Lecture 6: Graph & Search (BFS,DFS)

김수경

이화여자대학교 인공지능융합전공 소속

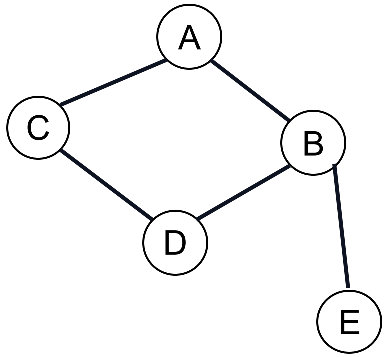
저작권 안내

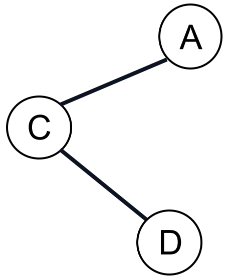
01

Graphs

Definitions

- **Graph** $G = (V, E)$, where
  - V is a set of vertices (nodes), and
  - E is the set of edges.
- **Subgraph**: a subset of a graph's vertices and edges.
- Two vertices are **adjacent** if they are joined by an edge.



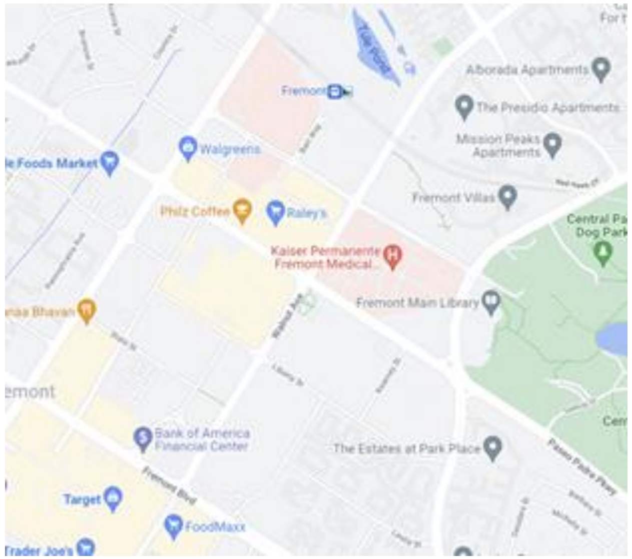


upstage Education

Graph Examples

수도권 광역전철 노선도
서울교통공사

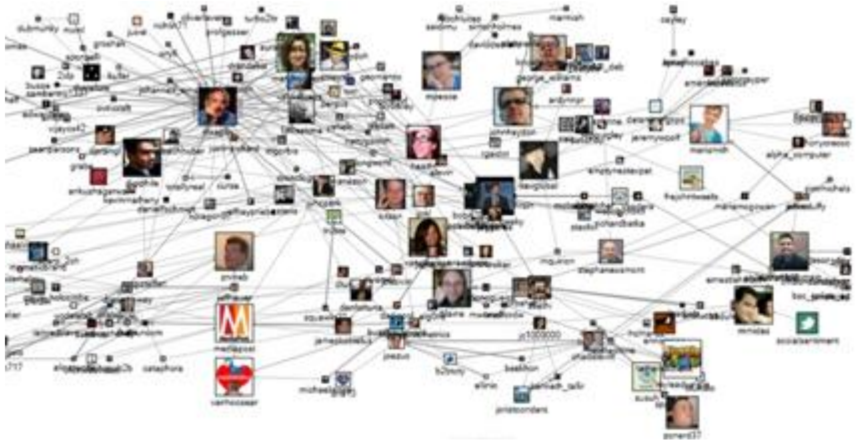
[1]



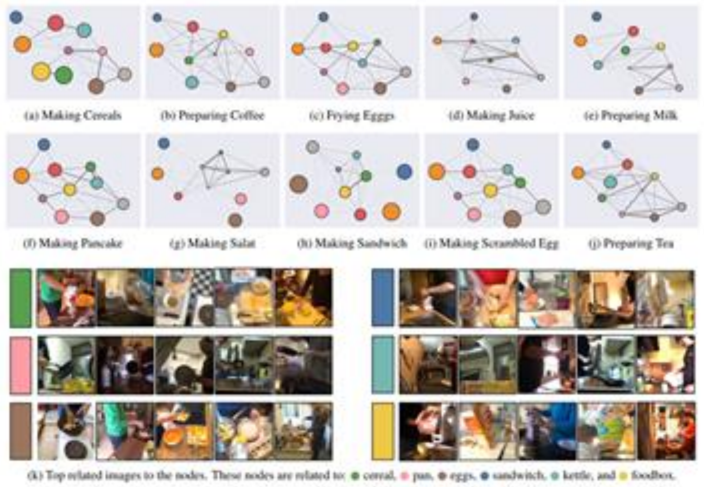
[2]

[1] 서울교통공사
[2] Google Map

Graph Examples



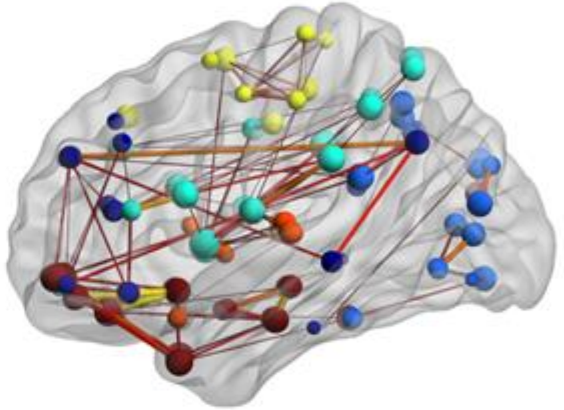
[1]

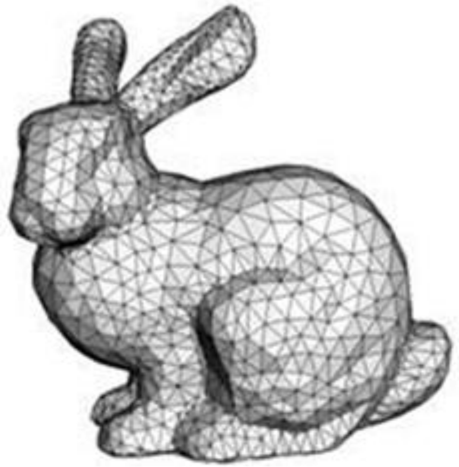


[2]

[1] https://www.smrfoundation.org/2010/04/28/mapping-the-twitter-network-of-www2010-with-nodexl/
[2] https://noureldien.com/research/videograph/index.html

