GRAPHS

School of Artificial Intelligence

GRAPHS

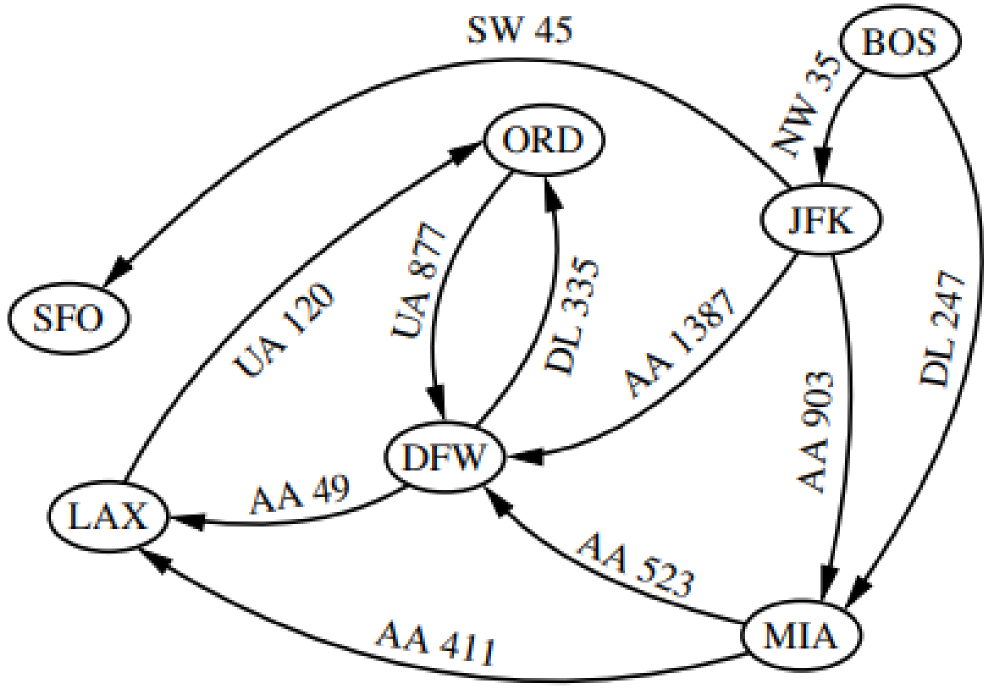
- **End vertices** (endpoints)
  - Two vertices joined by an edge
  - Directed edge: origin -> destination
- **Adjacent**(相邻的) vertices: u and v are adjacent if there is an edge whose end vertices are u and v
- **Incident**(入射): an edge is incident to a vertex if the vertex is one of the edge’s endpoints
- **Outgoing**(输出) edges of a vertex: directed edges whose origin is the vertex
- **Incoming** (输入) edges of a vertex: directed edges whose destination is the vertex
- **Degree** (度) of a vertex: # of incident edges of v
- **In-degree** (入度) : # of incoming edges of v
- **Out-degree** (出度) : # of outgoing edges of v

GRAPHS

- **Simple** graphs: graphs do not have parallel edges or self loops
- **Path**: sequence of vertices and edges, start at a vertex and ends at a vertex
  - Simple: if each vertex in the path is distinct
- **Cycle**: path that starts and ends at the same vertex, and includes at least one edge
  - Simple: if each vertex in the cycle is distinct(except for the first and last one)
- **Acyclic**: directed graph that has no directed cycles

GRAPHS

- Directed path
  - A path such that all edges are directed and are traversed along their direction
  - (BOS, NW35, JFK, AA1387, DFW)
- Directed cycle
  - A cycle such that all edges are directed and are traversed along their direction
  - (LAX, UA1200, ORD, UA877, DFW, AA49, LAX)

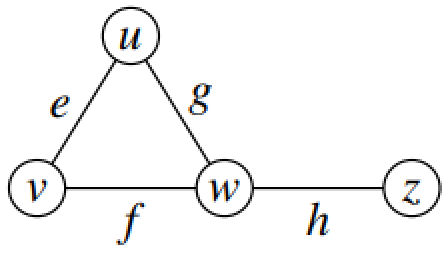


GRAPHS

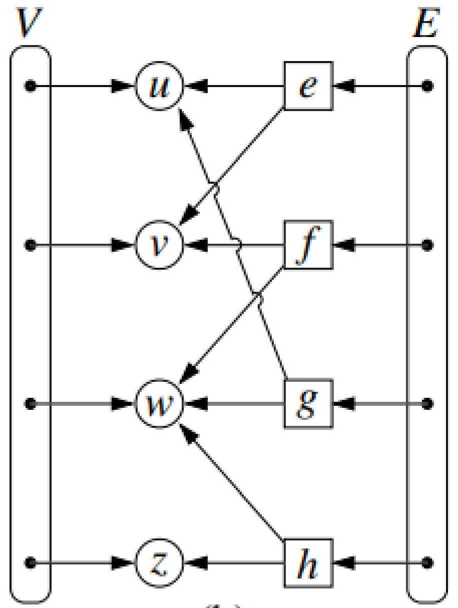
- **Reachability**
    - In a graph G, u **reaches** v, and v is reachable from u, if G has a path from u to v
- **Connectivity**
    - G is **connected**, if for any two vertices, there is a path between them
    - Directed graph: **strongly connected**
- **Subgraph**
    - Graph H whose vertices and edges are subsets of the vertices and edges of G
- **Spanning subgraph**
    - Subgraph of G that contains all the vertices of the graph
- **Forest**
    - Graph without cycles
- **Tree**
    - Connected forest
- **Spanning tree**
    - Spanning subgraph that is a tree

EDGE LIST

- Collections V and E are represented with doubly linked lists (PositionalList)
- Vertex objects
  - Reference to element x
  - Reference to position of the vertex instance in the list V
- Edge objects
  - Reference to element x
  - Reference to the vertex objects associated with the endpoint vertices of e
  - Reference to the position of the edge instance in list E



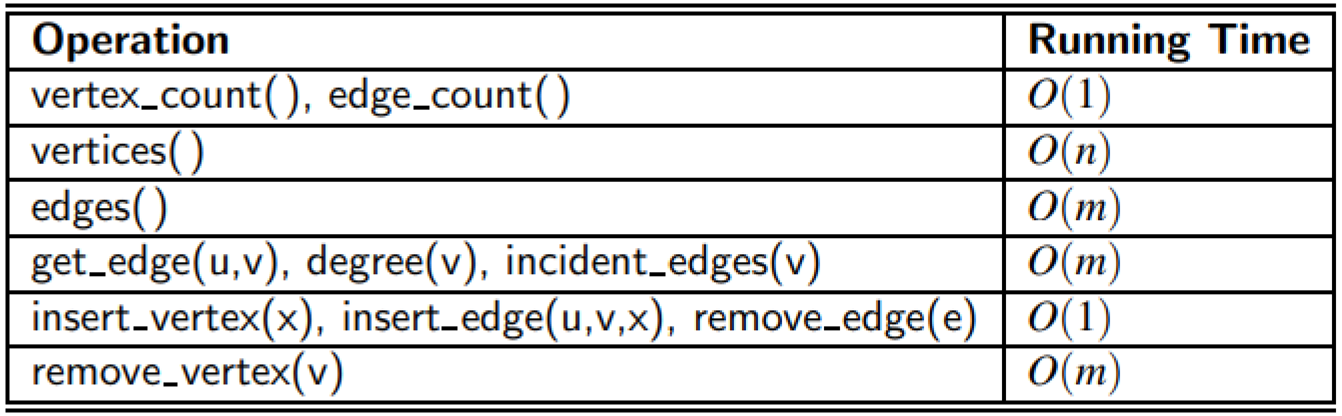
(a)



(b)

EDGE LIST

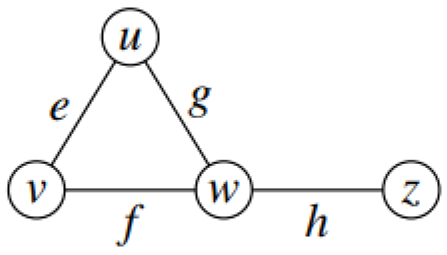
- Performance
- Space: O(n + m)
    - n vertices and m edges
- Running time
    - vertices(): O(n)
    - edges(): O(m)
    - get_edge(): O(m)
        - Most significant limitation
    - remove_vertex(v): O(m)
        - Why?



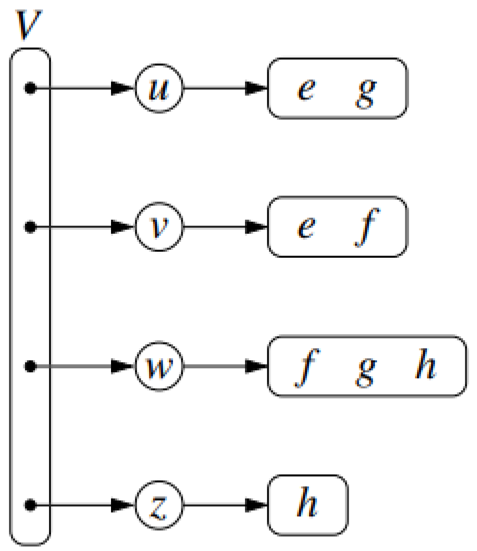
Operation	**Running Time**
vertex_count(), edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

ADJACENCY LIST

- Secondary containers for edges that are associated with each individual vertex
- For each v, maintain a collection I(v) called **incidence collection** of v
- Primary structure: collection V of vertices
  - Positional list
- Each vertex instance
  - Direct reference to its I(v) incidence collection



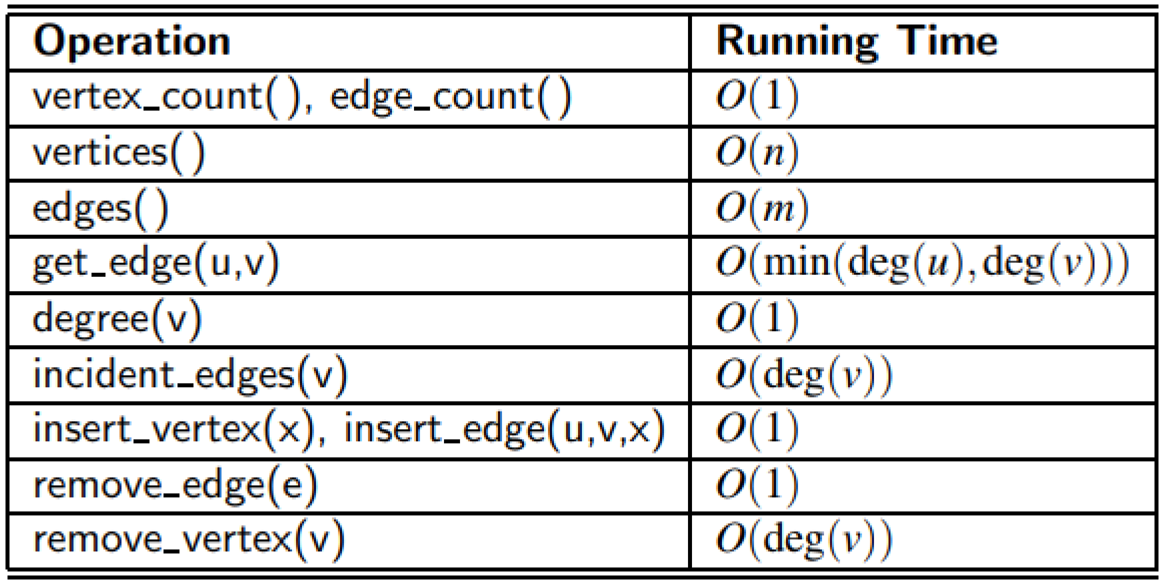
(a)



(b)

ADJACENCY LIST

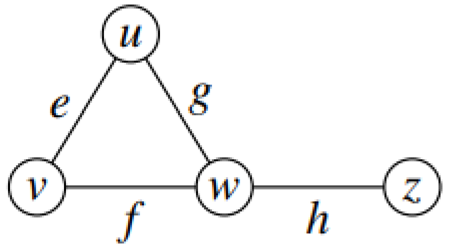
- Performance
- Space: O(n + m)
  - n vertices and m edges
- Running time
  - vertices(): O(n)
  - edges(): O(m)
  - get_edge(): O(min(deg(u), deg(v)))
    - Search through either I(u) or I(v)
  - remove_vertex(v): O(dev(v))



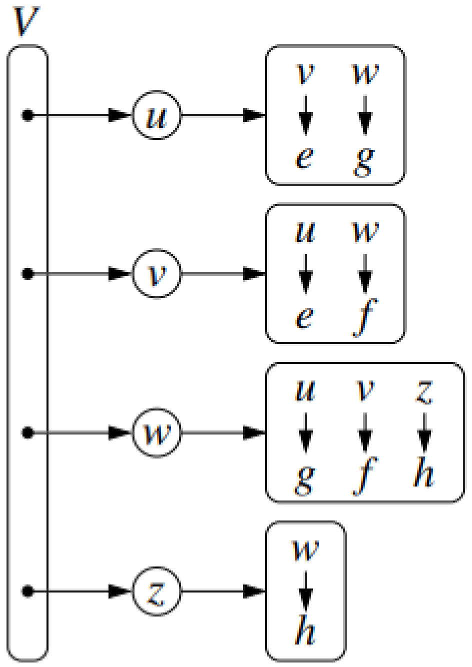
Operation	**Running Time**
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v)	$O(\min(\deg(u), \deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

ADJACENCY MAP

- Adjacency list
  - I(v) uses O(deg(v)) space
  - O(dev(v)) time
- Performance improvement
  - Hash-based map for I(v)
  - get_edge(u,v) can run in expected O(1) time, worst case O(min(deg(u), deg(v))



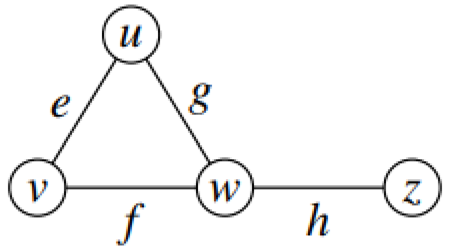
(a)



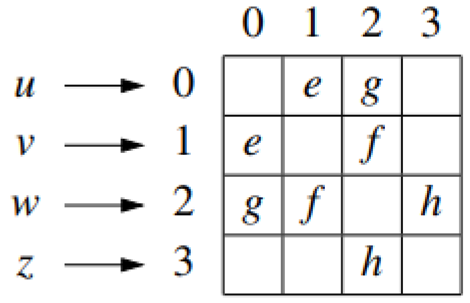
(b)

