Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Eulerian Tour

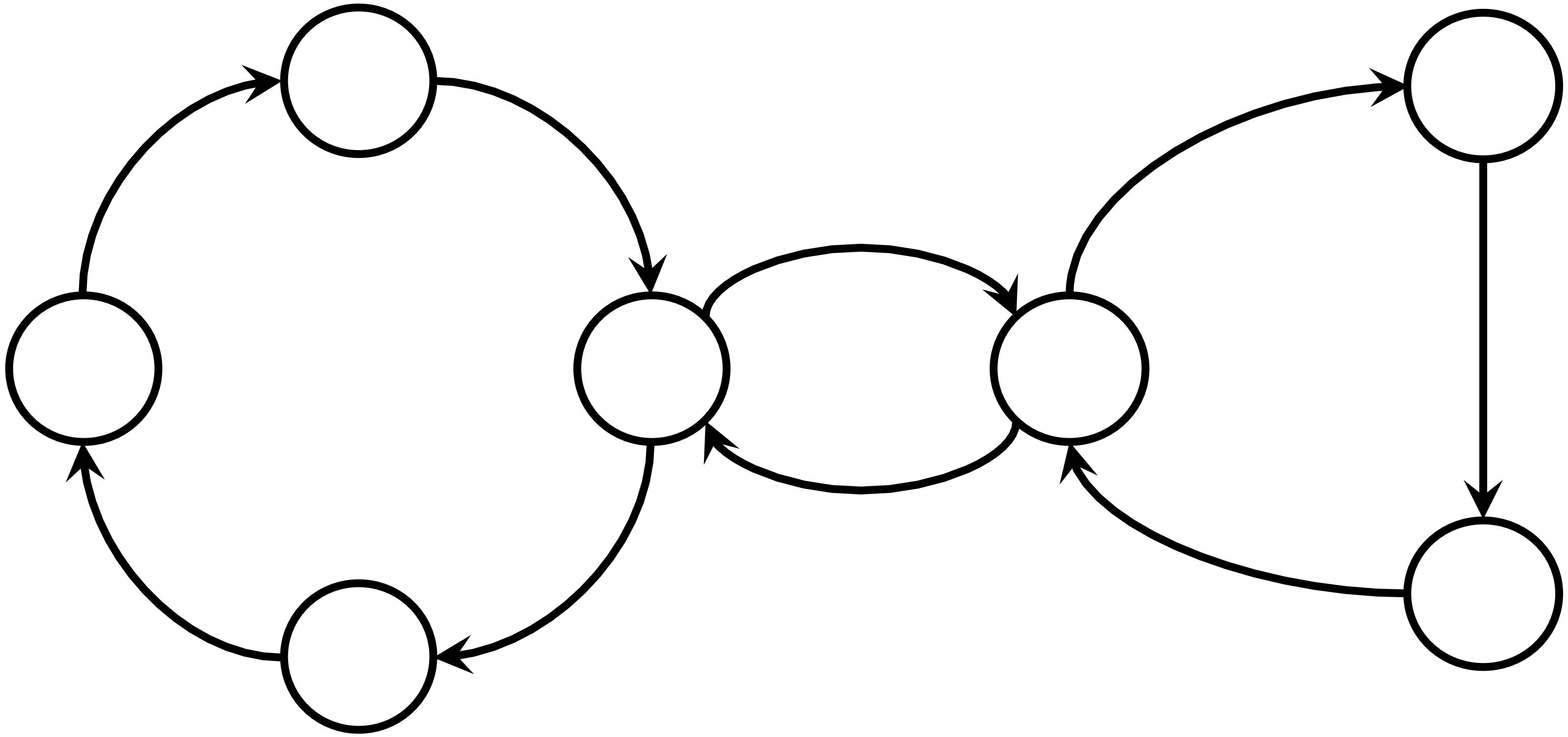
- We say a graph G has an **Eulerian tour** if:

1. There is a **trail** that **visits all edges** in G (exactly once).
2. And, the starting and the finishing vertex of the **trail** are the same.

- We usually assume that if G is undirected it has only one component, and it G is directed it has only one strongly connected component.

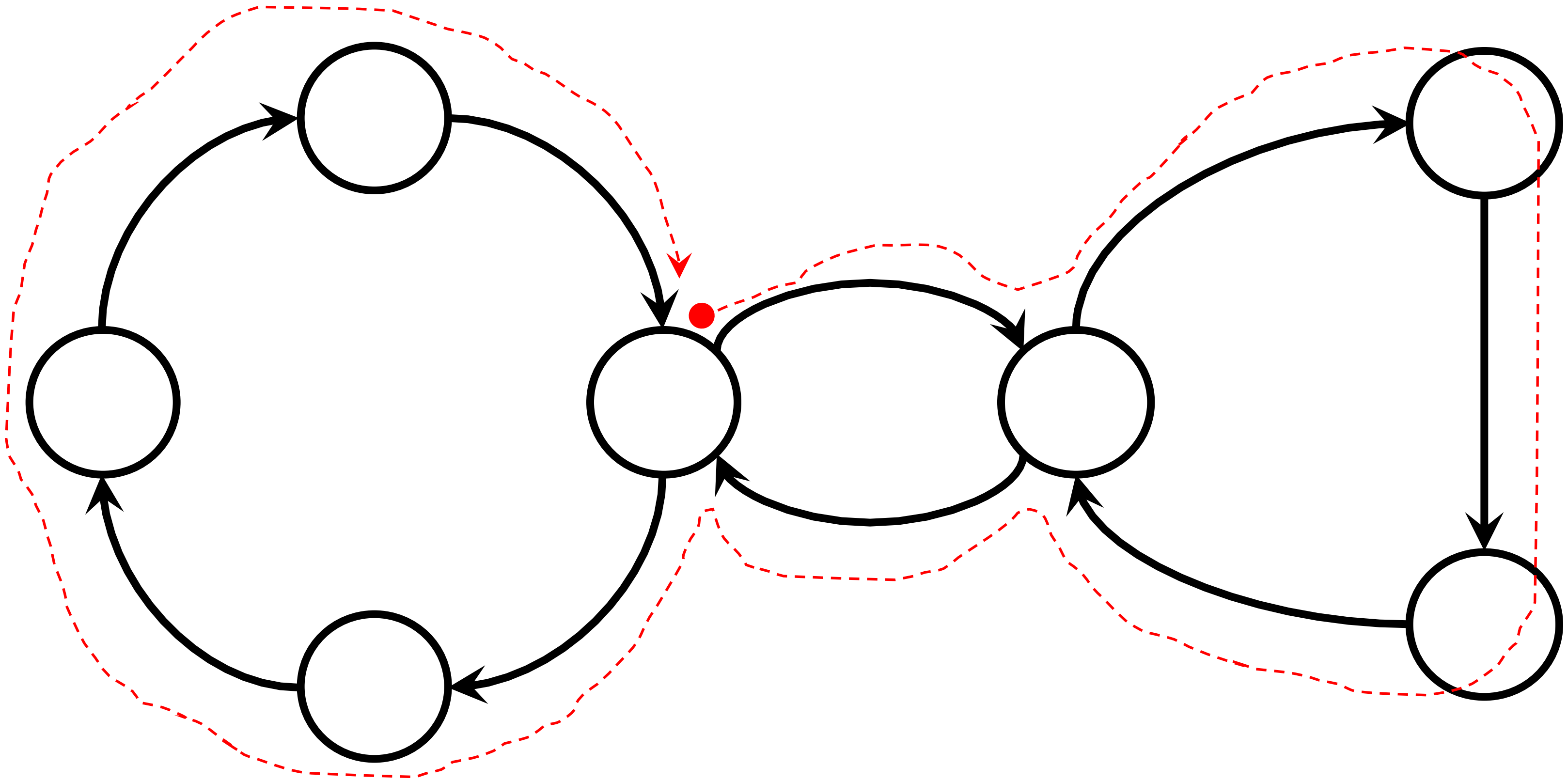
Eulerian Tour

- An Eulerian tour in a directed graph:



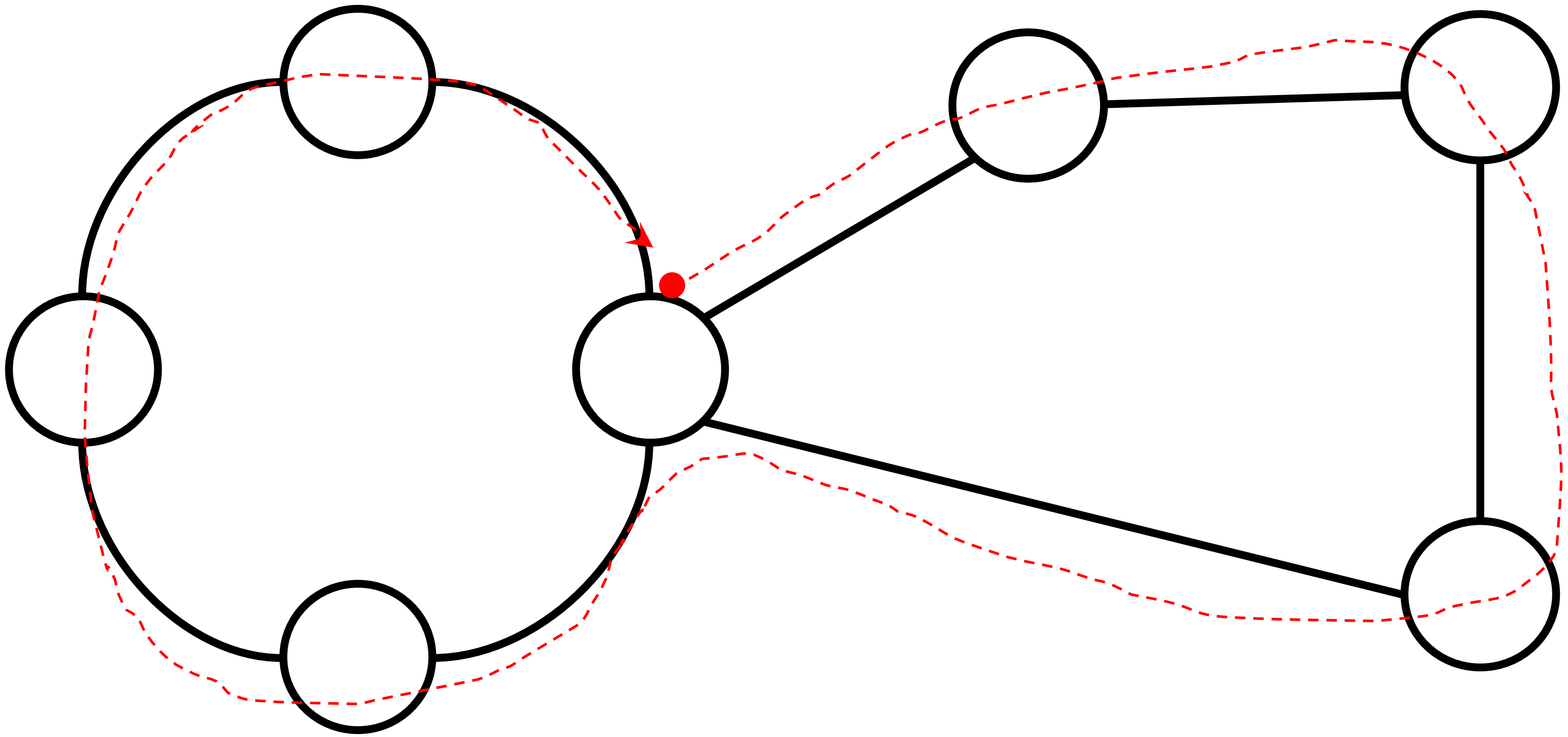
Eulerian Tour

- An Eulerian tour in a directed graph:



Eulerian Tour

- An Eulerian tour for an undirected graph:



Eulerian tour

Theorem: For a **directed graph** to have an Eulerian tour it must be that for all $v \in V: indegree(v) = outdegree(v)$

Eulerian tour

Theorem: For a **directed graph** to have an Eulerian tour it must be that for all $v \in V: indegree(v) = outdegree(v)$

- **Proof:** Each edge has to be visited **exactly once**. Imagine that we start the trail from node x. So, at the very beginning we leave x using some edge e and at the end we have to enter x using some edge e' again to finish the walk. Ignoring the initial and the last edges e, e', when we visit any vertex during the trail (including x), we have to **enter** the node and then **leave** it. So, each node has to have an outgoing edge for every incoming edge that it has. Also, for x, e is outgoing and e' is incoming. Therefore, each node must have equal number incoming and outgoing edges.

Eulerian tour

- We can use a similar argument to prove the following theorem:

Theorem: For an **undirected graph** to have an Eulerian tour it must be that for all $v \in V: degree(v)$ **is even.**

Eulerian tour

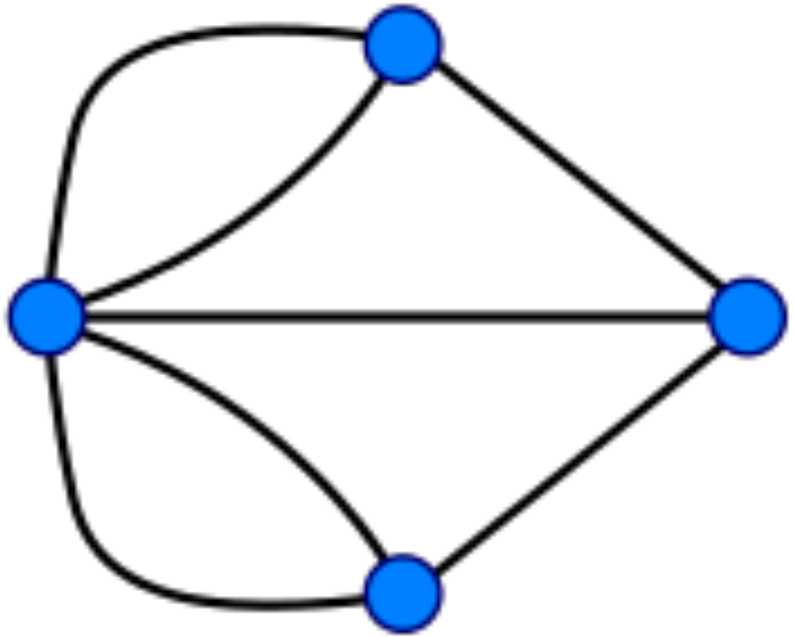
- We can use a similar argument to prove the following theorem:

Theorem: For an **undirected graph** to have an Eulerian tour it must be that for all $v \in V: degree(v)$ **is even.**

Proof: Each time we enter and leave a vertex we are visited 2 of its incident edges. So, the number of edges connected to a node must be a multiple of 2, i.e. even.

