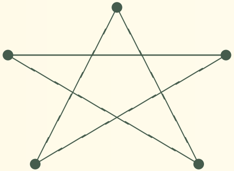
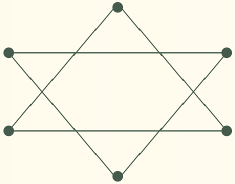
2 Connectivity in Graphs

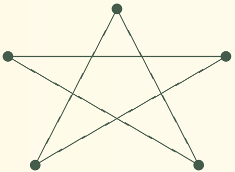
Graphs are often used to model various interconnecting networks, such as transport, pipe, or computer networks. In such models, one usually needs or wants to get from any place to any other place..., which naturally leads to the study of their "connectivity".

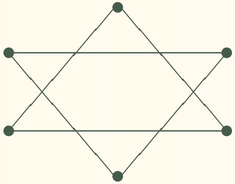




2 Connectivity in Graphs

Graphs are often used to model various interconnecting networks, such as transport, pipe, or computer networks. In such models, one usually needs or wants to get from any place to any other place. . . , which naturally leads to the study of their "connectivity".





Brief outline of this lecture

- Walks in a graph, the definition, connected components.
- Exploring a graph, search algorithms – BFS and DFS.
- Higher levels of connectivity, 2-connected graphs, Menger's theorem.
- Connectivity in directed graphs, strong components.
- Eulerian tours and trails, "Seven Bridges" and the even degree cond.

2.1 Graph Connectivity and Components

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

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

such that every its edge e_i has the ends v_{i-1}, v_i.

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.

2.1 Graph Connectivity and Components

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

$$(v_0, e_1, v_1, e_2, v_2, \dots, e_n, v_n)\,,$$

such that every its edge e_i has the ends v_{i-1}, v_i.

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.

Lemma 2.1. *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.*

2.1 Graph Connectivity and Components

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

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

such that every its edge e_i has the ends v_{i-1}, v_i.

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.

Lemma 2.1. *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. It is also symmetric since any undirected walk can be easily "reversed", and transitive since two walks can be concatenated at the common endvertex. $\square$

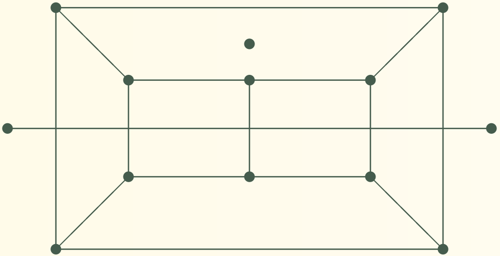
Components of a graph

Definition: The equivalence classes of the above relation $\sim$ (Lemma 2.1) on $V(G)$ are called the *connected components* of the graph G.

Components of a graph

Definition: The equivalence classes of the above relation $\sim$ (Lemma 2.1) on $V(G)$ are called the *connected components* of the graph G.

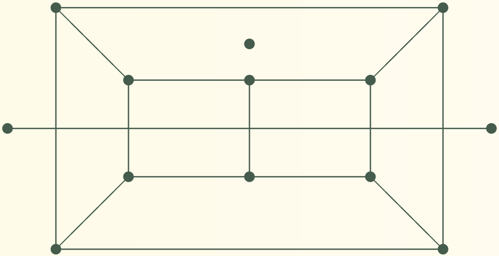
More generally, by connected components we also mean the subgraphs induced on these vertex set classes of $\sim$.



Components of a graph

Definition: The equivalence classes of the above relation $\sim$ (Lemma 2.1) on $V(G)$ are called the *connected components* of the graph G.

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



Can you see all the three components in this picture?

Recall a *path* in a graph — it is a walk without repetition of vertices.

Theorem 2.2. *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.*

Recall a *path* in a graph — it is a walk without repetition of vertices.

Theorem 2.2. *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. Let $(u = v_0, e_1, v_1, \ldots, e_n, v_n = v)$ be a walk of length n between u and v in G. We start building a new walk W from $w_0 = u$ which will actually be a path:

– Assume we have built a starting fragment $(w_0, e_1, w_1, \ldots, w_i) = W$ (inductively from $i = 0$, i.e. w_0) where $w_i = v_j$ for some $j \in \{0, 1, \ldots, n\}$.

Recall a *path* in a graph — it is a walk without repetition of vertices.

Theorem 2.2. *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. Let $(u = v_0, e_1, v_1, \ldots, e_n, v_n = v)$ be a walk of length n between u and v in G. We start building a new walk W from $w_0 = u$ which will actually be a path:

- Assume we have built a starting fragment $(w_0, e_1, w_1, \ldots, w_i) = W$ (inductively from $i = 0$, i.e. w_0) where $w_i = v_j$ for some $j \in \{0, 1, \ldots, n\}$.
- Find maximum index $k \geq j$ such that $v_k = v_j = w_i$ (repeated), and append W with $(\ \ldots \ , w_i = v_j = v_k,\ e_{k+1},\ w_{i+1} = v_{k+1}$.

Recall a *path* in a graph — it is a walk without repetition of vertices.

