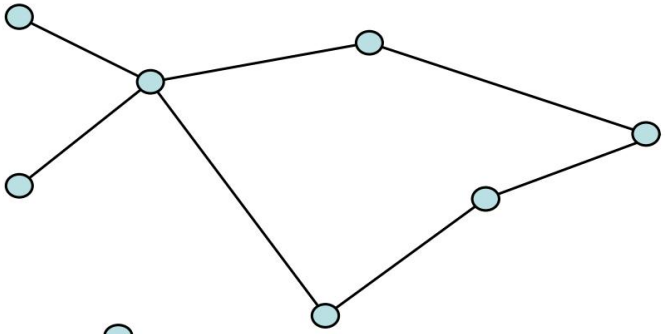
Graphene

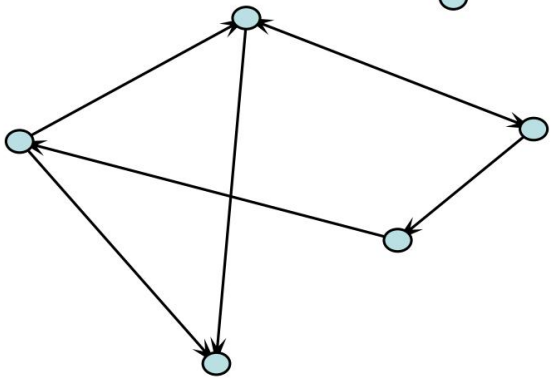
Graphs are a dominant data structure in computer science

Many problems in computer science can be described using graphs and solve using graph algorithms

Undirected graph

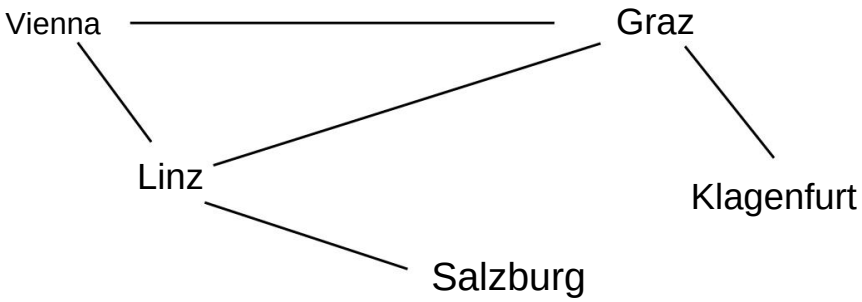


Directed graph



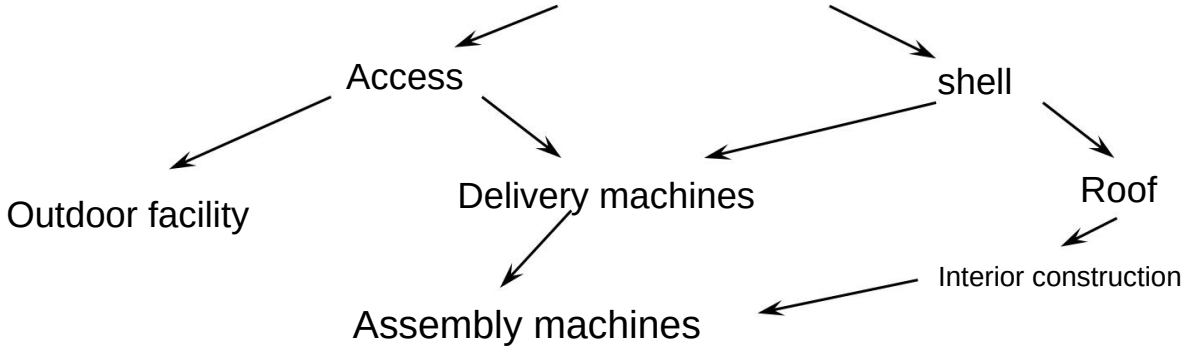
Graph examples

Local connections
e.g. trains



Process descriptions e.g.
project planning machine hall

Edges rep. Dependencies foundations



6.1 Undirected Graph

An ***undirected graph*** G = (,) consists of a set V (vertex) of ***vertices*** and a set E (edge) of ***edges,*** ie

V = {v1 , v2 , …, v|V|}, ***n*** = |V|

E = {e1 , e2 , … e|E|}, ***m*** = |E|

An ***edge*** e is an unordered pair of vertices from V, i.e. e = [vi , vj] with vi ,vj ÿ V and vi ÿ vj . vi and vj are involved in e .

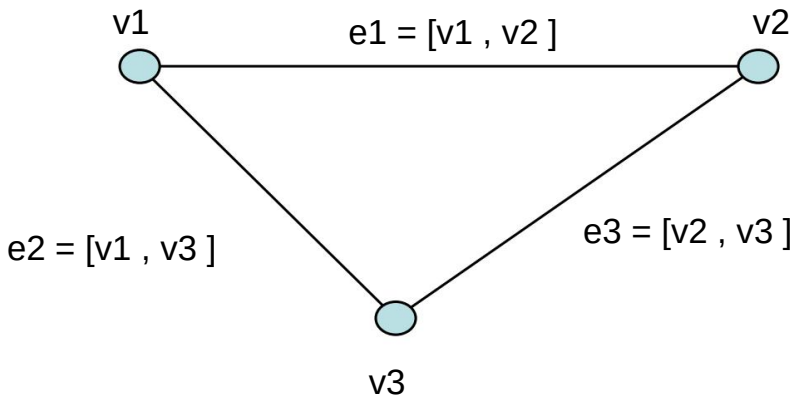
The number of edges a node participates in is the ***node's degree .***

G = (,)

V = {v1 , v2 , v3 }

E = {e1 , e2 , e3 }

City(v1) = 2



Number of edges

The number of edges m satisfies the following condition:

$$0 \ddot{y} \ddot{y} \quad \frac{(\ddot{y}1)}{2}$$

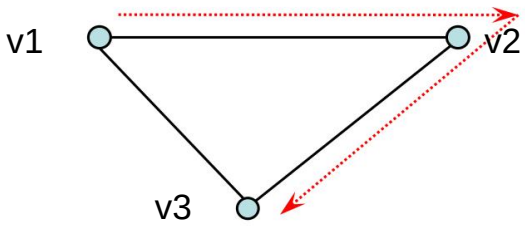
since every node has an edge to every other node can.

Furthermore: $\ddot{y}_{=1}$ degree$(=)$2

Edge sequence, path

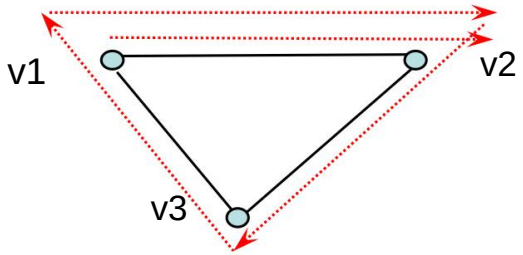
An ***edge sequence* (path)** from v1 to vk in a graph G is a finite sequence of edges [v1 ,v2], [v2 ,v3], ..., [vk-1 ,vk], where every 2 consecutive edges have a common end point

KF1 = [v1 ,v2], [v2 ,v3]



or

KF2 = [v1 ,v2], [v2 ,v3], [v3 ,v1], [v1 ,v2]



A ***path (simple path)*** is a sequence of edges in which all nodes are different (a single node is also considered a path) (KF1 is a path, KF2 is not)

The ***length*** of an edge sequence is the number of edges on the edge sequence.

Circle

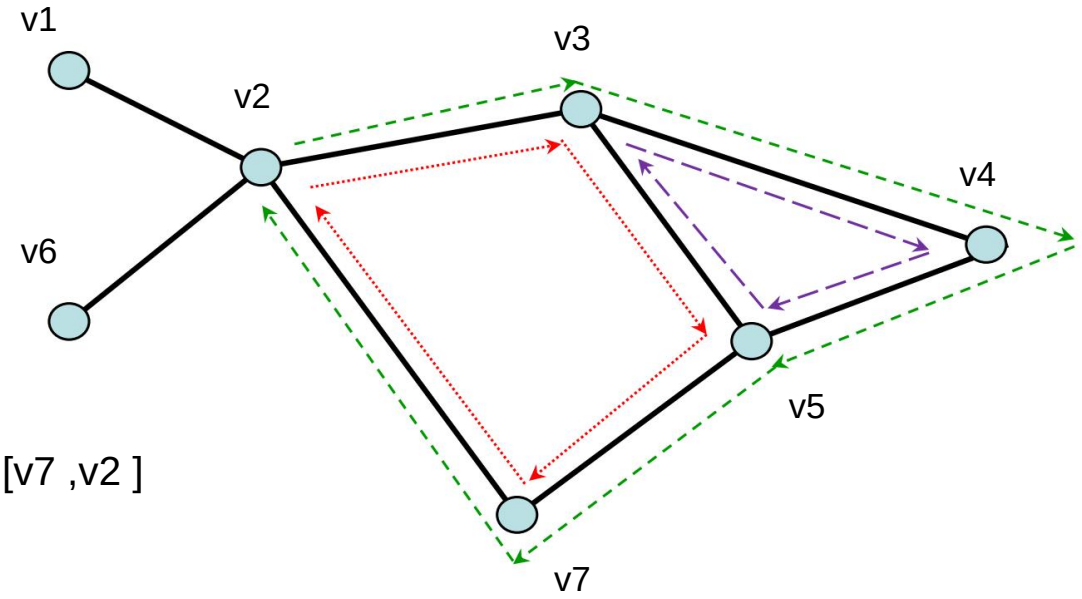
A ***cycle*** is a sequence of edges in which the nodes v1 , v2 ,…,vk-1 are all different, k ÿ 3 and vk = v1 .

Kreise:

[v2 ,v3], [v3 ,v5], [v5 , v7], [v7 ,v2]

[v3 ,v4], [v4 ,v5], [v5 , v3]

[v2 ,v3], [v3 ,v4], [v4 , v5], [v5 ,v7], [v7 ,v2]



A graph is said *to* ***be connected if*** for all possible node pairs vj ,vk there is a path that connects vj to vk .

A ***tree*** is therefore a connected circular (acyclic) graph.

Lemma 1: Every connected, acyclic graph G with
ÿ 2 node has at least one node with grade 1.

Proof: Take any node s and follow starting
from s a path in G.

After each node is visited at most once by , after at most -1 node
a node will be reached that has no edge to an unvisited node.
If an edge had an already visited node, there would be one

Circle in G, which is not possible. Therefore no edge is closed
an already visited node (except the edge through which it was
visited) and also no edge to an unvisited one
Node.

Therefore has degree 1.

Lemma 2: A tree with nodes has exactly ÿ 1 edges.

Proof: By induction over the number of nodes.

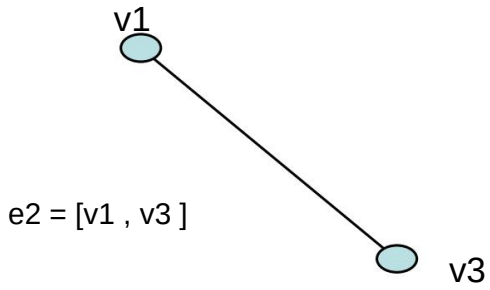
= 1: Every graph with 1 node is edgeless. Hence also a tree with 1 node edgeless. Therefore the induction claim that a tree with = 1 node is true exactly – 1 = 0 has edges.

> 1: Given a tree with nodes. According to Lemma 1, there is at least one node that only participates in one edge is. Remove from tree. The new graph is connected and circular, i.e. a tree again, with – 1 node.
We call him '. By the induction assumption ' has exactly - 2 edges. can be created by adding and the edge. Therefore has exactly – 1 edges.

Subgraph

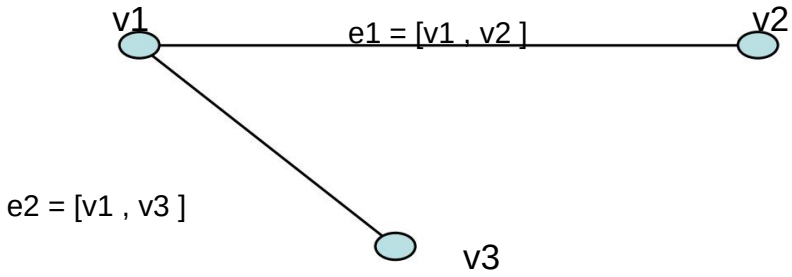
A graph = (ÿ and) is called ***a subgraph*** of G = (,), if ÿ ÿ ÿ ÿ ÿ .

ÿ = ( )
ÿ = { 1, 3 }
= { 2}



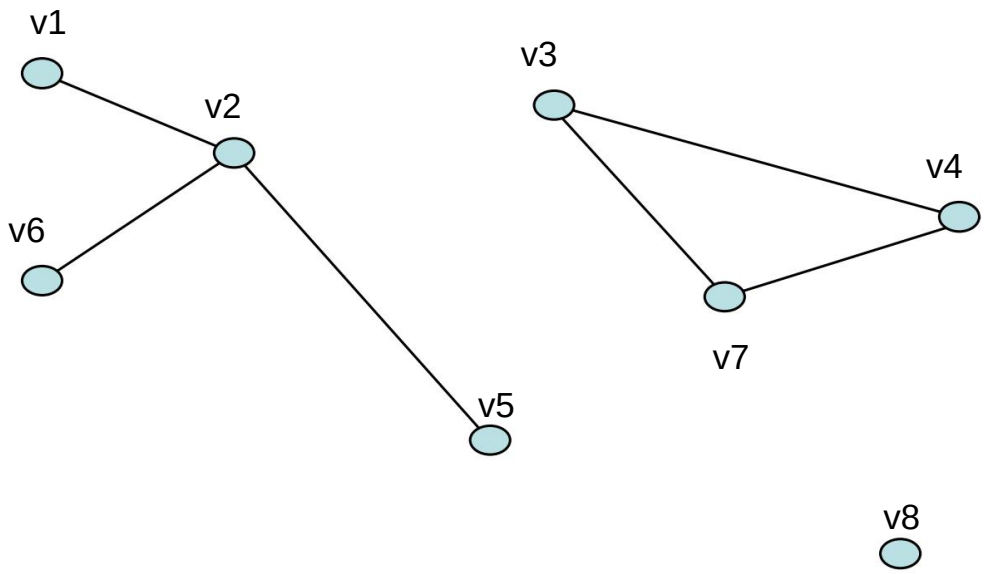
A subgraph) is an ÿÿ = ***exciting*** **tree** of a connected graph G = (,), = and if ÿÿ ÿÿ forms a tree.

ÿÿ = ( )
ÿ = { 1, 2, 3 }
= { 1, 2}



Components

Every maximal, connected subgraph is called ***a component*** of the Graphene.



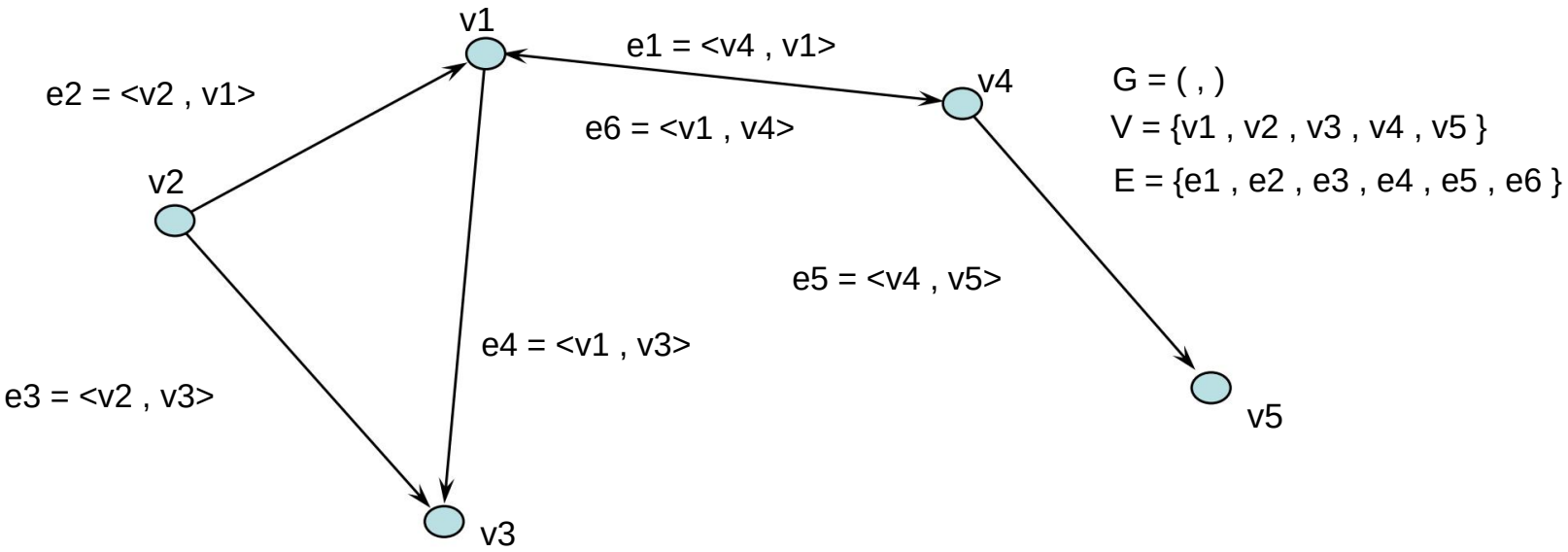
Komponenten:

{v1 , v2 , v5 , v6 } {v3 , v4 , v7 } {v8 }

6.2 Directed Graph

A ***directed graph*** G = (,) consists of a set V of nodes and a set E of edges, where the edges are *ordered* pairs < > or of nodes from V. (,)

One edge = (,) is called ***outgoing edge*** of and ***incoming edge*** of .



Between v1 and v4 there are 2 edges, a back and forth edge (also double edge).

Node degree

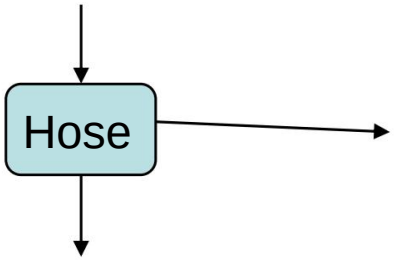
Entry and exit degrees of a node:

indegree(v) with ÿ : i.e. { ÿ | (ÿ,) ÿ | }

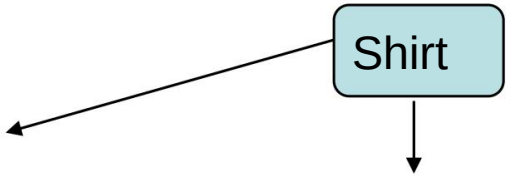
number of edges entering v

outdegree(v) mit ÿ : ÿ | { ÿ | (, ÿ) }

i.e. number of edges originating from v



indegree: 1, outdegree: 2



indegree: 0, outdegree: 2

Number of edges

The number of edges m satisfies the following condition:

$$0 \ddot{y} \ddot{y} (\ddot{y} 1),$$

since every node has an edge to every other node can.

Furthermore:

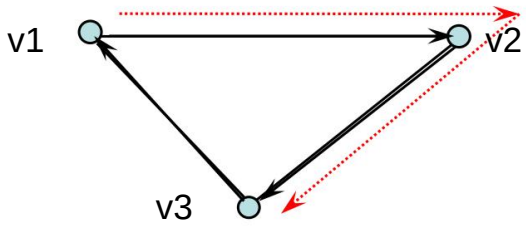
$$\ddot{y} \quad \text{indegree}(=)$$
$$=1$$

$$\ddot{y} \quad \text{outdegree}(=)$$
$$=1$$

Edge sequence, path

A ***(directed) edge sequence (directed path)*** from v1 to vk in a directed graph G is a finite sequence of edges <v1 ,v2>, <v2 ,v3>, ..., <vk-1 ,vk>, each with 2 consecutive edges Edges have a common end point or starting point

KF1 = <v1 ,v2>, <v2 ,v3>



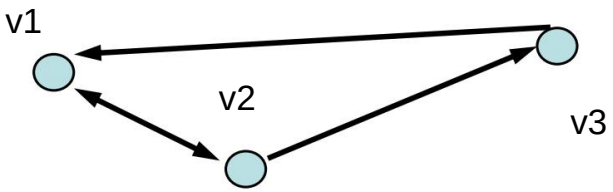
A ***simple directed path*** is one

Edge sequence in which all nodes are different (a single node is also considered a path)

The ***length*** of an edge sequence is the number of edges in the edge sequence.

Circle

A ***cycle*** is a sequence of edges in which the nodes v1 ,v2 ,…,vk-1 are all different and vk = v1 .



Circles

<v1 ,v2>, <v2 ,v1>

<v1 ,v2>, <v2 ,v3>, <v3 , v1>

A ***DAG (directed acyclic graph)*** is a directed circular (acyclic) graph.

Acyclic graphs

Lemma 3a: Every acyclic, directed graph has one node with no *outgoing* edge.

Proof:

Take any node and follow starting from one directed path in .

After each node is visited at most once by,
After at most ÿ 1 steps a node is reached that has no outgoing edge to an unvisited node.

Would have an outgoing edge to an already visited node, which is there would be a circle in not possible.

Therefore, there is no outgoing edge in .

A node without outgoing edges is also called ***a sink .***

Acyclic graphs

Lemma 3b: Every acyclic, directed graph has one node with no *incoming* edge.

Proof:

Let ' = (, ') be the ***inverse*** graph of = (,), i.e. the graph with edge set ' = { (,) | (,) ÿ . And is acyclic is also acyclic.

ÿ

Now Lemma 3a hat ÿ a node without an outgoing edge.

Then there is no incoming edge in .

A node with no incoming edges is also called ***a source .***

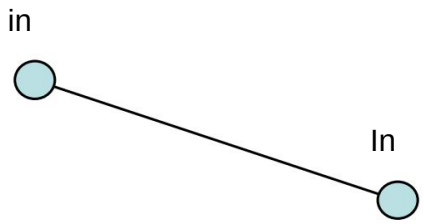
6.3 Storing Graphs

We distinguish between 2 methods for storing graphs:

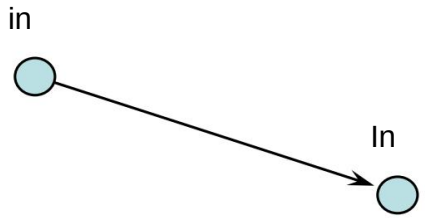
- Adjacency matrix representation
- Adjacency list representation

A node w is called ***adjacent (neighboring)*** to a node v, if an edge leads from v to w, e.g

undirected edges [v,w]: v adjacent to w w adjacent to v



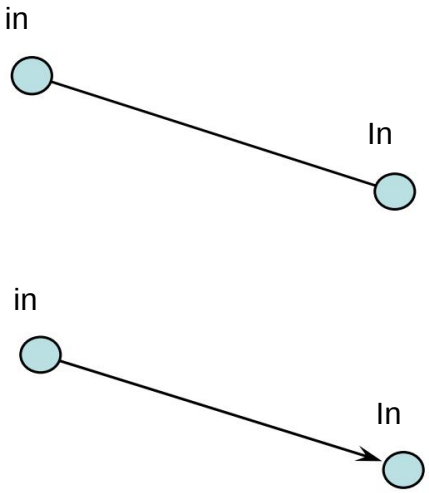
directed edge <v,w>: w adjacent to v v but NOT adjacent to w

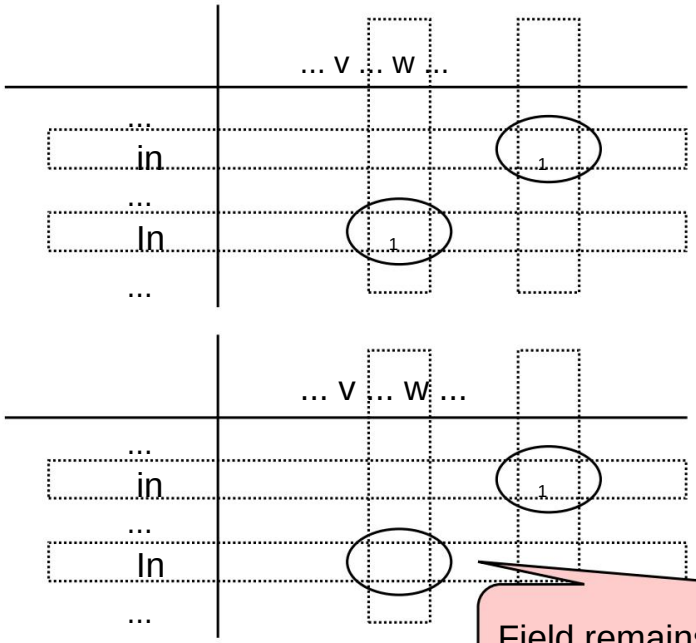


6.3.1 Adjacency matrix

In the adjacency matrix representation, the nodes represent Index values of a 2-dimensional matrix A

If node w is adjacent to node v, the field A[v,w] is set in the matrix, e.g. 1, 'true', value (edge weight), etc.



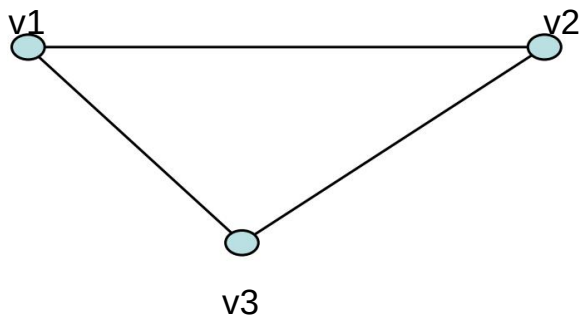


With un-directed Graphene is die Matrix symmetrical

Field remains empty (or contains 0) because v is not adjacent to w

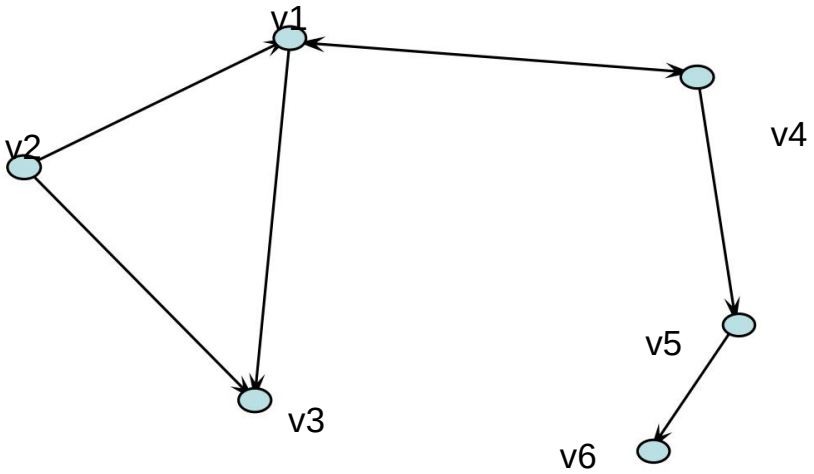
Adjacency matrix examples

Undirected graph



	v1	v2	v3
v1		1	1
v2	1		1
v3	1	1	

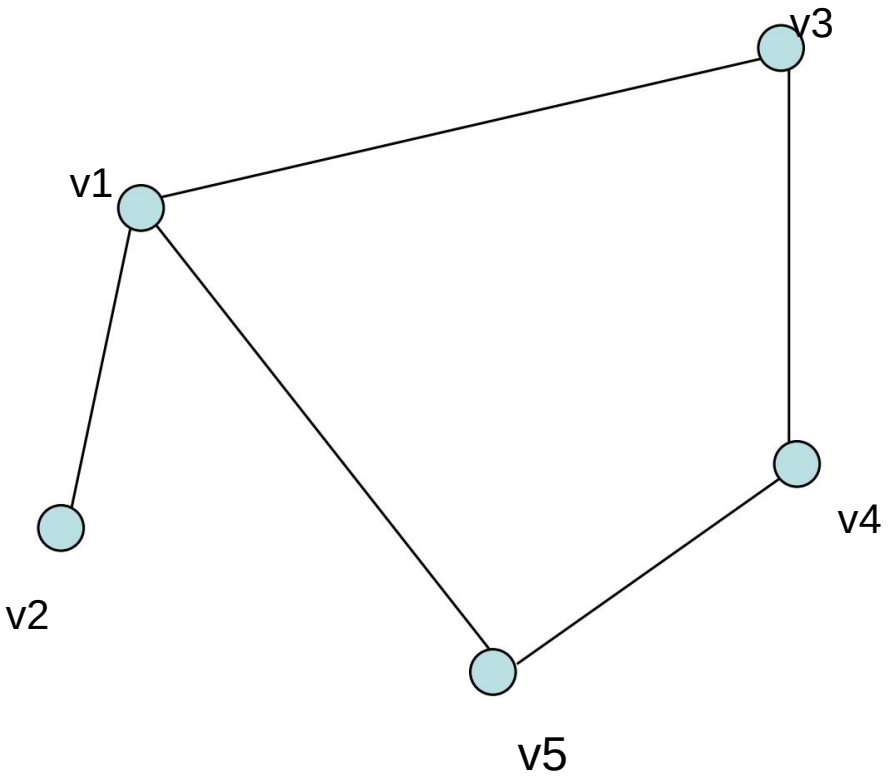
Directed graph



	v1	v2	v3	v4	v5	v6
v1		1		1		
v2			1			
v3						
v4					1	
v5						1
v6						

universität wien

Undirected Graph (Matrix)

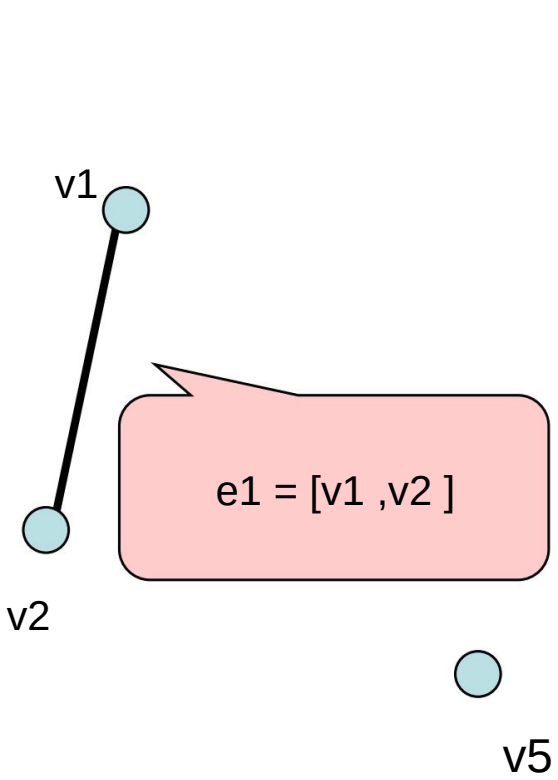


	v1	v2	v3	v4	v5
v1					
v2					
v3					
v4					
v5					

Diagonal not defined;
at the same time
Symmetrieachse

Example: Adj. Matrix - undirected graph (D2)

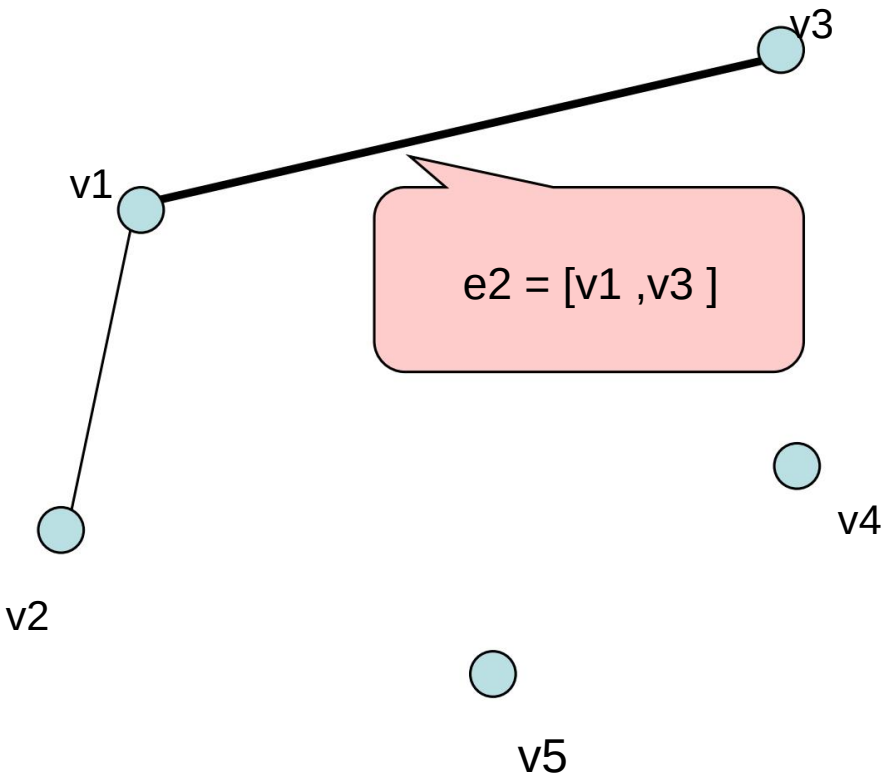
Undirected graph



	v1	v2	v3	v4	v5
v1		1			
v2	1				
v3					
v4					
v5					

Example: Adj. Matrix - undirected graph (D3)