Eulerian tour

- The graph of the Konigsberg town does not have an Eulerian tour, since there are vertices of odd degree.



Finding the tour

- Using the previous theorems **we can check** whether a graph has an Eulerian tour or not.

Finding the tour

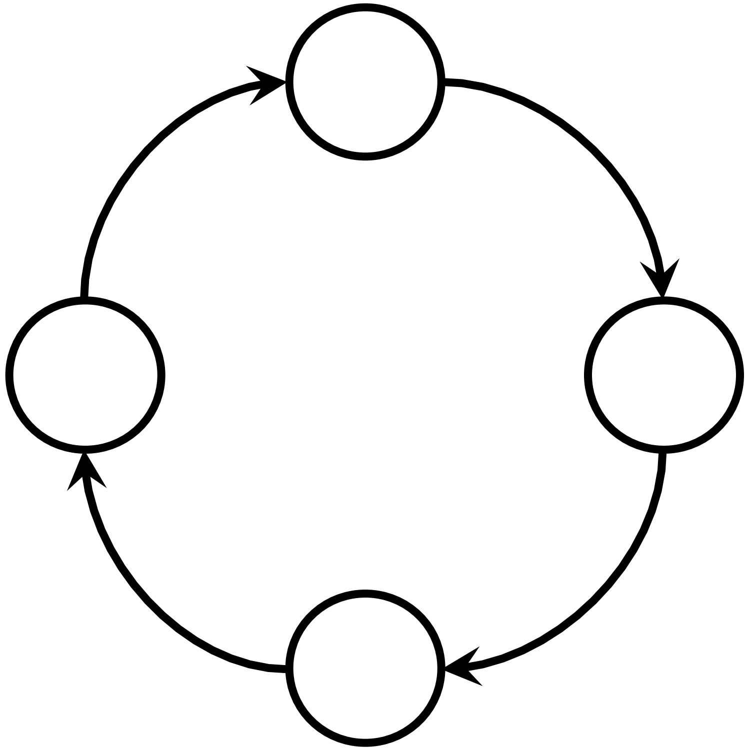
- Using the previous theorems **we can check** whether a graph has an Eulerian tour or not.

- But to actually find the tour, we use the following observations:

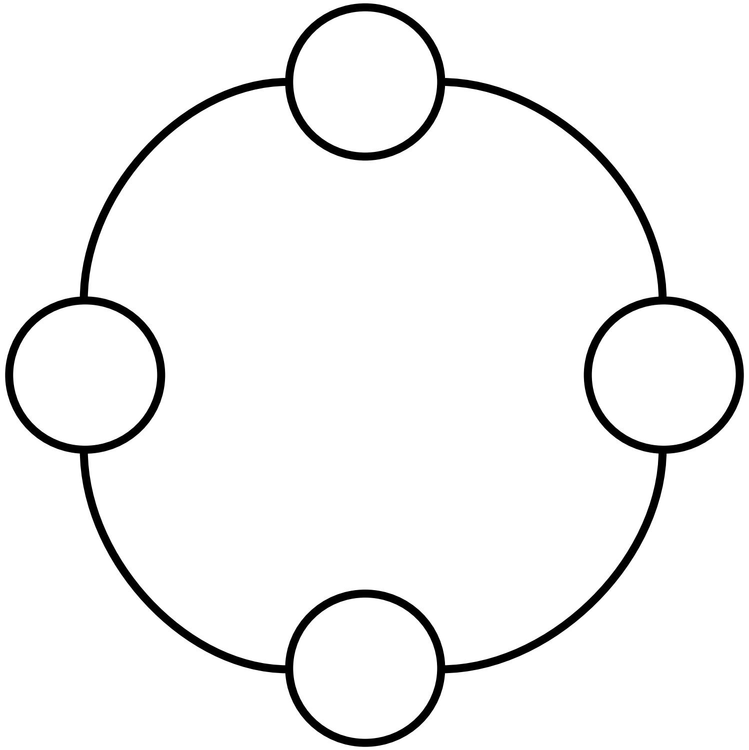
✓In a **directed cycle**, for $v \in V$, **indegree(v)=outdegree(v)=1**.

✓In an **undirected cycle** each node has **degree of 2**.

Finding the tour



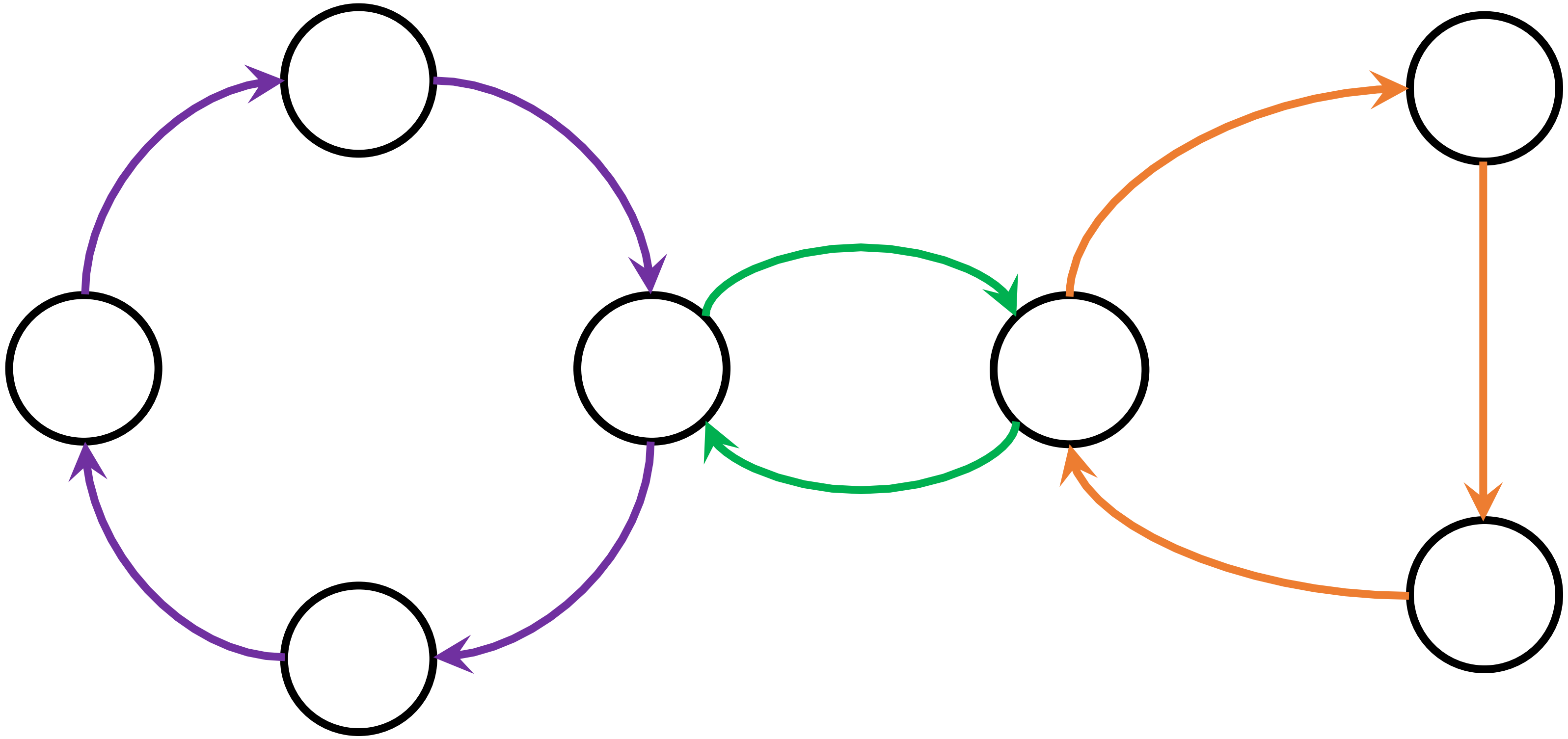
Each node has indegree and outdegree of 1



Each node has degree 2

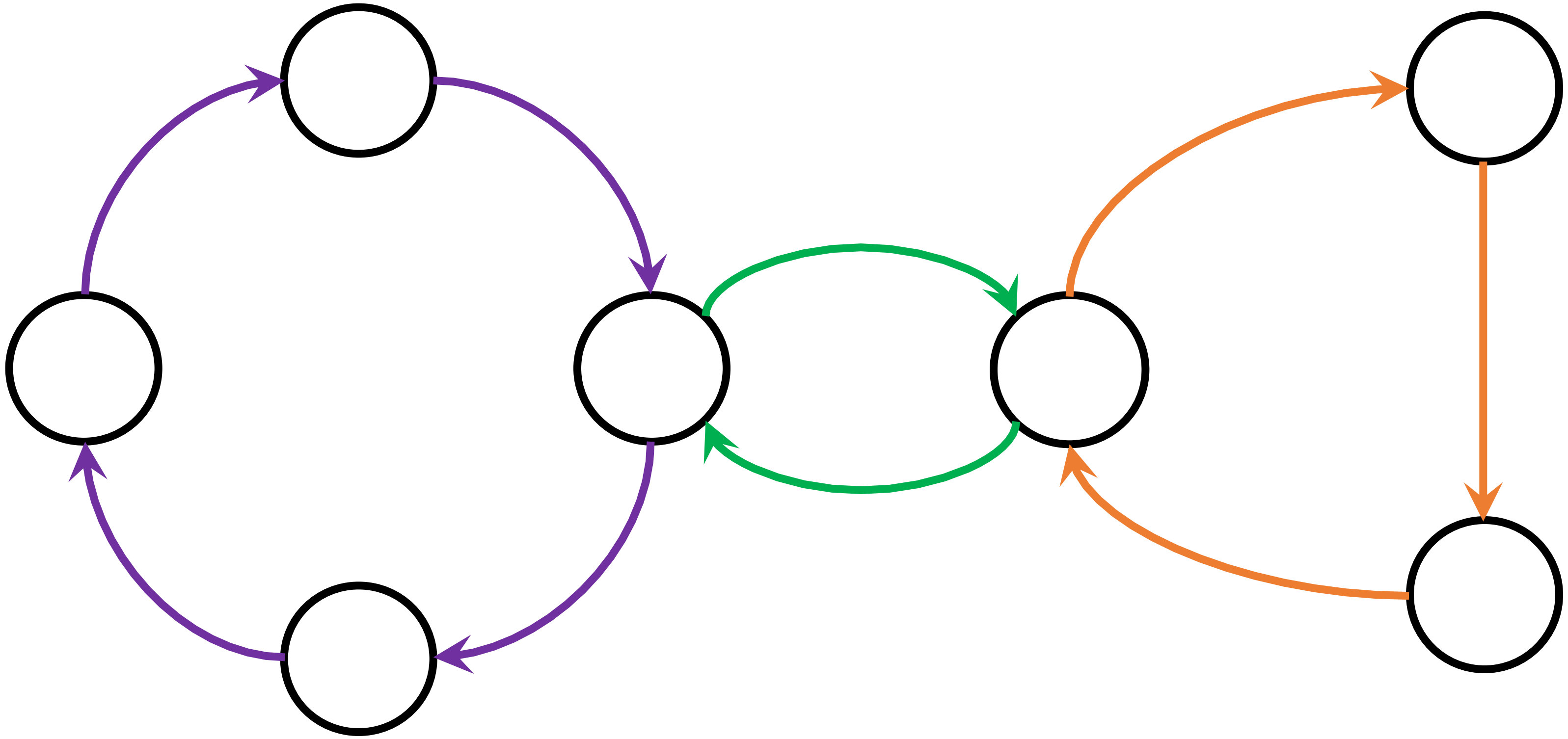
Finding the tour

- So, a graph that has an Eulerian tour can be viewed as a collection of **edge-disjoint cycles**. Example:



Finding the tour

- In fact, every node in the graph must be on at least one cycle.



Finding the tour

- So, a graph that has an Eulerian tour can be viewed as a collection of **edge-disjoint cycles**.
- Basically, the idea is to find cycles and remove the edges of the cycles from the graph.
- However, these edge-disjoint cycles should be merged in a way that they form a valid trail.

```
EULERIAN-TOUR()
    initialize tour as an empty list
    pick any arbitrary vertex v
    EULERIAN-TOUR-REC(v)
    reverse the list tour
    return tour
```

```
EULERIAN-TOUR-REC(x)
    while Adj[x] is not empty
        let y = Adj[x][0] //first remaining neighbor of x
        remove y from Adj[x] //also remove x from Adj[y] if the graph is undirected
        EULERIAN-TOUR-REC(y)
    if Adj[x] is empty
        append x to the end of tour
```

EULERIAN-TOUR()

```
initialize tour as an empty list
pick any arbitrary vertex v
EULERIAN-TOUR-REC(v)
reverse the list tour
return tour
```

The idea is similar to a DFS with the difference that we **allow a node to be visited multiple times**.

