L25: Minimum Spanning Trees (MST) and Disjoint Sets

Alex Steiger
CompSci 201: Spring 2024
4/15/2024
Logistics, coming up

• This Wednesday, 4/17
  • Midterm exam 3
  • APT 9 (last APTs) - extended to Thursday 4/18

• This Friday, 4/19
  • Semester / Final review in discussion

• Next Monday, 4/22
  • Project P6: Route (last project) due

• Tuesday after next, 4/30
  • Final exam, 9 am
Exploring a node with Dijkstra’s Algorithm, Pseudocode

While unexplored nodes remain
• Explore current = the closest unexplored node
• For each neighbor:
  • Update shortest path to neighbor if shorter to go through current

Just like BFS (explore closer nodes first) except... now we need to account for weights.
Practical Dijkstra Initialization

Add vertices to the queue once they are actually reached/visited.

Don’t need to add anything for all nodes yet.

```java
public Map<Character, Integer> stdDijkstra(char start, Map<Character, List<Character>> aList) {
    Map<Character, Integer> distance = new HashMap<>();
    distance.put(start, 0);
    Comparator<Character> comp = (a, b) -> distance.get(a) - distance.get(b);
    PriorityQueue<Character> toExplore = new PriorityQueue<>(comp);
    toExplore.add(start);
}
```
Practical Dijkstra search loop

Keep searching while there are unexplored nodes.

Choose to explore from the next closest (to start) unexplored node to start at each iteration.

```java
while (toExplore.size() > 0) {
    char current = toExplore.remove();
    int currDist = distance.get(current);
    for (char neighbor : alist.get(current)) {
        // ...}
    }
}
return distance;
```

Search all neighbors of current. If you find a shorter path to neighbor through current, update to reflect that.
Details: Checking each neighbor

All neighbors of current node

Distance to neighbor through current = distance to current + weight on edge from current to neighbor

```java
for (char neighbor : aList.get(current)) {
    int newDist = currDist + getWeight(current, neighbor);
    if (!distance.containsKey(neighbor)) {
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    } else if (newDist < distance.get(neighbor)) {
        // implement decreasePriority by removal and re-insertion
        toExplore.remove(neighbor);
        distance.put(neighbor, newDist);
        toExplore.add(neighbor);
    }
}
```

If neighbor newly discovered:
- Record new distance
- Add to priority queue

If neighbor already discovered, update:
- Remove from PQ
- Record new distance
- Add back to PQ
Implementing `decreasePriority`

- Most standard library binary heaps (including `java.util`) don’t support an efficient update/decrease priority operation.

```java
else if (newDist < distance.get(neighbor)) {
    // implement decreasePriority by removal and re-insertion
    toExplore.remove(neighbor);
    distance.put(neighbor, newDist);
    toExplore.add(neighbor);
}
```

- Our code works, but is $O(N)$ time
  - Java’s PQ takes $O(N)$ to remove *given* node ($O(1)$ for smallest)
  - Other PQ implementations support $O(\log N)$-time `decreasePriority`, but they are not in Java library
Is Dijkstra’s algorithm guaranteed to be correct? (Informal)

• **Claim.** Distance is correct shortest path distance for all nodes *explored* so far, and shortest path distance *through explored nodes* for all others.

• Formal proof is *by induction*, see CompSci 230.
  • Assume the property is true up to some point in the algorithm, then...
  • Consider the next node we explore:
Is Dijkstra’s algorithm guaranteed to be correct? (Informal)

The shortest path distance so far goes through explored nodes.

Suppose we explore from C this iteration.

The shortest path goes through explored nodes $W(A,C)$.

Can’t be another shorter path through an unexplored node! There would be a node that would be explored/removed from the PQ before C.
Runtime Complexity of Dijkstra’s Algorithm (with $N$ nodes, $M$ edges) assuming $O(\log N)$ decreasePriority

Like BFS, consider each node once and each edge twice, takes $O(\log N)$ time for each: $O((N+M)\log(N))$
Minimum Spanning Tree (MST) and Greedy Graph Algorithms
Minimum Spanning Tree (MST) Problem

- Given N nodes and M edges, each with a weight/cost...
- Find a set of edges that connect all the nodes with minimum total cost (will be a tree)

Weighted undirected graph with:
- Edges labeled with weights/costs
- Minimum spanning tree highlighted
Motivating/Applying Minimum Spanning Tree

• Create a connected cable/data network with the least cable/cost/energy possible.

