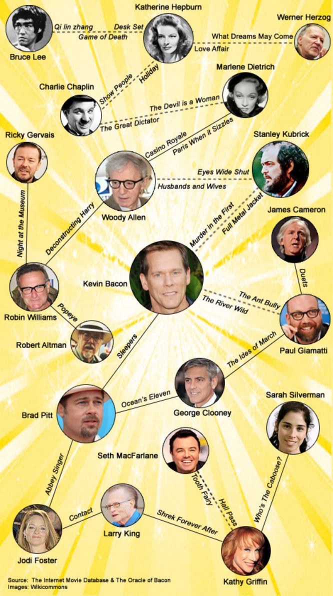
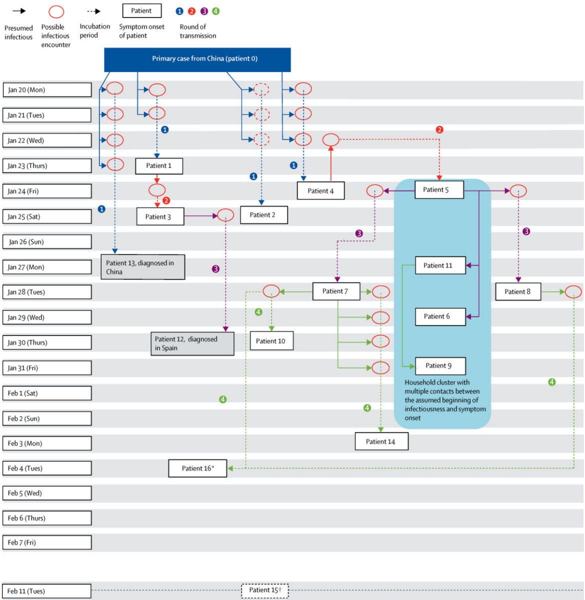
CS 3824: Introduction to Graphs

T. M. Murali

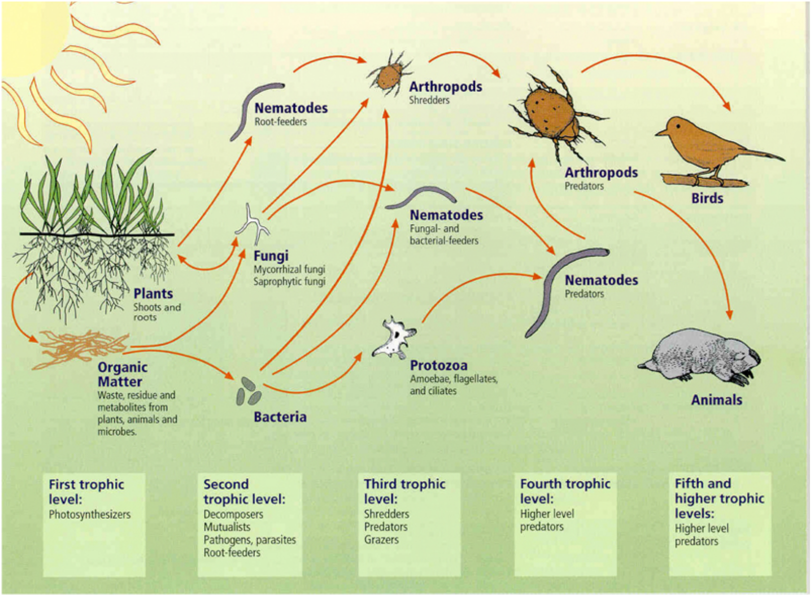
August 25, 30 2022

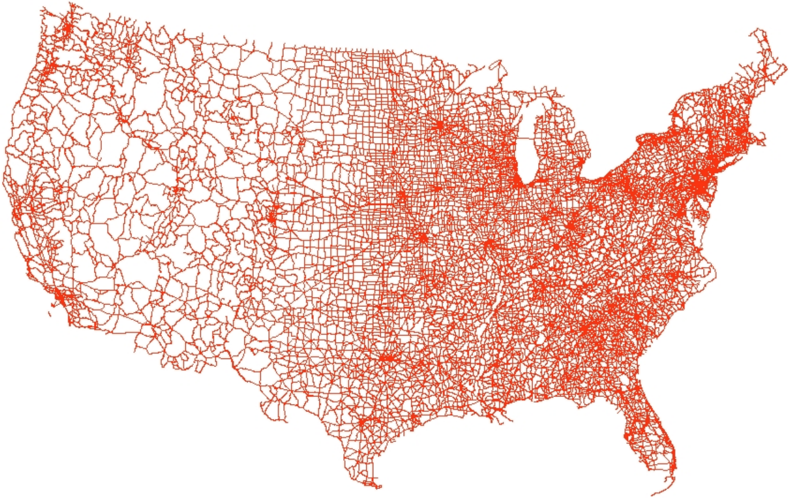


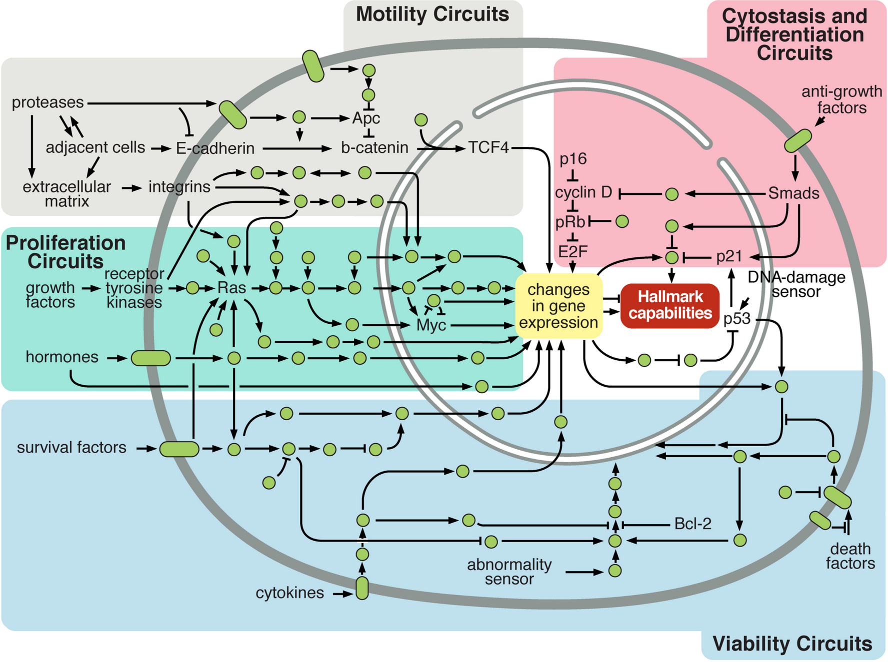
The Oracle of Bacon



(Böhmer *et al.*, *The Lancet*, May 15, 2020)


Arthropods
Shredders
Nematodes
Root-feeders
Arthropods
Predators
Birds
Nematodes
Fungal- and bacterial-feeders
Fungi
Mycorrhizal fungi
Saprophytic fungi
Plants
Shoots and roots
Nematodes
Predators
Organic Matter
Waste, residue and metabolites from plants, animals and microbes.
Protozoa
Amoebae, flagellates, and ciliates
Animals
Bacteria
First trophic level:
Photosynthesizers
Second trophic level:
Decomposers
Mutualists
Pathogens, parasites
Root-feeders
Third trophic level:
Shredders
Predators
Grazers
Fourth trophic level:
Higher level predators
Fifth and higher trophic levels:
Higher level predators




Motility Circuits
Cytostasis and Differentiation Circuits
proteases
adjacent cells
E-cadherin
extracellular matrix
integrins
Apc
b-catenin
TCF4
anti-growth factors
p16
cyclin D
pRb
E2F
Smads
p21
Proliferation Circuits
growth factors
receptor tyrosine kinases
Ras
Myc
changes in gene expression
Hallmark capabilities
DNA-damage sensor
p53
hormones
survival factors
Bcl-2
abnormality sensor
cytokines
death factors
Viability Circuits

Graphs

Graph $\equiv$ Network

- Model pairwise relationships (edges) between objects (nodes).