ADJACENCY MATRIX

- Matrix A (n by n) to locate an edge between a given pair of vertices in worst-case O(1) time
- Verticeis as integers in {0, 1, …, n-1}
- A[I, j] holds a reference to the edge (u, v) if one exists
- Edge (u, v) can be accessed in worst-case O(1) time
- O(n^2) space usage
- Matrix can be used to store only Boolean values, if edges do not store any additional data



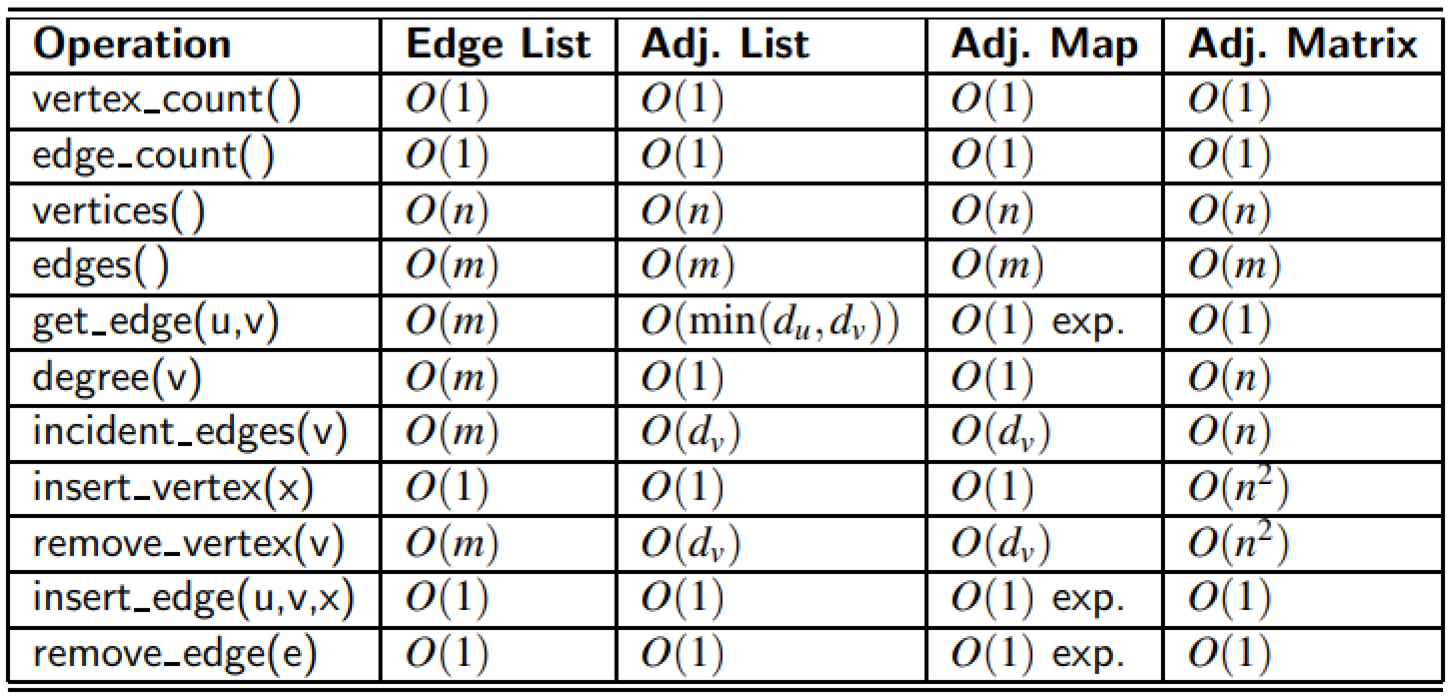
(a)



$$\begin{array}{ccc|c|c|c|c|}
 & & & 0 & 1 & 2 & 3 \\
\hline
u & \rightarrow & 0 & & e & g & \\
\hline
v & \rightarrow & 1 & e & & f & \\
\hline
w & \rightarrow & 2 & g & f & & h \\
\hline
z & \rightarrow & 3 & & & h & \\
\hline
\end{array}$$

(b)

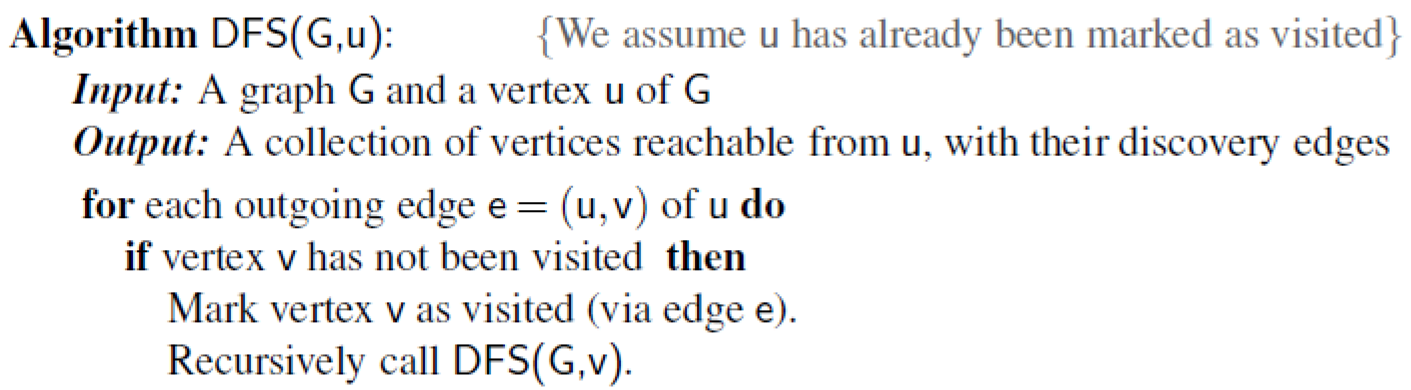
DATA STRUCTURES FOR GRAPHS



Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

GRAPH TRAVERSALS

- **Depth First Search**
  - Maze solving
- Begin at a starting vertex s
  - S is the current vertex u
- For each edge of s
  - If it leads to a visited vertex, ignore
  - If it leads to an unvisited vertex v, make v the current vertex u, make u "visited"
  - repeat

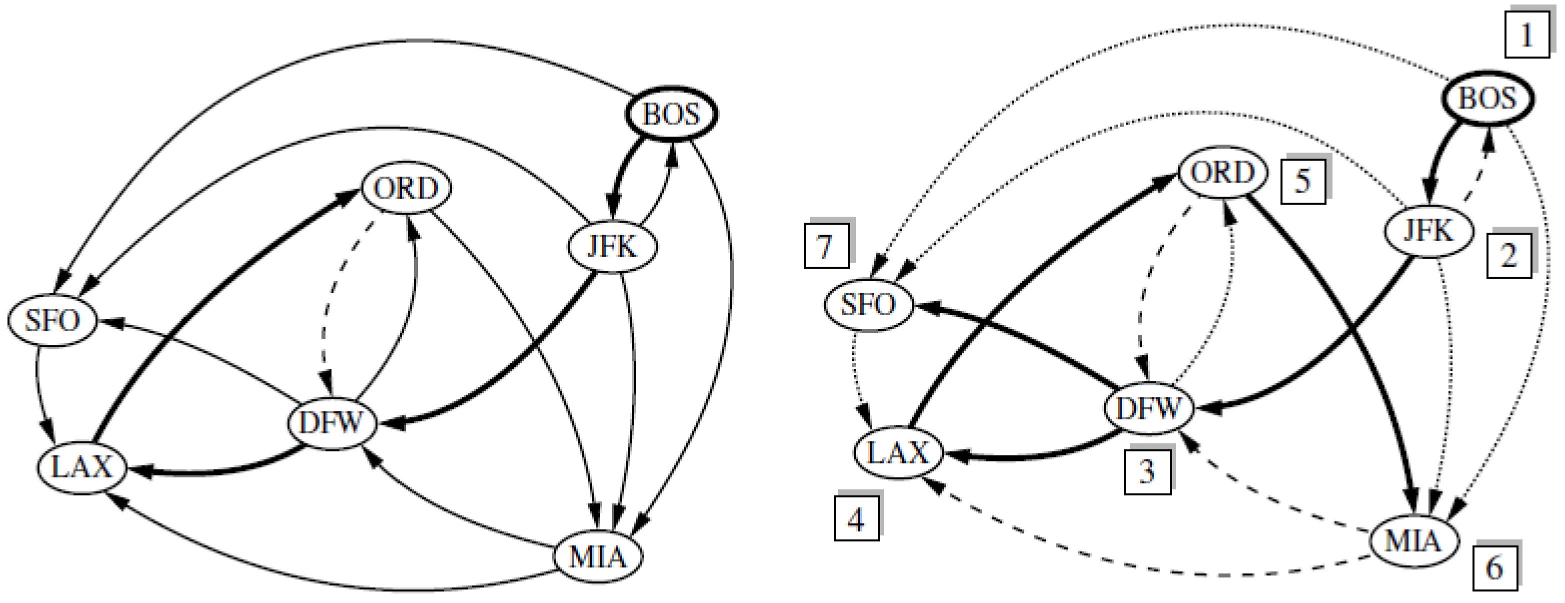


```
Algorithm DFS(G,u):            {We assume u has already been marked as visited}
   Input: A graph G and a vertex u of G
   Output: A collection of vertices reachable from u, with their discovery edges
   for each outgoing edge e = (u,v) of u do
      if vertex v has not been visited then
         Mark vertex v as visited (via edge e).
         Recursively call DFS(G,v).
```

DFS

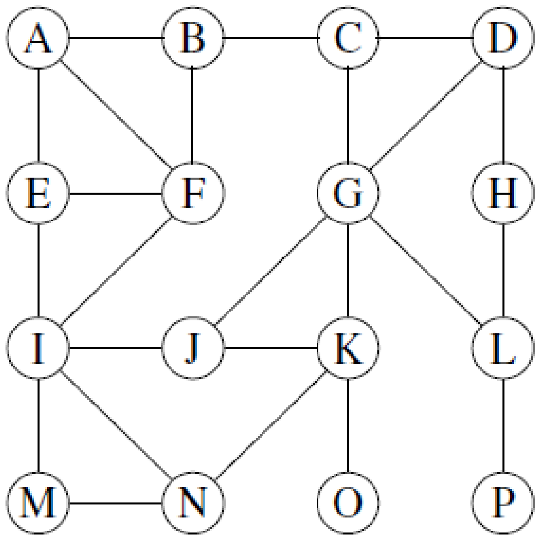
- **Depth First Search**
- Depth-first search tree: rooted at s
- When an edge d(u, v) is used to discover a new vertex v, the edge is known as a discovery edge (**tree edge**)
- **Nontree edges**
  - **Back edges**: connect a vertex to an ancestor in the DFS tree
  - **Forward edges**: connect a vertex to a descendant in the DFS tree
  - **Cross edges**: connect a vertex to a vertex that is neither its ancestor nor its descendant

DFS

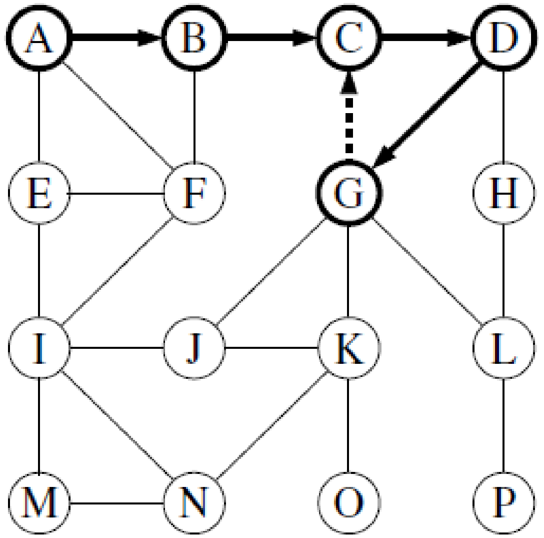


(a) (b)

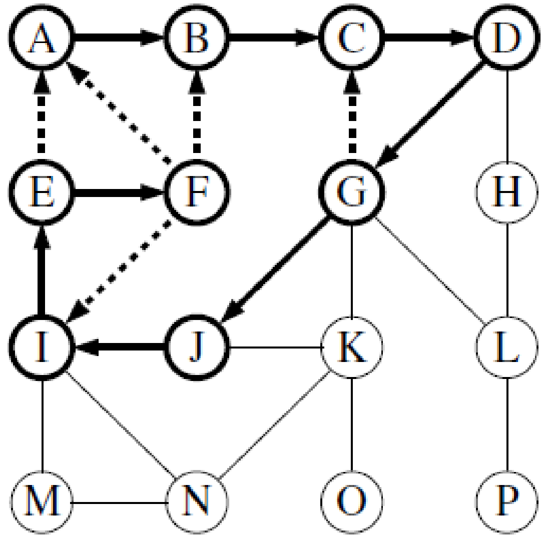
DFS


A B C D
E F G H
I J K L
M N O P

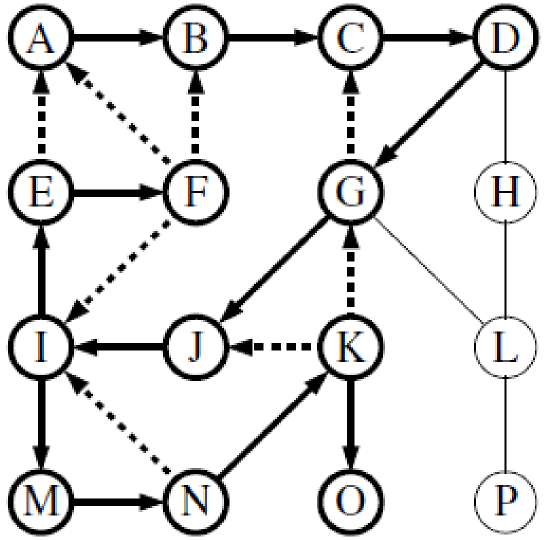
(a)