EULERIAN-TOUR-REC(x)

```
while Adj[x] is not empty
    let y = Adj[x][0] //first remaining neighbor of x
    remove y from Adj[x] //also remove x from Adj[y] if the graph is undirected
    EULERIAN-TOUR-REC(y)
if Adj[x] is empty
    append x to the end of tour
```

EULERIAN-TOUR-REC(x)

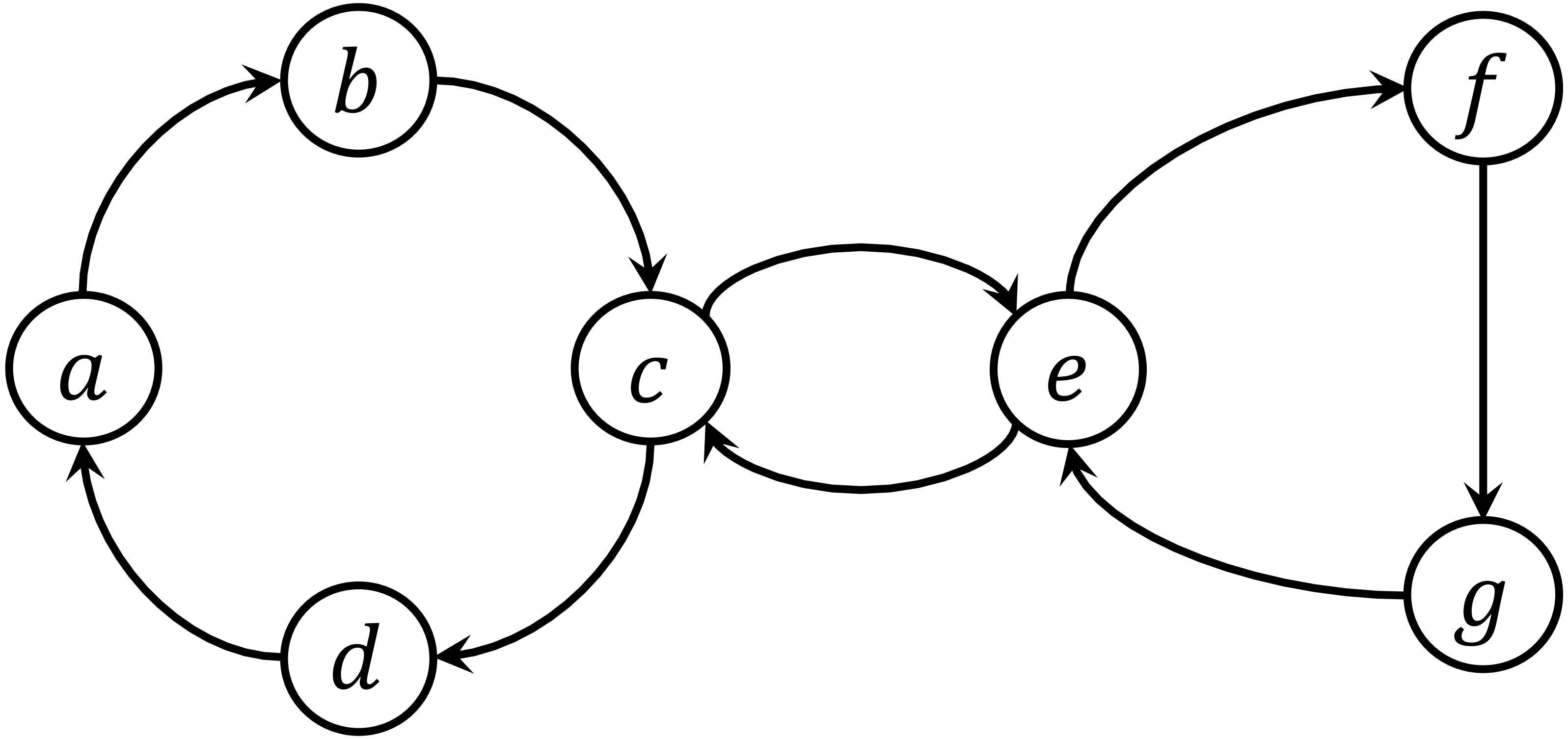
1. **while** *Adj*[x] is not empty
2. let y = Adj[x][0] //first remaining neighbor of x
3. remove y from *Adj*[x] //also remove x from *Adj*[y] if the graph is undirected
4. EULERIAN-TOUR-REC(y)
5. **if** *Adj*[x] is empty
6. append x to the end of *tour*

In step 3, **we basically remove the edge (x, y) from the graph**, so that we don't use the edge again for the tour.

In step 5, when after the while loop, the $Adj[x]$ becomes empty, it means that all cycles that had the node x in them, were recursively traversed and added to the list; so, we can now safely add the node x.

Finding the tour

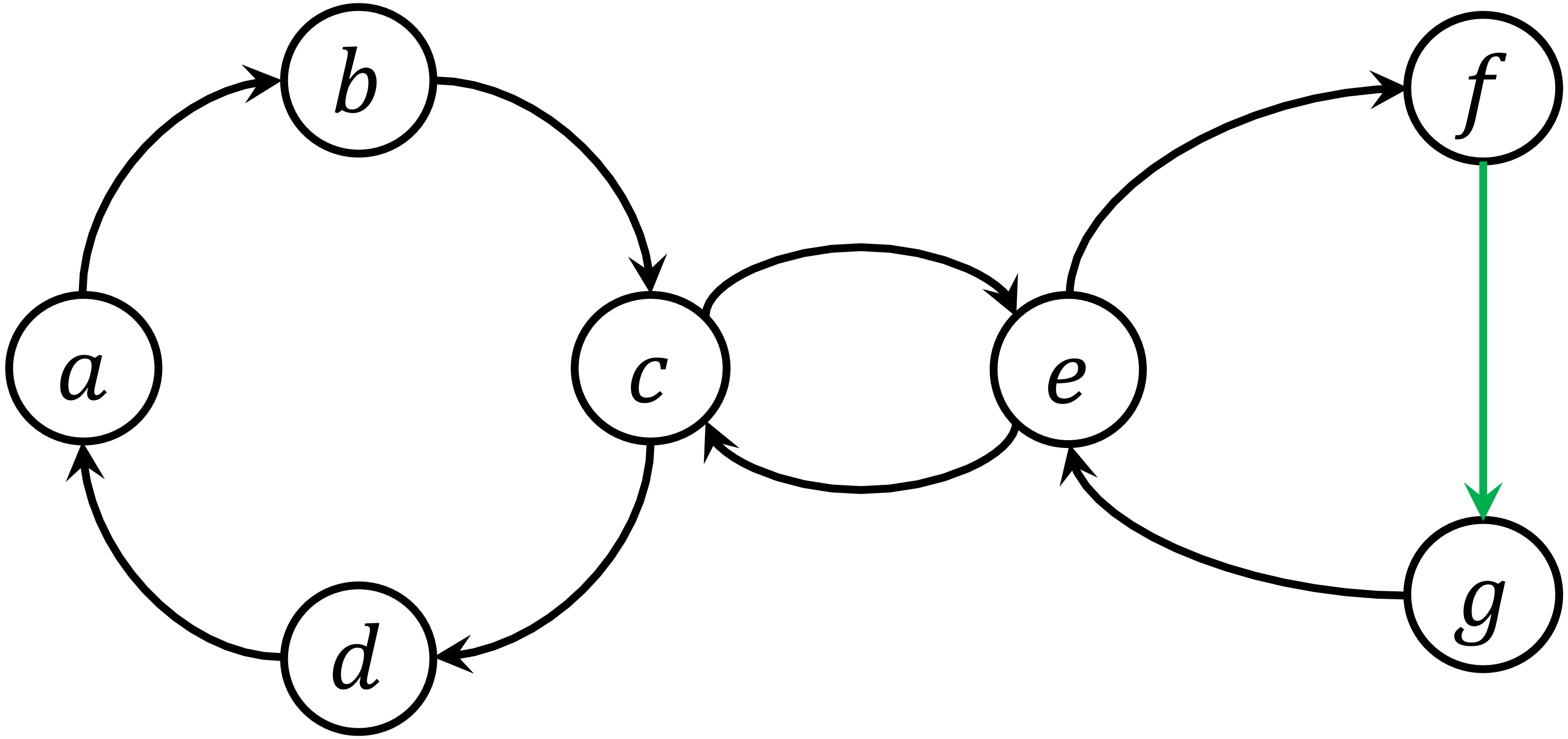
- Example, assume we start from node f



Finding the tour

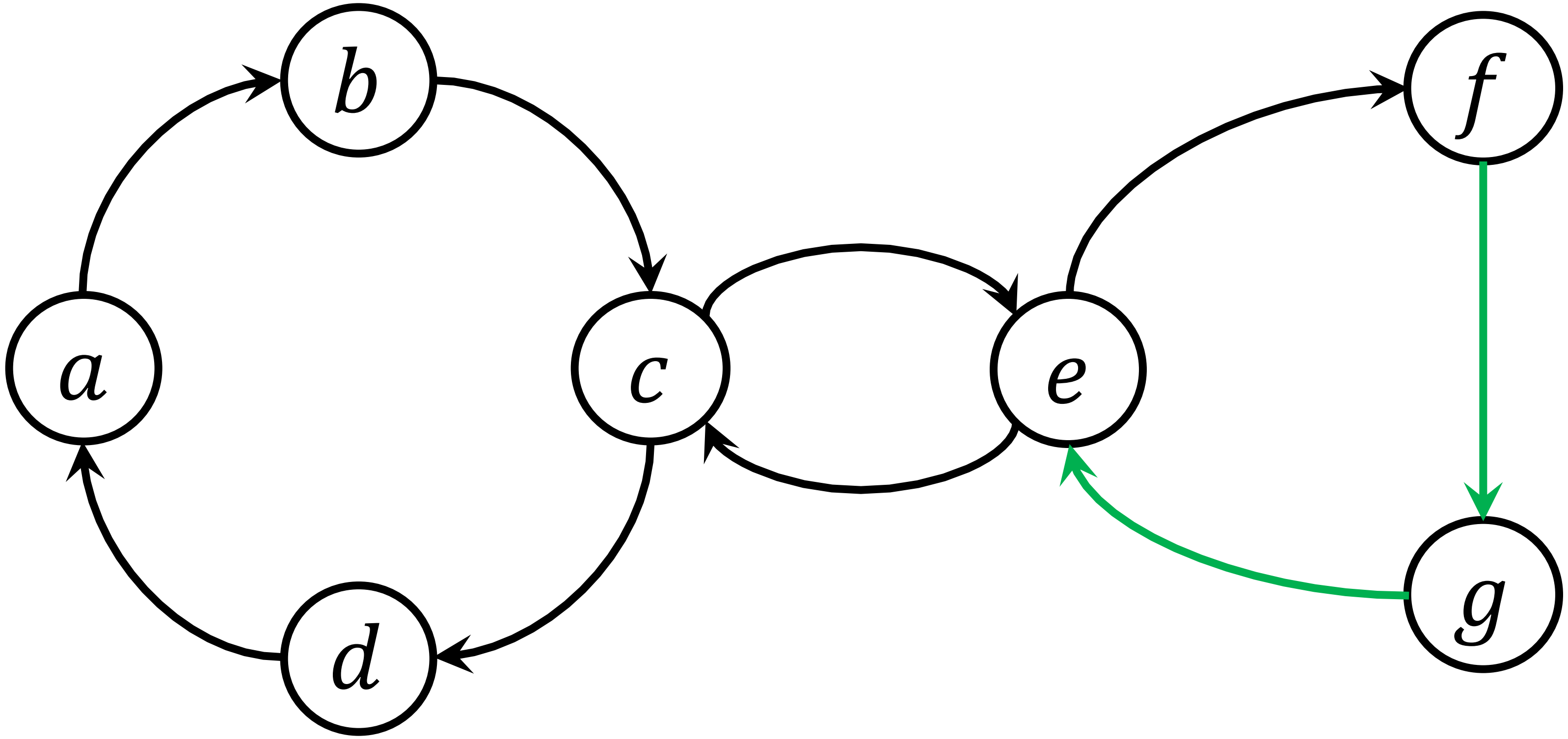
Green indicates a deleted

tour:



Finding the tour

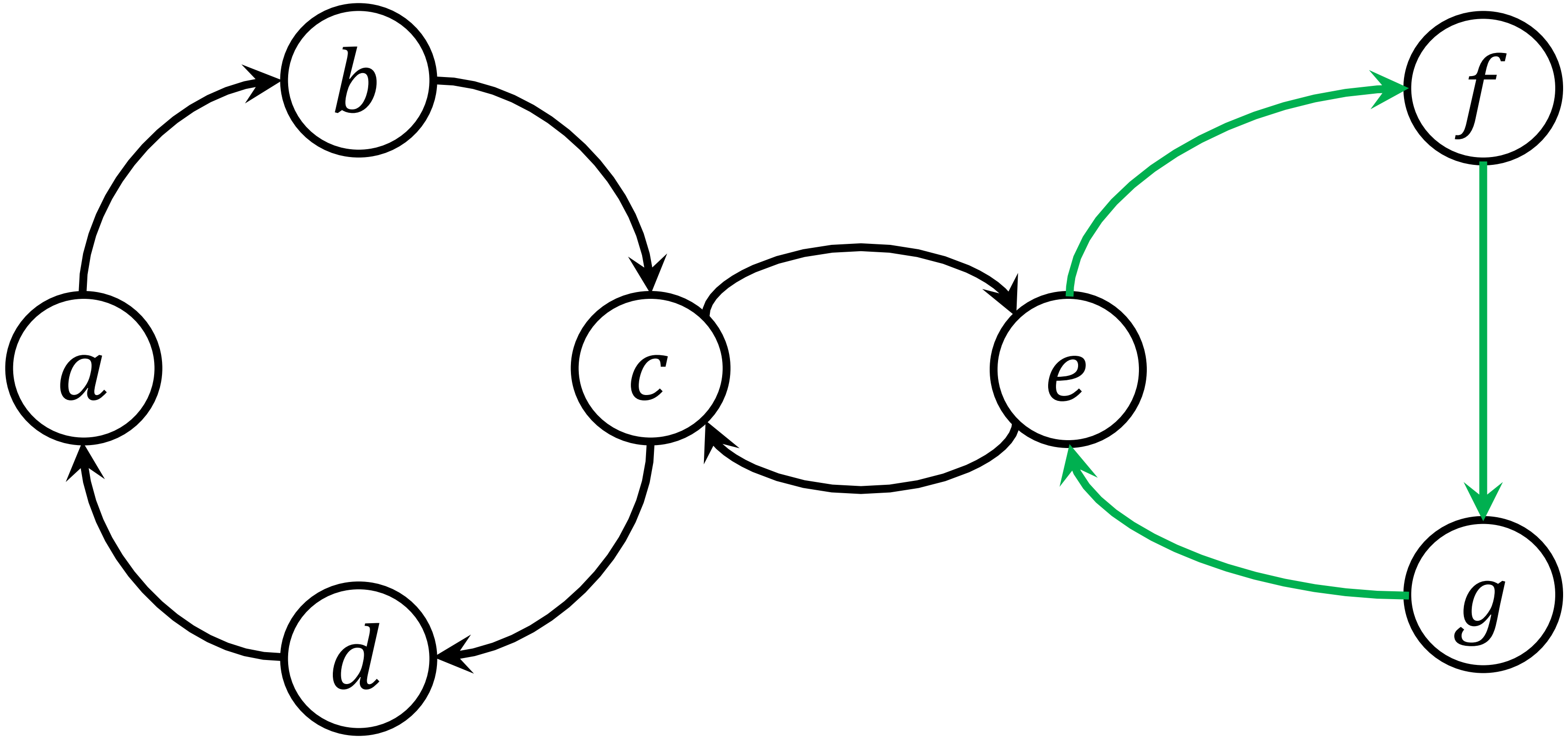
tour:



Finding the tour

tour: f

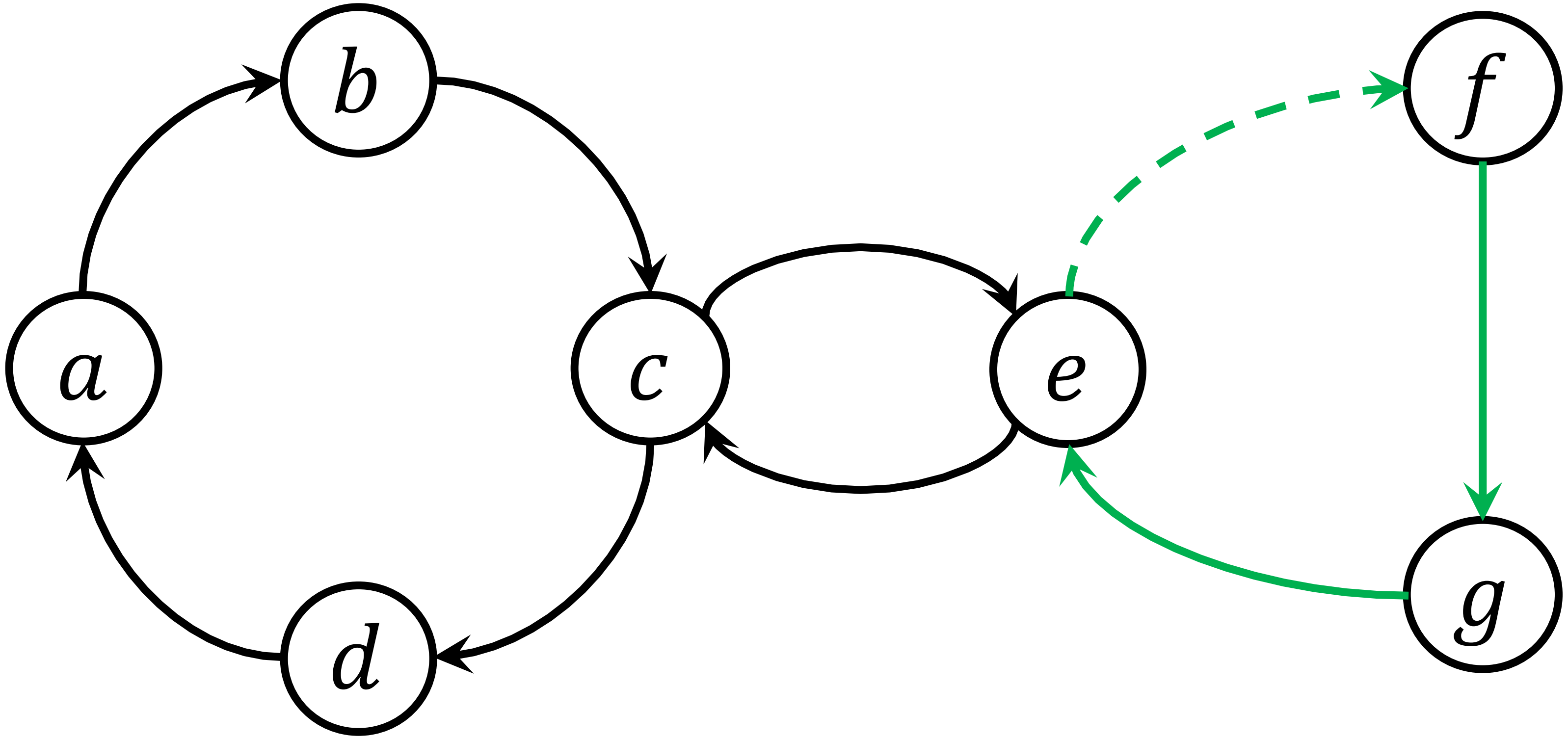
At this point all of edges in $Adj[f]$ are deleted, so we add f to the tour.



Finding the tour

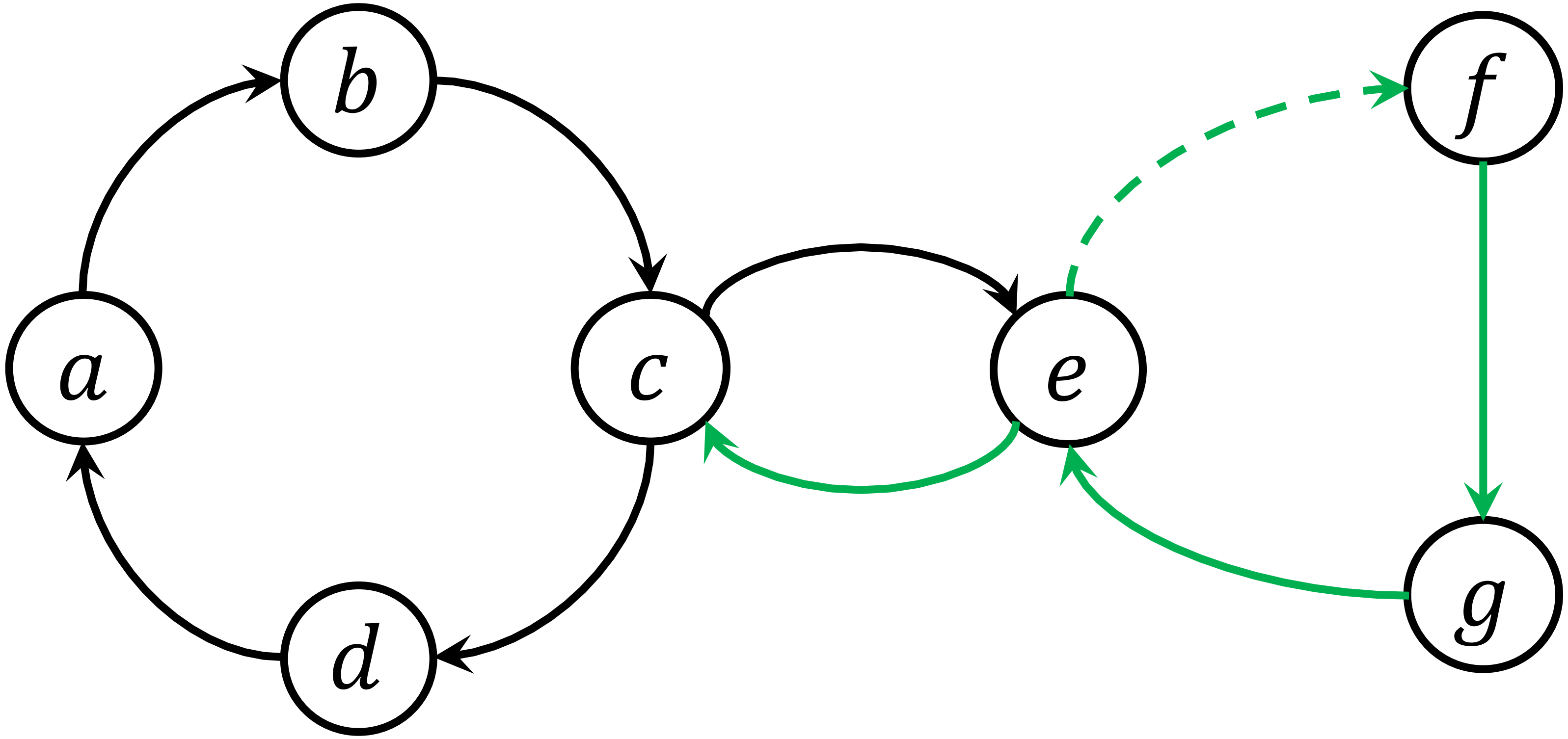
Now, we backtrack from f to e

$tour{:}\ f$



Finding the tour

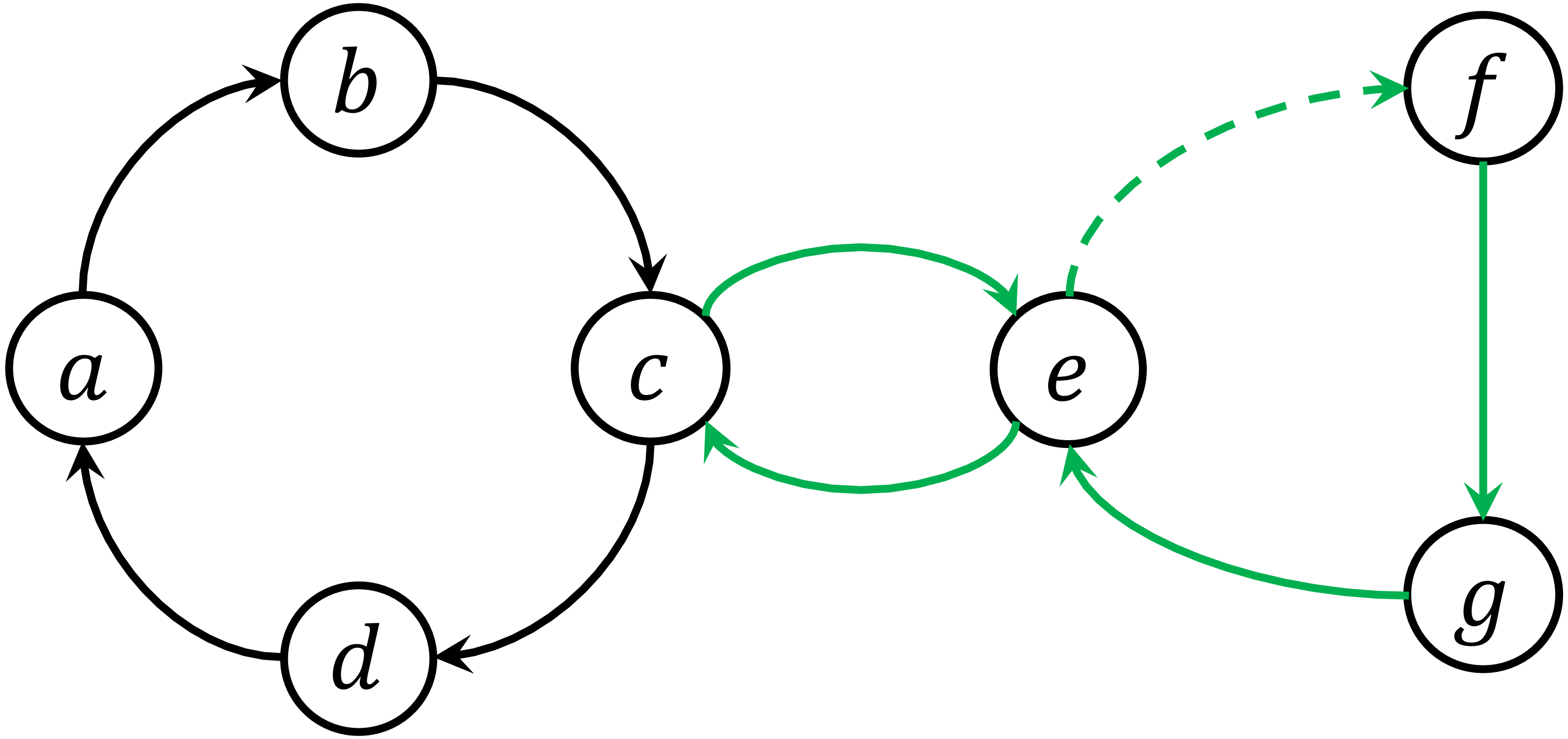
tour: f



Finding the tour

tour: f, e

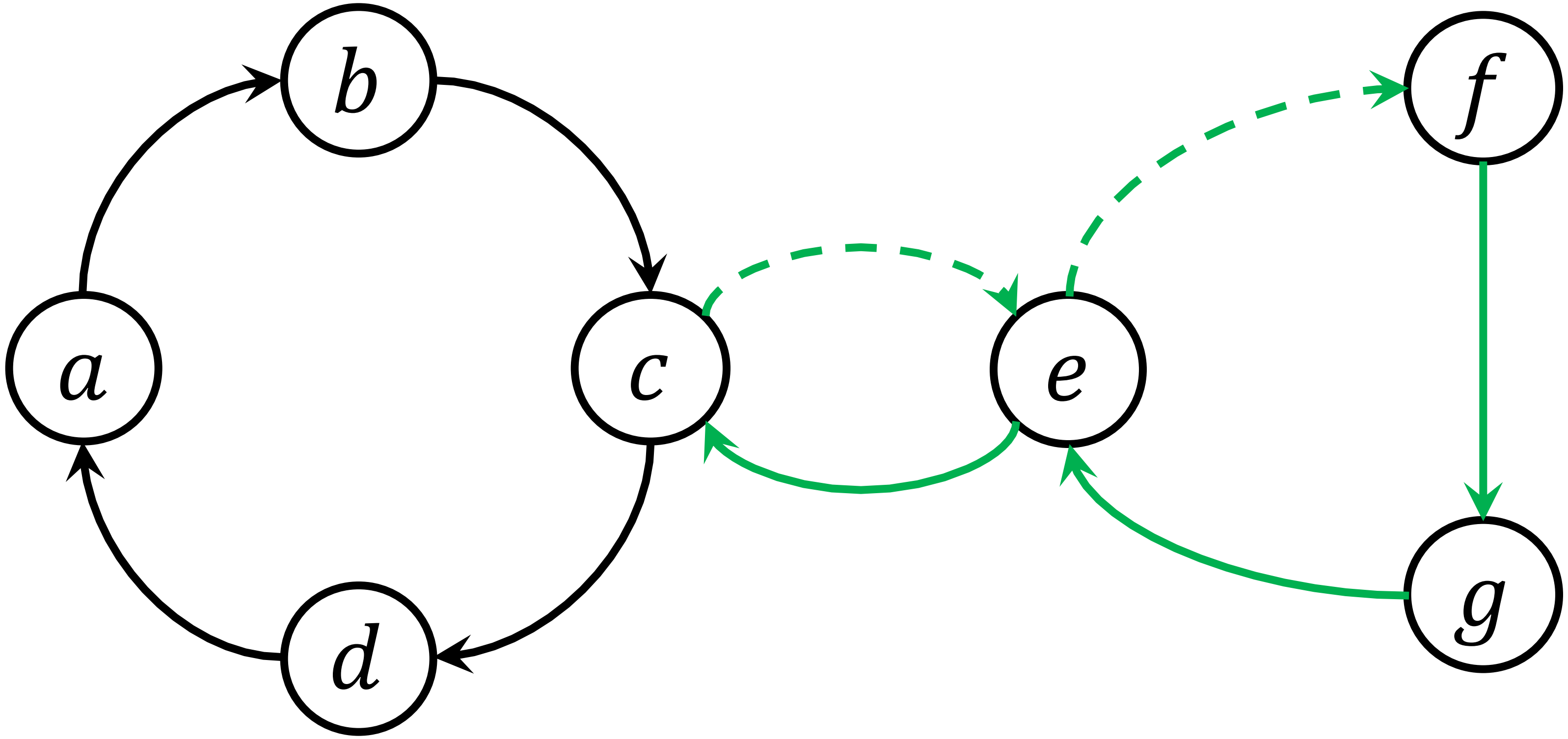
At this point all of edges in $Adj[e]$ are deleted, so we add e to the tour.



