Hamiltonian Graph Traveling Salesman Problem and NP-Completeness

Dr. Rajiv Misra

Associate Professor

Dept. of Computer Science & Engg.

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Preface

Recap of Previous Lecture:

- In previous lecture, we have discussed the characterization of Line Graphs, Edge-coloring, Chromatic index, Multiplicity and 1-factorization.

Content of this Lecture:

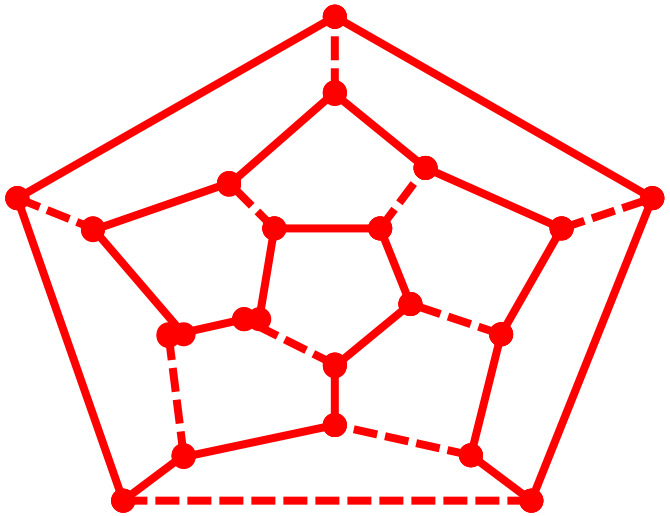
- In this lecture, we will discuss Hamiltonian Graph, Travelling Salesman Problem and NP-Completeness.

Hamiltonian Graph

Hamiltonian Cycles

- Studied first by **Kirkman [1856],** Hamiltonian cycles are named for **Sir William Hamiltonian**, who described a game on the graph of the dodecahedron in which one player specifies a 5-vertex path and the other must extend it to a spanning cycle.

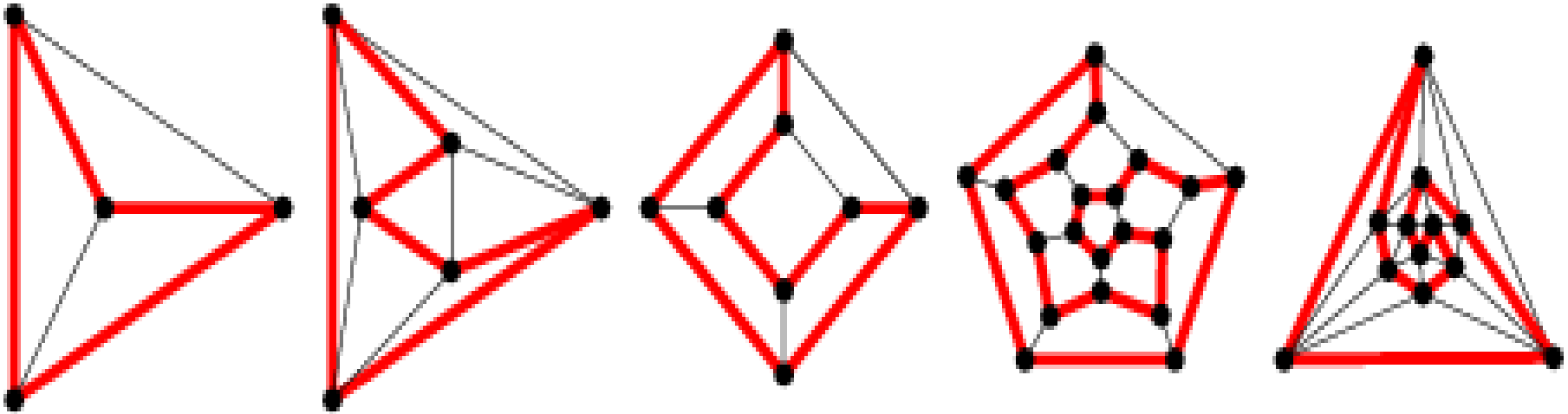
- The game was marketed as the "**Traveller's Dodecahedron**", a wooden version in which the vertices were named for 20 important cities.



Definition: Hamiltonian Graph

A **Hamiltonian graph** is a graph with a spanning cycle, also called a **Hamiltonian cycle**.

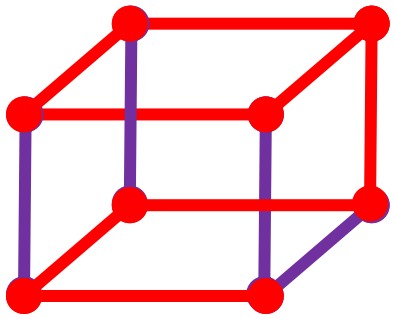
Example:



Hamiltonian Graphs

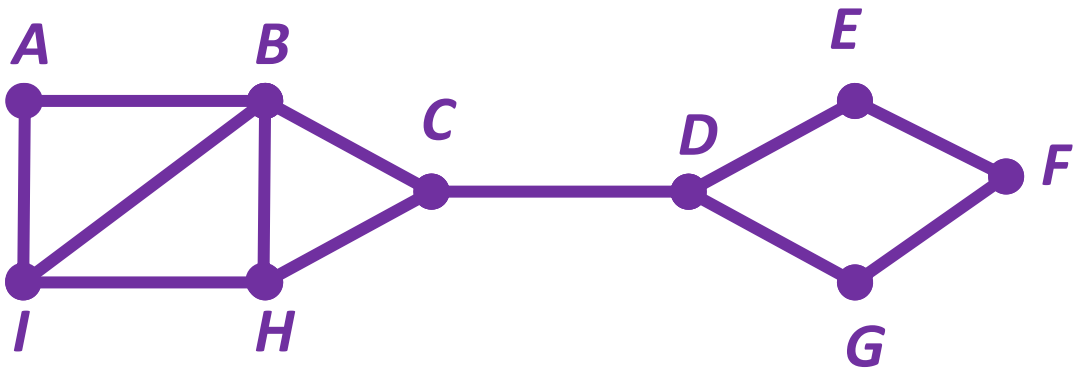
- **Hamiltonian Path:** Path that covers every vertex once
- **Hamiltonian Cycle:** Cycle that covers every vertex once
- **Hamiltonian Graph:** A graph containing a Hamiltonian Cycle is called a Hamiltonian Graph
- **Nonhamiltonian Graph:** A nonhamiltonian graph is a graph that is not Hamiltonian

Example: Hamiltonian

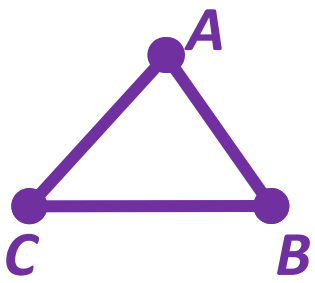


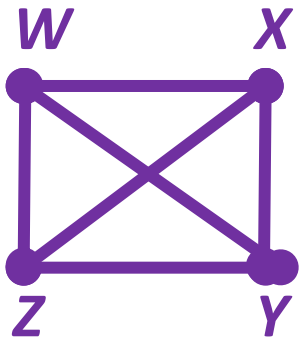
- Hamiltonian cycle can be converted to a Hamiltonian path by removing one edge.

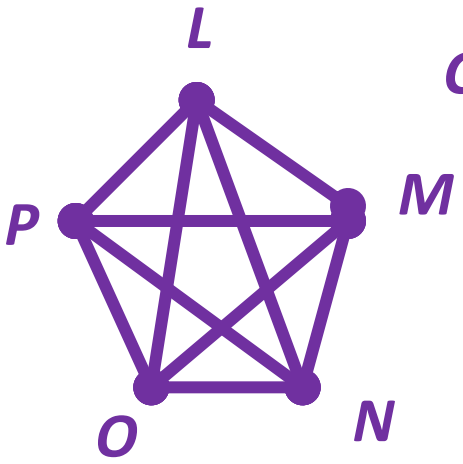
Example: Hamiltonian Path



Example: Hamiltonian Circuit

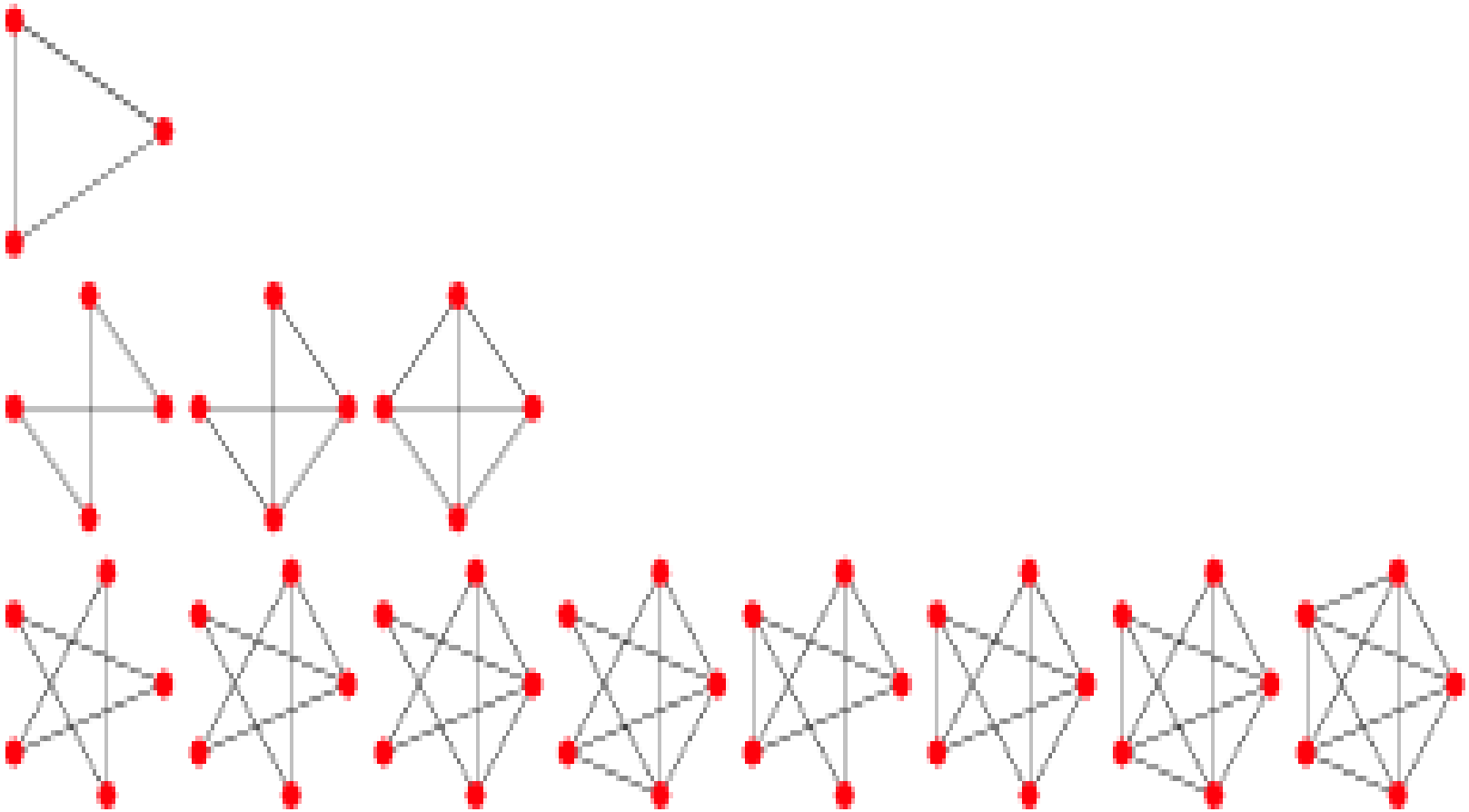






Complete Graphs

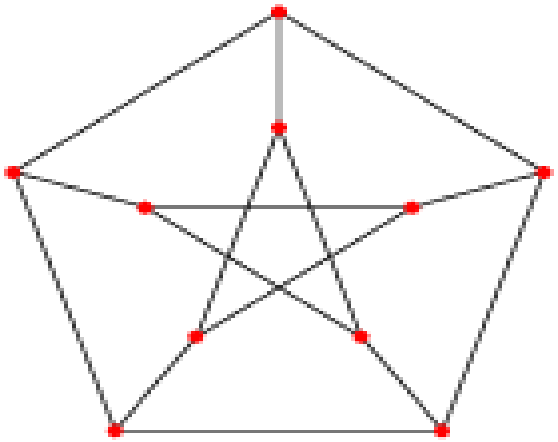
Example: Hamiltonian Graphs



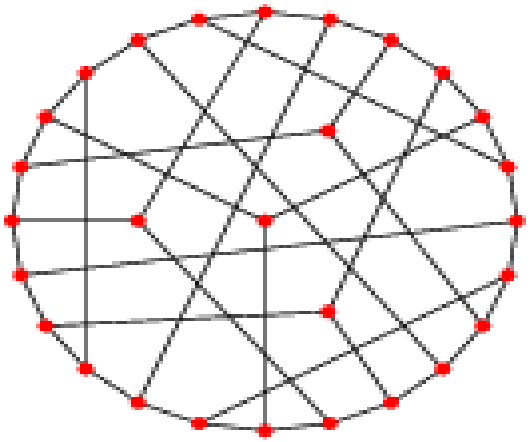
Example: Nonhamiltonian Graphs

2-path graph

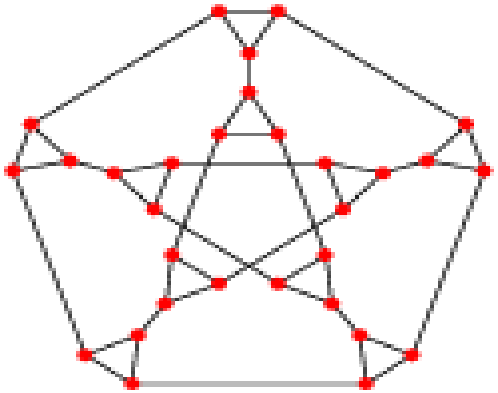
Petersen graph



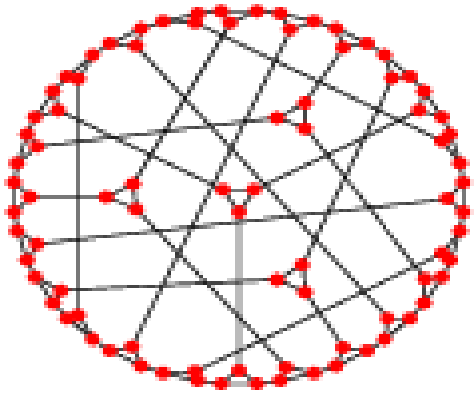
Coxeter graph



triangle-replaced Petersen graph



triangle-replaced Coxeter graph



Example: Nonhamiltonian Graphs

Graph	V(G)
Singleton graph	1
Claw graph	4
Theta-0 graph	7
PETERSEN GRAPH	10
HERSCHEL GRAPH	11
No perfect matching graph	16
First blanuša snark	18
Second blanuša snark	18
Flower snark	20
WALTHER GRAPH	25
COXETER GRAPH	28
Double star snark	30
THOMASSEN GRAPH	34
TUTTE'S GRAPH	46
SZEKERES SNARK	50
MEREDITH GRAPH	70

- Until the 1970s, interest in **Hamiltonian cycles** centered on their relation-ship to the **Four Color Problem**. Later study was stimulated by practical applications and by the issue of complexity.

