CS218- Data Structures
Week 14

Muhammad Rafi
December 11, 2020

Agenda

- Graphs
- Graphs Basic Terminologies
- Graph Data Structures
  - Graph Representation
  - Graph Algorithms
- Breadth First Search
- Depth First Search
- Shortest Path in a Graph
- All Pairs Shortest Paths
- Cycle Detections

Graphs

- Trees
  - Although Trees are very flexible data structures, they have some limitations
    - They can only model only a single type of hierarchical relationship – Parent -> Child
    - Knowledge about Sibling are hard to get through trees.
    - You can not climb back the tree once, you are down from a root pointer.
- Graphs
  - There is no such limitations in Graphs.
  - These are more flexible and representational in nature

Graphs

- Graphs
  - Mathematically Graph G=(V,E), where V is a non-empty set and E can be an empty set.
- Simple Graph
  - A simple graph G = (V, E) consists of a nonempty set V of vertices and a possibly empty set E of edges, each edge being a set of two vertices from V.
  - The number of and edges is denoted by |V| and |E |, respectively.

Graphs

- Multi Graph
  - A multigraph is a graph in which two vertices can be joined by multiple edges.
  - Geometric interpretation is very simple Formally, the definition is as follows: a multigraph G = (V,E,f) is composed of a set of vertices V, a set of edges E, and a function $f : E \rightarrow \{\{vi,vj\} : vi,vj \varepsilon V \text{ and } vi \neq vj\}$.
- Pseudograph
  - A pseudograph is a multigraph with the condition $vi \neq vj$ removed, which allows for loops to occur; in a pseudograph, a vertex can be joined with itself by an edge.

Graphs

- Directed Graphs
  - A directed graph, or a digraph, G = (V, E) consists of a nonempty set V of vertices and a set E of edges (also called arcs), where each edge is a pair of vertices from V.
  - The difference is that one edge of a simple graph is of the form {vi, vj}, and for such an edge, {vi,vj} = {vj,vi}.
  - In a digraph, each edge is of the form (vi,vj), and in this case, $(vi,vj) \neq (vj,vi)$

Graphs

- Weighted Graphs
  - A graph is called a weighted graph if each edge has an assigned number.
  - Depending on the context in which such graphs are used, the number assigned to an edge is called its weight, cost, distance, length, or some other name.

Graphs

- Complete Graphs
  - A graph with n vertices is called complete and is denoted K_n if for each pair of distinct vertices there is exactly one edge connecting them; that is, each vertex can be connected to any other vertex.
  - The number of edges in such a graph |E| = |V|^2.
- Connected Graph
  - All vertices are connected through some edges. A single component.
- Isolated Graph
  - The subsets of vertices may not be connected as a single component.

Graphs

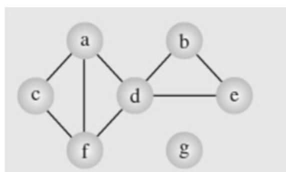
- Path
  - A path from v1 to vn is a sequence of edges edge(v1,v2), edge(v2,v3), . . . , edge (vn–1,vn) and is denoted as path v1, v2, v3, . . . , vn–1, vn. If v1 = vn and no edge is repeated, then the path is called a circuit.
- Cycle
  - If all vertices in a circuit are different, then it is called a *cycle.*

Graph Representations

- Graph Data Structures
  - There are several ways in which we can store graphs into memory.
  - The representation should be lossless. We can able to reconstruct the unambiguous graph.
  - Each representation is good for certain aspect of graph processing.
  - All graph processing libraries support internal conversion of these representations.
  - It is very common that we may change the representation before opting for an algorithm to solve any given problem on graph.

Graph Representations

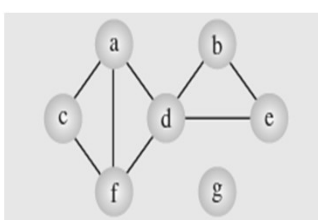
- Adjacency List
  - A simple representation is given by an adjacency list, which specifies all vertices adjacent to each vertex of the graph.
  - This list can be implemented as a table, in which case it is called a star representation, which can be forward or reverse.