Graphs

Graph ≡ Network

- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: gene and protein networks, our bodies (nervous and circulatory systems, brains).
- Other examples:

Graphs

Graph $\equiv$ Network

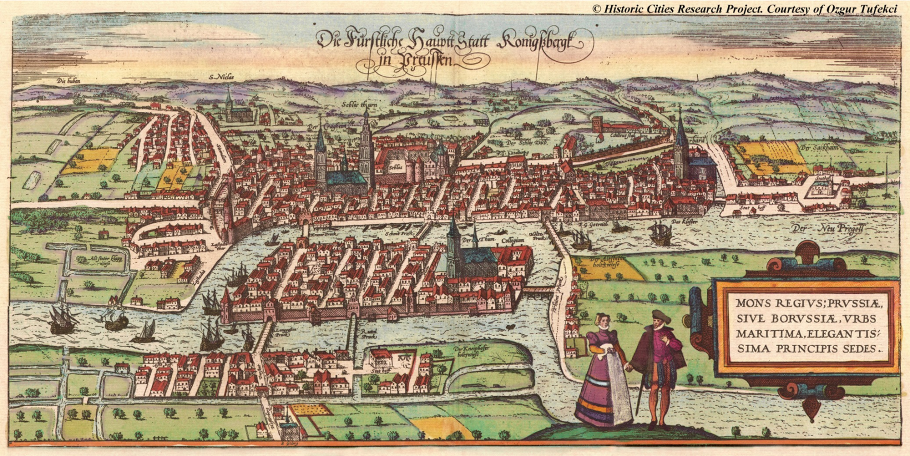
- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: gene and protein networks, our bodies (nervous and circulatory systems, brains).
- Other examples: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, transportation networks, ...

Graphs

Graph $\equiv$ Network

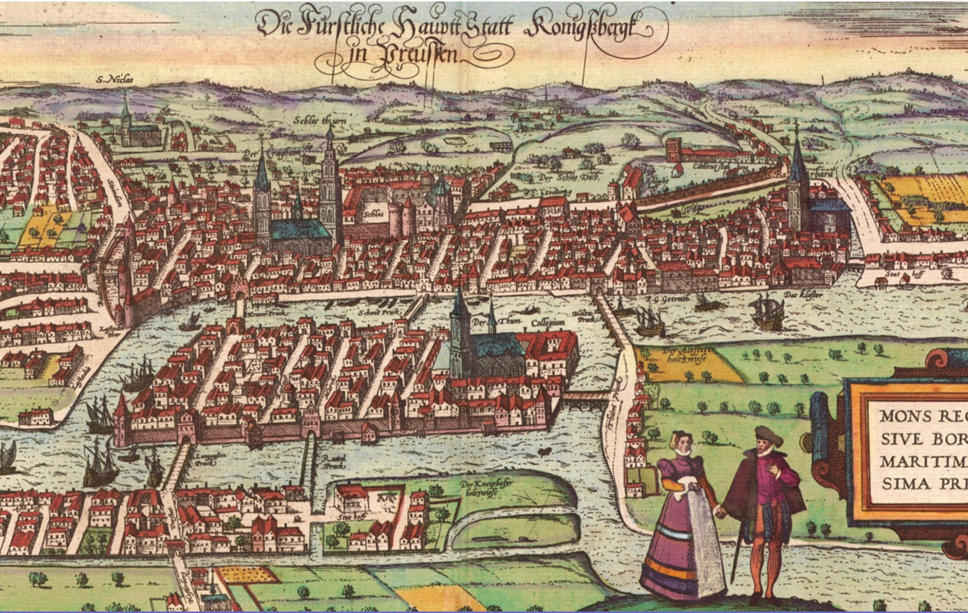
- Model pairwise relationships (edges) between objects (nodes).
- Useful in a large number of applications: gene and protein networks, our bodies (nervous and circulatory systems, brains).
- Other examples: computer networks, the World Wide Web, ecology (food webs), social networks, software systems, job scheduling, VLSI circuits, cellular networks, transportation networks, ...
- Problems involving graphs have a rich history dating back to Euler.

Euler and Graphs

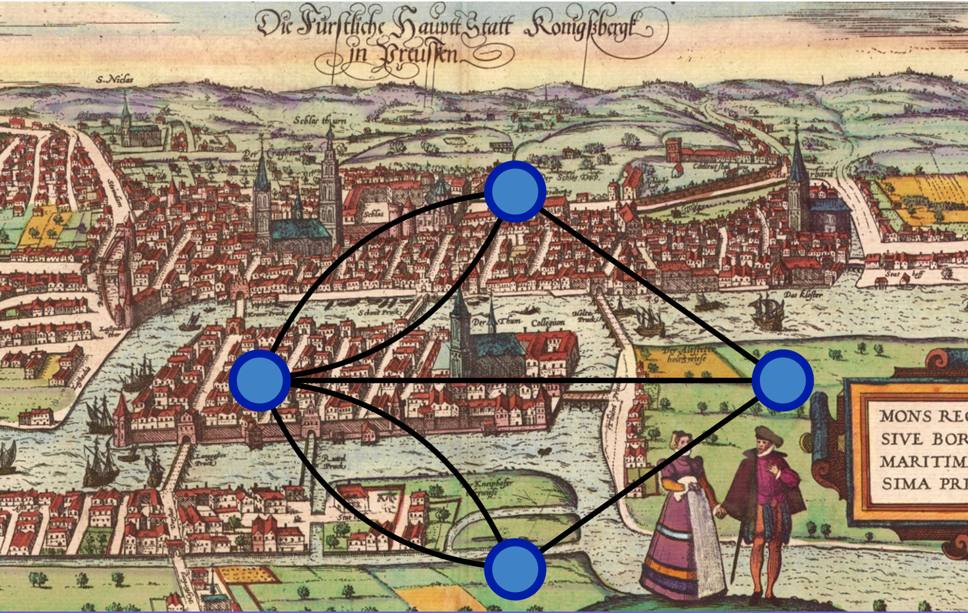


Devise a walk through the city that
crosses each of the seven bridges exactly once.

Euler and Graphs

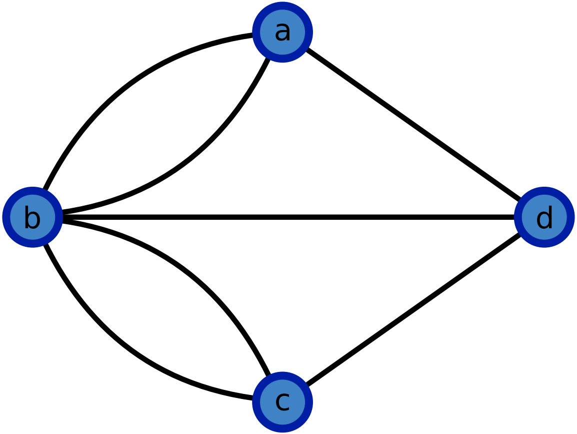


Euler and Graphs



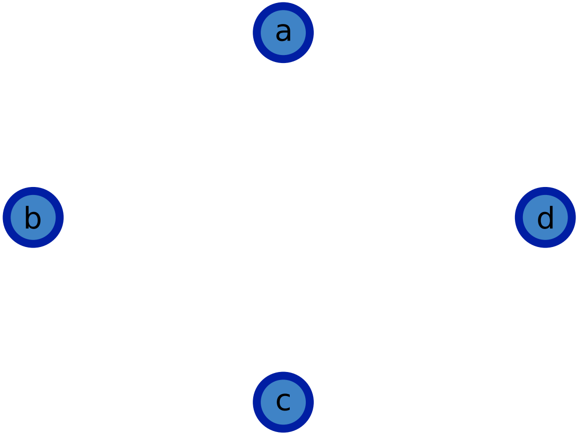
Definition of an Undirected Graph

- *Undirected graph* $G = (V, E)$: set V of nodes and set E of edges.
  - Each element of E is an unordered pair of nodes.
  - Edge (u, v) is *incident* on u, v; u and v are *neighbours* of each other.
  - G contains no self loops.



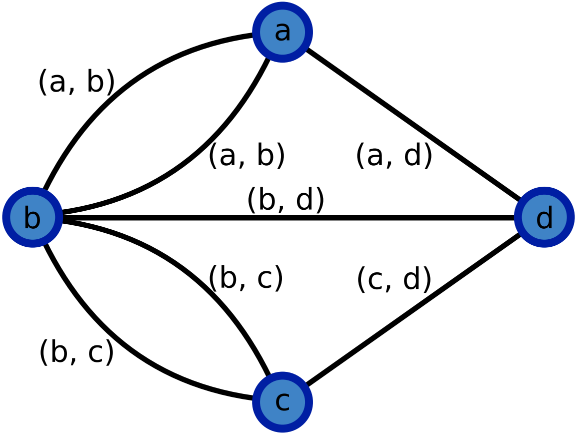
Definition of an Undirected Graph

- *Undirected graph* $G = (V, E)$: set V of nodes and set E of edges.
  - Each element of E is an unordered pair of nodes.
  - Edge (u, v) is *incident* on u, v; u and v are *neighbours* of each other.
  - G contains no self loops.