Graph Examples

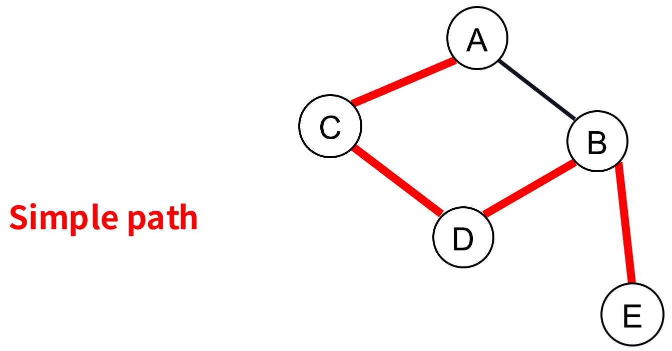

[1]

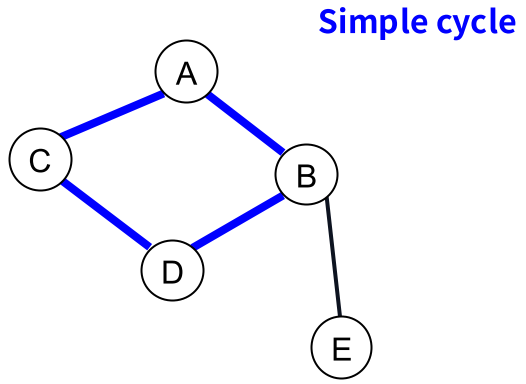

[2]

[1 https://www.nitrc.org/project/list_screenshots.php?group_id=504&screenshot_id=381
[2] https://www.researchgate.net/figure/Distinction-between-a-Graphical-Model-and-a-CAD-Model_fig1_361770871

Definitions

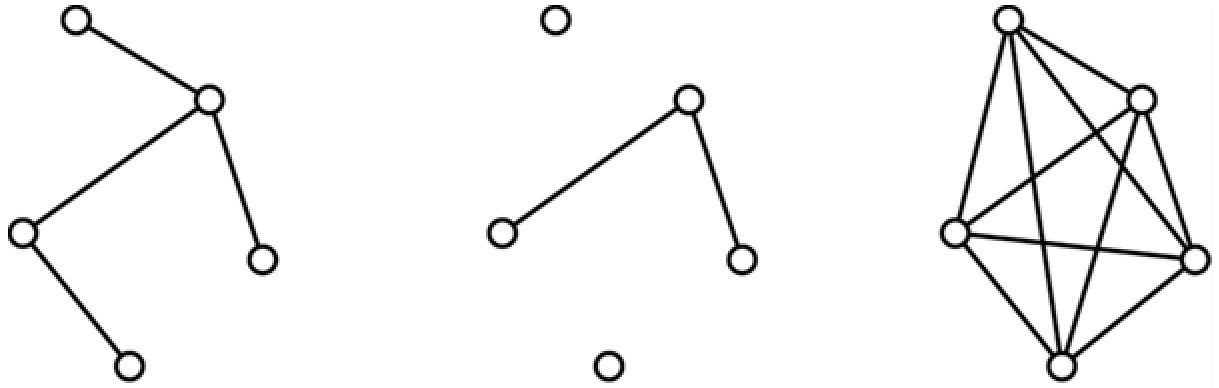
- **Path**: A sequence of connected edges
- **Cycle**: A path whose starting vertex and ending vertex are the same
- **Simple path**: A path that contains no cycle
- **Simple cycle**: A cycle that contains no cycle in it





Definitions

- **Connected graph**: Each pair of distinct vertices has a path between them
- **Complete graph**: Each pair of distinct vertices has an edge between them

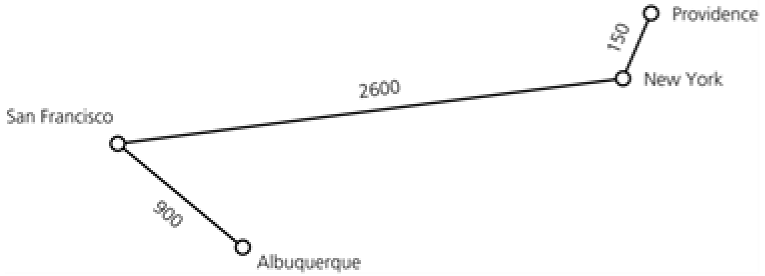

[1]

Connected graph	Disconnected graph	Complete graph

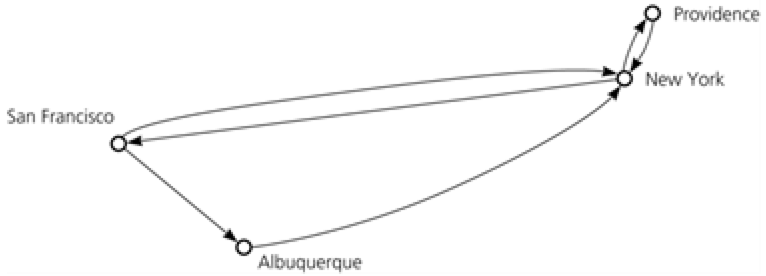
[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Definitions

- **Weighted** graph vs. **unweighted** graph: whether edges have weights
- **Directed** graph vs. **undirected** graph: whether edges have direction



Weighted graph



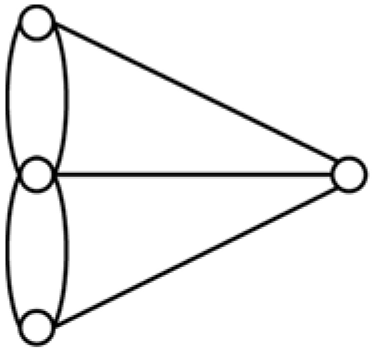
Directed graph

[1]

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

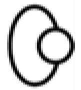
Definitions

- These are not allowed in (regular) graphs:
  - **Multigraph**: more than one edges allowed for a same pair of nodes.
  - **Self edge**: an edge starts from and arrives at the same node.



