ch22: Graph Algorithms

4 Dec 2012
CMPT231
Dr. Sean Ho
Trinity Western University

Outline for today

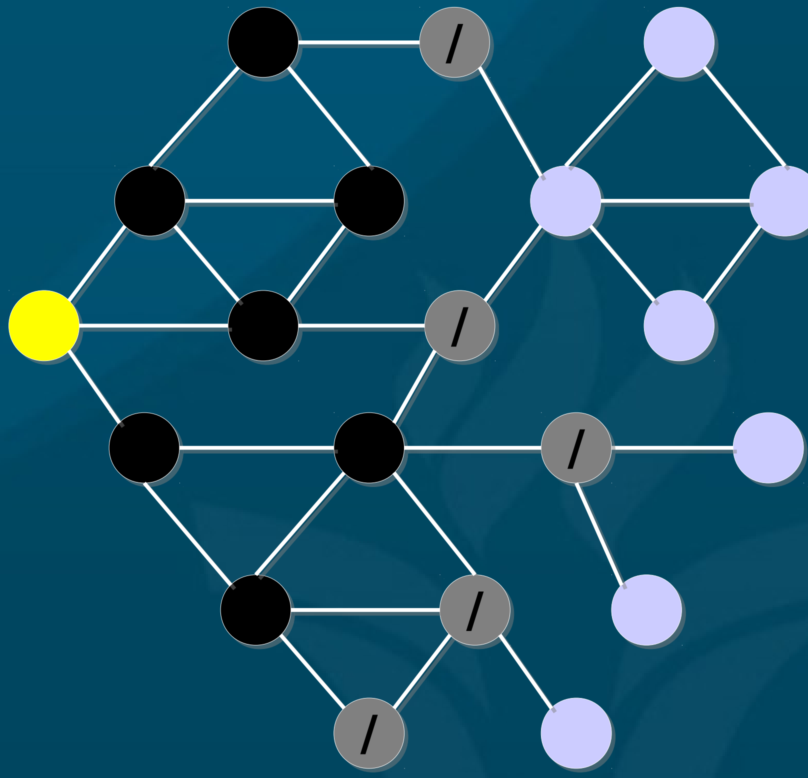
- Depth-first search
  - Parenthesis structure
  - Edge classification
  - Topological sort
  - Finding strongly-connected components
- Semester overview

Breadth-first search algorithm

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

V

E



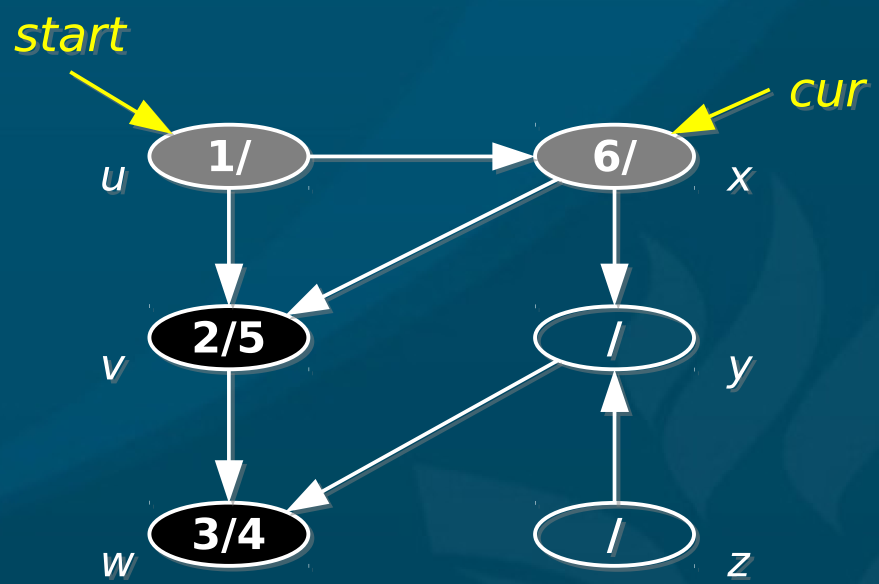
- Complexity: O(V + E)

Depth-first search

- Explore as deep as we can go
  - 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

