Universität Zürich^UZH

Institut für Informatik

Informatics II

Tutorial Session 13

Thursday, 25th of May 2022

Discussion of Exercise 12,
Graph Theory

14.00 – 15.45

BIN 0.B.06

Agenda

- Recitation of Graphs and Review of Exercise 12 (interleaved)
  - Graph Representation
  - Graph Traversals: DFS, BFS
  - (Topological Sorting, Edge Classification)
  - Minimum Spanning Tree
  - Dijkstra, Bellman-Ford
- Summary and Wrap-Up

Universität Zürich^UZH

Institut für Informatik

Administrivia

- Unofficial Q&A session

Unofficial Q&A Session

For people who want to ask questions, I will be available

tomorrow Friday, 28th of May

from 14.00 h

on Zoom

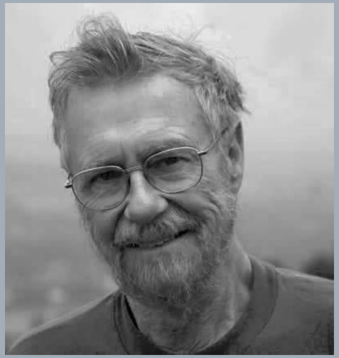
This will be an unofficial and informal Q&A session. I will not prepare any slides or materials but just (try to) answer any questions. The session will be recorded if people agree. If possible, send me any questions in advance.

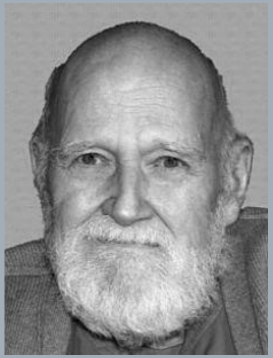
Universität Zürich UZH

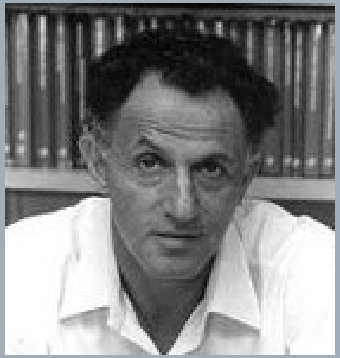
Institut für Informatik

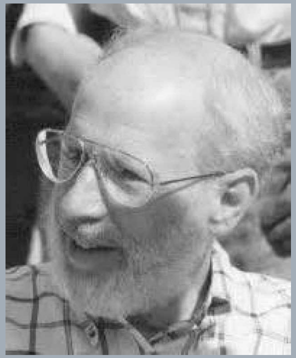
Graph Algorithms

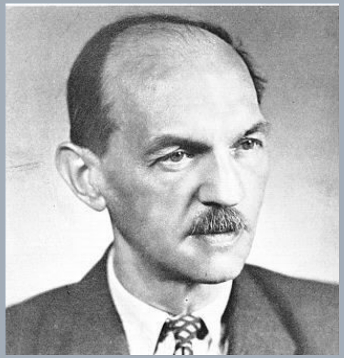
- Representation of Graphs: Adjacency Lists, Adjacency Matrices
- Graph Traversals: BFS and DFS
- Topological Sorting
- Minimum Spanning Trees, Prim-Jarník Algorithm
- SSSP, Dijkstra's Algorithm, Bellman-Ford Algorithm


Edsger Dijkstra


Lester Ford


Richard Bellman


Joseph Kruskal


Vojtěch Jarník

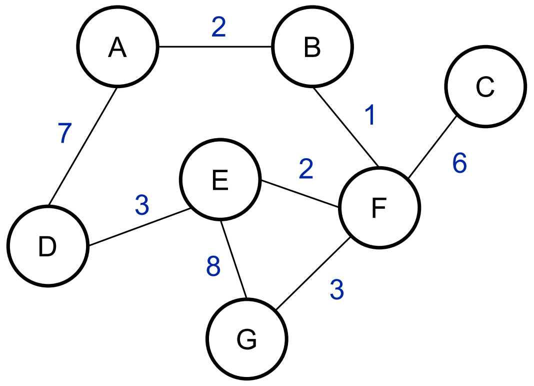
Robert Prim

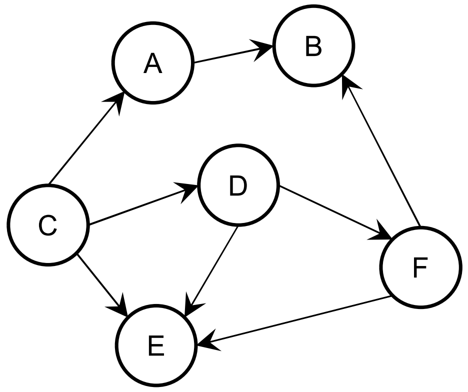
Graphs: Nomenclature and Different Types

Some of the most important properties to describe graphs are:

- directed / undirected
- weighted / unweighted
  - with / without negative weights
- connected / disconnected
- with / without cycles (acyclic)
- with / without self-edges
- with / without multiple edges between to nodes (multigraph)

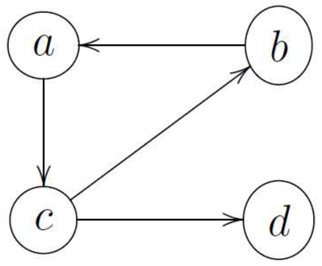
A given graph can fulfill one or more of these properties, producing a whole zoo of different kinds of graphs. For example a particularly important class are directed acyclic graphs (DAGs).

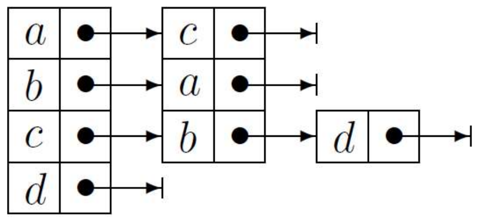




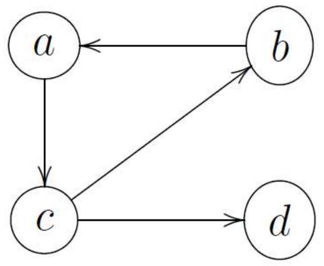
Representations of Graphs

Adjacency list





Adjacency matrix



$$\begin{array}{cc|cccc} & & \multicolumn{4}{c}{\textit{endpoint node}} \\ & & a & b & c & d \\ \hline \multirow{4}{*}{\textit{startpoint node}} & a & 0 & 0 & 1 & 0 \\ & b & 1 & 0 & 0 & 0 \\ & c & 0 & 1 & 0 & 1 \\ & d & 0 & 0 & 0 & 0 \end{array}$$

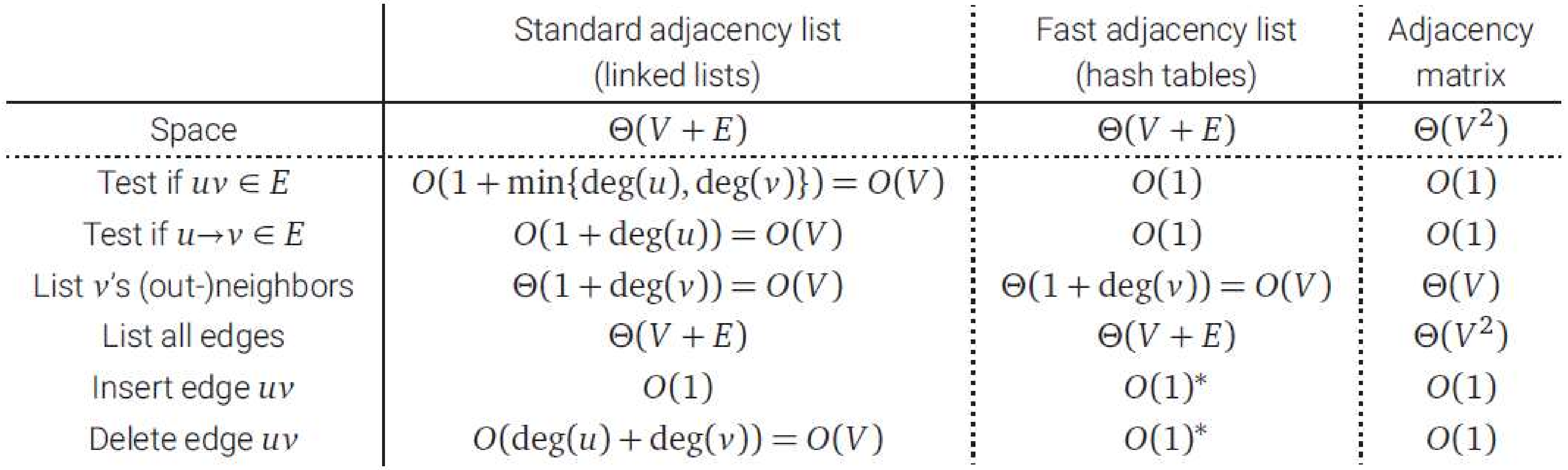
- Exercise: Give a graph with five nodes which has an adjacency list where each list has a length of exactly one.
- Comprehension questions:
  - How to distinguish between directed and undirected graphs from looking at those representations? How could such a test be implemented?
  - How would you implement a function which finds the out-degree (number of outgoing edges) for a given node? What about the in-degree?

Exercise: Adjacency List vs. Adjacency Matrix

For each of the following cases, decide whether you would use the adjacency list structure or the adjacency matrix structure. Justify your choices.

- **A:** The graph has 10'000 vertices and 20'000 edges, and it is important to use as little space as possible.
  adjacency list: The adjacency list will use 10'000 + 20'000 = 30'000 units of space whereas the adjacency matrix will use $(10'000)^2$ = 100'000'000 units of space.
- **B:** The graph has 10'000 vertices and 20'000'000 edges, and it is important to use as little space as possible.
  adjacency list: The adjacency list will use 10'000 + 20'000'000 = 20'010'000 units of space whereas the adjacency matrix will use $(10'000)^2$ = 100'000'000 units of space.
- **C:** The graph has 10'000 vertices and you need to answer the query «isAdjacentTo» as fast as possible, no matter how much space you use.
  adjacency matrix: Each list of neighours in the adjacency list has a length of at most 10'000, so a query «isAdjacentTo» requires a scan through a list of this length in the worst case. In contrast, the respective query using a adjacency matrix takes constant time.

