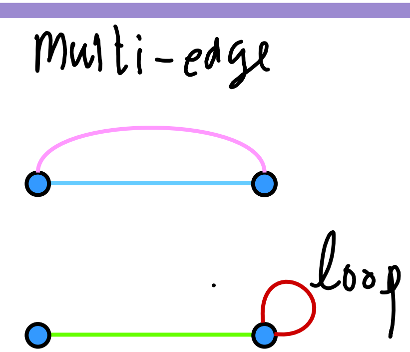
Graph Overview (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Simple Graph ⟷ Multi-Graph



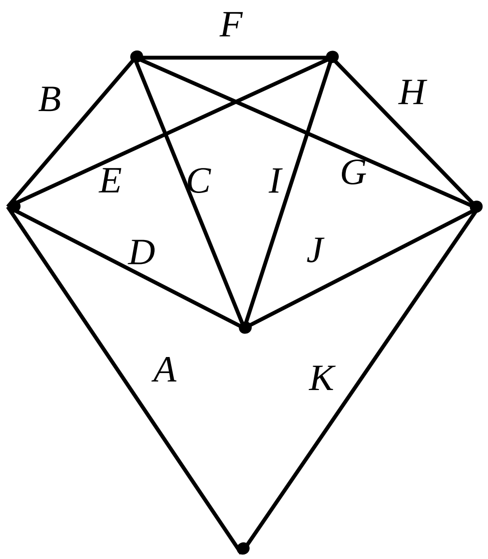
A simple graph is an undirected graph
without multiple edges or **loops**.

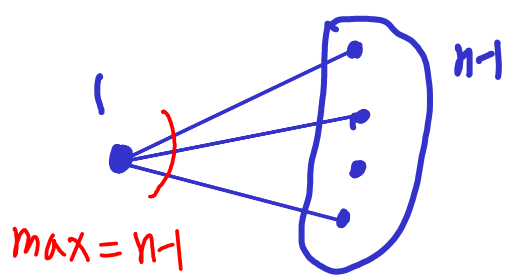
the edges form a set (rather than a multiset)
each edge is an unordered pair of distinct vertices.

can define a simple graph to be a **set V** of <u>vertices</u>
together with a **set E** of <u>edges</u>,

E are <u>2-element</u> <u>subsets</u> of **V**

with **n** <u>vertices</u>,
the **degree** of every <u>vertex</u> is <u>at most</u> **n − 1**



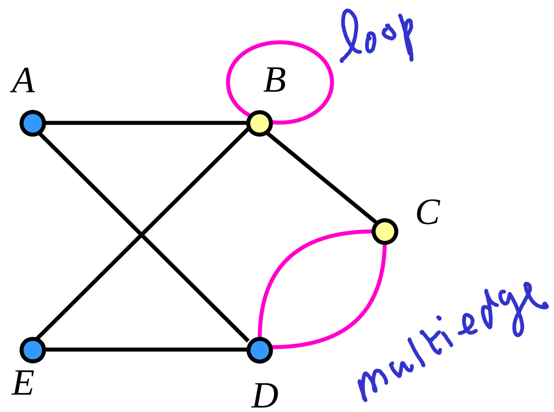


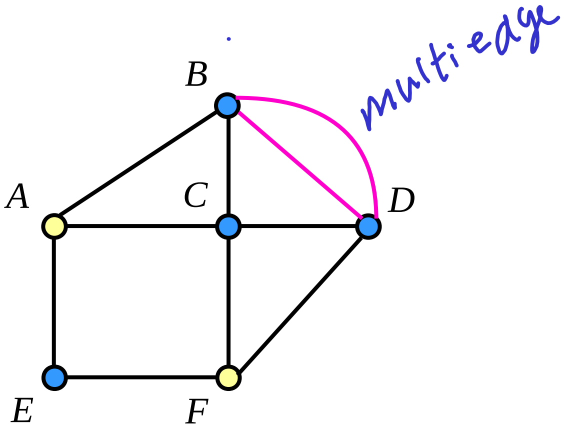
https://en.wikipedia.org/wiki/Travelling_salesman_problem

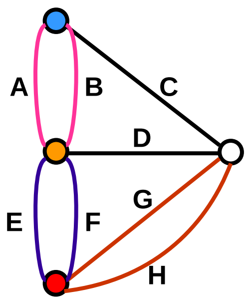
→ simple graph

Multi-Graph

A **multigraph**, as opposed to a **simple graph**, is an undirected graph in which **multiple edges** (and sometimes **loops**) are <u>allowed</u>.



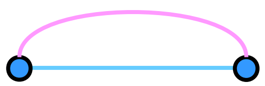




https://en.wikipedia.org/wiki/Travelling_salesman_problem

Multiple Edges

- multiple edges
- parallel edges
- Multi-edges



are two or more edges
that are incident to the same two vertices

A **simple graph** has no multiple edges.

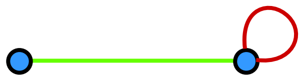
https://en.wikipedia.org/wiki/Travelling_salesman_problem

Loop

- a loop
- a self-loop
- a buckle

is an edge that connects a vertex to itself.

A **simple graph** contains no loops.

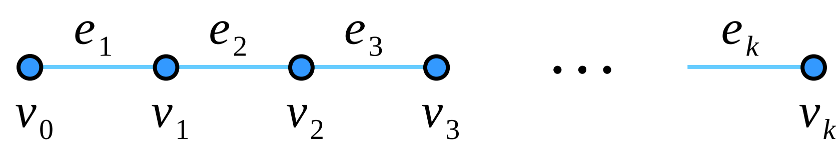


https://en.wikipedia.org/wiki/Travelling_salesman_problem

Walks

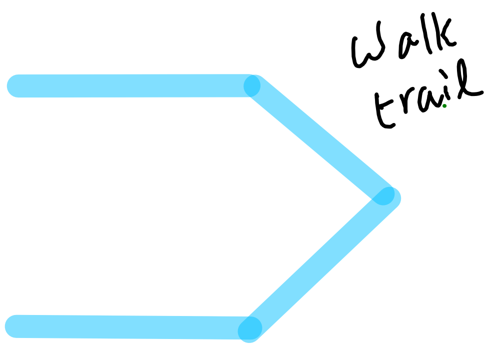
For a graph G= (V, E), a **walk** is defined as a sequence of alternating **vertices** and **edges** such as $v_0, e_1, v_1, e_2, \cdots, e_k, v_k$

where each edge $e_i = \{v_{i-1}, v_i\}$



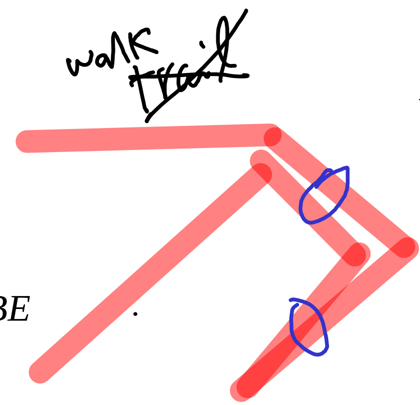
The length of this walk is k

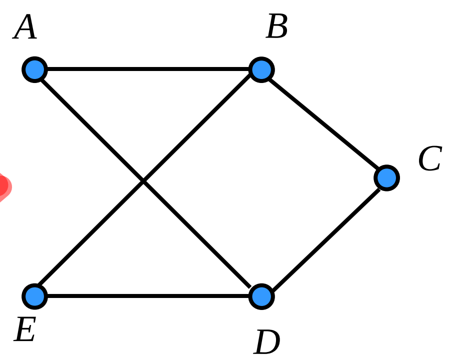
Edges are allowed to be repeated $e_i = e_j$ *for some* i, j



ABCDE

ABCDCBE





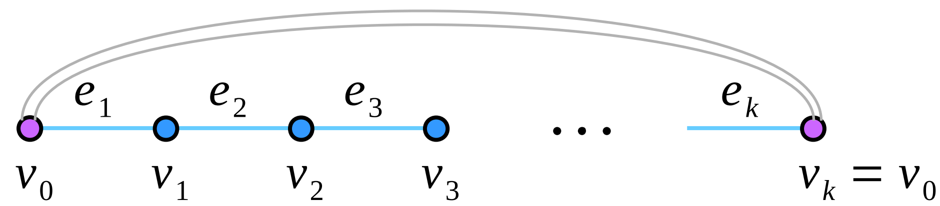
repeated edges

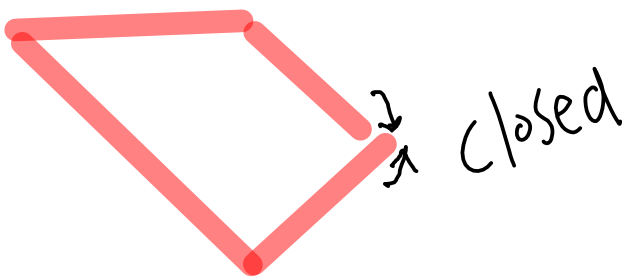
http://mathonline.wikidot.com/walks-trails-paths-cycles-and-circuits

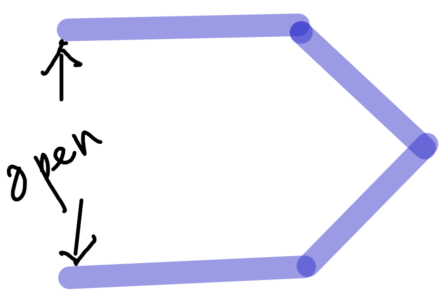
Open / Closed Walks

A walk is considered to be **closed** if the **starting** vertex is the same as the **ending** vertex.

Otherwise **open**



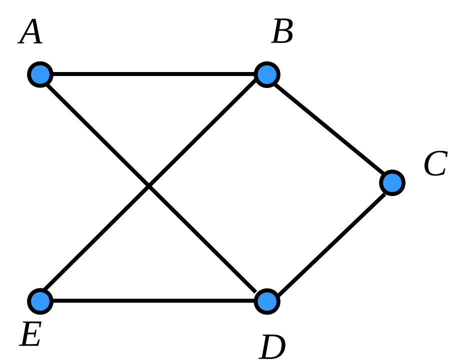


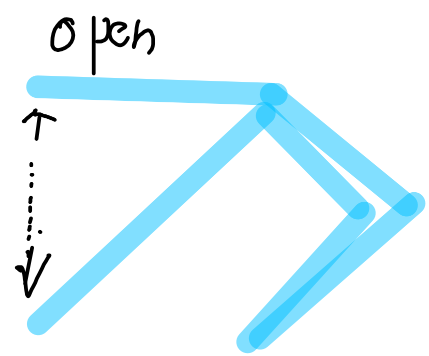


closed walk *ABCDA*

open walk *ABCDE*

open walk *ABCDCBE*

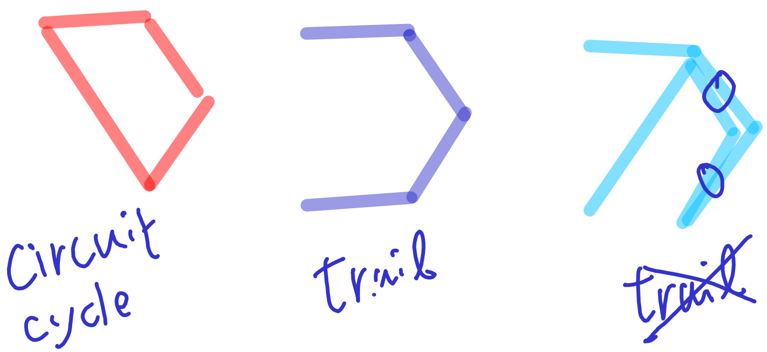


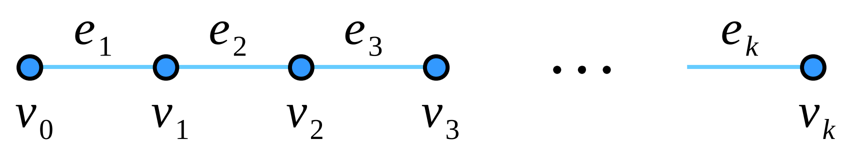


http://mathonline.wikidot.com/walks-trails-paths-cycles-and-circuits

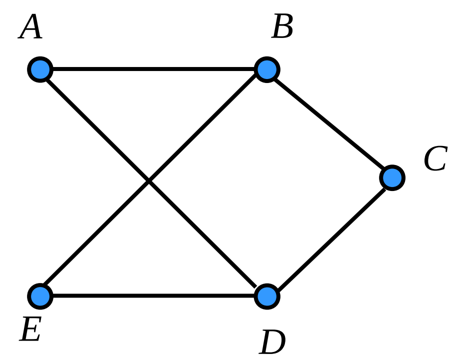
Trails

A **trail** is defined as a **walk** with no repeated **edges**. $e_i \neq e_j \;\; for\ all\ i, j$





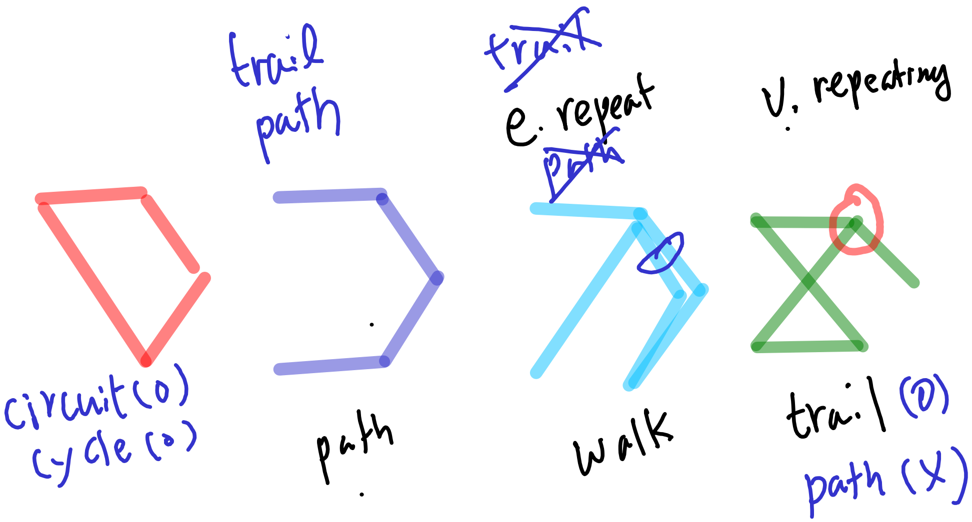
closed trail	closed walk	*ABCDA*
open trail	open walk	*ABCDE*
~~open trail~~	open walk	*ABCDCBE*

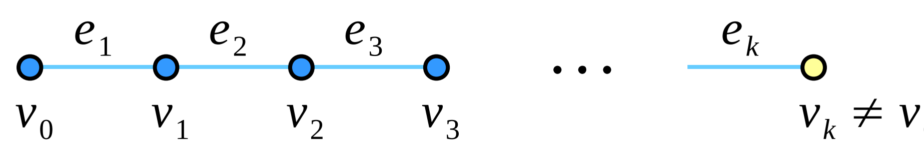


http://mathonline.wikidot.com/walks-trails-paths-cycles-and-circuits

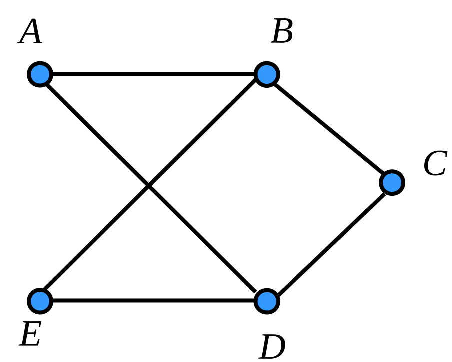
A **path** is defined as a **open trail** with no repeated **vertices**.

$$e_i \neq e_j \quad for\ all\ i,\ j$$
$$v_i \neq v_j \quad for\ all\ i,\ j$$





~~path~~	closed trail	closed walk	*ABCDA*
path	open trail	open walk	*ABCDE*
~~path~~	~~open trail~~	open walk	*ABCDCBE*
~~path~~	open trail	open walk	*BEDABC*



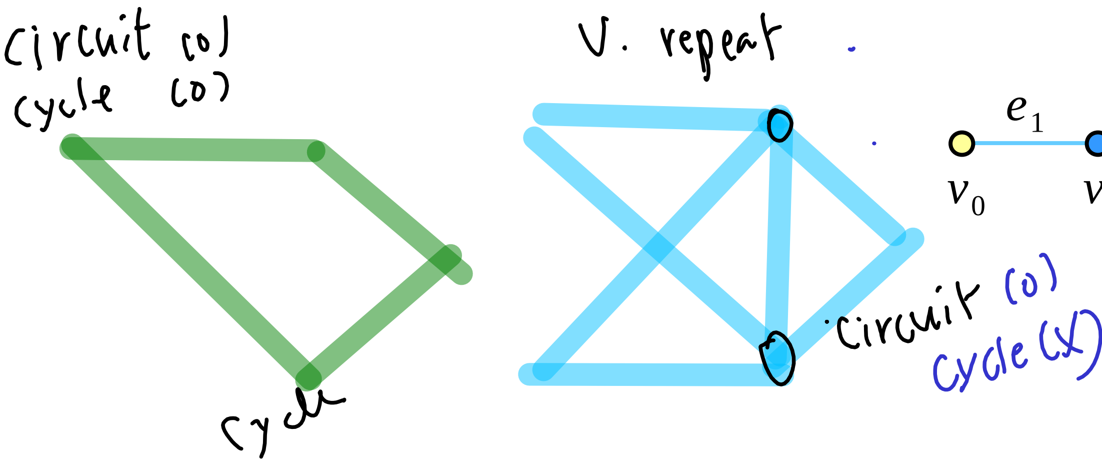
http://mathonline.wikidot.com/walks-trails-paths-cycles-and-circuits

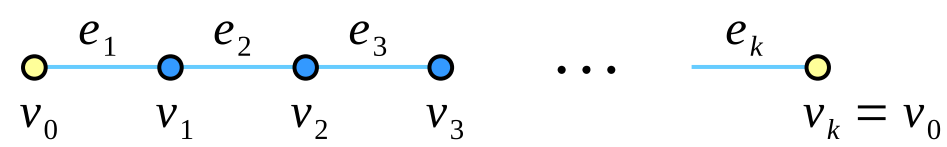
Cycles

A **cycle** is defined as a **closed trail** with no repeated **vertices** except the **start/end vertex**

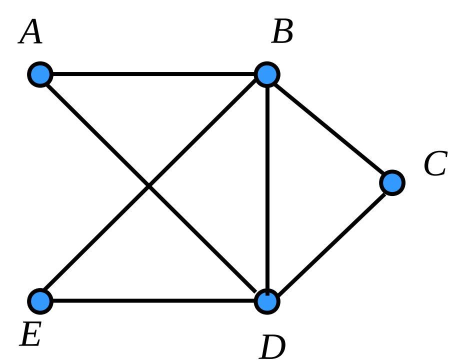
$e_i \neq e_j$ *for all* i, j

$v_i \neq v_j$ *for all* i, j





cycle	circuit	closed walk	*ABCDA*
~~cycle~~	circuit	closed walk	*ABCDEBDA*



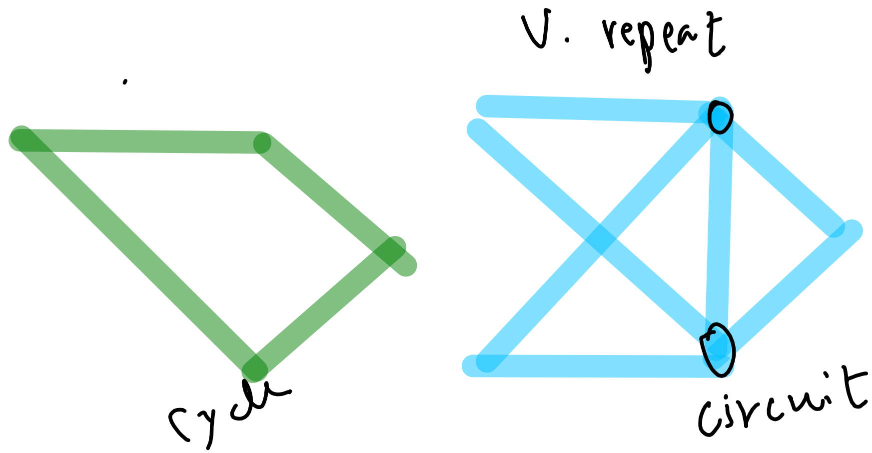
http://mathonline.wikidot.com/walks-trails-paths-cycles-and-circuits

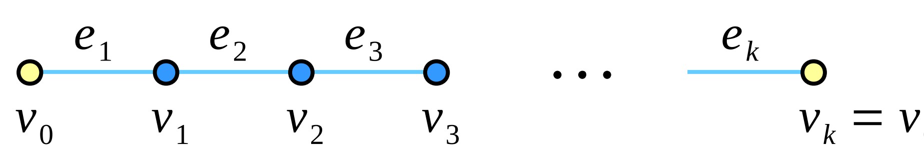
Circuits

A **circuit** is defined as a **closed trail** with possibly repeated **vertices** but with no repeated **edges**

$e_i \neq e_j \quad for\ all\ i, j$

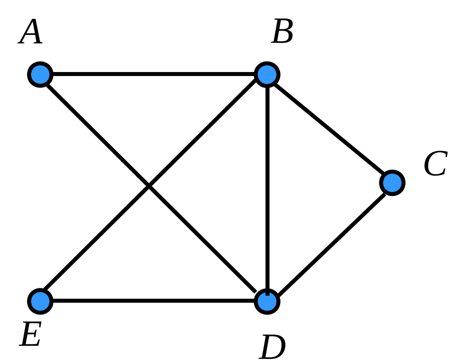
$v_i = v_j \quad for\ some\ i, j$





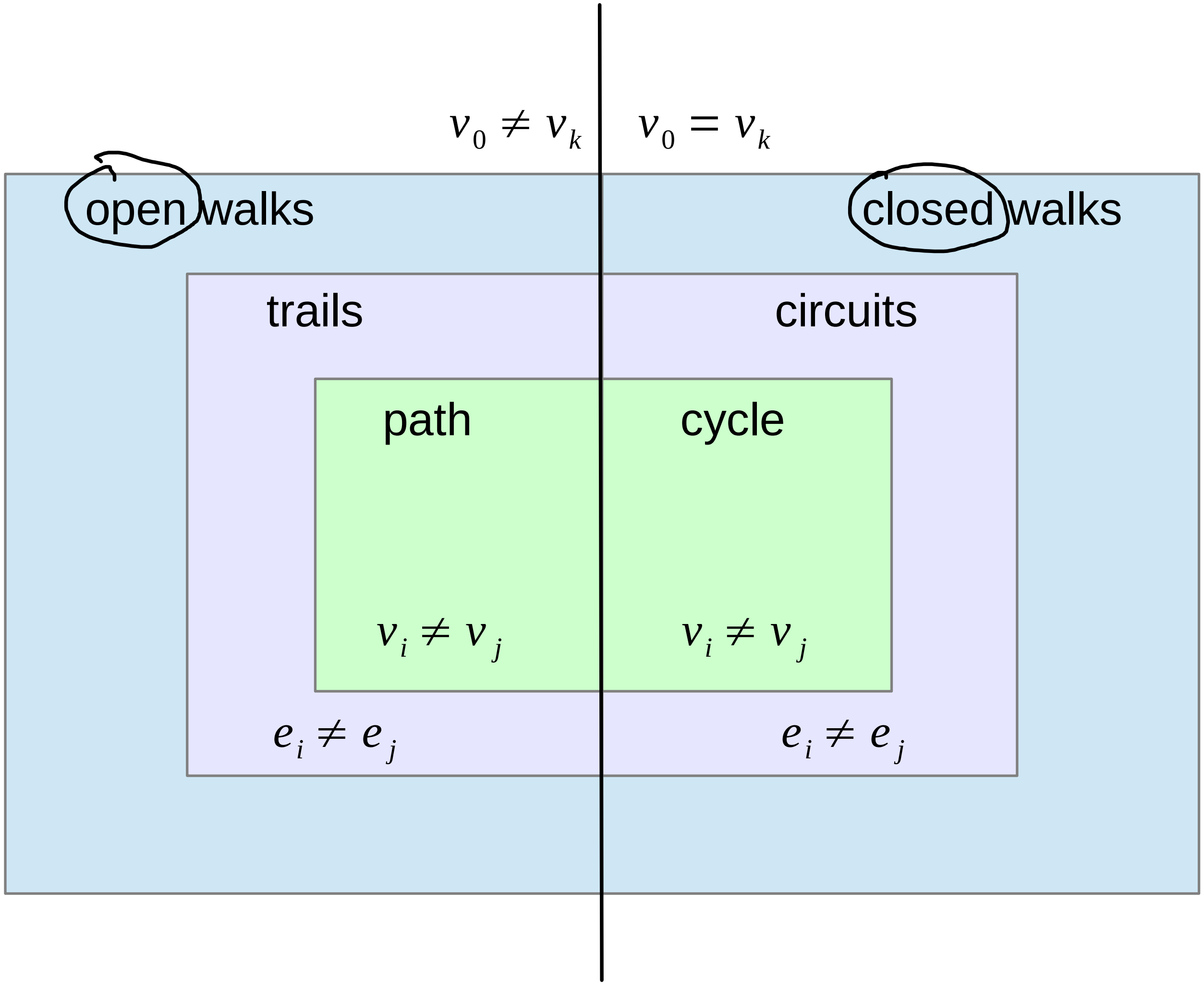
circuit closed walk *ABCDA*

circuit closed walk *ABCDEBDA*

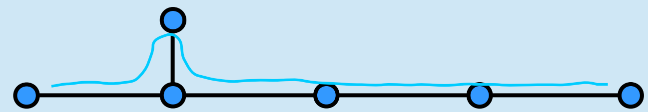
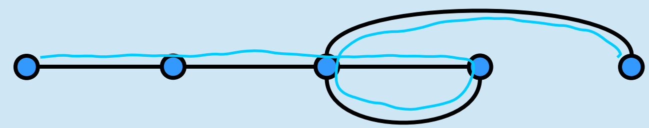
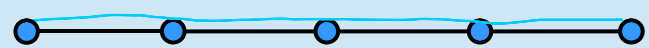
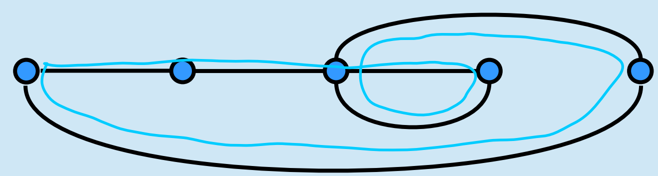
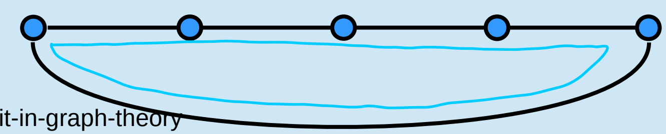


http://mathonline.wikidot.com/walks-trails-paths-cycles-and-circuits

Walk, Trail, Path, Circuit, Cycle



Walk, Trail, Path, Circuit, Cycle

	Vertices	Edges		
Walk	may repeat	may repeat	(Closed/Open)	
Trail	may repeat	cannot repeat	(Open)	
Path	cannot repeat	cannot repeat	(Open)	
Circuit	may repeat	cannot repeat	(Closed)	
Cycle	cannot repeat	cannot repeat	(Closed)	

https://math.stackexchange.com/questions/655589/what-is-difference-between-cycle-path-and-circuit-in-graph-theory

References

[1] http://en.wikipedia.org/
[2]

the same cycle

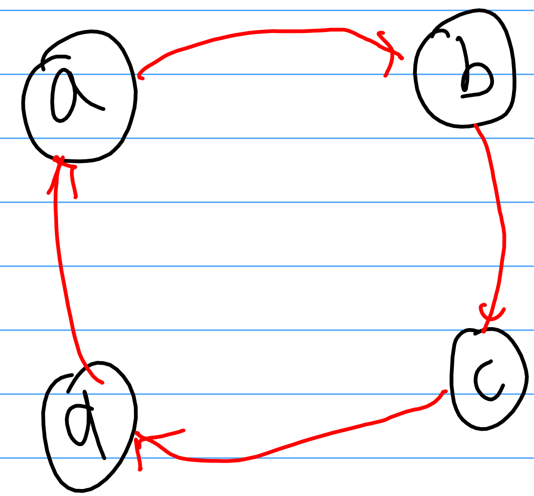
a - b - c - d

b - c - d - a

c - d - a - b

d - a - b - c

≡



Eulerian Cycle (2A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Path and Trail

A **path** is a **trail** in which all **vertices** are distinct. (except possibly the first and last)

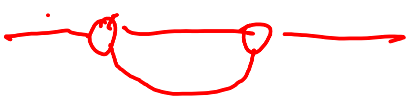
↔ cycle

V. repeat X
e. repeat X

A **trail** is a **walk** in which all **edges** are distinct. ↔ circuit

e. repeat X

	Vertices	Edges	
Walk	may repeat	may repeat	(Closed/Open)
Trail	may repeat	cannot repeat	(Open)
Path	cannot repeat	cannot repeat	(Open)
Circuit	may repeat	cannot repeat	(Closed)
Cycle	cannot repeat	cannot repeat	(Closed)



https://en.wikipedia.org/wiki/Eulerian_path

Simple Paths and Cycles

Most literatures require that all of the **edges** and **vertices** of a **path** be distinct from one another.

But, some do not require this and instead use the term **simple path** to refer to a **path** which contains no repeated **vertices**.

A **simple cycle** may be defined as a **closed walk** with no repetitions of **vertices** and **edges** allowed, other than the repetition of the **starting** and **ending vertex**

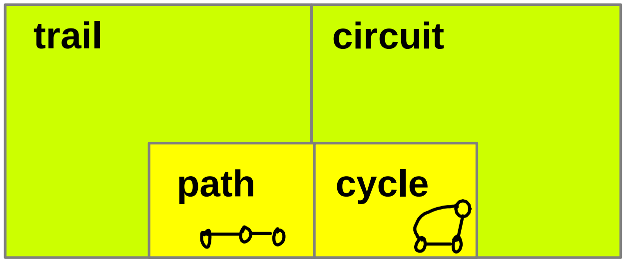
.

There is considerable variation of terminology!!!
Make sure which set of definitions are used...