Theorem 2.2. *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. Let $(u = v_0, e_1, v_1, \ldots, e_n, v_n = v)$ be a walk of length n between u and v in G. We start building a new walk W from $w_0 = u$ which will actually be a path:

– Assume we have built a starting fragment $(w_0, e_1, w_1, \ldots, w_i) = W$ (inductively from $i = 0$, i.e. w_0) where $w_i = v_j$ for some $j \in \{0, 1, \ldots, n\}$.

– Find maximum index $k \geq j$ such that $v_k = v_j = w_i$ (repeated), and append W with $(\ \ldots \ , w_i = v_j = v_k, \ e_{k+1}, \ w_{i+1} = v_{k+1}$.

– It remains to show, by easy means of a contradiction, that the new vertex $w_{i+1} = v_{k+1}$ have not occured in W yet.

– The procedure stops whenever $w_i = v$.

Recall a *path* in a graph — it is a walk without repetition of vertices.

Theorem 2.2. *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. Let $(u = v_0, e_1, v_1, \ldots, e_n, v_n = v)$ be a walk of length n between u and v in G. We start building a new walk W from $w_0 = u$ which will actually be a path:

- Assume we have built a starting fragment $(w_0, e_1, w_1, \ldots, w_i) = W$ (inductively from $i = 0$, i.e. w_0) where $w_i = v_j$ for some $j \in \{0, 1, \ldots, n\}$.
- Find maximum index $k \geq j$ such that $v_k = v_j = w_i$ (repeated), and append W with $(\ \ldots \ , w_i = v_j = v_k, \ e_{k+1}, \ w_{i+1} = v_{k+1}$.
- It remains to show, by easy means of a contradiction, that the new vertex $w_{i+1} = v_{k+1}$ have not occured in W yet.
- The procedure stops whenever $w_i = v$.

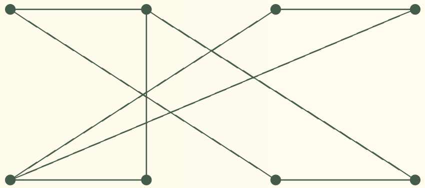
□

Proof; a shorter, but nonconstructive alternative.

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

Basic connectivity

Definition 2.3. **Graph G is connected** if G consists of at most one connected component. By Theorem 2.2, this means if every two vertices of G are connected by a path.



2.2 Exploring (Searching) a Graph

For an illustration, we present a very general scheme of searching through a graph. 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 we have find it along an edge,
  - *finished* – assigned after exploring all incident edges.
  - (Can also be post-processed, after finishing all its descendants.)
- **An edge**: having one of the states ...
  - *initial* – assigned at the beginning,
  - *processed* – whenever it has been processed at one of its endvertices.

2.2 Exploring (Searching) a Graph

For an illustration, we present a very general scheme of searching through a graph. 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 we have find it along an edge,
  - *finished* – assigned after exploring all incident edges.
  - (Can also be post-processed, after finishing all its descendants.)
- **An edge**: having one of the states ...
  - *initial* – assigned at the beginning,
  - *processed* – whenever 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.

Graph search has several variants mostly defined by the way vertices are picked from the depository. For greater generality, we actually record vertices together with their access edges. Specific programming tasks can be (are) performed at each vertex or edge of G while processing them.

Algorithm 2.4. Searching through a connected component G
This algorithm visits and processes every vertex and edge of a connected graph G.

```
input < graph G;
state(all vertices and edges of G) < initial;
stack U = {(∅, v_0)}, for any vertex v_0 of G;
search tree T = ∅;
while (U nonempty) {
    choose (e,v) ∈ U;
    U = U \ {(e,v)};
    if (e ≠ ∅) PROCESS(e);
    if (state(v) ≠ finished) {
        foreach (f incident with v) {
            w = the opposite vertex of f = vw;
            if (state(w) ≠ finished)
                U = U ∪ {(f,w)};
        }
        PROCESS(v);
        state(v) = finished;
        T = T ∪ {e,v};
    }
}
G is finished;
```

Implementation variants of graph searching

- *DFS* depth-first search – the depository `U` is a "LIFO" stack.

Implementation variants of graph searching

- *DFS* depth-first search – the depository `U` is a "LIFO" stack.
- *BFS* breadth-first search – the depository `U` is a "FIFO" queue.

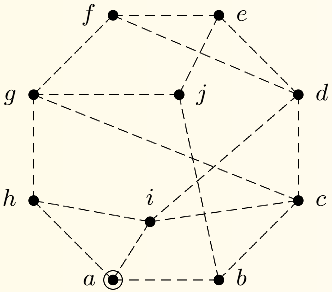
Implementation variants of graph searching

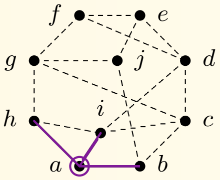
- *DFS* depth-first search – the depository `U` is a “LIFO” stack.
- *BFS* breadth-first search – the depository `U` is a “FIFO” queue.
- *Dijkstra’s* shortest path algorithm – the depository `U` always picks the vertex closest to the starting position v_0
  (similar, but more general, to BFS, see the next lecture).

