Union-Find (Disjoint-Set)

The *disjoint-set* (or *union-find*) ADT supports the following operations –

- makeset(x) – create a set containing a single element x
- find(x) – determine the set x belongs to
- union(x,y) – merge two sets x and y

In the context of Kruskal's algorithm –

- at the beginning, every vertex is in its own set – makeset(x)
- an edge (u,v) connects different sets if find(u) ≠ find(v)
- adding an edge (u,v) to the spanning tree combines two sets – union(u,v)

Kruskal's Algorithm

Running time using union-find?

- initialization: makeset(v) for each vertex
  - O(makeset) per iteration, n iterations
- finding the lowest-cost edge
  - can sort edges by weight, then iterate through
  - O(m log n) to sort + O(1) time per iteration, m iterations
- determine if an edge connects two separate chunks
  - O(find) per iteration, m iterations
- combine two chunks when an edge is chosen
  - O(union) per edge chosen, n-1 edges chosen

→ total: O(n ✕ makeset + m log n + m ✕ find + n ✕ union)

Implementing Union-Find

A set is an unordered collection of things.

One way to implement a set is with a doubly-linked list.

- makeset(x)
  - create a linked list with a single node containing x
  - O(1)
- union(x,y)
  - append y's list to x's list
  - O(1) if you have tail pointers – set x's tail's next to point to y's head
- find(x)
  - how to identify a set? – could use the head node as the representative of the set
  - given a node, it is O(size of list) to find the head of its list – follow prev pointers backwards from node to the head

→ total: O(n ✕ makeset + m log n + m ✕ find + n ✕ union)
= O(n + m log n + nm + n) = O(nm)

- (this is much worse than graph traversal, can we do better?)

Union-Find Summary

- union-by-rank list implementation yields O((n+m) log n) for Kruskal's algorithm
  - O(1) makeset(x)
  - O(1) find(x)
  - O(n log n) for a series of *n* union(x,y)
- union-by-rank tree implementation with path compression yields O(m log n) for Kruskal's algorithm
  - O(1) makeset(x)
  - effectively O(1) find(x) and union(x,y)
    - the tree height is a very slow-growing log*
    - *amortized* over a series of operations

Both are an improvement over our initial O(nm) algorithm.

Amortized vs. Average

Amortized time is a time-averaged running time.

- based on a worst-case analysis of the running time of an arbitrary sequence of operations
  - worst-case running time of any sequence of *n* operations / *n*
- gives the average worst-case performance of each operation
  - but any particular instance of the operation may be (far) worse
- useful when expensive cases exist but occur infrequently
  - e.g. dynamic array resizing
  - e.g. union-find with path compression
  - e.g. splay trees

Amortized vs. Average

Amortized time is a **time-averaged** running time.

- worst-case analysis of the running time of an arbitrary sequence of operations
  - worst-case running time of any sequence of *n* operations / *n*
- average worst-case performance of each operation
  - any single operation may be (far) worse
  - total for the sequence will not exceed n × operation time

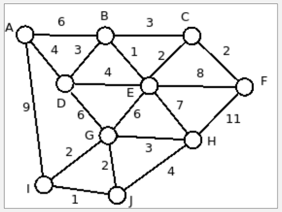
Average time is an **instance-averaged** running time.

- average-case analysis of the running time of an operation
  - based on the probability of each input instance occurring
- expected performance of each operation
  - any single operation may be (far) worse
  - low (but non-zero) probability that total for a sequence will exceed n × operation time

Algorithms for MST

Prim's algorithm –

- start with a tree T containing a single vertex S
- repeatedly add the cheapest edge connecting a vertex in S and a vertex in V-S to T



Prim's Algorithm

The idea:

- repeatedly add the cheapest edge connecting a vertex in S and a vertex in V-S to T

Implementation details:

- cheapest edge connecting S to V-S → ??
  - the set of eligible edges changes as new vertices are added to the tree → sounds like a priority queue ordered by edge weight

```
mark s as visited
add s's incident edges to PQ
while PQ is not empty (and T has fewer than n-1 edges)
  e ← PQ.removeMin()
  if e has an unvisited end vertex v,
    add e to T
    mark v as visited
    add v's incident edges to PQ (omitting those connecting
      to already-visited vertices)
```

Prim's Algorithm

```
mark s as visited
add s's incident edges to PQ
while PQ is not empty (and T has fewer than
 n-1 edges)
  e ← PQ.removeMin()
  if e has an unvisited end vertex v,
    add e to T
    mark v as visited
    add v's incident edges to PQ (omitting
      those connecting to already-visited
      vertices)
```

Running time?

