GRAPHS

Algorithms and Data Structures 2
Exercise – 2021W

Stefan Grünberger, Martin Schobesberger
Dari Trendafilov, Markus Weninger
Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

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

MOTIVATION

Graphs are one of the most basic data structures. Many problems can be characterized by graphs, such as:

- **Electric power grid**
  - Nodes: power distributors, transformer stations, etc.
  - Edges: wires
  - No defined direction → undirected graph
  - Cycles are possible
- **Material flow in manufacturing companies**
  - Nodes: work stations
  - Edges: band-conveyors
  - Raw materials only flow in one direction → directed graph
  - Limited capacity of band-conveyors → weighted graph
  - No cycles
- **Social distance in a set of persons**
  - Nodes: Humans
  - Edges: Relations

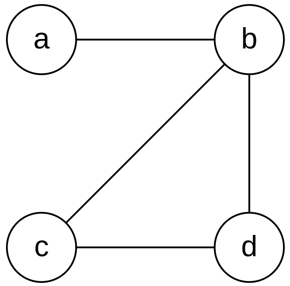
DEFINITION / TERMS

G = (V, E)

- **V... Set of vertices (or nodes)**
- **E... Set of edges**

Example:

- V = {a, b, c, d}
- E = {(a, b), (b, d), (c, d), (c, b)}



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

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

DEFINITION / TERMS

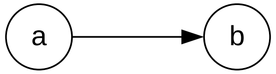
Degree of a vertex:
- Number of vertices that are adjacent to it (which is not necessarily equal to the number of edges)

Path: Sequence of adjacent vertices
- simple: No vertex occurs more than once.
- cyclic: At least one vertex occurs more than once.

Cyclic graph:
- Contains cyclic paths (otherwise: **acyclic** graph)

Directed edge: Connection from *a* to *b*.
- Directed graph: Contains only directed edges
- Directed, acyclic graph?

Loop:
- Edge (*v*, *v*) for vertex *v*



Component: connected part of a graph

DEFINITION / TERMS

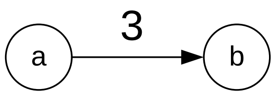
Connectivity

- Two vertices are called connected if there is a path (i.e., a sequence of edges) between them.
- Connected graph: Each pair of vertices in the graph is connected. This means that there is a path between every pair of vertices.
- Complete graph: Each pair of vertices is adjacent to each other (number of edges= *n(n-1)/2*)
- Strongly connected directed graph is a *complete* directed graph, i.e., compared to a complete undirected graph each edge is replaced by a pair of edges.
- Weakly connected directed graph is a directed graph whose underlying *undirected* graph is connected, i.e., if replacing all directed edges with undirected edges leads to a connected graph.

Tree: Connected, undirected graph without cycles

Forest: Set of trees

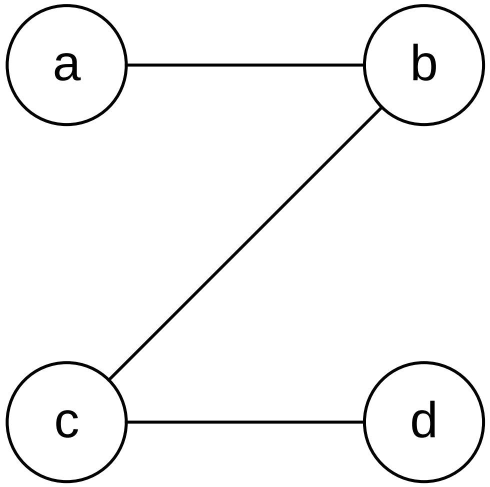
Weighted graph: Contains weighted edges.



