Algorithms

授課老師：張景堯

Breadth-First Search

Chapter.5-Graph Traversal

Traversing a Graph(圖的遍歷)

- One of the most fundamental graph problems is to traverse every edge and vertex in a graph.
- For ***efficiency***, we must make sure we visit each edge at most twice. (不要多走)
- For ***correctness***, we must do the traversal in a systematic way so that we don’t miss anything. (不能少走)
- Since a maze is just a graph, such an algorithm must be powerful enough to enable us to get out of an arbitrary maze. (最好可用來解迷宮)

https://d18l82el6cdm1i.cloudfront.net/image_optimizer/54429cab5dc05daafae44df6aae405409f240684.gif

Marking Vertices(標示頂點)

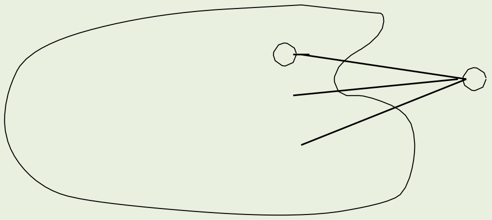
- The key idea is that we must mark each vertex when we first visit it, and keep track of what have not yet completely explored.
- Each vertex will always be in one of the following three states:
  - *undiscovered* – the vertex in its initial, virgin state.
  - *discovered* – the vertex after we have encountered it, but before we have checked out all its incident edges.
  - *processed* – the vertex after we have visited all its incident edges.
- Obviously, a vertex cannot be *processed* before we discover it, so over the course of the traversal the state of each vertex progresses from *undiscovered* to *discovered* to *processed*. (遍歷就是從未發現經已發現到已處理)

To Do List(放入清單以便遊歷)

- We must also maintain a structure containing all the vertices we have discovered but not yet completely explored.
- Initially, only a single start vertex is considered to be discovered.
- To completely explore a vertex, we look at each edge going out of it. For each edge which goes to an undiscovered vertex, we mark it *discovered* and add it to the list of work to do.
- Note that regardless of what order we fetch the next vertex to explore, each edge is considered exactly twice, when each of its endpoints are explored.
  (拜訪清單的資料結構關係到走的順序,都不要多走)

Correctness of Graph Traversal

- Every edge and vertex in the connected component is eventually visited.
- Suppose not, i.e. there exists a vertex which was unvisited whose neighbor was visited. This neighbor will eventually be explored so we would visit it:



(互聯的頂點終將遍歷，如果某頂點沒能走到，其相連的鄰近頂點終將被訪問，讓該頂點也能被遊歷到。)

Breadth-First Traversal(先寬徧歷)

- The basic operation in most graph algorithms is completely and systematically traversing the graph. We want to visit every vertex and every edge exactly once in some well-defined order.
- Breadth-first search is appropriate if we are interested in shortest paths on unweighted graphs.
(無加權圖上適用先寬徧歷來求取最短距離)

Data Structures for BFS

- We can use an attribute of Vertex class to maintain our knowledge about each vertex in the graph. For example, the attribute d means distance to start vertex.
- A vertex is initialized as *undiscovered*. i.e., d == -1
- A vertex is *discovered* the first time we visit it. i.e., d != -1
- A vertex is considered *processed* after we have traversed all outgoing edges from it.
- Once a vertex is discovered, it is placed on a **FIFO queue**. Thus the oldest vertices / closest to the root are expanded first.

GraphVertex Class for BFS

```
class GraphVertex:
    def __init__(self, label):
        self.label = label
        self.d = -1 #Distance to start vertex
        self.parent = None #The vertex discovers me
        self.edges = []  #Edges in Adjacency List
     def __str__(self):
        output = f"{self.label}(d={self.d}): "
        for e in self.edges:
            output += f"{self.label}->{e.label} "
        return output
```

BFS Implementation $\Theta(n + m)$

```
def BFS(start:GraphVertex):
    queue = []
    start.d = 0
    queue.append(start)
    while queue:
        curr = queue.pop(0)
        process_vertex_early(curr.label)
        for v in curr.edges:
            process_edge(curr.label, v.label)
            if v.d == -1: #Unvisited vertex
                v.d = curr.d + 1
                v.parent = curr
                queue.append(v)
        process_vertex_late(curr.label)
```

