Data Structure
Lec-11 Graph

Review: Objective - Disjoint Set

- Equivalence relations
- Equivalence classes
- Disjoint set Abstract Data Type
- Array Implementation
- Application: Maze generation

Review: Equivalence Relations

- A relation R is defined on a set S if for every pair of elements (*a*,*b*), *a,b* in S, *a* R *b* is true or false
- If *a* R *b* is true, we say that a is related to b
- R is an equivalence relation if
  - (Reflexive) *a* R *a* is true for all element *a* in set S
  - (Symmetric) *a* R *b* → *b* R *a*, *b* R *a*→ *a* R *b*
  - (Transitive) *a* R *b* and *b* R *c* implies *a* R *c*
- If R is an equivalence relation, then R divide S into several disjoint components, elements in the same component is equivalent to each other
  - The relation “same country” divide the world map into different countries

Review: Equivalence Relations

- Given an equivalence relation ~, the natural problem is to decide,

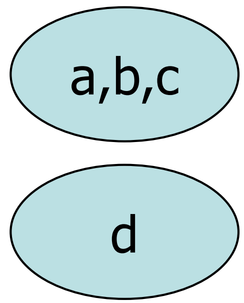
for any a and b, whether a ~ b.

	a1	a2	a3	a4	a5
a1	1	1			1
a2	1	1		1	
a3			1	1	
a4		1	1	1	
a5	1				1

a1 ~ a2
a3 ~ a4
a5 ~ a1
a4 ~ a2

Review: Equivalence Classes

- An equivalence class of an element a is the subset of all the elements that are equivalent to a
- Equivalence classes are disjoint
- Example:
  - Elements:a,b,c,d
  - if a~b,b~c
  - There are two equivalence classes
  - Elements can be represented like:



Exercise :

- Elements: a,b,c,d,e,f,g,h,i,j,k
- if a~b, b~c, b~d, e~f, g~h,i~e,j~k,k~c
- How many equivalence classes?
- Draw these equivalence classes

- Hint: use transitive rule to identify all the elements that are equivalent to ‘a’

Review: Disjoint Set ADT

- Value:
  - A set of items that belong to some data type ITEM_TYPE
  - Each item is associated with an equivalence class name

- Operations on DS
  - **Find**(ITEM_TYPE a)
    - ❑ return the equivalence class name for a
    - ❑ If Find(a)==Find(b), then a and b are in the same equivalence class
  - **Union**(ITEM_TYPE a, ITEM_TYPE b)
    - ❑ Combine a's equivalence class with b's equivalence class (make them the same name)
    - ❑ Precondition (Suppose): Find(a)==Find(c), Find(b)==Find(d)
    - ❑ Postcondition: a,b,c,d are in the same equivalence class (their equivalence class name are the same.)

Review: Array Implementation

- Array s[]
- s[i] stores the equivalence class name for i

0	1	2	3	4	5	6	7	8	9

- find(i): return s[i] worst case O(1)
- Union(i,j): change all the value s[j] into s[i] in the array: O(n)
- Example:
  - Union(1,2)
  - Union(3,1)
  - Find(1): return __

For Find(), the return value is not important.
The important thing is whether Find(i) and Find(j) are the same

0	1	2	3	4	5	6	7	8	9
0	1	1	3	4	5	6	7	8	9
0	3	3	3	4	5	6	7	8	9

Review: Array Implementation

```
class DisjointSet
{
        public:
                int Find(int )
                void Union(int,int)
        private:
                int name[MAX_SIZE]
}
```

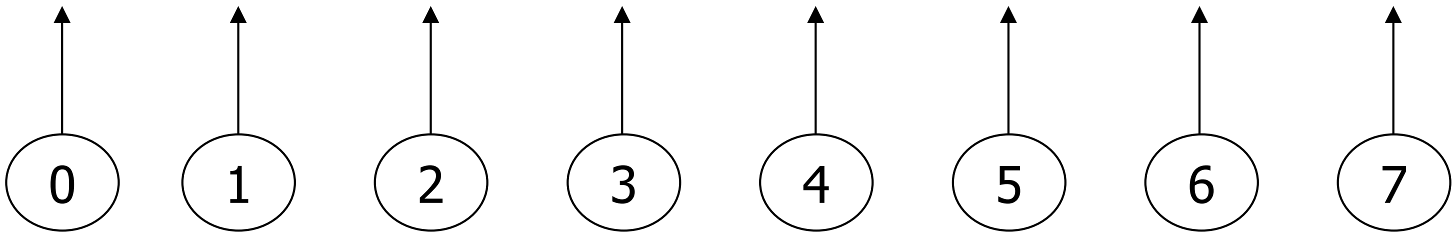
```
int DisjointSet::Find(int i)
{
        return name[i];
}



void DisjointSet::Union(int i, int j)
{
        if Find(i)!=Find(j)
        {
                int temp=name[j];

                for (k=0;k<MAX_SIZE;k++)
                {
                        if(name[k]==temp)
                                name[k]=name[i];
                }
        }
}
```

Review: Maze Generation

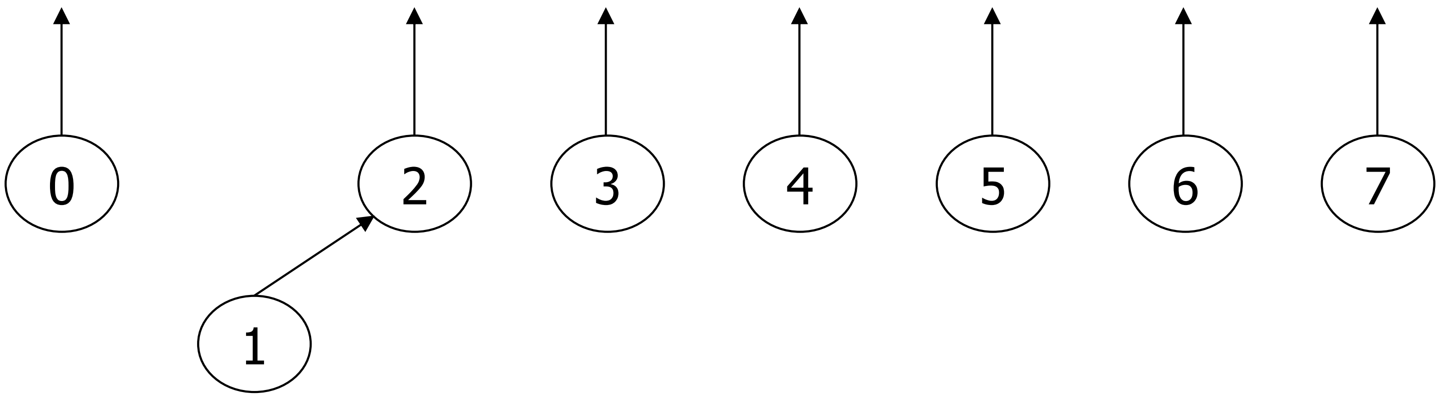
```
While the entrance cell is not connected to the exit cell
    Randomly pick a wall (two adjacent cells i and j);
    If (Find(i)!=Find(j))
        Union(i,j);      //Break the wall
```

Review: Disjoint Set (Tree Implement.)

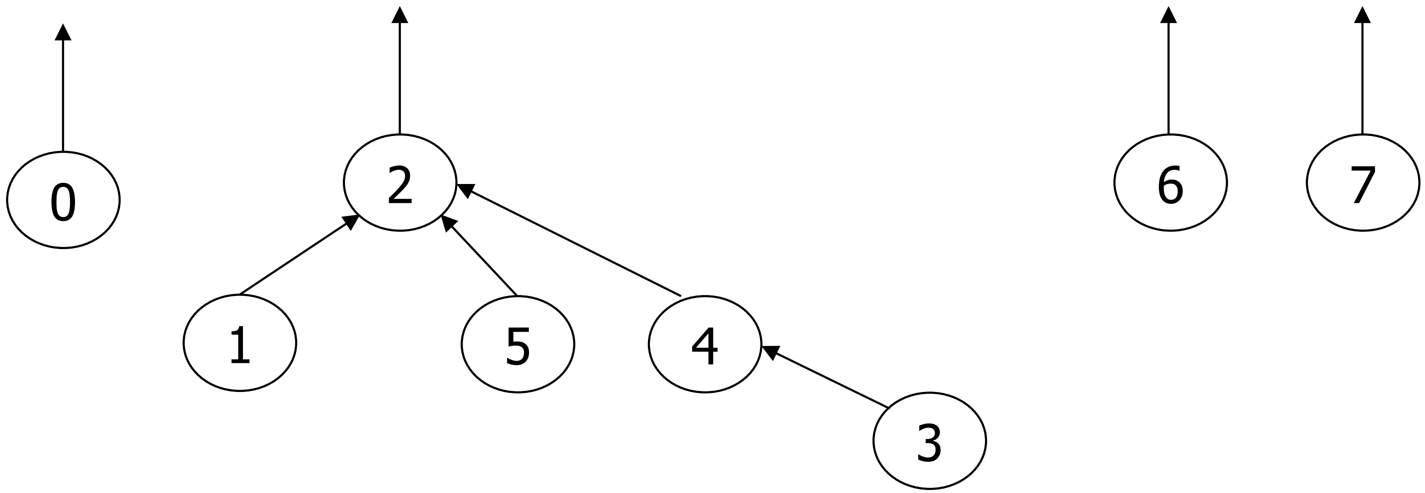
- Initially, all the nodes are independent



- After Union(1,2)



Review: use array to represent link



-1	2	-1	4	2	2	-1	-1
0	1	2	3	4	5	6	7

Review: Disjoint Set (Tree Implement. with array)

- Rule for Find(*i*)
  - Follow upward links of node *i* until we can go no further by following links

- Rule for Union(*i*,*j*)

  Find(i): Find the root r(*i*) of the tree which *i* belongs to

  Find(j): Find the root r(*j*) of the tree which *j* belongs to

  r(*i*)->next=r(*j*)

  Connecting rule:

  - smaller tree to larger tree
  - shallower tree to deeper tree

Review: Union-by-Height

- Guarantee all the trees have depth at most ***O(log N)***, where N is the total number of elements
- Running time for a find operation is ***O(log N)***
- The height of a tree increases one only when two equally deep trees are joined.

Review: Theorem 1

- **Theorem 1:** Union-by-height guarantees that all the trees have depth at most O(log N), where N is the total number of elements.

- **Lemma 1.** For any root x, size(x) $\geq 2^{\text{height}(x)}$, where size(x) is the number of nodes of x's tree, including x.