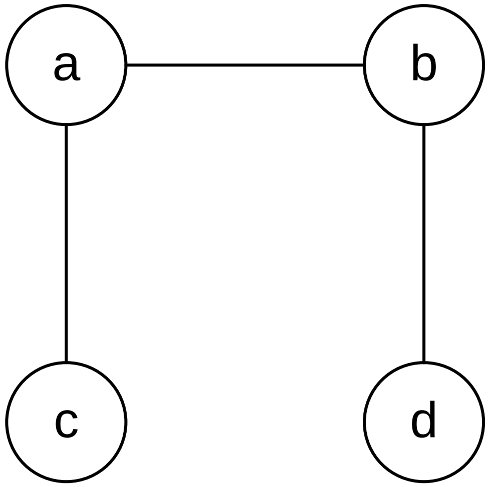
DEFINITION / TERMS

Spanning tree (ST): Subgraph of graph *G*, such that:

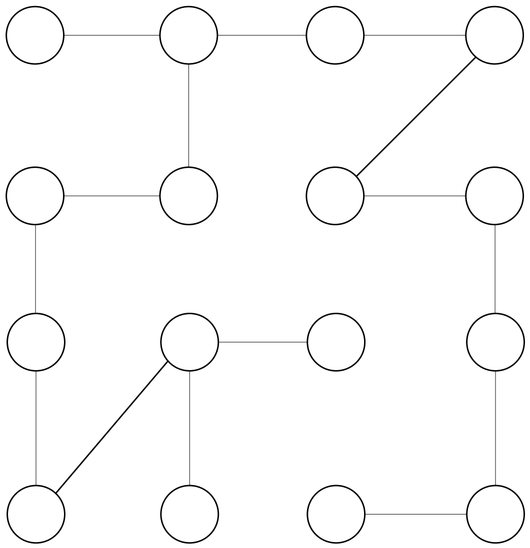
- ST is a tree
- ST contains all vertices of *G*
- By removing a single edge, the ST is no longer connected.



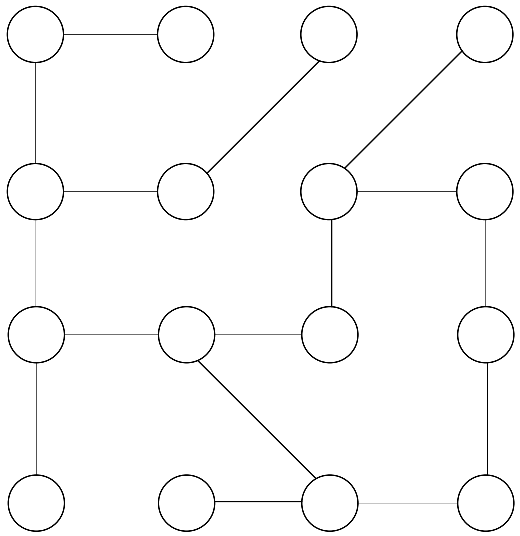
Example 1a



Example 1b



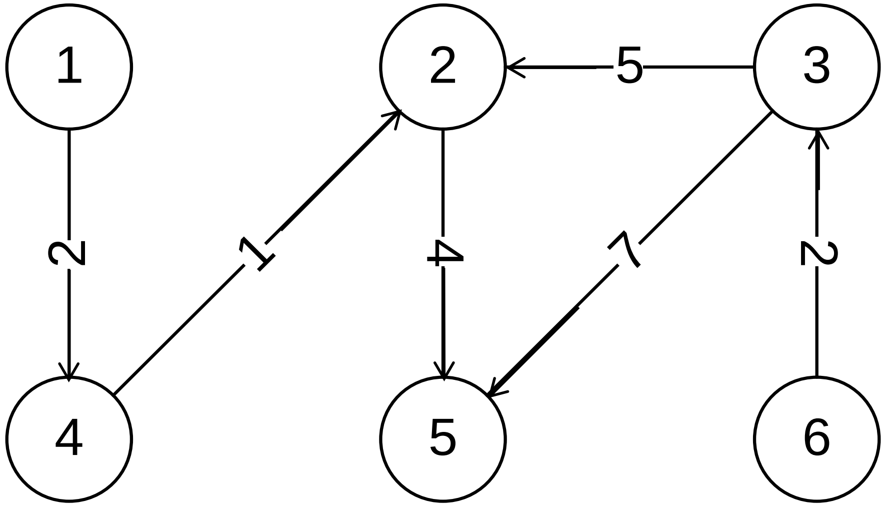
Example 2a



Example 2b

DATA STRUCTURES

Example:



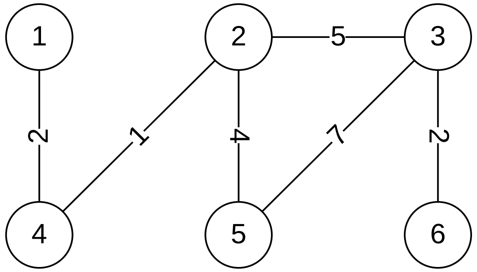
Out-degree
In-degree

Weighted:	yes
Directed:	yes
Degree(„2“):	(2,1)
Cyclic:	no
Loops:	0
Connected:	weak
Tree:	no

Graphs can be represented in form of a:

- **Edge list**
- **Adjacency matrix**

DATA STRUCTURES



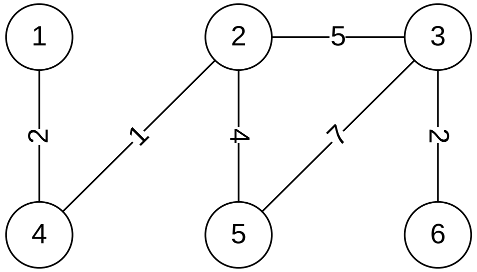
Edge list

- **Principle**: 2 data structures (for vertices and edges)
  - Array/List for vertices (add new vertices at the end)

```
class Vertex {
  toString() {...}
}
Vertex vertices[] // in class Graph
```

index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

DATA STRUCTURES



Edge list

- **Principle**: 2 data structures (for vertices and edges)
  - Array/List for vertices (add new vertices at the end)

```
class Vertex {
  toString() {...}
}
Vertex vertices[] // in class Graph
```

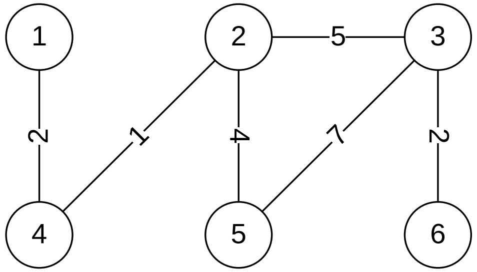
index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

  - Array/List for edges

```
class Edge {
  Vertex first, second  // the edge's vertices
  int weight   // edge weight
}
Edge edges[] // in class Graph
```

