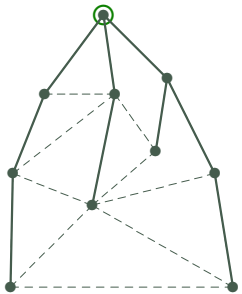
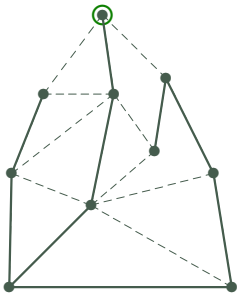
2 Graph Searching and Connectivity





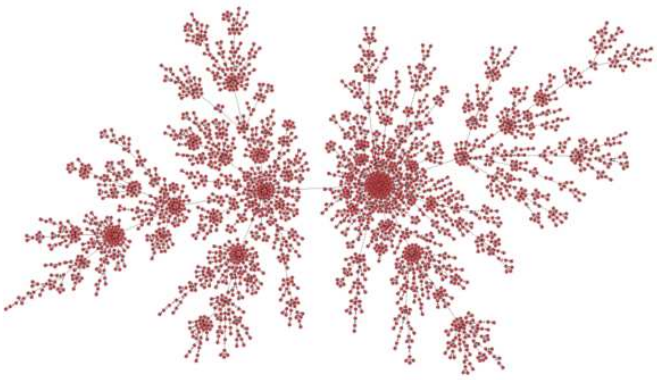
Brief outline of this lecture

- Generic graph search scheme, the BFS and DFS searches.
- Connectivity, components, Menger's theorem, 2-connected graphs.
- Minimum spanning tree problem (Jarník / Prim).
- Connectivity in directed graphs, strong components.

2.0 Exploring a Graph

How can one read a graph?

- Being humans, we look at a picture. . . □
- Being a computer, then what?



□

Algorithms need to employ local-search routines on huge input graphs. □

- We are going to present a general scheme of *searching through a graph*.

2.1 General Graph Search Scheme

This meta-algorithm works with the following data states and structures: □

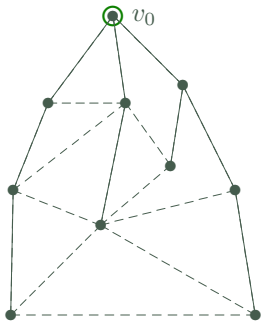
- **A vertex**: having one of the states ...
  - *initial* – assigned at the beginning,
  - *discovered* – after being found along an edge (stored for further processing),
  - *finished* – after checking and/or processing all the incident edges. □
  - (Can also be "post-processed", after finishing all of its descendants.) □
- **An edge**: having one of the states ...
  - *initial* – assigned at the beginning,
  - *processed* – after it has been processed at one of its endvertices. □
- **Stack (depository)**: is a supplementary data structure (a set) which
  - keeps all the discovered vertices, until they have been finished, and
  - stores an access edge for every discovery of a vertex (can be multiple). □

Graph search has many variants mostly given by the way vertices are picked from the stack. For greater generality, the scheme records vertices together with their access edges.

Spec. programming tasks can be performed at each vertex or edge of G while processing them.

Algorithm 2.1. ***Searching through a connected graph*** G.
This algorithm visits and processes every vertex and edge of a connected graph G.

```
state[ all vertices and edges of G ]  ← initial;
stack  U  ← {(∅, v0)},  for any starting vertex v0 of G;
search tree  T  ← ∅;
while (U ≠ ∅) {
    choose (e,v) ∈ U;
    U ← U \ {(e,v)};
    if (e ≠ ∅)  PROCESS(e; v);
    if (state[v] ≠ finished) {
        foreach (edge f incident with v) {
            w ← the opposite vertex of "f = vw";
            if (state[w] ≠ finished) {
                state[w] ← discovered;
                U ← U ∪ {(f,w)};
            }
        }
        PROCESS(v; e);
        state[v] ← finished;
        T ← T ∪ {e,v};
    }
}
G is finished;
```