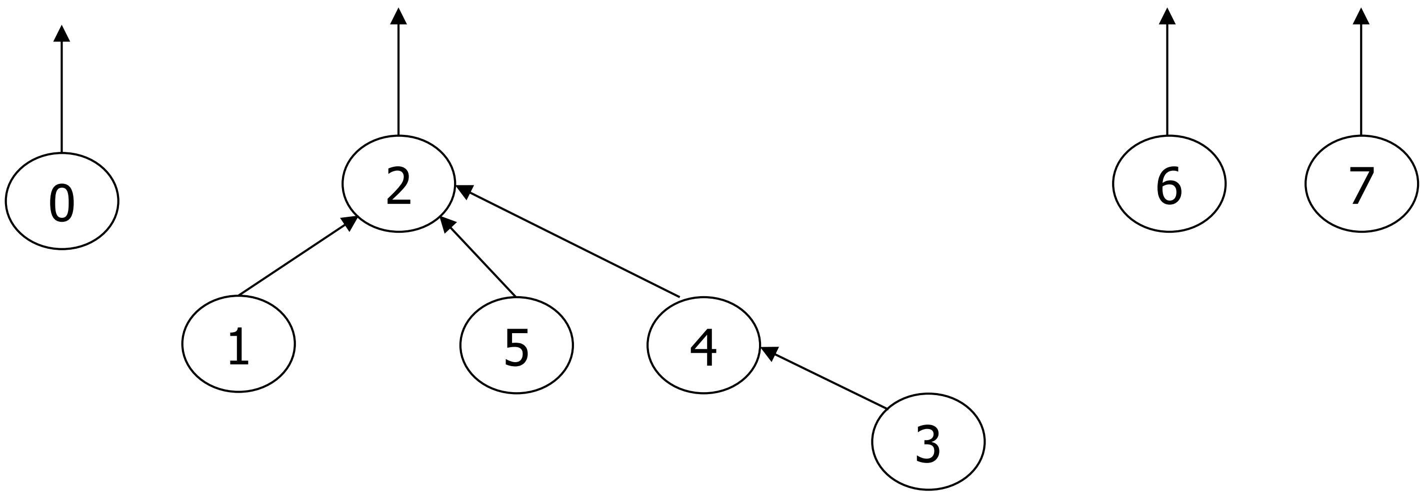
Review: Proof of Lemma 1 (1/3)

- Proof (by induction)
  - **Base case:** At beginning, all heights are 0 and each tree has size 1.
  - **Inductive step:** Assume true just before Union(x,y). Let size'(x) and height'(x) (or size'(y) and height'(y)) be the size and length of the merged tree after union, respectively.

Review: Proof of Theorem 1

- By Lemma 1, $size(x) \geq 2^{height(x)}$.
- Let n=size(x) and h=height(x).
- Every node has size $n \geq 2^h$
  => $\log n \geq h$
  => $h = O(\log n)$

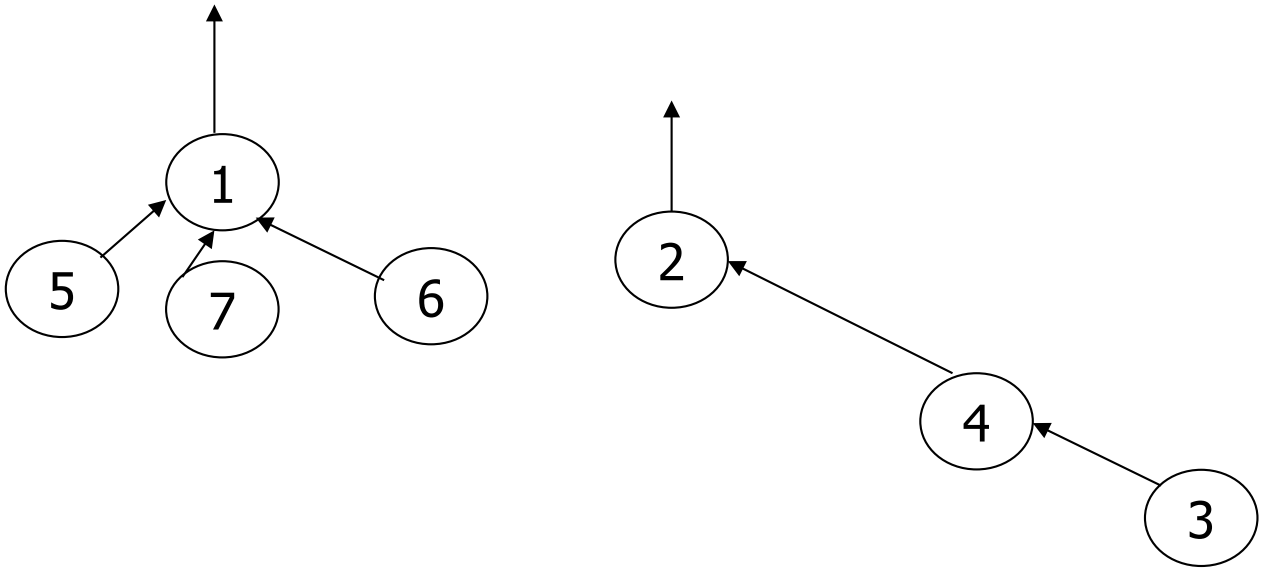
Represent link in a space efficient way



-1	2	-3	4	2	2	-1	-1
0	1	2	3	4	5	6	7

Review: Union by Size

- Smaller size tree link to bigger size tree



- Can also guarantee h=O(logn)
- How to prove?

Review: Path Compression

```
int Find (int element)   {
        if (A[element] < 0)
                return element;
        else
                return A[element] = Find(A[element]);
}
```

Path Compression!

Whenever Find() is performed, all items along the path update its parent to the topmost root

Next query (Find()) will be faster…

Exercise

Show the results of the following sequence of instructions (using an array), we have 10 elements (0-9):

Union(0, 1); Union(2, 3); Union(4, 5); Union(0, 2); Union(5, 6); Union(5, 7); Union(5, 8); Union(2, 4).

We perform them with Union-by-Height, and when the heights are the same, we connect the first tree to the second tree. After executing all these Union() operations, we execute Find() operation for each element.

3	**3**	**3**	**-2**	**3**	**3**	**3**	**3**	**3**	**-1**

Learning Objectives

1. Understand the concept of Disjoint Set
2. Able to analyze time complexities for operations on Disjoint Set
3. Understand the best implementation of Disjoint Set
4. Able to use Disjoint Set to solve problems

D:1; C:1,2; B:1,2,3; A:1,2,3,4

Objective - Suffix Array

- Suffixes
- Exact Pattern Matching
- Suffix Tree and Suffix Array
- Sorting Suffixes

Review: What is suffixes?

Given a string (or text)

$T = t_1\ t_2\ t_3\ \ldots\ t_{n-1}\ t_n$

then it has **n** suffixes,

they are

Suffix(T,i) = S[i]

$= t_i\ t_{i+1}\ t_{i+2}\ \ldots\ t_n$

for

$1 \le i \le n$

T =	innovation
S[1]=	innovation
S[2]=	nnovation
S[3]=	novation
S[4]=	ovation
S[5]=	vation
S[6]=	ation
S[7]=	tion
S[8]=	ion
S[9]=	on
S[10]=	n

Review: Why suffixes?

- Prefix of a string $\mathtt{T = t_1\ t_2\ t_3\ \dots\ t_{n-1}\ t_n}$
  - $\mathtt{Prefix(T,i) = t_1\ t_2\ t_3\ \dots\ t_{i-1}\ t_i}$
- Tricky (keep in mind please)
  - Any substring (or pattern) of **T**, must be a prefix of some suffix from **T**!
  - Example `T = mississippi,`

    `P =       ssip`

    then `P = Prefix( Suffix(T, 6), 4 )`

Review: Exact Pattern Matching

- How do you find the occurrence of a pattern **P** in a text **T**?
  - Test for each **i** whether **P** is a prefix of **Suffix(T,i)**

- Naïve implementation: **O(PT)** time, too slow!
- Knuth-Morris-Pratt (1977, SIAM J. Comput.)
  - Key Point: ignore testing impossible suffixes
  - **O(P)** preprocessing **P**
    - Calculate **Next(k)**: which suffix should try next when the first **k** chars of **P** are matched in the current suffix?
  - **O(P + T)** searching for any text **T**
  - Will be covered in CS 4335

Match Concurrently

- Naïve/KMP test suffixes in sequential order
  - Why not do the testing ***concurrently***?

Match Concurrently (e.g.)

Example

T= `mississippi`

P= `ssip`

```
mississippi
 ississippi
  ssissippi
   sissippi
    issippi
     ssippi
      sippi
       ippi
        ppi
         pi
          i
```

Match Concurrently (e.g.)

Example

T= mississippi

P= ssip

matching **s**

~~mississippi~~

~~ississippi~~

ssissippi

sissippi

~~issippi~~

ssippi

sippi

~~ippi~~

~~ppi~~

~~pi~~

~~i~~

Match Concurrently (e.g.)

Example

T= **mississippi**

P= **ssip**

matching **ss**

~~mississippi~~

~~ississippi~~

ssissippi 

s~~sissippi~~

~~issippi~~

ssippi 

s~~ippi~~

~~ippi~~

~~ppi~~

~~pi~~

~~i~~

Match Concurrently (e.g.)

