Week 9: Dynammic programming/Graph Algorithms

Agenda:

- LCS
- Basic Graph definitions
- BFS

Reading:

- Textbook: 350-356, 527-540

Longest common subsequence (LCS) problem:

Definitions:

- Sequence/string:
  `dynamicprogramming` is a sequence over the English alphabet
- Base/letter/character
- Subsequence:
  the given sequence with zero or more bases left out
  e.g., `dog` is a subsequence of `dynamicprogramming`
  WARNing: bases appear in the same order, but not necessarily consecutive
- Common subsequence
- LCS problem: given two sequences $X = x_1x_2 \ldots x_n$ and $Y = y_1y_2 \ldots y_m$, find a maximum-length common subsequence of them.

- The LCS problem has the "optimal substructure" ...
  - if x_n is NOT in the LCS (to be computed), then we only need to compute an LCS of $x_1x_2 \ldots x_{n-1}$ and $y_1y_2 \ldots y_m$ ...
  - similarly, if y_m is NOT in the LCS (to be computed), then we only need to compute an LCS of $x_1x_2 \ldots x_n$ and $y_1y_2 \ldots y_{m-1}$ ...
  - if x_n and y_m are both in the LCS (to be computed), then $x_n = y_m$ and we need to compute an LCS of $x_1x_2 \ldots x_{n-1}$ and $y_1y_2 \ldots y_{m-1}$;
    and then adding x_n to the end to form an LCS for the original problem

Longest common subsequence (LCS) problem (cont'd):

- Therefore, we define $DP[i, j]$ to be the length of LCS of $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$; for each $0 \leq i \leq n$ and $0 \leq j \leq m$.

  Letting $DP[n, m]$ to denote the length of an LCS of X and Y, then it is equal to

  $$\text{max length of} \begin{cases} LCS(x_1x_2\ldots x_{n-1},\ y_1y_2\ldots y_m), \\ LCS(x_1x_2\ldots x_n,\quad y_1y_2\ldots y_{m-1}), \\ LCS(x_1x_2\ldots x_{n-1},\ y_1y_2\ldots y_{m-1}) + `x_n', & \text{if } x_n = y_m \end{cases}$$

- Correctness

- In general, let $DP[i, j]$ denote the length of an LCS of $x_1x_2\ldots x_i$ and $y_1y_2\ldots y_j$.

- Recurrence:

  $$DP[i, j] = \max \begin{cases} DP[i-1, j], \\ DP[i, j-1], \\ DP[i-1, j-1] + 1, & \text{if } x_i = y_j \end{cases}$$

- Base cases ???

Longest common subsequence (LCS) problem (cont'd) — solving the recurrence:

- Divide-and-Conquer running time: $\Omega(3^{\min\{n,m\}})$
- Dynamic programming:

  Order of computations ???

procedure dpLCS(X, Y)

$n \leftarrow length[X]$
$m \leftarrow length[Y]$
for $i \leftarrow 1$ **to** m **do**
    $DP[i, 0] \leftarrow 0$
for $j \leftarrow 0$ **to** n **do**
    $DP[0, j] \leftarrow 0$
for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** m **do**
        if $x_i = y_j$ **then**
            $DP[i, j] \leftarrow DP[i-1, j-1] + 1$
        else if $DP[i-1, j] \geq DP[i, j-1]$ **then**
            $DP[i, j] \leftarrow DP[i-1, j]$
        else
            $DP[i, j] \leftarrow DP[i, j-1]$
return $DP[n, m]$

Longest common subsequence (LCS) problem (cont'd):

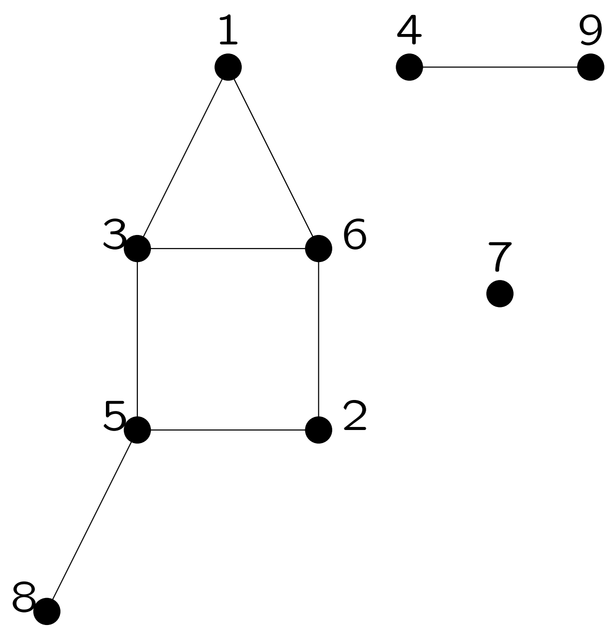
- Correctness
- Can return an associated LCS ... trace back
- Running time: $\Theta(n \times m)$

  There are $n \times m$ entries each takes constant time to compute.