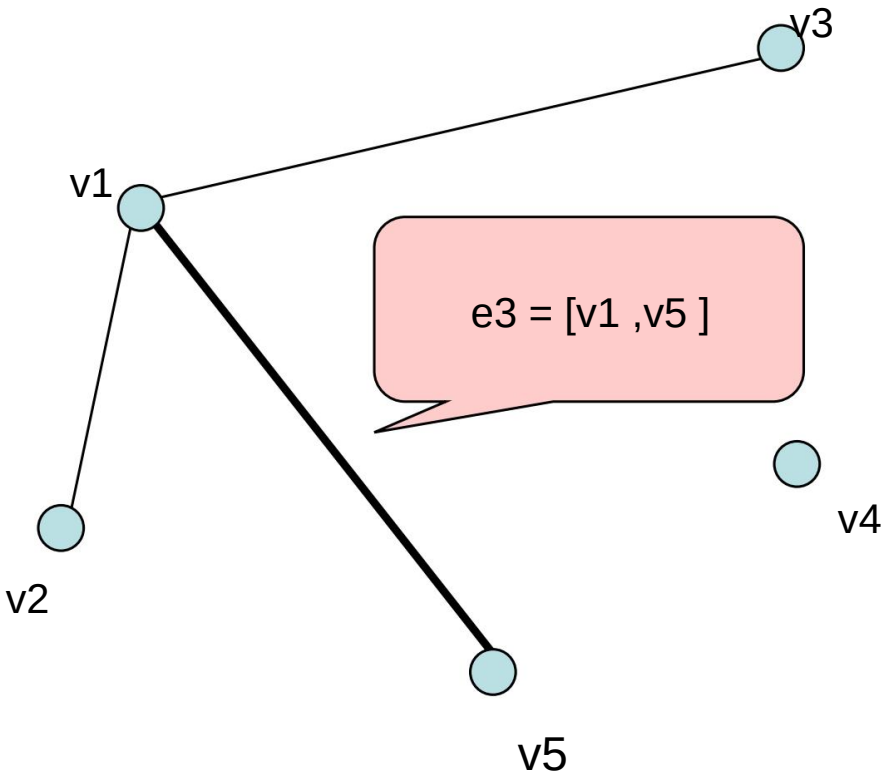
Undirected graph



	v1	v2	v3	v4	v5
v1			1		
v2			1		
v3	1	1			
v4					
v5					

Example: Adj. Matrix - undirected graph (D4)

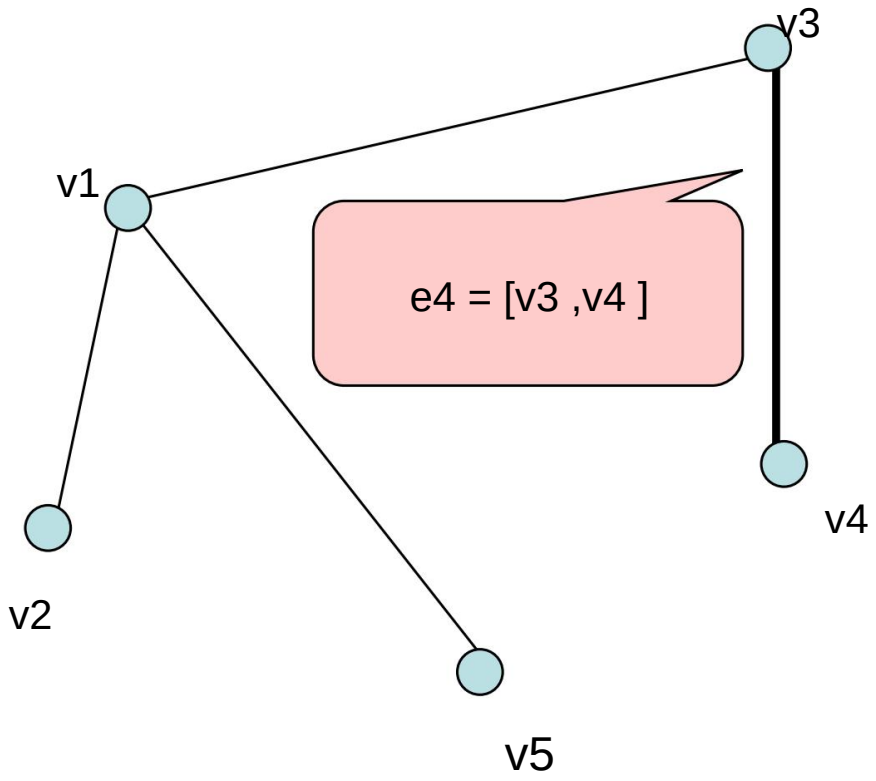
Undirected graph



	v1	v2	v3	v4	v5
v1					1
v2	1				
v3	1				
v4					
v5	1				

Example: Adj. Matrix - undirected graph (D5)

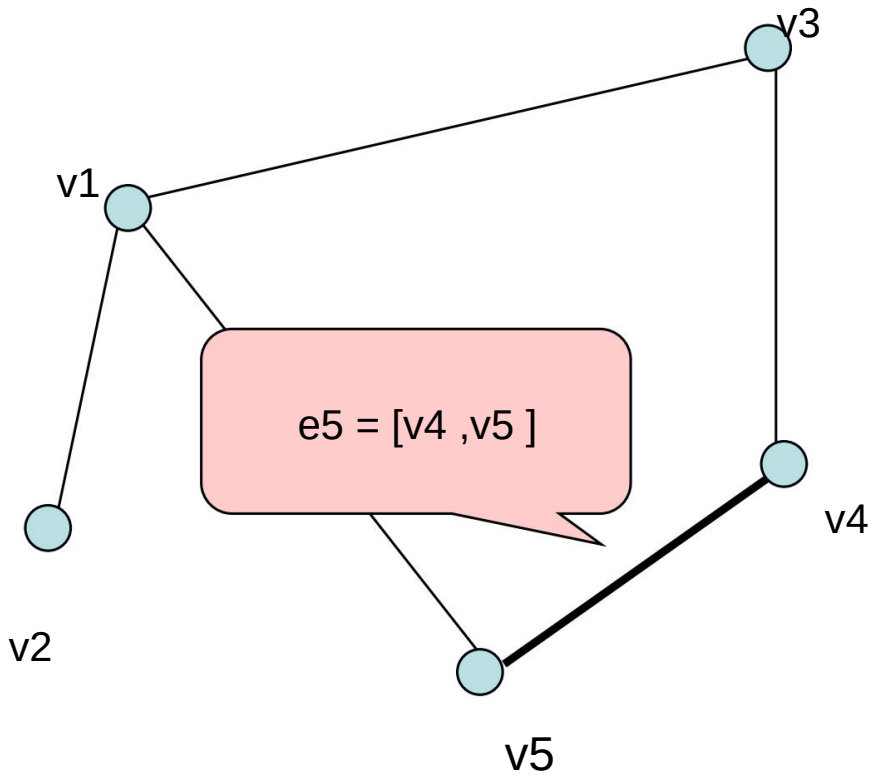
Undirected graph



	v1	v2	v3	v4	v5
v1		1	1		1
v2	1				
v3	1			1	
v4			1		
v5	1				

Example: Adj. Matrix - undirected graph (D6)

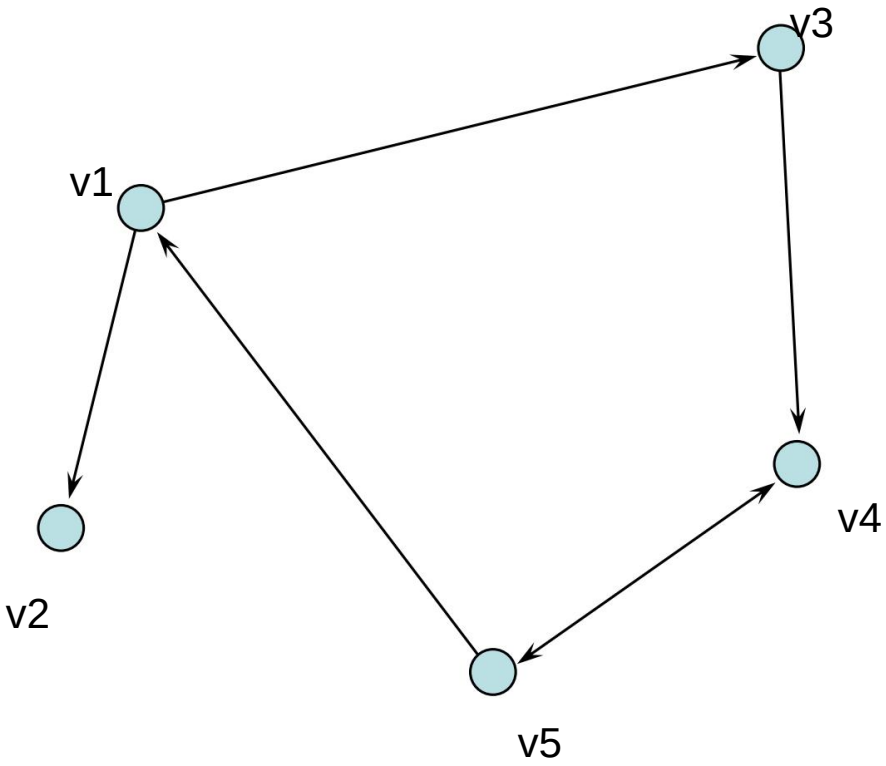
Undirected graph



	v1	v2	v3	v4	v5
v1		1	1		1
v2	1				
v3	1			1	
v4			1		1
v5	1				1

Example: Adj. Matrix - directed graph (D1)

Directed graph (matrix)

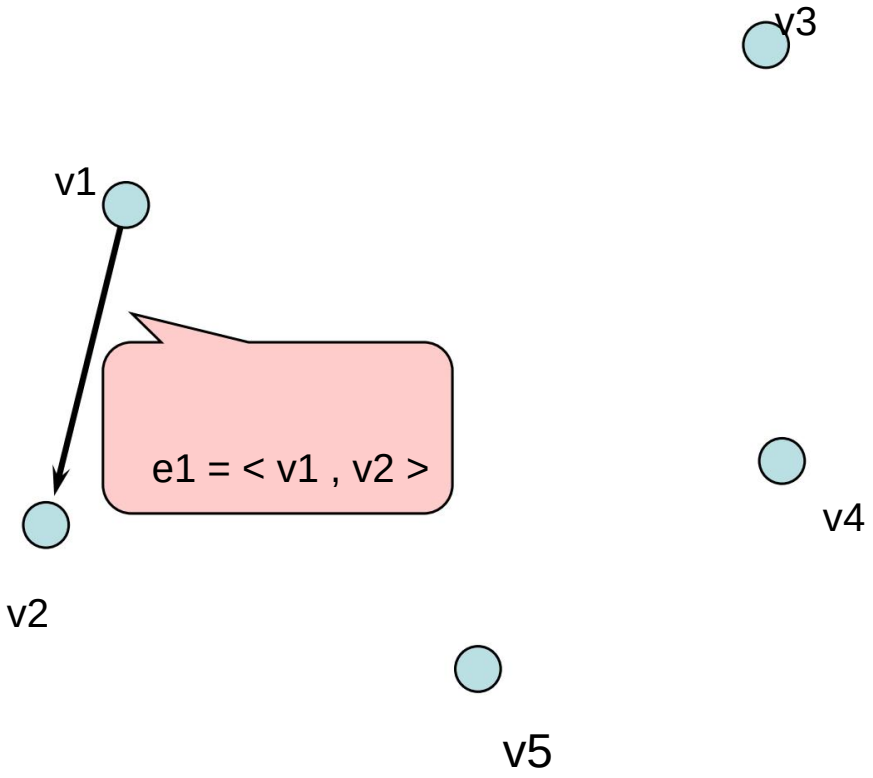


	v1	v2	v3	v4	v5
v1					
v2					
v3					

v4 Links to itself - not defined v5 No axis of symmetry!

Example: Adj. Matrix - directed graph (D2)

Directed graph

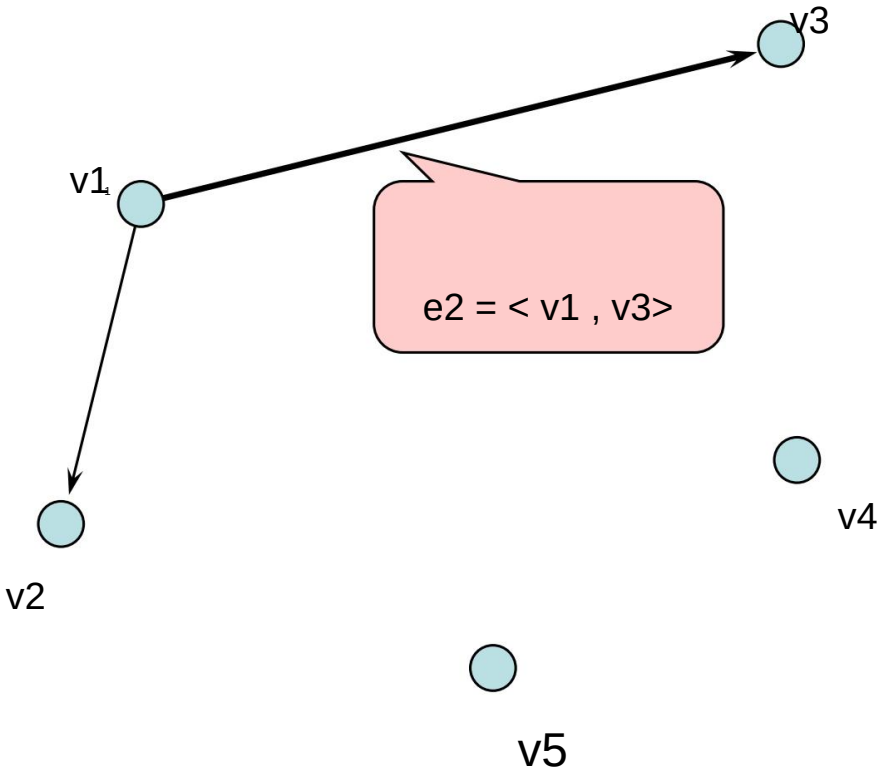


	v1	v2	v3	v4	v5
v1					
v2					
v3					
v4					
v5					

1

Example: Adj. Matrix - directed graph (D3)

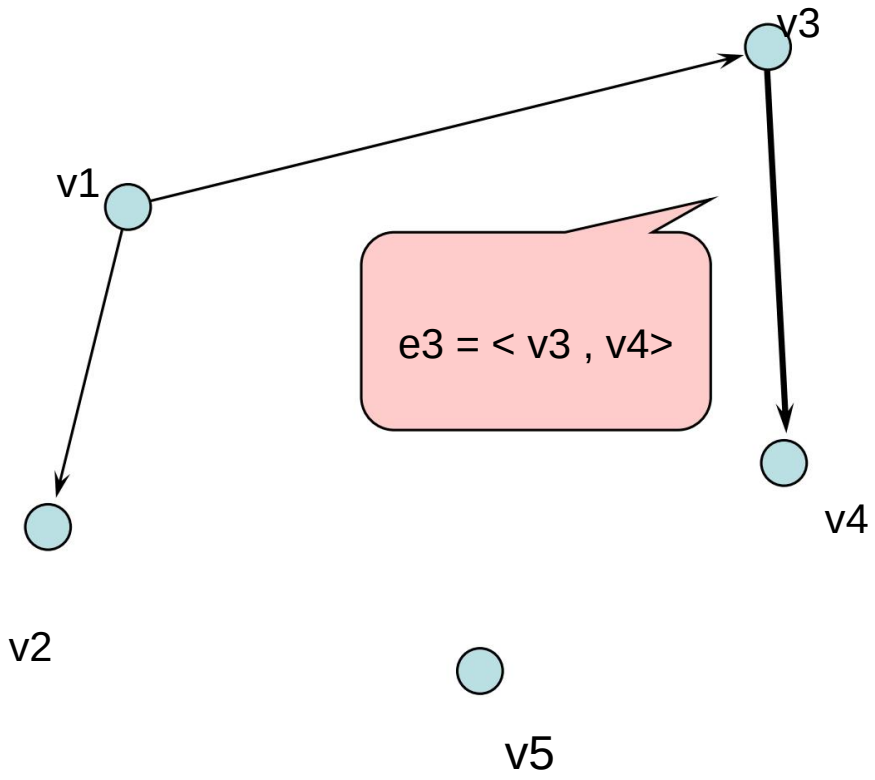
Directed graph



	v1	v2	v3	v4	v5	1	**1**
v1			○				
v2							
v3							
v4							
v5							

Example: Adj. Matrix - directed graph (D4)

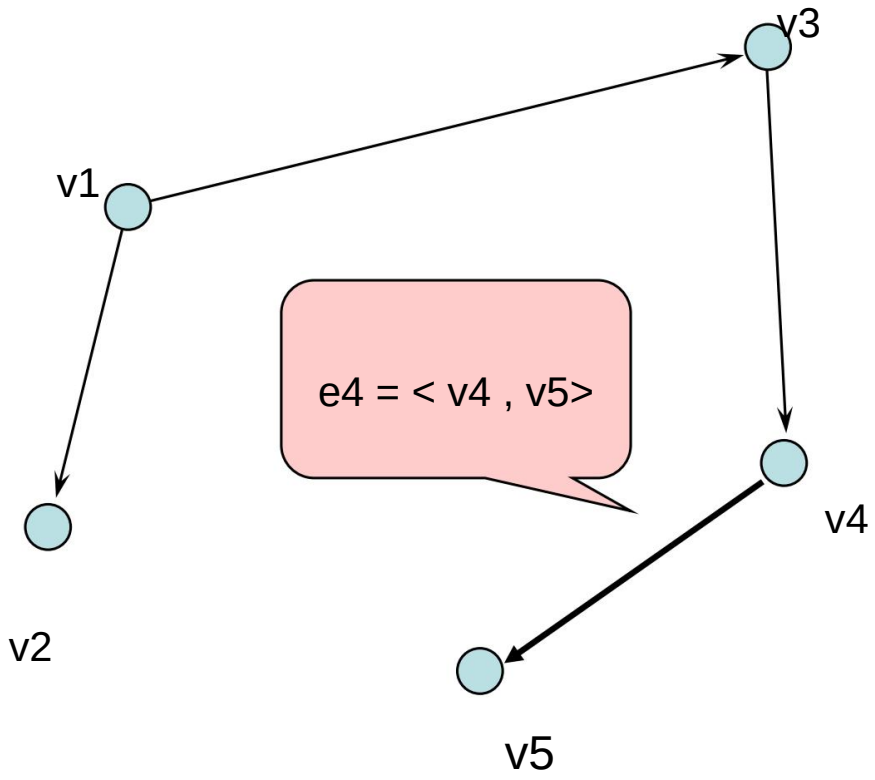
Directed graph



	v1	v2	v3	v4	v5
v1			1	1	
v2					
v3				1	
v4					
v5					

Example: Adj. Matrix - directed graph (D5)

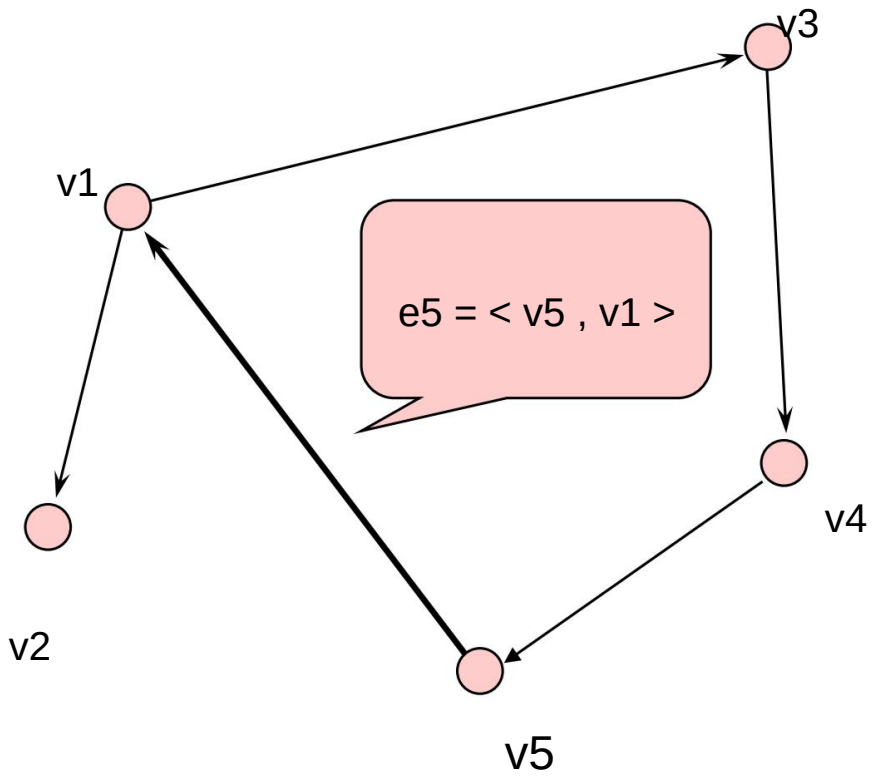
Directed graph



	v1	v2	v3	v4	v5
v1			1	1	
v2					
v3					1
v4					1
v5					

Example: Adj. Matrix - directed graph (D6)

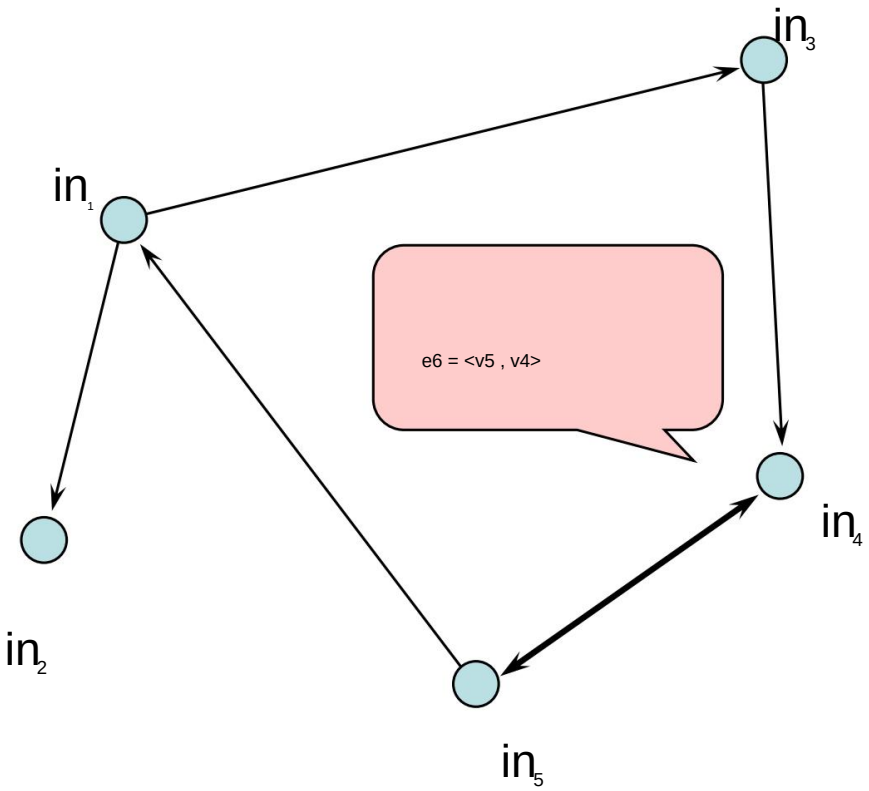
Directed graph



	v1	v2	v3	v4	v5
v1			1	1	
v2					
v3				1	
v4					1
v5	1				

Example: Adj. Matrix - directed graph (D7)

Directed graph



	v1	v2	v3	v4	v5
v1		1	1		
v2					
v3				1	
v4					1
v5	1			1	

Characteristics

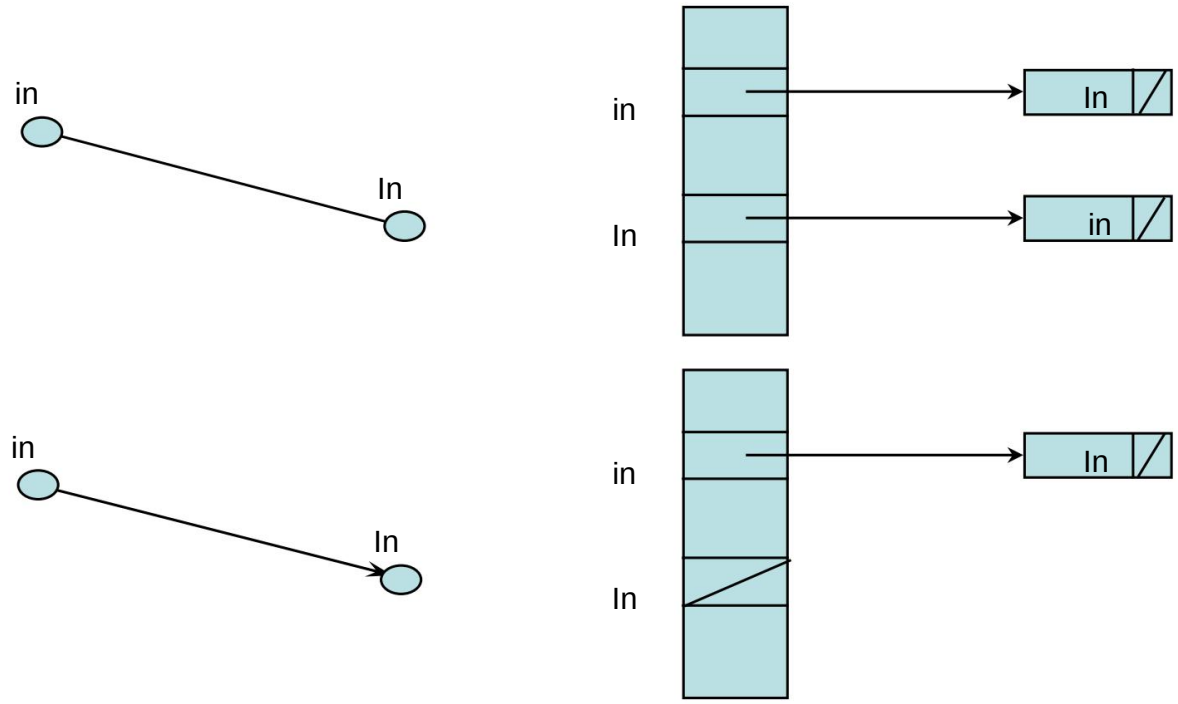
+ good for small graphs or graphs with many edges + checking adjacency property: O(1) + some algorithms simpler

- quadratic memory effort: O(|V|2) - poorly suited for traversal, calculation effort: O(|V|2)

6.3.2 Adjazenzlists

In the adjacency list representation, for each node, all adjacent nodes are stored in a linear list

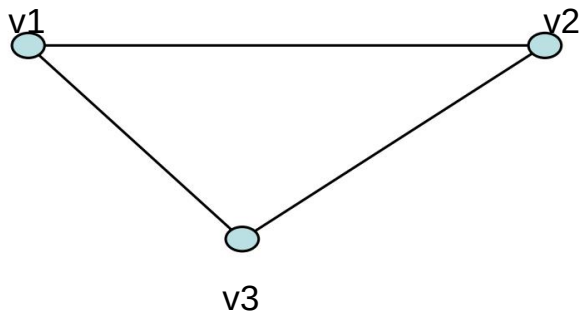
This means that only the edges that occur are noted

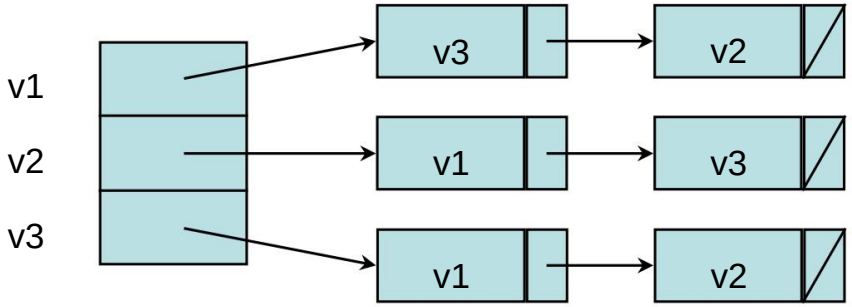


Possible C++ data structure:
class node { public: int v; node *next;} node *adjlist[maxV]; // adjacency list (maxV: maximum number of nodes)

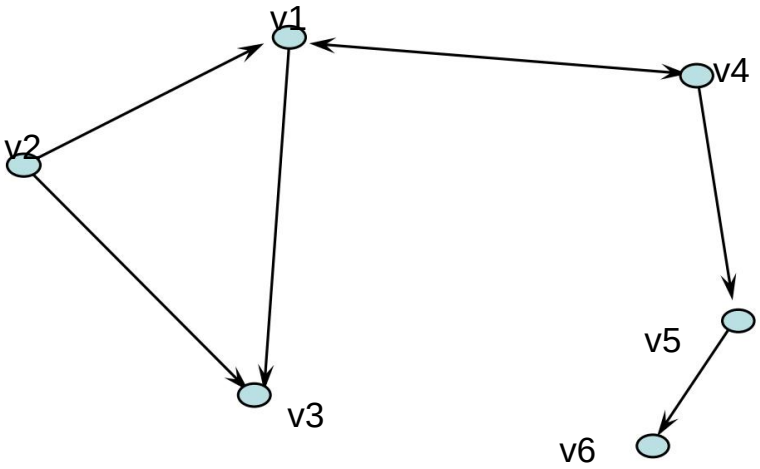
Adjacency list examples

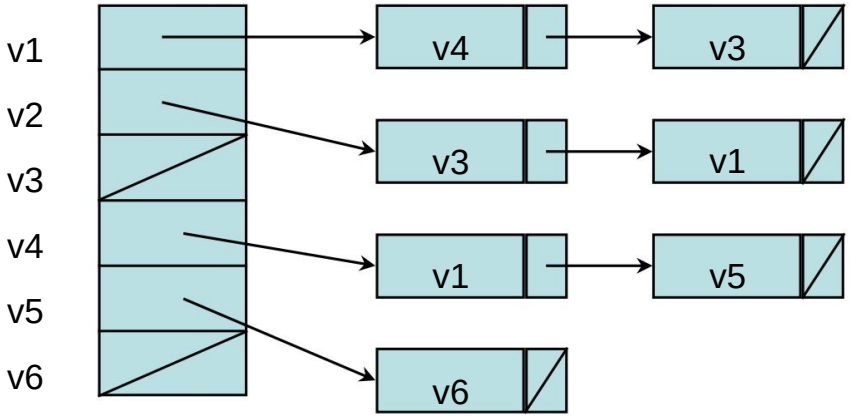
Undirected graph





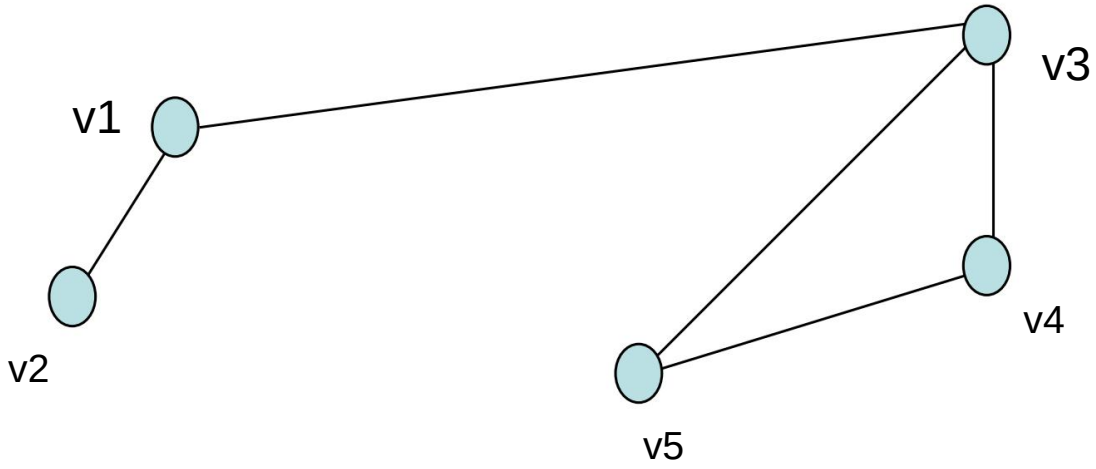
Directed graph

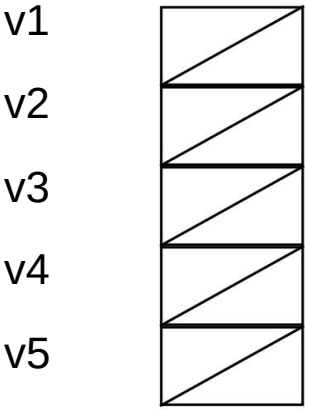




Example: Adj. List - undirected graph (D1)

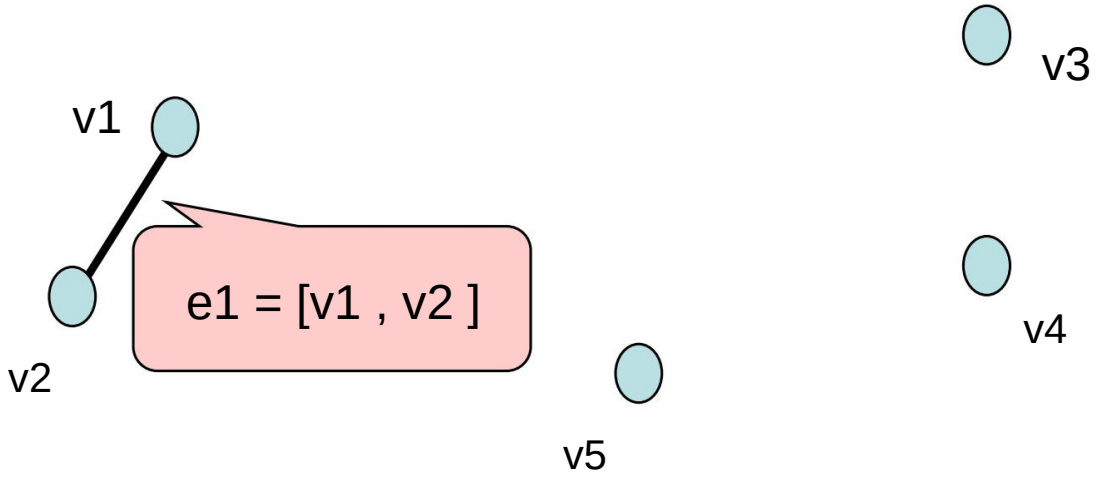
Undirected graph
(List)

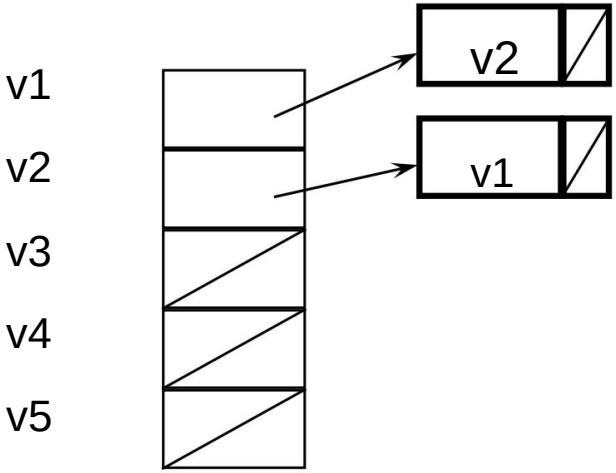




Example: Adj. List - undirected graph (D2)