Example

T= mississippi

P= ssip

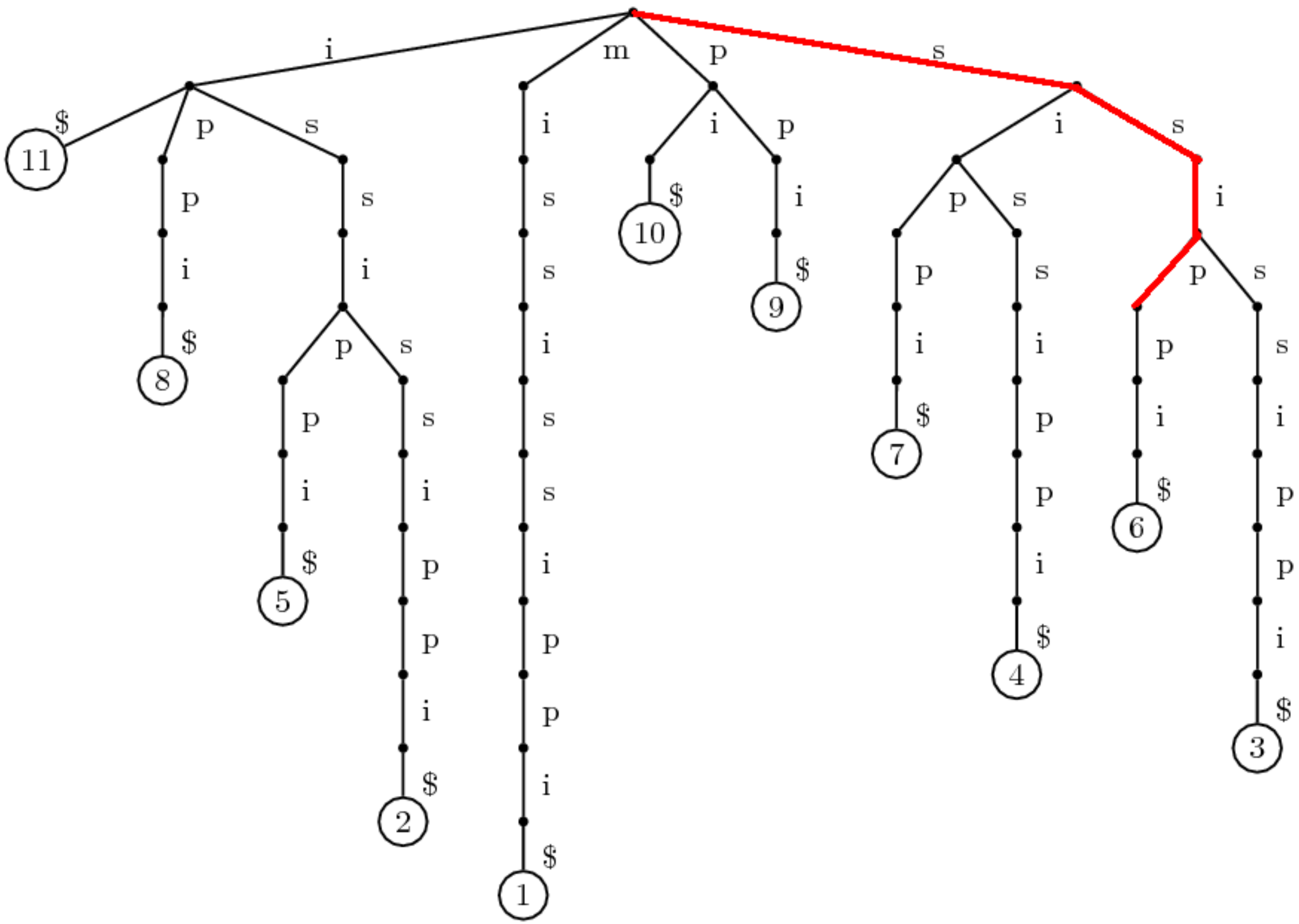
matching **ssip**

~~mississippi~~
~~ississippi~~
~~ssissippi~~
~~sissippi~~
~~issippi~~
ssippi
~~sippi~~
~~ippi~~
~~ppi~~
~~pi~~
~~i~~

Trie (“retrieval”)

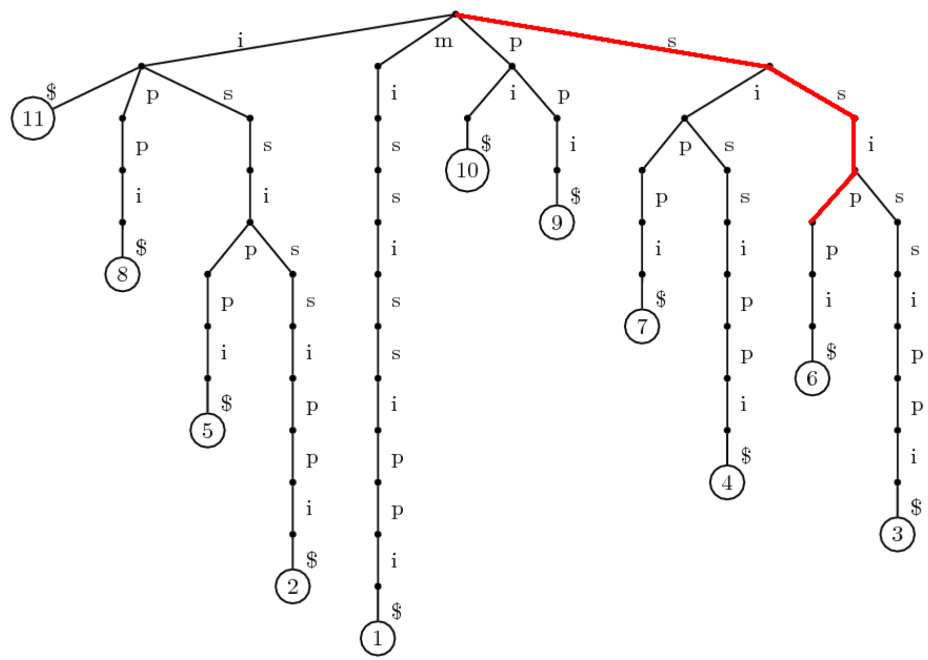
- But we do not have **|T|** CPUs!
- Put all suffixes into a Trie!

A Trie of *Mississippi*



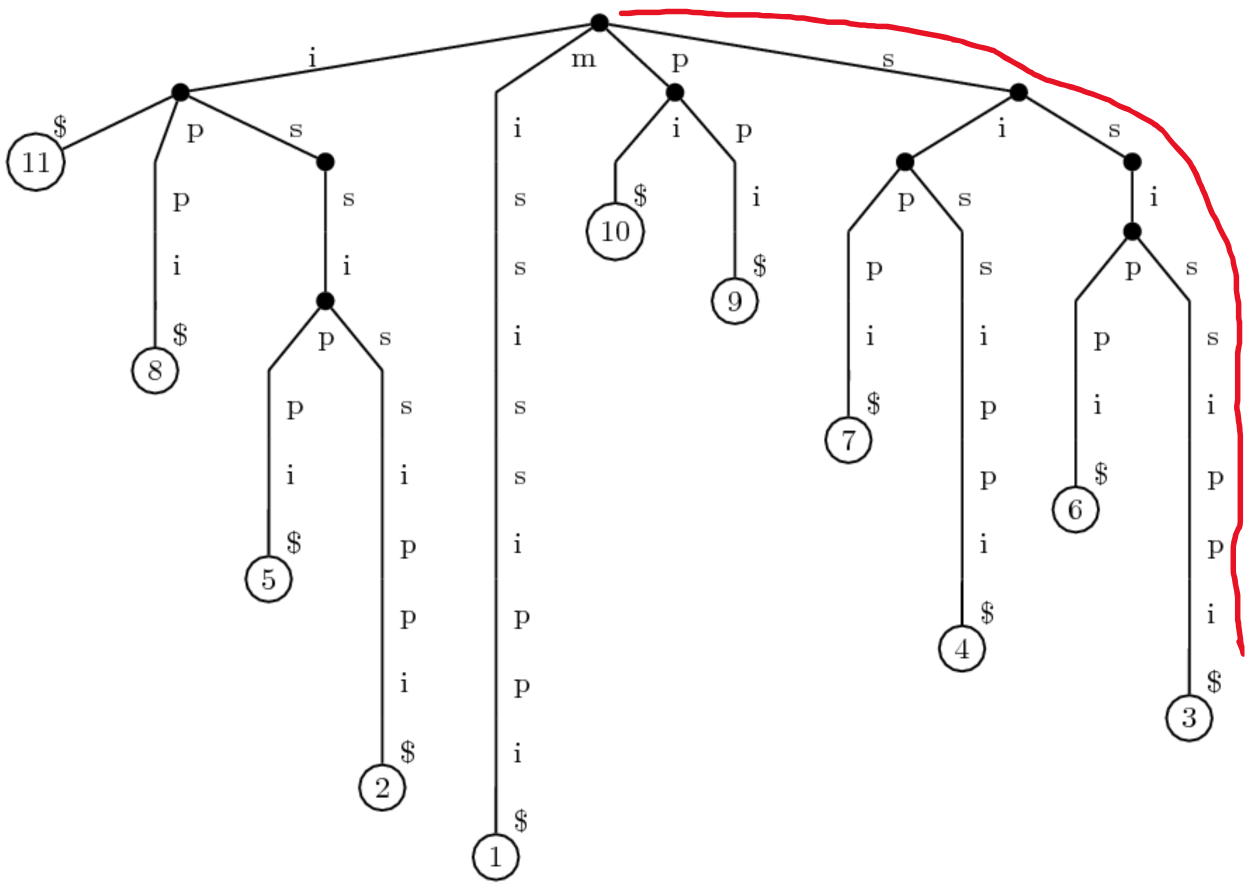
Trie (“retrieval”) (cont.)

- Put all suffixes into a Trie!
  - Every top-down path starts from root corresponds to a substring
  - Those paths ending with a leaf correspond to suffixes
- Complexity
  - Preprocessing **O($\mathtt{T}^2$)**
  - Matching **O($\mathtt{P}$)**



Suffix Tree

- Trie of all suffixes: too many nodes! **$O(T^2)$**
- Suffix Tree
  - Compact Trie
  - **$O(T)$** nodes

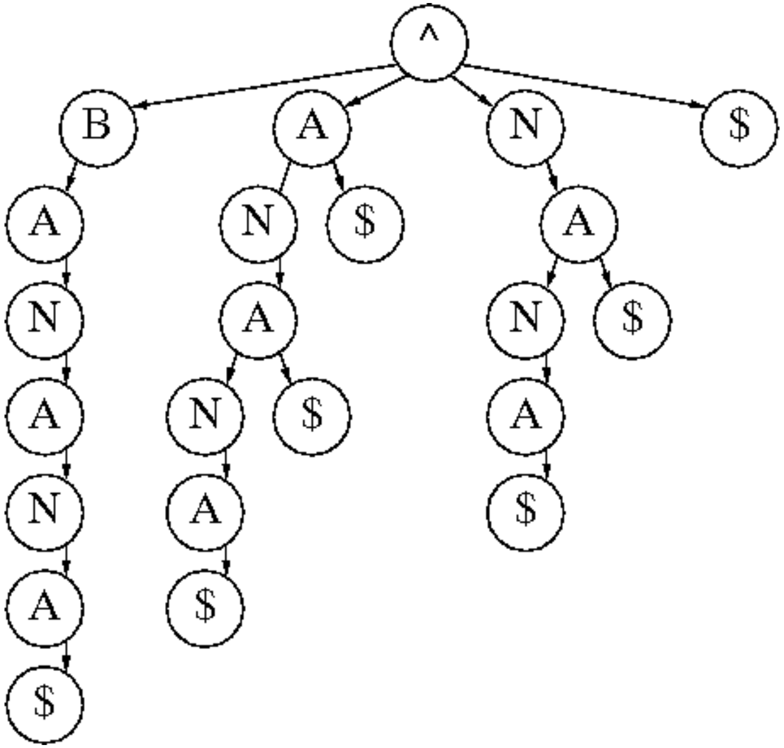


Suffix Tree

- Trie of all suffixes: too many nodes! **$O(T^2)$**
- Suffix Tree
  - Compact Trie of all suffixes
  - **$O(T)$** nodes
  - Compact representation of substring
    - e.g. `T[3..7] = ` $t_3 t_4 \ldots t_7$
  - **$O(T)$** storage

Exercise

Show the suffix tree when searching a pattern *P* on a text T= “banana”.



