CSCE 310J
Data Structures & Algorithms

Greedy Algorithms
and
Graph Optimization Problems

Dr. Steve Goddard
goddard@cse.unl.edu

http://www.cse.unl.edu/~goddard/Courses/CSCE310J

1

CSCE 310J
Data Structures & Algorithms

- Giving credit where credit is due:
  - » Most of slides for this lecture are based on slides created by Dr. David Luebke, University of Virginia.
  - » Some examples and slides are based on lecture notes created by Dr. Chuck Cusack, UNL, Dr. Jim Cohoon, University of Virginia, and Dr. Ben Choi, Louisiana Technical University.
  - » I have modified them and added new slides

2

Greedy Algorithms

- Main Concept: Make the *best* or *greedy* choice at any given step.
  - » This is what you did before you learned to not to think ahead and plan ☺
- Choices are made in sequence such that
  - » Each individual choice is best according to some limited "short-term" criterion, that is not too expensive to evaluate
  - » Once a choice is made, it cannot be undone!
    - ❖ Even if it becomes evident later that it was a poor choice
    - ❖ Sometimes life is like that ☹
- The goal is to make progress by choosing an action that
  - » Incurs the minimum short-term cost,
  - » With the hope that a lot of small short-term costs add up to small overall cost.

3

When to be Greedy

- Greedy algorithms apply to problems with
  - » The *greedy choice property*: an optimal solution can be obtained by making the greedy choice at each step.
  - » *Optimal substructures*: optimal solutions contain optimal sub-solutions.
  - » Many optimal solutions, but we only need one such solution.
- Unlike dynamic programming, we do not need to know the solutions to the sub-problems to make choices.
  - » Hence, greedy algorithms are often more efficient than dynamic programming.
- Possible drawback:
  - » Actions with a small short-term cost may lead to a situation, where further large costs are unavoidable.

4

The Fractional Knapsack Problem

- A thief breaks into a cookie store.
- She has a bag that can hold up to C pounds of cookies.
- There are n cookies in the store.
- The i^{th} cookie weighs w_i pounds and is worth v_i dollars.
- She can break the cookies and sell fractions of them.
- The thief wants to maximize the value of the cookies she steals, of course.
- How much of each cookie should she steal?

5

Some Greedy Observations

- The i^{th} cookie is worth $p_i = v_i/w_i$ dollars per pound.
- The item with the largest p_i has the most "bang for the buck," so the thief should take as much as she can of this item.
- If the thief takes x pounds of a cookie with $p_j < p_i$ instead of cookie i, her profit will be smaller for the same weight.

6

Fractional Knapsack Problem

- Problem
  - » Given a set of n objects where object i has value v_i per cookie and weight w_i and a knapsack capacity C, determine the fractional amount f_i of each object i to be included in the knapsack such that the profit is maximized while the weight of the included objects does not exceed the knapsack capacity
    - maximize $\sum_{i=1}^{n} v_i f_i$ such that $\sum_{i=1}^{n} w_i f_i \le C$

      where $0 \le f_i \le 1$

7

Strategy: Pick by Density

- Strategy
  - » Sort objects in non-increasing order of profit $p_i = v_i/w_i$
    - That is, $p_1=v_1/w_1 \ge p_2=v_2/w_2 \ge \dots \ge p_n=v_n/w_n$
  - » Consider objects by increasing order of subscript.
  - » When an object is considered choose a maximal amount such that the knapsack capacity is not violated.

8

A Greedy Solution

```
fractionalKnapsack(V, W, capacity, n, KnapSack) {
  sortByDescendingProfit(V,W,n)
  KnapSack = 0;
  capacityLeft = C;
  for (i = 1; (i <= n) && (capacityLeft > 0); ++i) {
     if (W[i] < capacityLeft)
         KnapSack[i] = 1;
         capacityLeft -= W[i];
     else
         KnapSack[i] = capacityLeft/W[i];
         capacityLeft = 0;
     }
  }
}
```

What is the complexity of this algorithm?

