Machine Representation of a Graph

September 22, 2026

Overview

- Walks
  - Hamiltonian
  - Euclidean
- Machine Representation
  - Adjacent Matrix
    - Isomorphisms
    - K-step walks
  - Adjacent List
    - Depth First Search
- Topological Sort

Walks and Cycles

Walks and Cycles

Definition (walks and cycles)

A walk, $\{v_1, v_2, ..., v_n\}$, in a graph is a sequence of vertices (not necessarily exhaustive nor distinct) for which $\{v_i, v_{i+1}\}$ is an edge for every $1 \leq i < n$.

If $v_1 = v_n$, the walk is called a cycle or a closed walk. Otherwise the walks is said to be open.

Hamiltonian Cycle

Definition (Hamiltonian Walk)

A Hamiltonian Walk is a walk which visits every vertex of a graph exactly once.

Hamiltonian Cycle

Definition (Hamiltonian Walk)

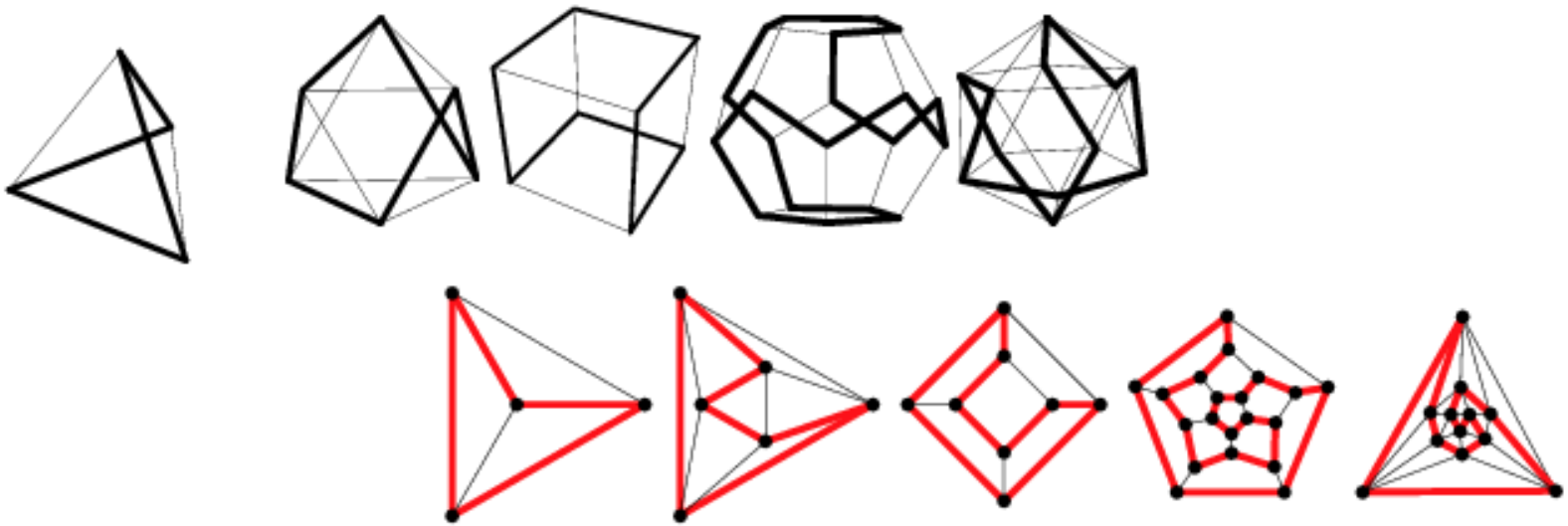
A Hamiltonian Walk is a walk which visits every vertex of a graph exactly once.

Definition (Hamiltonian Cycle)

A Hamiltonian Cycle, is a cycle, $\{v_1, v_2, ..., v_n\}$ for which $\{v_1, v_2, ..., v_{n-1}\}$ is a Hamiltonian walk.

Hamiltonian Cycle

The following image taken from Wolfram MathWorld illustrates some Hamiltonian cycles.



Hamiltonian Cycle

The problem of finding a Hamiltonian cycle is NP-complete.

The way to find one is to exhaust the possibilities.

The famous traveling salesman problem (TSP) is to find a shortest Hamiltonian cycle on a weighted graph.

Visiting all permutations

In Computer Programming we sometimes need to generate or at least visit all permutations. This is of course computationally expensive. For small number of objects, this can be done using a recursive function.

There are other ways. Many are explained in Donald Knuth: *Art of Computer Programming. Volume 4* Section 7.2.1.2. Also available as Volume 4 Fascicle 2.

Visiting all permutations

One thing to keep in mind is sometimes you will be asked to generate all permutations.

This is usually not possible, so whoever is asking you to do it needs to rethink his request.

Easier is to visit each permutation. Or visit some permutations, either a limited number, or until some time-out.

Eulerian Walks

Definition (Eulerian Walk)

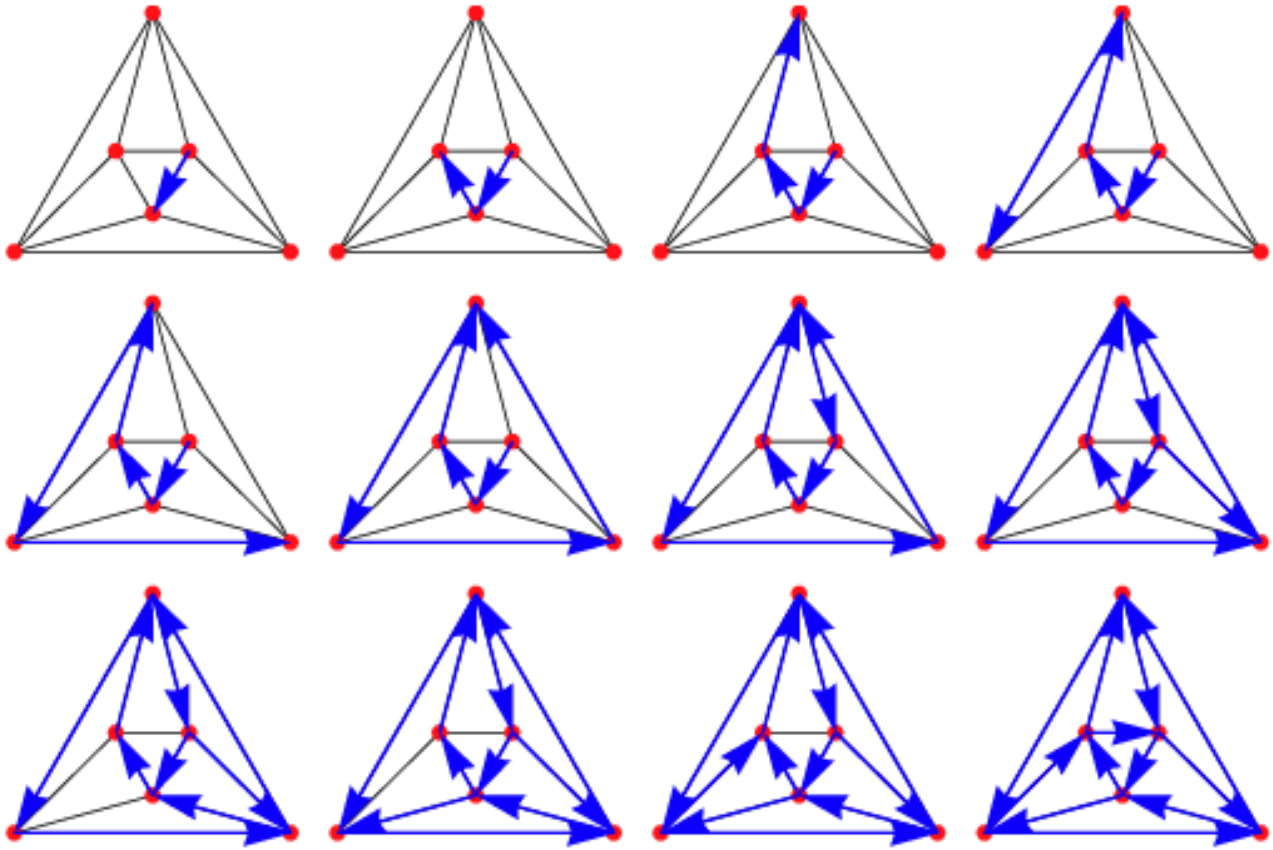
An Eulerian walk is a walk which visits every edge exactly once.

Definition (Eulerian Cycle)

An Eulerian cycle is a cycle which visits every edge exactly once.

Eulerian Walks

The following image taken from Wolfram MathWorld illustrates some Eulerian cycles.



Eulerian Walks

Finding an Eulerian cycle is easy.

Any graph which is connected and has only even-degree vertices, has an Eulerian cycle.

If a connected graph has exactly two odd-degree vertices, then it has an Eulerian walk which connects the two odd-degree vertices.

Eulerian Walks

Definition

A graph containing an Eulerian cycle is called an Eurlerian graph.

Note that some authors define Eurlerian graph as a graph for which every vertex has even degree.
QUESTION: Are these two definitions equivalent?

Eulerian Walks

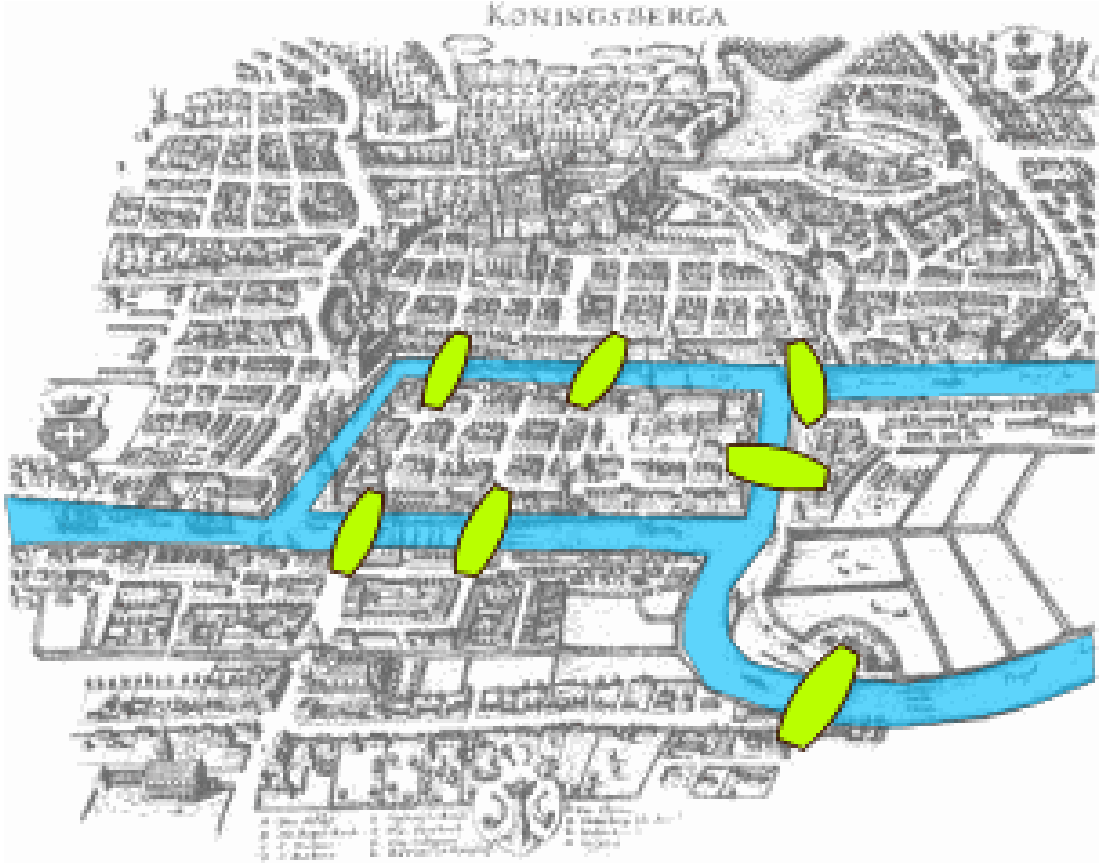
Definition

A graph containing an Eulerian cycle is called an Eurlerian graph.