Match Char by Char

Example

T= mississippi

P= ssip

```
S[ 1]= mississippi
S[ 2]=  ississippi
S[ 3]=   ssissippi
S[ 4]=    sissippi
S[ 5]=     issippi
S[ 6]=      ssippi
S[ 7]=       sippi
S[ 8]=        ippi
S[ 9]=         ppi
S[10]=          pi
S[11]=           i
```

Match Char by Char

Example

T= `mississippi`

P= `ssip`

Sort Suffixes First!

```
S[11]= i
S[ 8]= ippi
S[ 5]= issippi
S[ 2]= ississippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 7]= sippi
S[ 4]= sissippi
S[ 6]= ssippi
S[ 3]= ssissippi
```

Match Char by Char

Suffix Array

T= mississippi

Search **s**

```
S 11 = i
S  8 = ippi
S  5 = issippi
S  2 = ississippi
S  1 = mississippi
S 10 = pi
S  9 = ppi
S  7 = sippi
S  4 = sissippi
S  6 = ssippi
S  3 = ssissippi
```

Binary Search for Lower Bound

Binary Search for Upper Bound

Match Char by Char

Suffix Array

T= mississippi

Search **ssip**

```
S 11 = i
S  8 = ippi
S  5 = issippi
S  2 = ississippi
S  1 = mississippi
S 10 = pi
S  9 = ppi
S  7 = sippi
S  4 = sissippi
S  6 = ssippi
S  3 = ssissippi
```

Match Char by Char

Suffix Array

T= **mississippi**

```
S[11] = i
S[ 8] = ippi
S[ 5] = issippi
S[ 2] = ississippi
S[ 1] = mississippi
S[10] = pi
S[ 9] = ppi
S[ 7] = sippi
S[ 4] = sissippi
S[ 6] = ssippi
S[ 3] = ssissippi
```

Suffix Array

- Suffix Array (SA) : ***sorted indexes*** of all suffixes of a string in lexicographical order.
- Given the Suffix Array of **T**
  - Find all occurrences of **P** by a naïve binary search in **O(P * log T)** time
  - Can be done in **O(P)** time (more advanced topic)

- But how to get the Suffix Array?
  - In another words: How to sort suffixes?

Sorting Suffixes

- QSort or other comparison-based method
  - **$O(n^2 \log n)$**
  - Much faster in practice (real world problem)

- Radix Sort
  - **$O(n^2)$**

- **Doubling Algorithm**
  - Udi Manber and Gene Myers, ***Suffix arrays: a new method for on-line string searches,*** SODA 1990, SIAM J. Comput. 1993
  - **$O(n \log n)$**

- Skew Algorithm (for integer alphabet)
  - Kärkkäinen, Sanders and Burkhardt, ***Linear Work Suffix Array Construction***, Journal of the ACM, 2006
  - **$O(n)$**

L-order

- Definition: $S[i] \le_L S[j]$
  - Use the first **L** chars of each suffixes as **key**

- Examples:
  - ippi $<_2$ issippi
  - ssissippi $=_3$ ssippi
  - ssissippi $>_4$ ssippi

Doubling Algorithm

- Sort by 1-order ($\le_1$)

```
S[ 2]= ississippi
S[ 5]= issippi
S[ 8]= ippi
S[11]= i
S[ 1]= mississippi
S[ 9]= ppi
S[10]= pi
S[ 3]= ssissippi
S[ 4]= sissippi
S[ 6]= ssippi
S[ 7]= sippi
```

Doubling Algorithm

- Sort by 1-order ($\leq_1$)
- Sort by 2-order ($\leq_2$)

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Doubling Algorithm

- Sort by 1-order ($\leq_1$)
- Sort by 2-order ($\leq_2$)

Then…

- Sort by 3-order ($\leq_3$) ?
  - No! This is what ***Radix Sort*** do for general strings.
  - But we are sorting suffixes!
- Sort by 4-order ($\leq_4$) directly

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Extend 2-order to 4-order

- To compare
  - S[3]=ssissippi
  - S[6]=ssippi
- ss is sippi

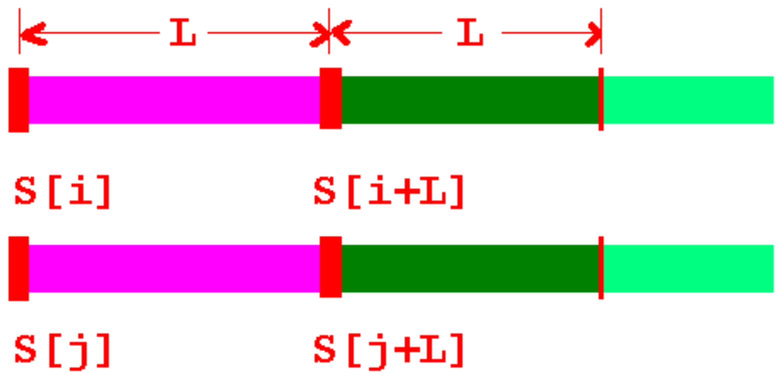
  ss ip pi
- issippi $>_2$ ippi

  from S[5] $>_2$ S[8]

```
S[11]= i
S[ 8]= ippi
S[ 2]= ississippi
S[ 5]= issippi
S[ 1]= mississippi
S[10]= pi
S[ 9]= ppi
S[ 4]= sissippi
S[ 7]= sippi
S[ 3]= ssissippi
S[ 6]= ssippi
```

Extend **L**-order to **2L**-order

- If we have the **L**-order, then **2L**-order could be obtain by
  - $S[i] <_L S[j] \rightarrow S[i] <_{2L} S[j]$
  - $S[i] >_L S[j] \rightarrow S[i] >_{2L} S[j]$
  - $S[i] =_L S[j]$
    - $S[i+L] <_L S[j+L] \rightarrow S[i] <_{2L} S[j]$
    - $S[i+L] >_L S[j+L] \rightarrow S[i] >_{2L} S[j]$



Complexity of Doubling Algorithm

- Doubling Algorithm
  - **L = 1,2,4,8,16,32,…** until exceeds **n**
  - **O(log n)** phases, each phase **O(n)**
  - Total Complexity
    - Time **O(n log n)**
    - Space **O(n)**
- Optimal for general alphabet set

Learning Objectives

1. Understand the concept of Suffix Tree and Suffix Array
2. Able to analyze complexities for sorting suffixes
3. Understand the implementation of Suffix Tree and Suffix Array
4. Able to use Suffix Tree and Suffix Array to solve problems

D:1; C:1,2; B:1,2,3; A:1,2,3,4

Outlines

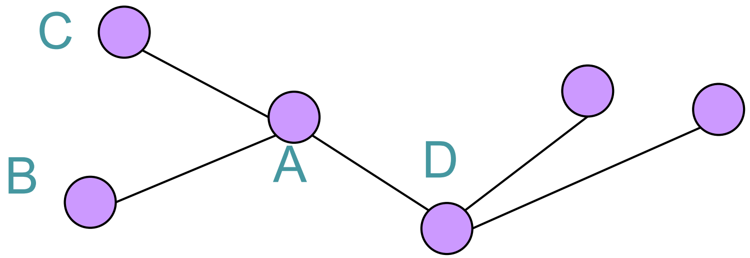
- Terms & Definitions
- Representations of graphs
- Searching graphs
  - BFS – Breath-first search
  - DFS - Depth-first search
- Applications
  - Minimum Spanning Trees
  - Shortest Path

Terms and definitions

- A graph G consists of:
  - A non-empty set of vertices: V
  - A set of edges: E
  - E & V are related in a way that the vertices on both ends of an edge are members of V
  - Usually written as $G = (V, E)$
- Usually, Vertices are used to represent a position or state meanwhile Edges are used to represent a transaction or relationship

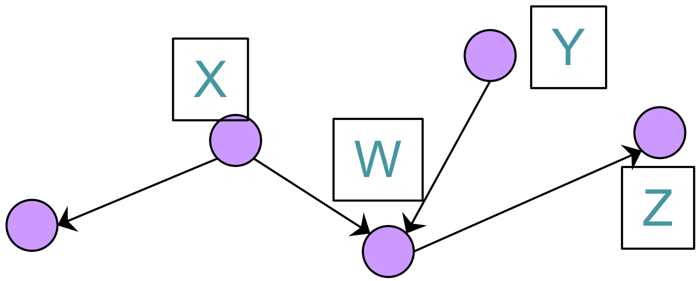
Terms and definitions: Degree

- Degree of a vertex is the number of edges connecting to it



Node A is having a degree of **3** (connects B, C, and D)

- For directed graph, degree is further classified as in-degree *(to this vertex)* & out-degree *(from this vertex)*

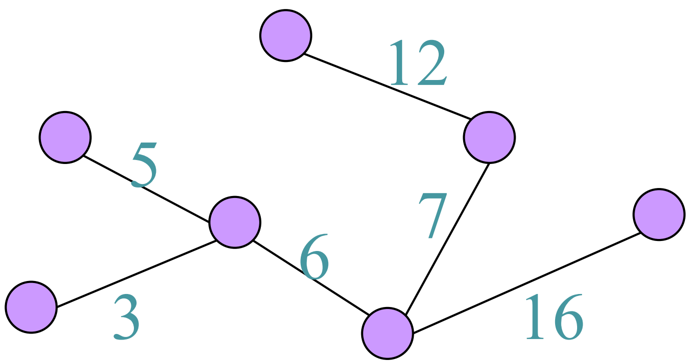


Node W is having an in-degree of **2** (X, Y) and an out-degree of **1** (Z)

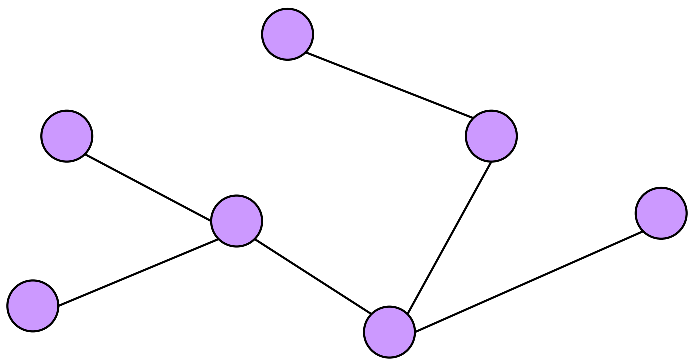
Terms and definitions: Weights

- Graph can be unweighted or weighted, in which a value is associated with each edge.
- In directed graph, the weights of edges going in opposite directions can be different.
- For example:
  - *Whether a bus can go from Shatin to CityU:* Unweighted (=1...)
  - *The bus fee it takes from Shatin to CityU:* Weighted