9

Knapsack Example

- The thief's knapsack holds 15 pounds
- The cookie inventory has the following properties:

i	1	2	3	4	5	6	7	8
v_i	12	4	5	3	8	8	12	1
w_i	4	3	5	6	1	4	10	4
p_i	3	1.33	1	0.5	8	2	1.2	.25

- Same properties sorted by profit

i	6	1	6	2	7	3	4	8
v_i	8	12	8	4	12	5	3	1
w_i	1	4	4	3	10	5	6	4
p_i	8	3	2	1.33	1.2	1	0.5	.25

10

Knapsack Example Solution

- The thief takes 1 pound of cookie 5, 4 pounds of cookie 1, 4 pounds of cookie 6, 3 pounds of cookie 2, and 3 pounds of cookie 7:

i	6	1	6	2	7	3	4	8
v_i	8	12	8	4	12	5	3	1
w_i	**1**	**4**	**4**	**3**	**3**/10	5	6	4
p_i	8	3	2	1.33	1.2	1	0.5	.25

- Thus, the profit is 8+12 + 8 + 4 + 3.6 = $35.6
- Solution property for non-trivial instances
  - » knapSack = $(1, 1, \dots 1, f_j, 0, 0, \dots 0)$ with $0 < f_j \le 1$
- Is it Optimal?

11

0/1 Knapsack

- f_i is restricted to be either 0 or 1
- Does pick by value work?
- Does pick by weight work?
- Does pick by density work?

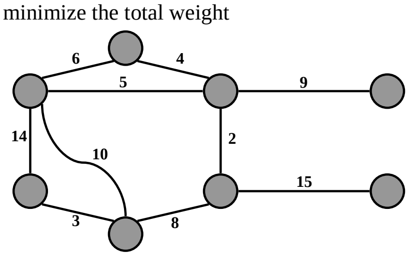
12

Minimum Spanning Tree (MST)

- A *spanning tree* for a connected, undirected graph, $G=(V,E)$ is
  1. a connected subgraph of G that forms an
  2. undirected tree incident with each vertex.
- In a weighted graph $G=(V,E,W)$,
  - » the weight of a subgraph is the sum of the weights of the edges in the subgraph.
- A *minimum spanning tree* (MST) for a weighted graph is
  - » a spanning tree with the minimum weight.

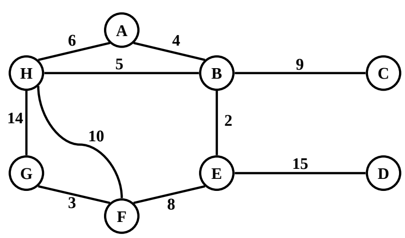
Minimum Spanning Tree

- Problem: given a connected, undirected, weighted graph: find a *spanning tree* using edges that minimize the total weight



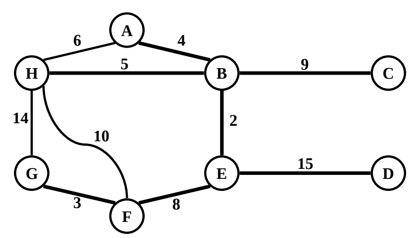
Minimum Spanning Tree

- Which edges form the minimum spanning tree (MST) of the below graph?



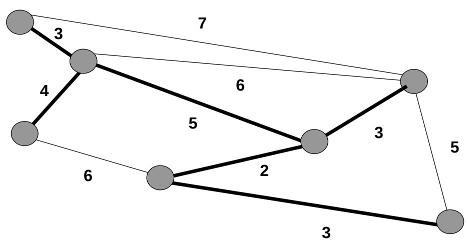
Minimum Spanning Tree

- Answer:



Another Example

- Given a weighted graph $G=(V, E, W)$, find a MST of G



MST Uniqueness

- Is a MST unique?
- What if the weights are (or are not) unique?
- Prove or disprove the uniqueness of a MST for a given graph G.

Minimum Spanning Tree

- MSTs satisfy the *optimal substructure* property: an optimal tree is composed of optimal subtrees
  - » Let T be an MST of G with an edge (*u*,*v*) in the middle
  - » Removing (*u*,*v*) partitions T into two trees T_1 and T_2
  - » Claim: T_1 is an MST of $G_1 = (V_1,E_1)$, and T_2 is an MST of $G_2 = (V_2,E_2)$ (*Do V_1 and V_2 share vertices? Why?*)
  - » Proof: $w(T) = w(u,v) + w(T_1) + w(T_2)$
    (There can't be a better tree than T_1 or T_2, or T would be suboptimal)

Minimum Spanning Tree

- Thm:
  - » Let T be MST of G, and let $A \subseteq T$ be subtree of T
  - » Let (*u*,*v*) be min-weight edge connecting A to V-A
  - » Then $(u,v) \in T$
- Proof: left as an exercise