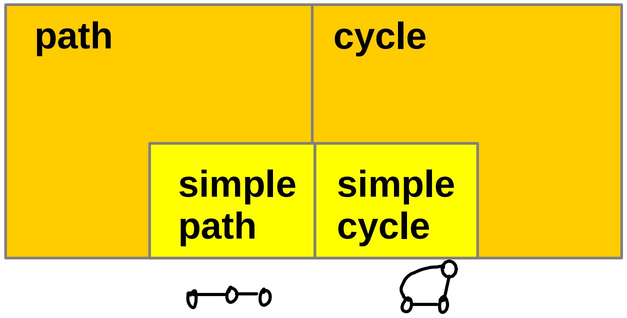
https://en.wikipedia.org/wiki/Eulerian_path

Simple Paths and Cycles

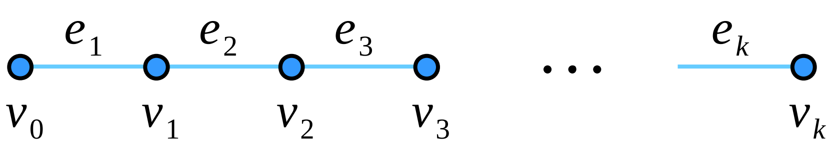
most



some

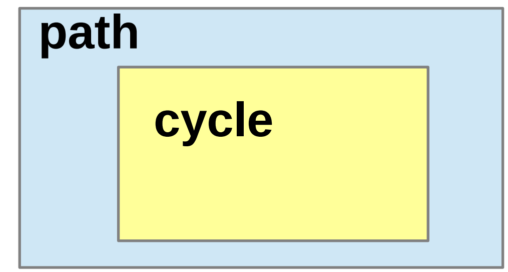


Paths and Cycles



path $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k$

cycle $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k$ $(v_0 = v_k)$



path $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k$ $(v_0 \neq v_k)$

cycle $v_{0,}\ e_{1,}\ v_{1,}\ e_{2,}\ \cdots,\ e_k,\ v_k$ $(v_0 = v_k)$

Euler Cycle

Some people reserve the terms **path** and **cycle** to mean non-self-intersecting path and cycle. — no repeating vertices

A (potentially) self-intersecting path is known as a **trail** or an **open walk**; — repeating vertices

and a (potentially) self-intersecting cycle, a **circuit** or a **closed walk**. — repeating vertices

This ambiguity can be avoided by using the terms **Eulerian trail** and **Eulerian circuit** when self-intersection is allowed — repeating vertices

https://en.wikipedia.org/wiki/Eulerian_path

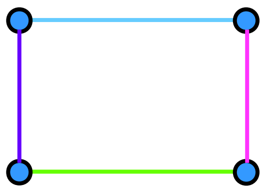
Euler Cycle

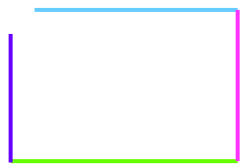
visits <u>every</u> <u>**edge**</u> exactly <u>once</u>

the existence of **Eulerian cycles**

all **vertices** in the graph have an **even** degree

connected graphs with **all vertices** of **even** degree h
ave an **Eulerian cycles**

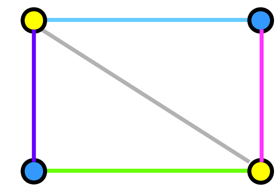




https://en.wikipedia.org/wiki/Eulerian_path

Euler Path

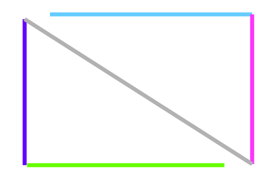
visits every **edge** exactly once



the existence of **Eulerian paths**

all the **vertices** in the graph have an **even** degree

except only **two** vertices with an **odd** degree



An **Eulerian path** starts and ends at different vertices
An **Eulerian cycle** starts and ends at the same vertex.

https://en.wikipedia.org/wiki/Eulerian_path

Conditions for Eulerian Cycles and Paths

An odd vertex = a vertex with an odd degree
An even vertex = a vertex with an even degree

# of **odd** vertices	Eulerian **Path**	Eulerian **Cycle**
0	No ←	**Yes**
2	**Yes** →	No
4,6,8, …	No	No
1,3,5,7, …	No such graph	No such graph

If the graph is connected

http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

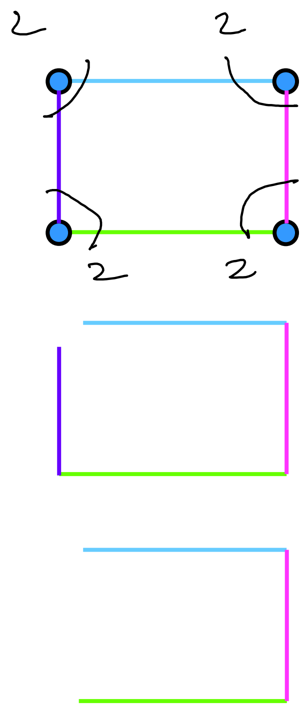
The number of odd vertices

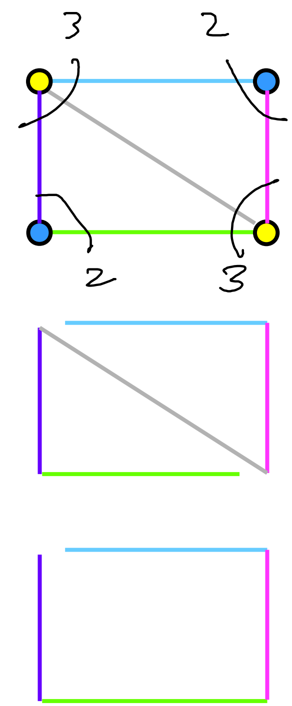
# of **odd** vertices	Eulerian **Path**	Eulerian **Cycle**
0	**No**	**Yes**
2	**Yes**	**No**

of **odd** vertices
= 0

Eulerian Cycle

No Eulerian Path





of **odd** vertices
= 2

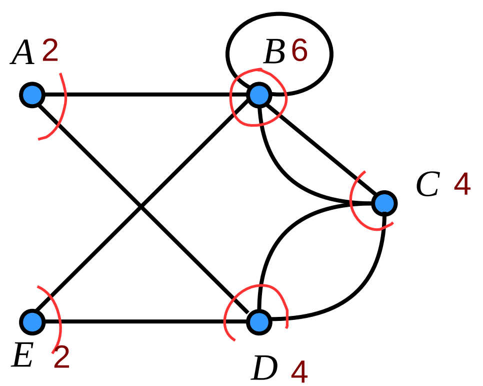
Eulerian Path

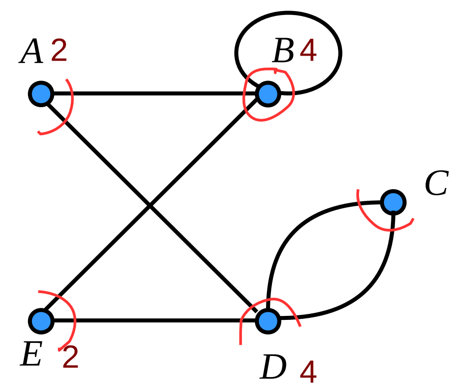
No Eulerian Cycle

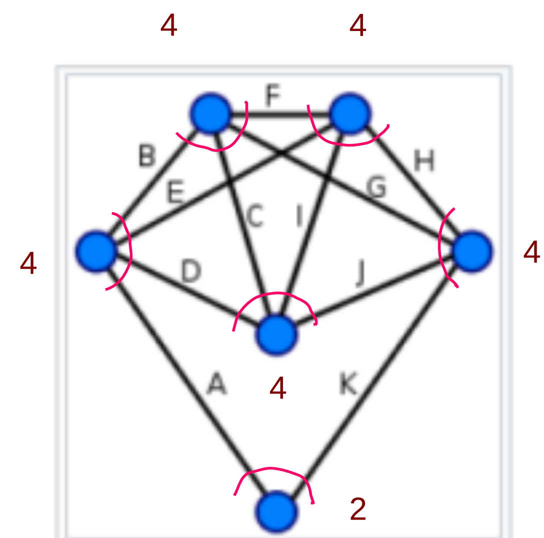
Eulerian Graph

Eulerian graph :
a graph with an **Eulerian cycle**
a graph with **every vertex** of **even degree**
(the number of **odd vertices** is 0)

These definitions coincide for connected graphs.

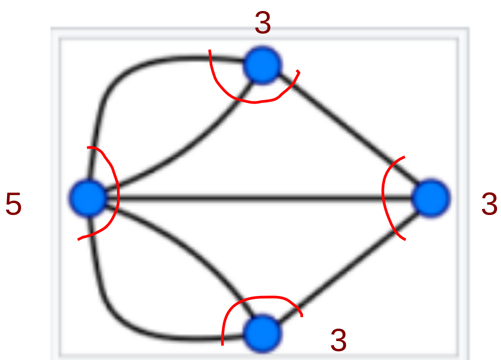






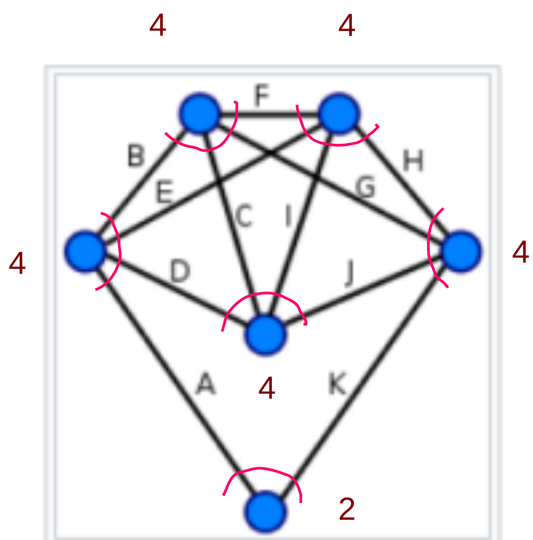
Every vertex of this graph has an even degree. Therefore, this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

Odd Degree and Even Degree



The Königsberg Bridges multigraph. This multigraph is not Eulerian, therefore, a solution does not exist.

All <u>odd</u> degree vertices

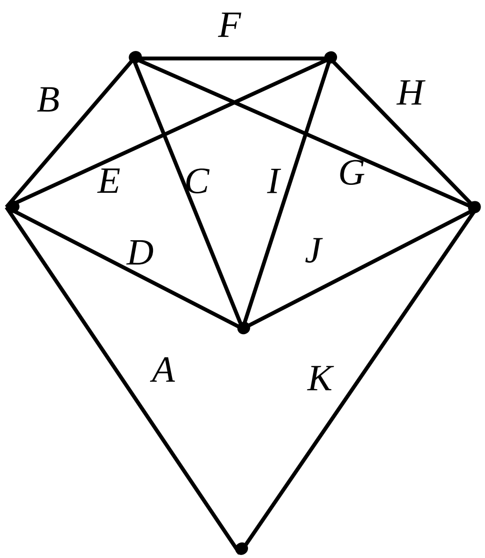


Every vertex of this graph has an even degree. Therefore, this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

All <u>even</u> degree vertices

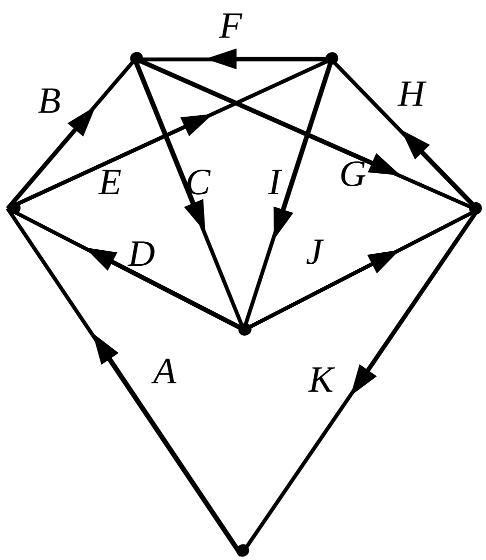
https://en.wikipedia.org/wiki/Eulerian_path

Euler Cycle Example



ABCDEFGHIJK

a path denoted by
the edge names



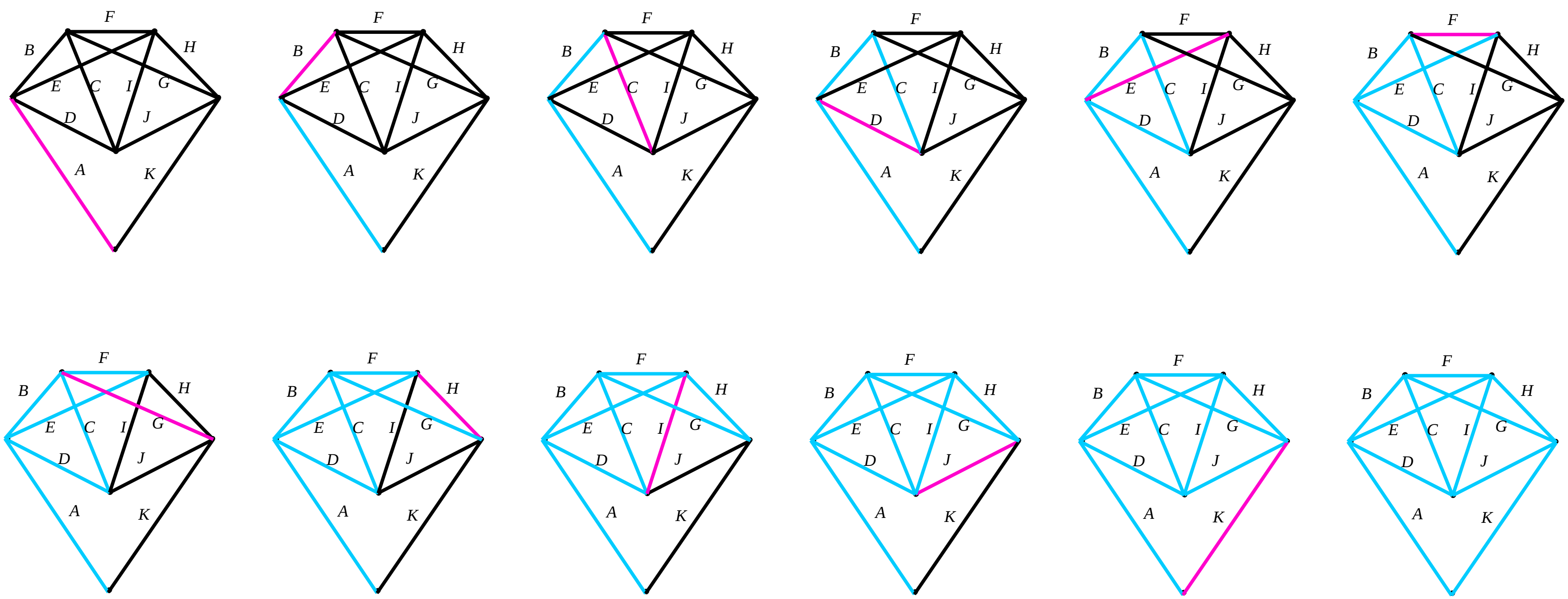
All <u>even</u> degree vertices

Eulerian Cycles

en.wikipedia.org

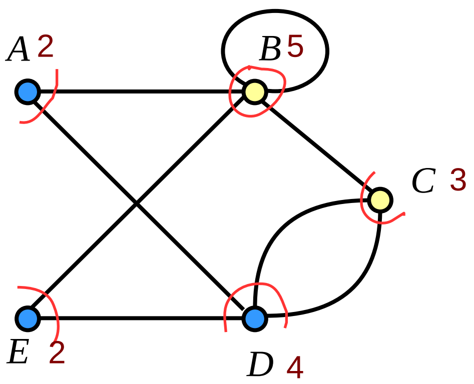
Euler Cycle Example

ABCDEFGHIJK

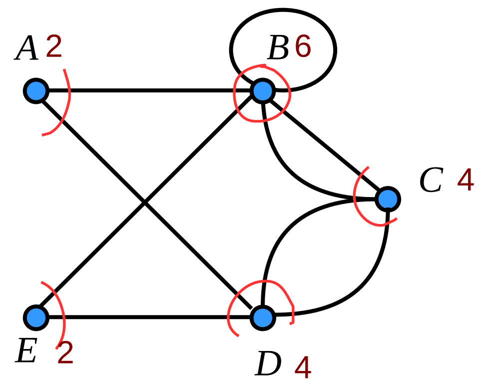


en.wikipedia.org

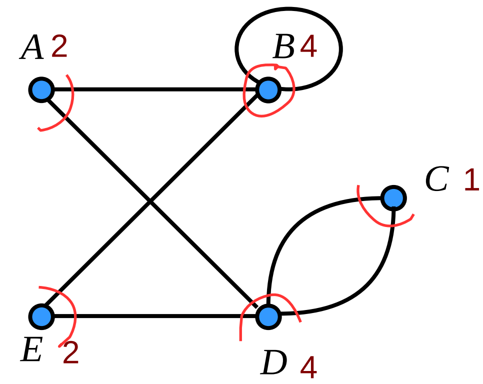
Euler Path and Cycle Examples



Eulerian Path
1. BBADCDEBC
2. CDCBBADEB



Euerian Cycle
1. CDCBBADEBC



Euerian Cycle
2. CDEBBADC

a path denoted by
the vertex names

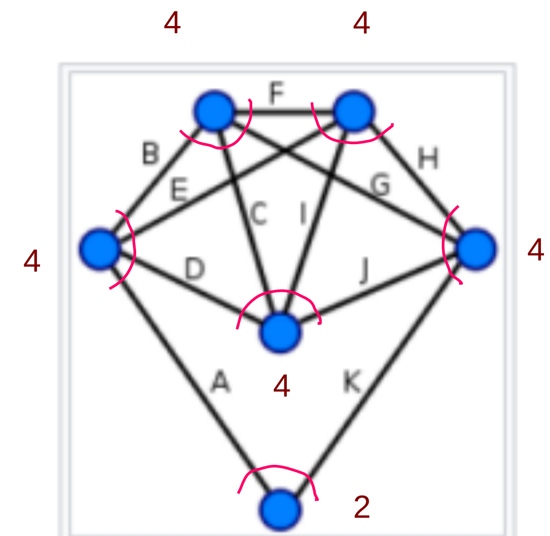
http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

Eulerian Cycles of Undirected Graphs

An **undirected** graph has an **Eulerian <u>cycle</u>**
if and only if every **vertex** has **even degree**,
and all of its **vertices** with **nonzero degree**
belong to a **single <u>connected</u> component**.

An **undirected** graph can be
decomposed into **edge-disjoint cycles**
if and only if all of its **vertices** have **even degree**.

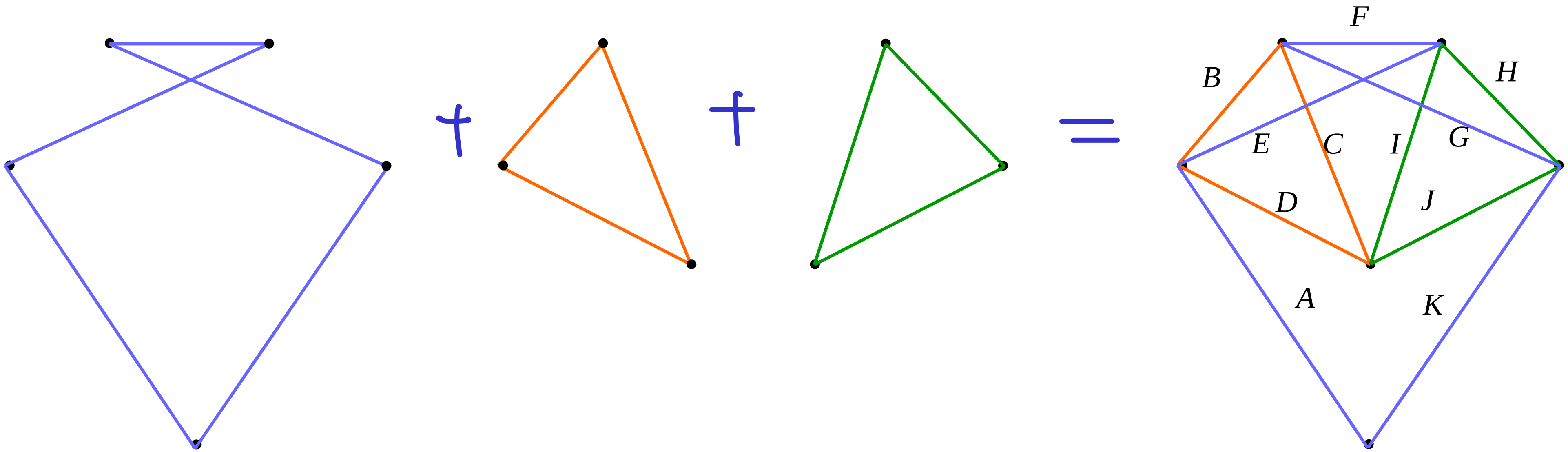
So, a graph has an **Eulerian <u>cycle</u>**
if and only if it can be decomposed
into **edge-disjoint cycles**
and its **nonzero-degree** vertices
belong to a **single connected component**.



Every vertex of this graph has an even degree. Therefore, this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

https://en.wikipedia.org/wiki/Eulerian_path

Edge Disjoint Cycle Decomposition



All even vertices ⟺ **Euerian Cycle** ⟺ **Edge Disjoint Cycles**

Eulerian Paths of Undirected Graphs

An undirected graph has an **Eulerian trail**
if and only if exactly **zero** or **two vertices** have **odd degree**,
and all of its vertices with **nonzero degree**
belong to a **single connected component**.

https://en.wikipedia.org/wiki/Eulerian_path

Eulerian Cycles of DiGraphs

A directed graph has an **Eulerian cycle**
if and only if every vertex has **equal in degree** and **out degree**,
and all of its vertices with nonzero degree
belong to a single strongly connected component.

Equivalently, a directed graph has an Eulerian cycle
if and only if it can be decomposed
into **edge-disjoint directed cycles**
and all of its vertices with nonzero degree
belong to a single strongly connected component.

https://en.wikipedia.org/wiki/Eulerian_path

Eulerian Paths of DiGraphs

A directed graph has an **Eulerian path**
if and only if **at most one** vertex has (out-degree) − (in-degree) = 1,
at most one vertex has (in-degree) − (out-degree) = 1,
every other vertex has equal in-degree and out-degree,
and all of its vertices with nonzero degree belong to a single connected component of the underlying undirected graph.

https://en.wikipedia.org/wiki/Eulerian_path

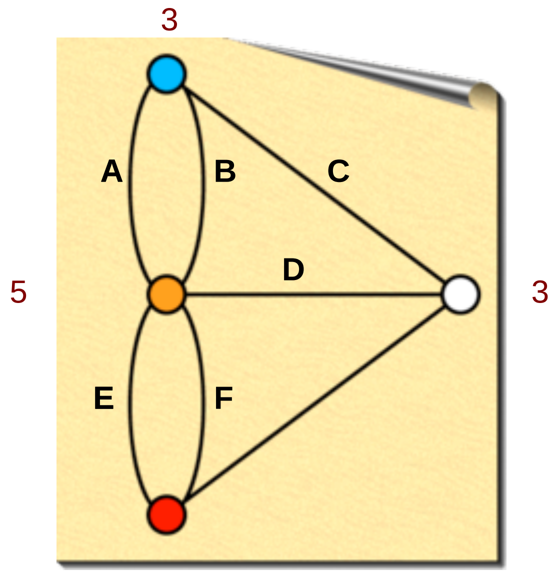
Seven Bridges of Königsberg

The problem was to devise a walk through the city that would cross each of those bridges once and only once.

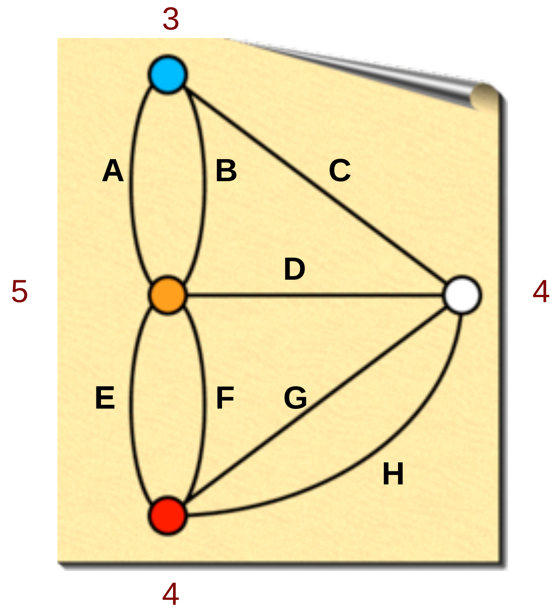
https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

Seven and Eight Bridges Problems

7 bridges problem



8 bridges problem



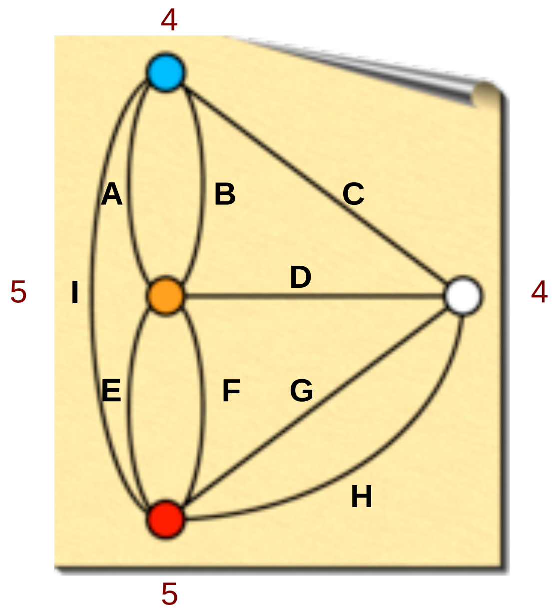
Eulerian Path

AEHGFDCB

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

Nine and Ten Bridges Problems

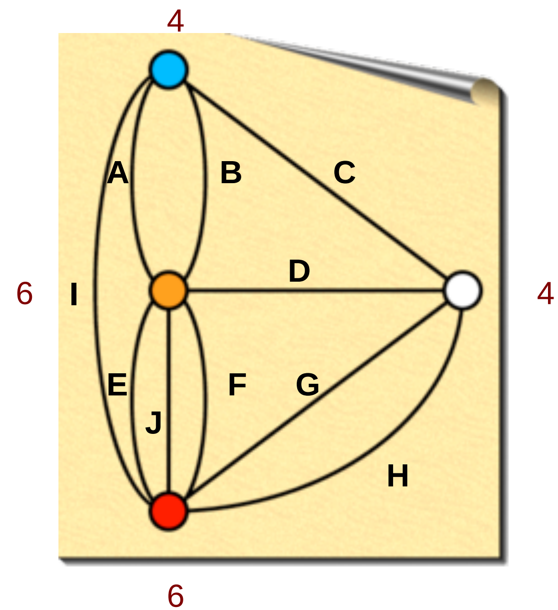
9 bridges problem



Eulerian Path

EHGFDCBAI

10 bridges problem

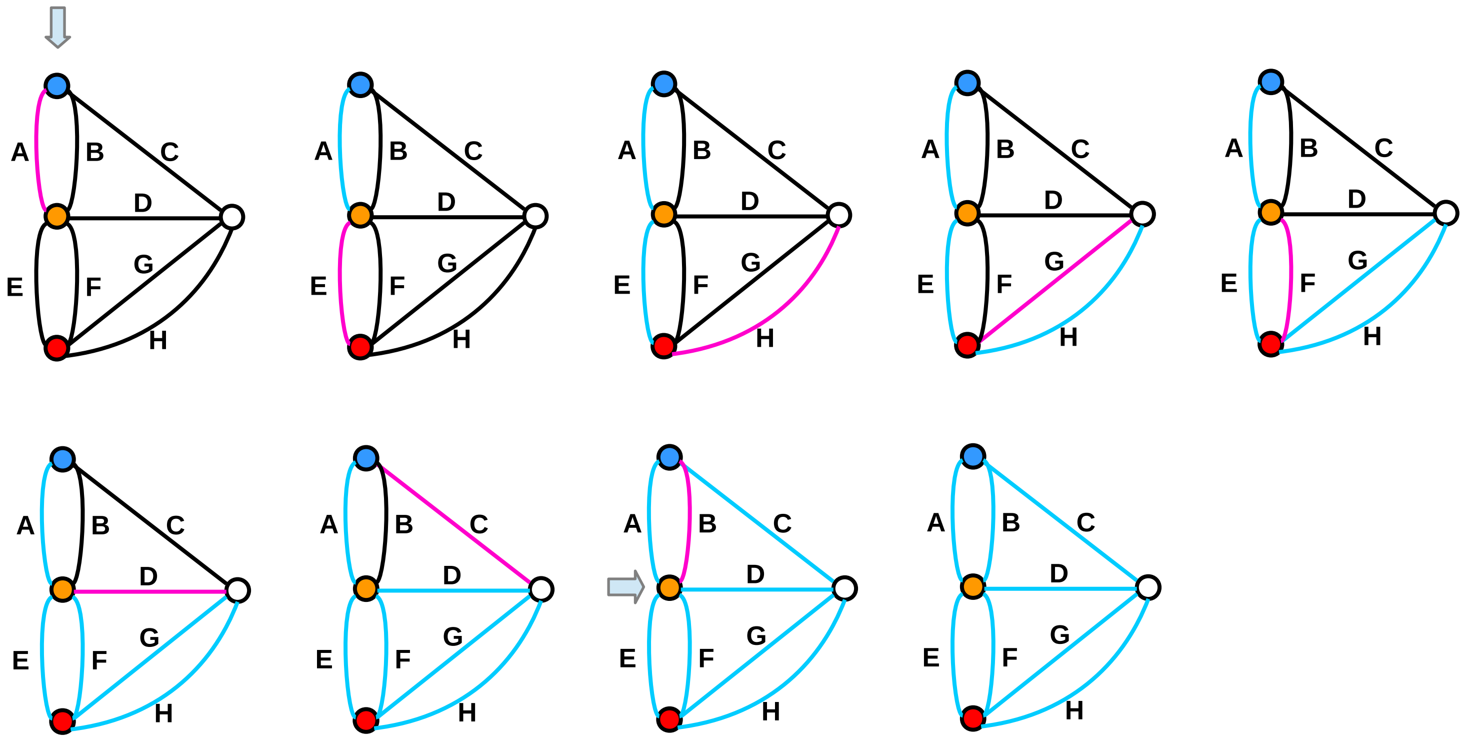


Eulerian Cycle

AEHGFDCBJI

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

8 bridges – Eulerian Path

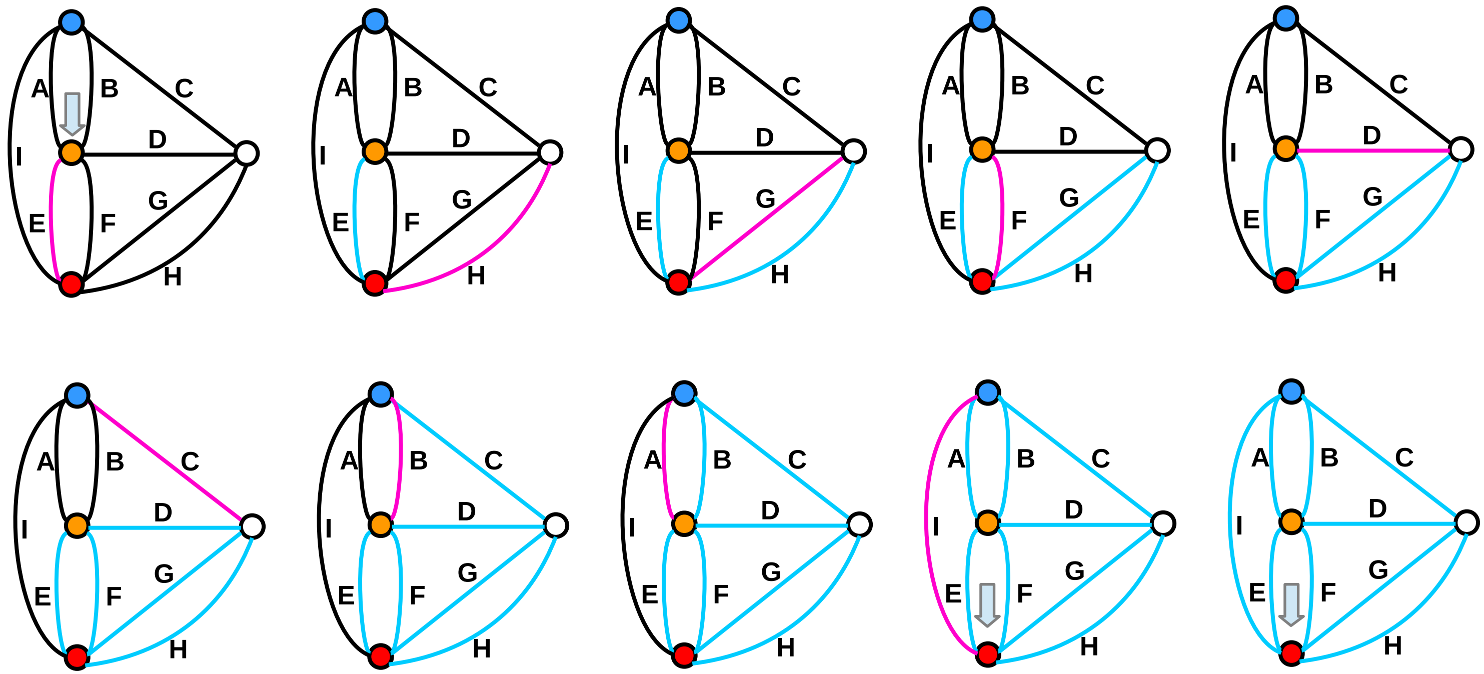


Eulerian Path

AEHGFDCB

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

9 bridges – Eulerian Path

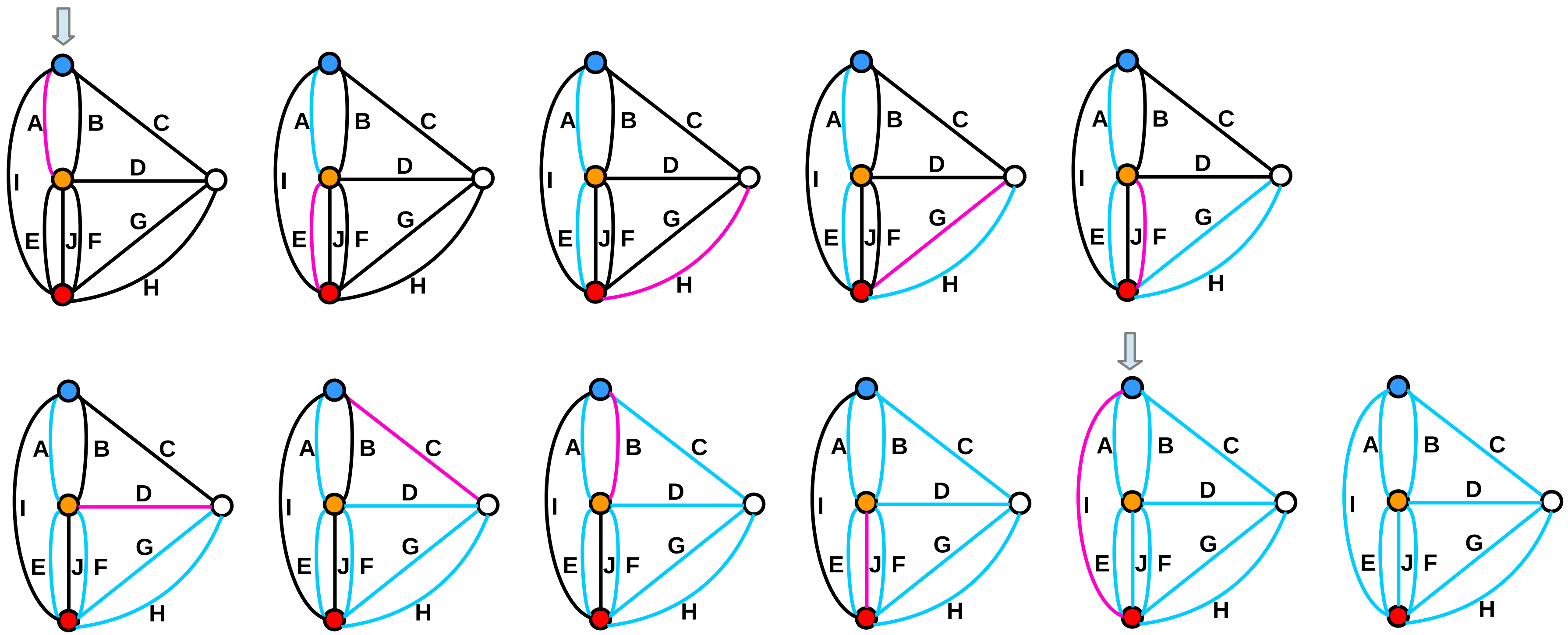


Eulerian Path

EHGFDCBAI

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

10 bridges – Eulerian Cycle



Eulerian Cycle

AEHGFDCBJI

https://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg

Fleury's Algorithm

To find an Eulerian path or an Eulerian cycle:

1. make sure the graph has either **0** or **2 odd** vertices

2. if there are **0 odd** vertex, start anywhere**.**
   If there are **2 odd** vertices, start at one of the two vertices

3. follow edges one at a time.
   If you have a choice between a **bridge** and a **non-bridge**,
   Always choose the **non-bridge**

4. stop when you run out of edge

http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

Bridges

A bridge edge

Removing a single edge from a connected graph
can make it disconnected

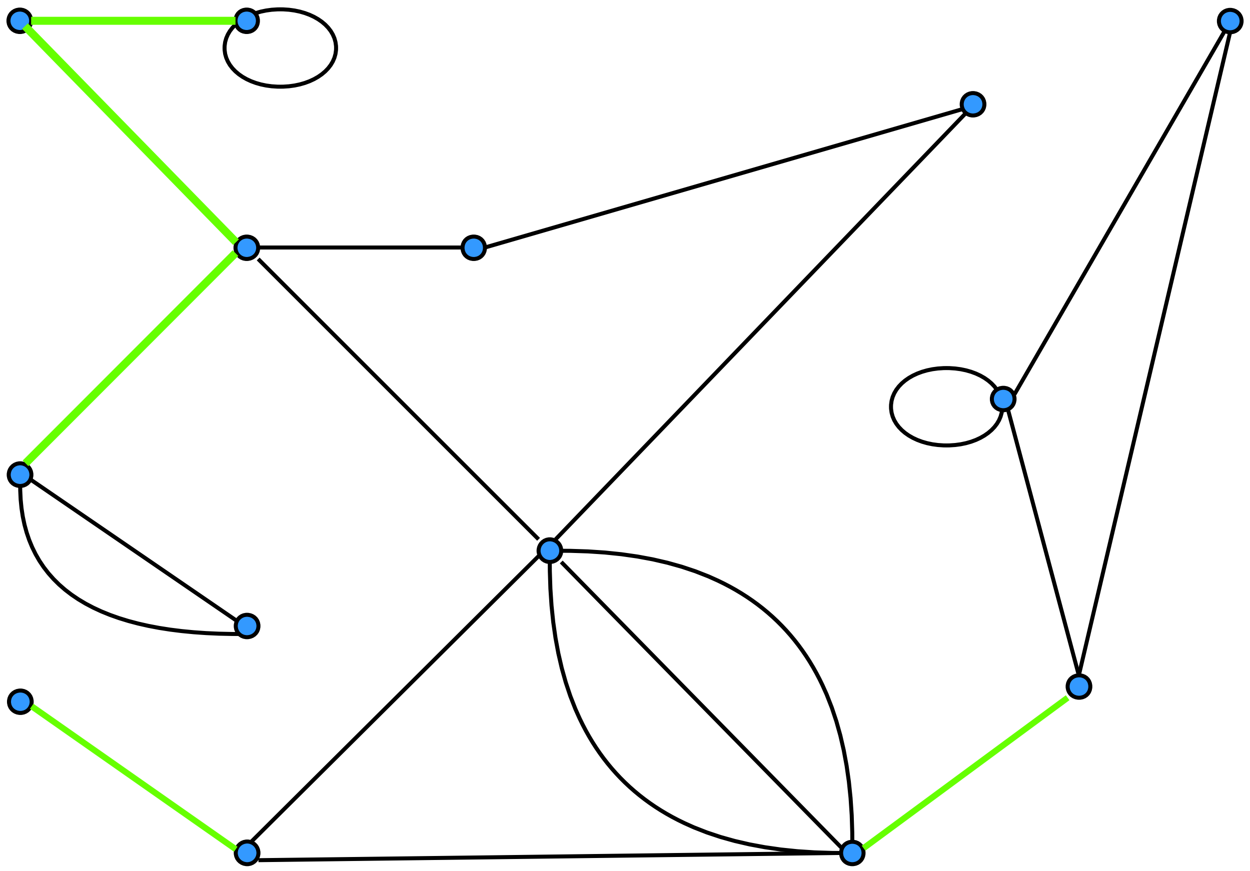
Non-bridge edges

Loops cannot be bridges

Multiple edges cannot be bridges

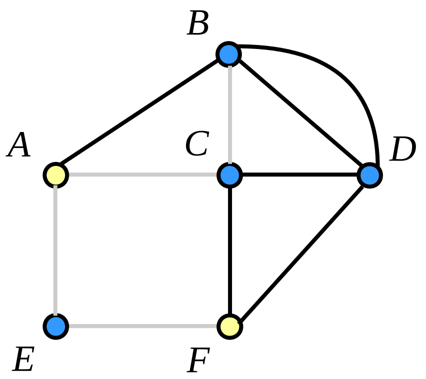
http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

Bridge examples in a graph

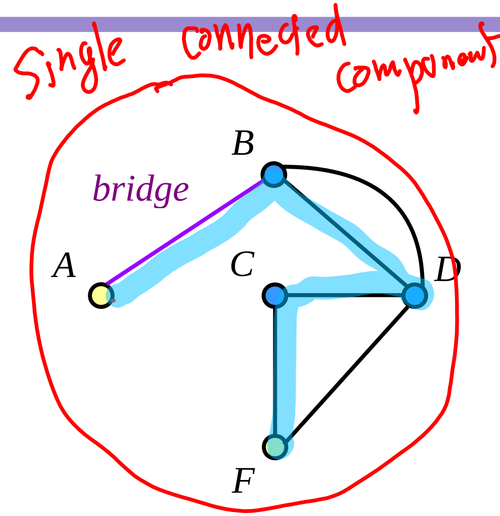


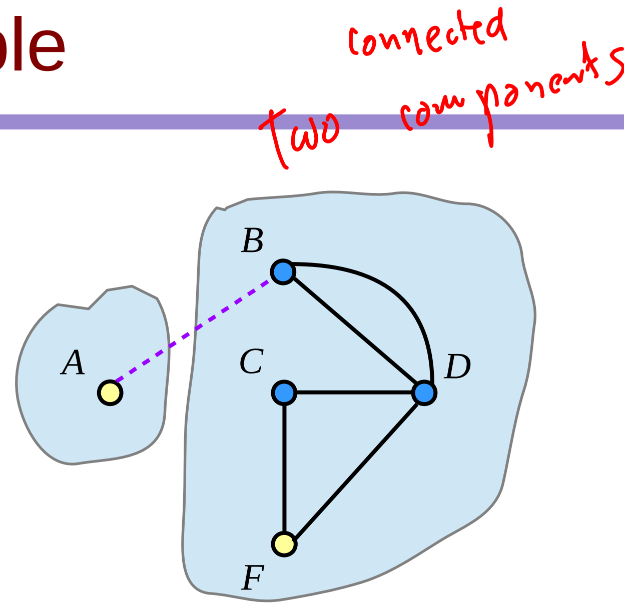
http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

Bridges must be avoided, if possible

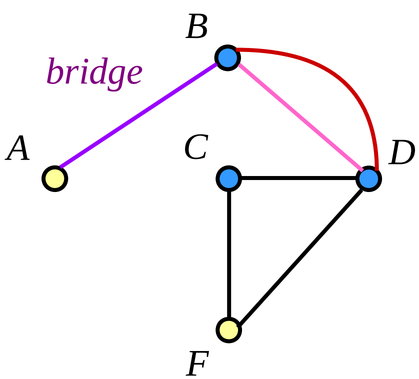


FEACB



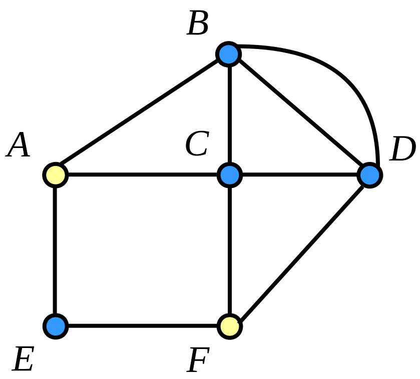


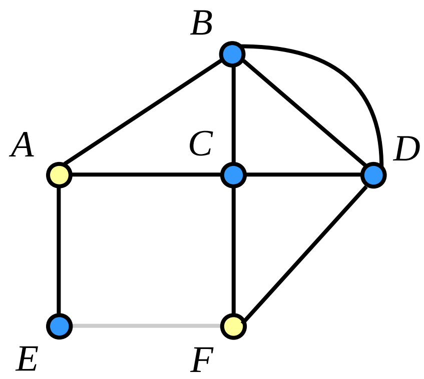
If there exists other choice other than a bridge
The bridge must <u>not</u> be chosen.



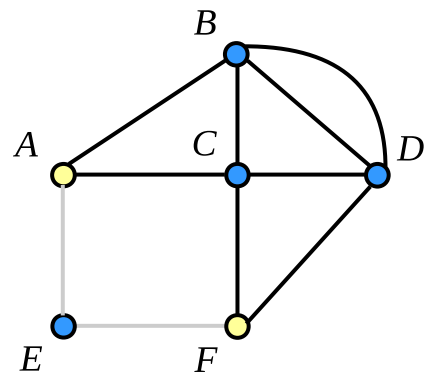
http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

Fleury’s Algorithm (1)

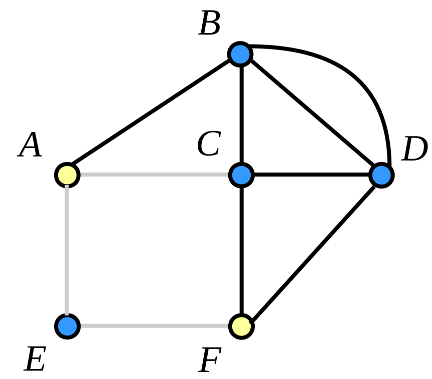




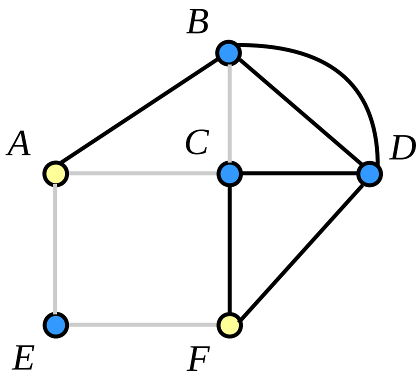
FE



FEA



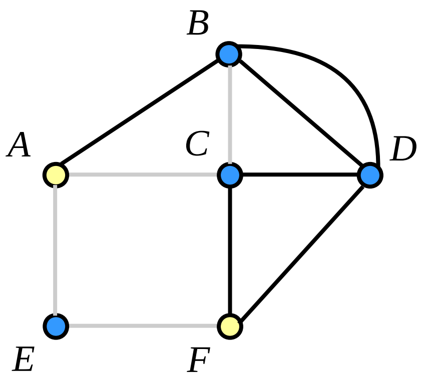
FEAC



FEACB

http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

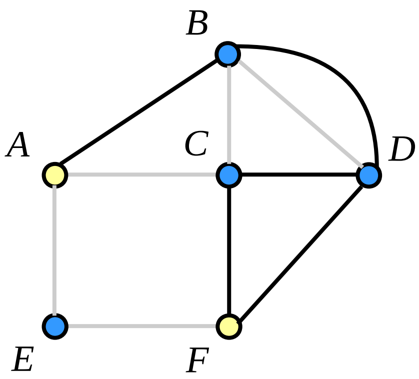
Fleury's Algorithm (2)



FEACB

BA : *bridge*

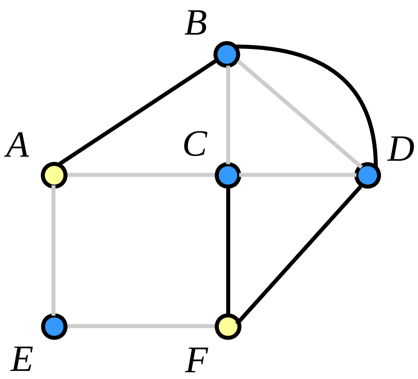
BD : *chosen*



FEACBD

DB : *bridge*

DC : *chosen*

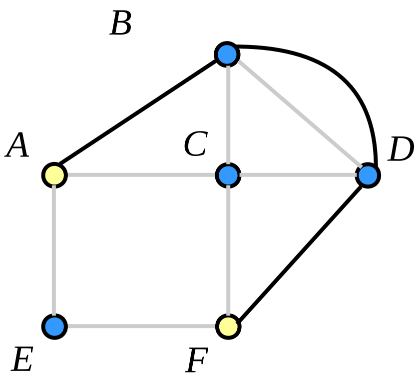


FEACBDC

CF : *bridge*

CF : *chosen*

no other choice



FEACBDCF

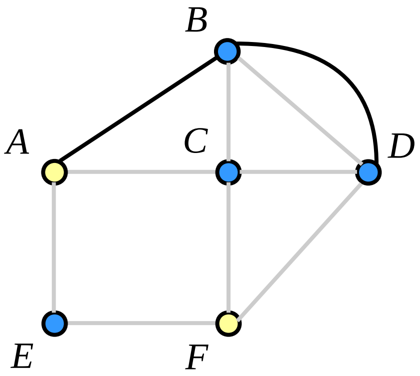
FD : *bridge*

FD : *chosen*

no other choice

http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

Fleury's Algorithm (3)

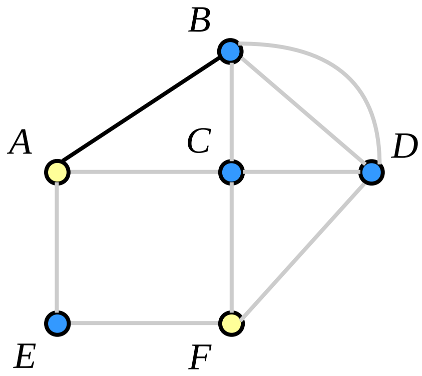


FEACBDCFD

DB : *bridge*

DB : *chosen*

no other choice

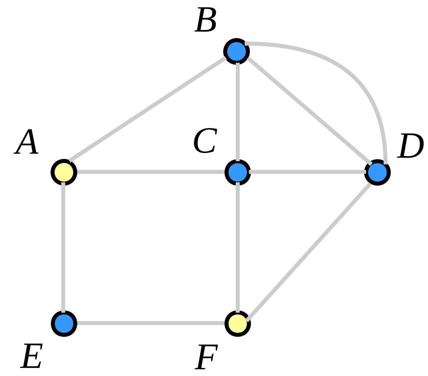


FEACBDCFDB

BA : *bridge*

BA : *chosen*

no other choice



http://people.ku.edu/~jlmartin/courses/math105-F11/Lectures/chapter5-part2.pdf

References

[1] http://en.wikipedia.org/
[2]

Hamiltonian Cycle (3A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Hamiltonian Cycles

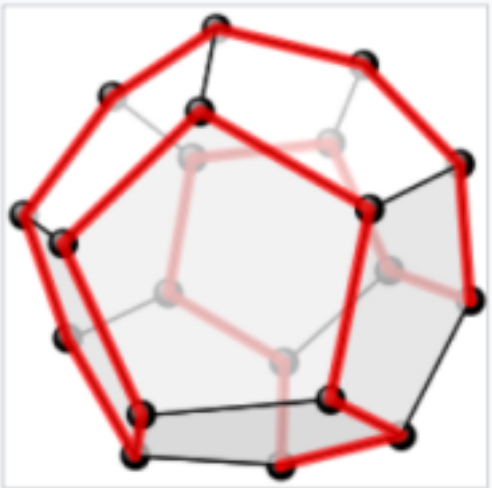
A Hamiltonian path is a path
in an undirected or directed graph
that visits **each vertex** exactly **once**.

A Hamiltonian cycle is
a Hamiltonian path that is a cycle.

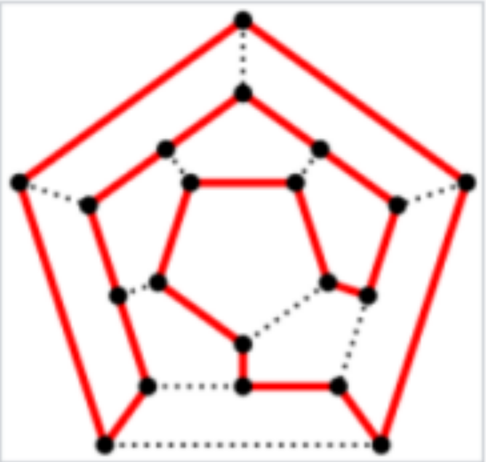
the Hamiltonian path problem is NP-complete.

https://en.wikipedia.org/wiki/Hamiltonian_path

Hamiltonian Cycles



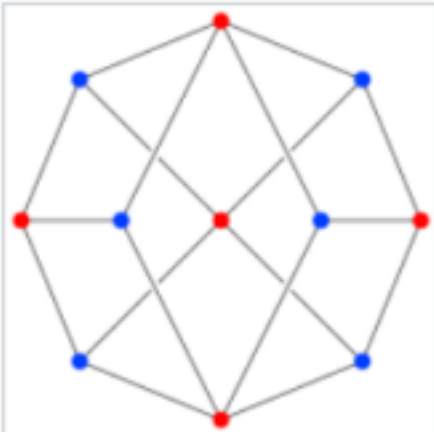
One possible Hamiltonian cycle through every vertex of a dodecahedron is shown in red – like all platonic solids, the dodecahedron is Hamiltonian



The above as a two-dimensional planar graph

https://en.wikipedia.org/wiki/Hamiltonian_path

Hamiltonian Cycles



The Herschel graph is the smallest possible polyhedral graph that does not have a Hamiltonian cycle.

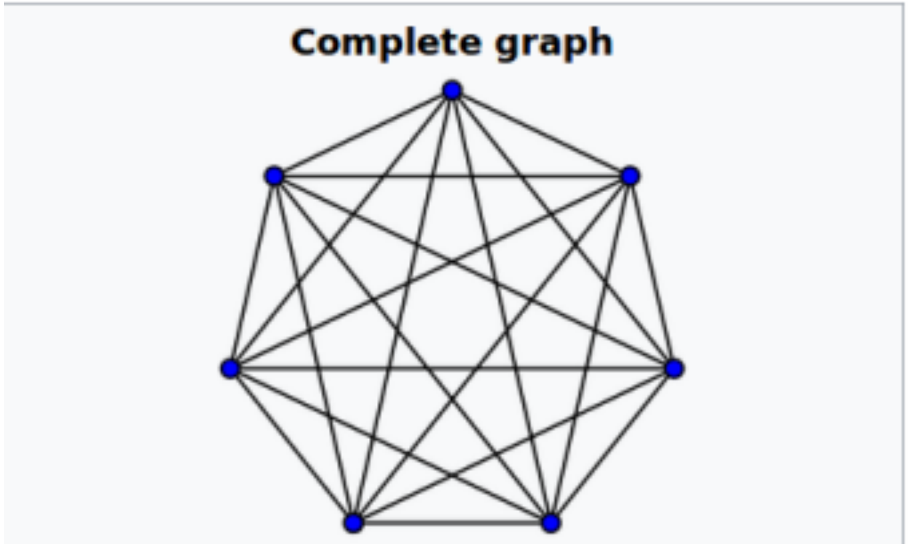
https://en.wikipedia.org/wiki/Hamiltonian_path

Hamiltonian Cycles

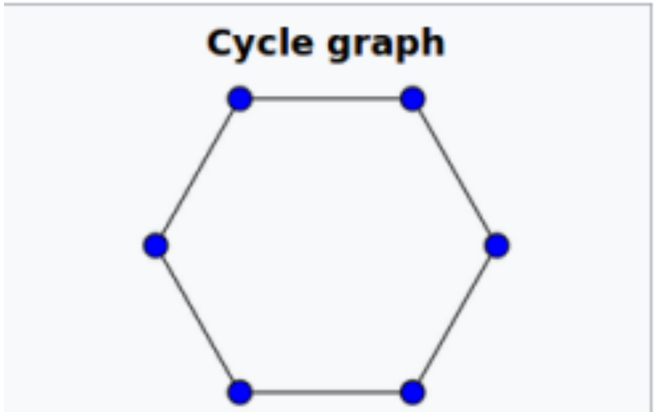
- a **complete graph** with more than two vertices is Hamiltonian
- every **cycle graph** is Hamiltonian
- every **tournament** has an odd number of Hamiltonian paths
- every **platonic solid**, considered as a graph, is Hamiltonian
- the **Cayley graph** of a finite **Coxeter** group is Hamiltonian

https://en.wikipedia.org/wiki/Hamiltonian_path

Complete Graphs and Cycle Graphs



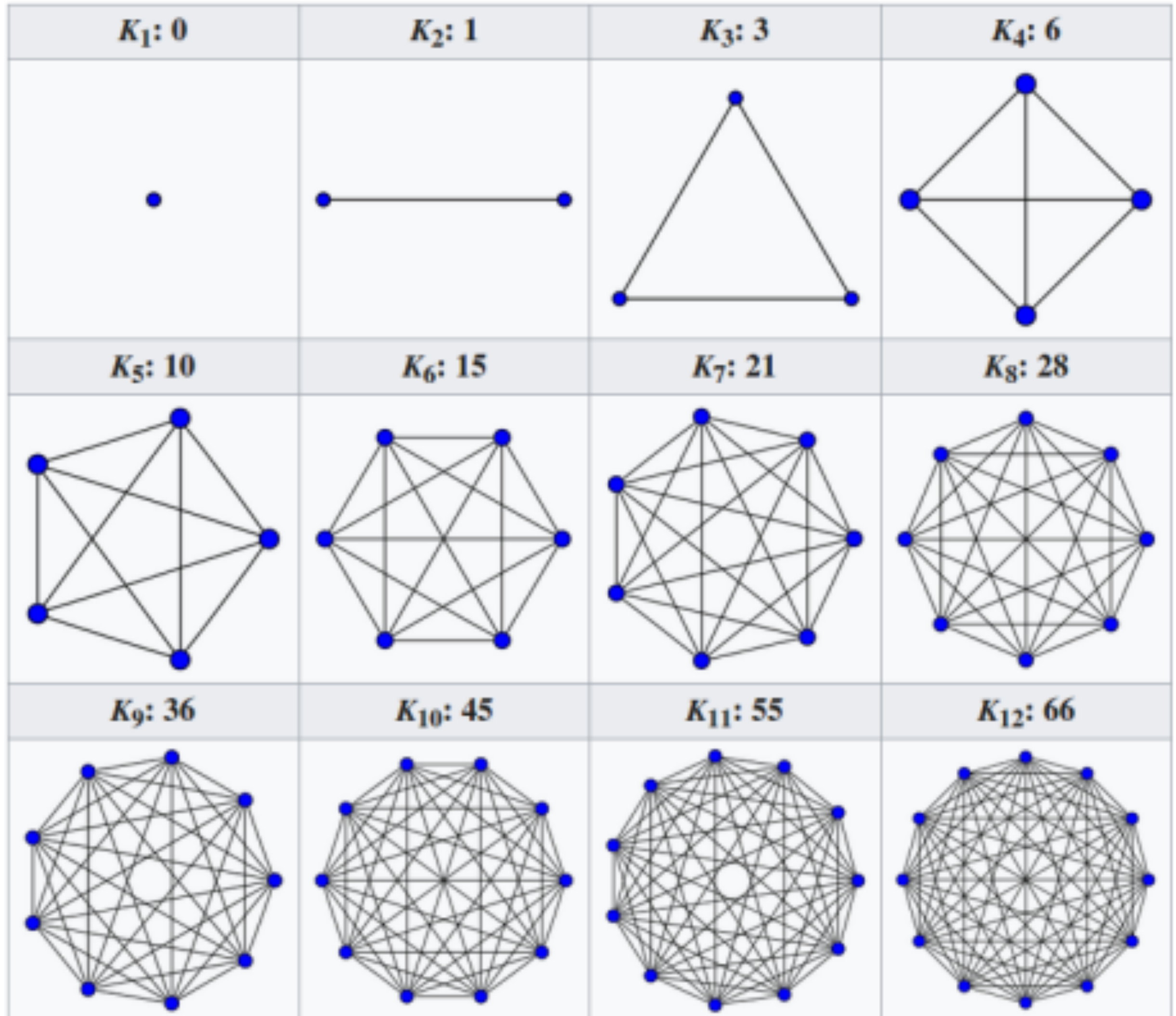
K_7, a complete graph with 7 vertices



A cycle graph of length 6

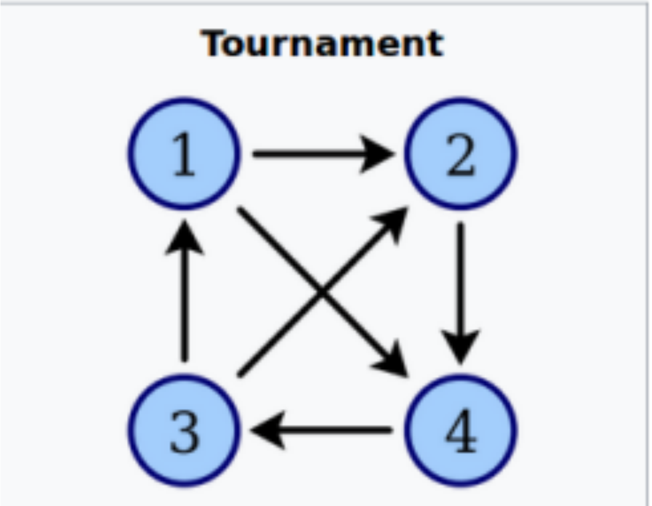
https://en.wikipedia.org/wiki/Complete_graph
https://en.wikipedia.org/wiki/Cycle_graph

Complete Graphs

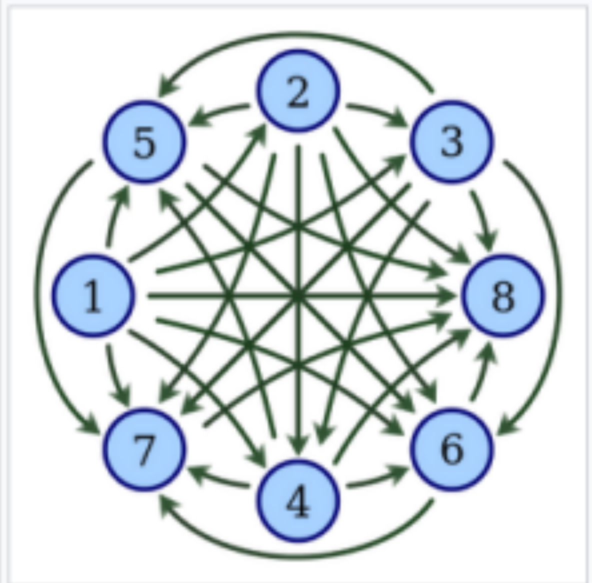


https://en.wikipedia.org/wiki/Complete_graph

Tournament Graphs



A tournament on 4 vertices



A transitive tournament on 8 vertices.

https://en.wikipedia.org/wiki/Tournament_(graph_theory

Platonic Solid Graphs

Tetrahedron	Cube	Octahedron	Dodecahedron	Icosahedron
Four faces	Six faces	Eight faces	Twelve faces	Twenty faces
				
(Animation)	(Animation)	(Animation)	(Animation)	(Animation)
(3D model)	(3D model)	(3D model)	(3D model)	(3D model)

https://en.wikipedia.org/wiki/Platonic_solid

Hamiltonian Cycles – Properties (1)

Any **Hamiltonian cycle** can be converted
to a **Hamiltonian path** by removing one of its edges,

but a **Hamiltonian path** can be extended to
Hamiltonian cycle only if its endpoints are adjacent.

All **Hamiltonian graphs** are **biconnected**, but a
biconnected graph need not be Hamiltonian

https://en.wikipedia.org/wiki/Hamiltonian_path

Biconnected Graph

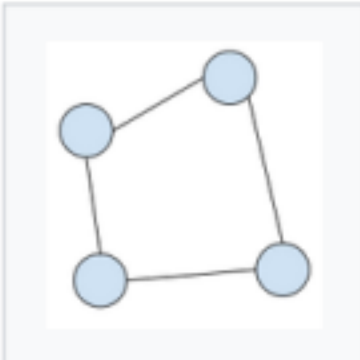
a biconnected graph is a connected and "nonseparable" graph, meaning that if any one **vertex** were to be removed, the graph will remain connected.

a biconnected graph has no articulation vertices.