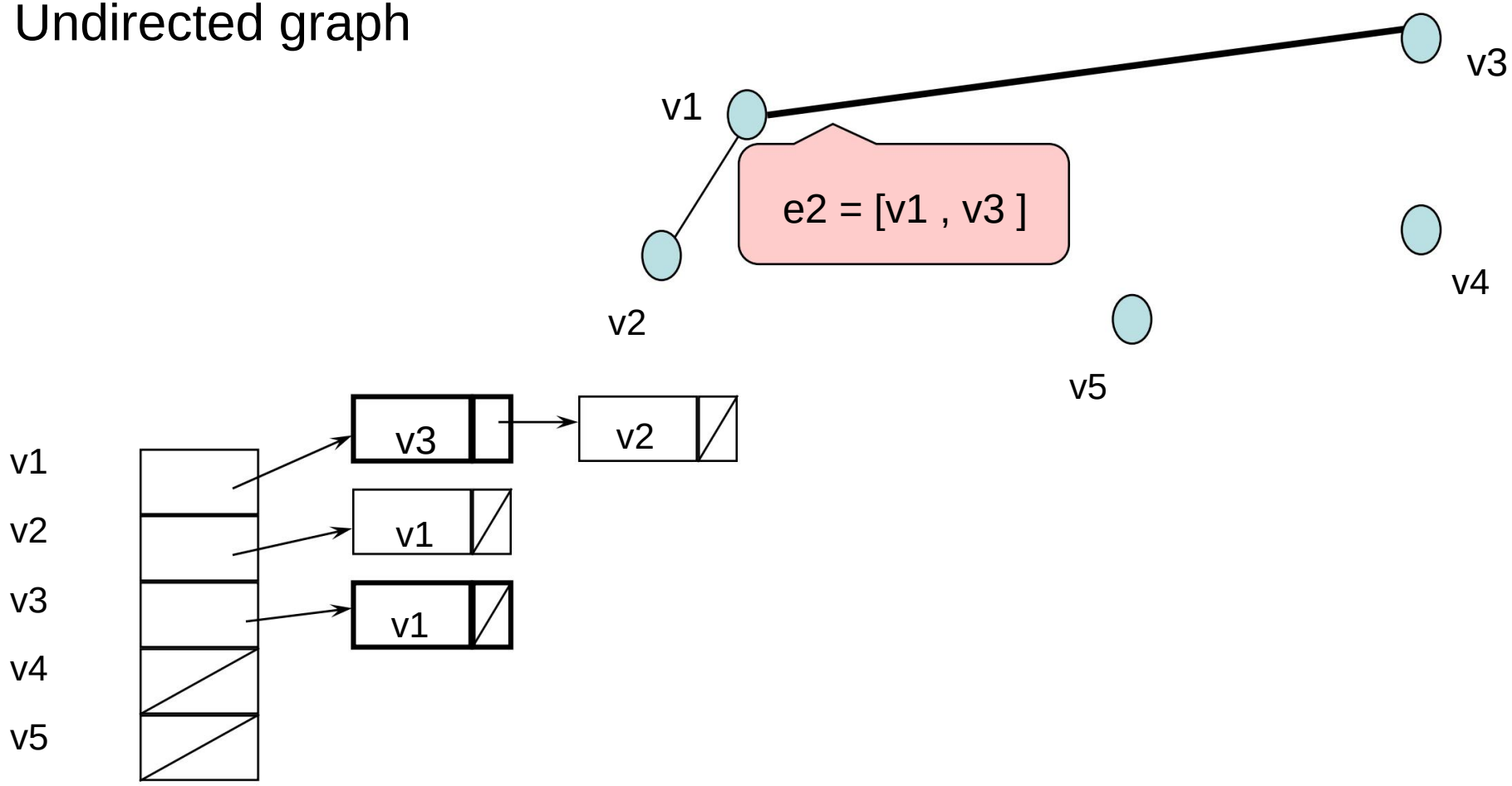
Undirected graph





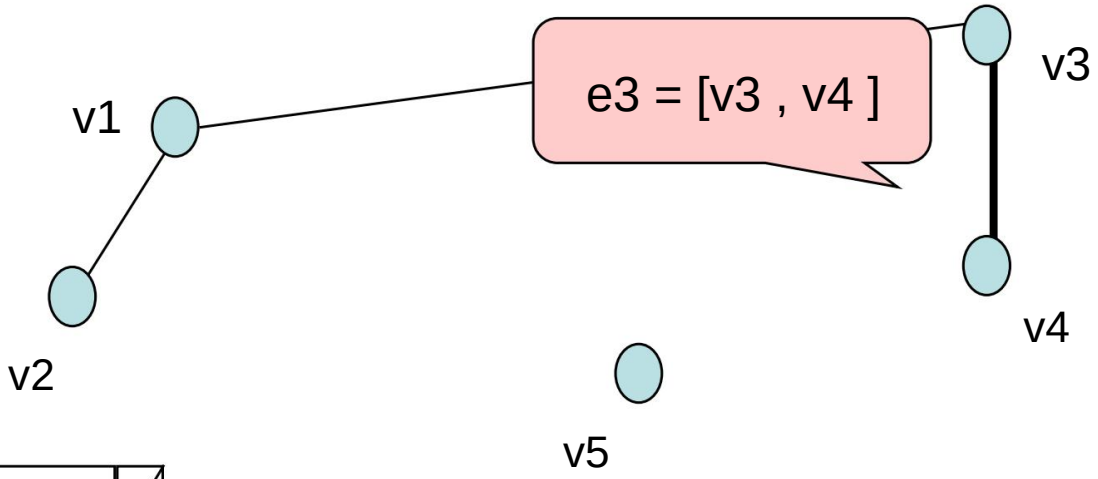
Example: Adj. List - undirected graph (D3)

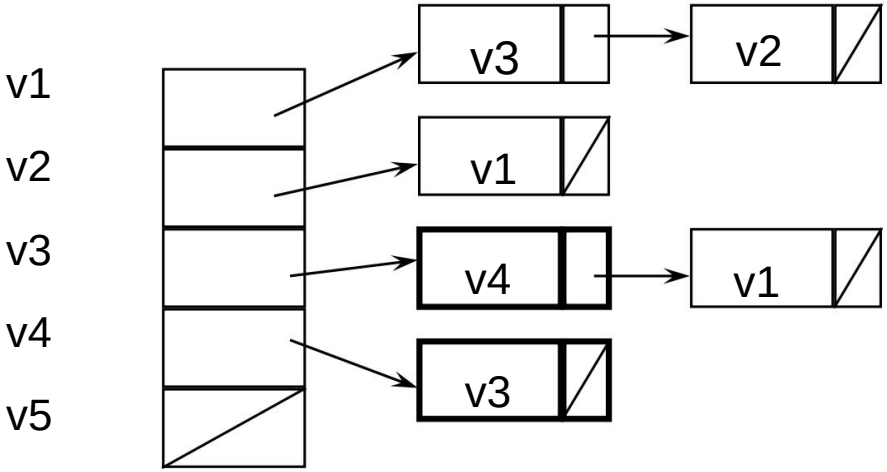
Undirected graph



Example: Adj. List - undirected graph (D4)

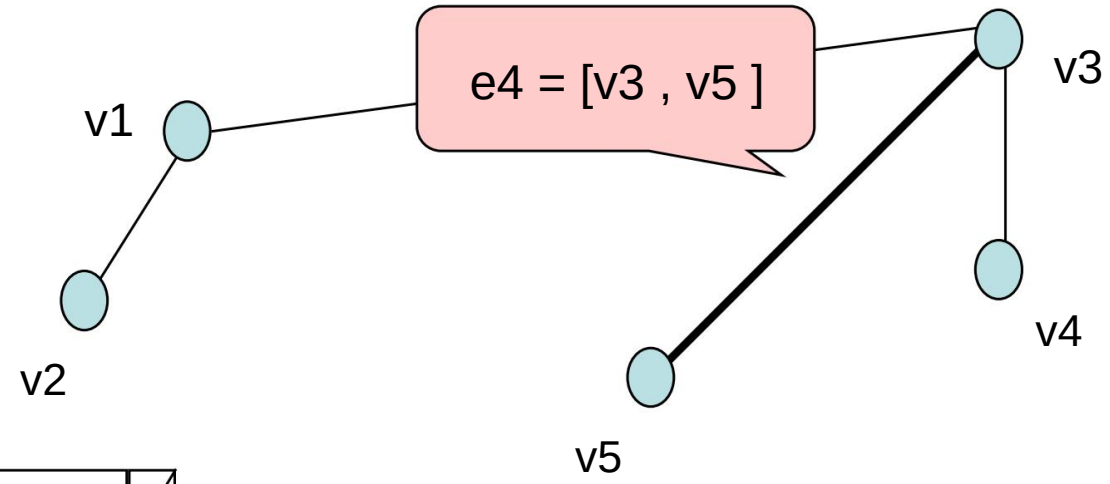
Undirected graph

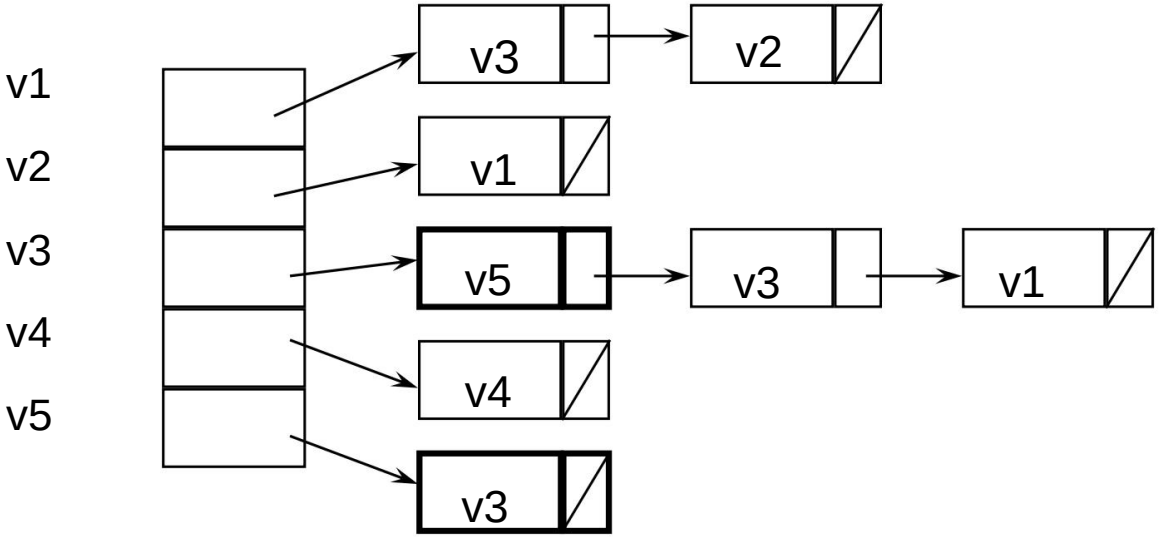




Example: Adj. List - undirected graph (D5)

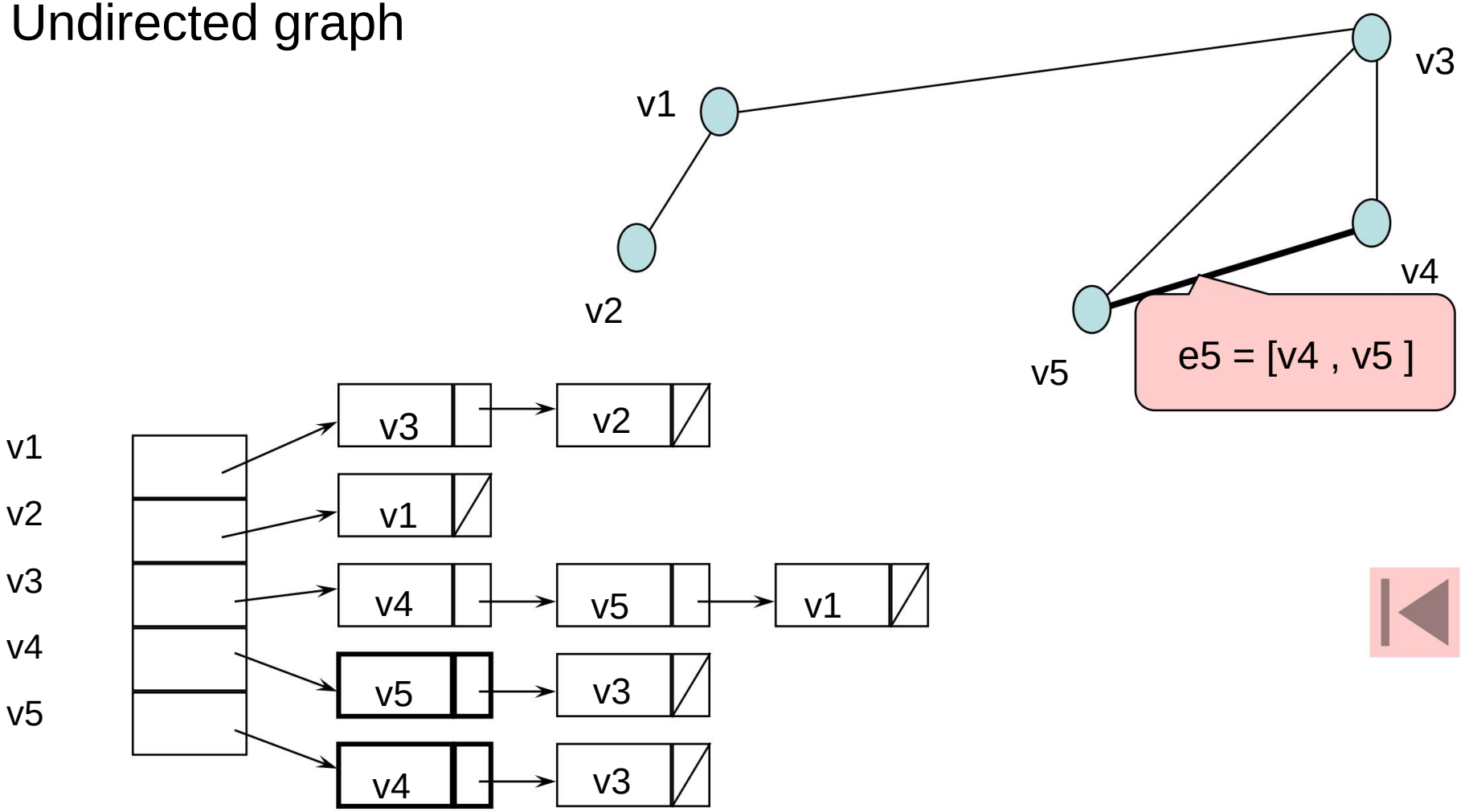
Undirected graph





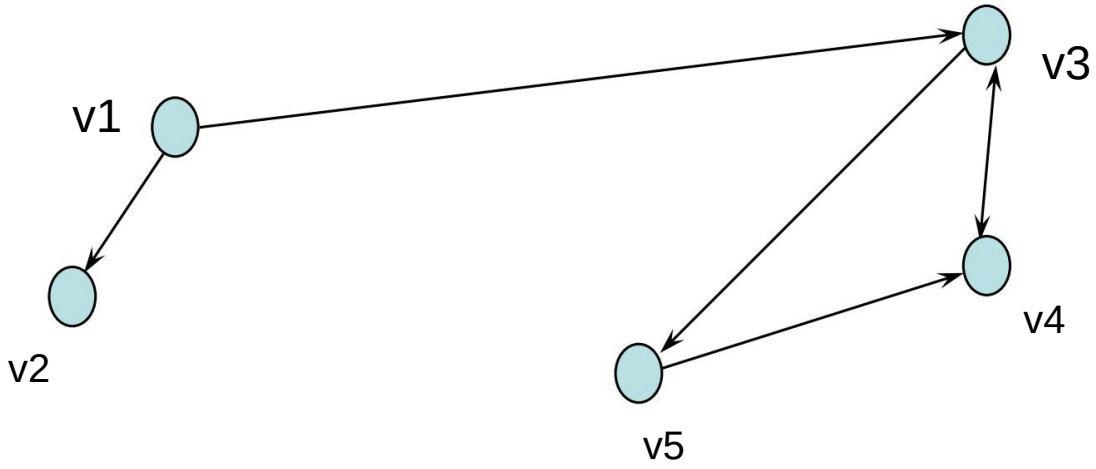
Example: Adj. List - undirected graph (D6)

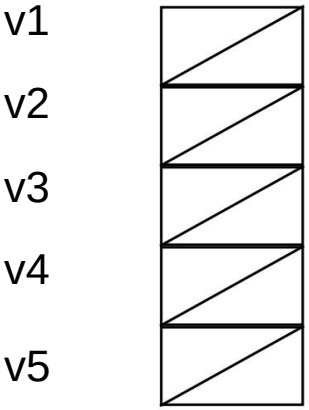
Undirected graph



Example: Adj. List - directed graph (D1)

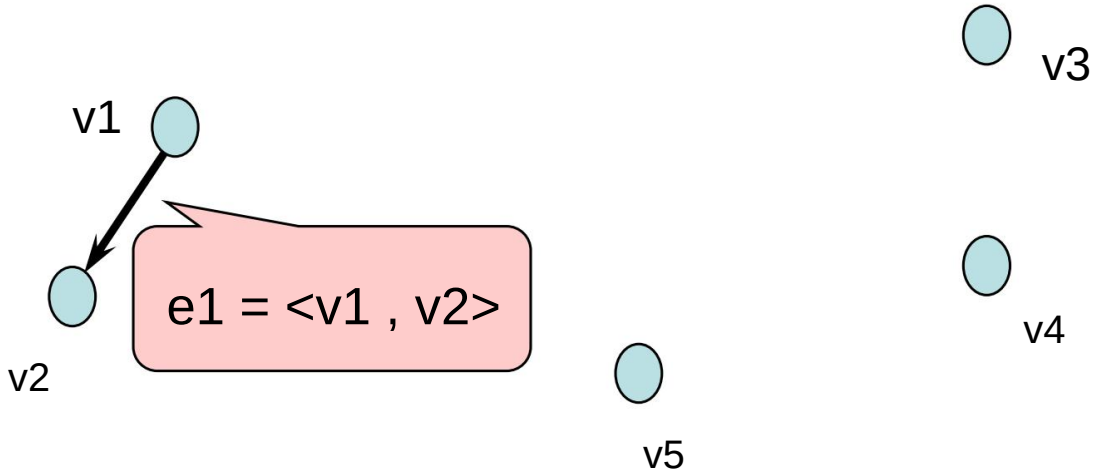
Directed graph
(List)

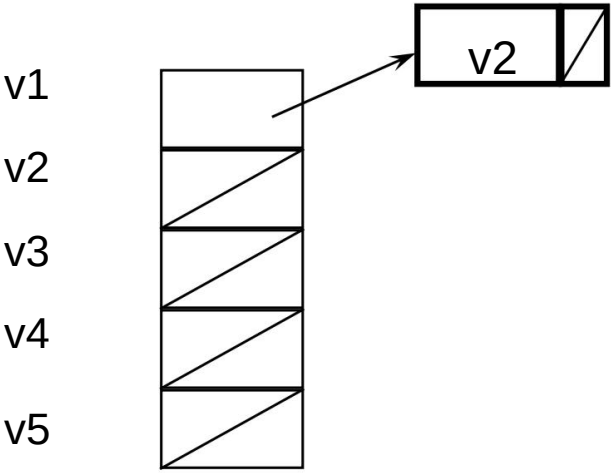




Example: Adj. List - directed graph (D2)

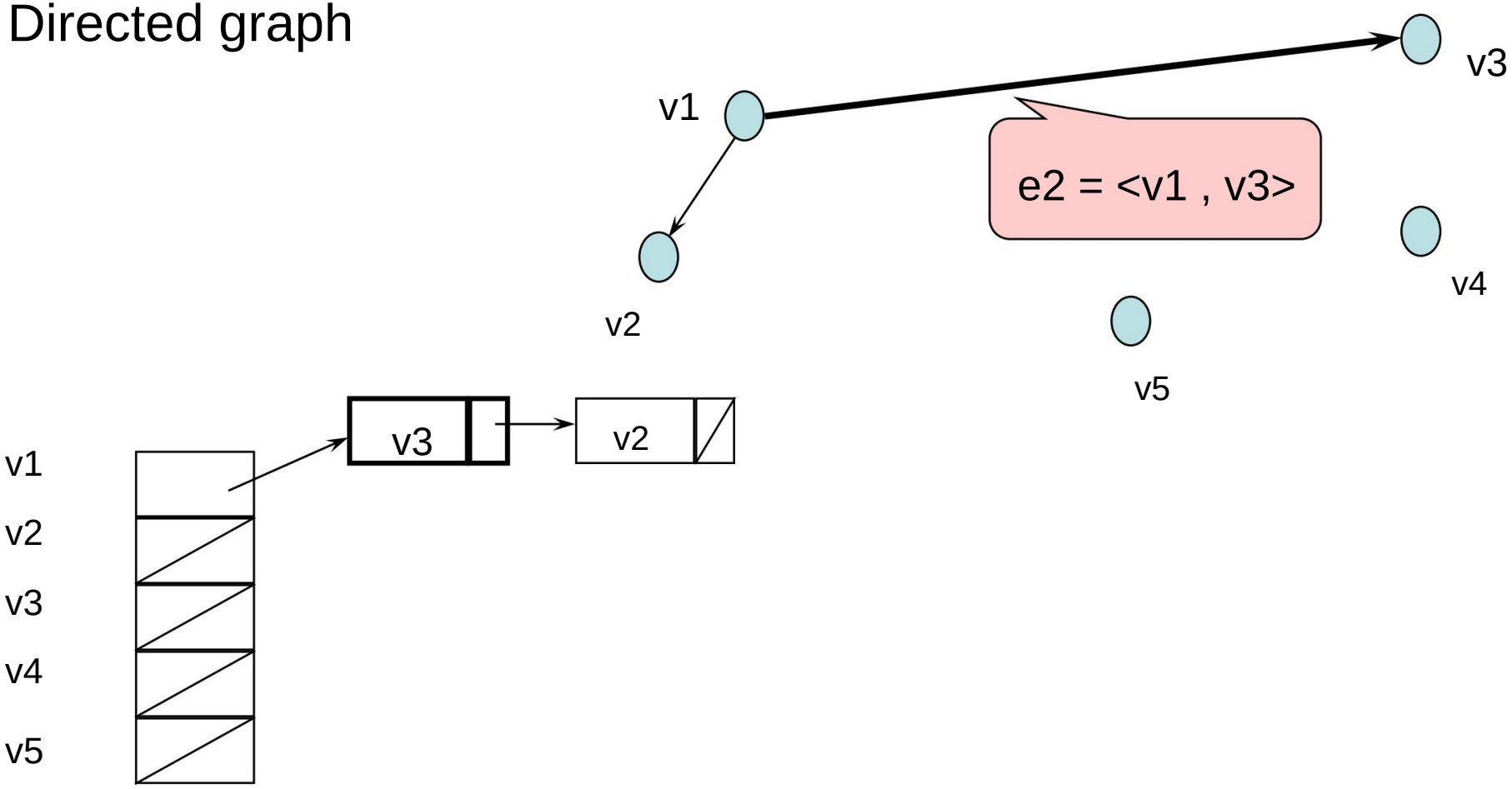
Directed graph





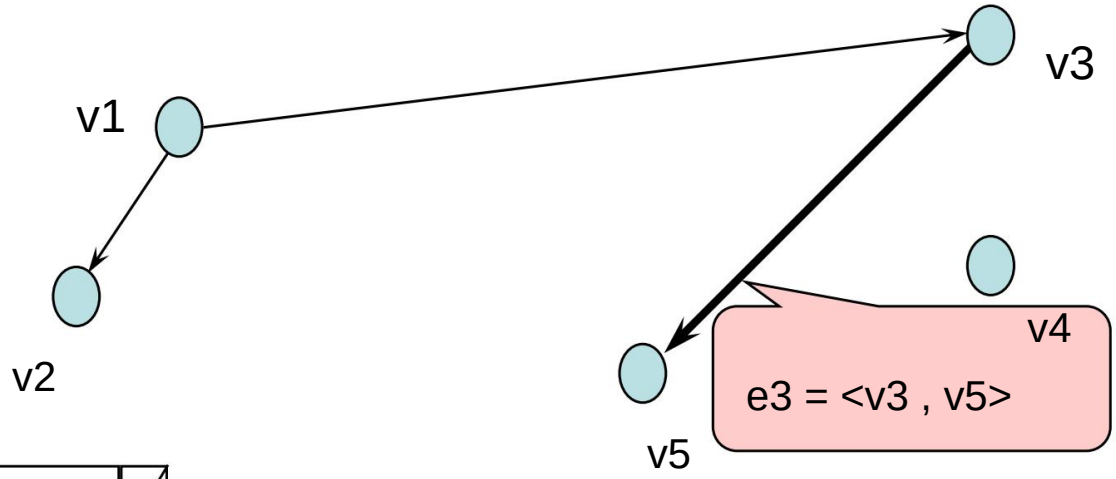
Example: Adj. List - directed graph (D3)

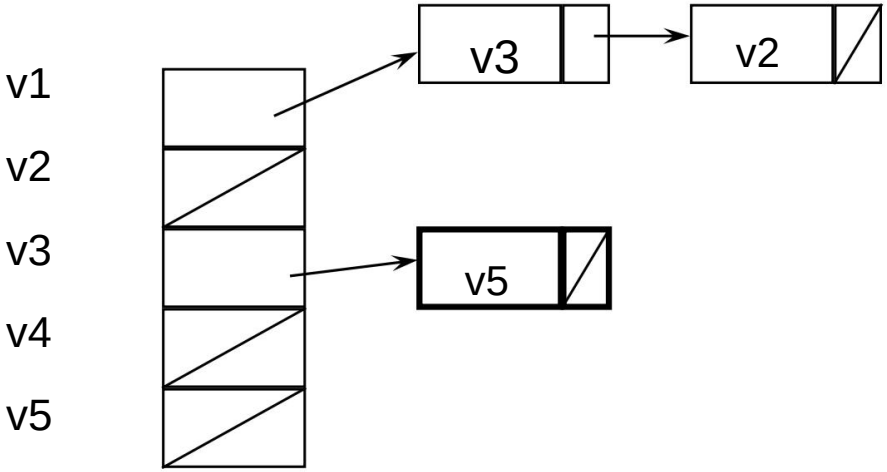
Directed graph



Example: Adj. List - directed graph (D4)

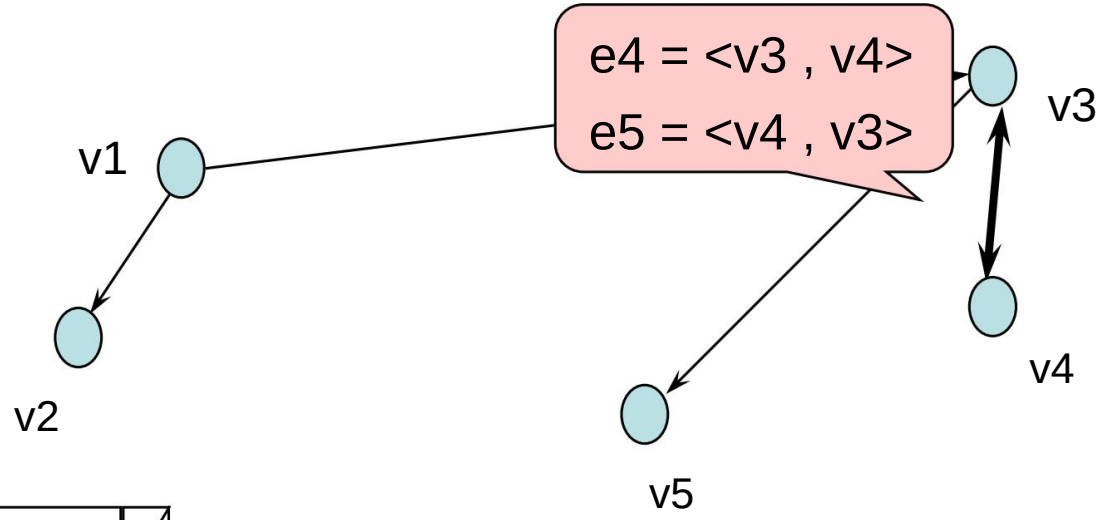
Directed graph

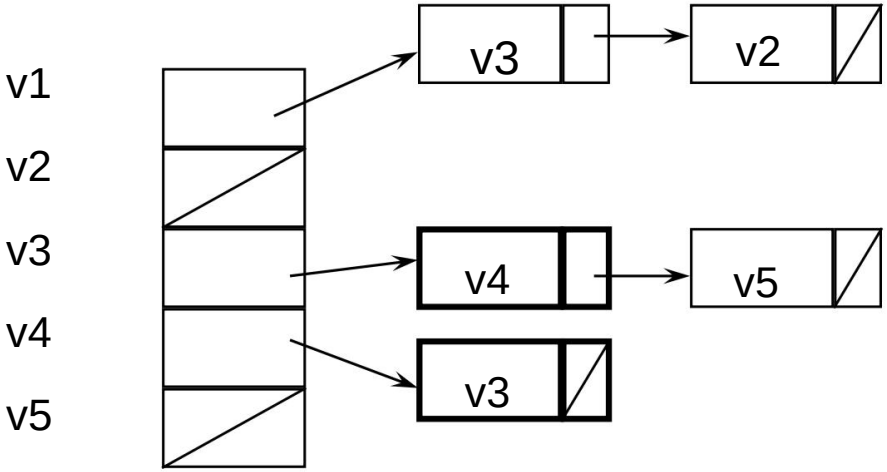




Example: Adj. List - directed graph (D5)

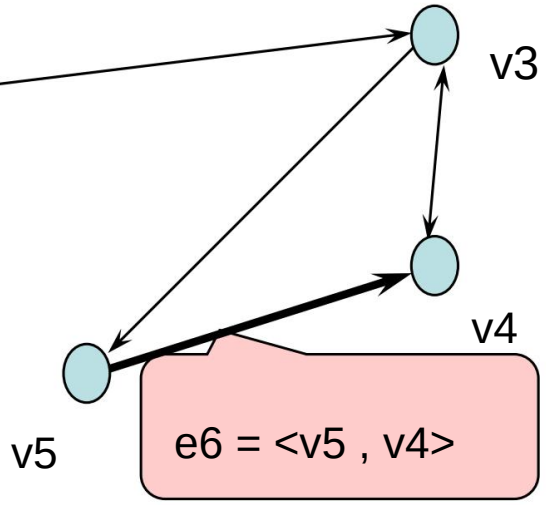
Directed graph

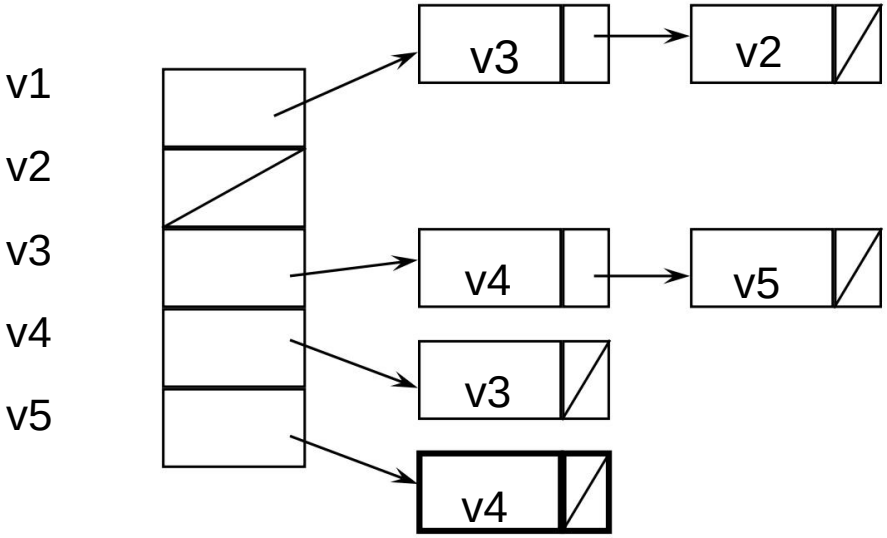




Example: Adj. List - directed graph (D6)

Directed graph





Characteristics

+ linear memory cost: O(|V|+|E|) + good for traversal, computational cost: O(|V|+|E|)

- Checking the adjacency property: O(|V|) - some algorithms more complex

6.4 Topological sorting

Problem: Starting from a binary relationship (e.g. “must be done before you can continue with”) of elements, it is necessary to clarify whether there is an order of the elements without violating any of the relationships.

Example:

Professor Bumstead gets dressed

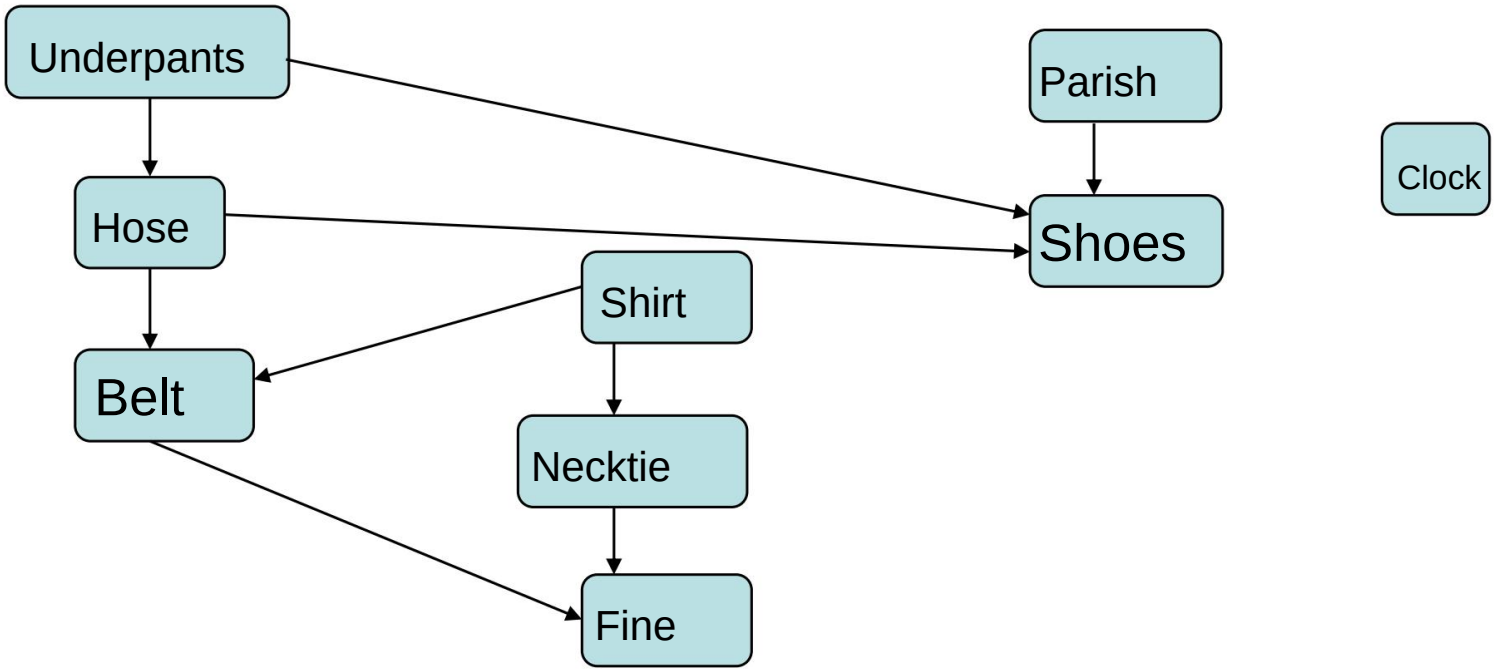
Items of clothing include: underpants, socks, shoes, trousers, belt, shirt, Tie, jacket, watch

Relationship: Garment A must be put on before B

Problem: Find a dressing order for Prof. Bumstead can attract.