  Can be reduced to $\Theta(n \times \frac{m}{\log m})$ (CMPUT 606)
- Space requirement ... $\Theta(n \times m)$

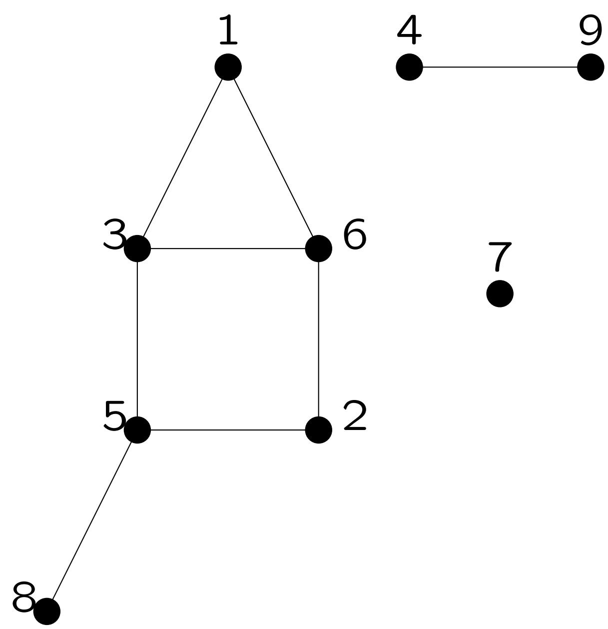
  Can be reduced to $\Theta(\min\{n, m\})$ (CMPUT 606)
- Applications:
  - Human (and other species) Genome Project
  - Detecting cheating :-)

An example:



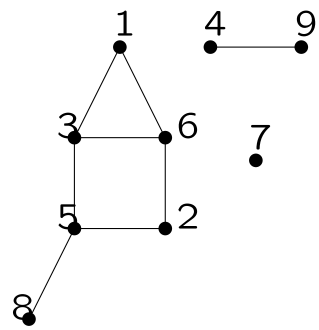
1:	3	6	
2:	5	6	
3:	1	5	6
4:	9		
5:	2	3	8
6:	1	2	3
7:			
8:	5		
9:	4		

An example:



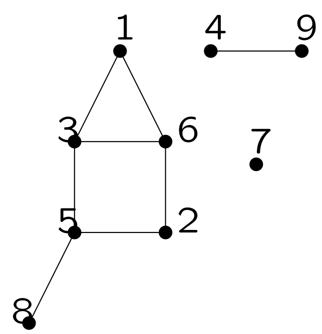
	1	2	3	4	5	6	7	8	9
1			*			*			
2					*	*			
3	*				*	*			
4									*
5		*	*					*	
6	*	*	*						
7									
8					*				
9				*					

Definitions:



- (simple, undirected) graph $G = (V, E)$
  - vertex set V
  - edge set E
    - an edge e is a pair of vertices v_1 and v_2
    - unordered — undirected
    - $v_1 \neq v_2$ — simple and no repeated edges.
- $V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  $E = \{\{1, 3\}, \{1, 6\}, \{2, 5\}, \{2, 6\}, \{3, 5\}, \{3, 6\}, \{4, 9\}, \{5, 8\}\}$
- Notions:
  - adjacent (vertex – vertex, edge – edge)
    e.g., 1 and 3 are adjacent; $(1, 3)$ and $(3, 5)$ are adjacent
  - incident (vertex – edge)
    e.g., 1 is incident with $(1, 3)$

Graph notions:



- Computer representations:
  - adjacency lists
  - adjacency matrix
- Neighborhood of a vertex
- Degree of a vertex — size of its neighborhood
- Walk (vertex – vertex), simple path

  e.g., $\langle 1, 3, 5, 2, 6, 3, 5, 8 \rangle$ and $\langle 1, 3, 5, 2, 6 \rangle$ the former (which has repeated nodes) is a walk and

  the latter is a simple path
- Connected (every pair of vertices is connected via a path)
- Subgraph $G' = (V', E')$ of $G = (V, E)$
  - it is a graph
  - $V' \subseteq V$
  - $E' \subseteq E$
- Connected component (maximal connected subgraph)