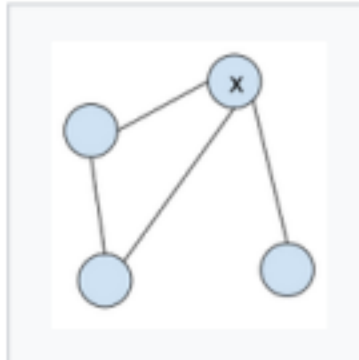
The property of being **2-connected** is equivalent to **biconnectivity**, with the caveat that the complete graph of two vertices is sometimes regarded as biconnected but not 2-connected.

https://en.wikipedia.org/wiki/Biconnected_graph

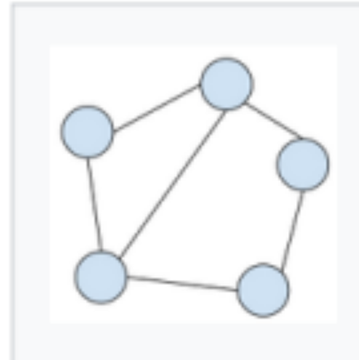
Biconnected Graph Examples



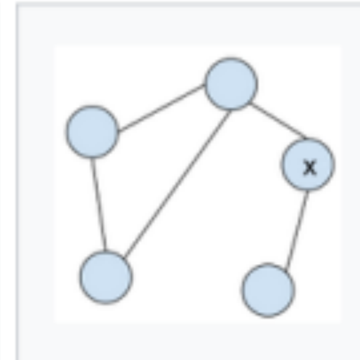
A biconnected graph on four vertices and four edges



A graph that is not biconnected. The removal of vertex x would disconnect the graph.



A biconnected graph on five vertices and six edges



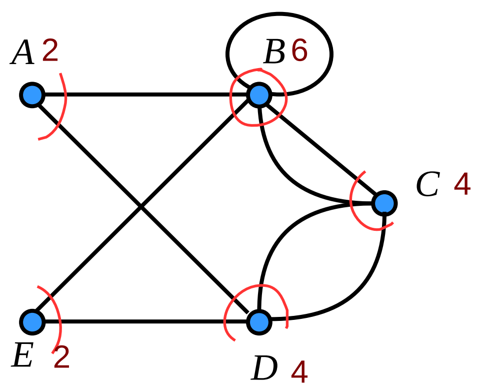
A graph that is not biconnected. The removal of vertex x would disconnect the graph.

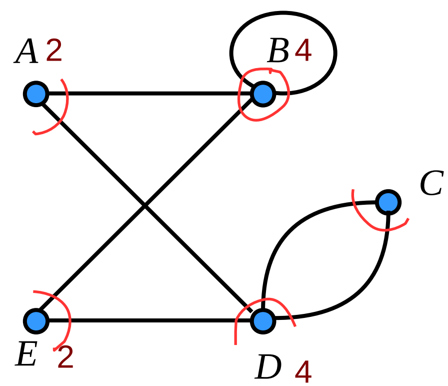
https://en.wikipedia.org/wiki/Biconnected_graph

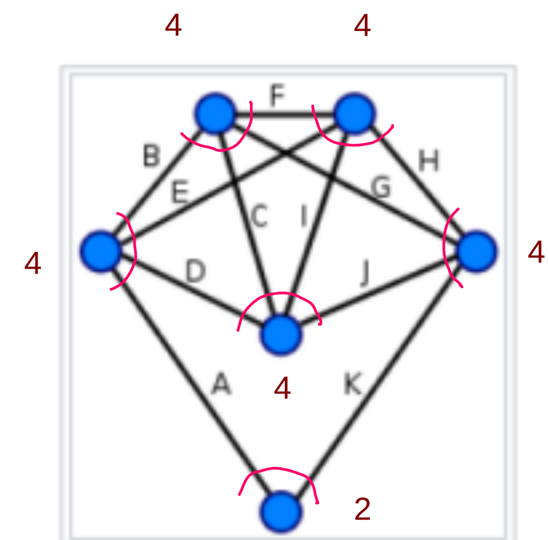
Eulerian Graph

An Eulerian graph G :
a **connected** graph in which
every **vertex** has **even degree**

An **Eulerian graph** G necessarily has an **Euler cycle**,
a closed walk passing through each **edge** of G exactly **once**.



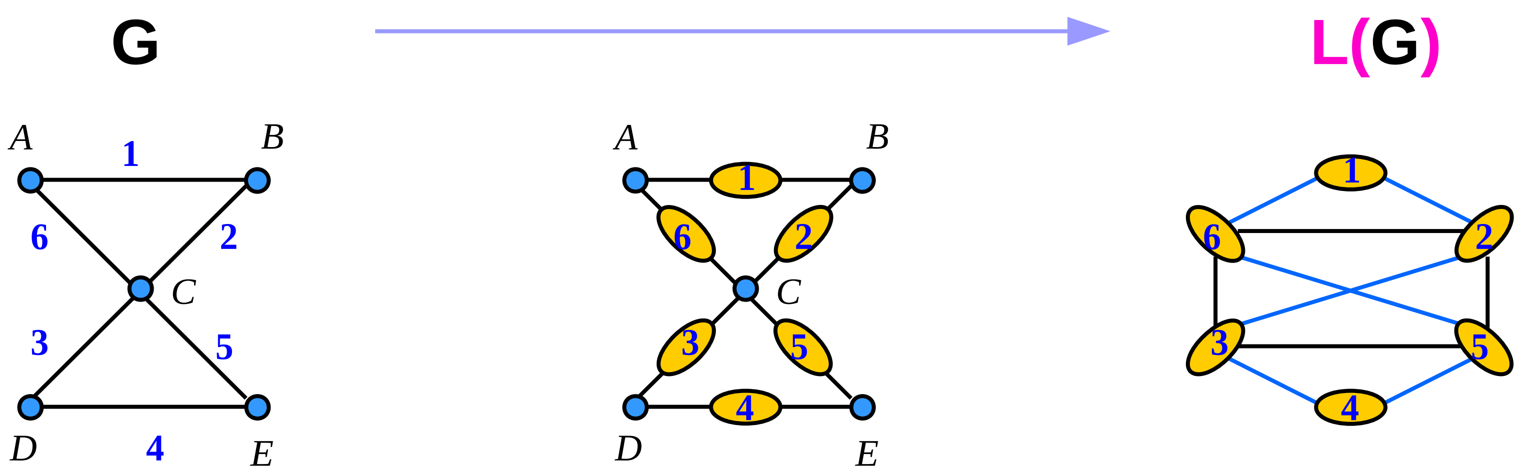




Every vertex of this graph has an even degree. Therefore, this is an Eulerian graph. Following the edges in alphabetical order gives an Eulerian circuit/cycle.

Eulerian Graph (1)

The **Eulerian cycle** corresponds to a **Hamiltonian cycle** in the **line graph L(G)**, so the **line graph** of every **Eulerian graph** is **Hamiltonian graph**.

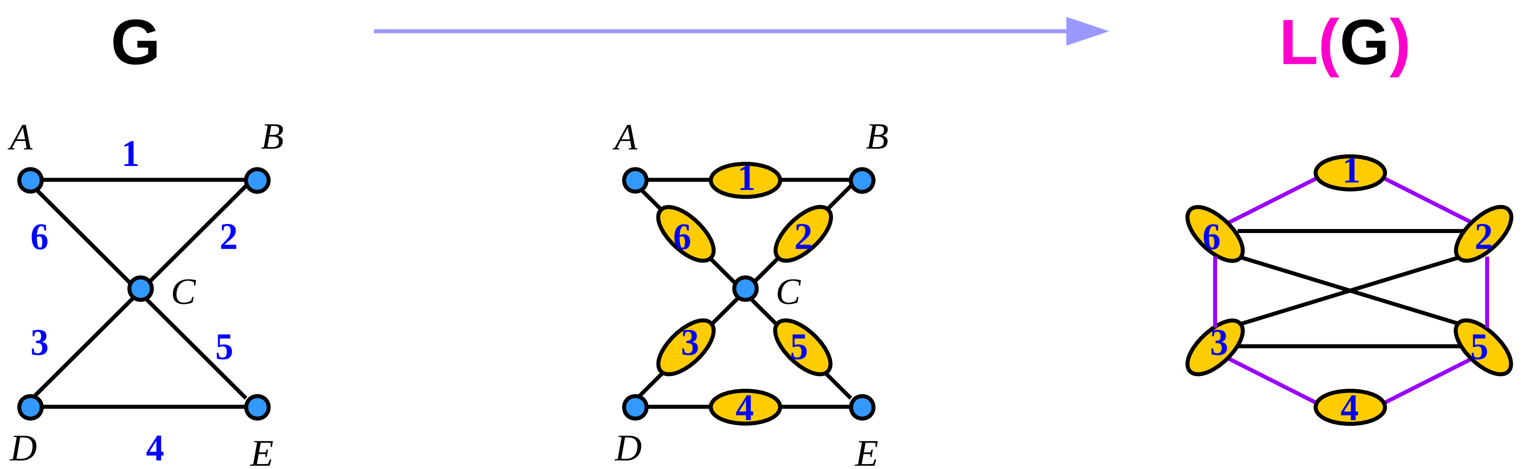


Eulerian Cycle
ABCDECA

Hamiltonian Cycle
1-2-3-4-5-6-1

Eulerian Graph (2)

The **Eulerian cycle** corresponds to a **Hamiltonian cycle** in the **line graph L(G)**, so the **line graph** of every **Eulerian graph** is **Hamiltonian graph**.



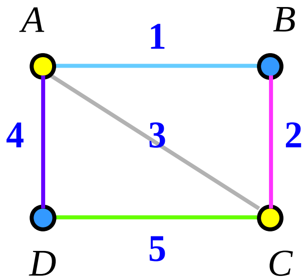
Eulerian Cycle
ABCEDCA

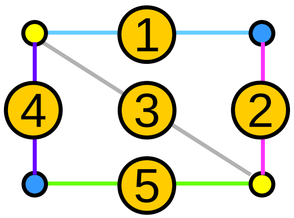
Hamiltonian Cycle
1-2-5-4-3-6-1

Eulerian Path (1)

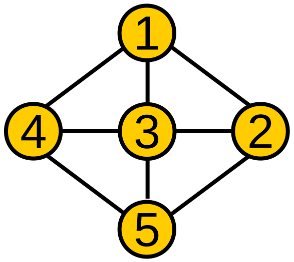
The **Eulerian path** corresponds to a **Hamiltonian path** in the **line graph L(G)**

G





L(G)



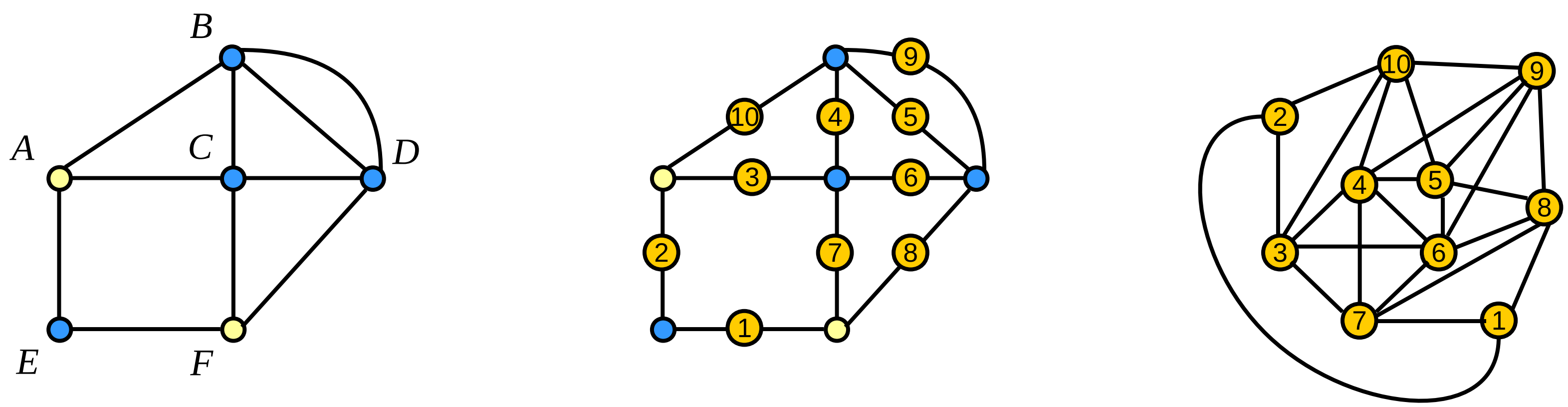
Eulerian Path
ABCADC

Hamiltonian Path
1-2-3-4-5

Eulerian Path (2)

Line graphs may have <u>other</u> **Hamiltonian cycles** that do <u>not</u> correspond to **Euler cycles**.

G → **L(G)**



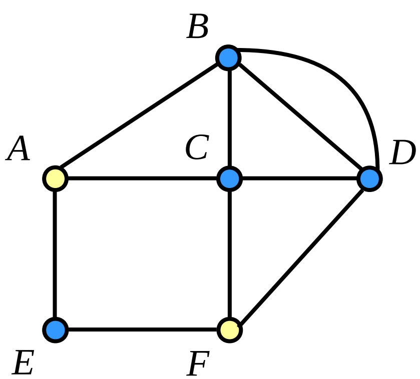
Eulerian Path
FEACBDCFDBA

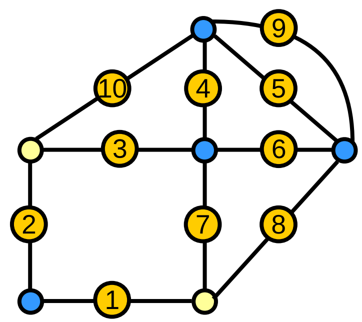
Hamiltonian Path
1-2-3-4-5-6-7-8-9-10

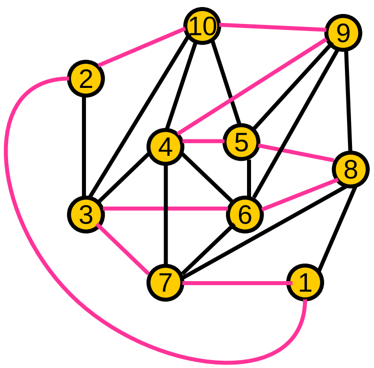
Eulerian Path (3)

Line graphs may have <u>other</u> **Hamiltonian cycles** that do <u>not</u> correspond to **Euler cycles**.

G → **L(G)**







not always

Eulerian Cycle X
Eulerian Path X

Hamiltonian Cycle
1-7-3-6-8-5-4-9-10-2-1

Hamiltonian Cycles – Properties (2)

This **Eulerian cycle** corresponds to a **Hamiltonian cycle** in the **line graph** L(G), so the **line graph** of every **Eulerian graph** is **Hamiltonian graph**.

Line graphs may have other Hamiltonian cycles that do not correspond to Euler paths.

The **line graph** L(G) of every **Hamiltonian graph** G is itself **Hamiltonian**, regardless of whether the graph G is **Eulerian**.

https://en.wikipedia.org/wiki/Hamiltonian_path

Line Graphs

In the mathematical discipline of graph theory, the line graph of an undirected graph G is another graph L(G) that represents the adjacencies between edges of G.

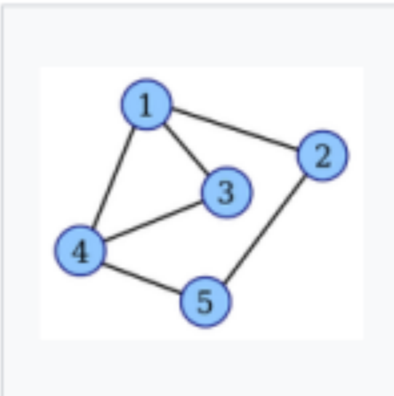
Given a graph G, its line graph L(G) is a graph such that

- each **vertex** of L(G) represents an **edge** of G; and
- two **vertices** of L(G) are **adjacent** if and only if their corresponding **edges** share a **common endpoint** ("are incident") in G.

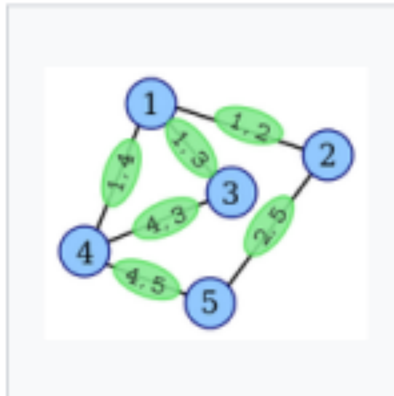
That is, it is the **intersection graph** of the **edges** of G, representing each edge by the set of its two endpoints.

https://en.wikipedia.org/wiki/Line_graph

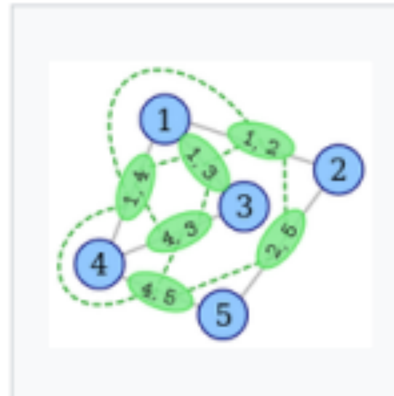
Line Graphs Examples



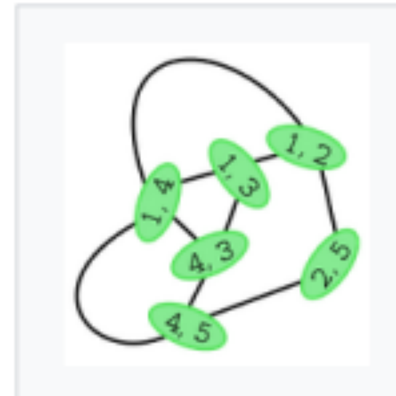
Graph G



Vertices in L(G) constructed from edges in G



Added edges in L(G)



The line graph L(G)

https://en.wikipedia.org/wiki/Line_graph

Hamiltonian Cycles – Properties (3)

A tournament (with more than two vertices) is Hamiltonian if and only if it is **strongly connected**.

The number of different Hamiltonian cycles
in a complete undirected graph on n vertices is $(n - 1)! / 2$
in a complete directed graph on n vertices is $(n - 1)!$.

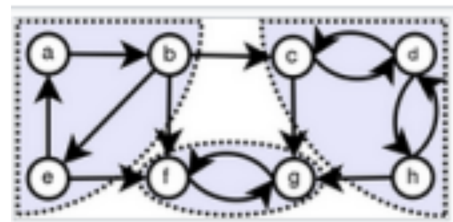
These counts assume that cycles that are the same apart from their starting point are not counted separately.

https://en.wikipedia.org/wiki/Hamiltonian_path

Strongly Connected Component

a directed graph is said to be **strongly connected** or **diconnected** if every **vertex** is reachable from every other **vertex.**

The **strongly connected components** or **diconnected components** of an arbitrary directed graph form a **partition** into **subgraphs** that are themselves **strongly connected**.



Graph with strongly connected components marked

https://en.wikipedia.org/wiki/Hamiltonian_path

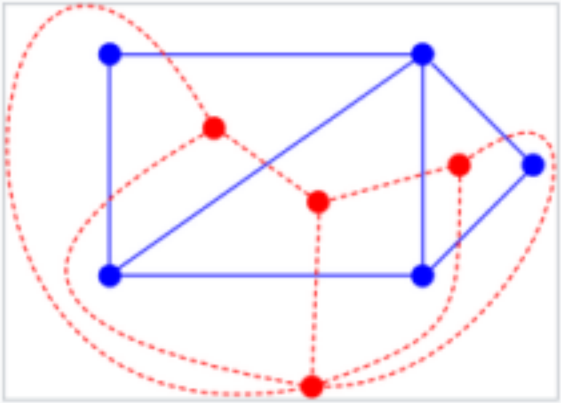
Dual Graph

the dual graph of a plane graph G is a graph that has a **vertex** for each **face** of G.

The dual graph has an **edge** whenever two **faces** of G are separated from each other by an **edge**,

and a **self-loop** when the same **face** appears on both sides of an **edge**.

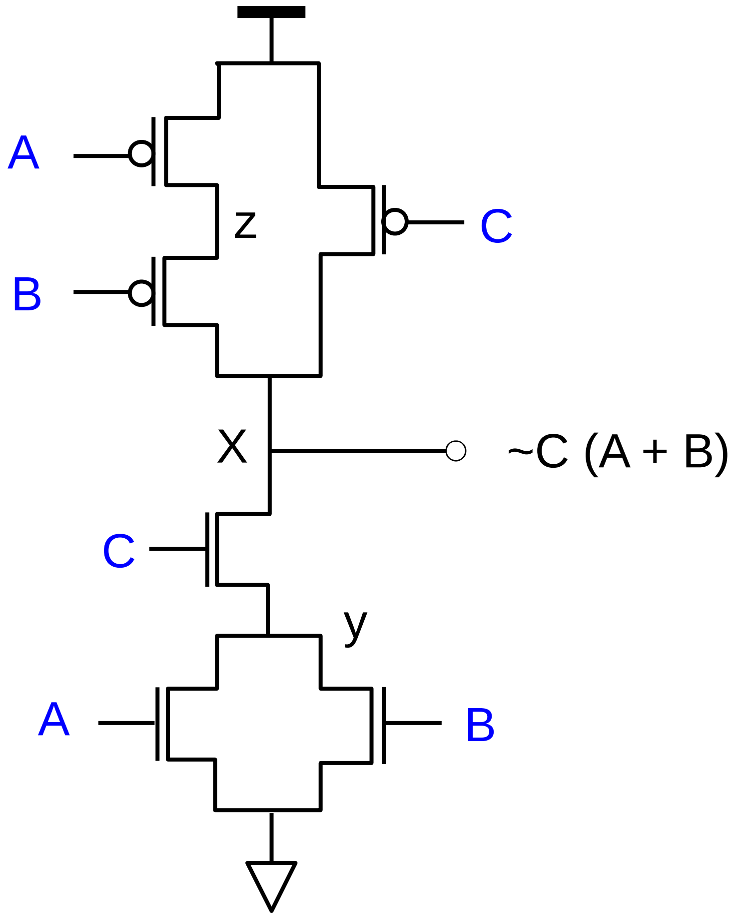
each **edge e** of G has a corresponding **dual edge**, whose endpoints are the **dual vertices** corresponding to the **faces** on either side of **e**.

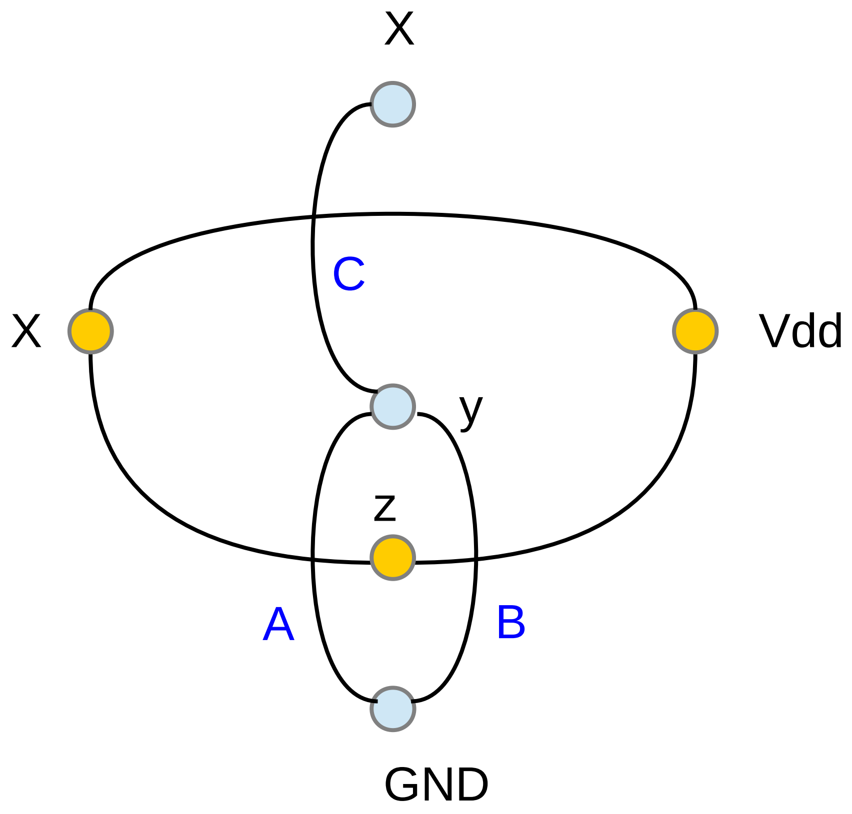


The red graph is the dual graph of the blue graph, and *vice versa*.

https://en.wikipedia.org/wiki/Hamiltonian_path

Dual Graph





https://en.wikipedia.org/wiki/Hamiltonian_path

References

[1] http://en.wikipedia.org/
[2]

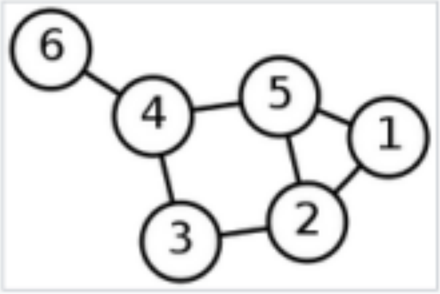
Shortest Path Problem (4A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

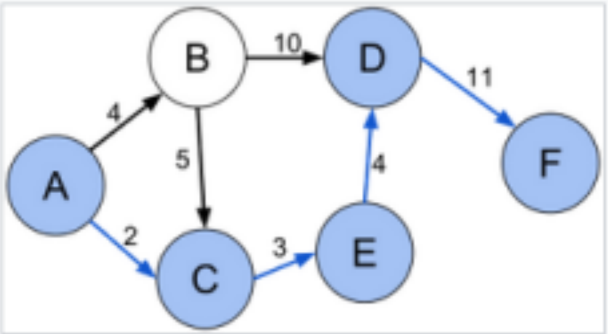
This document was produced by using LibreOffice and Octave.

Shortest Path Problem

the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized.



(6, 4, 5, 1) and (6, 4, 3, 2, 1) are both paths between vertices 6 and 1



Shortest path (A, C, E, D, F) between vertices A and F in the weighted directed graph

https://en.wikipedia.org/wiki/Shortest_path_problem

Types of Shortest Path Problems

The **single-pair shortest path problem:**
to find shortest paths from a **source** vertex v to a **destination** vertex w in a graph

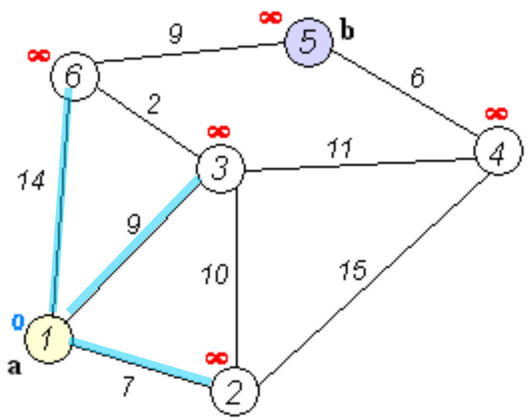
The **single-<u>source</u> shortest path problem:**
to find shortest paths from a **source** vertex v to **all** other vertices in the graph.

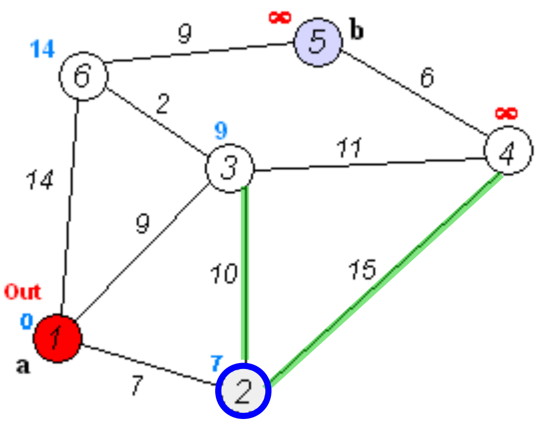
The **single-<u>destination</u> shortest path problem:**
to find shortest paths from **all** vertices in the directed graph to a single **destination** vertex v. This can be reduced to the single-source shortest path problem by reversing the arcs in the directed graph.

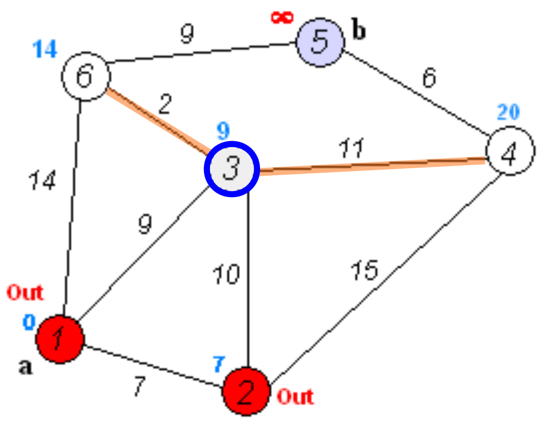
The **all-pairs shortest path problem:**
to find shortest paths between every **pair** of vertices v, v' in the graph.