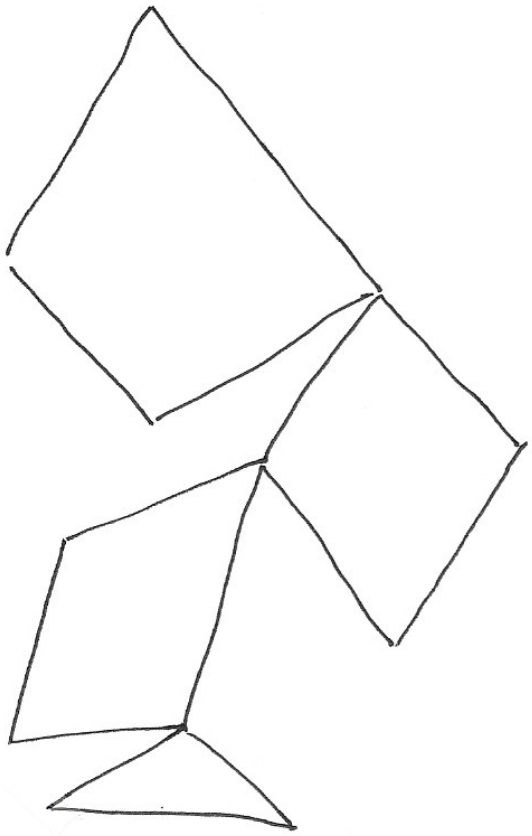
Note that some authors define Eurlerian graph as a graph for which every vertex has even degree.
QUESTION: Are these two definitions equivalent? The two definitions coincide if the graph is connected.

Eulerian Walks

A famous problem in graph theory, Seven Bridges of Königsberg is the question of finding an Eulerian cycle which crosses each of the Königsberg exactly once.

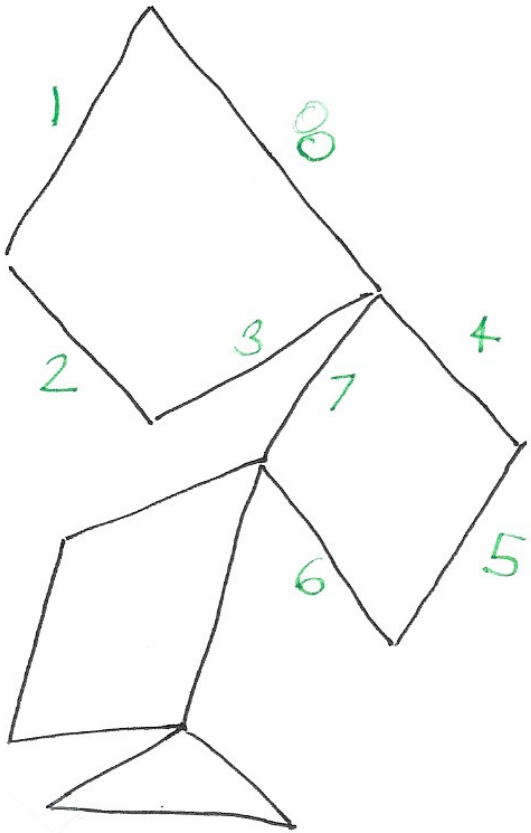


Example Eulerian Cycle



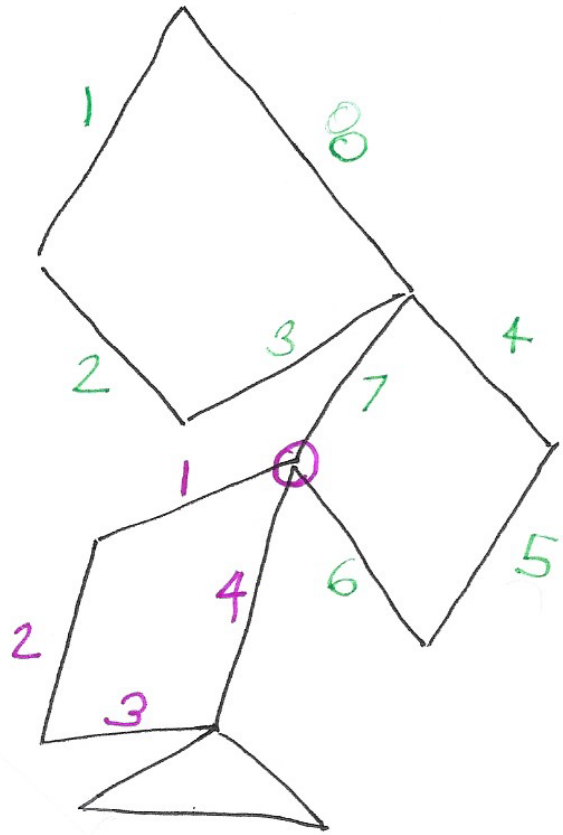
Just start anywhere walking edges.

Example Eulerian Cycle



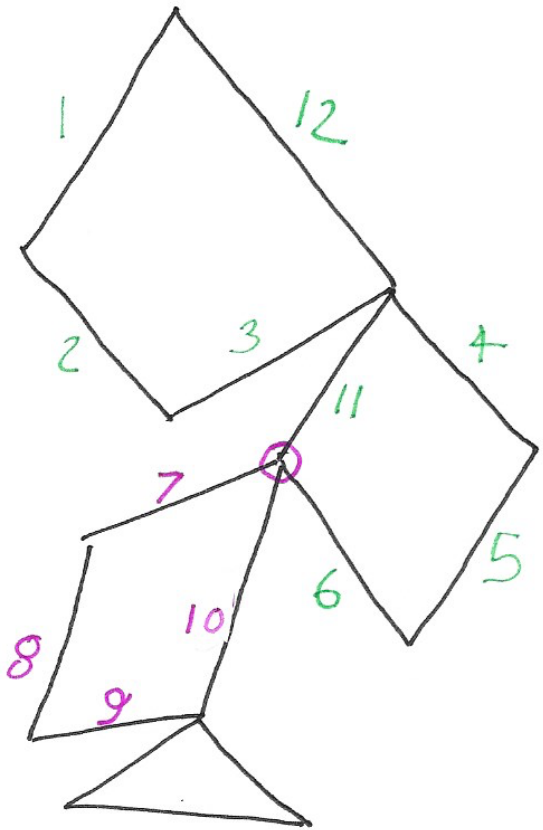
You may encounter the same vertex multiple times, but never walk the same edge twice.

Example Eulerian Cycle



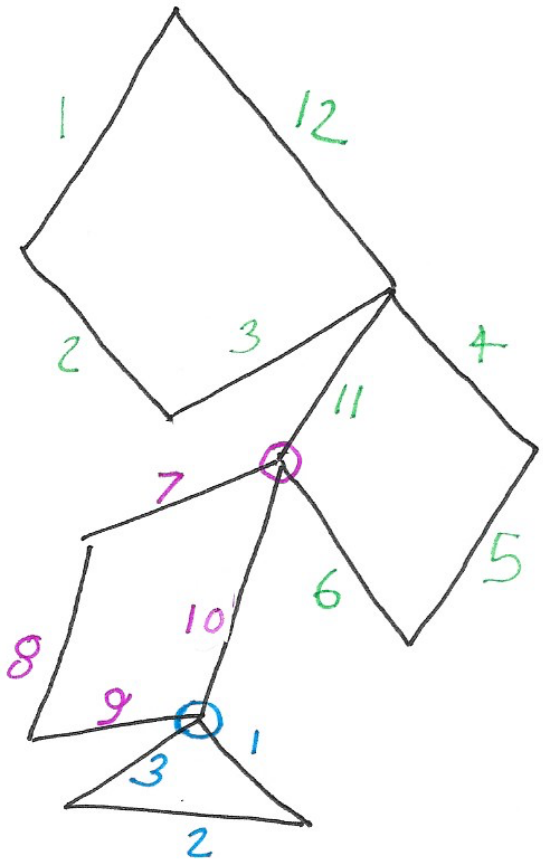
If edges remain, then some unused edges intersect the graph already walked; else the graph is not edge connected.

Example Eulerian Cycle



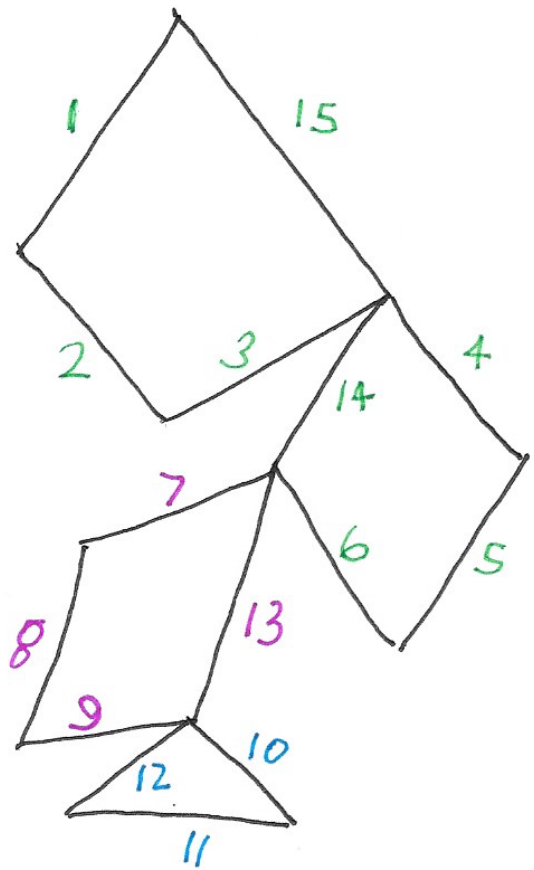
Insert the new cycle there.

