CSE 332 Data Abstractions: Disjoint Set Union-Find and Minimum Spanning Trees

Kate Deibel

Summer 2012

Making Connections

You have a set of nodes (numbered 1-9) on a network. You are given a sequence of pairwise connections between them:

3-5 4-2 1-6 5-7 4-8 3-7

Q: Are nodes 2 and 4 connected? Indirectly?

Q: How about nodes 3 and 8?

Q: Are any of the paired connections redundant due to indirect connections?

Q: How many sub-networks do you have?

Making Connections

Answering these questions is much easier if we create disjoint sets of nodes that are connected:

Start:	{1} {2} {3} {4} {5} {6} {7} {8} {9}
3-5	{1} {2} {3, 5} {4} {6} {7} {8} {9}
4-2	{1} {2, 4} {3, 5} {6} {7} {8} {9}
1-6	{1, 6} {2, 4} {3, 5} {7} {8} {9}
5-7	{1, 6} {2, 4} {3, 5, 7} {8} {9}
4-8	{1, 6} {2, 4, 8} {3, 5, 7} {9}
3-7	*no change*

Making Connections

Let's ask the questions again.

3-5 4-2 1-6 5-7 4-8 3-7

⇓

{1, 6} {2, 4, 8} {3, 5, 7} {9}

Q: Are nodes 2 and 4 connected? Indirectly?

Q: How about nodes 3 and 8?

Q: Are any of the paired connections redundant due to indirect connections?

Q: How many sub-networks do you have?

Disjoint Set Union-Find ADT

Separate elements into disjoint sets
- If set x ≠ y then x ∩ y = ∅ (i.e. no shared elements)

Each set has a name (usually an element in the set)

union(x,y): take the union of the sets x and y (x ∪ y)
- Given sets: {3,5,7} , {4,2,8}, {9}, {1,6}
- union(5,1) → {3,5,7,1,6}, {4,2,8}, {9},

find(x): return the name of the set containing x.
- Given sets: {3,5,7,1,6}, {4,2,8}, {9},
- find(1) returns 5
- find(4) returns 8

Disjoint Set Union-Find Performance

Believe it or not:
- We can do Union in constant time.
- We can get Find to be ***amortized*** constant time with worst case O(log n) for an individual Find operation

Let's see how…

Beware of Minotaurs

FIRST, LET'S GET LOST

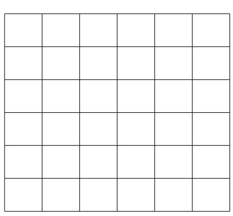
What Makes a Good Maze?

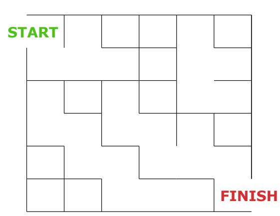
- We can get from any room to any other room (connected)
- There is just one simple path between any two rooms (no loops)
- The maze is not a simple pattern (random)

Making a Maze

A high-level algorithm for a random maze is easy:

- Start with a grid
- Pick Start and Finish
- Randomly erase edges





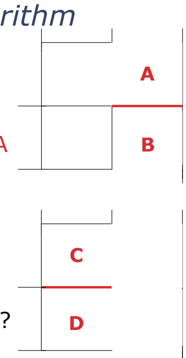
The Middle of the Algorithm

So far, we've knocked down several walls while others still remain.

Consider the walls between A and B and C and D

- Which walls can we knock down and maintain both our connectedness and our no cycles properties?

How do we do this efficiently?



Maze Algorithm: Number the Cells

Number each cell and treat as disjoint sets:

- S ={ {1}, {2}, {3}, {4},… {36} }

Create a set of all edges between cells:

- W ={ (1,2), (1,7), (2,8), (2,3), … } 60 walls total.