* Procházení souvislým grafem *
state = stav, stack / depository = úschovna

Remarks

- Algorithm 2.1 is only a general scheme, and not a complete implementation. □
- In particular, implementing the stack `U` and the procedures `PROCESS()` would likely require additional ("hidden") variables and/or data structures. □
- Algorithm 2.1 altogether performs $\mathcal{O}(|E(G)|)$ (linearly many) elementary operations, calls to '`PROCESS()`' and updates of the stack `U`. □
- Given a particular implementation of `U` and `PROCESS()`, the general scheme can often be simplified (such as, by avoiding duplicates in U). □
- There are no mupliple calls to '`PROCESS(a)`' for any vertex or edge a of G.
  - True for vertices $a = v$ by '`state[v]` $\leftarrow$ *finished*',
  - and for edges $a = f$ since one end v of f gets finished together with adding of f into `U`, and then we have:

```
if (state[w] ≠ finished) { ... U ← U ∪ {(f,w)}; }
```

Correctness of the search scheme

Theorem 2.2. *Let G be a connected graph and $v_0 \in V(G)$ an arbitrary vertex. Then Algorithm 2.1 on G, when started from v_0, processes all the vertices and edges of G.* □

Proof.

1. Every element of G (vertex or edge) that gets into U will event. be processed:

```
choose (e,v) ∈ U;
if (e ≠ ∅)  PROCESS(e);
if (state[v] ≠ finished) { ...   PROCESS(v); ...} □
```

2. If an edge $f = vw \in E(G)$ gets into U, then both ends v, w will be processed:

```
if (state[w] ≠ finished) { ... U ← U ∪ {(f,w)}; }
```

   Hence v already is processed at this moment and w will be processed by 1. □

3. Assume an edge $f \in E(G)$ that never gets into U. By connectivity of G, one may choose f closest possible to v_0, and so some $h \in E(G)$ sharing a vertex v with f gets in; $(h, v) \in U$. However, then

```
foreach (edge f incident with v) { ... U ← U ∪ {(f,w)}; }
```

   a contradiction to the assumption that f never gets into U. □

Correctness..., II, the Search tree

Various applications of the graph search scheme, moreover, use the outcome T and refer to it as to the *search tree* of G. To be math. precise, we have to justify this fact:

Proposition 2.3. *The subgraph $T \subseteq G$ computed by Algorithm 2.1 is a tree spanning all the vertices of G. (See also further Def. 2.12.)* □

Proof. Using induction on the number of iterations of the main cycle

```
while (U ≠ ∅) {  ... ...    T ← T ∪ {e,v}; } }
```

we prove that T is a tree spanning the set of already processed ("finished") vertices. □

- After the first iteration, $T = \{\emptyset, v_0\}$ which represents a single-node tree v_0. □
- Let, after iteration i, the induction claim hold. Iteration $i+1$ chooses $(e, v) \in U$, and if v is finished, then nothing new happens.

  ```
  if (state[v] ≠ finished) { ...   T ← T ∪ {e,v}; } □
  ```

  Hence v is not finished yet, and $T \leftarrow T \cup \{e, v\}$ means we are adding to T a new vertex v with an edge e to some existing vertex of T. Then $T \cup \{e, v\}$ is again connected and acyclic, hence a tree, too.

□

Variants and Applications of the graph search scheme:

```
    while (U nonempty) {
(*)     choose (e,v) ∈ U;
        ......
```

The many variants of graph search mostly differ by the way (*) vertices are chosen (picked up) from the stack... □

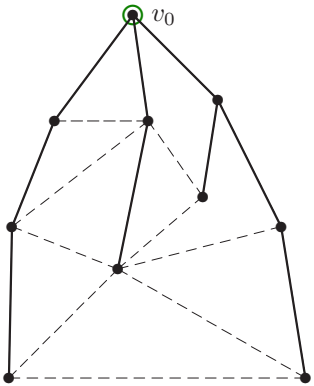
- *BFS* breadth-first search – the stack `U` is a "fifo" queue
  - perhaps the simplest variant, and trivially implementable. □
- *DFS* depth-first search – the stack `U` is a "lifo" stack
  - very important to store the vertices to `U` multiple times → last one "wins". □
- Searching a *disconnected graph* (in any way)
  - simply repeat Alg. 2.1 with arbitrary starting vertices in the parts... □
- Searching a *directed graph*
  - in Alg. 2.1, instead of `foreach (edge f incident with v)`, just use `foreach (arc f starting in v)` ...

disconnected graph = nesouvislý graf
DFS = prohl. do hloubky, BFS = prohl. do šířky