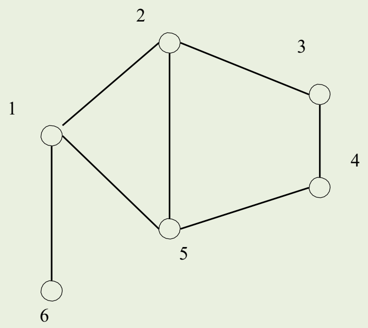
Exploiting Traversal

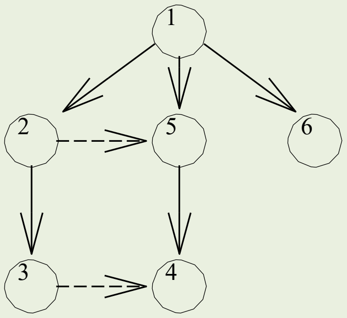
- We can easily customize what the traversal does as it makes one official visit to each edge and each vertex. By setting the functions to

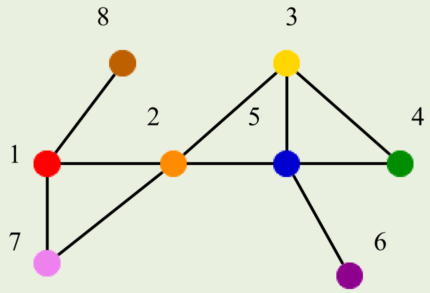
```
def process_vertex_early(label:int):
    print("Processing vertex",label)

def process_edge(i:int, j:int):
    print(f"Processing edge ({i}, {j})")

def process_vertex_late(label:int):
    print(f"Vertex {label} processed")
```

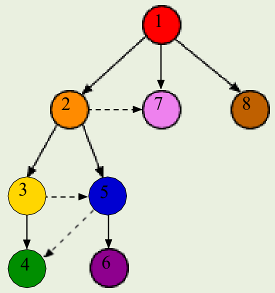
- we print each vertex and edge exactly once.

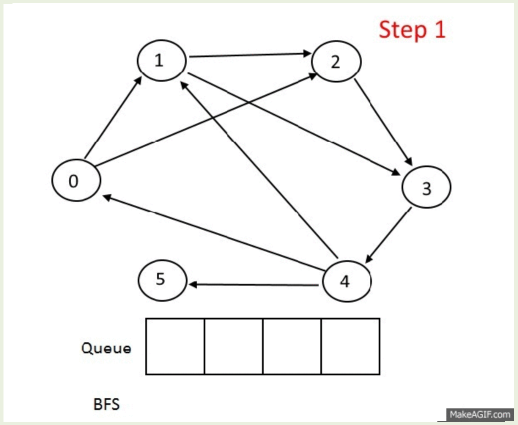
BFS Examples



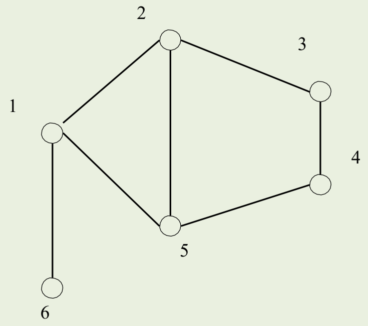


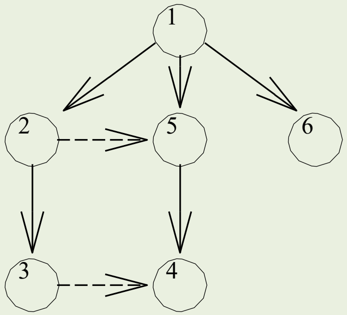






BFS Result





```
edge_list=[[1,2],[2,1],[1,5],[5,1],[1,6],[6,1],[2,3],[3,2],[2,5],[5,2],[3,
4],[4,3],[4,5],[5,4]]

graph = initial_graph(7, edge_list)

BFS(graph[1])

for v in graph:

    print(v)
```

Shortest Paths and BFS

- In BFS vertices are discovered in order of increasing distance from the root, so this tree has a very important property.
- The unique tree path from the root to any node $x \in V$ uses the smallest number of edges (or equivalently, intermediate nodes) possible on any $\underline{\text{root-to-}x}$ path in the graph.