Binary equivalence relation:

- A relation $\sim$ involving two elements (in a set A)
  for example, "$\leq$" relation for real numbers
- Reflexive: $a \sim a$ for any $a \in A$
- Symmetric: $a_1 \sim a_2$ iff $a_2 \sim a_1$
- Transitive: $a_1 \sim a_2$ and $a_2 \sim a_3$ imply $a_1 \sim a_3$
- Binary equivalence relation:
  reflexive + symmetric + transitive
  e.g., "$=$" relation for real numbers
- Equivalence class of a

  the subset of elements b such that $a \sim b$

  Therefore, the equivalence class of a contains b implies it is also the equivalence class of b ...
- The equivalence classes form a partition of A
  - union to A
  - disjoint

Connected component:

- A binary equivalence relation $\sim$ on vertex set V

  $v_1 \sim v_2$ iff "there is a path connecting v_1 and v_2"

- The connected component containing vertex v is the equivalence class of v:
  - the connected components form a partition of G, such that
  - no edge crossing the components

Biconnected component:

- Two paths connecting v_1 and v_2 are vertex-disjoint if share no common internal vertex (other than v_1 and v_2).
- Biconnected graph: $|V| \geq 2$, connected, and every pair of vertices are connected via two vertex-disjoint (simple) paths
- Notes:
  - connectivity does NOT implies biconnectivity
  - articulation vertex — cut vertex: its removal disconnects G
  - bridge — cut edge : its removal disconnects G
- Biconnected component — maximal biconnected subgraph
  - a partition of E (not necessarily a partition of V)

More notions:

- Notions on simple, undirected graphs:
  - cycle — a path with two ending vertices collapsed
  - simple cycle
  - acyclic graph — a graph containing no cycles — also called *forest*
  - tree — connected forest
  - complete graph ($|E| = \dfrac{|V| \times (|V| - 1)}{2}$)
    every pair of vertices are adjacent
  - induced subgraph on a subset of vertices, say $U \subset V$
    $(U, E[U])$, where $E[U] = \{(v_1, v_2) : (v_1, v_2) \in E \&\& v_1, v_2 \in U\}$
  - clique (subset of vertices) — the induced subgraph is complete
  - independent set (of vertices) — the induced subgraph contains no edge

- Graph variants:
  - multigraph (remove "simple"), may have loops or parallel edges.
  - digraph (remove "undirected"), every edge is an ordered pair of vertices.
  - edge-weighted graph (every edge has a weight or cost)

More notions:

- The following properies can be proved for a tree:
  - Every tree on n nodes has $n-1$ edges.
  - Every node of degree 1 in a tree is called a leaf; Each tree of size at least 2 has at least two leaves.
  - Adding any edge uv to a tree creates exactly one cycle which consists of the edge uv and the unique path between u and v in the tree.
  - A spanning subgraph is a subgraph containing all the vertices; A spanning tree is a spanning subgraph that is a tree
  - A graph is connected if and only if it has a spanning tree.
- Graph traversal:

  The most elementary graph algorithm:
  - goal: visit all vertices, by following all edges in some order
  - *e.g.*, maze traversal
  - the most common graph traversal with a list storing "waiting" vertices
    1. FIFO list (queue) — breadth first search
    2. LIFO list (stack) — depth first search
    3. recursive — depth first search

Two representations:

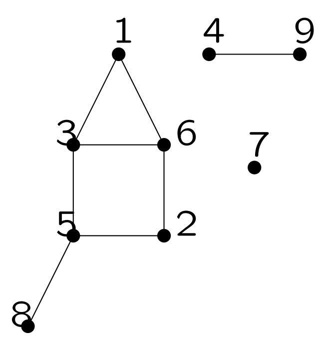
- Adjacency lists: for example,

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

- Adjacency matrix: for example,

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

They both describe the following graph (graphical view):



Breadth First Search (BFS):

- Input: simple undirected graph $G = (V, E)$ and start vertex s
- Output: distance (smallest number of edges) from s to each reachable vertex
  (in a same connected component, if G is not connected)
- Pseudocode:

```
procedure BFS(G, s)          **G = (V, E), s ∈ V start vertex

for each v ∈ V − s do
    c[v] ← WHITE             **unknown yet
    d[v] ← ∞                 **distance from s
    p[v] ← NIL               **predecessor
Q ← ∅                        **waiting vertex queue
enqueue(Q, s)
c[s] ← GRAY                  **in queue Q
d[s] ← 0
while Q ≠ ∅ do
    u ← dequeue(Q)
    for each v ∈ Adj[u] do
        if c[v] = WHITE then
            c[v] ← GRAY
            d[v] ← d[u] + 1
            p[v] ← u
            enqueue(Q, v)
    c[u] ← BLACK             **visited
```

