ch22: Breadth-First Search and Depth-First Search

5 Nov 2013
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

- Huffman coding
- Graph algorithms
  - Breadth-first search
- Depth-first search
  - Parenthesis structure
  - Edge classification
  - Topological sort
  - Finding strongly-connected components

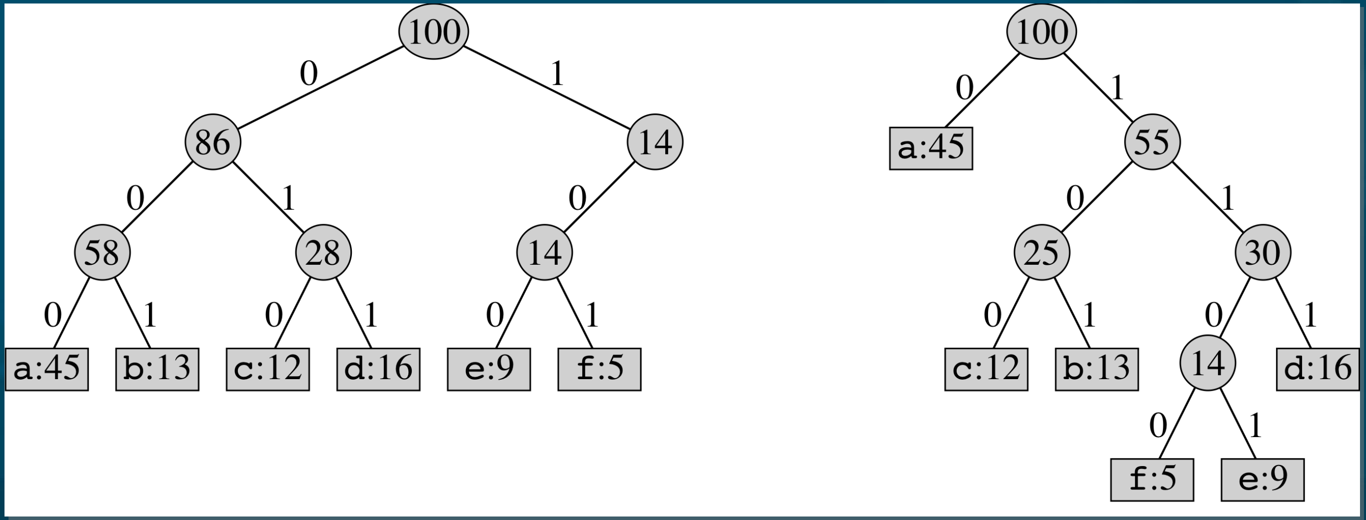
Encoding

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- Given a text with a known set of characters
  - Encode each character with a binary codeword
- Fixed-length code: all codewords same length
  - “cafe” ⇒ 010 000 101 100
- Variable-length code: some codes lower cost
  - “cafe” ⇒ 100 0 1100 1101
  - Compression: choose shorter codes for more frequent characters
- Prefix code: no code is a prefix of another
  - Unique parsing; don't need to delimit chars
  - “cafe” ⇒ 100011001101

Code trees

- Prefix code ⇒ code tree: binary tree where nodes represent prefixes; characters are at leaves
  - Fixed-length code ⇒ leaves all at same level
  - Decoding = walk down the tree
    - Cost of a char = depth in tree
- Total cost of encoding a file using a given tree:
  - Σ_c [freq(c) * depth(c)]



Huffman coding

- Build tree bottom-up
  - Start with two least-common chars
  - Merge to make new subtree with combined freq
- Min-priority queue manages the greedy choice
- Input: array of char nodes with .freq attribs

```
def huffman(chars):
  • Q = new MinQueue(chars)
  • for i in 1 .. length(chars)-1:
    • z = new Node
    • z.left = Q.popmin()
    • z.right = Q.popmin()
    • z.freq = z.left.freq + z.right.freq
    • Q.push(z)
  • return Q.popmin()
```

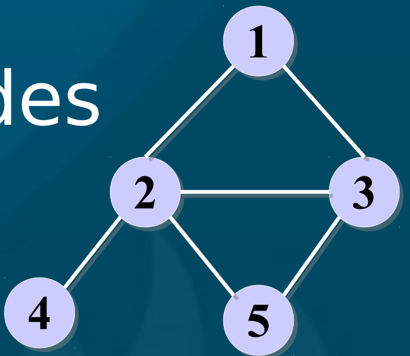
char	freq
a	15
b	5
c	9
d	7
e	18
f	10

Outline for today

- Huffman coding
- Graph algorithms
  - Breadth-first search
- Depth-first search
  - Parenthesis structure
  - Edge classification
  - Topological sort
  - Finding strongly-connected components