(在unweighted graph BFS所遍歷的路程即是從起點到該節點的最短路徑)

Finding Paths

- The parent field array set within bfs() is very useful for finding interesting paths through a graph.
- The vertex which discovered vertex t is defined as t.parent.
- The parent relation defines a tree of discovery with the initial search node as the root of the tree.

(從某頂點 t 透過parent欄位倒推就能得到從初始頂點到該頂點的路徑)

Recursion and Path Finding

- We can reconstruct this path by following the chain of ancestors from x to the root. Note that we have to work backward. We cannot find the path from the root to x, since that does not follow the direction of the parent pointers. Instead, we must find the path from x to the root.

```python
def find_path(start:GraphVertex, end:GraphVertex):
    if start == end or end is None:
        print(start.label, end=' ')
    else:
        find_path(start, end.parent)
        print(end.label, end=' ')

for v in graph:
    if v.d != -1:
        print('1->', v.label, end=': ')
        find_path(graph[1], v)
        print()
```

Connected Components (連接組件)

- The *connected components* of an undirected graph are the separate "pieces" of the graph such that there is no connection between the pieces.(可得到幾個不相連的子圖)
- Many seemingly complicated problems reduce to finding or counting connected components. For example, testing whether a puzzle such as Rubik's cube or the 15-puzzle can be solved from any position is really asking whether the graph of legal configurations is connected.
- Anything we discover during a BFS must be part of the **same connected component**. We then repeat the search from any undiscovered vertex (if one exists) to define the next component, until all vertices have been found:

Implementation $\Theta(n + m)$

```
def connected_components(graph:list):
    c = 0
    for v in graph:
        if v.d == -1: #unvisited
            c += 1
            print("Component {0}:".format(c))
            BFS(v)
```

Two-Coloring Graphs

- The *vertex coloring* problem seeks to assign a label (or color) to each vertex of a graph such that no edge links any two vertices of the same color.
- A graph is *bipartite* if it can be colored without conflicts while using only two colors. Bipartite graphs are important because they arise naturally in many applications.
- For example, consider the graph of students and the courses they are registered for. No edges go between student pairs or course pairs, so the graph is bipartite.

Depth-First Search

Chapter.5-Graph Traversal

Depth-First Search $\Theta(n+m)$

- DFS has a neat *recursive* implementation which eliminates the need to explicitly use a stack.

```python
class GraphVertex:
    def __init__(self, label):
        self.label = label
        self.d = 0 #Distance to start vertex
        self.discovered = False
        self.processed = False
        self.parent = None #The vertex discovers me
        self.edges = []  #Edges in Adjacency List
    def __str__(self):
        output = f"{self.label}(d={self.d}): "
        for e in self.edges:
            output += f"{self.label}->{e.label} "
        return output
```

Depth-First Search $\Theta(n + m)$(Cont.)

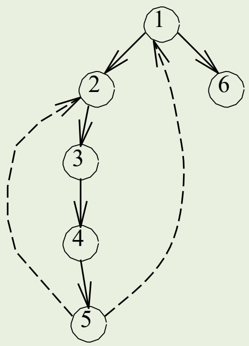
```
def DFS(v:GraphVertex):
    v.discovered = True
    process_vertex_early(v)
    for t in v.edges:
        if not t.discovered:
            t.parent = v
            t.d = v.d + 1
            process_edge(v, t)
            DFS(t)
        elif v.parent != t:
            process_edge(v, t)
     process_vertex_late(v)
     v.processed = True
```

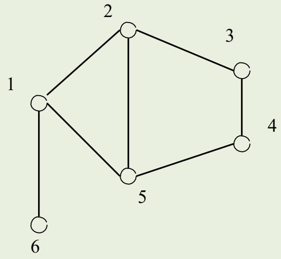
Backtracking and Depth-First Search

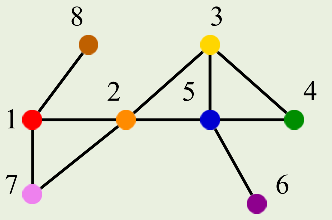
- **Depth-first search** uses essentially the same idea as backtracking. (DFS基本上就是回溯法)
- Both involve *exhaustively* searching all possibilities by advancing if it is possible, and backing up as soon as there is no unexplored possibility for further advancement. Both are most easily understood as recursive algorithms.