A B C D
E F G H
I J K L
M N O P

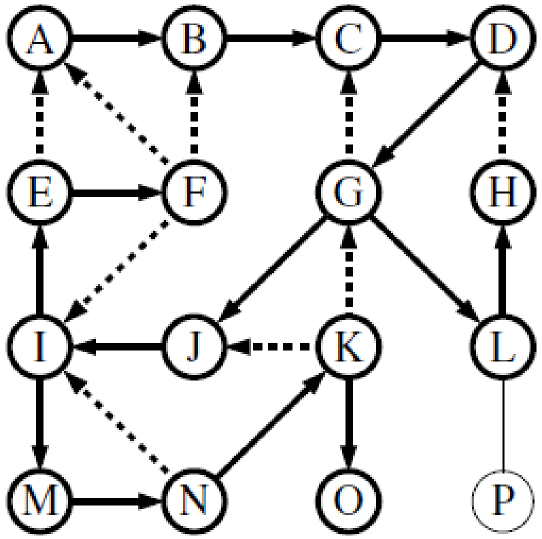
(b)


A B C D
E F G H
I J K L
M N O P

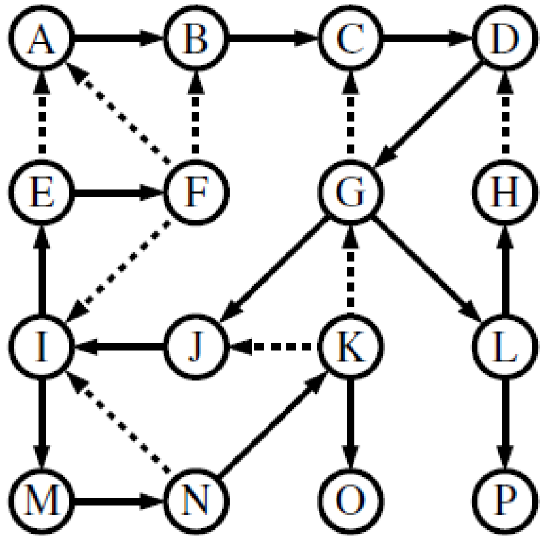
(c)


A B C D
E F G H
I J K L
M N O P

(d)


A B C D
E F G H
I J K L
M N O P

(e)


A B C D
E F G H
I J K L
M N O P

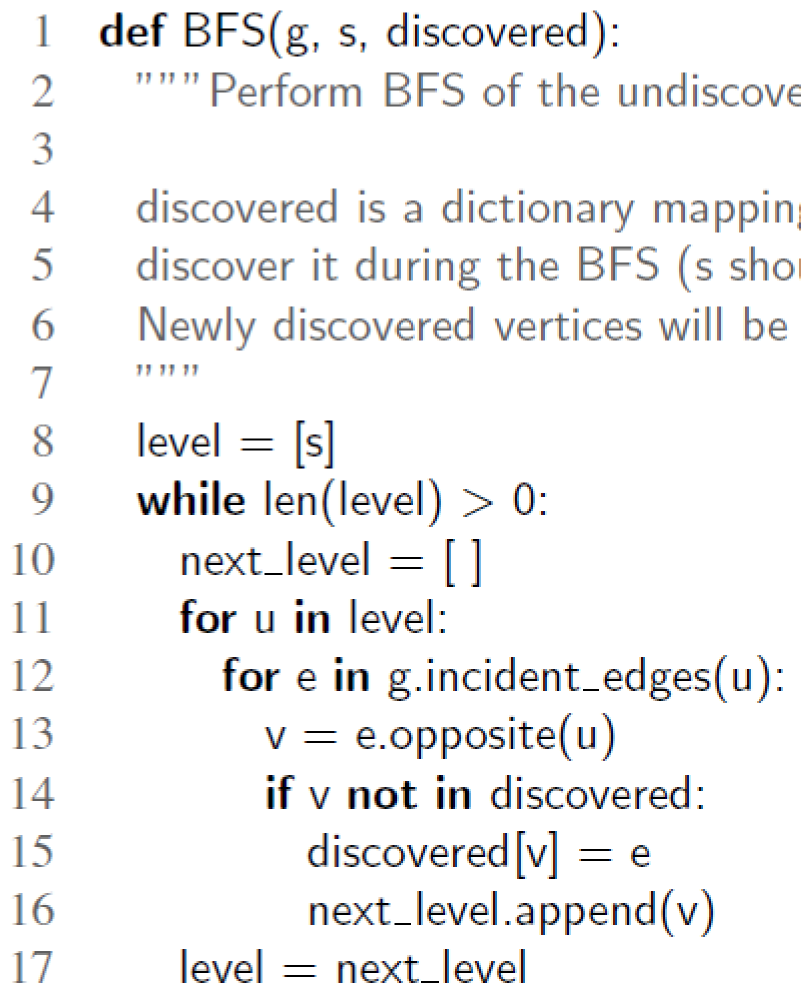
(f)

DFS

- **Depth First Search**
  - G = undirected graph, when a DFS is performed at a starting vertex s, the traversal visits all vertices in the connected component of s, and the discovery edges form a spanning tree of the connected component of s
  - G = directed graph, a DFS on G starting at vertex s visits all the vertices of G that are reachable from s. Also the DFS tree contains directed paths from s to very vertex reachable from s
- Running time
  - $O(n_s + m_s)$
    - n_s : # of vertices reachable from a vertex s
    - m_s : # of incident edges to vertices reachable from s

BFS

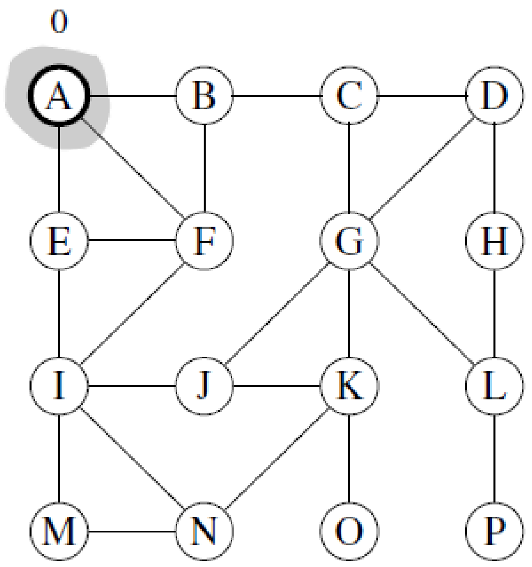
- **Breadth First Search**
  - Web crawling
  - Social network
  - Garbage collection
  - Rubic's cube
- Start from vertex s
- One 'level' at a time
- Level = all adjacent vertex from s,
  - If vertex u is unvisited, mark u as 'visited'
  - If vertex u is visited, discard
- Proceed to the next 'level'



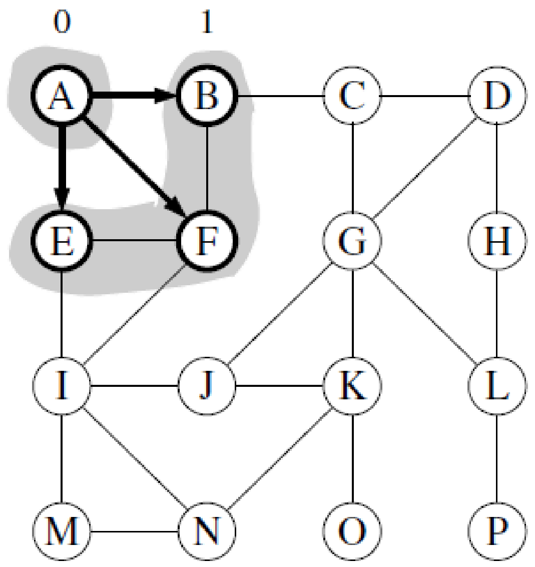
```
def BFS(g, s, discovered):
  """Perform BFS of the undiscove

  discovered is a dictionary mappin
  discover it during the BFS (s sho
  Newly discovered vertices will be
  """
  level = [s]
  while len(level) > 0:
    next_level = [ ]
    for u in level:
      for e in g.incident_edges(u):
        v = e.opposite(u)
        if v not in discovered:
          discovered[v] = e
          next_level.append(v)
    level = next_level
```

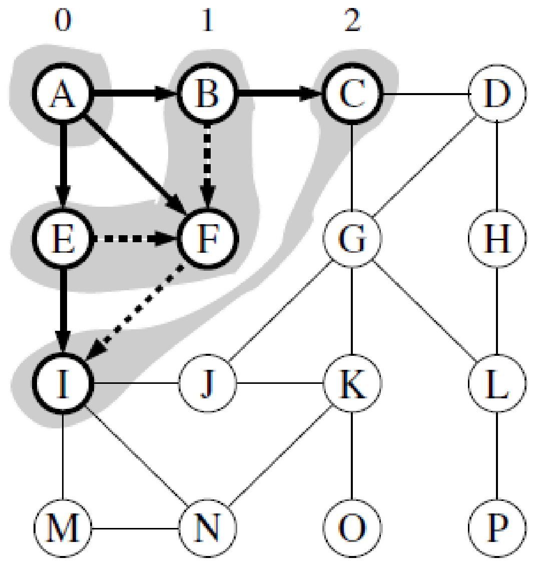
BFS


0
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

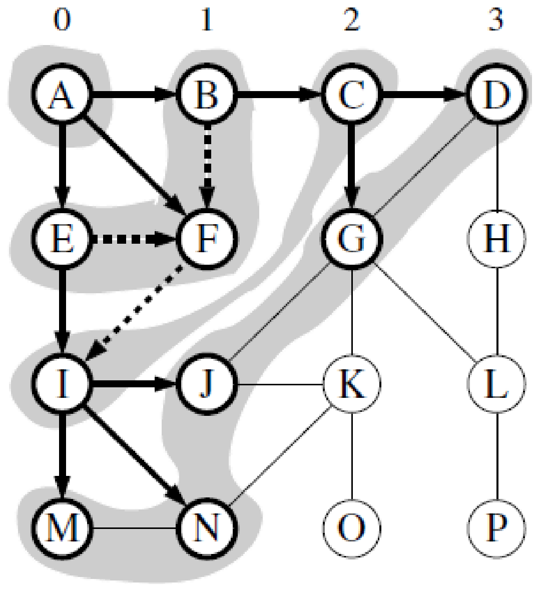
(a)


0
1
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

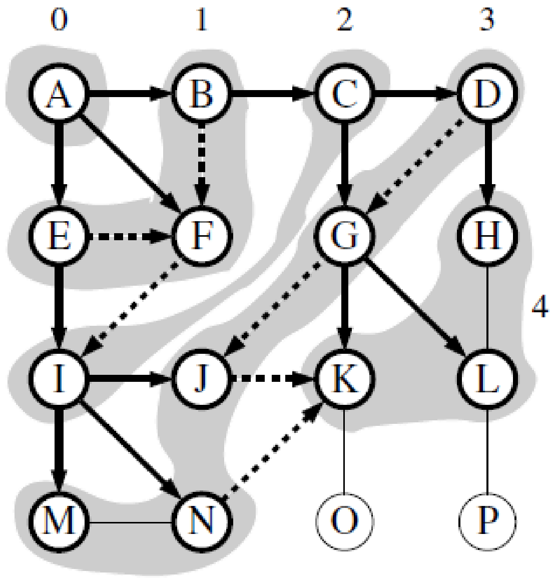
(b)


0
1
2
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

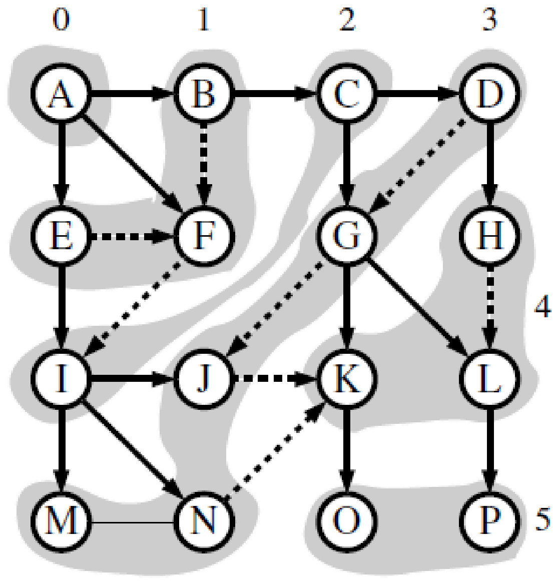
(c)


0
1
2
3
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

(d)


0
1
2
3
4
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

(e)


0
1
2
3
4
5
A
B
C
D
E
F
G
H
I
J
K
L
M
N
O
P

(f)

BFS

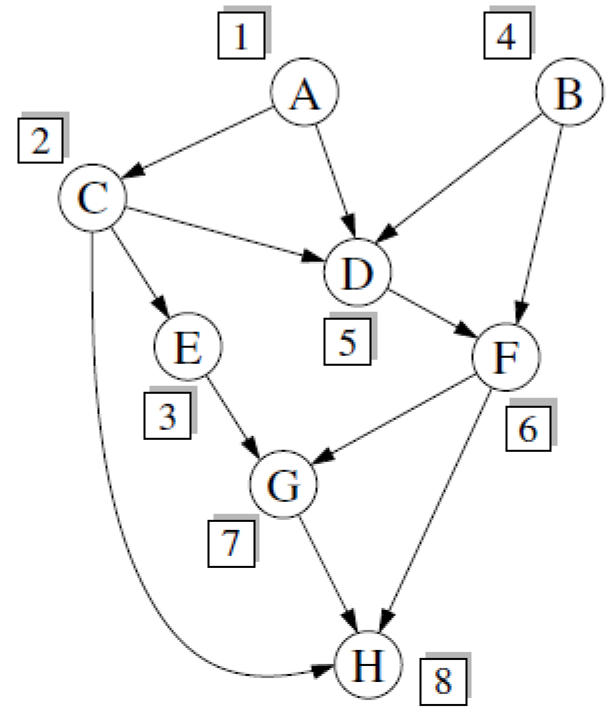
- **Nontree edges**
  - Undirected graph: all non tree edges are cross edges
  - Directed graph: back edges or cross edges
- Properties:
  - If G is a graph on which a BFS traversal starting at vertex s has been performed
    - The traversal visits all vertices of G that are reachable from s
    - For each vertex v at level i, the path of the BFS tree T between s and v has i edges and any other path of G from s to v has at least i edges
    - If (u, v) is an edge that is not in the BFS tree, then the level number of v can be at most 1 greater than the leve number of u
  - If G is a graph with n vertices and m edges, BFS traversal of G takes O(n+m) time