START	1	2	3	4	5	6	
	7	8	9	10	11	12	
	13	14	15	16	17	18	
	19	20	21	22	23	24	
	25	26	27	28	29	30	
	31	32	33	34	35	36	FINISH

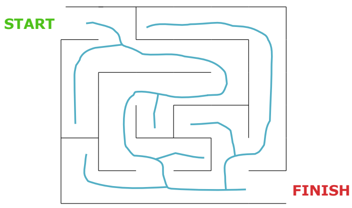
Maze Algorithm: Building with DSUF

Algorithm sketch:

- Choose a wall at random.
- Erase wall if the neighbors are in disjoint sets (this avoids creating cycles)
- Take union of those cell's sets
- Repeat until there is only one set
  - Every cell is thus reachable from every other cell

The Secret To Why This Works

Notice that a connected, acyclic maze is actually a Hidden Tree



This suggests how we should implement the Disjoint Set Union-Find ADT

I promise the first twenty minutes of this section will not be the saddest trees you have ever seen…

IMPLEMENTING DSUF WITH UP TREES

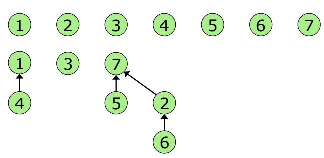
Up Trees for Disjoin Set Union-Find

Up trees
- Notes point to parent, not children
- Thus only one pointer per node

In a DSUF
- Each disjoint set is its own up tree
- The root of the tree is the name for the disjoint set

Initial State

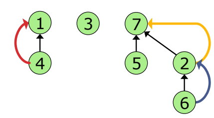
After Unions



Find Operation

find(x): follow x to the root and return the root (the name of the disjoint set)

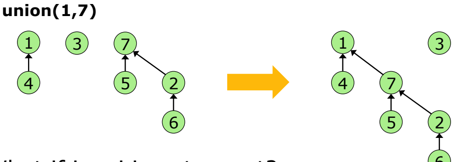
find(1) = 1
find(3) = 3
find(4) = 1
find(6) = 7



Find Operation

union(i,j): assuming i and j are roots, point root i to root j

union(1,7)



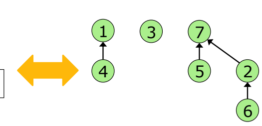
What if i or j is not a root?
- Run a find on i and j first and use the returned values for the joining

Why do we join roots and not just the nodes?

Simple Implementation

Once again, it is better to implement a tree using an array than with node objects
- Leave up[0] empty (or # of disjoint sets)
- up[x] = i means node x's parent is node i
- up[x] = 0 means x is a root

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



Performance

Using array-based up trees, what is the cost for
- union(i,j)?
- find(x)?

union(i,j) is O(1) if i and j are roots
- Otherwise depends on cost of find

find(x) is O(n) in worst-case
- What does the worst-case look like?

Performance – Doing Better

The problem is that up trees get too tall

In order to make DSUF perform as we promised, we need to improve both our union and find algorithms:
- Weighted Union
- Path Compression

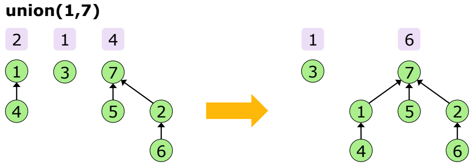
Only with BOTH of these will we get find to average-case O(log n) and amortized O(1)

Weighted Union

Instead of arbitrarily joining two roots, always point the smaller tree to the root of the larger tree
- Each up tree has a weight (number of nodes)
- The idea is to limit the height of each up tree
- Trees with more nodes tend to be deeper

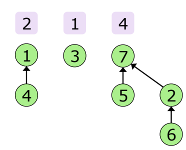
Union by rank or height are similar ideas but more complicated to implement

union(1,7)



Weighted Union Implementation

We can just use an additional array to store weights of the roots…

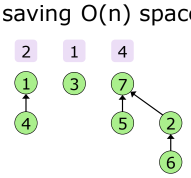


	1	2	3	4	5	6	7
up	0	7	0	1	7	2	0
weight	2		1				4

Weighted Union Implementation

…or we use negative numbers to represent roots and their weights

But generally, saving O(n) space is not critical



	1	2	3	4	5	6	7
up	-2	7	-1	1	7	2	-4

Weighted Union Performance

Weighted union gives us guaranteed worst-case O(log n) for find
- The union rule prevents linear up trees
- Convince yourself that it will produce at worst a fairly balanced binary tree

However, we promised ourselves O(1) *amortized* time for find
- Weighted union does not give us enough
- Average-case is still O(log n)

Motivating Path Compression

Recall splay trees

- To speed up later finds, we moved searched for nodes to the root
- Also improved performance for finding other nodes
- Can we do something similar here?