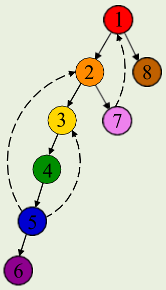
The *Key* Idea with DFS

- A depth-first search of a graph organizes the edges of the graph in a precise way.
- In a DFS of an undirected graph, we **assign a direction** to each edge, from the vertex which discover it:









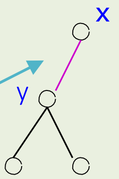
(無向圖經過DFS後，從起始頂點賦予每個邊方向性)

Edge Classification for DFS

- Every edge is either:

1. A Tree Edge

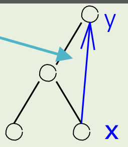
```
y.parent is x
```



2. A Back Edge

   to an ancestor

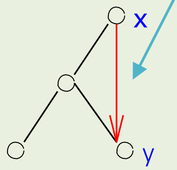
```
y.discovered and not y.processed and x.d > y.d
```



3. A Forward Edge

   to a decendant

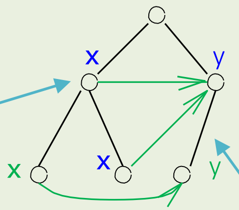
```
y.processed and x.d < y.d
```



4. A Cross Edge

   to a different node

```
y.discovered and not y.processed and x.d <= y.d or y.processed and x.d >= y.d
```



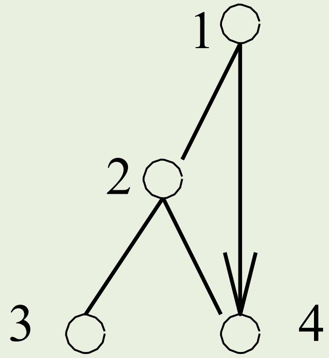
- On any particular DFS or BFS of a directed or undirected graph, each edge gets classified as one of the above.

Edge Classification Implementation

```
from enum import Enum
class EdgeType(Enum):
    TREE = 0
    BACK = 1
    FORWARD = 2
    CROSS = 3

def edge_classification(x:GraphVertex, y:GraphVertex)->EdgeType:
    if y.parent is x:
        return EdgeType.TREE
    elif y.discovered and not y.processed:
        return EdgeType.BACK if x.d > y.d else EdgeType.CROSS
    elif y.processed:
        return EdgeType.FORWARD if x.d < y.d else EdgeType.CROSS
    print(f"Warning: unclassified edge ({x.label},{y.label})")
    return None
```

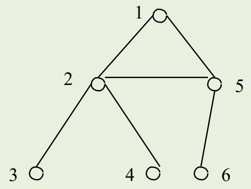
DFS: Tree Edges and Back Edges Only

- The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.
- In a DFS of an **undirected graph**, every edge is either a tree edge or a back edge.
- Why? Suppose we have a forward edge. We would have encountered (4, 1) when expanding 4, so this is a back edge.

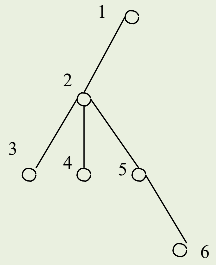


No Cross Edges in DFS

- Suppose we have a cross-edge



When expanding 2, we would discover 5, so the tree would look like:



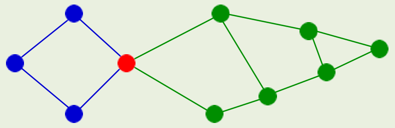
DFS Application: Finding Cycles

- Back edges are the key to finding a cycle in an undirected graph.
- Any back edge going from x to an ancestor y creates a cycle with the path in the tree from y to x.

```
def process_edge(i:GraphVertex, j:GraphVertex):
    if graph[j].parent != graph[i]: #find back edge in DFS on undirected graph
        find_path(graph[j], graph[i])
        print(f" Cycle from {i} to {j}")
```

Articulation Vertices(肢接頂點)

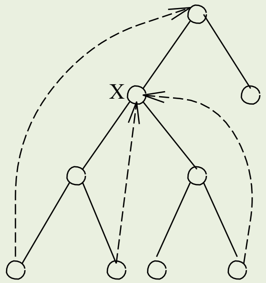
- Suppose you are a terrorist, seeking to disrupt the telephone network. Which station do you blow up?