- No easily testable characterization is known for Hamiltonian graphs; we will study necessary conditions. Loops and multiple edges are irrelevant; a graph is Hamiltonian if and only if the simple graph obtained by keeping one copy of each non-loop edge is Hamiltonian.

- Therefore, in this lecture we will restrict our discussion to simple graphs; this is relevant when discussing conditions involving vertex degrees.

Necessary Conditions

- **Every Hamiltonian graph is 2-connected**, because deleting a vertex leaves a subgraph with a spanning path. Bipartite graphs suggest a way to strengthen this necessary condition.

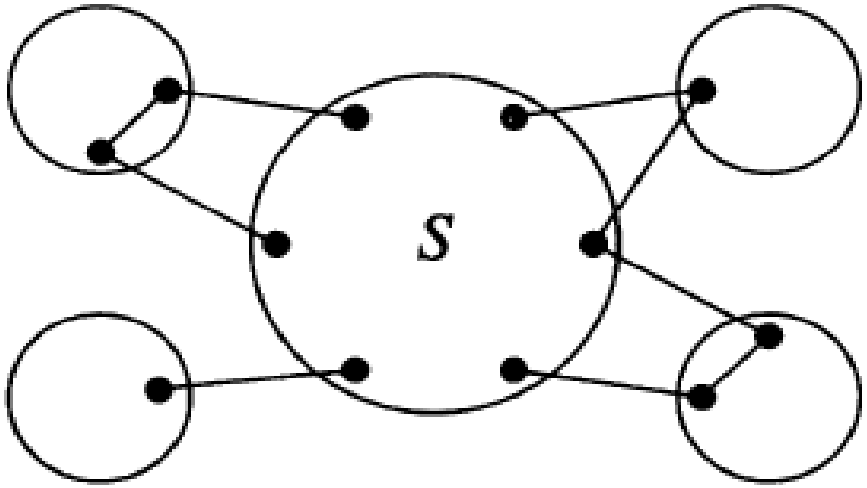
Example: Bipartite graphs. A spanning cycle in a bipartite graph visits the two partite sets alternatively, so there can be no such cycle unless the partite sets have the same size.

Hence $K_{m,n}$ is Hamiltonian only if m = n. Alternatively, we can argue that the cycle returns to different vertices of one partite set after each visit to the other partite set.

Proposition 7.2.3

- If G has a Hamiltonian cycle, then for each nonempty set $S \subseteq V$, the graph G-S has at most |S| components.

Proof: When leaving a component of G-S, a Hamiltonian cycle can go only to S, and the arrivals in S must use distinct vertices of S. Hence S must have at least as many vertices as G-S has components.

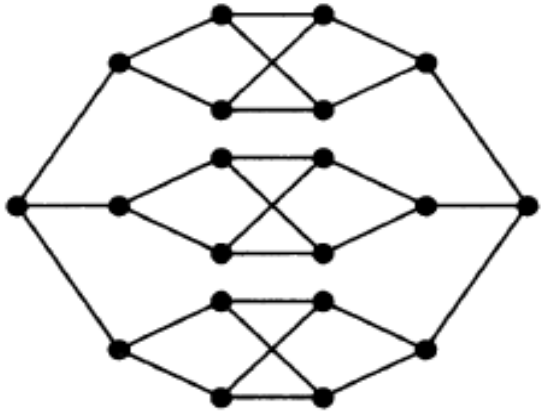


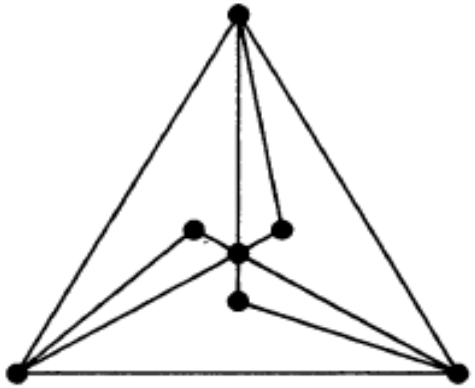
Definition 7.2.4

- Let c(H) denote the number of components of a graph H.
- Thus the necessary condition is that $c(G-S) \leq |S|$ for all $\phi \neq S \subseteq V$.
- This condition gurantees that G is 2-connected (deleting one vertex leaves at most one component), but it does not guarantee a Hamiltonian cycle.

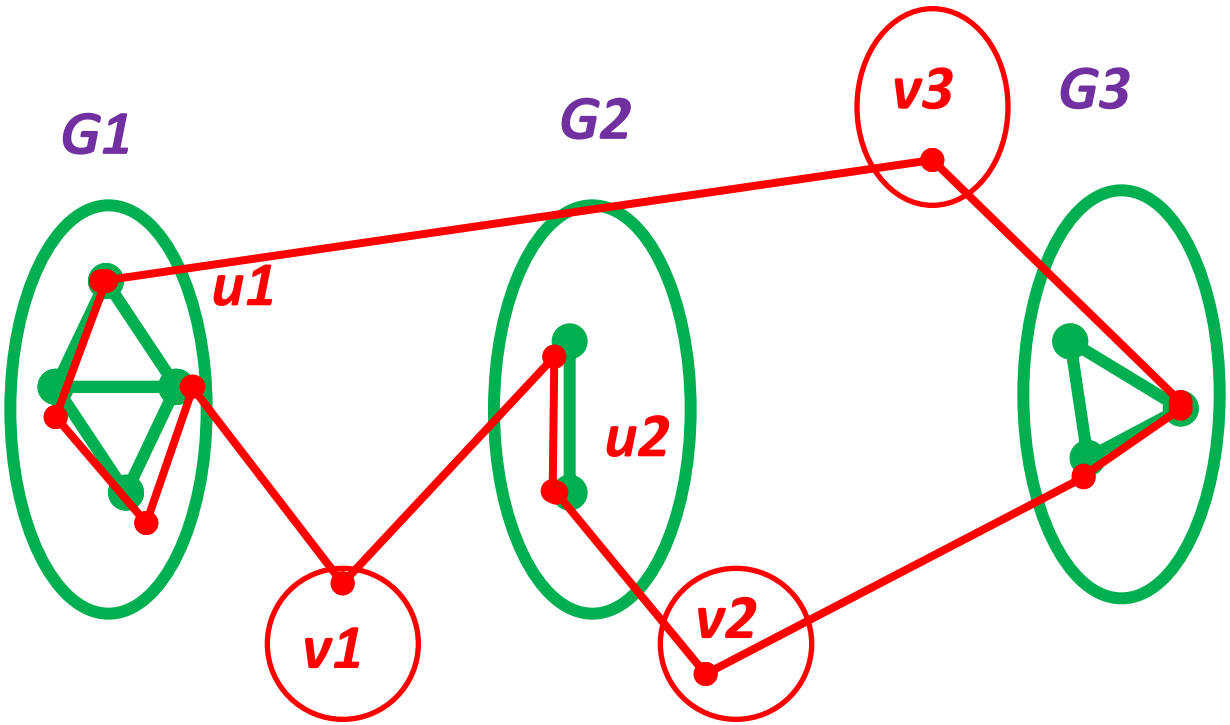
Example 7.2.5

- **Example:** The graph on the left below is bipartite with partite sets of equal size. However, it fails the necessary condition of Proposition 7.2.3. **Hence it is not Hamiltonian.**





- The graph on the right shows that the necessary condition is not sufficient. This graph satisfies the condition but has no spanning cycle. All edges incident to vertices of degree 2 must be used, but in this graph that requires three edges incident to the central vertex.
- The **Petersen graph is another non-Hamiltonian graph** satisfying the condition. We proved in previous lecture that $2C_5$ is the only 2-factor of the Petersen graph, so it has no spanning cycle.



Because C is a cycle and u, vi ϵ E(G) for all i

For each i=1,…,k there is a vi ϵ S |S| ≥ k

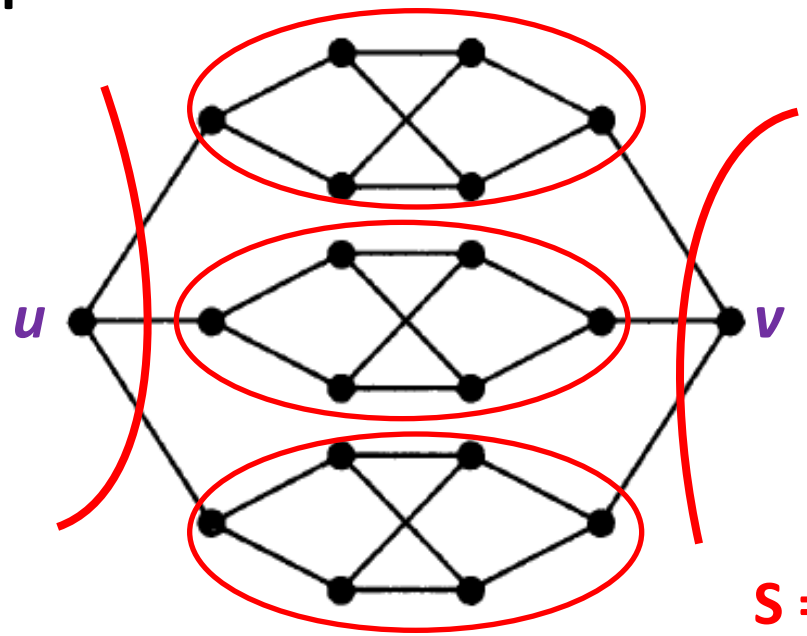
k = c(G-S) ≤ |S|

Example:

Lemma: If G is hamiltonian, then for every nonempty subset $S \subseteq V(G)$, $c(G-S) \leq |S|$

Number of connected components of G-S

Example:



$\phi \neq S \subseteq V(G)$
$C(G-S) > |S|$

Then G is non-hamiltonian

S = {u,v} **c(G-S) = 3**

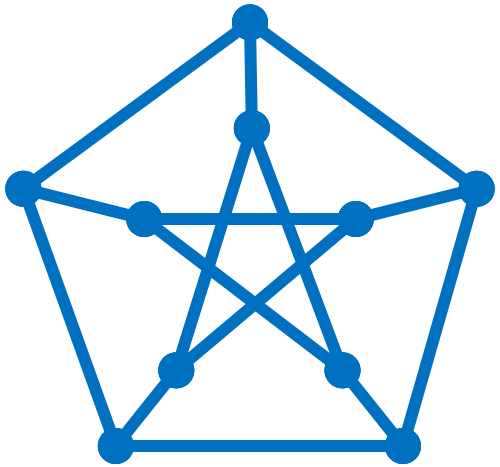
Example: Petersen Graph

Petersen Graph

Satisfies

$\phi \neq S \subseteq V(G)$

$C(G-S) \leq |S|$

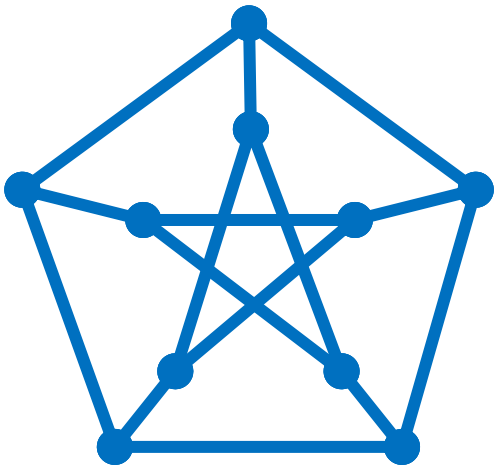


However, it has no Hamiltonian cycle.

Example: Petersen Graph is Not Hamiltonian

1) Find a Hamiltonian path in the Petersen graph

2) Prove that the Petersen graph has no Hamiltonian cycle.

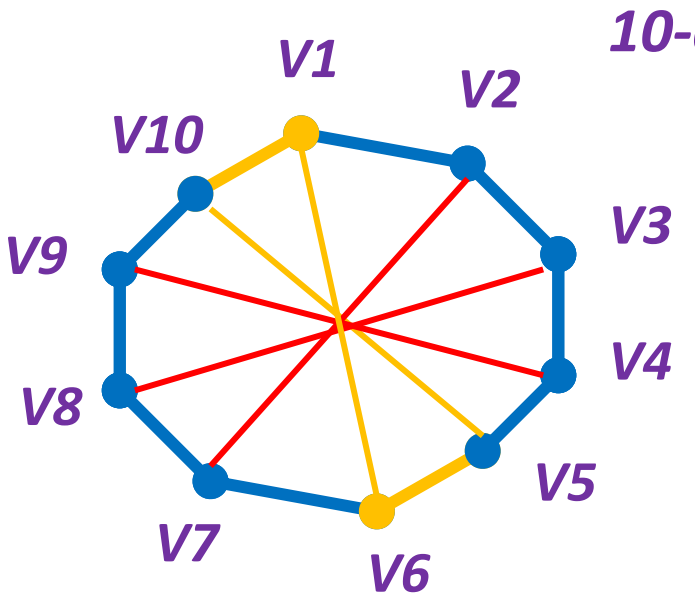


Hamiltonian Path

Observe: The Petersen graph has no cycle on ≤ 4 vertices

Proof (By Contradiction):

- Suppose C is a Hamiltonian cycle of the Petersen graph.



- Then the Petersen graph consists of the 10-cycle C=(v1,v2,….,v10) together with 5 “chord” edges that connect vertices not already adjacent in C.

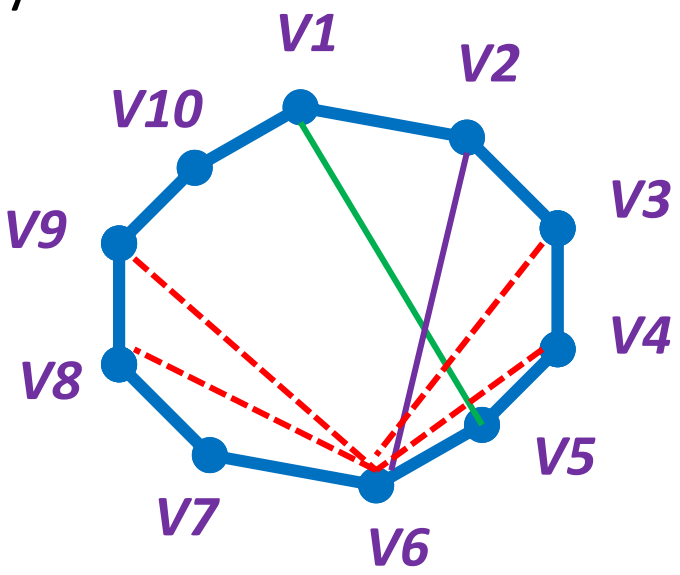
If each chord joins vertices opposite on C then ∃ a 4-cycle

Proof (Continue)

v1v5 ϵ E

There is a chord that joins vertices at distance <5 in C, these vertices must be at distance 4 in C.

WLOG say v1v5ϵ E



Now v6 cannot be incident with any chord edge without making a cycle of length ≤ 4.

The Petersen graph is NOT Hamiltonian.

Traveling Salesman Problem (TSP)

Traveling Salesman Problem (TSP)

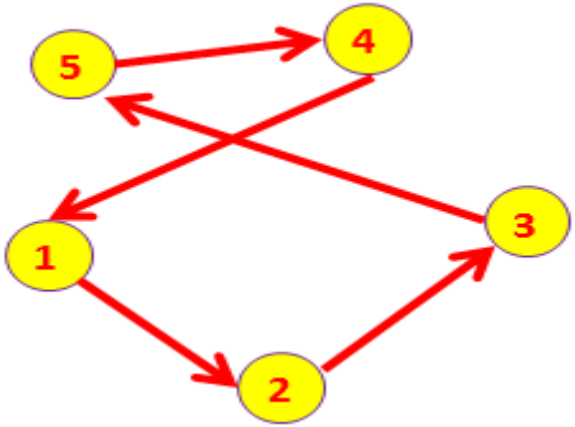
- A salesman plans to visit n-1 other cities and return home. The natural objective is to minimize the total travel time.

