CSE 373 Analysis of Algorithms
Fall 2016
Instructor: Prof. Sael Lee

LEC12: MINIMUM SPANNING TREES (PP.191-205)
(CH6 WEIGHTED GRAPH ALGORITHMS)

Lecture slide courtesy of Prof. Steven Skiena

WEIGHTED GRAPH ALGORITHMS

- Beyond DFS/BFS exists an alternate universe of algorithms for *edge-weighted graphs*.
- Our adjacency list structure consists of an array of linked lists, such that the outgoing edges from vertex *x* appear in the list edges[x]:

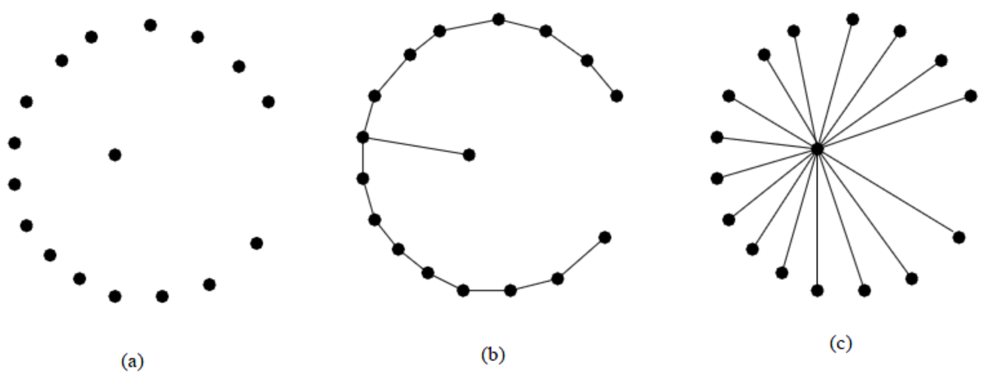
```
typedef struct {
        edgenode *edges[MAXV+1];    /* adjacency info */
        int degree[MAXV+1];         /* outdegree of each vertex */
        int nvertices;              /* number of vertices in graph */
        int nedges;                 /* number of edges in graph */
        int directed;               /* is the graph directed? */
} graph;
```

- Each edgenode is a record containing three fields, the first describing the second endpoint of the edge (y), the second enabling us to annotate the edge with a weight (weight), and the third pointing to the next edge in the list (next):

```
typedef struct {
        int y;                      /* adjacency info */
        int weight;                 /* edge weight, if any */
        struct edgenode *next;      /* next edge in list */
} edgenode;
```

MINIMUM SPANNING TREES

- A **tree** is a connected graph with no cycles.
- A **spanning tree** is a subgraph of G which has the same set of vertices of G and is a tree.
- A **minimum spanning tree** of a weighted graph G is the spanning tree of G whose edges sum to minimum weight.
- There can be more than one minimum spanning tree in a graph
  - Ex> consider a graph with identical weight edges.



(a) Two spanning trees of point set;
(b) the minimum spanning tree, and
(c) The shortest path from center tree

WHY MINIMUM SPANNING TREES?

