notes for
CS419

an introduction to graph algorithms

primary references:

(Baase) computer algorithms
(McHugh) algorithmic graph theory
(Cormen/Leiserson/Rivest) intro to algorithms
(M. Atallah) alg'ms & theory of comp'n handbook

part I: introduction

- graph theory
  - history and motivation
  - graphs and variants
  - representations
  - basic terminology
  - overview
- algorithmic techniques
  - recursion
  - divide and conquer
  - dynamic programming
  - the greedy method
  - backtracking
  - approximation
  - transformation
  - probabilistic methods
  - integer programming
  - tree-based search (*)
  - geometric algorithms (*)
- data structures
  - stacks and queues
  - priority queues
  - Fibonacci heaps

part II: basic graph algorithms

- traversal and connectivity
  - depth first search
  - breadth first search
  - biconnectivity
  - strong connnectivity
  - randomized connectivity
- minimum spanning trees
  - Boruvka, Dijkstra, Kruskal
- distance and shortest paths
  - unweighted single source
  - single source: Dijkstra
  - digraph single source digraph: Bellman-Ford
  - digraph all pairs: Floyd-Warshall
- acyclic digraphs
  - topological sorting
  - pert analysis
- tours
  - Eulerian tours
  - Hamiltonian cycles
- isomorphism and generation

part III: advanced graph algorithms

- matchings and network flows
  - bipartite matching
  - matching (also randomized)
  - network flow
  - min cut (also randomized)
  - min cost flows
  - multi-commodity flows
- drawing and planarity
- colouring
  - vertex colouring
  - edge colouring
  - approximation algorithms
  - chordal graphs and lexicographic BFS
  - perfect graphs
- other hard problems and approximation algorithms

part IV: selected topics

- treewidth
- dynamic graph algorithms
- parallel graph algorithms
- geometric graph algorithms

References

[A] M. Atallah (ed.), Algorithms and Theory of Computation Handbook, CRC Press (1999) ISBN 0-8493-2649-4

[B] S. Baase & A.V. Elder, Computer Algorithms (3rd ed.) Addison-Wesley (2000) ISBN 0-201-61244-5

[CO] G. Chartrand & O. Oellermann, Applied and Algorithmic Graph Theory, McGraw Hill (1993) ISBN 0-07-557101-3

[Gi] A. Gibbons, Algorithmic Graph Theory, Cambridge University Press (1985) ISBN 0-521-28881-9 [out of print?]

[Go] M.C. Golumbic, Algorithmic Graph Theory and Perfect Graphs, Academic Press (1980) ISBN 0-12-289260-7 [specialized research primer, out of print]

[M] J.A. McHugh, Algorithmic Graph Theory, Prentice Hall (1990) ISBN 0-13-023615-2 [out of print]

[P] E.M. Palmer, Graphical evolution, an introduction to the theory of random graphs, John Wiley & Sons (1985) ISBN 0-471-81577-2 [out of print]

[R] J.H. Reif (ed.), Synthesis of Parallel Algorithms, Morgan Kaufmann (1993) ISBN 1-55860-135-X

[We] D. West, Introduction to Graph Theory, Prentice Hall (1996) ISBN 0-13-227828-6 [grad level]

[Wi] R. Wilson, Introduction to Graph Theory (4th ed.) Longman (1996) ISBN 0-582-24993-7 [inexpensive graph theory primer]

1-1: basic concepts

terminology

- **graph**, vertex, edge, order (n), size (m), trivial graph, adjacent, endpoint, adjacency set (neighbourhood) ADJ(S), isolated,
- **degree**, min(G), max(G), degree sequence,
- **Thm (deg sum)** $\Sigma\, d(v) = 2|E|$
- (spanning/induced) subgraph, union, edge sum, complement,
- **path**, trail, walk, cycle, circuit, length, **connected**, component, disconnected, cut-vertex, bridge, biconnected, block (bicomponent), vertex/edge connectivity, k-connected, k-edge-connected, **distance**, eccentricity, center, radius, diameter,
- **digraph** initial/terminal vertex, adjacent to/from, in/out-degree, di-path/cycle/$\ldots$, semi-path/cycle/ldots, strongly connected, reachable, transitive closure
- **graph variants** multi-, weighted-, loop-, mixed graph

special (di-)graphs

- acyclic, dag, tree, forest, complete, regular, hamlitonian, eulerian, (complete) bipartite, (complete) k-partite (multipartite),

graphs as models

- assignment problem, data flow diagrams

isomorphism

- invariants (degree sequence, cycle lengths, etc)
- complexity? $n!$, backtracking, automorphism,