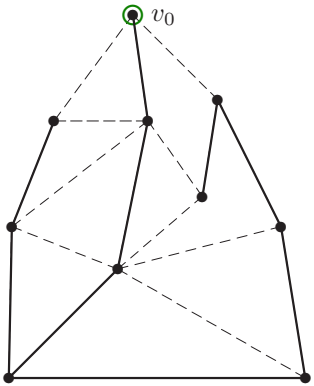
DFS and BFS search trees

Recalling the concept of a *search tree* T from Proposition 2.3, the following are particularly recognized variants of it:

T of BFS ("fifo")



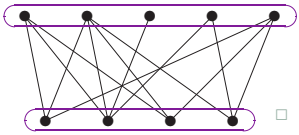
T of DFS ("lifo")



"fifo" = fronta, "lifo" = zásobník

Variants and Applications..., II

- Some slightly more involved variants (using additional implicit variables):
  - testing *bipartite graphs* – one simply runs BFS such that it "switches sides" with each `PROCESS(e)`, until none or some conflict is found. (Lecture 6)



  - *Jarník's* minimum spanning tree algorithm – the stack `U` always offers the vertex closest to any processed vertex. (Section 2.3) □
  - *Dijkstra's* shortest path algorithm – the stack `U` always offers the vertex closest to the starting position v_0. (Lecture 3)

2.2 Connectivity and Components

Recall: connected graphs are those graphs G such that, for any two vertices $u, v \in V(G)$, there is a path in G between the ends u and v. □

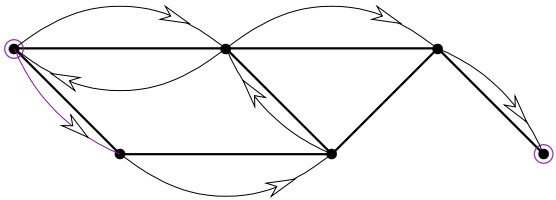
Definition: A *walk* W of length n in a graph G is a sequence of alternating vertices and edges

$$W = (v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n), \square$$

such that every its edge e_i has the ends v_{i-1}, v_i, for $i = 1, \ldots, n$.

We speak about a *walk from* $u = v_0$ *to* $v_n = v$, even if the graph G is undirected.

Such a sequence really is a "*walk*" through the graph, see for instance how an IP packet is routed through the internet – it often repeats vertices.



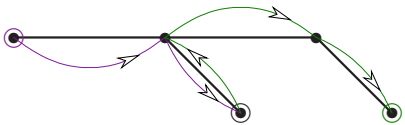
walk = procházka v grafu, alternating = střídající

Components of a graph

Lemma 2.4. *Let $\sim$ be a binary relation on the vertex set $V(G)$ of a graph G, such that $u \sim v$ if, and only if, there exists a walk in G starting in u and ending in v. Then $\sim$ is an equivalence relation.* □

Proof. The relation $\sim$ is

- reflexive since every vertex itself forms a walk of length 0,
- symmetric since any undirected walk can be easily "reversed", and □
- transitive since two walks can be concatenated at the common endvertex.



□

Definition 2.5. ***The connected components*** of a graph G are formed by the equivalence classes of the above relation $\sim$ (Lemma 2.4) on $V(G)$.□

More generally, by connected components we also mean the subgraphs induced on these vertex set classes of $\sim$.

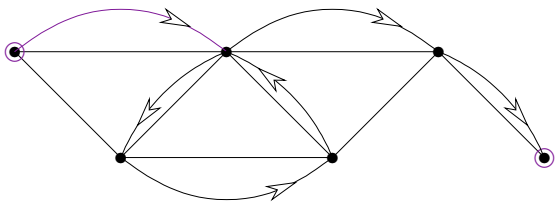
connected component = souvislá komponenta

Connected graphs revisited

Fact: By the definition, a *path* in a graph is a walk without repetition of vertices.

Lemma 2.6. *If there exists a walk between vertices u and v in a graph G, then there also exists a path from u to v in this G.* □

Proof. Among all the walks between u and v in G, we choose the (one of) shortest walk as $W \subseteq G$. It is then clear that if the same vertex is repeated on W, then W could be shortened further, a contradiction. Hence W is a path in G.



□

Corollary 2.7. *A graph G is connected if, and only if, G consists of at most one connected component.*

Higher Levels of Connectivity

Various netw. applications demand not only that a graph is connected, but that it stays connected even after failure of a small number of nodes (vertices) or links (edges).