[1]

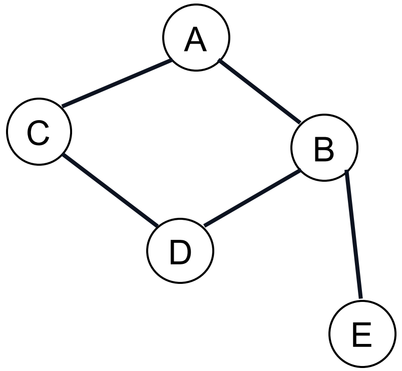
Multigraph

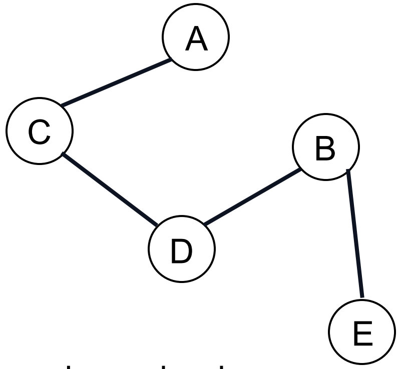


Self edge

Tree *vs.* Graph?

- **Tree** is a special case of graph, where
  - all nodes are **connected**, and
  - there is **no cycle**.





Any node can be the root.

Exercise

- Tree *vs.* Graph?

  Graph

- Connected *vs.* disconnected?

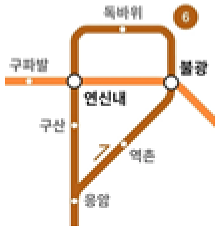
  Connected

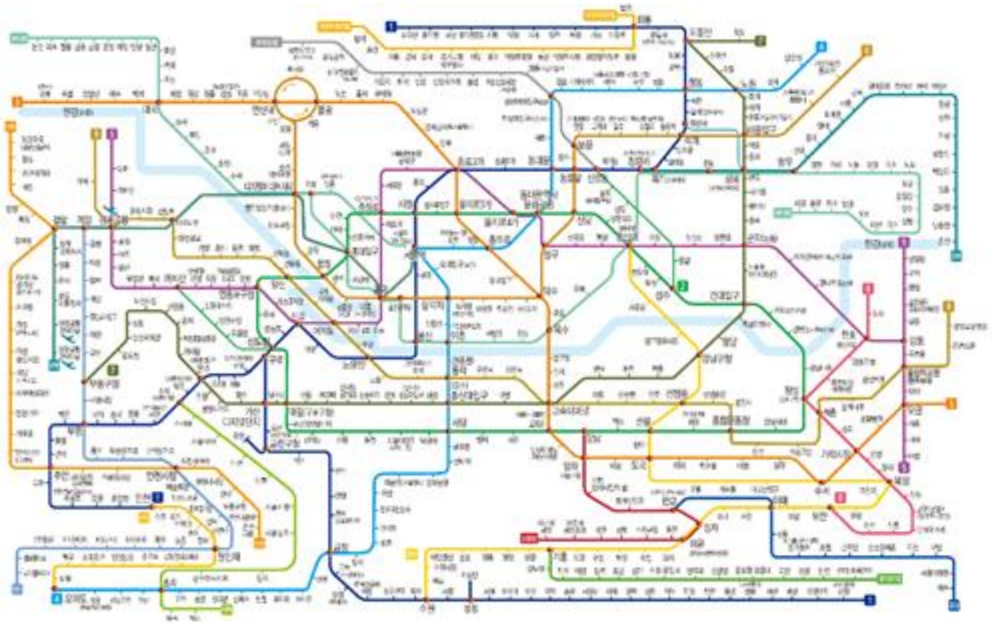
- Cyclic *vs.* acyclic?

  Cyclic

- Directed *vs.* undirected?

  Directed



[1]


[1] 서울교통공사

02

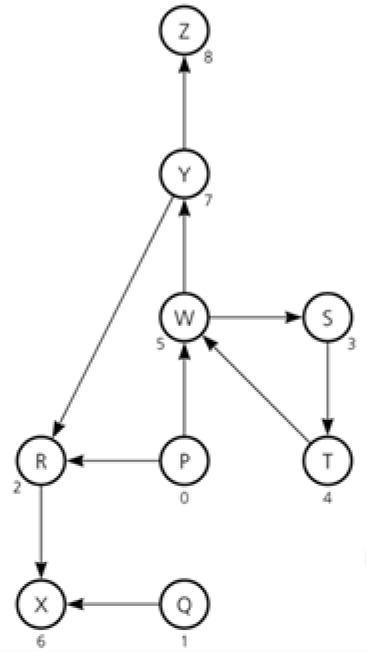
Graph Representation

Adjacency Matrix

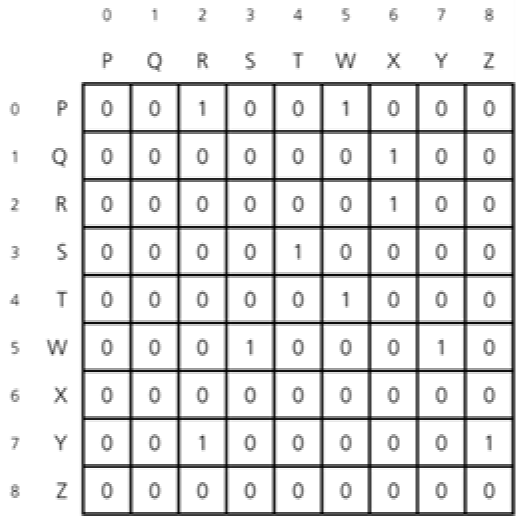
- Adjacency matrix
  - A graph is represented by an ***N* × *N* matrix (2D array)**, where `matrix[i][j]` is 1 if there is an edge between vertex `i` and vertex `j`, and 0 otherwise.
  - In case of a *directed* graph, `matrix[i][j]` is 1 if there is an edge from vertex `i` to vertex `j`.
  - In case of a *weighted* graph, `matrix[i][j]` has the weight value, instead of 1.
- This is analogous to the **array-based** implementation of other data structures.
  - We use fixed size of memory (corresponding to $N \times N$), regardless of how many edges we have.
  - (+) random access is done at O(1).
  - (-) inefficient use of memory if the graph is sparsely connected.