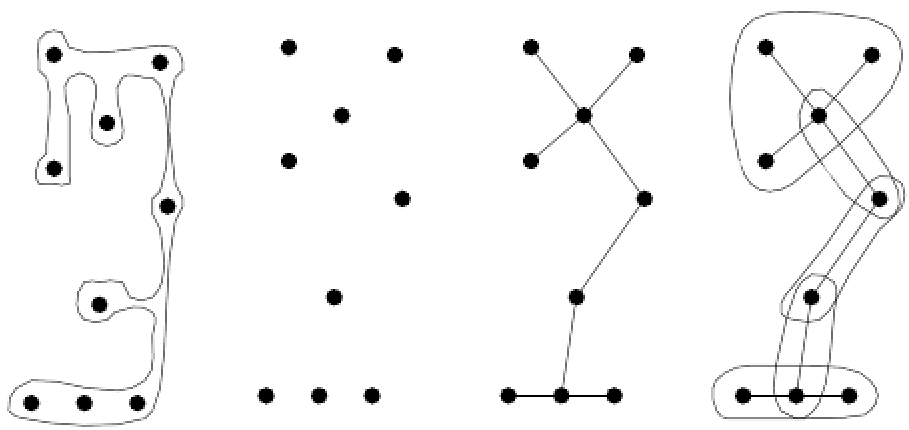
- The minimum spanning tree problem has a long history
  - the first algorithm dates back at least to 1926!.
- Minimum spanning tree is always taught in algorithm courses since
  - (1) it arises in many applications,
  - (2) it is an important example where *greedy* algorithms always give the optimal answer, and
  - (3) Clever data structures are necessary to make it work.
- In **greedy algorithms**, we make the decision of what next to do by selecting the <u>best local option</u> from all available choices<u>, without regard to the global structure</u>.

APPLICATIONS OF MINIMUM SPANNING TREES

- Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire.
- Minimum spanning trees provide a reasonable way for *clustering* points in space into natural groups.
- What are natural clusters in the friendship graph?

MINIMUM SPANNING TREES AND NET PARTITIONING

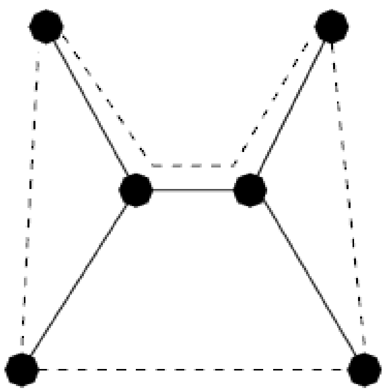
- One of the war stories in the text describes how to partition a graph into compact subgraphs by deleting large edges from the minimum spanning tree.



(a) (b) (c) (d)

MINIMUM SPANNING TREES AND TSP

- When the cities are points in the Euclidean plane, the minimum spanning tree provides a good heuristic **for traveling salesman problems.**
- The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.



The Option Traveling System tour is at most twice the length of the minimum spanning tree.

Note: There can be more than one minimum spanning tree considered as a group with identical weight edges.

PRIM'S ALGORITHM

- If G is connected, every vertex will appear in the minimum spanning tree.
- If not, we can talk about a minimum spanning forest.
- Prim's algorithm starts from one vertex and grows the rest of the tree an edge at a time.
- As a greedy algorithm, which edge should we pick?
  - The cheapest edge with which can grow the tree by one vertex without creating a cycle.

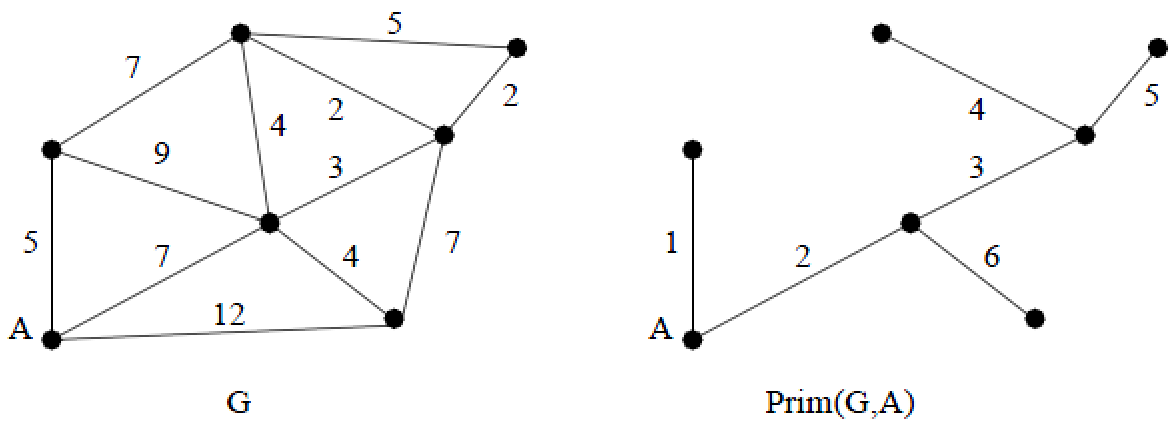
PRIM'S ALGORITHM (PSEUDOCODE)