Weighted graph

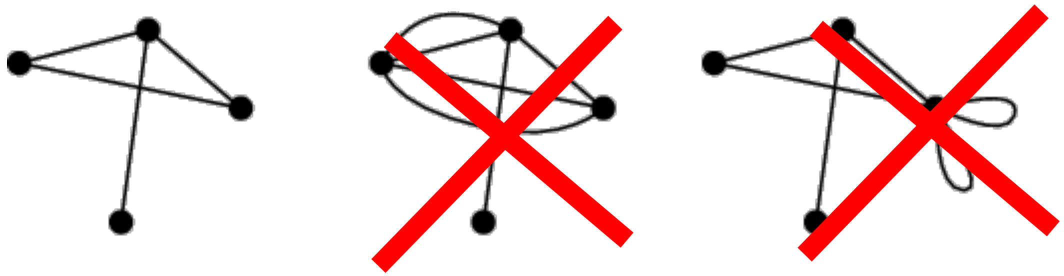


Unweighted graph

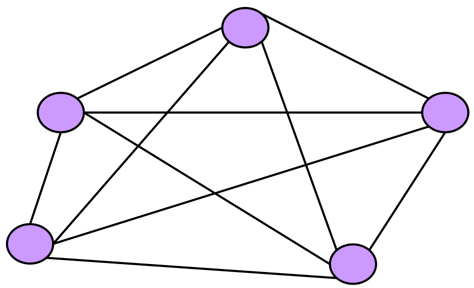


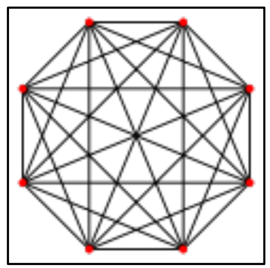
Terms and definitions

- Simple graph:
  - an un-weighted, undirected graph containing no graph loops or multiple edges



- Complete graph:
  - A simple graph in which every pair of vertices are connected directly.
  - If number of vertices (||*V*||) = *n*, number of edges = ?





Representations of graphs

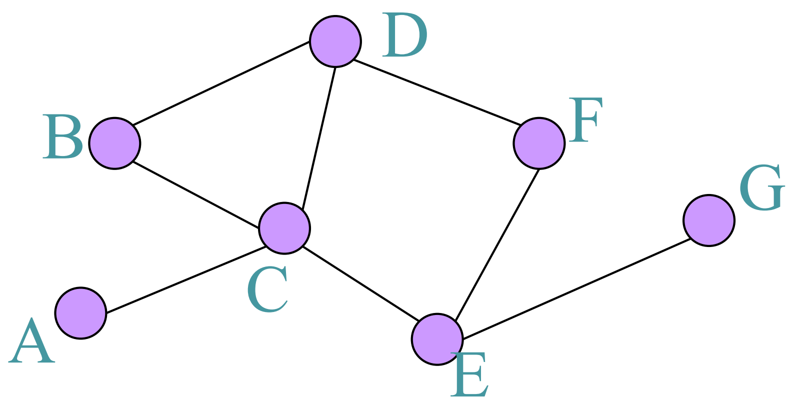
- When working with graph, we always perform one of the following operation:
  - Get the list of vertices connecting a given vertex.
  - Is vertices A & B connected?
  - What is the weight of edge from A to B?
  - What is the in/out degree of a vertex?

- 3 standard representations
  - Adjacency Matrix
  - Adjacency List
  - Compressed Sparse Row (CSR)

Adjacency Matrix

- Use *N*N* 2D array to represent the weight (or T/F) of one vertex to another

An undirected graph



	A	B	C	D	E	F	G
A	0	0	1	0	0	0	0
B	0	0	1	1	0	0	0
C	1	1	0	1	1	0	0
D	0	1	1	0	0	1	0
E	0	0	1	0	0	1	1
F	0	0	0	1	1	0	0
G	0	0	0	0	1	0	0

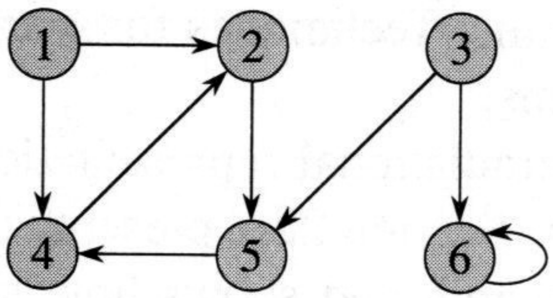
Adjacency Matrix

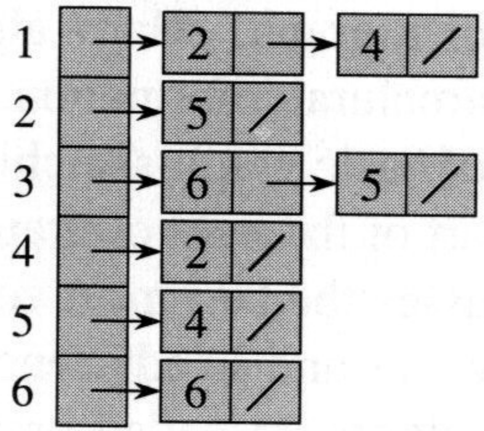
- For undirected graph, only half of the array is used
- Fast query on edge weight & connection

- Total memory used: N^2 (what if num of vertex = 10K?)
- Waste memory if the graph is sparse
  - i.e. #Edge is much smaller than half of $(\#Vertex)^2$, a large proportion of the array will be zero
- Slow when querying the list of neighboring vertices if the graph is sparse

Adjacency List

- Use link list (or equivalent) to store the list of neighboring vertex.
- Save memory if the graph is sparse.
- Query on edge weight / connection can be slow.
- Graph update is slow (especially if one have to maintain order of neighbors)
- Enumeration of all neighbors is fast





Compressed Sparse Row (CSR)

- An efficient way to store sparse matrices or graphs
- The edge array is sorted by the source of each edge, but contains only the targets for the edges.
- The vertex array stores offsets into the edge array, providing the offset of the first edge outgoing from each vertex.

$$\boldsymbol{A} = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

Adjacency matrix

Edge ID: 0 1 2 3 4 5 6 7 8

$$\boldsymbol{edge\text{-}array} = [0, 1, 1, 2, 0, 2, 3, 1, 3]$$

$$\boldsymbol{vertex\text{-}array} = [0, 2, 4, 7, 9]$$

Vertex ID: 0 1 2 3

Representations of graph

- If memory is sufficient and graph update is in-frequent, can represent the graph in all methods at the same time…
  - *If you need fast enumeration of neighbors together with fast query of weight/connection*
  - *e.g. List out all the neighbors of vertex A which do not connect to vertex B or vertex C…*
- Link-list can be replaced by 1D array with count *(enumeration of neighbor and query on degree will be fast)*
- For un-weighted adjacency matrix, you may consider other possibilities other than 0 & 1….

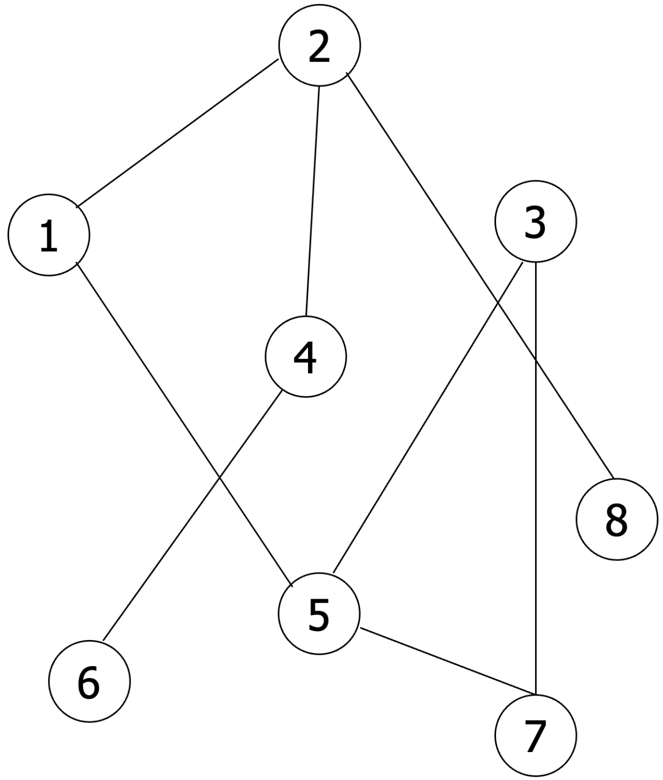
Graph Searching

- To determine whether two vertices are connected (indirectly via some intermediate)
  - A is a relative of B, B is a relative of C, are A & C relative?
- To list out all members of a *connected-component*
  - List out all the direct/indirect family members of A…
- To find the shortest path (of un-weighted graph) from one vertex to another
  - Travel from Shatin to Central with minimum <u>number of changes</u> of transportation…