Finding a MST

- Principal greedy methods: algorithms by Prim and Kruskal
- Prim
  - » Grow a single tree by repeatedly adding the least cost edge that connects a vertex in the existing tree to a vertex not in the existing tree
    - ❖ Intermediary solution is a subtree
- Kruskal
  - » Grow a tree by repeatedly adding the least cost edge that does not introduce a cycle among the edges included so far
    - ❖ Intermediary solution is a spanning forest

Prim's Algorithm

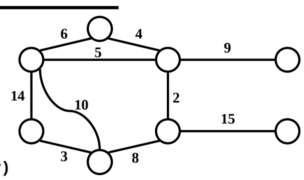
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

Grow a single tree by repeatedly adding the least cost edge that connects a vertex in the existing tree to a vertex not in the existing tree

Intermediary solution is a subtree

Prim's Algorithm

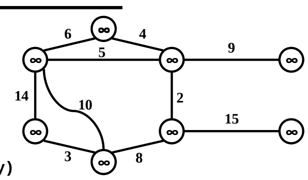
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

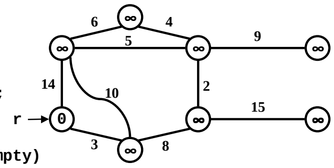
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Run on example graph

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

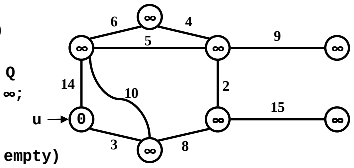


Pick a start vertex r

25

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

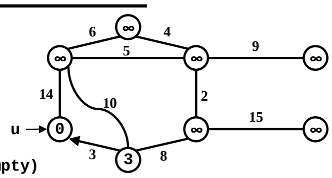


Red vertices have been removed from Q

26

Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

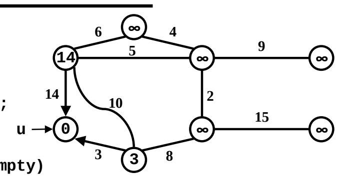


Red arrows indicate parent pointers

27

Prim's Algorithm

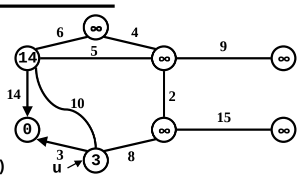
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



28

Prim's Algorithm

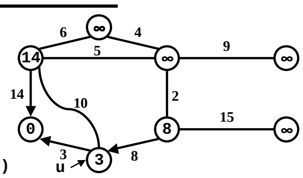
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



29

Prim's Algorithm

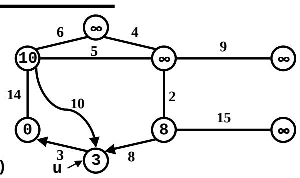
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



30

Prim's Algorithm

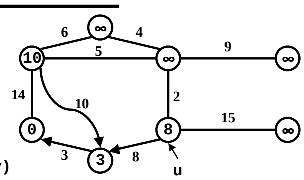
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



31

Prim's Algorithm

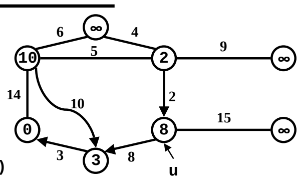
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



32

Prim's Algorithm

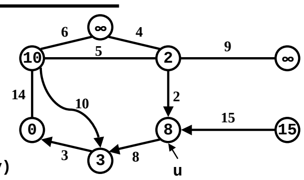
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



33

Prim's Algorithm

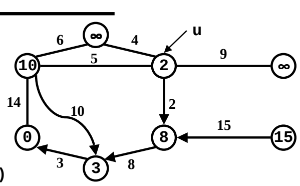
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



34

Prim's Algorithm

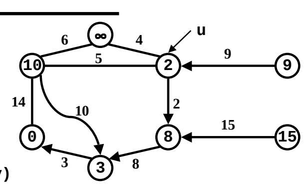
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