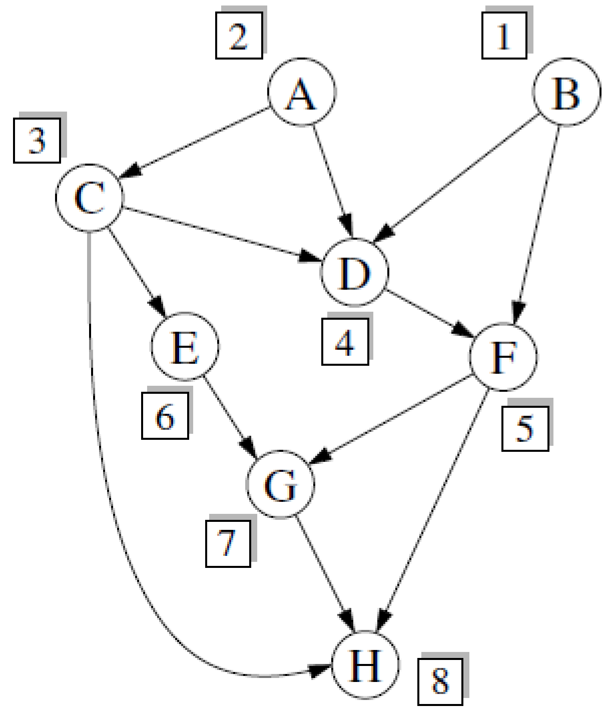
TRANSITIVE CLOSURE (传递闭包)

- Reachability problem in a directed graph
  - DFS or BFS: O(n + m) time
- In certain applications, we want to answer many reachability queries more efficiently
- Transitive closure of a directed graph G
  - Is itself a directed graph G', such that
  - The vertices of G' are the same as the vertices of G
  - G' has an edge (u, v), when ever G has a directed path from u to v
- Computation of transitive closure O(n(n+m))
  - N DFSs
- Floyd-Warshall: O(n^3) – not covered in this course

DIRECTED ACYCLIC GRAPHS

- Directed Acyclic Graph (DAG)
  - Directed graphs without directed cycles
- Applications
  - Prerequisites between courses of a degree program
  - Inheritance between classes of an object-oriented program
  - Scheduling constraints between the tasks of a project
- Topological Ordering
  - If G is a directed graph with n vertices
  - Topological ordering of G: an ordering v1, v2, …, vn of vertices of G such that for every edge (vi, vj) of G, it is the case that i < j


(a)

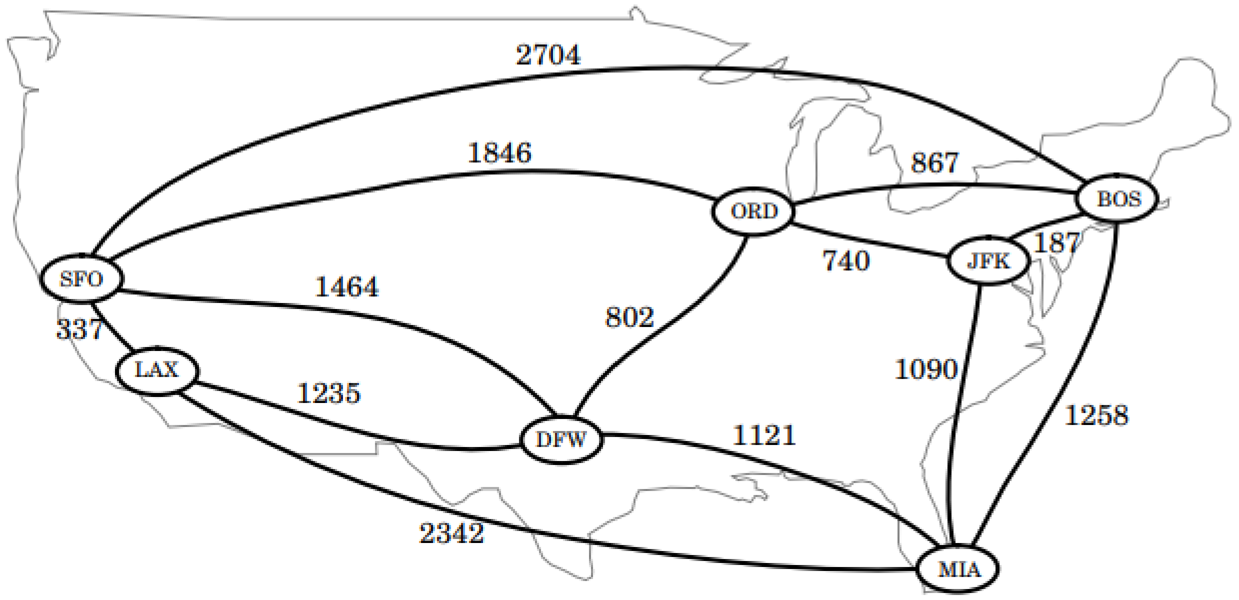

(b)

```
def topological_sort(g):
  """Return a list of verticies of directed acy

  If graph g has a cycle, the result will be inc
  """
  topo = [ ]              # a list of vertices
  ready = [ ]             # list of vertices th
  incount = { }           # keep track of in-
  for u in g.vertices( ):
    incount[u] = g.degree(u, False)      # pa
    if incount[u] == 0:                  # if
      ready.append(u)                    # it
  while len(ready) > 0:
    u = ready.pop( )                     # u
    topo.append(u)                       # ac
    for e in g.incident_edges(u):        # co
      v = e.opposite(u)
      incount[v] −= 1                    # v
      if incount[v] == 0:
        ready.append(v)
  return topo
```

THIS LECTURE: SHORTEST PATHS

- Shortest paths, why?
  - Map as a graph, to find the shortest route to destination
  - Computer network as a graph, minimal number of routing possible
- Graphs so far
  - Set of Vertices: V
  - Collection of edges: E
- To find the shortest path, edges need to be weighted
  - e = (u, v)
  - w(u, v) = w(e)

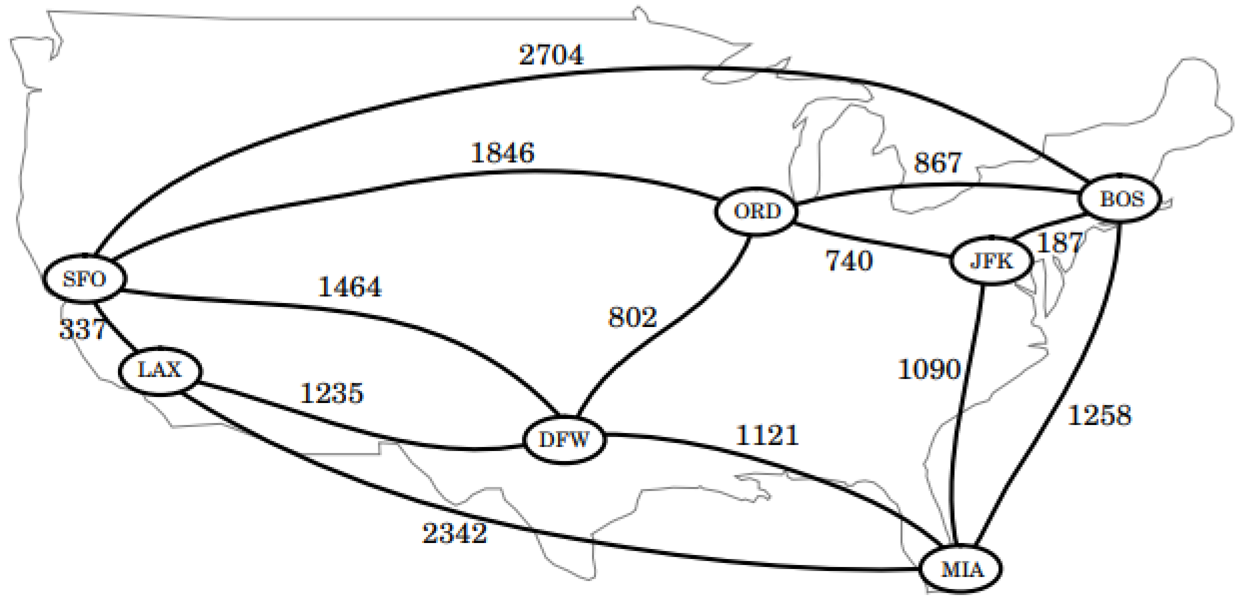


SHORTEST PATHS

- Shortest Path in a weighted graph
  - If P = $((v_0, v_1),(v_1, v_2), \ldots, (v_{k-1}, v_k))$
  - Length of P, w(P) is

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

  - Distance from u to v d(u, v) is the length of a shortest path from u to v (if it exists)
- d(u, v) = ∞ if there is no path from u to v
- For now we consider w(u, v) = non-negative

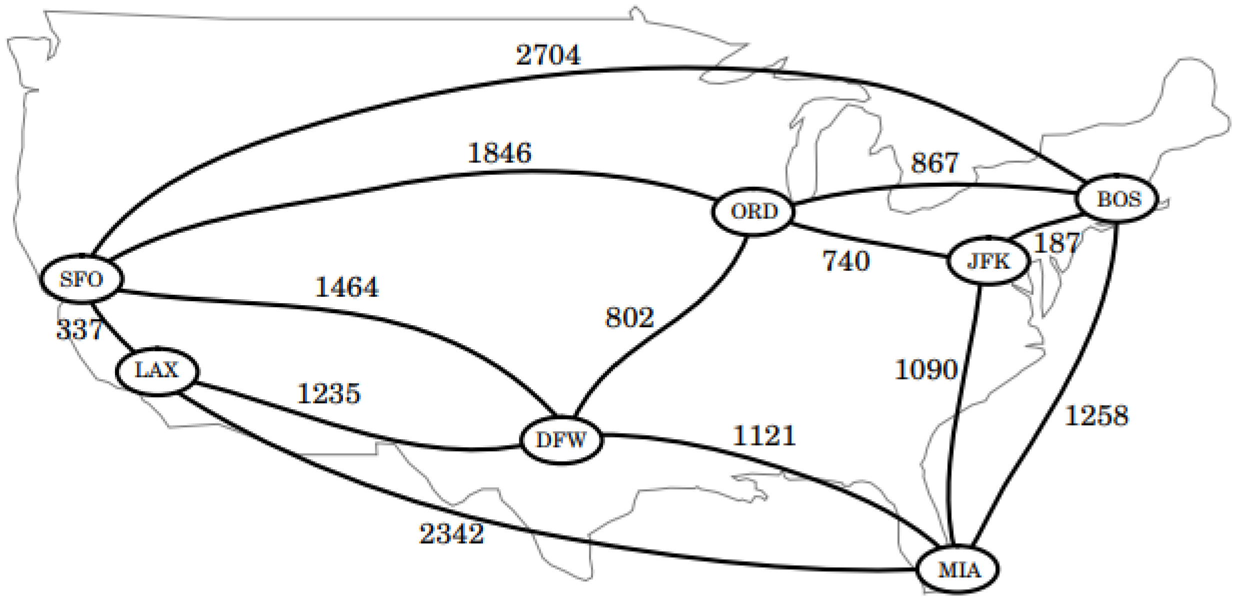


DIJKSTRA'S SHORTEST PATH ALGORITHM

- BFS at the start vertex s
- Edge relaxation
  - Label D[v] for each vertex v in V
  - D[s] = 0 and D[v] = ∞
  - Breadth first search
    - Update D[v] for each v in the search level

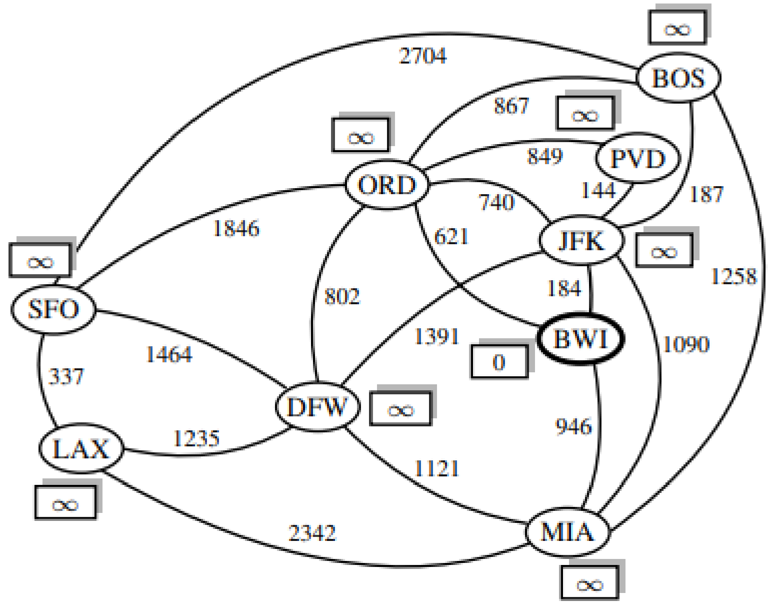
Edge Relaxation:

```
if $D[u] + w(u,v) < D[v]$ then
    $D[v] = D[u] + w(u,v)$
```



DIJKSTRA'S SHORTEST PATH ALGORITHM

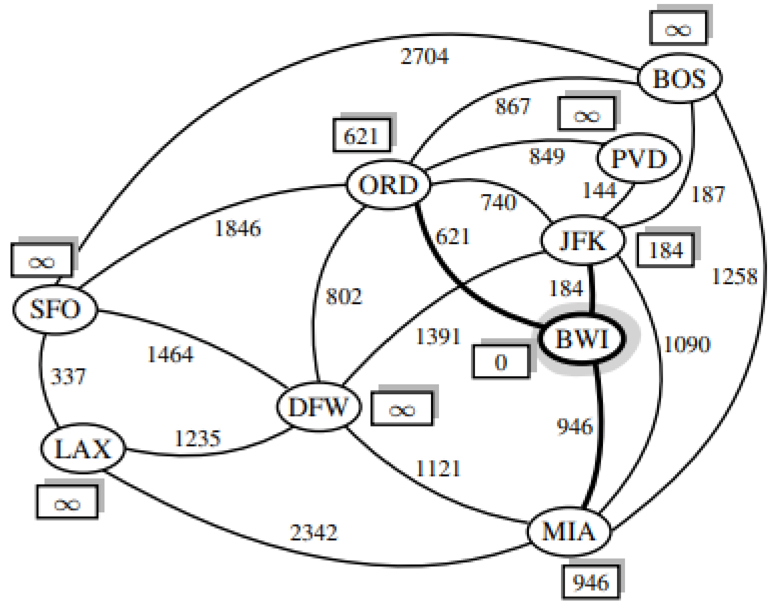
```
Algorithm ShortestPath($G, s$):
    Input: A weighted graph $G$ with nonnegative edge weights, and a distinguished
      vertex $s$ of $G$.
    Output: The length of a shortest path from $s$ to $v$ for each vertex $v$ of $G$.
    Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
    Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
    while $Q$ is not empty do
      {pull a new vertex $u$ into the cloud}
      $u$ = value returned by $Q$.remove_min()
      for each vertex $v$ adjacent to $u$ such that $v$ is in $Q$ do
        {perform the relaxation procedure on edge $(u, v)$}
        if $D[u] + w(u, v) < D[v]$ then
          $D[v] = D[u] + w(u, v)$
          Change to $D[v]$ the key of vertex $v$ in $Q$.
    return the label $D[v]$ of each vertex $v$
```



