CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie

Graph ADT

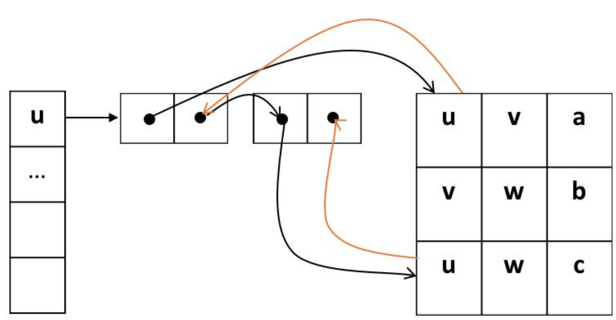
- **Data**: all vertices, all edges, and the structure to maintain relations between vertices and edges.
- **Functions**:
  - insert vertex/edge,
  - remove vertex/edge,
  - find incident edges,
  - check if two vertices are adjacent, and
  - In case of directed graph find origin/destination.

Question: Implementation 2 runs in either O(1) or O(n), while Implementation 1 runs in either O(1) or O(m). Which one is better?

- Depends on the data. If the graph is not connected Implementation 1 is better even though it seems bad when we look at the worst case running time.
- Before we decide on the implementation we need to consider what our data set looks like.

Graph implementation 3 : ADJ List

- We will maintain a hash table of vertices, and every vertex in the table has a linked list of pointers which point to edges in the edge list.
- Elements from the edge list will point back to the hash table.



- The running time of **insert vertex** would be O(1) → we would just add a vertex to the hash table.
- The running time of **remove vertex** would be O(deg(v)):
  - loop through the list of incident edges → each node v has deg(v) incident edges, so we have to go over deg(v) edges to remove a vertex.

- To **check adjacent nodes**, we need to go through incident edges of one of the vertices. We will chose the vertex with smaller list. The running time is O(min(deg(v1), deg(v2))).
- **Find incident edge**s takes O(deg(v)).
- **InsertEdges**: O(1) + O(1) + O(1) = O(1)

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie

Summary of graph implementation run time

	Edge List	Adjacency Matrix	Adjacency List
space	n+m	n^2	n+m
insert vertex	1 😀	n	1 😀
remove vertex	m	n	deg(v) 😀
insert edge	1 😀	1 😀	1 😀
remove edge	1 😀	1 😀	1 😀
incident edges	m	n	deg(v) 😀
are adjacent	m	1 😀	min(deg(v), deg(w))

- When removing a vertex deg(v) is always the best: if m = 0, deg(v) = 0; if the graph is simple, deg(v) = n-1.
- If we care about areAdjacent we are going to use adjacency matrix. On the other hand, if we care about incident vertices, we will use adjacency list. Insert/remove and edge takes O(1) because we are just adding/removing from the front of the linked list.
- Some possible cases:
    - Sparse graphs: the graph is not connected → $m < n$ which implies $deg(v) < n$. So in the case of sparse graph, we want to use adjacency list implementation.
    - Dense graphs: the graph is almost fully connected → $m \sim n^2$. So in the case of dense graph, we can use either adjacency list or adjacency matrix. It depends on the operations we need (are adjacent or insert vertex).

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie

Traversal

- Objective: Visit every vertex and every edge exactly once
- Purpose: Search for interesting substructures in the graph
- We've done it on trees, but it was easier

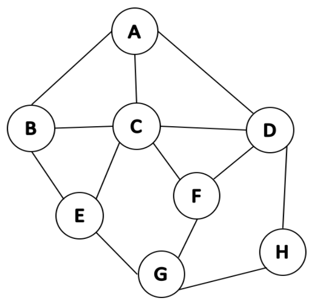
Trees	Graphs
1. Ordered → we always go from parents to children. 2. Obvious start → we start at the root node. 3. Notion of completeness → we are done when we reach leaf nodes.	1. Unordered → no notion of children nodes, just neighbours. 2. No obvious start → we can start anywhere. 3. No notion of completeness → we need to know when we have visited all nodes.