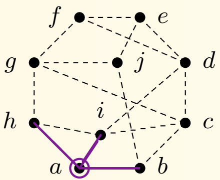
Implementation variants of graph searching

- *DFS* depth-first search – the depository U is a "LIFO" stack.
- *BFS* breadth-first search – the depository U is a "FIFO" queue.
- *Dijkstra's* shortest path algorithm – the depository U always picks the vertex closest to the starting position v_0
(similar, but more general, to BFS, see the next lecture).

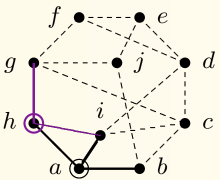
Example 2.15. *An example of a breadth-first search run from a vertex a.*



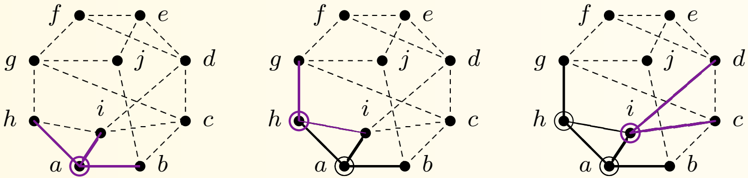

f
e
g
j
d
i
h
c
a
b

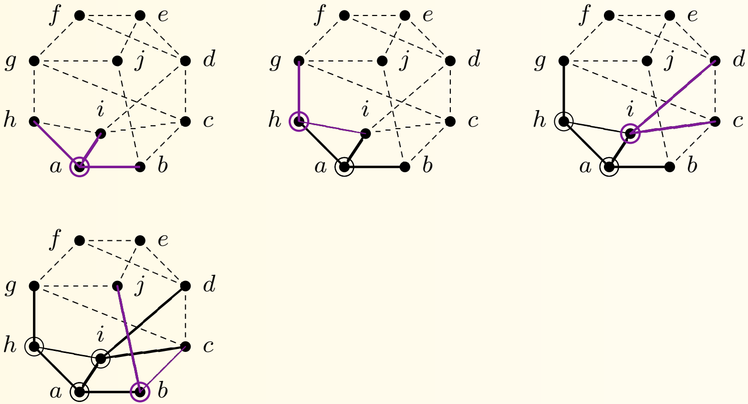


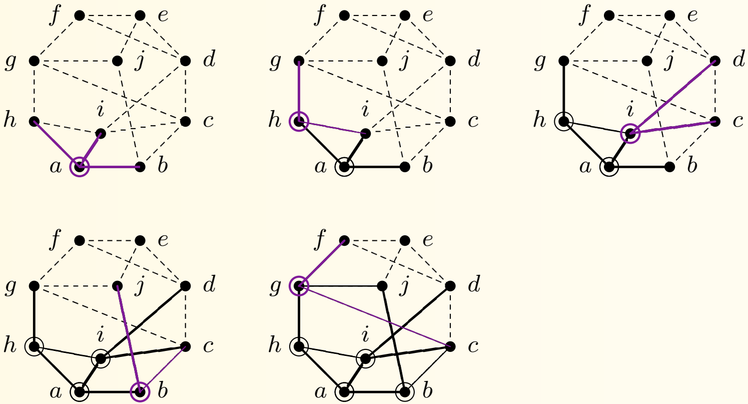
f
e
g
j
d
h
i
c
a
b

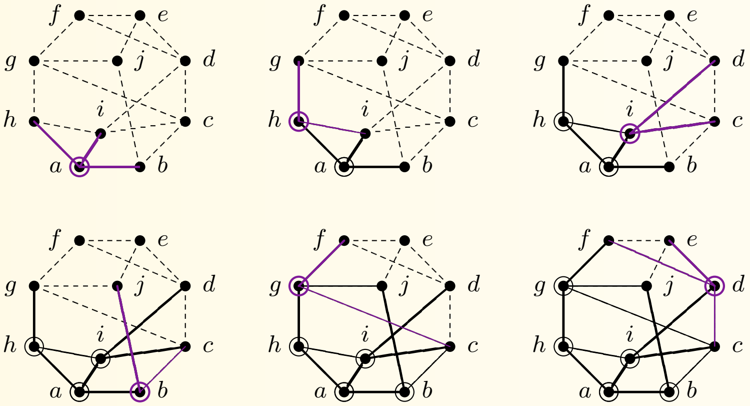