Adjacency Matrices vs. Adjacency Lists: Asymptotic Complexities



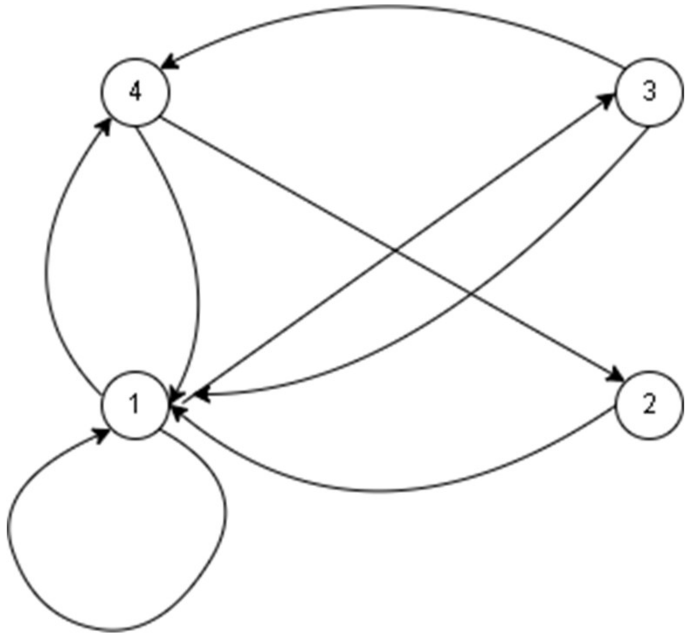
	Standard adjacency list (linked lists)	Fast adjacency list (hash tables)	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Test if $uv \in E$	$O(1+\min\{\deg(u),\deg(v)\}) = O(V)$	$O(1)$	$O(1)$
Test if $u{\to}v \in E$	$O(1+\deg(u)) = O(V)$	$O(1)$	$O(1)$
List v's (out-)neighbors	$\Theta(1+\deg(v)) = O(V)$	$\Theta(1+\deg(v)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)^*$	$O(1)$
Delete edge uv	$O(\deg(u)+\deg(v)) = O(V)$	$O(1)^*$	$O(1)$

Universität Zürich^UZH

Institut für Informatik

Exercise 12, Task 1.3

Show adjacency matrix of the given graph.



$$\begin{bmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Exercise 12, Task 1.1

For an undirected graph G with n vertices and m edges, which statement is correct?

– A. if $m < n$, G is unconnected

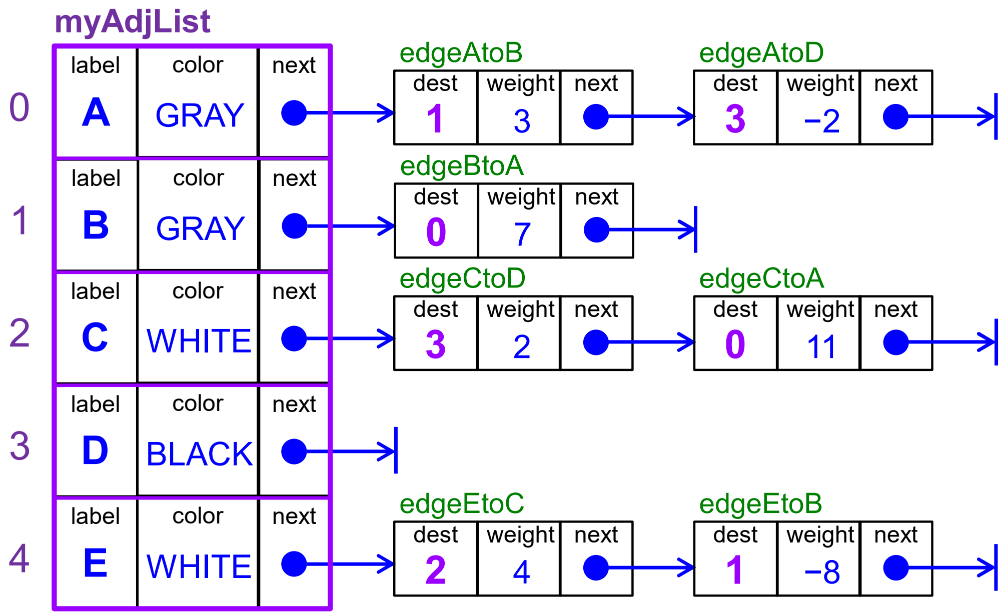
– B. if $m \geq n$, there is a loop in G

– C. if $m > n$, G is connected

– D. if $m < n$, there is no loop in G

→ **B**

Implementation of a Graph in C



```
struct AdjacencyListNode {
    int destination;
    int weight;
    struct AdjacencyListNode* next;
};

struct GraphVertex {
    char label;
    int color;
    struct AdjacencyListNode* head;
};

struct Graph {
    struct GraphVertex* adjacency_list;
    int number_of_vertices;
};
```

Comprehension Question

Why do we represent graphs using adjacency lists or adjacency matrices? Why not just model them with structs which are interlinked with pointers as we do it with binary trees for example?

Additional Exercise: Find Cycles, Check Connectivity

– Write a C program that checks whether a graph contains a loop.

– Write a C program that checks whether a graph is connected.

Graph Traversals: Introduction

There are two basic ways how the nodes in a graph can be visited:

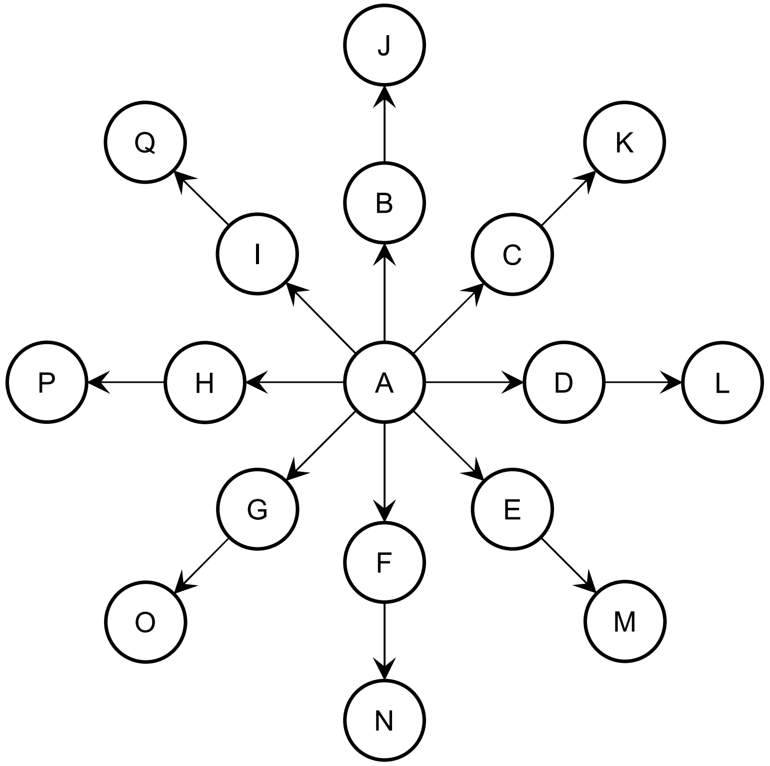
- **Breadth-first search (BFS):** starting at some selected node, explore all neighboring nodes first and only if there are no unvisited neighbours left, proceed to do the same with the neighbours of the neighbours.
- **Depth-first search (DFS):** starting at some selected node, explore neighboring nodes as far as possible until a dead end (i.e. node with no more unvisited neighbours) is reached.

Both algorithms can be applied on directed or undirected graphs and will ignore any weights.

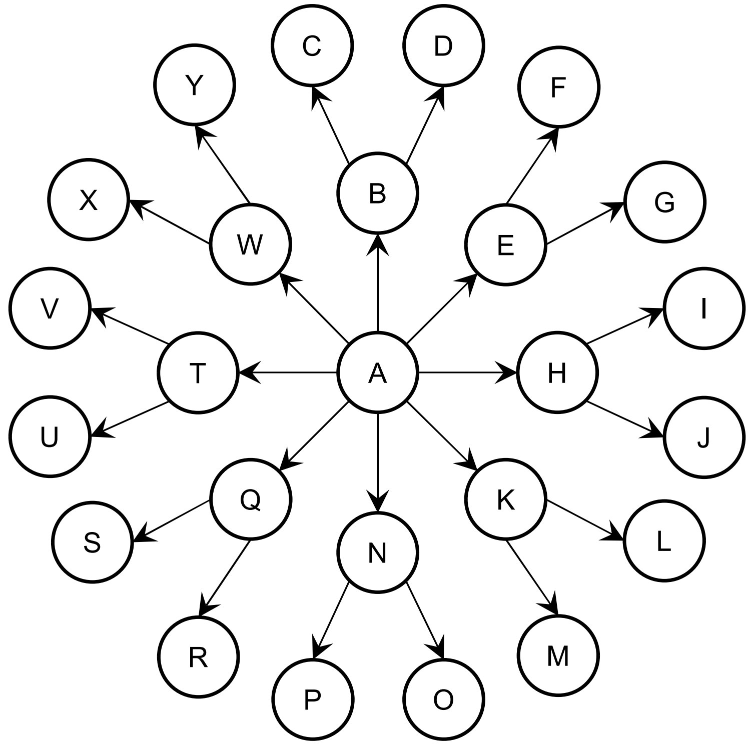
Remark: Tree traversals preorder, inorder and postorder are examples of depth-first search applied on a binary tree. A levelorder tree traversal is an example of a breadth-first search applied on a binary tree.

Breadth-First Search: Analogy

BFS expands like the waves around a stone thrown into a pond.


$$\begin{tikzcd}
 & & & J & & & \\
 & Q & & & & K & \\
 & & & B \arrow[uu] & & & \\
 & & I \arrow[uul] & & C \arrow[uur] & & \\
P & H \arrow[l] & & A \arrow[ll] \arrow[rr] \arrow[u] \arrow[ul] \arrow[ur] \arrow[dl] \arrow[d] \arrow[dr] & & D \arrow[r] & L \\
 & & G \arrow[ddl] & F \arrow[dd] & E \arrow[ddr] & & \\
 & & & & & & \\
 & O & & N & & M &
\end{tikzcd}$$

Depth-First Search: Principle



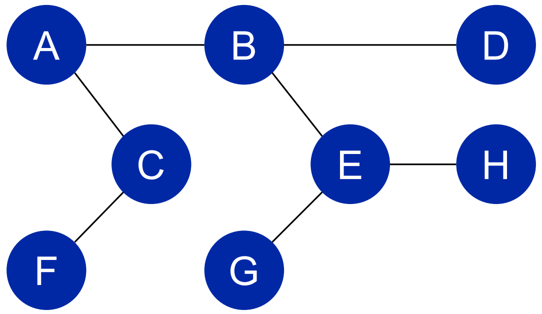
Graph Traversals: Book-Keeping Information

For some particular applications which we will see later, we will keep track of additional information while traversing the graph:

- For both BFS and DFS: color, predecessor
- BFS: distance (= number of edges traversed from starting node)
- DFS: start time, end time

Breadth-First Search: Example

– What is the order in which nodes are visited in the following graph, when applying BFS and starting at node A?



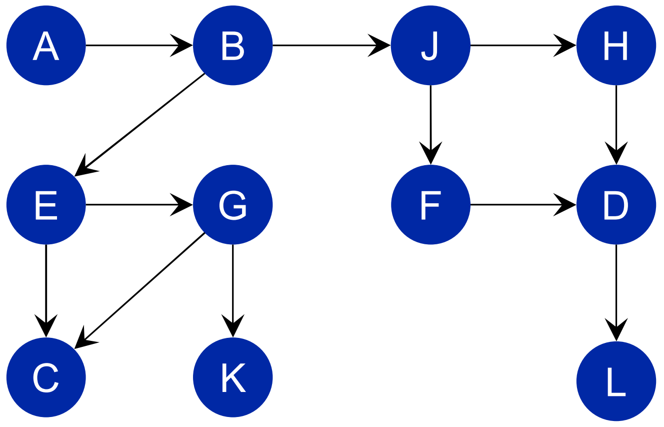
– Comprehension question: What other name could you assign to this particular way of passing through this graph?