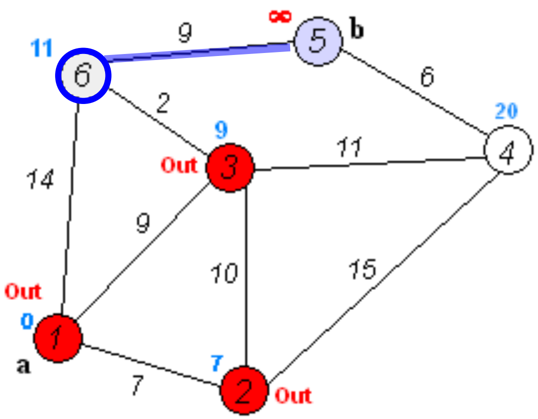
https://en.wikipedia.org/wiki/Shortest_path_problem

Dijkstra’s Algorithm Example Summary









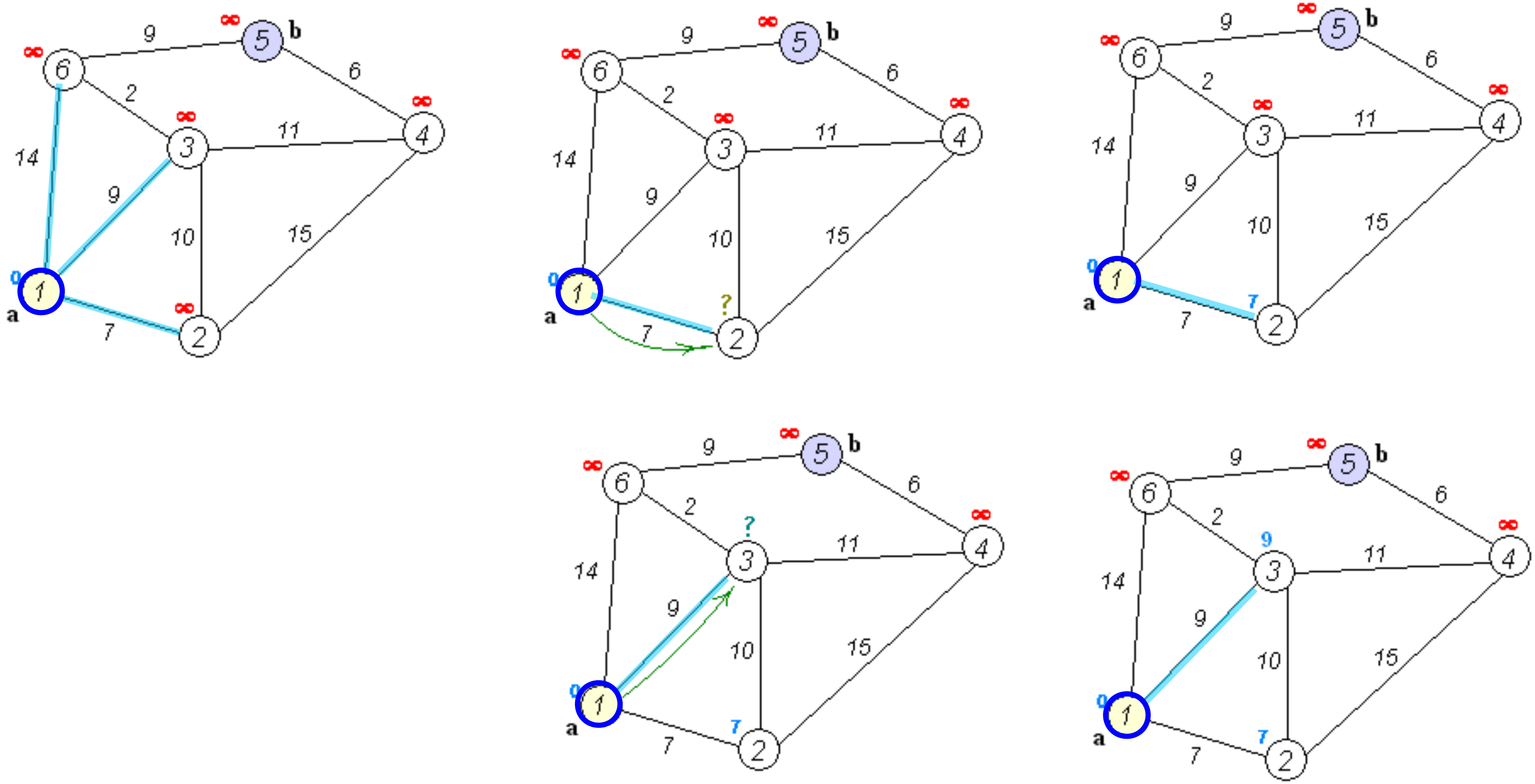
the initial node

the current node

the visited nodes

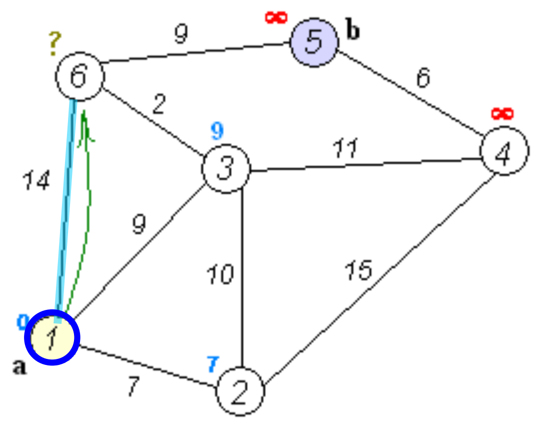
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

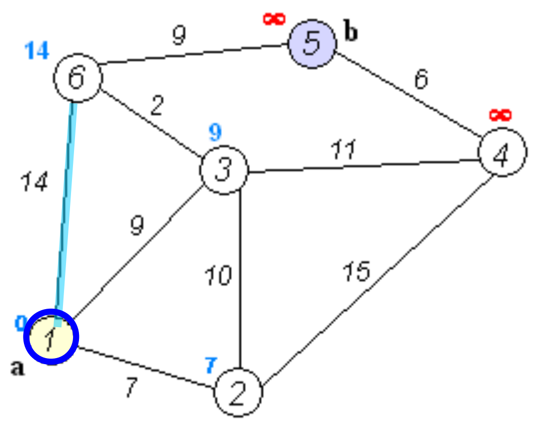
Dijkstra’s Algorithm Example (1)

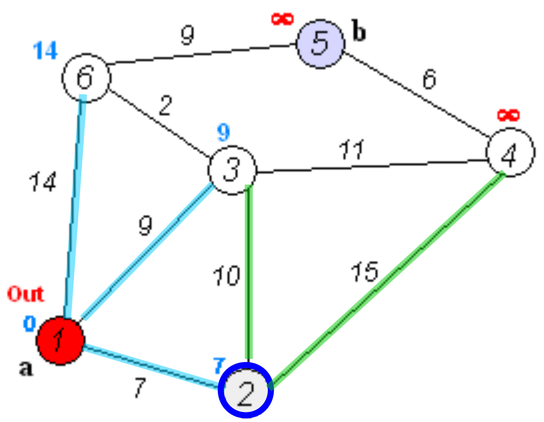


https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra's Algorithm Example (2)

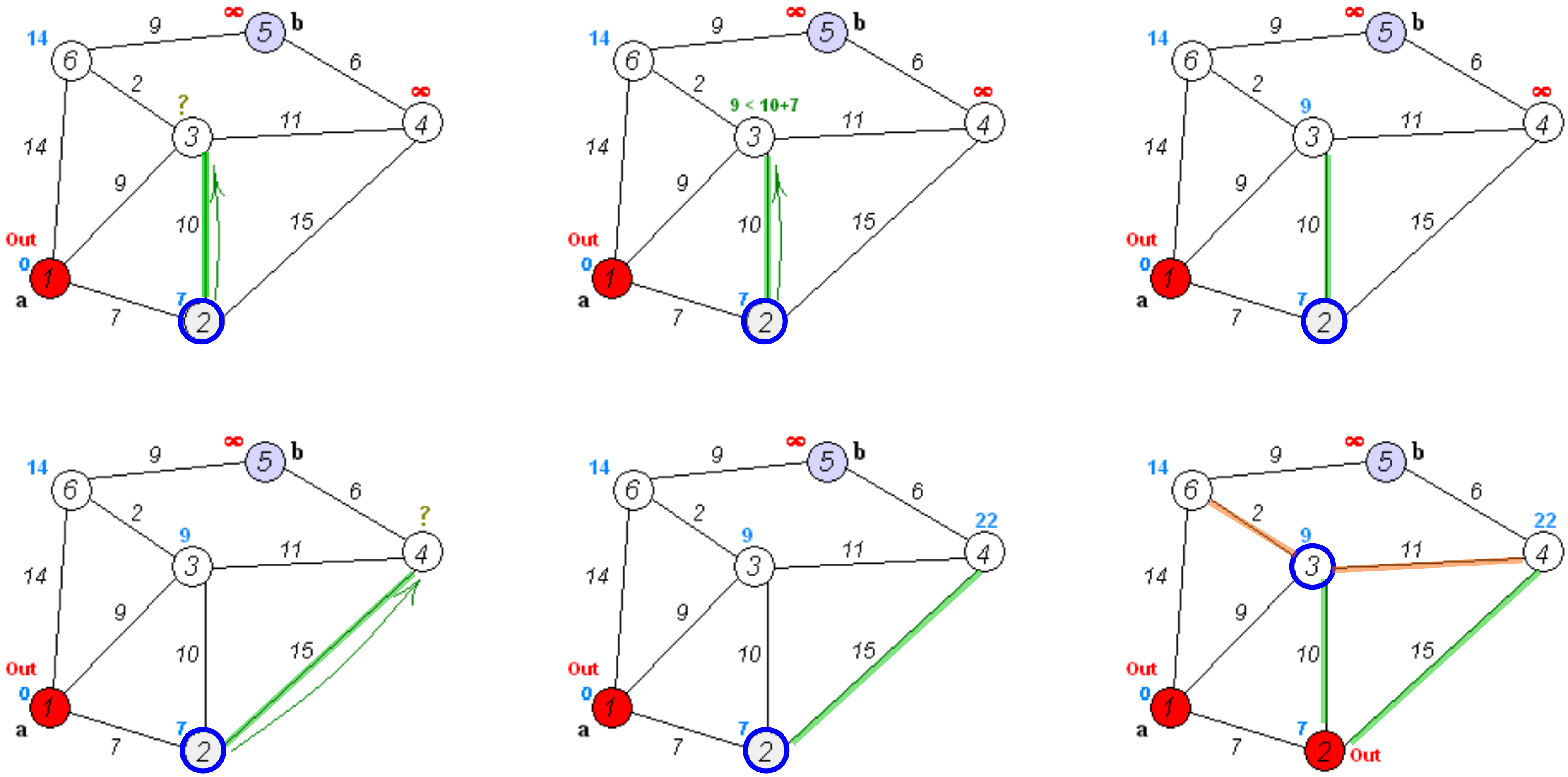






https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

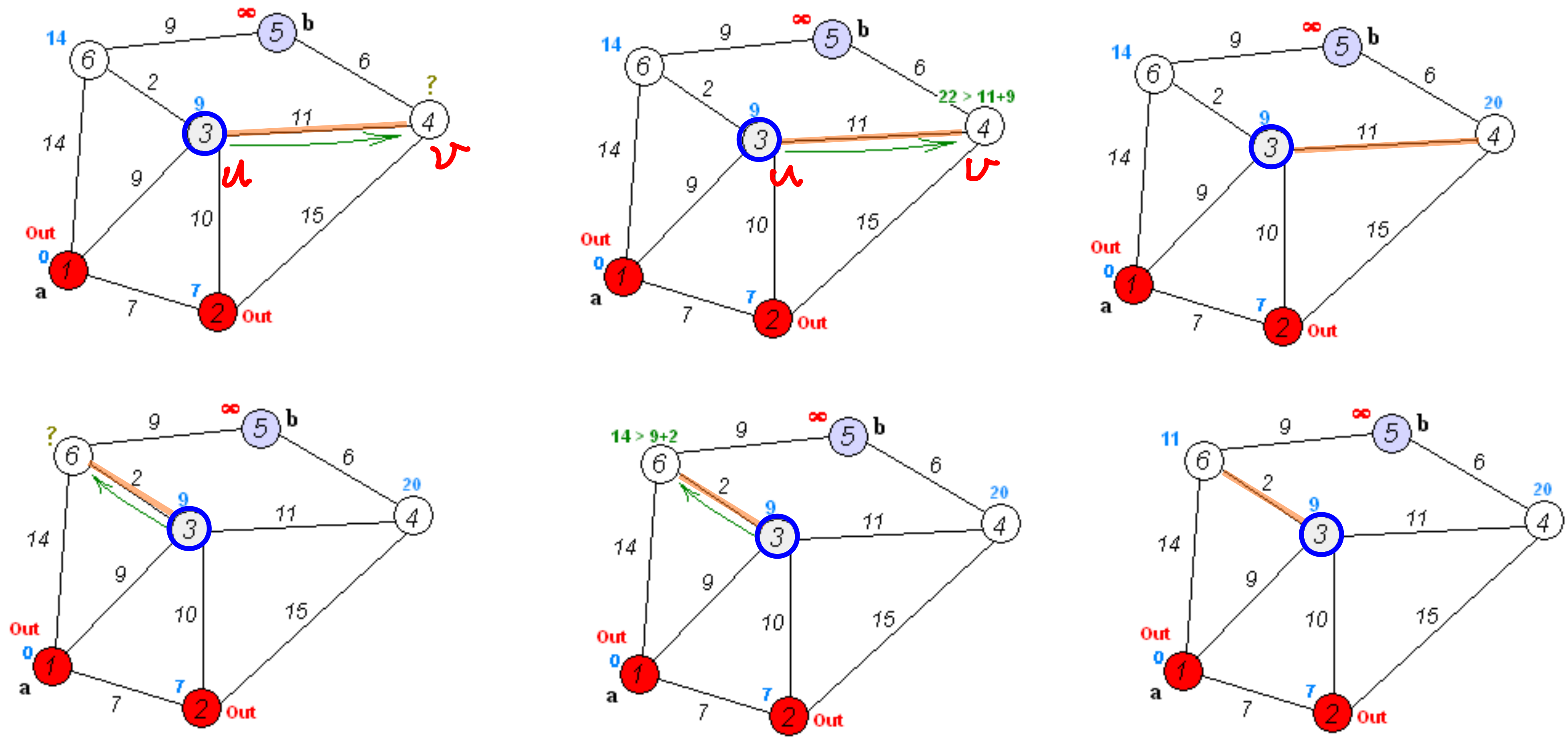
Dijkstra’s Algorithm Example (3)



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

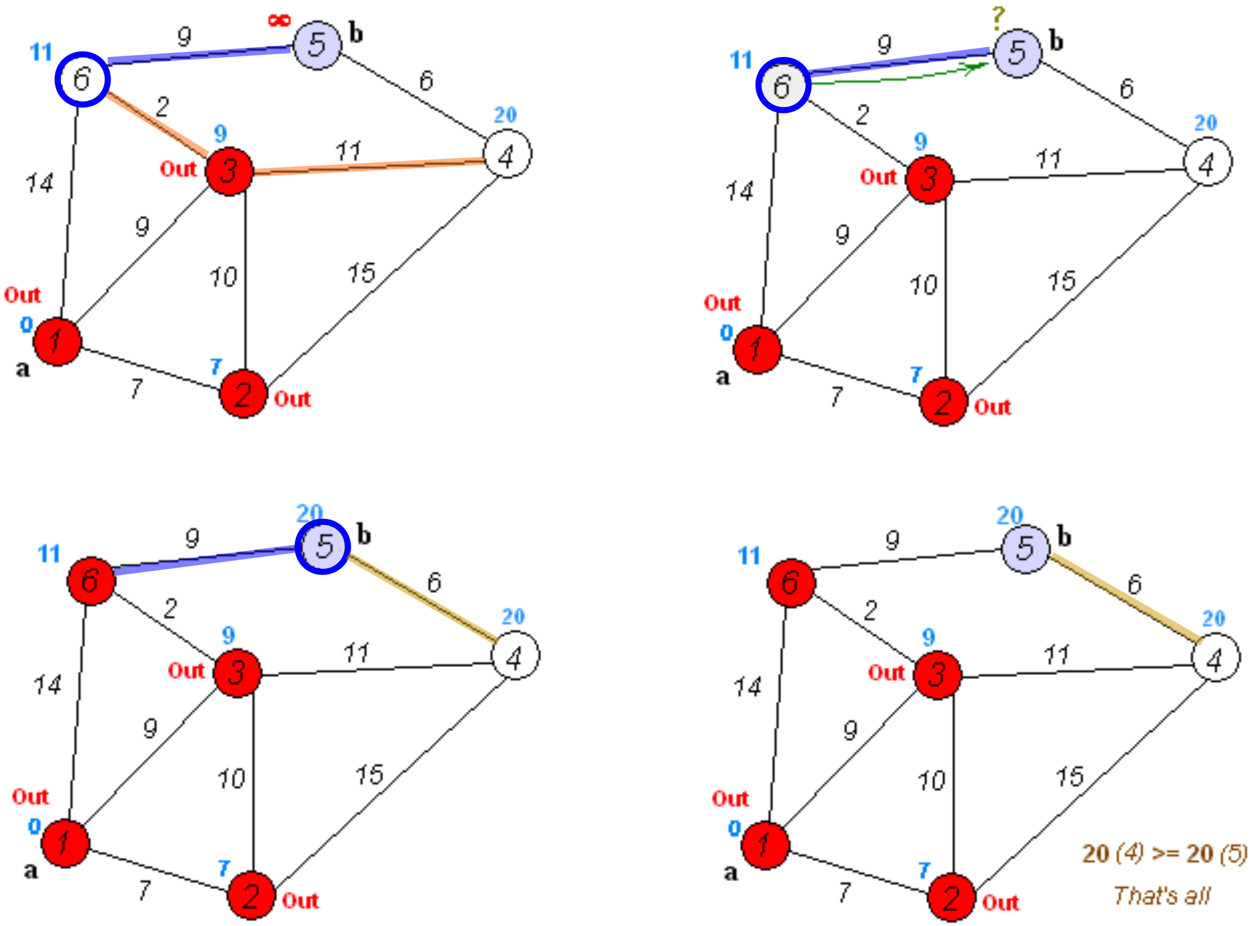
Dijkstra's Algorithm Example (4)

$L(u)=9 \quad w(u.v)=11$ $\quad$ $L(u)+w(u,v) = 20 < L(v)=22$



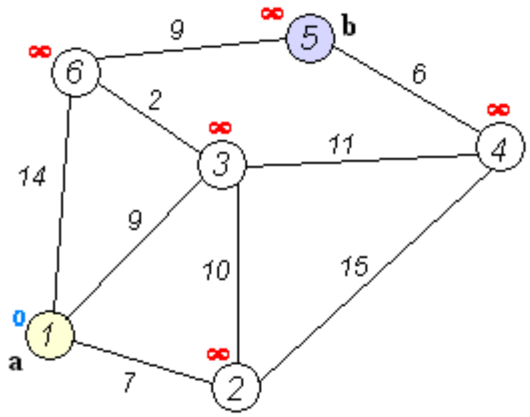
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra’s Algorithm Example (5)



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Hamiltonian Cycles



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra’s Algorithm (1)

Let the node at which we are starting
be called the **initial node**.
Let the **distance** of node Y be
the **distance** from the **initial node** to Y.
Dijkstra's algorithm will assign some **initial distance values** and will try to improve them step by step.

1. Mark all nodes **unvisited**.
Create a set of all the unvisited nodes called the **unvisited set**.

2. Assign to every node a **tentative distance value**:
set it to **zero** for our initial node and
to **infinity** for all other nodes.
Set the **initial node** as **current**.

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

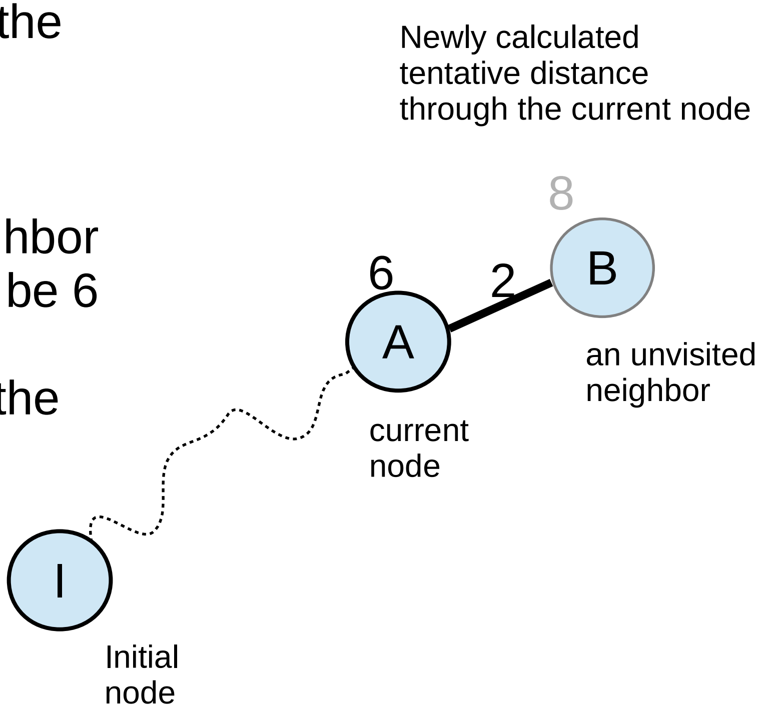
Dijkstra's Algorithm (2)

3. Remove the **current node** from the **unvisited set**

For all the **unvisited neighbors** of the **current node**, calculate their **tentative distances** through the **current** node.

Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.

For example, if the current node A is marked with a distance of 6, and the edge connecting it with a neighbor B has length 2, then the distance to B through A will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.



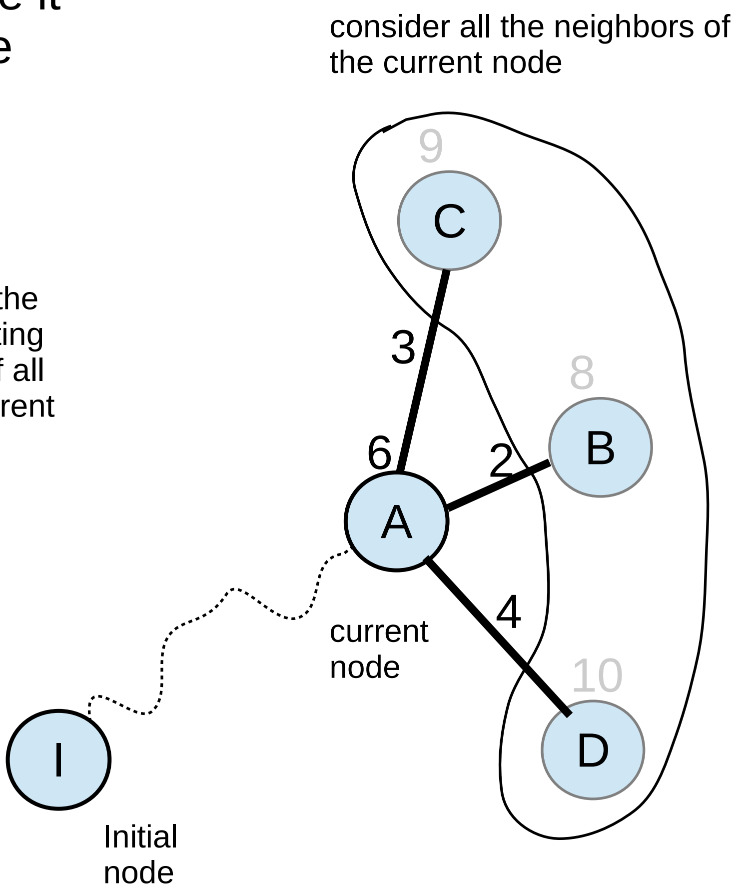
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra's Algorithm (3)

4. After considering <u>all</u> of the **neighbors** of the **current node**, mark the **current** node as **visited** and remove it from the **unvisited set**. A **visited node** will never be checked again.

current node : chosen node with the <u>smallest</u> tentative distance from the **unvisited set**

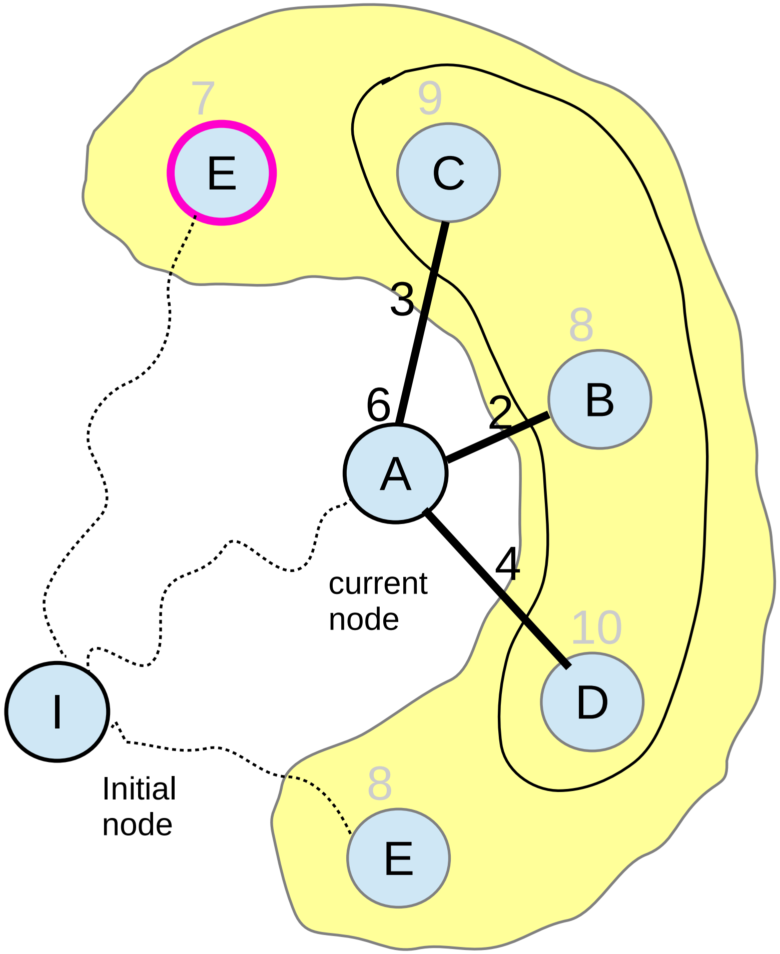
current node : move to the **visited** set, after calculating the tentative distances of all the **neighbors** of the current node



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra's Algorithm (4)

5. Move to the **next unvisited node** with the smallest tentative distances and repeat the above steps which check neighbors and mark visited.



https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra’s Algorithm (5)

5-a. If the **destination** node has been marked **visited** (when planning a route between two specific nodes)

or if the smallest tentative distance among the nodes in the unvisited set is **infinity** (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes),

then stop. The algorithm has finished.

5-b. Otherwise, select the **unvisited** node that is marked with the smallest tentative distance,
set it as the new **current node**, and go back to step 3.

https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra’s Algorithm – Pseudocode 1

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:              // Initialization
        dist[v] ← INFINITY                   // Unknown distance from source to v
        prev[v] ← UNDEFINED                  // Previous node in optimal path from source
        add v to Q                           // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                         // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]     // Node with the least distance
                                             // will be selected first
        remove u from Q

        for each neighbor v of u:            // where v is still in Q.        for each v in Q:
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:                // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

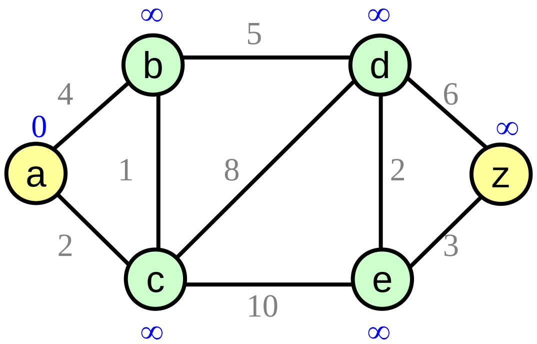
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#/media/File:Dijkstra_Animation.gif

Dijkstra's Algorithm – Pseudocode 2

Procedure Dijkstra(**G**: weighted connected simple graph, with all positive weights)
{**G** has vertices $a = v_0, v_1, \ldots, v_n = z$ and length $w(v_i, v_j)$
where $w(v_i, v_j) = \infty$ if $\{v_i, v_j\}$ is not an edge in **G**}
for i := 1 to n
    L(v_i) := ∞
L(a) := 0
S := { }
{the labels are now initialized so that the label of a is 0 and
All other labels are ∞, and S is the empty set}
while z ∉ S
    u := a vertex not in S with L(u) minimal
    S := S ∪ {u}
    for all vertices v not in S
        if L(u) +w(u,v) < L(u) then L(v) := L(u) + w(u,v)
        {this adds a vertex to S with minimal label and
        updates the labels of vertices not in S}
return L(z) {L(z) = length of a shortest path from a to z}

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (0)



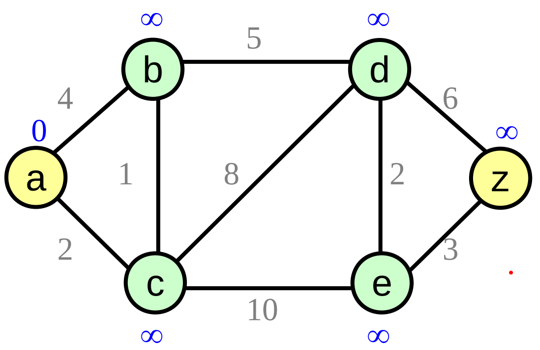
	a	*b*	*c*	*d*	*e*	*z*
a	∞	4	2	∞	∞	∞
b	4	∞	1	5	∞	∞
c	2	1	∞	8	10	∞
d	∞	5	8	∞	2	6
e	∞	∞	10	8	∞	3
z	∞	∞	∞	6	3	∞

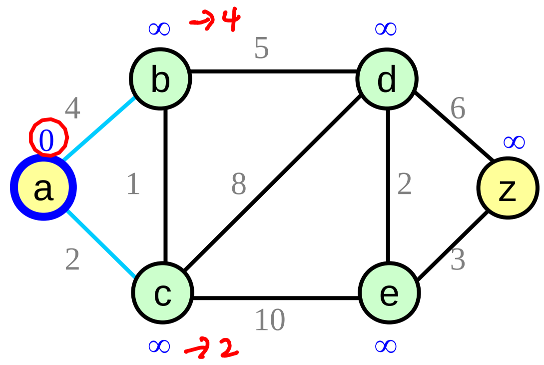
∞ for no direct connection

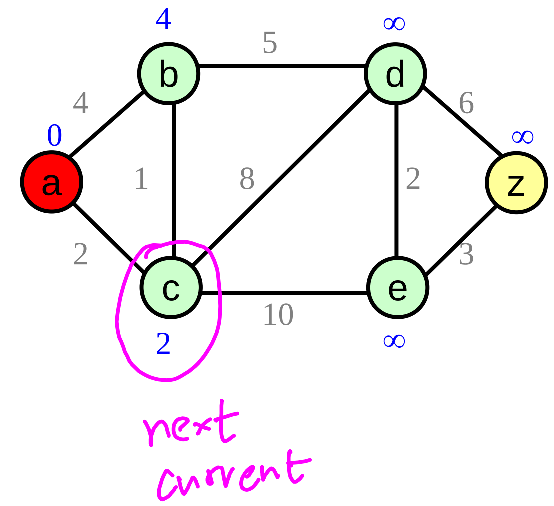
$$w(u_i, u_j)$$

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (1)







$S = \{a\}$

$L(a) + w(a,b) = 0 + 4 \;<\; L(b) = \infty$

$L(a) + w(a,c) = 0 + 2 \;<\; L(c) = \infty$

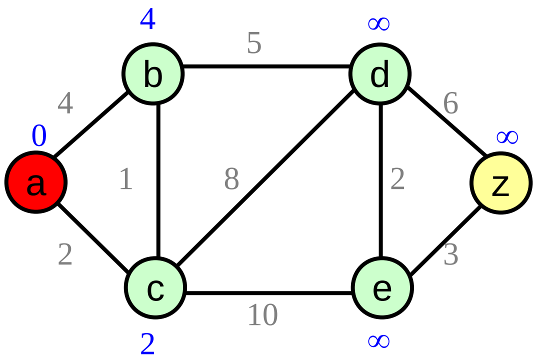
$L(a) + w(a,d) = 0 + \infty \;=\; L(d) = \infty$

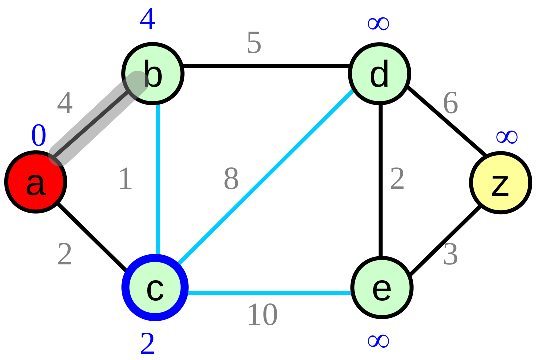
$L(a) + w(a,e) = 0 + \infty \;=\; L(e) = \infty$

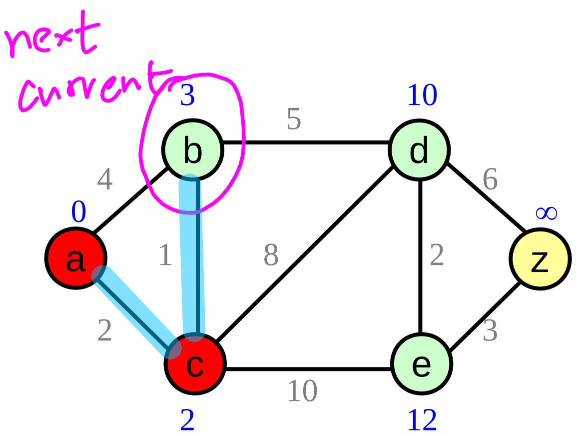
$L(a) + w(a,z) = 0 + \infty \;=\; L(z) = \infty$

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (2)







$S = \{a, c\}$

$L(c) + w(c, b) = 2 + 1 < L(b) = 4$ $\qquad P(a, c, b) < P(a, b)$

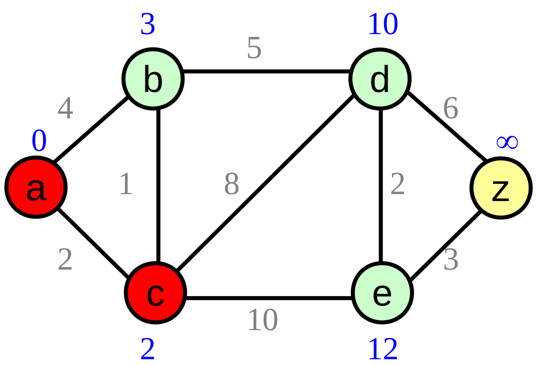
$L(c) + w(c, d) = 2 + 8 < L(d) = \infty$