1-2: representations

- **static** adj. list, adj. matrix, edge list, incidence matrix

1-3: bipartite graphs

- matching, alternating paths, **Hall's thm**, **odd cycle thm**,
- linear recognition (bfs)

1-4: regular graphs

- $m = rn$, **regular supergraph thm**,
- **König thm**: G bipartite implies $\chi'(G) = \Delta(G)$ (implies PRBM)

1-5: maximum matching algorithms later
1-6: planar graphs later
1-7: eulerian graphs later
1-8: hamiltonian graphs later
Chapter 2: Algorithmic Techniques later
read on your own, make sure you recall all techniques

Topic 1: graph traversal

- process of visiting each vertex in graph
- many graph algorithms require traversal
- most general version: traversal with list (below)
- most common variants: BFS, DFS
  - FIFO list (queue): version* of breadth first search
  - LIFO list (stack): version** of depth first search
  - recursive version: depth first search

* only need add never-enqueued neighbours
** push neighbours in reverse order

```
graph_traverse(G)
  for each vertex v do
    visited[v] <- FALSE
  endfor
  for each vertex v do
    if not visited[v] then
      component_traverse(v) endif
  endfor
end_graph_traverse

component_traverse(s)
  list <- {s}
  while not empty(list) do
    remove_from(list,t)
    if not visited[t] then
      visit(t)
      for each neighbour w of t do (*)
        add_to(list,w) endfor
     endif
   endwhile
end_component_traverse

component_DFS(s)
  visit(s)
  for each neighbour t of s do
    if not visited[t] then
      DFS(t) endif
  endfor
end_component_DFS
```

- analysis:
  - adjacency matrix representation
    - space $\qquad\qquad$ $\Theta(n^2)$
    - time $\quad \Theta(n + \sum_x n = n + n^2)$ $\qquad\qquad$ $\Theta(n^2)$
  - adjacency list representation
    - space $\quad \Theta(n + \sum_x d(x) = n + 2m)$ $\qquad\qquad$ $\Theta(n + m)$
    - time $\quad \Theta(n + \sum_x d(x) = n + 2m)$ $\qquad\qquad$ $\Theta(n + m)$

- applications:
  - connected components — any traversal
  - bipartite/odd cycle — any traversal
  - unweighted distance — BFS
  - chordal graph algorithms — lexicographic BFS
  - bi-connected components — DFS
  - strongly connected components — DFS

ordered trees and traversal

- ordered trees
- recursion and procedure call tree
- ordered tree traversal (pre/in/post/level orders)

traversal trees

- for any graph, a component traversal defines an ordered tree, and the edges of the graph associated with the component can be partitioned based on the tree, as follows:
  - breadth first (node level is graph distance)
    - tree edge to child
    - forward edge to child of previous node at same level
    - cross edge within level
  - depth first
    - tree edge to child
    - back edge to ancestor (other than parent)
- dfs: pre-order tree traversal
- bfs: level-order tree traversal
- given a bfs/dfs tree, show the possible other edges

biconnected components

- *cut vertex* — removal increases number of components
  for some other x, y, on every x-y-path
- *biconnected graph* — connected and no cut vertex
- *bicomponent* — maximal biconnected subgraph
- *rooted tree* — tree with one root vertex
- in a rooted tree,
  - *ancestor* of v — any vertex on v-to-root path
  - *proper ancestor* — ancestor other than vertex itself
  - z *descendant* of v — iff v ancestor of z
  - *proper descendant* — descendant other than vertex itself
  - *parent* of v — neighbour of v on v-to-root path
  - z *child* of v — iff v parent of z
- v cut vertex iff, w.r.t. dfs tree