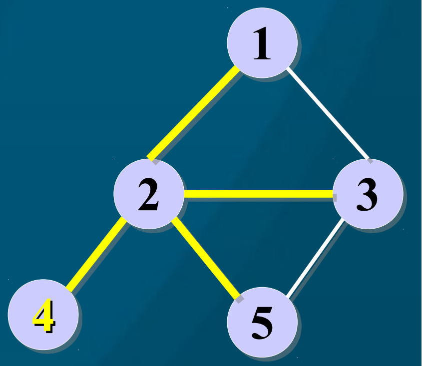
Intro to graph algorithms

- Representing graphs: G = (V, E)
  - V: vertices/nodes (e.g., via array or linked-list)
  - E: edges connecting vertices (directed or un)
- Representing edges:
  - Edge list: array/list of (u,v) pairs of nodes
  - Adjacency list: indexed by start node
    - What about undirected graphs?
    - How to find (out)-degree of every vertex?
  - Adjacency matrix: A[i,j]=1 if (i,j) is an edge
    - What about undirected graphs?
    - Weighted graph: A[i,j] not limited to 0/1



Graph traversal: breadth-first

- Traverse: visit each node exactly once
- BFS: overlay a breadth-first tree
  - Path in the tree = shortest path from chosen start node
  - BFS tree not necessarily unique
- Graph ≠ tree: could have loops
  - ⇒ Need to track which nodes we've seen
- Assign colour: white = unvisited, grey = on border (some unvisited neighbours), black = no unvisited neighbours
- Use FIFO queue to manage grey nodes



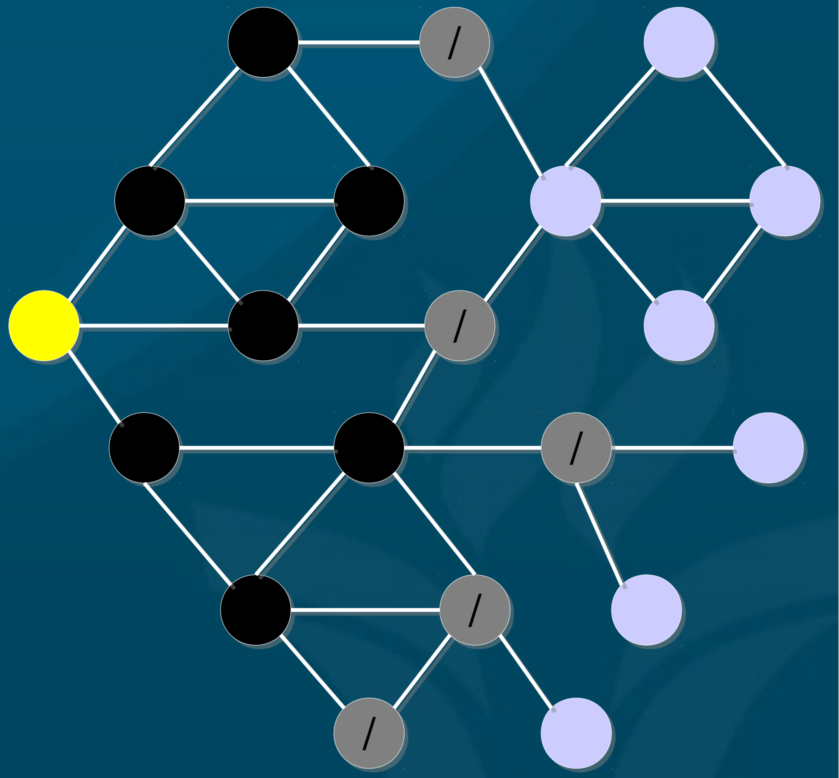
Breadth-first search: algorithm

- Input: vertex list, adjacency list (linked lists), start
- Output: modify vertex list to add parent pointers

V

E

- def BFS(V, E, start):
  - initialise all vertices to be white, with NULL parent
  - initialise start to be grey
  - initialise FIFO: Q.push(start)
  - while Q.notempty():
    - u = Q.pop()
    - for each v in E.adj[u]:
      - if v.colour == white:
        - v.colour = grey
        - v.parent = u
        - Q.push(v)
    - u.colour = black



- Complexity: O(V + E)

Outline for today

- Huffman coding
- Graph algorithms
  - Breadth-first search
- Depth-first search
  - Parenthesis structure
  - Edge classification
  - Topological sort
  - Finding strongly-connected components