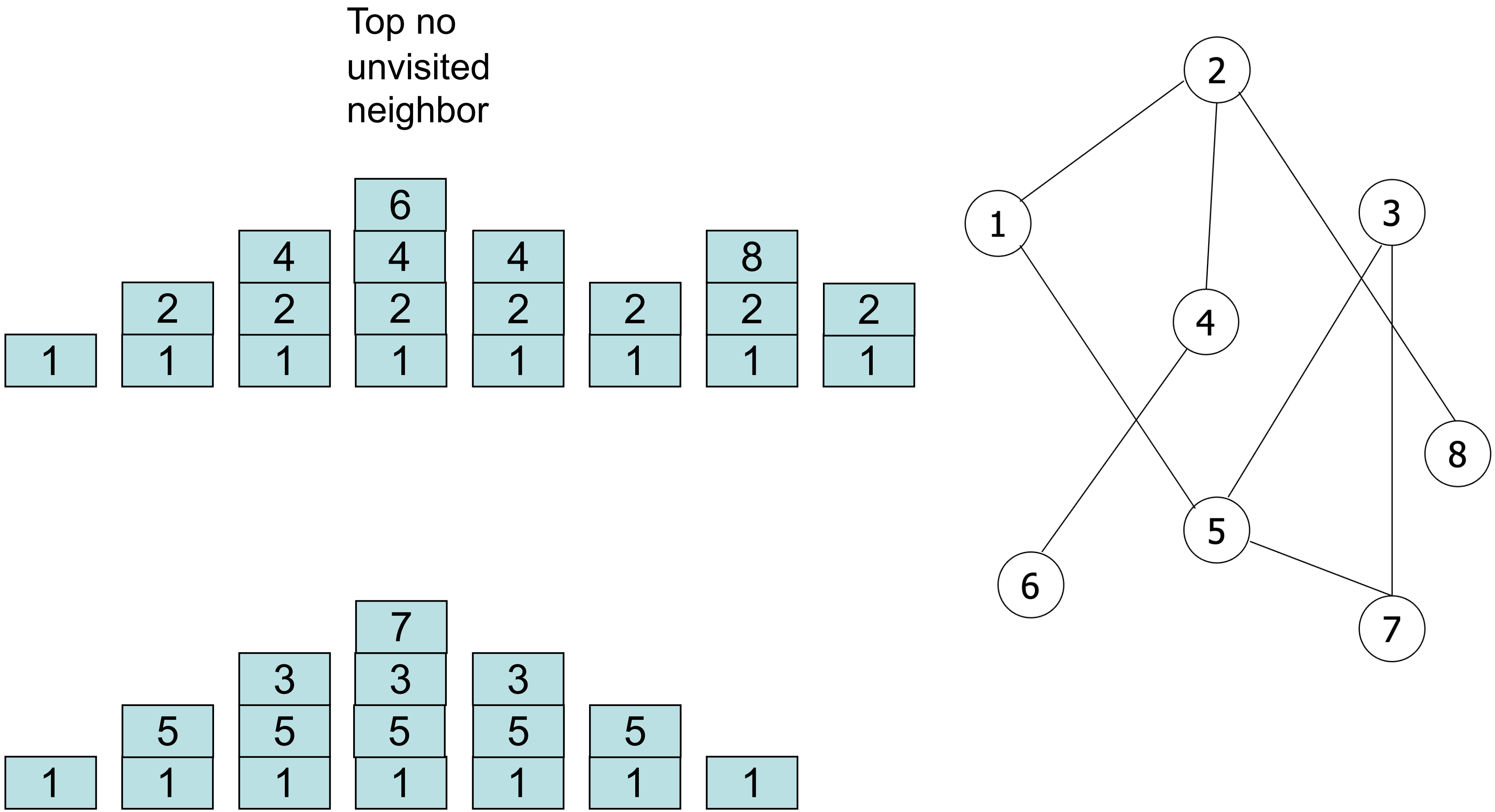
- TWO algorithms:
  - DFS (Depth First Search)
  - BFS (Breadth First Search)

DFS on Graphs

- Go as deep as you can
- Example DFS order (starting from 1):
  - 1,2,4,6,8,5,3,7
  - 1,5,7,3,2,8,4,6
- Using Stack to store nodes
  - Put the starting node into the stack
  - Repeat checking the top
- If top is unvisited
  - Print this element
- If top has unvisited neighbors
  - Push one of the neighbors on stack
- If top has no unvisited neighbors
  - Pop one element



Example

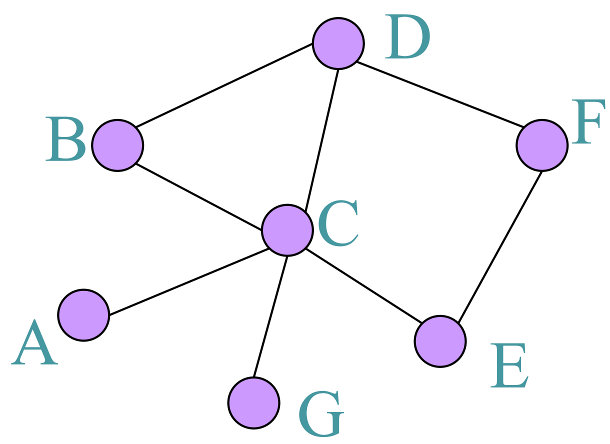


Depth first search (DFS)

- Starts with vertex *v*:

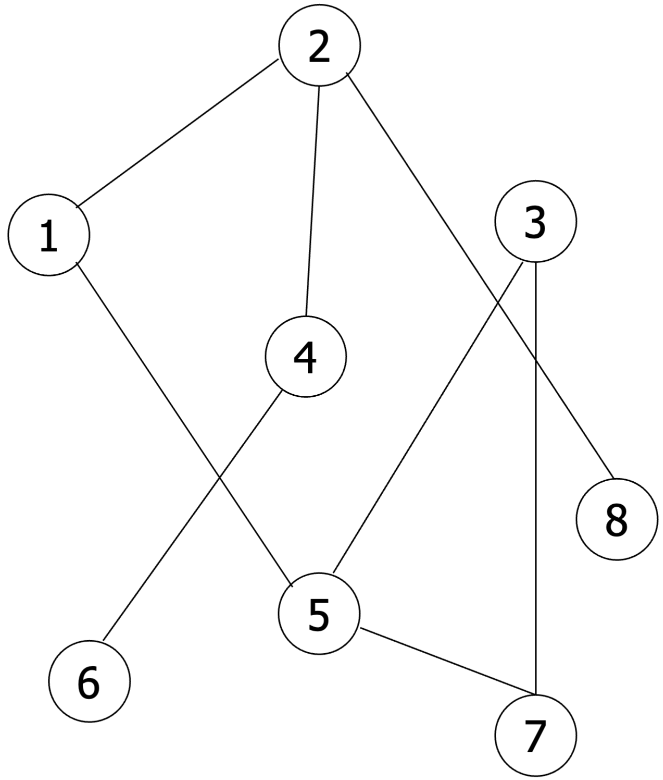
```
DFS (v) {
    visited[v] = true;
    for each vertex w adjacent to v {
      if (! visited[w])
          DFS (w); //Recursion
    }
}
```

DFS(A) = A, C, B, D, F, E, G



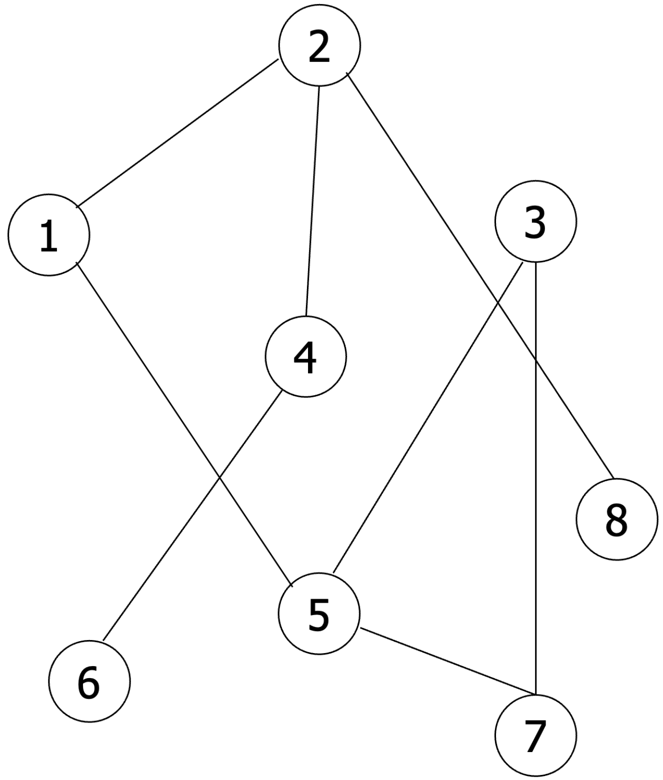
BFS on Graphs

- Go as broad as you can
- Example BFS order (starting from 1):
  - 1,2,5,4,8,3,7,6
  - 1,5,2,7,3,8,4,6
- Using Queue to store nodes
  - Put the starting node into queue
  - Repeat the following "Remove"
- Remove:
  - Remove a node from the queue
  - Print this element
  - Insert all his unvisited (haven't been in the queue) neighbors into the queue

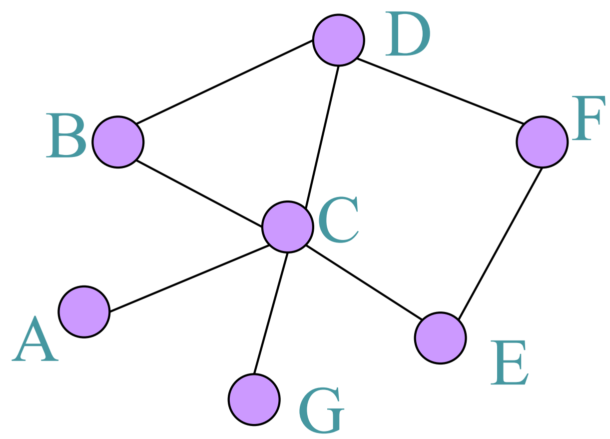


Example

1							
	2	5					
		5	4	8			
			4	8	3	7	
				8	3	7	6
					3	7	6
						7	6
							6



BFS (v)

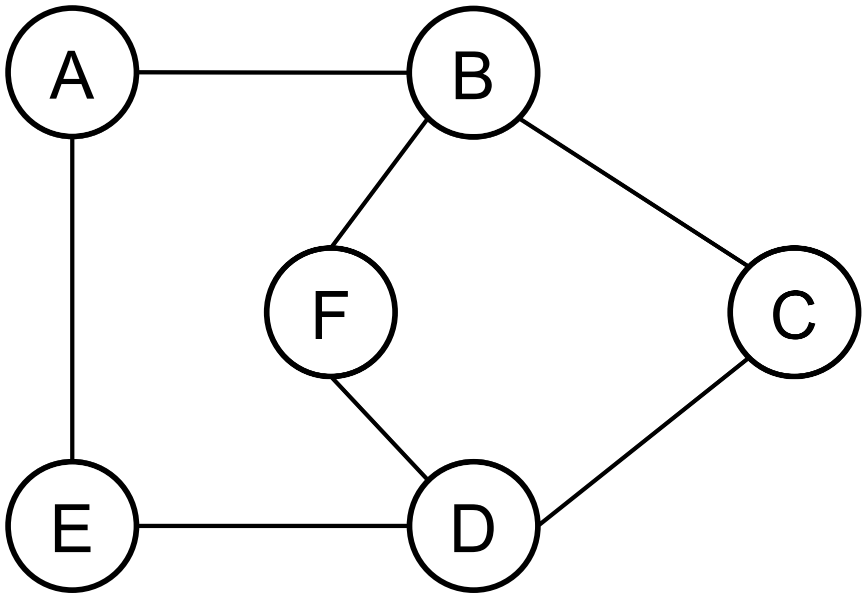



```
BFS(v) {
   visited[v] = true;
   Enqueue(v);
   While queue not empty {
   x = Dequeue();
         for each vertex w adjacent to x {
         if (! visited[w]) {
               Enqueue (w);
               visited[w] = true;
        }
      }
   }
}
```

BFS(A) = A, C, B, D, E, G, F

Exercise

Please illustrates the DFS and BFS graph traversal for the following graph from Node *C*. When there are multiple choices, visit the vertices in alphabetical order



Application 1: Spanning Trees

- Given (connected) graph $G(V,E)$,
  a spanning tree $T(V',E')$:
  - Is a subgraph of G; that is, $V' \subseteq V$, $E' \subseteq E$.
  - Spans the graph ($V' = V$)
  - Forms a tree (no cycle);
  - So, E' has $|V| - 1$ edges

Minimum Spanning Trees (MST)

- Edges are weighted: find minimum cost spanning tree
- Applications
  - Find cheapest way to wire your house
  - Find minimum cost to send a message on the Internet

Strategy for Minimum Spanning Tree

- For any spanning tree T, inserting an edge e_{new} not in T creates a cycle
- But
  - Removing any edge e_{old} from the cycle gives back a spanning tree
  - If e_{new} has a lower cost than e_{old} we have progressed!