Binary relationship between elements is directed edge between corresponding nodes

Prof. Bumstead



Topological order of a DAG

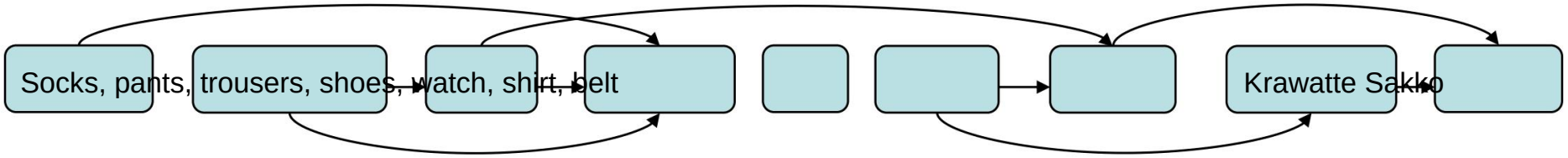
A ***topological order*** of a directed, acyclic
Graphs G ***(directed acyclic graph, DAG)*** is a linear graph
Arrange all nodes so that u comes before v in the
arrangement if G contains an edge <u, v>

If the graph is not acyclic, there is no topological order

A ***topological numbering*** of a DAG G is one
Function f: V ÿ {1, …, |V|} such that (1) f(u) ÿ f(v) if u ÿ v and
(2) f(u) < f(v) if u ÿ v and es there is a path from u to v in G.

Topological numbering implies topological ordering

Topological order: Professor Bumstead



One possible approach

Topological numbering:

1. lfdNr = 0
2. Test whether there is a node v without an incoming edge.
   If not, go to 6.

   /* If there is no such node, then the graph by Lemma 3b is not acyclic. Therefore, there is no topological order and the algorithm can stop. */
3. Arrange the node v in the topological order, i.e. increase serial number by 1 and give v number serial number.
4. Delete v from the graph
5. Go to 2.
6. If there are still nodes, give error “no topologic order exists”.

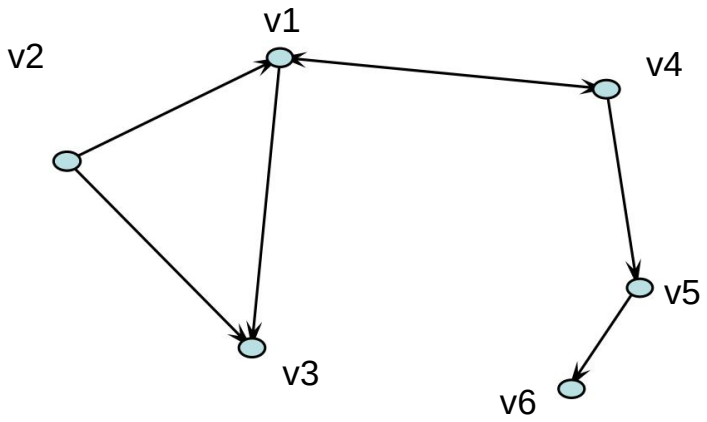
Induced subgraph

A graph = ( ) is called ***an induced subgraph (induced Subgraph, induced subgraph)*** of = (,), if

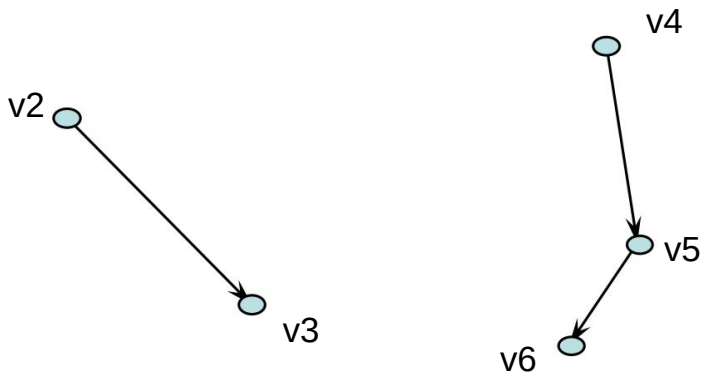
ÿ and ÿ = ÿ ÿ × ÿ }.

{ The subgraph of = (,) induced by ÿ is also written as [ ] for short.

V={v1 , v2 , v3 , v4 , v5 , v6 }



V' = V \ {v1 } = {v2 , v3 , v4 , v5 , v6 }



Topological sorting algorithm

```
lfdNr = 0;
while (ÿ v ÿ V: indegree(v) = 0) {
    lfdNr = lfdNr + 1;
    N[v] = lfdNr;
    G = G[V\{v}];                    // delete v from G
}
if (V = {}) {
    return N;                        // G is acyclic
} else {
    raise error;                     // there is no top. Order
}
```

Topological sorting algorithm
Runtime with adjacency matrix

lfdNr = 0;

// indegree(v): // ( ) Time per node

ÿ O n (2) Time per loop iteration

// 3 **n** The iteration ÿ O n ( ) Total time

while (ÿ v ÿ **V: indegree(v) = 0) {**

lfdNr = lfdNr + 1;

N[v] = lfdNr;

G = G[V\{v}]; // delete v from G

// O n ( ) Zeit for Iteration ÿ **n** Iterationen ÿ O n (2) Total time

}

if (V = {}) { return N; } // G is acyclic

else { raise error; } // no top exists. Order

Topological sorting algorithm

Implementation with *adjacency matrix representation:*

- Test at the beginning of the while loop takes time $(^{2})$.
- Removing a node takes time $( )$.
- At most iterations of the while loop

ÿ total (^{3}) Duration

Topological sorting algorithm

Implementation with *adjacency matrix representation* and additional Data structure:

NEW: store input degree of each node in array **IND**

- **Initialization** of **IND** before while loop: Calculate **IND** using the adjacency matrix A by summing the entries of the column of each node w and storing them in IND[w]) time once ÿ (^{2}
- **Test at the beginning of the while loop** runs over **IND** and looks for the first node v with IND[v] = 0 • ÿ Time( )per iteration

Remove from a node v: Set all entries in the row of v in the adjacency matrix to 0 and for every positive entry A[v,w] reduce IND[w] by 1. Set IND[v] to -1 (to show that v is no longer exists) ÿ time per iteration ( )

- At most iterations of the while loop

ÿ total (^{2}) Duration

Topological sorting algorithm

Implementation with *adjacency list representation* and additional Data structure:

Store input degree of each node in array **IND.**

• **Initialize** IND before while loop: Initialize **IND** with 0 **for** every node. Traverse all edges, i.e. all adjacency lists and increase IND[v] by 1 for each edge <u,v> ÿ + time ( )

• **Test at the beginning of the while loop** runs over **IND** and looks for the first node v with IND[v] = 0 • **Remove from a node** v: ÿ Time ( ) per iteration

Iterate over the adjacency list of v and for each node w reduce IND[w] by 1 . (Optional: put after adjlist[v] to NIL to remove v.)

ÿ ( ) ÿ( ) Time per iteration

• At most iterations of the while loop

ÿinsgesamt (+ + ^{2}) = (^{2}) Duration

Topological sorting algorithm

Implementation with *adjacency list representation* and additional

Data structures:

Store input degree of each node in array **IND.**

NEW: Store nodes with input degree 0 in a list **SRC.**

- **Initialize** SRC and **IND** before while loop: **Initialize IND**

  with 0 for each node. Traverse all edges, i.e. all adjacency lists, and increase IND[v] by 1 for each edge <u,v>. Iterate over all nodes and store each node v with IND[v] = 0 in **SRC** ÿ + time ( )

- **Test at the beginning of the while loop** checks whether **SRC** is empty and extracts otherwise any node v ÿ 1 time ( ) per iteration

- **Remove from a node** v: Iterate over the adjacency list of v and for each node w, reduce IND[w] by 1. If this reduces IND[w] to 0 , add w to the list **SRC .**

ÿ ( ) Time per iteration

- Exactly one node is processed in each iteration of the while loop

ÿinsgesamt (+ + ÿ (1 + ( )))ÿ= (+)

6.5 Traversing a Graph

This is what ***traversing*** a graph means
systematic and complete visits to all nodes of the Graphene

In principle, two approaches can be distinguished:

Depth-first search, dfs

depth-first search - Traversierung

Breadth-first search, bfs

breadth-first search - Traversierung

Almost all important problems on graphs can be solved using these two approaches , e.g

Find a path from node v to w?

Does the graph have a cycle?

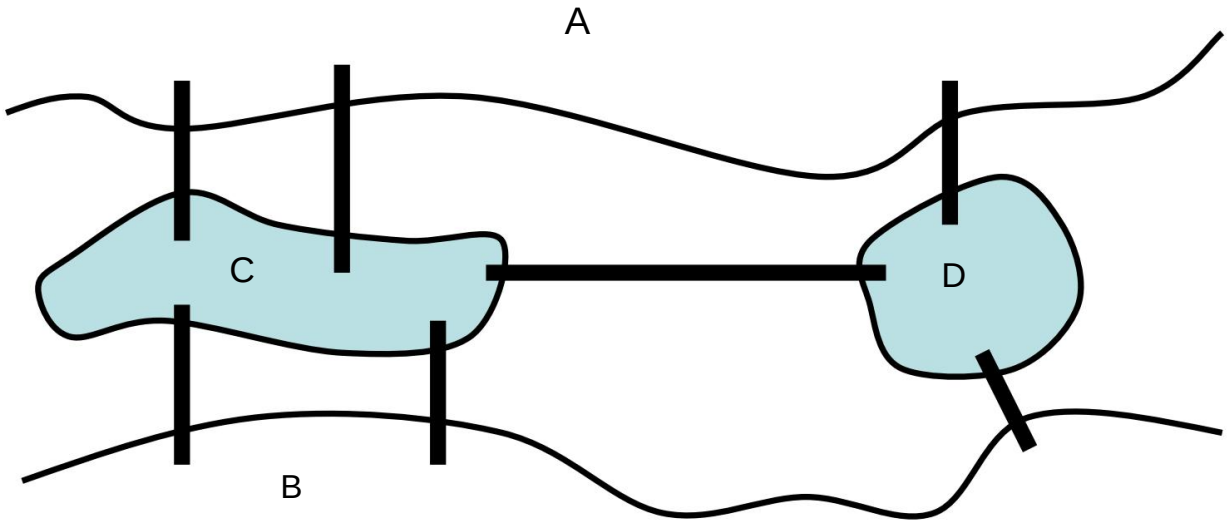
Find all components?

...

The “Königsberg Bridges” problem

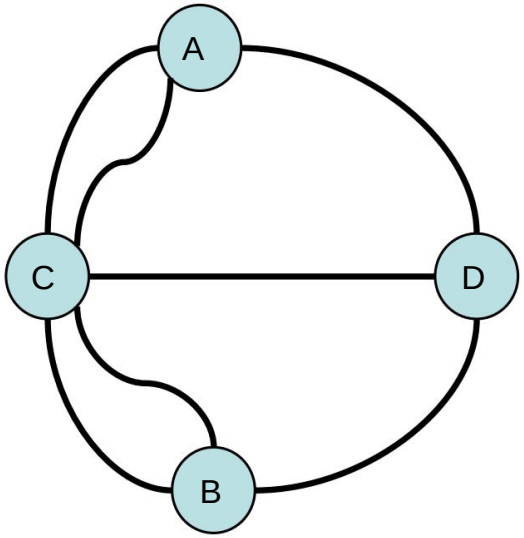
Leonhard Euler, 1736

The city of Königsberg (Kaliningrad) is located on the banks and on 2 islands of the Pregel River and is connected by 7 bridges



Question: Is there a way I can visit the city, cross all the bridges exactly once and return to the starting point?

Question: Is there a circle in the graph that has all edges exactly once contains?



Euler solved the problem by proving that such a circle is possible if and only if the graph is connected and the node degree of all nodes is even

Such graphs are called ***Eulerian graphs***

called.

This obviously does not apply to Kaliningrad.

Eulerian graphs are considered the first solved problem of Graph theory considered

Proof

Theorem: A graph is Eulerian *if and only if* it is connected and all vertices have even degrees.

Execution (ÿ): If a graph is Eulerian, it is connected and all vertices have even degree.

Each node must be entered as often as left ÿ even degree

All edges are part of the same circle ÿ graph is connected

Back direction (ÿ): When a graph is connected and all nodes have an even degree, it is Eulerian.

Proof by induction:

Induction beginning: An empty graph with 0 vertices and 0 edges trivially has an (empty) circle that contains all edges exactly once.

Hypothesis: A connected graph with < edges in which all nodes have even degrees is Eulerian.

Induction step: Let be a connected graph with > 0 edges in which all nodes have even degrees. Remove a circle (even node degree, even number of edges) from . In the remaining graph all nodes have even degrees, but...

ÿ

Proof (sketch)

Problem: ÿ could be in components ÿ 1 , 2 ÿ , … , ÿ disintegrate, but each

Component meets the conditions of the hypothesis. So the application results in circles included.

1, 2, … , which all edges of and +1: = to ÿ 1 , 2 ÿ , … , ÿ

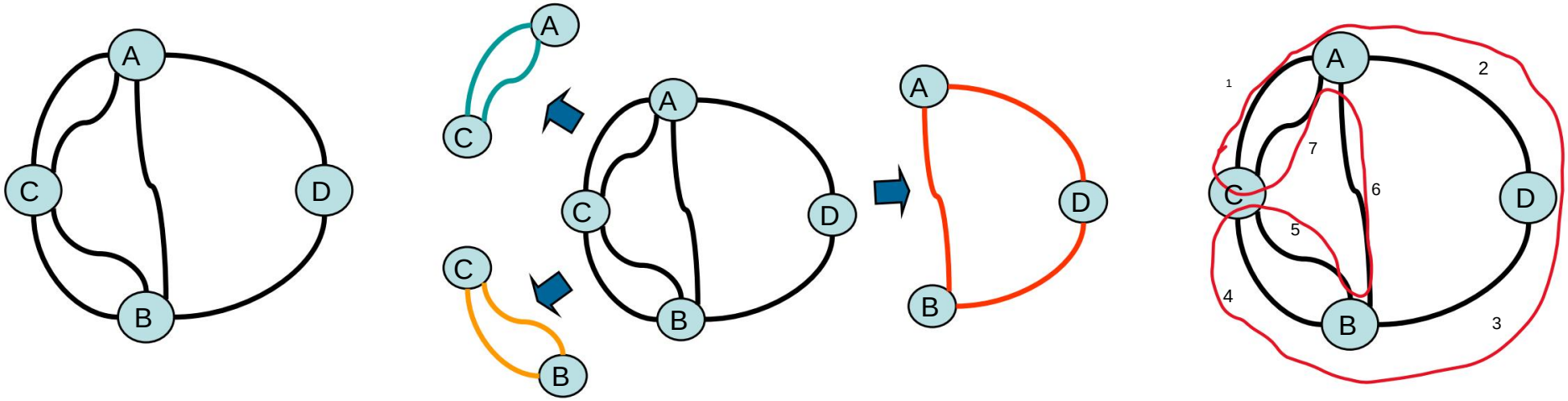
Solution: Combine 1, 2, … , form a complete circle as follows: and follow 1 until one

Start at any node in the circle to a node that is also to 1

a node that is also to ÿ (ÿ 1) belongs. From there, trace (1 ÿ

ÿ) heard, etc. If there is no node that belongs

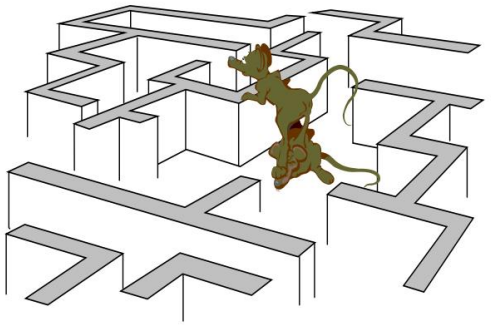
to a circle that has not yet been seen, follow the current circle to the starting point...



How do you combine the circles systematically? How do you traverse a graph? ÿ next topic!

6.5.1 Depth-first Search (DFS)

Idea



We interpret the graph as
Labyrinth, where the edges represent paths
and the nodes represent intersections .

With the depth-first search approach, we try to cover as long a new path as possible.
If we can't find a new path, we go back and try the next path that hasn't been visited yet.

How do we find the next path that has not yet been visited?
When we come to an unmarked intersection, we mark it (in the labyrinth with a stone) and remember all possible alternative paths.

If we come to an intersection that has already been marked, we go back *the same* way until we come to an intersection with an unused alternative route. This is the next path that has not yet been visited.

DFS approaches

Analogy

Read a book, search for detailed information, decision tree, Selection criteria, etc.

2 approaches

Recursive without explicit (further) data structure

Iterative (non-recursive) with stack

Can be applied to directed and undirected graphs become.

Here: undirected (directed works analogously)

6.5.1.1 Recursive approach

Recursive algorithm

nodes to be visited are (automatically) added to the recursion stack noted.

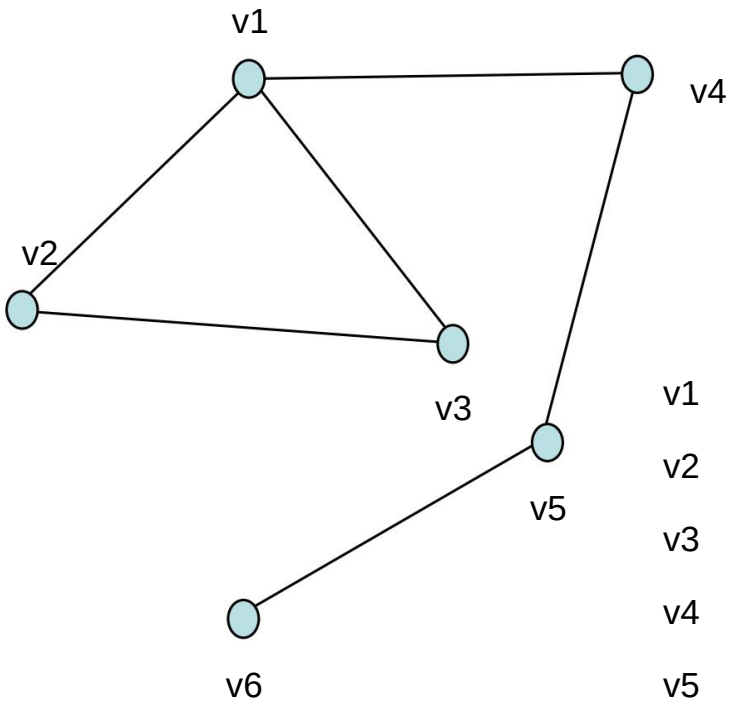
“First go deep, then go wide”

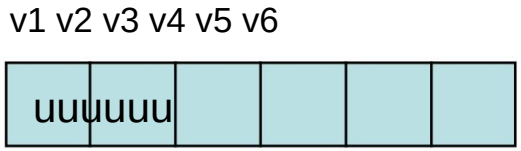
```
void visits-dfs ( node x ) {
    mark node x with 'visited';
    for (every vertex v adjacent to x)
        if (v is not yet visited)
            visits-dfs ( v );
}
```

Recursive approach

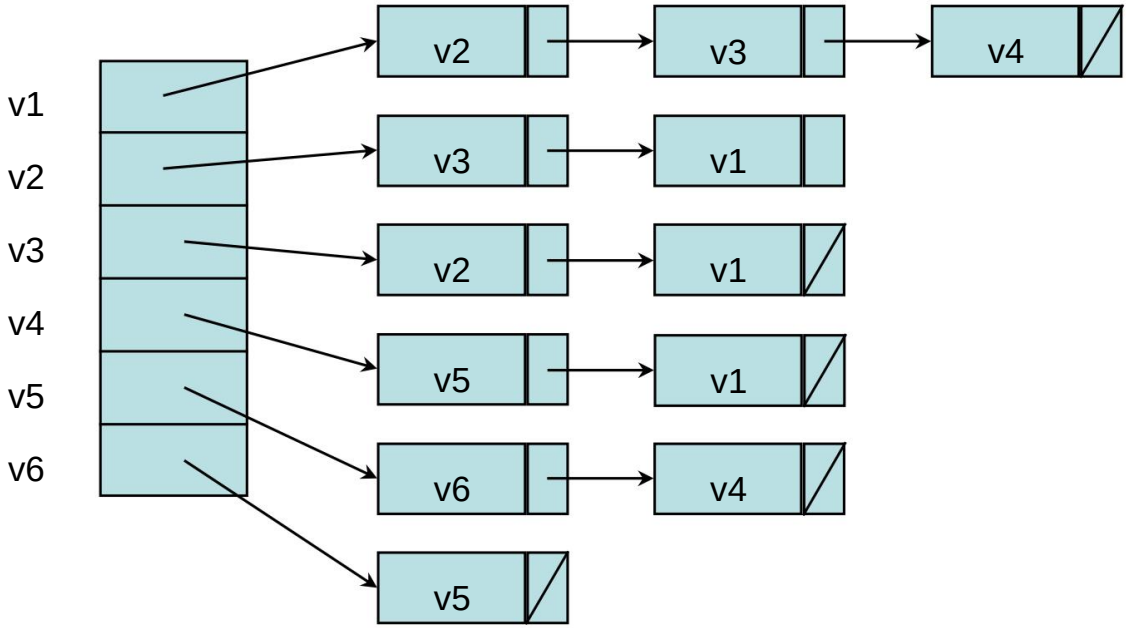
Marker: A field with the node names as an index, initialized with 'unvisited' (no stones).

Example, dfs

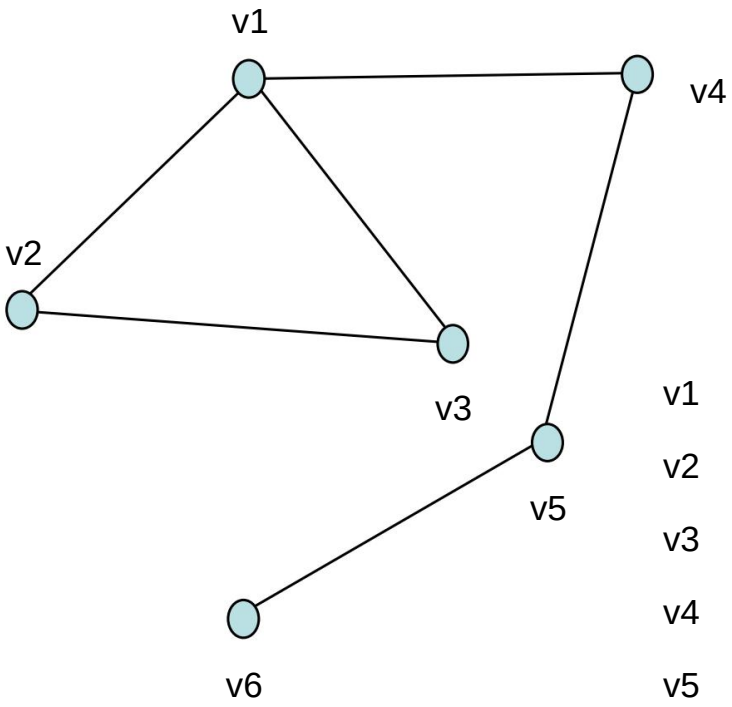




Adjazenzliste



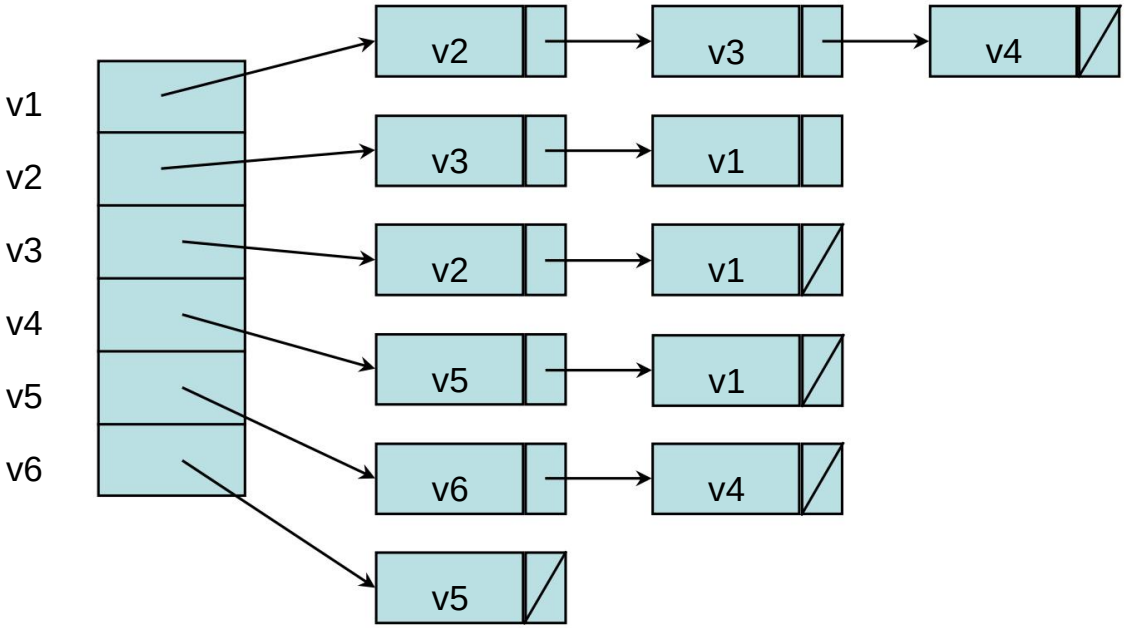
Example, dfs



v1 v2 v3 v4 v5 v6

mark

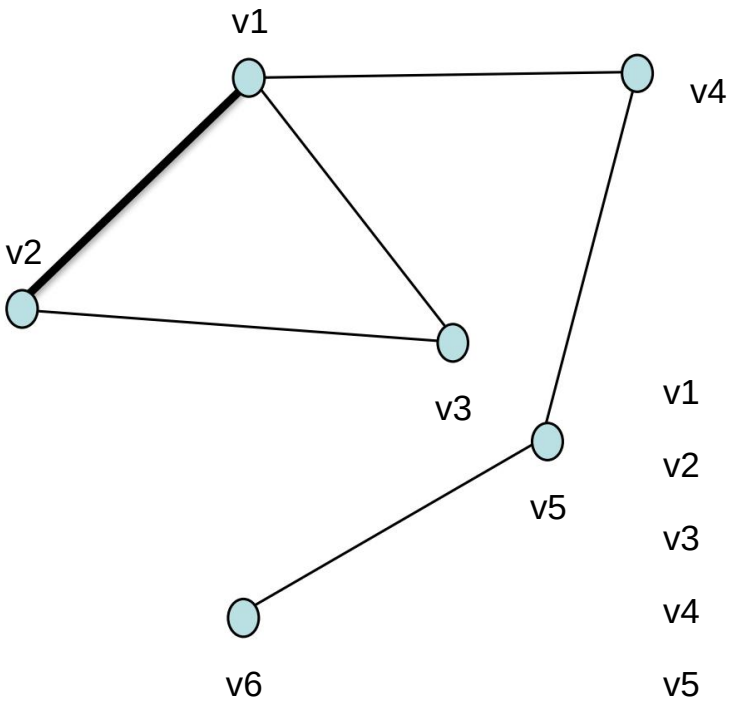
Adjazenzliste



visit-dfs (node

)

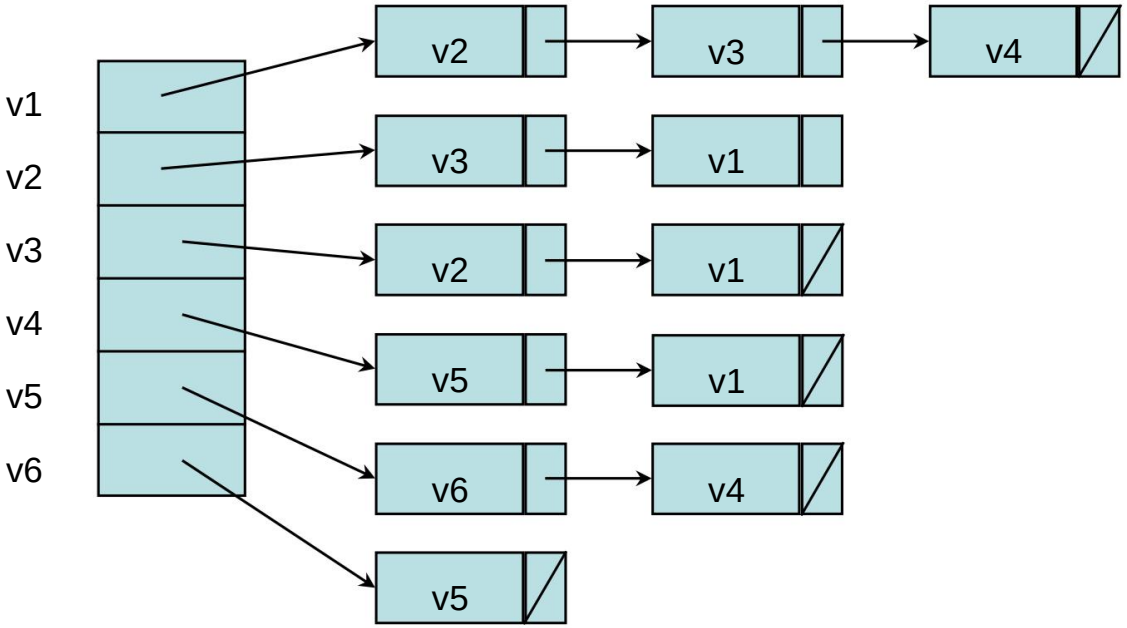
Example, dfs



v1 v2 v3 v4 v5 v6

mark

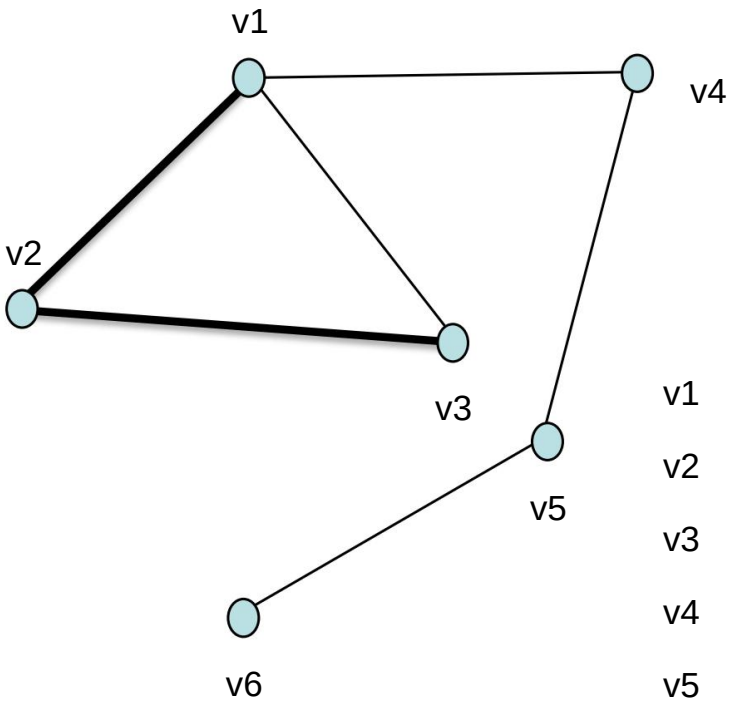
Adjazenzliste



visit-dfs (node

)