Finding the tour

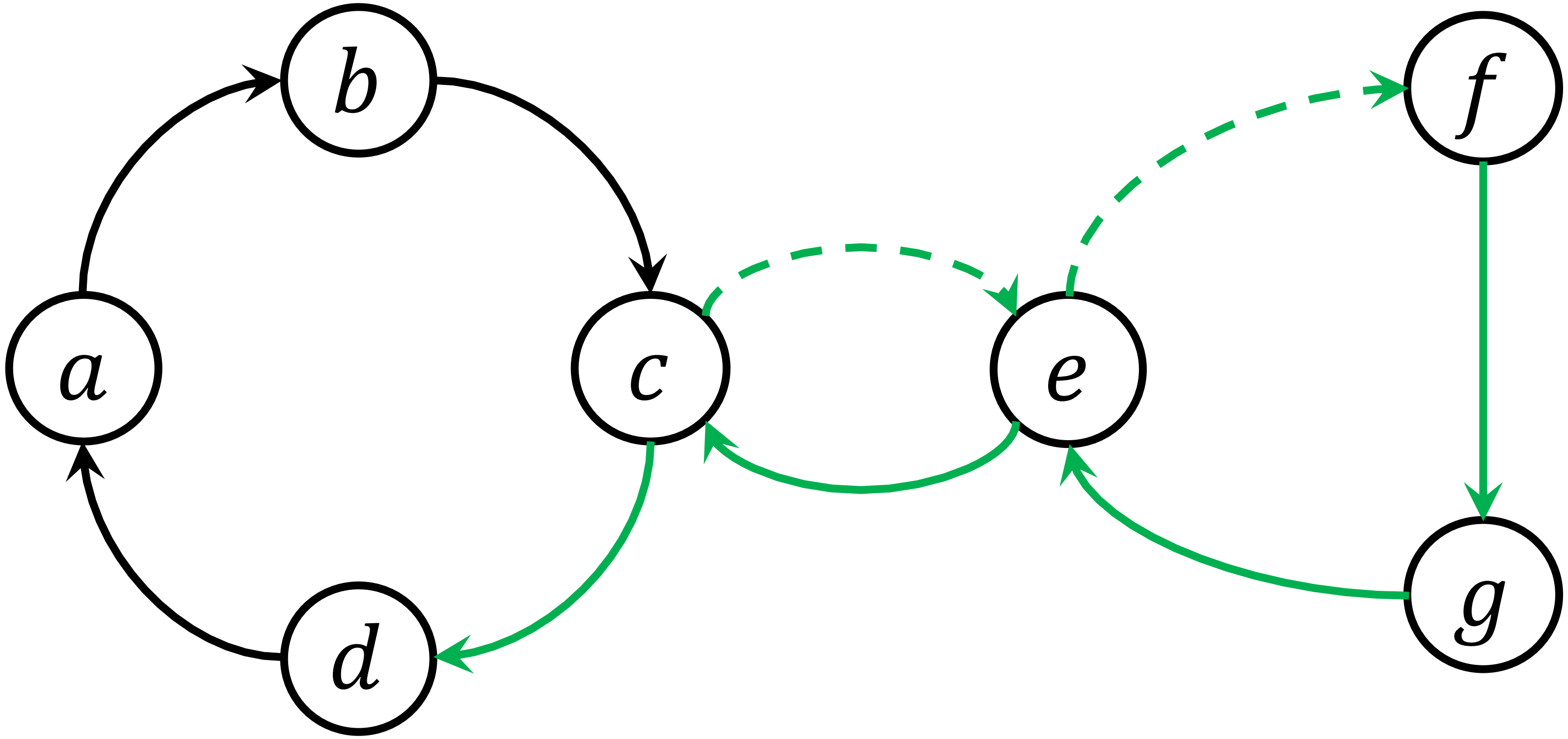
Backtrack from e to c

$tour: f, e$



Finding the tour

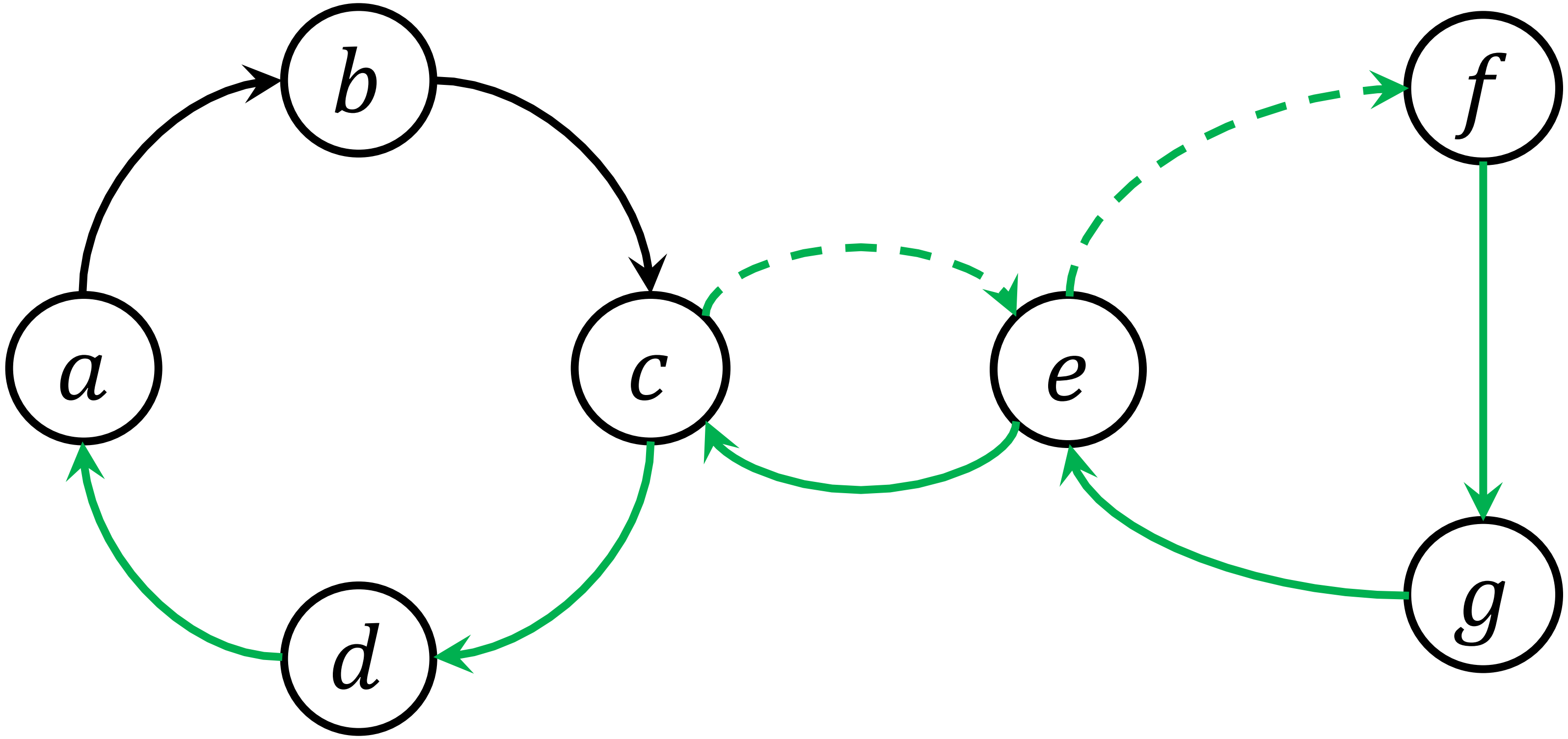
tour: f, e



Finding the tour

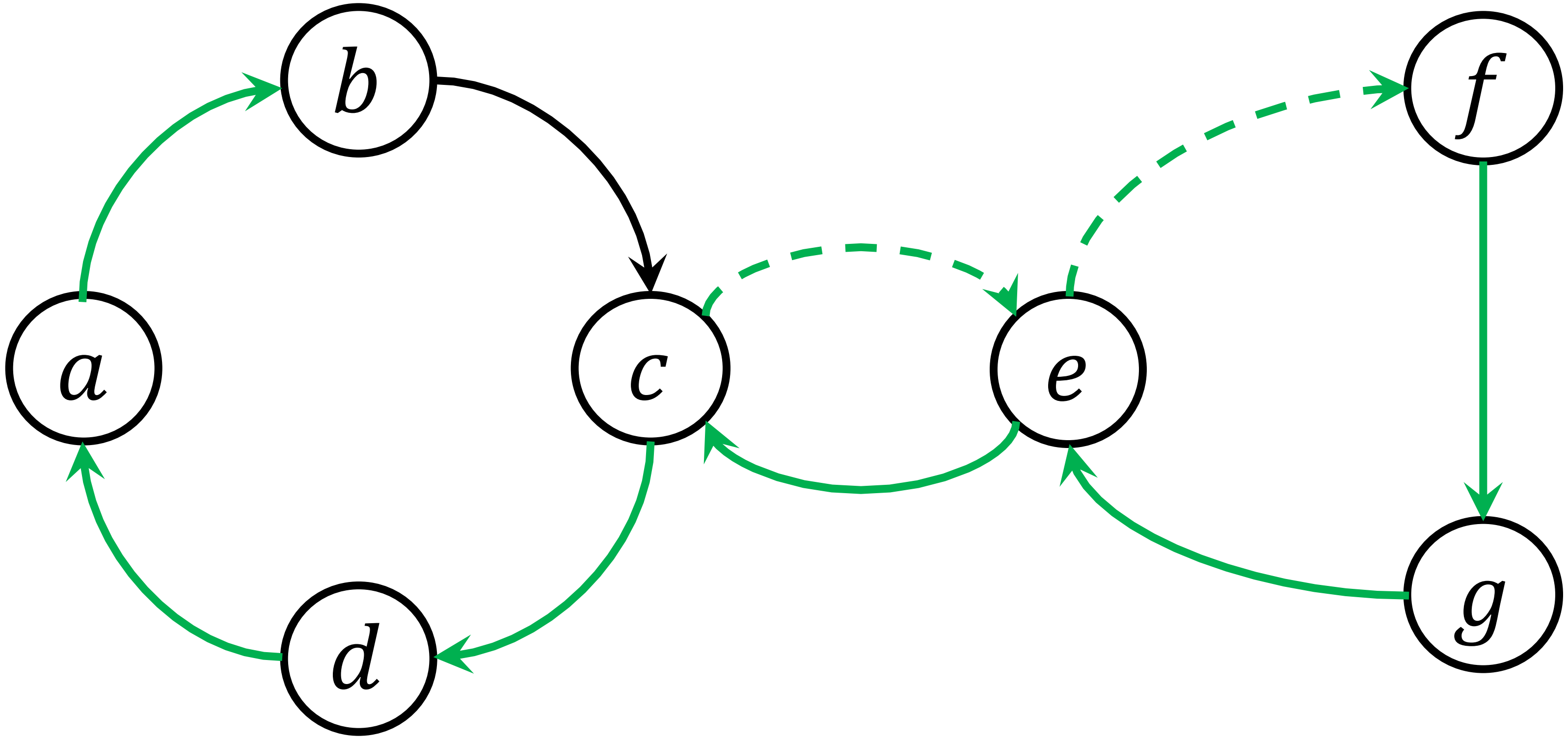
tour: f, e

At this point all of edges in $Adj[e]$ are deleted, so we add e to the tour.