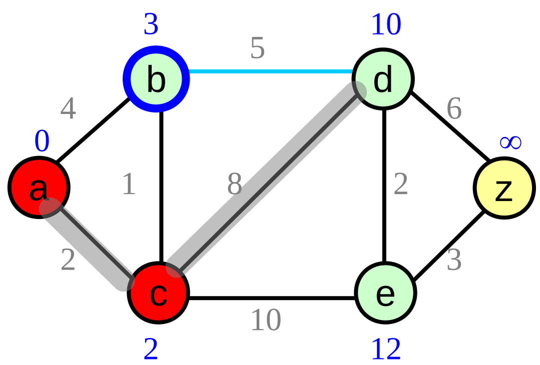
$L(c) + w(c, e) = 2 + 10 < L(e) = \infty$

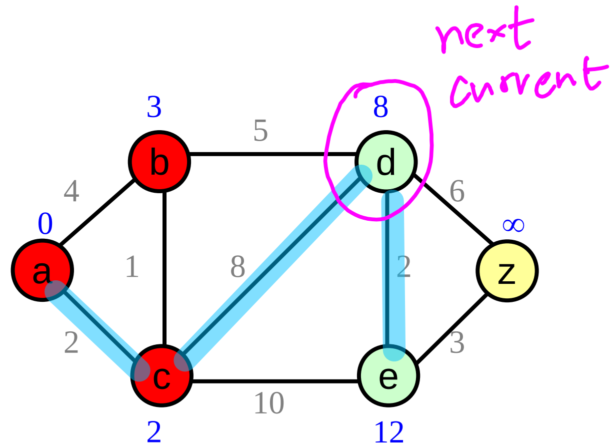
$L(c) + w(c, z) = 2 + \infty = L(z) = \infty$

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (3)







$S = \{a, c, b\}$

$L(b) + w(b, d) = 3 + 5 < L(d) = 10$

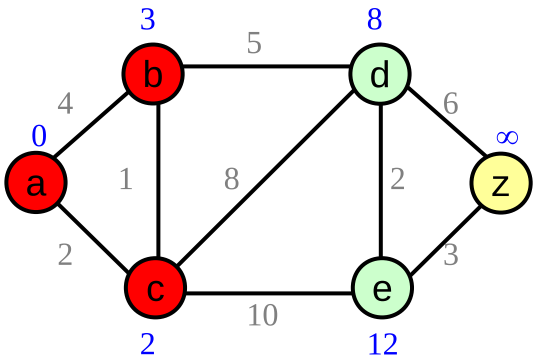
$L(b) + w(b, e) = 3 + \infty > L(e) = 12$

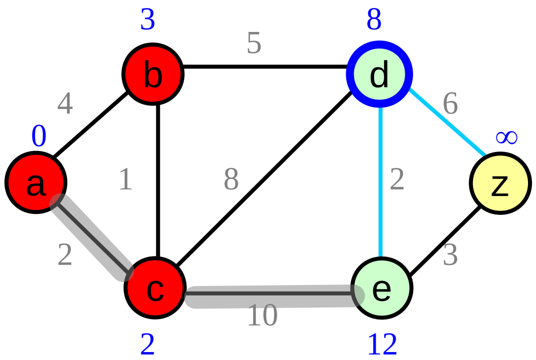
$L(b) + w(b, z) = 3 + \infty = L(z) = \infty$

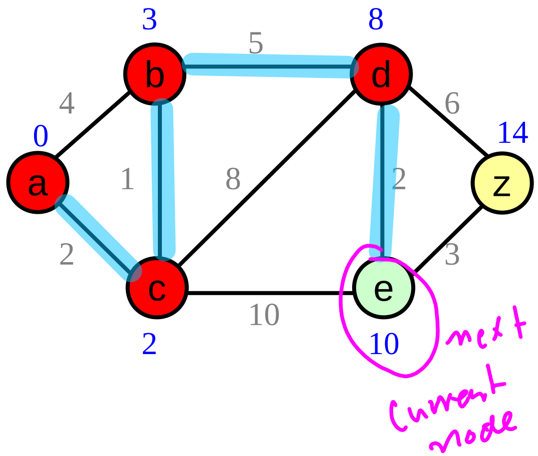
$P(a, c, b, d) < P(a, c, d)$

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (4)







$S = \{a, c, b, d\}$

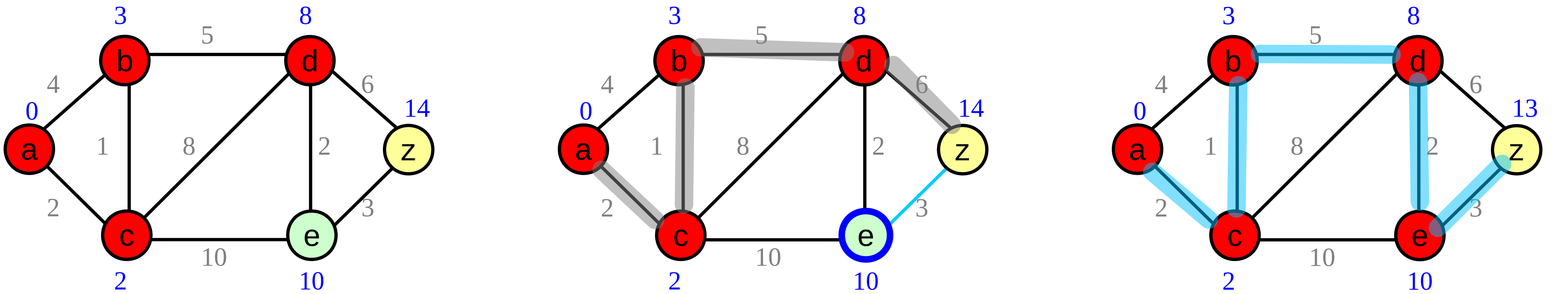
$L(d) + w(d, e) = 8 + 2 < L(e) = 12$

$L(d) + w(d, z) = 8 + 6 < L(z) = \infty$

$P(a, c, b, d, e) < P(a, c, e)$

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (5)

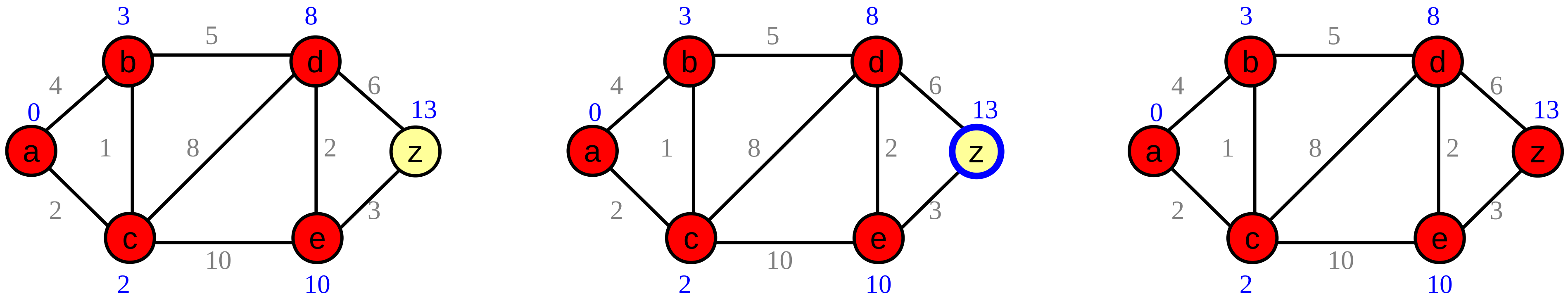


$S = \{a, c, b, d, e\}$

$L(e) + w(e, z) = 10 + 3 < L(z) = 14$ $\quad P(a, c, b, d, e, z) < P(a, c, b, d, z)$

Discrete Mathematics and It's Applications, K. H. Rosen

Dijkstra Algorithm Pseudocode 2 Example (6)



$S = \{a, c, b, d, e, z\}$

Discrete Mathematics and It's Applications, K. H. Rosen

References

[1] http://en.wikipedia.org/
[2]

Minimum Spanning Tree (5A)

Minimum　최소　Σweight : min

Spanning　신장　all veritices

Tree　트리　— cycle X

① Borůvka

② Kruskal

③ Prim

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

Minimum Spanning Tree

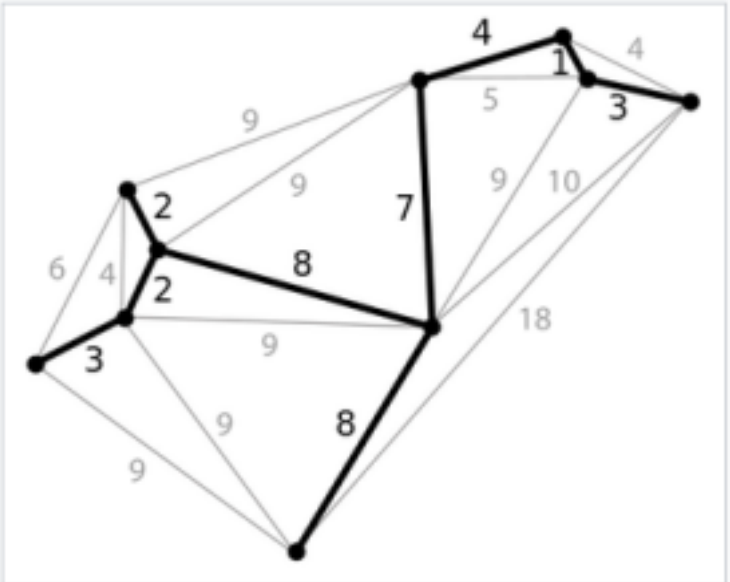
a **subset** of the **edges** of a connected, edge-weighted (un)directed graph that connects **all** the **vertices** together, without any **cycles** and with the **minimum** possible total edge **weight**.

a spanning tree whose sum of edge weights is as small as possible.

More generally, any edge-weighted undirected graph (not necessarily connected) has a minimum spanning **forest**, which is a **union** of the minimum spanning **trees** for its connected components.

https://en.wikipedia.org/wiki/Minimum_spanning_tree

Types of Shortest Path Problems



A planar graph and its minimum spanning tree. Each edge is labeled with its weight, which here is roughly proportional to its length.

https://en.wikipedia.org/wiki/Minimum_spanning_tree

Properties (1)

Possible multiplicity
If there are **n vertices** in the graph,
then each spanning tree has **n−1 edges**.

Uniquenss
If each edge has a distinct weight
then there will be only one, unique minimum spanning tree.
this is true in many realistic situations

Minimum-cost subgraph
If the weights are positive, then a minimum spanning tree is in fact a minimum-cost subgraph connecting **all vertices**, since subgraphs containing cycles necessarily have more total weight.

https://en.wikipedia.org/wiki/Minimum_spanning_tree

Properties (2)

Cycle Property

For any **cycle C** in the graph, if the weight of an **edge e** of **C** is larger than the individual weights of all other **edges** of **C**, then this edge cannot belong to a MST.

Cut property

For any **cut C** of the graph, if the weight of an **edge e** in the **cut-set** of **C** is strictly smaller than the weights of all other edges of the **cut-set** of **C**, then this edge belongs to all MSTs of the graph.

https://en.wikipedia.org/wiki/Minimum_spanning_tree

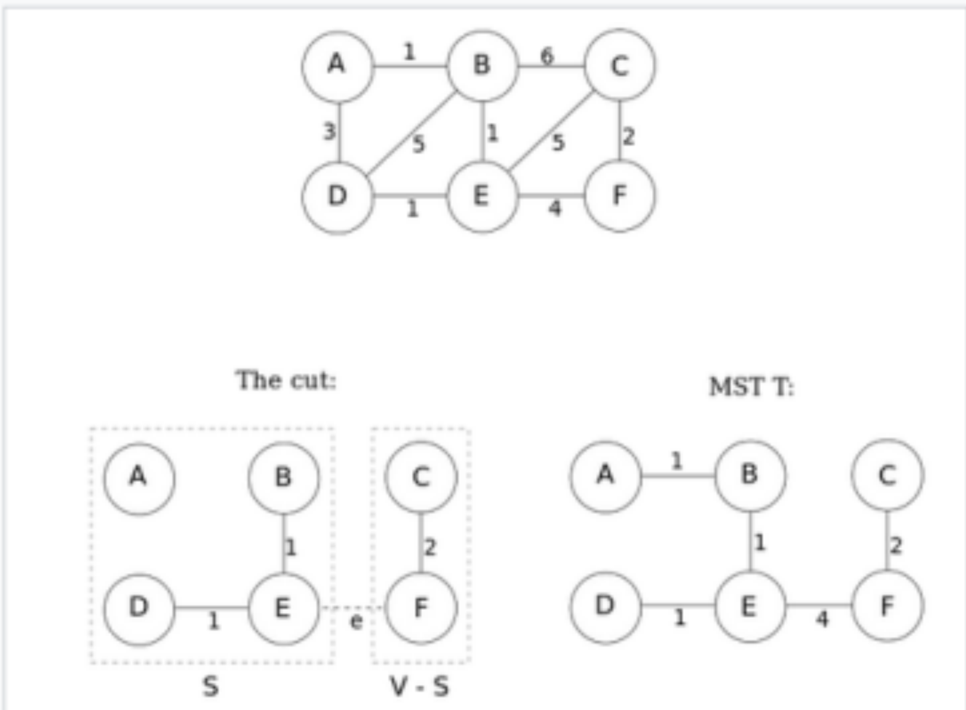
Properties (3)

Minimum-cost edge
If the minimum cost **edge e** of a graph is unique, then this edge is included in any MST.

Contraction
If **T** is a **tree** of **MST edges**, then we can contract **T** into a single vertex while maintaining the invariant that the MST of the contracted graph plus T gives the MST for the graph before contraction.

https://en.wikipedia.org/wiki/Minimum_spanning_tree

Cut property examples



This figure shows the cut property of MSTs. T is the only MST of the given graph. If S = {A,B,D,E}, thus V-S = {C,F}, then there are 3 possibilities of the edge across the cut(S,V-S), they are edges BC, EC, EF of the original graph. Then, e is one of the minimum-weight-edge for the cut, therefore S ∪ {e} is part of the MST T.

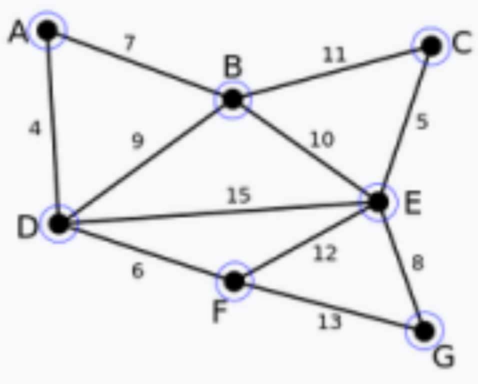
https://en.wikipedia.org/wiki/Minimum_spanning_tree

Borůvka's algorithm

Input: A graph G whose edges have distinct weights
Initialize a forest **F** to be a set of one-vertex trees,
one for each vertex of the graph.
While F has more than one component:
Find the connected components of F and
label each vertex of G by its component
Initialize the cheapest edge for each component to "None"
For each edge **uv** of **G**:
If u and **v** have different component labels:
If uv is <u>cheaper</u> than the <u>cheapest</u> edge
for the component of **u**:
Set **uv** as the <u>cheapest</u> edge for the component of **u**
If uv is <u>cheaper</u> than the <u>cheapest</u> edge
for the component of **v**:
Set **uv** as the <u>cheapest</u> edge for the component of **v**
For each component whose <u>cheapest</u> edge
is not "None":
Add its <u>cheapest</u> edge to **F**
Output: **F** is the minimum spanning forest of **G**.

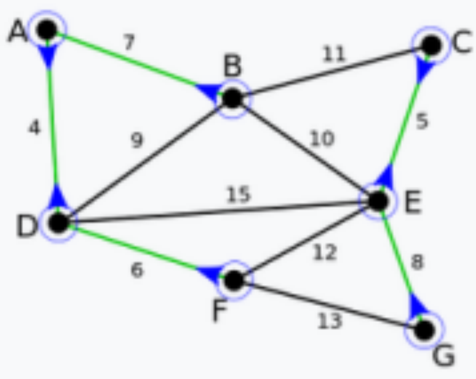
https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Borůvka's algorithm examples (1)

Image	components	Description
	{A} {B} {C} {D} {E} {F} {G}	This is our original weighted graph. The numbers near the edges indicate their weight. Initially, every vertex by itself is a component (blue circles).

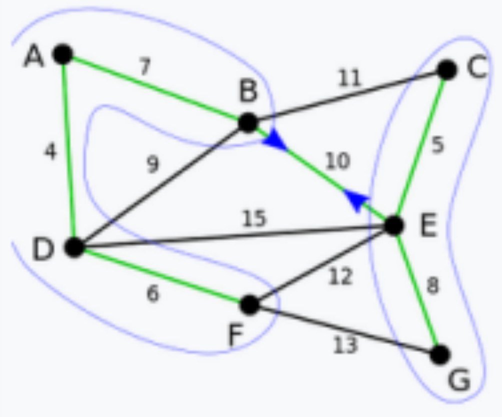
https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Borůvka's algorithm examples (2)

	{A,B,D,F} {C,E,G}	In the first iteration of the outer loop, the minimum weight edge out of every component is added. Some edges are selected twice (AD, CE). Two components remain.

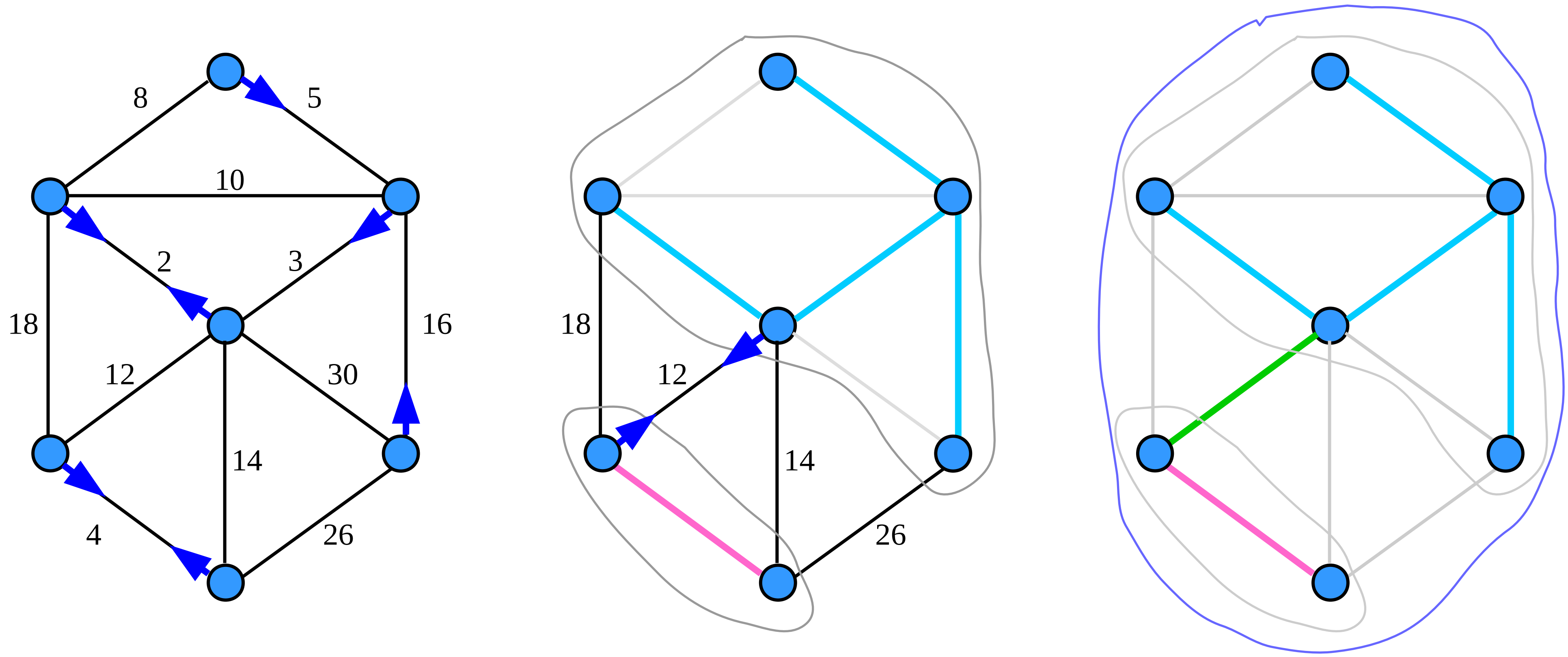
https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Borůvka's algorithm examples (3)

	{A,B,C,D,E,F,G}	In the second and final iteration, the minimum weight edge out of each of the two remaining components is added. These happen to be the same edge. One component remains and we are done. The edge BD is not considered because both endpoints are in the same component.

https://en.wikipedia.org/wiki/Bor%C5%AFvka%27s_algorithm

Borůvka's algorithm examples (4)



http://jeffe.cs.illinois.edu/teaching/algorithms/notes/20-mst.pdf

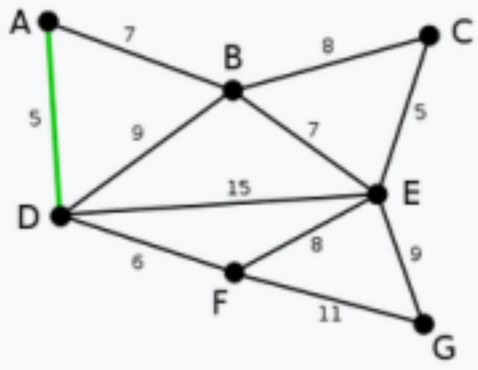
Kruskal's algorithm

```
KRUSKAL(G):
A = ∅
foreach v ∈ G.V:
   MAKE-SET(v)
foreach (u, v) in G.E ordered by weight(u, v), increasing:
   if FIND-SET(u) ≠ FIND-SET(v):
      A = A ∪ {(u, v)}
      UNION(u, v)
return A
```

Scan all edges in increasing weight order; if an edge is safe, add it to A

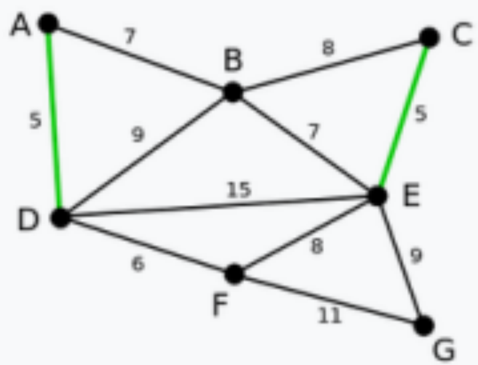
https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Kruskal's algorithm examples (1)



{ 5, 5, 6, 7, 7, 8, 8, 9, 9, 11, 15 }

AD and **CE** are the shortest edges, with length 5, and **AD** has been arbitrarily chosen, so it is highlighted.

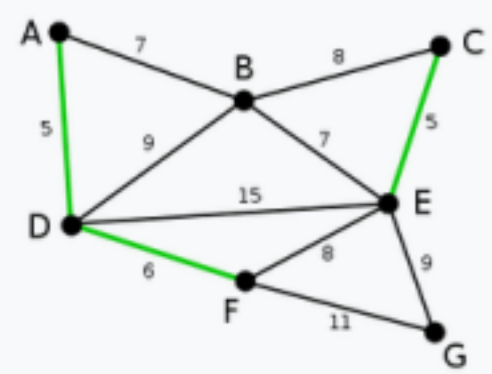


{ 5, 5, 6, 7, 7, 8, 8, 9, 9, 11, 15 }

CE is now the shortest edge that does not form a cycle, with length 5, so it is highlighted as the second edge.

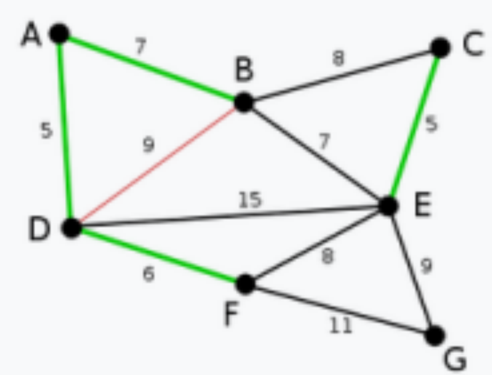
https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Kruskal's algorithm examples (2)



{ 5, 5, 6, 7, 7, 8, 8, 9, 9, 11, 15 }

The next edge, **DF** with length 6, is highlighted using much the same method.

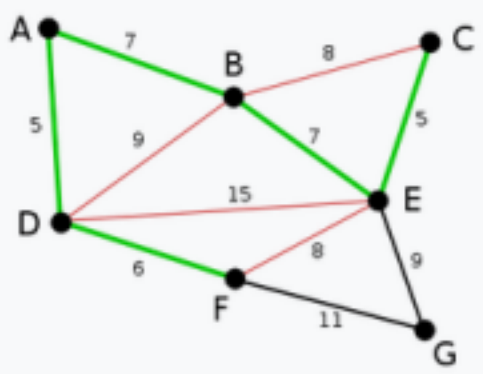
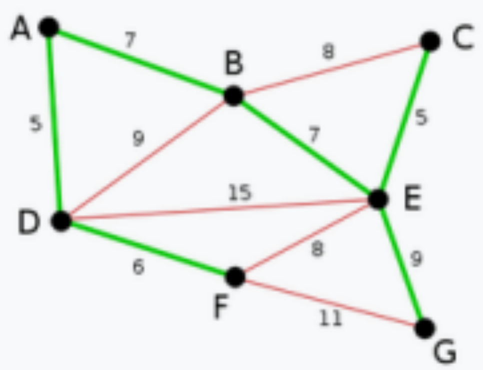


{ 5, 5, 6, 7, 7, 8, 8, ~~9,~~ 9, 11, 15 }

The next-shortest edges are **AB** and **BE**, both with length 7. **AB** is chosen arbitrarily, and is highlighted. The edge **BD** has been highlighted in red, because there already exists a path (in green) between **B** and **D**, so it would form a cycle (**ABD**) if it were chosen.

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

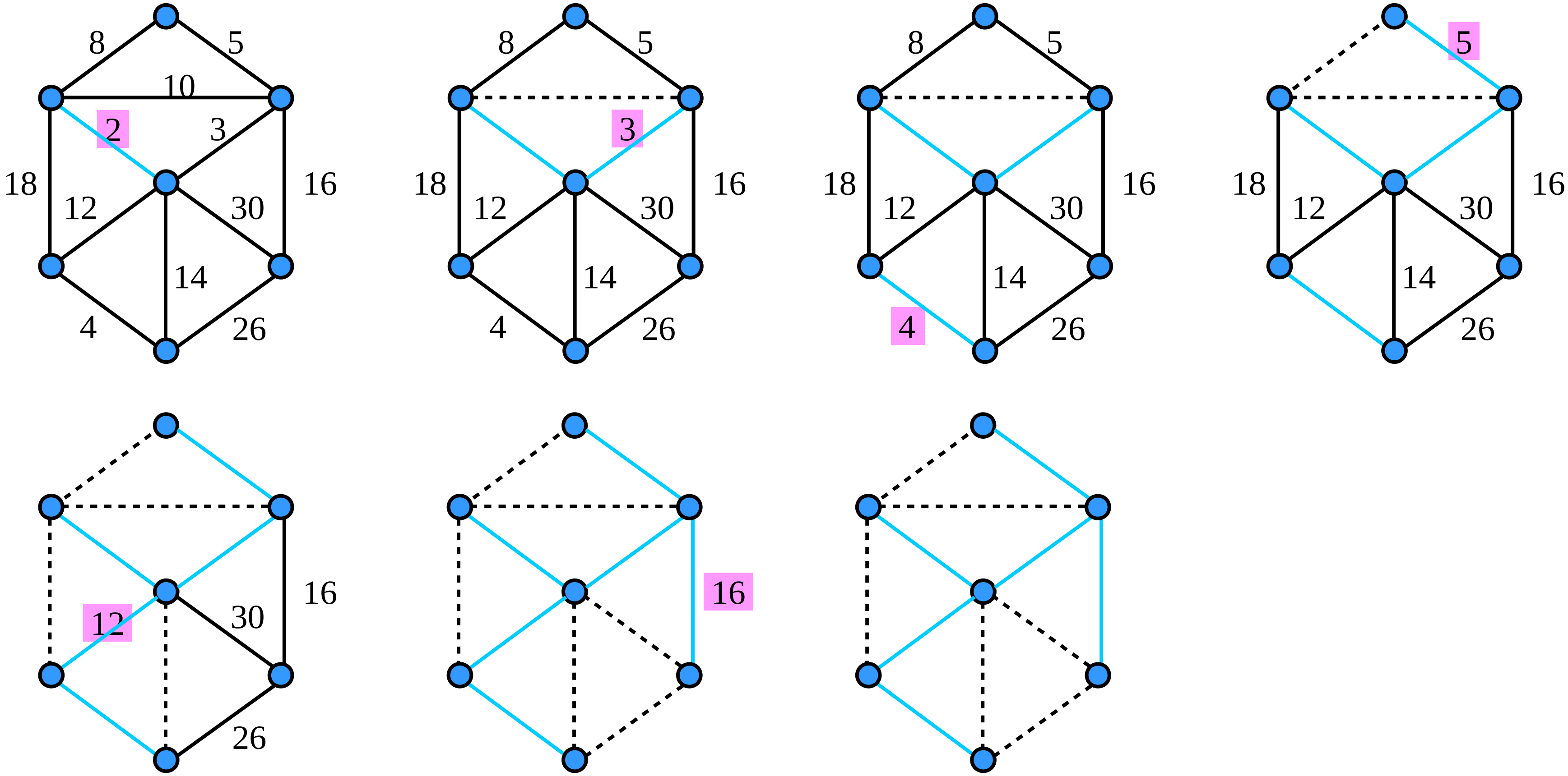
Kruskal's algorithm examples (3)

	$\{5, 5, 6, 7, 7, 8, 8, 8, 9, 11, 15\}$ The process continues to highlight the next-smallest edge, **BE** with length 7. Many more edges are highlighted in red at this stage: **BC** because it would form the loop **BCE**, **DE** because it would form the loop **DEBA**, and **FE** because it would form **FEBAD**.
	$\{5, 5, 6, 7, 7, 8, 8, 8, 9, 11, 15\}$ Finally, the process finishes with the edge **EG** of length 9, and the minimum spanning tree is found.

https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Kruskal's algorithm examples (4)

$\{2, 3, 4, 5, 8, 10, 12, 14, 16, 18, 26, 30\}$



http://jeffe.cs.illinois.edu/teaching/algorithms/notes/20-mst.pdf

Prim's algorithm

a greedy algorithm that finds a minimum spanning tree
for a weighted undirected graph.

operates by building this tree one vertex at a time,
from an arbitrary starting vertex,
at each step adding the cheapest possible connection
from the tree to another vertex.

Repeatedly add a safe edge to the tree

1. Initialize a tree with a single vertex,
   chosen arbitrarily from the graph.
2. Grow the tree by one edge:
   of the edges that connect the tree to vertices
   not yet in the tree, find the minimum-weight edge,
   and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).

https://en.wikipedia.org/wiki/Prim%27s_algorithm

Prim's algorithm

1. Associate with each vertex **v** of the graph
   a number **C[v]** (the cheapest cost of a connection to v)
   and an edge **E[v]** (the cheapest edge).
   Initial values: C[v] = +∞, E[v] = flag for no connection

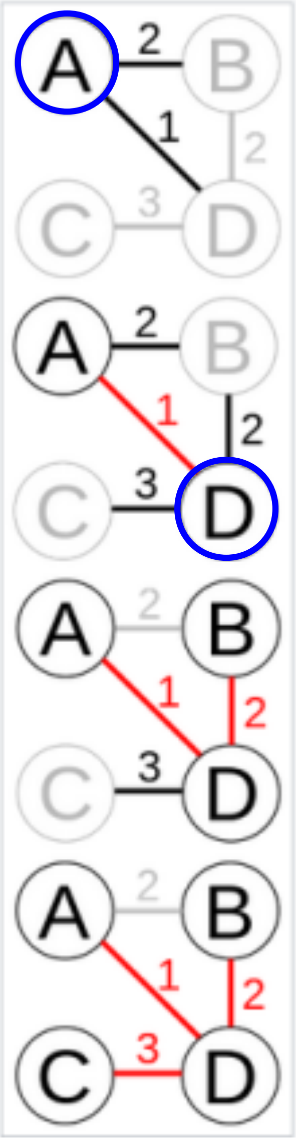
2. Initialize an empty **forest F** and a **set Q** of **vertices**
   that have not yet been included in **F**

3. Repeat the following steps until **Q** is empty:
   a. Find and remove a vertex **v** from **Q**
      having the minimum possible value of **C[v]**
   b. Add **v** to **F** and, if **E[v]** is not the special flag value,
      also add E[v] to F
   c. Loop over the edges **vw** connecting **v** to other
      vertices **w**. For each such edge, if w still belongs to Q
      and **vw** has smaller weight than **C[w]**,
      perform the following steps:
      I) Set **C[w]** to the cost of edge **vw**
      II) Set **E[w]** to point to edge **vw**.