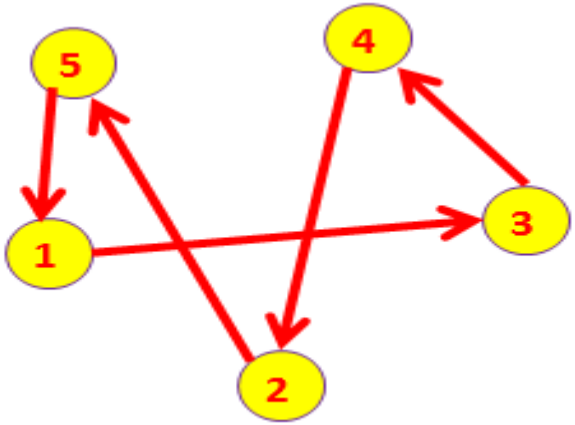
- If we assign each edge of K_n a weight equal to the travel time between the corresponding cities, then we seek the spanning cycle of minimum total weight. This is the famous **Traveling Salesman Problem (TSP).** Seemingly analogous to the Minimum Spanning Tree problem, the TSP as yet has no good algorithm.

Traveling Salesman Problem (TSP)

Traveling Salesman Problem:

- Sales man has to start from one place
- Go to all other city **just once**
- And come back to original city
- Let's see an example
- Say there are five cities then
- TSP is all about finding the least cost (distance) path with above conditions





Why it is computationally difficult ?

In 4 cities problem

In case of four cities, you have four solutions

A-B-C-D-A,

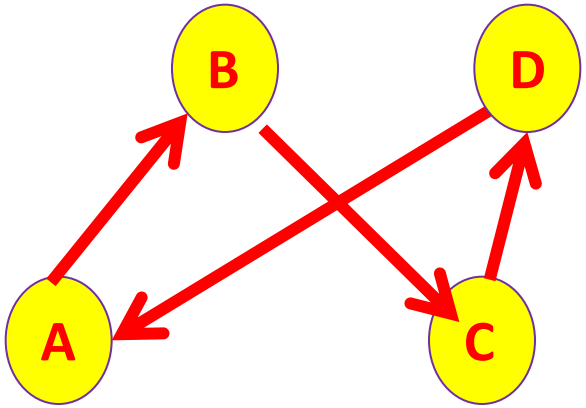
A-B-D-C-A,

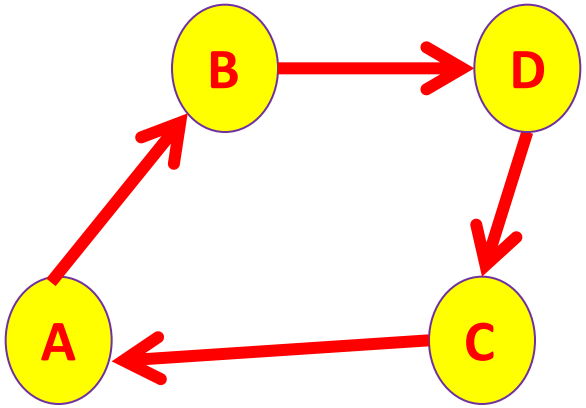
A-D-B-C-A

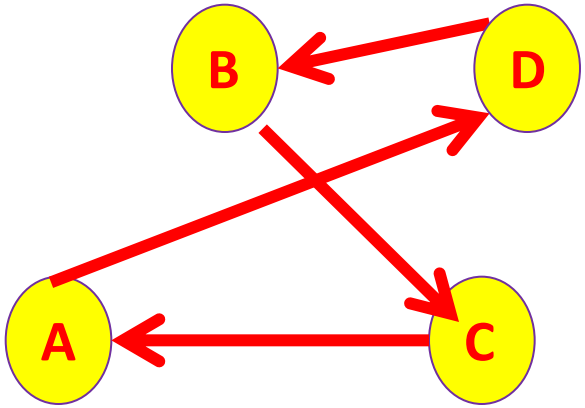
There will no be any other possibilities here

So possibilities are given by (n-1)! / 2

For 3 cities, 1 possibility , 4 cities 3!/2 = 3 possibilities







Heuristics and Bounds

- Our traveling salesman still awaits instruction. NP-completeness does not eliminate the need for an answer.

- We seek heuristic algorithms that find solutions close to optimal. Perhaps we can prove a guarantee about how far from the optimum the result may be. For example, we may be content to have a solution whose cost is at most twice the optimum, if we have an algorithm that can quickly generate such a solution.

- An **approximation algorithm** always generates a solution whose ratio to the optimum is bounded by a constant.

Nearest-Neighbor Heuristic for TSP

- Nevertheless, the greedy algorithm works well on large graphs generated randomly.

- Next we consider simple heuristic for the TSP, where {v1,…,vn} are the vertices and w_{ij} denotes the weight (cost) of edge vivj. From an arbitrary starting vertex, it seems reasonable to move to a new vertex via the least-cost incident edge.

- We iteratively move to the closest unvisited neighbor of the current vertex. This is a “greedy” algorithm and runs quickly. It is the **nearest-neighbor heuristic**.

Example: Failure of the Nearest-Neighbor Heuristic

- Consider a TSP with weight 0 on a Hamiltonian path P, weight n^2 on all other edges incident to the endpoints of P, and weight 1 on all remaining edges.

- This example has many spanning cycles of weight n, but the nearest-neighbor heuristic yields a cycle of weight at least n^2 from any starting vertex.

- Thus the cost of the cycle produced by the algorithm is not bounded by a constant multiple of the optimal cost, and it is not an approximation algorithm.

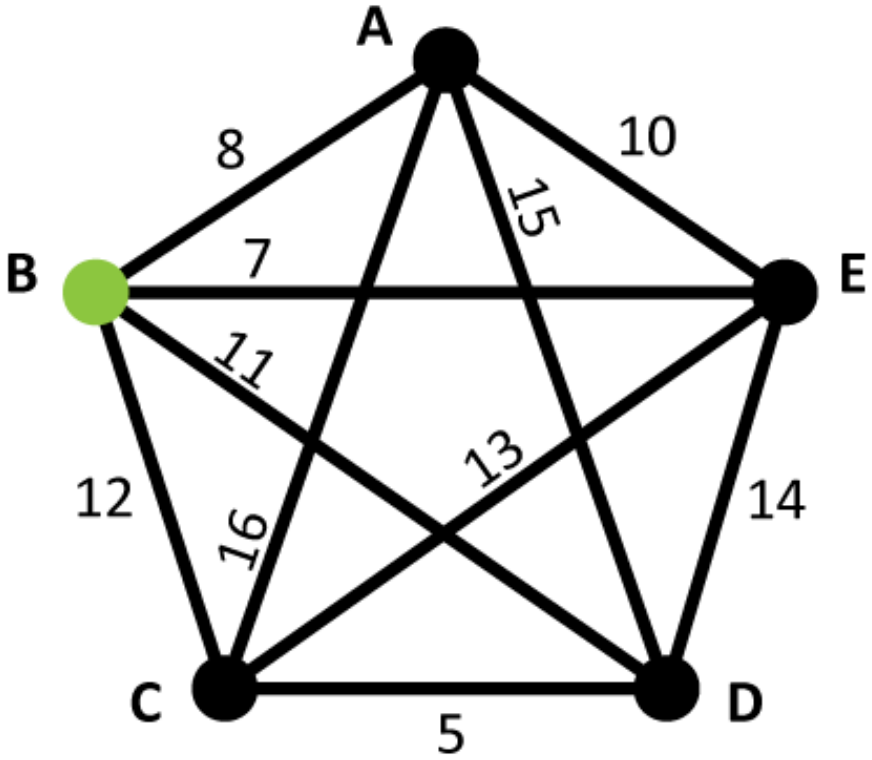
Theorem (Sahni-Gonzalez [1976])

- If there is a constant $c \geq 1$ and a polynomial-time algorithm A such that A produces for each instance of the TSP a spanning cycle with cost at most c times the optimum, then P = NP.

Example: Nearest-Neighbor Algorithm for TSP

- We have to have a starting point

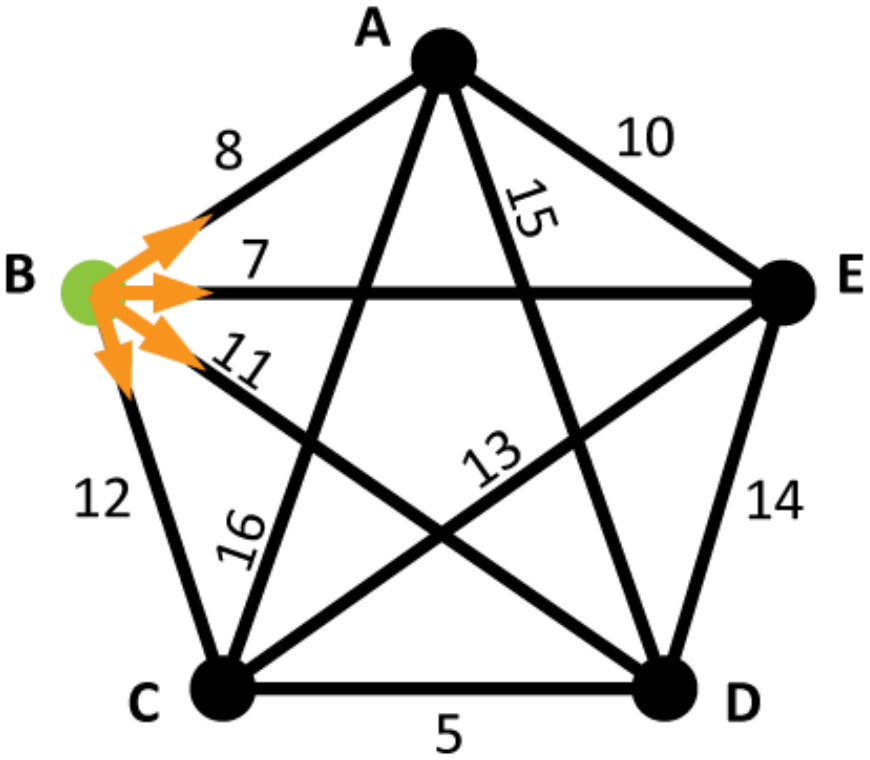
- We will choose our second vertex by finding the “nearest neighbor”



Example: Nearest-Neighbor Algorithm for TSP

- Where do we go first?

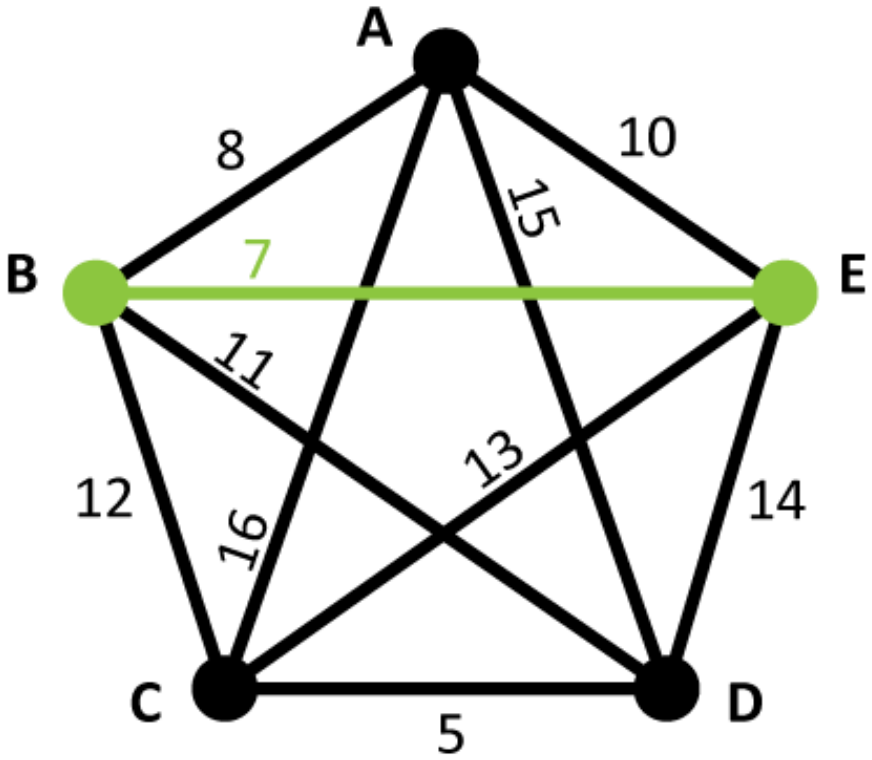
- Choose the cheapest edge



Example: Nearest-Neighbor Algorithm for TSP

- Choose the cheapest edge

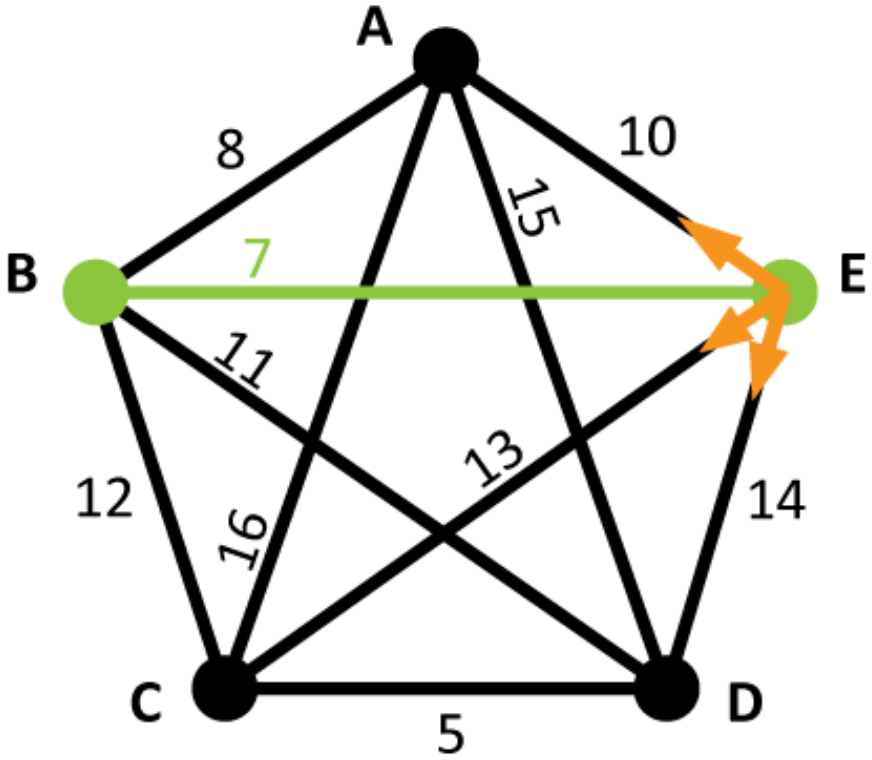
- In this case, we go from B to E (7)



Example: Nearest-Neighbor Algorithm for TSP

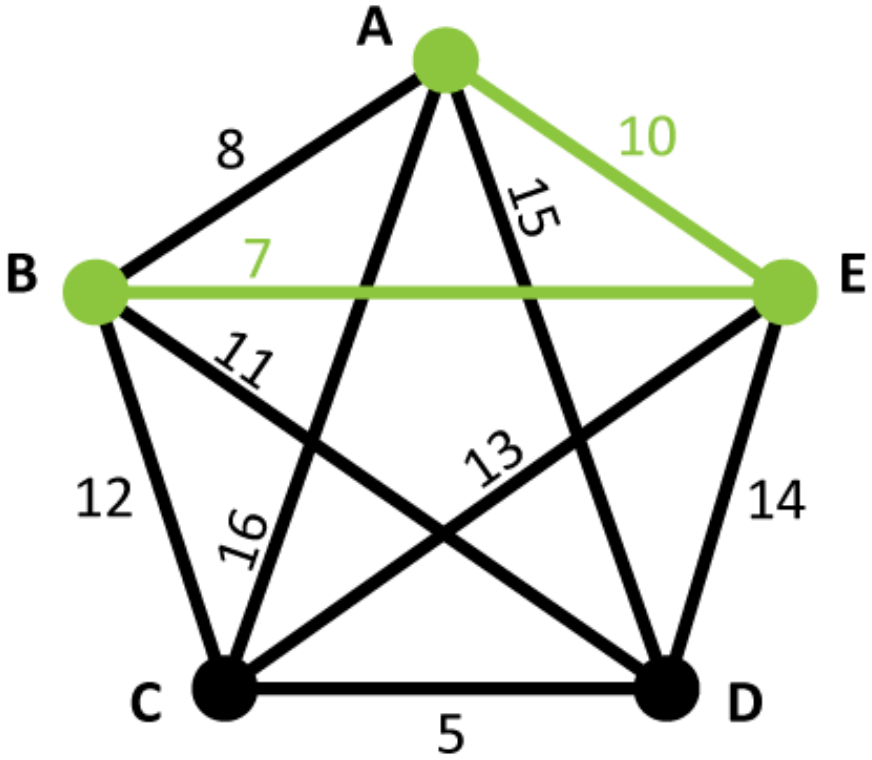
- Now where do we go?

