Overview

- Spanning trees
- Minimum spanning tree (MST)
  - Prim's algorithm
  - Kruskal's algorithm
  - Union-Find

Spanning Tree

- Set of edges connecting all nodes in graph
  - need N-1 edges for N nodes
  - no cycles, can be thought of as a tree
- Can build tree during traversal

Spanning Tree Construction

- Recursive algorithm
  
  Known = { start }
  explore ( start );

  void explore (Node X) {
    for each successor Y of X
      if (Y is not in Known)
        Parent[Y] = X
        Add Y to Known
        explore(Y)
  }

- Iterative algorithm
  
  Known = { start }
  Discovered = { start }
  while ( Discovered ≠ ∅ ) {
    take node X out of Discovered
    for each successor Y of X
      if (Y is not in Known)
        Parent[Y] = X
        Add Y to Discovered
        Add Y to Known
  }

Breadth & Depth First Spanning Trees

Breadth-first

Depth-first
Depth-First Spanning Tree Example

Breadth-First Spanning Tree Example

Spanning Tree Construction

- Many spanning trees possible
  - Different breadth-first traversals
  - Nodes same distance visited in different order
  - Different depth-first traversals
  - Neighbors of node visited in different order
  - Different traversals yield different spanning trees

Minimum Spanning Tree (MST)

- Spanning tree with minimum total edge weight

Minimum Spanning Tree (MST)

- Possible to have multiple MSTs
  - Different spanning trees with same weight

- Example applications
  - Minimize length of telephone lines for neighborhood
  - Minimize distance of airplane routes serving cities

Algorithms for Finding MST

- Three well known algorithms
  1. Borůvka’s algorithm [1926]
     - For constructing efficient electricity network
     - Rediscovered by Sollin in 1960s
  2. Prim’s algorithm [1957]
     - First discovered by Vojtěch Jarník in 1930
     - Similar to Dijkstra’s algorithm
  3. Kruskal’s algorithm [1956]
     - By Prof. Clyde Kruskal’s uncle
Algorithms for Finding MST

1. **Borůvka’s algorithm**
   - Add vertices to MST in parallel

2. **Prim’s algorithm**
   - Add vertices to MST
   - One at a time
   - Closest vertex first

3. **Kruskal’s algorithm**
   - Add edges to MST
   - One at a time
   - Lightest edge first

Shortest Path – Dijkstra’s Algorithm

S = ∅
P[ ] = none for all nodes
C[start] = 0, C[ ] = ∞ for all other nodes
while ( not all nodes in S )
  find node K not in S with smallest C[K]
  add K to S
  for each node J not in S adjacent to K
    if ( C[K] + cost of (K,J) < C[J] )
      C[J] = C[K] + cost of (K,J)
      P[J] = K
Optimal solution computed with greedy algorithm

MST – Prim’s Algorithm

S = ∅
P[ ] = none for all nodes
C[start] = 0, C[ ] = ∞ for all other nodes
while ( not all nodes in S )
  find node K not in S with smallest C[K]
  add K to S
  for each node J not in S adjacent to K
    if ( /* C[K] + */ cost of (K,J) < C[J] )
      C[J] = /* C[K] + */ cost of (K,J)
      P[J] = K
Keeps track of vertex w/ minimal distance to current tree
Optimal solution computed with greedy algorithm

MST – Kruskal’s Algorithm

sort edges by weight (from least to most)
tree = ∅
for each edge (X,Y) in order
  if it does not create a cycle
    add (X,Y) to tree
  stop when tree has N–1 edges
Keeps track of
  lightest edge remaining
  whether adding edge to MST creates cycle
Optimal solution computed with greedy algorithm

MST – Kruskal’s Algorithm Example

When does adding (X,Y) to tree create cycle?

Two approaches to finding cycles
1. Traversal
2. Connected subgraph
MST – Kruskal’s Algorithm

Traversal approach
- Traverse tree starting at X
- If we can reach Y, adding (X,Y) would create cycle

Example
- Question
  - Add (X,Y) to MST?
- Answer
  - No, since can already reach Y from X by traversing MST

MST – Connected Subgraph Approach

- Maintain set of nodes for each connected subgraph
- Initialize one connected subgraph for each node
- If X, Y in same set, adding (X,Y) would create cycle
- Otherwise
  1. Add edge (X,Y) to spanning tree
  2. Merge sets containing X, Y

To test set membership
- Use Union-Find algorithm

Union-Find Algorithm

- Algorithm & data structure
  - Very efficient for testing membership in disjoint sets

Problem description
- Start with n nodes, each in different subgraph
- Support two operations
  - Find – are nodes x & y in same subgraph?
  - Union – merge subgraphs containing x & y

Basic approach
- Each node has a parent pointer
- Find – follow parent pointer(s) to root of tree
- Union – point root of 1st tree to root of 2nd tree

Example
- Union( a, b ); union( c, d); union( b, d)
**Union-Find Algorithm**

- **Path compression**
  - Speeds up future Find( ) operations
  1. Follow parent pointer(s) to root of tree
  2. Update all nodes along path to point to root

- **Example**
  - Find(d)

```
        a
       /|
      /  |
     b   c
     / \
    d   d
```

So how fast is Union-Find?

**Ackermann’s Function**

- **Function**
  - \( A( ) \) grows fast

```c
int A(x, y) {
    if (x == 0)
        return y + 1;
    if (y == 0)
        return A(x - 1, 1);
    return A(x - 1, A(x, y - 1));
}
```

- \( A(2, 2) = 7 \)
- \( A(3, 3) = 61 \)
- \( A(4, 2) = 2^{65536} - 3 \)
- \( A(4, 3) = 2^{65536} - 3 \)
- \( A(4, 4) = 2^{65536} - 3 \)

**Inverse Ackermann’s Function**

- **Definition**
  - \( \alpha(n) \) is the inverse Ackermann’s function
  - \( \alpha(n) = \) the smallest \( k \) such that \( A(k, k) \geq n \)

- **Fun fact**
  - \( \alpha(\text{number of atoms in universe}) = 4 \)

- **Union-find**
  - A sequence of \( n \) operations requires \( O(n \alpha(n)) \) time
  - Practically speaking, indistinguishable from \( O(n) \)

**Graph Summary**

- **Graph data structure**
  - Very useful in practice
  - Different representations

- **Many graph algorithms**
  - Traversal
  - Shortest path
  - Minimum spanning tree

**Algorithms / Data Structures**

- **Introduction to data structures in 132**
  - Lists, Trees, Graphs, Sets / Maps

- **Much more to learn in future courses**
  - 351 – Introduction to Algorithms
    - Dynamic programming, recurrences, reductions, NP-completeness…
  - 420 – Data Structures
    - Balanced trees, quadtrees, k-d trees…
  - 451 – Design and Analysis of Computer Algorithms
    - Correctness proofs, analyzing complexity…