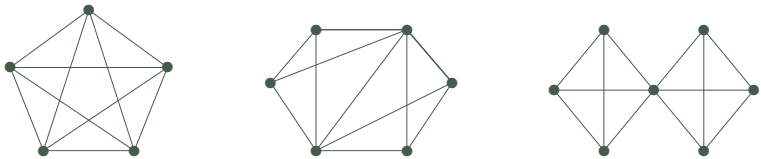
This can be studied in theory as "higher levels" of graph connectivity. □

Definition: A graph G is *edge-k-connected*, $k > 1$, if G stays connected even after removal of any subset of $\leq k - 1$ edges. □

Definition: A graph G is *vertex-k-connected*, $k > 1$, if G has more than k vertices and G stays connected even after removal of any subset of $\leq k - 1$ vertices.

Specially, the complete graph K_n is defined to be vertex-$(n-1)$-connected but not n-connected for $n > 1$, and K_1 is vertex-1-connected. □

Graphs that are "vertex / edge-1-connected" are just connected.



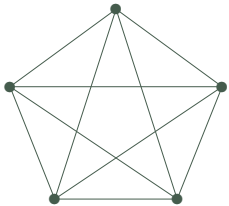
Sometimes we speak about a k-connected graph, and then we usually mean it to be vertex-k-connected. High vertex connectivity is a (much) stronger requirement than edge connectivity...

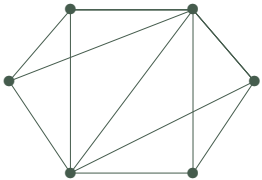
* Vyšší stupně souvislosti *
vertex- / edge-k-connected = vrcholově / hranově k-souvislý

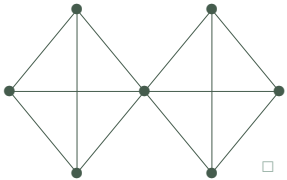
Menger's theorem and related

Theorem 2.8. *A graph G is edge-k-connected if, and only if, G has (at least) k edge-disjoint paths between any pair of vertices (the paths may share vertices).*

(Menger) *A graph G is vertex-k-connected if, and only if, G has (at least) k internally-disjoint paths between any pair of vertices (the paths may share only their ends).*







□

A few notes:

- For $k = 1$ the statement is trivial (note that this holds also for K_1). □
- For $k = 2$ the theorem (second part) reads:

  A graph G is vertex-2-conn. iff every two vertices of G lie on a common cycle. □
- The full proof, for any k, will be given in Lecture 4.

edge-disjoint paths = hranově disjunktní cesty
internally-disjoint paths = vnitřně disjunktní (tj. bez konců) cesty

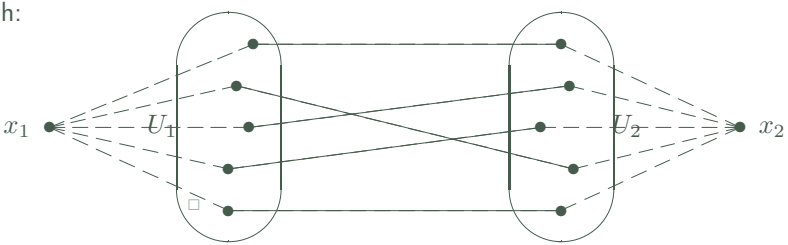
Menger's theorem, II

Recall: A graph G is vertex-k-connected if, and only if, there exist (at least) k internally-disjoint paths between any pair of vertices (the paths may share only their ends).

A straightforward reformulation of Theorem 2.8 is: □

Theorem 2.9. *Assume G is a k-connected graph, $k \geq 2$. Then, for every two disjoint sets $U_1, U_2 \subset V(G)$, $|U_1| = |U_2| = k$, there exist k pairwise disjoint paths from the terminals of U_1 to U_2.*

A sketch:

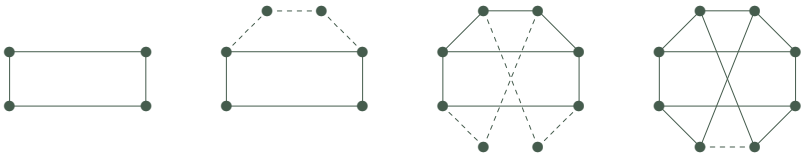


add two new vertices x_1, x_2, each adjacent to one of U_1, U_2, and take the k internally-disjoint paths from Theorem 2.8 between x_1 and x_2.