Yes, but we cannot move the node to the root

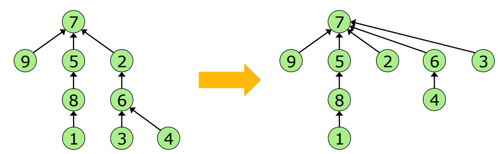
- Roots are the names of the disjoint set
- Plus, we want to move associated nodes up at the same time
- Why not move all nodes touched in a find to point directly to the root?

Path Compression

On a find operation point all the nodes on the search path directly to the root

- Keep a stack/queue as you traverse up
- Then empty to the stack/queue to repoint each stored node to the root

find(3)



Digression: Ackermann Function

The Ackermann function is a recursive function that grows exceptionally fast

$$A(x,y) = \begin{cases} y+1, & x = 0 \\ A(x-1,1), & y = 0 \\ A(x-1, A(x, y-1)), & otherwise \end{cases}$$

If $\mathrm{ack}(x) = A(x,x)$, then the first few values are:

$ack(0) = 1$
$ack(1) = 3$
$ack(2) = 7$
$ack(3) = 61$
$ack(4) = 2^{2^{2^{65536}}} - 3$ (WOW!!)

Digression: Inverse Ackermann

Just as fast as the Ackermann function grows, its inverse, $ack^{-1}(n)$, grows veeeeeeeerrrrrrrrrrrryyyyy slowly

In fact, $ack^{-1}(n)$ grows more slowly than the following:

- Let $\log^{(k)}$ n = $\underbrace{\text{log (log (log … (log n)))}}_{k \text{ logs}}$
- Then, let log* n = minimum *k* such that $\log^{(k)}$ n ≤ 1

How fast does $\log^*$ n grow?

$\log^*$ (2) = 1
$\log^*$ (4) = 2
$\log^*$ (16) = 3
$\log^*$ (65536) = 4
$\log^*$ (2^{65536}) = 5 (a 20,000 digit number!)
$\log^*$ ($2^{2^{65536}}$) = 6

Optimized Disjoint Set Union-Find

Tarjan (1984) proved that **m** weighted union and find with path compression operations on a set of **n** elements have worst case complexity O(m·ack^{-1}(n))

- For **all** practical purposes this is amortized constant time as ack^{-1}(n) < 5 for reasonable n

More generally, the total cost of m finds (with at most n-1 unions—why?), the total work is: O(m+n)

- Again, this is O(1) amortized with O(1) worst-case for union and O(log n) worst-case for find
- One can also show that any implementation of find and union cannot both be worst-case O(1)

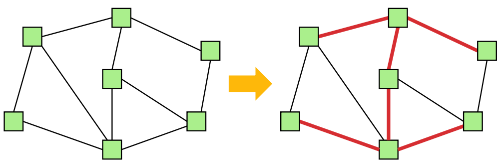
With no surprise, DSUF will be very useful here

MINIMUM SPANNING TREES

General Problem: Spanning a Graph

A simple problem: Given a *connected* graph G=(V,E), find a minimal subset of the edges such that the graph is still connected

- A graph G_2=(V,E2) such that G_2 is connected and removing any edge from E_2 makes G_2 disconnected

