Admin

Assignment 8 out: don’t reinvent the wheel

Assignment schedule updated for the rest of the semester

Groups optional this week

MORE GRAPH ALGORITHMS

David Kauchak

CS 140 — Spring 2024

1

2

Connectedness

Given an undirected graph, for every node \( u \in V \), can we reach all other nodes in the graph?

Algorithm + running time

Run BFS or DFS: visit (one pass) and mark nodes as we visit them. If we visit all nodes, return true. Otherwise false.

Running time: \( O(|V| + |E|) \)

3

Strongly connected

Given a directed graph, can we reach any node \( v \) from any other node \( u \)?

Can we do the same thing?

4
Transpose of a graph

Given a graph G, we can calculate the transpose of a graph G^T by reversing the direction of all the edges.

Running time to calculate G^T? \( \Theta(|V| + |E|) \)

Strongly connected

Strongly-Connected(G):
- Run DFS-Visit or BFS from some node u
- If not all nodes are visited: return false
- Create graph G^T
- Run DFS-Visit or BFS on G^T from node u
- If not all nodes are visited: return false
- return true

Is it correct?

What do we know after the first pass?
- Starting at u, we can reach every node.

What do we know after the second pass?
- All nodes can reach u.
  - We can get from u to every node is G, therefore, if we reverse the edges (i.e. G), then we have a path from every node to u.

Which means that any node can reach any other node. Given any two nodes s and t we can create a path through u.

Runtime?

Strongly-Connected(G):
- Run DFS-Visit or BFS from some node u
- If not all nodes are visited: return false
- Create graph G^T
- Run DFS-Visit or BFS on G^T from node u
- If not all nodes are visited: return false
- return true

\( O(|V| + |E|) \)
Minimum spanning trees

What are they?
What do you remember about them?
What algorithms do you remember?

Minimum spanning trees

What is the lowest weight set of edges that connects all vertices of an undirected graph with positive weights.

Input: An undirected, positive weight graph, G=(V,E)
Output: A tree T=(V,E') where E' ⊆ E that minimizes

\[ \text{weight}(T) = \sum_{e \in E'} w_e \]

MST example

Can an MST have a cycle?

MSTs
MSTs

Can an MST have a cycle?

Applications?

Connectivity
- Networks (e.g., communications)
- Circuit design/wiring
- Hub/spoke models (e.g., flights, transportation)

Traveling salesman problem?

Algorithm ideas?

Cuts

A cut is a partitioning of the vertices into two sets $S$ and $V-S$.

An edge "crosses" the cut if it connects a vertex $v \in V$ and $v \not\in V-S$.
Minimum cut property

Given a partition \( S \), let edge \( e \) be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge \( e \).

**Proof by contradiction?**

Consider an MST with edge \( e' \) that is not the minimum edge.

Using \( e \) instead of \( e' \) still connects the graph, but produces a tree with smaller weights.

Minimum cut property

If the minimum cost edge that crosses the partition is not unique, then some minimum spanning tree contains edge \( e \).
Kruskal’s algorithm

Given a partition $S$, let edge $e$ be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge $e$.

Kruskal’s algorithm

1. For all $v \in V$
2. Make-Set($v$)
3. $S \leftarrow \emptyset$
4. Sort the edges of $E$ by weight
5. For all edges $(u, v) \in E$ in increasing order of weight
6. If Find-Set(u) $\neq$ Find-Set(v)
7. Add edge to $S$
8. Union(Find-Set(u), Find-Set(v))

Add smallest edge that connects two sets not already connected

MST

G
Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

A 1
C

B
D

F

MST

G

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

A 1
C

B
D

F

MST

G

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

A 1
C

B
D

F

MST

G

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

A 1
C

B
D

F

MST

G
Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected
Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

Kruskal’s algorithm
Add smallest edge that connects two sets not already connected

Correctness of Kruskal’s
Never adds an edge that connects already connected vertices
Always adds lowest cost edge to connect two sets. By min cut property, that edge must be part of the MST

Running time of Kruskal’s

Kruskal(G)
1 for all v ∈ V
2    MAKESET(v)
3    F ← ∅
4 sort the edges of E by weight
5 for all edges (u, v) ∈ E in increasing order of weight
6     if FIND-SSET(u) ≠ FIND-SSET(v)
7         add edge (u, v)
8     UNION(FIND-SSET(u), FIND-SSET(v))

3/27/24
Running time of Kruskal’s

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>E</td>
<td>V</td>
<td>E</td>
</tr>
</tbody>
</table>

- \( V \) calls to MakeSet
- \( O(\log E) \) calls to FindSet
- \( 2E \) calls to FindSet
- \( V \) calls to Union

Disjoint set data structures

Represents a collection of one or more sets

Operations:
- MakeSet: Add a new value to the collection and make the value its own set
- FindSet: Given a value, return the set the value is in
- Union: Merge two sets into a single set

Disjoint set data structure

MakeSet(A), MakeSet(B), MakeSet(C), MakeSet(D), MakeSet(E)

Disjoint set data structure

FindSet(A)

Disjoint Set

\[ A \]  [B]  [C]  [D]  [E]
Disjoint set data structure

FindSet(A)

Union(FindSet(A), FindSet(E))

Union(FindSet(A), FindSet(E))

Union(FindSet(C), FindSet(D))
Disjoint set data structure

Union(FindSet(C), FindSet(D))

Disjoint Set

A B C

FindSet(D)?