- pick any starting vertex
  - → O(1)
- one iteration per edge
  - → O(m)
- removeMin
  - → O(log m) per iteration, up to m iterations
- traverse incident edges
  - → O(m) total
    - iterate through 2m edges (once from each end) at O(1) per
  - need incident edges → choose adjacency list implementation for graph
- add incident edges to queue
  - → O(m log n) total
    - O(log m) to add to queue; each edge is added at most once
- mark as visited / check status
  - → O(1) per

→ total: O(m log n)

- using heap implementation of priority queue

Prim's Algorithm

Can we do better?

- O(m log n) isn't an improvement over O((n+m) log n) or O(m log n) for Kruskal's algorithm

The running time is dominated by the queue operations.

More efficient insert and remove isn't that feasible (we need both), but what about doing fewer operations?

- many of the edges in the priority queue aren't useful because they connect within S
- alternative: store the vertices in V-S in the priority queue instead of edges, ordered by the cost of the cheapest edge between the vertex and a vertex of S
  - the idea is to maintain a collection of what could be the next vertex added to the spanning tree, along with the cheapest cost of connecting that vertex

---

Prim's Algorithm

```
algorithm prim(G,w)
 input: connected undirected
  graph G with edge weights w
 output: MST defined by the
  'prev' labels

for all u in V
  cost[u] ← ∞
  prev[u] ← null
s ← a vertex of G
cost[s] ← 0

PQ ← makeQueue(V)
while PQ is not empty
  v ← PQ.removeMin()
  for each edge (v,z) in E
    if cost[z] > w(v,z) then
      cost[z] = w(v,z)
      prev[z] = (v,z)
      PQ.decreaseKey(z)
```

For each vertex in V-S, keep track of the cheapest known edge connecting it to S.

- prev(v) = the cheapest known edge connecting v to S
- cost(v) = weight of edge prev(v)

“Known” edges are those incident on vertices in S.

- the information is complete for any vertex in V-S connected to one in S
- update prev/cost information when we add a vertex to S

Prim's Algorithm

Running time?

- same structure as Dijkstra's algorithm, same running time
  - O((n+m) log n) for a heap-based priority queue
  - can do better with a fancier PQ implementation – O(n log n) for sparse, O(n^2) for dense

MST

Prim's or Kruskal's?

- can achieve better running time with Prim's algorithm and a fancy PQ implementation
- (standard) PQ is a more common data structure than union-find (or a fancy PQ)
- need to repeat Prim's on each connected component if the graph is not connected
  - Kruskal's handles disconnected graphs without anything additional

---

Takeaways

- definitions: spanning tree, minimum spanning tree
- algorithms for MST – kruskal's, prim's
  - what the algorithm is – be able to trace
  - running time and pros/cons of each algorithm
- union-find data structure
  - operations – makeset, find, union
  - union-by-rank list implementation – what it is, running time
  - union-by-rank tree implementation – running time
  - as an example of an incremental approach to data structure development

---

Recap

- graph algorithms
  - BFS-based algorithms – reachability, connected components, unweighted shortest path, 2-coloring
  - DFS-based algorithms – reachability, cycle detection, cut vertices, cut edges, strongly connected components, topological sort
  - shortest weighted paths – Dijkstra's algorithm, Bellman-Ford, Floyd-Warshall (all pairs shortest path)
  - MST – Kruskal's and Prim's algorithms
  - max flow, min flow, …
- new data structure
  - union-find
- a surprising insight
  - sometimes the simple solutions are better (or at least not worse)
- and a less-surprising observation
  - the best implementation depends on the situation

Recap

- tactics for designing efficient data structures
  - basic steps: e.g. store elements, store structural relationships
  - adapt known structures with good properties
  - store extra information for efficiency
  - balance storage with computation
  - don't waste work
  - piggyback computation on top of access operations as long as big-Oh is not affected