BFS Algorithm

```
Algo: BFS(G,s)
foreach $v \in G.V$ do
    v.col = W; v.dist = $\infty$; v.pred = NIL
s.col = G; s.dist = 0;
InitQueue(Q);
Enqueue(Q,s);
while $Q \neq \emptyset$ do
    v = Dequeue(Q);
    foreach $u \in v.adj$ do
        if $u.col$==$W$ then
            u.col = G; u.dist = v.dist+1; u.pred = v; Enqueue(Q,u)
    v.col = B;
return (m,c);
```

Depth-First Search: Example

– What is the order in which nodes are visited in the following graph, when applying DFS and starting at node A?



DFS Algorithm

```
**Algo:** DFS-Tree(G,s)
---
s.col = G; time = time+1; s.time = time;
foreach $u \in s.adj$ do
    if $u.col$==$W$ then
        u.pred = s;
        DFS-Tree(G,u)
s.col = B; time = time+1; s.etime = time;
```

DFS Algorithm

```
Algo: DFS(G)
foreach $v \in G.V$ do
    v.col = W; v.pred = NIL
time = 0;
foreach $v \in G.V$ do
    if $v.col$==$W$ then DFS-Tree(G,v);
```

Breadth-First Search and Depth-First Search

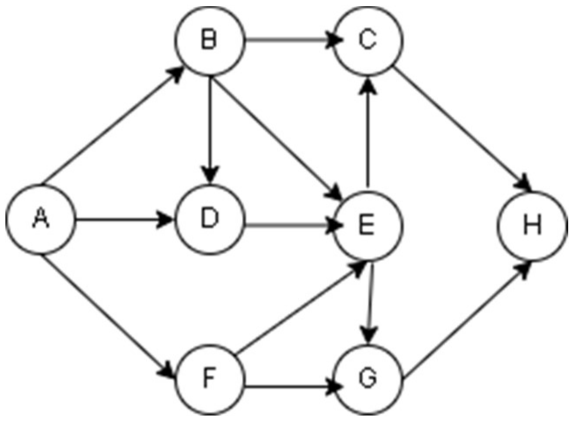
	Breadth-first search	Depth-first search
Principle	Considers all directions	Persues one direction
Results	Single tree (distance of each visited node to the starting node)	Set of trees (start time, end time, edge classification)
Data structure	Queue	Stack
Complexity	O(|V| + |E|)	O(|V| + |E|)
Applications	Finding shortest paths	Cycle detection, topological sort, planarity test

Exercise 12, Task 1.2

A possible sequence of Depth-First search on the graph is…

- A. ABDFCEGH
- B. ABCHDEGF
- C. ADECHBFG
- D. AFBDCEGH

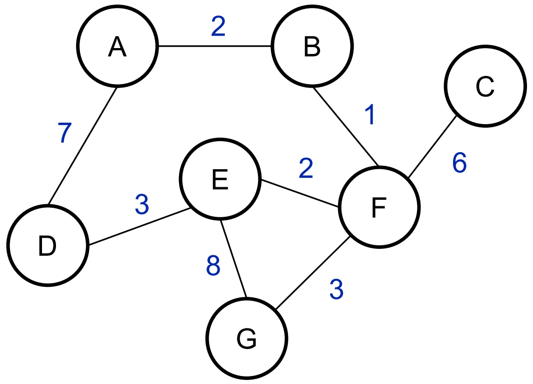
→ **B, C**

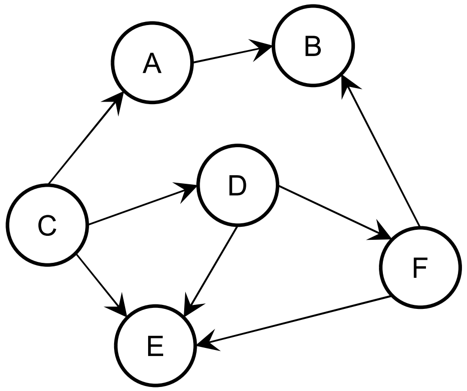

$$\begin{tikzcd}
 & B \arrow[r] \arrow[d] \arrow[dr] & C \arrow[drr] & & \\
A \arrow[ur] \arrow[r] \arrow[dr] & D \arrow[r] & E \arrow[u] \arrow[d] & & H \\
 & F \arrow[ur] \arrow[r] & G \arrow[urr] & &
\end{tikzcd}$$

Spanning Trees

A spanning tree of a graph is a tree (a graph with no cycles) which contains all nodes of a graph.

Both BFS and DFS traversals of a graph produce (by considering the edges which were used for visiting the nodes) spanning trees (or potentially a set of trees, a forest, for unconnected graphs or for directed graphs in certain situations).

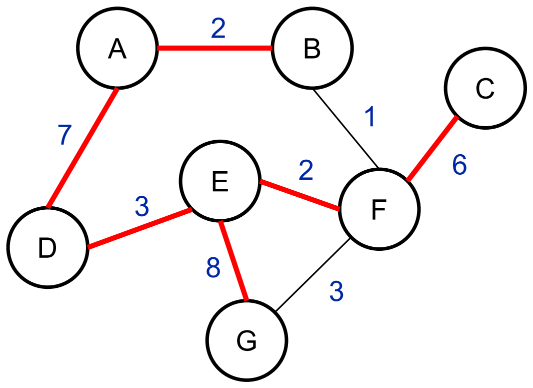


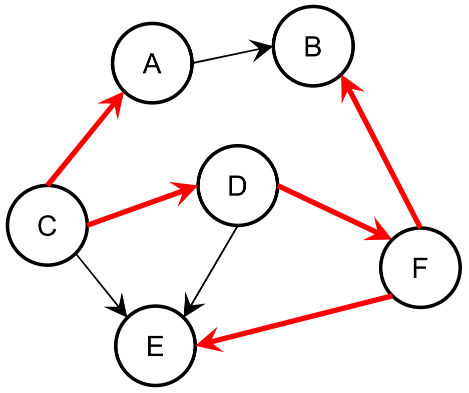


Spanning Trees

A spanning tree of a graph is a tree (a graph with no cycles) which contains all nodes of a graph.

Both BFS and DFS traversals of a graph produce (by considering the edges which were used for visiting the nodes) spanning trees (or potentially a set of trees, a forest, for unconnected graphs or for directed graphs in certain situations).





DFS: Additional Information

By applying the DFS algorithm, additional information about the nodes and edges in a graph can be gathered and stored:

- start time and end time of each node
- edge classification for each edge

This information will be valuable for various purposes as we will see later on.

DFS: Start Time and End Time

While performing the DFS traversal of a graph, we can note down for each node the occurence of two events:

- Start time: when we first visited a node (encountered / started a node, made it grey, put the node in the stack)
- End time: when we last visited a node (finished a node, made it black, popped it from the stack)

Remark:

- «time» is a bit odd since not any actual measured time in the physical sense is meant but just the sequence / position / counter

Edge Classification with DFS

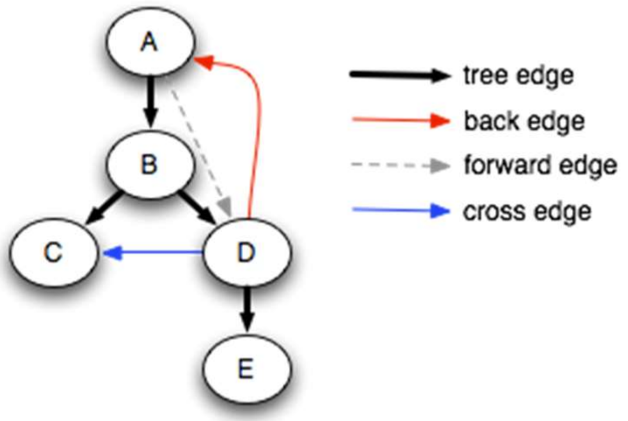
With regard to the spanning tree produced by the DFS algorithm, we can distinguish four different types of edges:

– tree edges: from a parent to a child, used for visiting nodes in DFS algorithm, edges forming the spanning tree produced by DFS algorithm

– forward edges: to a non-child descendant / to a finished vertex discovered after the current vertex

– back edges: to an ancestor / to a discovered but unfinished node

– cross edges: everything else (i.e. to a vertex finished before the current vertex's discovery); cross edges connect independent subtrees



Universität Zürich^UZH

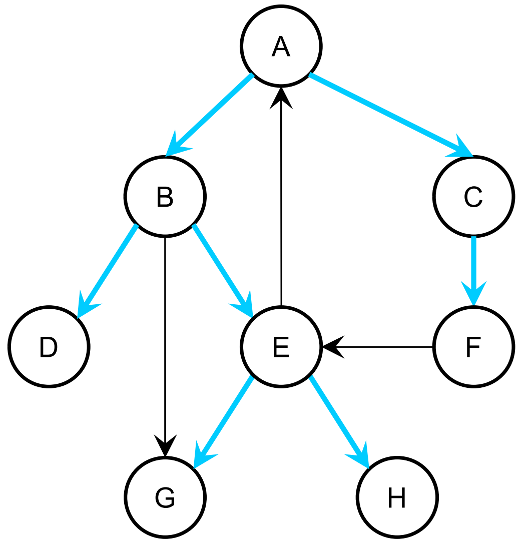
Institut für Informatik

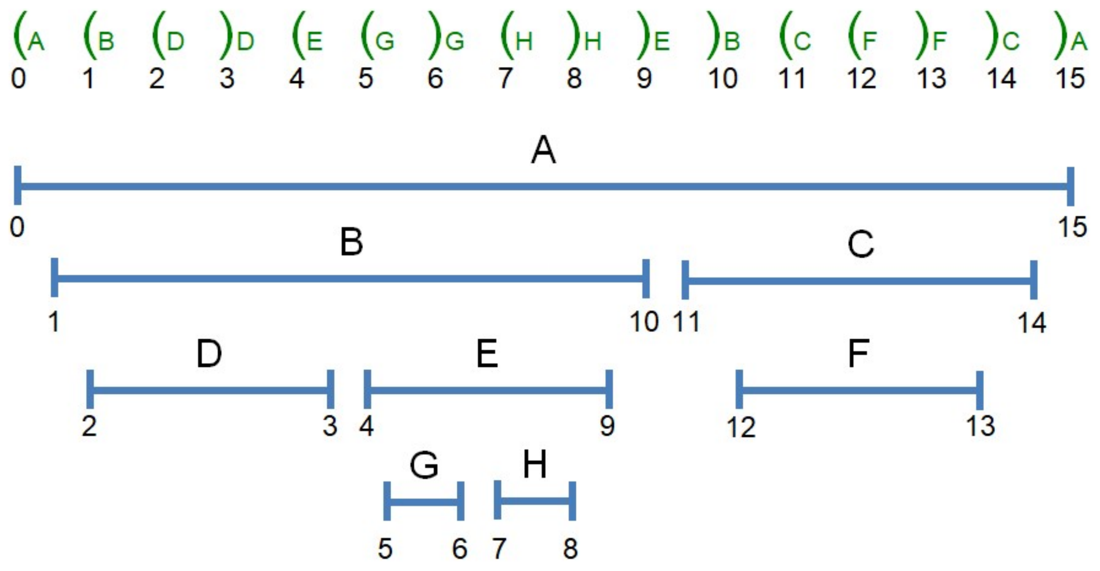
Examples / Exercises Edge Classification

$$\begin{tikzcd}
A \arrow[rr] \arrow[dr] & & C \arrow[rr] & & F \arrow[dl] \\
& B \arrow[rr] \arrow[dl] \arrow[dr] & & E \arrow[ulll] \arrow[dl] \arrow[dr] & \\
D & & G & & H
\end{tikzcd}$$

$$\begin{tikzcd}
A \arrow[drr] & & C \arrow[ll] \arrow[r, bend left] & D \arrow[l, bend left] & E \arrow[l] \\
& & B \arrow[u] \arrow[ur] \arrow[urr] & F \arrow[ur] \arrow[l] \arrow[dll] & \\
& G \arrow[rr] \arrow[uul] & & H \arrow[u] & J \arrow[l] \arrow[lll, bend left]
\end{tikzcd}$$

