Kruskal's MST algorithm

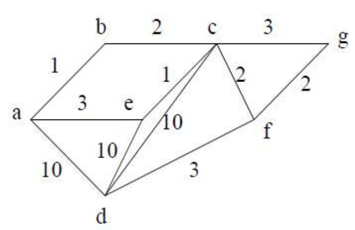
- **Prim's** algorithm adds the next closest vertex.
- **Kruskal's** algorithm adds the next lowest weight edge *that* ***doesn't form a cycle***.

Kruskal's Algorithm for MST

```
E1: edges in MST so far
E2: remaining edges

E1=EMPTYSET,E2=E
Sort edges in E2 by weight
while |E1| < |V|-1 edges and E2 not EMPTYSET
      Pick min cost edge e(i,j) from E2
      E2 = E2 \ e(I,j)
      if V(i),V(j) are not in same MST-so far, then
            E1 = E1 Union e(I,j)
            unite MSTs with V(i) and V(j)
```


Kruskal's




```
if V(i),V(j) are not in same MST-so far, then
            unite MSTs with V(i) and V(j)
```

- Prevents cycles (not in same MST-so far)
- Unites MSTs (new edge in MST-so far)

Kruskal's algorithm

Sort edges:

- ? log ?

E* get next edge and check for cycle

E * merge subsets

Kruskal's Algorithm for MST

```
E1: edges in MST so far
E2: remaining edges

E1=EMPTYSET,E2=E
Sort edges in E2 by weight
while |E1| < |V|-1 edges and E2 not EMPTYSET
      Pick min cost edge e(i,j) from E2
      E2 = E2 \ e(I,j)
      if V(i),V(j) are not in same MST-so far, then
            E1 = E1 Union e(I,j)
            unite MSTs with V(i) and V(j)
```

??


```
if V(i),V(j) are not in same MST-so far, then
            unite MSTs with V(i) and V(j)
```

- Prevents cycles (not in same MST-so far)
- Unites MSTs (new edge in MST-so far)

Sounds easy, but…

…. **requires new data structure and algorithm**

- Disjoint-set data structure
- Union-find algorithm

Union-find

- Have disjoint (non-overlapping) subsets
  - Find: Which subset is an element in?
  - Union: Join two subsets into a single subset
- For Kruskal's algorithm:
  - **Find**: Is the new edge in an existing subset?
    - If yes, this is a cycle! – don't use!
  - **Union**: Does the new edge join two subjects?
    - If yes, join the two subsets

Union-find

- Have disjoint (non-overlapping) subsets
  - Find: Which subset is an element in?
  - Union: Join two subsets into a single subset

Union-find

- Have disjoint (non-overlapping) subsets
  - Find: Which subset is an element in?
  - Union: Join two subsets into a single subset
- Naïve union-find: array

Union-find: array

- Have disjoint (non-overlapping) subsets
  - Find: Which subset is an element in?
  - Union: Join two subsets into a single subset
- Naïve union-find: array
- id[]

0	1	2	3	4	5	6	7

Union-find: array

- Start: Singleton Sets
- id[]

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

Example: Put (2, 3) in same set, and (4,6):

- change entry in id[]: choose representatives

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

Union-find: array

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

Union-find: array

- Naïve algorithm, using array:
- Find:
  - id[p] == id[q]
  - O(?)
- Union:
  - id[p and all in same subset] = id[q]
  - O(?)

Speeding up the Union in Union-Find

- Speed up union: tree-based approach
  - id[] is a parent array
  - **Root** is the **representative of the subset**

To union two subsets – make the root of one the parent of the root of the other

- O(?)

0	1	2	3	4	5	6	7

Find in Tree-based Union-Find

- Find:
  - Traverse back through parent array to root
  - Nodes are in the same subset if they have the same root
  - O(?)

Time for trace depends on depth of tree

0	1	2	3	4	5	6	7

Improvements in Union-Find

- Find:
  - Time for trace depends on depth of tree
  - Weighted: merge smaller tree into larger
    - keeps tree broader
  - Path compression
- Analysis: E union-finds on V vertices
  - Naïve: O(EV)
  - Weighted or path compress: O(V + E log V)
  - Weighted AND path compress: O(E+V) α(V)

    ≈ O(E+V)

1-44

Union-Find Analysis

- Analysis: E union-finds on V vertices
  - Naïve: O( )
    - Array: O( ) find; O( ) union
    - Tree: O( ) union; O( ) find
  - Weighted OR path compress: O(V + E log V)
  - Weighted AND path compression:
    - O(E*α(E,V) + V)
    - α(n): inverse Ackermann function, small constant
    - ≈ O(E+V)

1-45

Kruskal's: Analysis with best union-find

- Sort edges:
  - ? log ?
- E finds and E unions:
  - E+V
- O(E log E + E + V) = O(E log E)

- Time is dominated by sorting the edges!