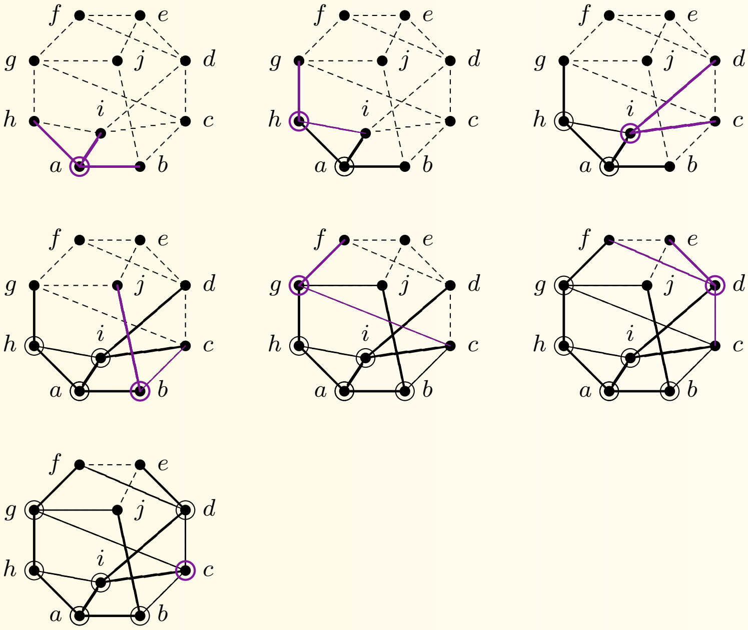
f
e
g
j
d
h
i
c
a
b

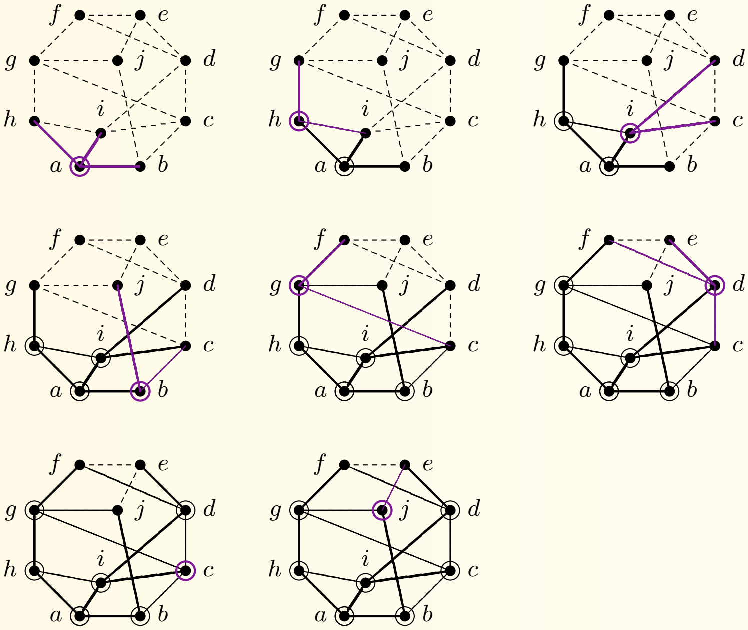

f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b

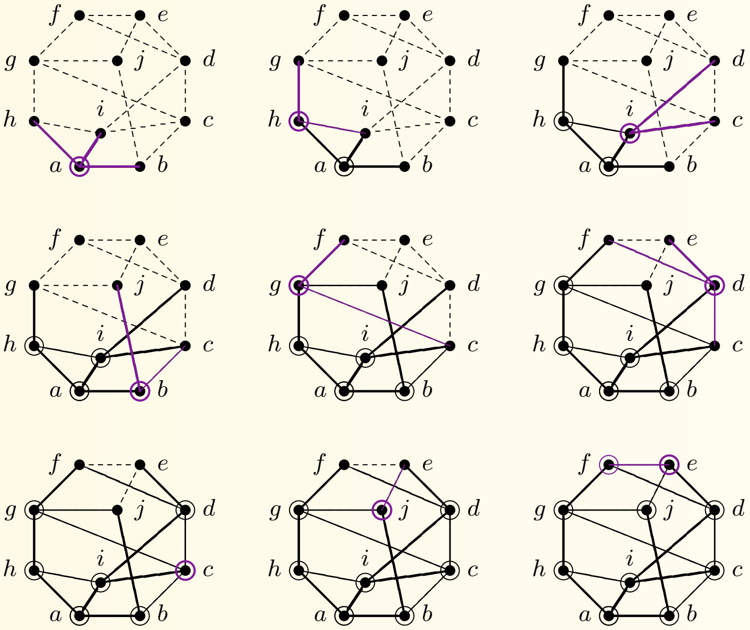

f
e
g
j
d
h
i
c
a
b


f e
g j d
h i c
a b
f e
g j d
h i c
a b
f e
g j d
h i c
a b
f e
g j d
h i c
a b
f e
g j d
h i c
a b


f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b
f
e
g
j
d
h
i
c
a
b

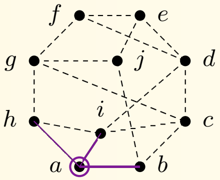

f
e
g
j
d
h
i
c
a
b


f
e
g
j
d
h
i
c
a
b

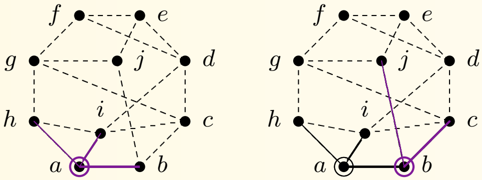

f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b
f e g j d h i c a b

□

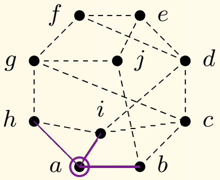
Example 2.16. *An example of a depth-first search run from a vertex a.*