More on 2-connected graphs

To better illustrate the interesting properties of highly connected graphs, we give the following alternative characterizations of 2-connected ones. (A similar one exists for 3-conn. graphs.)

Theorem 2.10. *A simple graph is 2-connected if, and only if, it can be constructed from a cycle by "adding ears"; i.e. by iterating the operation which adds a new path (of arbitrary length, even an edge, but not a parallel edge) between two existing vertices of a graph.*



A sketch: find the first cycle, and "grow it" to the largest possible subgraph G' by adding ears. A closest vertex of the graph "outside" of G' has two connections to G'—one a direct edge and another via a path—making together a new ear. □

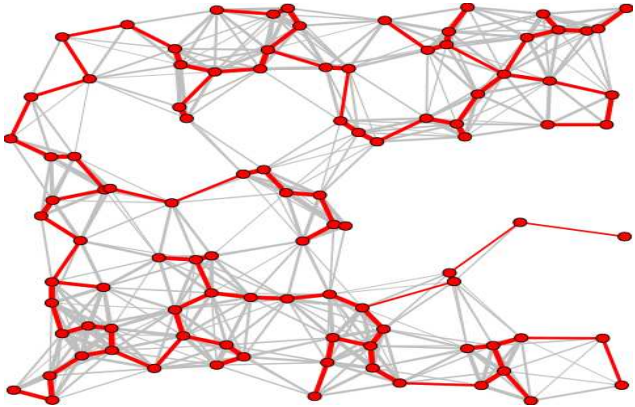
Theorem 2.11. *G is a 2-connected graph if, and only if, every two edges of G belong to a common cycle.* □

Proof. For two edges $e_i = u_i v_i$, $i = 1, 2$, apply Theorem 2.9 to $U_i = \{u_i, v_i\}$. □

2.3 Minimum Spanning Tree problem

Recall (Section 1.3); trees are "minimal connected graphs" on a given vertex set...

Definition 2.12. ***A spanning tree*** of a connected graph G is a subgraph $T \subseteq G$ such that T is a tree and $V(T) = V(G)$.



* *Problém minimální kostry* *
spanning tree = kostra grafu

The MST problem formulation

Definition: A *weighted graph* is a pair of a graph G together with a weighting w of the edges by real numbers $w : E(G) \to \boldsymbol{R}$ (edge lengths in this case).
A *positively weighted graph* (G, w) is such that $w(e) > 0$ for all edges e. □

Looking for a minimal interconnection in a graph (wrt. weights w) hence reads:

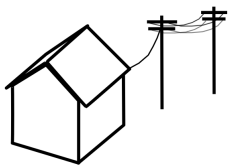
Problem 2.13. ***The minimum spanning tree (MST) problem***
Given a weighted connected graph (G, w) with (positive) edge weights w; the problem is to find a spanning tree T in G that minimizes the total weight. □*Formally*

$$MST := \min_{\text{sp. tree } T \subset G} \left(\sum_{e \in E(T)} w(e) \right).$$

weighted graph = vážený či ohodnocený graf

A bit of history – it all started in Brno!

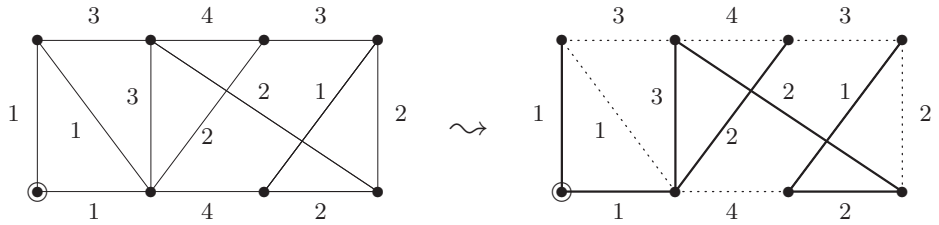
- Long time ago, in the past century, the task was to connect towns and villages in South Moravia by electric power lines of minimum total length...