Adjacency Matrix: Example

Directed graph



[1]

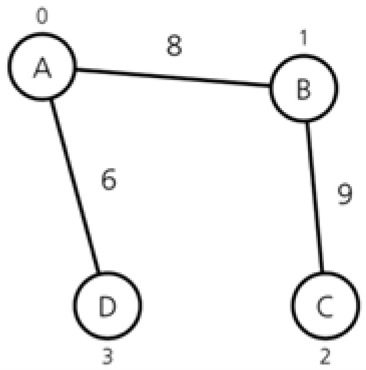


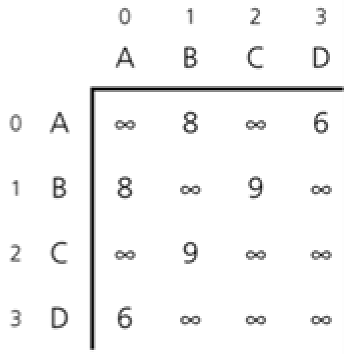
		0	1	2	3	4	5	6	7	8
		P	Q	R	S	T	W	X	Y	Z
0	P	0	0	1	0	0	1	0	0	0
1	Q	0	0	0	0	0	0	1	0	0
2	R	0	0	0	0	0	0	1	0	0
3	S	0	0	0	0	1	0	0	0	0
4	T	0	0	0	0	0	1	0	0	0
5	W	0	0	0	1	0	0	0	1	0
6	X	0	0	0	0	0	0	0	0	0
7	Y	0	0	1	0	0	0	0	0	1
8	Z	0	0	0	0	0	0	0	0	0

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Adjacency Matrix: Example

Weighted undirected graph





		0	1	2	3
		A	B	C	D
0	A	∞	8	∞	6
1	B	8	∞	9	∞
2	C	∞	9	∞	∞
3	D	6	∞	∞	∞

[1]

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Adjacency List

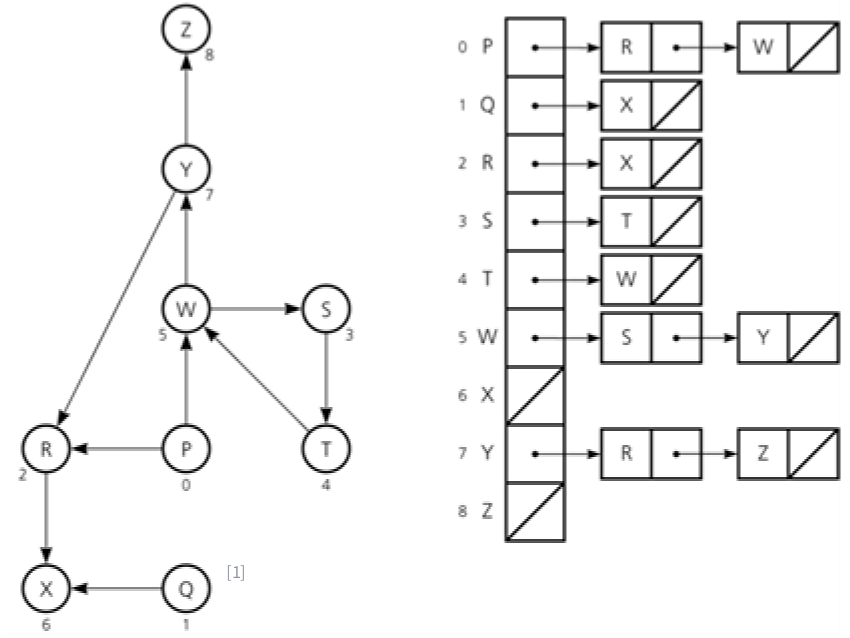
- Adjacency list
  - A graph is represented by ***N* linked lists** where `list[i]` is the list of vertices that is adjacent to vertex `i`.
  - In case of weighted graph, the list also contains the weight values.

- This is analogous to the **reference-based** implementation of other data structures.
  - For each node, we use variable size of memory, depending on the number of edges connected from/to it.
  - (+) efficient use of memory if the graph is sparsely connected.
  - (-) need linear search for an edge. (Not a big problem if the graph is sparse enough.)

upstage Education

Adjacency List: Example

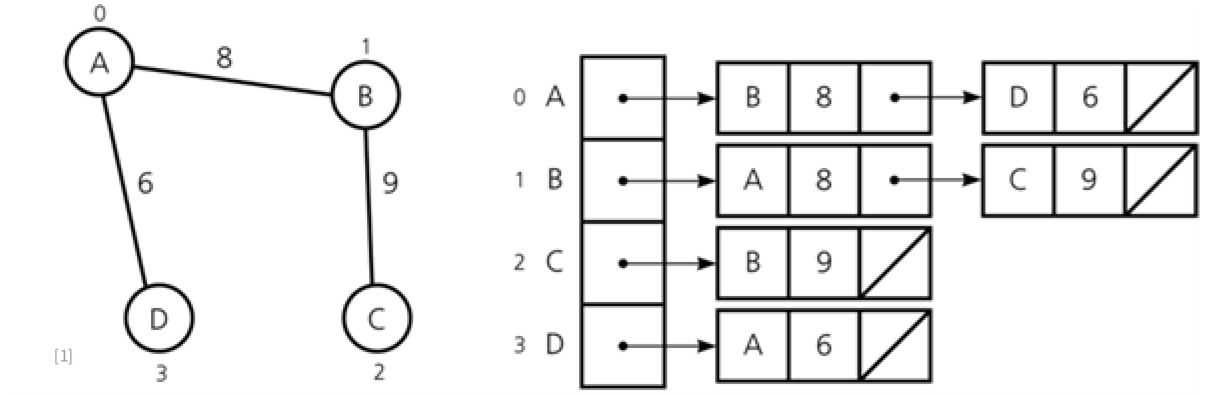
Directed graph



[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Adjacency List: Example

Weighted undirected graph



[1]

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Adjacency List: Implementation

```
class undirected_graph():
  def __init__(self, nodes, edges):
    self.v = nodes[:]
    self.e = {}
    for node in nodes:
      self.e[node] = []

    for (u, v) in edges:
      self.e[v].append(u)
      self.e[u].append(v)
```

```
v = ['a', 'b', 'c']
e = [('a', 'b'), ('b', 'c')]
graph = undirected_graph(v, e)
print(graph.e)
```

```
{'a': ['b'], 'b': ['a', 'c'], 'c': ['b']}
```

- Can you modify this to
  - directed graph?
  - weighted graph?

Google Interview Question

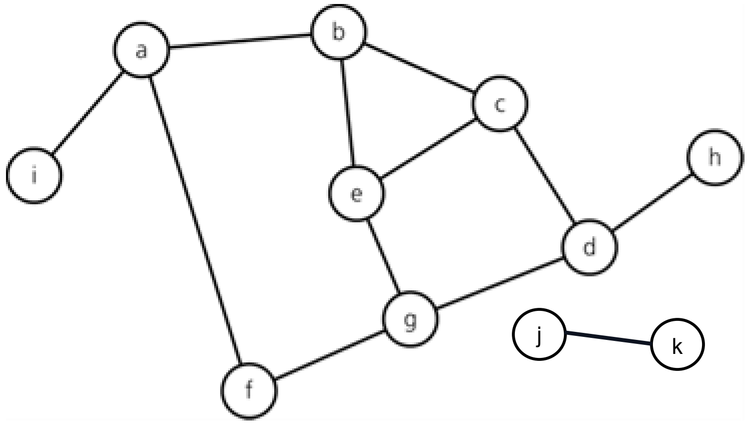
- Design a Graph class.
    - What is the graph for? What data to be stored?
    - **How big** will be the graph?
    - Is the graph expected to be **sparse or dense**?
    - **What operations** are we going to use most frequently?
    - Is the node & edge likely **static or dynamic**?