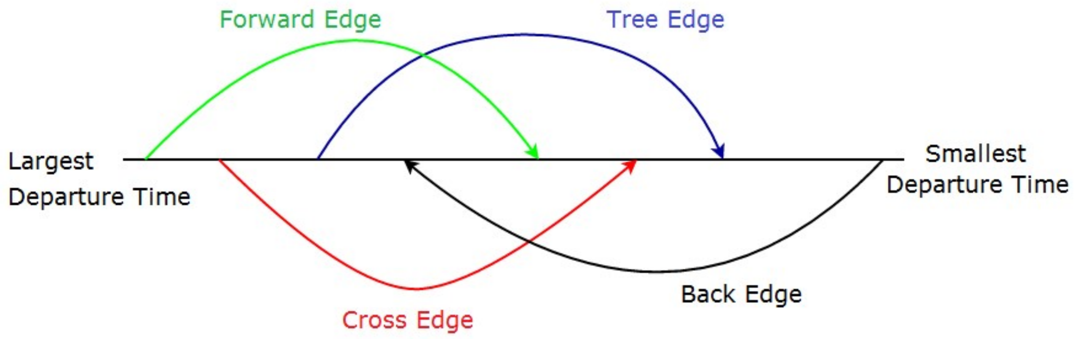
Examples / Exercises Edge Classification: Solution





Edge Classification, Start and End Times

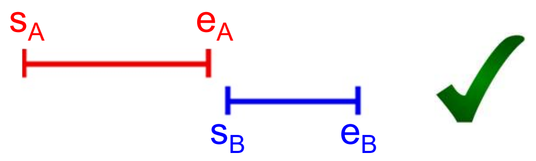
- How can edge detection be implemented? And how do edge classification and start / end time relate to eachother?



- These considerations also point to the parentheses theorem (and they are also important for topological sorting)…

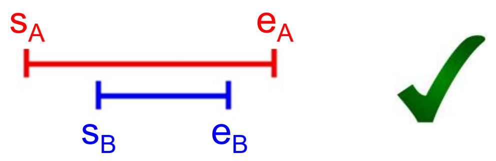
Parentheses Theorem

This means that, for two nodes A and B, the intervals as given by the start and end times of a node, $[s_A, e_A]$ and $[s_B, e_B]$, must be in one of the following two configurations:



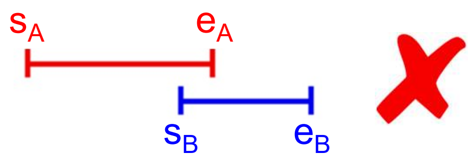
disjoint intervals

- The intervals of two nodes A and B are entirely disjoint.
- Start and end times of the second interval of node B are both encountered *after* the intervall of the first node A has finished.



nested intervalls

- The intervals of one node (here: A) is contained entirely within the interval of another node (here: B).
- The inner interval B is encountered after A has started and B ends before A ends.

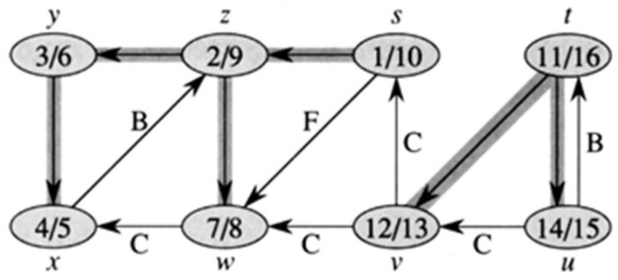


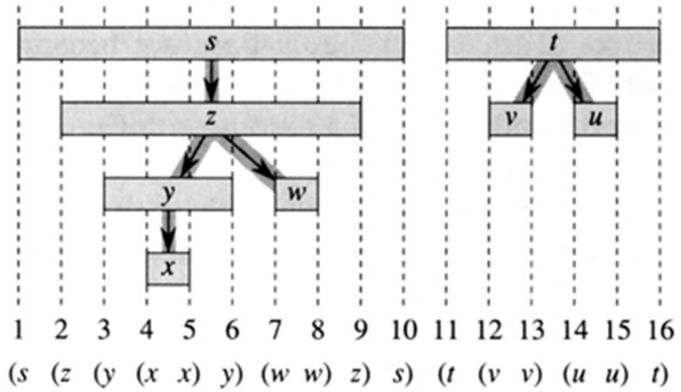
overlapping intervals

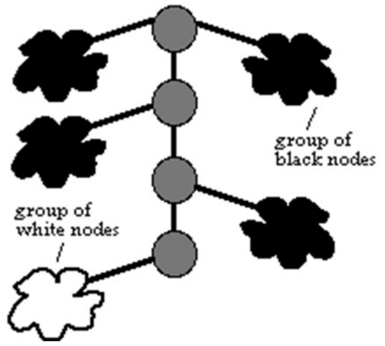
- The two intervals of two nodes overlap each other.
- The second interval B starts before the first A has ended but B finishes after A.
- This configuration cannot occur.

Parentheses Theorem

The parentheses theorem can be derived directly from the properties of the spanning tree.







DFS Application: Cycle Detection

DFS provides an easy way to determine whether a given graph has a cycle: If the spanning tree produced by the DFS traversal contains a back edge, the graph contains a cycle.

Topological Sorting: Introduction and Example

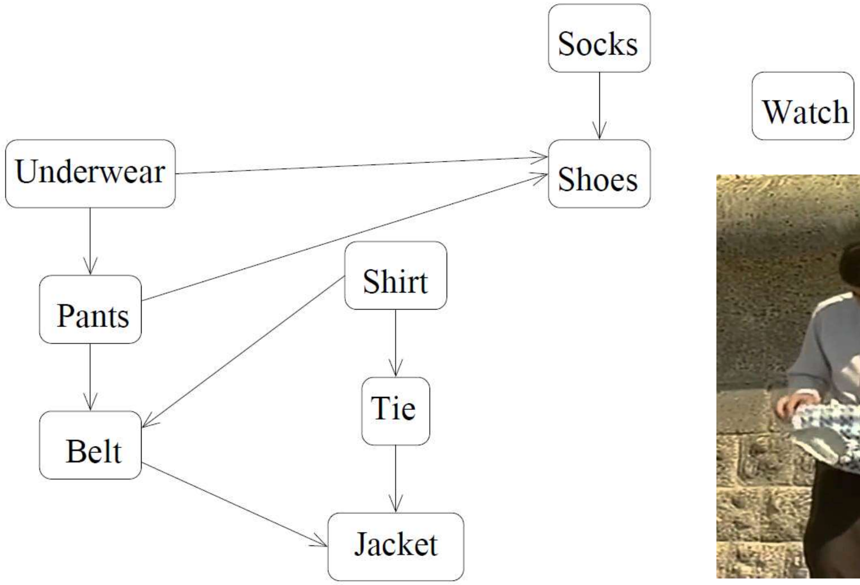
– Any directed acyclic graph (DAG) has a topological order (and normally even many different ones).

– If a graph has a cycle, a topological order cannot exist.

Example: How to get dressed in the morning

– The adjacent graph shows the dependencies of which pieces of clothing have to be put on necessarily before other ones. For example, the socks have to be put on before the shoes are put on.

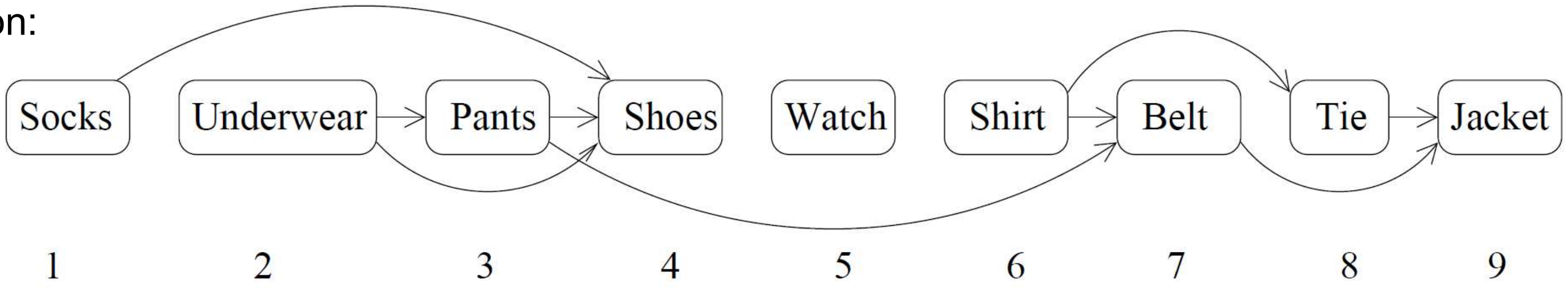
– In which order can the clothes be put on in accordance with this graph?



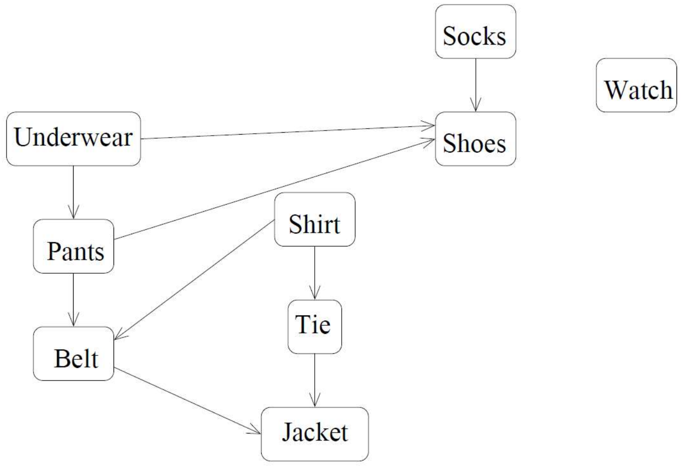
$$\begin{tikzcd}
 & & & \text{Socks} \arrow[d] & \text{Watch} \\
\text{Underwear} \arrow[rrr] \arrow[d] & & & \text{Shoes} & \\
\text{Pants} \arrow[rrru] \arrow[d] & & \text{Shirt} \arrow[lld] \arrow[d] & & \\
\text{Belt} \arrow[rrd] & & \text{Tie} \arrow[d] & & \\
 & & \text{Jacket} & &
\end{tikzcd}$$

Topological Sorting: Example: How to Get Dressed in the Morning

– Possible solution:



– Other solutions are possible.