- Does it sound familiar? Yes—we have to find a *minimum spanning tree* in the graph of all possible connections between these settlements. □
- Did it really happen? Yes, indeed, and an algorithmic solution was found in 1926 by Otakar Borůvka, a Brno mathematician. □

  A simpler and precise algorithm of Vojtěch Jarník then followed in 1930. □
- Unfortunately, both there works were published only in Czech language, and so most of the world knows only the work of Kruskal, giving another algorithm in 1956, and of Prim, who later rediscovered Jarník's algorithm.

Jarník's MST Algorithm



Algorithm 2.14. ***Jarník's (later known as Prim's) algorithm for MST.***

Given is a weighted graph (G, w) with edge weights w (which are commonly assumed positive, but this is not necessary). □

- Run Algorithm 2.1 with an implementation of the step

  `choose (e,v)` $\in$ `U;`

  such that $(e, v) \in U$ minimizes $w(e)$. □
- U is thus implemented as a *min-heap with the key $w(e)$* for $(e, v) \in U$. □
- The resulting search tree T is a minimum spanning tree of G.

min-heap = halda s minimálním klíčem

Jarník's MST algorithm, proof

Algorithm 2.14:

- Run Algorithm 2.1 with an implementation of the step

  `choose (e,v) ∈ U;`

  such that $(e, v) \in U$ minimizes $w(e)$.
- U is thus implemented as a *min-heap with the key* $w(e)$ for $(e, v) \in U$.
- The resulting search tree T is a minimum spanning tree of G.

Proof of Algorithm 2.14. We have the following setup;

- let T be the spanning tree (Proposition 2.3) computed by the algorithm, □
- $T_1 = \{v_0\}, T_2, \ldots, T_n = T$ be the seq. of part. solutions after each iter., and
- $T^{opt} \neq T$ be some optimal MST solution, maximizing index k s.t. $T_k \subseteq T^{opt}$.□

Then $k < n$. Let $\{e\} = E(T_{k+1}) \setminus E(T_k)$ be the first "wrong" edge chosen by the alg.

By Corollary 1.12, $T^{opt} + e$ (by adding the edge e to T^{opt}) contains exactly one cycle $C \subseteq T^{opt} + e$. Then C has at least two edges leaving $V(T_k)$; e and some f. □

Now $T^{alt} = (T^{opt} + e) \setminus f$ is a spanning tree again, and since $w(f) \geq w(e)$ (by the algorithm, $w(e)$ was minimized when choosing $(e, v) \in U$), also T^{alt} is an optimal MST solution. This contradicts our choice of max. k. □

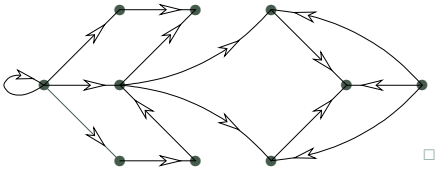
2.4 Connectivity in Directed Graphs

For start, we proceed analogously to the undirected case...

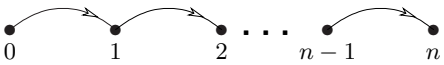
Definition: A *directed walk* W of length n in a digraph D is a sequence of alternating vertices and oriented edges

$$W = (v_0, e_1, v_1, e_2, v_2, \ldots, e_n, v_n),$$

such that every edge (arc) e_i in W is of the form $e_i = (v_{i-1}, v_i)$.



Lemma 2.15. *If there exists a directed walk from u to v in a digraph D, then there also exists a directed path from u to v in this D.*



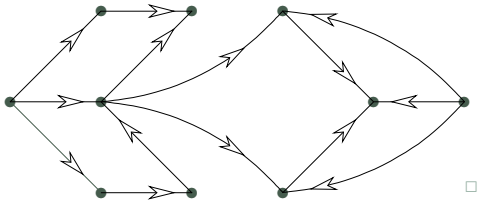
directed walk = orientovaná procházka

Views of directed connectivity

- The *weak connectivity* view does not care about directions of arcs.
  Not so usable or interesting... □

- The *reachability* view is as follows:

Definition: A digraph D is *out-connected* if there exists a vertex $v \in V(D)$ such that for every $x \in V(D)$ there is a directed walk from v to x (all vertices reachable from v).