- An *articulation vertex* is a vertex of a connected graph whose deletion disconnects the graph. (刪了肢接頂點圖就不相連了)
- Clearly connectivity is an important concern in the design of any network.
- Articulation vertices can be found in $O(n(m+n))$ – just delete each vertex to do a DFS on the remaining graph to see if it is connected.

A Faster $O(n+m)$ DFS Algorithm

- In a DFS tree, a vertex v (other than the root) is an articulation vertex iff v is not a leaf and some subtree of v has no back edge incident until a proper ancestor of v.



The root is a special case since it has no ancestors.

X is an articulation vertex since the right subtree **does not have a back edge to a proper ancestor**.

Leaves cannot be articulation vertices

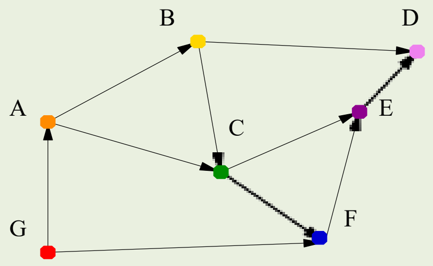
(找肢接頂點不用一個個刪然後DFS檢測連接性，只要一次DFS就能判定：非root、非leaf、有子樹沒back edge到上代頂點，就是了)

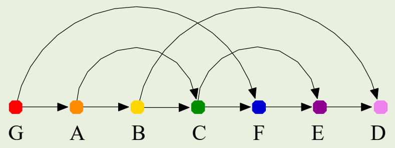
Topological Sorting

Chapter.5-Graph Traversal

Topological Sorting(拓蹼排序)

- A directed, acyclic graph (*DAG*) has no directed cycles.

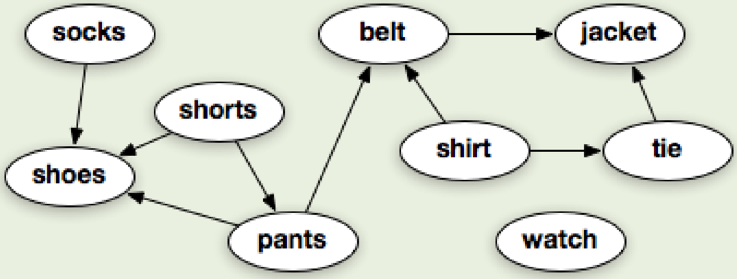


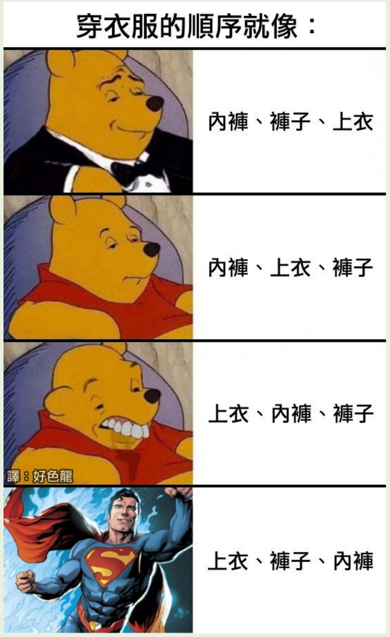


- A topological sort of a graph is an ordering on the vertices so that all edges go from left to right.
- DAGs (and only DAGs) has at least one topological sort (here *G, A, B, C, F, E, D*).

Applications of Topological Sorting

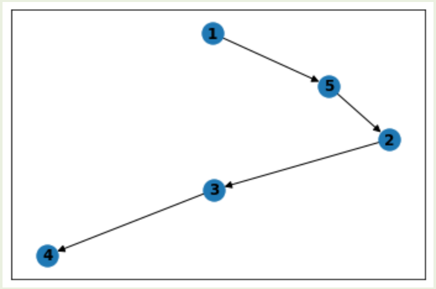
- Topological sorting is often useful in scheduling jobs in their proper sequence. In general, we can use it to order things given precedence constraints.
- Example: Dressing priority schedule





Example: Identifying errors in DNA fragment assembly

- Certain fragments are constrained to be to the left or right of other fragments, unless there are errors.



(1)A G T A C
(2)A C A D A
(3)A D A G T
(4)D A G T A
(5)T A C A D

```
A G T A C A D A G T A
---------------------
A G T A C
    T A C A D
      A C A D A
          A D A G T
            D A G T A
```

- Solution – build a DAG representing all the left-right constraints. Any topological sort of this DAG is a consistent ordering. If there are cycles, there must be errors.