- During execution each vertex v is either in the tree, *fringe* (meaning there exists an edge from a tree vertex to v) or *unseen* (meaning v is more than one edge away).

```
Prim-MST(G)
        Select an arbitrary vertex s to start the tree from.
        While (there are still non-tree vertices)
                Select the edge of min weight between a tree
                Add the selected edge and vertex to the tree T_prim.
```

- This creates a spanning tree, since no cycle can be introduced, but is it minimum?

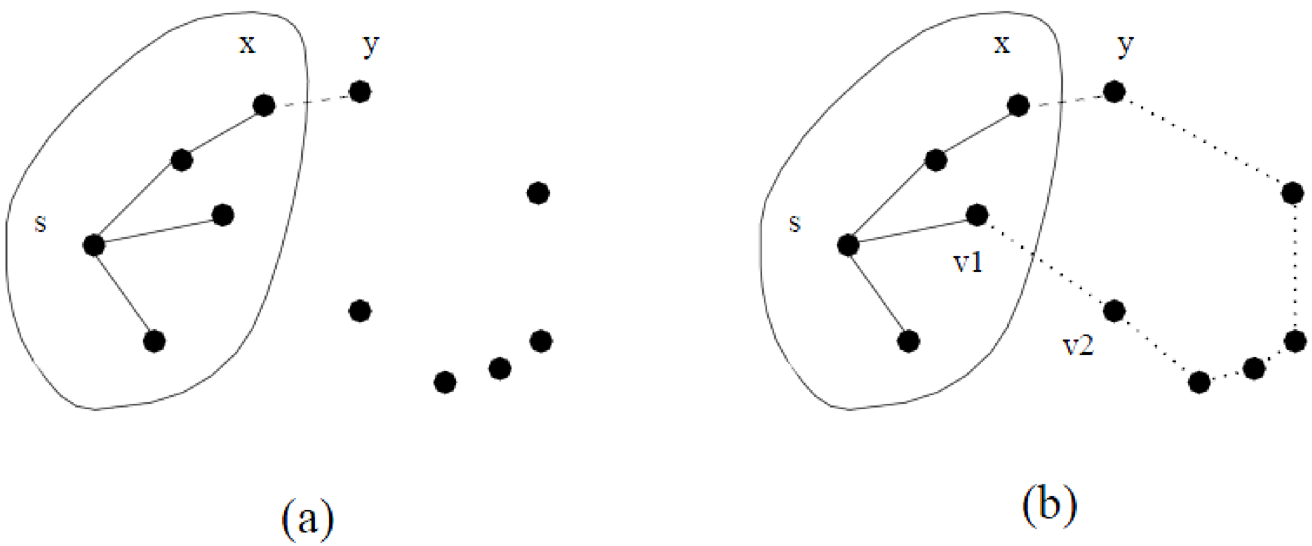
PRIM'S ALGORITHM IN ACTION



WHY IS PRIM CORRECT?

- We use a proof by contradiction:
- Suppose Prim's algorithm does not always give the minimum cost spanning tree on some graph.
- If so, there is a graph on which it fails.
- And if so, there must be a first edge *(x, y)* Prim adds such that the partial tree V' cannot be extended into a minimum spanning tree.
- But if (x; y) is not in MST(G), then there must be a path in MST(G) from x to y since the tree is connected.
- Let (v, w) be the first edge on this path with one edge in V' Replacing it with (x, y) we get a spanning tree with smaller weight, since W(v, w) > W(x, y).
- Thus you did not have the MST!!