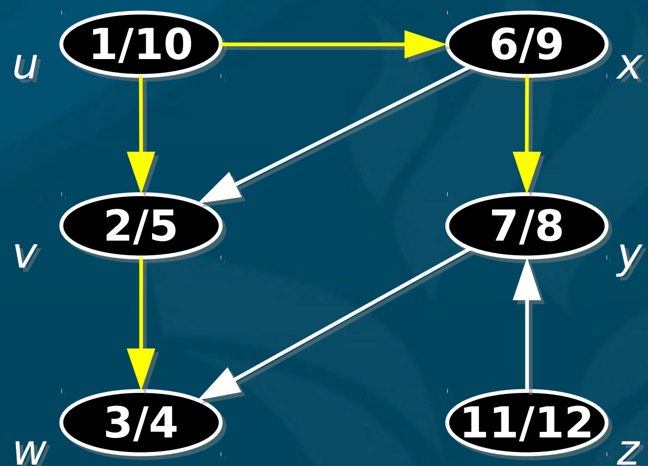
DFS algorithm

- def DFS(G):
  - initialise all vertices to be white, with NULL parent
  - time = 0
  - for u in vertices: ← *why not just call DFS-Visit once?*
    - if u is white: DFS-Visit(G, u)
- 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



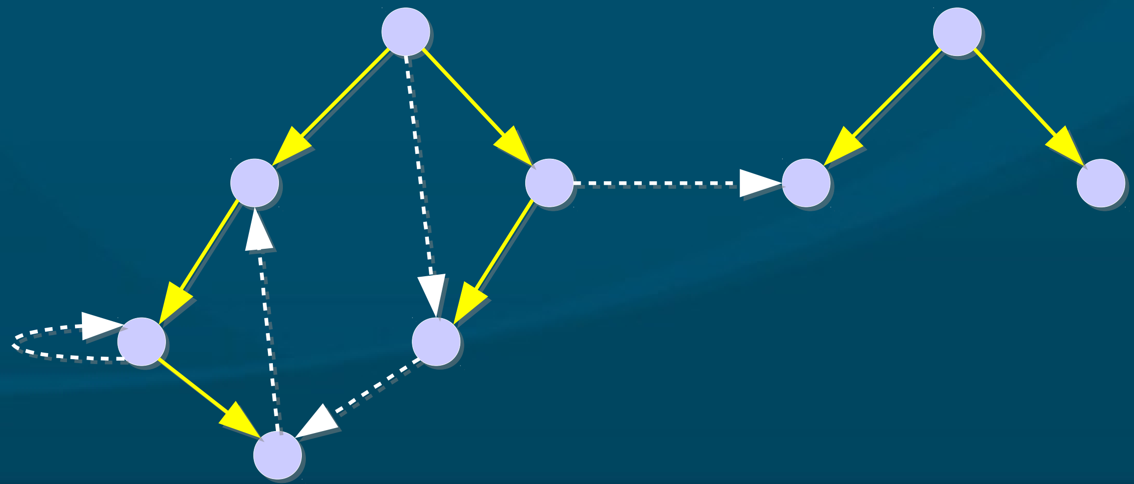
Parenthesis structure

- 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



Edge classification

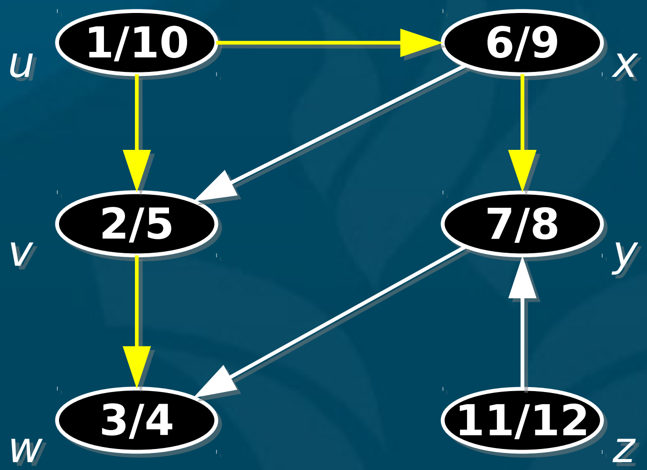
- Edges in a graph are either:
  - Tree edges: in the DFS forest
  - Back edges: from a node to an ancestor in the same DFS tree (including self-loop)
  - Forward edges: from a node to a descendant
  - Cross edges: between nodes in different subtrees or different DFS trees



