# Lecture 21: Friday October 26, 2001

## today

- graphs
  - component definition: equivalence relation
- disjoint set union-find problem
  - DS ADT
  - DS application: graph components
  - DS implementation
    - simple: representative array
    - better: forest of rooted trees (via parent array)

## announcements

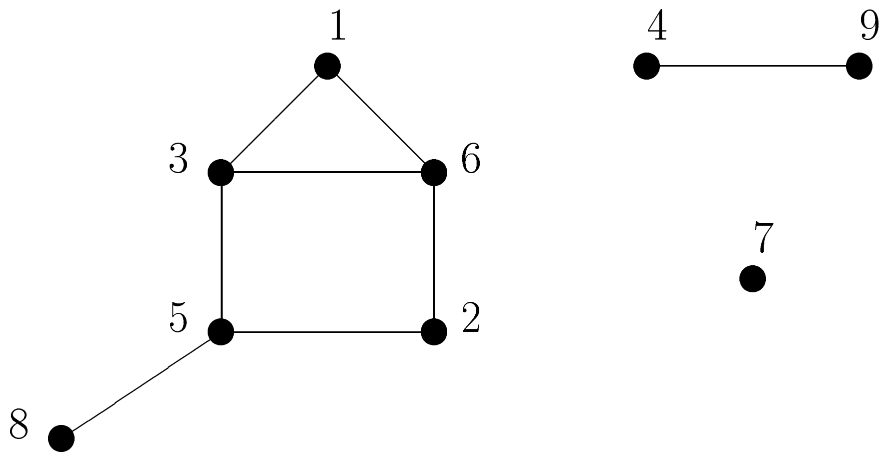
- assignment 5: due Monday October 29
- acknowledge references
- no office hours next week ☺ ☹

## some graph definitions

- *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
- in a graph, a vertex sequence $(v_0, v_1, \ldots, v_t)$ is a *path*

  if each consecutive pair of vertices is adjacent
- *connected* graph: for each pair of vertices, a path between them
- *subgraph* of $G = (V, E)$: a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$
- 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




| | |
|---|---|
| 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 |

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

## disjoint sets union-find problem [CLRS Ch. 21]

- the DSUF ADT
- analysis: sequence of operations
- implementation: array of representatives
- application: graph components
- implementation: forest of rooted trees
  - basic imp'n
  - improvement: union by rank
  - improvement: compressed find

## disjoint sets: abstract data type

- maintain: pairwise disjoint sets
- one element of each set is *representative*
- operations
  - `MakeSet(x)` $S_x \leftarrow \{x\}$
  - `Find(x)` return representative of set containing $x$
  - `Union(x,y)` $S_z \leftarrow S_{f(x)} \cup S_{f(y)}$

## disjoint sets analysis: over a sequence of operations

- first: `MakeSet(1), ..., MakeSet(n)`
- then: all `Union`, `Find` op'ns
- $n$: number of elements $m$: total number operations
- time for all $m = n + |U| + |F|$ operations?

## simplest DS implementation: array R of representatives

```
proc makeSet(n)   { initialize all rep's
  for x <- 1 to n do
    R[x] <- x

proc find(x)
  return R[x]     { representative of x

proc union(x,y)
  rx <- R[x]
  ry <- R[y]
  for j <- 1 to n do
    if R[j]=ry then R[j] <- rx
```

```
***** example ***********************************
sets at start     {1} {2} {3} {4} {5} {6} {7} {8}

indices             1   2   3   4   5   6   7   8
-------------------------------------------------
R                 [ 1   2   3   4   5   6   7   8 ]

U(1,4)            [ 1   2   3   1   5   6   7   8 ]
F(1)
F(4)
U(2,3)            [ 1   2   2   1   5   6   7   8 ]
U(5,1)            [ 5   2   2   5   5   6   7   8 ]
U(1,8)            [ 5   2   2   5   5   6   7   5 ]
U(6,5)            [ 6   2   2   6   6   6   7   6 ]
F(6)
F(3)

sets at finish:      {1,4,5,6,8}   {2,3}   {7}
```

## a DS/UF application: finding components of a graph

```
Connected-Components(G)
  for each vertex v in V(G) do
    MakeSet(v)
  for each edge xy in E(G) do
    if not SameComponents(x,y) then Union(x,y)

Same-Components(x,y,G)
  return FindSet(x)=FindSet(y)   { boolean value returned
```

## DS simplest implementation: analysis

- recall: $n$ MakeSets, then $m - n$ Union/Find
- each MakeSet: $\Theta(1)$ time
- each Find(x): $\Theta(1)$ time
- each Union(x,y): $\Theta(n)$ time
- worst case time $\Theta(n + (m - n)n))$ $\Theta(mn)$ if $m >> n$
- *amortized*: average per operation, over a sequence
- for $m >> n$, amortized time per operation $\Theta(n)$

## better(?) DS implementation: forest of rooted trees

- elements of set $\leftrightarrow$ nodes of rooted tree
- representative of set $\leftrightarrow$ root of tree
- each node needs only parent $\Rightarrow$ implement forest via array

```
proc makeSet(n)   { initialize all parents
  for x <- 1 to n do
    P[x] <- x     { parent of root is root

proc find(x) { return element at root
  while P[x]<>x do
    x <- P[x]   { parent of x
  return x

proc union(x,y) { make root of x's tree child of root of y's
  rx <- find(x)
  ry <- find(y)
  P[rx] <- ry
```

```
 same example ***************************************
sets at start  {1} {2} {3} {4} {5} {6} {7} {8}

indices         1   2   3   4   5   6   7   8
---------------------------------------------------
P              [1   2   3   4   5   6   7   8]

U(1,4)         [4   2   3   4   5   6   7   8]
F(1)
F(4)
U(2,3)         [4   3   3   4   5   6   7   8]
U(5,1)         [4   3   3   4   4   6   7   8]
U(1,8)         [4   3   3   8   4   6   7   8]
U(6,5)         [4   3   3   8   4   8   7   8]
F(6)
F(3)

sets at finish:    {1,4,5,6,8}   {2,3}   {7}

forest at finish:       8          3      7
                       / \         |
                      4   6        2
                     / \
                    1   5
```