- We can't go back to B



Example: Nearest-Neighbor Algorithm for TSP

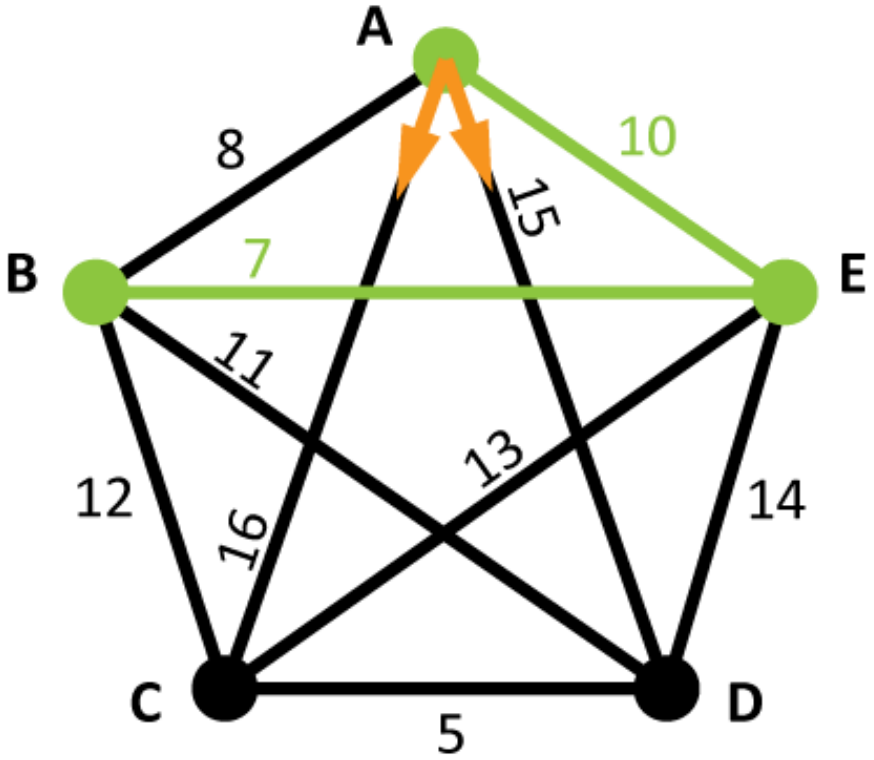
- Now where do we go?
- We can't go back to B
- Again choose the cheapest edge



Example: Nearest-Neighbor Algorithm for TSP

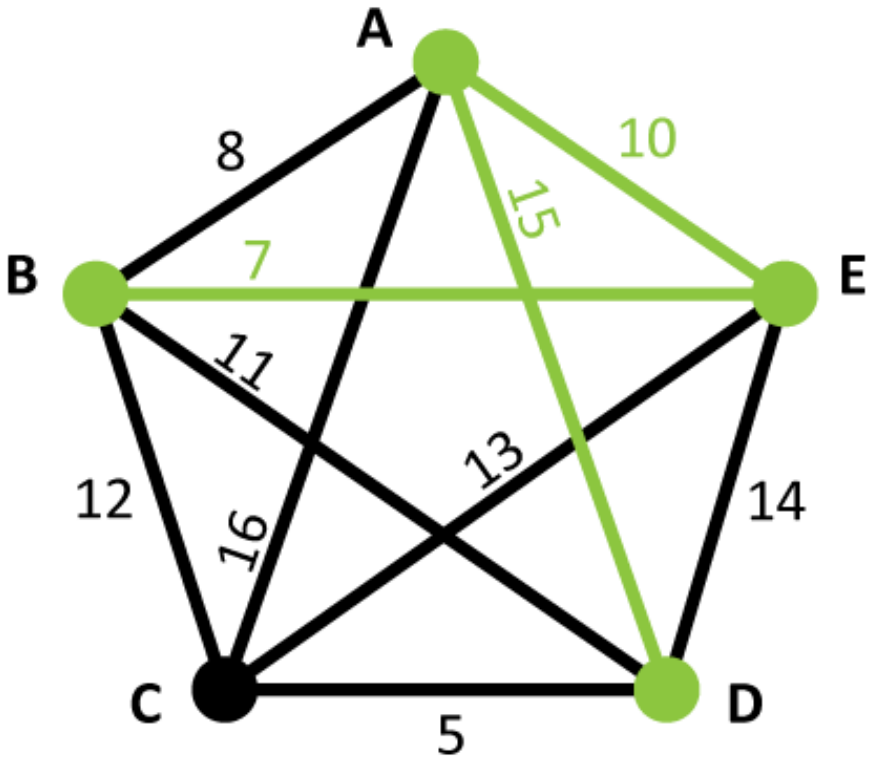
- Now where do we go?

- We can't go back to E, but we also can't go to B



Example: Nearest-Neighbor Algorithm for TSP

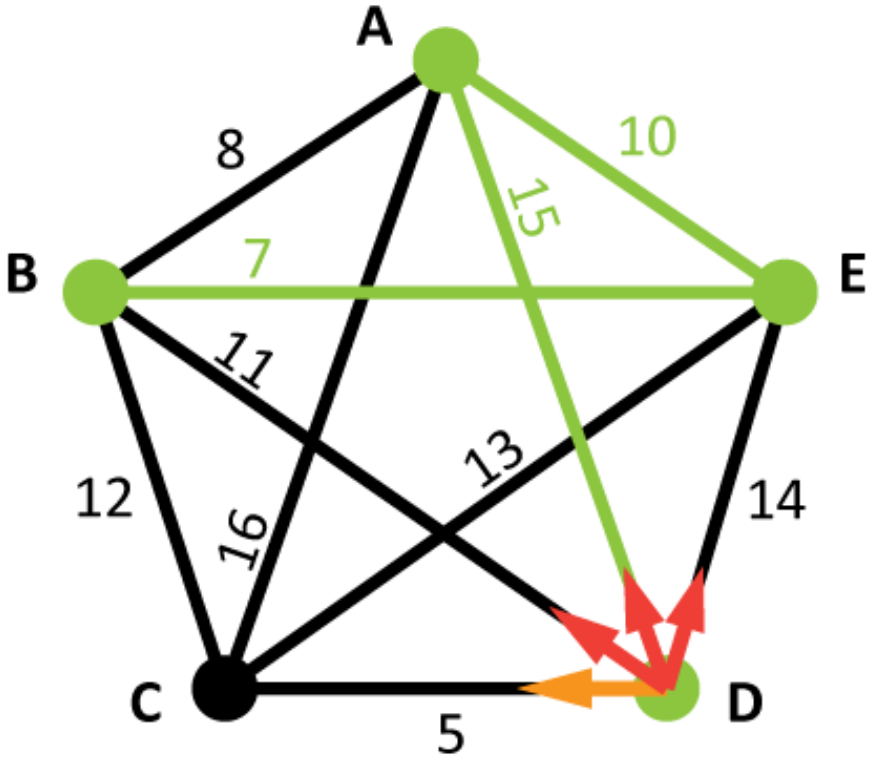
- The rule is “nearest neighbor”: always choose the lowest cost edge, unless that would take you back to a vertex you have already been to



Example: Nearest-Neighbor Algorithm for TSP

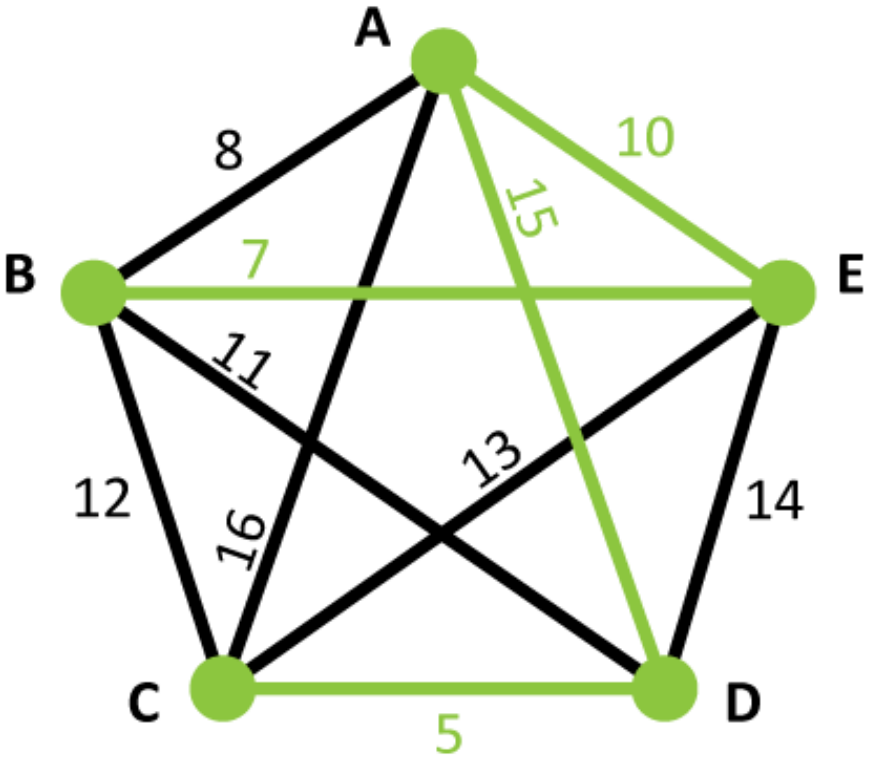
- Now we only have one choice

- We can't go back to A or E, and we can't return to B because that would leave out C



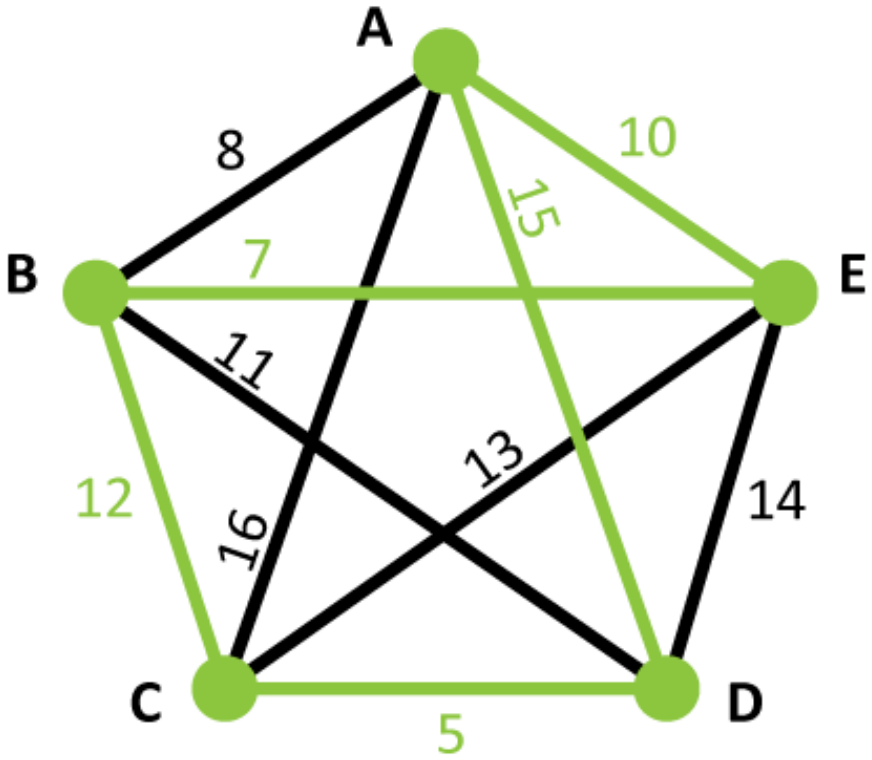
Example: Nearest-Neighbor Algorithm for TSP

- Now we only have one choice
- We can't go back to A or E, and we can't return to B because that would leave out C
- So we must go to C



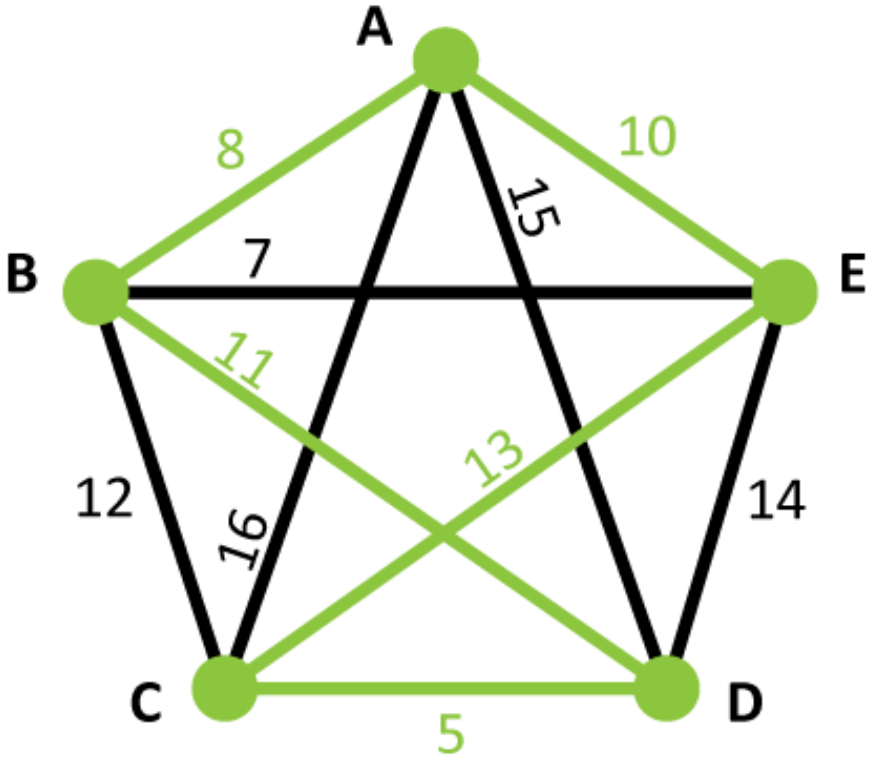
Example: Nearest-Neighbor Algorithm for TSP