COMP 20003 Algorithms and Data Structures 1-46

Kruskal's: Analysis with best union-find

- Time is dominated by sorting the edges!

Any ideas for what we might do?

COMP 20003 Algorithms and Data Structures 1-47

Improvement to Kruskals: Partial sort

When sorting dominates performance, partial sorting can help…

… only need the smallest V-1 edges

e.g. quicksort-like partition, but

- Works if graph is connected
- Doesn't work if longest edge needs to be in MST
  - *e.g.* tight clusters connected by one or more long edges

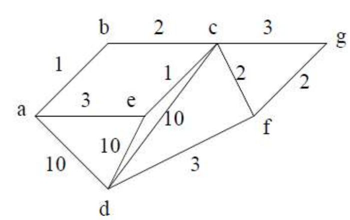
	Prim	Kruskal
General	?	?
Dense Graph	E log V	E log E
V<<E, Prim's is faster		
Sparse Graph	V log V	V log V
Kruskal's is faster because of the data structures		

Kruskal's algorithm: an overview (Skiena)

A large-scale view of Kruskal's algorithm:

http://www.cs.sunysb.edu/~skiena/combinatorica/animations/mst.html

Kruskal's *vs.* Prim's



More advanced MSTs

- Euclidean MSTs:
  - Given points on a plane, build MST
  - Could construct complete graph, then use Prim's. – Slow!

Other more clever algorithms exist

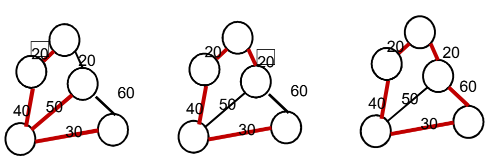
More advanced MSTs

- Randomized MST algorithm
  - Random partition of the graph
  - Expected time linear, but bad worst case
  - Karger, David R.; Klein, Philip N.; Tarjan, Robert E. (1995). "A randomized linear-time algorithm to find minimum spanning trees". *JACM* **42** (2): 321–328.
  - Linear MST algorithms exist for restricted types of graphs

The general solution for linear time MST creation is an open research problem

MST and the Travelling Salesperson Problem

- Travelling salesperson problem (TSP):
  - Given a list of cities and the distances between each pair of cities, find:
    - shortest possible route that
    - visits each city exactly once
    - and returns to the origin city



MST and the Travelling Salesperson Problem

- Travelling salesperson problem (TSP):
  - Given a list of cities and the distances between each pair of cities, find:
    - shortest possible route that
    - visits each city exactly once
    - and returns to the origin city
- Much harder than MST!
- Greedy (nearest neighbor) doesn't work!

Graph algorithms

- Graph search:
  - Depth-first
  - Breadth-first
  - Priority-first
- Undirected graphs
- Directed graphs

COMP 20003 Algorithms and Data Structures 1-56

Graph algorithms

- Graph search
- Algorithms on undirected graphs
- Algorithms on directed graphs

COMP 20003 Algorithms and Data Structures 1-57

Graph algorithms

- Graph search
  - Depth-first search
  - Breadth-first search
  - Priority-first search
  - (Connected components)
- Algorithms on undirected graphs
- Algorithms on directed graphs

COMP 20003 Algorithms and Data Structures 1-58

Graph algorithms

- Graph search
- Algorithms on undirected graphs
- Algorithms on directed graphs
  - Single source shortest path (Dijkstra's)
  - Transitive closure (Warshall)
  - All pairs shortest path (Floyd-Warshall)

COMP 20003 Algorithms and Data Structures 1-59

Graph algorithms

- Graph search
- Algorithms on undirected graphs
  - Minimum spanning tree
    - Prim's
    - Kruskal's
  - Travelling salesperson
- Algorithms on directed graphs

Graphs in the real world

- Many real-world problems can be modelled as graphs
- Many specialized types of graphs allow modelling of complex problems
- People have been working on graph algorithms for a long time, so
- Huge library of algorithms available

Take away lesson

If you can model a problem as a graph, there is a very good chance that there is already an algorithm to solve the problem…

… or evidence that the problem is intractible