UNIT 5: GRAPHS

<u>L13. BREADTH-FIRST SEARCH (BFS)</u>

GRAPH REPRESENTATIONS: (DATA STRUCTURES)

Adjacency lists

Array Adj of $|V|$ linked lists

- for each vertex $u \in V, \mathrm{Adj}[u]$ stores u 's neighbors, i.e., $\{v \in V \mid (u, v) \in E\}. \quad (u, v)$ are just outgoing edges if directed.
- in Python: $Adj =$ dictionary of list/set values; vertex $=$ any hashable object (e.g., int, tuple)
- advantage: <u>**multiple graphs on same vertices**</u>

Implicit Graphs:

$\mathrm{Adj}(u)$ is a function - compute local structure on the fly (e.g., Rubik's Cube).

- <u>**This requires "Zero" Space.**</u>

Object-oriented Variations:

- object for each vertex u
- u.neighbors $=$ list of neighbors i.e. $\mathrm{Adj}[u]$

BREADTH-FIRST SEARCH

Explore graph level by level from s

- level $0 = \{s\}$
- level $i =$ vertices reachable by path of i edges but not fewer
- build level $i > 0$ from level $i - 1$ by trying all outgoing edges, but ignoring vertices from previous levels

BREADTH-FIRST-SEARCH ALGORITHM

```
BFS (V,Adj,s):    # See CLRS for queue-based implementation
  level = { s: 0 }
  parent = {s : None }
  i = 1
  frontier = [s]    # previous level, i − 1
  while frontier:
    next = [ ]    # next level, i
    for u in frontier:
      for v in Adj [u]:
        if v not in level:    # not yet seen
          level[v] = i    # = level[u] + 1
          parent[v] = u
          next.append(v)
    frontier = next
    i + =1
```

- $O(V+E)$ ("LINEAR TIME")

Shortest Paths

- for every vertex v, fewest edges to get from s to v is

$$\begin{cases}\text{level }[v] \text{ if } v \text{ assigned level} \\ \infty \quad \text{else (no path)}\end{cases}$$

- parent pointers form shortest-path tree $=$ union of such a shortest path for each v $\Longrightarrow$ to find shortest path, take v, parent $[v]$, parent $[$ parent $[v]]$, etc., until s (or None)

THINKING

本讲主要介绍了Graph的数据结构表达，以及BFS。

BFS在6.001和6.009中都有涉及，但这里是一个更偏向数据结构的介绍，也是更为泛化的介绍。

1. 不同数据结构针对不同类型的图（graph）有不同的特点：有些图的子节点可以通过方法生成，有些节点可能组成不同的图等等
2. BFS可以搜索整个图建立所有节点到目标节点的步数和最短路径上父节点的字典（在节点数有限的情况下）

另外开始介绍了魔方问题，作为图和BFS的引子，有趣的是，魔方问题有极多的节点，因此边长大于3的魔方就已经无法知道最短路径（类似于围棋？）

R13. BREADTH-FIRST SEARCH (BFS)

HANDSHAKING LEMMA

connected graph

connected component

GRAPH REPRESENTATION

Adjacency Matrix

For a graph $G=(V, E)$, we assume that the vertices are numbered $1,2, \ldots|V|$ in some arbitrary order. Then the adjacency matrix representation of G consists of a $|V| \times|V|$ matrix $A=\left(a_{i j}\right)$ such that

$$a_{i j}= \begin{cases}1, & \text { if }(i, j) \in E, \\ 0, & \text { otherwise }\end{cases}$$

Representation Tradeoffs

Space:

- Adjacency lists uses one node per edge, and two machine words per node. So space is $\Theta(E w)$ bits ($w=$ word size)
- Adjacency matrix uses V^2 entries, but each entry can be just one bit. So space can be $\Theta\left(V^2\right)$ bits.