Topological Sorting: Implementation with DFS

Topological order of a graph G can be established using the depth-first search (DFS) algorithm:

- Use DFS(G) to get start and end times for all vertices in G.
- Return list of vertices in reverse order of end times.

Additional Exercise: Islands / Ecosystems

(Task 3 of assignment 6 from FS 2018)

You are given a binary 2D matrix M, each element of the matrix represents an island, islands with value 0 are uninhabited and islands with value 1 are inhabited. Each island represented by a cell in the matrix can have up to 8 neighbors (north, east, west, south and 4x diagonally). A group of neighboring inhabited islands form an ecosystem. The goal is to find the number of distinct ecosystems in the archipelago of islands represented by the 2D matrix M.

1	1	0	0	1
0	1	0	0	0
1	0	0	1	1
0	0	0	0	0
1	0	1	0	1

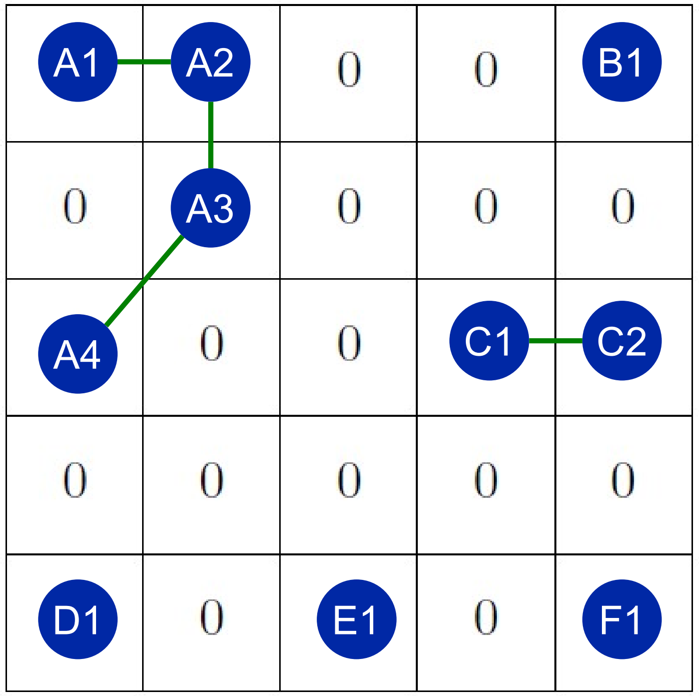
Additional Exercise: Islands / Ecosystems

(Task 3 of assignment 6 from FS 2018)

You are given a binary 2D matrix M, each element of the matrix represents an island, islands with value 0 are uninhabited and islands with value 1 are inhabited. Each island represented by a cell in the matrix can have up to 8 neighbors (north, east, west, south and 4x diagonally). A group of neighboring inhabited islands form an ecosystem. The goal is to find the number of distinct ecosystems in the archipelago of islands represented by the 2D matrix M.

Idea for solution:

Call DFS on each grid position if there is a value of 1 and the node was not yet visited by another call of DFS. Count the number of times, DFS was called.



Additional Exercise: Islands / Ecosystems

```c
#include <stdio.h>
#include <string.h>

#define ROW 5
#define COL 5

static int M[ROW][COL] = {
    {1, 1, 0, 0, 1},
    {0, 1, 0, 0, 0},
    {1, 0, 0, 1, 1},
    {0, 0, 0, 0, 0},
    {1, 0, 1, 0, 1}
  };

int isSafe(int row, int col, int visited[ROW][COL]) {
  return (row >= 0) && (row < ROW) &&
         (col >= 0) && (col < COL) &&
         (M[row][col] && !visited[row][col]);
}
```

```c
void DFS(int row, int col, int visited[ROW][COL]) {
  int k;
  int rowNbr[] = {-1, -1, -1,  0, 0,  1, 1, 1};
  int colNbr[] = {-1,  0,  1, -1, 1, -1, 0, 1};
  visited[row][col] = 1;

  for (k = 0; k < 8; ++k)
    if (isSafe(row + rowNbr[k], col + colNbr[k], visited) ) {
      DFS(row + rowNbr[k], col + colNbr[k], visited);
  }
}
```

```c
int countEcosystems() {
  int i;
  int j;
  int visited[ROW][COL];
  memset(visited, 0, sizeof(visited));

  int count = 0;
  for (i = 0; i < ROW; ++i)
    for (j = 0; j < COL; ++j)
      if (M[i][j] && !visited[i][j]) {
        DFS(i, j, visited);
        ++count;
      }
  return count;
}
```

```c
void main() {
  printf("Number of ecosystems is: %d\n", countEcosystems());
}
```

Overview of Some DFS and BFS Applications

DFS applications:

- Cycle detection
- Topological sorting
- Connected components identification/counting
- (Planarity test)

BFS applications:

- Calculate the distance from a start node to each other (reachable) node (e.g.: Erdős number)
- (With extensions: finding shortest paths as part of Dijkstra’s algorithm)

Algorithm of Prim and Jarník

– Finds a minimum spanning tree (MST) for a given graph. Operates on weighted trees (directed or undirected).

– *Steps:*

(1) Select an arbitrary node as starting node.

(2) Repeatedly follow the least costly edge from the set of nodes which have already been visited until all nodes have been visited.

– The algorithm grows a tree from the starting node. It is a node oriented algorithm.

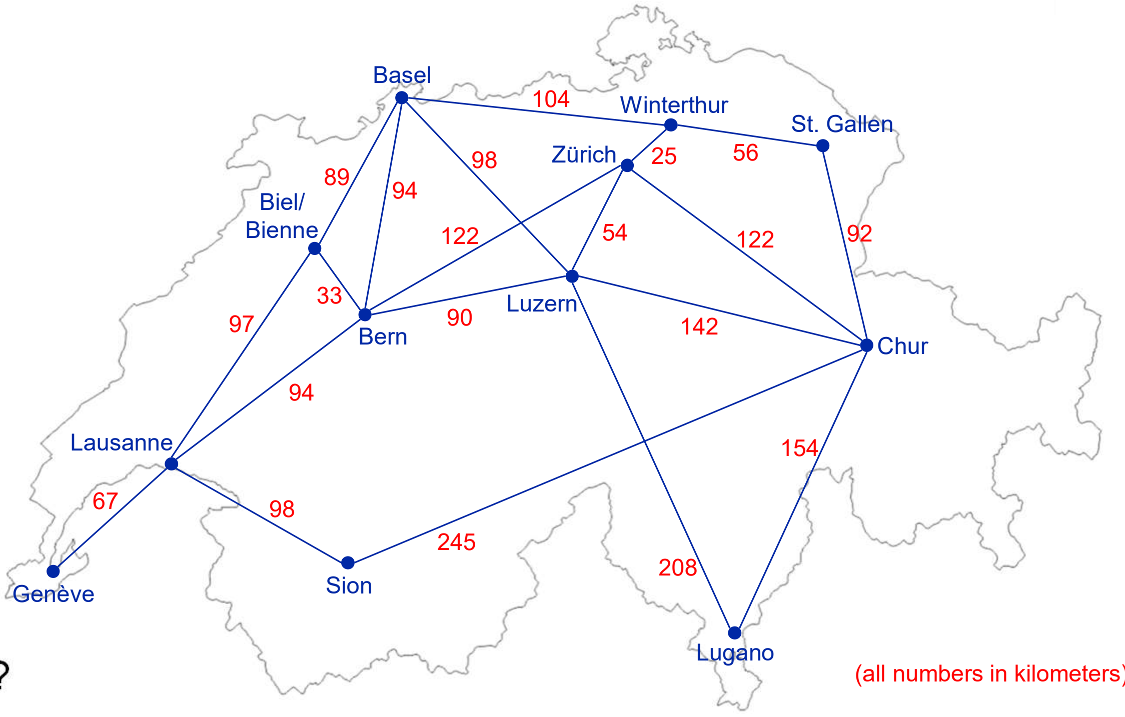
– Remark: There is another algorithm for this purpose called Kruskal’s algorithm which instead creates the minimum spanning tree by selecting edges with the smallest weight from the set of all edges (not considering whether their nodes have already been visited). Kruskal’s algorithm is edge oriented.

Prim-Jarník Algorithm: Example / Exercise

Considering the adjacent, simplified distance map of Switzerland, use the algorithm of Prim and Jarník to find a minimum spanning tree (MST).

Additional questions:

- What could be a possible use case for a minimum spanning tree like the one created in this task?
- Will the resulting MST always be the same, indepent on which node is used as start node (not only for this graph, but for arbitrary graphs)?



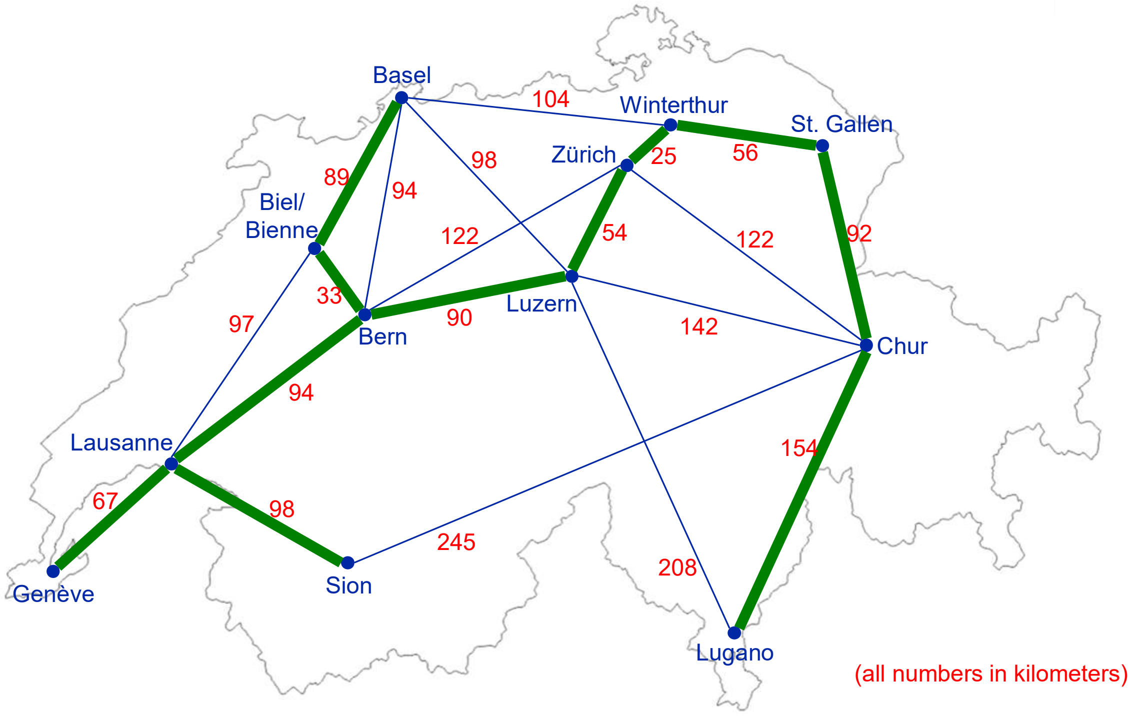
(all numbers in kilometers)

Prim-Jarník Algorithm: Example / Exercise

Considering the adjacent, simplified distance map of Switzerland, use the algorithm of Prim and Jarník to find a minimum spanning tree (MST).