index	1	2	3	4	5	6
edge	1, 2; 2	2, 3; 1	3, 4; 4	3, 5; 5	4, 5; 7	5, 6; 2

DATA STRUCTURES



Adjacency matrix

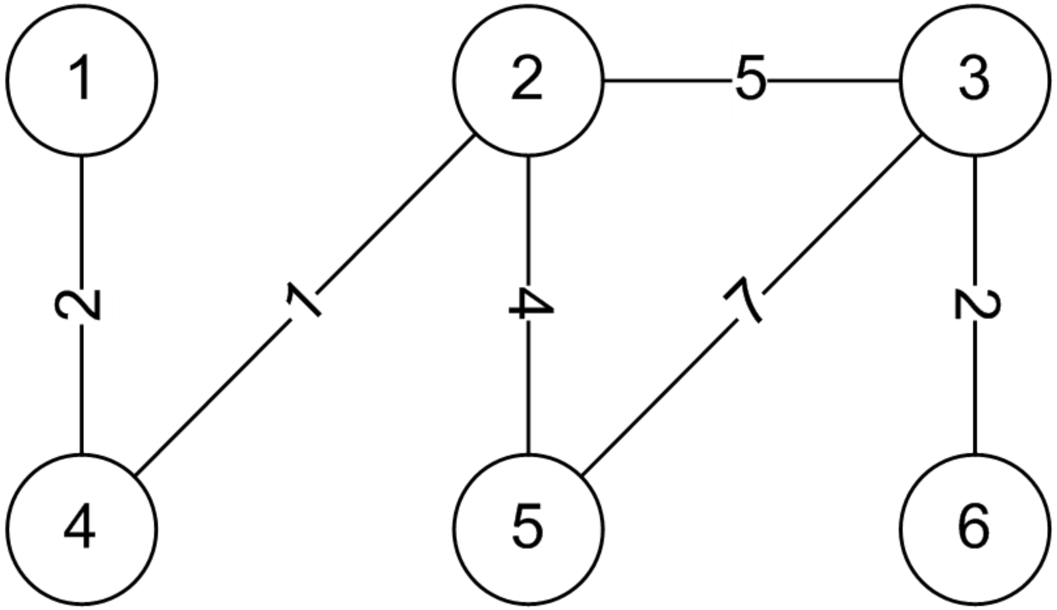
- **Principle**: Graph with *n* vertices is represented by an *n* x *n* matrix
  - Vertices are numbered from 1 to *n.*
  - Relation of the vertices are entered in the matrix.
  - True is entered in the i^{th} row and j^{th} column if vertices *i* and *j* are connected by an **unweighted edge**, otherwise false.
  - The adjacency matrix is symmetrical if the graph does not contain **any directed edges**.
  - For **weighted graphs** enter the edge weight (1...∞)
  - The **main diagonal** remains free if the graph contains no **loops**
  - If there is no edge, e.g., -1 can be entered.

	1	2	3	4	5	6
1				2		
2			5	1	4	
3		5			7	2
4	2	1				
5		4	7			
6			2			

TRAVERSAL

Two ways of traversing graphs (i.e. visiting all edges):

- **Breadth First Search (BFS)**
- **Depth First Search (DFS)**



DFS/BFS can be used to check:
- Is a graph *G* connected?
- Number of components in *G*?
- Is *G* cyclic?

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)

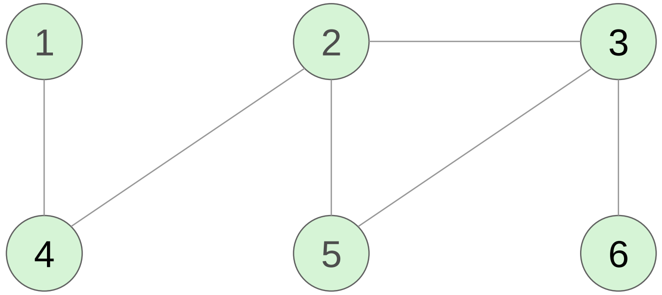
Principle

- Start with any vertex *v*:
  - Traverse *v*
  - Traverse (recursively) any unvisited vertex connected to *v*.

Implementation hint

- Usage of an auxiliary array to note which vertices have already been visited.

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

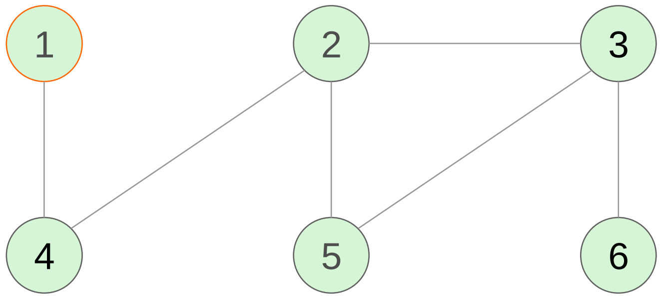
start vertex 1

mark ‚1‘ / check ‚4‘ (has not been visited yet)