Finding the tour

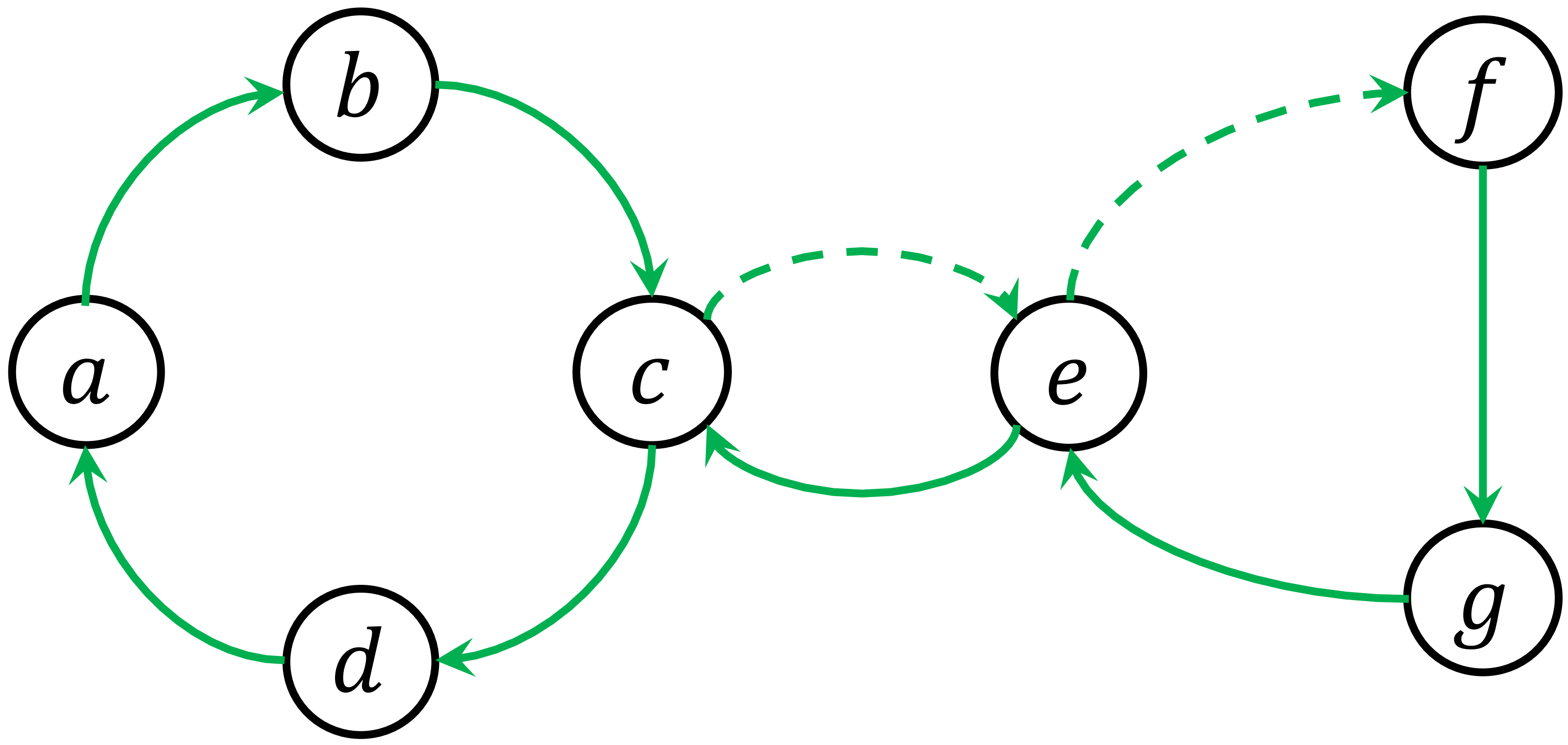
tour: f, e



Finding the tour

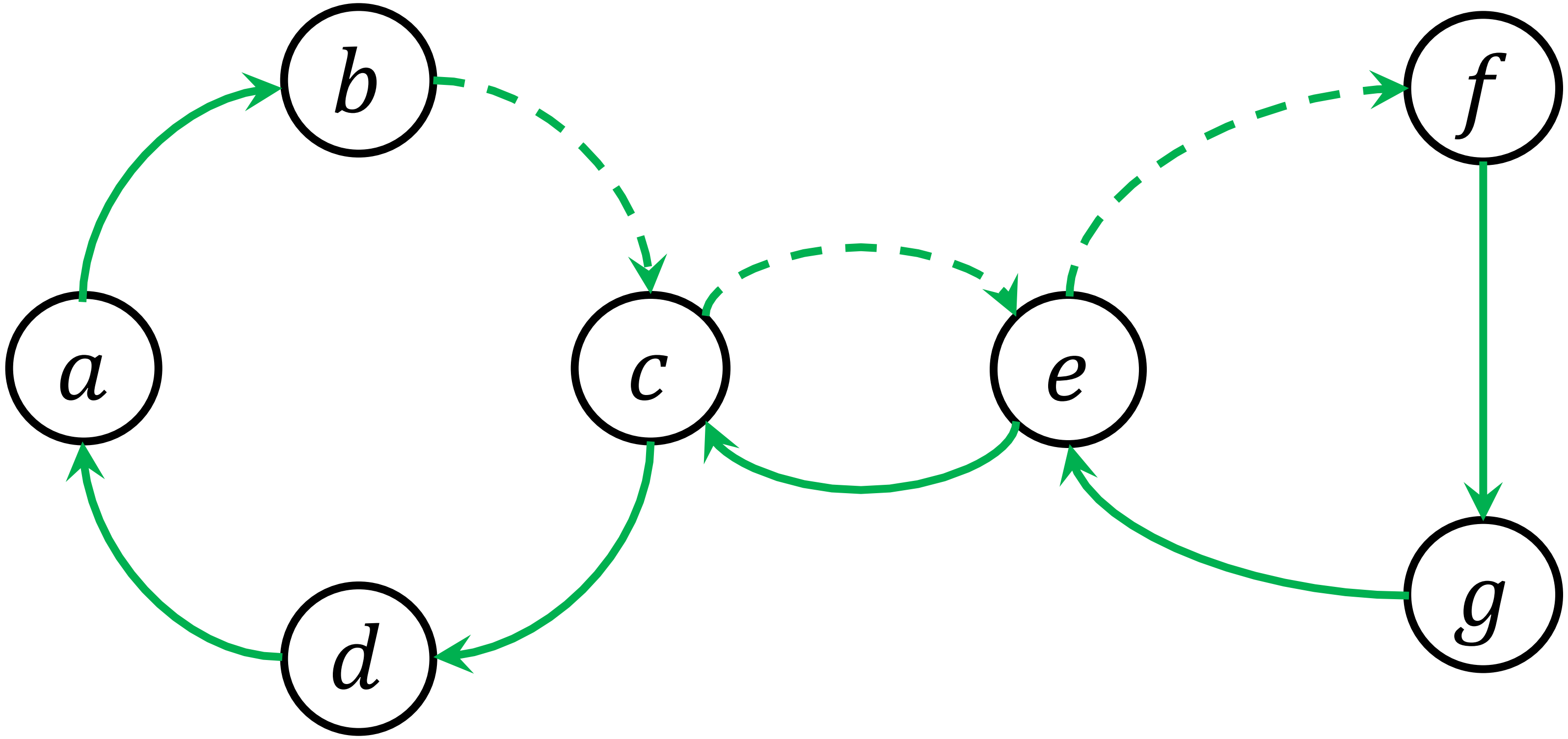
tour: f, e, c

At this point all of edges in $Adj[c]$ are deleted, so we add c to the tour.