- An example:

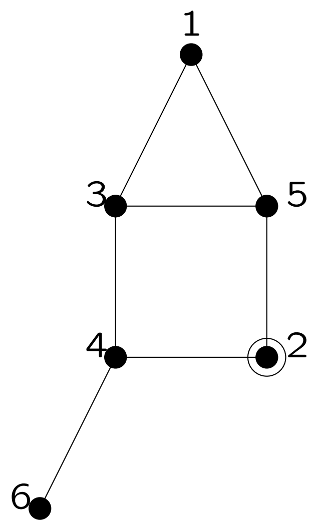
  $V = \{1, 2, 3, 4, 5, 6\}$

  $E = \{\{1, 3\}, \{1, 5\}, \{2, 4\}, \{2, 5\}, \{3, 4\}, \{3, 5\}, \{4, 6\}\}$

  $s = 2$

BFS example:

- $V = \{1, 2, 3, 4, 5, 6\}$
  $E = \{\{1,3\}, \{1,5\}, \{2,4\}, \{2,5\}, \{3,4\}, \{3,5\}, \{4,6\}\}$
  $s = 2$



Adjacency lists:

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

BFS example:

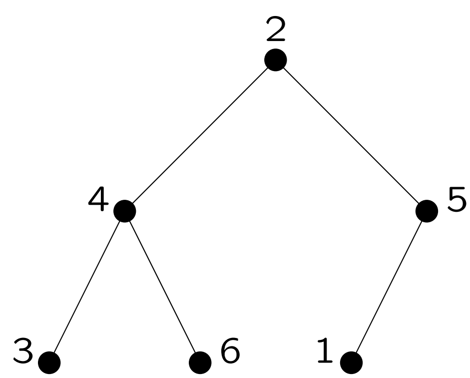
	1	2	3	4	5	6	Q
color	W	G	W	W	W	W	$\{2\}$
distance	∞	0	∞	∞	∞	∞	
parent	NIL	NIL	NIL	NIL	NIL	NIL	
color	W	B	W	G	G	W	$\{4, 5\}$
distance	∞	0	∞	1	1	∞	
parent	NIL	NIL	NIL	2	2	NIL	
color	W	B	G	B	G	G	$\{5, 3, 6\}$
distance	∞	0	2	1	1	2	
parent	NIL	NIL	4	2	2	4	
color	G	B	G	B	B	G	$\{3, 6, 1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	G	B	B	B	B	G	$\{6, 1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	G	B	B	B	B	B	$\{1\}$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	
color	B	B	B	B	B	B	$\emptyset$
distance	2	0	2	1	1	2	
parent	5	NIL	4	2	2	4	

BFS example:

- Adjacency lists:

1:	3	5	
2:	4	5	
3:	1	4	5
4:	2	3	6
5:	1	2	3
6:	4		

- BFS tree:



Notes:

- root is the start vertex s
- parent of x is predecessor $p[x]$
- left-to-right child order *depends* on neighbor ordering (in $Adj[u]$)

BFS analysis:

- $n = |V|$, $m = |E|$
- Handshaking Lemma: $\sum_{v \in V} \texttt{degree}(v) = 2m$
- Analysis:
  - each vertex enqueued exactly once: WHITE $\rightarrow$ GRAY
  - each vertex dequeued exactly once: GRAY $\rightarrow$ BLACK
  - running time:
    1. adjacency list representation:
       $\Theta(n + \sum_{v \in V} \texttt{degree}(v)) = n + 2m) = \Theta(n + m)$
    2. adjacency matrix representation:
       $\Theta(n + \sum_{v \in V} n = n + n^2) = \Theta(n^2)$
  - space complexity:
    1. adjacency list representation:
       $\Theta(n + \sum_{v \in V} \texttt{degree}(v)) = n + 2m) = \Theta(n + m)$
    2. adjacency matrix representation:
       $\Theta(\sum_{v \in V} n = n^2) = \Theta(n^2)$
- BFS product:
  1. every s-to-v shortest path (tracing the parents)
  2. putting these paths together forms the BFS tree
- Warning: vertices in other connected components wouldn't be discovered !!!

  EXERCISE: modify the pseudocode to discover ALL vertices