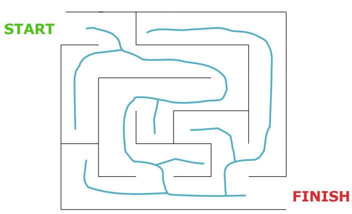


Observations

1. Any solution to this problem is a tree
   - Recall a tree does not need a root; just means acyclic
   - For any cycle, could remove an edge and still be connected
   - We usually just call the solutions spanning trees
2. Solution not **unique** unless original graph was already a tree
3. Problem ill-defined if original graph not connected
   - We can find a spanning tree per connected component of the graph
   - This is often called a *spanning forest*
4. A tree with **|V|** nodes has **|V|-1** edges
   - This every spanning tree solution has **|V|-1** edges

We Saw This Earlier

Our acyclic maze consisted of a tree that touched ever square of the grid



Motivation

A spanning tree connects all the nodes with as few edges as possible

Example: A "phone tree" so everybody gets the message and no unnecessary calls get made

- Bad example since would prefer a balanced tree

In most compelling uses, we have a *weighted* undirected graph and want a tree of least total cost

- Minimize electrical wiring for a house or wires on a chip
- Minimize road network if you cared about asphalt cost

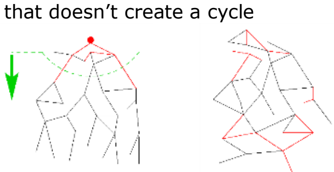
This is the minimum spanning tree problem

- Will do that next, after intuition from the simpler case

Finding Unweighted Spanning Trees

Different algorithmic approaches to the spanning-tree problem:

1. Do a graph traversal (e.g., depth-first search, but any traversal will do) and keep track of edges that form a tree
2. or, iterate through edges and add to output any edge that doesn't create a cycle



Spanning Tree via DFS

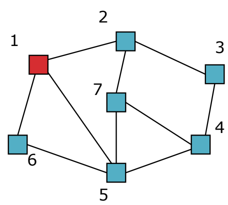
```
spanning_tree(Graph G) {
  for each node i: i.marked = false
  for some node i: f(i)
}
f(Node i) {
  i.marked = true
  for each j adjacent to i:
    if(!j.marked) {
      add(i,j) to output
      f(j) // DFS
    }
}
```

Correctness:
DFS reaches each node. We add one edge to connect it to the already visited nodes. Order affects result, not correctness.

Time: $O(|E|)$

DFS Spanning Tree Example

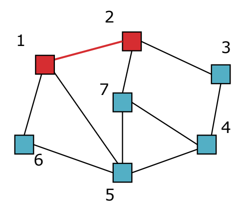
Stack
f(1)



Output:

DFS Spanning Tree Example

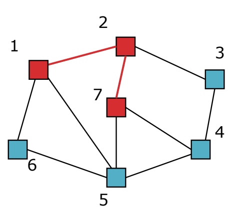
Stack
f(1)
f(2)



Output: (1,2)

DFS Spanning Tree Example

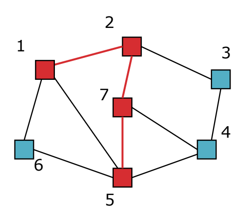
Stack
f(1)
f(2)
f(7)



Output: (1,2), (2,7)

DFS Spanning Tree Example

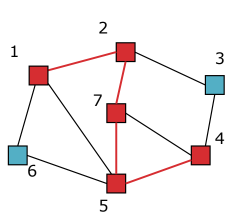
Stack
f(1)
f(2)
f(7)
f(5)



Output: (1,2), (2,7), (7,5)

DFS Spanning Tree Example

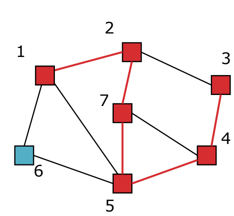
Stack
f(1)
f(2)
f(7)
f(5)
f(4)



Output: (1,2), (2,7), (7,5), (5,4)

DFS Spanning Tree Example

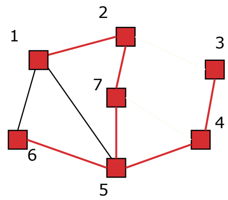
Stack
f(1)
f(2)
f(7)
f(5)
f(4)
f(3)