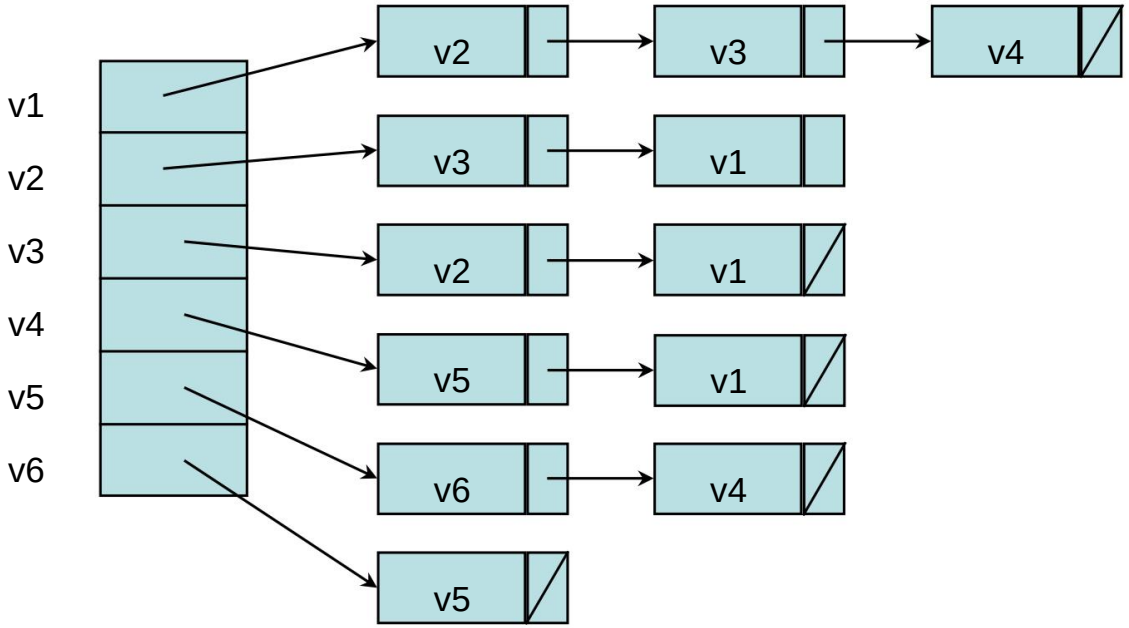
Example, dfs



v1 v2 v3 v4 v5 v6

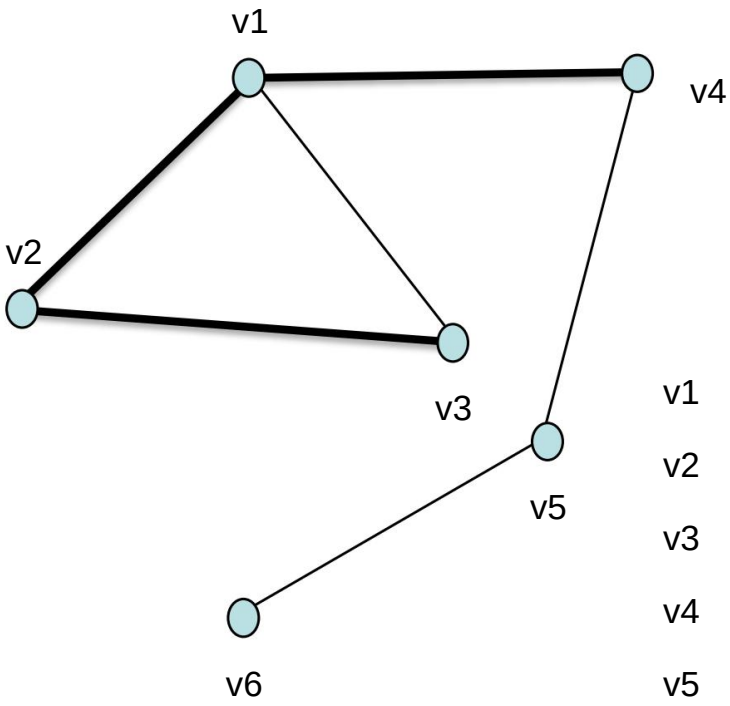
mark **b b b** u u u

Adjazenzliste



visit-dfs (node)

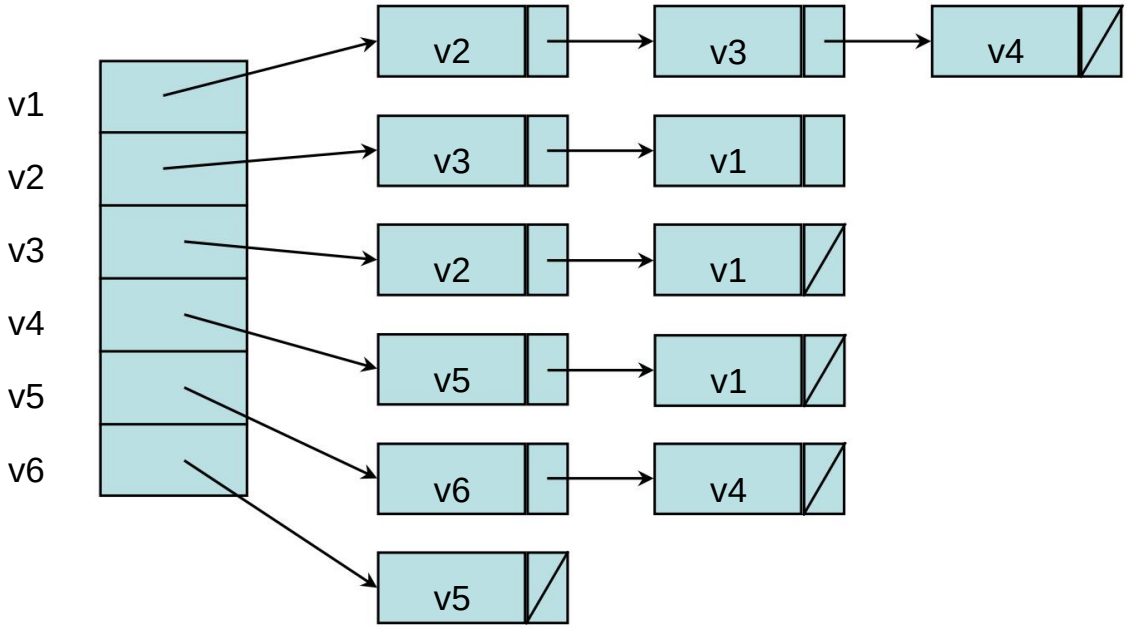
Example, dfs



v1 v2 v3 v4 v5 v6

mark **bbbb** uu

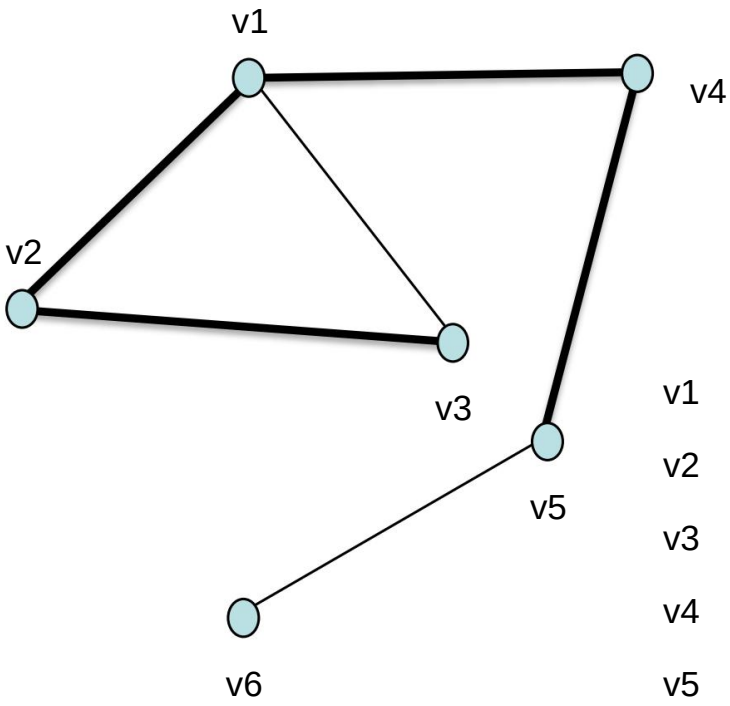
Adjazenzliste



visit-dfs (node

)

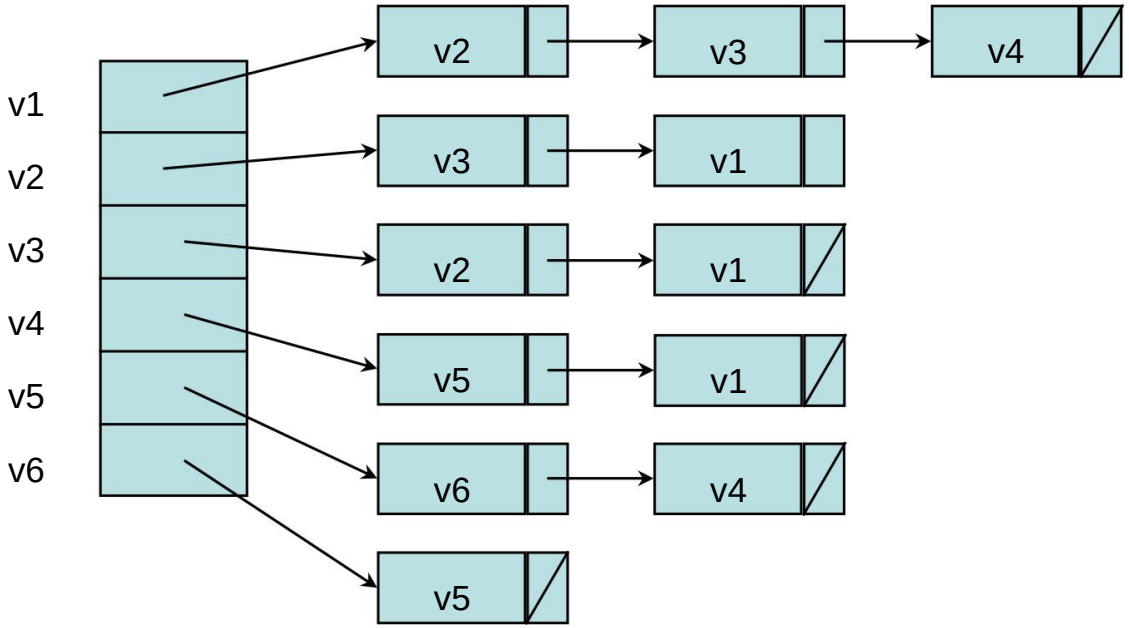
Example, dfs



v1 v2 v3 v4 v5 v6

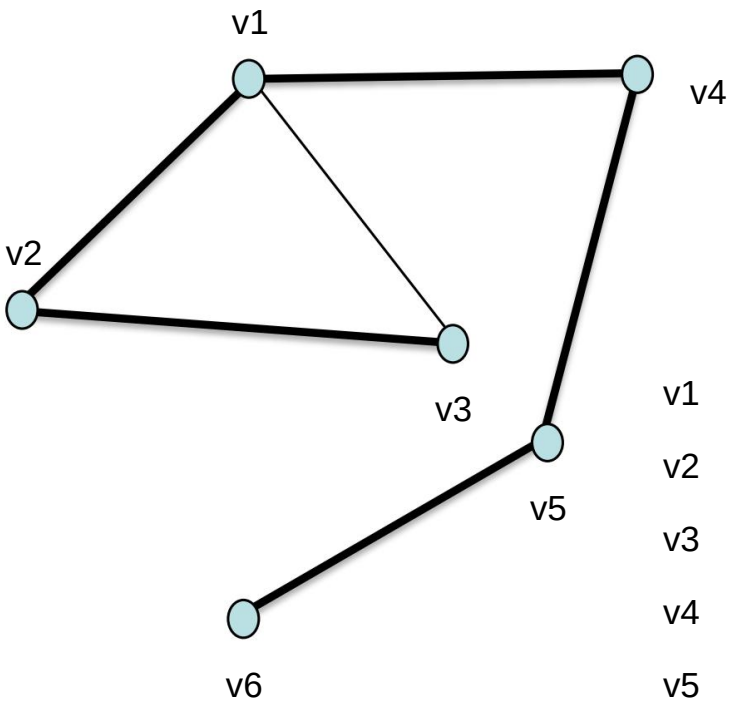
mark

Adjazenzliste



visit-dfs (node)

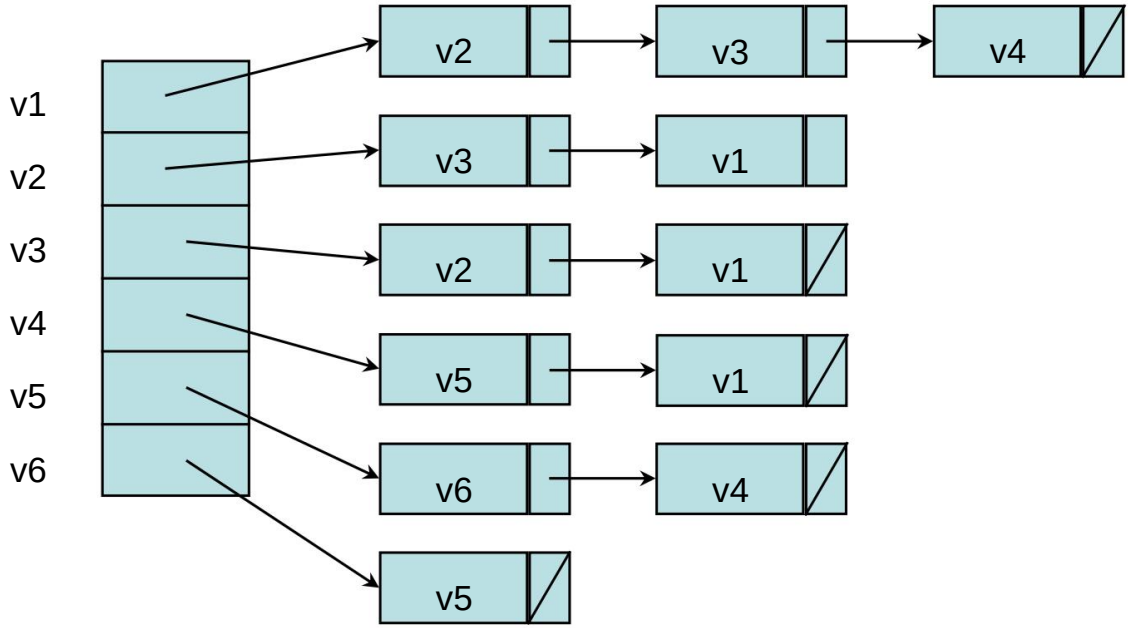
Example, dfs



v1 v2 v3 v4 v5 v6

mark **bbbbbb**

Adjazenzliste



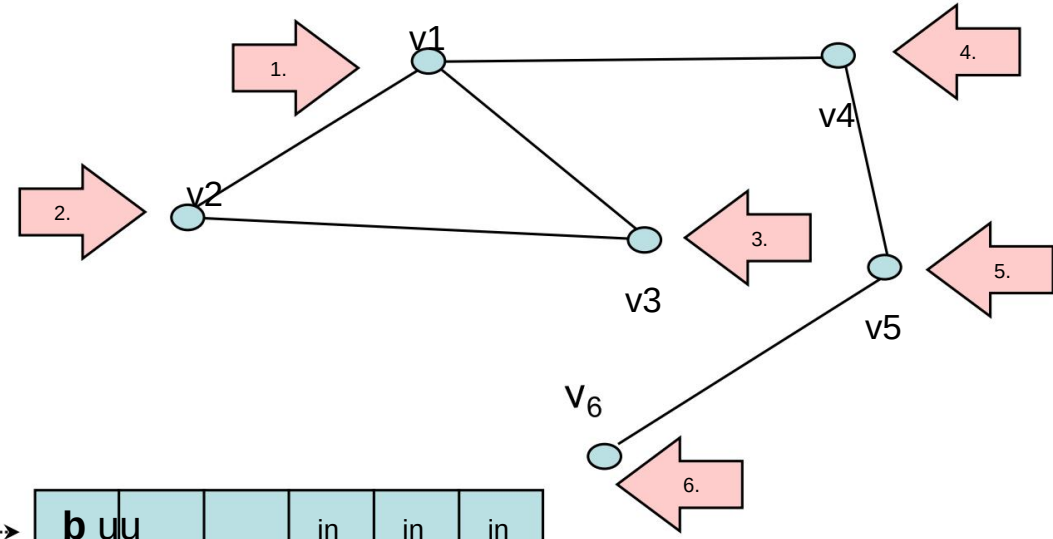
visit-dfs (node)

Recursive Method dfs

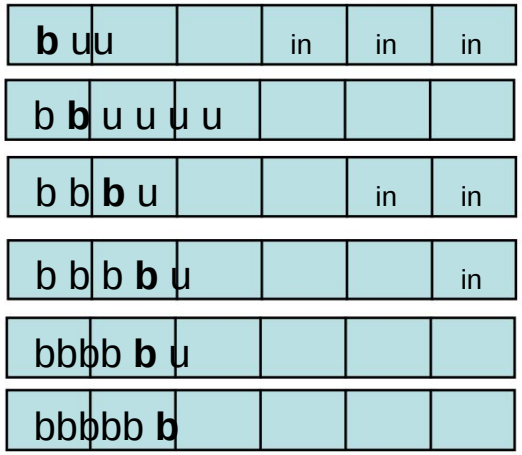
dfs traversal of a graph

Starting from node v1
the graph is traversed
using the dfs approach.

Visited nodes
are noted on the recursion
stack.



visits-dfs(1)
visits-dfs(2)
visits-dfs(3)
visits-dfs(4)
visits-dfs(5)
visits-dfs(6)



How does that know?
Program which edges from node 1 not yet are checked?

C++ code for recursive approach

```
#define visited 1 #define
unvisited 0 // maxV defined
elsewhere

class node { public: int v; node *next;} node
*adjliste[maxV]; // Adjazenzliste int mark[maxV]; //
Knotenmarkierung

void traversiere() { int k; for
    (k = 0; k
    < maxV; ++k) mark[k] = unbesucht;
        for (k = 0; k < maxV; ++k)

        if (mark[k] == unvisited) visit-dfs(k);

}

void besuche-dfs(int k) { mark[k] =
    besucht; for (node *t =
    adjliste[k]; t != NULL; t = t->next)
        if (mark[t->v] == unvisited) visit-dfs(t->v);

}
```

universität wien

Iterative approach

Nodes to visit are stored in a stack data structure,
not in the (automatic) recursion stack

```
void visit-dfs( node x ) { push node x onto
the stack; while( stack not empty ) {

    pop (pop) last node y from the stack; if ( y already
    'visited' ) continue; mark node y 'visited'; for (every
    vertex adjacent to y v ) { if ( v already
    'visited' ) continue; // optionally push v onto the stack; } //
        for } // while

}
```

Iterative approach

Iterative Methode dfs

dfs traversal of a graph

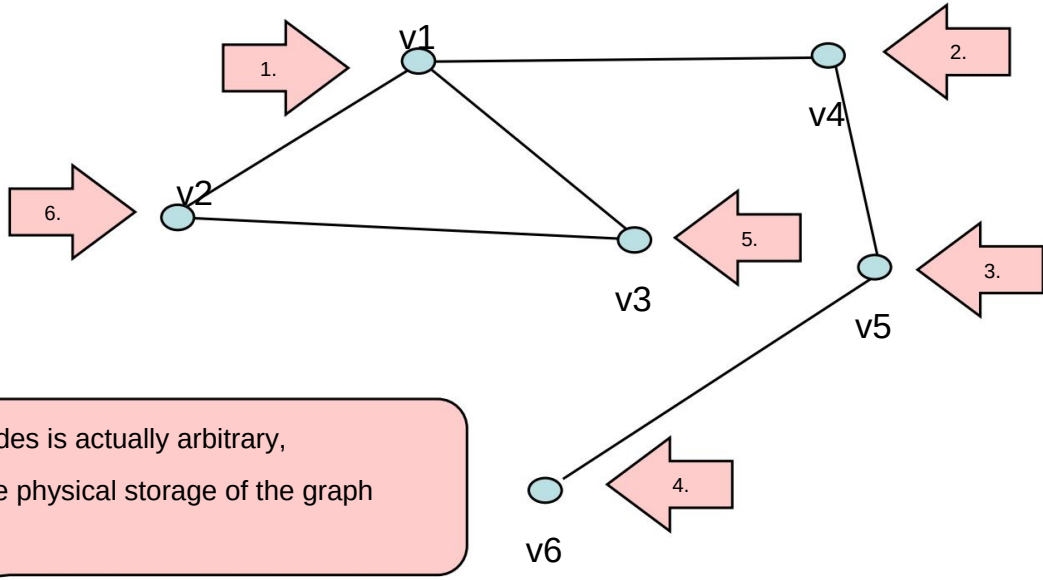
Starting from node v1

the graph is traversed using the dfs approach.

Nodes yet to be visited

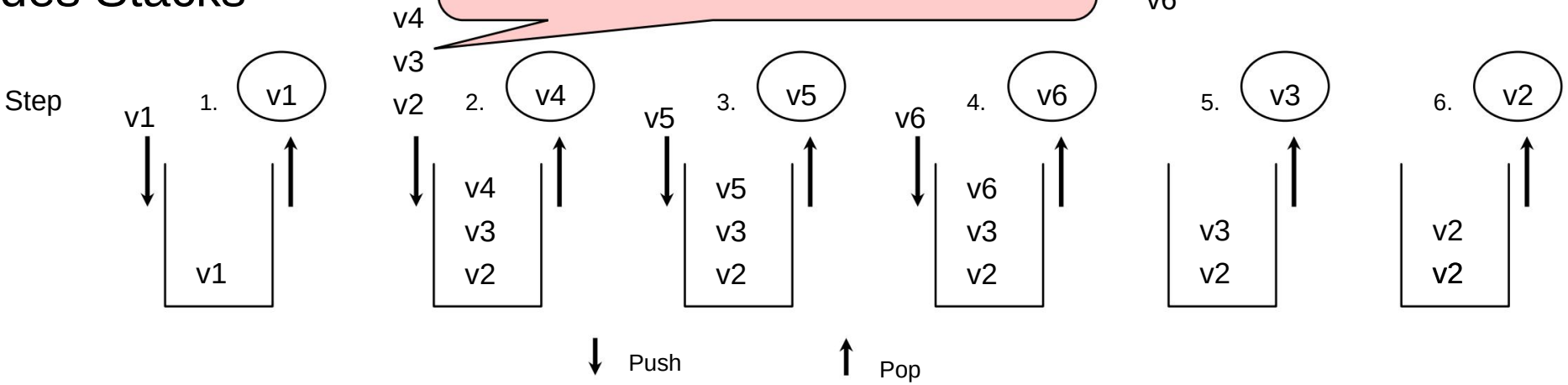
are on a stack

noted.



Behave des Stacks

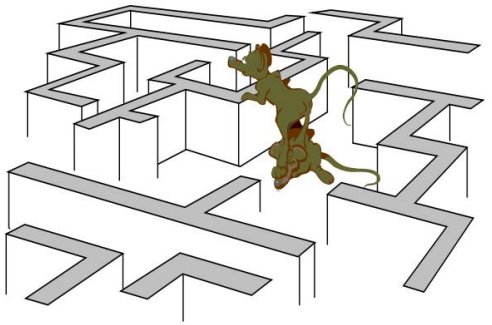
The order of the nodes is actually arbitrary, depending. from the physical storage of the graph



Non-recursive C++ code

```
// besucht, unbesucht defined as before, maxV defined elsewhere

class node { public: int v; node *next;} node
*adjliste[maxV]; // Adjazenzliste int mark[maxV]; //
Knotenmarkierung
Stack stack(maxV*(maxV-1)/2);

void traverse() { int k; for(k =
    0; k <
    maxV; ++k) mark[k] = unvisited; for(k = 0; k < maxV; ++k) if(mark[k]
    == unvisited) visit-dfs(k);


} void besuche-dfs(int k)
    { stack.Push(k);
    while(!stack.IsStackEmpty()) {
        k = stack.Pop(); if
        (mark[k] == besucht) continue; mark[k] =
         besucht; for(node *t =
        adjliste[k]; t != NULL; t = t->next){
            if(mark[t->v] == unbesucht) stack.Push(t-
                >v);
        }
    }
}
```

Effort (runtime analysis) both approaches



Method traverse without visits-dfs:

( ) Steps

Method visits-dfs:

With adjacency list:

For each node: (1 + | [ ]|) Steps to find the Successor node

Each node is visited once ÿ For all nodes: 1 + ÿ (1 + |

(p ÿ [ ]|)) = (p ÿ ÿ [ ]|) = (+)

Total: + (+ = (+))

With adjacency matrix:

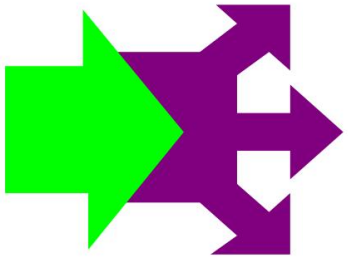
For each node: ( ) to find the successor nodes

Each node is visited once ÿ For all nodes: (2)

Total: + (2) = (2)

And ÿ (2) Adjacency lists are generally faster

6.5.2 Breadth-first Search



Idea

- You empty a pot of ink onto it
  - Start node. The ink pours in all directions (across all edges) at once
- With the breadth-first search approach, all possible alternatives are explored at once, across the entire range of possibilities
- This means that first all possible ones from a node
  - Moving away edges are examined and then the next node is moved on

Analogy

- Get an overview of the book, search for general information, hierarchical learning approach, selection overview, wave, etc.

One approach

- Iterative with queue data structure

Iterative approach

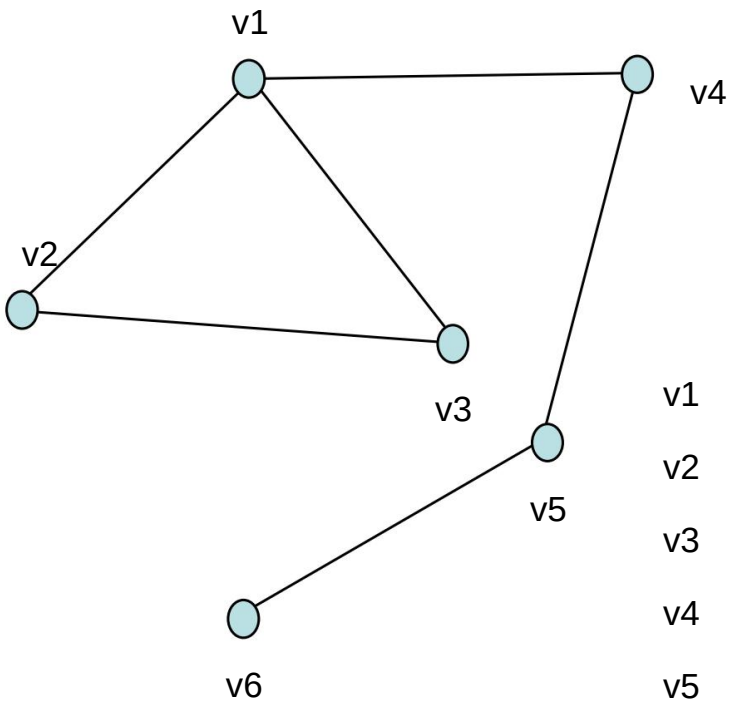
Nodes to be visited are noted in a queue

“First go broad, then go deeper”

```
void visits-bfs(node x) { place (enqueue)
    node x in queue; mark node x 'visited';
    while(Queue not empty) { get
    (Dequeue) first node y from the
        queue; for(all unvisited nodes v adjacent to y) { put (enqueue)
        v into the queue; mark node v 'visited'; } // for } // while
}
```

Iterativer Approach

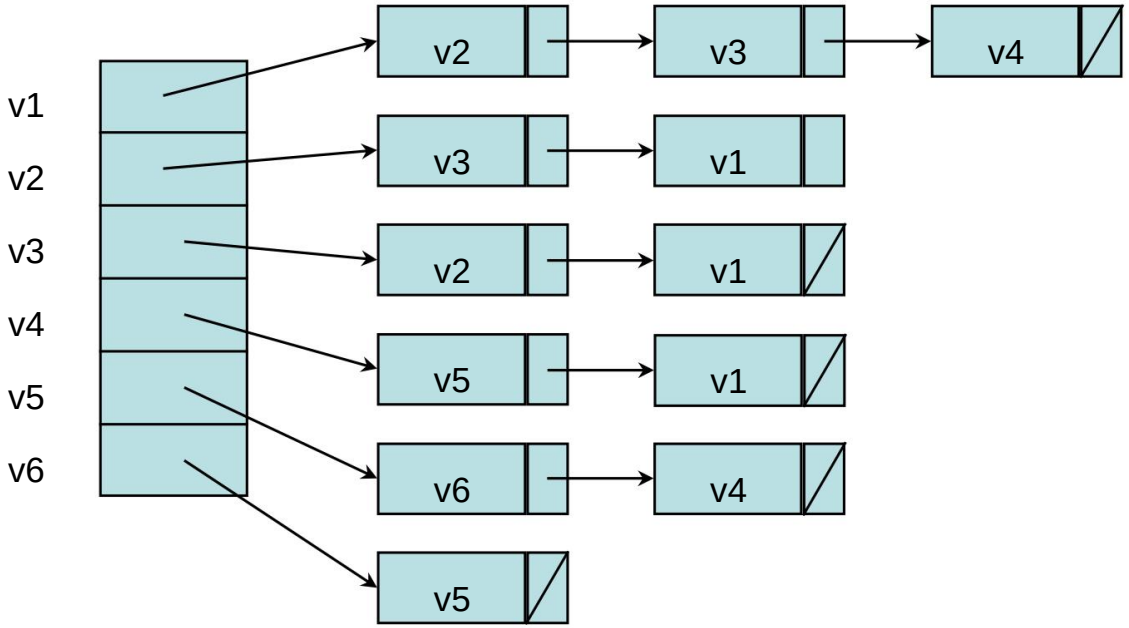
Example 1, bfs



v1 v2 v3 v4 v5 v6

mark

Adjazenzliste



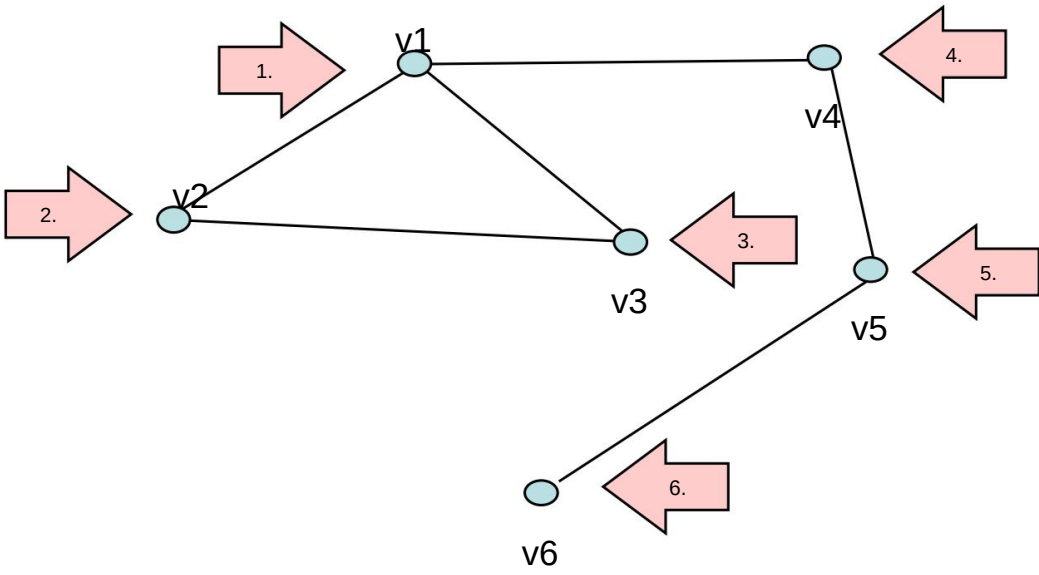
Method bfs

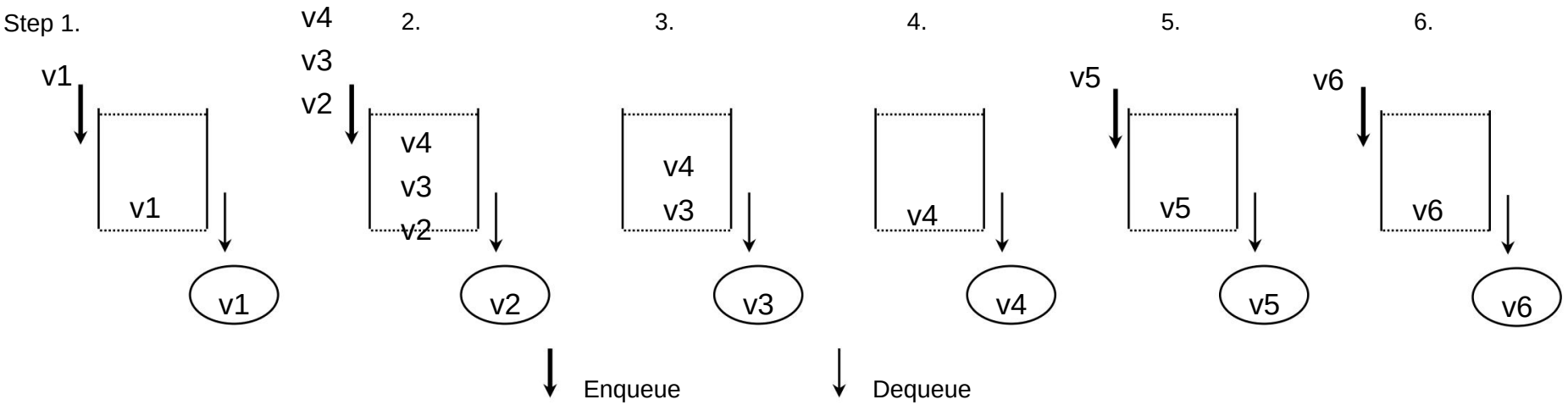
bfs traversal of a graph

Starting from node v1 , the graph is traversed using the bfs approach

Visited nodes are noted in a queue

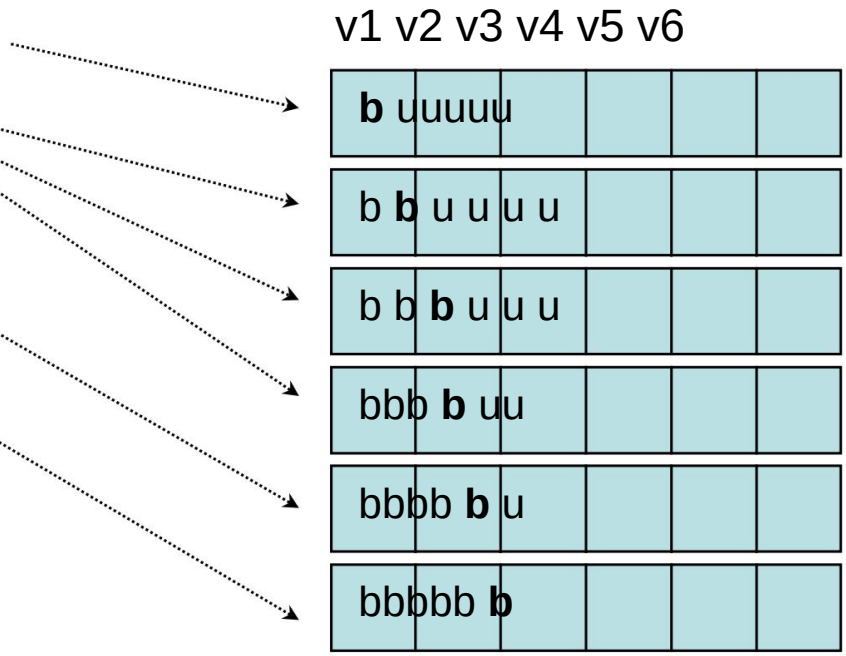
Behavior of the queue



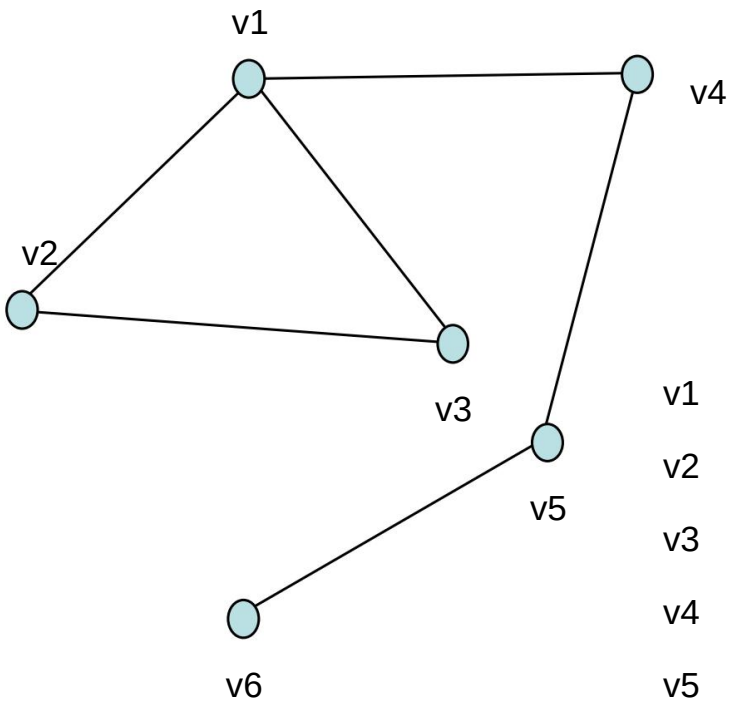


Example 1, bfs (2)

visits-bfs(1)
- Enqueue(1), Dequeue(1)
- Enqueue(2,3,4)
- Tail(2,3,4)
- Enqueue(5)
- Dequeue(5)
- Enqueue(6)
- Dequeue(6)