Example Eulerian Cycle



Continue as long as edges remain.

Example Eulerian Cycle



Find an Eulerian Cycle (if there is one)

1. Start by finding any cycle.
   - Start anywhere. Keep walking without repeating edges until you can't go any further.
   - You'll automatically end up where you started.
   - If not, quit. There is no Eulerian Cycle.
2. If that cycle encompasses all the edges, you're done.
   - If not, remove the edges and find another cycle, starting at some vertex, v, already examined.
3. Now, join the two cycles at v (which is common to both cycles).
4. Continue this process until all edges are visited.

There are many ways to optimize this algorithm. See wikipedia for example.

Machine Representation of a Graph

Representing a graph

Definition

A simple graph, (V, E), is an object consisting of two finite sets V and E, such that $V \neq \emptyset$ and E is a set of two-element subsets of V. V is called the vertex set. E is called the edge set.

We might attempt to represent a graph programmically similar to its mathematical definition.

Naive Implementation

In keeping with this definition, one might attempt a programmatic definitions somewhat like the following.

```
class Graph[T] {
  type Edge = UnorderedPair[T]
  def V:Set[T]
  def E:Set[Edge]
}
```

Naive Implementation

- With such a class definition we can represent a graph whose Integer vertices as `Graph[Integer]`, similarly `Graph[String]`, or `Graph[Array[Boolean]]`.
- Unfortunately, this is rarely done because of performance.
- This representation does not facilitate common operations.
- For example: given a vertex, find the adjacent vertices.

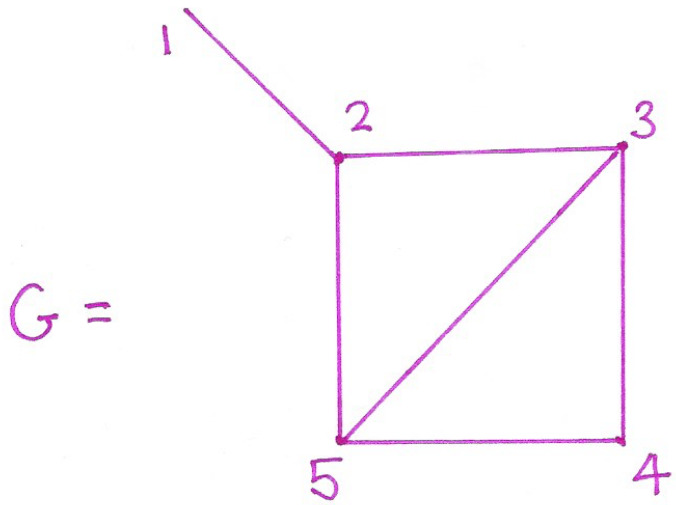
Common Implementation

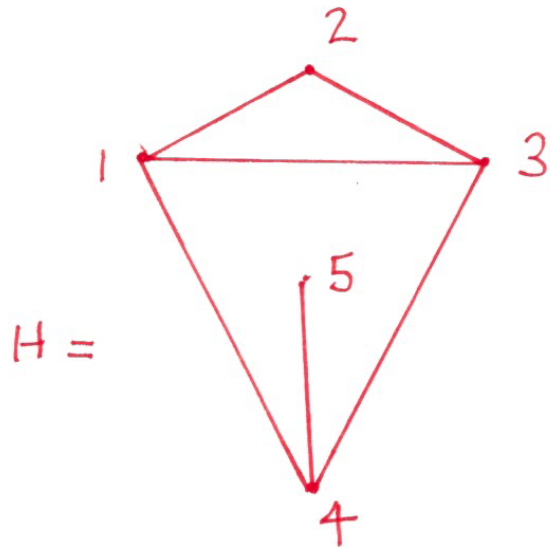
Two common representations are Adjacency Matrix and Adjacency List.

Adjacency Matrix

Adjacency Matrix

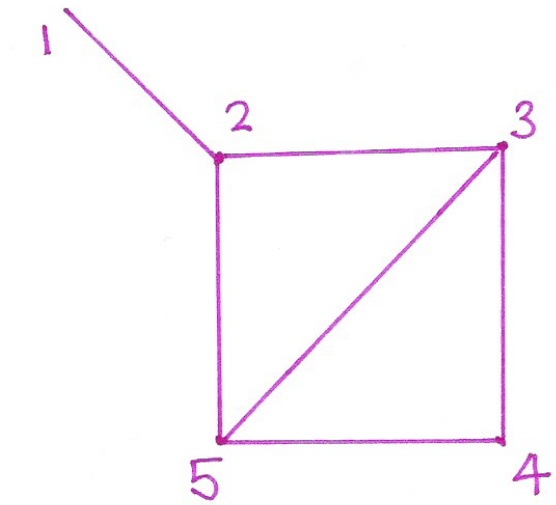
Let's look at two simple graphs, G and H.





Adjacency Matrix

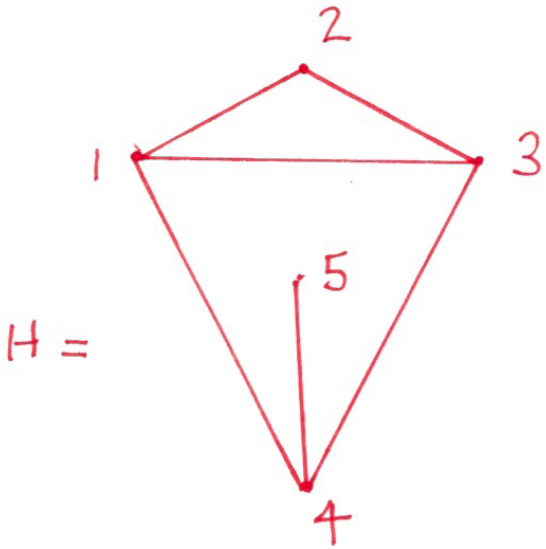
The adjacency matrix of G is:

$G =$ 

$$A_G = \begin{array}{c} \begin{matrix} ① & ② & ③ & ④ & ⑤ \end{matrix} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} ① \\ ② \\ ③ \\ ④ \\ ⑤ \end{matrix}$$

Adjacency Matrix

The adjacency matrix of H is:



$$A_H = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix}$$

Adjacency Matrix

Definition

The adjacency matrix of a graph G is the matrix with 1 at row i column j whenever G has an edge connecting vertices v_i and v_j, and 0 otherwise.

Adjacency Matrix

$$A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \quad A_H = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Adjacency Matrix and Isomorphisms

Adjacency Matrix and Isomorphisms

G and H are isomorphic if and only if there exists an orthogonal matrix P_{HG} such that

$$P_{HG} A_H P_{HG}^{-1} = A_G \, .$$

In such a case P_{HG} is called a permutation matrix going from H to G.

Orthogonal Matrix

Definition

An orthogonal matrix, M, has orthogonal unit vectors as rows and columns and has the special property that

$$M^{-1} = M^T$$

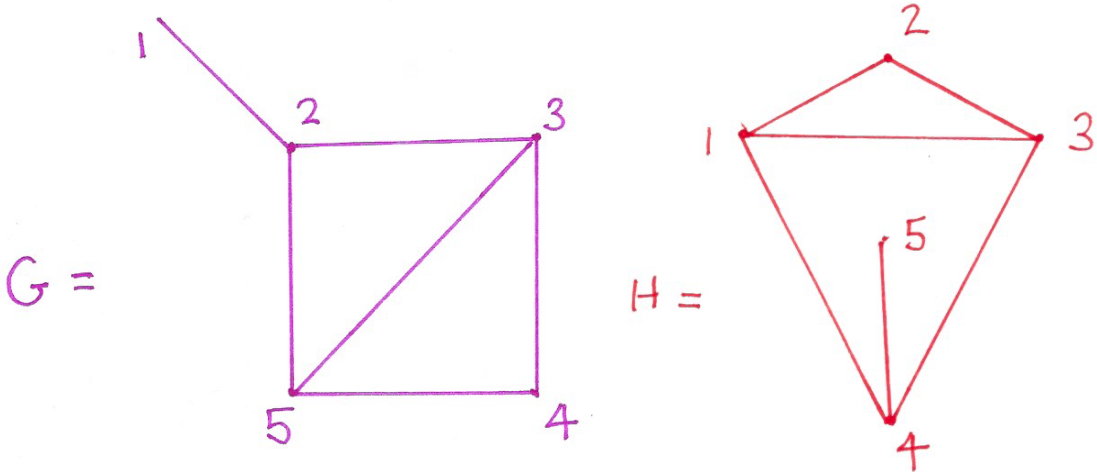
its inverse is identical to its transpose.

So to calculate the inverse of P_{HG}, we simply transpose it. The formula becomes:

$$P_{HG} A_H P_{HG}^T = A_G \, .$$

Verifying an Isomorphism

Finding P_{HG} is difficult, but verifying it is straightforward.



$$P_{HG} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Verifying an Isomorphism

Let's check whether $P_{HG} A_H P_{HG}^T = A_G$.

$$P_{HG} A_H P_{HG}^T = \overbrace{\underbrace{\begin{bmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \end{bmatrix}}_{P_{HG}} \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}}^{\text{Associative, so we can multiply these first.}} \underbrace{\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}}_{P_{HG}^T}$$

Verifying an Isomorphism

Let's check whether $P_{HG} A_H P_{HG}^T = A_G$.

$$P_{HG} A_H P_{HG}^T = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Swapping rows.

Verifying an Isomorphism

Let's check whether $P_{HG} A_H P_{HG}^T = A_G$.

$$P_{HG} A_H P_{HG}^T = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} = A_G$$

Swapping columns.

Extracting the Isomorphism

Given $P_{HG} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$,
we can *extract* the isomorphism.

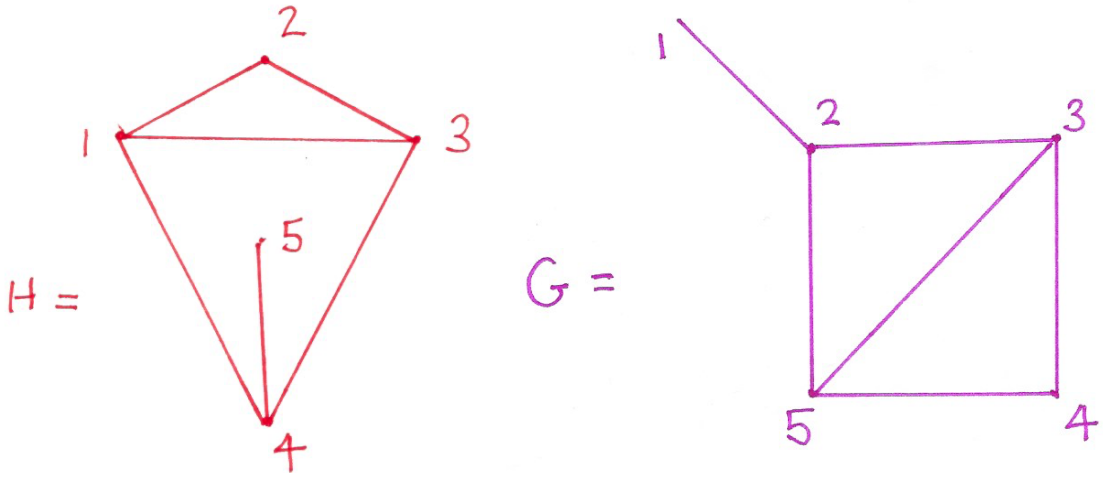
To do so, we note that any 1 in row i column j means that vertex v_i of A_H maps to vertex v_j of A_G.