Lemma (22.11):
For directed graphs,
acyclic ⟺ no back edges

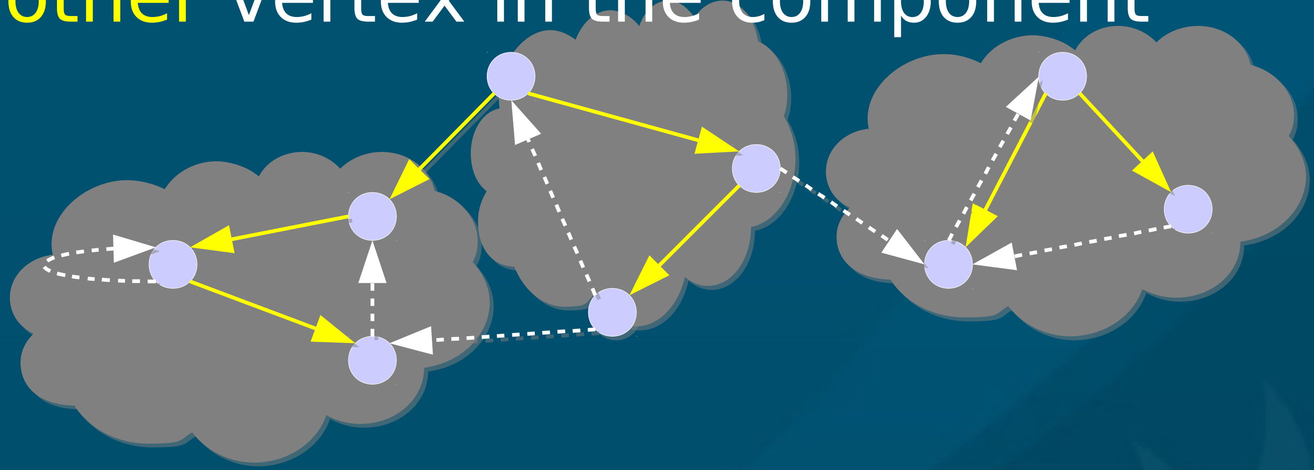
Topological sort

- 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 / Gannt chart
- Tweak DFS: as each vertex is finished, insert it at the head of a linked list
- e.g.: z, u, x, y, v, w
- DFS might not be unique, so topo sort might not be unique



Strongly-connected component

- 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) but iterate over neighbours in decreasing order of finishing time from step 1
  - ⇒ Each tree in DFS(G^T) is a separate component

Outline for today

- Depth-first search
  - Parenthesis structure
  - Edge classification
  - Topological sort
  - Finding strongly-connected components
- Semester overview

Semester overview

- ch1-4: Intro/definitions
  (complexity, recurrences, divide-conquer)
- ch6-8,11: Sorting
  comparison sorts (insertion, merge, heap, quick)
  linear sorts (counting, radix, bucket), hash tables
- ch10,12,18: Data Structures
  (linked lists, stacks/queues, trees, BST, B-trees)
- ch15,16: Algorithms
  (dynamic programming, greedy)
- ch22: Graph algorithms
  (BFS, DFS, topo sort, components)

Exam1: ch1-4

- Algorith. complexity: Θ(=), O(≤), Ω(≥), o(<), ω(>)
  - Know their technical definitions!
  - Proofs!
- Solving recurrences: induction, master method
- Algorithms to be familiar with:
  - Insertion sort, bubble, merge, max subarray
  - Matrix multiply (3 algorithms!)

Exam2: ch6, 7

- Hand-simulation, complexity analysis
  - "What if?" questions: tweaks to std algorithms
- Ch6: Heapsort
  - Trees
  - Max heaps: max-heap property, heapify()
  - Heapsort: building a heap, using it for sorting
  - Priority queue: ops, complexity
- Ch7: Quicksort
  - Naive quicksort with fixed pivot
  - Randomised pivot
  - Complexity analysis: expected running time E[]

Exam2: ch8, 11

- Ch8: Linear-time sorts (assumptions!)
  - Decision tree model, why $\Omega(n \lg n)$ comparisons
  - Counting sort (census + move): $\Theta(n + k)$
  - Radix sort (with r-bit digits): $\Theta(d(n + k))$
  - Bucket sort: $\Theta(n)$ expected time
- Ch11: Hash tables
  - Hash function, hash collisions, chaining
  - Load factor $\alpha = n/(\text{\# buckets})$, search in $\Theta(1+\alpha)$
  - Hashes: div, mul, universal hashing
  - Open addressing: linear, quad, double-hash

Exam3: ch10, 12, 18, 15

- ch10: Linked-lists (dbl, circ), stacks/queues
  - Implementation and complexity
- ch12: Trees (terms, expression trees)
  - BSTs (traversal, search, insert, del)
- ch18: B-trees (motivation, design, variants B*, B+)
  - Operations: search, insert, del
  - Complexity analysis: CPU and disk
- ch15: Dynamic programming
  - Optimal substructure ⇒ bottom-up solution
  - Rod-cutting, Fib, matrix-chain, optimal wt BST

Last chapters: ch16, 22

- ch16: Greedy algorithms
  - Activity selection
  - Fractional knapsack problem
  - Huffman coding
- ch22: Graph algorithms
  - Breadth-first search
  - Depth-first search
    - Edge types, finding cycles
  - Topological sort
  - Finding strongly-connected components