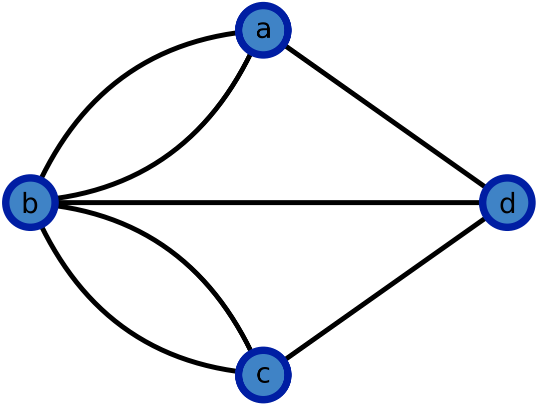


Definition of an Undirected Graph

- *Undirected graph* $G = (V, E)$: set V of nodes and set E of edges.
  - Each element of E is an unordered pair of nodes.
  - Edge (u, v) is *incident* on u, v; u and v are *neighbours* of each other.
  - G contains no self loops.

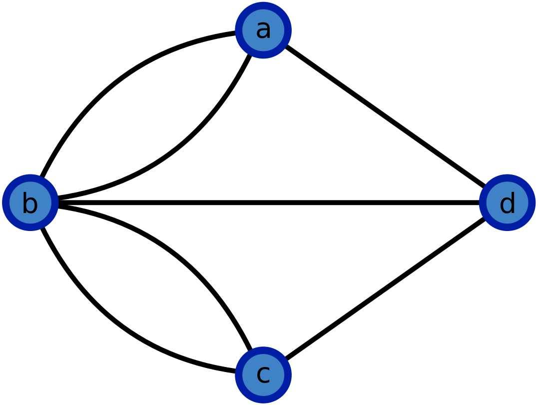


Paths and Cycles in Graphs



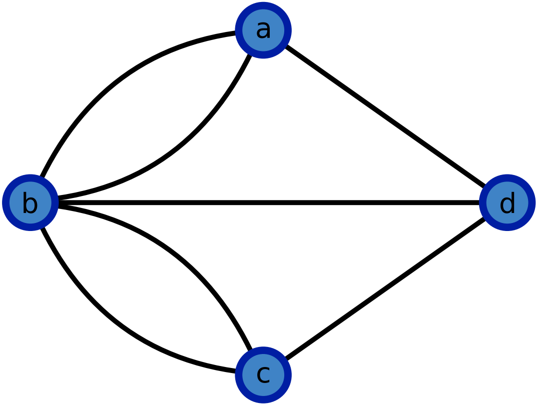
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that for every $i, 1 \leq i < k$, (v_i, v_{i+1}) is an edge in E.

Paths and Cycles in Graphs



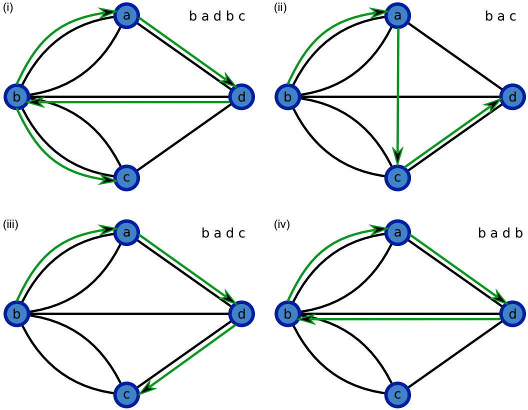
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that for every $i, 1 \leq i < k$, (v_i, v_{i+1}) is an edge in E.
- A path is *simple* if all its nodes are distinct.

Paths and Cycles in Graphs



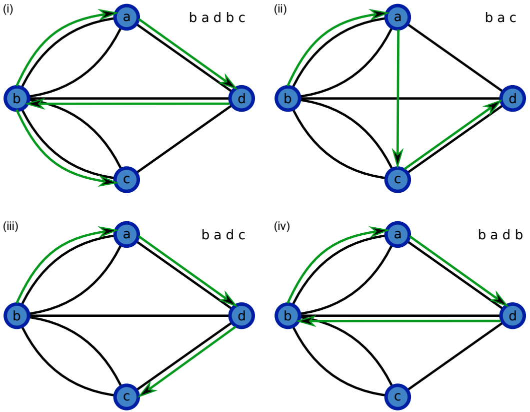
- A v_1-v_k *path* in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that for every $i, 1 \leq i < k$, (v_i, v_{i+1}) is an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where the first $k - 1$ nodes are distinct and $v_1 = v_k$. Poll

Paths and Cycles in Graphs



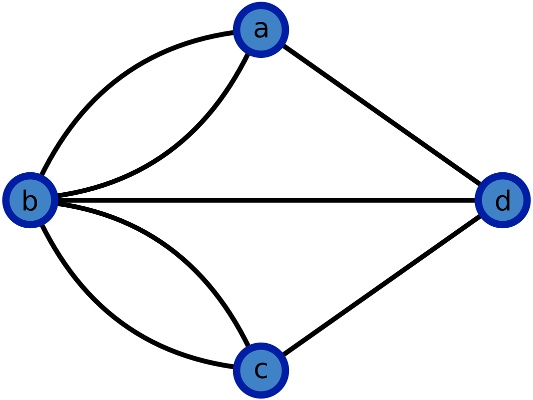
- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that for every $i, 1 \leq i < k$, (v_i, v_{i+1}) is an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where the first $k - 1$ nodes are distinct and $v_1 = v_k$. Poll

Paths and Cycles in Graphs



- A *v_1-v_k path* in an undirected graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_{k-1}, v_k \in V$ such that for every $i, 1 \leq i < k$, (v_i, v_{i+1}) is an edge in E.
- A path is *simple* if all its nodes are distinct.
- A *cycle* is a path where the first $k-1$ nodes are distinct and $v_1 = v_k$. ▸ Poll
- An undirected graph G is *connected* if for every pair of nodes $u, v \in V$, there is a u-v path in G.

Bridges to Graphs

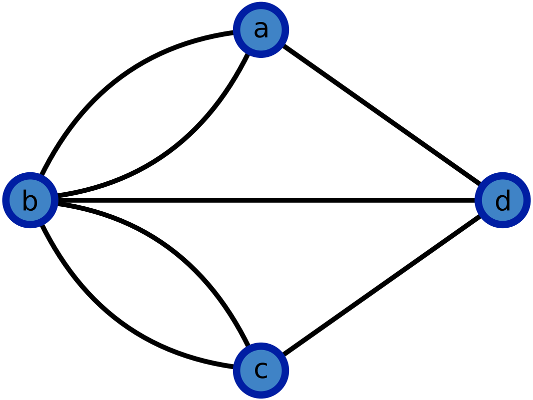


EULERIAN TOUR

Given an undirected graph $G(V, E)$,

construct an *Eulerian tour*, i.e., a path in G that traverses each edge in E exactly once, .