Sequence of choosing the edges when starting in Zürich:

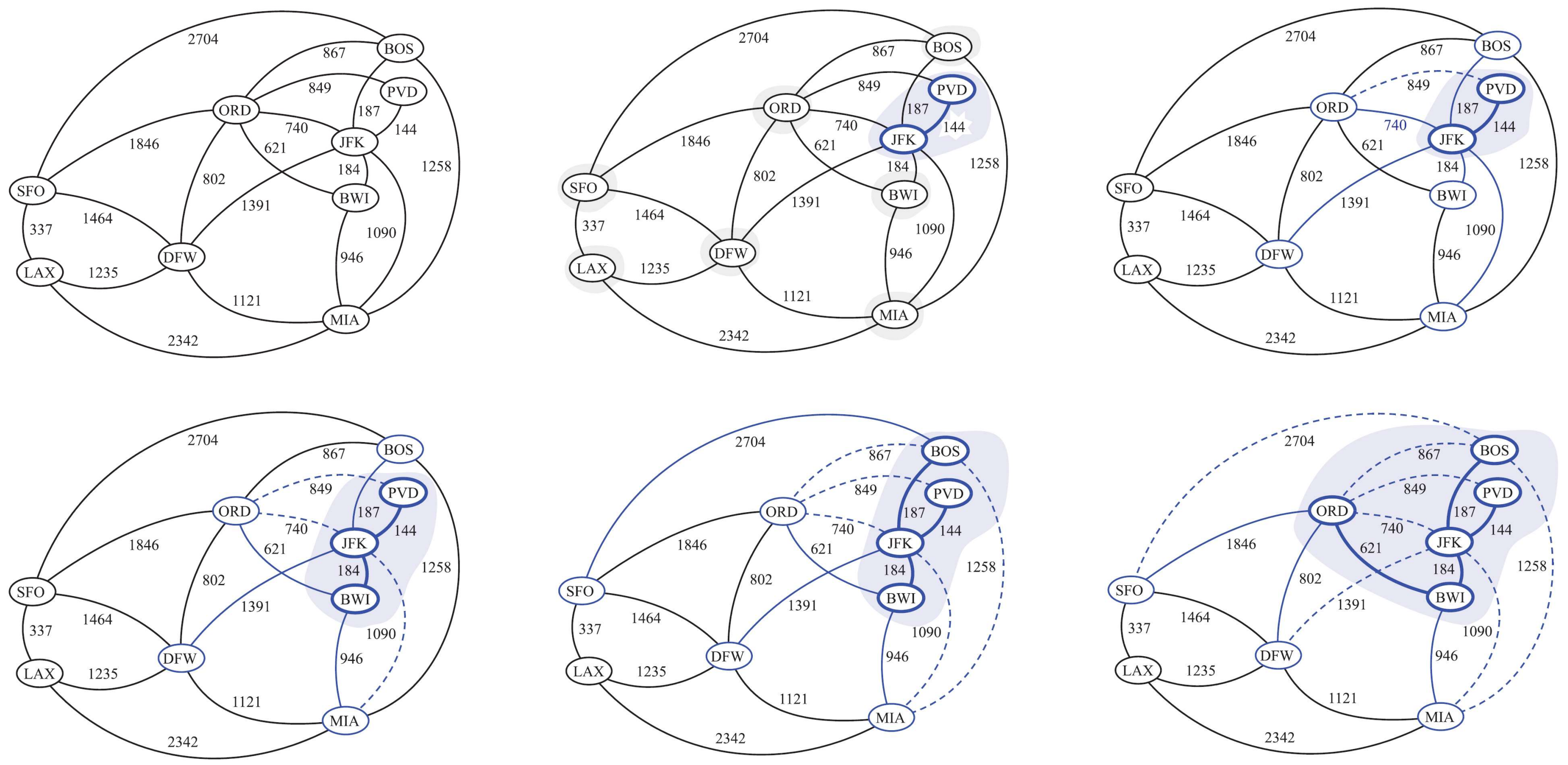
1) Zürich – Winterthur (25 km)
2) Zürich – Luzern (54 km)
3) Winterthur – St. Gallen (56 km)
4) Luzern – Bern (90 km)
5) Bern – Biel/Bienne (33 km)
6) Biel/Bienne – Basel (89 km)
7) St. Gallen – Chur (92 km)
8) Bern – Lausanne (94 km)
9) Lausanne – Genève (67 km)
10) Chur – Lugano (154 km)



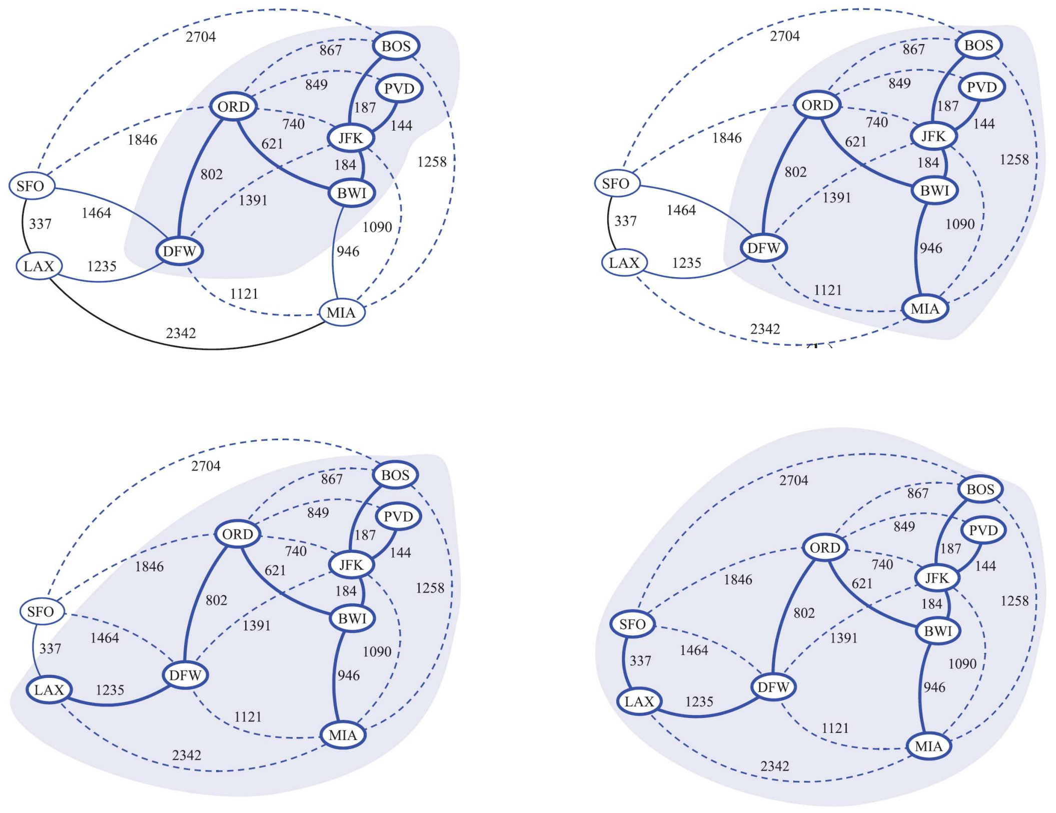
Universität Zürich^UZH

Institut für Informatik

Prim-Jarník Algorithm Visualized



Prim-Jarník Algorithm Visualized



from: Michael T. Goodrich, Roberto Tamassia, David M. Mount: Data Structures and Algorithms in C++ (second edition, 2011). John Wiley & Sons; p. 652f.

Algorithm of Prim and Jarník: Implementation

Algo: PrimJarnik(G,w,s)

```
foreach $v \in G.V$ do
    v.key = $\infty$;
    v.pred = NIL;
s.key = 0;
InitMinPriorityQueue(PQ,G.V);
while $PQ \neq \emptyset$ do
    v = ExtractMin(PQ);
    foreach $u \in v.adj$ do
        if $u \in PQ \land w(v,u) < u.key$ then
            u.key = w(v,u);
            u.pred = v;
            DecreaseKey(PQ,u,w(u,v))
```

Exercise 12, Task 1.4

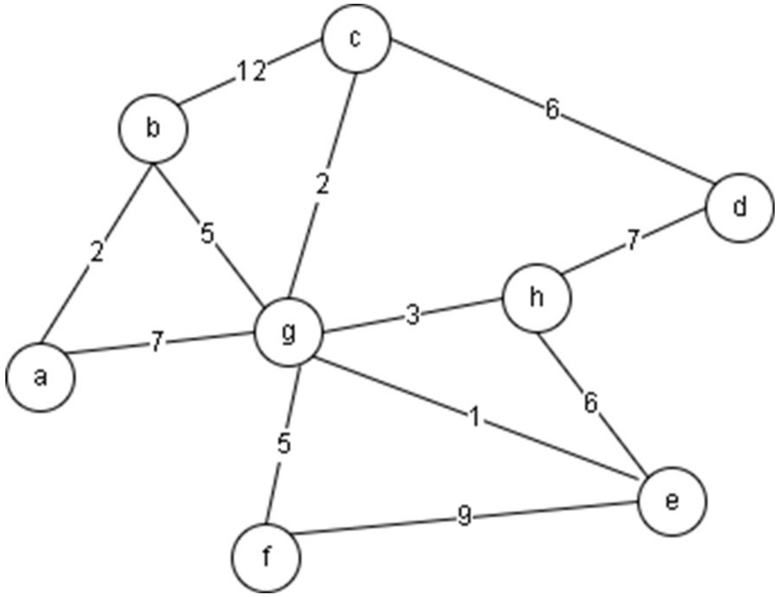
For an undirected weighted graph, its minimum spanning tree may not exist, but if it exists, it might not be unique.