Extracting the Isomorphism

Given $P_{HG} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{bmatrix}$,
we can *extract* the isomorphism.

	row		col
$v_{i,j}$	A_H	$\rightarrow$	A_G
$v_{5,1}$	5	$\rightarrow$	1
$v_{4,2}$	4	$\rightarrow$	2
$v_{1,3}$	1	$\rightarrow$	3
$v_{2,4}$	2	$\rightarrow$	4
$v_{3,5}$	3	$\rightarrow$	5

Extracting the Isomorphism



$v_{i,j}$	row A_H	$\rightarrow$	col A_G
$v_{5,1}$	5	$\rightarrow$	1
$v_{4,2}$	4	$\rightarrow$	2
$v_{1,3}$	1	$\rightarrow$	3
$v_{2,4}$	2	$\rightarrow$	4
$v_{3,5}$	3	$\rightarrow$	5

Summary

To summarize, the adjacency matrix can be used to verify a suspected isomorphism.

But to calculate the isomorphism matrix given only the graphs is difficult.

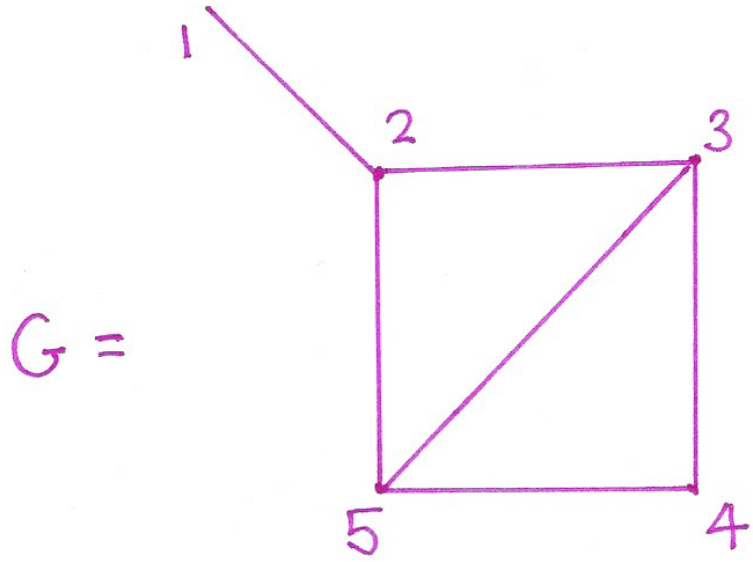
K-step Walks

K-step Walks

Another use of the adjacency matrix is to count the number of k-step walks between two given vertices.

- If A is the adjacency matrix, then $(A^k)_{i,j} = (A^k)_{j,i}$ is precisely the number of k-step walks from vertex v_i to vertex v_j,
- and consequently from vertex v_j to vertex v_i.

K-step Walks



Let's look again at .

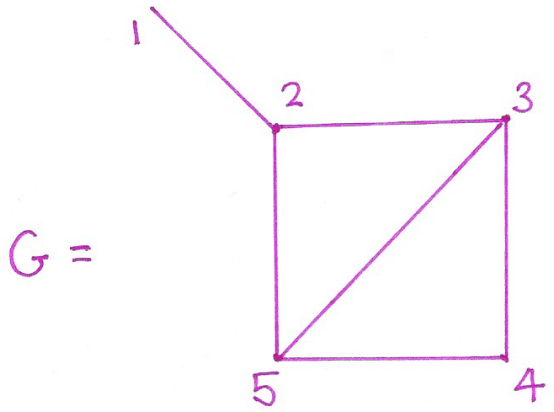
$$A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

2-step Walks

$$A_G{}^2 = A_G \cdot A_G = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 3 \end{bmatrix}$$

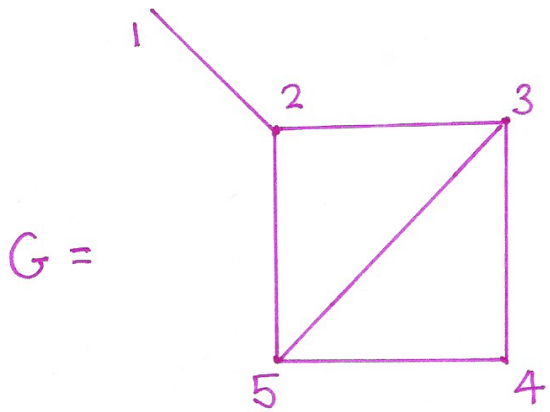
2-step Walks



$$A_G{}^2 = A_G \cdot A_G = \begin{bmatrix} 1 & 0 & 1 & 0 & 1 \\ 0 & 3 & 1 & 2 & 1 \\ 1 & 1 & 3 & 1 & 2 \\ 0 & 2 & 1 & 2 & 1 \\ 1 & 1 & 2 & 1 & 3 \end{bmatrix}$$

$$3 = A^2_{2,2} = A^2_{3,3} = A^2_{5,5}$$

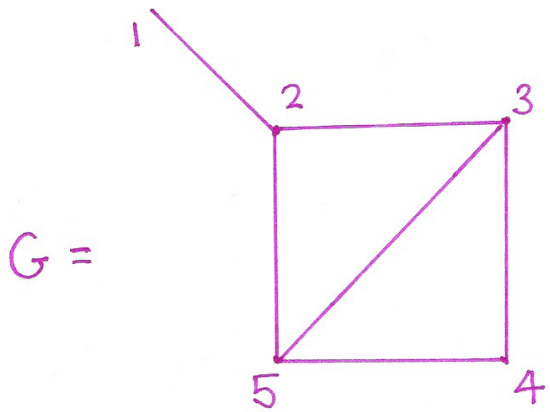
2-step Walks



There are exactly 3 2-step walks from v_2 to itself, namely

1. $v_2 \to v_1 \to v_2$,
2. $v_2 \to v_3 \to v_2$, and
3. $v_2 \to v_5 \to v_2$;

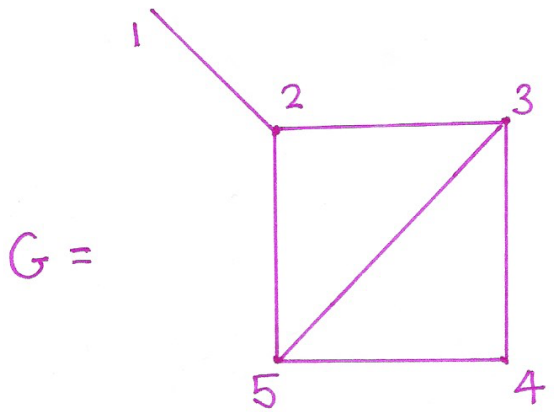
2-step Walks



There are exactly 3 2-step walks from v_3 to itself, namely

1. $v_3 \to v_2 \to v_3$,
2. $v_3 \to v_4 \to v_3$, and
3. $v_3 \to v_5 \to v_3$;

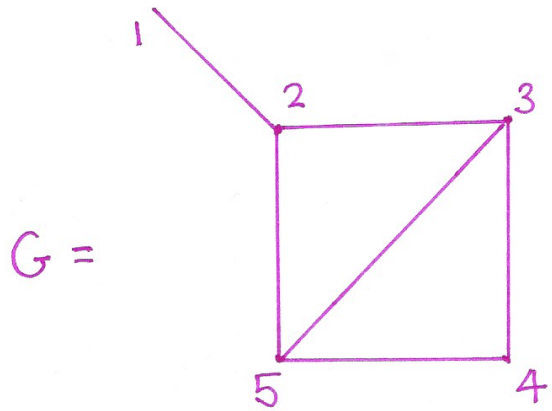
2-step Walks



There are exactly 3 2-step walks and from v_5 to itself, namely

1. $v_5 \to v_2 \to v_5$,
2. $v_5 \to v_3 \to v_5$, and
3. $v_5 \to v_4 \to v_5$.

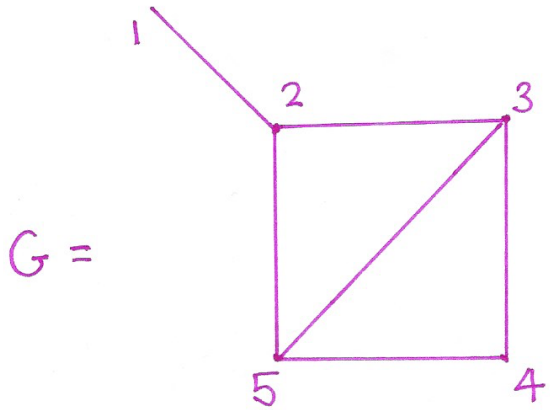
2-step Walks



Likewise, there is a 2 at row 4 column 2 (and at row 2 column 4) meaning that there are 2 2-step walks from vertex v_2 to vertex v_4, namely

1. $v_2 \to v_3 \to v_4$, and
2. $v_2 \to v_5 \to v_4$.

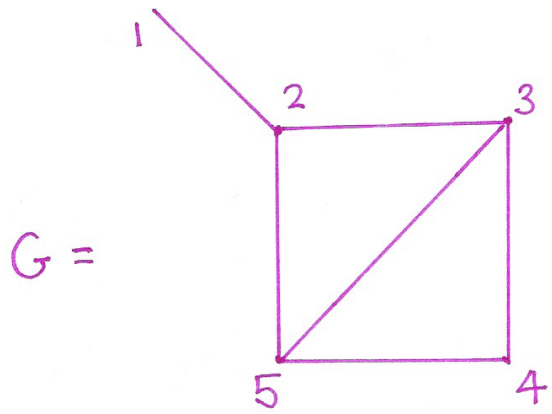
3-step Walks



We can use A_G^2 to compute A_G^3 giving the 3-step walks.

$$A_G{}^3 = A_G{}^2 \cdot A_G = \begin{bmatrix} 0 & 3 & 1 & 2 & 1 \\ 3 & 2 & 6 & 2 & 6 \\ 1 & 6 & 4 & 5 & 5 \\ 2 & 2 & 5 & 2 & 5 \\ 1 & 6 & 5 & 5 & 4 \end{bmatrix}$$

3-step Walks



The calculation of $A_G{}^3$ predicts that there are 6 3-step walks between v_2 and v_5, namely

1. $v_2 \to v_1 \to v_2 \to v_5$,
2. $v_2 \to v_3 \to v_2 \to v_5$,
3. $v_2 \to v_5 \to v_2 \to v_5$,
4. $v_2 \to v_3 \to v_4 \to v_5$,
5. $v_2 \to v_5 \to v_3 \to v_5$, and
6. $v_2 \to v_5 \to v_4 \to v_5$.