(a)

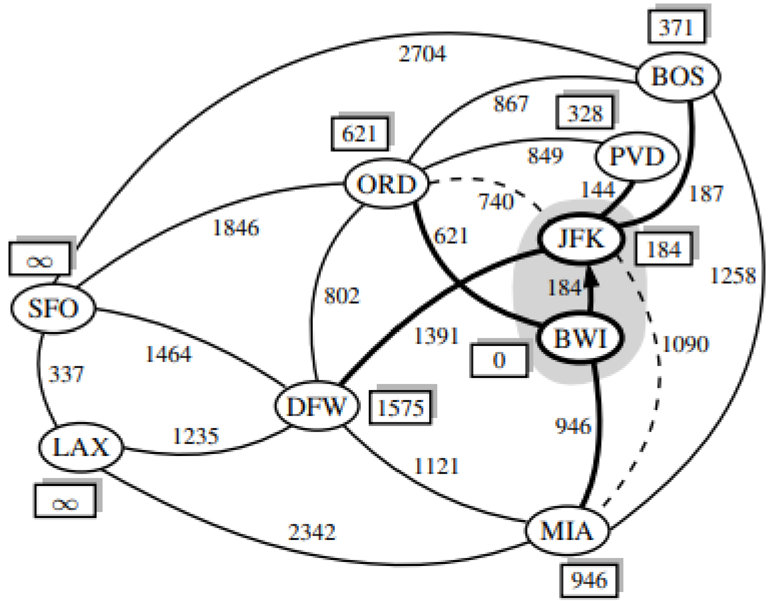
DIJKSTRA'S SHORTEST PATH ALGORITHM

```
Algorithm ShortestPath($G, s$):
    Input: A weighted graph $G$ with nonnegative edge weights, and a distinguished
        vertex $s$ of $G$.
    Output: The length of a shortest path from $s$ to $v$ for each vertex $v$ of $G$.
    Initialize $D[s] = 0$ and $D[v] = \infty$ for each vertex $v \neq s$.
    Let a priority queue $Q$ contain all the vertices of $G$ using the $D$ labels as keys.
    while $Q$ is not empty do
        {pull a new vertex $u$ into the cloud}
        $u$ = value returned by $Q$.remove_min()
        for each vertex $v$ adjacent to $u$ such that $v$ is in $Q$ do
            {perform the relaxation procedure on edge $(u, v)$}
            if $D[u] + w(u, v) < D[v]$ then
                $D[v] = D[u] + w(u, v)$
                Change to $D[v]$ the key of vertex $v$ in $Q$.
    return the label $D[v]$ of each vertex $v$
```

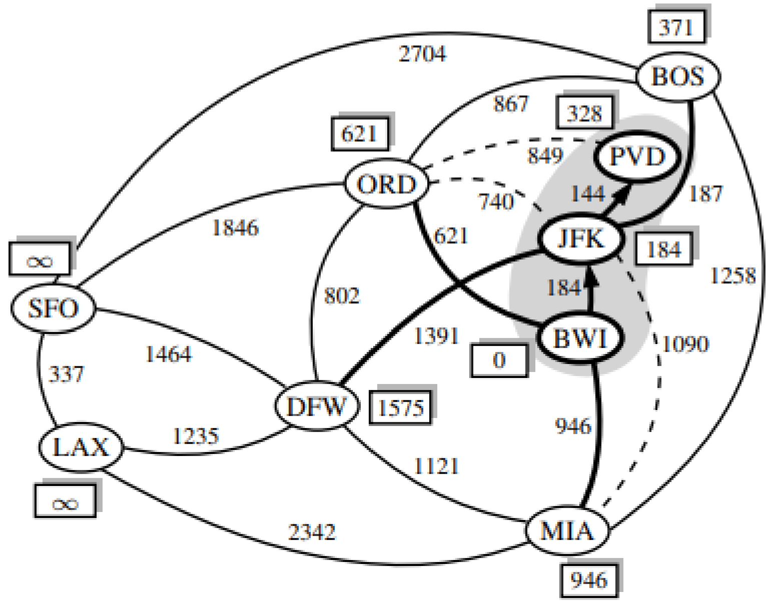


(b)

DIJKSTRA'S SHORTEST PATH ALGORITHM

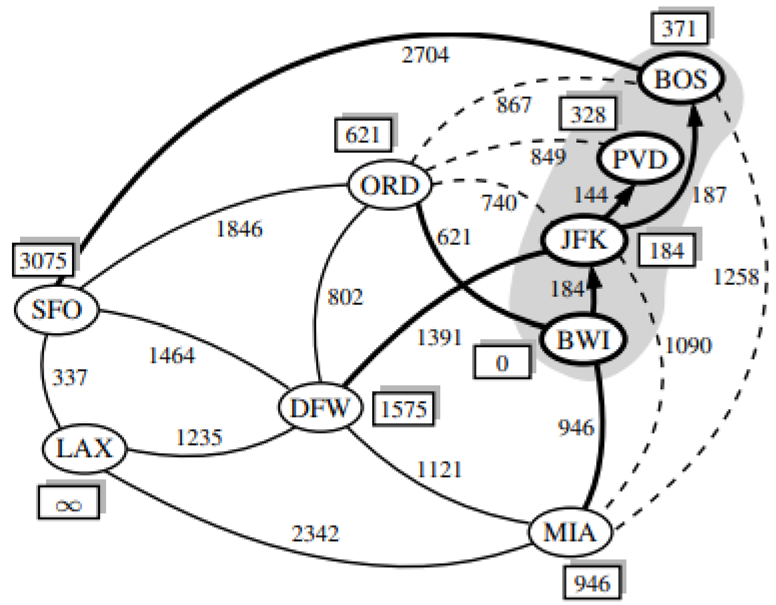


(c)

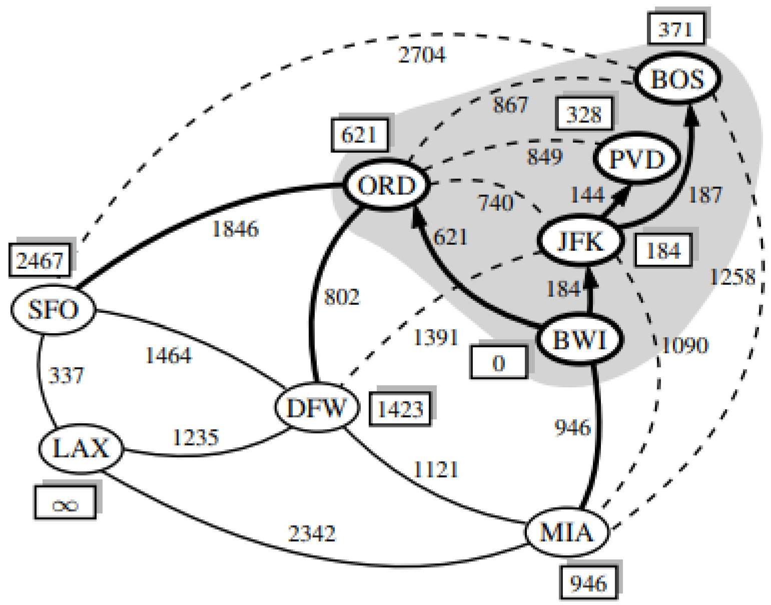


(d)

DIJKSTRA'S SHORTEST PATH ALGORITHM

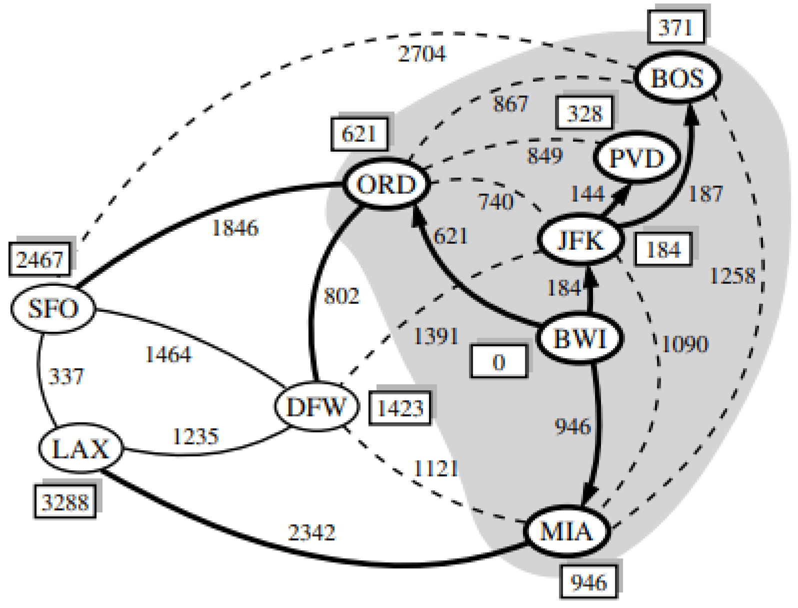


(e)

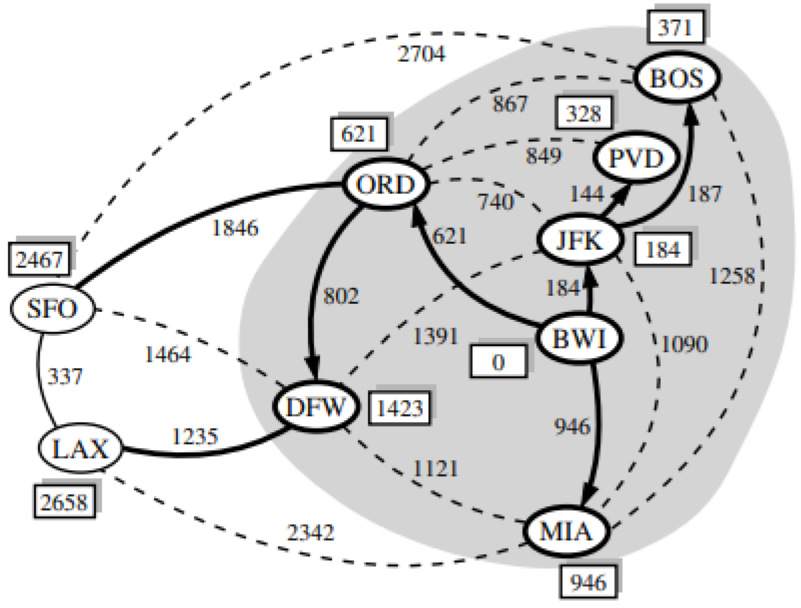


(f)

DIJKSTRA'S SHORTEST PATH ALGORITHM

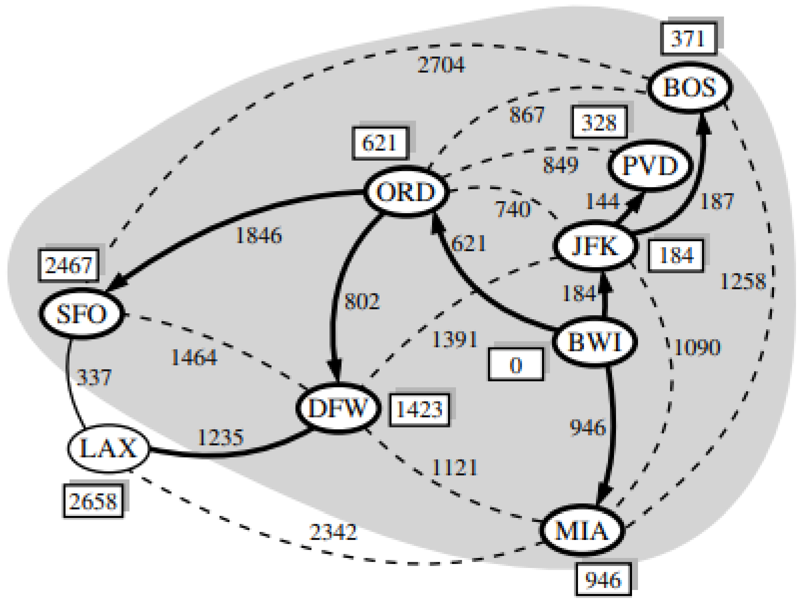


(g)

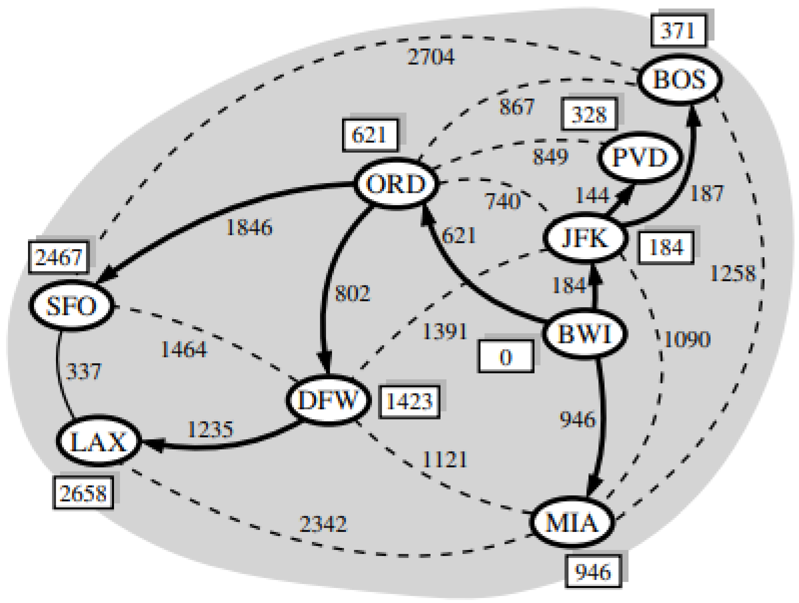


(h)

DIJKSTRA'S SHORTEST PATH ALGORITHM



(i)



(j)

DIJKSTRA'S SHORTEST PATH ALGORITHM

- Why it works
  - Whenever a vertex u is visited, its label D[u] stores the correct length of a shortest path from v to u
  - When the algorithm terminates, it computes the shortest path from s to every vertex of G
- Running time
  - Assumption: adjacency list or adjacency map for G
  - O((V+E)log V)
- Limitation
  - Does not work on negative weights, why?