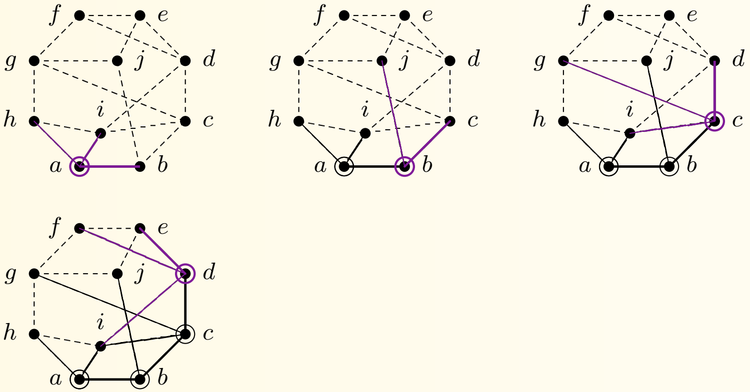
Example 2.16. *An example of a* *depth-first* *search run from a vertex* a.



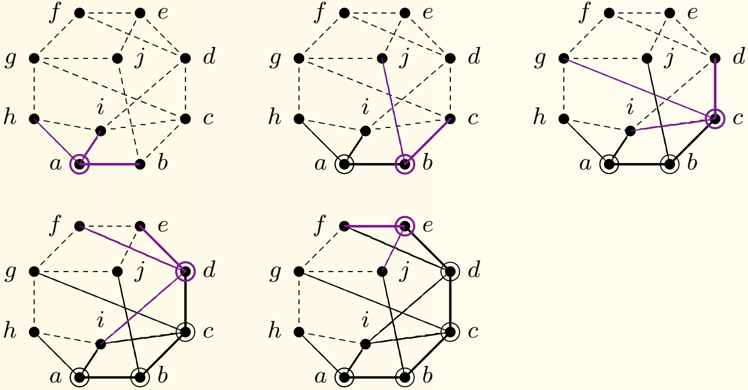
Example 2.16. *An example of a depth-first search run from a vertex a.*



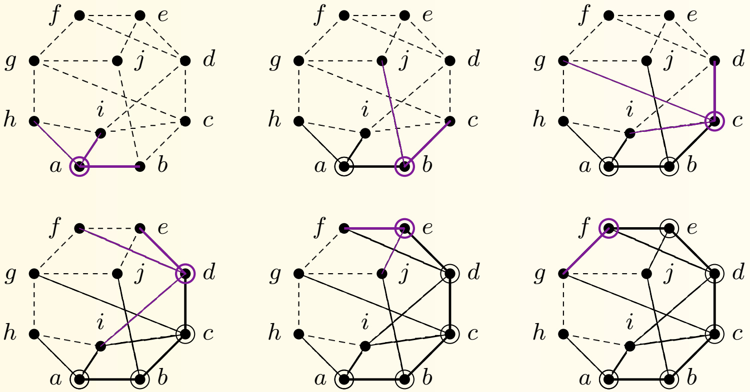
Example 2.16. *An example of a depth-first search run from a vertex a.*



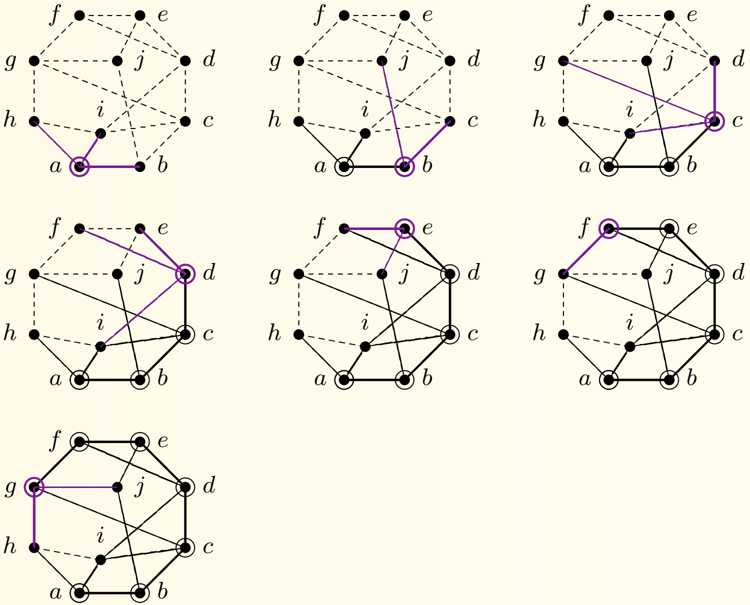
Example 2.16. *An example of a depth-first search run from a vertex a.*