BFS

- Setup:
  - Maintain a queue
  - Maintain a table of vertices with following features:
    - Boolean flag - visited
    - Distance from the start
    - Predecessor
    - List of adjacent vertices

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie



key	visited	dist.	pred.	adj. vertices
A				C B D
B				A E C
C				A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

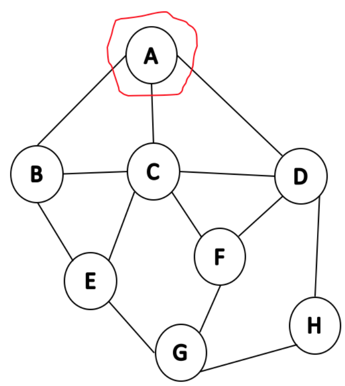
Queue

- Algorithm:
    1. Add the starting point
    2. While the queue is not empty
        a. Dequeue **v**
        b. For all of the <u>unlabeled</u> edges adjacent to **v**
            - If an adjacent edge "discovers" a new vertex **t**:
                - Label the edge a "<u>discovery edge</u>"
                - Enqueue **t**, update the information of **t** (distance = dist(**v**) + 1, predecessor = **v**)
            - If an adjacent edge is between two visited vertices
                - Label the edge a "<u>cross edge</u>"

- **Example:**
    - A as a starting point:

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie

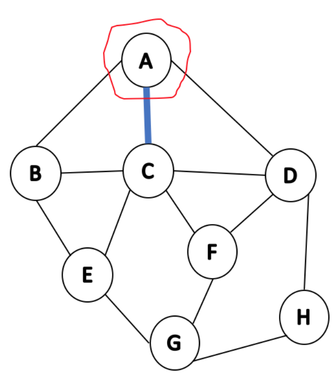


key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B				A E C
C				A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

Queue

A								

- dequeue A and examine vertices C, B, and D.
- First examine C. It hasn't been visited, so we add a discovery edge, update visited flag, set distance to parent's distance plus vertex distance which is one in this case, set predecessor to A, and add C to the queue.

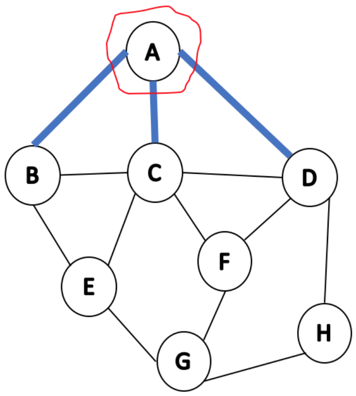


key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B				A E C
C	✓	1	A	A B D E F
D				A C F H
E				B C G
F				C D G
G				E F H
H				D G

Queue

~~A~~	C							

- We do the same for B and D.



key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E				B C G
F				C D G
G				E F H
H				D G

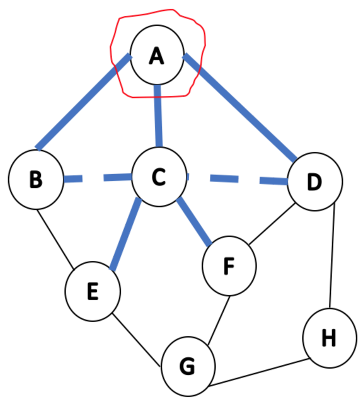
Queue

~~A~~	C	B	D					

- Repeat this process until the queue is empty.
- Dequeue C and examine its adjacent edges. A has been visited and the edge is labeled as discovery edge, so it is just ignored. B and D have been visited from another node, so we add a cross edge but do not update anything. E and F have not been discovered yet, therefore we add discovery edges to the graph, add vertices to the queue, and update the table.