root: more than one subtree

not root: child subtree has no back edge to proper v-ancestor

finding bicomponents via depth first search

- algorithm: for each v, for each child w, keep track of furthest back edge from w-subtree
- in component traversal, each v is encountered deg(v) times
- in dfs component traversal,
  - 1st w encounter: tree edge created from parent
  - last w encounter: subtree traversed, back up to parent
- how to implement above algorithm using dfs?
  - 1st encounter of child w of parent v
    - recurse from w
  - last encounter of w, just before backing up to v
    - check whether v cuts off w-subtree
  - maintain `num`, `back`, `parent` for each v
    - update `back` when backedge 1st encountered
    - update `back` when backing up
  - maintain edge stack
    - push edge when edge 1st encountered
    - pop edges when cutpoint discovered
  - **Warning:** this algorithm differs slightly from that of Baase (2nd ed'n) She pushes each back-edge once and each tree-edge twice; edges are popped until the edge initially on top reappears; this avoids the need for maintaining a `parent` array.

```
bicomponents()    (*version RBH99, differs from Baase*)
  empty stack; dfn <- 0
  for all v do
    parent[v] <- 0; num[v] <- 0; back[v] <- n+1
  for all v do
    if num[v]=0 then bidfs(v)
end_bicomponents

bidfs(v)
  inc(dfn); num[v] <- dfn; back[v] <- dfn
  for each neighbour w do
    if num[w]=0 then                     (*1st w encounter*)
      push [vw]; parent[w] <- v                (*tree edge*)
      bidfs(w)
      (* backup up from w to v*)
      if back[w] >= num[v] then  (*v root or cuts off w? yes*)
        print ‘new bicomponent’
        repeat: pop and print edge
        until  popped edge is [vw]
      else                        (*v root or cuts off w? no*)
        back[v] <- min {back[v],back[w]}
      (*end backup from w to v*)
    elsif num[w]<num[v] and w<>parent[v] then  (*back edge*)
      push [vw]; back[v] <- min {num[w],back[v]}
end_bidfs
```

- example trace: execute **`bidfs(4)`** on the graph below, assuming no previous **`bidfs()`** calls (answer on the next page)

```
                 1: 2 5 7 8    dfs(4) tree  4
   2     5       2: 1 8                    /
  / \   /        3: 6 7 9                 7
 /   \ /         4: 7 8                  / \
8-----1      9   5: 1                   1   3
|     |     /|   6: 3 9                / \   \
|     |    / |   7: 1 3 4             2   5   6
4-----7---3--6   8: 1 2 4            /         \
                 9: 3 6             8           9
```

- correctness? the truth is out there ☺
- complexity? (our/Baase versions)
  - time: constant for each vertex/edge encounter
    $\Theta(c_1 n + c_2 \sum_v deg(v) = c_1 n + 2c_2 m) \;=\; \Theta(n+m)$
  - space: assume adjacency list representation
    - graph, arrays of size n, edge stack, runtime stack
    - edge stack: $O(m)$ since each edge pushed once (our version) at most twice (Baase version)
    - runtime stack: $O(n)$ since at most n constant size activation records
    - $\Theta(n+m) + \Theta(n) + O(m) + O(n) \;=\; \Theta(n+m)$

```
                          back[1 2 3 4 5 6 7 8 9]
bidfs(4)                      * * * 1 * * * * *
4} tree[47]
4} bidfs(7)                   * * * 1 * * 2 * *
4} 7} tree[71]
4} 7} bidfs(1)                3 * * 1 * * 2 * *
4} 7} 1} tree[12]
4} 7} 1} bidfs(2)             3 4 * 1 * * 2 * *
4} 7} 1} 2} tree[28]
4} 7} 1} 2} bidfs(8)          3 4 * 1 * * 2 5 *
4} 7} 1} 2} 8} back[81]       3 4 * 1 * * 2 3 *
4} 7} 1} 2} 8} back[84]       3 4 * 1 * * 2 1 *
4} 7} 1} 2} backup noout      3 1 * 1 * * 2 1 *
4} 7} 1} backup noout         1 1 * 1 * * 2 1 *
4} 7} 1} tree[15]
4} 7} 1} bidfs(5)             1 1 * 1 6 * 2 1 *
4} 7} 1} backup               out[15]
4} 7} backup noout            1 1 * 1 6 * 1 1 *
4} 7} tree[73]
4} 7} bidfs(3)                1 1 7 1 6 * 1 1 *
4} 7} 3} tree[36]
4} 7} 3} bidfs(6)             1 1 7 1 6 8 1 1 *
4} 7} 3} 6} tree[69]
4} 7} 3} 6} bidfs(9)          1 1 7 1 6 8 1 1 9
4} 7} 3} 6} 9} back[93]       1 1 7 1 6 8 1 1 7
4} 7} 3} 6} backup noout      1 1 7 1 6 7 1 1 7
4} 7} 3} backup               out[93][69][36]
4} 7} backup                  out[73]
4} backup                     out[84][81][28][12][71][47]
```