BELLMAN-FORD

- Bellman_ford(G,w,s):

```
Initialise()
for i = 1 to |V| -1:
    for each edge (u, v) in E
            relax(u, v, w)
For each edge (u, v) in E
    if d[v] > d[u] + w(u, v)
            there is negative cycle
```

FINAL EXAM：题型

- 判断题：7题14分
  - 只需回答对错，不用证明
- 填空题：5题10分
  - 中英文均可
- 单选题：5题10分
- 多选题：8题16分
  - 少选得1分，选错得0分
- 简答题：5题50分
  - 算法：伪代码，注意格式，注意注释

FINAL EXAM: 复习范围

- 算法渐进性分析
  - 递归算法的分析
- 数组
- 双向链表
- 哈希表与哈希函数
  - 哈希函数实现（hashing与compressing）
  - 哈希冲突的解决办法
- 优先队列
  - 堆：插入；从底向上构建的方法

FINAL EXAM: 复习范围

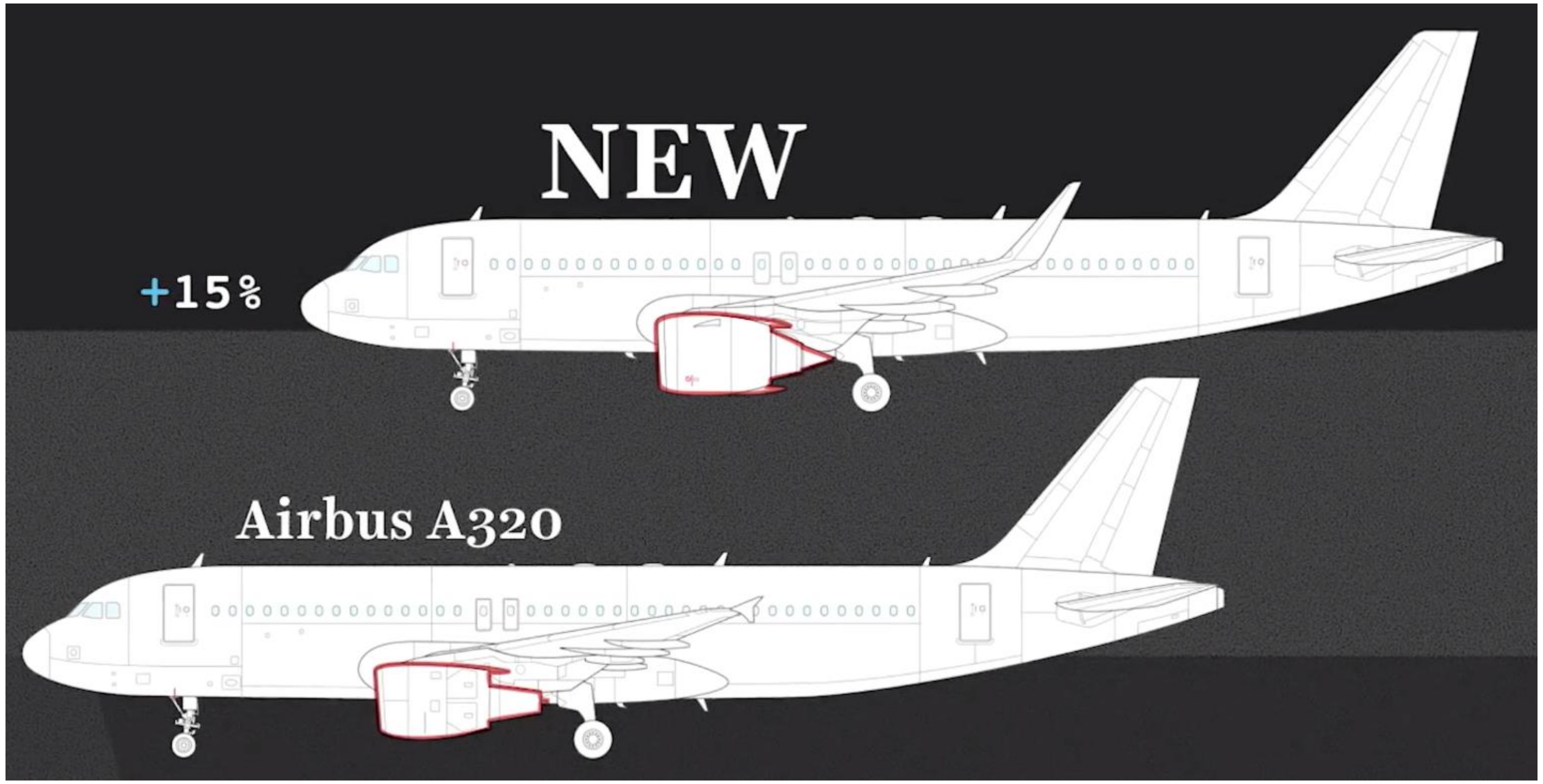
- 搜索树
  - AVL树：执行操作的效率；旋转
  - 红黑树：红黑树保持平衡的属性；在红黑树插入元素的操作
- 排序算法
  - 分治相关的排序
  - 线性排序算法
- 图
  - 表示图的数据结构
  - 有向图与无向图的DFS和BFS算法
    - 以上算法中的边的分类问题

研究方向
系统安全

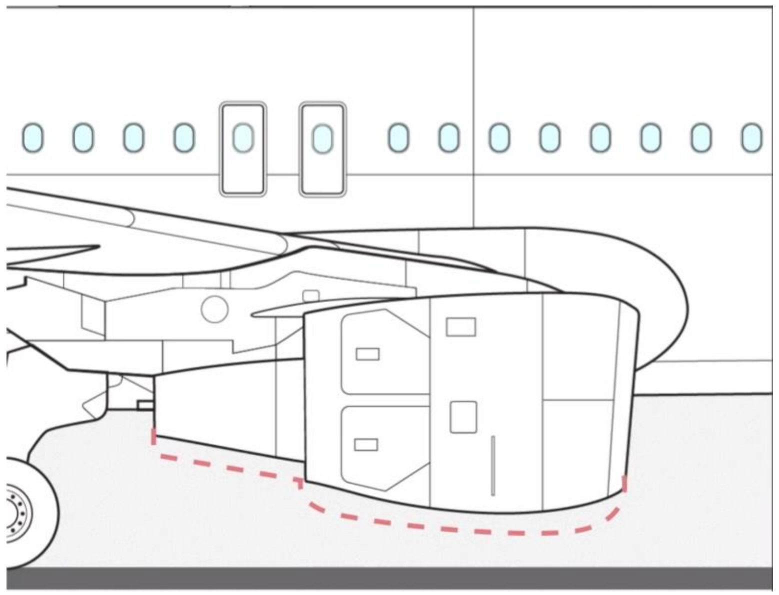
- 2019年03月10日, 埃塞俄比亚航空 ET302 航班在起飞6分钟后坠落，机上157人全部遇难。
- 2018年10月29日, 印尼狮航 610 航班在起飞13分钟后坠入爪哇海，机上189人全部遇难。