03

Graph Traversal

Graph Traversal

- Goal: Visiting all nodes once in the graph.
  - *E.g.*, for searching a specific node
  - Caution: graphs may have **cycles** and may be **disconnected**. We still should visit all nodes, but should not visit the same node more than once, and finish once we visit all of them.

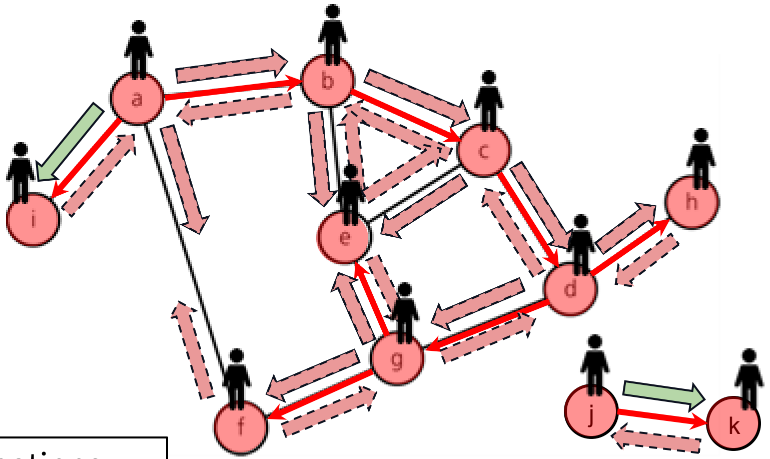


[1]

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Depth First Search (DFS)

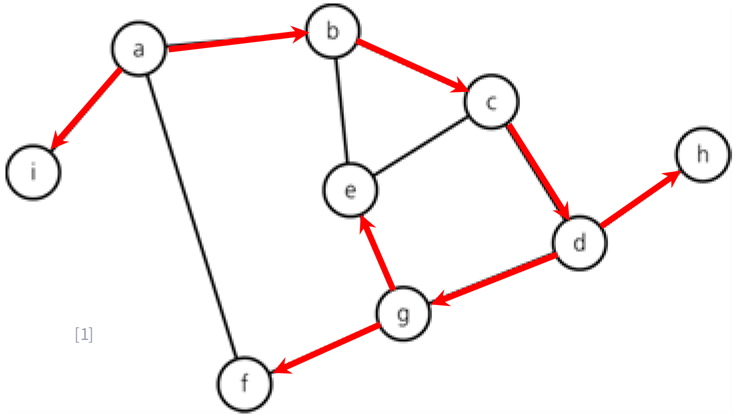
- Start from an arbitrary node.
- Visit any connected unvisited node from there.
- If no more node to visit, **backtrack** to the previous one, and keep going.



This slide is best seen with animations.

Depth First Search (DFS)

- To backtrack, we need to trace our footprints!
- What data structure would work best?
  - We backtrack to the most recently visited previous node!
  - **Last-in, First-out**: **Stack** would work nicely here!



[1]

Node visited	Stack (bottom to top)
a	a
b	a b
c	a b c
d	a b c d
g	a b c d g
e	a b c d g e
(backtrack)	a b c d g
f	a b c d g f
(backtrack)	a b c d g
(backtrack)	a b c d
h	a b c d h
(backtrack)	a b c d
(backtrack)	a b c
(backtrack)	a b
(backtrack)	a
i	a i
(backtrack)	a
(backtrack)	(empty)

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Depth First Search (DFS)

```
def dfs(self):
  unvisited = self.v.copy()
  stack = Stack()
  while not unvisited.is_empty():
    visit(unvisited[0])   # visit origin
    stack.push(unvisited[0])
    del unvisited[0]

    while not stack.is_empty():
      curr = stack.peek()
      if there remains an unvisited city from curr:
        next = select an unvisited city from curr
        visit(next)
        stack.push(next)
        delete next from unvisited
      else:
        stack.pop()   # backtracking
```

We need this loop to take care of **disconnected nodes**!

Do something when we **visit** the node. (*e.g.*, print, compare, ⋯)

For each connected node from `curr` node (accessed by internal **adjacency matrix or list**), check if it is in the `unvisited`.

upstage Education

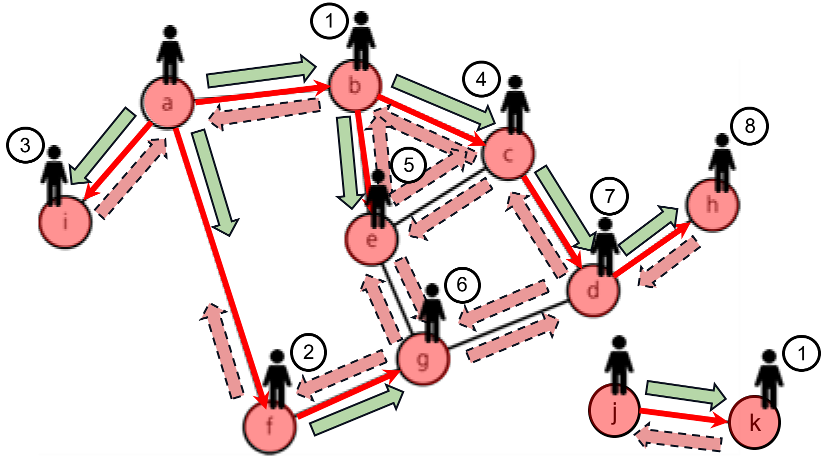
Time Complexity of DFS

With adj. matrix **With adj. list**

- At each step,
  - We visit a node *x*. **O(1)**
  - For the node *x*, we search the list of adjacent nodes. **O(|V|)** ← Total number of search = number of edges (|E|)
  - If there is an unvisited adjacent node, move to there. **O(1)**
  - If not, backtrack to the previously visited node. **O(1)**
- How many such steps? **O(|V|)**
- In total: **O(|V|²)** **O(|V| + |E|)**

Breadth First Search (BFS)

- Start from an arbitrary node.
- Visit all connected unvisited node from there.
- In the next step, repeat the same thing on those nodes. Keep going!