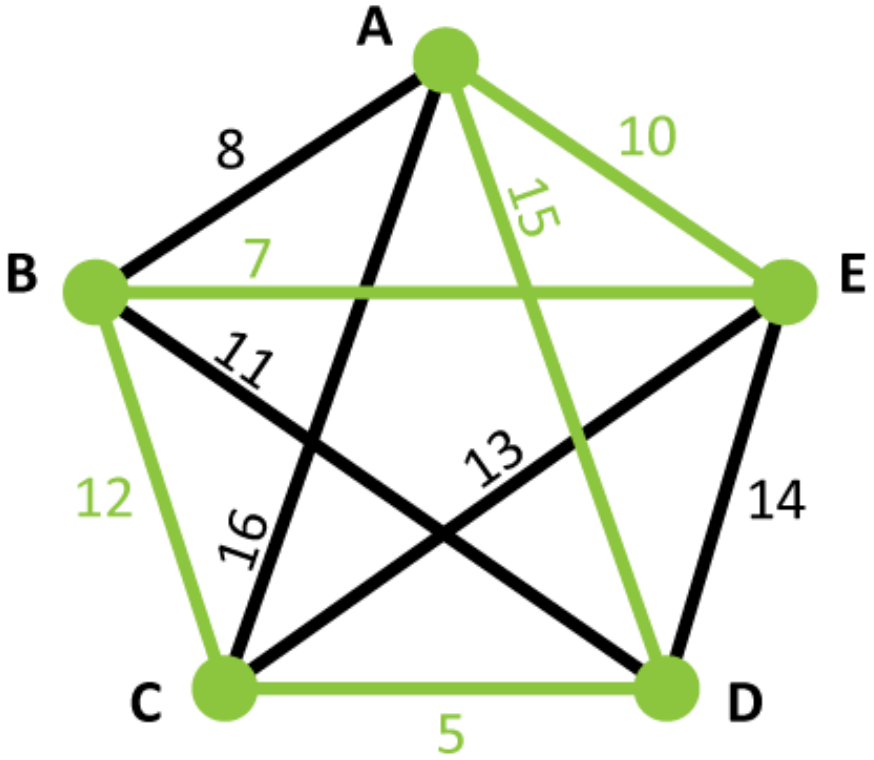
- We have now visited all of the vertices, so we finally return to B
- This circuit has a total cost of 49
- Is it the best circuit?



Example: Nearest-Neighbor Algorithm for TSP

- It is *not* the best! The solution on the left has a total cost of 47





Starting Vertex	Path	Total Cost
A	A-B-E-C-D-A	48
B	B-E-A-D-C-B	49
C	C-D-B-E-A-C	49
D	D-C-B-E-A-D	49
E	E-B-A-D-C-E	48

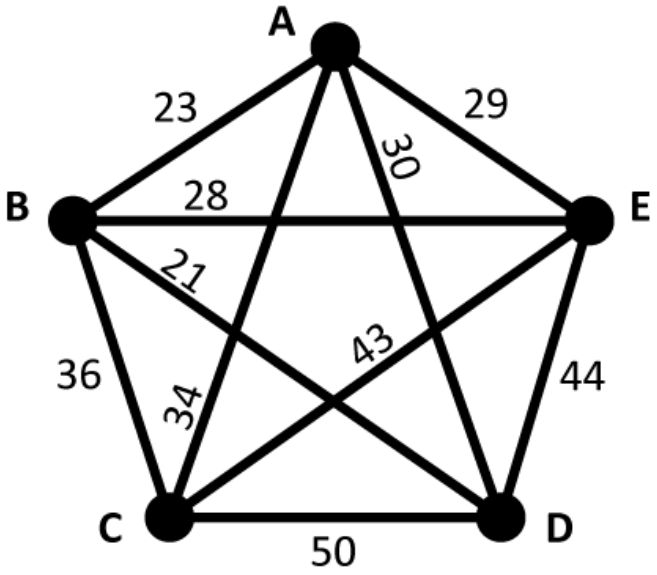
Example: Nearest-Neighbor Algorithm for TSP

1. From the starting vertex, choose the edge with the smallest cost and use that as the first edge in your circuit.
2. Continue in this manner, choosing among the edges that connect from the current vertex to vertices you have not yet visited.
3. When you have visited every vertex, return to the starting vertex.

Your Turn: Nearest-Neighbor

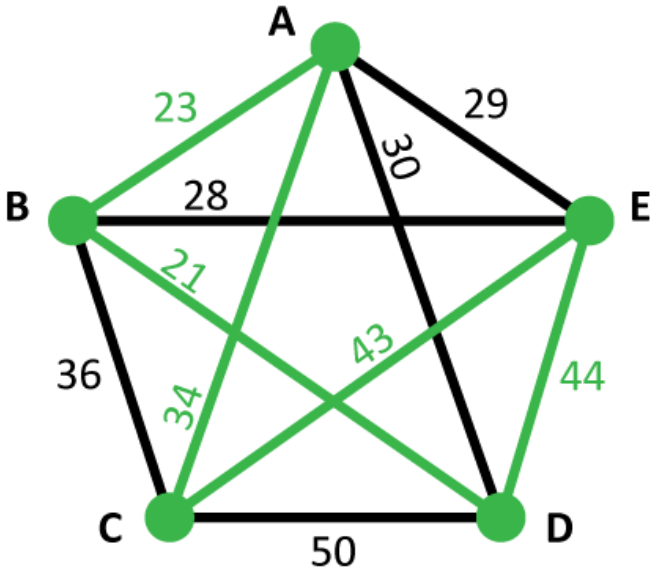
1. From the starting vertex, choose the edge with the smallest cost and use that as the first edge in your circuit.
2. Continue in this manner, choosing among the edges that connect from the current vertex to vertices you have not yet visited.
3. When you have visited every vertex, return to the starting vertex.

For this example, start at C



Your Turn: Nearest-Neighbor

- The solution is shown here
- This circuit has a total cost of 165
- If we had chosen a different starting point, we may have produced a different solution



Travelling Salesperson Lower Bound

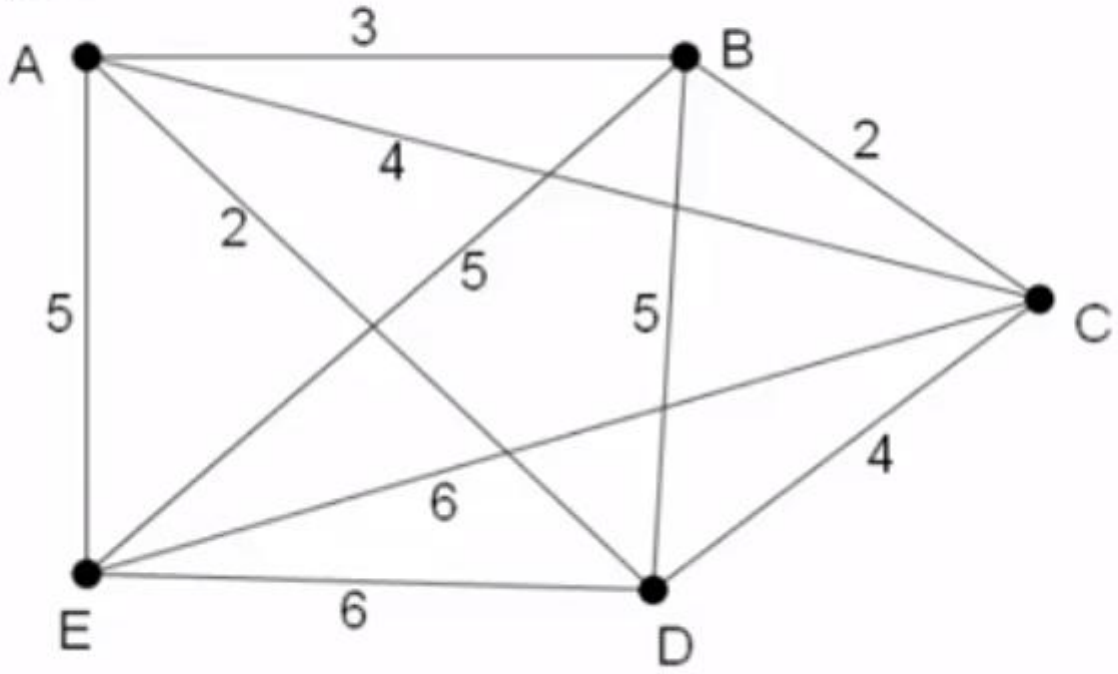
Finding a lower bound

To find a lower bound for the weight of the minimum Hamiltonian cycle:

- Choose an arbitrary node. Delete that node and all its arcs.
- Find the length of the minimum connector for the remaining arcs.
- Add the weights of the two least weight arcs from the deleted node to the weight of the minimum connector.

Finding a lower bound

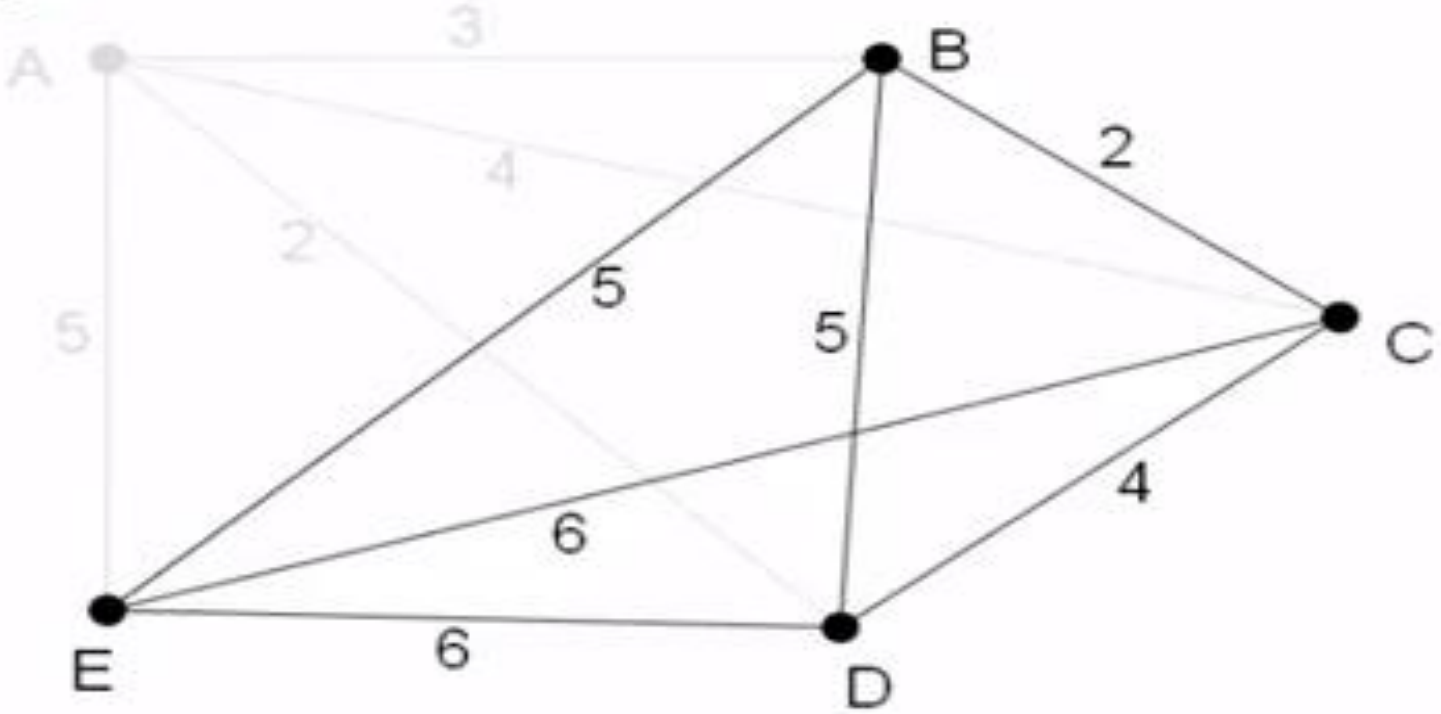
Example



First, delete A and its arcs.

Finding a lower bound

Example

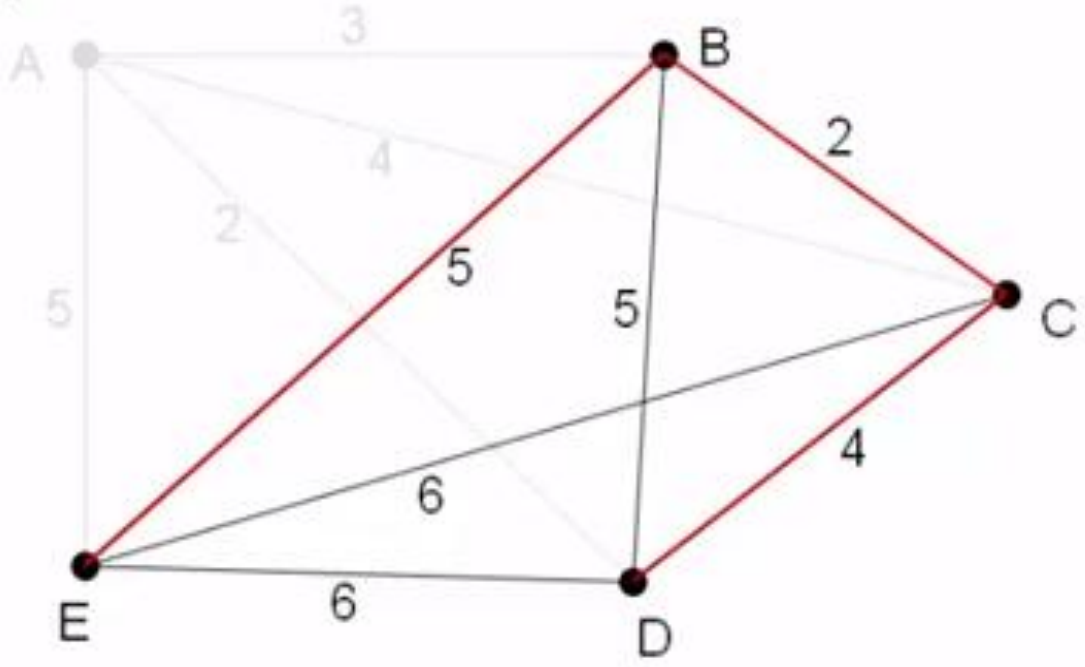


First, delete A and its arcs.

Finding a lower bound

Weight of minimum connector = 11

Example



First, delete A and its arcs.