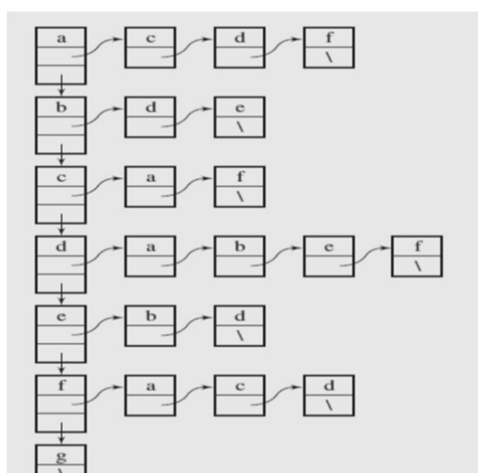


a	c	d	f	
b	d	e		
c	a	f		
d	a	b	e	f
e	b	d		
f	a	c	d	
g				

Graph Representations

- Adjacency List

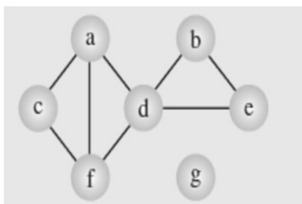




(c)

Graph Representations

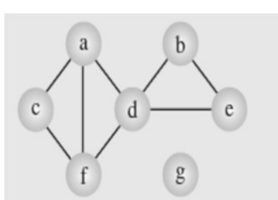
- Adjacency Matrix
  - An adjacency matrix of graph G = (V,E) is a binary |V| × |V| matrix.
  - Such that each entry of this matrix a_{ij} is 1 if v_i is connected to v_j.



	a	b	c	d	e	f	g
a	0	0	1	1	0	1	0
b	0	0	0	1	1	0	0
c	1	0	0	0	0	1	0
d	1	1	0	0	1	1	0
e	0	1	0	1	0	0	0
f	1	0	1	1	0	0	0
g	0	0	0	0	0	0	0

Graph Representations

- Incidence Matrix
  - An Incidence matrix of graph G = (V,E) is a binary |V| × |E| matrix.
  - Such that each entry of this matrix a_{ij} is 1 if v_i is connected to any edge.



	ac	ad	af	bd	be	cf	de	df
a	1	1	1	0	0	0	0	0
b	0	0	0	1	1	0	0	0
c	1	0	0	0	0	1	0	0
d	0	1	0	1	0	0	1	1
e	0	0	0	0	1	0	1	0
f	0	0	1	0	0	1	0	1
g	0	0	0	0	0	0	0	0

Adjacency List vs. Adjacency Matrix

Operations	Adjacency Matrix	Adjacency List
Storage Space	This representation makes use of VxV matrix, so space required in worst case is **O(|V|^2)**.	In this representation, for every vertex we store its neighbours. In the worst case, if a graph is connected O(V) is required for a vertex and O(E) is required for storing neighbours corresponding to every vertex .Thus, overall space complexity is O(|V|+|E|).
Adding a vertex	In order to add a new vertex to VxV matrix the storage must be increases to (|V|+1)^2. To achieve this we need to copy the whole matrix. Therefore the complexity is **O(|V|^2)**.	There are two pointers in adjacency list first points to the front node and the other one points to the rear node.Thus insertion of a vertex can be done directly in **O(1) time.**

Adjacency List vs. Adjacency Matrix

Operations	Adjacency Matrix	Adjacency List
Adding an edge	To add an edge say from i to j, matrix[i][j] = 1 which requires **O(1)** time.	Similar to insertion of vertex here also two pointers are used pointing to the rear and front of the list. Thus, an edge can be inserted in **O(1)**time.
Removing a vertex	In order to remove a vertex from V*V matrix the storage must be decreased to |V|^2 from (|V|+1)^2. To achieve this we need to copy the whole matrix. Therefore the complexity is **O(|V|^2)**.	In order to remove a vertex, we need to search for the vertex which will require O(|V|) time in worst case, after this we need to traverse the edges and in worst case it will require O(|E|) time.Hence, total time complexity is **O(|V|+|E|)**.

Adjacency List vs. Adjacency Matrix

Operations	Adjacency Matrix	Adjacency List
Removing an edge	To remove an edge say from i to j, matrix[i][j] = 0 which requires **O(1)** time.	To remove an edge traversing through the edges is required and in worst case we need to traverse through all the edges. Thus, the time complexity is **O(|E|)**.
Querying	In order to find for an existing edge the content of matrix needs to be checked. Given two vertices say i and j matrix[i][j] can be checked in **O(1)** time.	In an adjacency list every vertex is associated with a list of adjacent vertices. For a given graph, in order to check for an edge we need to check for vertices adjacent to given vertex. A vertex can have at most O(|V|) neighbours and in worst can we would have to check for every adjacent vertex. Therefore, time complexity is **O(|V|)** .