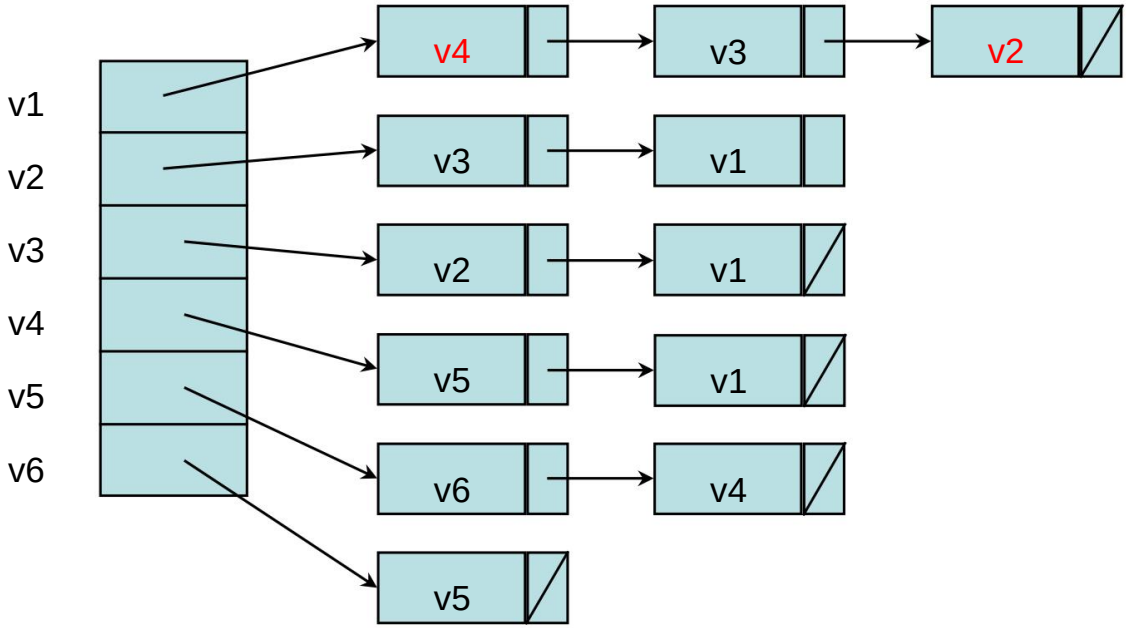
Example 2, bfs



v1 v2 v3 v4 v5 v6

mark

Adjazenzliste



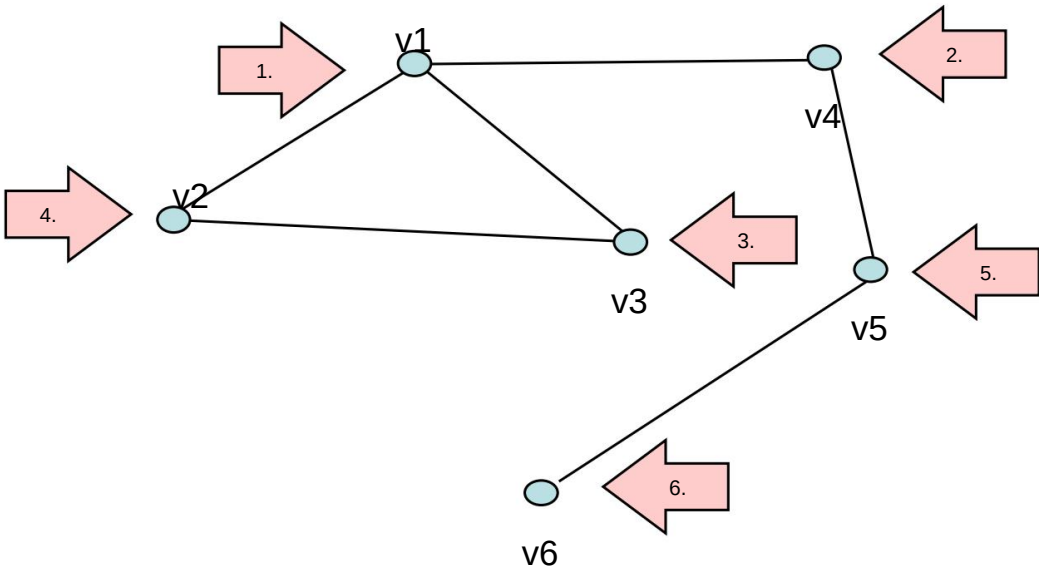
Method bfs, example 2

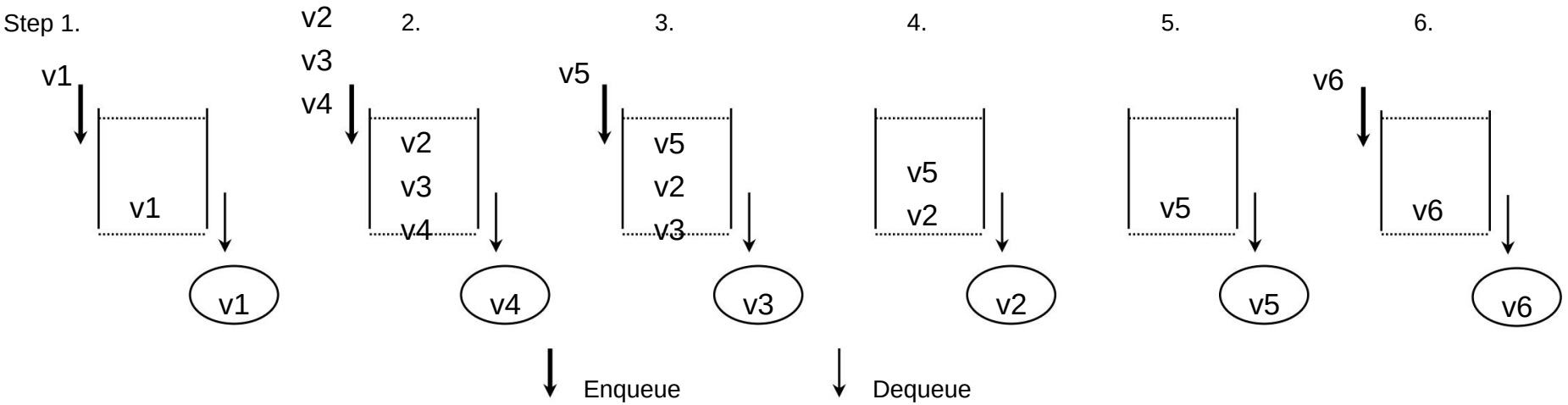
bfs traversal of a graph

Starting from node v1 , the graph is traversed using the bfs approach

Visited nodes are noted in a queue

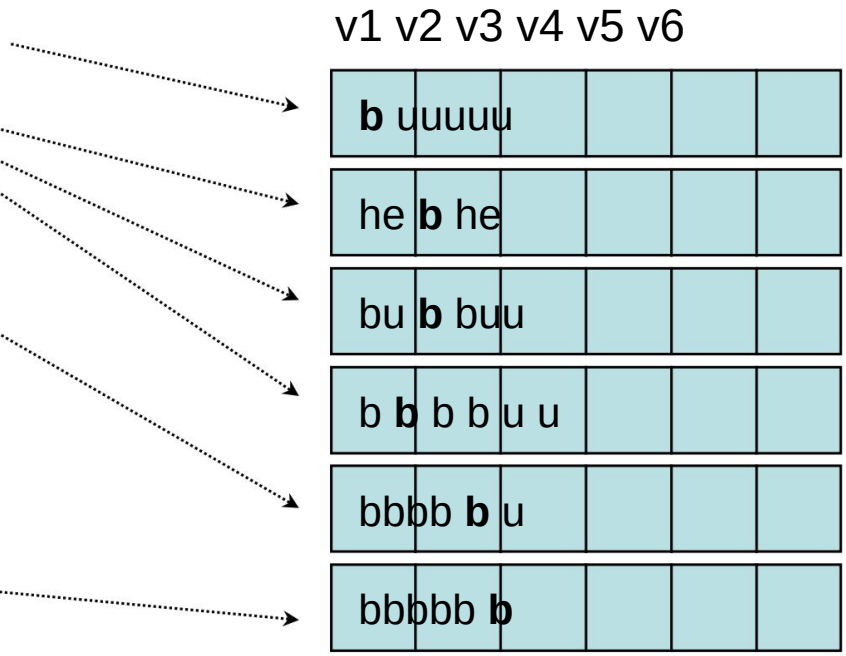
Behavior of the queue





Example 2, bfs (3)

visits-bfs(1)
- Enqueue(1), Dequeue(1)
- Enqueue(4,3,2)
- Dequeue(4)
- Enqueue(5)
  check 1 visited
- Dequeue(3)
  check 2.1 visited
- Dequeue(2)
  check 3.1 visited
- Dequeue(5)
- Enqueue(6)
  check 4 visited
- Dequeue(6)
  check 5 visited



Order of node visits in bfs traversal

Different representations of the graph lead to different order of node visits, but

Regardless of the representation, BFS visits…

- ... first all nodes whose shortest path to v1 has length 1,
- ... then all nodes whose shortest path to v1 has length 2,
- ... then all nodes whose shortest path to v1 has length 3
- ...

C++ - Code

```
// besucht, unbesucht defined as before, maxV defined elsewhere

class node { public: int v; node *next;} node
*adjliste[maxV]; // Adjazenzliste int mark[maxV]; //
Knotenmarkierung
Queue queue(maxV);
void traverse() { int k; for(k
    = 0; k <
    maxV; ++k) mark[k] = unvisited; for(k = 0; k < maxV; ++k)
    if(mark[k] == unvisited) visit-bfs(k);


} void visits-bfs(int k)
    { queue.Enqueue(k);
    mark[k]= visited; while(!
    queue.IsQueueEmpty()) { k =
        queue.Dequeue(); for(node
        *t = adjlist[k]; t != NULL; t = t->next){
            if(mark[t->v] == unvisited)
                { queue.Enqueue(t->v);
                mark[t->v] = visited;
            }
        }
    }
}
```

Effort from dfs and bfs

Even!

Why?

Same traversal principle, just different

- Data structure (stack or queue) with the same runtime for Insert and delete operations

General iterative approach to traversing a graph with Help 2 (simpler) lists

- OpenList
  - Saves known but not yet visited nodes
- CloseList
  - Saves all nodes already visited

Expand a node

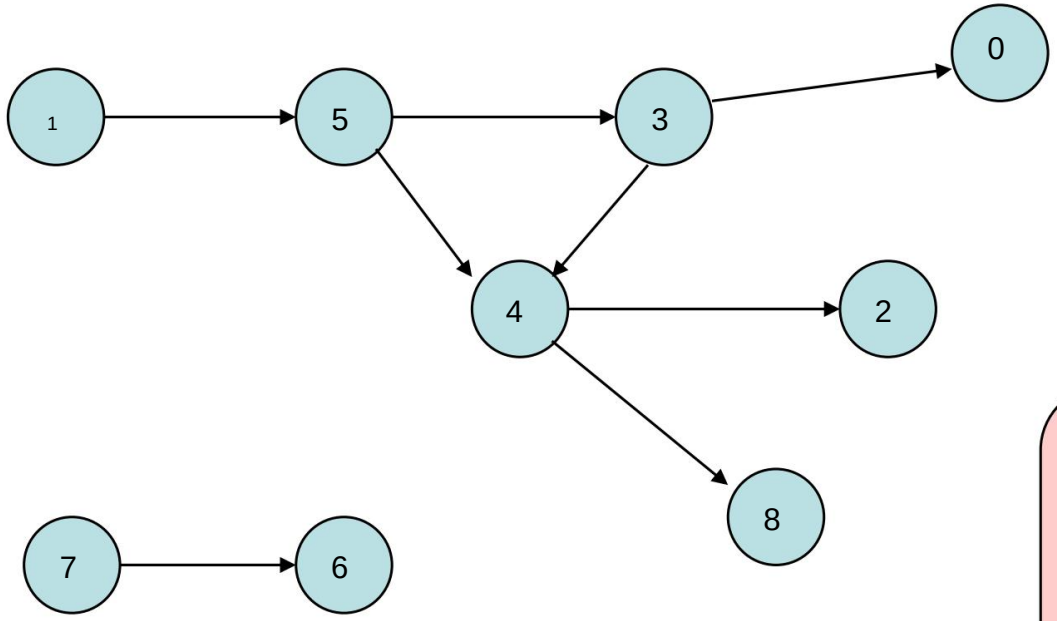
- Generate the successors of a node
- i.e. OpenList contains all generated (known) but not yet expanded nodes, CloseList contains all expanded nodes

Without further control, the traversal approach is unsystematic

Unsystematic traversal

```
Search(node v) {
    OpenList = [v];
    CloseList = [];
    while (OpenList != []) { Act =
        anyNodeFromOpenList; // ???
        OpenList = OpenList \ [Act]; CloseList =
        CloseList + [Act]; process(act); // whatever
        to do for(all nodes adjacent to act v1){

            if (v1 ÿ OpenList && v1 ÿ CloseList)
                OpenList = OpenList + [v1];
        }
    }
}
```

Example (F)



All nodes that are reachable from the start node are visited .

Difference directed – undirected graph?

z.B.: **Search(5)**

Open List: 5 | 3.4 | 4.0 | 0,2,8 | 2.8 | 8|

CloseList: 5,3,4,0,2,8

Akt: 5 | 3 | 4 | 0 | 2 | 8

Extension of **Search**

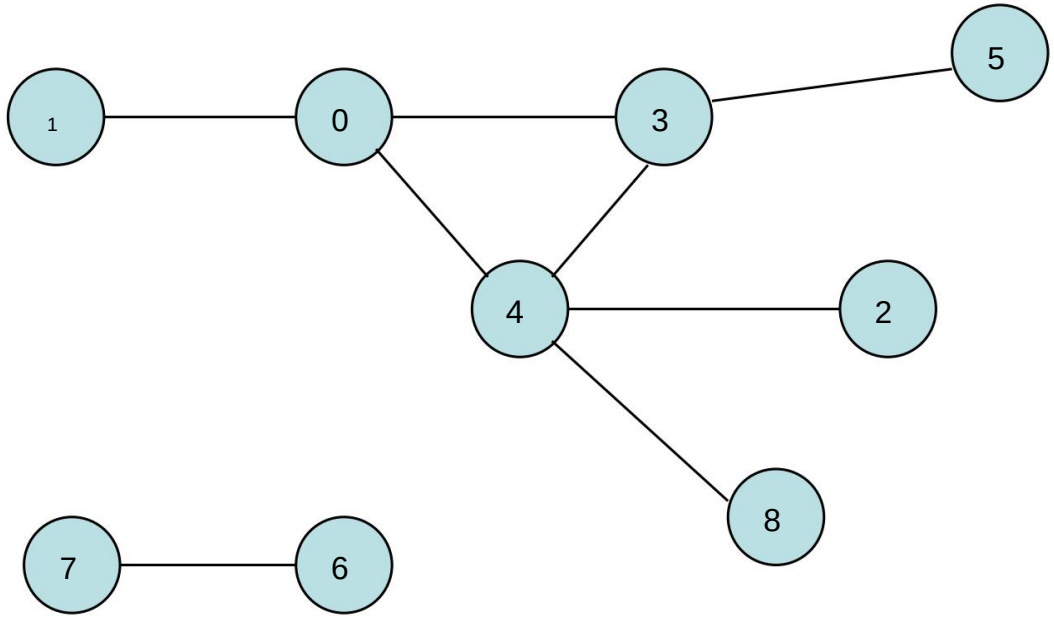
Field for noting the components K[n]

```
int K_num = 0;
for (int i = 0; i < n; i++)
    if(K[i] == 0) { // i is unvisited
        K_num = K_num + 1;
        SearchAndMark(i, K_num);
    }

SearchAndMark(node v, int K_num) {
    version of Search with
        process(Akt) { K[Akt] = K_num; }
}
```

Connective component

```
SearchAndMark(node v, int K_num) {
    OpenList = [v];
    CloseList = [];
    while (OpenList != []) {
        Act = anyNodeFromOpenList;
        OpenList = OpenList \ [Act];
        CloseList = CloseList + [Akt];
        K[Act] = K_num;
        for(all nodes adjacent to act v1){
            if (v1 ÿ OpenList && v1 ÿ CloseList)
                OpenList = OpenList + [v1];
        }
    }
}
```

Example (F)



Ask:
What about a directed graph?

OpenList: 0 | 1,3,4 | 3.4 | 4.5 | 5,2,8 | 2.8 | 2 | 6| 7

CloseList: 0,1,3,4,5,2,8 | 6,7

Akt: 0 | 1 | 3 | 4 | 5 | 2 | 8 | 6 | 7

K

0 1		2 3 4 5 6 7 8						
1	1	1	1	1	1	2 2		1

Systematization of traversal

Systematic traversal is already known

- Depth-first search dfs
- Breadth-first search bfs

Unclear program part

of any node from Open list

Systematization by organizing the selection

- Structure of the list
  - Position of the element to be inserted in the list
  - bfs: **OpenList = OpenList + [v1];** at the end
  - "pseudo" dfs/dfs-like: **OpenList = [v1] + OpenList;** at the beginning

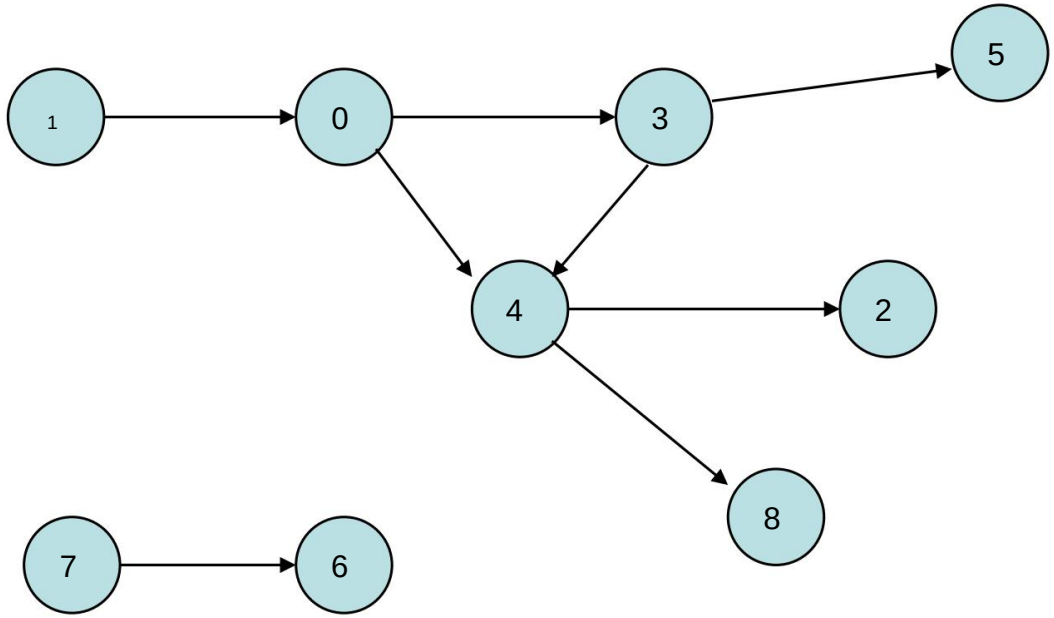
Difference to “real” dfs?

- Access to the OpenList
  - Access: **First(OpenList);** Always access the first element

Systematic traversal

```
Search(Node v) { OpenList
    = [v]; CloseList = [];
    while (OpenList != [])
    { Act = First(OpenList); OpenList
        = OpenList \ [Act]; CloseList =
        CloseList + [Act]; process(act); for(all
        nodes adjacent to act v1){

            if (v1 ÿ OpenList && v1 ÿ CloseList)
                Pseudo-dfs: OpenList = [v1] + OpenList; bfs: OpenList
                = OpenList + [v1];
        }
    }
}
```

Example (F)



z.B.: **Search(0)** mit Pseudo-dfs OpenList: 0|3,4|5,4|4|2,8|

8| CloseList: 0,3,5,4,2,8 Akt: 0|3|5|4|2|8

e.g.: **Search(0)** with bfs OpenList: 0|3.4|4.5|

5.2.8|2.8|8| CloseList: 0,3,4,5,2,8 Act: 0|3|4|5|2|8

Another small problem

Systematization not yet complete

The instruction
for(all unvisited nodes adjacent to act v1)…

Leads to undefined order of all adjacent nodes during further processing

ÿ Order defined by the internal graph storage
(adjacency list, adjacency matrix)

Assumption: Adjacent nodes are entered in ascending order

Calculating an exciting one Baumes

Input: connected graph G=(V,E), node v in V **Search(node v)**
{ OpenList = [v]; CloseList
= []; T = []; while (OpenList != []) { Act =

***First(OpenList);* OpenList =**
OpenList \ [Act]; CloseList =
CloseList + [Act]; process(act); for(all
nodes adjacent to act v1){

if (v1 ÿ OpenList && v1 ÿ CloseList){
Pseudo-dfs: ***OpenList = [v1] + OpenList;*** bfs: ***OpenList***
= OpenList + [v1];
T = T + [[Akt, v1]]
}}}

}

Calculating an exciting one Baumes

If G is connected, then T is an exciting tree of G.

- Traversal with BFS: T is called *BFS tree*
- Traversal with ***real*** DFS: T is called *DFS tree*

Layers concept for problem solving

Application level:
Examples
- cheapest way from a to b -
good moves in a game

Tool level:
Examples
- Graph traversal from v1 to v2
- Topological arrangement

Implementationsebene:
Examples
- Recursive traversing
- Iterative traversal

universität wien

Examples

Find path from node x to y

Starting from x, traverse the graph until you get to y (i.e. y is marked becomes).

Find any cycle in the undirected graph

Start traversing the graph from each node. Circle is found, if you come to a marked node (you've been there before).

Find any circle in the directed graph

Start traversing the graph from each node. Circle is found if you come to a marked node.

Important for directed graphs: Markings must be set Climbing back can be deleted again.

...

6.5.4 The “Farmer, Wolf, Goat and Cabbage” Problem

Classic problem

A farmer wants to cross a river with a wolf, a goat and a cabbage. He has a small boat at his disposal, but there is only room for two.

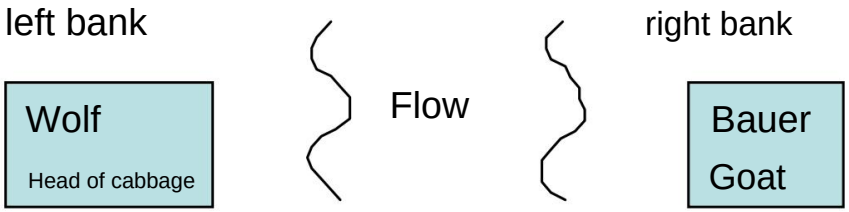
Furthermore, the problem arises that only the farmer can row, and the wolf cannot be left alone with the goat and the goat with the cabbage, otherwise one will eat the other.

A transport sequence should be found so that all of them are 'uneaten' reach other shores.

Coding the problem (1)

The problem is represented by a graph, where the nodes represent the positions of the things to be transported and the edges represent the boat trips.

A position (= location = node) defines who is on which side of the river,

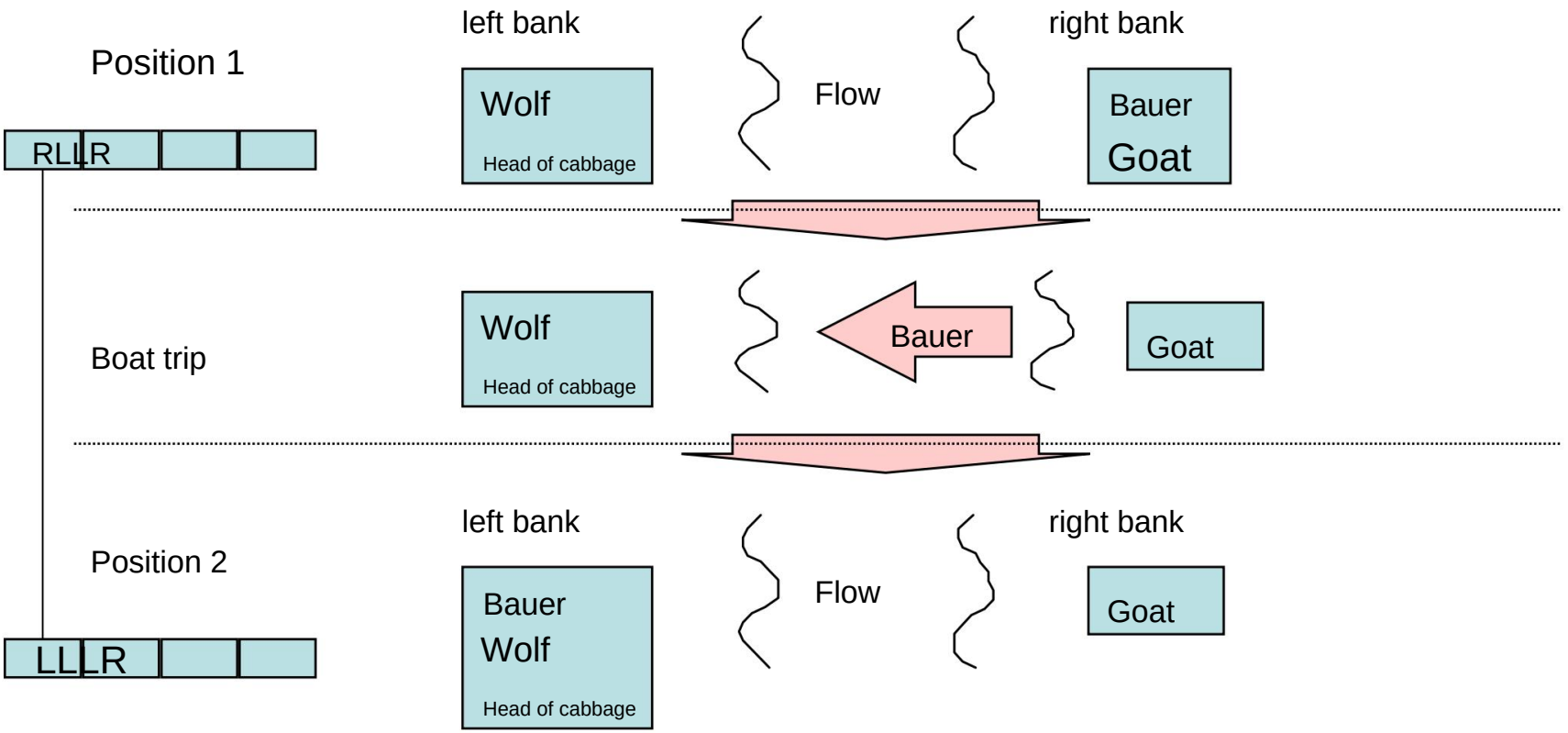


This is a 'safe' position as no one will be eaten.

A position can be encoded in a 4-element vector, where a vector element is assigned to each of the 4 items to be transported and L denotes the left bank or R the right, ie the following vector corresponds to the above position

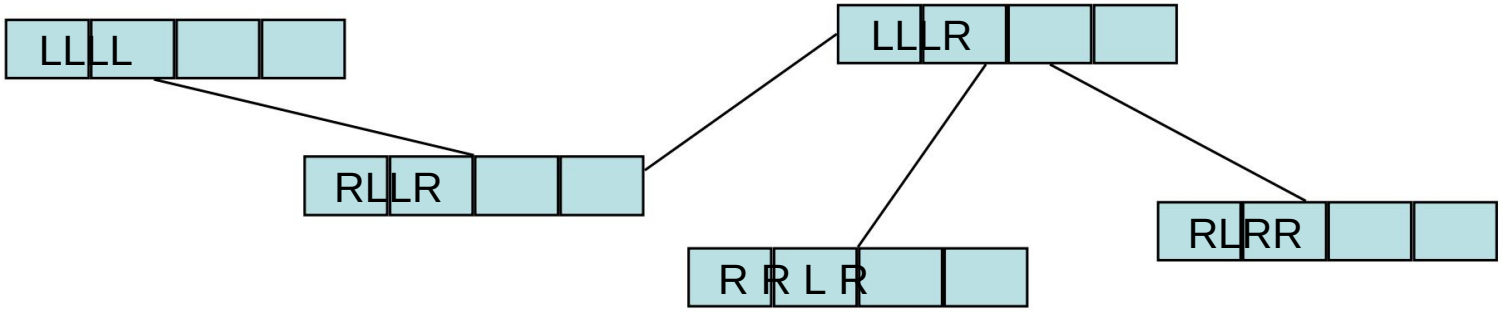
Coding the problem (2)

A boat ride (edge) indicates who in the boat translates, i.e., leads one position into the next, i.e



Problem representation

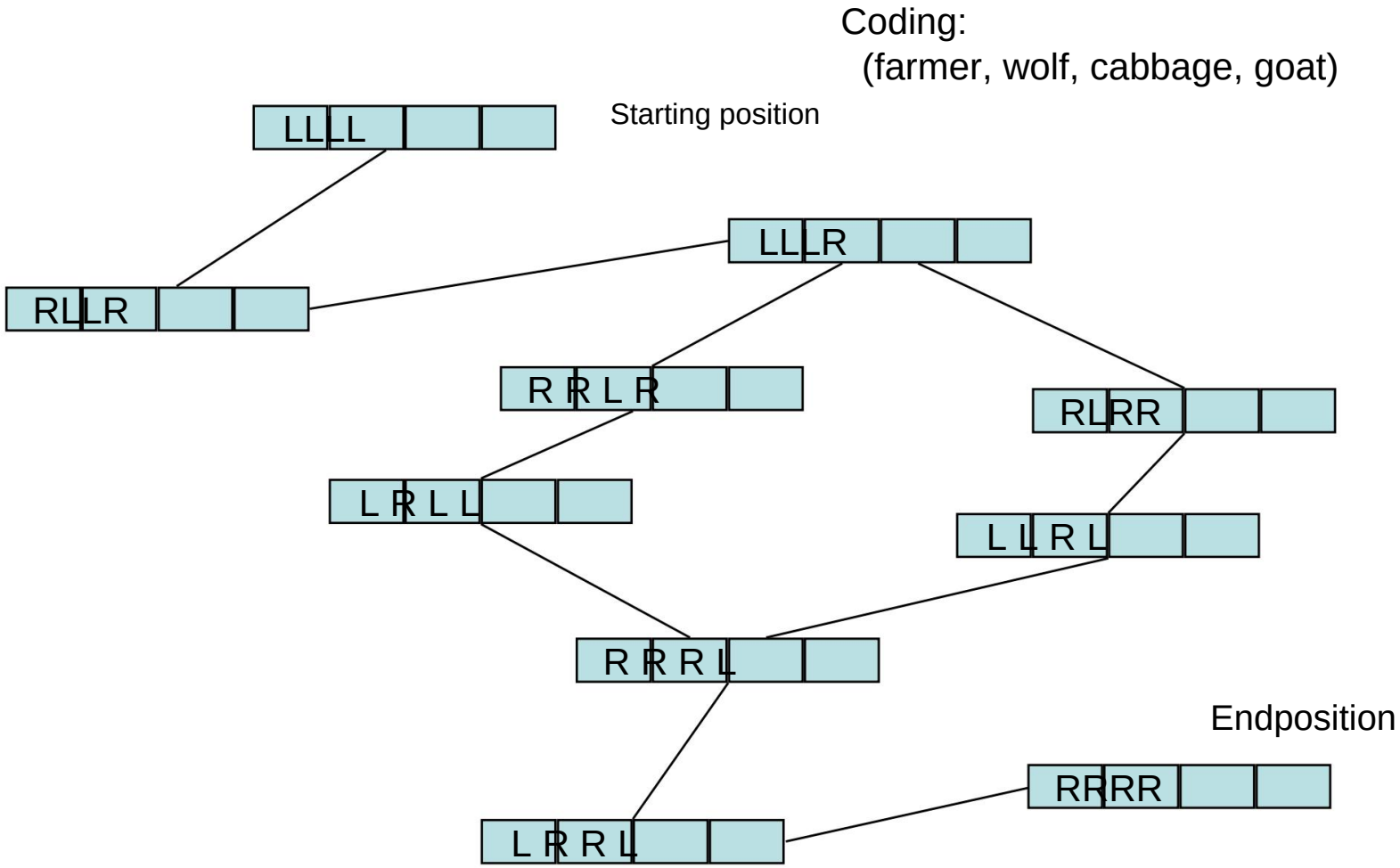
The entire problem area can therefore be solved by one Represent graphs, where the positions are represented by the nodes and the boat trips by the edges, e.g. (detail)



The solution is thus determined by a path that leads from the initial position (all 4 on the left bank = LLLL) to the final position (all 4 on the right bank = RRRR).