Topological Sorting via DFS

- A directed graph is a DAG if and only if no back edges are encountered during a depth-first search.
- The conceptually simplest linear-time algorithm for topological sorting performs a depth-first search of the DAG to identify the complete set of ***source vertices***, where source vertices are vertices of **in-degree zero**.
(從沒有進入邊的頂點開始)
- Labeling each of the vertices in the **reverse order** that they are marked *processed* finds a topological sort of a DAG.
(過程中將標註'已處理'的頂點**反序**呈現就是拓樸排序順序了)
- Why?

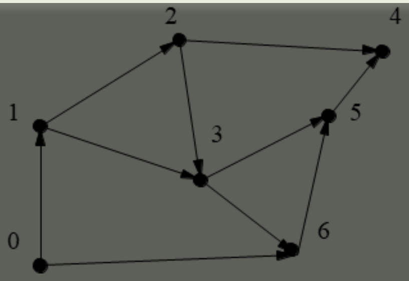
Case Analysis for directed edge {x,y}

■ Consider what happens to each directed edge $\{x, y\}$ as we encounter it during the exploration of vertex x:

- If y is currently *undiscovered*, then we then start a DFS of y before we can continue with x. Thus y is marked *completed* before x is, and x appears before y in the topological order, as it must.
  (如果y未發現，我們會先做y的遍歷，反而比x先完成)
- If y is *discovered* but not *completed*, then $\{x, y\}$ is a back edge, which is forbidden in a DAG.
  (如果y已發現但未完成，$\{x, y\}$就是back edge，不應發生)
- If y is *completed*, then it will have been so labeled before x. Therefore, x appears before y in the topological order, as it must.
  (y已完成遍歷，屬forward或cross edge更是表示拓樸序在x後)

Topological Sorting Implementation

```
order = []
def process_vertex_late(label:int):
    order.insert(0, label)


def process_edge(i:GraphVertex, j:GraphVertex):
    if edge_classification(i, j) == EdgeType.BACK:
        raise Exception(f"Cycle from {i.label} to {j.label}")
```



- We insert each vertex at the first place of a list soon as we have evaluated all outgoing edges. The first vertex always has no incoming edges from any vertex in the list, repeatedly yields a topological ordering.

Topological Sorting by Indegree-0 Node Removal $\Theta(n + m)$

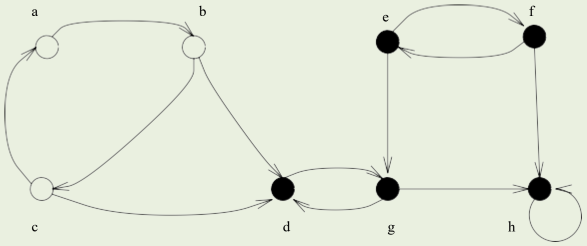
```
def topsort(g:GraphVertex)->list:
    zeroin = [] # queue for vertices with in-degree 0
    sorted = []
    indeg = [0 for v in graph] # in-degrees of all vertices
    for v in graph: # compute in-degrees
        for e in v.edges:
            indeg[e.label] += 1
    for idx, i in enumerate(indeg):
        if i == 0:
            zeroin.append(graph[i])
```

Topological Sorting by Indegree-0 Node Removal $\Theta(n + m)$ (Cont.)

```
    j = 0
    while zeroin:
        j += 1
        x = zeroin.pop(0)
        sorted.append(x.label)
        for e in x.edges:
            indeg[e.label] -= 1
            if indeg[e.label] == 0:
                zeroin.append(e)
    if j != len(graph):
        print(f"Not a DAG -- only {j} vertices found.")
    return sorted
```

Applications of DFS：Strongly Connected Components(強連接組件)

- A directed graph is strongly connected iff there is a directed path between any two vertices.
  (正方向做DFS，建立一個反方向的Graph也做DFS就能判斷)
- The strongly connected components of a graph is a partition of the vertices into subsets (maximal) such that each subset is strongly connected.

