Data Structures in Java

Session 18
Instructor: Bert Huang
http://www.cs.columbia.edu/~bert/courses/3134
Announcements

- Homework 5 posted, due 11/24
- Old homeworks, midterm exams
Review

- Shortest Path algorithms
- Breadth first search
- Dijkstra’s Algorithm
- All-Pairs Shortest Path
Today’s Plan

• Minimum Spanning Tree
• Prim’s Algorithm
• Kruskal’s Algorithm
• Disjoint Sets
Minimum Spanning Tree

Problem Definition

- Given connected graph $G$, find the connected, acyclic subgraph $T$ with minimum edge weight
- A tree that includes every node is called a spanning tree
- The method to find the MST is another example of a greedy algorithm
Motivation for Greed

- Consider any spanning tree
- Adding another edge to the tree creates exactly one cycle
- Removing an edge from that cycle restores the tree structure
Prim’s Algorithm

- Grow the tree like Dijkstra’s Algorithm
- Dijkstra’s: grow the set of vertices to which we know the shortest path
- Prim’s: grow the set of vertices we have added to the minimum tree
- Store shortest edge $D[ ]$ from each node to tree
Prim’s Algorithm

• Start with a single node tree, set distance of adjacent nodes to edge weights, infinite elsewhere

• Repeat until all nodes are in tree:
  • Add the node \( v \) with shortest known distance
  • Update distances of adjacent nodes \( w \):
    \[ D[w] = \min(D[w], \text{weight}(v, w)) \]
Prim’s Example
Prim’s Example

Diagram:
- Nodes: 1, 2, 3, 4, 5, 6
- Edges with weights: (1, 3: 4), (2, 4: 5), (2, 7: 5), (3, 6: 6), (4, 5: 9)
- Starting node: 1

Weights:
- Node 1: 0
- Node 2: ∞
- Node 3: ∞
- Node 4: ∞
- Node 5: ∞
- Node 6: ∞
Prim’s Example

Diagram: A graph with nodes labeled 9, 3, 4, and ∞, connected by edges labeled 3, 7, 5, 6.
Prim’s Example
Prim’s Example
Prim’s Example
Prim's Example

Graph with vertices and edges labeled with weights.
Prim’s Example
Prim’s Example
Prim’s Example
Prim’s Example
Prim’s Example

Diagram of a graph with labeled edges.
Implementation Details

- Store “previous node” like Dijkstra’s Algorithm; backtrack to construct tree after completion
- Of course, use a priority queue to keep track of edge weights. Either
  - keep track of nodes inside heap & decreaseKey
  - or just add a new copy of the node when key decreases, and call deleteMin until you see a node not in the tree
Prim’s Algorithm Justification

- At any point, we can consider the set of nodes in the tree $T$ and the set outside the tree $Q$.
- Whatever the MST structure of the nodes in $Q$, at least one edge must connect the MSTs of $T$ and $Q$.
- The greedy edge is just as good structurally as any other edge, and has minimum weight.
Prim’s Running Time

- Each stage requires one deleteMin $O(\log |V|)$, and there are exactly $|V|$ stages.
- We update keys for each edge, updating the key costs $O(\log |V|)$ (either an insert or a decreaseKey).
- Total time:
  $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$
Kruskal’s Algorithm

- Somewhat simpler conceptually, but more challenging to implement
- Algorithm: repeatedly add the shortest edge that does not cause a cycle until no such edges exist
- Each added edge performs a union on two trees; perform unions until there is only one tree
- Need special ADT for unions (Disjoint Set)
Kruskal’s Example
Kruskal’s Example
Kruskal’s Example
Kruskal’s Example
Kruskal’s Example
Kruskal’s Example
Kruskal’s Example
Kruskal’s Justification

- At each stage, the greedy edge $e$ connects two nodes $v$ and $w$
- Eventually those two nodes must be connected;
  - we must add an edge to connect trees including $v$ and $w$
- We can always use $e$ to connect $v$ and $w$, which must have less weight since it's the greedy choice
Kruskal’s Running Time

- First, buildHeap costs $O(|E|)$
- In the worst case, we have to call $|E|$ deleteMins $|E| \leq |V|^2$
- Total running time $O(|E| \log |V|)$; but

\[ O(|E| \log |V|^2) = O(2|E| \log |V|) = O(|E| \log |V|) \]
MST Summary

- Connect all nodes in graph using minimum weight tree
- Two greedy algorithms:
  - Prim’s: similar to Dijkstra’s. Easier to code
  - Kruskal’s: easy on paper
Disjoint Sets
Motivating Example

- One interpretation of Kruskal’s Algorithm:
  - Think of trees as sets of connected nodes
  - Merge sets by connecting nodes
  - Never merge nodes that are in the same set
- Simple idea, but how can we implement it?
Equivalence Relations