- Thus we cannot go wrong with the greedy strategy the way we could with the traveling salesman problem.



Where Prim's algorithm goes bad? No, because $d(v_1, v_2) \geq d(x, y)$

HOW FAST IS PRIM'S ALGORITHM?

- That depends on what data structures are used.
- In the simplest implementation, we can simply mark each vertex as tree and non-tree and search always from scratch:

  Select an arbitrary vertex to start.

  While (there are non-tree vertices)

  select minimum weight edge between tree and fringe

  add the selected edge and vertex to the tree

- This can be done in *O(nm)* time, by doing a DFS or BFS to loop through all edges, with a constant time test per edge, and a total of n iterations.

PRIM'S IMPLEMENTATION

- To do it faster, we must identify fringe vertices and the minimum cost edge associated with it fast.

```
prim(graph *g, int start)
{
    int i;                    /* counter */
    edgenode *p;              /* temporary pointer */
    bool intree[MAXV+1];      /* is the vertex in the tree yet? */
    int distance[MAXV+1];     /* cost of adding to tree */
    int v;                    /* current vertex to process */
    int w;                    /* candidate next vertex */
    int weight;               /* edge weight */
    int dist;                 /* best current distance from start */

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }
```

```
        distance[start] = 0;
        v = start;
        while (intree[v] == FALSE) {
                intree[v] = TRUE;
                p = g->edges[v];
                while (p != NULL) {
                        w = p->y;
                        weight = p->weight;
                        if ((distance[w] > weight) && (intree[w] == FALSE)) {
                                distance[w] = weight;
                                parent[w] = v;
                        }
                        p = p->next;
                }
                v = 1;
                dist = MAXINT;
                for (i=1; i<=g->nvertices; i++)
                        if ((intree[i] == FALSE) && (dist > distance[i])) {
                                dist = distance[i];
                                v = i;
                        }
        }
}
```

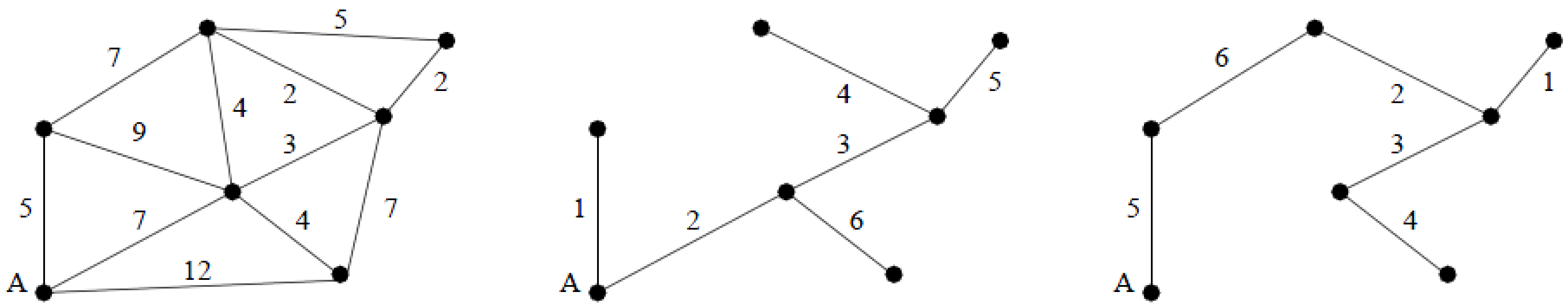
PRIM'S ANALYSIS

- Finding the minimum weight fringe-edge takes O(n) time – just bump through fringe list.
- After adding a vertex to the tree, running through its adjacency list to update the cost of adding fringe vertices(there may be a cheaper way through the new vertex) can be done in O(n) time.
- Total time is **$O(n^2)$.**

KRUSKAL'S ALGORITHM

- Since an easy lower bound argument shows that every edge must be looked at to find the minimum spanning tree, and the number of edges $m = O(n^2)$, Prim's algorithm is optimal in the worst case.
- The complexity of Prim's algorithm is independent of the number of edges.
- Can we do better with sparse graphs? Yes!
- **Kruskal's algorithm** is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.