Depth-first search

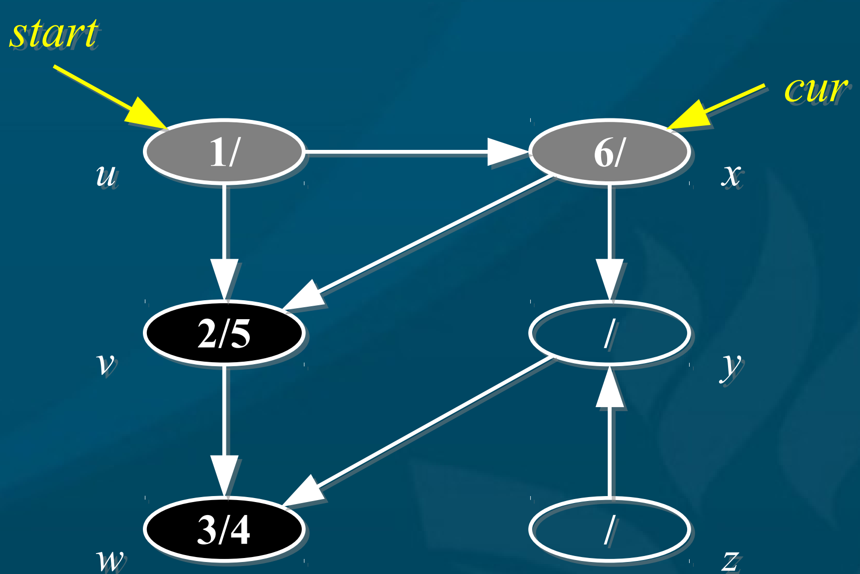
- Explore as deep as we can first
  - Backtrack to explore other paths
  - Recursive algorithm
- Colouring: white = undiscovered
  - Grey = discovered
  - Black = finished (visited all neighbours)
- Add timestamps on discover and finish
- Overlays a forest on the graph
  - Subtree at a node = nodes visited between this node's discovery and finish

Depth-first search: algorithm

- def DFS(G):
  - initialise all vertices to be white, with NULL parent
  - time = 0
  - for u in vertices:
    - if u is white: DFS-Visit(G, u)

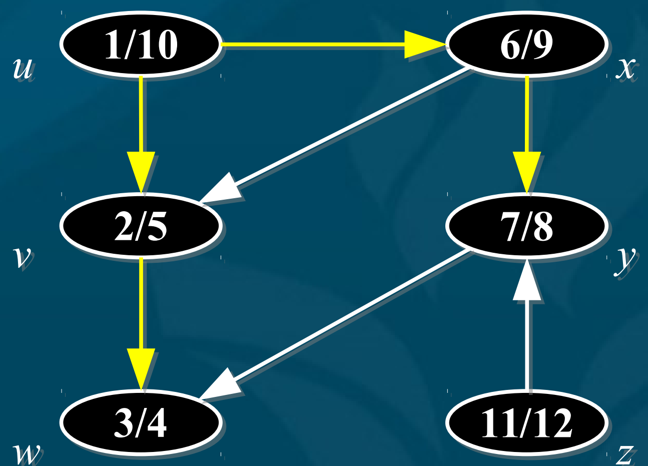
why not just call DFS-Visit once?

- def DFS-Visit(G,u):
  - time++
  - u.discovered = time
  - u.colour = gray
  - for v in u's neighbours:
    - if v is white:
      - v.parent = u
      - DFS-Visit(G, v)
  - u.colour = black
  - time++
  - u.finished = time



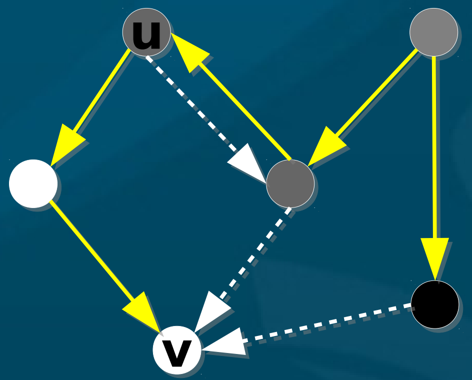
DFS: parenthesis structure

- Subtree at a node is visited between the node's discovery and finish times
- Print a "$(_u$" when we discover a node u, and "$)_u$" when we finish it:
  - Output will be a valid parenthesisation
  - e.g., $(_u\ (_v\ (_w\ )_w\ )_v\ (_x\ (_y\ )_y\ )_x\ )_u\ (_z\ )_z$
  - but not: $(_u\ (_v\ )_u\ )_v$
- The (discover, finish) intervals for two vertices are either:
  - Completely disjoint, or
  - One contained in the other



