Basic Algorithm Lec 8

Graphs are structures to store "relations." Generalizations of trees. (not always hierarchical)

E.g. Social networks.

\[ G = (V, E) \]

\( V \) = set of vertices (nodes/points)

\( E \subseteq V \times V \rightarrow \) edges (relations).

E.g. Software dependency
These are directed graphs.

Undirected: Networks (Internet / Phone).

NY region

HIGHWAYS.

NYC

BOSTON

Philly

TIME WARNER

XFINITY

CHICAGO
weighted

Add info to the edges to give meanings to the relations.

E.g. Currency Exchange

\[
\begin{align*}
&\text{US $} \rightarrow \text{£ pound} \\
&0.75 \\
&0.76 \rightarrow 1.75 \\
&\text{€} \rightarrow 0.65 \rightarrow \text{C $} \\
&1.53 \rightarrow 83.72 \\
&\text{¥ yen}
\end{align*}
\]

Almost always $y$ is going to be directed.

\((v_i, v_j) \in E \Rightarrow v_j \text{ is a neighbour of } v_i \text{. (successor/adjacent)}\)

Path: $v_1, v_2, \ldots, v_k$

\((v_i, v_{i+1}) \in W \forall i < k\).

Simple path \(\Rightarrow v_i \neq v_j \forall i \neq j\)

Simple graph \(\Rightarrow (v_i, v_j) \notin E \forall v_i, v_j \in V\).

Can have \(\leq 2\) only one edge between \((v_i, v_j)\) and \((v_j, v_i)\).
Representing a graph

**Adjacency Matrix.**

The edges can be stored in a matrix of size $|V| \times |V|$, 0 if no-edge 1/0.

**E.g.**

```
   1 -- 2 -- 3
  ^    |    |
  |    v    |
  3     4
```

```
   1  2  3  4

   1 | 0 1 1 1 |
   2 | 0 0 0 0 |
   3 | 0 0 0 1 |
   4 | 0 0 0 0 |
```

Can store weights instead of 0/1 as well.

**Adjacency List.**

Only store the edges.

**E.g.**

```
2 -> [4]
3 -> [4]
4
```

**Java:** `adj -> [list]`

**C++:** `adj -> [list]`
if it is a weighted graph, store pairs in the list.

e.g.

```
1 -> [2, 10] -> [5, 2]
2 -> [4, 17] -> [3, 15]
3
4
5 -> [3, 29]
```

Pairs / Tuples in Java → need to make your own
std::pair <int, int> → C++.

---

A lot of other representations.

e.g. for games: chess

→ represent state of the board. → nodes.

Each next configuration can be made on the fly

"Implicit graph"
Edge representation

Edge objects: start, finish
weight.
Can store other data.

Advantages: fast edge searching.
good for very sparse graphs.

Breadth First Search
Explore the graph level by level.
Start from 8

```
8  →  lev 0

```

E.g.

```
1 → 4 ← 7 ← 8
2 → 3 → 5 → 9
              \        /
               6       10
                    \  /
                     11
```

BFS (8) creates a tree with back edges e.g. 7 -> 4
level edges 7 -> 9

notice 6, 8 are skipped.

// Make a tree using
queue q.
q.push (1)

while q is not empty:
  v = q.pop()
  for nhb of v:
    if not visited [nhb]
      parent [nhb] = v
      level [nhb] = level [v] + 1
      visited [v] = true
Analysis for running time.

1) Each vertex is put into q only once.
2) Every edge is analyzed once, when the starting point of the edge is popped out of q.

\[ T(q) = \sum_{v \in V} |\text{adj}[v]| + O(|V|) \]
\[ = O(|E| + |V|) \]

In general \(|E| \rightarrow m, |V| \rightarrow n\)

\[ T(q) = O(n+m) \] linear time.

Applications of BFS

Given a directed graph \(G := (V, E)\)
find the shortest path from \(s\) to \(t\)
BFS(s), give level \(t\).
Real world uses.

* Every Rubik's cube configuration can be solved in less than or equal to 20 moves.
* Find shortest path between Wikipedia articles by following links.

---

**Depth first search**

Instead of level by level, travel all the way down a path and then backtrack.

The same code as BFS except replace queue by stack.

e.g.

```
1
  \rightarrow
2  \rightarrow
  \rightarrow
3  \rightarrow
  \rightarrow
5  \rightarrow
    \rightarrow
7  \rightarrow
  \rightarrow
10
  \rightarrow
4
  \rightarrow
9
  \rightarrow
11
```

types of edges:
- **forward edges**
  
  \(1 \rightarrow 4; \ 5 \rightarrow 9\)
other types.

\[
\text{back edges:} \quad \begin{array}{c}
3 \rightarrow 1 \\
\end{array}
\]

Doing a BFS / DFS at a single node gives a tree (specified by the parent array).

- But some nodes may not be visited by this.
- Do a BFS at each node to get a "forest". (collection of trees)

\[
\text{e.g.}
\]

1 \rightarrow 3 \leftarrow 4

\downarrow \downarrow \downarrow

2 \leftarrow 5 \rightarrow 6

\uparrow \downarrow \downarrow

8 \rightarrow 7 \rightarrow 9
a forest with 3 trees.
2 of which have 1 node
(no back edges)

$1 \rightarrow 5 \rightarrow 9$ are forward edges.