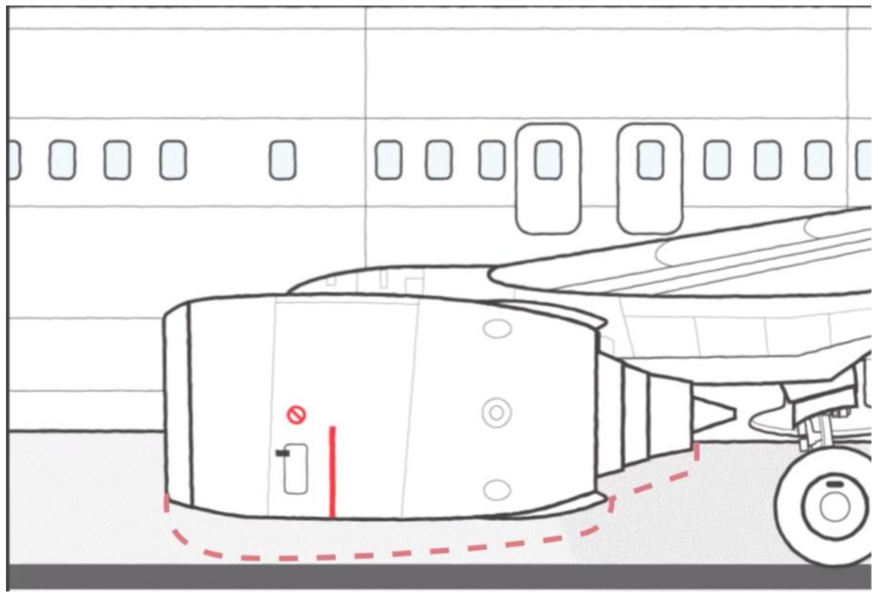
- 均疑似因迎角传感器故障，以及机动特性增强系统(MCAS)反应过度。

研究方向
系统安全

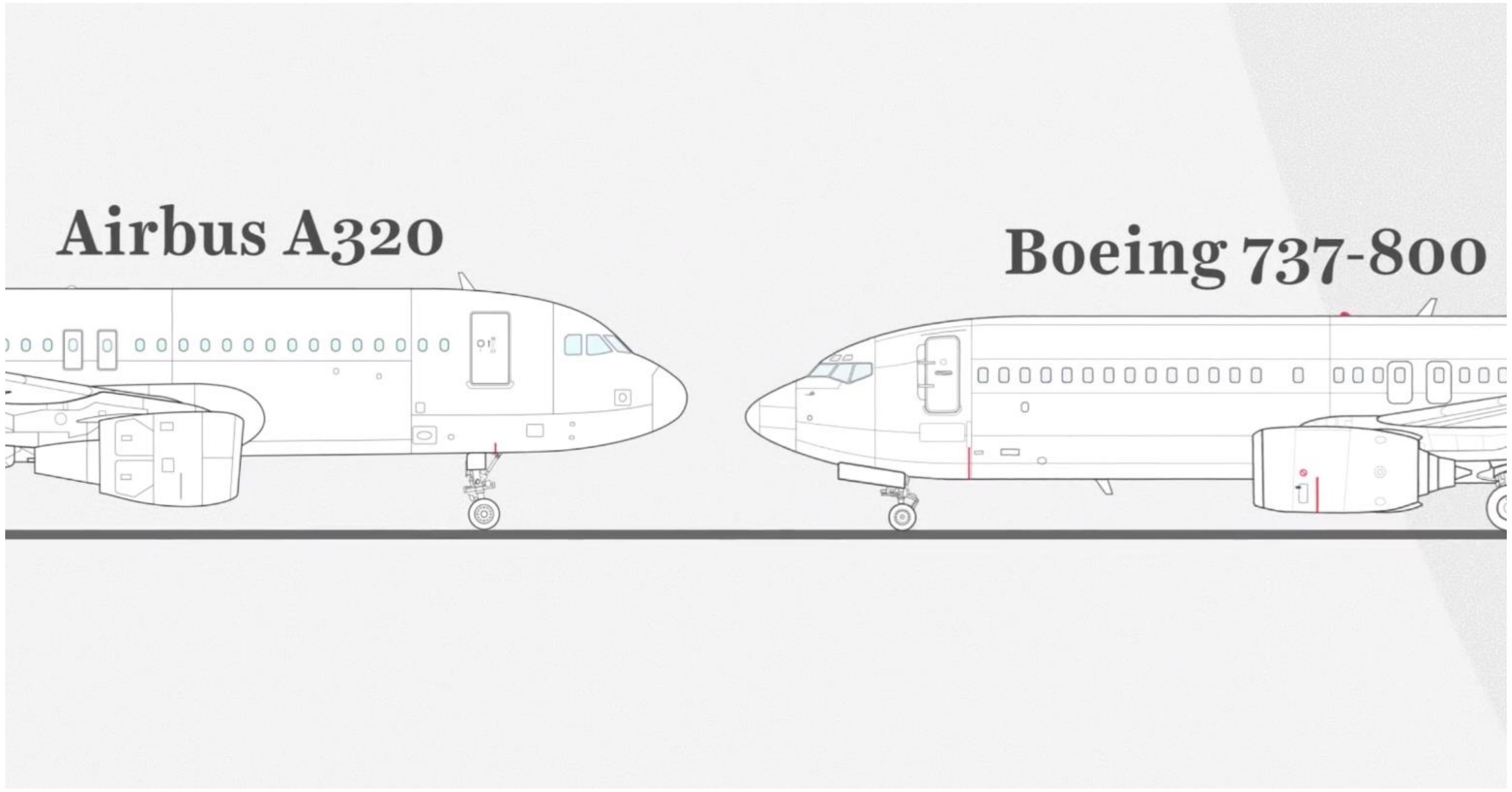


研究方向
系统安全

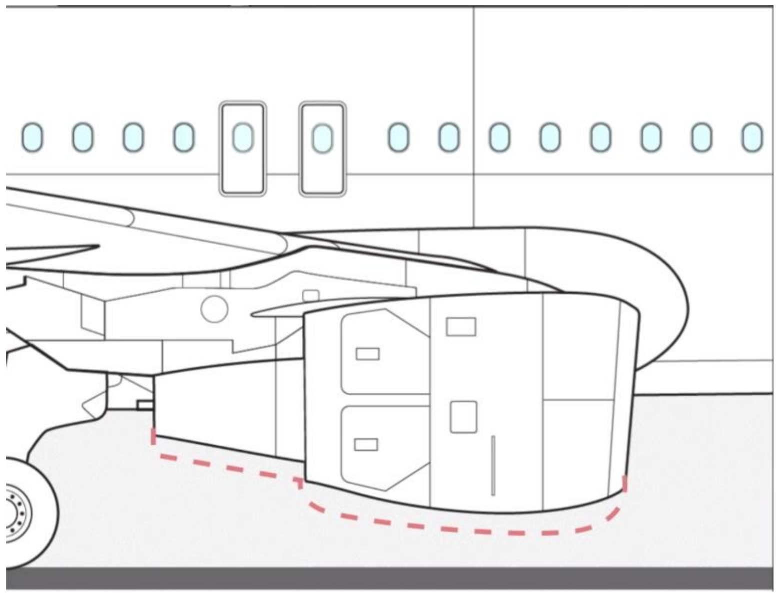


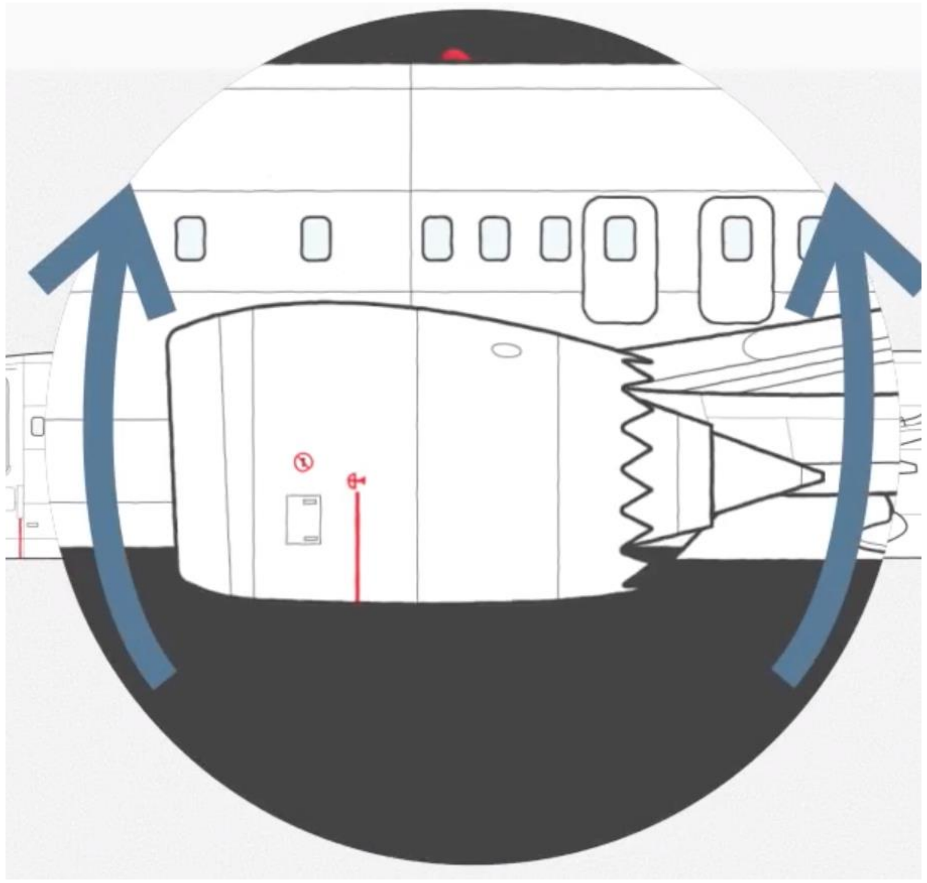


研究方向
系统安全

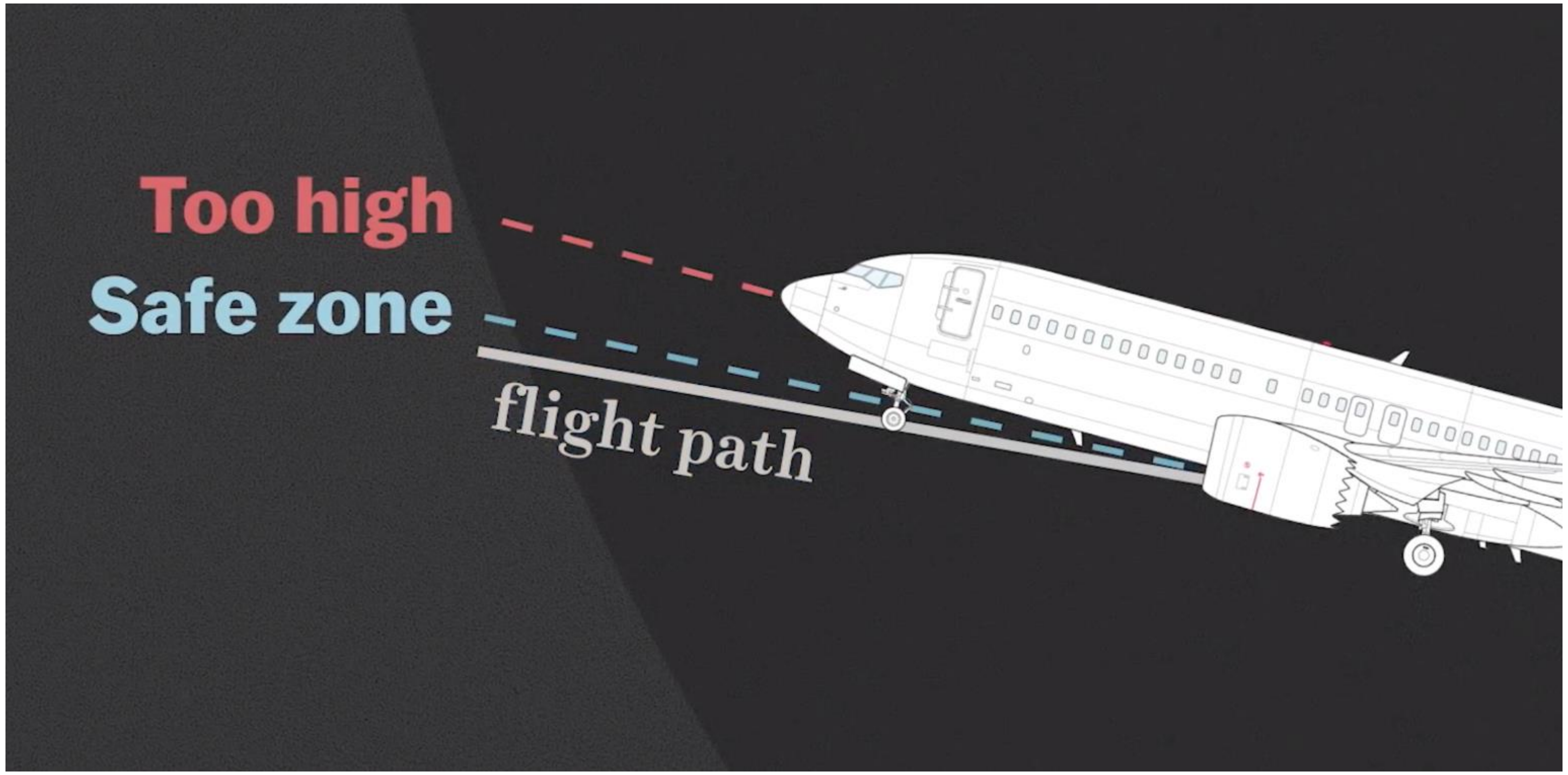


研究方向
系统安全



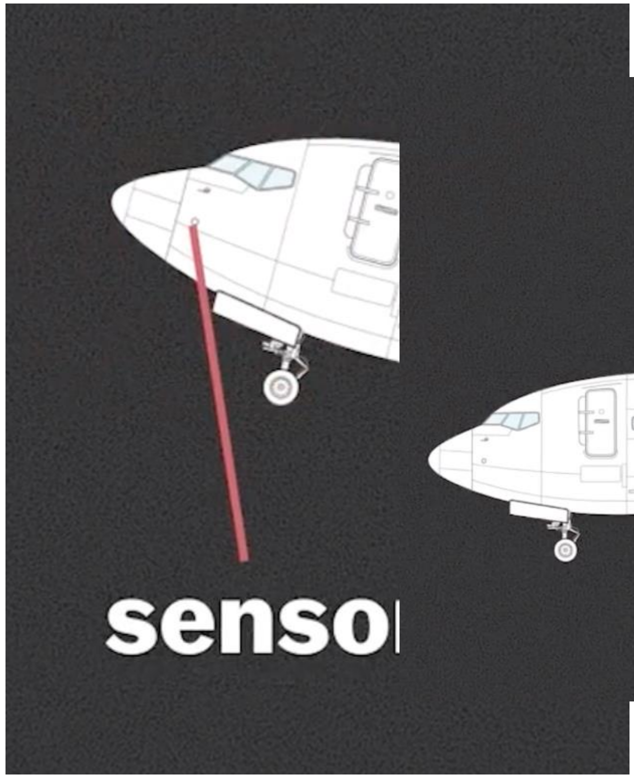


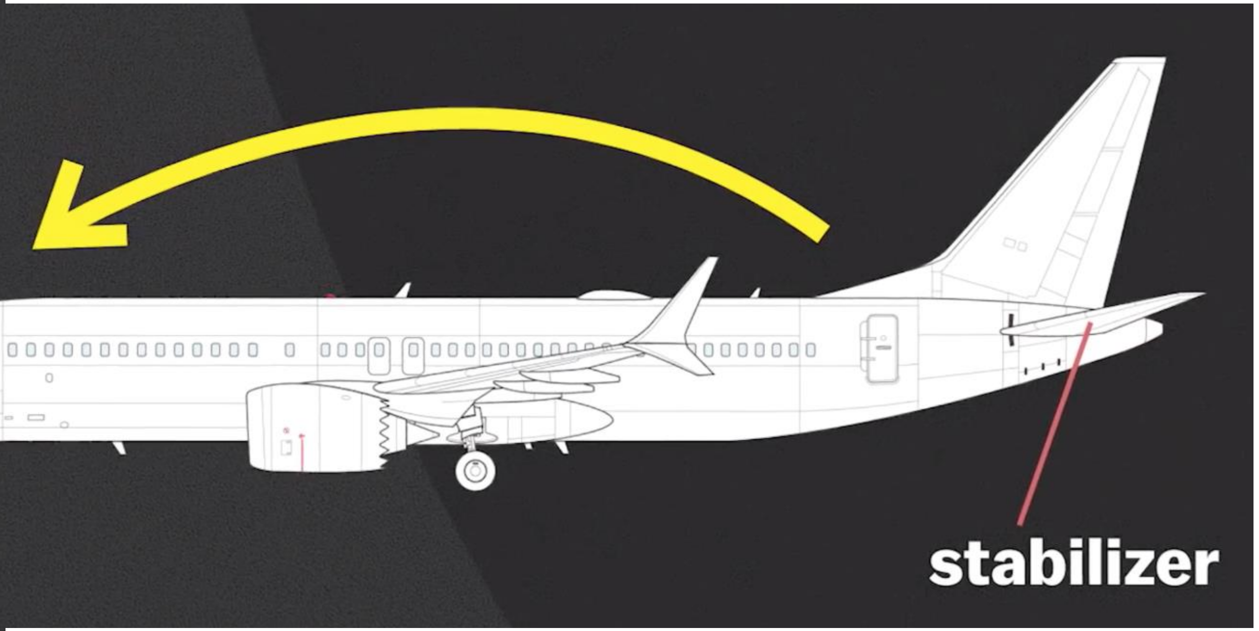
研究方向
系统安全



研究方向
系统安全

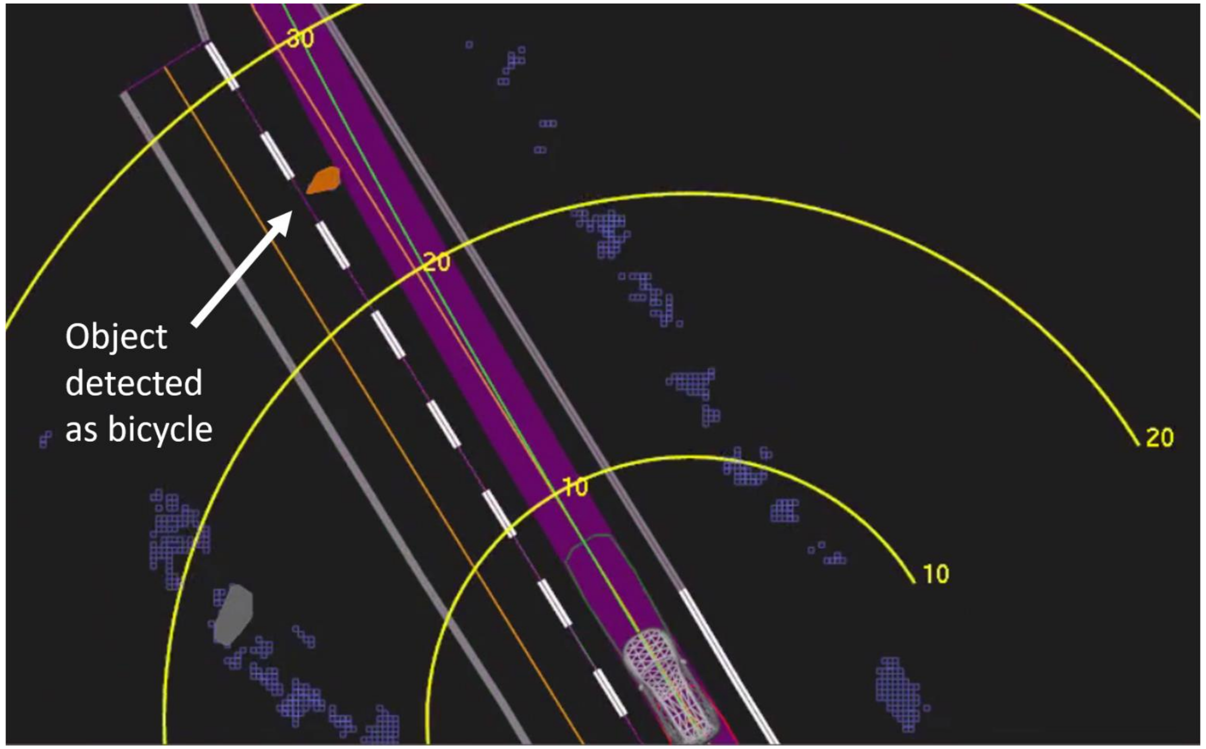
Maneuvering Characteristics Augmentation System
MCAS