Bridges to Graphs

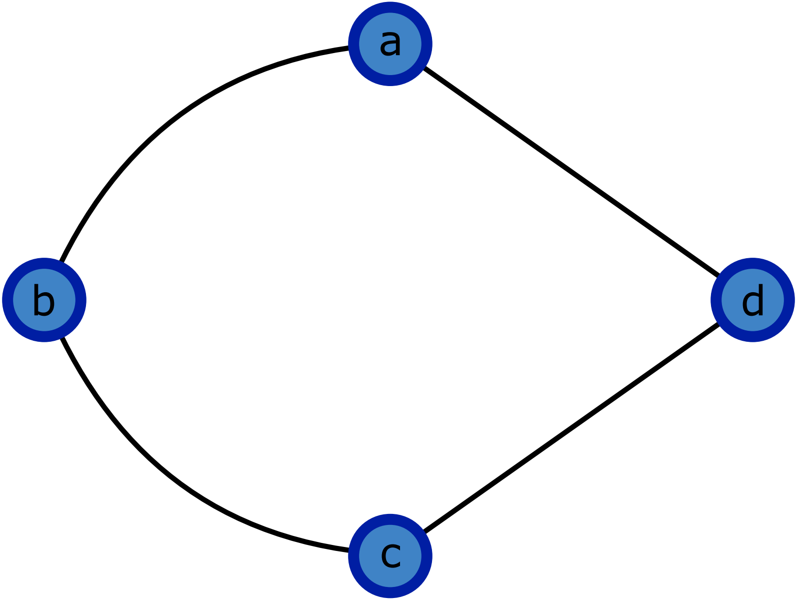


EULERIAN TOUR

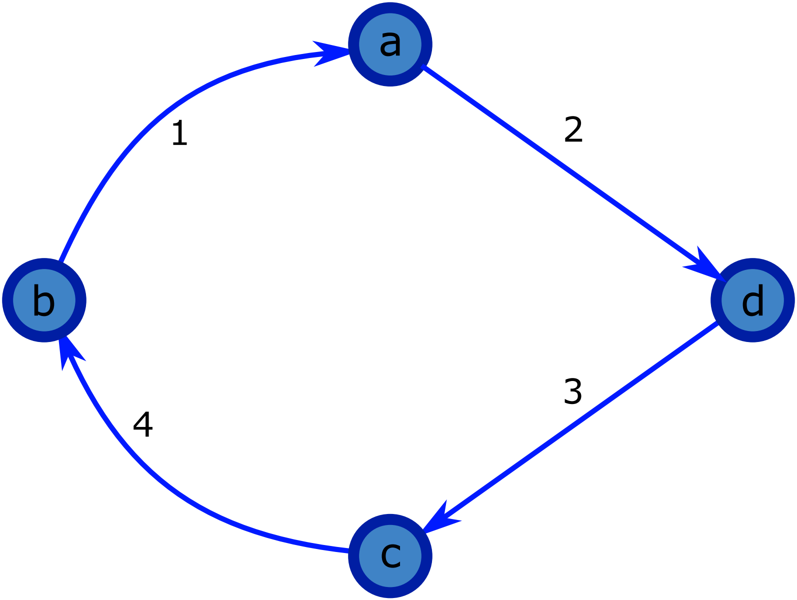
Given an undirected graph $G(V, E)$,

construct an *Eulerian tour*, i.e., a path in G that traverses each edge in E exactly once, if such a tour exists.