- A. True
- B. False

$\rightarrow$ **A**

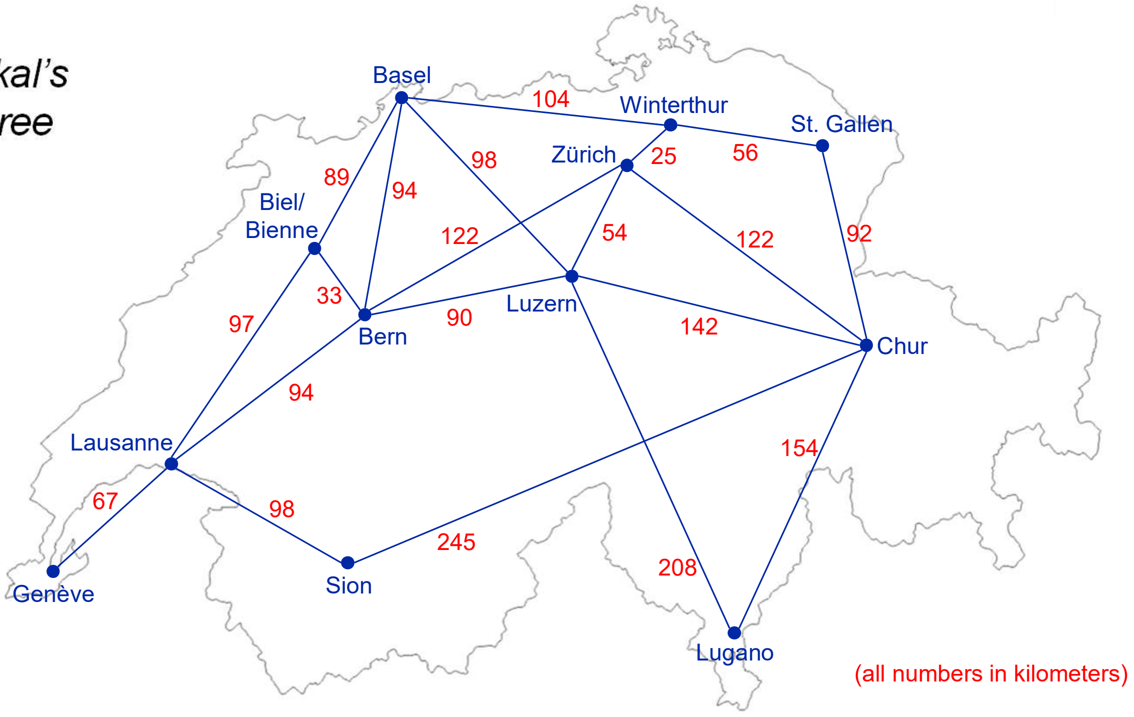
Exercise 12, Task 1.5

Use the Prim-Jarnik algorithm to compute to compute a Minimum Spanning Tree for the graph below (start from node 'a').



Kruskal’s Algorithm: Example / Exercise

Considering the adjacent, simplified distance map of Switzerland, use Kruskal’s algorithm to find a minimum spanning tree (MST).

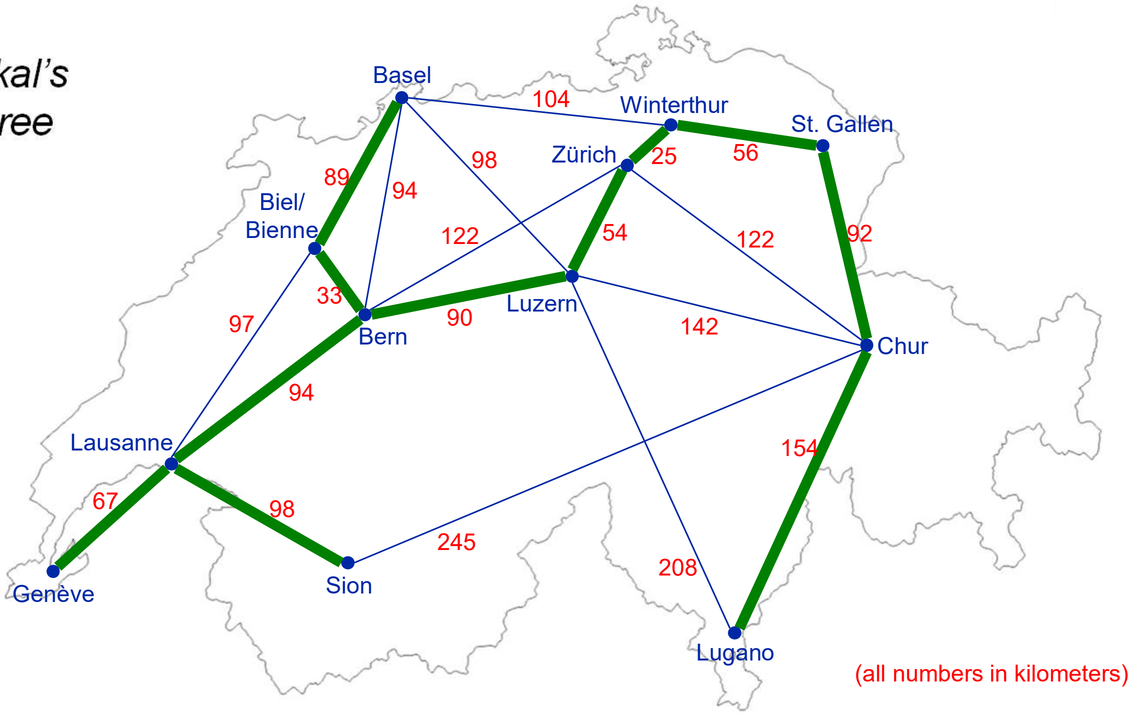


Kruskal’s Algorithm: Example / Exercise

Considering the adjacent, simplified distance map of Switzerland, use Kruskal’s algorithm to find a minimum spanning tree (MST).

Sequence of choosing the edges:

1) Winterthur – Zürich (25 km)
2) Bern – Biel/Bienne (33 km)
3) Winterthur – St. Gallen (56 km)
4) Genève – Lausanne (67 km)
5) Basel – Biel/Bienne (89 km)
6) Bern – Luzern (90 km)
7) St. Gallen – Chur (92 km)
8) Bern – Lausanne (94 km)
9) Lausanne – Sion (98 km)
10) Chur – Lugano (154 km)



Dijkstra’s Algorithm: Introduction

- «Dijkstra = greedy with second thoughts»
- Greedy: always does what currently looks best
- Finds shortest paths from a given start node to all other nodes (SSSP)

- Idea: Grows a set of finalized nodes for which the closest distance to the starting node is definitively known. Subsequently add new nodes reachable from the finalized set and than relax all edges leading to nodes from the finalized set of nodes.

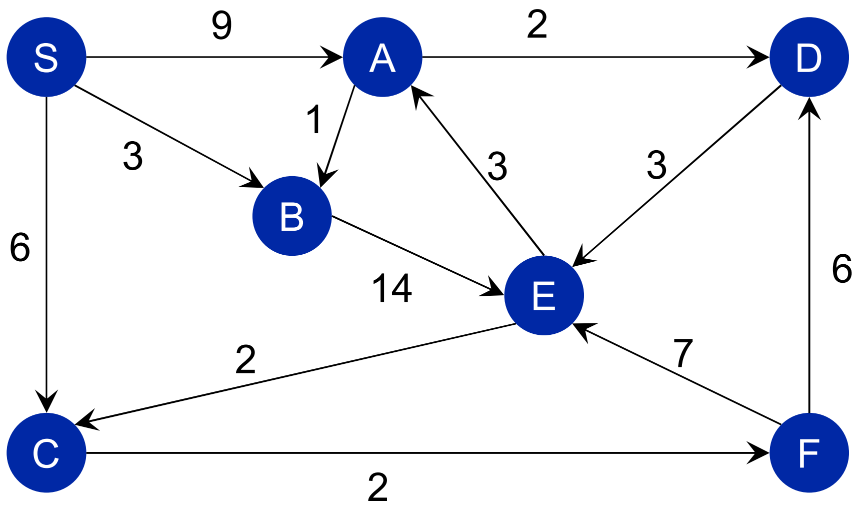
- Uses a minimum priority queue (= min heap) using their distances as keys with the following two additional operations:
  - decreaseKey(u, d): change key (distance) of node u to new value d (reorder PQ accordingly)
  - extractMin(): removes node with minimal key (distance) from the queue and returns its value

Relaxation

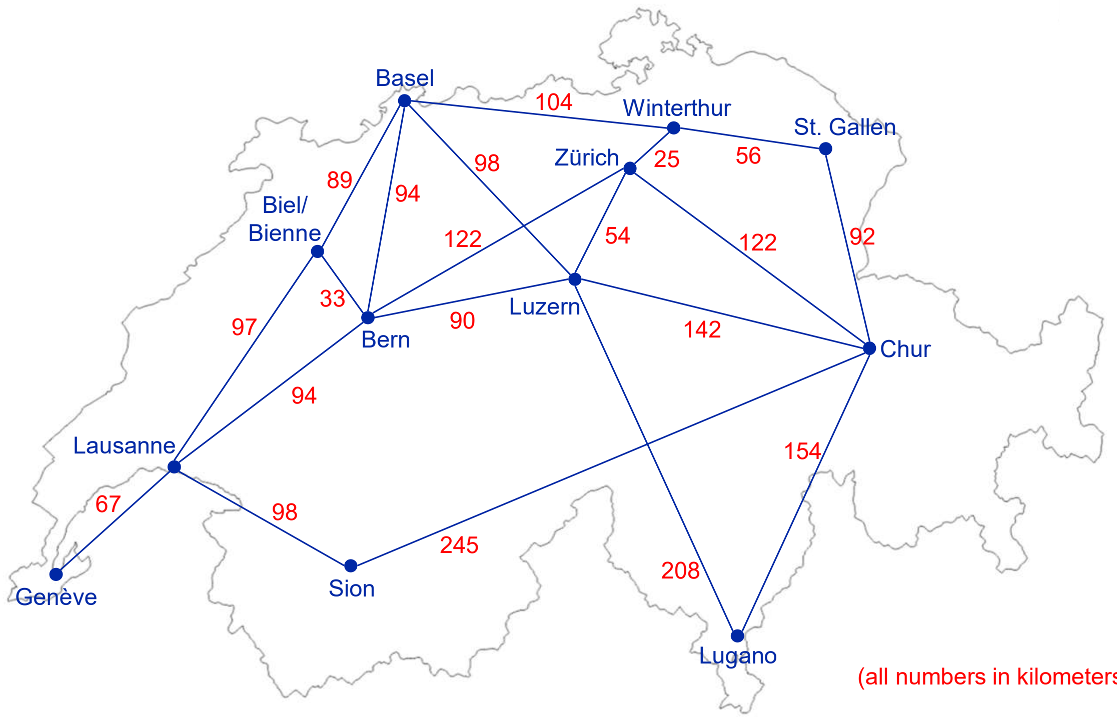
$$\begin{tikzcd}
 & A \arrow[r] & B \arrow[r] & C \arrow[dr] & & {} \\
S \arrow[ur] \arrow[r] \arrow[dr] & F \arrow[r] & G \arrow[r] & H \arrow[r] & Q \arrow[ur] \arrow[dr] & \\
 & K \arrow[r] & L \arrow[r] & M \arrow[ur] & & {}
\end{tikzcd}$$

Dijkstra’s Algorithm: Example

Apply Dijkstra’s algorithm to find the shortest paths from node S to all other nodes:



Dijkstra’s Algorithm: Another Example



(all numbers in kilometers)

Dijkstra's Algorithm: Implementation

edge weights

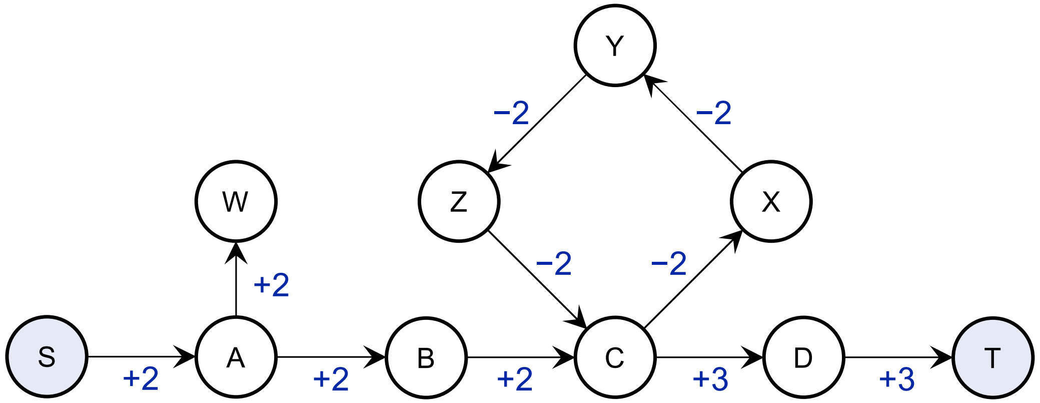
```
Algo: Dijkstra(G,w,s)
foreach $v \in G.V$ do
    v.dist = $\infty$; v.pred = NIL;
s.dist = 0;
InitMinPQ(PQ,G.V);
while $PQ \neq \emptyset$ do
    v = ExtractMin(PQ);
    foreach $u \in v.adj$ do
        Relax(u,v,w);
        DecreaseKey(PQ,u,u.dist)
```

as long as the priority queue is not empty

put all nodes into a minimum priority queue (with their distances as keys); priority queue will serve as a kind of to-do list (for all nodes which are still in the queue, the definite shortest path has not yet been found)

Is There Always a Shortest Path and Does Dijkstra Always Work?