Output: (1,2), (2,7), (7,5), (5,4), (4,3)

DFS Spanning Tree Example

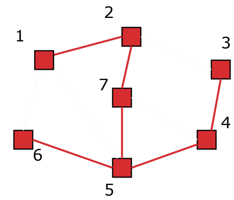
Stack
f(1)
f(2)
f(7)
f(5)
f(4)
f(3)
f(6)



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

DFS Spanning Tree Example

Stack



Output: (1,2), (2,7), (7,5), (5,4), (4,3), (5,6)

Second Approach

Iterate through edges; output any edge that does not create a cycle

Correctness (hand-wavy):

- Goal is to build an acyclic connected graph
- When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
- The graph is connected, we consider all edges

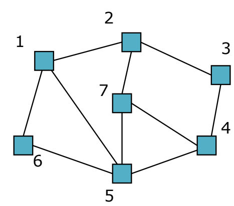
Efficiency:

- Depends on how quickly you can detect cycles
- Reconsider after the example

Example

Edges in some arbitrary order:

(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

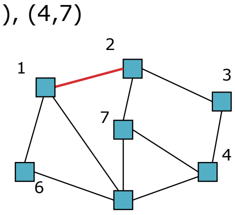


Output:

Example

Edges in some arbitrary order:

, (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

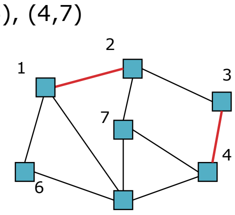


Output: (1,2)

Example

Edges in some arbitrary order:

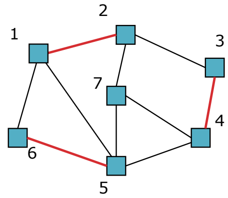
, , (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4)

Example

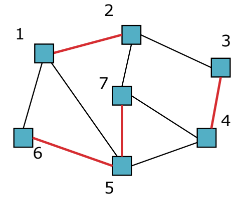
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7),(1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6)

Example

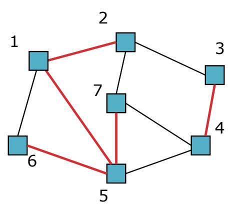
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7)

Example

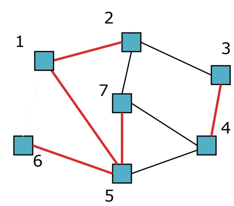
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

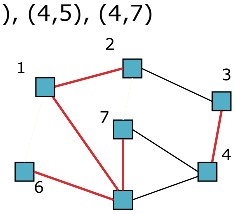
Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)



Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

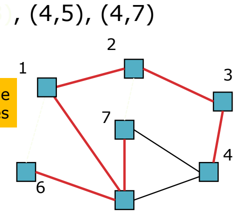


Output: (1,2), (3,4), (5,6), (5,7), (1,5)

Example

Edges in some arbitrary order:
(1,2), (3,4), (5,6), (5,7), (1,5), (1,6), (2,7), (2,3), (4,5), (4,7)

Can stop once we have |V|-1 edges



Output: (1,2), (3,4), (5,6), (5,7), (1,5), (2,3)

Cycle Detection

To decide if an edge could form a cycle is *O*(|V|) because we may need to traverse all edges already in the output

- So overall algorithm would be *O*(|V||E|)

But it is faster way to use the DSUF ADT

- Initially, each vertex is in its own 1-element set
- find(u): what set contains u?
- union(u,v): combine the sets containing u and v

Using Disjoint-Set to Detect Cycles

Invariant:
u and **v** are connected in output-so-far if and only if **u** and **v** in the same set

Algorithm:

- Initially, each node is in its own set
- When processing edge **(u,v)**:
  - If **find(u)==find(v)**, then do not add the edge
  - Else add the edge and **union(u,v)**

Summary so Far

The spanning-tree problem

- Add nodes to partial tree approach is *O*(**|E|**)
- Add acyclic edges approach is *O*(**|E| `log` |V|**)

But what we really want to solve is the minimum-spanning-tree problem

- Given a weighted undirected graph, find a spanning tree of minimum weight
- The above approaches suffice with minor changes
- Both will be *O*(**|E| `log` |V|**)

Like vi versus emacs except people do not typically fight over which one is better (emacs and Kruskal are best!)