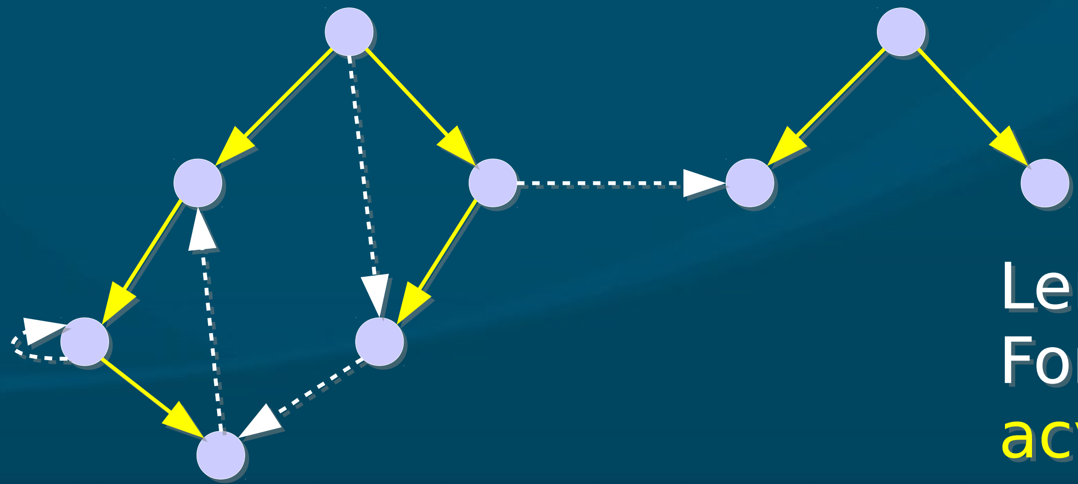
DFS: white-path theorem

- From parenthesis structure: $u.d < v.d < v.f < u.f$
  i.e., the (discover, finish) interval for v is contained / nested within the interval for u,
  $\iff$ v is a descendant of u in the DFS
- White-path theorem:
  v is a descendant of u in the DFS $\iff$
  when u is discovered, there is a path from u → v with only white vertices



DFS for edge classification

- Edges in a graph are either:
  - Tree edges: in the DFS forest
  - Back edges: from a node to an ancestor in the same DFS tree (including self-loop)
  - Forward edges: from a node to a descendant
  - Cross edges: between nodes in different subtrees or different DFS trees



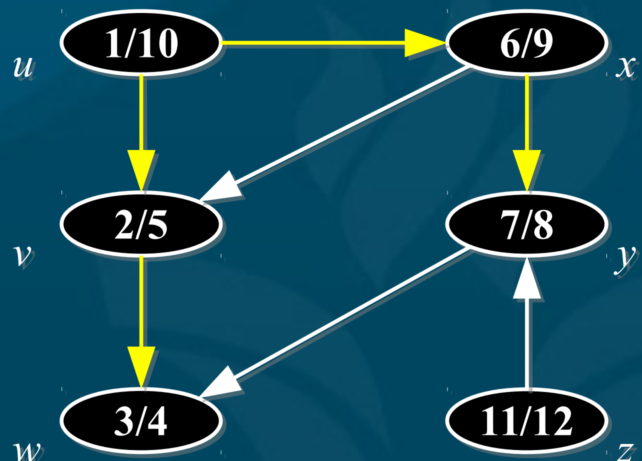
Lemma (22.11):
For directed graphs,
acyclic $\Longleftrightarrow$ no back edges