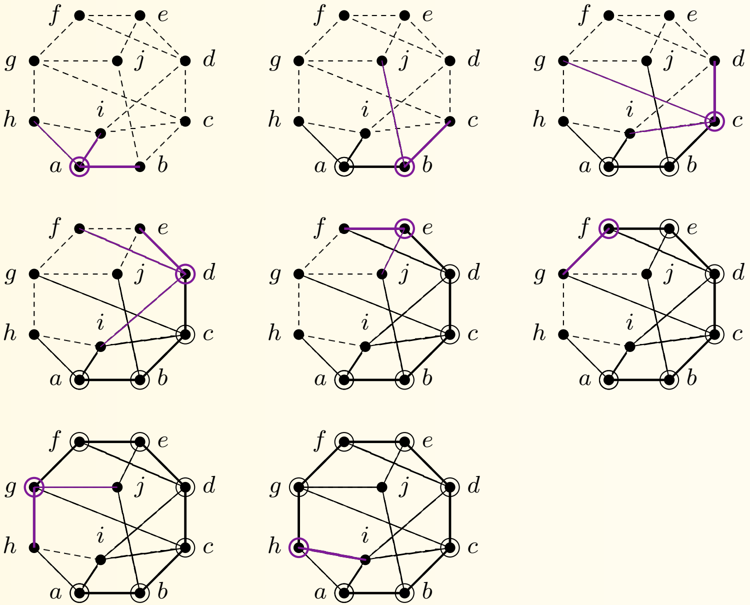
Example 2.16. *An example of a* *depth-first* *search run from a vertex* a.



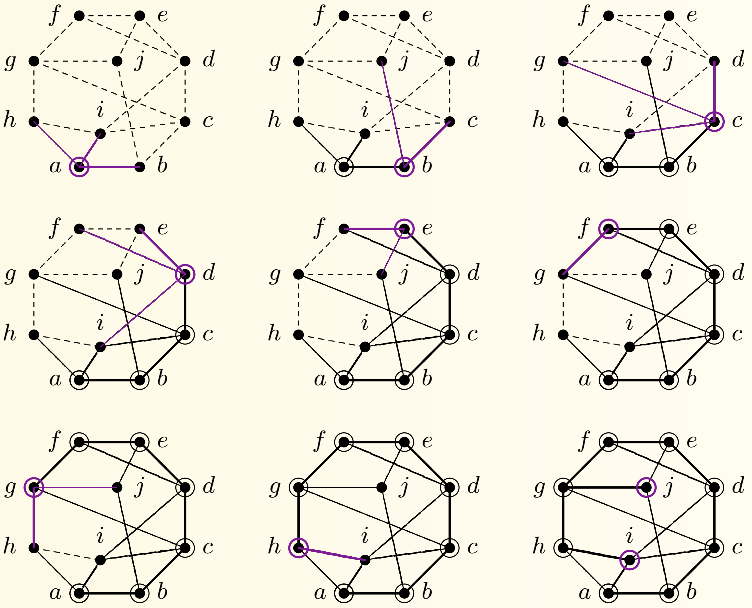
Example 2.16. *An example of a depth-first search run from a vertex a.*



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



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



□

2.3 Higher Leves of Connectivity

Various networking applications are often demanding not only that a graph is connected, but that it will stay connected even after failure of some small number of nodes (vertices) or links (edges).

This can be studied in theory as “higher levels” of graph connectivity.

2.3 Higher Leves of Connectivity

Various networking applications are often demanding not only that a graph is connected, but that it will stay connected even after failure of some 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.

2.3 Higher Leves of Connectivity

Various networking applications are often demanding not only that a graph is connected, but that it will stay connected even after failure of some 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 stays connected even after removal of any subset of $\leq k-1$ vertices.
Specially, the complete graph K_n is vertex-$(n-1)$-connected.

2.3 Higher Leves of Connectivity

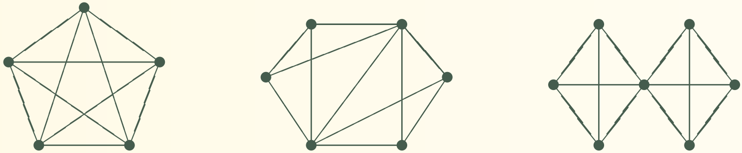
Various networking applications are often demanding not only that a graph is connected, but that it will stay connected even after failure of some small number of nodes (vertices) or links (edges).

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 stays connected even after removal of any subset of $\leq k-1$ vertices.
Specially, the complete graph K_n is vertex-$(n-1)$-connected.

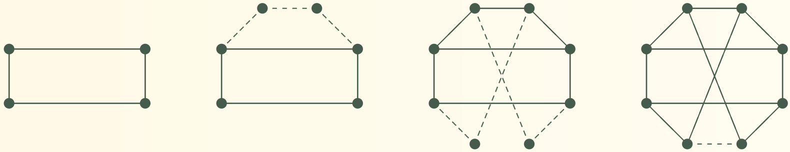
Graphs that are "vertex / edge-1-connected" are simply connected.



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...