0-step Walks

Since

$$A_G^0 = I = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

we see that there is exactly one 1-step walk from any vertex to itself, because

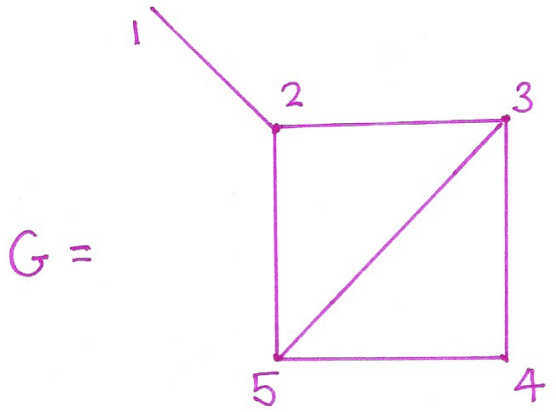
$$A_{i,i}^0 = 1\,.$$

Complexity of Adjacency Matrix

- Space requirement for the adjacency matrix is $\Theta(|V|^2)$.
- This can become $\Theta(|V| + |E|)$ if a sparse matrix is used.
- We can determine whether v_i connects to v_j in constant time.
- We can determine the list of connecting vertices of v_i in linear time.

Adjacency List

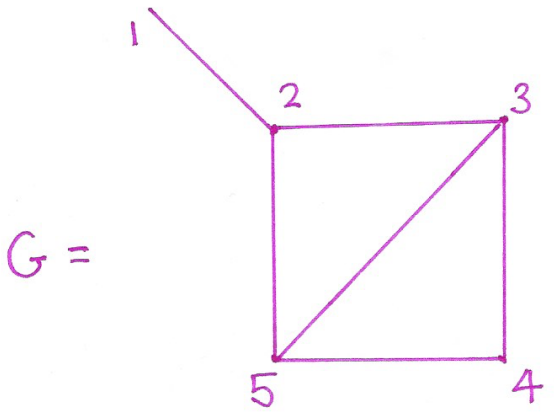
The Adjacency List



$$A[1] = \{2\}$$
$$A[2] = \{1, 3, 5\}$$
$$A[3] = \{2, 4, 5\}$$
$$A[4] = \{2, 3\}$$
$$A[5] = \{2, 3, 4\}$$

- An array indexed by vertex.
- Associated with index i is a list (or set or array) of neighbors of v_i.

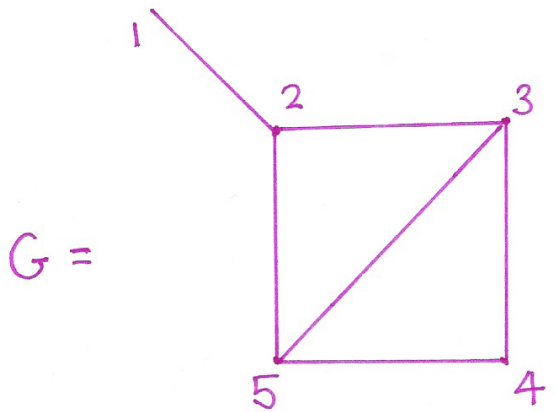
The Adjacency List



$$A[1] = \{2\}$$
$$A[2] = \{1, 3, 5\}$$
$$A[3] = \{2, 4, 5\}$$
$$A[4] = \{2, 3\}$$
$$A[5] = \{2, 3, 4\}$$

If the graph is un-directed, then $i \in A[j] \implies j \in A[i]$.

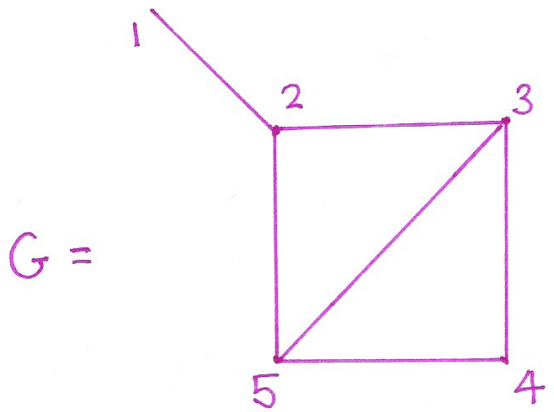
The Adjacency List



$$A[1] = \{2\}$$
$$A[2] = \{1, 3, 5\}$$
$$A[3] = \{2, 4, 5\}$$
$$A[4] = \{2, 3\}$$
$$A[5] = \{2, 3, 4\}$$

To determine whether v_i connects to v_j we must do a membership check, $i \in A[j]$ or $j \in A[i]$.

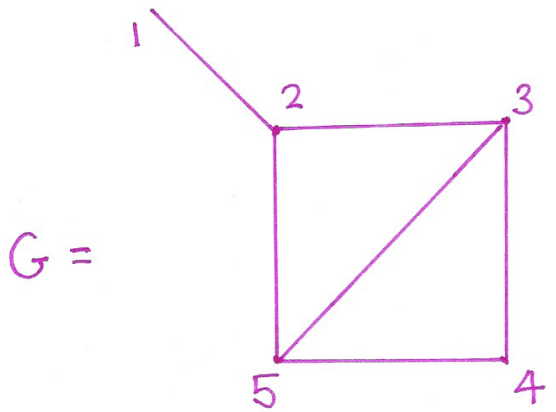
The Adjacency List



$$A[1] = \{2\}$$
$$A[2] = \{1, 3, 5\}$$
$$A[3] = \{2, 4, 5\}$$
$$A[4] = \{2, 3\}$$
$$A[5] = \{2, 3, 4\}$$

If the vertices are integers from 0 to $n-1$ or from 1 to n, then A can be a simple array. If not you may have to apply some mapping strategy.

The Adjacency List



$$A[1] = [2]$$
$$A[2] = [1, 3, 5]$$
$$A[3] = [2, 4, 5]$$
$$A[4] = [2, 3]$$
$$A[5] = [2, 3, 4]$$

If you need to support parallel edges, use array or list rather than set.

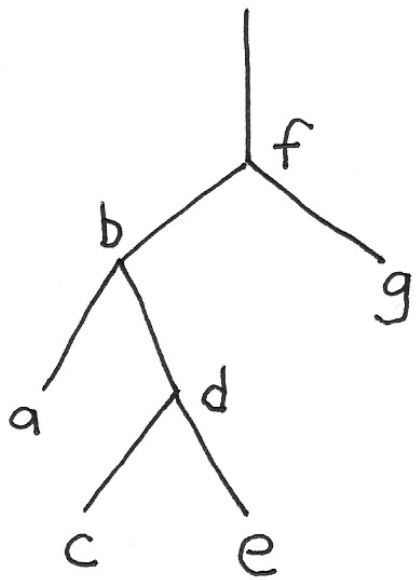
Complexity of Adjacency List

- Space requirement for the adjacency matrix is $\Theta(|V| + |E|)$.
- Efficient when $|E| << |V|$.
- We can determine whether v_i connects to v_j in linear or log time, $v_j \in A[v_i]$
- We can determine the neighbors of v_i in constant time: $A[v_i]$

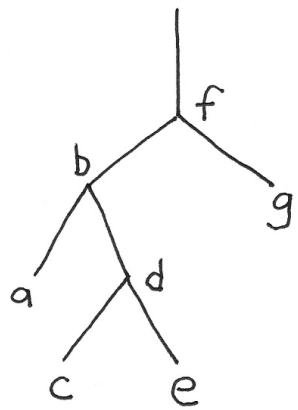
Depth First Search

Depth First Search, DFS

To understand DFS, first think of a walk over a binary tree without loops and without diamonds.



Binary Search Algorithm



Algorithm 1: DFS-binary-tree(v,f)

Input : v vertex of a binary tree
Input : f function to apply to each vertex

```
1 if v.left then
2 |   DFS-binary-tree(v.left,f)
3 f(v)
4 if v.right then
5 |   DFS-binary-tree(v.right,f)
```

DFS

We'd like to extend this concept to an arbitrary graph, which:

- may not have a designated root,
- may not be connected,
- maybe has loops, or
- maybe has multiple edges (> 2) per vertex.

We first look at `DFS-visit` which is almost what we want but has some deficiencies which we will note.

DFS-visit

Algorithm 2: DFS-visit(A,v,f,parent)

Input : A the adjacency list, mapping each vertex to list of adjacent vertices

Input : v a starting vertex, $v \in V$

Input : f a function to be applied to each vertex

Input : *parent* maps each vertex to a *parent* vertex or special value.

```
1 foreach neighbor ∈ A[v] do
2 |   if not parent[neighbor] then
3 |   |   parent[neighbor] ← v
4 |   |   DFS-visit(A,neighbor,f,parent)
5 f(v)
```

What is the special value?

Optimization: the `parent[]` array serves both as 1) the back-pointer to the vertex we arrived from, and 2) as a Boolean to indicate whether the vertex has already been visited, thus avoiding infinite loops.

But we must set `parent[s]` to a special value, not a vertex, before calling `DFS-visit(...s...)`.

Be careful, in Python:

- If `parent[s] == 0` remember, `0 == False`.
- If `parent[s] == None` remember, `None == False`.

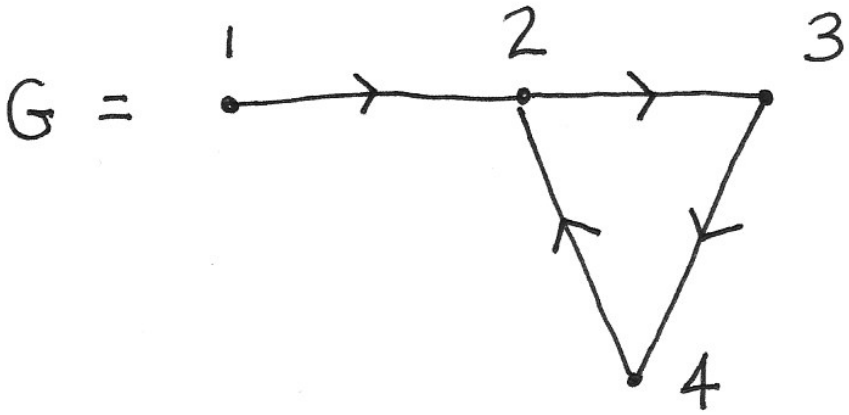
Advise about `DFS-visit`

Implementing `if not parent[neighbor] then` in Python.