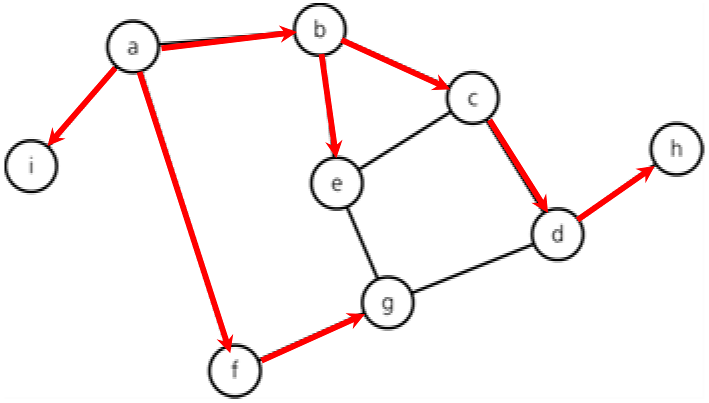


This slide is best seen with animations.

Breadth First Search (BFS)

- We keep track of the order to process: ①, ②, ③, ⋯
- What data structure would work best?
  - We process the waiting nodes in the order of our visit.
  - **Fast-in, First-out**: **Queue** would work nicely here!

[1]



Node visited	Queue (front to back)
	a
a	b f i
b	f i c e
f	i c e g
i	c e g
c	e g d
e	g d
g	d
d	h
h	(empty)

[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Breadth First Search (BFS)

```
def bfs(self):
  unvisited = self.v.copy()
  queue = Queue()
  while not unvisited.is_empty():
    queue.enqueue(some unvisited node)

    while not queue.is_empty():
      curr = queue.dequeue()
      visit(curr)
      delete curr from unvisited

      for city in all unvisited cities connected from curr:
        queue.enqueue(city)
```

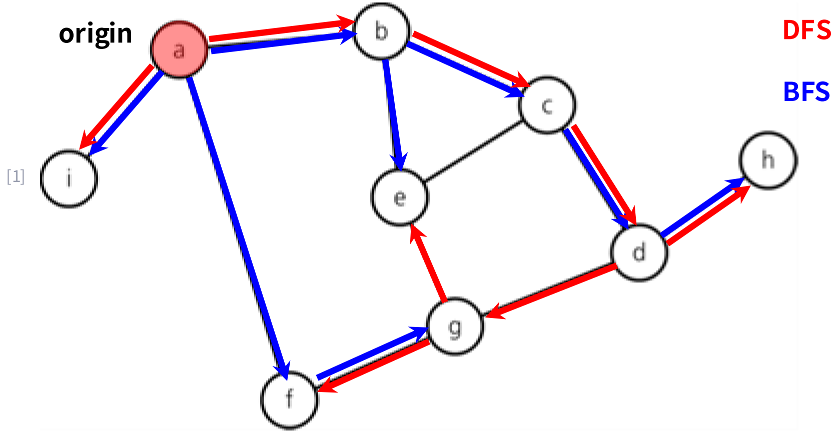
We need this loop to take care of **disconnected nodes**!

Do something when we **visit** the node. (*e.g.*, print, compare, ⋯)

For each connected node from `curr` node (accessed by internal **adjacency matrix or list**), check if it is in the `unvisited`.

DFS *vs.* BFS

- DFS and BFS visit the nodes in different order.
  - BFS visits nodes closer to the origin first. With an unweighted graph, the path discovered by BFS is a **shortest path** from the origin to that node.
- Following the arrows, we build a tree.
  - Recall that the tree is a special graph, connected without a cycle.



[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Time Complexity of BFS

With adj. matrix With adj. list

- At each step,
  - Dequeue a node from the queue and visit it (call it *x*). O(1)
  - From the node *x*, enque all unvisited adjacent nodes. O(|V|)

Total number of search = number of edges (|E|)

- How many such steps? O(|V|)
- In total: O(|V|^2) O(|V| + |E|)

04

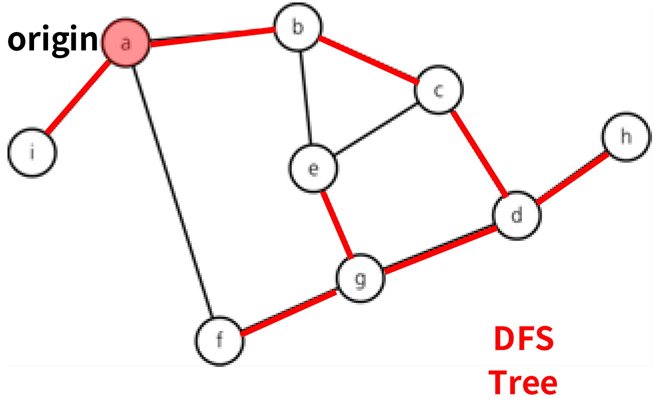
Minimum Spanning Trees

upstage Education

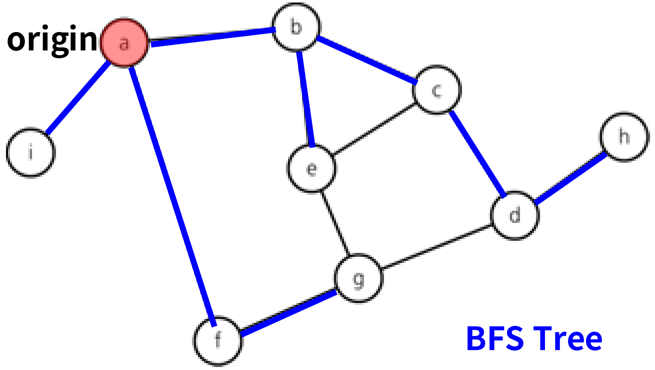
Spanning Trees

- Given an undirected connected graph *G*, find a tree as a subgraph of *G* that contains all of *G*'s vertices.
  - Recall that a Tree is a connected graph with no cycles, and it must contain |V| - 1 edges.