PRIM AND KRUSKAL'S ALGORITHMS

One Problem, Two Algorithms

Algorithm #1: Prim's Algorithm

- Shortest-path is to Dijkstra's Algorithm as Minimum Spanning Tree is to Prim's Algorithm
- Both based on expanding cloud of known vertices, basically using a priority queue

Algorithm #2: Kruskal's Algorithm

- Exactly our forest-merging approach to spanning tree but process edges in cost order

Idea: Prim's Algorithm

Central Idea:

- Grow a tree by adding an edge from the "known" vertices to the "unknown" vertices.
- Pick the edge with the smallest weight that connects "known" to "unknown."

Recall Dijkstra picked "edge with closest known distance to source."

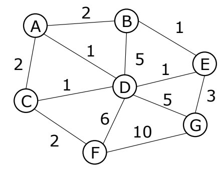
- But that is not what we want here
- Otherwise identical
- Feel free to look back and compare

Pseudocode: Prim's Algorithm

1. For each node `v`, set `v.cost = ∞` and `v.known = false`
2. Choose any node `v`.
   a) Mark `v` as known
   b) For each edge `(v,u)` with weight `w`, set `u.cost = w` and `u.prev = v`
3. While there are unknown nodes in the graph
   a) Select the unknown node `v` with lowest cost
   b) Mark `v` as known and add `(v, v.prev)` to output
   c) For each edge `(v,u)` with weight `w`,

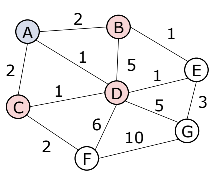
```
if(w < u.cost) {
   u.cost = w;
   u.prev = v;
}
```


Example: Prim's Algorithm



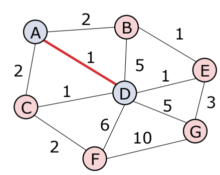
vertex	known?	cost	prev
A			
B			
C			
D			
E			
F			
G			

Example: Prim's Algorithm



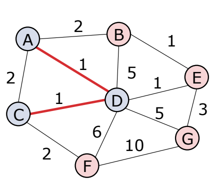
vertex	known?	cost	prev
A	Y	0	-
B		2	A
C		2	A
D		1	A
E			
F			
G			

Example: Prim's Algorithm



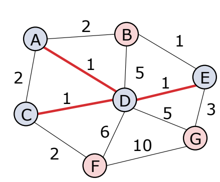
vertex	known?	cost	prev
A	Y	0	-
B		2	A
C		~~2~~ 1	~~A~~ D
D	Y	1	A
E		1	D
F		6	D
G		5	D

Example: Prim's Algorithm



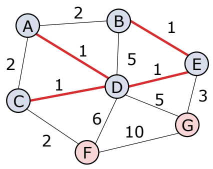
vertex	known?	cost	prev
A	Y	0	-
B		2	A
C	Y	~~2~~ 1	~~A~~ D
D	Y	1	A
E		1	D
F		~~6~~ 2	~~D~~ C
G		5	D

Example: Prim's Algorithm



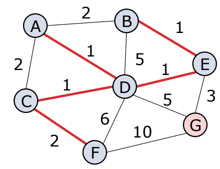
vertex	known?	cost	prev
A	Y	0	-
B		~~2~~ 1	~~A~~ E
C	Y	~~2~~ 1	~~A~~ D
D	Y	1	A
E	Y	1	D
F		~~6~~ 2	~~D~~ C
G		~~5~~ 3	~~D~~ E