This can be solved by a simple traversal (e.g. DFS or BFS) of the graph.

solution



C program

We choose a bfs approach (iterative, using a queue)

- The position is encoded in binary (place value) in an integer variable
  - Farmer 3rd bit, wolf 2nd bit, cabbage 1st bit, goat 0th bit, where 0 means left bank (L) and 1 means right bank (R). Example: **1001** right: farmer, goat
- The functions **farmer, wolf, cabbage, goat** return the current position: **0** (left) or **1** (right)
- The **safe** function determines whether a position makes sense, i.e. no one gets eaten.
- The **PrintLocation** function is used to output the positions in a comprehensible manner.
- The nodes to be visited are noted in the **train** queue.
- The path taken (traversal) is saved in the **Path** field (max. 16 positions).
  - C specialties 0x08 Hexadecimal representation of a number

^	bitwise exclusive or, XOR
&	bitwise And
|	bitwise or
<<	Left shift operator

```
int Ort = …;
if (Ort & 0x08 == 0) {
        // test pawn bit, pawn is on the right
} else { … }
    // only pawn changes side
int nOrt = Ort ^ 0x08;
    // 0001 : goat bit
int Pers = 1;
    // Farmer and goat change sides
nOrt = Ort ^ (0x08 | Pers);
    // 0010 : Cabbage bit
nPers = Pers <<= 1;
```

Auxiliary functions

```
int Bauer(int Ort) {return 0 != (Ort & 0x08);}
// Location & 0x08 == 1 if pawn is on the right, == 0 if pawn on the left
      is

int Wolf(int location) {return 0 != (location & 0x04);} int Kohl(int
location) {return 0 != (location & 0x02);} int Goat(int location) {return
0 != (location & 0x01);}

bool sure(int location)
      { if((goat(location) == cabbage(location)) &&
            (Goat(Location) != Farmer(Location))) return false;
      if((Wolf(Place) == Goat(Place)) &&
            (Wolf(Location) != Farmer(Location))) return false;
      return true;
}

void DruckeOrt(int Ort) {
      cout << ((Ort & 0x08) ? "R " cout << ((Ort     : "L ");
      & 0x04) ? "R " cout << ((Ort & 0x02) ? "R     : "L ");
      " cout << ((Ort & 0x01) ? "R " cout <<         : "L ");
      endl;                                         : "L ");


}
```

Auxiliary functions

```
int Page(int Location, int Pers) if (Pers
      == 8) return Location & 0x08;
// if pers = pawn and pawn on the right, == 0 if pawn on the left
      is
       if (Pers == 4) return Ort & 0x04;
       if (Pers == 2) return Ort & 0x02;
       if (Pers == 1) return Ort & 0x01;
}
```

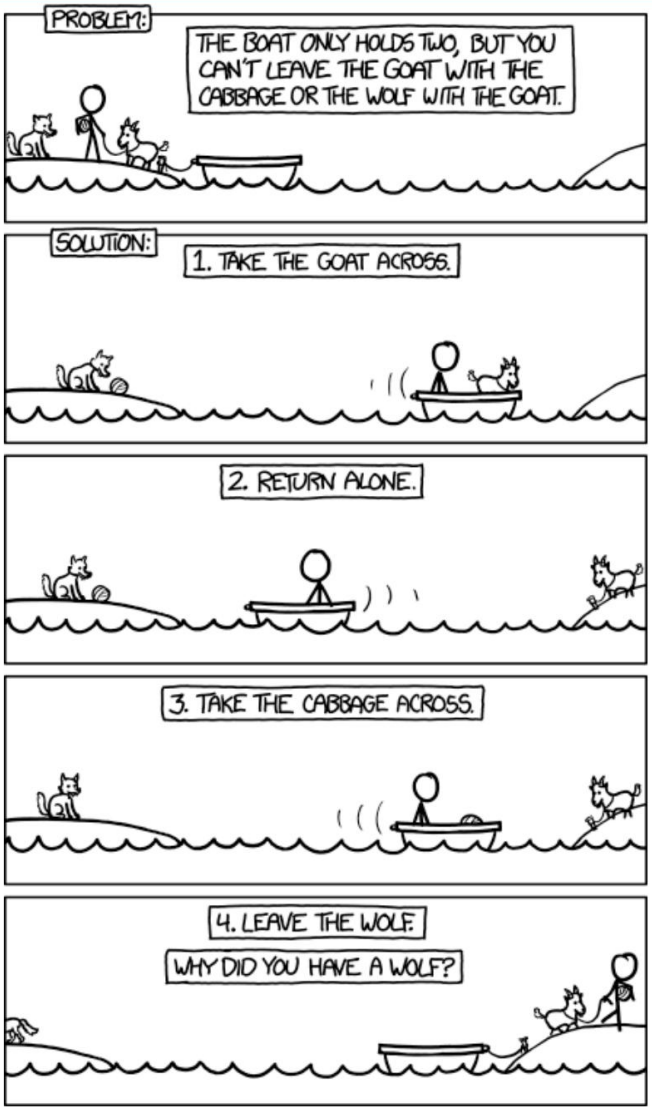
traversal

```
void main() {
    Tail<int> Zug; // BFS tail
    int way[16];                    // specifically path from LLLL to RRRR
    for(int i = 0; i < 16; i++) way[i] = -1; // Initialization
    Train.Enqueue(0x00); // start at LLLL
    while(!Zug.IsEmpty()) {
      int Location = Train.Dequeue(); // current node
      for (int Pers = 1; Pers <= 8; Pers <<= 1) { // create adjacent edges: Pers only takes
          values 1, 2, 4, 8
            if (Page(Place, Pers) != Pawn(Place)) continue; // Pers not on the same side as
          Pawn, so no edge that changes Pers' position can exist

                                          ^     (0x08 | pers); // neighboring node: int nOrt = location
                Farmer and pers change sides
            if(sure(nPlace) && (Way[nPlace] == -1)) {
            // Edge exists and nOrt not yet visited
                  Way[nLocation] = Location;
                  Train.Enqueue(nLocation);
            }
      }
    }
}
```

traversal

```
void main()
    { Queue<int> Zug; int
    Weg[16]; for(int i
    = 0; i < 16; i++) Weg[i] = -1; Zug.Enqueue(0x00); while(!
    Zug.IsEmpty()) {

      int Ort = Zug.Dequeue(); for (int
      Pers = 1; Pers <= 8; Pers <<= 1) {
            if (Page(Place, Pers) != Pawn(Place)) continue; int nOrt = Location
            if(sure(nOrt) &&              ^     (0x08 | Pers);
            (Way[nLocation] == -1)) { Way[nLocation] = Location;
                  Train.Enqueue(nLocation);


            }

      }

    }
// Output of the solution cout <<
    "Path:\n"; for(int location
    = 15; location > 0; location = path[location]) printLocation(location); cout << '\n';


}
```

The “optimal solution”



https://xkcd.com/1134/

6.6 Minimal Spanning Tree

An edge-weighted graph = (, ,) is an undirected or directed graph in which the weight function assigns a value to each edge, i.e. for each edge (ÿ) there is a weight (,), which indicates the costs/values of the edge. [.] ÿ
(.)

A ***Minimum Spanning Tree MSB (minimum spanning tree)*** is a spanning tree of a **connected,** undirected graph whose sum of edge values is minimal.

Goal: Find an MSB, i.e. an acyclic subset ÿ that connects all nodes and for which:

( ) = ÿ[,]ÿT (,) is a minimum

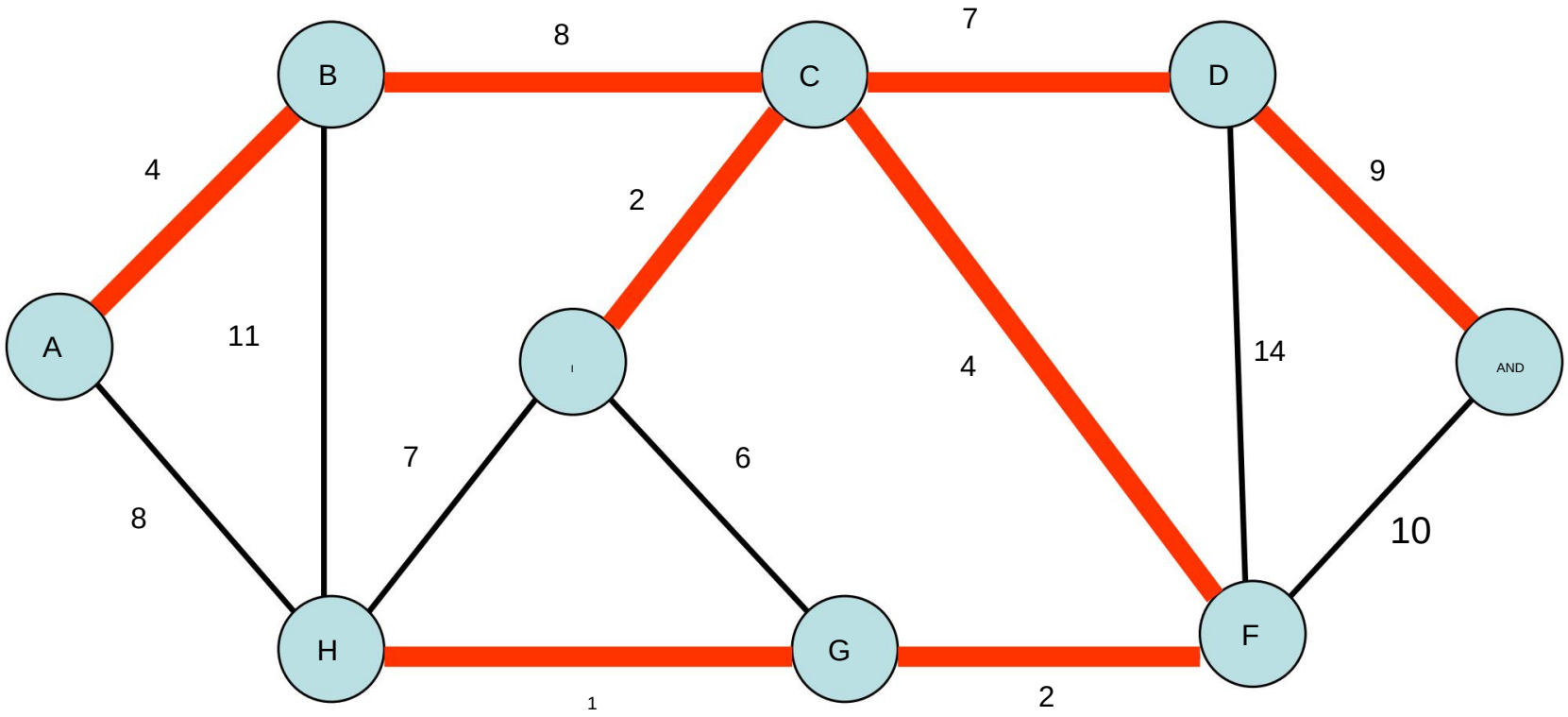
Example:

Wiring problem in an electronic circuit diagram

All pins must be wired together, keeping the wire length to a minimum

i.e. V is the set of pins, E is the set of possible connections between pin pairs, w(u,v) wire length between pin u and v

Example: Minimal Spanning Tree



Weight of MSB: 37

MSB is not unique, edge [B, C] could be replaced by [A, H].

Generic MSB algorithm

Approach through greedy algorithm

1. Algorithm manages a set A of edges, which is always one subset of a possible MSB
2. MSB is created step by step, edge by edge, where for each edge checks whether it can be added to A without violating condition 1.

Such an edge is called ***a "safe edge for*** A" : an edge e such that A ÿ {e} is a subset of a possible MSB

How do I find such a safe edge?

```
Generic-MSB(G=(V, E, w)) {
    A = {};
    while (A does not form an MSB) {
       find an edge [u,v] that is "safe" for A
       A = A ÿ { [u,v] }
    }
}
```

A **forest** is an acyclic graph.

Difference Between Forest and Tree:

A tree must connect all nodes in the graph, i.e. be exciting.
A forest does not have to connect all nodes.

6.6.1 Kruskal algorithm

Basic idea:

The set of nodes represents a forest consisting of |
| Components

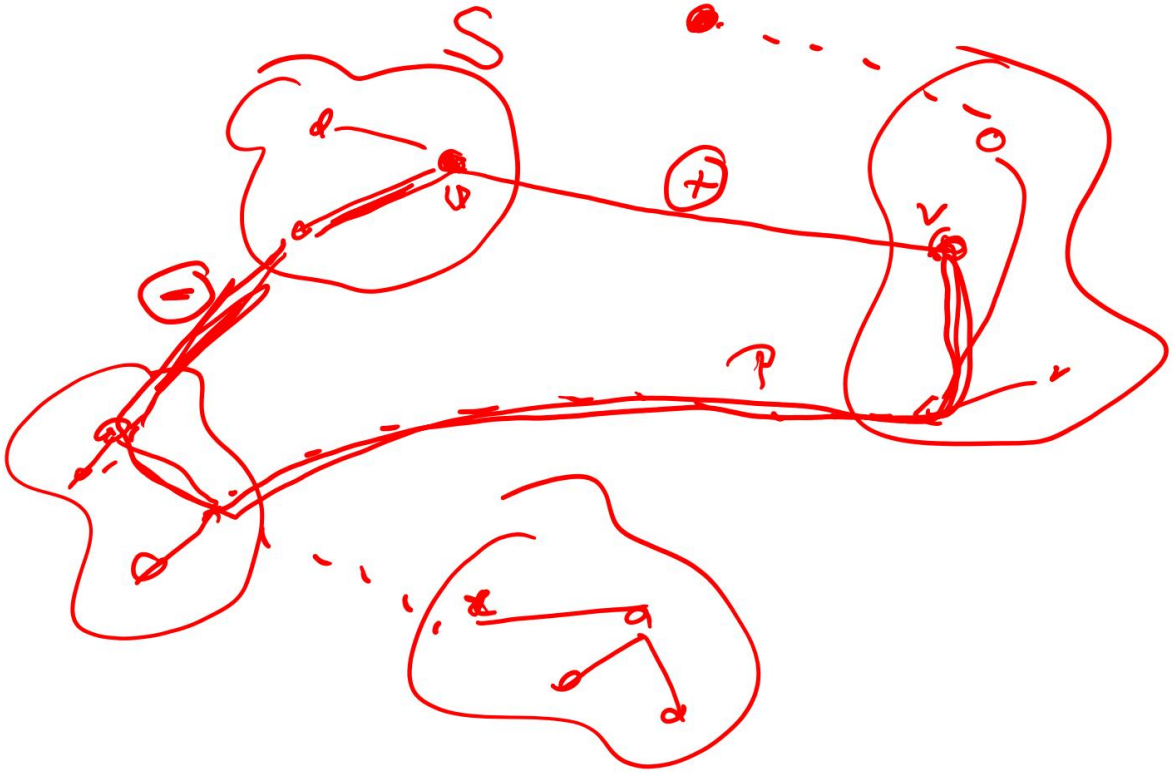
ÿ no edge at the beginning

Lemma 4: *Let be a set of edges such that* (,) is *a subgraph of an MSB of* = (, ,) . *The edge with the lowest weight that connects* two different components of (,) is a safe edge for .

Greedy approach:

In each step, the edge with the lowest weight becomes the Forest added.

Proof of Lemma 4



Proof of Lemma 4

Let = (, ,) be a graph and X = (,) a subgraph of an MSB of ie ÿ ÿ . Let ÿ be an edge with minimal weight, the two [.]

components in connects. Let be a component of such that ÿ and ÿ . We want to show that [,] is a safe edge for.

We assume that [,] is not safe for and generate a

contradiction. This proves that [,] is safe for.

If [,] is not safe for, then [,] does not belong to any MSB, which is all

edges of contains. It follows that there is a path from to that does not contain [,].

Let [,] be the first edge on such that x ÿ and ÿ . [,] cannot belong to since ÿ . Since [,] the component with another

Component connects is according to the definition of [, ÿ (,).] In (.)

Let ÿ { , } also an exciting tree. But [w is] ÿ is acyclic and connected, i.e. is a

MSB. Since A ÿ { , } to assume that (ÿ) = ( ) (.) + (.) ÿ ( ), i.e. belong, ÿ

there is no MSB containing A ÿ { , }, i.e. that is [ ] this contradicts the

not safe for. [ ] [.]

Why is ÿ an exciting tree?

Removing [,] breaks into 2 components, namely and ÿ such that ÿ and ÿ . Therefore,

adding [,] to T connects ÿ

[,] the two components and V ÿ again.

thats why ÿ = ÿ , ÿ {{[} connected.]} [ ]

If we use [, }, Adding] to T without removing [,] produces exactly that

edge [,] lies on the path and [,]. Since a circle in ÿ { , by the definition of [,] the

to , removing [,] destroys exactly the circle .

thats why ÿ = ÿ , {[ ]} ÿ { , [} acyclic.

It follows that is. ÿ acyclic and connected, and therefore an exciting tree

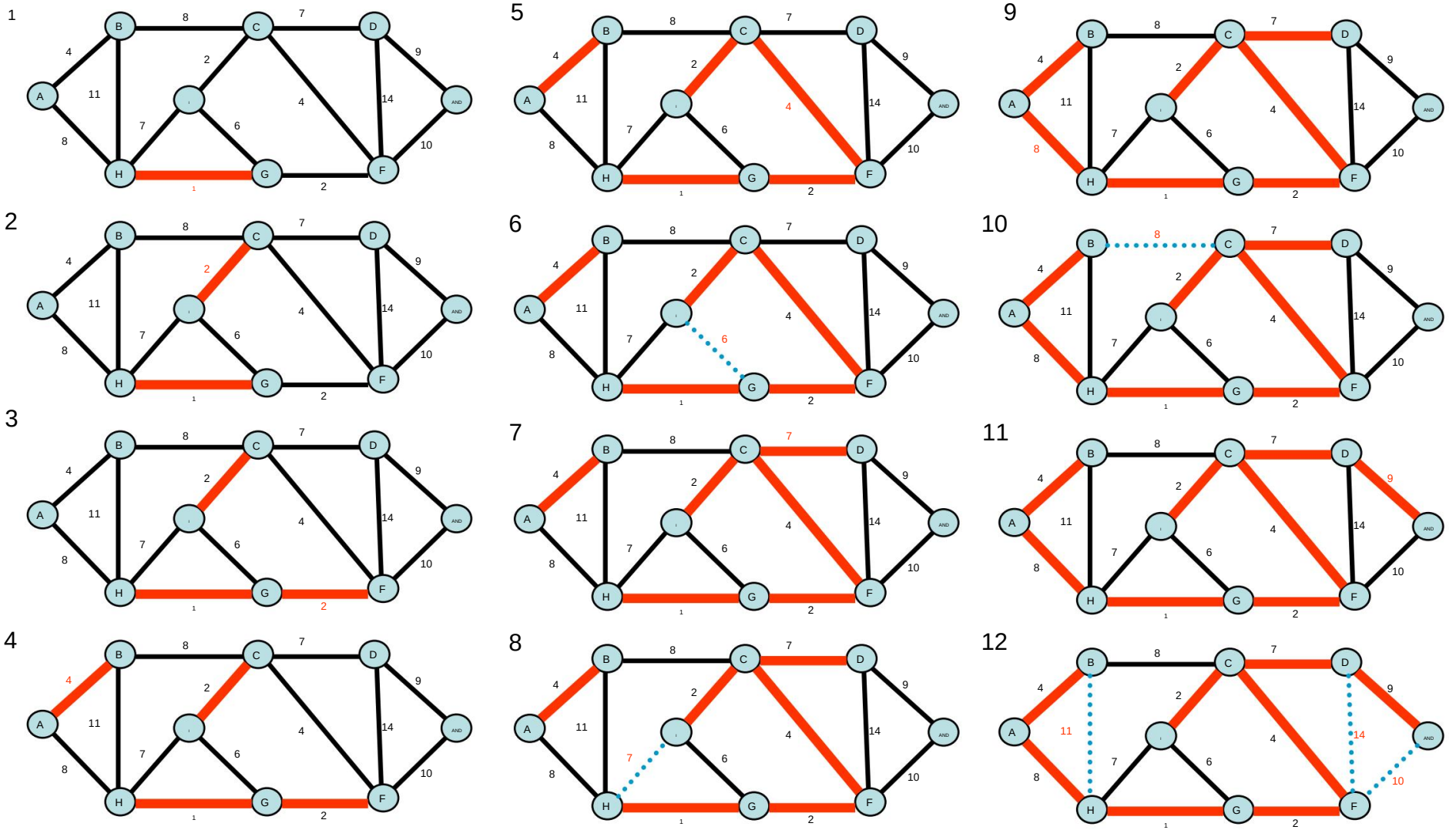
Kruskal algorithm (2)

```
MSB-Kruskal(G=(V, E, w)) {
    A = {}; for
     (each node vÿ V)
        make-set(v);
    sort the edges from E according to increasing weight w
    for(each edge [u,v]ÿE in the order of weights)
        if(find-set(u) != find-set(v)) {
            A = A ÿ { [u,v] }
            union(u, v)
    }
    return A;
}
```

w contains the weights between the nodes, e.g. weight between u and v = **w(u,v) make-set(v)** creates a set consisting of the element v **find-set(v)** returns a representative element for the set which v contains **union(u,v)**

combines sets of u and v into one set Here: Each set corresponds to the nodes in a component of (V,A)

Example: Kruskal algorithm



Expense from MSB-Kruskal

- (log = ()log) Sort edges*, plus
- make-set The operation, plus
- 2 find-set The operation, plus
- ÿ ÿ 1 union operations

  (a tree has at most ÿ 1 edges)

A data structure containing make-set, find-set, and union Operations implemented is called ***union-find data structure***

The effort depends on the union-find data structure that is used.

* Note: There < ^{2} , it follows that log < log ^{2} = 2 log .

Union-Find data structure

Idea

Set is represented by the root of a tree

- make-set(v): Generate a new tree with a node v
- find-set(u): Return the root of the tree containing u out of
- union(u,v): Make the 2 trees u and v contain, a tree in which the root of one tree becomes a child of the root of the other tree

Simple union-find data structure

```
make-set(x) { p[x]
    = x;

} link(x, y) { p[y]
    = x;

} find-set(x) {
    if(x != p[x]) return find-set(p[x]); return p[x];

}
```

```
union(x, y)
    { link(find-set(x), find-set(y));
}
```

p[x] contains the parent node of x (representative of the subset) **p[x] == x** if x is the root

Effort: Tree can have height ÿ 1 •

make-set(x): (1) •

link(x,y): (1) • find-set(x): ( ) •

union(x,y): ( ) ÿ

MSB-Kruskal: log +(ÿ = ( ) ÿ) with this one Data structure

Example: Union find

Elemente: A, B, C, D, E, F, G, H, I p-

Werte für

make-set(A), make-set(B), …, make-set(I)

union(H,G)

union(F,G)

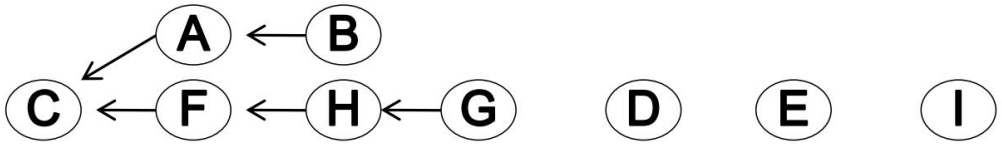
union(A,B)

union(C,F)

union(H,A)

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**					
A	B	C	D	E	F	G	H	I					
A	B	C	D	E	F	H	H	I					
A	B	C	D	E	F	H	F	I					
A	A	C	D	E	F	H	F	I					
A	A	C	D	E	C	H	F	I					
C	A	C	D	E	C	H	F	I					

At this point, the p-values have the following structure:



find-set(B) braucht 2 Look-ups, find-set(G) 3 Look-ups.

Better union-find data structure (union-by-rank)

Idea: Reduce the effort of find-set by link balancing the tree

```
make-set(x) {
    p[x] = x;
    rank[x] = 0;
}
link(x, y) {
    if (rank[y] > rank[x]) p[x] = y;
    else {
        p[y] = x;
        if (rank[x] == rank[y])
            rank[x] = rank[x] + 1;
    }
}
find-set(x) {
    if(x != p[x]) return find-set(p[x]);
    return p[x];
}
```

```
union(x, y) {
    link(find-set(x), find-set(y));
}
```

p[x] contains the parent node of x (representative of the subset)
rank[x] contains the height of x (Length of the longest path between x and a leaf in Under tree)

Example: Union find

Elemente: A, B, C, D, E, F, G, H, I

p- und rank-Werte für

make-set(A), make-set(B), …, make-set(I) union(H,G)

union(F,G)

union(A,B)

union(C,F)

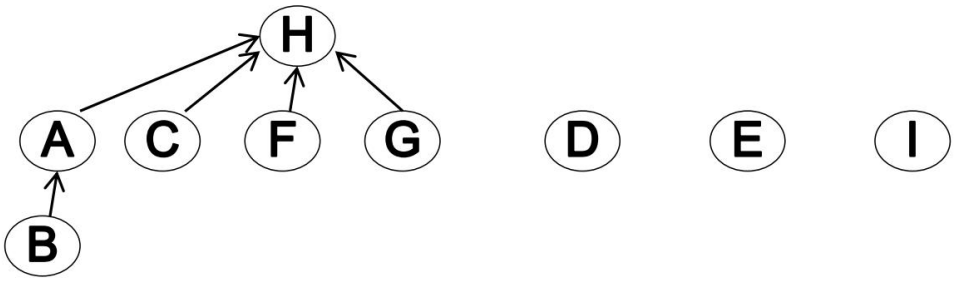
union(H,A)

A	B	C	D	E	F	G	H	I					
A,0	B,0	C,0	D,0	E,0	F,0	G,0	H,0	I,0					
A,0	B,0	C,0	D,0	E,0	F,0	H,0	H,1	I,0					
A,0	B,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0					
A,1	A,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0					
A,1	A,0	H,0	D,0	E,0	H,0	H,0	H,1	I,0					
H,1	A,0	H,0	D,0	E,0	H,0	H,0	H,2	I,0					

At this point, the p-values have the following structure:

find-set(B) needs
2 look-ups,

find-set(G) just more
1 Look-up



Property of union-by-rank

Lemma 5: A node with rank = has at least 2 [ ] node in its subtree.

Proof: Induction on k.

= 0: Each node lies in its own subtree, ie each subtree 0 has at least 1 = 2 Node, also the subtree of a node with rank 0.

ÿ 1 ÿ : The rank of a node only increases to if there is a node with the same rank (before the increase) and becomes a child of. By the induction assumption we know that at this point in time in the subtree of as well as in the subtree of at least 2 ÿ1 nodes are. Therefore, the unified tree has at least 2 ÿ1+2 ÿ1= 2 nodes.

ÿ If there are elements, then there are ÿ nodes in every ÿ . It Baum, d. 2 k follows that k ÿ log, i.e. every node ÿ log, i.e. has [ ] every tree height at most log.

Union-by-rank effort

Effort: Every tree has height ÿ log

- make-set(x): (1)
- link(x,y): (1)
- find-set(x): (log)
- union(x,y): (log)

$m < n^2$

$\log m < \log n^2 = 2 \log n$

ÿ MSB-Kruskal: (log + log = (log) with this)

Data structure

Best Union-find data structure (union-by-rank with path compression)

```
make-set(x) {
    p[x] = x;
    rank[x] = 0;
}
link(x, y) {
    if (rank[y] > rank[x]) p[x] = y;
    else {
        p[y] = x;
        if (rank[x] == rank[y])
            rank[x] = rank[x] + 1;
    }
}
find-set(x) {
    if(x != p[x]) p[x] = find-set(p[x]); return p[x];

}
```

```
union(x, y) {
    link(find-set(x), find-set(y));
}
```

p[x] contains the parent node of x (representative of the subset)

rank[x] is an *upper bound*

the height of x (number of Edges between x and one

Successor leaf **find-set(v)** looks for the root and then gives it to everyone node as parent node

Path compression: All nodes on the path become the root Children of the Root

Example: Union find

Elements: A, B, C, D, E, F, G, H,

I p and r values for

make-set(A), make-set(B), …, make-set(I)

union(H,G)

union(F,G)

union(A,B)

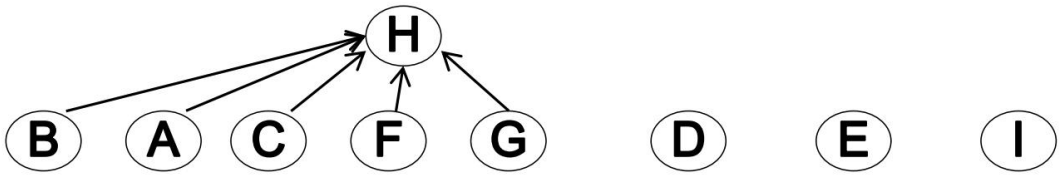
union(C,F)

union(H,A)

find-set(B)

A	B	C	D	E	F	G	H	I					
A,0	B,0	C,0	D,0	E,0	F,0	G,0	H,0	I,0					
A,0	B,0	C,0	D,0	E,0	F,0	H,0	H,1	I,0					
A,0	B,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0					
A,1	A,0	C,0	D,0	E,0	H,0	H,0	H,1	I,0					
A,1	A,0	H,0	D,0	E,0	H,0	H,0	H,1	I,0					
H,1	A,0	H,0	D,0	E,0	H,0	H,0	H,2	I,0					
H,1	H,0	H,0	D,0	E,0	H,0	H,0	H,2	I,0					

At this point, the p-values have the following structure:



Effort of union-by-rank with path compression

Complicated analysis

- make-set(x): (1)
- link(x,y): (1)
- find-set(x): (( )) (fast (1))
- union(x,y): (( )) (fast (1))

(,): Ackermann function, extremely fast growing function

( ) ÿ (,) ÿ1

( ) ÿ ( ), Extremely slow growing function

( ) < 5 practical for everyone relevant

ÿ MSB-Kruskal: (log + ÿ ( )) = (log) with this Data structure

6.6.2 Prim Algorithm

Lemma 4: *Let be a set of edges such that* (,) is *a subgraph of an MSB of* = (, ,) . *The edge with the lowest weight that connects* two *different components of* (,) is a safe edge for .

Lemma 4a: *Let be a set of edges such that* (,) *is a subgraph of an MSB of* = (, ,) *and let be a component of A. The edge with the lowest weight that has another component of* (,)

connects is a safe edge for .

Proof: In the proof of Lemma 4, replace

"Let ÿ [be an] edge with minimum weight containing two components such that ÿ and ÿ ."

connects. Let be a component of by:

"Be [ ] ÿ an edge with minimal weight, that with another

a component of connects."

6.6.2 Prim Algorithm

Lemma 4a: The edge with the lowest weight that *connects to another component of* (,) is a safe edge for .

Basic idea:

1. The set forms a single tree 2. The safe

edge that is added is always the edge [,]
with the lowest weight that connects T to a node u that is not yet in T

1. For each node that is not yet in, store the weight of the “lightest” edge that connects to in the variable [ ].
2. The node with minimum value is the [node] that is not yet in is and (of all such nodes) an edge with lowest has weight

ÿ Greedy approach

Advantage: Sorting the edges by weight is not necessary

Prim algorithm

```
MSB-Prim(G=(V,E,w), r) {
    Q = V;
    for(each node u ÿ Q)
        key[u] = ÿ;
    key[r] = 0;
    pred[r] = NIL;
    while(Q != {}) {
        u = extract-min(Q)
        if (pred(u) != NIL) {
                A = A ÿ { [u, pred[u]] }
        }
        for(every node v adjacent to u) if(v ÿ Q &&
            w(u,v) < key[v]) {
                pred[v] = u;
                decrease-key(v, w(u,v));
            }
    }}
```

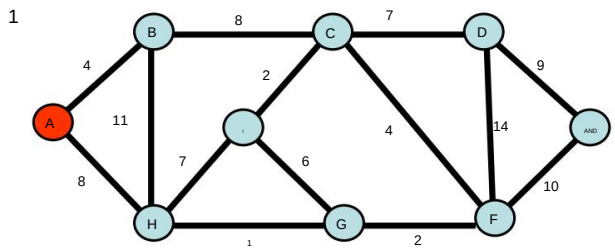
r is the root (start node) of the MSB T

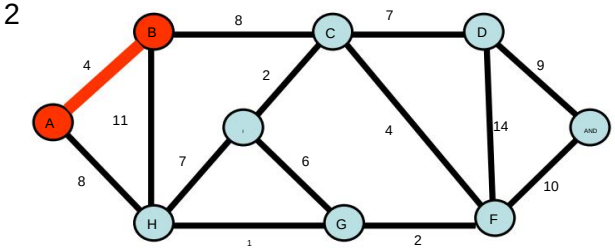
key[v] stores the lightest weight from v to a node in T

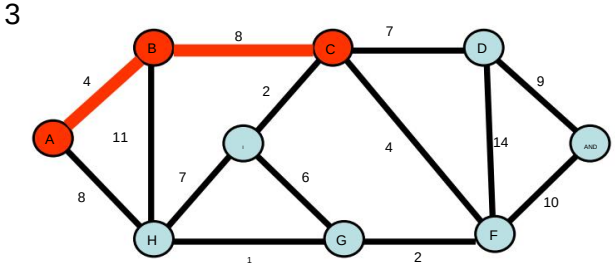
pred[v] stores a node **u** such that **[u,v]** has weight **key[v].**

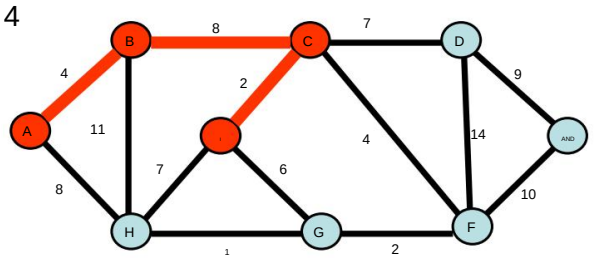
Q is a priority queue that stores all nodes not in T according to their **key** value ;

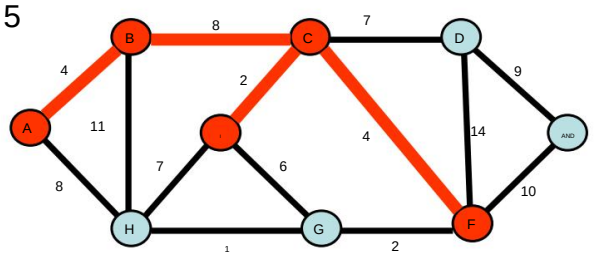
Example: Prim algorithm

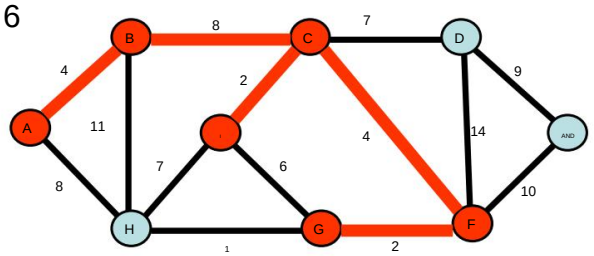


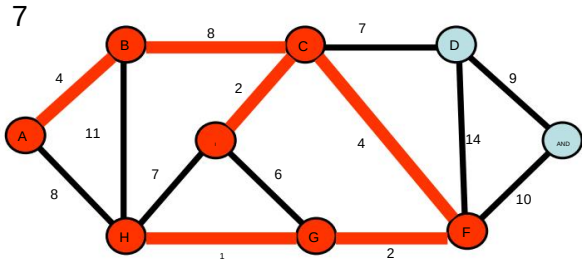


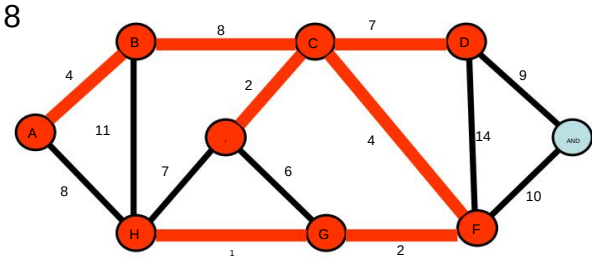


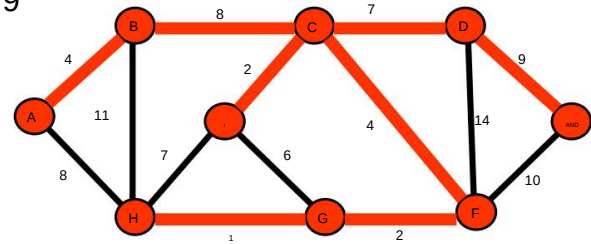












Effort of Prim Algorithm

- (+), plus
- Insert operations in Q, plus
- Extract-Min Operation, plus
- ÿ The Decrease-Key Operation

ÿThe effort of the algorithms depends heavily on the implementation of the set operations or the data structures for set management (priority queue, ...)