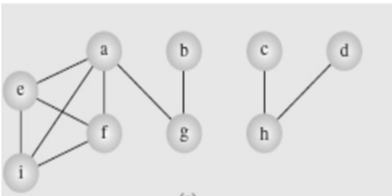
Graph Traversals

- Graph traversals are more challenging than Tree traversals
  - Cycles in the graphs
  - Isolated Graphs – disconnected components.
- Graph Search
  - Breadth First Search
  - Depth First Search

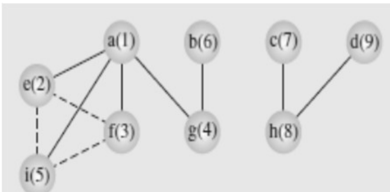
Breadth First Search

```
breadthFirstSearch()
   for all vertices u
      num(u) = 0;
   edges = null;
   i = 1;
   while there is a vertex v such that num(v) is 0
      num(v)=i++;
      enqueue(v);
      while queue is not empty
         v = dequeue();
         for all vertices u adjacent to v
            if num(u) is 0
               num(u) = i++;
               enqueue(u);
               attach edge(vu) to edges;
   output edges;
```

Breadth First Search

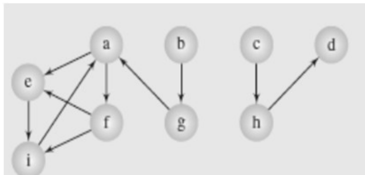


(a)

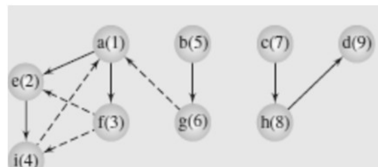


(b)

Breadth First Search



(a)

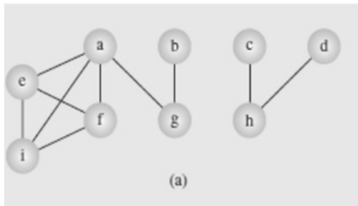


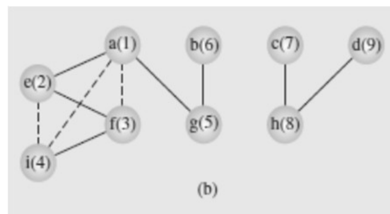
(b)

Depth First Search

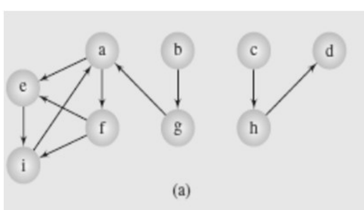
```
DFS(v)
   num(v)= i++;
   for all vertices u adjacent to v
      if num(u) is 0
         attach edge(uv) to edges;
         DFS(u);

depthFirstSearch()
   for all vertices v
      num(v) = 0;
   edges = null;
   i = 1;
   while there is a vertex v such that num(v) is 0
      DFS(v);
   output edges;
```

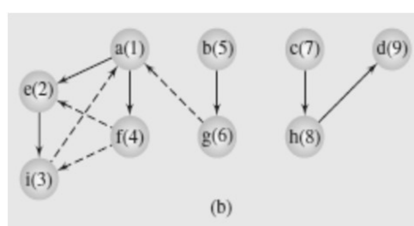
Depth First Search





Depth First Search





Shortest Path in a Graph

- Finding the shortest path is a classical problem in graph theory, and a large number of different solutions have been proposed.
- Edges are assigned certain weights representing different measure like cost of the path, distance, etc.
- When determining the shortest path from vertex v to vertex u, information about distances between intermediate vertices w has to be recorded.

Shortest Path in a Graph

- When determining the shortest path from vertex v to vertex u, information about distances between intermediate vertices w has to be recorded.
- The methods of finding the shortest path rely on these labels.
- There are two kinds of methods:
  - Label-setting methods
  - Label correcting methods.

Shortest Path in a Graph

- Label-setting methods
  - For label-setting methods, in each pass through the vertices still to be processed, one vertex is set to a value that remains unchanged to the end of the execution.
  - These methods limit to processing graphs with only positive weights.

Shortest Path in a Graph

- Label-Correcting methods
  - The label-correcting methods, allows for the changing of any label during application of the method. These methods limit to processing graphs with only positive weights.
  - These methods can be applied to graphs with negative weights and with no negative cycle.
  - Negative Cycle -a cycle composed of edges with weights adding up to a negative number—but they guarantee that, for all vertices, the current distances indicate the shortest path only after the processing of the graph is finished.

