Lecture 23: Wednesday October 31, 2001

today

- disjoint set union-find problem
  - clarification: union by rank
  - improvement: compressed find
- graphs
  - basic definitions
  - traversal: breadth first search, depth first search

announcements

- assignment 6: due Monday November 5
- midterm II: Wednesday November 7
- pre-midterm office hours
  - today 3:30-4:30
  - Friday 3:30-4:30
  - Monday 3:30-4:30
  - none on Tuesday
- Hallowe'en

clarification

- following should be called **union by rank**, not weighted union
- `rUnion` with ordinary `find` rank[x]=height(x)
- `rUnion` with compressed `cFind` rank[x] $\leq$ height(x)
- (weighted union: based on $|T(x)|$, $|T(y)|$ instead of rank[x],rank[y])

```
proc rUnion(x,y) { make smaller rank root child of other root
  rx <- find(x)  { root of x's tree
  ry <- find(y)  { root of y's tree
  if rank[rx]>rank[ry]  {compare ranks; initially 0
    then P[ry] <- rx
    else { rank[rx] less/equal rank[ry]
      P[rx] <- ry
      if rank[rx]=rank[ry]
        then INC[rank[ry]]
```

- recall: union/find WC $\Theta(n+(m-n)n)$
- recall: rUnion/find WC $\Theta(n+(m-n)\lg n)$

even better(!!): union by rank, compressed find

- cFind: on node-root path, change parent of each node to root

```
non-recursive                          * recursive
                                       *
                                       *
proc cFind(x)                          *
  t <- x                               * proc cFind(x)
  while P[t]<>t do { find root         *   if x<>P[x] then
    t <- P[t]                          *     P[x] <- cFind(P[x])
  root <- t                            *   return P[x]
  t <- x                               *
  while P[t]<>t do { compress path *
    x <- t                             *
    t <- P[t]                          *
    P[x] <- root                       *
  return root                          *
```

analysis: rUnion and cFind

- recall: `rUnion` used with `cFind` height $\leq$ rank

- $\lg^* n$: smallest t so that $2^{2^{2^{\cdots}}} \geq n$, where t is number of 2's
- $\lg^* n \leq 5$ for $n \leq 2^{65536}$ virtually constant

```
  n     2  ...4  ...16  ...65536  ...2^{65536}
lg*n    1    2     3      4             5
```

DSUF conclusion

- U/F WC $\Theta(n + (m-n)n)$
- rU/F WC $\Theta(n + (m-n)\lg n)$
- rU/cF WC $O(n + (m-n)\alpha(n))$

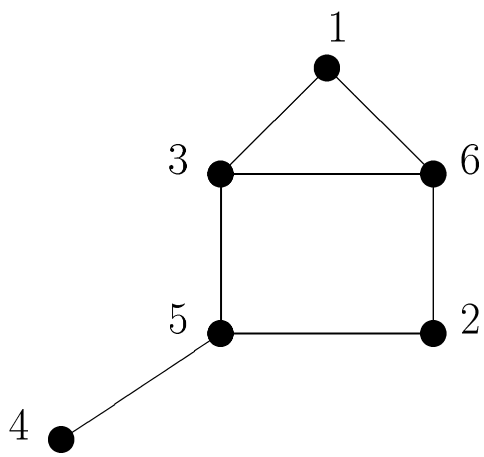
$f(n) = 1 \in o(\alpha(n))$ and $f(n) = \alpha(n) \in o(\lg^* n)$

graph algorithms

- definitions [CLRS Appendix B.4]
- elementary algorithms [CLRS Ch 21]
  - breadth first search
  - depth first search
  - biconnected components
- minimum spanning tree [CLRS Ch 23]