□

No, this graph is not out-connected – see the right-most vertex... □

- The *strong* (bidirectional) view builds on the following:

weak connectivity = slabá souvislost
reachability = dosažitelnost

Strong connectivity

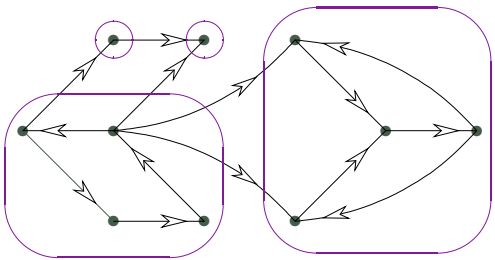
Lemma 2.16. *Let $\approx$ be a binary relation on the vertex set $V(D)$ of a directed graph D such that $u \approx v$ if, and only if, there exist two directed walks in D – one starting in u and ending in v and the other starting in v and ending in u.*

Then $\approx$ is an equivalence relation. □

Definition 2.17. ***The strong components*** of a digraph D are formed by the equivalence classes of the above relation $\approx$ (Lemma 2.16) on $V(D)$.□

A digraph is *strongly connected* if it has at most one strong component.

See the four strong components in this illustration picture:



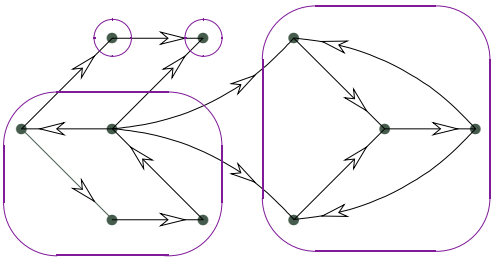
strong components = silné komponenty
strongly connected = silně souvislý

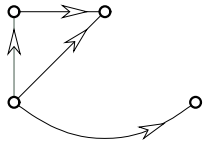
Condensation of a digraph

Definition: A digraph Z whose vertices are the strong components of D, and the arcs of Z exist exactly between those pairs of distinct components of D such that D contains an arc between them, is called a *condensation* of D. □

Definition: A digraph is *acyclic* (a "DAG") if it does not contain a directed cycle.□

Proposition 2.18. *The condensation of any digraph is an acyclic digraph.*



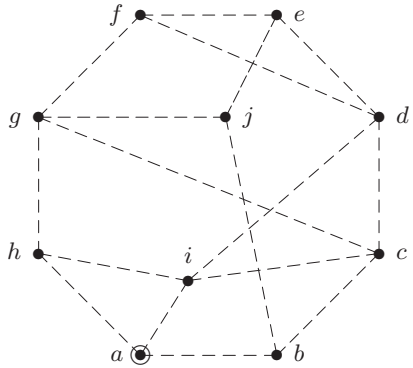


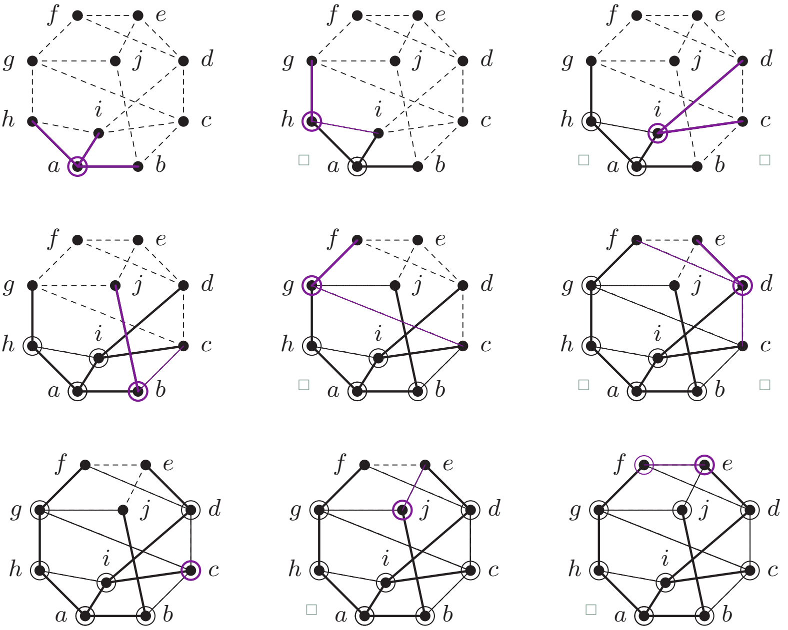
acyclic = acyklický, condensation = kondenzace

2.5 Appendix: DFS and BFS examples

Following on Algorithm 2.1...