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie

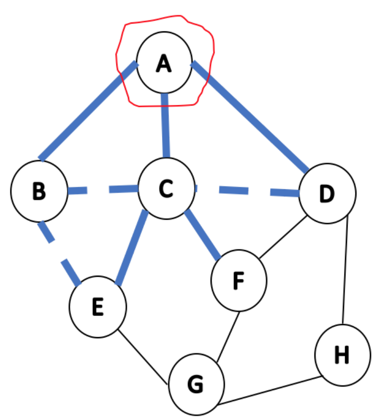


key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E	✓	2	C	B C G
F	✓	2	C	C D G
G				E F H
H				D G

Queue

~~A~~	~~C~~	B	D	E	F			

- Next, dequeue B and add a cross edge to E.



key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E	✓	2	C	B C G
F	✓	2	C	C D G
G				E F H
H				D G

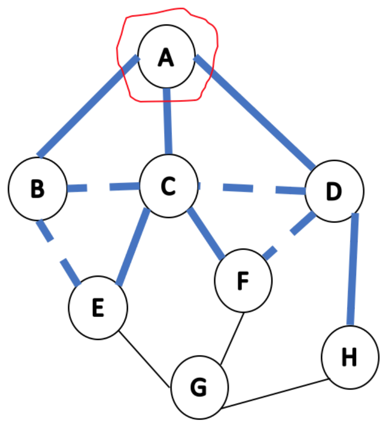
Queue

~~A~~	~~C~~	~~B~~	D	E	F			

- Dequeue D, add cross edge to F, and update H.

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie



key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E	✓	2	C	B C G
F	✓	2	C	C D G
G				E F H
H	✓	2	D	D G

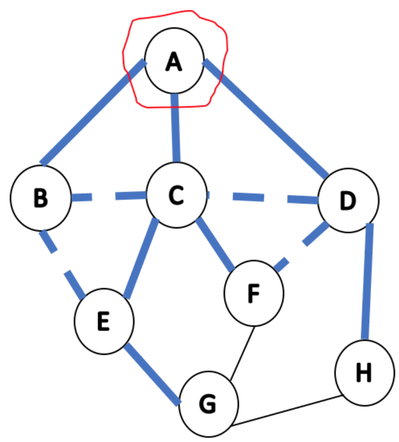
Queue

~~A~~	~~C~~	~~B~~	~~D~~	E	F	H		

- Then, dequeue E and update G.

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie



key	visited	dist.	pred.	adj. vertices
A	✓	0	null	C B D
B	✓	1	A	A E C
C	✓	1	A	A B D E F
D	✓	1	A	A C F H
E	✓	2	C	B C G
F	✓	2	C	C D G
G	✓	3	E	E F H
H	✓	2	D	D G

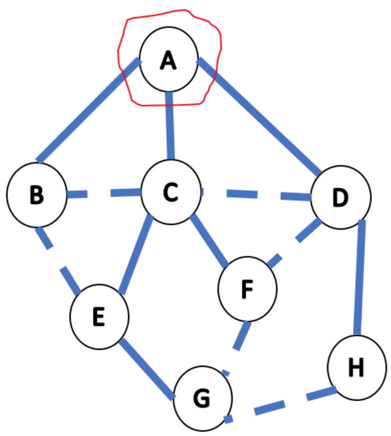
Queue

~~A~~	~~C~~	~~B~~	~~D~~	~~E~~	F	H	G	

- Dequeue F and add cross edge to G; Dequeue H and add cross edge to G.
- Finally dequeue G and we are done.

CS 225 Spring 2019 :: TA Lecture Notes
4/12 Graphs III

By Wenjie



key	visited	dist.	pred.	adj. vertices
A	√	0	null	C B D
B	√	1	A	A E C
C	√	1	A	A B D E F
D	√	1	A	A C F H
E	√	2	C	B C G
F	√	2	C	C D G
G	√	3	E	E F H
H	√	2	D	D G

Queue

~~A~~	~~C~~	~~B~~	~~D~~	~~E~~	~~F~~	~~H~~	~~G~~	