Time:

- Add an edge: both data structures are $O(1)$.
- Find if there is an edge from u to v : matrix is $O(1)$, and adjacency list must be scanned.
- Visit all neighbors of v (very common): matrix is $\Theta(V)$, and adjacency list is O (neighbors). This means BFS will take $O\left(V^2\right)$ time if we use adjacency matrix

representation.
- Remove an edge: similar to find and add.

The adjacency list representation provides a compact way to represent sparse graphs - those for which $|E|$ is much less than $|V^2| -$ it is usually the method of choice. We may prefer an adjacency matrix representation, however, when the graph is dense $-|E|$ is close to $|V^2| -$ or when we need to be able to tell quickly if there is an edge connecting two given vertices.

THINKING

讲解了另一个图的表示方法：Adjacency Matrix，很方便查看两个node是否相连，存储空间也只需要 $O(V^2)$ bits，但对于BFS来说查找每个节点的子节点都需要 $O(V)$, 因此总复杂度为 $O(V^2)$，而其他数据结构只需要 $O(V+E)$ (queue) 或 $O(E)$ (dictionary)

L14. DEPTH-FIRST SEARCH (DFS), TOPOLOGICAL SORT

DEPTH FIRST SEARCH ALGORITHM

- follow path until you get stuck
- backtrack along breadcrumbs until reach unexplored neighbor
- recursively explore
- careful not to repeat a vertex

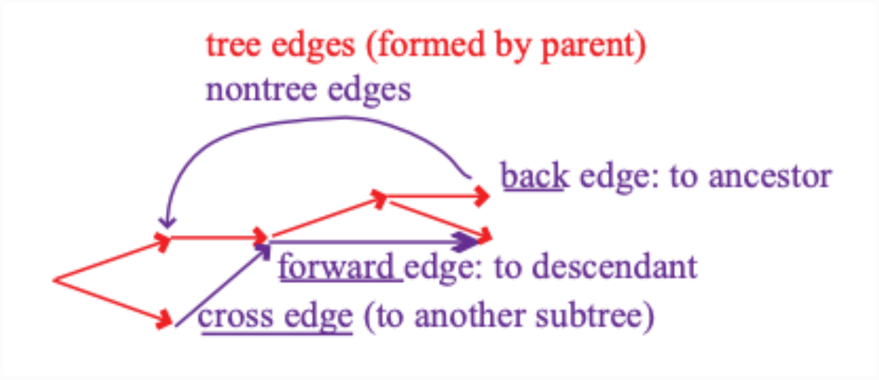
```
parent = {s: None}

DFS-visit (V, Adj, s):   # search from start vertex s (only see stuff reachable from s)
 for v in Adj [s]:
  if v not in parent:
    parent [v] = s
    DFS-visit (V, Adj, v)

DFS (V, Adj):   # explore entire graph (could do same to extend BFS)
 parent = { }
 for s in V:
  if s not in parent:
    parent [s] = None
    DFS-visit (V, Adj, s)
```

- Complexity: $O(V + E)$

EDGE CLASSIFICATION


tree edges (formed by parent)
nontree edges
back edge: to ancestor
forward edge: to descendant
cross edge (to another subtree)

- to compute this classification (back or not), mark nodes for duration they are "on the stack"
- only tree and back edges in undirected graph

CYCLE DETECTION

Graph G has a cycle $\Leftrightarrow$ DFS has a back edge

JOB SCHEDULING

Given Directed Acylic Graph (DAG), where vertices represent tasks & edges represent dependencies, order tasks without violating dependencies