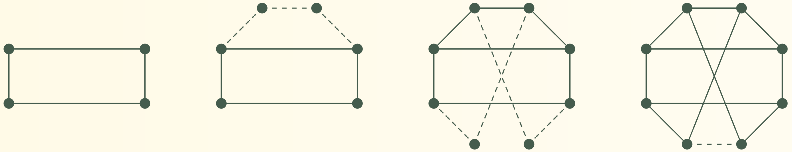
About 2-connected graphs

Theorem 2.5. *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.*



About 2-connected graphs

Theorem 2.5. *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.*

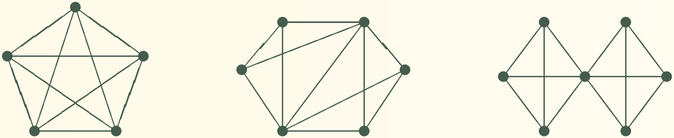


Theorem 2.6. *Assume* G *is a* 2*-connected graph. Then every two edges in* G *lie on a common cycle.*

Menger's theorem and related

Theorem 2.7. *A graph G is edge-k-connected if, and only if, there exist (at least) k edge-disjoint paths between any pair of vertices (the paths may share vertices).*

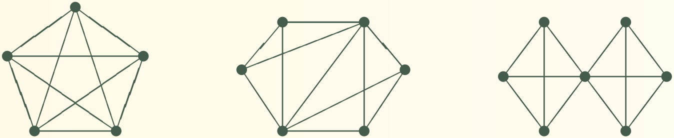
A graph G is vertex-k-connected if, and only if, there exist (at least) k internally disjoint paths between any pair of vertices (the paths may share only their ends).



Menger's theorem and related

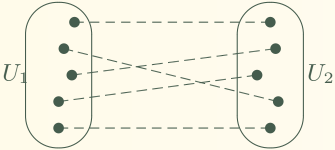
Theorem 2.7. *A graph G is edge-k-connected if, and only if, there exist (at least) k edge-disjoint paths between any pair of vertices (the paths may share vertices).*

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).



Some direct corollaries of the theorem are the following:

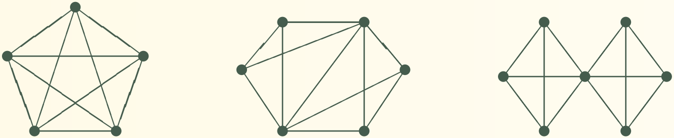
Theorem 2.8. *Assume G is a k-connected graph, $k \geq 2$. Then, for every two disjoint sets $U_1, U_2 \subset V(G)$, $|U_1| = |U_2| = k$, there exist k pairwise disjoint paths from the terminals of U_1 to U_2.*



Menger's theorem and related

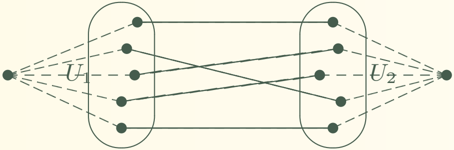
Theorem 2.7. *A graph G is edge-k-connected if, and only if, there exist (at least) k edge-disjoint paths between any pair of vertices (the paths may share vertices).*

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).



Some direct corollaries of the theorem are the following:

Theorem 2.8. *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.*



2.4 Connectivity in Directed Graphs

At the beginning we proceed analogically to the undirected case...

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

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

such that every edge e_i in it is $e_i = (v_{i-1}, v_i)$.

2.4 Connectivity in Directed Graphs

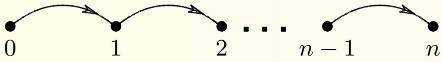
At the beginning we proceed analogically to the undirected case...

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

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

such that every edge e_i in it is $e_i = (v_{i-1}, v_i)$.

Theorem 2.9. *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.*



Views of directed connectivity

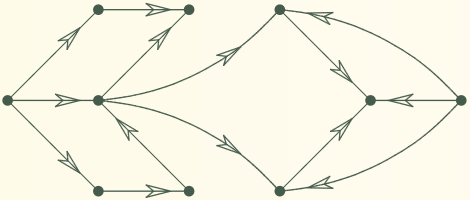
- The *weak connectivity* does not care about directions of arcs.
  Not so usable or interesting...

Views of directed connectivity

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

- A *reachability* view, 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).

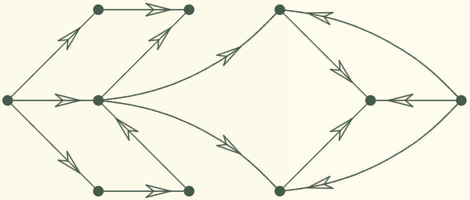


Views of directed connectivity

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

- A *reachability* view, 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).



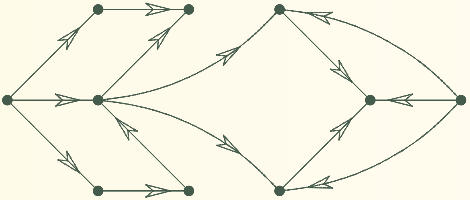
No, this graph is not out-connected – see the right-most vertex...

Views of directed connectivity

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

- A *reachability* view, 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...

- A *strong* (bidirectional) view, as follows:

Strong connectivity

Lemma 2.10. *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.

Strong connectivity

Lemma 2.10. *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 2.11. **The strong components** of a digraph D are formed by the equivalence classes of the above relation $\approx$ (Lemma 2.10) on $V(D)$.