- If `parent` is a `list`, use `None` as initializing value, use `'None'` as special value.
  `if parent[neighbor] is not None:`
- If `parent` is a `dict`, use `None` as special value
  `if neighbor not in parent:`

Caveats of DFS-visit

If the graph is not connected, DFS-visit will fail to visit some vertices. Also if the graph is connected but directed, there is a chance of missing some vertices.



In this graph DFS-visit(...1...) would visit every vertex, but DFS-visit(...3...) would never visit vertex 1.

We need an outer loop.

DFS Algorithm

Algorithm 3: DFS(A,f)

Input : A the adjacency list, mapping each vertex to list of adjacent vertices

Input : f a function to be applied to each vertex

```
1 parent ← ∅
2 foreach s ∈ A do
3 |   if not parent[s] then
  |   |   /* if s has not yet been visited        */
4 |   |   parent[s] ← None
5 |   |   DFS-visit(s)
```

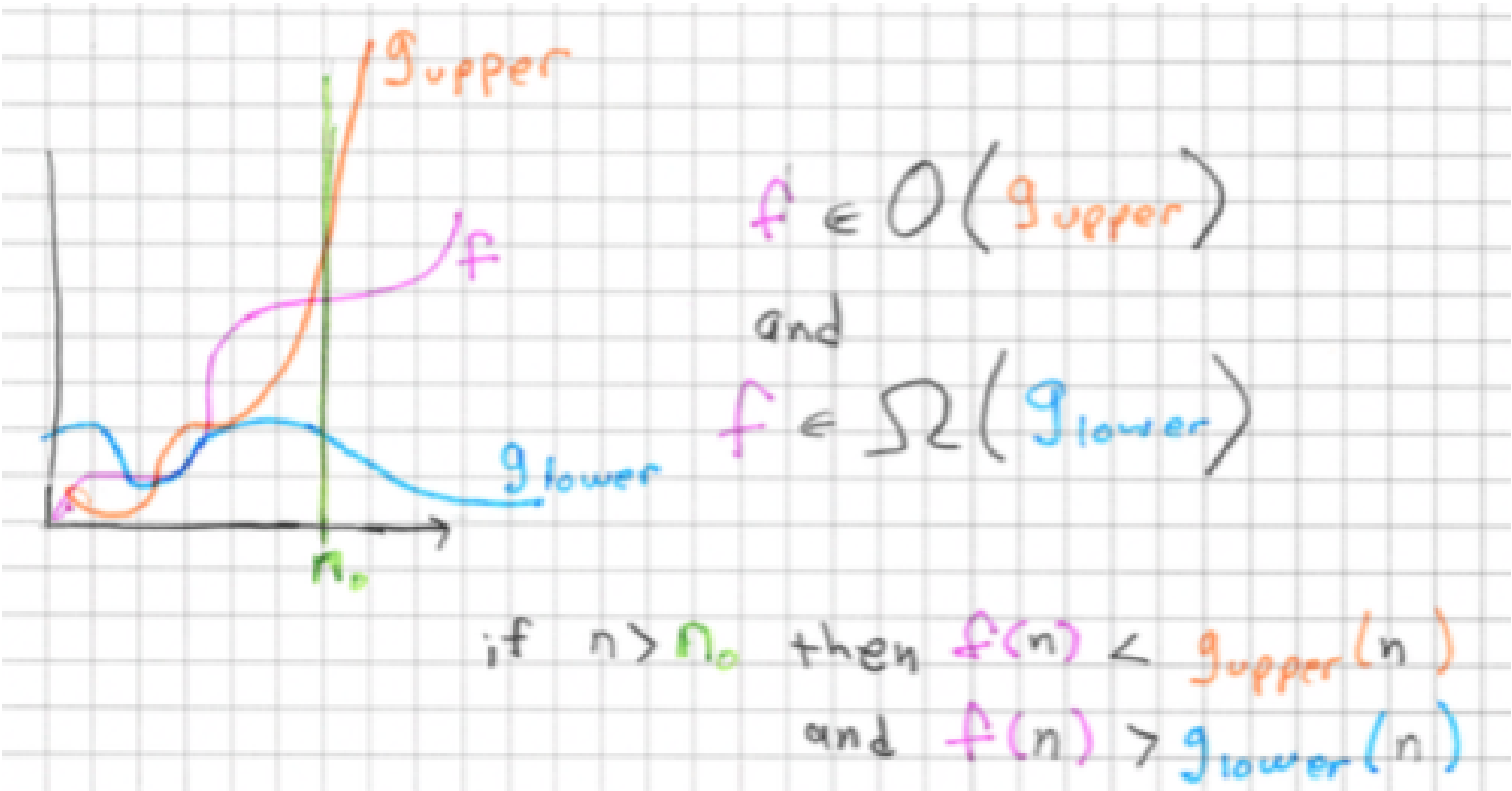
Complexity of DFS

To analyze the complexity of `DFS` we know that `DFS-visit` is called exactly once per vertex, and for vertex v_i we perform $O(1) + O(1) \times degree(v_i)$ amount of time assuming $f(i)$ is constant time.

$$
\begin{aligned}
Complexity &= \sum_{i=v_1}^{v_n} O(1) + O(1) \times degree(v_i) \\
&= |V| \times O(1) + O(1) \times \sum_{i=v_1}^{v_n} degree(v_i) \\
&= |V| \times O(1) + O(1) \times 2 \times |E| \\
&= O(|V| + |E|)
\end{aligned}
$$

Big O Notation

Recall that if $f \in O(g)$, then $f(n)$ is eventually bounded above by $cg(n)$ for $n >> 0$, and if $f \in \Omega(g)$, then $f(n)$ is eventually bounded below by $cg(n)$ for $n >> 0$.



Topological Sort

Topological Sort

Topological sort is a way to order a set which is constrained by pairwise orderings.

- In terms of a graph, topological sort is a way of selecting an order of the vertices of an directed graph such that if (a, b) is an edge of the directed graph, then a precedes b in the sequence of ordered vertices.
- The problem is normally stated as a set of constraints, and you are asked to produce an ordering which does not violate any of the constraints.

Example of Topological Sort

Consider the following Common Lisp class definitions.

```
(defclass food () ...)
(defclass fruit (food) ...)
(defclass spice (food) ...)
(defclass apple (fruit) ...)
(defclass cinnamon (spice) ...)
(defclass ingredient (apple cinnamon) ...)
```

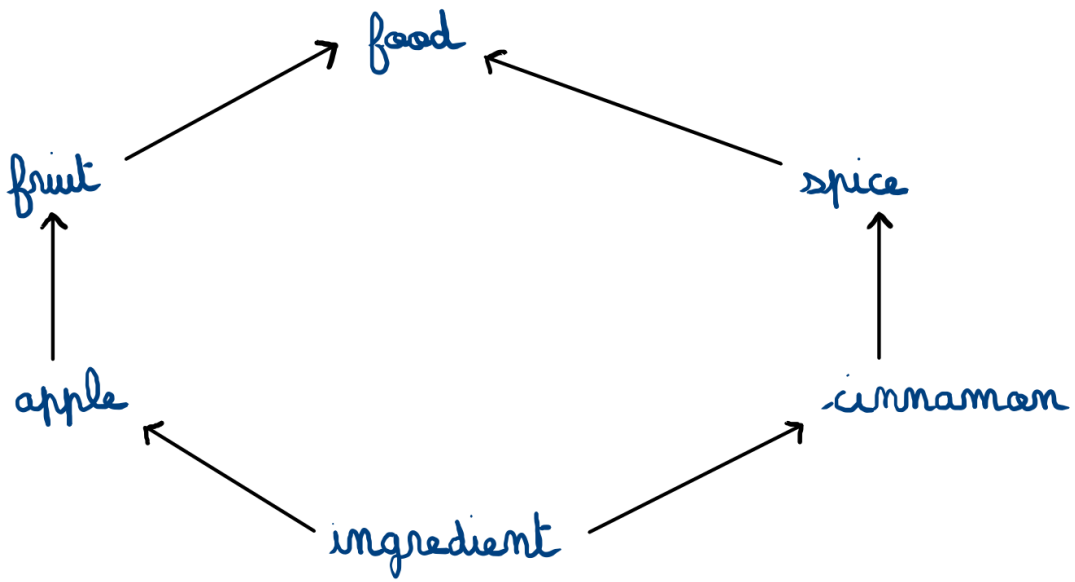
Example of Topological Sort

The following subclass relations hold:

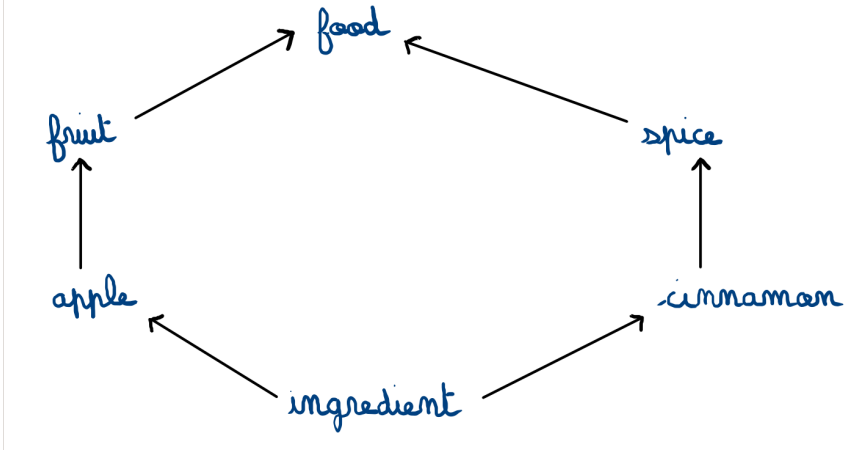
$$\begin{aligned} \textit{ingredient} &\subset \textit{apple} \\ \textit{ingredient} &\subset \textit{cinnamon} \\ \textit{apple} &\subset \textit{fruit} \\ \textit{cinnamon} &\subset \textit{spice} \\ \textit{fruit} &\subset \textit{food} \\ \textit{spice} &\subset \textit{food} \end{aligned}$$

Example of Topological Sort

Leading to the class graph:



Example of Topological Sort



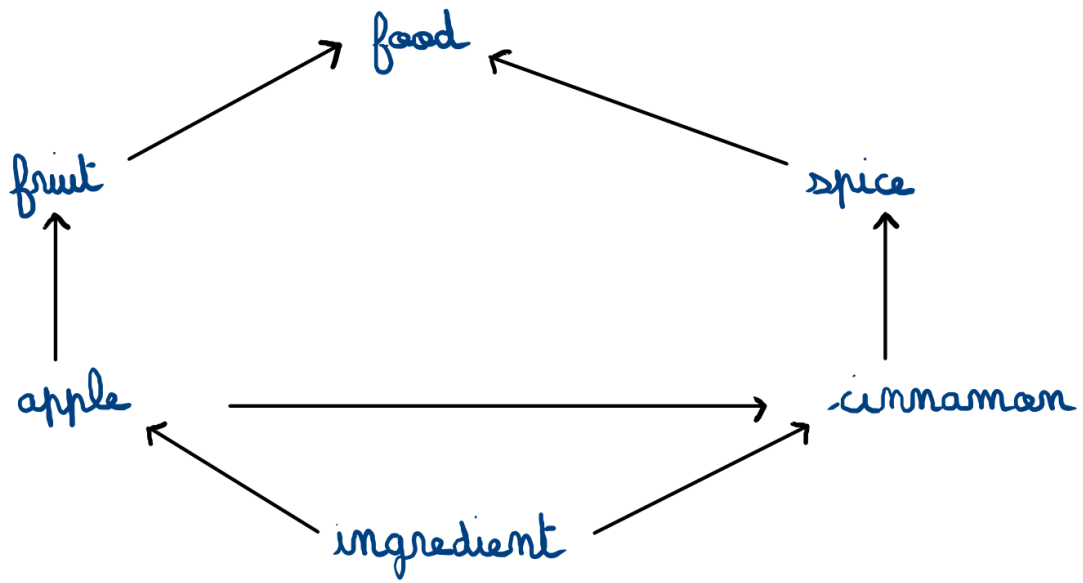
We would like to linearize these classes, respecting all the subclss relations. There are six such orders.

$$ingredient \to apple \to cinnamon \to fruit \to spice \to food$$
$$ingredient \to apple \to cinnamon \to spice \to fruit \to food$$
$$ingredient \to apple \to fruit \to cinnamon \to space \to food$$
$$ingredient \to cinnamon \to apple \to fruit \to spice \to food$$
$$ingredient \to cinnamon \to apple \to spice \to fruit \to food$$
$$ingredient \to cinnamon \to spice \to apple \to fruit \to food$$

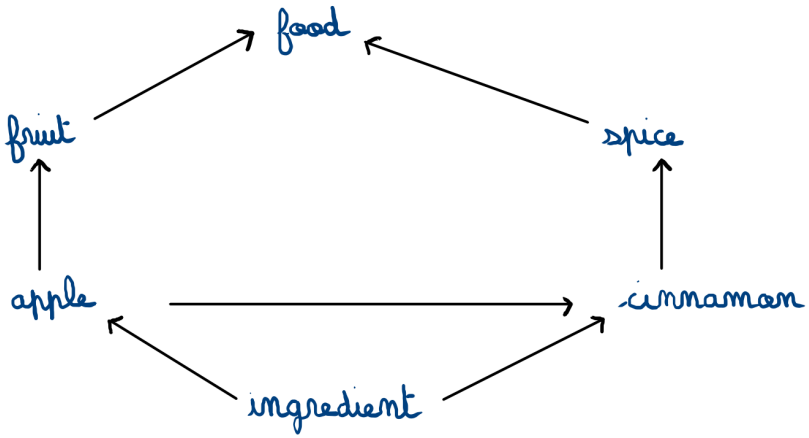
Example of Topological Sort

```
(defclass ingredient (apple cinnamon) ...)
```

Since the class definition of `ingredient` above mentions `apple` before `cinnamon`, CL requires that `apple` appear before `cinnamon` in the CPL. This adds an additional constraint in the graph.



Example of Topological Sort



However, even with this additional constraint, there are still 3 possible orderings.

$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow fruit \rightarrow spice \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow spice \rightarrow fruit \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$

Example of Topological Sort

$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow fruit \rightarrow spice \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow spice \rightarrow fruit \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$

Next we must to choose either `apple->cinnamon` or `apple->fruit`.

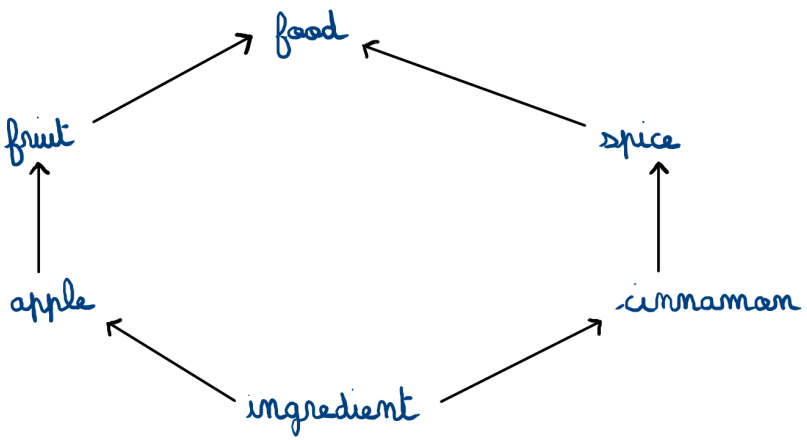
We need a tie-breaker.

When selecting between `cinnamon` and `fruit` CL requires that we ask which of these two has a direct subclass farthest to the right in the list computed so far.

Example of Topological Sort

$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow fruit \rightarrow spice \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow spice \rightarrow fruit \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$

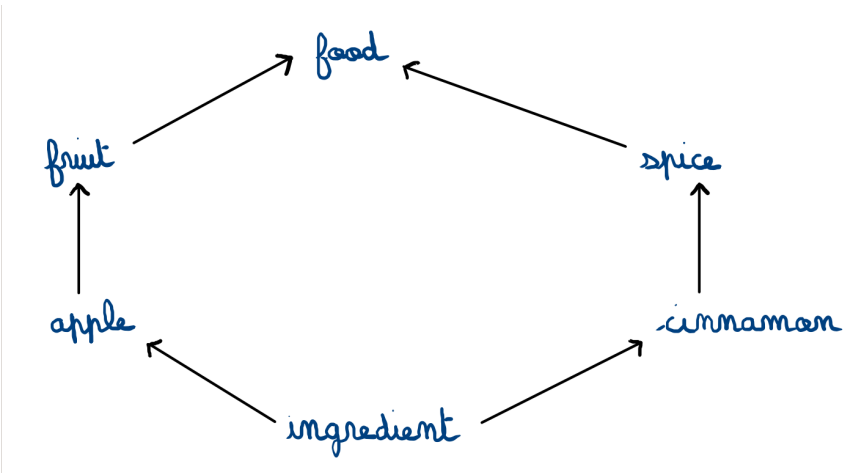
The list computed so far is $ingredient \rightarrow apple$.



Which of `cinnamon` or `fruit` has a subclass farther to the right in the list computed so far?

Example of Topological Sort

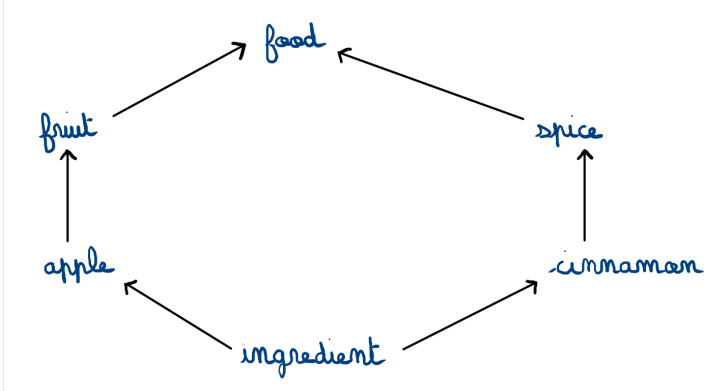
$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow fruit \rightarrow spice \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow cinnamon \rightarrow spice \rightarrow fruit \rightarrow food$$
$$ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$



In this case $apple \subset fruit$ and $ingredient \subset cinnamon$.

Example of Topological Sort

$$\textit{ingredient} \rightarrow \textit{apple} \rightarrow \textit{cinnamon} \rightarrow \textit{fruit} \rightarrow \textit{spice} \rightarrow \textit{food}$$
$$\textit{ingredient} \rightarrow \textit{apple} \rightarrow \textit{cinnamon} \rightarrow \textit{spice} \rightarrow \textit{fruit} \rightarrow \textit{food}$$
$$\textit{ingredient} \rightarrow \textit{apple} \rightarrow \textit{fruit} \rightarrow \textit{cinnamon} \rightarrow \textit{spice} \rightarrow \textit{food}$$



As `apple` is farther right than `ingredient` in $\textit{ingredient} \rightarrow \textit{apple}$, we select $\textit{apple} \rightarrow \textit{fruit}$, thus leaving only one possibility.

$$\textit{ingredient} \rightarrow \textit{apple} \rightarrow \textit{fruit} \rightarrow \textit{cinnamon} \rightarrow \textit{spice} \rightarrow \textit{food}$$

Kahn Algorithm

Kahn Algorithm (1962)

Algorithm 4: Topological Sort (Kahn)

```
1  L → ∅ list that will contain the sorted elements;
2  S → Set of all nodes with no incoming edge ;
3  while S ≠ ∅ do
4  |   remove a node n from S ;
5  |   concatenate n to tail of L ;
6  |   foreach node m with an edge e from n to m do
7  |   |   remove edge e from the graph;
8  |   |   if m has no other incoming edges then
9  |   |   |   insert m into S ;
10 if graph has edges then
11 |   return error (graph has at least one cycle) ;
12 else
13 |   return L (a topologically sorted order)
```

Kahn Algorithm

The original Kahn algorithm did not explicitly mention a user defined tie-breaker function.

Let's look at the same set of constraints as before.

List the constraints:

$$
\begin{aligned}
\mathit{ingredient} &\rightarrow \mathit{apple} \\
\mathit{ingredient} &\rightarrow \mathit{cinnamon} \\
\mathit{apple} &\rightarrow \mathit{cinnamon} \\
\mathit{apple} &\rightarrow \mathit{fruit} \\
\mathit{cinnamon} &\rightarrow \mathit{spice} \\
\mathit{fruit} &\rightarrow \mathit{food} \\
\mathit{spice} &\rightarrow \mathit{food}
\end{aligned}
$$

Kahn Algorithm

Start with the class being defined. This class is unconstrained, nothing is inheriting from it by definition, because we are in the process of defining it. I.e., initialize

$$CPL = ingredient$$

and mark all constraints $ingredient \rightarrow x$ as satisfied.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
$apple \rightarrow cinnamon$	
$apple \rightarrow fruit$	
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Kahn Algorithm

~~*ingredient* → *apple*~~	SATISFIED
~~*ingredient* → *cinnamon*~~	SATISFIED
apple → *cinnamon*	
apple → *fruit*	
cinnamon → *spice*	
fruit → *food*	
spice → *food*	

Now find the set of unconstrained class(es). The only such class is `apple`; add it to the end.

$$CPL = ingredient \rightarrow apple$$

Kahn Algorithm

$$CPL = ingredient \rightarrow apple$$

And remove all constraints on `apple`.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Kahn Algorithm

$$CPL = ingredient \rightarrow apple$$

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
$cinnamon \rightarrow spice$	
$fruit \rightarrow food$	
$spice \rightarrow food$	

Now find the set of unconstrained classes.
They are $\{cinnamon, fruit\}$.