lexicographic bfs

- in ordinary bfs, the order in which neighbours of a vertex are placed on the queue is arbitrary and so gives no information on the structure of the graph
- in lexbfs, the queue of vertices is replaced with a queue of vertex subsets; at all times, the subsets partition the collective neighbourhood of vertices already visited;
- the label of each queue element is its neighbourhood of already visited vertices; vertices are labelled in decreasing order as they are visited; queue elements are maintained in lexicographically decreasing order, namely the 1st item in the queue is lexicographically largest
- each iteration, an arbitrary vertex is removed from the first set of the queue, and the queue partition is refined
- *lexicographic order* for finite integer subsets S_j and S_k is defined as follows:

  sets are lex'ly equal if and only if they are equal

  set S_j is lexĺy less than S_k iff the largest integer which is in exactly one of S_j, S_k is in S_k.
- e.g.: $\{1\} < \{1,2,4\} < \{3,4\}$

```
lexbfs (store lexicographic labels of each vertex)
for each vertex v do
  lexLabel[v] <- { }
  lexorder[v] <- 0  endfor
for p <- n downto 1 do  (* p is priority *)
  v <- any unvisited vertex with lex'ly largest label
  lexorder[v] <- n+1-p (*put v next in lexbfs order*)
  for each unvisited nbr w of v do
    add p to lexLabel[w] endfor endfor

lexbfs (refine a queue of sets in lexicographic order)
for each vertex v do
  lexorder[v] <- 0 endfor
Q <- one set containing all vertices, status old
for j <- 1 to n do  (*don't need p*)
  v <- remove any vertex from first set in Q
  lexorder[v] <- j
  for each unvisited nbr x of v do
    X <- set in Q containing x
    W <- set in Q preceding X
    remove x from X
    if W is not new then
      insert in Q a new set {x} preceding X
    else
      add x to W endif endfor
  for each new set do
    change its status to old endfor endfor
```

lexbfs and chordal graphs

- *chordal* graph (a.k.a. triangulated): no induced $C_{k \geq 4}$
- *simplicial vertex*: neighbourhood induces a clique
- *simplicial (elimination) ordering*:
  $(v_1, \ldots, v_n)$ with v_j simplicial in $G[\{v_1, \ldots, v_j\}]$
- chordal $\iff$ graph has simplicial ordering
- chordal iff lexbfs order is simplicial order
- can check whether order simp'l in linear time, so linear chordal graph recognition
- simplicial orders also lead to optimal chordal graph colouring/clique size/generation

other chordal graph results

- graph *perfect*: for each vertex induced subgraph, chromatic number equals size of largest clique
- chordal graphs are perfect
- complements of chordal graphs (co-chordal) graphs are perfect
- chordal graph $\iff$

  intersection graph of subtrees of a tree
- chordal graphs have $O(n)$ maximal cliques

$\Theta(n + m)$ time/space simp'l order recognition

```
A[u]: list with rep'ns; vertices which must see u

for all v do
  A[v] <- { } endfor
for j <- n downto 2 do
  v <- j'th vertex in the lexbfs order
  U <- nbrs of v which precede v in lexbfs order
  if U non-empty then
    u <- vertex in U with highest position in lexbfs order
    add U - {u} to A[u] endif
  if A[v] - nbr[v] <> { } then return NO endif endfor
return YES
```

- correctness: u is simplicial in $G[v_1, \ldots, u]$
- complexity:
  - line -2 can be done in $O(|A[v]| + |nbr[v]|)$ time/space using an array of size n initialized to all zero, and reset to all zero after each test
  - $\sum |A[u]| < \sum |nbr[v]|$
  - total time/space $\Theta(|V| + \sum |nbr[v]| + \sum |A[v]|) = \Theta(n + m)$

chordal graph algorithms

- linear recognition
  - lexBFS (queue of sets)
  - verify simp'l order
- max'l cliques
  - each has form $\{v\} \cup$ Prev(v)
  - at most n
  - can output in linear time
- linear colour/clique
  - simp'l order
  - greedy: in forward order, each col(v) $< -$ smallest colour not in col(Prev(v))
- linear clique cover/stable set
  - simp'l order
  - greedy: in reverse order, each clique $< - \{v\} \cup$ Prev(v)