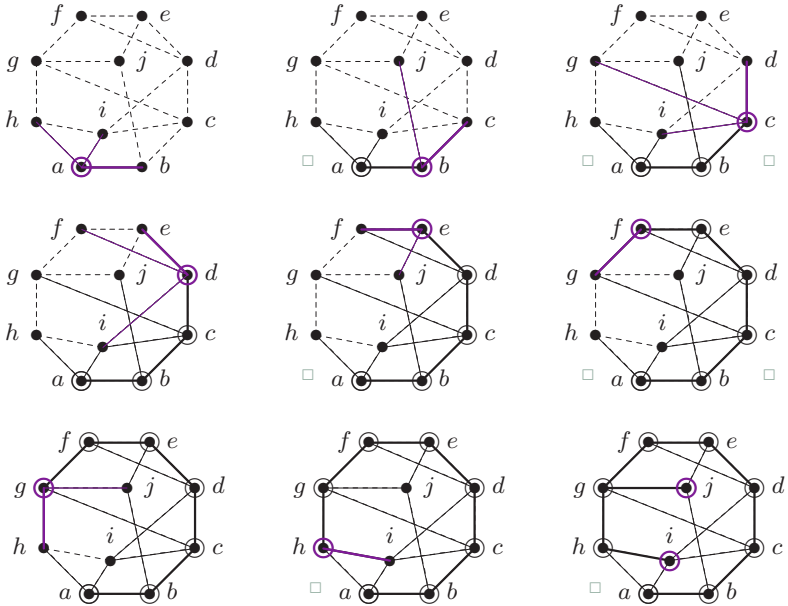
Example 2.15. *An example of a breadth-first search run from the vertex a.*




f
e
g
j
d
h
i
c
a
b

□

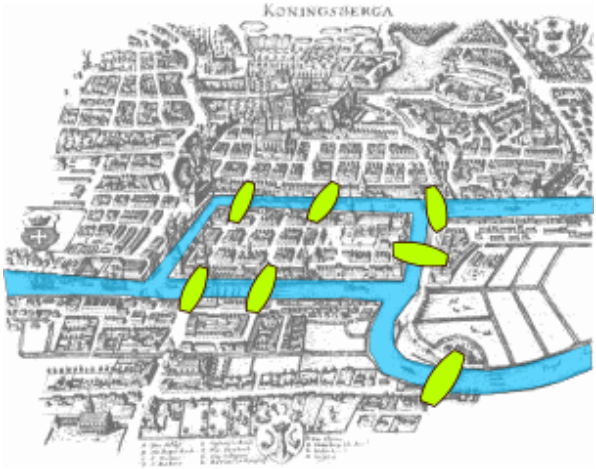
Example 2.16. *An example of a depth-first search run from the vertex a.*



□

2.6 Appendix: Eulerian Trail

Perhaps the oldest recorded result of graph theory comes from famous Leonhard Euler — it is the "*7 bridges of Königsberg*" (Královec, now Kaliningrad) problem.

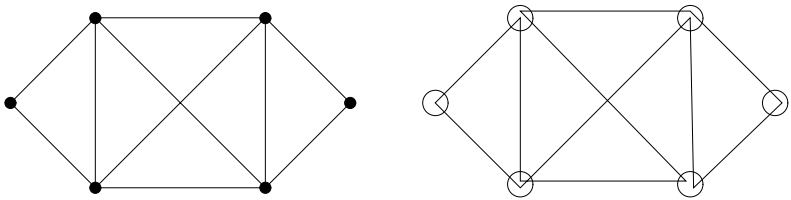


So what was the problem? The city majors that time wanted to walk through the city while crossing each of the 7 bridges exactly once...

** Eulerovský tah ("kreslení jedním tahem") **

This problem led Euler to introduce the following:

Definition: A *trail* in a (multi)graph is a walk which does not repeat edges.
A *closed trail (tour)* is such a trail that ends in the same vertex it started with. The opposite is an *open trail*.



And the (perhaps) oldest graph theory result by Euler reads: □

Theorem 2.17. (Euler) *A (multi)graph G consists of one closed trail if, and only if, G is connected and all the vertex degrees in G are even.* □

Corollary 2.18. *A (multi)graph G consists of one open trail if, and only if, G is connected and all the vertex degrees in G but two are even.*

Analogous results hold true also for digraphs (the proofs are the same)...

trail = tah (uzavřený / otevřený)