Dijkstra’s algorithm is not applicable (will fail) for some particular graphs with negative edge weights.



- The shortest path from S to A is 2.
- The shortest path from S to B is 4.
- The shortest path from S to W is 4.
- The shortest path from S to C is −∞.
- The shortest path from S to all other nodes is −∞.

– Why is this the case? What will happen if Dijkstra’s algorithm is applied in this case?

– How can the shortest path from a single source be found in a graph with negative edge weights?

Bellman-Ford Algorithm: Implementation

```
Algo: BellmanFord(G,w,s)
foreach $v \in G.V$ do
    v.dist = $\infty$; v.pred = NIL;
for $i = 1$ to $|G.V|-1$ do
    foreach $(u, v) \in G.E$ do
        Relax(u,v,w);
foreach $(u, v) \in G.E$ do
    if $v.dist > u.dist + w(u, v)$ then
        return FALSE;
return TRUE;
```

Exercise 12, Task 2.1

Which algorithm can help us get the minimum weight from 'a' to 'z'? What is the weight?

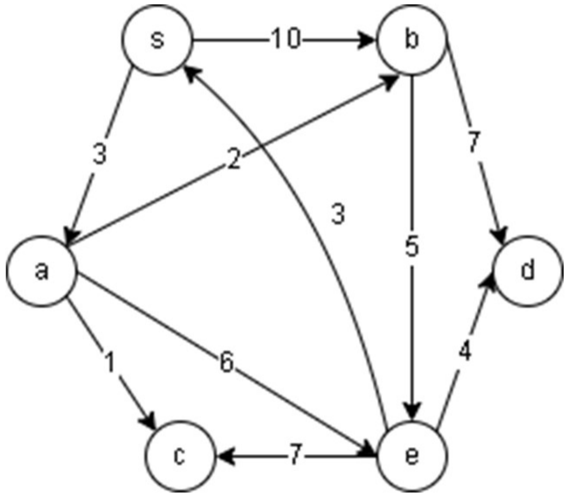
– A. Bellman-Ford 21
– B. Bellman-Ford 16
– C. Dijkstra 21
– D. Dijkstra 16

$\rightarrow$ **B**

a – b – c – l – h – g – f – z

Exercise 12, Task 2.2

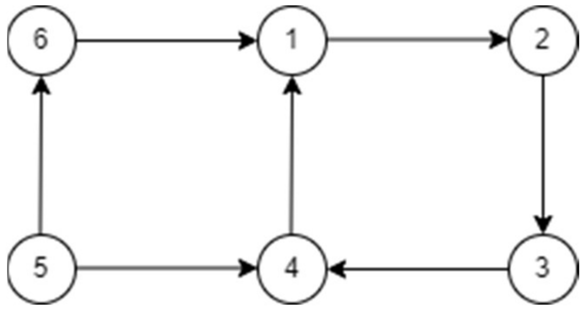
Use Dijkstra algorithm to compute shortest path from start node 's' for the graph.



Exercise 12, Task 3

A root vertex of a directed graph is a vertex u with a directed path from u to v for every pair of vertices (u, v) in the graph. In other words, all other vertices in the graph can be reached from the root vertex. Given a graph, write C code that finds the root vertex using Breadth First Search as well as Depth First Search approach. A graph can have multiple root vertices. In such cases, the solution should find all the root vertices.

For example, the root vertex is 5 since it has a path to every other vertex in the following graph:



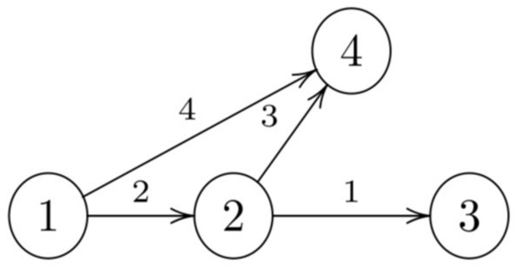
Universität Zürich^UZH

Institut für Informatik

Exercise 12, Task 4

Consider a directed weighted graph G = (V, E). Let n = |V| and m = |E|. Each vertex of V has a unique integer label between 1 and n. Each edge of E has a distinct weight between 1 and m.

The edges of E are stored in an array A. Each array element is a record with three fields: f (from), t (to), and w (weight). The array elements are sorted by the edge weight in ascending order. Note that position in A starts at 1.



$G = (V, E)$

	1	2	3	4
f	2	1	2	1
t	3	2	4	4
w	1	2	3	4

Array A representing G

Exercise 12, Task 4.1

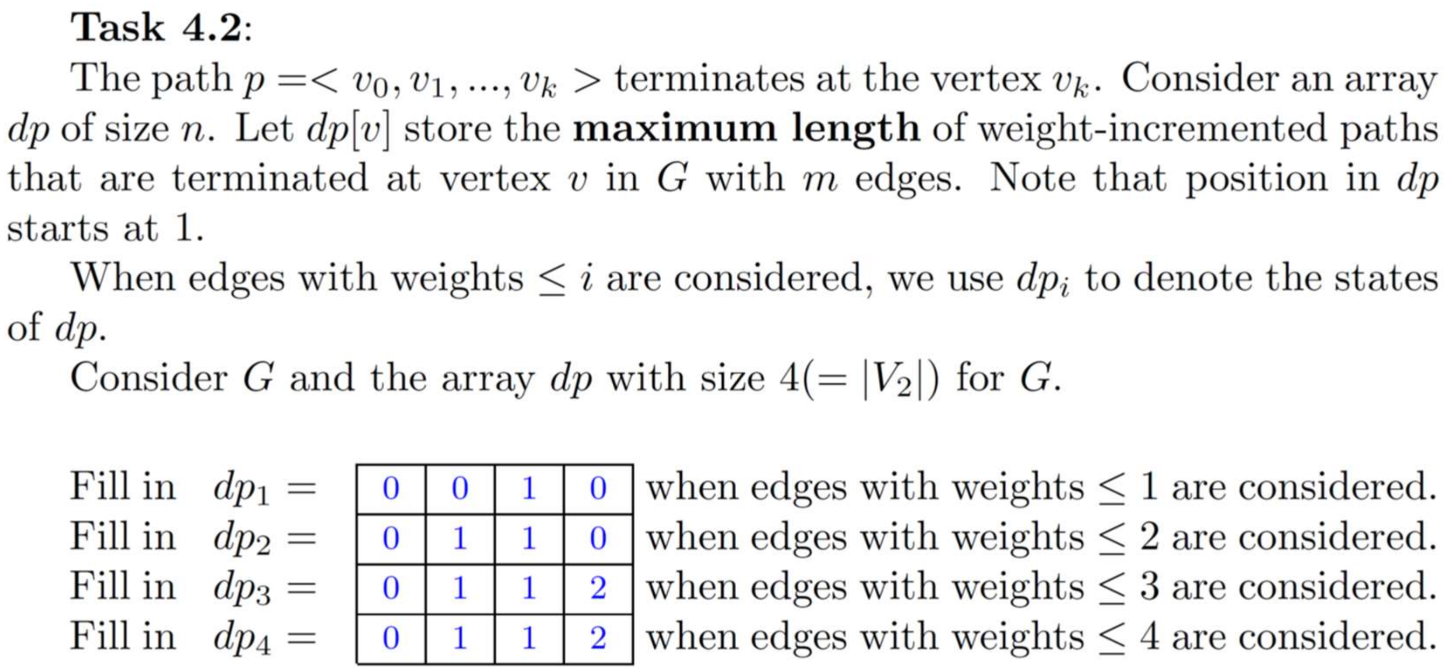
Task 4.1:

A path $p =< v_0, v_1, ..., v_k >$ is a **weight-incremented path** if and only if the weights of two neighboring edges in p are strictly increased, *i.e.*, $w(v_{i-1}, v_i) < w(v_i, v_{i+1})$ where $1 \leq i < k$. A path with only one edge is also consider as a weight-incremented path.

Determine the maximum length of weight-incremented paths in G_2.

Answer:2; 1-2-4.

Exercise 12, Task 4.2



Task 4.2:

The path $p =< v_0, v_1, ..., v_k >$ terminates at the vertex v_k. Consider an array dp of size n. Let $dp[v]$ store the **maximum length** of weight-incremented paths that are terminated at vertex v in G with m edges. Note that position in dp starts at 1.

When edges with weights $\leq i$ are considered, we use dp_i to denote the states of dp.

Consider G and the array dp with size $4(= |V_2|)$ for G.

Fill in $dp_1 =$	0	0	1	0	when edges with weights ≤ 1 are considered.
Fill in $dp_2 =$	0	1	1	0	when edges with weights ≤ 2 are considered.
Fill in $dp_3 =$	0	1	1	2	when edges with weights ≤ 3 are considered.
Fill in $dp_4 =$	0	1	1	2	when edges with weights ≤ 4 are considered.

Exercise 12, Task 4.3

Task 4.3:Determine the recursive problem formulation for $dp_i[v]$. Assume that the directed edge (a, b) has the weight i.

$$\begin{cases} dp_i[v] = dp_{i-1}[v] & \texttt{v} \neq \texttt{b} \\ dp_i[v] = max(dp_{i-1}[a] + 1, dp_{i-1}[v]) & \texttt{v = b} \end{cases}$$

Exercise 12, Task 4.3

Task 4.4: Write the pseudocode algorithm `maximum_len(`A`, `m`)` that returns the maximum length of weight-incremented paths in $G = (V, E)$ with m ($m = |E|$) edges.

Requirement: The asymptotic complexity of your solution should be $O(m)$.

Algorithm: MAXIMUM_LEN(A, m)

```
initialize an array dp[n];
for $i$ = 1 to $n$ do
    dp[i] = 0;
max_len = 0;
for $i = 1$; $i \leq m$; $i = i + 1$ do
    $u = A[i].f$
    $v = A[i].t$
    dp[$v$] = max(dp[$v$], dp[$u$] + 1)
    max_len = max(max_len, dp[$v$])
return max_len;
```

Universität Zürich^UZH

Institut für Informatik

Wrap-Up

– Summary
– Feedback
– Outlook
– Questions

Wrap-Up

– Summary

Questions?

Thank you for your attention.