• City planning: Connect several metro stops with least tunneling

• Image Segmentation

• Clustering

https://slideplayer.com/slide/11413693/
You are given an array `points` representing integer coordinates of some points on a 2D-plane, where `points[i] = [x_i, y_i]`.

The cost of connecting two points `[x_i, y_i]` and `[x_j, y_j]` is the **manhattan distance** between them: `|x_i - x_j| + |y_i - y_j|`, where `|val|` denotes the absolute value of `val`.

Return the minimum cost to make all points connected. All points are connected if there is exactly one simple path between any two points.
Intuitive Inductive Reasoning

• Suppose we have the MST on N-1 vertices.

• We consider the next vertex to get the MST on N vertices.
  • Must use the cost 2 or the cost 5 edge regardless of the rest of the MST
  • Might as well use the cheaper cost 2 edge
Greedy Optimization: Prim’s Algorithm

• Initialize?
  • Choose an arbitrary vertex

• Partial solution?
  • MST connecting *subset* of the vertices.

• Greedy step?
  • Choose the cheapest / least weight edge that connects a new vertex to the partial solution.
Visualizing Prim’s Algorithm

In the visualization:

• Edges between all pairs of vertices
• Weights are implicit by distances
• Algorithm greedily grows by choosing closest unconnected vertex

By Shiyu Ji - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=54420894
More Intuitive Inductive Reasoning

• Suppose we have chosen some spanning trees so far.

• Must connect all of them, might as well choose the **cheapest** edge connecting two trees.

![Graph with spanning trees and edges labeled with weights.](https://commons.wikimedia.org/w/index.php?curid=644030)
Greedy Optimization Again: Kruskal’s Algorithm

• Initialize?
  • All nodes in \textit{disjoint sets}

• Partial solution?
  • Forest of spanning trees in disjoint sets

• Greedy step?
  • Choose the cheapest / least weight edge that connects two disjoint sets / trees, connect them.
Visualizing Kruskal’s Algorithm

In the visualization:

• Edges between all pairs of vertices
• Weights are implicit by distances
• Algorithm greedily grows by cheapest edge that connects disjoint sets/trees.

By Shiyu Ji - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=54420894
Kruskal’s Algorithm in *Pseudocode*

Input: N node, M edges, M edge weights

• Initialize MST as empty set

• Let S be a collection of N *disjoint sets*, one per node

• While S has more than 1 set:
  • Let \((u, v)\) be the minimum cost remaining edge
  • *Find* which sets \(u\) and \(v\) are in. If different sets:
    • *Union* the sets together
    • Add \((u, v)\) to MST

• Return MST
Kruskal’s Algorithm Runtime?

Input: N node, M edges, M edge weights
- Initialize MST as empty set
- Let S be a collection of N disjoint sets, one per node
- While S has more than 1 set:
  - Let (u, v) be the minimum cost remaining edge
  - Find which sets u and v are in. If different sets:
    - Union the sets together
    - Add (u, v) to MST
- Return MST

Overall: $O(M(\log(M)+C))$ where $C$ is time for Union/Find
Disjoint Sets and Union-Find

DIYDisjointSets implementation viewable here: coursework.cs.duke.edu/cs-201-spring-24/diydisjointsets
Union-Find Data Structure

- AKA Disjoint-Set Data Structure
- Start with $N$ distinct (disjoint) sets
  - consider them labeled by integers: 0, 1, ...
- **Union** two sets: create set containing both
  - label with one of the numbers
- **Find** the set containing a number
  - Initially self, but changes after unions
Disjoint-Set Forest Implementation

- Each set will be represented by a parent “tree”: Instead of child pointers, nodes have a parent “pointer”.
- Everything starts as its own tree: a single node

<table>
<thead>
<tr>
<th>parent</th>
<th>parent</th>
<th>parent</th>
<th>parent</th>
<th>parent</th>
<th>parent</th>
<th>parent</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
Disjoint-Set Forest Union

- Union(7,8)
Disjoint-Set Forest Union

- **Union(3,4)**
Disjoint-Set Forest Union

- Union(3,8)
- parent[8] is not the root anymore—Need to find its root first
  - Use Find(8) operation
Disjoint-Set Forest Find