Prev(v): all previous, in simp'l order, neighbours of v

Topic: MST algorithms

- Boruvka, Prim/Dijkstra, Kruskal algorithms
- P/D implementations
  - naive $O(n^3)$
  - store best edge for each fringe vertex $O(n^2)$
  - as previous, use priority queue (heap imp'n) $O(m \lg n)$
  - as previous, (Fibonacci heap imp'n) $O(m + n \lg n)$ [did not cover]
- Kruskal implementations
  - sort all edges first (not so good: why?)
  - priority queue (heap imp'n) $O(m \lg m)$
  - sideline: union/find trees
  - n w-unions, m finds take $O(m \lg n)$
  - n w-unions, m c-finds take $O(m \lg^* n)$
- REFERENCES: my 204 notes (on the web)

randomized graph algorithms

- r.a.: has some randomized step: among a collection of items, pick one randomly
- classical cs r.a.: qsort (WC $\Theta(n^2)$ AC $\Theta(n \lg n)$)
- mot'n: if perform repeatedly, guarantees AC performance
- classical graph r.a.: min cut

def'n cut (of connected graph): edge set, removal disconnects graph

- deterministic alg'ms use network flow

```
input: connected graph with n vertices, edges labelled
output: a minimal cut (not nec. minimum)
repeat n-2 times:
  e <- *random* edge of remaining loopless multigraph
  contract e (remove loops; leave parallel edges)
cut <- edges between two remaining vertices
```

- references: Randomized Algorithms, by Motwani/Raghavan

graph drawing

- active research area
- graph *drawing*: embedding of a graph on some surface, usually the plane, s.t.
  - vertex $\leftrightarrow$ point
  - edge $\leftrightarrow$ continuous line joining two vertex points
  - lines of adjacent edges intersect only at the point of incidence
  - lines of non-adjacent edges intersect at most once (and if so, at a crossing point, and not at a 'touching' or tangent point)
- crossing number $\nu(G)$: over all drawings, minimum number of line crossings
- planar graph: $\nu(G) = 0$
- $\nu(G) \leq k$? NP-complete (for fixed k?)
- G planar? polynomial (linear time)

graph drawing: math background

- closed continuous line partitions plane (inside/outside)
- $K_{3,3}$ and K_5 non-planar
- *homeomorphic* graphs: can be obtained from same graph by inserting vertices of degree two into edges
- Kuratowski's thm: planar iff no subgraph homeomorphic to K_5 or $K_{3,3}$
- related thm: planar iff no subgraph contractible to K_5 or $K_{3,3}$
- *face*: (roughly) region of planar embedding
- Euler's thm: for planar embedding of connected graph, n+f=m+2
- Corollary: for planar graph, $m \leq 3n - 6$
- *thickness* $t(G)$: min number of planar graphs which can be superimposed to form G
- $t(G) \geq \frac{m}{3n-6}$
- planar embedding of connected planar graph G has *geometric dual* graph G^*:
  - vertex of G^* $\leftrightarrow$ face of G
  - edge of G^* $\leftrightarrow$ join two faces of G which share an edge of G
  - this concept goes back to Euclid, and is important in computational geometry

a planarity testing algorithm

- simple $O(n^3)$ divide and conquer algorithm
  [Graph Drawing, di Battista et. al.] p74
- Hopcroft and Tarjan implementation: $O(n)$

```
algorithm is_planar(G,C)
Input:  biconnected G with m <= 3n-6 and separating cycle C
Output: yes/no
1. find pieces of G with respect to C
2. for each piece P of G that is not a path
   (a) P' <- graph obtained by adding P to C
   (b) C' <- cycle of P' obtained from C by replacing
          the portion of C btwn 2 consec. attachments
          with a path of P btwn them
   (c) if not is_planar(P',C') then return NO
3. compute interlacement graph I of pieces
4. return is_bipartite(I)
```