• An equivalence relation is a relation operator that observes three properties:
  • **Reflexive**: \((a \ R \ a)\), for all \(a\)
  • **Symmetric**: \((a \ R \ b)\) if and only if \((b \ R \ a)\)
  • **Transitive**: \((a \ R \ b)\) and \((b \ R \ c)\) implies \((a \ R \ c)\)
  • Put another way, equivalence relations check if operands are in the same **equivalence class**
Equivalence Classes

- Equivalence class: the set of elements that are all related to each other via an equivalence relation
- Due to transitivity, each member can only be a member of one equivalence class
- Thus, equivalence classes are disjoint sets
- Choose any distinct sets S and T, \( S \cap T = \emptyset \)
Disjoint Set ADT

- Collection of objects, each in an equivalence class
- **find**(x) returns the class of the object
- **union**(x, y) puts x and y in the same class
  - as well as every other relative of x and y
- Even less information than hash; no keys, no ordering
One simple implementation would be to store the class label for each element in an array

- $O(1)$ lookup for **find**, $O(N)$ for **union**

- If we store equivalent elements in linked lists, we avoid scanning the whole set during **union**

- We can change the labels of the smaller class
Data Structure

- Store elements in equivalence (general) trees
- Use the tree’s root as equivalence class label
- `find` returns root of containing tree
- `union` merges tree
- Since all operations only search up the tree, we can store in an array
Implementation

- Index all objects from 0 to N-1
- Store a parent array such that \( s[i] \) is the index of i’s parent
- If i is a root, store the negative size of its tree*
- find follows \( s[i] \) until negative, returns index
- union(x,y) points the root of x’s tree to the root of y’s tree
Analysis

- **find** costs the depth of the node
- **union** costs $O(1)$ after **finding** the roots
- Both operations depend on the height of the tree
- Since these are general trees, the trees can be arbitrarily shallow
Union by Size

• Claim: if we union by pointing the smaller tree to the larger tree’s root, the height is at most $\log N$

• Each union increases the depths of nodes in the smaller trees

• Also puts nodes from the smaller tree into a tree at least twice the size

• We can only double the size $\log N$ times
Union by Size Figure

3 b
a  c

2 e
d

3 b
a  c  e

d
Union by Height

- Similar method, attach the tree with less height to the taller tree
- Overall height only increases if trees are equal height
Union by Height

Figure 1
Union by Height proof

- Induction: tree of height $h$ has at least $2^h$ nodes
- Let $T$ be tree of height $h$ with least nodes possible via union operations
- At last union, $T$ must have had height $h-1$, because otherwise, it would have been a smaller tree of height $h$
- Since the height was updated, $T$ unioned with another tree of height $h-1$, each had at least $2^{h-1}$ nodes resulting in at least $2^h$ nodes for $T$
Path Compression

- Even if we have log N tall trees, we can keep calling `find` on the deepest node repeatedly, costing $O(M \log N)$ for $M$ operations.
- Additionally, we will perform **path compression** during each `find` call.
- Point every node along the find path to root.
Path Compression

Figure
Union by Rank

- Path compression messes up union-by-height because we reduce the height when we compress.
- We could fix the height, but this turns out to gain little, and costs `find` operations more.
- Instead, rename to `union by rank`, where `rank` is just an overestimate of height.
- Since heights change less often than sizes, rank/height is usually the cheaper choice.
Worst Case Bound

- The algorithms described have been proven to have worst case $\Theta(M\alpha(M, N))$
  where $\alpha$ is the inverse of Ackermann’s function:

  - $A(1, j) = 2^j$
  - $A(i, 1) = A(i - 1, 2)$
  - $A(i, j) = A(i - 1, A(i, j - 1))$

  - $\alpha(M, N) = \min\{i \geq 1 | A(i, \lfloor M/N \rfloor) > \log N\}$
Worst Case Bound

- A slightly looser, but easier to prove/understand bound is that any sequence of operations will cost $O(M \log^* N)$ running time.

- $\log^* N$ is the number of times the logarithm needs to be applied to $N$ until the result is $\leq 1$.

- e.g., $\log^*(65536) = 4$ because $\log(\log(\log(\log(\log(65536)))))) = 1$.
Log* Plots
Log* Steps

| $\log^* N = 1$ | (1, 2] |
| $\log^* N = 2$ | (2, 4] |
| $\log^* N = 3$ | (4, 16] |
| $\log^* N = 4$ | (16, 65536] |
| $\log^* N = 5$ | (65536, $2^{65536}$] |
Note about Kruskal’s

- With this bound, Kruskal’s algorithm needs $N-1$ unions, so it should cost almost linear time to perform unions.
- Unfortunately the algorithm is still dominated by heap deleteMin calls, so asymptotic running time is still $O(E \log V)$. 
Reading

- Weiss 9.5 (MST)
- Weiss 8.1-8.5 (Disjoint Sets)