KRUSKAL'S ALGORITHM IN ACTION



G Prim(G,A) Kruskal(G)

WHY IS KRUSKAL'S ALGORITHM CORRECT?

- Again, we use proof by contradiction.
- Suppose Kruskal's algorithm does not always give the minimum cost spanning tree on some graph.
- If so, there is a graph on which it fails.
- And if so, there must be a first edge (x,y) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree.
- When we added (x,y) there previously was no path between x and y, or it would have created a cycle
- Thus if we add (x, y) to the optimal tree it must create a cycle.
- At least one edge in this cycle must have been added after (x, y), so it must have a heavier weight.
- Deleting this heavy edge leave a better MST than the optimal tree? A contradiction!

HOW FAST IS KRUSKAL'S ALGORITHM?

- What is the simplest implementation?
  - Sort the m edges in O(mlgm) time.
  - For each edge in order, test whether it creates a cycle the forest we have thus far built – if so discard, else add to forest.
    - With a BFS/DFS, this can be done in O(n) time (since the tree has at most n edges).
- The total time is *O(mn)*, but can we do better?

FAST COMPONENT TESTS GIVE FAST MST

- Kruskal's algorithm builds up connected components.
- Any edge where both vertices are in the same connected component create a cycle.
- Thus if we can maintain which vertices are in which component fast, we do not have test for cycles!
  - *Same component*(v_1,v_2) – Do vertices v_1 and v_2 lie in the same connected component of the current graph?
  - *Merge components*(C_1,C_2) – Merge the given pair of connected components into one component.

FAST KRUSKAL IMPLEMENTATION

Put the edges in a heap
count = *0*
while (*count* < *n-1*) do
    get next edge *(v, w)*
    if (component (v) != component(w))
        add to T
        component (v)=component(w)

- If we can test components in *O(log n)*, we can find the MST in *O(mlogm)*!
- *Question:* Is O(mlogn) better than O(mlogm)?

UNION-FIND PROGRAMS

- We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together.
- These can be implemented by *union* and *find* operations, where
  - *Find(i)* – Return the label of the root of tree containing element i, by walking up the parent pointers until there is no where to go.
  - *Union(i,j)* – Link the root of one of the trees (say containing i) to the root of the tree containing the other (say j) so find(i) now equals find(j).

COMPONENT TESTS

- Same Component Tests

  Is $s_i \equiv s_j$

  $t = \text{Find}(s_i)$
  $u = \text{Find}(s_j)$
  Return (Is $t = u$?)

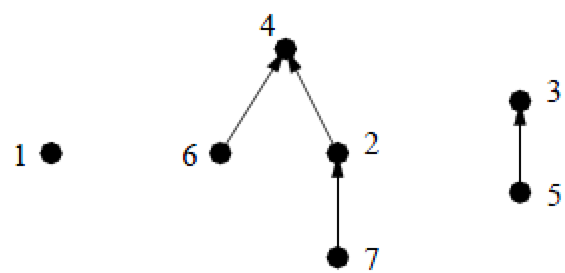
- Merge Components Operation

  Make $s_i \equiv s_j$

  $t = d(s_i)$
  $u = d(s_j)$
  Union(t, u)

UNION-FIND "TREES"

- We are interested in minimizing the time it takes to execute *any* sequence of unions and finds.
- A simple implementation is to represent each set as a tree, with pointers from a node to its parent. Each element is contained in a node, and the *name* of the set is the key at the root:



(l)

1	2	3	4	5	6	7
0	4	0	0	3	4	3

(r)

