

## Disjoint Sets (Partitions)

- need data structure that can do fast:
  - merge
  - find
- **Union-Find** data structure
- trees in arrays
  - $a[5] =$  index of parent of 5
  - if smallest element of set: pointer to itself
- each set is rooted tree
  - merge:  $O(1)$
  - find:  $O(n)$
- if we always merge smaller trees into bigger ones:
  - find becomes  $O(\log n)$

© 2005-06. 324-1-59

## Disjoint Sets: Example

- Disjoint sets:  $\{1,3,4\}$  and  $\{2,5,6\}$
- Tree representation:



- Array representation:

index	1	2	3	4	5	6
value	1	2	1	1	2	5

© 2005-06. 324-1-60

## Greedy algorithms

- take whatever you can get!
- Making change

```
MultiSet change( n ) {
    const Set coins = { 100, 25, 10, 5, 1 };
    MultiSet result = MultiSet. getEmptySet();
    int sum = 0;
    while( ! sum == n ) {
        x = largest item in coins so that sum+x<=n
        if( there is no x ) {
            return error( "no solution exists" );
        }
        result = union( result, {x} );
        sum += x;
    }
    return result;
}
```

© 2005-06. 324-1-61

## Structure of a Greedy Algorithm

```
Set greedy( Set candidates ) {
    Set result = Set. getEmptySet();
    while( ! candidates. isEmpty() && ! isSolution( result ) ) {
        Object x = select( candidates );
        candidates. remove( x );
        if( feasible( union( result , x ) ) ) {
            result. add( x );
        }
    }
    if( isSolution( result ) ) {
        return result ;
    } else {
        error( "there are no solutions" );
    }
}
```

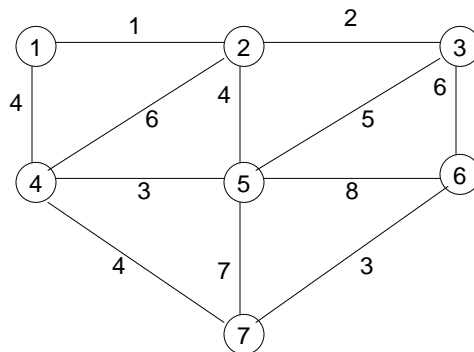
© 2005-06. 324-1-62

## Minimum Spanning Trees

- Problem:
  - we have a graph
    - edges have a cost
  - we need a spanning tree that's cheap
  - number of edges in a spanning tree?
  - sample application: creating a network to connect cities
- set of candidates?
- isSolution function?
- isFeasible function?

© 2005-06. 324-1-63

## Kruskal's Algorithm: Example



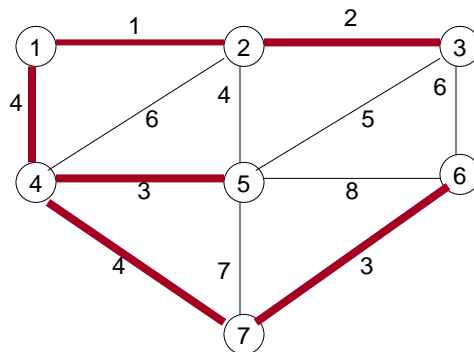
© 2005-06. 324-1-64

## Kruskal's Algorithm

1. make a partition for each node
2. repeat until one partition left:
  1. pick shortest remaining edge
  2. if it is inside one partition: rejectelse:
  1. merge the two partitions
  2. add edge to solution set
3. done.

© 2005-06. 324-1-65

## Kruskal's Algorithm: Example



© 2005-06. 324-1-66

## Kruskal's Algorithm

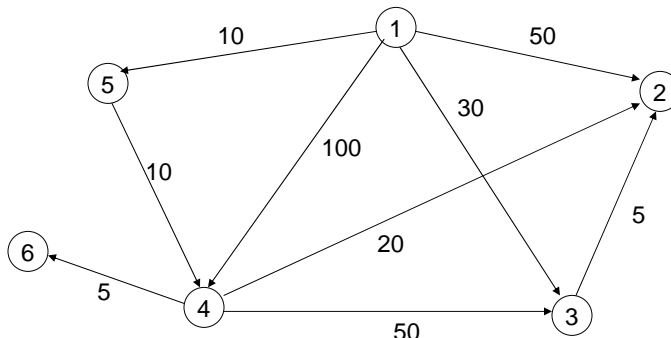
```
Edge[] minimumSpanningTree( Graph g ) {
    int e = g. getNumberOfEdges();
    int n = g. getNumberOfNodes();
    Edge[] mst = new Edge[ n-1 ];
    UnionFind uf = new UnionFind( n );
    Edge[] candidates = g. getEdges();
    sort( candidates );
    for( int i = 0, k = 0; i < e && k < n-1; i++ ) {
        Set set1 = uf. find( candidates[ i ]. firstNode() );
        Set set2 = uf. find( candidates[ i ]. secondNode() );
        if( set1 != set2 ) {
            uf. merge( set1, set2 );
            mst[ k++ ] = candidates[ i ];
        }
    }
    return mst;
}
```

- Runtime:  $O( e \log n )$ 
  - e: edges, n: nodes

© 2005-06. 324-1-67

## Shortest Paths: Dijkstra's Algorithm

- How cheap (fast) can you get from one node to all others?
  - directed graph with cost
  - sample application: distances from one city to other cities



© 2005-06. 324-1-68

## Dijkstra's Algorithm

- candidates = set of nodes
- result = array of (preliminary) distances to start node for all other nodes
- in each step:
  - pick the candidate that has shortest distance to start node so far
  - check if it makes the distance to any other node shorter
  - if so, update distances

© 2005-06. 324-1-69

## Dijkstra's Algorithm: Example

step	node considered	remain. candid.	result[2]	result[3]	result[4]	result[5]	result[6]
1							
2							
3							
4							
5							

© 2005-06. 324-1-70

## Dijkstra's Algorithm

```
int[] shortestPath( graph ) { // node 1: start node
    Set candidates = graph. getSetofNodes();
    candidates. remove( 1 );
    int[] result = new int[];
    for( i = 2; i <= n; i++ ) {
        result[ i ] = graph. getCost( 1, i ); // may be ∞
    }
    while( candidates. size() > 0 ) {
        int node = element of candidates so that result[ node ] minimal
        candidates. remove( node );
        for( int c: candidates ){
            result[ c ] = min( result[ c ], result[ node ] + graph. getCost(
                node, c )
        )
    }
    return result;
}
}

```

- Runtime:  $O(n^2)$ 
  - $O(n \log n)$  if inverted heap is used for candidate nodes (which is better?)