Strong connectivity

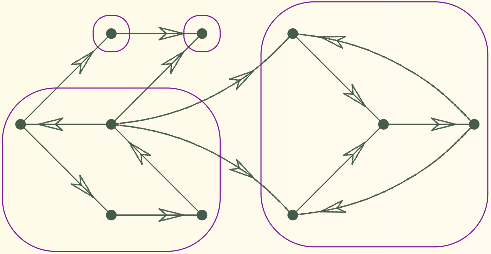
Lemma 2.10. *Let $\approx$ be a binary relation on the vertex set $V(D)$ of a directed graph D such that $u \approx v$ if, and only if, there exist two directed walks in D – one starting in u and ending in v and the other starting in v and ending in u.*

Then $\approx$ is an equivalence relation.

Definition 2.11. **The strong components** of a digraph D are formed by the equivalence classes of the above relation $\approx$ (Lemma 2.10) 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:



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.

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.

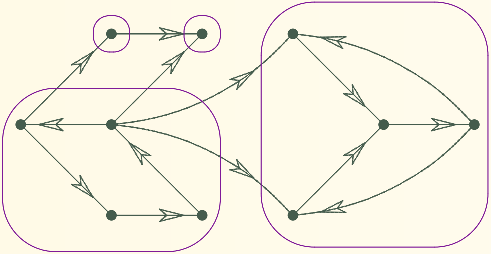
Definition: A digraph is *acyclic* (a "DAG") if it does not contain a directed cycle.

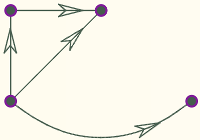
Condensation of a digraph

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.

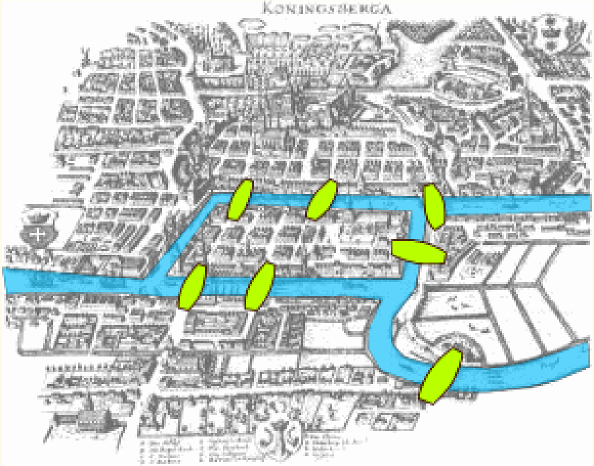
Proposition 2.12. *The condensation of any digraph is an acyclic digraph.*





2.5 Eulerian Trails

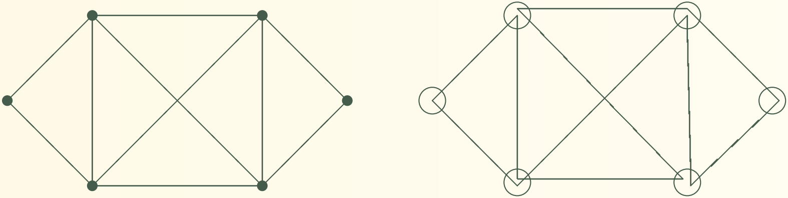
Perhaps the oldest recorded result of graph theory comes from famous Leonardo Euler — it is the "7 *bridges of Königsberg*" (Královec, now Kaliningrad) problem.



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...

This problem led Euler to introduce the following:

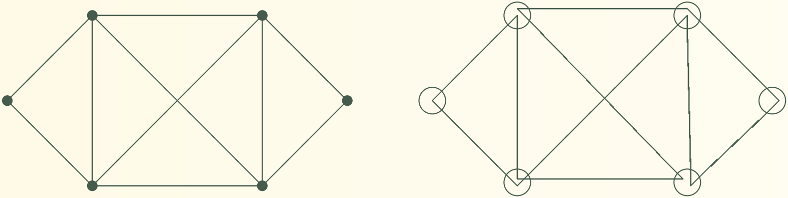
Definition: A *trail* in a graph is a walk which does not repeat edges.
A *closed trail (tour)* is such a trail that ends in the same vertex it started with. The opposite is an *open trail*.



And the oldest graph theory result by Euler reads:

This problem led Euler to introduce the following:

Definition: A *trail* in a 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*.

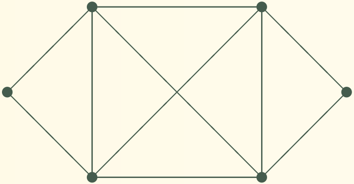


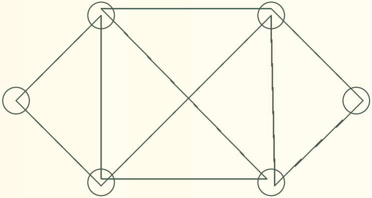
And the oldest graph theory result by Euler reads:

Theorem 2.13. *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.*

This problem led Euler to introduce the following:

Definition: A *trail* in a 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 oldest graph theory result by Euler reads:

Theorem 2.13. *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 2.14. *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). . .