Detecting a cycle.

$G$ has a cycle $\Rightarrow$ DFS of $G$ has a back edge

e.g.

$3 \rightarrow 1$ back edge.
Say \( G \) has a cycle

\[ v_1 \rightarrow v_2 \]

- Say we start the DFS at some node \( U \).
- By reach some vertex \( v_1 \) of the cycle.
- Then we would travel along the subtrees of \( v_1 \) before finishing all its descendants.
- We would visit \( v_2 \).
- Similarly, we would visit \( v_3, \ldots, v_k \).

\( \Rightarrow \) we would see the back edge \( v_k \rightarrow v_1 \).

\( \Leftarrow \) If there is a back edge

\[ v_1 \]

\[ v_2 \]

\[ v_3 \]

\[ v_4 \]

\[ v_5 \]

\[ v_k \]

then \( v_1, \ldots, v_k \) is a cycle.
DFS with coloring.

- Hard to identify back edges if we only store parents & visited.
- How to distinguish between cross/forward/back edges?

Idea: color a node white/grey/black.
  white: not started DFS at node
  grey: currently exploring subtree of node
  black: finished exploring subtree.

Recursive formulation of DFS

DFS(node)
  if color[node] = white:
    color[node] = grey
    for u in node:
      if color[u] = white:
        DFS(u), parent[u] = node.
      elif color[u] = grey:
        "node -> u" is a back edge.
      else:
        "node -> u" is a cross/forward edge.
    color[node] = black.

edge.
A graph without cycles is called a Directed Acyclic Graph (DAG).

Most common uses of DAGs are dependency relationships. The software installation example:

What order to install software?

E.g.: Nodes $\rightarrow$ represent software.

Edges $\rightarrow$ dependencies

\[ \text{.NET} \quad \uparrow \quad \downarrow \]

\[ \text{DIRECT-X} \quad \text{V.C++ Runtime} \]

\[ \text{GTA V} \]

Can install in two orders:

- .NET, DIRECT-X, V.C++, GTA V
- .NET, V.C++, DIRECT-X, GTA V.
These orders are called topologically sorted orderings of V.

Properties:

- Edges only go in the forward direction.

Thus:
Every DAG has a topological ordering.
& every graph that has a topological ordering is a DAG.

Say G has a top sorted order

\[ v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \ldots \rightarrow v_n \]

So G cannot have a cycle as there are no edges in the reverse direction.

\[ \Rightarrow \] Say G is a DAG with n vertices.

Claim is that G has a vertex with 0 indegree (number of incoming edges) & 0 outdegree (number of outgoing edges).

Proof: Travel on the graph, to reach at the vertex that you have already visited.

\[ \Rightarrow \neg \text{ cycle} \Rightarrow n \]
Move detailed proof.

we will first show \( \exists \) a vertex with 0-out degree.

Suppose not

- Start from a vertex \( x \)
  and start travelling on a path.
- We can keep travelling as each vertex has at least one outgoing edge.
- By pigeonhole principle after \( n \) steps we will have visited some vertex twice.

\[ \Rightarrow \text{ a cycle} \Rightarrow \exists \]

\[ \therefore \exists \text{ a vertex with out degree 0.} \]

for indegree 0 vertex, reverse each edge of the graph, which is still a DAG & do the above.

Now to show that \( G \) has a top sorted ordering.

Suppose that \( \forall G' \) of size \( n-1 \) vertices we have a top sorted ordering.
we will prove by induction, trivial for \( n=1 \)
we know that $G$ has an indegree 0 vertex. 

say $v_1$

remove $v_1, G \backslash v_1$ has $n-1$ vertices & is still a DAG

$\Rightarrow G \backslash v_1$ has a top sort ordering

say $u_1, u_2 \cdots u_{n-1}$

then

$v_1 \notin u_1, u_2 \cdots u_{n-1}$

is a top sort ordering for $G$ as there are no back edges.

This gives an algorithm to find a top sort of $G$.

1) ind[i, ..., n] → calculate indegrees

2) roots = $\emptyset$ → empty set of indegree 0 vertices

3) for $i = 1$ to $n$
   if ind[i] = 0
      roots.add(i)

4) while roots is not empty:
   node = roots.pop()
   print(node)
   for u in neighbours of node:
      ind[u] -= 1
      if ind[u] = 0 & roots.add(u).
To analyze the time complexity.
- Each node is inserted & popped only once from root. \( \Rightarrow O(1V1) \)
- We analyze each edge out of a node exactly once \( \Rightarrow O(1E1) \)

\[ T(G) = O(n+m) \Rightarrow \text{linear} \]

Kahn's Algorithm for Top sort \( \uparrow \)

Modify this to get lexically smallest top sort.

Only a small modification

\( \Rightarrow \) make root a min heap.

Each time you remove an element it will give the smallest in the dictionary order.

\[ T(G) = O(n \log n + m \log n) \]

\( \downarrow \)

To do deletion \( \downarrow \)

(POP)