- For unweighted graphs, we may use DFS or BFS:



[1]

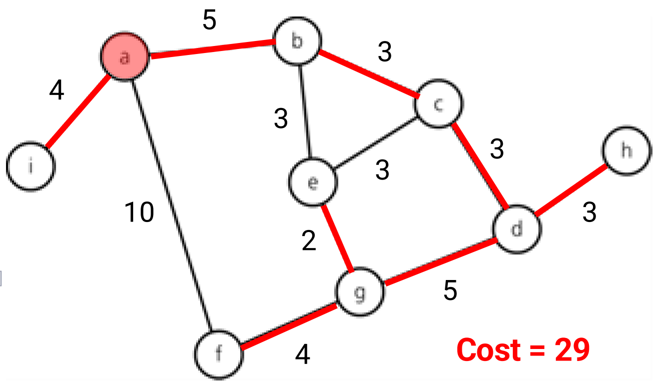


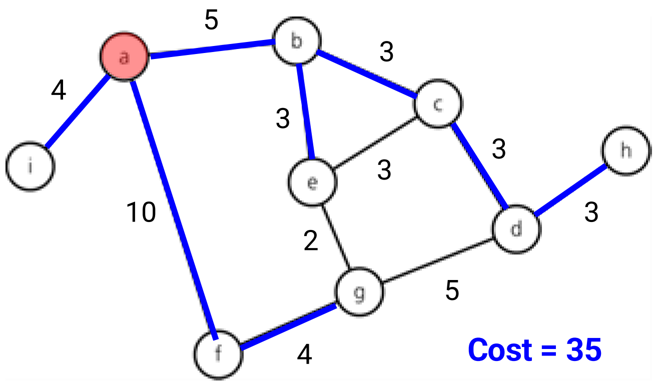
[1] https://user.ceng.metu.edu.tr/~tcan/ceng301_s1516/Schedule/week14.pdf

Minimum Spanning Trees

- What about the graph is **weighted**?
  - Each edge is no longer equal; they have different **cost**!
  - There may be multiple spanning trees.
    → We are interested in building a valid spanning tree with **minimal cost**.

[1]

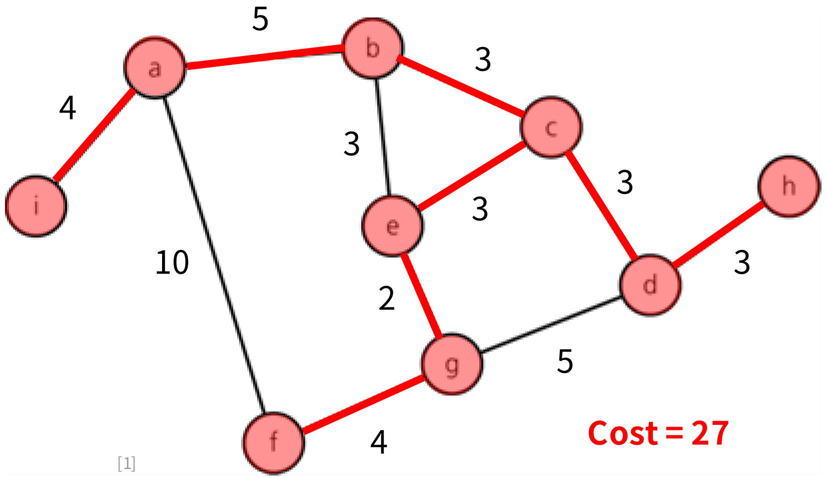




Minimum Spanning Trees

- Main idea: let's try a **greedy** approach (choose the option that looks like best at the current moment, not worrying about the future.)
  - Start with a given node selected only.
  - At each step, choose a **least-cost edge connecting the visited region and the unvisited region**.
  - Keep doing this, until all nodes are within the visited region.
- **Prim's algorithm** is a rare case that a greedy algorithm guarantees global optimality.
- Starting from different initial node may result in different tree, but the minimum cost is always the same.

Minimum Spanning Trees: Illustration



Minimum Spanning Trees: Prim's Algorithm

```
def prims_mst(graph, start):
  visited = [start]
  unvisited = graph.nodes.copy().remove(start)
  mst = []

  while not unvisited.is_empty():
    e = least-cost edge from visited to unvisited
    u = the node consisting of e from the unvisited side
    visited.append(u)
    unvisited.remove(u)
    mst.append(e)
```

Time complexity? **$O(|V| \cdot (|V|+|E|))$**

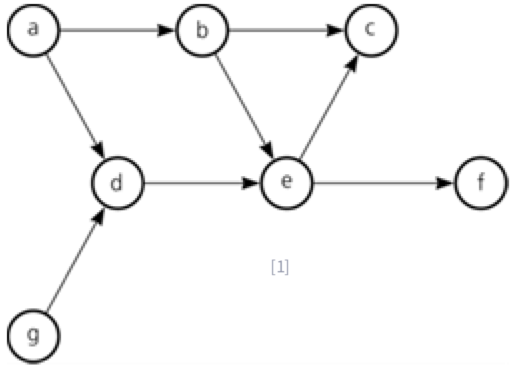
Note that this can be further improved by using Priority queue.
This is beyond the scope of this course.

05

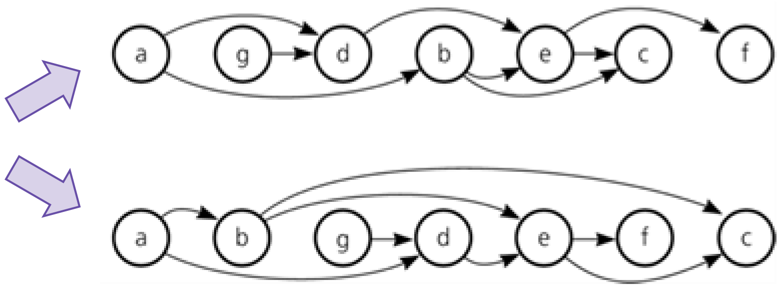
Topological Sorting

Topological Sorting

- On a directed graph without cycles, list the nodes in “topological order”:
  - An order of vertices in which vertex *x* precedes vertex *y* if there is an edge from *x* to *y*.
  - Usually, there are multiple topological orders for a directed graph.