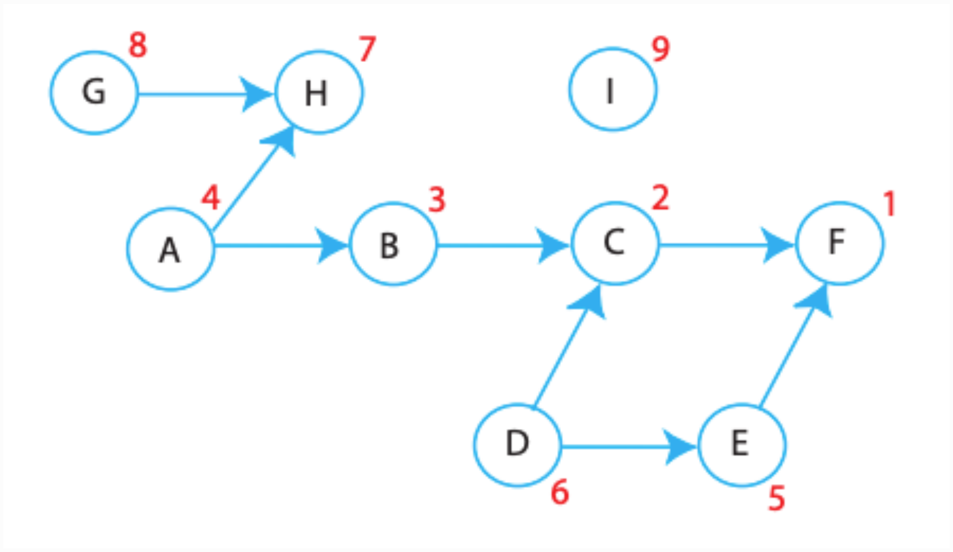


Figure 6: Dependence Graph: DFS Finishing Times

Topological Sort

only works on directed acyclic graph (DAG)

Reverse of DFS finishing times (time at which DFS-Visit(v) finishes)

```
DFS-Visit(v):
  ...
  order.append(v)

order.reverse()
```

Correctness

For any edge $(u, v) - u$ ordered before v, i.e., v finished before u

*Because of Recursion and Backtracking

THINKING

本讲主要介绍了DFS，同样是从更为抽象的角度来介绍。也因此有更多的应用。

这里的DFS是使用recursion和backtracking来实现的，因此子节点的结束时间会比父节点的结束时间要早，通过节点的结束时间可以进行很多操作：

1. 进行边分类，从而查找图中是否有环。
    - 如存在back edge，则有环
    - 确定back edge的方法为DFS过程中如果探索该边时目标节点为已经开始但还未结束，则该边为back edge
2. 然后在确定无环的图中，可以进行topological sort（拓扑排序），方法为将结束时间倒序排列。

R14. DEPTH-FIRST SEARCH (DFS)

```python
class DFSResult:
  def __init__(self):
    self.parent = {}
    self.start_time = {}
    self.finish_time = {}
    self.edges = {} # Edge classification for directed graph.
    self.order = []
    self.t = 0

def dfs(g):
  results = DFSResult()
  for vertex in g.itervertices():
    if vertex not in results.parent:
      dfs_visit(g, vertex, results)
  return results

def dfs_visit(g, v, results, parent = None):
  results.parent[v] = parent
  results.t += 1
  results.start_time[v] = results.t
  if parent:
    results.edges[(parent, v)] = 'tree'

  for n in g.neighbors(v):
    if n not in results.parent: # n is not visited.
      dfs_visit(g, n, results, v)
    elif n not in results.finish_time:
      results.edges[(v, n)] = 'back'
    elif results.start_time[v] < results.start_time[n]:
      results.edges[(v, n)] = 'forward'
    else:
      results.edges[(v, n)] = 'cross'

  results.t += 1
  results.finish_time[v] = results.t
  results.order.append(v)


def topological_sort(g):
  dfs_result = dfs(g)
  dfs_result.order.reverse()
  return dfs_result.order
```

THINKING

这节复习课详细介绍了DFS中边的分类，然后介绍了边分类和拓扑排序的代码实现，使用了一个DFSResult类记录DFS过程中的所有结果，并使用这些结果生成其他需要的结果（如记录访问节点的起止时间，用于边分类和拓扑排序）