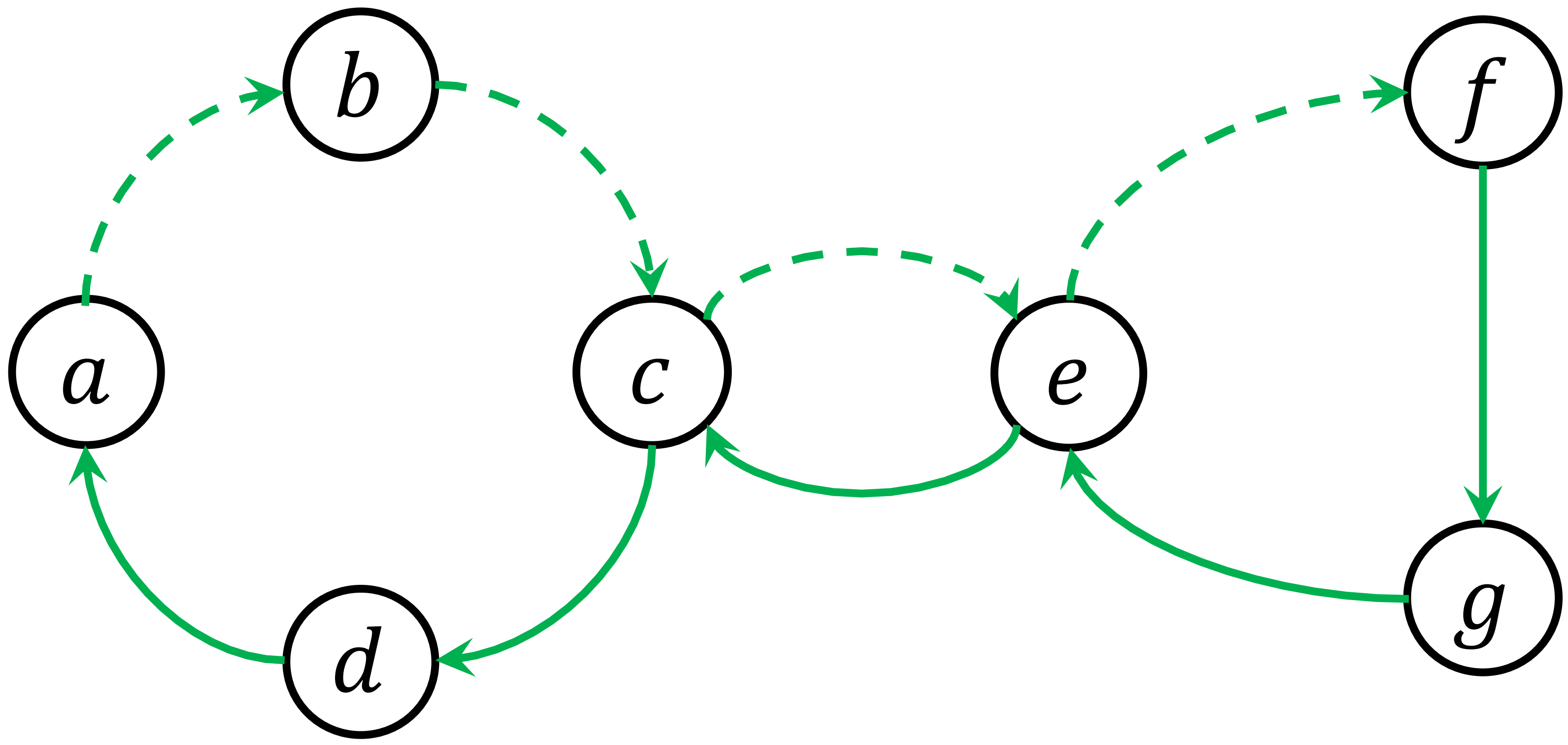
Finding the tour

tour: f, e, c, b



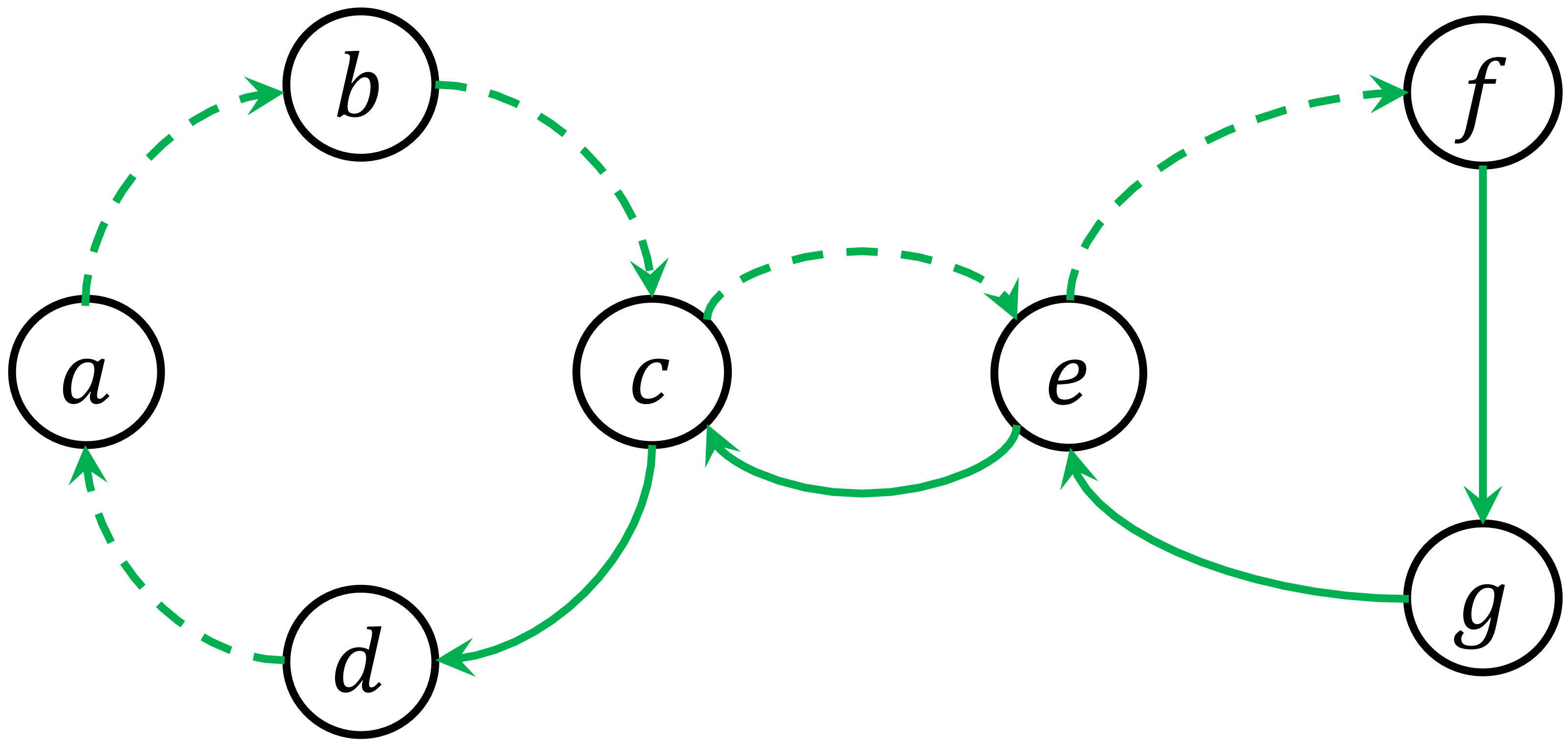
Finding the tour

tour: f, e, c, b, a



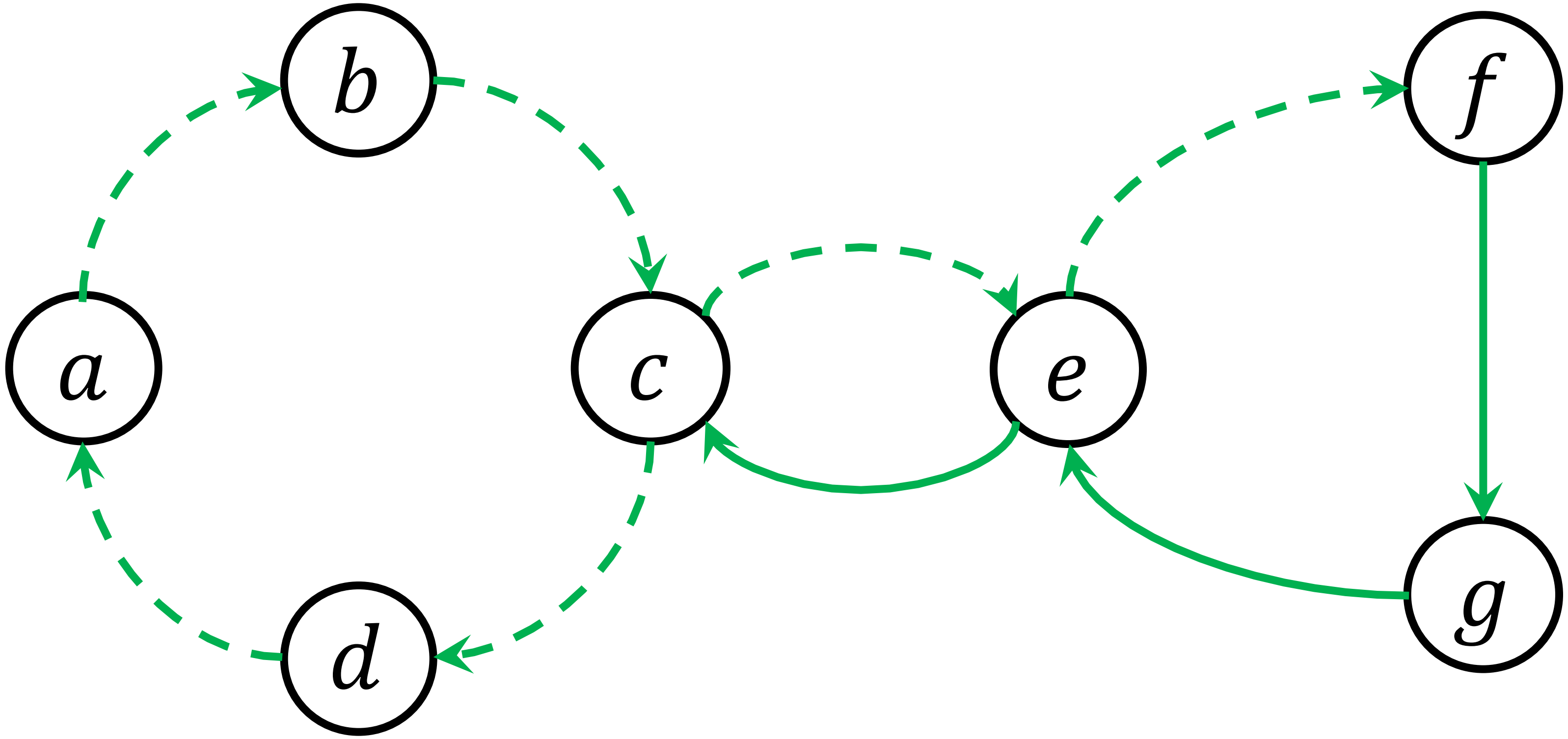
Finding the tour

tour: f, e, c, b, a, d



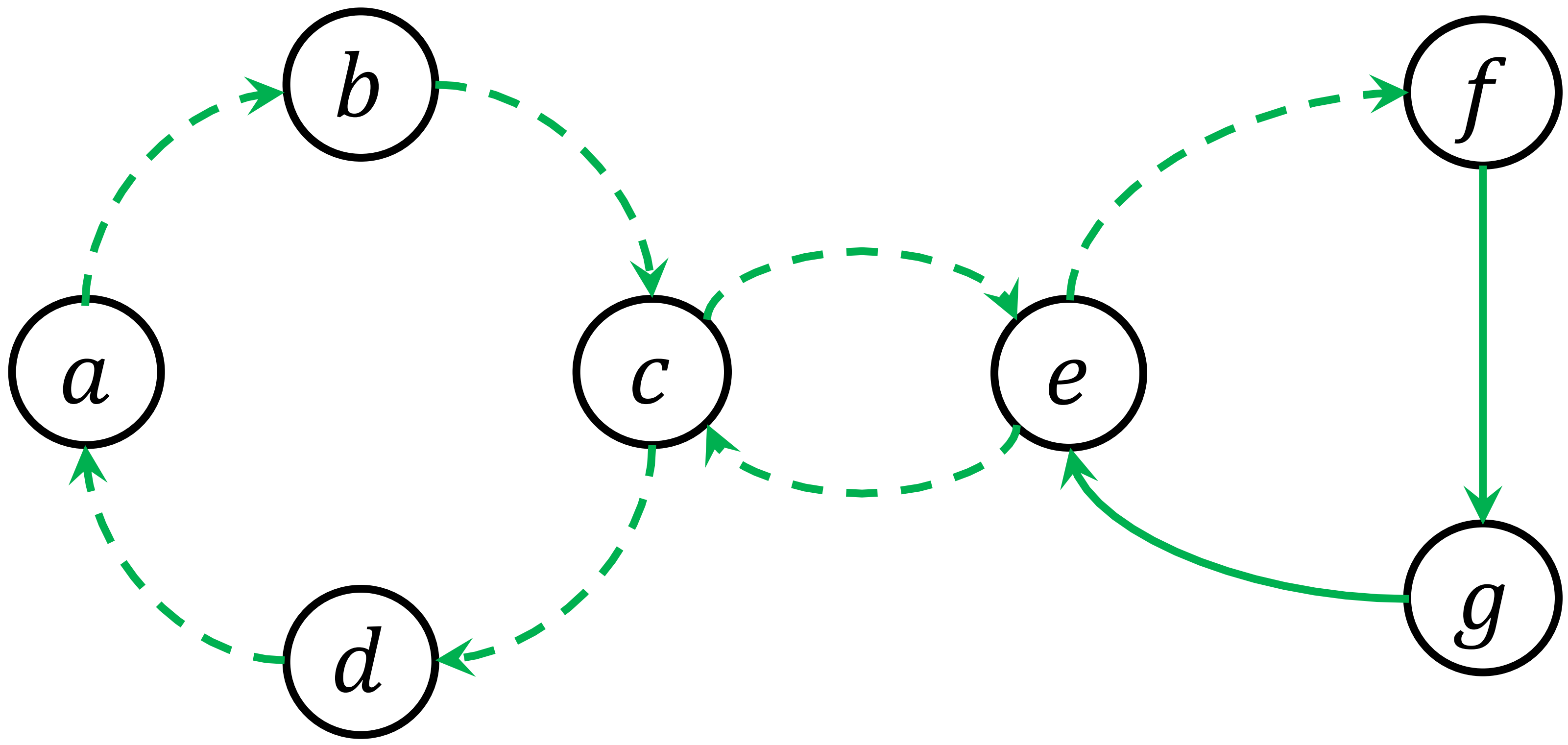
Finding the tour

tour: f, e, c, b, a, d, c



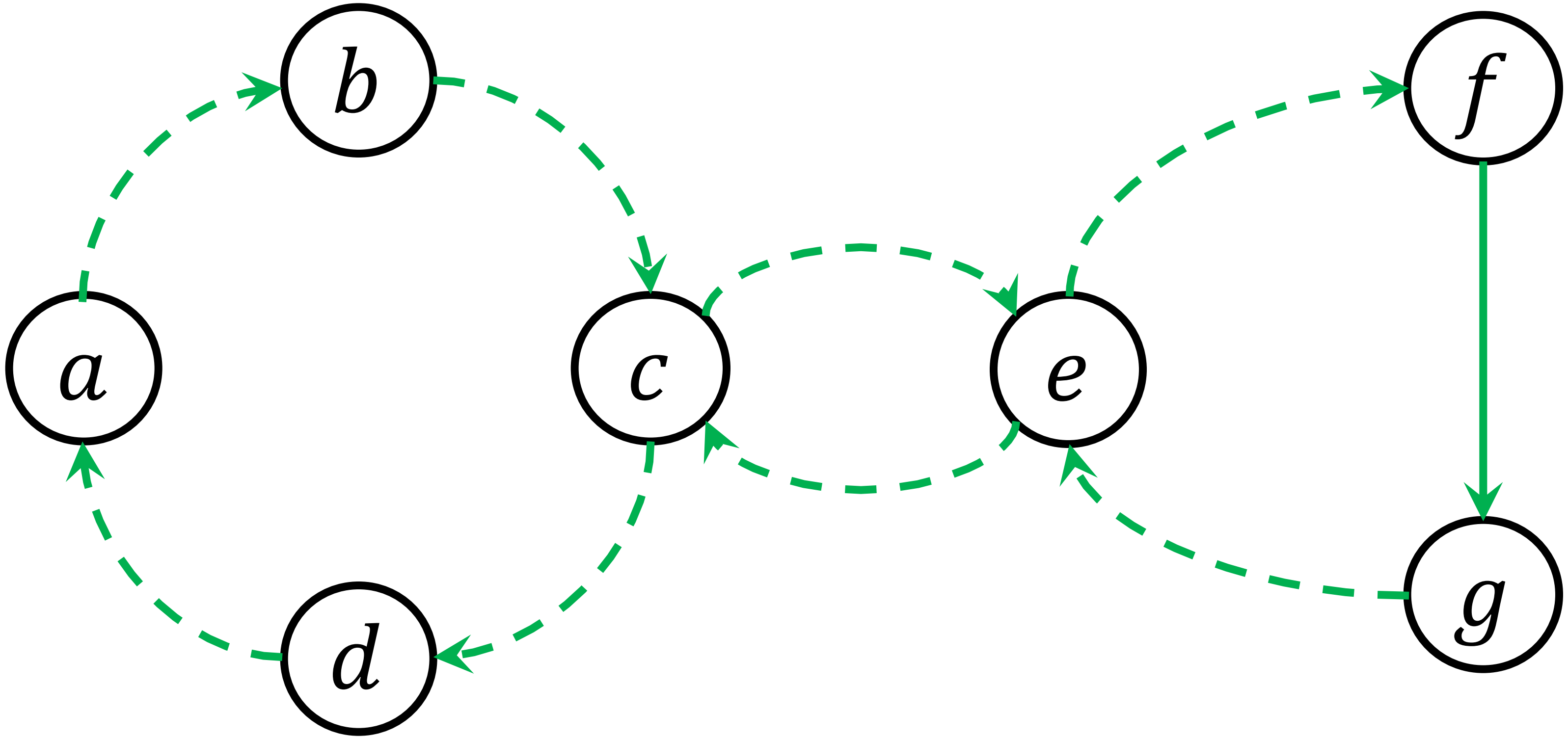
Finding the tour

tour: f, e, c, b, a, d, c, e



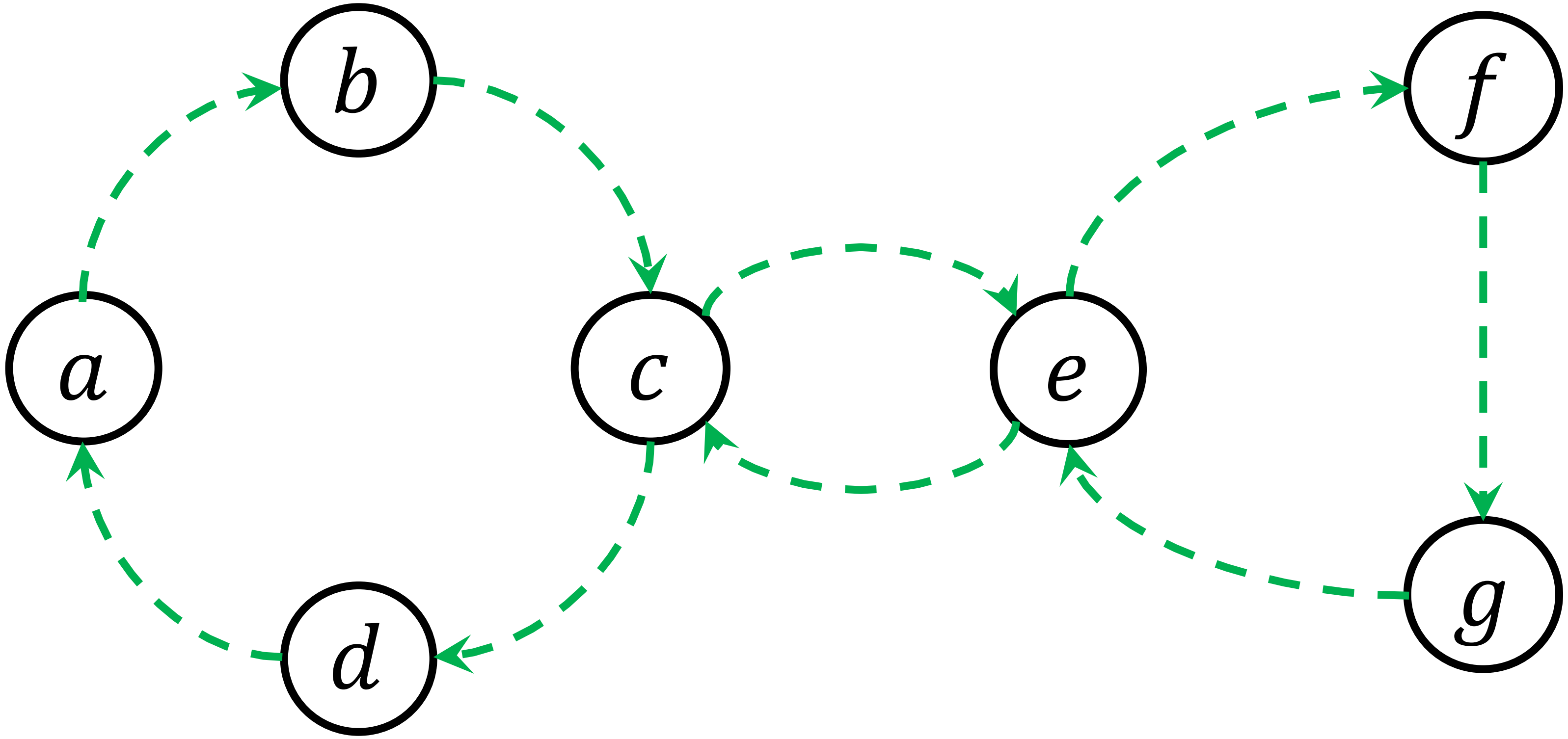
Finding the tour

tour: $f, e, c, b, a, d, c, e, g$



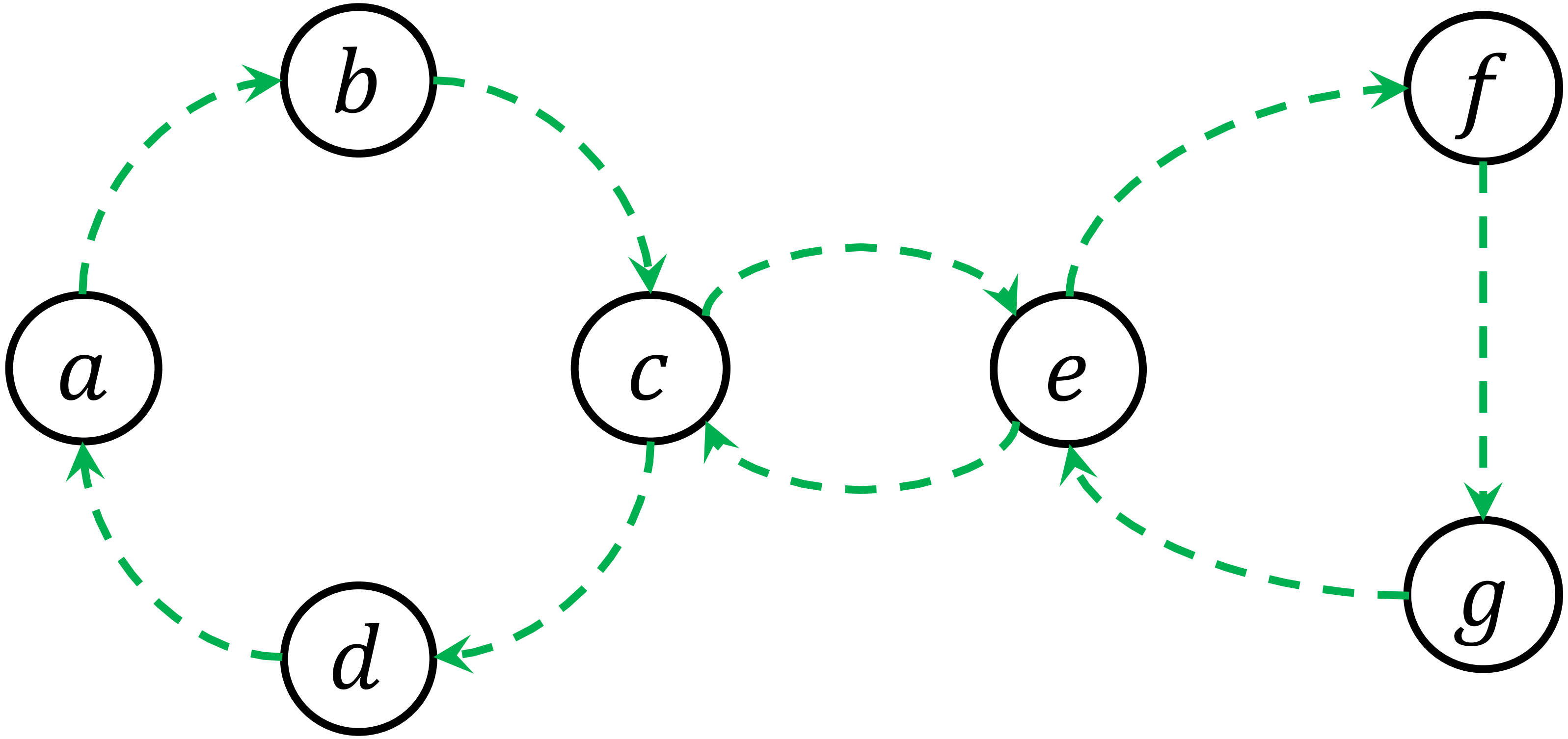
Finding the tour

$tour: f, e, c, b, a, d, c, e, g, f$



Finding the tour

Now we **reverse the list** to obtain the Eulerian tour.