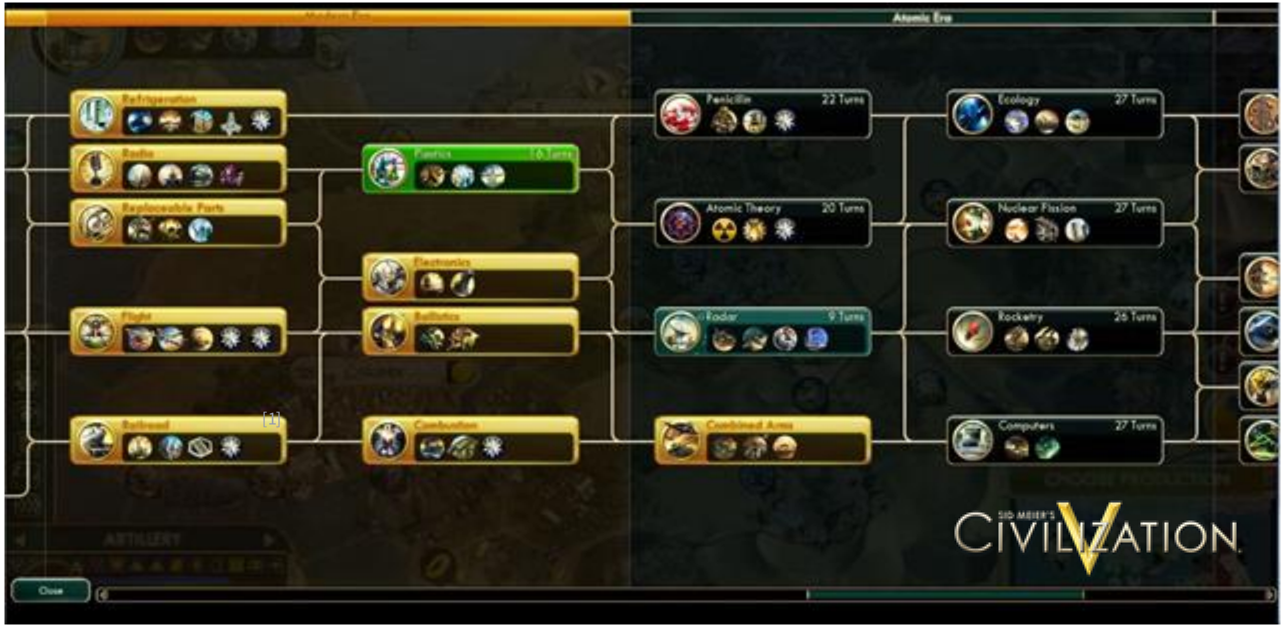


[1]



[1]

Examples



[1]

[1] https://harshpopculture.wordpress.com/2013/09/02/civilization-v-not-all-sequels-are-as-bad-as-duke-nukem-forever/

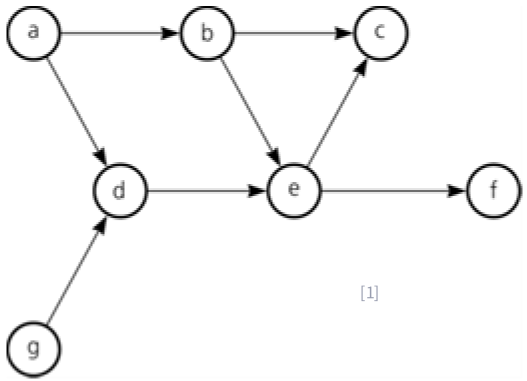
Topological Sorting

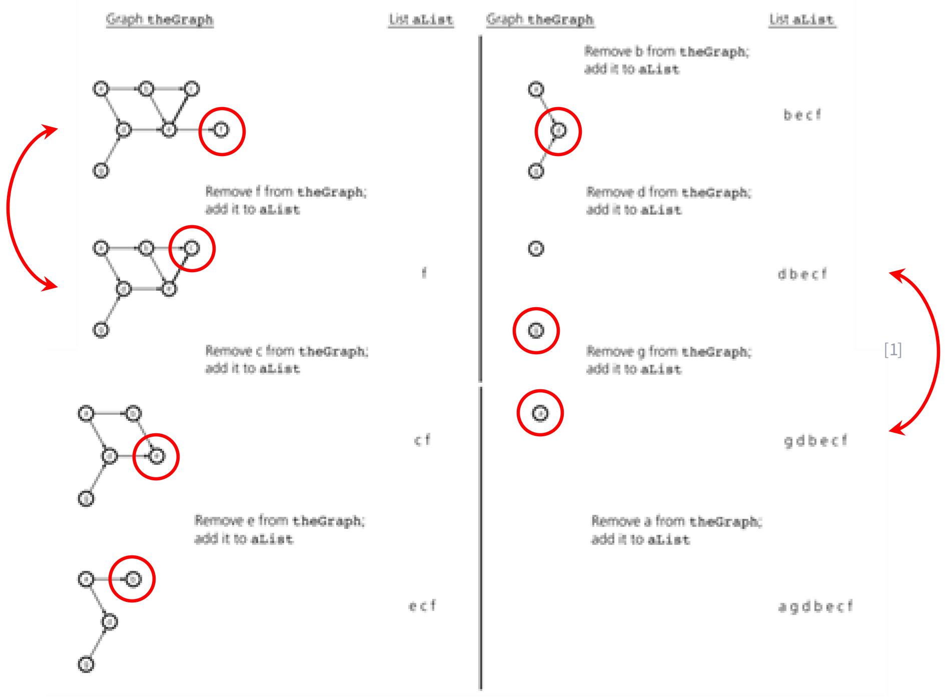
- Main idea: We should put a node without successors at the end of the output!
  - Choose a node without successors, put it at the end of the current list.
  - Detach the chosen node and all edges to it.
  - Recursively solve with the remaining graph, until no node remains.

- As we delete all edges to already chosen nodes, the next highest ones will become available to choose.

upstage Education

Topological Sorting: Illustration




Graph theGraph
List aList
Graph theGraph
List aList
Remove f from theGraph; add it to aList
f
Remove c from theGraph; add it to aList
c f
Remove e from theGraph; add it to aList
e c f
Remove b from theGraph; add it to aList
b e c f
Remove d from theGraph; add it to aList
d b e c f
Remove g from theGraph; add it to aList
g d b e c f
Remove a from theGraph; add it to aList
a g d b e c f
[1]

upstage Education

Topological Sorting: Implementation

```
def topological_sort(nodes, edges):
  output = []
  curr_last = Set of all nodes with no incoming edge

  while curr_last is not empty:
    target = curr_last[0]
    output.append(target)
    del curr_last[0]
    for each node m with an incoming edge e from target:
      Remove e from the graph
      if m has no other incoming edges:
        curr_last.append(m)

  if len(output) < len(nodes):
    return error  # graph has at least one cycle
  else:
    return output
```

O(|E|)

Up to |V| times!
Each node can be selected up to once.

Up to |E| times!
Each edge can be removed up to once.

At this point, we have no more nodes at the last. If there is no cycle, we should be done now.

Time complexity? **O(|V| + |E|)**

upstage

Building intelligence for the future of work