Strategy for Minimum Spanning Tree

- Strategy for construction:
  - Add an edge of minimum cost that does not create a cycle (greedy algorithm)
  - Repeat |V| -1 times

It is correct since if we could replace an edge with one of lower cost, the algorithm would have picked it up

Two Algorithms

- Prim: (build tree incrementally)
  - Pick lower cost edge connected to known (incomplete) spanning tree that does not create a cycle and expand to include it in the tree

- Kruskal: (build forest that will finish as a tree)
  - Pick lowest cost edge not yet in a tree that does not create a cycle. Then expand the set of included edges to include it. (It will be somewhere in the forest.)

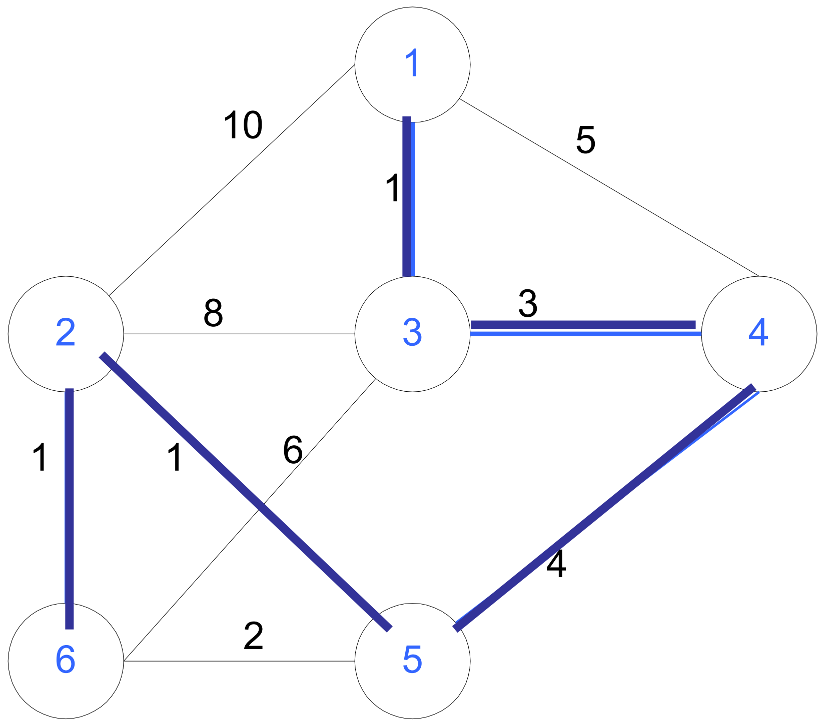
Prim's algorithm

Repeat until all vertices have been chosen

V={1,3,4,5,2,6}

E'={(1,3),(3,4),(4,5),(5,2),(2,6)}

Final Cost: 1 + 3 + 4 + 1 + 1 = 10



Prim's algorithm Implementation

- Assume adjacency list representation
  - Initialize connection cost of each node to "inf" and "unmark" them
  - Choose one node, say v and set cost[v] = 0 and prev[v] =0
  - While they are unmarked nodes
    - Select the unmarked node **u** with minimum cost; mark it
    - For each unmarked node **w** adjacent to **u**

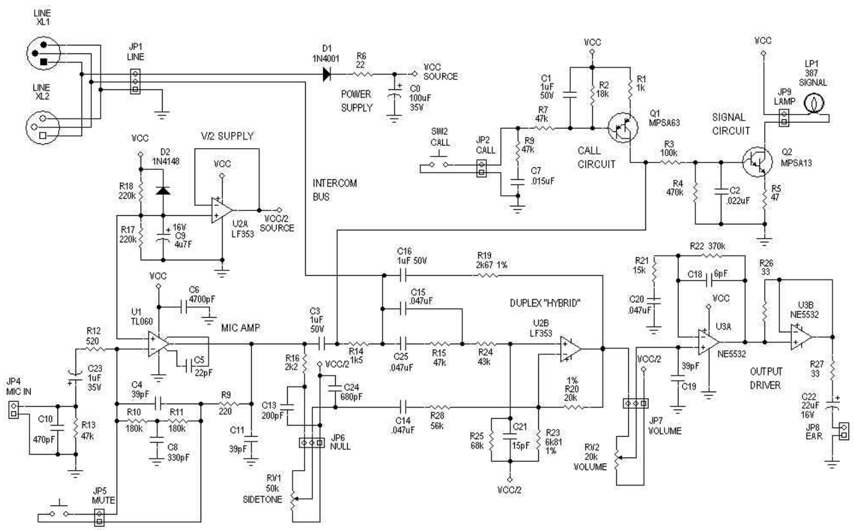
      if cost(u,w) < cost(w) then cost(w) := cost (u,w)

      prev[w] = u

- If the "Select the unmarked node u with minimum cost" is done with binary heap, then O((n+m)logn)

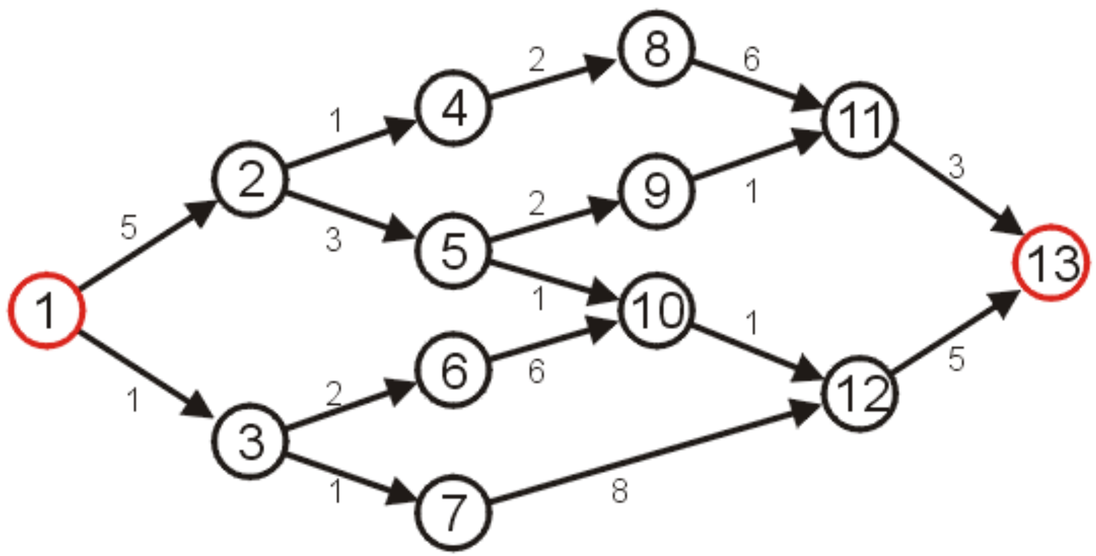
Application 2: Shortest Path

- Given a weighted directed graph, one common problem is finding the shortest path between two given vertices
- Recall that in a weighted graph, the *length* of a path is the sum of the weights of each of the edges in that path
- Application: in circuit design, the time it takes for a change in input to affect an output depends on the shortest path



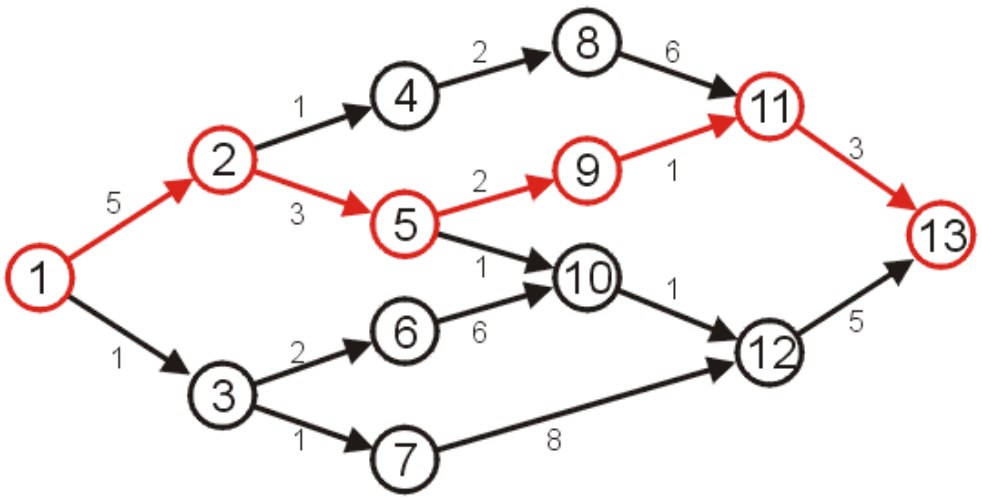
Shortest Path

- Given the graph below, suppose we wish to find the shortest path from *vertex 1* to *vertex 13*



Shortest Path

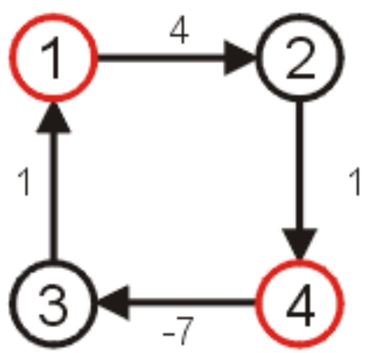
- After some consideration, we may determine that the shortest path is as follows, with length 14