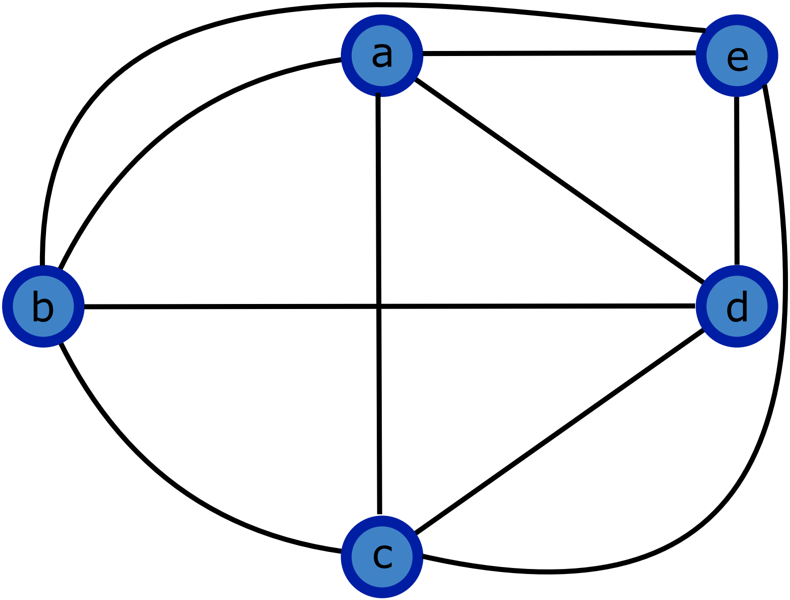
Examples of Euler Tours



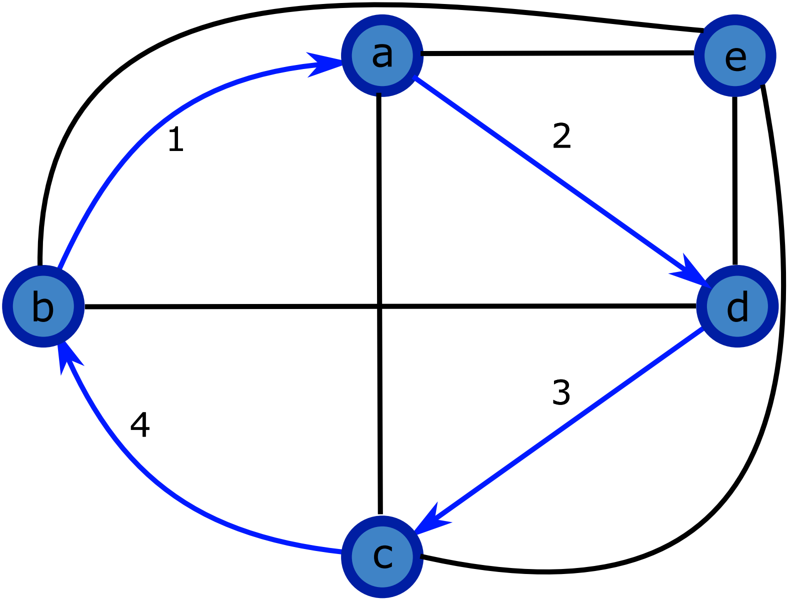
Examples of Euler Tours



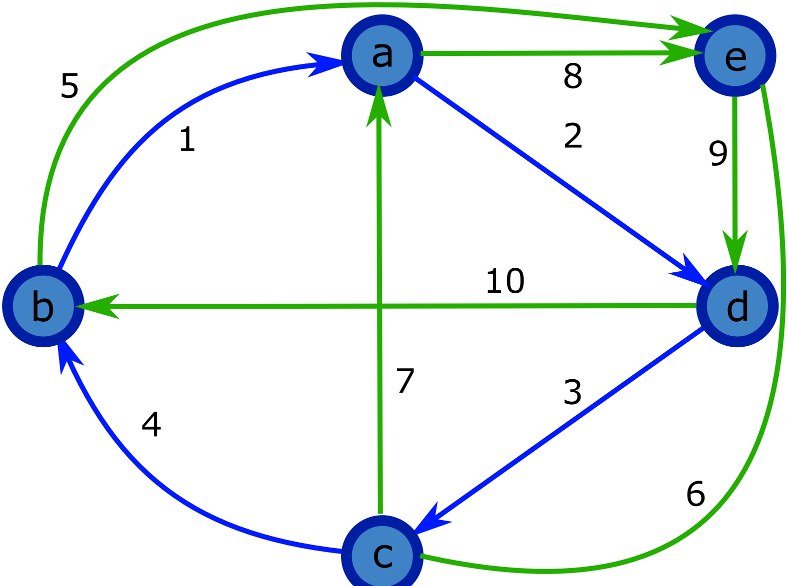
Examples of Euler Tours



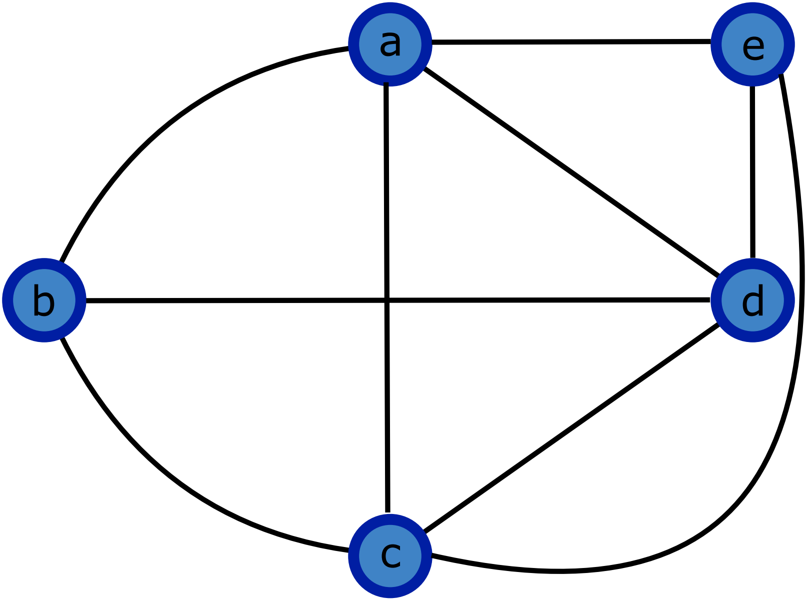
Examples of Euler Tours



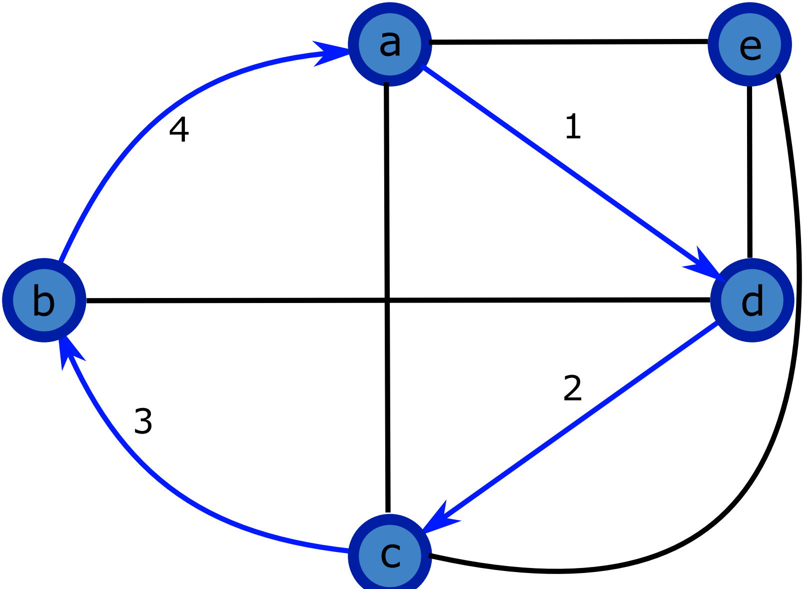
Examples of Euler Tours



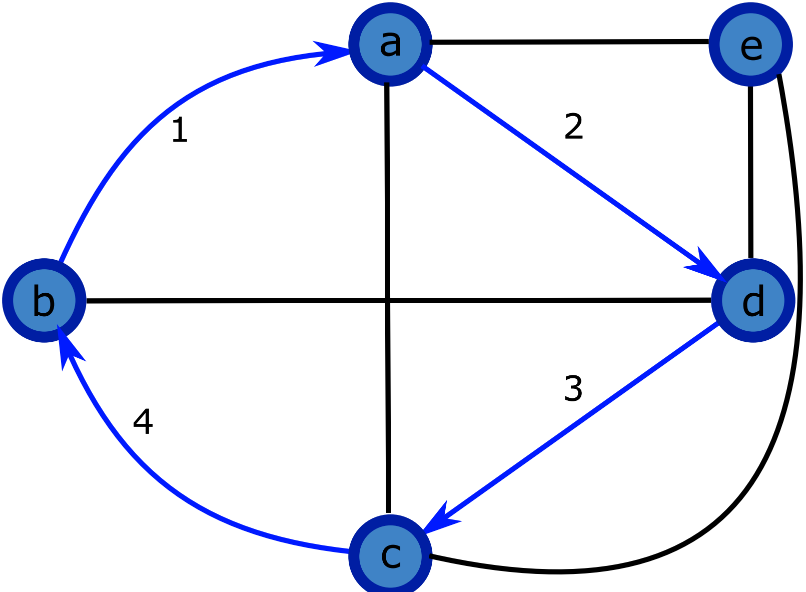
Examples of Euler Tours



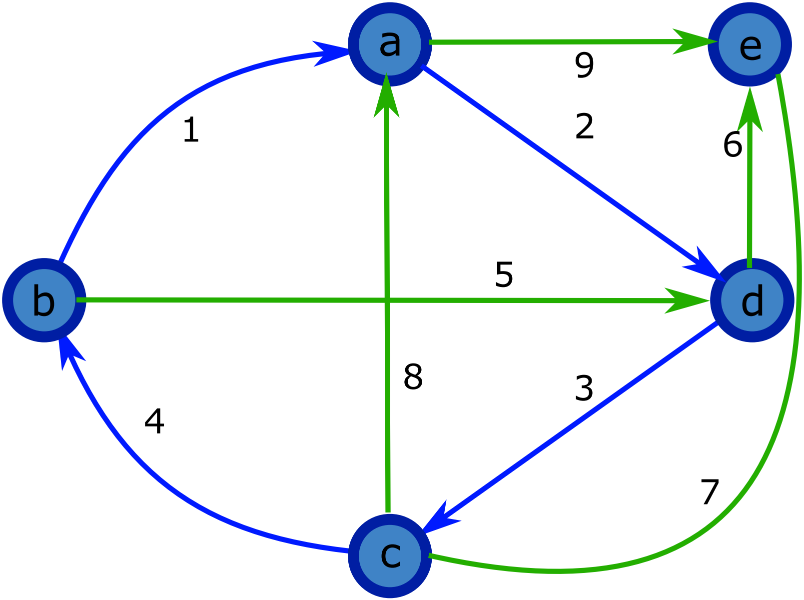
Examples of Euler Tours



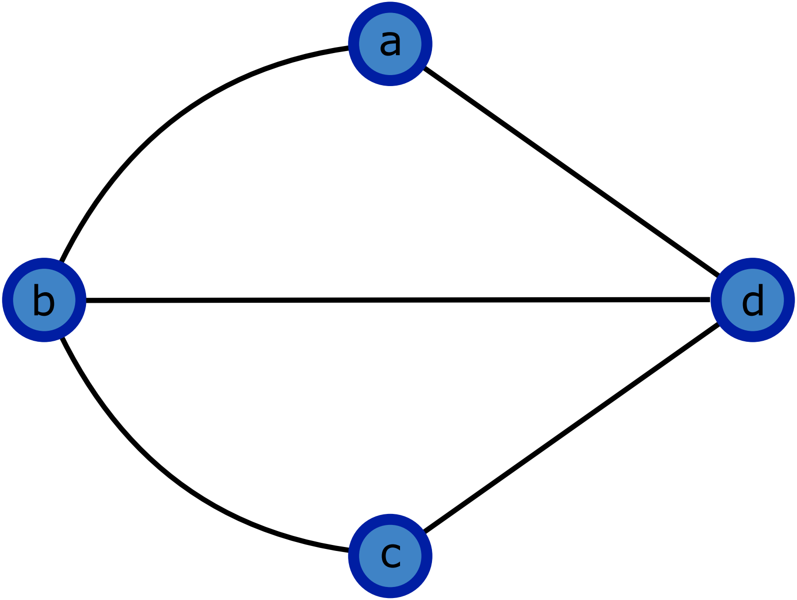
Examples of Euler Tours



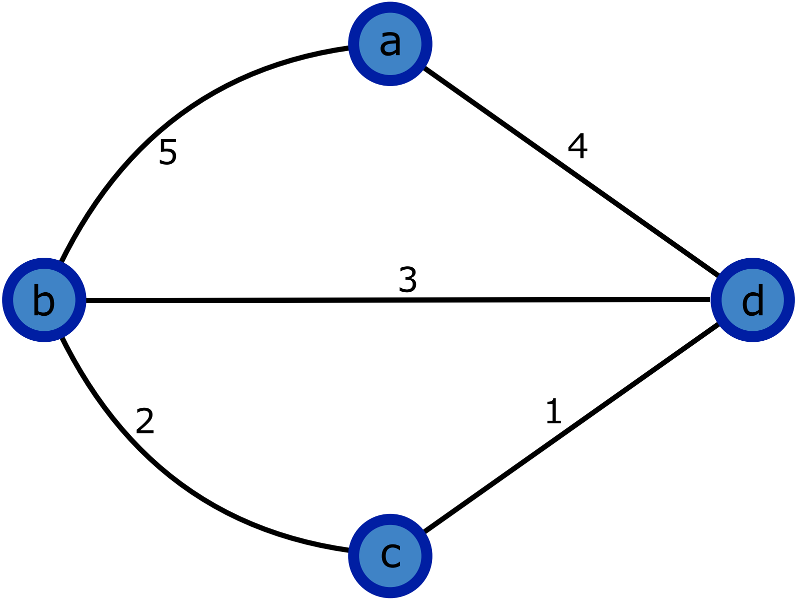
Examples of Euler Tours



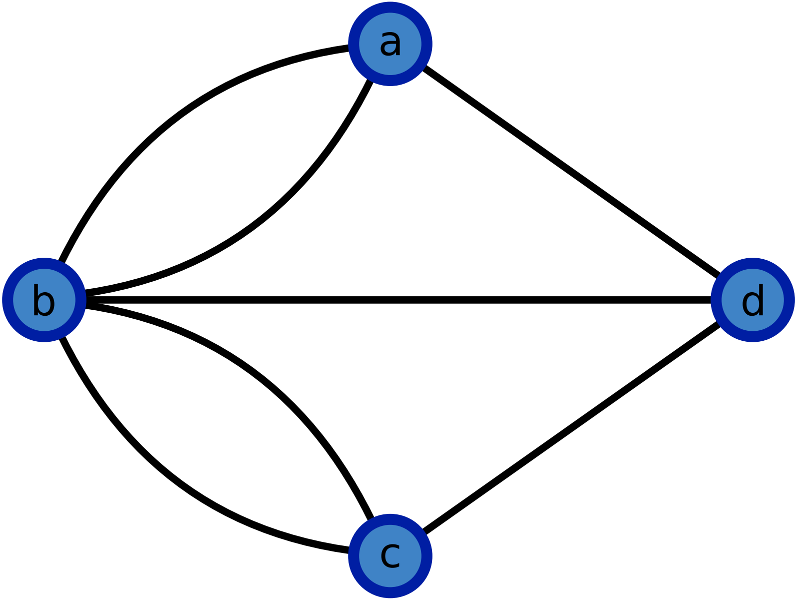
Examples of Euler Tours



Examples of Euler Tours



Examples of Euler Tours



What Euler Proved

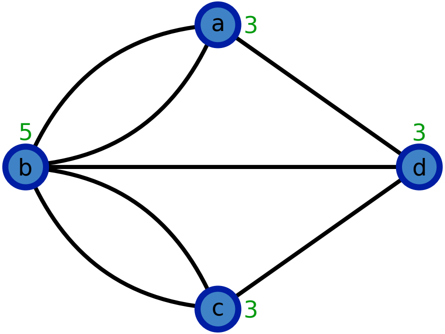
AD GEOMETRIAM SITVS PERTINENTIS. 139

§. 19. Praeterea ſi duo tantum numeri litteris A, B, C etc. adſcripti fuerint impares, reliqui vero omnes pares, tum ſemper deſideratus tranſitus ſuccedet, ſi modo curſus ex regione ad quam pontium impar numerus tendit incipiatur. Si enim pares numeri biſecentur atque etiam impares vnitate aucti, vti praeceptum eſt, ſumma harum medietatum vnitate erit maior quam numerus pontium, ideoque aequalis ipſi numero praefixo. Ex hocque porro perſpicitur, ſi quatuor vel ſex vel octo etc. fuerint numeri impares in ſecunda columna, tum ſummam numerorum tertiae columnae maiorem fore numero praefixo, eumque excedere vel vnitate, vel binario vel ternario etc. et idcirco tranſitus fieri nequit.

§. 20. Caſu ergo quocunque propoſito ſtatim facillime poterit cognoſci, vtrum tranſitus per omnes pontes ſemel inſtitui queat an non, ope huius regulae. Si fuerint plures duabus regiones, ad quas ducentium pontium numerus eſt impar, tum certo affirmari poteſt, talem tranſitum non dari. Si autem ad duas tantum regiones ducentium pontium numerus eſt impar, tunc tranſitus fieri poterit, ſi modo curſus in altera harum regionum incipiatur. Si denique nulla omnino fuerit regio, ad quam pontes numero impares conducant, tum tranſitus deſiderato modo inſtitui poterit, in quacunque regione ambulandi initium ponatur. Hac igitur data regula problemati propoſito pleniſſime ſatisfit.

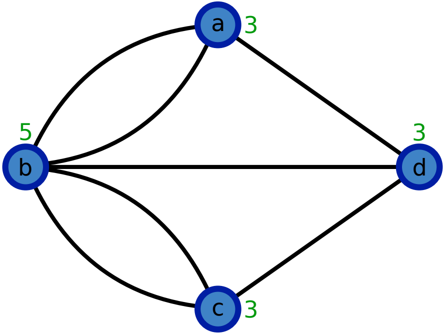
S 2 §. 21.

What Euler Proved (in English)



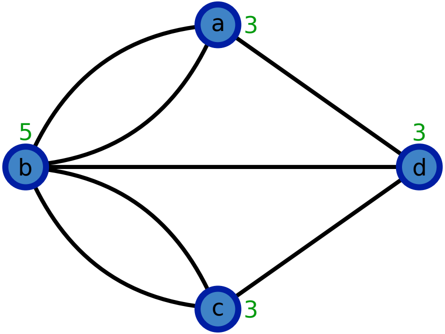
- *Degree $d(v)$ of a node v* is the number of edges incident on it.

What Euler Proved (in English)



- *Degree $d(v)$ of a node v* is the number of edges incident on it.
- Euler's conclusion:
  1. If there are more than two nodes with odd degree, then the graph has no Eulerian tour.
  2. If exactly two nodes have odd degree, then there is tour that starts at one of these nodes and ends at the other node.
  3. If all nodes have even degree, then there exists a tour starting at any node.

What Euler Proved (in English)



- *Degree $d(v)$ of a node v* is the number of edges incident on it.