   Return F

https://en.wikipedia.org/wiki/Prim%27s_algorithm

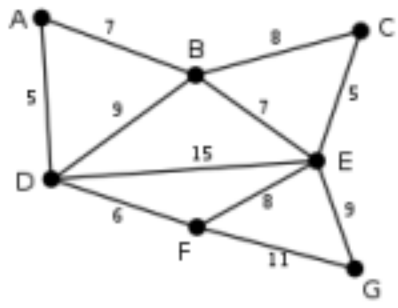
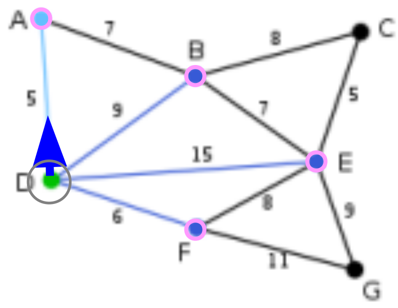
Prim's algorithm



Prim's algorithm starting at vertex A.
In the third step, edges BD and AB both have weight 2, so BD is chosen arbitrarily.
After that step, AB is no longer a candidate for addition to the tree because it links two nodes that are already in the tree.

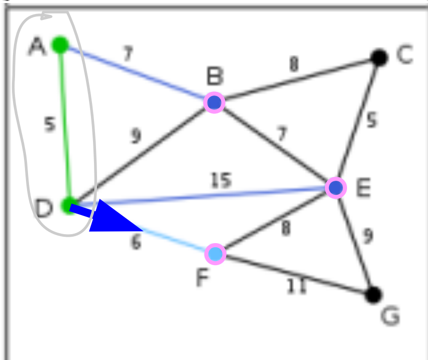
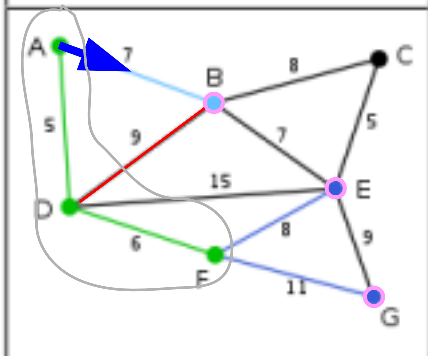
https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

Prim's algorithm examples (1)

Image	Description	Not seen	In the graph	In the tree
	This is the initial weighted graph. It is not a tree, since to be a tree it is required that there are no cycles, and in this case there is. The numbers near the edges indicate the weight. None of the edges is marked, and vertex **D** has been chosen arbitrarily as the starting point.	C, G	A, B, E, F	D
	The second vertex is closest to **D** : **A** is 5 away, **B is** 9, **E is** 15, and **F is** 6. Of these, 5 is the smallest value, so we mark the **DA** edge. $\{5, 6, 9, 15\}$	C, G	B, E, F	A, D

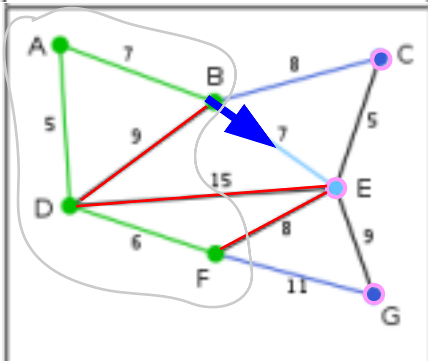
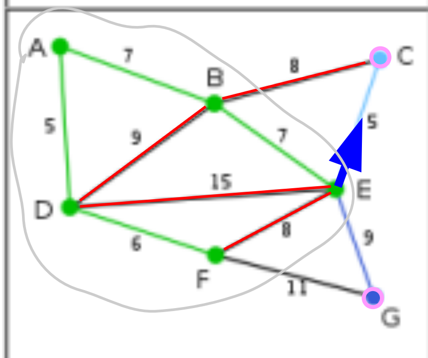
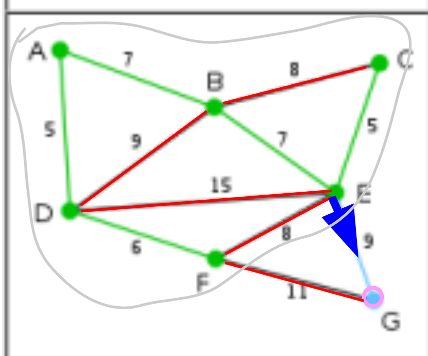
https://es.wikipedia.org/wiki/Algoritmo_de_Prim

Prim's algorithm examples (2)

Image	Description	Not seen	In the graph	In the tree
	The next vertex to choose is the closest to **D** or **A.** **B** is 9 away from **D** and 7 away from **A** , **E** is at 15, and **F** is at 6. 6 is the smallest value, so we mark the vertex **F** and the edge **DF** .	C	B, E, G	A, D, F
	The algorithm continues. The vertex **B** , which is at a distance of 7 from **A** , is the next one marked. At this point the edge **DB** is marked in red because its two ends are already in the tree and therefore can not be used.	null	C, E, G	A, D, F, B

https://es.wikipedia.org/wiki/Algoritmo_de_Prim

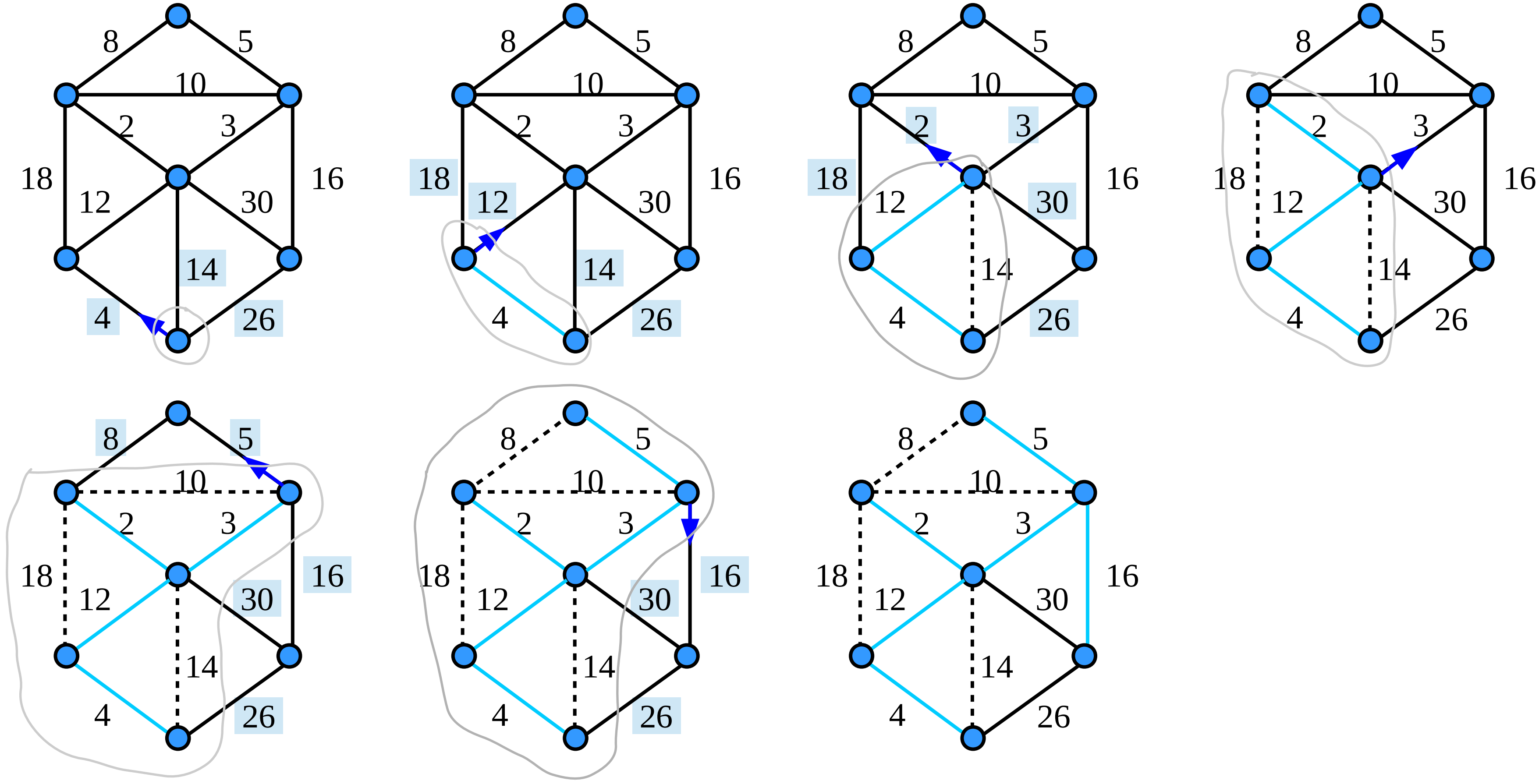
Prim's algorithm examples (3)

Image	Description	Not seen	In the graph	In the tree
	Here you have to choose between **C** , **E** and **G. C** is 8 away from **B** , **E** is 7 away from **B** , and **G** is 11 away from **F. E** is closer, so we mark the vertex **E** and the edge **EB** . Two other edges were marked in red because both vertices that join were added to the tree.	null	C, G	A, D, F, B, E
	Only **C** and **G** are available. **C** is 5 away from **E** , and **G is** 9 away from **E.** Choose **C** , and mark with the arc **EC** . The **BC** arc is also marked with red.	null	G	A, D, F, B, E, C
	**G** is the only outstanding vertex, and it is closer to **E** than to **F** , so **EG** is added to the tree. All vertices are already marked, the minimum expansion tree is shown in green. In this case with a weight of 39.	null	null	A, D, F, B, E, C, G

https://es.wikipedia.org/wiki/Algoritmo_de_Prim

Prim's algorithm examples (4)

$\{2, 3, 4, 5, 8, 10, 12, 14, 16, 18, 26, 30\}$



http://jeffe.cs.illinois.edu/teaching/algorithms/notes/20-mst.pdf

References

[1] http://en.wikipedia.org/
[2]

Tree Traversal (1A)

Please send corrections (or suggestions) to youngwlim@hotmail.com.

This document was produced by using LibreOffice and Octave.

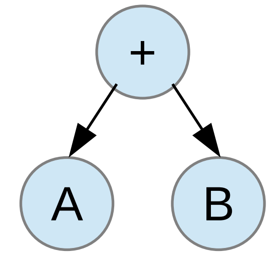
Infix, Prefix, Postfix Notations

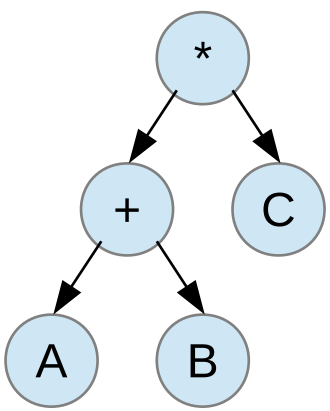
Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
A * (B + C)	* A + B C	A B C + *
A / B + C / D	+ / A B / C D	A B / C D / +
((A + B) * C) – D	– * + A B C D	A B + C * D –

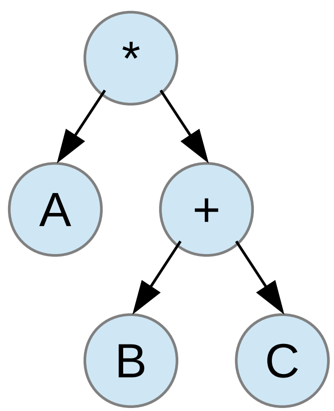
https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html

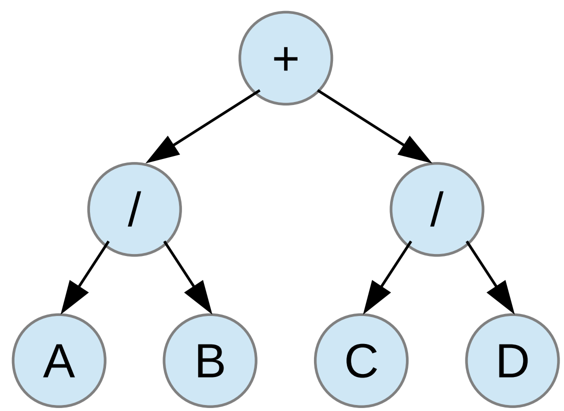
Infix, Prefix, Postfix Notations and Binary Trees

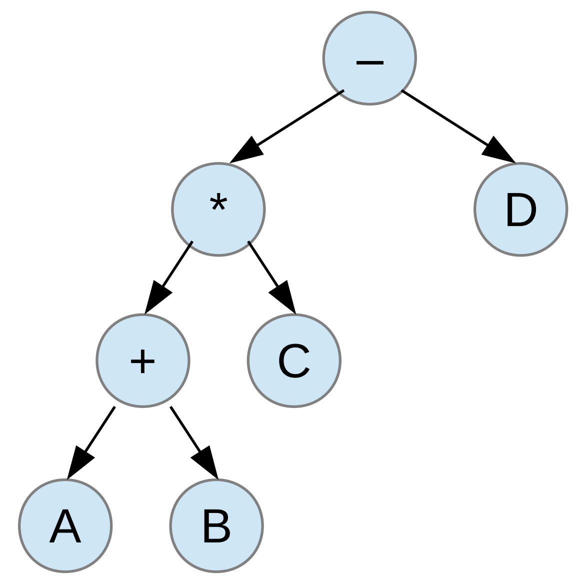
Infix Notation	Prefix Notation	Postfix Notation
A + B	+ A B	A B +
(A + B) * C	* + A B C	A B + C *
A * (B + C)	* A + B C	A B C + *
A / B + C / D	+ / A B / C D	A B / C D / +
((A + B) * C) – D	– * + A B C D	A B + C * D –





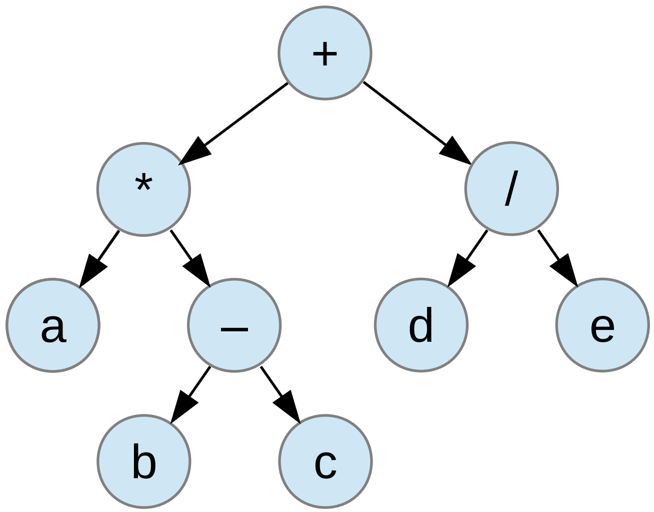


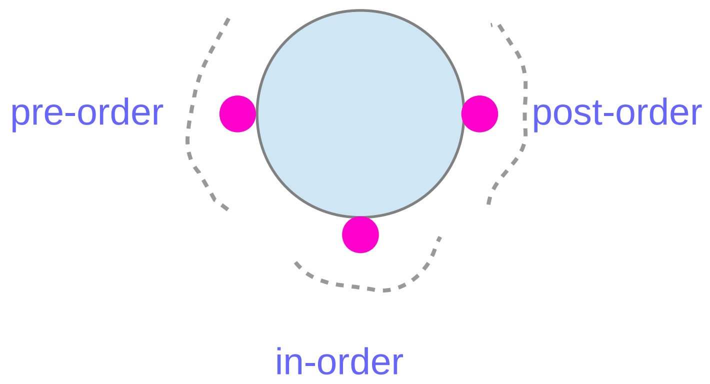




Depth First Search
- Pre-Order
- In-order
- Post-Order

Breadth First Search



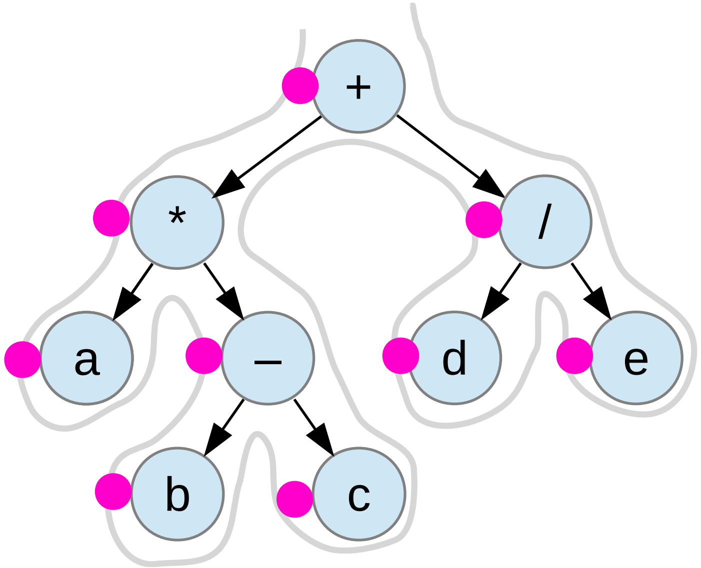


(a*(b-c))+(d/e)

a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

https://en.wikipedia.org/wiki/Morphism

Pre-Order Binary Tree Traversals

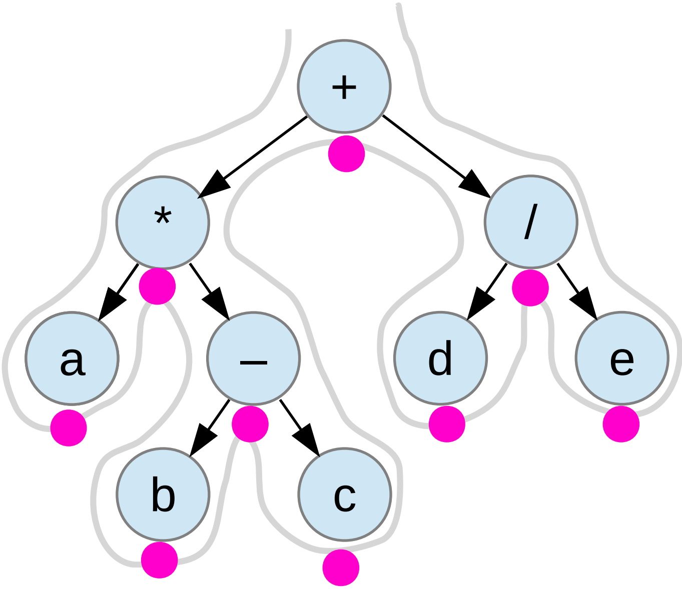


(a*(b-c))+(d/e)

a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

https://en.wikipedia.org/wiki/Morphism

In-Order Binary Tree Traversals

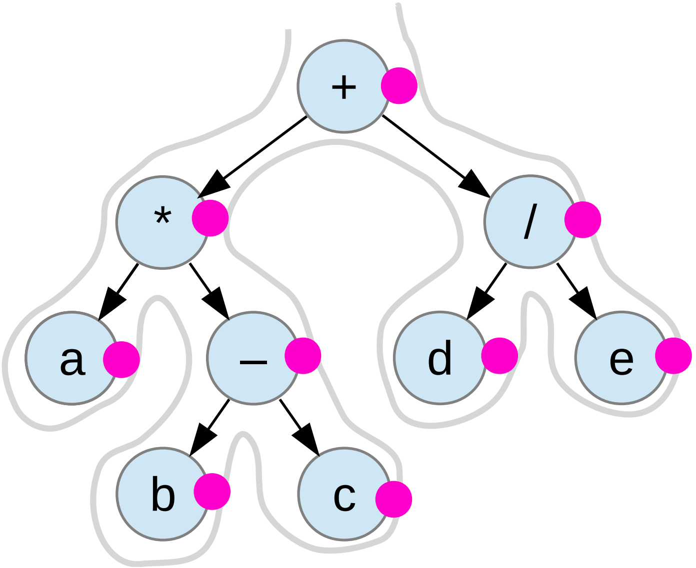


(a*(b-c))+(d/e)

a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

https://en.wikipedia.org/wiki/Morphism

Post-Order Binary Tree Traversals



(a*(b-c))+(d/e)

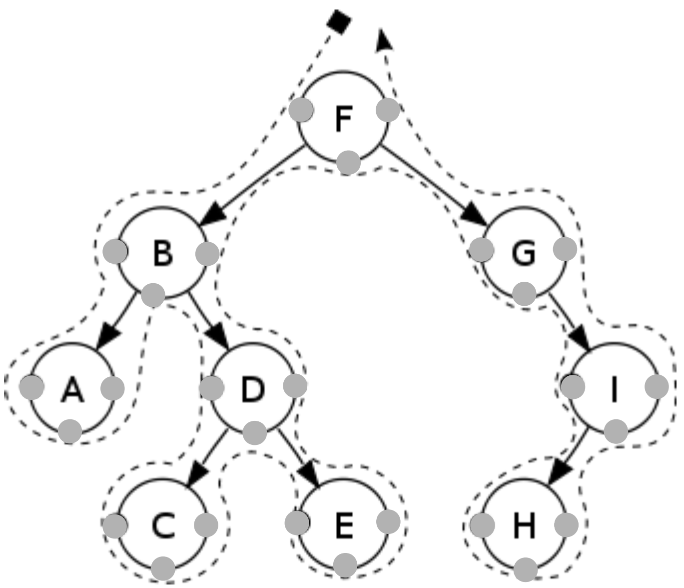
a * b – c + d / e	Infix notation
+ * a – b c / d e	Prefix notation
a b c – * d e / +	Postfix notation

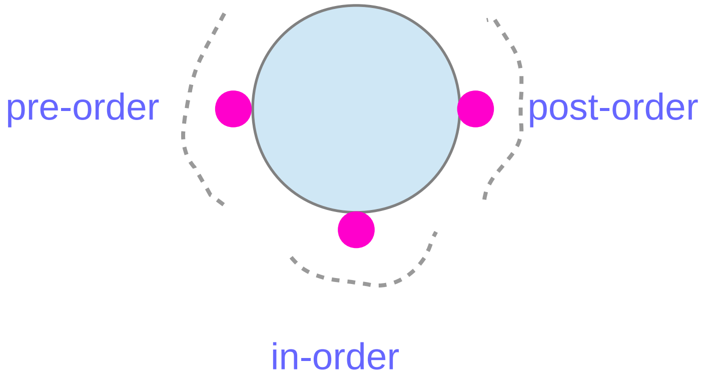
https://en.wikipedia.org/wiki/Morphism

Tree Traversal

Depth First Search
- Pre-Order
- In-order
- Post-Order

Breadth First Search



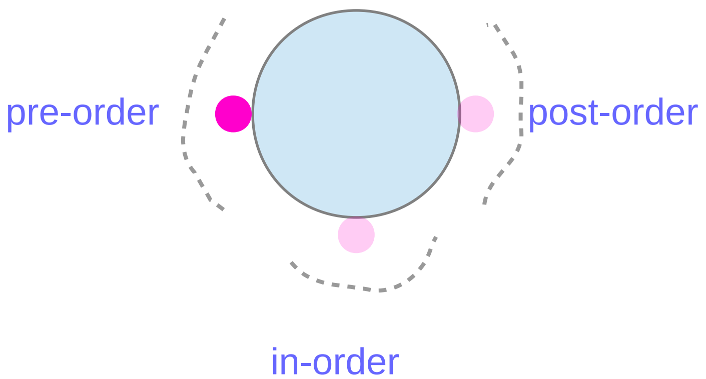


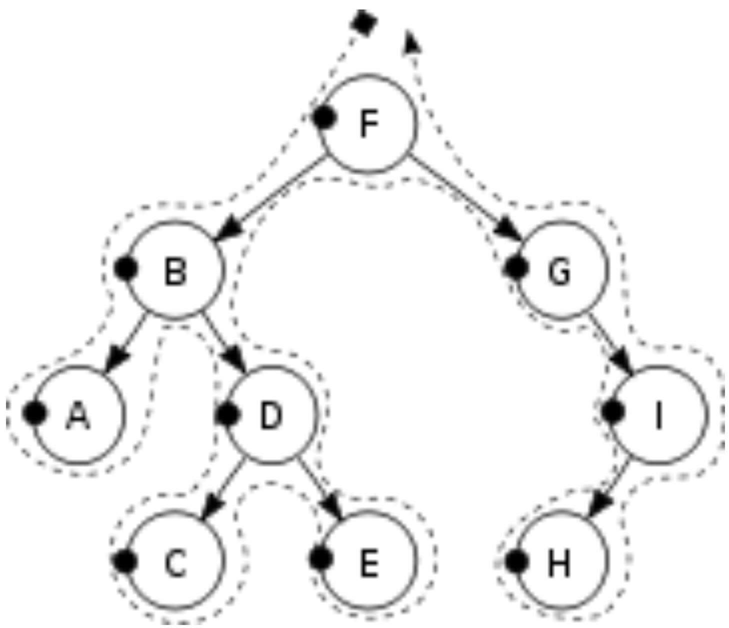
https://en.wikipedia.org/wiki/Morphism

Pre-Order

pre-order function
Check if the current node is empty / null.
Display the data part of the root (or current node).
Traverse the **left** subtree by recursively calling the **pre-order** function.
Traverse the **right** subtree by recursively calling the **pre-order** function.

FBADCEGIH





https://en.wikipedia.org/wiki/Morphism

In-Order

in-order function

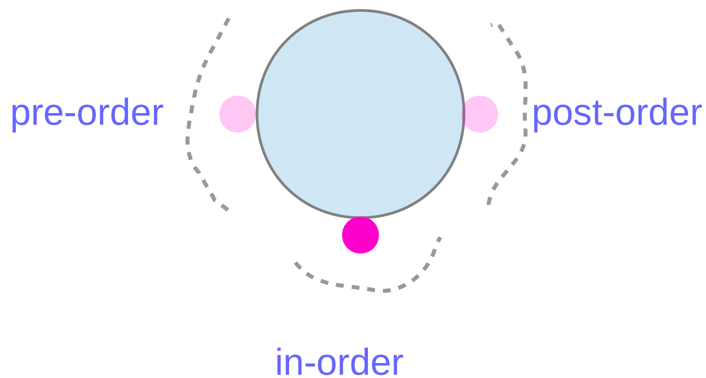
Check if the current node is empty / null.

Traverse the left subtree by recursively calling the **in-order** function.

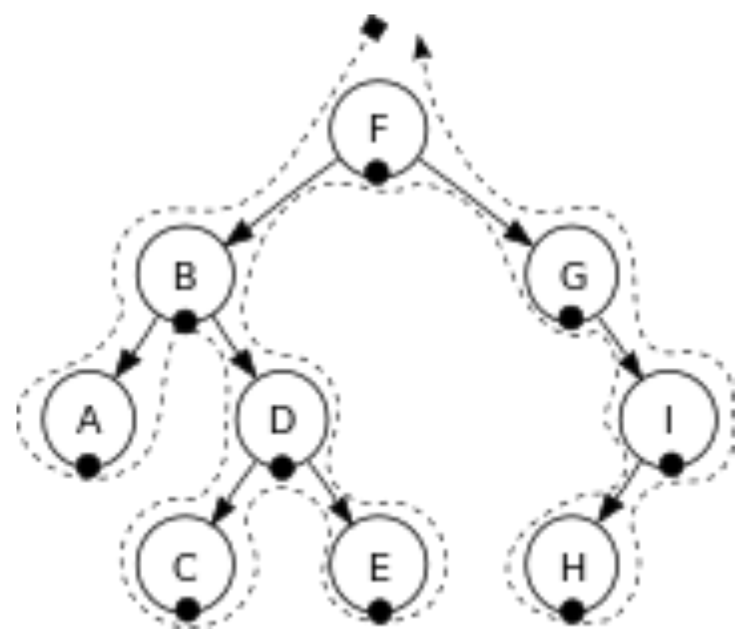
<u>**Display**</u> the data part of the root (or current node).

Traverse the right subtree by recursively calling the **in-order** function.

ABCDEFGHI



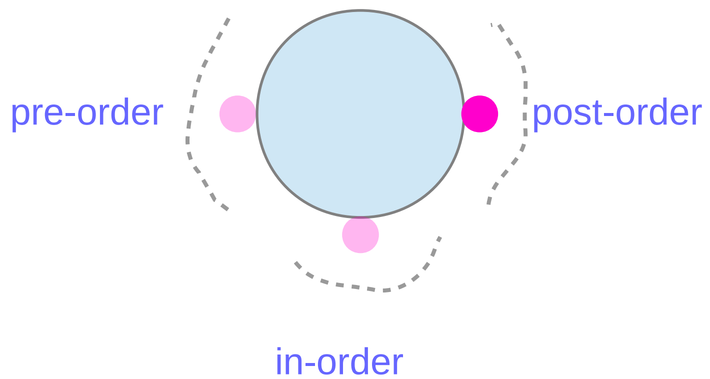
https://en.wikipedia.org/wiki/Morphism

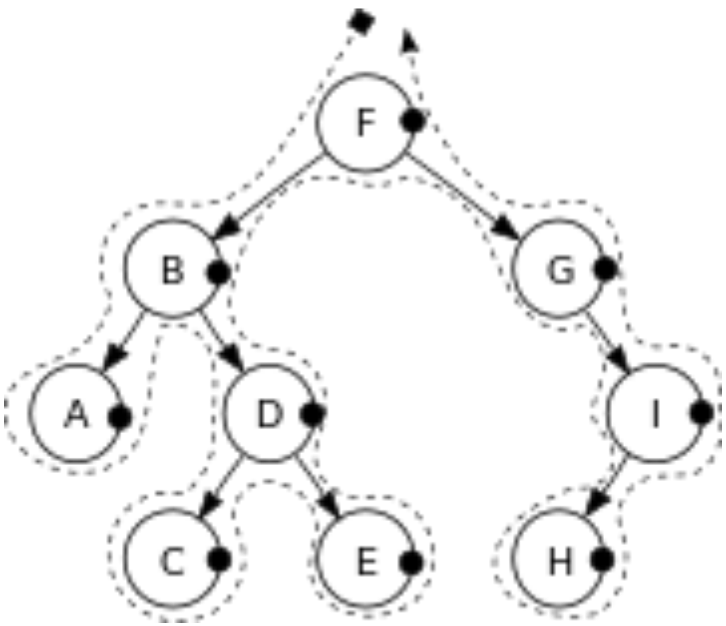


Post-Order

post-order function
Check if the current node is empty / null.
Traverse the left subtree by recursively calling the **post-order** function.
Traverse the right subtree by recursively calling the **post-order** function.
Display the data part of the root (or current node).