Disjoint Set

A B C

FindSet(D)?

Disjoint Set

A B C

Union(FindSet(D), FindSet(B))

Disjoint Set

A B C

A B C
Disjoint set data structure

Union(FindSet(D), FindSet(B))

A

Disjoint Set

B

C

D

E

How would we implement it with a list of linked lists?
MakeSet?
FindSet?
Union?

A

Disjoint Set

B

C

E

Disjoint set: union

A

B

C

D
Disjoint set: union

[Diagram showing union of sets A, B, C, D]

Running time?

O(1)
Disjoint set: find-set

Disjoint set: find-set

Disjoint set: find-set

Disjoint set: find-set

Disjoint set data structure

<table>
<thead>
<tr>
<th></th>
<th>MakeSet (V calls)</th>
<th>FindSet (E calls)</th>
<th>Union (V calls)</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linked lists</td>
<td>O(V)</td>
<td>O(E)</td>
<td>O(V)</td>
<td>O(V)</td>
</tr>
<tr>
<td>Linked lists +</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>heuristics</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

O(n) = n = number of things in set

Search each linked list

Running time?

Running time of Kruskal’s

O(\log E)
Prim’s algorithm

Start at some root node and build out the MST by adding the lowest weighted edge at the frontier.

\[
\text{Prim}(v, V, E, w) = \begin{cases} 
1 & \text{for all } u \in V, \text{dist}(u, v) = \infty \\
2 & \text{dist}(v, v) = 0 \\
3 & \text{dist}(v, u) \leftarrow w(u, v) \\
4 & \text{dist}(v, u) \leftarrow \text{dist}(v, u) \\
5 & \text{dist}(v, u) = \text{dist}(v, u) \\
6 & \text{while } \exists u \in V \setminus V' \\
7 & \text{dist}(v, u) = \infty \\
8 & \text{for each edge } (u, v) \in E \\
9 & \text{if dist}(v, u) < \text{dist}(v, u) \\
10 & \text{dist}(v, u) \leftarrow \text{dist}(v, u) \\
11 & \text{dist}(v, u) \leftarrow \text{dist}(v, u) \\
12 & \text{dist}(v, u) = \text{dist}(v, u)
\end{cases}
\]

Prim’s algorithm

Start at some root node and build out the MST by adding the lowest weighted edge at the frontier.

\[
\text{Prim}(v, V, E, w) = \begin{cases} 
1 & \text{for all } u \in V, \text{dist}(u, v) = \infty \\
2 & \text{dist}(v, v) = 0 \\
3 & \text{dist}(v, u) \leftarrow w(u, v) \\
4 & \text{dist}(v, u) \leftarrow \text{dist}(v, u) \\
5 & \text{dist}(v, u) = \text{dist}(v, u) \\
6 & \text{while } \exists u \in V \setminus V' \\
7 & \text{dist}(v, u) = \infty \\
8 & \text{for each edge } (u, v) \in E \\
9 & \text{if dist}(v, u) < \text{dist}(v, u) \\
10 & \text{dist}(v, u) \leftarrow \text{dist}(v, u) \\
11 & \text{dist}(v, u) \leftarrow \text{dist}(v, u) \\
12 & \text{dist}(v, u) = \text{dist}(v, u)
\end{cases}
\]
```plaintext
0 while (E ≠ ∅)
  1 e = ExtractMin(E)
  2 visited[e] = true
  3 for each edge (u,v) ∈ E
  4 if visited[u] and visited[v] < ∞
  5 DecreaseKey(v, w(u,v))
  6 prev[v] = u
```
Correctness of Prim’s?

Can we use the min-cut property?
- Given a partition, let edge be the minimum cost edge that crosses the partition. Every minimum spanning tree contains edge e.

Let S be the set of vertices visited so far.

The only time we add a new edge is if it’s the lowest weight edge from S to V-S.

Running time of Prim’s

```
Prim(G, s)
1: s ← Extract-Min(D)
2: S ← {s}
3: dist(s) ← 0
4: for each u ∈ V - {s}
5:    dist(u) ← ∞, prev(u) ← nil
6: while (S ≠ V)
7:    u ← Extract-Min(D)
8:    for each v ∈ V - S
9:        if dist(v) > w(u, v)
10:           dist(v) ← w(u, v)
11:           prev(v) ← u
```

Running time of Prim’s

1. Make a heap $|V|$ calls to ExtractMin
2. $|E|$ calls to DecreaseKey

Total:
- **Array:** $O(|V|^2)$, $O(|E| log |V|)$, $O((|V|+|E|) log |V|)$
- **Binary heap:** $O(|V| log |V|)$, $O(|E|)$, $O(|E| log |V|)$
- **Fib heap:** $O(|V| log |V|)$, $O(|E|)$, $O(|E| log |V| + |E|)$
- **Kruskal’s:** $O(|E| log |E|)$