- Euler's conclusion:
  1. If there are more than two nodes with odd degree, then the graph has no Eulerian tour.
  2. If exactly two nodes have odd degree, then there is tour that starts at one of these nodes and ends at the other node.
  3. If all nodes have even degree, then there exists a tour starting at any node. Is there a missing case?! ▸ Poll

What Didn't Euler Prove?

- Implicit assumption: G is connected.

What Didn't Euler Prove?

- Implicit assumption: G is connected.
- Euler's conditions (1741) were necessary. Hierholzer proved their sufficiency (1873).

What Didn't Euler Prove?

- Implicit assumption: G is connected.
- Euler's conditions (1741) were necessary. Hierholzer proved their sufficiency (1873).
- What about constructing such a tour if it exists?

What Didn't Euler Prove?

140 *SOLVTIO PROBLEMATIS AD GEOM. &c.*

§. 21. Quando autem inuentum fuerit talem tranſitum inſtitui poſſe, quaeſtio ſupereſt quomodo curſus ſit dirigendus. Pro hoc ſequenti vtor regula; tollantur cogitatione quoties fieri poteſt, bini pontes, qui ex vna regione in aliam ducunt, quo pacto pontium numerus vehementer plerumque diminuetur, tum quaeratur, quod facile fiet, curſus deſideratus per pontes reliquos, quo inuento pontes cogitatione ſublati hunc ipſum curſum non multum turbabunt, id quod paululum attendenti ſtatim patebit; neque opus eſſe indico plura ad curſus reipſa formandos praecipere.

THEO-

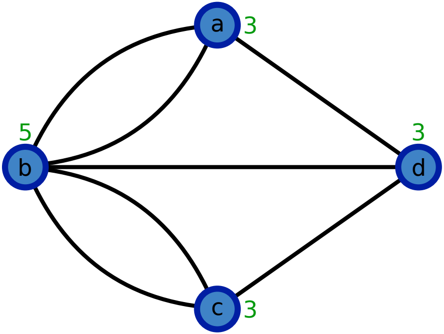
What Didn't Euler Prove?

- Implicit assumption: G is connected.
- Euler's conditions (1741) were necessary. Hierholzer proved their sufficiency (1873).
- What about constructing such a tour if it exists?
  - We must go through the effort to write out a path that is correct.
  - Method to accomplish this was trivial, and Euler did not want to spend a great deal of time on it.

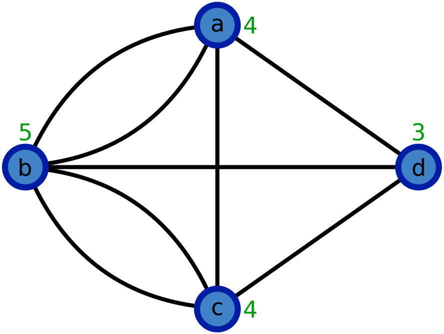
What Didn't Euler Prove?

- Implicit assumption: G is connected.
- Euler's conditions (1741) were necessary. Hierholzer proved their sufficiency (1873).
- What about constructing such a tour if it exists?
  - We must go through the effort to write out a path that is correct.
  - Method to accomplish this was trivial, and Euler did not want to spend a great deal of time on it.
- Hierholzer provided an algorithm.

Hierholzer's Algorithm

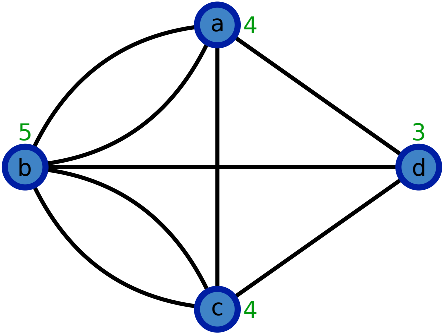


Hierholzer's Algorithm



- If there are two nodes in G with odd degree, call them s and t.
- Otherwise, let s be any node in G.

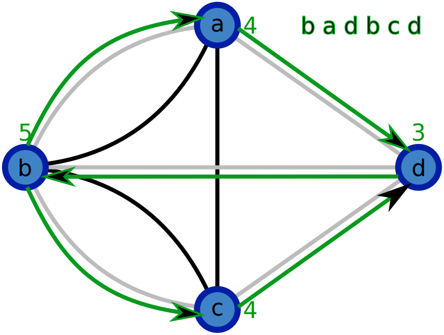
Hierholzer's Algorithm



- If there are two nodes in G with odd degree, call them s and t.
- Otherwise, let s be any node in G.

$u \leftarrow s$ #u denotes the currently-visited node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

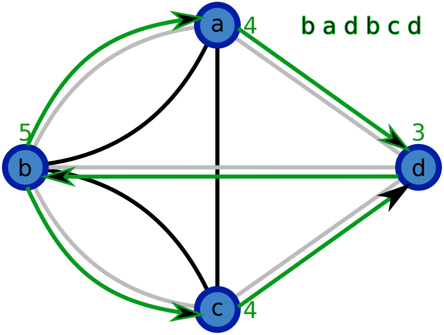
Hierholzer's Algorithm



- If there are two nodes in G with odd degree, call them s and t.
- Otherwise, let s be any node in G.