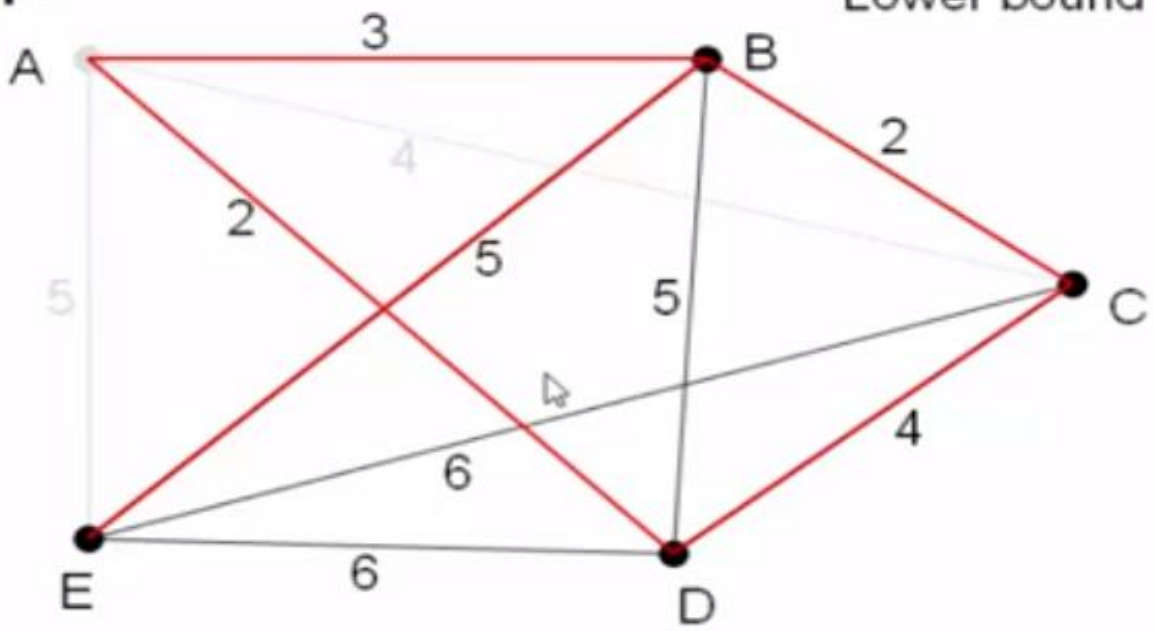
Find the weight of the minimum connector for the remaining network.

Finding a lower bound

Weight of minimum connector = 11

Lower bound = 11 + 2 + 3 = 16

Example



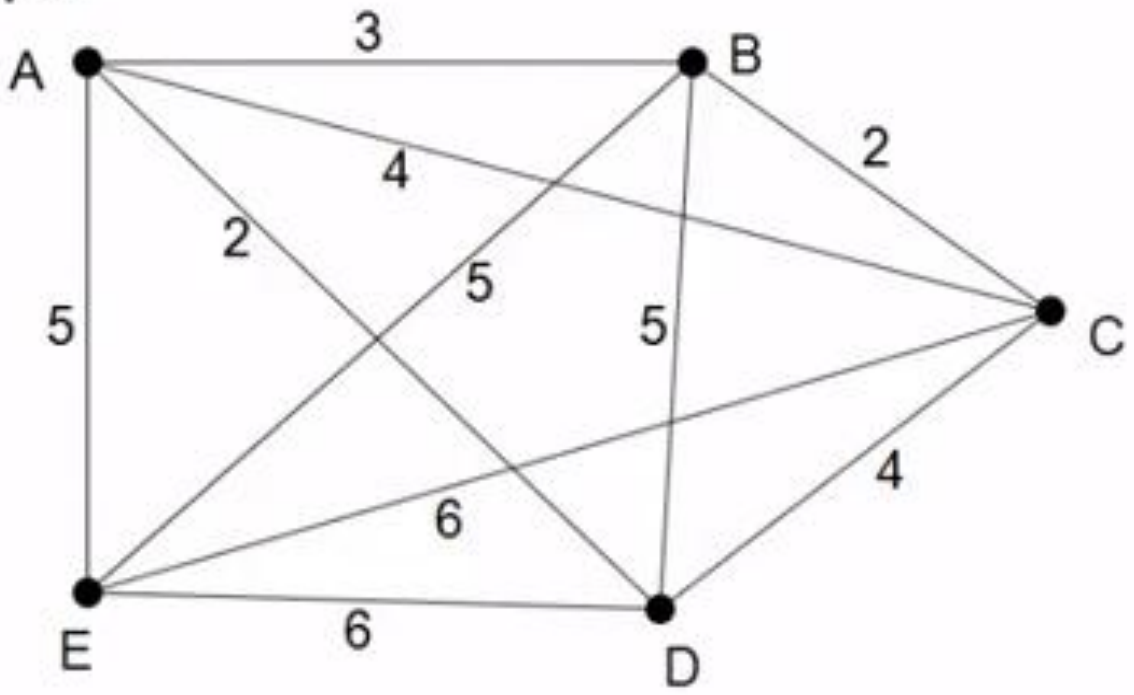
First, delete A and its arcs.

Find the weight of the minimum connector for the remaining network.

Add the two least weight arcs from A to give a lower bound.

Finding a lower bound

Example

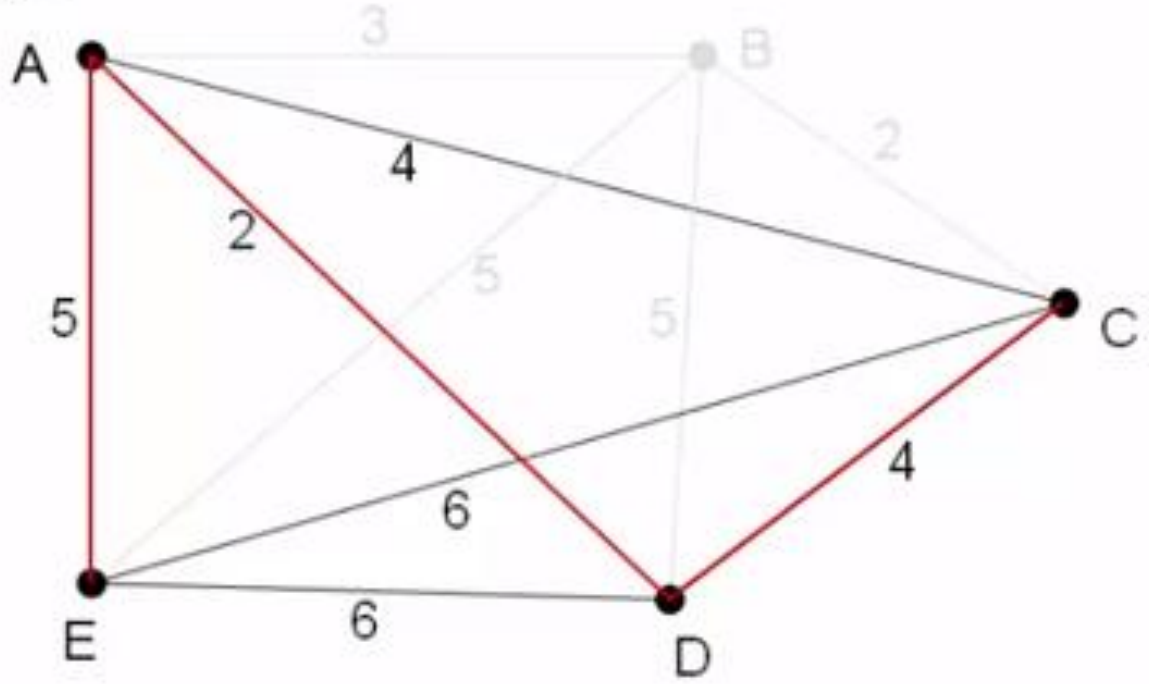


Now, delete B and its arcs.

Finding a lower bound

Weight of minimum connector = 11

Example



Now, delete B and its arcs.

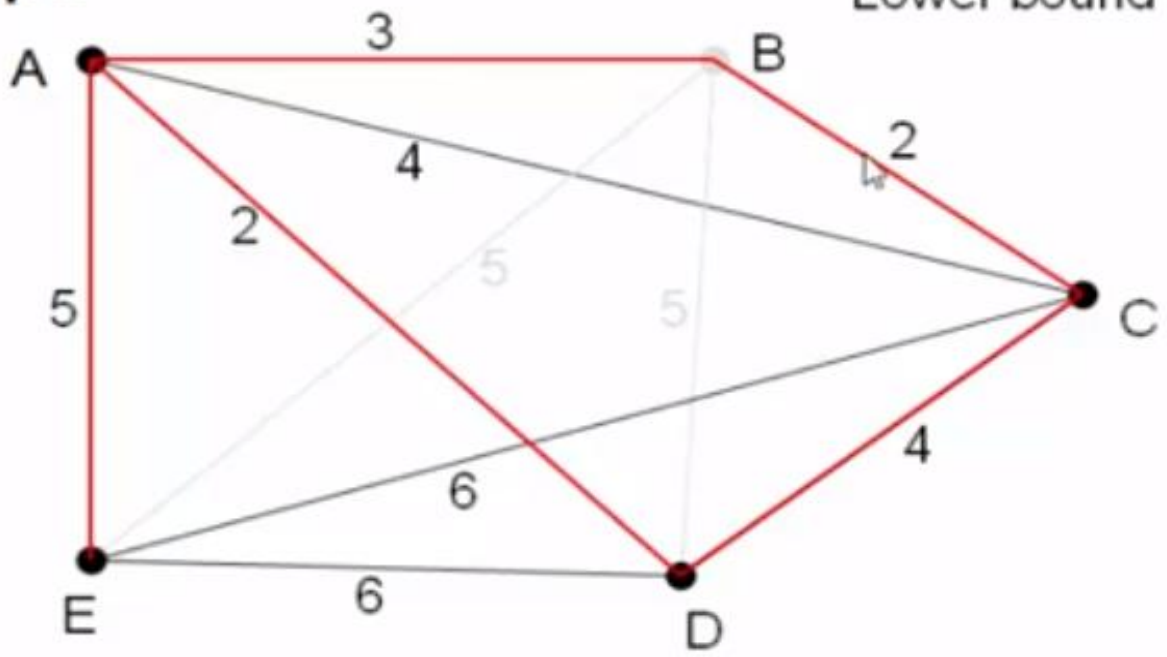
Find the weight of the minimum connector for the remaining network.

Finding a lower bound

Weight of minimum connector = 11

Lower bound = 11 + 2 + 3 = 16

Example



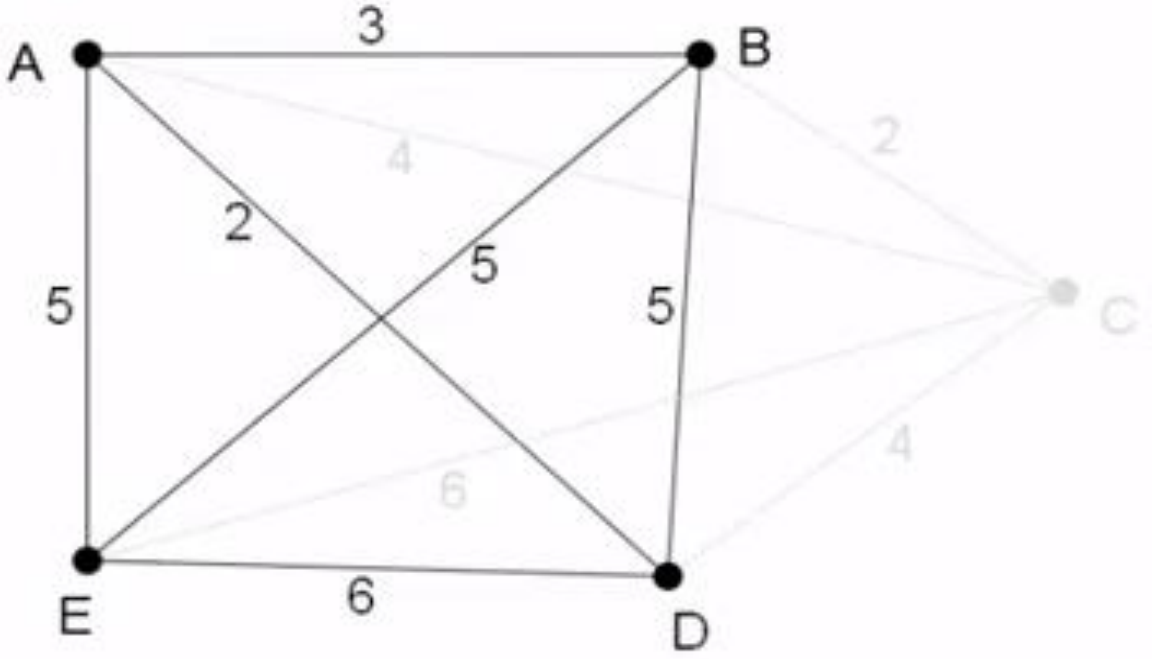
Now, delete B and its arcs.

Find the weight of the minimum connector for the remaining network.

Add the two least weight arcs from B to give a lower bound.

Finding a lower bound

Example



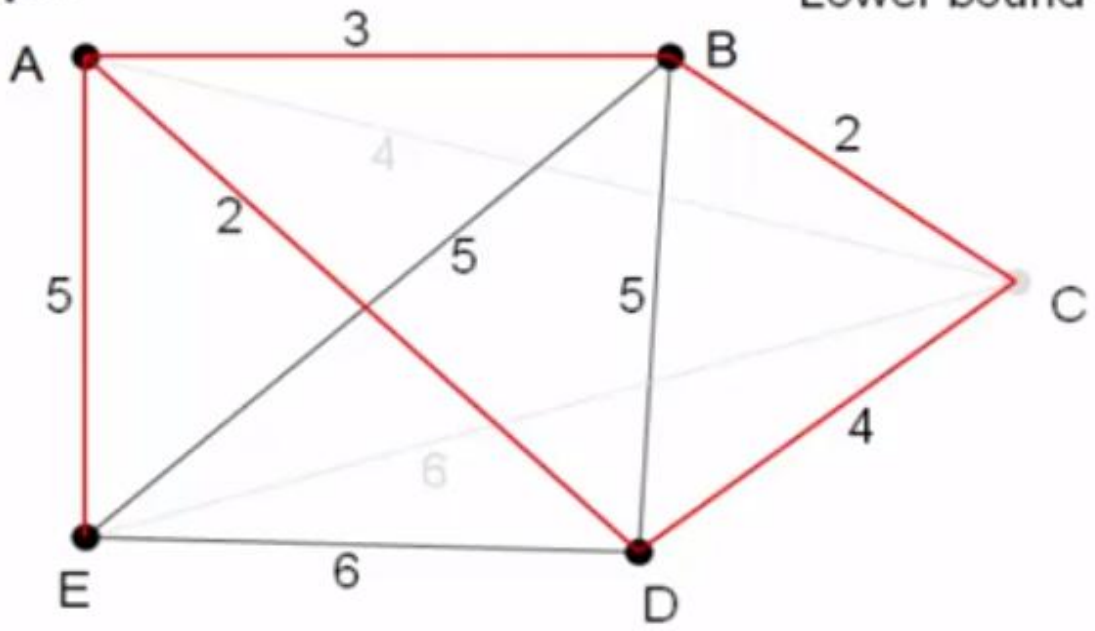
Now, delete C and its arcs.

Finding a lower bound

Weight of minimum connector = 10

Lower bound = 10 + 2 + 4 = 16

Example



Now, delete C and its arcs.

Find the weight of the minimum connector for the remaining network.