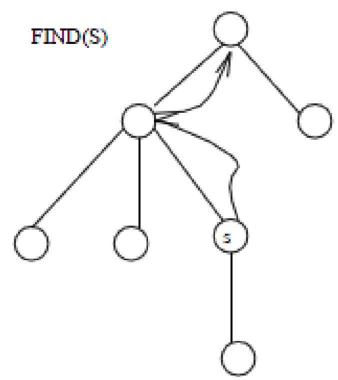
WORST CASE FOR UNION FIND

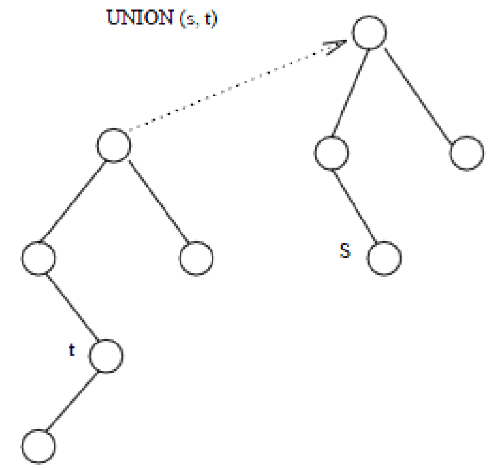
- In the worst case, these structures can be very unbalanced:

```
For i = 1 to n=2 do
        UNION(i,i+1)
For i = 1 to n=2 do
        FIND(1)
```

WHO'S THE DADDY?

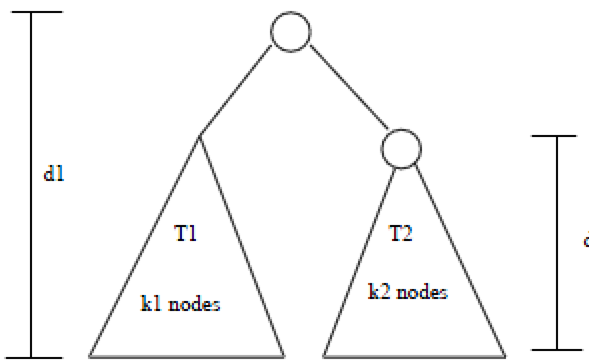
- We want the limit the height of our trees which are effected by *union*'s.
- When we union, we can make the tree with fewer nodes the child.
- Since the number of nodes is related to the height, the height of the final tree will increase only if both subtrees are of equal height!
- If *Union(t, v)* attaches the root of v as a subtree of t **iff** the number of nodes in t is greater than or equal to the number in v, after any sequence of unions, any tree with h/4 nodes has height at most $\lfloor \lg h \rfloor$.





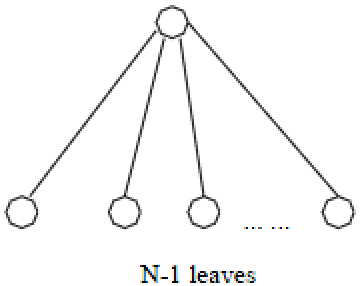
PROOF

- By induction on the number of nodes k, k = 1 has height 0.
- Let d_i be the height of the tree t_i
- If ($d_1 > d_2$) then $d = d_1 \le \lfloor \lg k_1 \rfloor \le \lfloor \lg(k_1 + k_2) \rfloor = \lfloor \lg k \rfloor$
- If ($d_1 \le d_2$), then
- $d = d_2 + 1 \le \lfloor \lg k_2 \rfloor + 1 = \lfloor \lg 2k_2 \rfloor \le \lfloor \lg(k_1 + k_2) \rfloor = \log k$



CAN WE DO BETTER?

- We can do *unions* and *finds* in O(log n), good enough for
- Kruskal's algorithm. But can we do better?
- The ideal *Union-Find* tree has depth 1:



- On a find, if we are going down a path anyway, why not
- change the pointers to point to the root?

- This path compression will let us do better than O(n log n) for n union-finds.
- O(n)? Not quite . . . Difficult analysis shows that it takes O(n(n)) time, where (n) is the inverse Ackerman function and (number of atoms in the universe)= 5.