Example: Prim's Algorithm



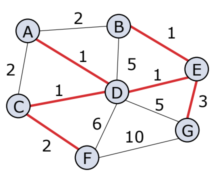
vertex	known?	cost	prev
A	Y	0	-
B	Y	~~2~~ 1	~~A~~ E
C	Y	~~2~~ 1	~~A~~ D
D	Y	1	A
E	Y	1	D
F		~~6~~ 2	~~D~~ C
G		~~5~~ 3	~~D~~ E

Example: Prim's Algorithm



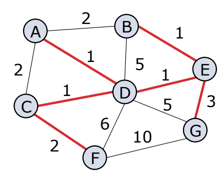
vertex	known?	cost	prev
A	Y	0	-
B	Y	~~2~~ 1	~~A~~ E
C	Y	~~2~~ 1	~~A~~ D
D	Y	1	A
E	Y	1	D
F	Y	~~6~~ 2	~~D~~ C
G		~~5~~ 3	~~D~~ E

Example: Prim's Algorithm



vertex	known?	cost	prev
A	Y	0	-
B	Y	~~2~~ 1	~~A~~ E
C	Y	~~2~~ 1	~~A~~ D
D	Y	1	A
E	Y	1	D
F	Y	~~6~~ 2	~~D~~ C
G	Y	~~5~~ 3	~~D~~ E

Example: Prim's Algorithm



Output:

(A, D) (C, F)
(B, E) (D, E)
(C, D) (E, G)

Total Cost: 9

vertex	known?	cost	prev
A	Y	0	-
B	Y	~~2~~ 1	~~A~~ E
C	Y	~~2~~ 1	~~A~~ D
D	Y	1	A
E	Y	1	D
F	Y	~~6~~ 2	~~D~~ C
G	Y	~~5~~ 3	~~D~~ E

Analysis: Prim's Algorithm

Correctness

- Intuitively similar to Dijkstra's algorithm

Run-time

- Same as Dijkstra's algorithm
- O(|E| `log` |V|) using a priority queue

Idea: Kruskal's Algorithm

Central Idea:

- Grow a forest out of edges that do not grow a cycle, just like for the spanning tree problem.
- But now consider the edges in order by weight

Basic implementation:

- Sort edges by weight → O(|E| `log` |E|) = O(|E| `log` |V|)
- Iterate through edges using DSUF for cycle detection → O(|E| `log` |V|)

Somewhat better implementation:

- Floyd's algorithm to build min-heap with edges → O(|E|)
- Iterate through edges using DSUF for cycle detection and deleteMin to get next edge → O(|E| `log` |V|)
- Not better worst-case asymptotically, but often stop long before considering all edges

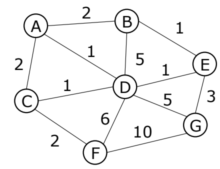
Pseudocode: Kruskal's Algorithm

1. Put edges in min-heap using edge weights
2. Create DSUF with each vertex in its own set
3. While output size < |V|-1
   a) Consider next smallest edge (u,v)
   b) if find(u,v) indicates u and v are in different sets
      - output (u,v)
      - union(u,v)

Recall invariant:

u and v in same set if and only if connected in output-so-far

Example: Kruskal's Algorithm

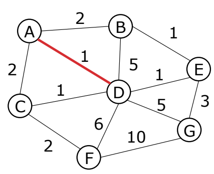


Edges in sorted order:
1: (A,D) (C,D) (B,E) (D,E)
2: (A,B) (C,F) (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A) (B) (C) (D) (E) (F) (G)
Output:

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

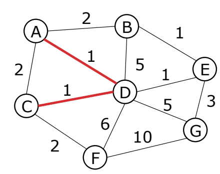


Edges in sorted order:
1: ~~(A,D)~~ (C,D) (B,E) (D,E)
2: (A,B) (C,F) (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,D) (B) (C) (E) (F) (G)
Output: (A,D)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