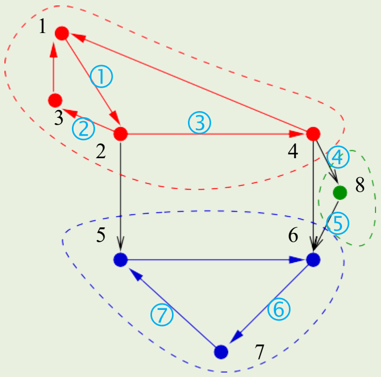


- Observe that no vertex can be in two maximal components, so it is a partition.

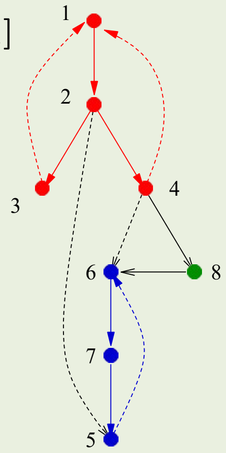
Strongly Connected Components (Cont.)

vertices processed order:[3,5,7,6,8,4,2,1]

○: tree edges discovered order

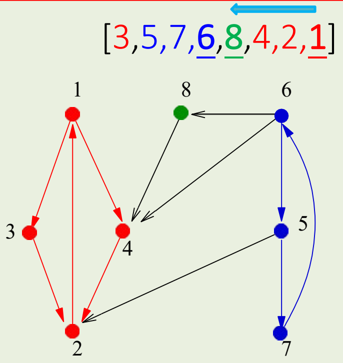


G



$DFS(G)$

[3,5,7,<u>6</u>,<u>8</u>,4,2,<u>1</u>]



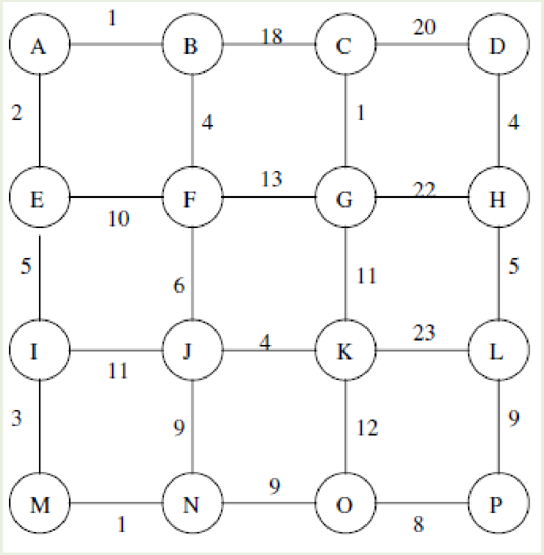
$DFS(G^T)$

- There is an elegant, linear time algorithm to find the strongly connected components of a directed graph using DFS which is similar to the algorithm for biconnected components. (正方向做DFS後得一頂點遍歷順序，再從反方向的Graph根據該順序由右至左反向逐個做DFS就能得強連接組件個數)

Exercises

Problems of the Day

- For the following two graphs, report the order of the vertices encountered on a (1)breadth-first search and (2)depth-first search starting from vertex A. Break all ties by picking the vertices in alphabetical order (i.e., A before Z).



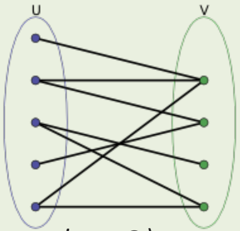
Problems of the Day (Cont.)

- When traversing undirected graph by BFS, What kind of edges could exist in BFS tree?
- When traversing undirected graph by DFS, What kind of edges could exist in DFS tree?
- When traversing directed graph by DFS, What kind of edges could exist in DFS tree?
- When traversing directed acyclic graph by DFS, What kind of edges could exist in DFS tree?

Program Exercises (Moodle CodeRunner)

■Exercise 06 (close at 4/22 23:59)

- Making Wired Connections (Bipartite):
  - ➢ *To design an algorithm that takes a set of pins and a set of wires connecting pairs of pins representing as an adjacency list of **Pin** class, and determines if it is possible to place some pins on the left half of a PCB, and the remainder on the right half, such that each wire is between left and right halves.*
  - ➢ *Return **True**, if such a division exists.*



- Search a Maze :
  - ➢ *Given a 2D array of black(wall, value 1) and white(open area, value 0) entries representing a maze with designated entrance and exit points, find a path from the entrance to the exit, if one exists.*