Kahn Algorithm

$$CPL = ingredient \rightarrow apple$$

Deciding between `cinnamon` and `fruit`.
Kahn's original algorithm didn't consider tie breaking.
So by that original algorithm, you could take either `cinnamon` or `fruit` as you prefer.
We use the same tie-breaker as before and choose `fruit`.

Kahn Algorithm

Kahn's original algorithm didn't consider tie breaking.
So by that original algorithm, you could take either `cinnamon` or `fruit` as you prefer.
We use the same tie-breaker as before and choose `fruit`.

$$CPL = ingredient \rightarrow apple \rightarrow fruit$$

And remove all constraints on `fruit`.

Kahn Algorithm

And remove all constraints on `fruit`.

~~*ingredient* → *apple*~~	SATISFIED
~~*ingredient* → *cinnamon*~~	SATISFIED
~~*apple* → *cinnamon*~~	SATISFIED
~~*apple* → *fruit*~~	SATISFIED
cinnamon → *spice*	
~~*fruit* → *food*~~	SATISFIED
spice → *food*	

Kahn Algorithm

~~*ingredient* → *apple*~~	SATISFIED
~~*ingredient* → *cinnamon*~~	SATISFIED
~~*apple* → *cinnamon*~~	SATISFIED
~~*apple* → *fruit*~~	SATISFIED
cinnamon → *spice*	
~~*fruit* → *food*~~	SATISFIED
spice → *food*	

Kahn Algorithm

~~*ingredient* → *apple*~~	SATISFIED
~~*ingredient* → *cinnamon*~~	SATISFIED
~~*apple* → *cinnamon*~~	SATISFIED
~~*apple* → *fruit*~~	SATISFIED
cinnamon → *spice*	
~~*fruit* → *food*~~	SATISFIED
spice → *food*	

Now find the set of unconstrained class(es). The only such class is `cinnamon`; add it to the end.

Kahn Algorithm

Now find the set of unconstrained class(es). The only such class is `cinnamon`; add it to the end.

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon$$

Remove constraints on `cinnamon`.

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
~~$cinnamon \rightarrow spice$~~	SATISFIED
~~$fruit \rightarrow food$~~	SATISFIED
$spice \rightarrow food$	

Kahn Algorithm

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon$$

~~$ingredient \rightarrow apple$~~	SATISFIED
~~$ingredient \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow cinnamon$~~	SATISFIED
~~$apple \rightarrow fruit$~~	SATISFIED
~~$cinnamon \rightarrow spice$~~	SATISFIED
~~$fruit \rightarrow food$~~	SATISFIED
$spice \rightarrow food$	

Now the only unconstrained class is `spice`, so add it to the end.

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice$$

Kahn Algorithm

And remove all constraints on `spice`.

~~*ingredient* → *apple*~~	SATISFIED
~~*ingredient* → *cinnamon*~~	SATISFIED
~~*apple* → *cinnamon*~~	SATISFIED
~~*apple* → *fruit*~~	SATISFIED
~~*cinnamon* → *spice*~~	SATISFIED
~~*fruit* → *food*~~	SATISFIED
~~*spice* → *food*~~	SATISFIED

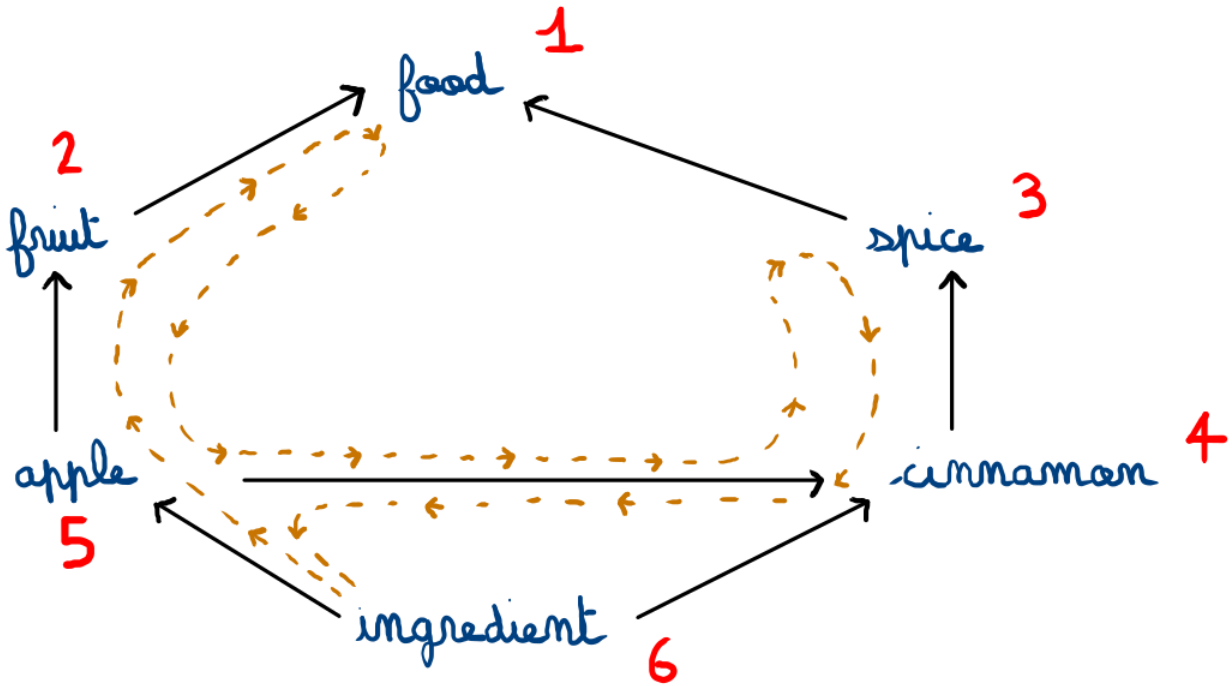
The only remaining class is `food`, so add it to the end.

$$CPL = ingredient \rightarrow apple \rightarrow fruit \rightarrow cinnamon \rightarrow spice \rightarrow food$$

Tarjan Algorithm

Tarjan Algorithm (1976)

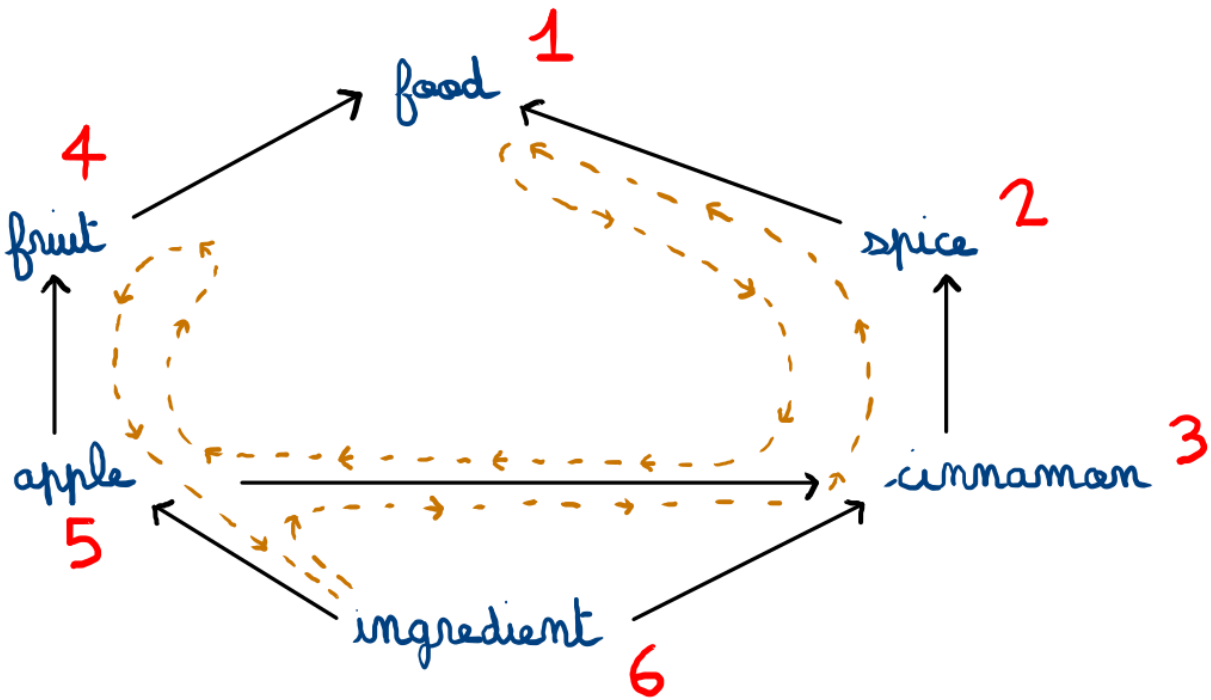
Perform DFS. Annotate vertices after visiting all neighbors. Depending on the semantics of the edge directions, either the resulting order or its reverse is the topological order.



$$\overbrace{\textit{ingredient}}^{6} \rightarrow \overbrace{\textit{apple}}^{5} \rightarrow \overbrace{\textit{fruit}}^{4} \rightarrow \overbrace{\textit{cinnamon}}^{3} \rightarrow \overbrace{\textit{spice}}^{2} \rightarrow \overbrace{\textit{food}}^{1}$$

Tarjan Algorithm

Neighbors are unordered;



multiple topological orders are possible.

$$\overbrace{ingredient}^{6} \rightarrow \overbrace{apple}^{5} \rightarrow \overbrace{cinnamon}^{4} \rightarrow \overbrace{spice}^{3} \rightarrow \overbrace{fruit}^{2} \rightarrow \overbrace{food}^{1}$$

Kahn vs Tarjan

Let $G = (V, E)$, with $V = \{a, b, c, d, e, f\}$.
There are many ways to topological sort the vertices of this graph subject to the constraints $\{(a, b), (a, c), (b, c), (b, d), (d, f), (c, e), (e, f)\}$. I.e., a precedes b, a precedes c, b precedes c etc.

- a b c d e f
- a b c e d f
- a b d c e f

Kahn vs Tarjan

Find the order of the vertices which comes first in lexographical order. With Kahn algorithm we can use a tie-breaker rule that uses `alpha-less-than`. This gives

- a b c d e f

This order is impossible with Tarjan, because in every DFS of the graph, c and e are adjacent.