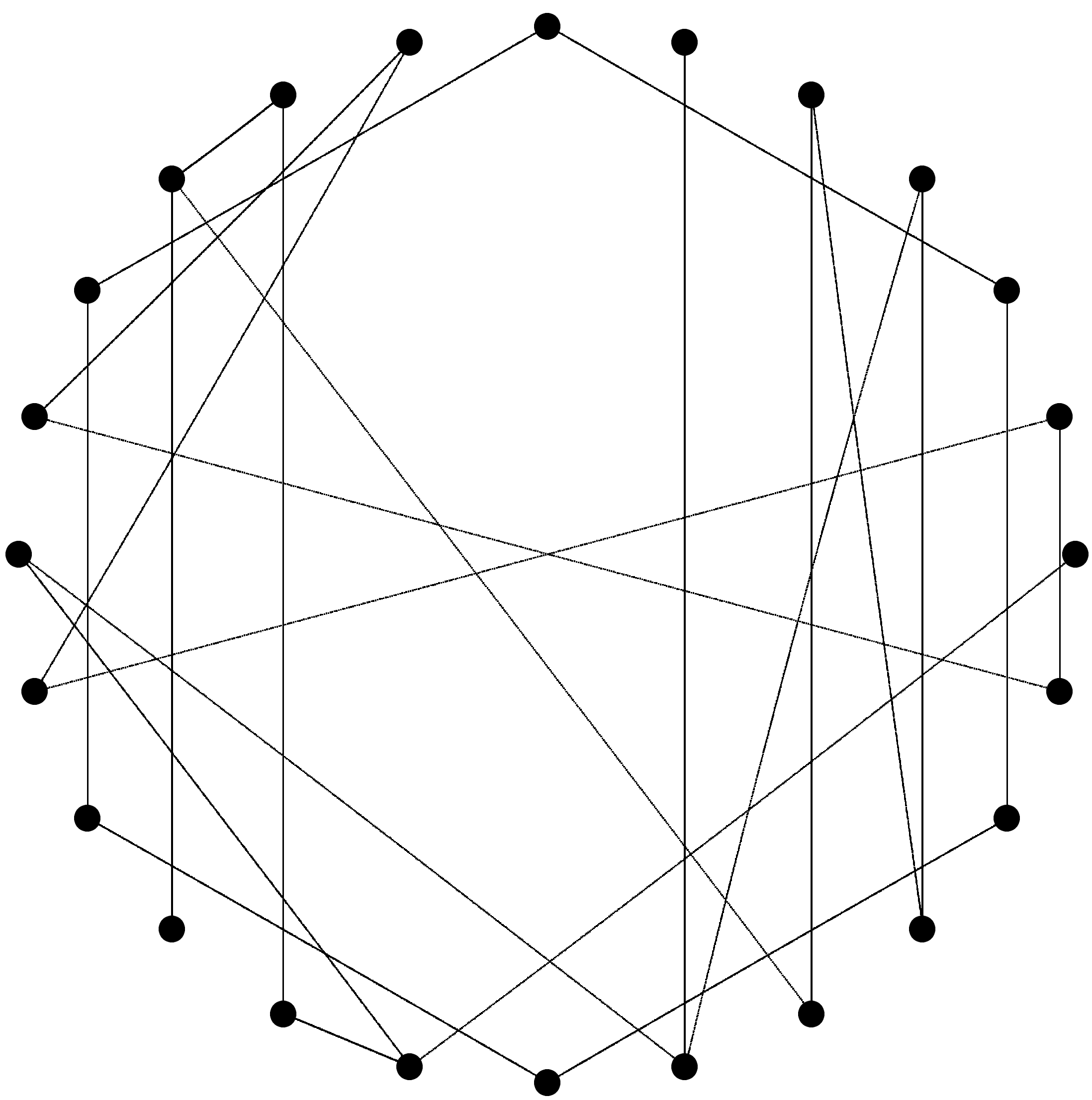
graph definitions

- graph $G = (V, E)$:
  vertex set V and edge set E consisting of unordered vertex pairs
- adjacent, incident, degree (sequence), $\sum_{v \in V} d(v) = 2|E|$
- graph variants
  - simple graph: no loops, multiple edges
  - digraph $D = (V, A)$: arcs are ordered vertex pairs
  - multigraph $M^* = (V, E^*)$: E^* is an edge multi-set
  - multi-digraph $D^* = (V, A^*)$: A^* is an arc multi-set
  - weighted (di/multi)graph: edges/arcs have weights/distances
- computer representation
  - adjacency list
  - adjacency matrix
- equal versus isomorphic
- maximum/minimum versus maximal/minimal
- subgraph
- clique, independent set
- (simple) path, connected, (rooted) tree




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

	1	2	3	4	5	6
1			*			*
2					*	*
3	*				*	*
4					*	
5		*	*	*		
6	*	*	*			



- *adjacent* vertices induce an edge; *adjacent* edges share a vertex
- *incident* edge and vertex: if the vertex is in the edge
- vertex *neighbourhood*: all vertices adjacent to the vertex
- vertex *degree*: size of its neighbourhood
- *equal* graphs: equal vertex sets and equal edge sets
- two graphs are *isomorphic* if the vertices of one graph can be relabelled so that the resulting graphs are equal
- $G' = (V', E')$ is a *subgraph* of graph $G = (V, E)$ if V' is a subset of V and E' is a subset of E
- *maximum/minimum*: largest/smallest
- *clique*: vertex subset, each pair adjacent
- w.r.t. a graph and a property P, a vertex subset is *maximal/minimal* if it satisfies P and no proper superset/subset satisfies P e.g. maximal clique
- in a graph, a vertex sequence $(v_0, v_1, \ldots, v_t)$ is a
  - *path* if $\{v_j, v_{j+1}\}$ is an edge, for $0 \leq j < t$,
  - *simple path* if a path, and no vertex is repeated
  - *cycle* if a path, and also $\{v_0, v_t\}$ is an edge, no cycle edge is repeated
  - *simple cycle* if a cycle, and no vertex is repeated

- *connected* graph: path between every two vertices
- graph *component*: maximal connected subgraph
- binary *equivalence relation*: reflexive, symmetric, transitive
- *equivalence class*: maximal sets of equivalent pairs

  e.g. for any graph, the relation $R(x, y)$: *there is a path* $(x, \ldots, y)$ is an equivalence relation whose equivalence classes are the graph's components
- *forest*: acyclic graph
- *tree*: connected forest
- *rooted tree*: directed tree with one vertex as the root
  for each arc (x, y), x is the *parent* of y, and y is the *child* of x
- for a weighted graph, *weight* of a subgraph: sum of the weights of the edges of the subgraph

most elementary graph algorithm: traversal

- problem: visit all vertices, by following all edges (in some order)
- e.g. maze traversal
- general graph traversal with list to store "waiting" vertices
  - FIFO list (queue): version of breadth first search
  - LIFO list (stack): version of depth first search
  - recursive version: depth first search

traversal applications

- connected components — any traversal
- two-colouring — any traversal
- odd cycle — any traversal
- unweighted distance — BFS
- biconnected components — DFS
- strongly connected components — DFS

breadth first search

```
BFS(G,s)                { G=(V,E)  s: start vertex
  for each v in V do
    color[v] <- WHITE  { unknown
    d[v]     <- infty  { distance from s
    p[v]     <- NIL    { predecessor

  makeEmpty(Q)
  enqueue(Q,s)
  color[s] <- GRAY     { in Q
  d[s]     <- 0
  while not empty(Q) do
    u <- dequeue(Q)
    for each v in Adj[u] do   { nbrs of u
      if color[v]= WHITE then
        color[v] <- GRAY
        d[v] <- d[u]+1
        p[v] <- u
        enqueue(Q,v)
    color[u] <- BLACK         { visited
```