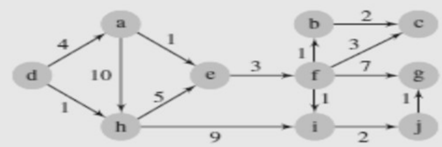
General Shortest Path

```
genericShortestPathAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    initialize toBeChecked;
    while toBeChecked is not empty
        v = a vertex in toBeChecked;
        remove v from toBeChecked;
        for all vertices u adjacent to v
            if currDist(u) > currDist(v) + weight(edge(vu))
                currDist(u) = currDist(v) + weight(edge(vu));
                predecessor(u) = v;
                add u to toBeChecked if it is not there;
```

Dijkstra Algorithm

```
DijkstraAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    toBeChecked = all vertices;
    while toBeChecked is not empty
        v = a vertex in toBeChecked with minimal currDist(v);
        remove v from toBeChecked;
        for all vertices u adjacent to v and in toBeChecked
            if currDist(u) > currDist(v)+ weight(edge(vu))
                currDist(u) = currDist(v)+ weight(edge(vu));
                predecessor(u) = v;
```

Dijkstra Algorithm



(a)

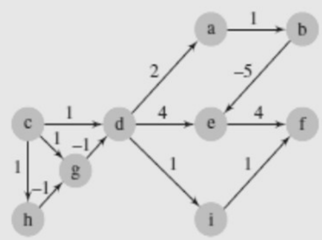
iteration:	init	1	2	3	4	5	6	7	8	9	10
active vertex:		**d**	**h**	**a**	**e**	**f**	**b**	**i**	**c**	**j**	**g**
a	∞	4	4								
b	∞	∞	∞	∞	∞	9					
c	∞	∞	∞	∞	∞	11	11	11			
d	0										
e	∞	∞	6	5							
f	∞	∞	∞	∞	8						
g	∞	∞	∞	∞	∞	15	15	15	15	12	
h	∞	1									
i	∞	∞	10	10	10	9	9				
j	∞	∞	∞	∞	∞	∞	∞	11	11		

(b)

Ford's Algorithm

```
FordAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    while there is an edge(vu) such that currDist(u) > currDist(v)+ weight(edge(vu))
        currDist(u) = currDist(v)+ weight(edge(vu));
```

Ford's Algorithm



(a)

the order of edges: *ab be cd cg ch da de di ef gd hg if*

		iteration				
	init	**1**		**2**	**3**	**4**
a	∞	3		2	1	
b	∞			4	3	2
c	0					
d	∞	1	0	–1		
e	∞	5		–1	–2	–3
f	∞	9	3	2	1	
g	∞	1	0			
h	∞	1				
i	∞	2		1	0	

(b)

Label-Correcting Algorithm

```
labelCorrectingAlgorithm(weighted simple digraph, vertex first)
    for all vertices v
        currDist(v) = ∞;
    currDist(first) = 0;
    toBeChecked = {first};
    while toBeChecked is not empty
        v = a vertex in toBeChecked;
        remove v from toBeChecked;
        for all vertices u adjacent to v
            if currDist(u) > currDist(v)+ weight(edge(vu))
                currDist(u) = currDist(v)+ weight(edge(vu));
                predecessor(u) = v;
                add u to toBeChecked if it is not there;
```

Label-Correcting Algorithm

FIGURE 8.9 An execution of `labelCorrectingAlgorithm()`, which uses a queue.

		active vertex																						
		c	d	g	h	a	e	i	d	g	b	f	a	e	i	d	b	f	a	i	e	b	f	e
queue		d g h	g h a e i	h a e i d	a e i d g	e i d g b	i d g b f	d g b f	g b f a e i	b f a e i d	f a e i d	a e i d	e i d b	i d b	d b f	b f a i	f a i e	a i e	i e b	e b f	b f	f e	e	
a	∞	∞	3	3	3	3	3	3	2	2	2	2	2	2	2	1								
b	∞	∞	∞	∞	∞	4	4	4	4	4	4	4	3	3	3	3	3	3	2					
c	0																							
d	∞	1	1	0	0	0	0	0	0	–1														
e	∞	∞	5	5	5	5	5	5	4	4	–1	–1	–1	–1	–1	–1	–2	–2	–2	–2	–2	–3		
f	∞	∞	∞	∞	∞	∞	9	3	3	3	3	3	3	3	2	2	2	2	2	1				
g	∞	1	1	1	1	0																		
h	∞	1																						
i	∞	∞	2	2	2	2	2	2	1	1	1	1	1	1	1	0								