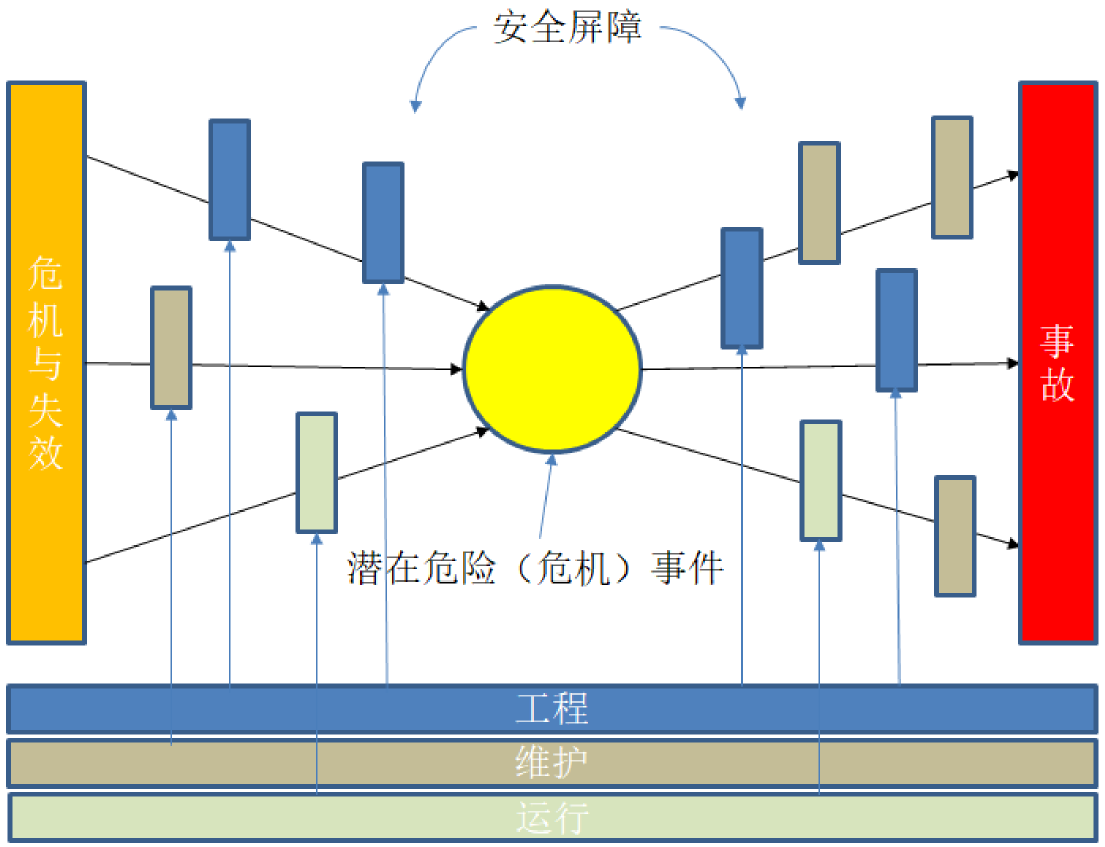
研究方向
系统安全

- 特斯拉 – 2016年01月20日，在自动驾驶模式下撞向一辆静止状态的卡车，司机在事故中身亡。
- 特斯拉 – 2016年05月07日，传感器没有正确识别出一辆大型18轮卡车, 司机在事故中身亡。
- 特斯拉 – 2018年03月23日，车辆未正确识别马路标识，司机在事故中身亡。
- 优步 – 2018年03月18日，车辆在测试过程中撞向一名横穿马路的行人，并导致其身亡。此次事故主要由于车辆未在规定时间内及时制动以及相关人为因素导致。

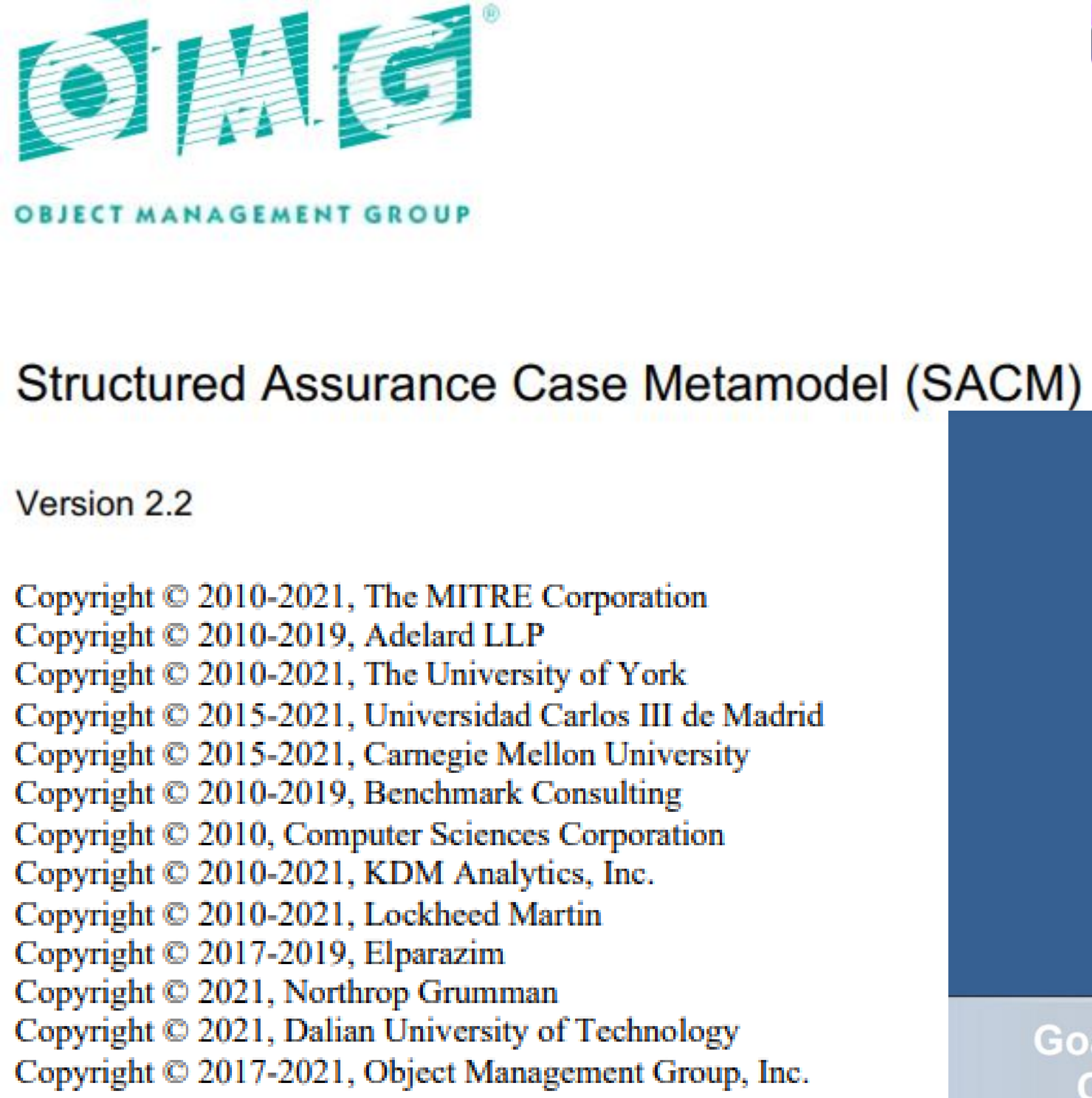


研究方向
系统安全

- 安全性：系统所具有的不导致人员伤亡、系统损坏、重大财产损失或不危机人员健康和环境的能力。
- 可靠性：系统再规定条件下和规定时间内完成规定功能的能力，以概率表示。
- 军工“六性”之二



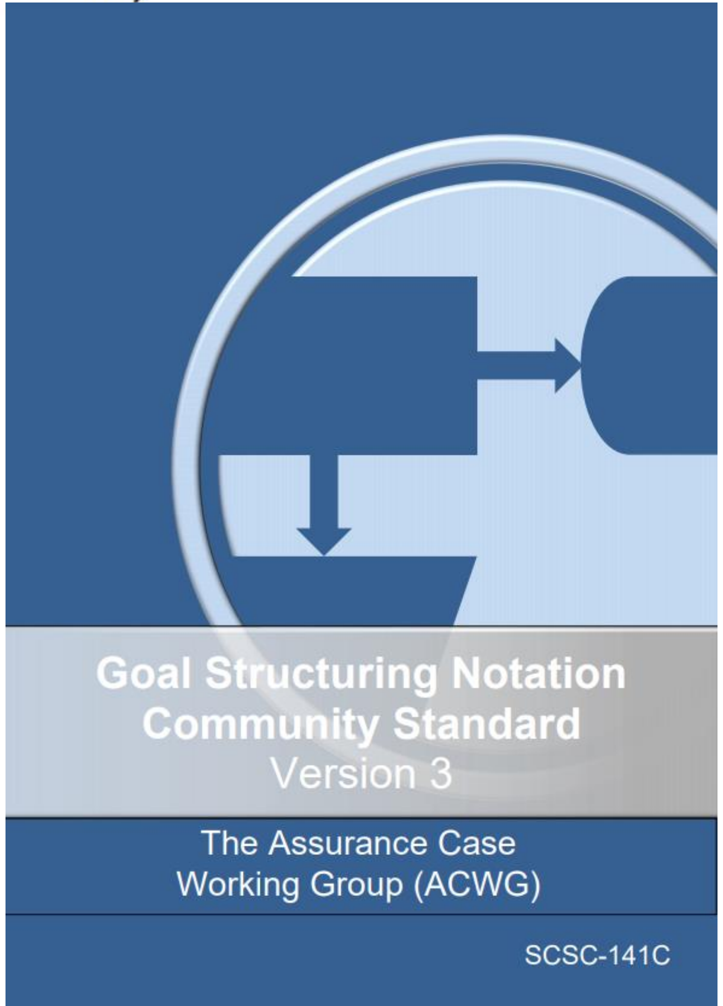
研究方向
系统安全



OBJECT MANAGEMENT GROUP

Structured Assurance Case Metamodel (SACM)

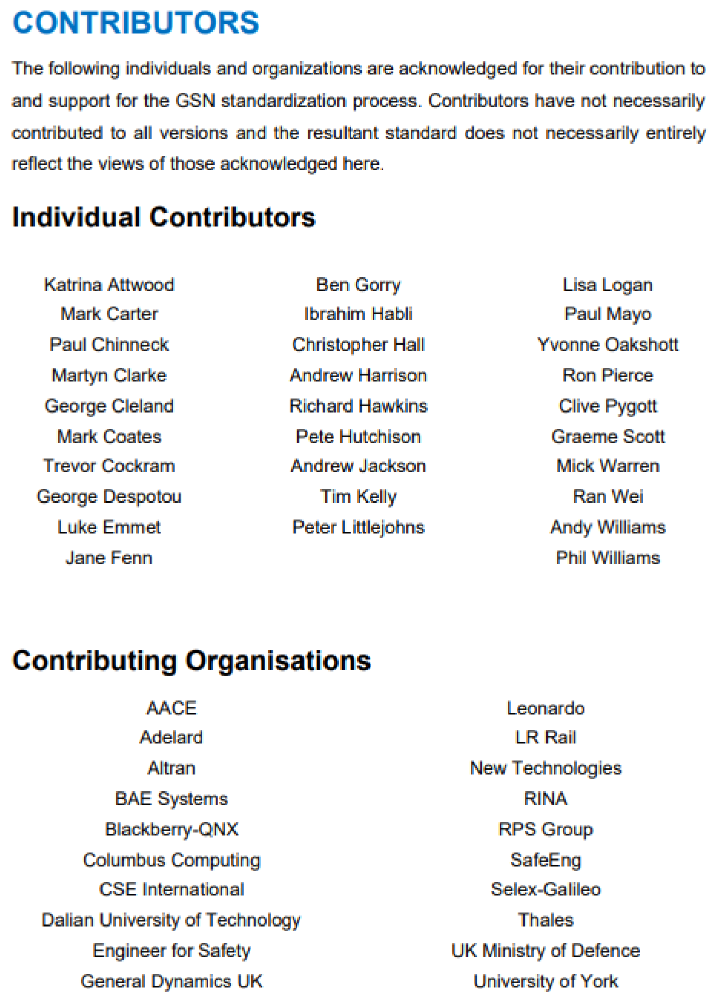
Version 2.2



Goal Structuring Notation Community Standard
Version 3

The Assurance Case Working Group (ACWG)

SCSC-141C



CONTRIBUTORS

The following individuals and organizations are acknowledged for their contribution to and support for the GSN standardization process. Contributors have not necessarily contributed to all versions and the resultant standard does not necessarily entirely reflect the views of those acknowledged here.

Individual Contributors

Katrina Attwood	Ben Gorry	Lisa Logan
Mark Carter	Ibrahim Habli	Paul Mayo
Paul Chinneck	Christopher Hall	Yvonne Oakshott
Martyn Clarke	Andrew Harrison	Ron Pierce
George Cleland	Richard Hawkins	Clive Pygott
Mark Coates	Pete Hutchison	Graeme Scott
Trevor Cockram	Andrew Jackson	Mick Warren
George Despotou	Tim Kelly	Ran Wei
Luke Emmet	Peter Littlejohns	Andy Williams
Jane Fenn		Phil Williams

Contributing Organisations

AACE	Leonardo
Adelard	LR Rail
Altran	New Technologies
BAE Systems	RINA
Blackberry-QNX	RPS Group
Columbus Computing	SafeEng
CSE International	Selex-Galileo
Dalian University of Technology	Thales
Engineer for Safety	UK Ministry of Defence
General Dynamics UK	University of York

研究方向
系统安全

- 研究内容
  - 模型化安全保障
  - 实时嵌入式系统/操作系统
- 适用行业
  - 机器人与自主系统（Robotics and Autonomous Systems）
  - 自动驾驶
  - 开放自适应信息物理系统
  - 军工系统安全
  - 航天系统安全

研究方向
高精度电磁计量

- 计量级高精度电磁仪器
  - 电压、电流信号调理
  - 标准电流信号发生器
  - 功率放大器
  - ．．．
- 解决“卡脖子”问题
- 应用
  - 科学计量
  - 航空航天
  - 军工、国防

THANKS

See you in the next session!