Outline for today

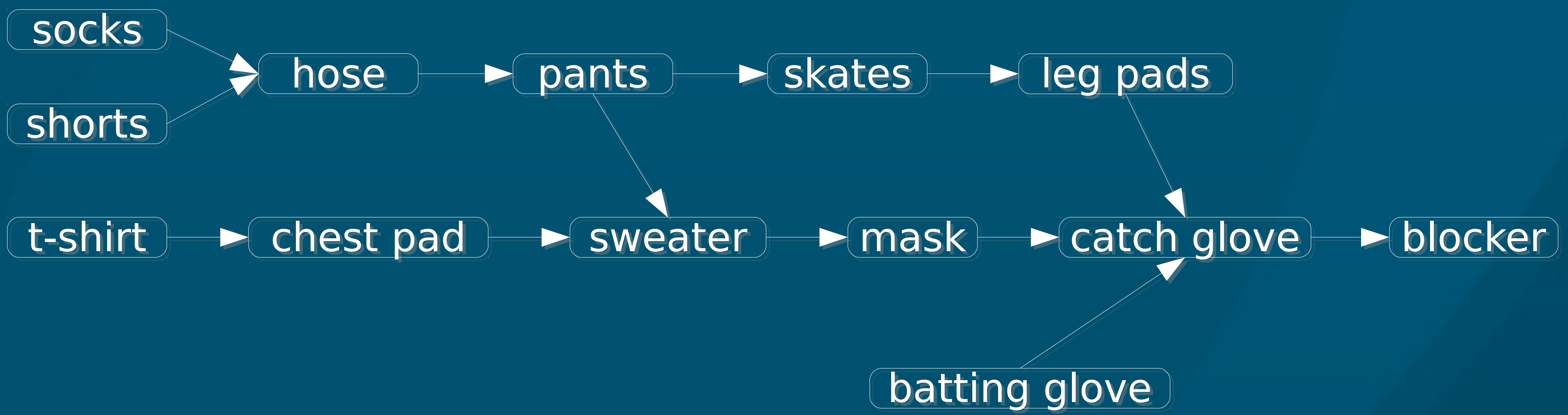
- Huffman coding
- Graph algorithms
  - Breadth-first search
- Depth-first search
  - Parenthesis structure
  - Edge classification
  - Topological sort
  - Finding strongly-connected components

DFS for topological sort

- Linear ordering of vertices such that
  if u → v is an edge, then u comes before v in sort
  - Assumes no cycles! (DAG: directed acyclic)
  - Applications: dependency resolution,
    compiling files, task planning / Gantt chart
- Tweak DFS: as each vertex is finished,
  insert it at the head of a linked list
  - i.e., sort by decr finish time
- e.g.: z, u, x, y, v, w
- DFS might not be unique,
  so topo sort might not be unique



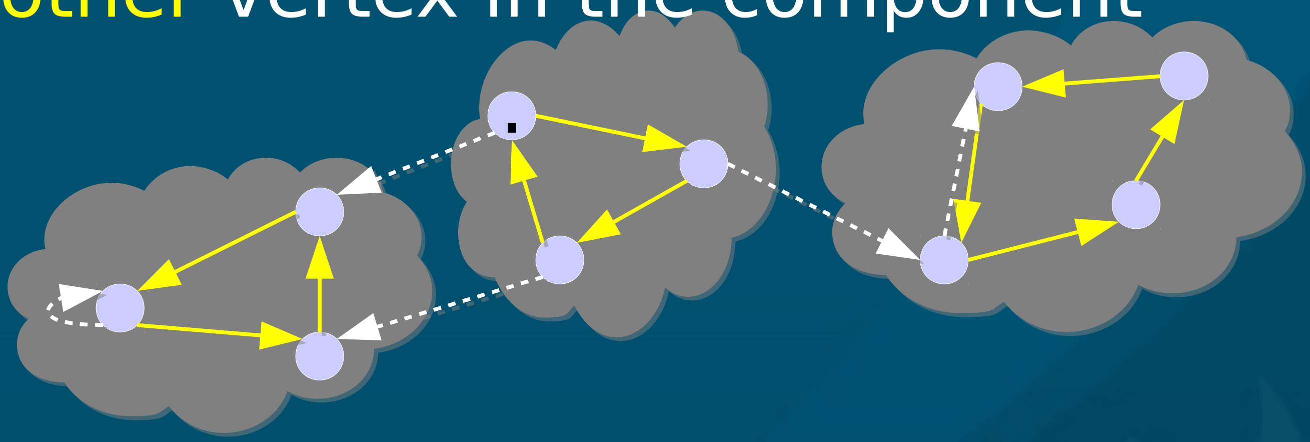
Topo sort: example



- Proof of correctness: $(u,v) \in E \Rightarrow v.f < u.f$
- When DFS explores (u,v), what colour is v?
  - v is gray: means v is an ancestor of u $\Rightarrow (u,v)$ is a back edge $\Rightarrow$ graph has a loop
  - v is white: becomes child: $u.d < v.d < v.f < u.f$
  - v is black: v done, but u not done yet: $v.f < u.f$

DFS for connected components

- Largest completely-connected set of vertices:
  - Every vertex in the component has a path to every other vertex in the component



- Algorithm:
  - Compute DFS(G) to find finishing times
  - Let G^T (transpose) be G with all edges reversed
  - Compute DFS(G^T) starting at vertex with highest finishing time from step 1
  - ⇒ Each tree in DFS(G^T) is a separate component

Connected components

- Original graph G and DFS (DFS tree shaded)
- Transpose graph G^T and $DFS(G^T)$
  - Start DFS at highest-finish (b.fin == 16)
- Combine vertices: component graph