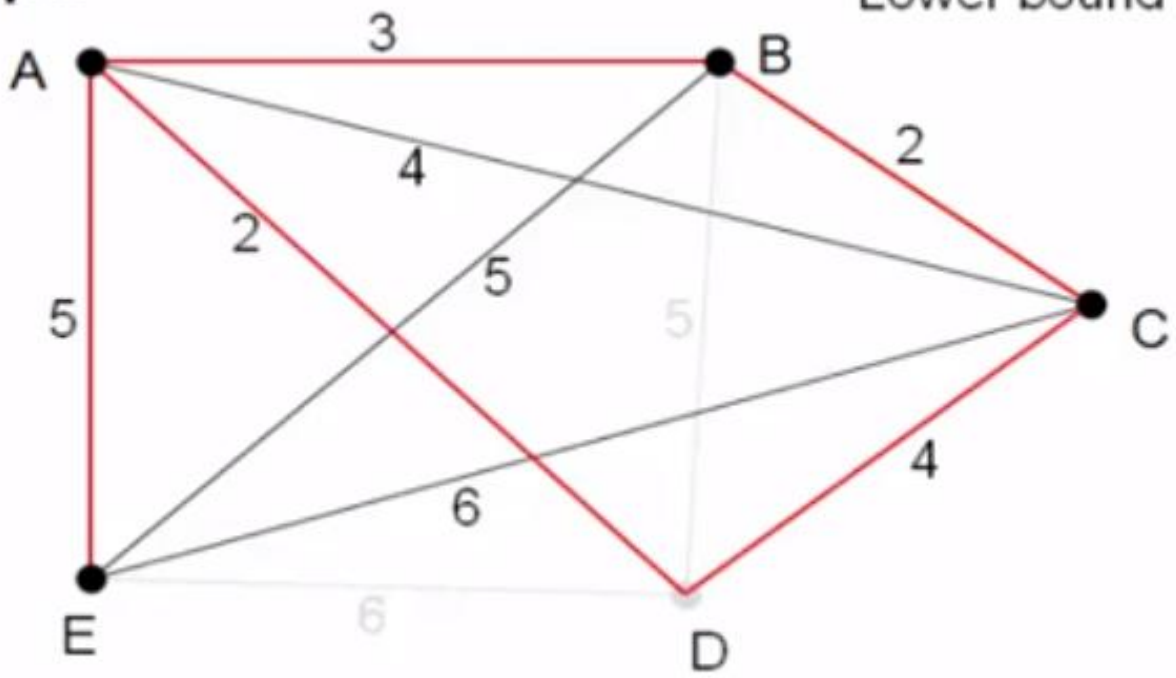
Add the two least weight arcs from C to give a lower bound.

Finding a lower bound

Weight of minimum connector = 10

Lower bound = 10 + 2 + 4 = 16

Example



Now, delete D and its arcs.

Find the weight of the minimum connector for the remaining network.

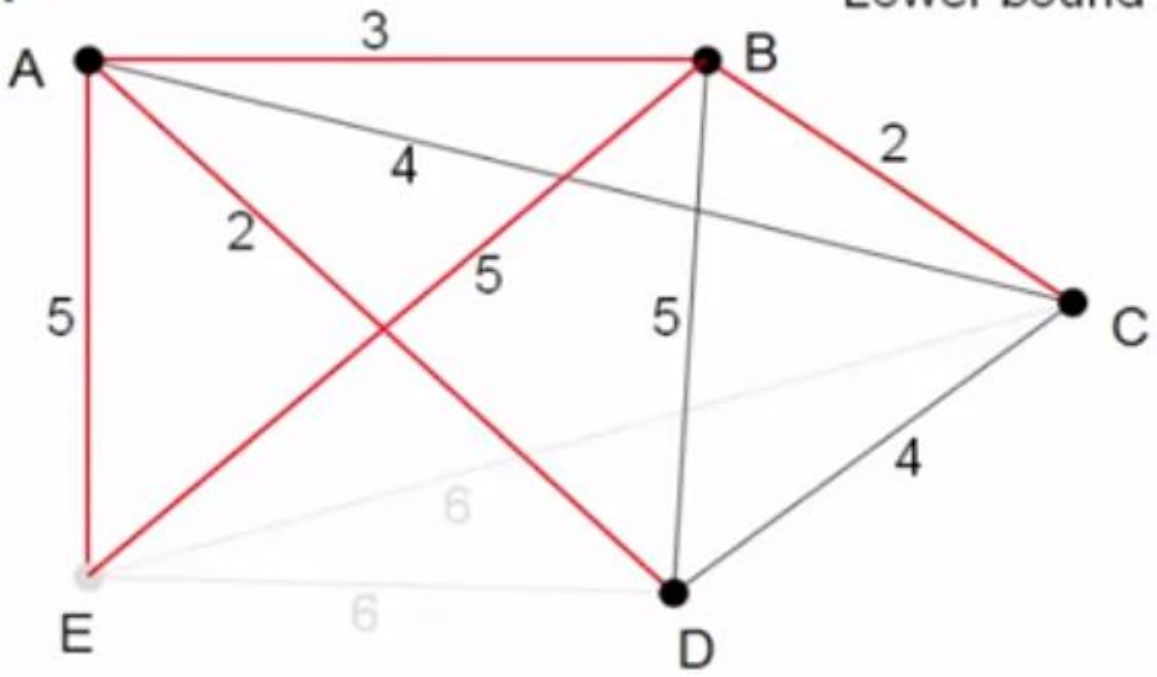
Add the two least weight arcs from D to give a lower bound.

Finding a lower bound

Weight of minimum connector = 7

Lower bound = 7 + 5 + 5 = 17

Example



Finally, delete E and its arcs.

Find the weight of the minimum connector for the remaining network.

Add the two least weight arcs from E to give a lower bound.

Finding a lower bound

Vertex deleted	Lower bound
A	16
B	16
C	16
D	16
E	17

The greatest lower bound is 17, by deleting vertex E.

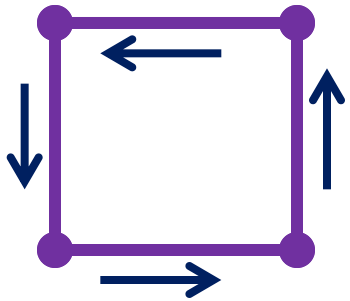
So the lower bound for the travelling salesperson problem is 17.

Reductions from Hamiltonian Cycle to the Traveling Salesman Problem (TSP)

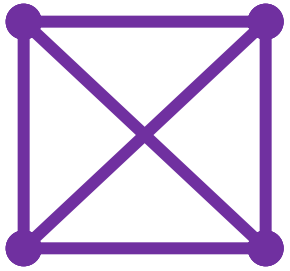
- Reducing from big problem (decision) to a little problem that will solve/ decide the big problem.
- **Hamiltonian Cycle**

  -does not imply that the graph is complete

  - does the graph contain a cycle that visits each vertex exactly once



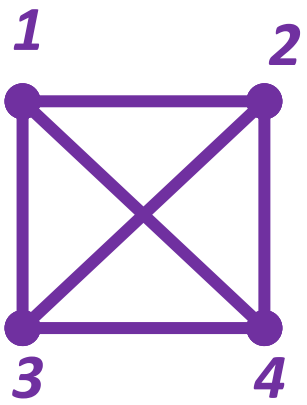
Traveling Salesman Problem (TSP)



-graph must be complete

- does the graph contain a cycle that visits each vertex exactly once AND has total length ≤ k

- **Hamiltonian Cycle Problem**

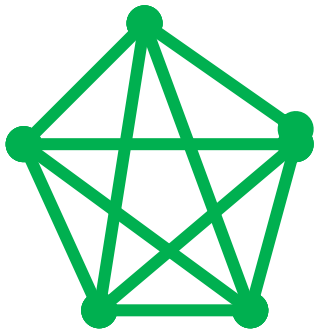


4 vertices, 4 edges

Traveling Salesman Problem (TSP)

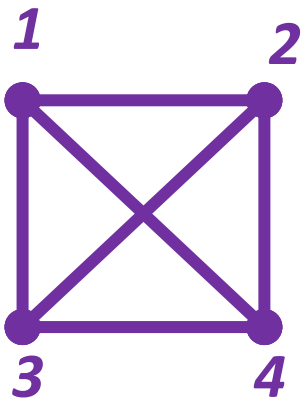
-Complete the graph

- how do we set the weights (lengths)?
  (remember total length ≤ k)
- how do we pick number of k ?



5 vertices, 5 edges

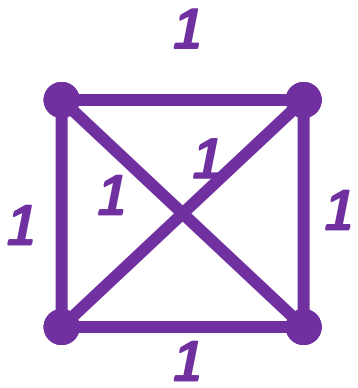
- **Hamiltonian Cycle Problem** – if you have n vertices, you need n edges.



Traveling Salesman Problem (TSP)

-Complete the graph

- how do we set the weights (lengths)?
  (remember total length ≤ k)
- how do we pick number of k ?

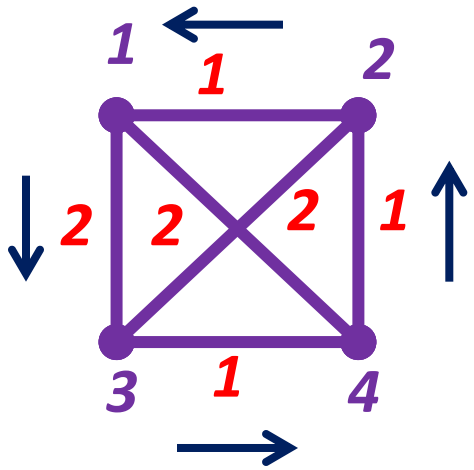


n = 4

Greedy heuristic

k = 4

- **Hamiltonian Cycle Problem** – if you have n vertices, you need n edges.



Traveling Salesman Problem (TSP)

-Complete the graph

- how do we set the weights (lengths)?
  (remember total length ≤ k)
- how do we pick number of k ?

Total length (want it to be ≤ 4)

1+1+2+1 = 5

Input- original graph G = (V,E)

Output- graph corresponding to TSP

- k

minimum k =n

n = 4

k = 4

NP-Completeness

Intractability

- We defined a **good algorithm** to be an algorithm that runs (correctly) in time bounded by a polynomial function of the input size.

- One algorithm for the TSP considers all spanning cycles and selects the cheapest one. This is not a good algorithm, because K_n has (n-1)!/2 spanning cycles, and this has grows faster than every polynomial function of n.

- The computation takes too long for graphs of any substantial size. Practical applications require solving TSPs on graphs with hundreds or thousands of vertices.

Intractability

- No one has found a good algorithm, and no one has proved that none exists. The TSP belongs to a large class of problems having the property that a good algorithm for any one of them will yield a good algorithm for every one of them. A good algorithm for B yields a good algorithm for A if we can "**reduce** " problem A to problem B.
- As an easy example of this, we can use a good algorithm for the TSP (problem B) to recognize Hamiltonian graphs (problem A). From a graph G, form an instance of the TSP on vertex set V(G) by assigning weight 0 to vertex pairs that are edges of G and weight 1 to pairs that are not. The graph G has a Hamiltonian cycle if and only if the optimal solution to this instance of the TSP has cost 0. The time for the transformation is polynomial in n(G), so a good algorithm for the TSP produces a good algorithm to test for spanning cycles. We conclude that the TSP is at least as hard as the Hamiltonian cycle problem.

Decision Problem

- In the formal discussion, we consider only **decision problems**, where the answer is YES on NO. This makes sense for recognizing Hamiltonian graphs, but the TSP is an optimization problem. When formulated as a decision problem (called **MINIMUM SPANNING CYCLE**), the input for the TSP is a weighted graph G and a number k, and the problem is to test whether G has a spanning cycle with weight at most k.

- Repeated applications of this decision problem (at most a polynomial number of applications) can be used to find the minimum weight of a spanning cycle. Similarly, **MAXIMUM INDEPENDENT SET** takes a graph G and an integer k as input and test $\alpha(G) \geq k$.

NP-Hard, NP-Complete

- A problem is **NP-hard** if a polynomial-time algorithm for it could be used to construct a polynomial-time algorithm for each problem in NP.
- It is **NP-complete** if it belongs to NP and is NP-hard. If some NP-complete problem belongs to P, then P=NP.
- No polynomial-time algorithm is known for any of the many NP-complete problems. This supports the prevailing belief that $P \neq NP$.
- Given one NP-complete problem, NP-completeness of other problems follows by reduction arguments as suggested earlier.

Examples NP-complete and NP-hard problems

NP-complete

Hamiltonian Paths

Optimization Problem: Given a graph, find a path that passes through every vertex exactly once

Decision Problem: Does a given graph have a Hamiltonian Path ?

NP-hard

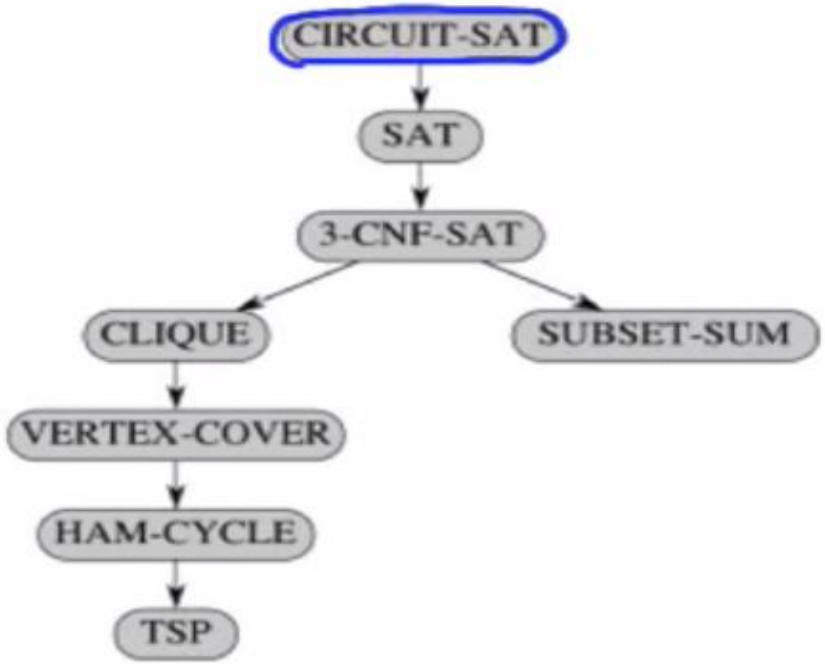
Traveling Salesman

Optimization Problem: Find a minimum weight Hamiltonian Path

Decision Problem: Given a graph and an integer k, is there a Hamiltonian Path with a total weight at most k ?

NP Completeness Proofs by Reduction

We can now populate the set NPC by transitive reduction: Reducing one problem in NPC to a new problem means that all problems in NPC transitively reduce to that new problem.