1	2	3	4	5	6
T	f	f	f	f	f

Auxiliary array for visited vertices

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

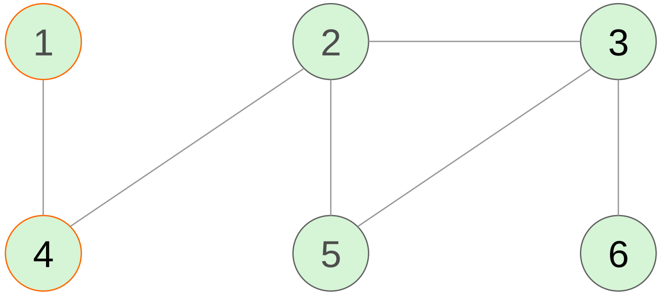
start vertex 1

mark ‚4‘ / check ‚1,2‘

1	2	3	4	5	6
T	T	f	f	f	f

Vertex ‚1‘ already visited, therefore visit ‚2‘ next

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



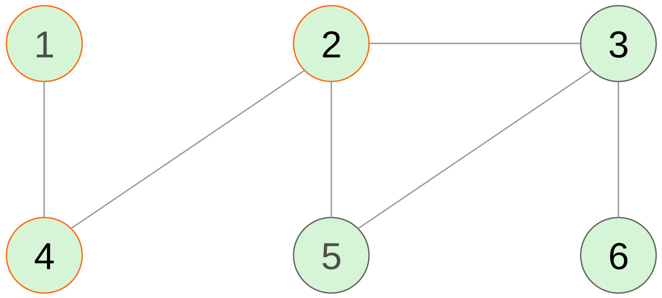
index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

start vertex 1

mark ‚2‘ / check ‚3,4,5‘

1	2	3	4	5	6
T	T	T	f	f	f

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



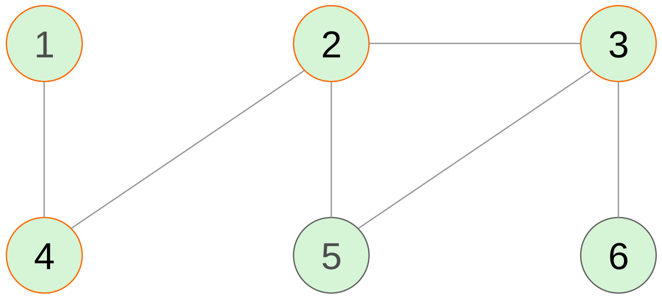
index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

start vertex 1

mark ‚3‘ / check ‚2,5,6‘

1	2	3	4	5	6
T	T	T	f	T	f

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



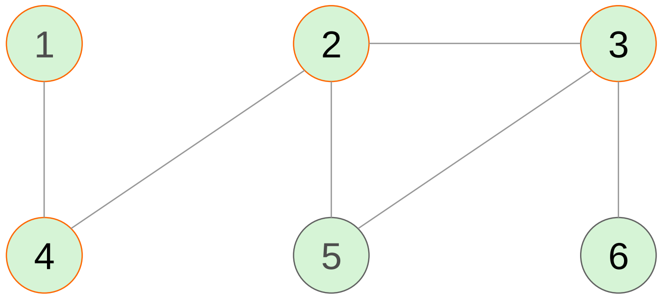
index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

start vertex 1

mark ‚5‘ / check ‚2,3‘ (cycle candidates: 1,2,4 | 3 is no candidate because it was the last one visited)

1	2	3	4	5	6
T	T	T	T	T	f

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

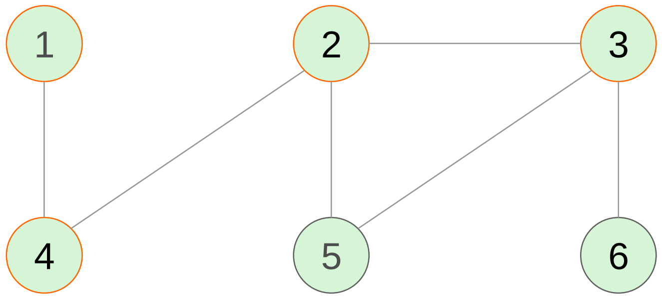
start vertex 1

overlap

mark ‚5' / check ‚2,3' (cycle candidates: 1,2,4)