Label-Correcting Algorithm

FIGURE 8.10 An execution of `labelCorrectingAlgorithm()`, which applies a deque.

		active vertex												
		c	d	g	d	h	g	d	a	e	i	b	e	f
deque		d g h	g h a e i	d h a e i	h a e i	g a e i	d a e i	a e i	e i b	i b f	b f	e f	f	
a	∞	∞	3	3	2	2	2	1						
b	∞	∞	∞	∞	∞	∞	∞	∞	2					
c	0													
d	∞	1	1	0	0	0	–1							
e	∞	∞	5	5	4	4	4	3	3	3	3	–3		
f	∞	∞	∞	∞	∞	∞	∞	∞	∞	7	1			
g	∞	1	1	1	1	0								
h	∞	1												
i	∞	∞	2	2	1	1	1	0						

All Pairs Shortest Path

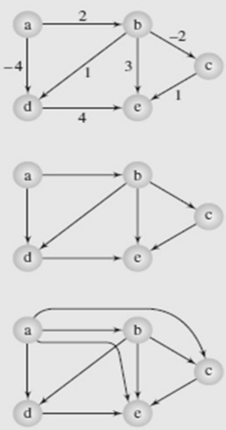
- All pairs shortest path seems more complicated than simple single source shortest path at first.
- W. Floyd algorithm is an elegant implementation of this problem.

All Pairs Shortest Path

```
WFIalgorithm(matrix weight)
   for i = 1 to |V|
      for j = 1 to |V|
         for k = 1 to |V|
            if weight[j][k] > weight[j][i] + weight[i][k]
               weight[j][k] = weight[j][i] + weight[i][k];
```

All Pairs Shortest Path

FIGURE 8.11 An execution of `WFIalgorithm()`.



(a)

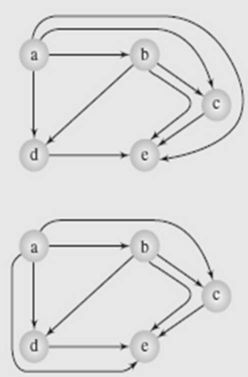
	a	b	c	d	e
$\boxed{a}$	0	2	∞	−4	∞
b	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

(b)

	a	b	c	d	e
a	0	2	0	−4	5
$\boxed{b}$	∞	0	−2	1	3
c	∞	∞	0	∞	1
d	∞	∞	∞	0	9
e	∞	∞	∞	∞	0

(c)

All Pairs Shortest Path



	a	b	c	d	e
a	0	2	0	−4	1
b	∞	0	−2	1	−1
$\boxed{c}$	∞	∞	0	∞	1
d	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

(d)

	a	b	c	d	e
a	0	2	0	−4	0
b	∞	0	−2	1	−1
c	∞	∞	0	∞	1
$\boxed{d}$	∞	∞	∞	0	4
e	∞	∞	∞	∞	0

(e)

Cycle Detection

- As it can be seen with all previous algorithm that cycle is quite challenging in algorithm development for graph.
- A simple cycle detection algorithm is very much required.
  - Undirected Graph
  - Directed Graph

Cycle Detection

```
cycleDetectionDFS(v)
    num(v) = i++;
    for all vertices u adjacent to v
        if num(u) is 0
            pred(u) = v;
            cycleDetectionDFS(u);
        else if edge(vu) is not in edges
            pred(u) = v;
            cycle detected;
```

```
DFS(v)
    num(v)= i++;
    for all vertices u adjacent to v
        if num(u) is 0
            attach edge(uv) to edges;
            DFS(u);

depthFirstSearch()
    for all vertices v
        num(v) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num(v) is 0
        DFS(v);
    output edges;
```

Di-graph Cycle Detection

```
digraphCycleDetectionDFS(v)
    num(v) = i++:
    for all vertices u adjacent to v
        if num(u) is 0
            pred(u) = v;
            digraphCycleDetectionDFS(u);
        else if num(u) is not ∞
            pred(u) = v;
            cycle detected;
    num(v) = ∞;
```

```
DFS(v)
    num(v)= i++;
    for all vertices u adjacent to v
        if num(u) is 0
            attach edge(uv) to edges;
            DFS(u);

depthFirstSearch()
    for all vertices v
        num(v) = 0;
    edges = null;
    i = 1;
    while there is a vertex v such that num(v) is 0
        DFS(v);
    output edges;
```