• Find(8):
  • Find root of tree containing 8.
  • Follow parent pointers starting at parent[8]
• In this example, parent[7]

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Disjoint-Set Forest Find

- Back to $\text{Union}(3,8)$
  - Set root of parent[8], which is $\text{Find}(8) = \text{parent}[7]$, to root parent[3]

```plaintext
parent

<table>
<thead>
<tr>
<th>itemID</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
</table>
```

**Diagram:**
- Node 3 is the root of the parent[8] because $\text{Find}(8) = \text{parent}[7] = 7$.

Find(8)
Disjoint-Set Forest Array Representation

- The “nodes” and “pointers” are just conceptual – can represent with a simple array, like binary heap.
- Parent array just stores what the itemID node points to.

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Disjoint-Set Forest Find

```java
18     public int find(int id) {
19         while (id != parent[id]) {
20             id = parent[id];
21         }
22         return id;
23     }
```

root is just when `parent[i] = i`

Else go to next “node up”

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Disjoint-Set Forest Union Revisited

```java
25    public void union(int set1, int set2) {
26        int root1 = find(set1);
27        int root2 = find(set2);
28        parent[root2] = root1;
```

Make one "point to" other

roots from initial set1 and initial set2 "nodes"

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>3</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Worst-Case Runtime Complexity?

```
25    public void union(int set1, int set2) {
26        int root1 = find(set1);
27        int root2 = find(set2);
28        parent[root2] = root1;
```

What if we...  
union(7,8)  
union(6,7)  
union(5,6)  
...  
union(0,1)

Now `find(8)` would have linear runtime complexity!!

<table>
<thead>
<tr>
<th>parent</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>itemID</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Optimization 1: Union by Size

Be careful in how you union. Always make the “root” for the set with fewer elements point to the “root” for the set with more elements.

Sufficient for worst case logarithmic efficiency.
Claim. Each element to root path has length at most $O(\log(N))$ with union by size optimization.

Proof.
- Consider an element $a$, initially a set of size 1.
- Each time the path length increases, the size of the set must at least double.
- Can happen at most $O(\log(N))$ times with $N$ initial sets.
Optimization 1: Union by Size

```java
public void union(int set1, int set2) {
    int root1 = find(set1);
    int root2 = find(set2);
    if (root1 == root2) { return; }
    if (setSizes[root1] < setSizes[root2]) {
        parent[root1] = root2;
        setSizes[root2] += setSizes[root1];
    } else {
        parent[root2] = root1;
        setSizes[root1] += setSizes[root2];
    }
    size--;}
```
Lazy Path Compression

- Lazy path compression: When ever you traverse a path in `find`, connect all the pointers to the top.

- Sufficient for amortized logarithmic runtime complexity for union/find operations.

```
find(5)  
```

```
5  6  7  8  
```

```
5  
```

```
6  7  8  
```
Disjoint Set Forest Path Compression

```java
public int find(int id) {
    int idCopy = id;
    while (id != parent[id]) {
        id = parent[id];
    }
    int root = id;
    id = idCopy;
    while (id != parent[id]) {
        parent[idCopy] = root;
        id = parent[id];
        idCopy = id;
    }
    return id;
}
```
Optimized Runtime Complexity

• Optimizations considered separately:
  • Union by size: Worst case logarithmic
  • Path compression: Amortized logarithmic

• Considered together...?
  • Worst case logarithmic, and amortized inverse Ackermann function $a(n)$.
  • $a(n) < 5$ for $n < 2^{2^{2^{16}}} = 2^{2^{65536}}$
  • Number of atoms in observable universe only $\sim 10^{80}$
  • Practically constant for any $n$ you can write down
Remember Kruskal’s Algorithm Runtime?

Input: N node, M edges, M edge weights

• Let MST to an empty set
• Let S be a collection of N disjoint sets, one per node
• While S has more than 1 set:
  • Let (u, v) be the minimum cost remaining edge
  • Find which sets u and v are in. If different sets:
    • Union the sets
    • Add (u, v) to MST
• Return MST

• Remove from binary heap, O(log(M))

O(M(log(M)+C)) because C < log(M) for our optimized union find

Looping over (worst case) all M edges