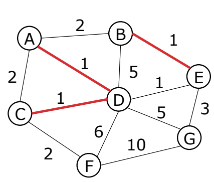


Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ (B,E) (D,E)
2: (A,B) (C,F) (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,C,D) (B) (E) (F) (G)
Output: (A,D) (C,D)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

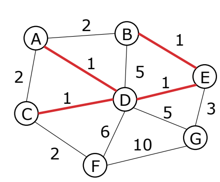


Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ (D,E)
2: (A,B) (C,F) (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,C,D) (B,E) (F) (G)
Output: (A,D) (C,D) (B,E)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

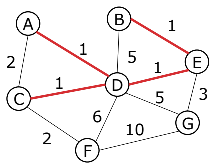


Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~
2: (A,B) (C,F) (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,B,C,D,E) (F) (G)
Output: (A,D) (C,D) (B,E) (D,E)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

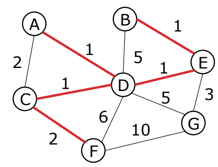


Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~
2: ~~(A,B)~~ (C,F) (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,B,C,D,E) (F) (G)
Output: (A,D) (C,D) (B,E) (D,E)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

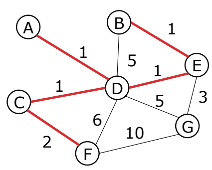


Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~
2: ~~(A,B)~~ ~~(C,F)~~ (A,C)
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,B,C,D,E,F) (G)
Output: (A,D) (C,D) (B,E) (D,E) (C,F)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm

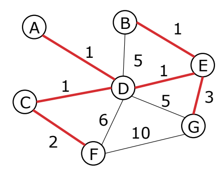


Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~
2: ~~(A,B)~~ ~~(C,F)~~ ~~(A,C)~~
3: (E,G)
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,B,C,D,E,F) (G)
Output: (A,D) (C,D) (B,E) (D,E) (C,F)

At each step, the union/find sets are the trees in the forest

Example: Kruskal's Algorithm



Edges in sorted order:
1: ~~(A,D)~~ ~~(C,D)~~ ~~(B,E)~~ ~~(D,E)~~
2: ~~(A,B)~~ ~~(C,F)~~ ~~(A,C)~~
3: ~~(E,G)~~
5: (D,G) (B,D)
6: (D,F)
10: (F,G)

Sets: (A,B,C,D,E,F,G)
Output: (A,D) (C,D) (B,E) (D,E) (C,F) (E,G)

At each step, the union/find sets are the trees in the forest

Analysis: Kruskal's Algorithm

Correctness: It is a spanning tree
- When we add an edge, it adds a vertex to the tree (or else it would have created a cycle)
- The graph is connected, we consider all edges

Correctness: That it is minimum weight
- Can be shown by induction
- At every step, the output is a subset of a minimum tree

Run-time
- $O(|\mathbf{E}| \log |\mathbf{V}|)$

So Which Is Better?

Time/space complexities essentially the same

Both are fairly simple to implement

Still, Kruskal's is slightly better
- If the graph is not connected, Kruskal's will find a forest of minimum spanning trees

sniff

WRAPPING UP DATA ABSTRACTIONS

That's All Folks

Disjoint Set Union-Find and minimum spanning trees are the last topics we will get to cover

Still, there are plenty more data structures, algorithms and applications out there to learn

You have the basics now

Your Programming Mind has Changed

Before, you often thought first about code
- Declare a variable, a for-loop here, an if-else statement there, etc.

Now, you will see a problem and also think of the data structure
- Lots of lookups… use a hashtable
- Is this a graph and shortest path problem?
- Etc.

Most Important Lesson

There is rarely a best programming solution

Every solution has strengths and weaknesses

The key is to be able to argue in favor of your approach over others

Just remember:
Even though QuickSort's name says it is fast, it is not always the best sort every time

Cheers, Thanks, Whee!

Take care

Fill out the evaluations… I read these!!

Good luck on the final

Remember: Optional Section on Thursday
- Get your final back
- Free doughnuts!
- And maybe another cool data structure