35

Prim's Algorithm

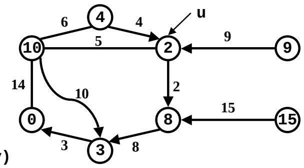
```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



36

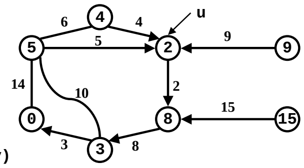
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



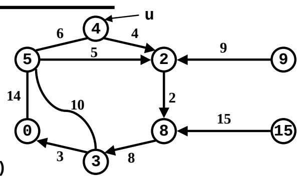
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



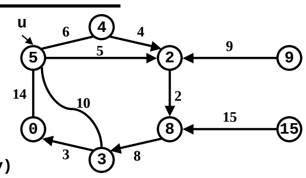
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



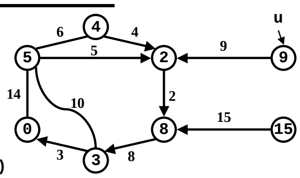
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



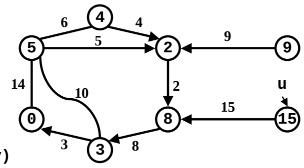
Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```



Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What is the hidden cost in this code?

43

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

44

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                DecreaseKey(v, w(u,v));
```

How often is ExtractMin() called?
How often is DecreaseKey() called?

45

Review: Prim's Algorithm

```
MST-Prim(G, w, r)
    Q = V[G];
    for each u ∈ Q
        key[u] = ∞;
    key[r] = 0;
    p[r] = NULL;
    while (Q not empty)
        u = ExtractMin(Q);
        for each v ∈ Adj[u]
            if (v ∈ Q and w(u,v) < key[v])
                p[v] = u;
                key[v] = w(u,v);
```

What will be the running time?

There are n=|V| ExtractMin calls and m=|E| DecreaseKey calls. Thus, the worst case is $O(n^2+m)$.

The priority Q implementation has a large impact on performance.

E.g., O((n+m)lg n) = O(m lg n) using binary heap for Q
Can achieve O(n lg n + m) with Two-pass pairing or Fibonacci heaps

46

Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

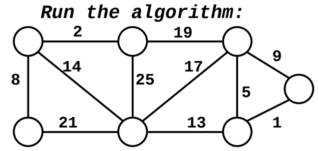
Grow a tree by repeatedly adding the least cost edge that does not introduce a cycle among the edges included so far

Intermediary solution is a spanning forest

47

Kruskal's Algorithm

Run the algorithm:

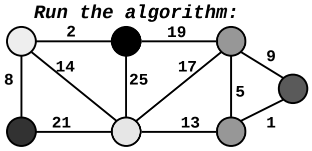



```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

48

Kruskal's Algorithm

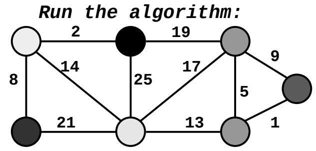
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

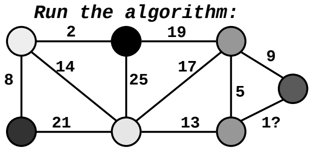
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

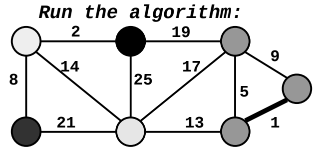
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

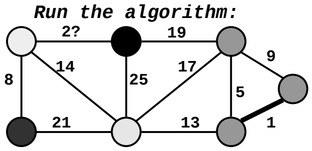
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

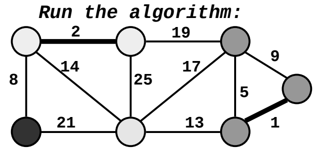
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

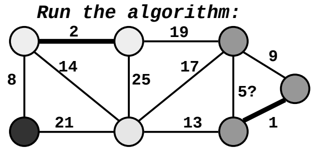
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

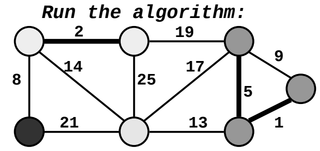
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

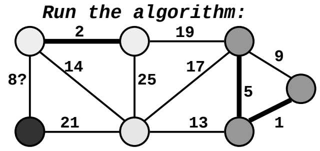
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

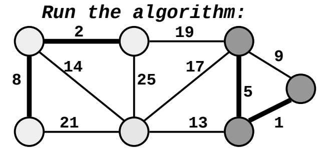
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

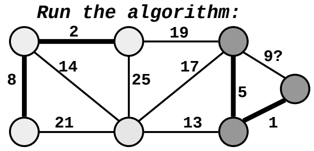
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

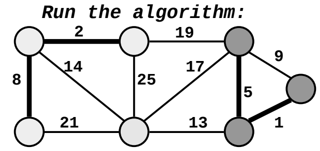
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

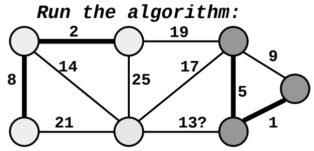
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

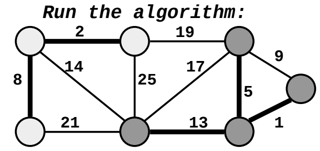
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

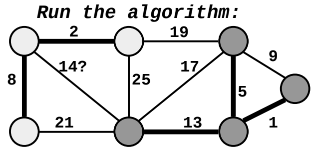
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

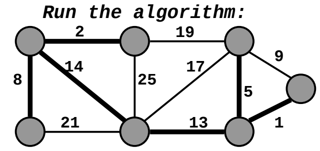
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

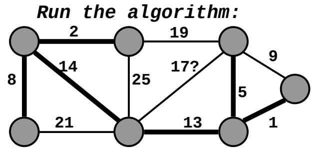
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

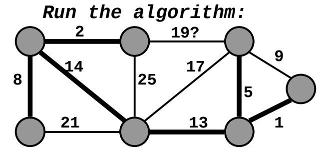
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

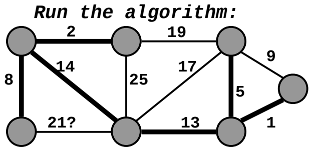
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

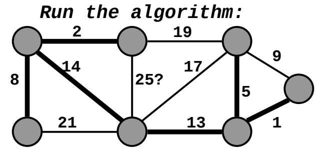
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

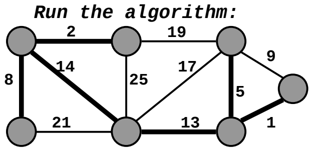
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

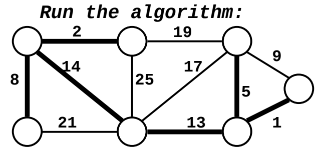
Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Kruskal's Algorithm

Run the algorithm:




```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T U {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```


Correctness Of Kruskal's Algorithm

- Sketch of a proof that this algorithm produces an MST for *T*:
  - » Assume algorithm is wrong: result is not an MST
  - » Then algorithm adds a wrong edge at some point
  - » If it adds a wrong edge, there must be a lower weight edge (cut and paste argument)
  - » But algorithm chooses lowest weight edge at each step. Contradiction
- Again, important to be comfortable with cut and paste arguments

Kruskal's Algorithm

What will affect the running time?

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

Kruskal's Algorithm

```
Kruskal()
{
   T = ∅;
   for each v ∈ V
      MakeSet(v);
   sort E by increasing edge weight w
   for each (u,v) ∈ E (in sorted order)
      if FindSet(u) ≠ FindSet(v)
         T = T ∪ {{u,v}};
         Union(FindSet(u), FindSet(v));
}
```

What will affect the running time?
Let n=|V| and m=|E|
1 Sort
O(n) MakeSet() calls
O(m) FindSet() calls
O(n) Union() calls

Kruskal's Algorithm: Running Time

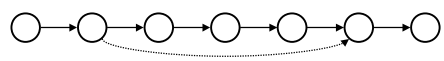
- To summarize:
  - » Sort edges: O(m lg m)
  - » O(n) MakeSet()'s
  - » O(m) FindSet()'s
  - » O(n) Union()'s
- Upshot:
  - » Best disjoint-set union algorithm makes above 3 operations take O(m·α(m,n)), α almost constant
  - » Overall thus O(m lg m), almost linear w/o sorting

Single-Source Shortest Path

- Problem: given a weighted directed graph G, find the minimum-weight path from a given source vertex *s* to another vertex *v*
  - » "Shortest-path" = minimum weight
  - » Weight of path is sum of edges
  - » E.g., a road map: what is the shortest path from Minneapolis to Lincoln?

Shortest Path Properties

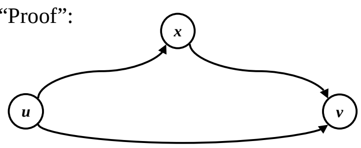
- Again, we have *optimal substructure*: the shortest path consists of shortest subpaths:



  - » Proof: suppose some subpath is not a shortest path
    - There must then exist a shorter subpath
    - Could substitute the shorter subpath for a shorter path
    - But then overall path is not shortest path. Contradiction