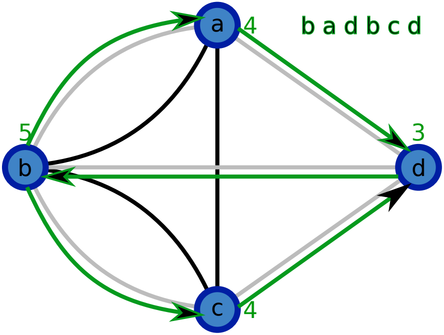
$u \leftarrow s$ #u denotes the currently-visited node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

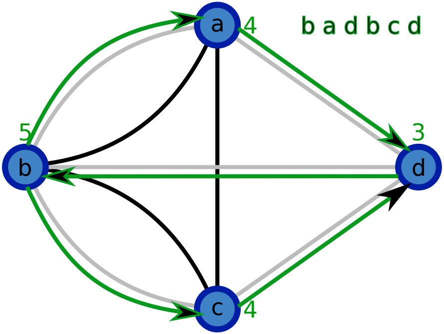
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph?
- If it terminates, what can we say about node u at termination?
- Will all edges of G have been traversed upon termination?

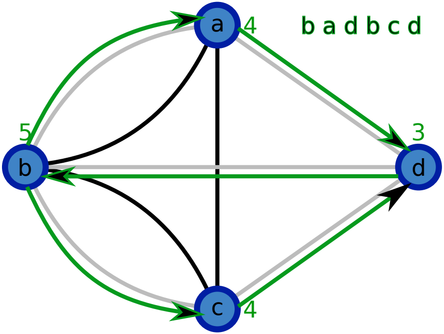
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
- Will all edges of G have been traversed upon termination?

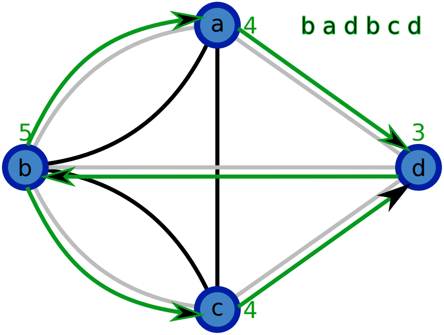
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
  - If G had no nodes of odd degree, then $u = s$.
  - If G had two nodes of odd degree, then $u = t$.
- Will all edges of G have been traversed upon termination?

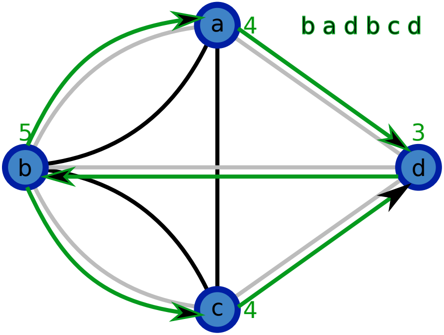
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
  - If G had no nodes of odd degree, then $u = s$.
  - If G had two nodes of odd degree, then $u = t$.
- Will all edges of G have been traversed upon termination? No!

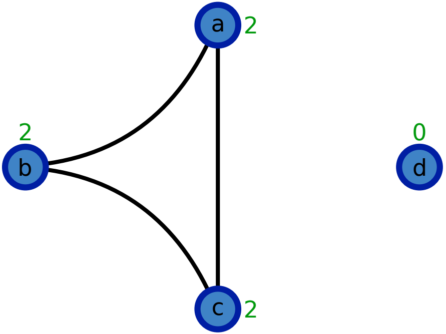
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
  - If G had no nodes of odd degree, then $u = s$.
  - If G had two nodes of odd degree, then $u = t$.
- Will all edges of G have been traversed upon termination? No! Set u to be any node in the remaining graph and repeat.

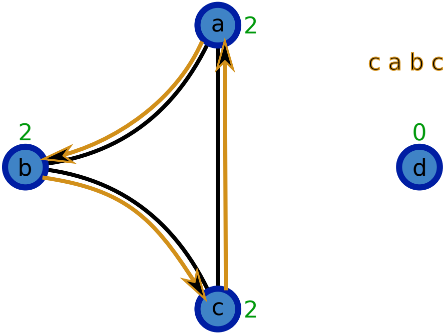
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
  - If G had no nodes of odd degree, then $u = s$.
  - If G had two nodes of odd degree, then $u = t$.
- Will all edges of G have been traversed upon termination? No! Set u to be any node in the remaining graph and repeat.

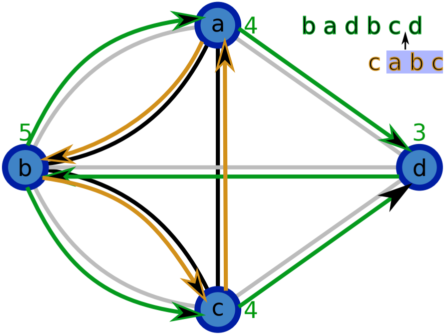
Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
  - If G had no nodes of odd degree, then $u = s$.
  - If G had two nodes of odd degree, then $u = t$.
- Will all edges of G have been traversed upon termination? No! Set u to be any node in the remaining graph and repeat.

Properties of Heilholzer's Algorithm



$u \leftarrow s$
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

- Will the algorithm terminate for every connected graph? Yes, because we traverse a new edge in each iteration.
- If it terminates, what can we say about node u at termination?
  - If G had no nodes of odd degree, then $u = s$.
  - If G had two nodes of odd degree, then $u = t$.
- Will all edges of G have been traversed upon termination? No! Set u to be any node in the remaining graph and repeat.
- Algorithm's running time is $O(|V| + |E|)$, i.e., linear in the size of G.

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n - 1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?		
Insert edge (i, j)		
Delete edge (i, j)		
Iterate over all nbours of node i		
Space used		

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?	$O(1)$ time	
Insert edge (i, j)	$O(1)$ time	
Delete edge (i, j)	$O(1)$ time	
Iterate over all nbours of node i		
Space used		

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?	$O(1)$ time	
Insert edge (i, j)	$O(1)$ time	
Delete edge (i, j)	$O(1)$ time	
Iterate over all nbours of node i	$O(n)$ time	
Space used		

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$. ▸ Poll

Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?	$O(1)$ time	
Insert edge (i, j)	$O(1)$ time	
Delete edge (i, j)	$O(1)$ time	
Iterate over all nbours of node i	$O(n)$ time	
Space used	$O(n^2)$	

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?	$O(1)$ time	$O(d(i))$ time
Insert edge (i, j)	$O(1)$ time	$O(1)$ time
Delete edge (i, j)	$O(1)$ time	$O(d(i) + d(j))$ time
Iterate over all nbours of node i	$O(n)$ time	
Space used	$O(n^2)$	

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

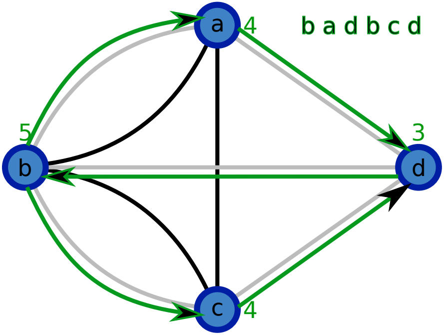
Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?	$O(1)$ time	$O(d(i))$ time
Insert edge (i, j)	$O(1)$ time	$O(1)$ time
Delete edge (i, j)	$O(1)$ time	$O(d(i) + d(j))$ time
Iterate over all nbours of node i	$O(n)$ time	$O(d(i))$ time
Space used	$O(n^2)$	

Representing Undirected Graphs

- Graph $G = (V, E)$ has two input parameters: $|V| = n, |E| = m$.
  - Size of the graph is defined to be $m + n$.
  - Strive for algorithms whose running time is linear in graph size, i.e., $O(m + n)$.
- Assume $V = \{1, 2, \ldots, n-1, n\}$.
- *Adjacency matrix*: $n \times n$ Boolean matrix, where the entry in row i and column j is 1 iff the graph contains the edge (i, j).
- *Adjacency list*: array `Adj`, where `Adj`$[v]$ stores a linked list of all nodes adjacent to v.
  - An edge $e = (u, v)$ appears twice: in `Adj`$[u]$ and `Adj`$[v]$.