ACEDBHIGH





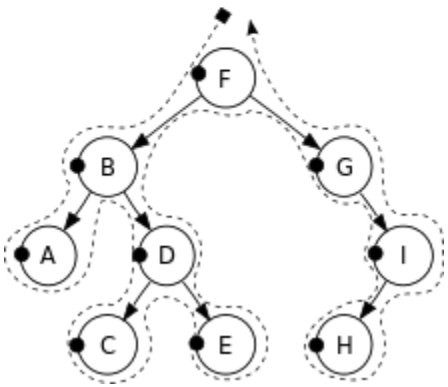
https://en.wikipedia.org/wiki/Morphism

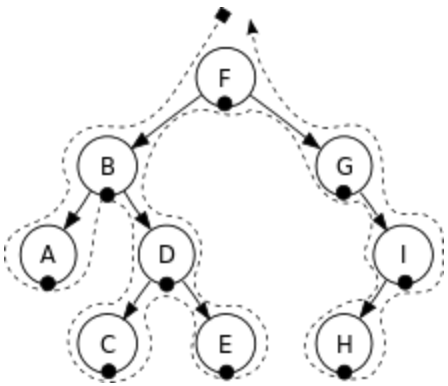
Recursive Algorithms

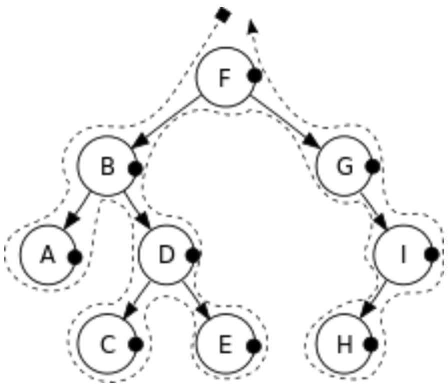
```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```

```
inorder(node)
  if (node = null)
    return
  inorder(node.left)
  visit(node)
  inorder(node.right)
```

```
postorder(node)
  if (node = null)
    return
  postorder(node.left)
  postorder(node.right)
  visit(node)
```







https://en.wikipedia.org/wiki/Tree_traversal

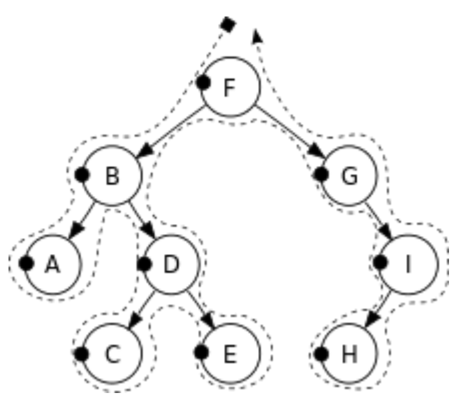
Iterative Algorithms

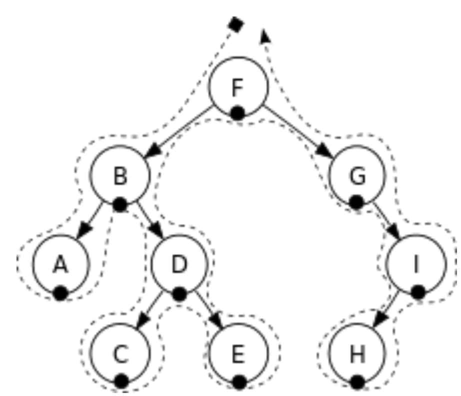
```
iterativePreorder(node)
  if (node = null)
    return
  s ← empty stack
  s.push(node)

  while (not s.isEmpty())
    node ← s.pop()
    visit(node)
    // right child is pushed first
    // so that left is processed first
    if (node.right ≠ null)
      s.push(node.right)
    if (node.left ≠ null)
      s.push(node.left)
```

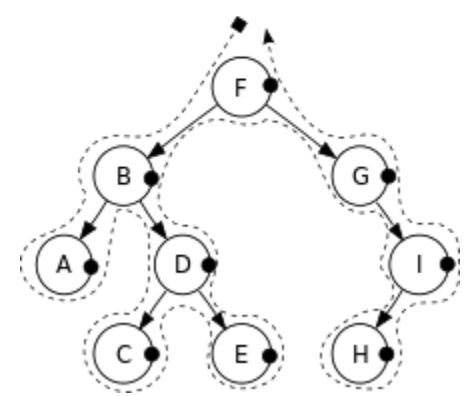
https://en.wikipedia.org/wiki/Tree_traversal

```
iterativeInorder(node)
  s ← empty stack

  while (not s.isEmpty() or
           node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      node ← s.pop()
      visit(node)
      node ← node.right
```

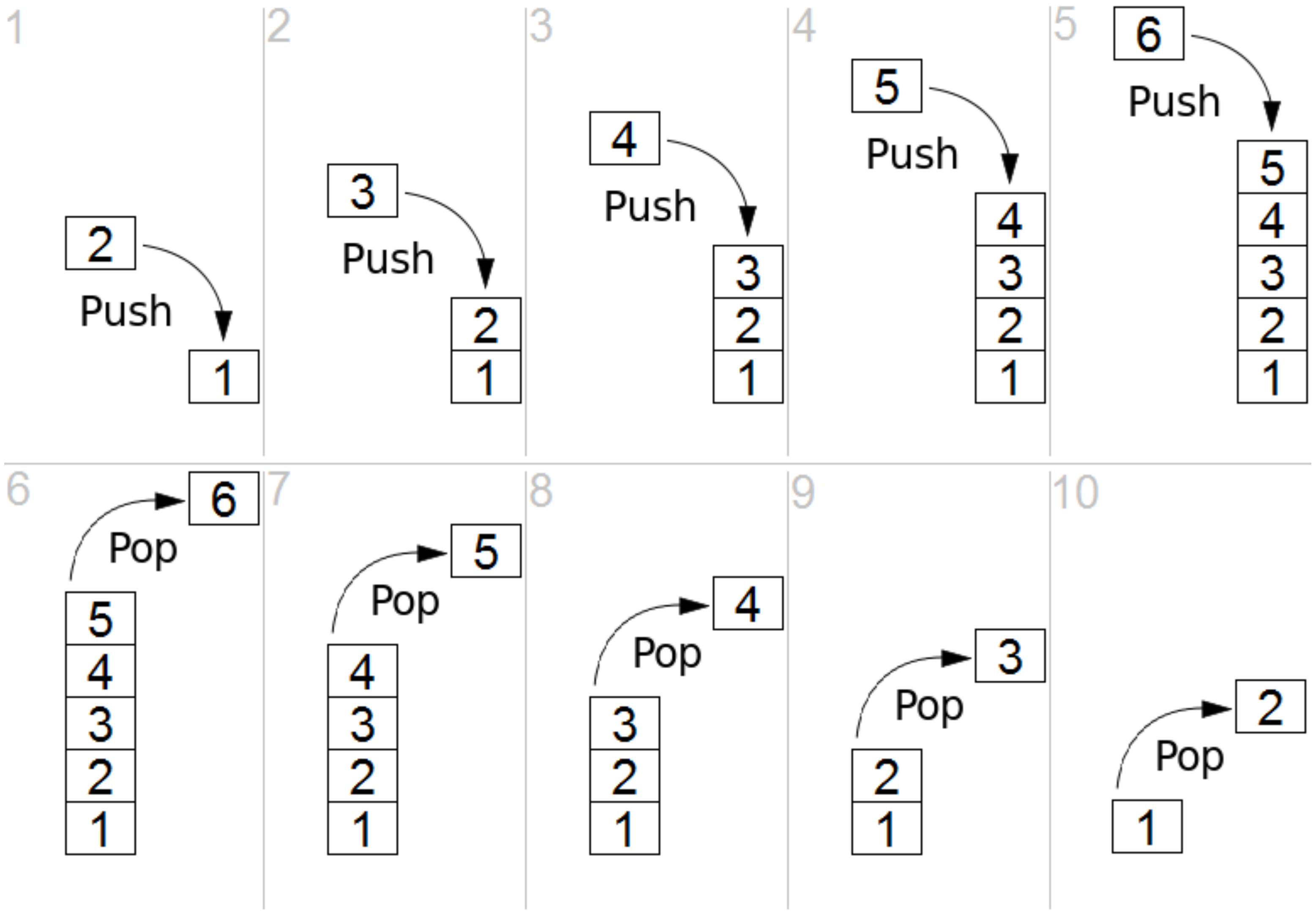
```
iterativePostorder(node)
  s ← empty stack
  lastNodeVisited ← null

  while (not s.isEmpty() or node ≠ null)
    if (node ≠ null)
      s.push(node)
      node ← node.left
    else
      peekNode ← s.peek()
      // if right child exists and traversing
      // node from left child, then move right
      if (peekNode.right ≠ null and
         lastNodeVisited ≠ peekNode.right)
        node ← peekNode.right
      else
        visit(peekNode)
        lastNodeVisited ← s.pop()
```





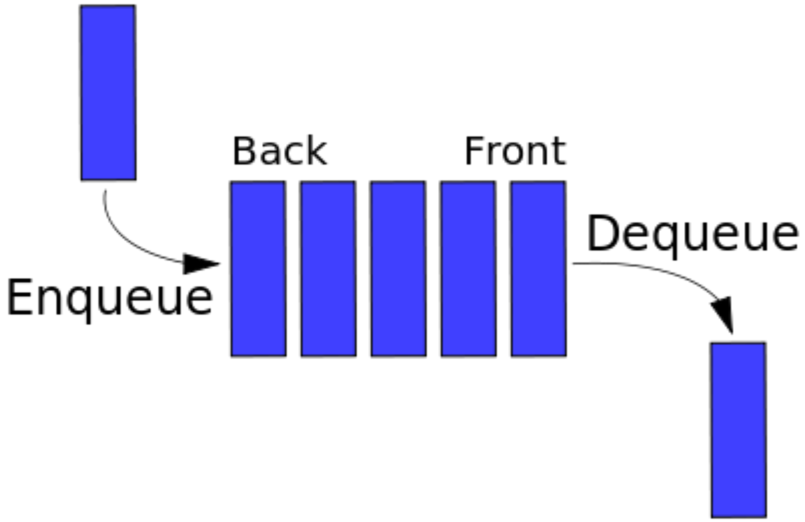


Stack



https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

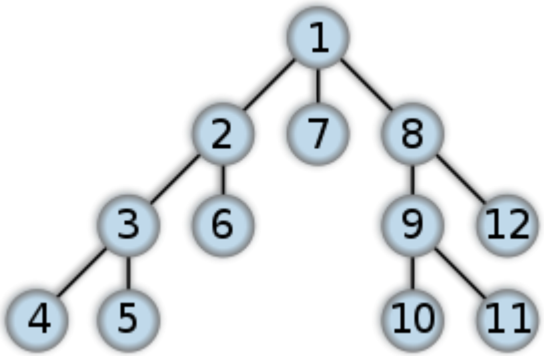
Queue



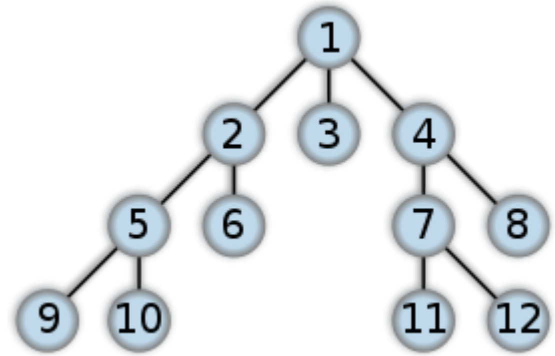
https://en.wikipedia.org/wiki/Queue_(abstract_data_type)#/media/File:Data_Queue.sv
g

Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)



https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

DFS Algorithm

A recursive implementation of DFS:

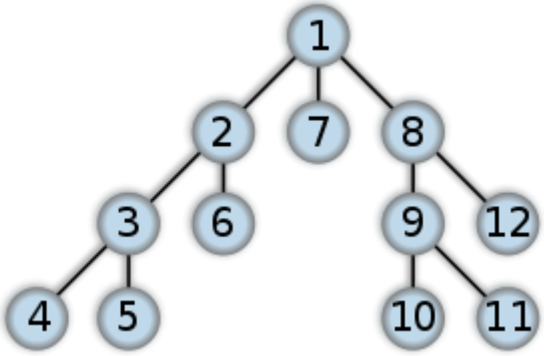
```
procedure DFS(G,v):
    label v as discovered
    for all edges from v to w in G.adjacentEdges(v) do
        if vertex w is not labeled as discovered then
            recursively call DFS(G,w)
```

A non-recuUrsive implementation of DFS:

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

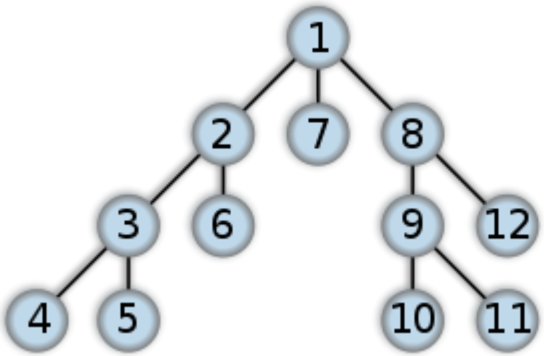
https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

DFS (Depth First Search)

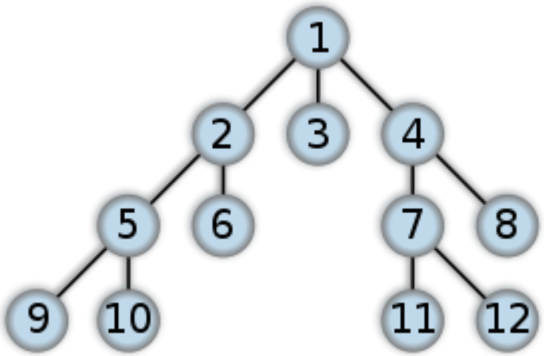


Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)

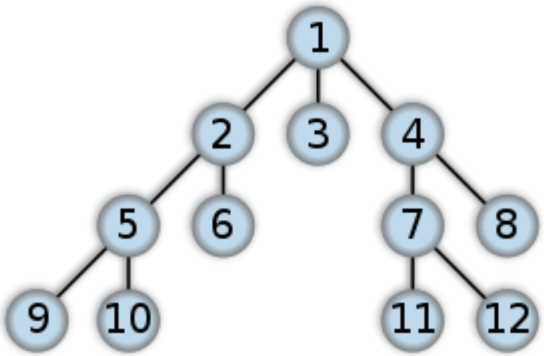


https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

BFS Algorithm

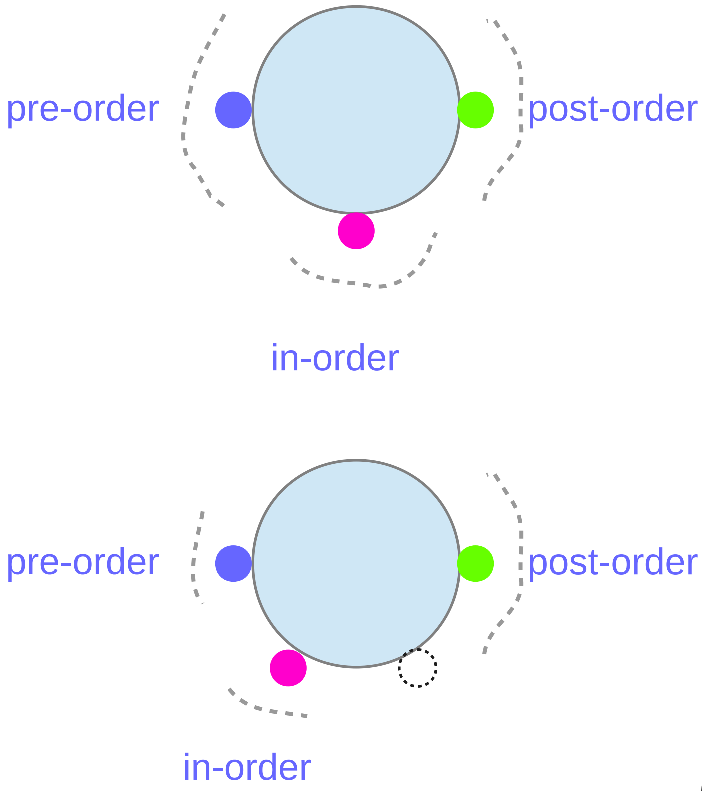
```
Breadth-First-Search(Graph, root):

    create empty set S
    create empty queue Q

    add root to S
    Q.enqueue(root)

    while Q is not empty:
        current = Q.dequeue()
        if current is the goal:
            return current
        for each node n that is adjacent to current:
            if n is not in S:
                add n to S
                n.parent = current
                Q.enqueue(n)
```

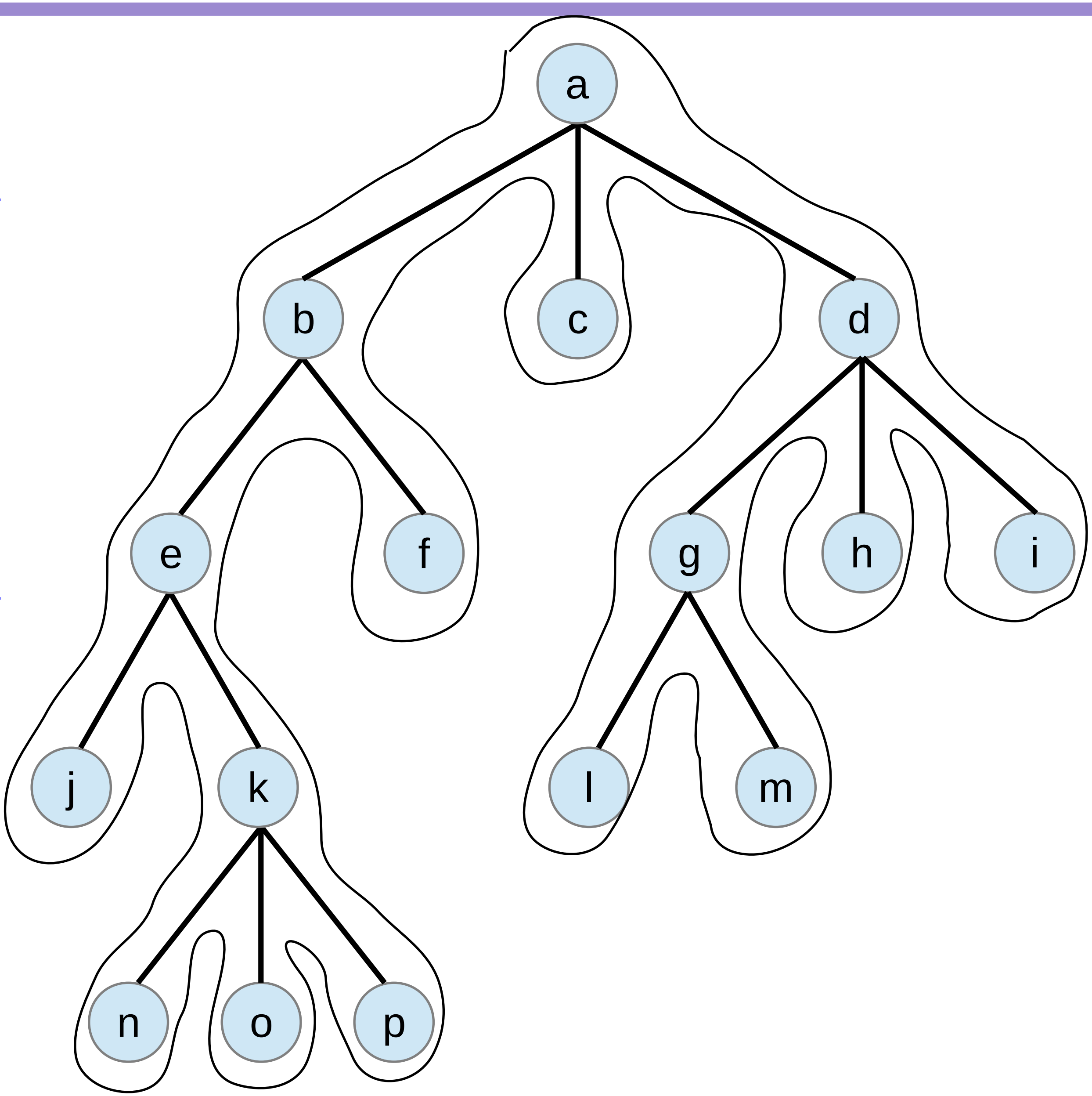
BFS (Breadth First Search)



https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search

In-Order

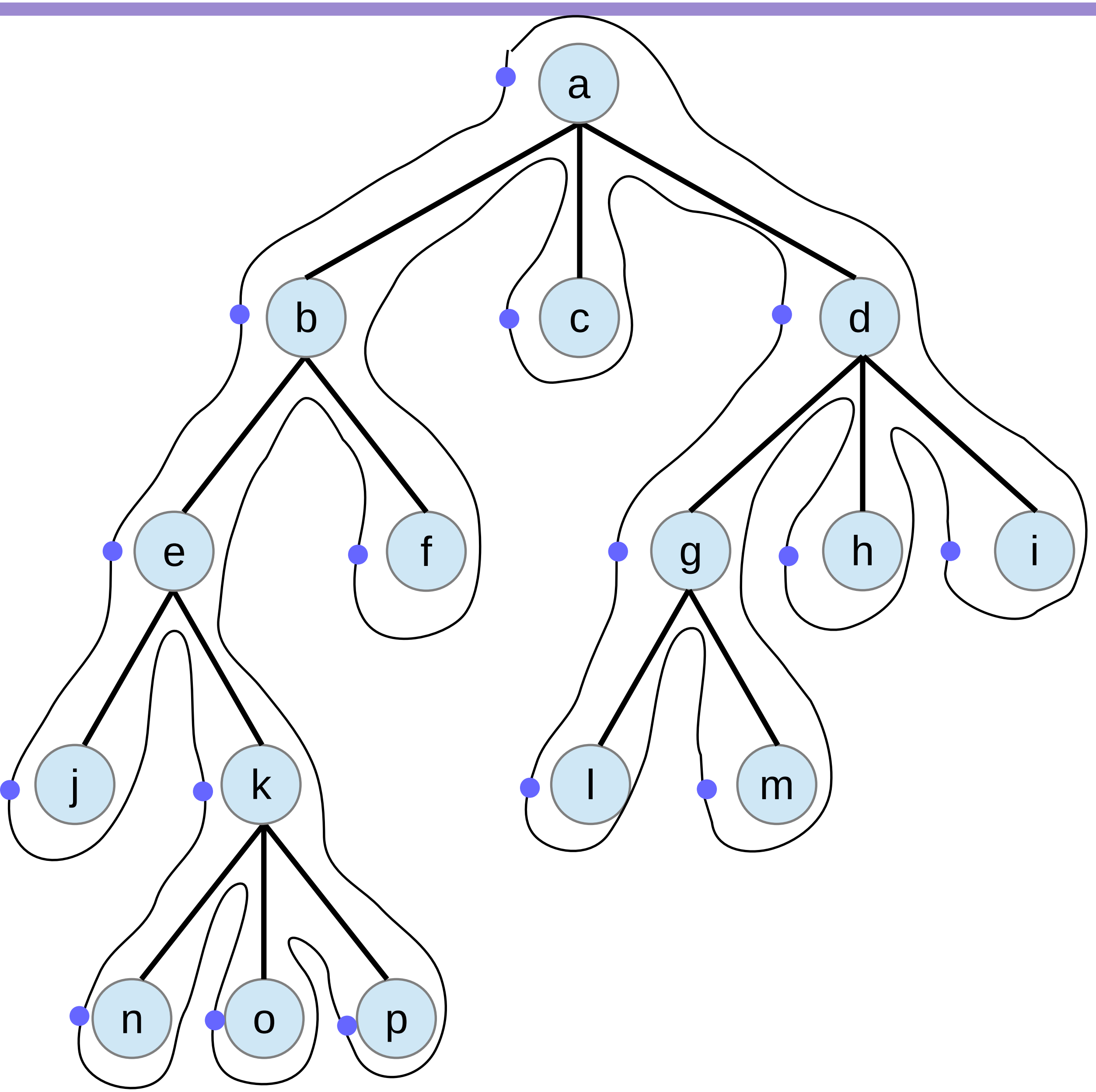




Rosen

Ternary Tree

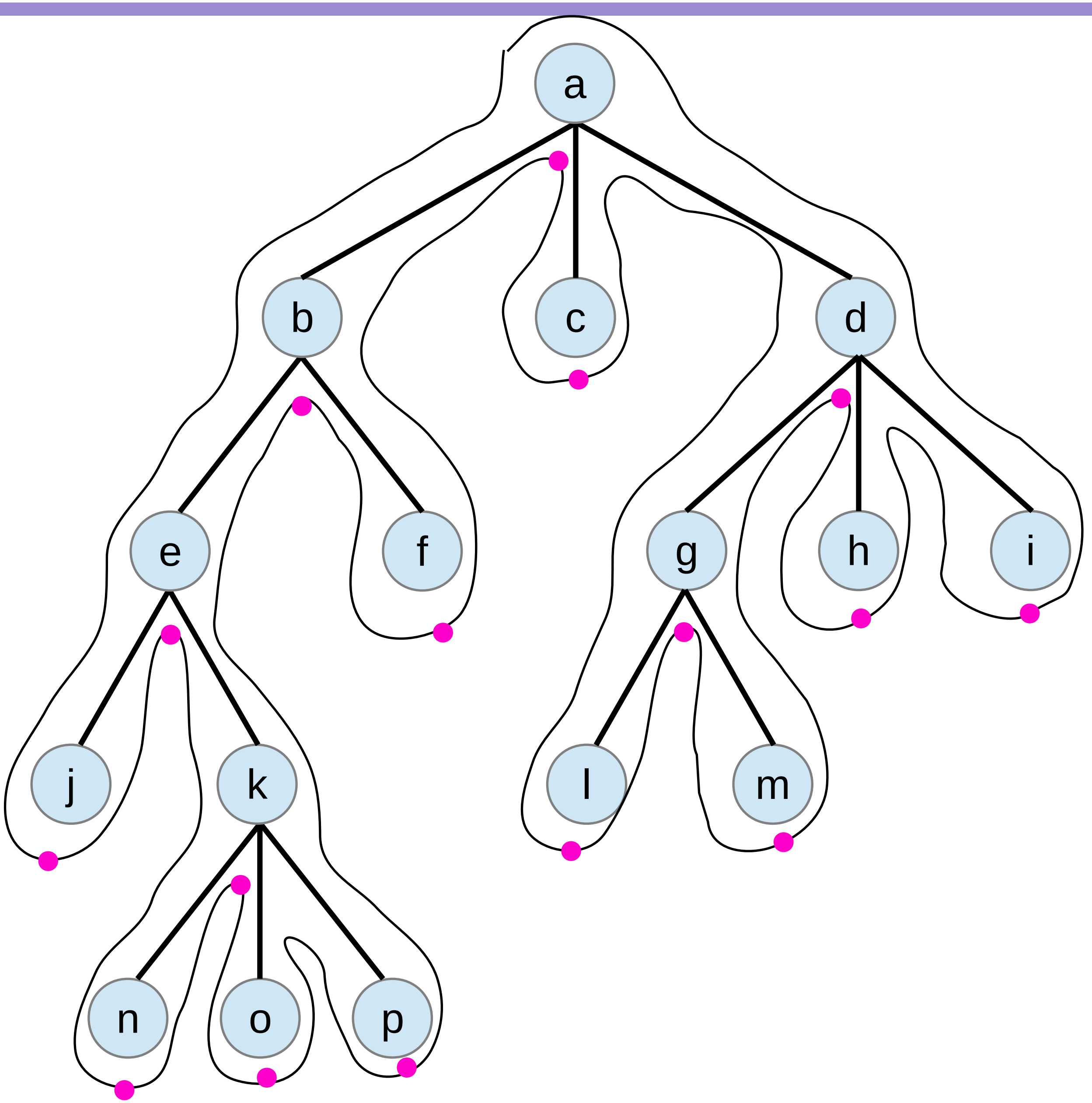
a-b-e-j-k-n-o-p-f-c-d-g-l-m-h-i



Rosen

In-Order

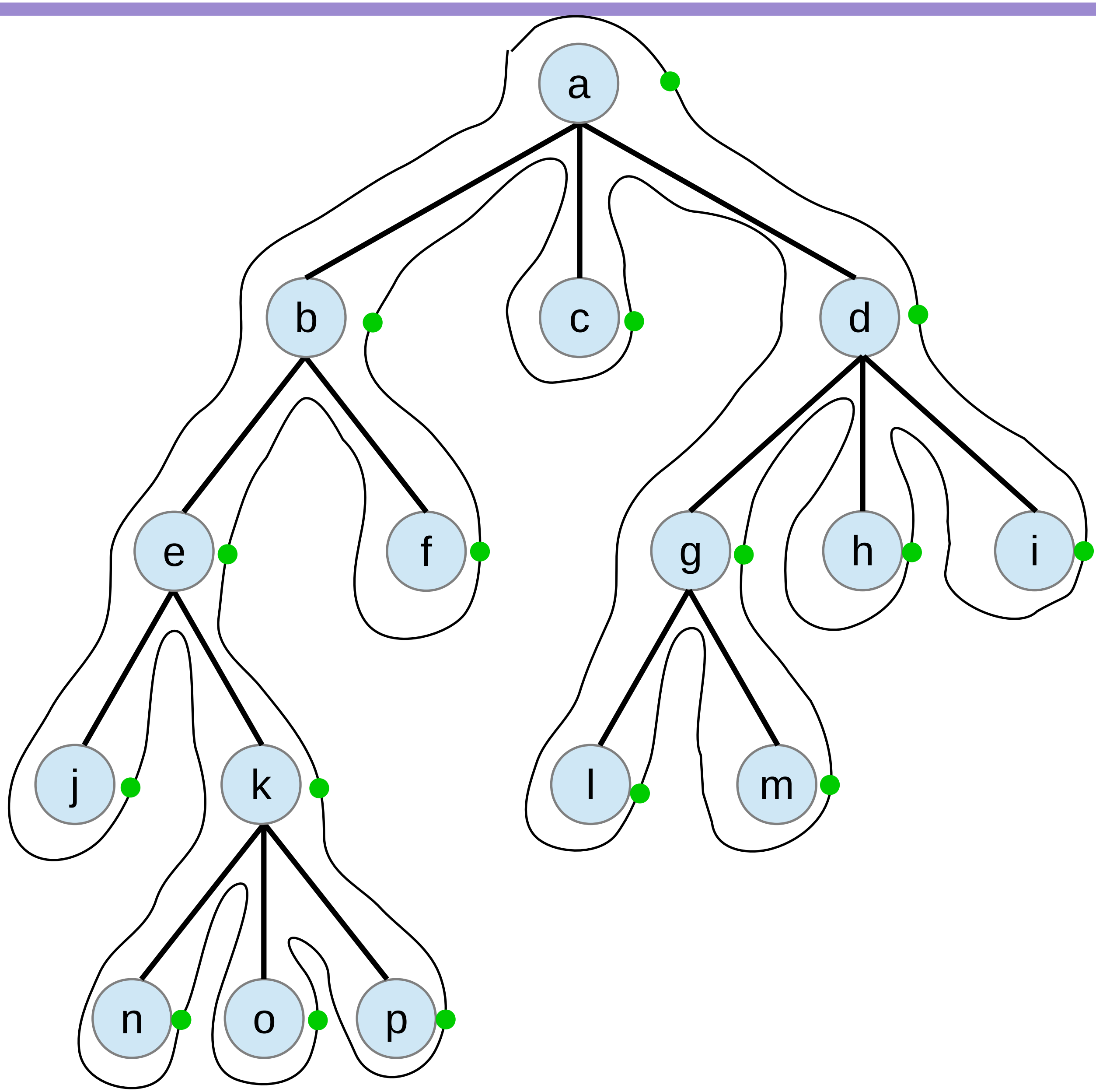
j-e-n-k-o-p-b-f-a-c-l-g-m-d-h-i



Rosen

Post-Order

j-n-o-p-k-e-f-b-c-l-m-g-h-i-d-a



Rosen

Ternary

Ternary

Etymology
Late Latin ternarius (“consisting of three things”), from terni (“three each”).
Adjective

ternary (not comparable)
Made up of three things; treble, triadic, triple, triplex
Arranged in groups of three
(mathematics) To the base three [quotations ▼]
(mathematics) Having three variables

https://en.wiktionary.org/wiki/ternary

The sequence continues with **quaternary**, **quinary**, **senary**, **septenary**, **octonary**, **nonary**, and **denary**, although most of these terms are rarely used. There's no word relating to the number eleven but there is one that relates to the number twelve: **duodenary**.

https://en.oxforddictionaries.com/explore/what-comes-after-primary-secondary-tertiary

References

[1] http://en.wikipedia.org/
[2]