- Other paths exists, but they are longer

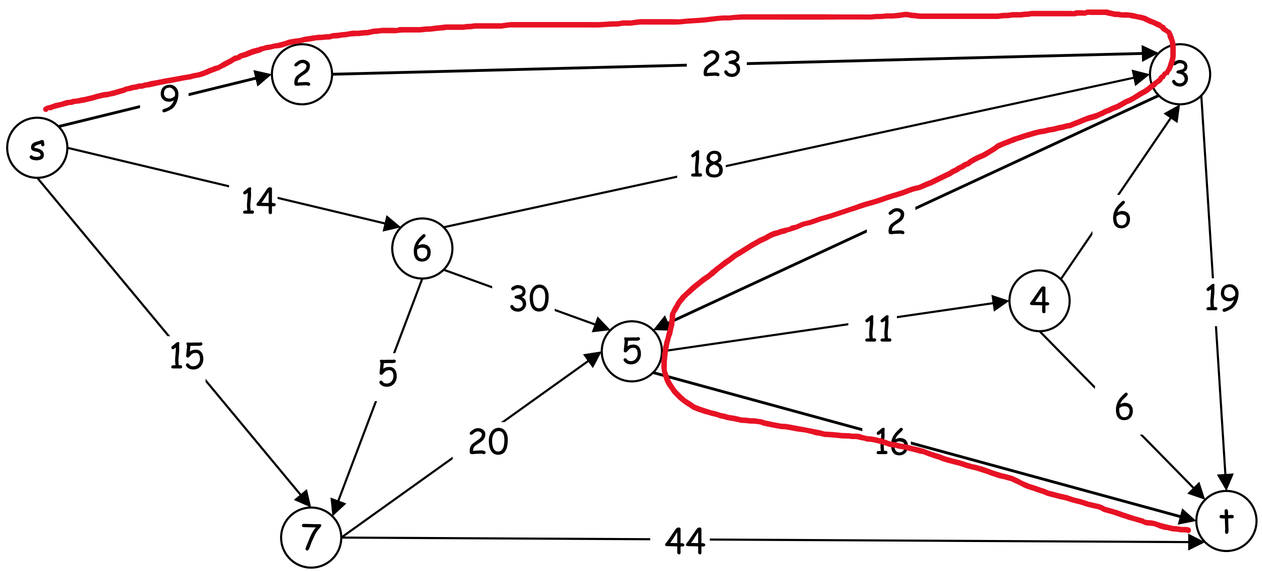
Negative Cycles

- Clearly, if we have negative vertices, it may be possible to end up in a cycle whereby each pass through the cycle decreases the total *length,* thus a shortest length would be undefined for such a graph
- Consider the shortest path from vertex 1 to vertex 4.
- We will only consider non-negative weights.



Shortest Path Example

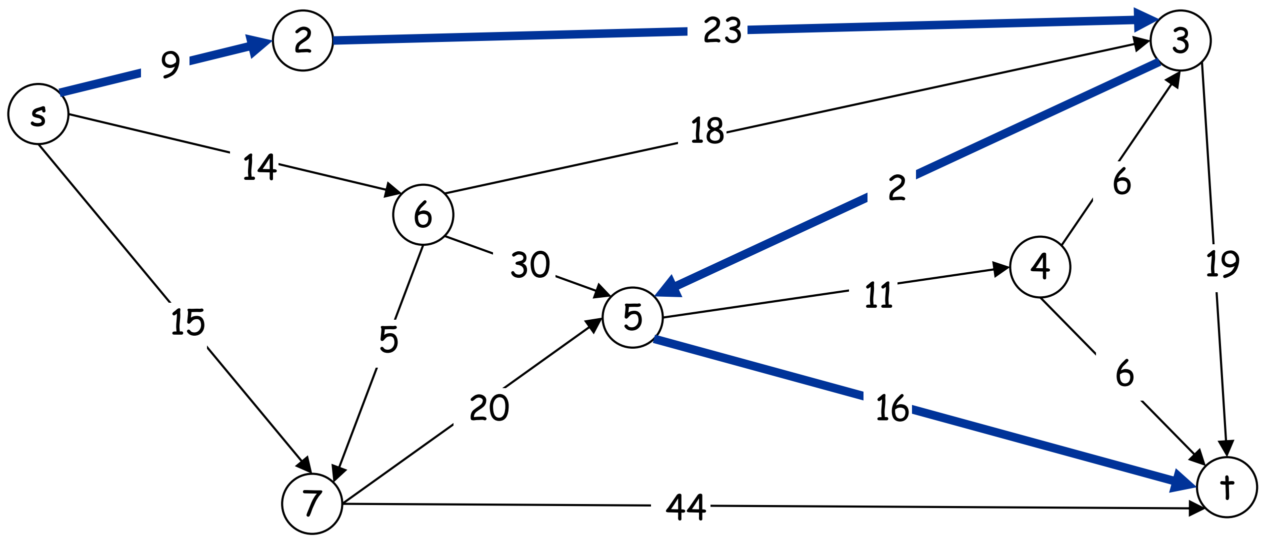
- Given:
  - Weighted Directed graph G = (V, E).
  - Source *s*, destination *t*.
- Find shortest directed path from *s* to *t*.



Cost of path s-2-3-5-t
= 9 + 23 + 2 + 16
= 48.

Discussion Items

- How many possible paths are there from *s* to *t*?
- Can we safely ignore cycles? If so, how?
- Any suggestions on how to reduce the set of possibilities?
- Can we determine a lower bound on the complexity like we did for comparison sorting?



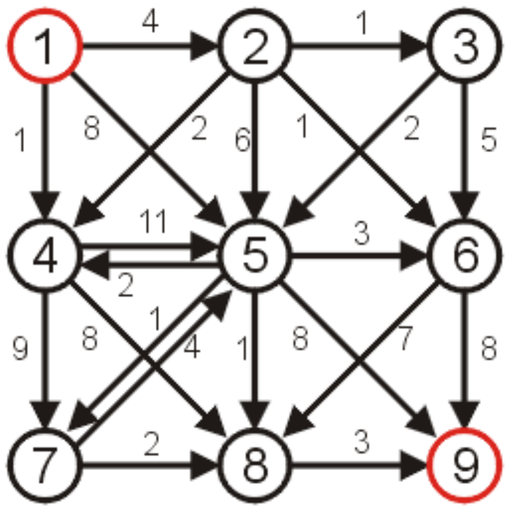
Key Observation

- If the shortest path of *s*->*t* contains the node *v*, then:
  - It will only contain *v* once
  - The path *s* -> *v* must be the shortest path to *v* from *s*.
  - The path *v* -> *t* must be the shortest path to *t* from *v*.

Dijkstra’s algorithm

- Works when all of the weights are positive.
- Provides the shortest paths from a source to **all** other vertices in the graph.

- Consider the following graph with positive weights and cycles.



Dijkstra's algorithm

$d[s] \leftarrow 0$
for each $v \in V - \{s\}$
    $d[v] \leftarrow \infty$
    $p[v] \leftarrow$ undefined
$S \leftarrow \varnothing$
$Q \leftarrow V$ ⊳ Q is a priority queue maintaining $V - S$

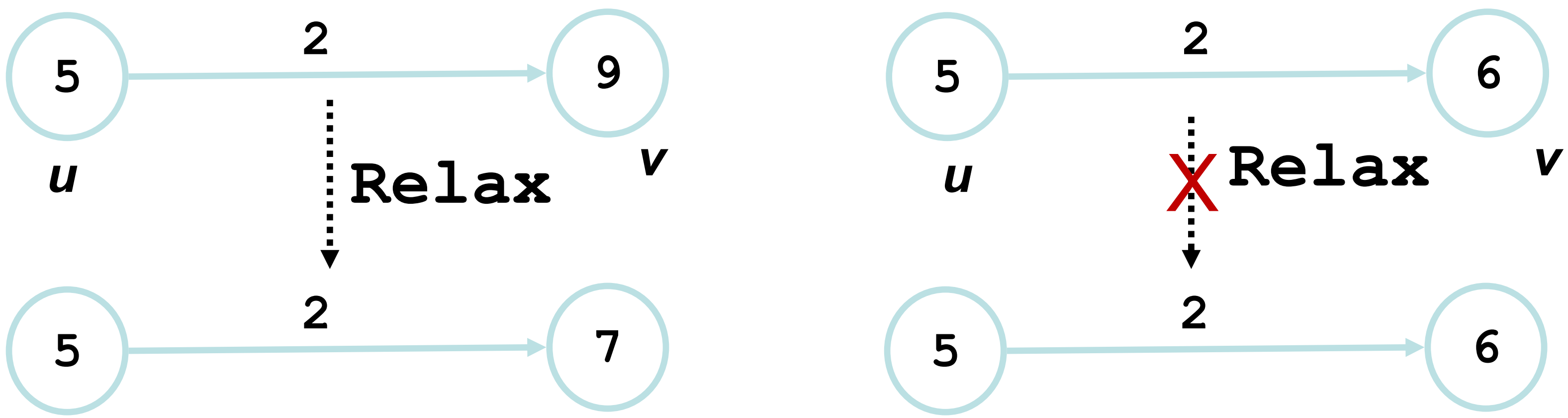
while $Q \neq \varnothing$
    $u \leftarrow$ Extract-Min(Q)
    $S \leftarrow S \cup \{u\}$
    for each $v \in Adj[u]$
        if $d[v] > d[u] + w(u, v)$
            $d[v] \leftarrow d[u] + w(u, v)$
            $p[v] \leftarrow u$

relaxation

Relaxation

- Maintaining this shortest discovered distance d[*v*] is called **relaxation**:

```
Relax(u,v,w) {
      if (d[v] > d[u]+w)
            d[v]=d[u]+w;
}
```



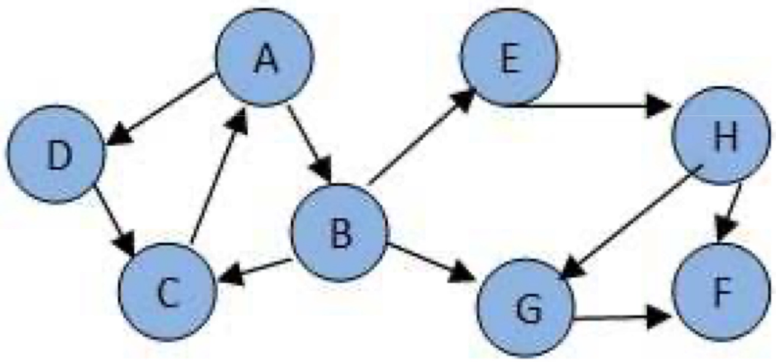
Further topics

- More problems on graphs
  - Shortest path
  - Maximum flow
  - …
  - To be covered by CS3391 and CS4335
- More searching techniques
  - Binary search (not just finding a number)
  - Branch and cut/Branch and bound
  - …

Exercise

Consider the following graph. In what order will the nodes be visited using a BFS? In what order will the nodes be visited using a DFS?

If there is ever a decision between multiple neighbor nodes in the BFS or DFS algorithms, assume we always choose the letter closest to the beginning of the alphabet first.



Learning Objectives

1. Able to do BFS and DFS manually on a Graph
2. Able to do topological sorting
3. Able to write programs to implement BFS&DFS
4. Able to solve minimum spanning tree using disjoint set

D:1; C:1,2; B:1,2,3; A:1,2,3,4