$tour\ (reversed)\!: f, g, e, c, d, a, b, c, e, f$

Proof (not mandatory)

To be able to understand the algorithm you have to think **recursively**. Let's say we start with node x, then this node x must belong to a cycle C. So, in order to print the Eulerian tour we should print the nodes in cycle C and whenever we get to a node in this cycle that could lead to other potential cycles, we print those cycles recursively and then come back to print the rest of the cycle C.

This is exactly the reason that we put a node in the list only when its adjacency list becomes empty. For simplicity let's say on this cycle C there is only one node y that is connected to other cycles (the we can generalize the argument if there are multiple nodes like y).

Say cycle C: $x = x_1, x_2, x_3, \dots, x_i = y, x_{i+1}, x_{i+2}, \dots, x_k = x$.

When we get to the end of C then all nodes have emptied their lists. So, we put $x_k, x_{k-1}, \dots, x_{i+1}$ to the list. The cycles visited from y are already in the list, and then we add $x_{i-1}, x_{i-2}, \dots x$ to the list. So, to get the actual order we reverse the list and then return it.

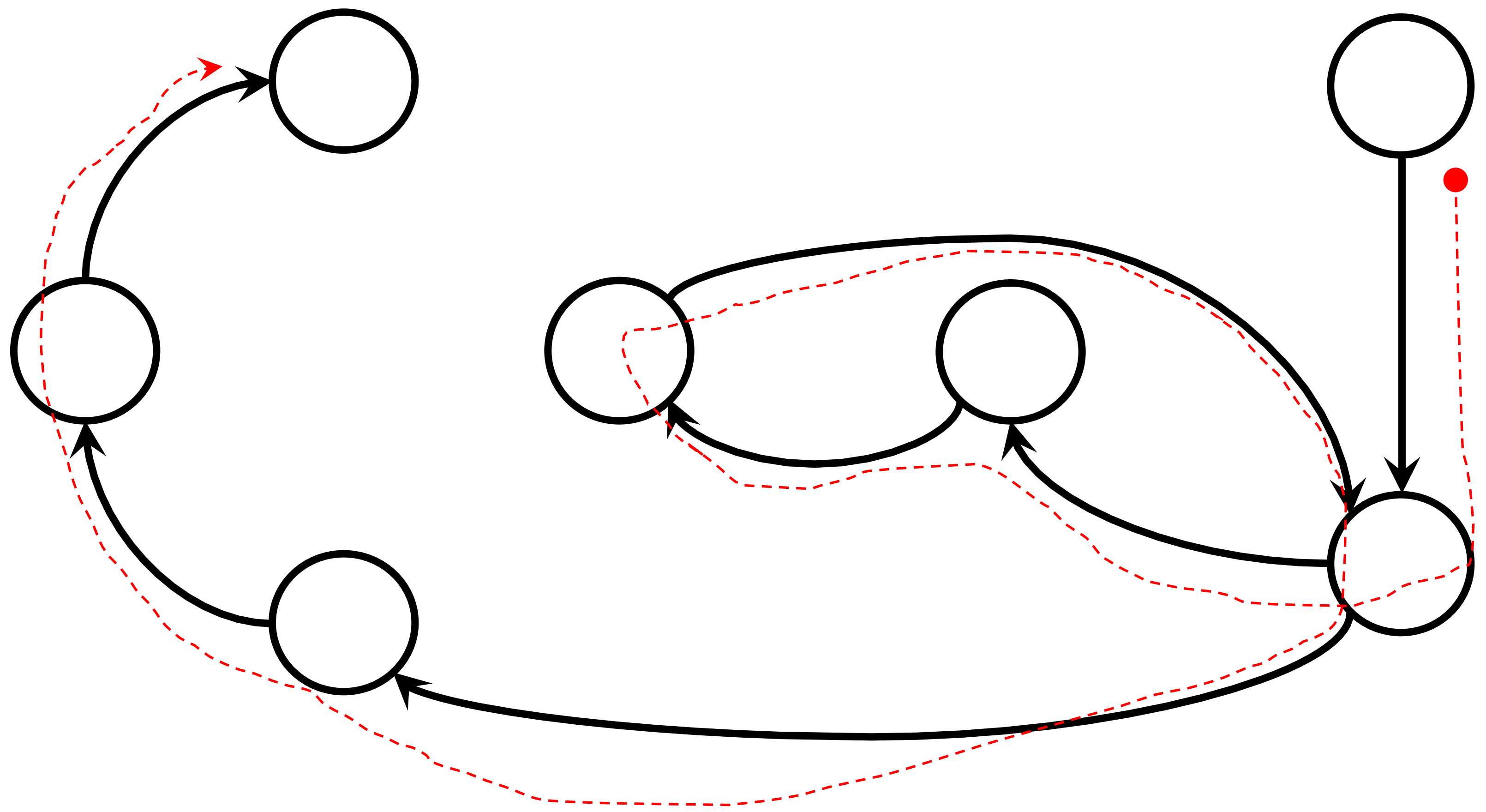
Eulerian path

- We say a graph G has an **Eulerian path** if:

1. There is a **trail** that **visits all edges** in G (exactly once).

- Note that in this variation the start and the end of the trail are not the same.

Eulerian path



Eulerian path

- A **directed graph** has an **Eulerian path** if

1. There are exactly two vertices v, u, such that:

$$outdegree(v) = indegree(v) + 1$$
$$indegree(u) = outdegree(u) + 1$$

2. For all other vertices, indegree and outdegree are equal

- In this case, we can pick v as the start point to call $\textsc{Eulerian-Tour-Rec}$.

Eulerian path

- An **undirected graph** has an Eulerian path if there are exactly **two** vertices of **odd degree**.

- In this case, we can pick any of the odd-degree nodes as the start point to call EULERIAN-TOUR-REC.

Diameter

- **Diameter** of a graph is defined as

$$\max_{u,v} \delta(u, v)$$

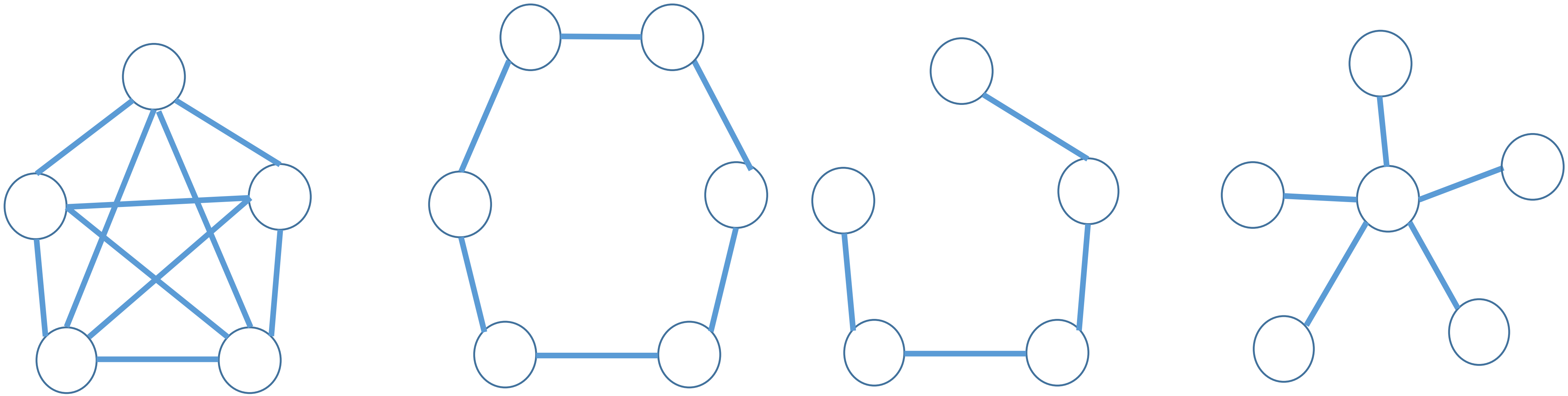
- $\delta(u, v)$ is the **length of the shortest** path between u and v.

✓Diameter is an interesting parameter of a graph it corresponds to the distance of the two furthest nodes.

✓We consider the problem of finding the diameter in an undirected graph.

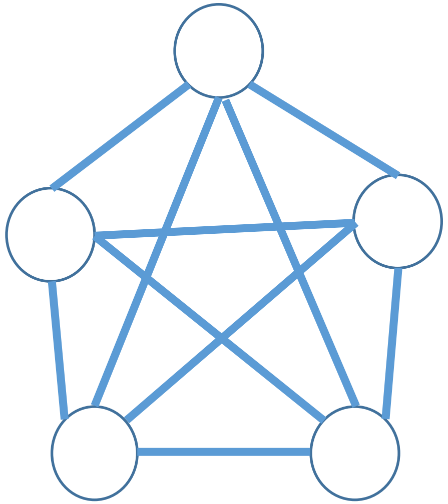
Diameter

- Examples:

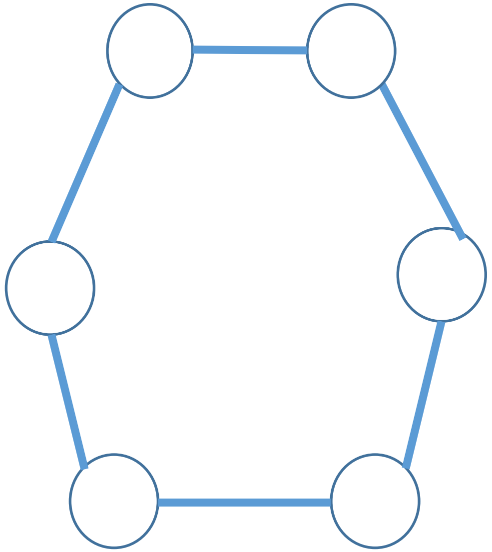


Diameter

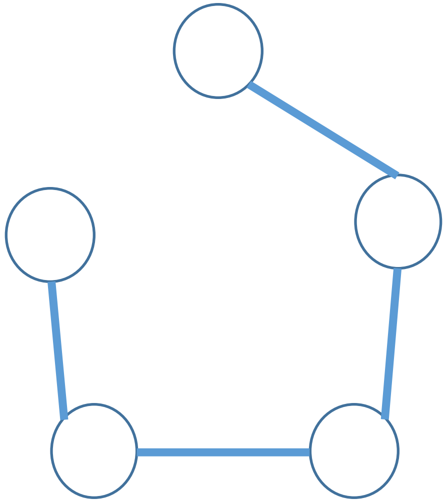
- Examples:



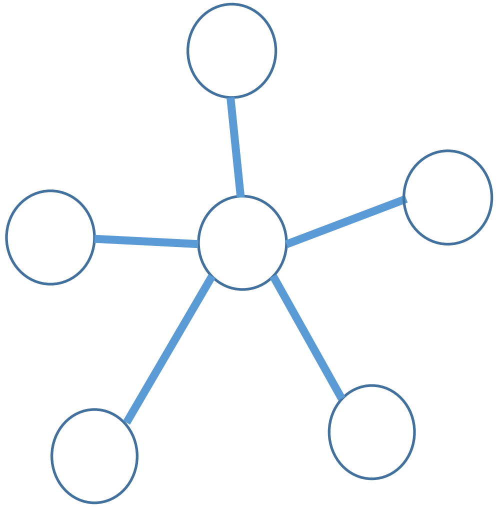
Complete graph
Diameter = 1



Cycle graph
Diameter = floor of half the size of the cycle= 3



Path graph
Diameter = length of the path = 4



Star graph
Diameter = 2

Diameter

- **Question:** How can we find the diameter of a graph?

Diameter

- **Question:** How can we find the diameter of a graph?
- **Answer:** If the graph is disconnected the answer is ∞. However, if it's connected we can run a BFS search from every node, and find the maximum shortest path from that node. Then, we get the maximum of all of these values over all nodes to obtain the diameter. The complexity is $O\big(n \times (m+n)\big) = O(n^3)$.

Diameter

- **Question:** How can we find the diameter of a graph?
- **Answer:** If the graph is disconnected the answer is ∞. However, if it's connected we can run a BFS search from every node, and find the maximum shortest path from that node. Then, we get the maximum of all of these values over all nodes to obtain the diameter. The complexity is $O\big(n \times (m+n)\big) = O(n^3)$.

- **However,** if the graph is a tree we can do it in $O(n+m)$, i.e. linear time!

Diameter

- In a tree there is **exactly one path** between any pairs of nodes. So, each path in a tree is actually a shortest path.

- The longest shortest path should be **between two leaves** because otherwise we can extend the two ends of the path to reach to leaves.

Algorithm

DIAMETER(T) //T is a tree

1. pick any node u in T
2. run a BFS from u
3. let x be some node at maximum distance from u
4. run another BFS from x
5. **return** the maximum distance found at step 4

Proof

- The claim is that ***x* is on some shortest path of maximum length**.

- So, the second BFS will find that path, and the returned diameter is correct.

Proof (not mandatory)

- Let u be the root of the first shortest path tree.
- Let x be a node that is at max distance from u.
- We use proof by contradiction. Say there are two nodes s, t such that the $\delta(s, t)$ is strictly bigger than the max distance between x and any other node.
- There are two cases:

1. The $s - t$ path and $u - x$ path do not intersect.
2. They intersect.

- In either case we can show that the $x - t$ path or the $x - s$ path is at least as long as the $s - t$ path.