1	2	3	4	5	6
T	T	T	T	T	f

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

start vertex 1

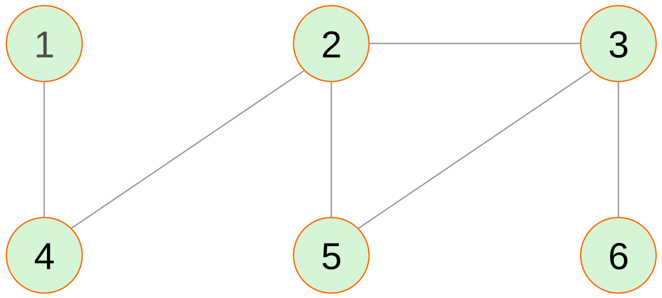
overlap

mark ‚5‘ / check ‚2,3‘ (cycle candidates: 1,2,4)

1	2	3	4	5	6
T	T	T	T	T	f

Overlap between the vertex to be checked (adjacent) and cycle candidate (visited but not previous) → cyclic graph!
Vertices ‚2,3‘ already marked, therefore go back in recursion and visit vertex ‚6‘.

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



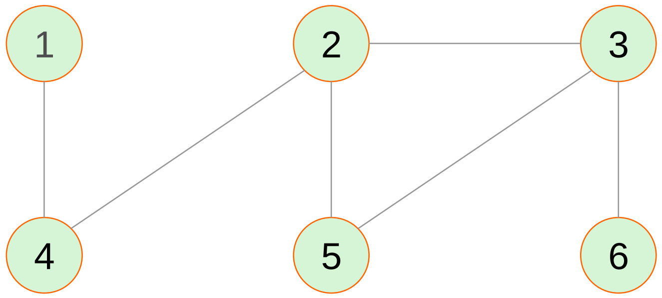
index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

mark ‚6‘ / check ‚3‘ (cycle candidate: 1,2,4,5)

1	2	3	4	5	6
T	T	T	T	T	T

Vertex ‚3‘ already marked, therefore go back in recursion to the start.

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)



index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

mark ‚6‘ / check ‚3‘ (cycle candidate: 1,2,4,5)

1	2	3	4	5	6
T	T	T	T	T	T

Auxiliary array filled completely
→ graph connected!

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)

Is graph *G* connected (method `boolean isConnected()`)**?**

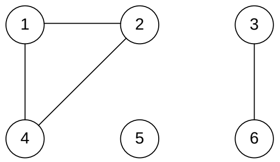
Does graph *G* contain cycles (method `boolean isCyclic()`)**?**

1. Start with vertex ***v* -** Mark ***v*** as traversed
   (Set value in auxiliary array at index of v to true. If value was already true, then there is a cycle → do not traverse this vertex again).
2. Determine the set of all vertices **AD(v)** that are adjacent to ***v***
   (Iterate over edge list and determine indices of adjacent vertices).
3. For each of these vertices ***n*** start with 1., where ***n*** instead of ***v*** is used.
   (Can be implemented recursively).
4. If the auxiliary array is completely filled at the end, the graph is connected.
5. If, during the traversal of a vertex, the value of the auxiliary array at the index of the vertex is already true, there is a cycle in the graph.
   (Cycles can only occur with a minimum of 3 vertices).

TRAVERSAL :: DEPTH FIRST SEARCH (DFS)

What is the number of components in the graph (method `int getNumOfComponents()`)?

Graph:



Vertex-Array:

index	1	2	3	4	5	6
vertex	1	4	2	5	3	6

- DFS is called once (starting with vertex 1) and return the following array (= 1. component)

index	1	2	3	4	5	6
visited	T	T	T	f	f	f

Then call DFS until all fields are marked (continuing with the next unmarked field, here index 4)

- DFS ends for the 2. time (= 2. component)

index	1	2	3	4	5	6
visited	T	T	T	T	f	f

- DFS ends for the 3. time (= 3. component)

index	1	2	3	4	5	6
visited	T	T	T	T	T	T

GRAPHS

Algorithms and Data Structures 2
Exercise – 2021W

Stefan Grünberger, Martin Schobesberger
Dari Trendafilov, Markus Weninger
Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

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