Shortest Path Properties

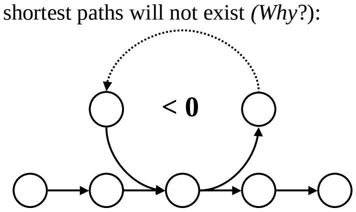
- Define δ(u,v) to be the weight of the shortest path from u to v
- Shortest paths satisfy the *triangle inequality*: δ(u,v) ≤ δ(u,x) + δ(x,v)
- "Proof":



This path is no longer than any other path

Shortest Path Properties

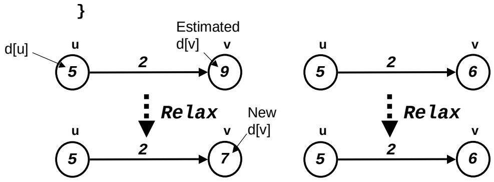
- In graphs with negative weight cycles, some shortest paths will not exist *(Why?)*:



Relaxation

- A key technique in shortest path algorithms is *relaxation*
  - » Idea: for all v, maintain upper bound d[v] on $\delta(s,v)$

```
Relax(u,v,w) {
    if (d[v] > d[u]+w) then d[v]=d[u]+w;
}
```

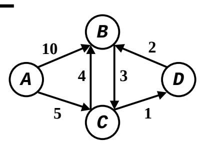


Dijkstra's Algorithm

- If no negative edge weights, we can beat BFS
- Similar to breadth-first search
  - » Grow a tree gradually, advancing from vertices taken from a queue
- Also similar to Prim's algorithm for MST
  - » Use a priority queue keyed on d[v]

Dijkstra's Algorithm

```
Dijkstra(G,s)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ Adj[u]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```



Ex: run the algorithm

Relaxation Step (the last two lines)

Note: this is really a call to Q->DecreaseKey() (the line d[v] = d[u]+w(u,v);)

Dijkstra's Algorithm

```
Dijkstra(G,s)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ Adj[u]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

How many times is ExtractMin() called?

How many times is DecraseKey() called?

What will be the total running time?

Analysis of Dijkstra's Algorithm

```
Dijkstra(G,s)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪ {u};
      for each v ∈ Adj[u]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

There are n=|V| ExtractMin calls and m=|E| DecreaseKey calls. Thus, the worst case is O(n^2+m).

The priority Q implementation has a large impact on performance.

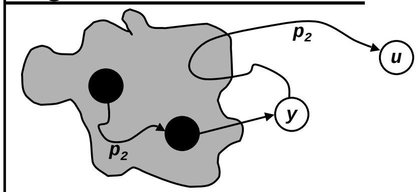
E.g., O((n+m)lg n) = O(m lg n) using binary heap for Q
Can achieve O(n lg n + m) with Fibonacci heaps

Dijkstra's Algorithm

```
Dijkstra(G,s)
   for each v ∈ V
      d[v] = ∞;
   d[s] = 0; S = ∅; Q = V;
   while (Q ≠ ∅)
      u = ExtractMin(Q);
      S = S ∪{u};
      for each v ∈ Adj[u]
         if (d[v] > d[u]+w(u,v))
            d[v] = d[u]+w(u,v);
```

Correctness: we must show that when u is removed from Q, it has already converged

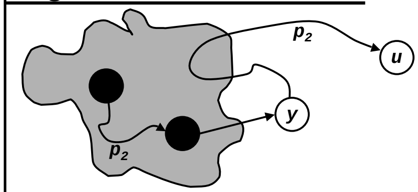
Correctness Of Dijkstra's Algorithm



- Note that d[v] ≥ δ(s,v) ∀v
- Let u be first vertex picked s.t. ∃ shorter path than d[u] ⇒d[u] > δ(s,u)
- Let y be first vertex ∈ V-S on actual shortest path from s→u ⇒ d[y] = δ(s,y)
  - » Because d[x] is set correctly for y's predecessor x ∈ S on the shortest path, and
  - » When we put x into S, we relaxed (x,y), giving d[y] the correct value

85

Correctness Of Dijkstra's Algorithm



- Note that d[v] ≥ δ(s,v) ∀v
- Let u be first vertex picked s.t. ∃ shorter path than d[u] ⇒d[u] > δ(s,u)
- Let y be first vertex ∈ V-S on actual shortest path from s→u ⇒ d[y] = δ(s,y)
- d[u] > δ(s,u)
  = δ(s,y) + δ(y,u) (*Why?*)
  = d[y] + δ(y,u)
  ≥ d[y] But if d[u] > d[y], wouldn't have chosen u. Contradiction.

86