General procedure for proving that L is in NPC:

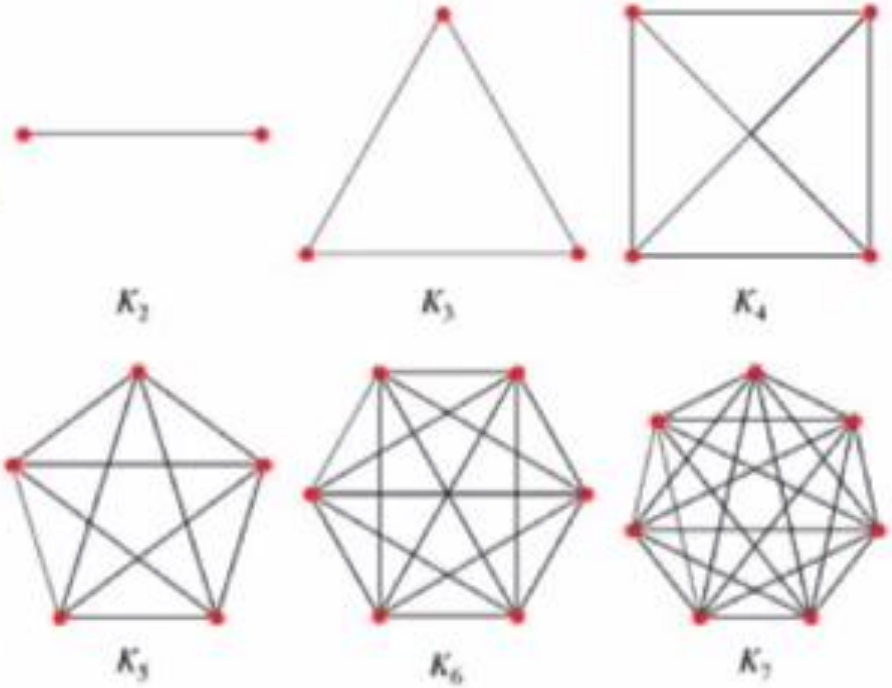
1. **Prove $L \in$ NP** (show one can check solutions in polynomial time).
2. **Prove L is NP-Hard**:
   1. Select a known language L' in NPC
   2. Describe an algorithm A that computes function f mapping *every* instance $x \in \{0, 1\}^*$ of L' to *some appropriately constructed* instance $f(x)$ of L.
   3. Prove that $x \in L'$ iff $f(x) \in L$, $\forall\, x \in \{0, 1\}^*$.
   4. Prove that A runs in polynomial time.

Why doesn't mapping *every* instance of L to *some* instances of L' work?

Clique

CLIQUE

A **clique** in an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$, each pair of which is connected by an edge in E (a complete subgraph of G). (Example cliques are shown.)



The **clique problem** is the problem of finding a clique of maximum size in G. This can be converted to a decision problem by asking whether a clique of a given size k exists in the graph:

$$\text{CLIQUE} = \{\langle G, k\rangle : G \text{ is a graph containing a clique of size } k\}$$

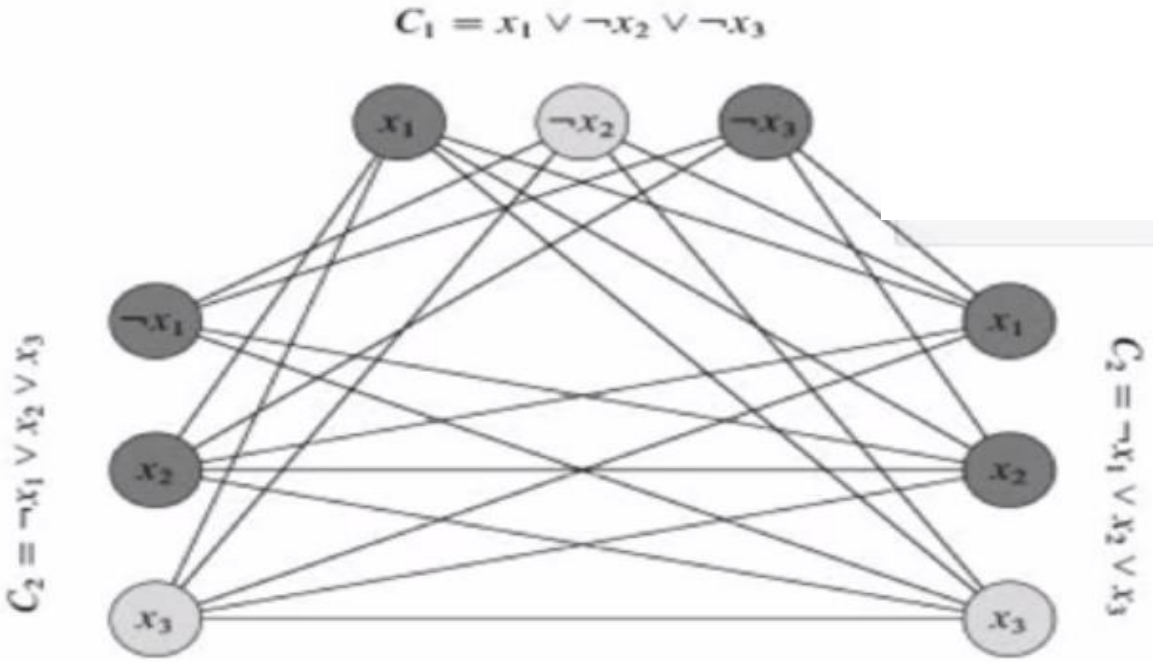
Clique is NP Complete

CLIQUE is NP Complete

One can check a solution in polynomial time.

3-CNF-SAT is reduced to CLIQUE by a clever reduction illustrated in the figure.

- There is a vertex for every literal
- There is an edge between vertices only if the corresponding literals are in different triples *and* the literals are consistent.



If there are k clauses we ask whether the graph has a k-clique. Such a clique exists if and only if there is a satisfying assignment.

- The fact that there is a k-clique means there are k vertices that are all connected to each other.
- The fact that two vertices are connected to each other means that they can receive a consistent boolean assignment, *and* that they are in different clauses.
- Since there are k vertices then a literal from each of the k clauses must be satisfied.

Vertex Cover

A vertex cover of an undirected graph $G = (V, E)$ is a subset $V' \subseteq V$ such that if $(u, v) \in E$ then $u \in V'$ or $v \in V'$ or both.

Each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E.

The **Vertex Cover Problem** is to find a vertex cover of minimum size in G. Phrased as a decision problem,

$$\text{VERTEX-COVER} = \{\langle G, k\rangle : \text{graph } G \text{ has a vertex cover of size } k\}$$

Vertex Cover is NP Complete

Vertex Cover is NP Complete

VERTEX-COVER = $\{\langle G, k\rangle$: graph G has a vertex cover of size $k\}$

There is a straightforward reduction of CLIQUE to VERTEX-COVER, illustrated in the figure.

Given an instance $G=(V,E)$ of CLIQUE, one computes the complement of G, which we will call $G_c = (V,\bar{E})$, where $(u,v) \in \bar{E}$ iff $(u,v) \notin E$.

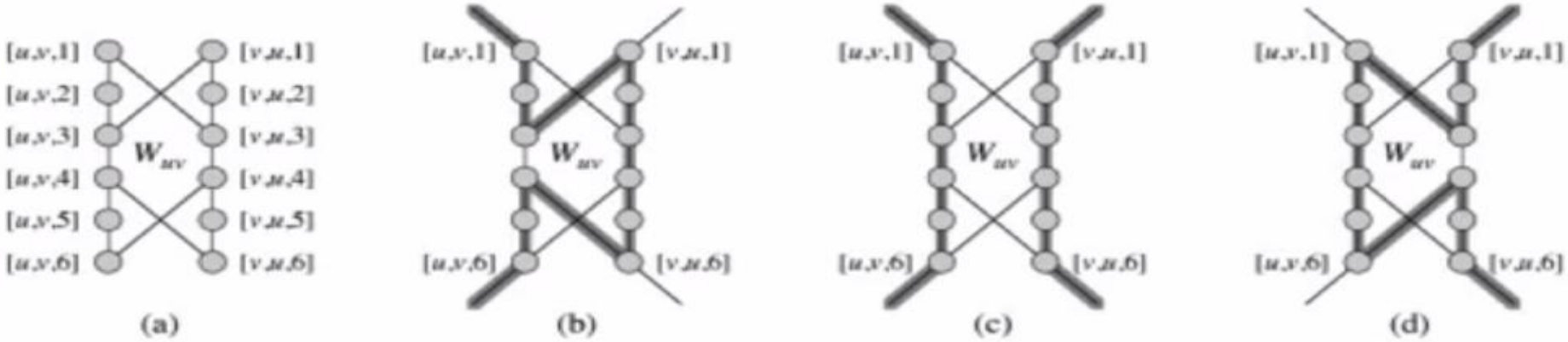
The graph G has a clique of size k iff the complement graph has a vertex cover of size $|V| - k$.

Hamiltonian Cycle (HAM-CYCLE)

Hamiltonian Cycle (HAM-CYCLE)

HAM-CYCLE = {⟨G⟩ : G is a Hamiltonian graph}

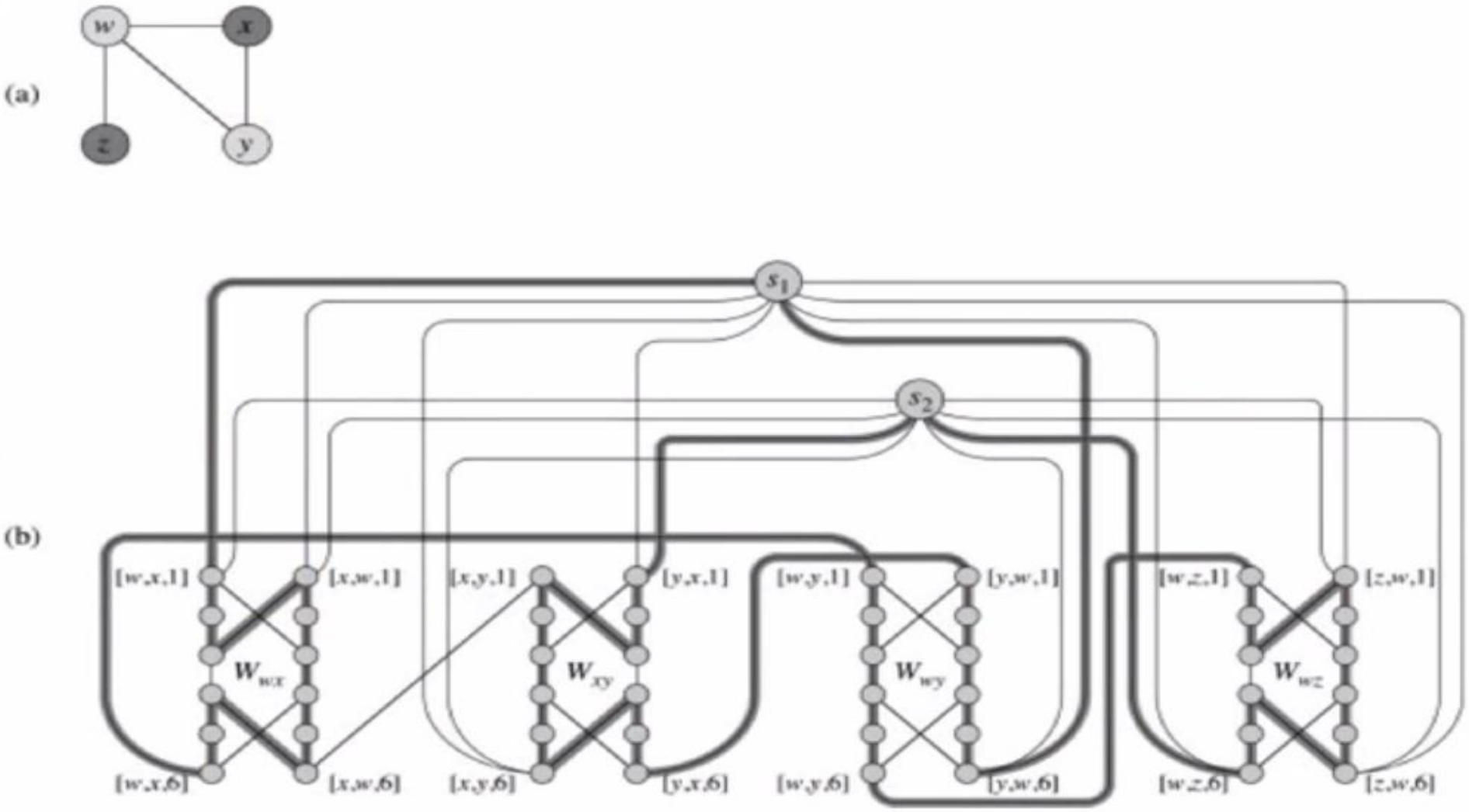
We reduce VERTEX-COVER to HAM-CYCLE by converting each edge of VERTEX-COVER instance G into subgraph "widgets" through which one can find a portion of a Hamiltonian Cycle only if one or both of the vertices of the edge are in a covering set.



Any Hamiltonian cycle must include all the vertices in the widget, and there are only three ways to pass through each widget. If only vertex u is included in the cover, we will use path (b); if only vertex v then path (d); otherwise path (c) to include both.

The widgets are wired in sequences that chain all the widgets for each given vertex, so if the vertex is selected all of the widgets corresponding to its edges are reached.

Hamiltonian Cycle (HAM-CYCLE)



Finally, k selector vertices are added, and wired such that each will select the kth vertex in the cover of size k.

Traveling Salesperson Problem (TSP)

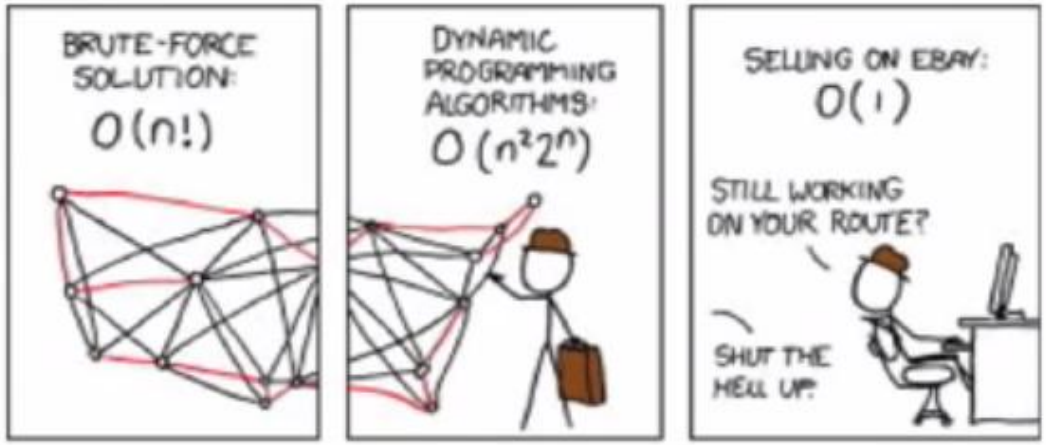
Traveling Salesperson Problem (TSP)

Finally, one of the more famous problems: Suppose you are a traveling salesperson, and you want to visit n cities exactly once in a Hamiltonian cycle, but choosing a **tour** with minimum cost.

$$\begin{aligned}\text{TSP} = \{\langle G, c, k\rangle : & \ G = (V, E) \text{ is a complete graph,} \\ & c : V \times V \to \mathbf{N}, \\ & k \in \mathbf{N}, \text{ and} \\ & G \text{ has a traveling-salesperson tour with cost at most } k\}\end{aligned}$$

Only exponential solutions have been found to date, although it is easy to check a solution in polynomial time.

The reduction represents a HAM-CYCLE problem as a TSP problem on a complete graph, but with the cost of the edges in TSP being 0 if the edge is in the HAM-CYCLE problem, or 1 if not.



Conclusion

- In this lecture, we have discussed the Hamiltonian Graphs, Traveling Salesman Problem (TSP), Intractability, Decision Problems, NP-Hard, NP-Complete Problems, NP Completeness Proofs by Reduction, Heuristics and Bounds, Nearest-Neighbor Algorithm for TSP and also discuss the Reductions from Hamiltonian Cycle to the Traveling Salesman Problem (TSP)