ÿHeaps from 5.9.1: Insert operations can be modified to Decrease-Key operations

Aufwand pro Insert, Extract-Min, Decrease-Key Operation: (log)

Total effort: + + log(+ log = (log))

Can be improved to (log +) using Fibonacci heap

6.7 Shortest paths

In a weighted graph, weights between nodes can be viewed as ***path lengths*** or costs

The question often arises as to which is the shortest Path(s) between

- a node and all other nodes
- two nodes
- all nodes

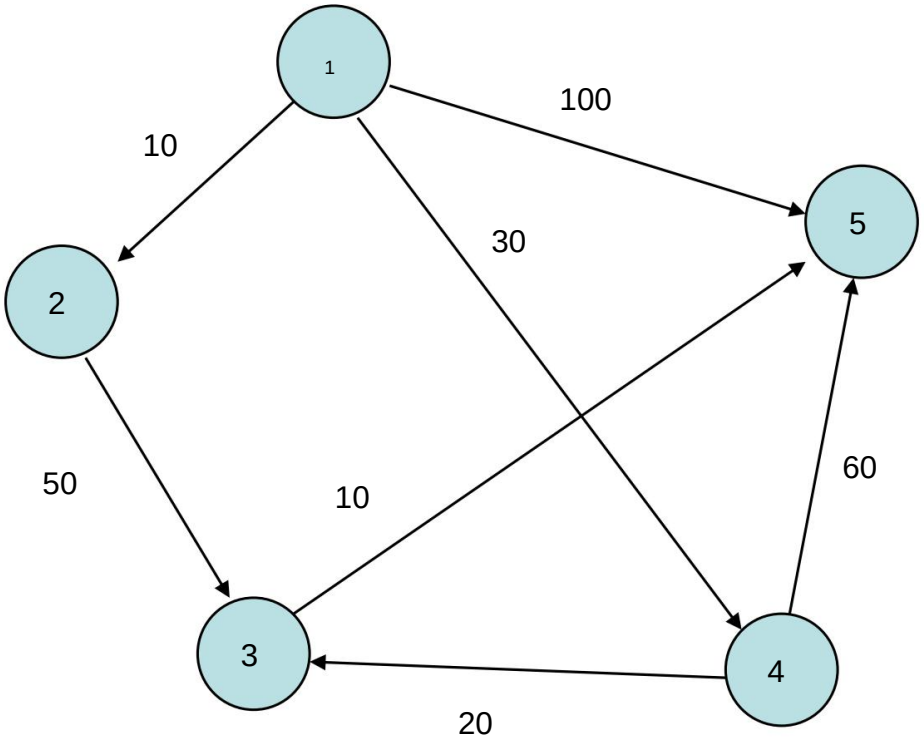
Assumption: all costs are stored in the cost matrix C (C[u,v] contains costs from the edge (u,v) if it exists and ÿ if (u,v) does not exist.)

The question exists in directed and undirected graphs, here: directed graphs

Example: route network of an airline

Assumption: only positive edge values

Some algorithms only work with positive values



6.7.1 Dijkstra Algorithm Single Source Shortest Paths

Algorithm for finding the shortest path from one starting node s to all others node

Assumption: *only positive edge values*

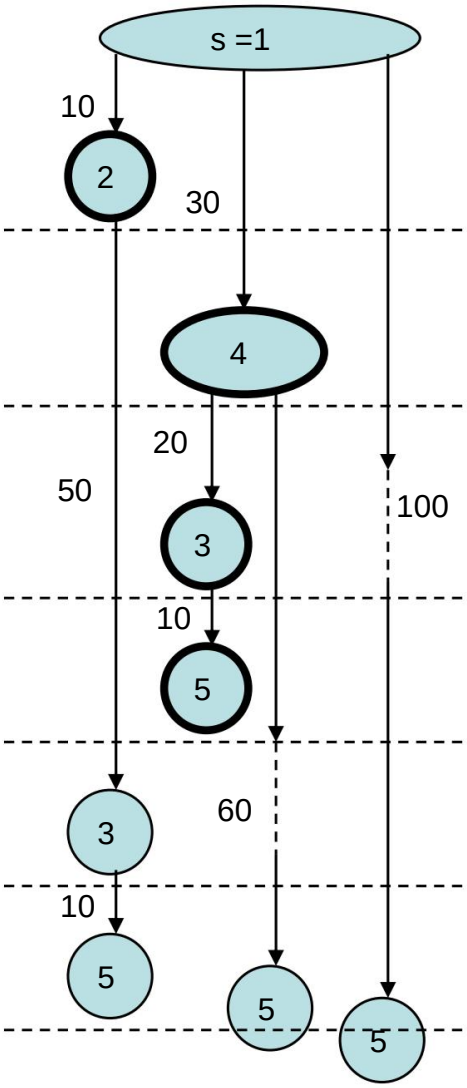
Idea

Start at the start node s.

Find the node with the shortest distance to s, then the node with the second shortest distance, etc. Note the respective shortest paths. ÿ Greedy approach, similar to Prim's algorithm

If all edge lengths = 1, then order of node visits in Dijkstra's algorithm is equal to the order of BFS ÿ Algorithm

constructs a shortest-path tree with root s. (What is the difference to MST?)



Dijkstra's algorithm

Basic idea:

1. is the set of nodes that have already been processed

2. The next node added is always a node
   to from ÿ which has the shortest path with inner nodes only from

   1. For each node that is not yet in, store the length of the shortest path from to
      in the variable minC[v].
      Nodes of used as inner nodes
   2. The node with minimum minC[u] value is the node that still
      is not in and (of all such nodes) has the shortest path from with inner nodes only from
      - (inner nodes = not start or end nodes of the path)

Greedy approach

Prim algorithm

```
MSB-Prim(G=(V,E,w), r) {
    Q = V;
    for(each node u ÿ Q)
        key[u] = ÿ;
    key[r] = 0;
    pred[r] = NIL;
    while(Q != {}) {
        u = Extract-Min(Q)
        if pred(u) != NIL) {
                A = A ÿ { [u, pred[u]] }
        }
        for(every node v adjacent to u) if(v ÿ Q && w(u,v)
            < key[v]) {
                pred[v] = u;
                Decrease-Key(v, w(u,v));
            }
}
```

r is the root (start node) of the MSB T

key[v] stores the lightest weight from v to a node in T

pred[v] stores a node **u** such that **[u,v]** has weight **key[v].**

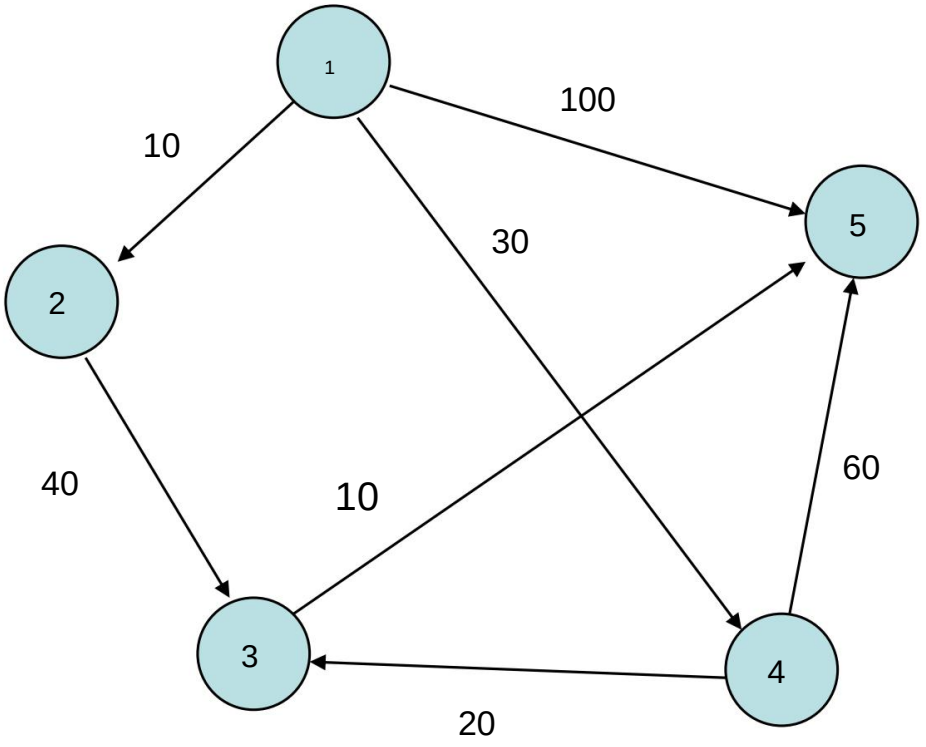
Q is a priority queue that stores all nodes not in T according to their **key** value;

Dijkstra's algorithm

```
Dijkstra(G=(V,E,w), s) {
1.Q = V; S = {};
2. for(each node v ÿ Q)
3.          MinC[v] = ÿ;
4. MinC[s] = 0;
5. while(Q != {}){
6. v = Extract-Min(Q);
7. if MinC[v] < ÿ {
8.          S = S ÿ { v };
9.          for (every node w adjacent to v)
10.              if ((w ÿ Q) && (MinC[v] + C[v,w] < MinC[w])) {
11.                  Decrease-Key(w, MinC[v] + C[v,w]);
12.              }
13. }
14.}
```

S Amount of nodes already processed
v Node currently being edited
C cost matrix
MinC shortest path known so far
Q is a priority queue (=heap), which is all Nodes not in S according to their **MinC** Saves value

Calculation for s = 1



S Act MinC

{ } [0, ÿ, ÿ, ÿ, ÿ ÿ

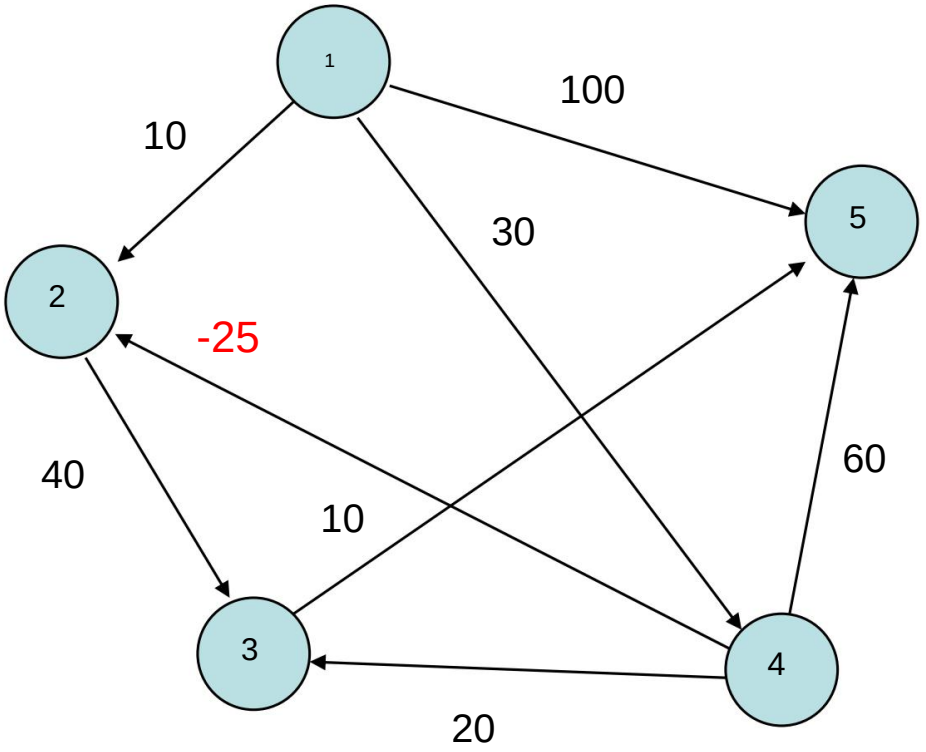
{1} [0, 10, ÿ, 30, 100]

{1,2} [0, 10, 50, 30, 100]

{1,2,4} 4 [0, 10, 50, 30, 90]

{1,2,3,4} 3 [0, 10, 50, 30, 60]

What happens if edge costs are negative?



s Act MinC

{ } [0, ÿ, ÿ, ÿ, ÿ ÿ

{1} 0 [0, 10, ÿ, 30, 100]

{1,2} 2 [0, 10, 50, 30, 100]

{1,2,4} 4 [0, 10, 50, 30, 90]

2 is no longer in Q. Therefore, the shortest paths that go through 2 can no longer be corrected.

{1,2,3,4} 3 [0, 10, 50, 30, 60] incorrect values

Expense

- (+ (as) with BFS), plus
- Insert and extract min operations

• ÿ The Decrease-Key Operation

ÿHeaps from 5.9.1: Insert operations can be modified to Decrease-Key operations

Aufwand pro Insert, Extract-Min, Decrease-Key Operation: (log)

Total effort: log + log = (log))

Can be determined by Fibonacci

Improve heap to (+ log)

6.7.2 All Pairs Shortest Paths (APSP)

Before we address the question of the shortest paths between all of them
To clarify nodes, we first want to deal with the (simpler)
question of which nodes actually exist
ways

Leads to 2 algorithms

Transitive cover: which nodes are connected by paths

APSP: all shortest paths between nodes

Transitive cover

Question which nodes are connected by paths (reachable), leads to the question of the ***transitive closure***

A directed graph) becomes = (,) ***transitive*** (and ***reflexive)***

Cover of a graph = (,) if and only if:

(,) ÿ ÿ ÿ there is a path from to in

Starting point: Adjacency matrix of G

universität wien

Idea 1

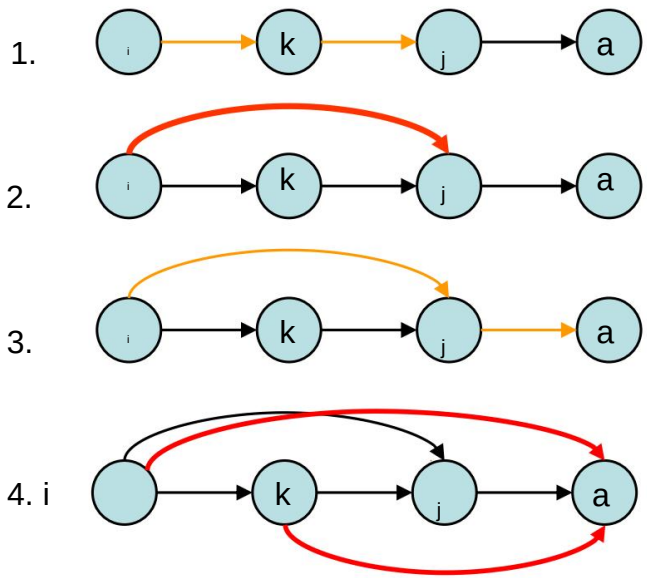
Iterative addition of edges in the graph for paths of length 2

i.e. path (i,k) and (k,j) is described by new edge (i,j).

subsequently it will be natural paths (i,j) and (j,a) were also found to be paths of length 2 (in reality clearly paths of length 3), etc.

This causes the new edges to describe longer and longer paths until all possible paths have been found

Approach through 3 nested loops, similar to matrix multiplication



Approach: **Dynamic programming:**
Solve “smaller sub-problems” to solve the overall problem

Floyd-Warshall algorithm (2)

In which order should the nodes/paths be checked?

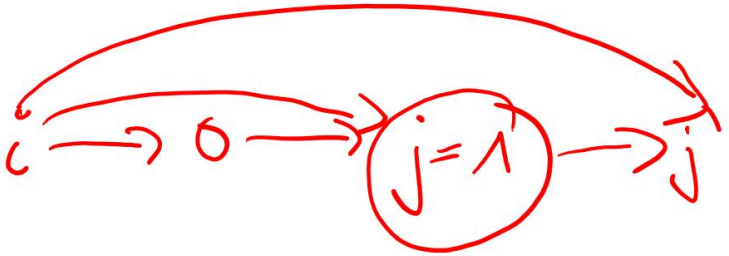
Idea 2: For all node pairs (i,j): •

First loop pass: Test whether there is a path through node 0.

- If there are edges (i,0) and (0, j), add edge (i,j).

- Second loop pass: Test whether there is a path through node 1.
  - If there are edges (i,1) and (1, j), add edge (i,j).
  - After all paths through 0 to 1 have already been found, this one finds
    Also step all paths (i, 0), (0,1) and (1,j). The same applies to all paths from 1 through 0, i.e. all paths (i,1) and (1, 0) (0,j).
  - Therefore, this step finds all paths that are considered inner nodes only nodes Use {0,1}
- General invariant: The execution of the outermost loop finds all paths from i to j for the respective value k, which only use nodes from {0, …, k} as inner nodes
  (inner nodes = not start or end nodes of the path)

Floyd-Warshall algorithm (2)




```
Transitive cover (matrix a) {
    n = a.numberofRows();
    for(int k = 0; k < n; ++k)
        /* Test all pairs (i,j) */
        for(int i = 0; i < n; ++i)
            for(int j = 0; j < n; ++j) a[i,j] = a[i,j] |
                a[i,k] & a[k,j];
}
```

a Adjazenzmatrix
at **a** as a bit matrix

| binary OR operator

& binary AND operator

Adds path as new
Insert an edge (if it
does not already
exist) into the graph

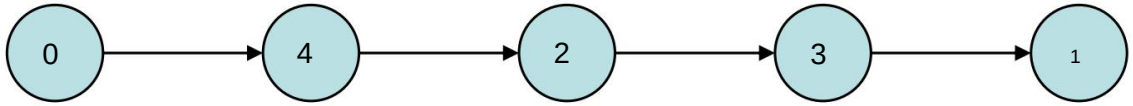
True (1) if there is a
path between i
and j via k

Expense

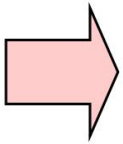
With adjacency matrix representation:

- ( ) for the innermost loop,
- ( ) ^{2} for the two inner loops,
- (^{3}) for all three loops

Example: Floyd-Warshall for transitives Cover (1)



	0	1	2	3	4		
0	0	0	0	0	1		
1	0	0	0	0	0		
2	0	0	0	1	0		
3	0	1	0	0	0		
4	0	0	1	0	0		



	0	1	2	3	4		
0	0	1		1	1	1	
1	0	0	0	0	0		
2	0	1	0	1	0		
3	0	1	0	0	0		
4	0	1		1	1	0	

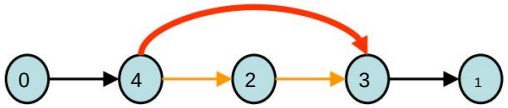
Example: Floyd-Warshall for transitives Cover (2)

Values for k (runs outer loop):

0: no edge, since a(?,0) is not possible

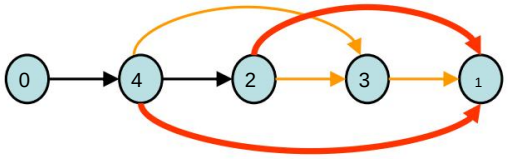
1: no edge, since a(1,?) is not possible

2: Kante a(4,2), a(2,3) ÿ a(4,3)



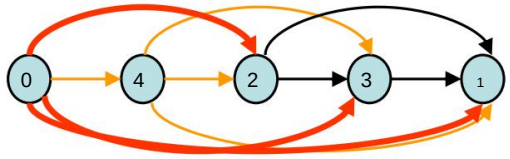
	0	1	2	3	4		
0	0	0	0	0	1		
1	0	0	0	0	0		
2	0	0	0	1	0		
3	0	1	0	0	0		
4	0	0	1			**1**	0

3: a(2,3), a(3,1) ÿ a(2,1) a(4,3), a(3,1) ÿ a(4,1)



	0	1	2	3	4		
0	0	0	0	0	1		
1	0	0	0	0	0		
2	0	**1**	0	1	0		
3	0	1	0	0	0		
4	0	**1**		1		1	0

4: a(0,4), a(4,1) ÿ a(0,1) a(0,4), a(4,2) ÿ a(0,2) a(0,4), a(4,3) ÿ a(0,3)



	0	1	2	3	4		
0	0	**1**		1	1	1	
1	0	0	0	0	0		
2	0	1	0	1	0		
3	0	1	0	0	0		
4	0	1		1		1	0

Floyd-Warshall algorithm for shortest paths

Calculating the shortest paths between all nodes is
can be easily derived from the algorithm for transitive hulls

Difference

- Adjacency matrix stores the path lengths between nodes (corresponds to the cost matrix)
  - Instead of 0/1 values the travel costs
- When determining a path over 2 edges, simple calculation of the Path length of this new edge and comparison with current path length (if already existing)
  - Instead of logical AND, the calculation of the cost sum

Floyd-Warshall algorithm (2)

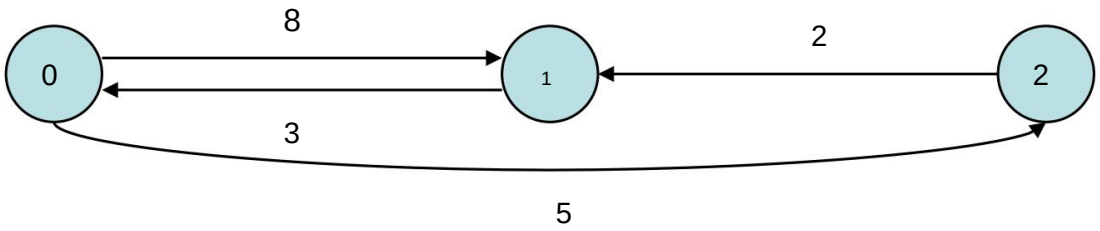
```
Floyd-Warshall (Matrix a) {
    n = a.numberofRows();
    for(int k = 0; k < n; k++)
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n; j++) {
                int newPathLength = a[i,k] + a[k,j];
                if(newPathLength < a[i,j])
                    a[i,j] = newPathLength;
            }
}
```

Entry in the cost matrix if the new route is shorter than the possibly existing old route

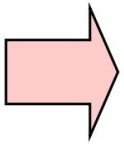
Calculation of Path length via new path

Invariant: The kth execution (counting starting from 0) of the outermost Loop finds the length of the shortest path from i to j, which only uses nodes from {0, …, k} as inner nodes

Example: Floyd-Warshall for shortest ways



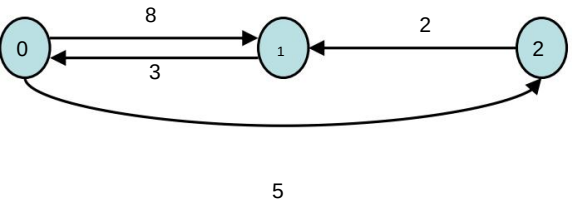
	0	1	2	
0	0	8	5	
1	3	0	ÿ	
2	ÿ	2	0	



	0	1	2	
0	0	7	5	
1	3	0	8	
2	5	2	0	

Example: Floyd-Warshall for shortest ways (2)

Starting position

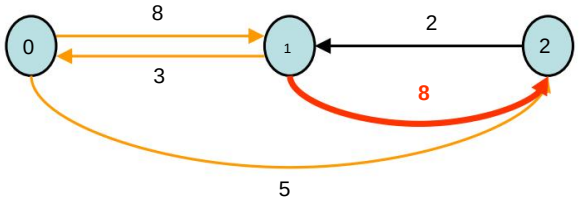


	0	1	2
0	0	8	5
1	3	0 ÿ	
2 ÿ 2			0

k: 0

(1.0) (0.2) ÿ (1.2), Cost: 8

(1.0) (0.1) ÿ (1.1), Cost: 11



	0	1	2
0	0	8	5
1	3	0	8
2 ÿ 2			0

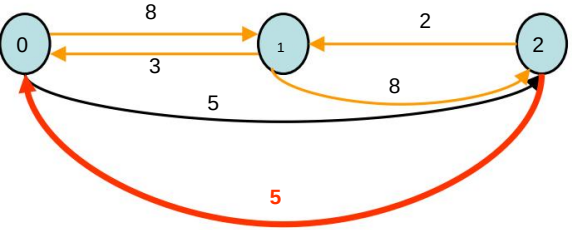
k: 1

(0,1) (1,2) ÿ (0,2): 16

(0,1) (1,0) ÿ (0,0): 11

(2,1) (1,0) ÿ (2,0): 5

(2,1) (1,2) ÿ (2,2): 10



	0	1	2
0	0	8	5
1	3	0	8
2	5 2		0

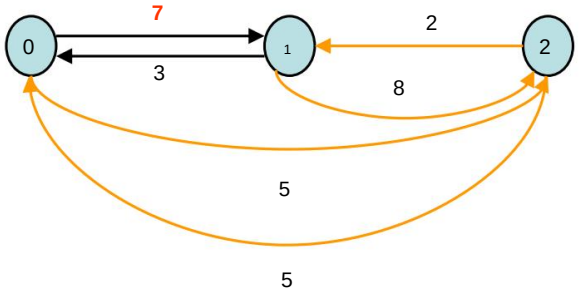
k: 2

(0,2) (2,0) ÿ (0,0): 10

(0,2) (2,1) ÿ (0,1): 7

(1,2) (2,0) ÿ (1,0): 13

(1,2) (2,1) ÿ (1,1): 10

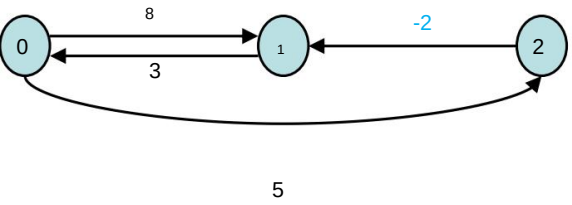


	0	1	2
0	0	7	5
1	3	0	8
2	5	2	0

Floyd-Warshall for shortest routes
What happens if costs are negative?

universität wien

Starting position

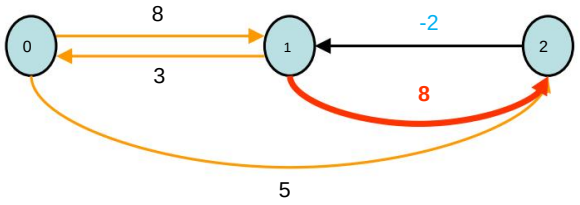


	0	1	2
0	0	8	5
1	3	0 ÿ	
2 ÿ -2			0

k: 0

(1.0) (0.2) ÿ (1.2), Cost: 8

(1.0) (0.1) ÿ (1.1), Cost: 11



	0	1	2
0	0	8	5
1	3	0	8
2 ÿ -2			0

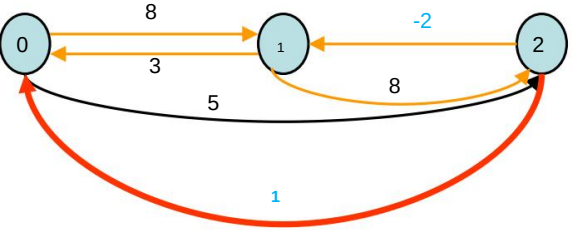
k: 1

(0,1) (1,2) ÿ (0,2): 16

(0,1) (1,0) ÿ (0,0): 11

(2,1) (1,0) ÿ (2,0): 1

(2,1) (1,2) ÿ (2,2): 6



	0	1	2
0	0	8	5
1	3	0	8
2	1	-2	0

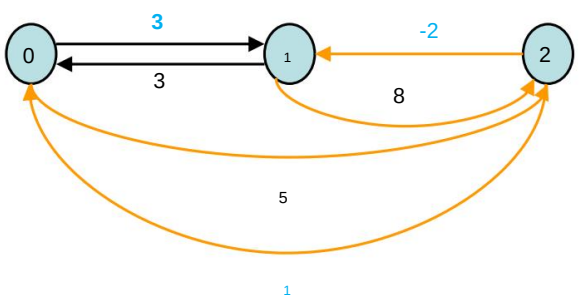
k: 2

(0,2) (2,0) ÿ (0,0): 6

(0,2) (2,1) ÿ (0,1): 3

(1,2) (2,0) ÿ (1,0): 9

(1,2) (2,1) ÿ (1,1): 6



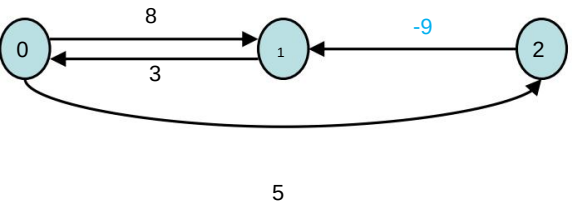
	0	1	2
0	0	3	5
1	3	0	8
2	1	-2	0

What happens if edge costs are negative?

Negative edge costs do not cause a problem as long as they do not produce cycles where the sum of the Edge cost is negative ***(negative cycle)***

Floyd-Warshall for shortest routes
What happens during negative cycles?

Starting position

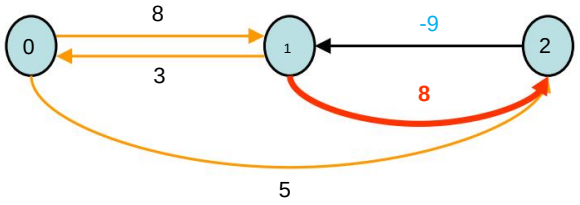


	0	1	2
0	0	8	5
1	3	0 ÿ	
2 ÿ -9			0

k: 0

(1.0) (0.2) ÿ (1.2), Cost: 8

(1.0) (0.1) ÿ (1.1), Cost: 11



	0	1	2
0	0	8	5
1	3	0	8
2 ÿ -9			0

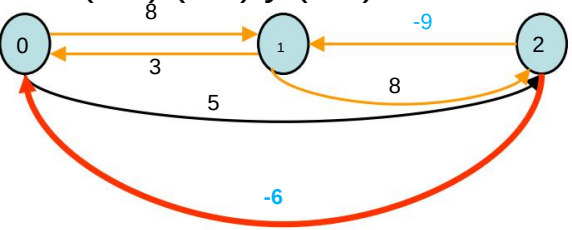
k: 1

(0,1) (1,2) ÿ (0,2): 16

(0,1) (1,0) ÿ (0,0): 11

(2,1) (1,0) ÿ (2,0): -6

(2,1) (1,2) ÿ (2,2): -1



	0	1	2
0	0	8	5
1	3	0	8
2 -6		-9	-1

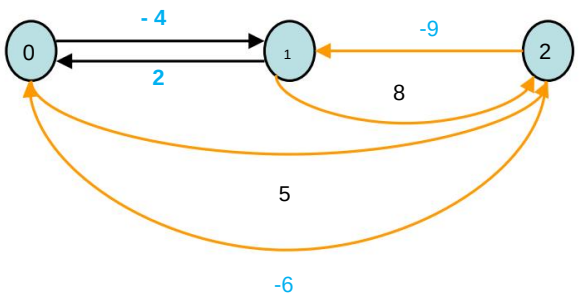
k: 2

(0,2) (2,0) ÿ (0,0): -1

(0,2) (2,1) ÿ (0,1): -4

(1,2) (2,0) ÿ (1,0): 2

(1,2) (2,1) ÿ (1,1): -1



	0	1	2
0	-1	-4 5	
1	2 -1		8
2	-6	-9	-1

What happens during negative cycles?

For negative cycles, there is one at the end of the algorithm

Node i with a[i,i] < 0

If there are no negative cycles, there is none in the end

Node i with a[i,i] < 0

ÿFloyd-Warshall can be used to **test** the existence of negative cycles

Expense: $(\ ^{3} \ )$

Attention: If there is a path between two nodes

contains negative cycle, their distance is ÿÿ / undefined.

What do we take with us?

- Graphene
  - The definition
  - Directed and undirected graphs
- Topological sorting
- traversal
  - BFS and DFS
  - “Farmer, Wolf, Goat and Cabbage” problem
  - General iterative approach
- Exciting trees
- Shortest routes