Operation/Space	**Adj. matrix**	**Adj. list**
Is (i, j) an edge?	$O(1)$ time	$O(d(i))$ time
Insert edge (i, j)	$O(1)$ time	$O(1)$ time
Delete edge (i, j)	$O(1)$ time	$O(d(i) + d(j))$ time
Iterate over all nbours of node i	$O(n)$ time	$O(d(i))$ time
Space used	$O(n^2)$	$O(n + \sum_{v \in G} d(v)) = O(n + m)$

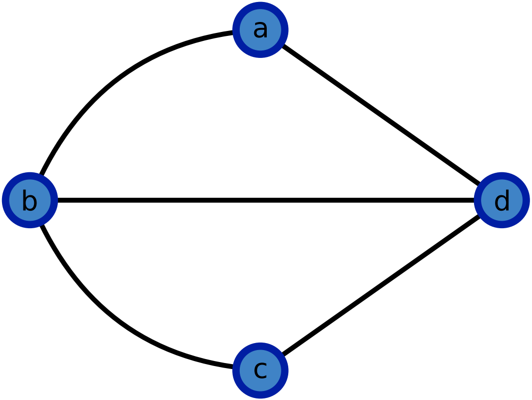
Running Time of Hierholzer's Algorithm



- If there are two nodes in G with odd degree, call them s and t.
- Otherwise, let s be any node in G.

$u \leftarrow s$ #u denotes the currently-visited node.
while $d(u) > 0$ **do**
  Output u.
  Let v be a neighbour of u.
  Delete the edge (u, v) from G.
  $u \leftarrow v$
end while

Visiting Nodes Rather than Edges

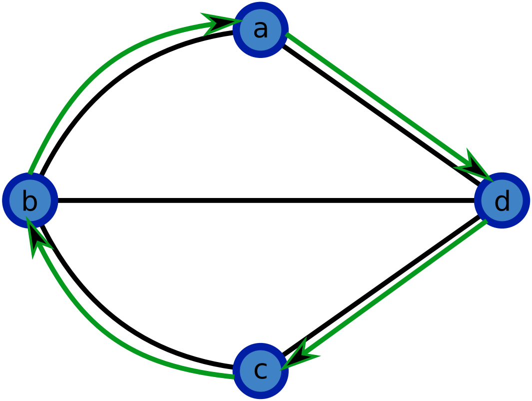


EULERIAN TOUR

Given an undirected graph $G(V, E)$,

construct an *Eulerian tour*, i.e., a path in G that traverses each edge in E exactly once, if such a tour exists.

Visiting Nodes Rather than Edges

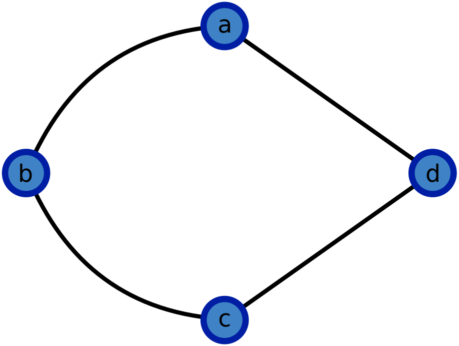


Hamiltonian cycle

Given an undirected graph $G(V, E)$,

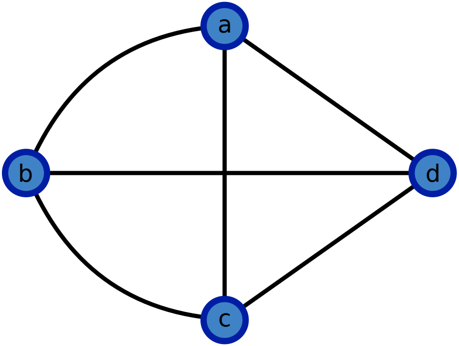
construct an *Hamiltonian cycle*, i.e., a cycle in G that traverses each node in V exactly once, if such a tour exists.

Conditions for Existence of Hamiltonian Cycle



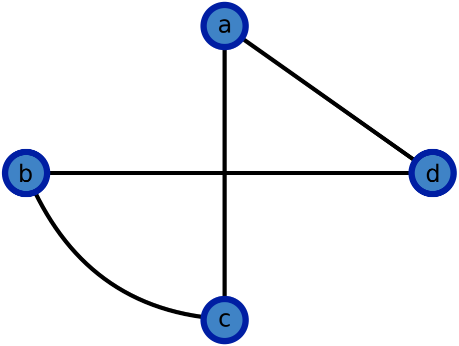
- G has a Hamiltonian cycle if G is a cycle.
- An n-node graph G has a Hamiltonian cycle

Conditions for Existence of Hamiltonian Cycle



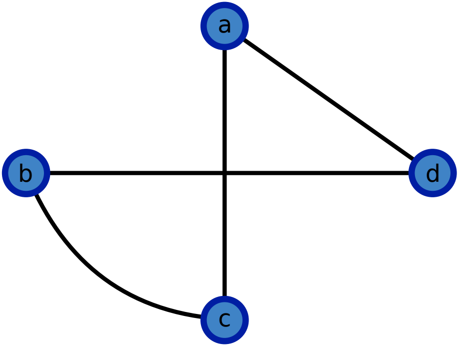
- G has a Hamiltonian cycle if G is a cycle.
- An n-node graph G has a Hamiltonian cycle
  - if each node has degree $n-1$.

Conditions for Existence of Hamiltonian Cycle



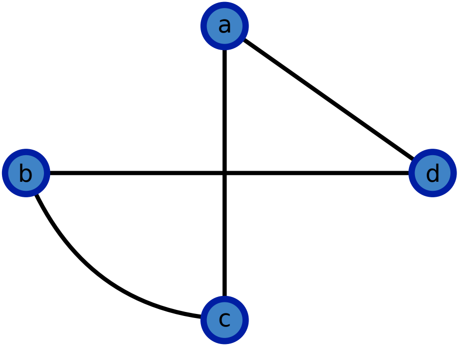
- G has a Hamiltonian cycle if G is a cycle.
- An n-node graph G has a Hamiltonian cycle
  - if each node has degree $n-1$.
  - each node has degree $n-2$.

Conditions for Existence of Hamiltonian Cycle



- G has a Hamiltonian cycle if G is a cycle.
- An n-node graph G has a Hamiltonian cycle
  - if each node has degree $n - 1$.
  - each node has degree $n - 2$.
  - each node has degree $\geq n/2$ (Dirac, 1952).

Conditions for Existence of Hamiltonian Cycle



- G has a Hamiltonian cycle if G is a cycle.
- An n-node graph G has a Hamiltonian cycle
  - if each node has degree $n-1$.
  - each node has degree $n-2$.
  - each node has degree $\geq n/2$ (Dirac, 1952).
  - two disconnected nodes with sum of degrees $\geq n$ (Ore, 1952).

Status of Hamiltonian Cycle Problem

HAMILTONIAN CYCLE

Given an undirected graph $G(V, E)$,

construct an *Hamiltonian cycle*, i.e., a cycle in G that traverses each node in V exactly once, if such a tour exists.

- The Hamiltonian cycle problem is NP-complete,

Status of Hamiltonian Cycle Problem

HAMILTONIAN CYCLE

Given an undirected graph $G(V, E)$,

construct an *Hamiltonian cycle*, i.e., a cycle in G that traverses each node in V exactly once, if such a tour exists.

- The Hamiltonian cycle problem is NP-complete, it is very unlikely that we will find a polynomial time algorithm to check if an undirected graph contains such a cycle.

Status of Hamiltonian Cycle Problem

HAMILTONIAN CYCLE

Given an undirected graph $G(V, E)$,

construct an *Hamiltonian cycle*, i.e., a cycle in G that traverses each node in V exactly once, if such a tour exists.

- The Hamiltonian cycle problem is NP-complete, it is very unlikely that we will find a polynomial time algorithm to check if an undirected graph contains such a cycle.
- Algorithms for computing Hamiltonian cycle:

Status of Hamiltonian Cycle Problem

HAMILTONIAN CYCLE

Given an undirected graph $G(V, E)$,

construct an *Hamiltonian cycle*, i.e., a cycle in G that traverses each node in V exactly once, if such a tour exists.

- The Hamiltonian cycle problem is NP-complete, it is very unlikely that we will find a polynomial time algorithm to check if an undirected graph contains such a cycle.
- Algorithms for computing Hamiltonian cycle:
  - Brute force: try all permutations. Running time is $O(n^2 n!)$.

Status of Hamiltonian Cycle Problem

HAMILTONIAN CYCLE

Given an undirected graph $G(V, E)$,

construct an *Hamiltonian cycle*, i.e., a cycle in G that traverses each node in V exactly once, if such a tour exists.

- The Hamiltonian cycle problem is NP-complete, it is very unlikely that we will find a polynomial time algorithm to check if an undirected graph contains such a cycle.
- Algorithms for computing Hamiltonian cycle:
  - Brute force: try all permutations. Running time is $O(n^2 n!)$.
  - Dynamic programming: running time of $O(n^2 2^n)$ (Held and Karp 1962).
  - Fastest known algorithm runs in time $O(1.657^n)$ (Björklund 2010).