Admin

Assignment 8

No mentor hours Friday

Practice

Solution

\[ \text{Sum} = 8 + 8 + 9 + 9 + 11 + 11 + 12 + 14 = 82 \]
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 $E' \subseteq E$ that minimizes

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

MST example

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 $e$.

Prim's algorithm

Start at some root node and build out the MST by adding the lowest weighted edge out of the MST constructed so far.
Correctness of Prim's?

Can we use the min-cut property?

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

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(V, E)
1. for all $v \in V$
2. $h[v] \leftarrow \infty$
3. $p[v] \leftarrow \text{null}$
4. $\text{heap}() \leftarrow \emptyset$
5. $\text{heap}() \leftarrow \text{MakeHeap}(\text{heap}())$
6. $u \leftarrow \text{Extract-Min}(\text{heap}())$
7. $\text{visited}[u] \leftarrow \text{true}$
8. for each edge $(u, v) \in E$
9.  if $\text{visited}[v]$ is $\text{false}$
10.  $\text{heap}() \leftarrow \text{Decrease-Key}(\text{heap}(), v, u)$
11.  $\text{visited}[v] \leftarrow \text{true}$
12.  $p[v] \leftarrow u$

Running time of Prim’s

$O(V)\times E$
1. 1 call to MakeHeap
2. $O(V)$ calls to Extract-Min
3. $O(E)$ calls to Decrease-Key
Running time of Prim’s

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Bin heap</th>
<th>Fib heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeHeap</td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td>(</td>
<td>V</td>
<td>) ExtractMin</td>
<td>(O(</td>
</tr>
<tr>
<td>(</td>
<td>E</td>
<td>) DecreaseKey</td>
<td>(O(</td>
</tr>
<tr>
<td>Total</td>
<td>(O(</td>
<td>V</td>
<td>^2))</td>
</tr>
</tbody>
</table>

Kruskal’s: \(O(|E| \log |E|)\)

When should we use Kruskal’s or Prim’s?

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Bin heap</th>
<th>Fib heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>MakeHeap</td>
<td>(O(</td>
<td>V</td>
<td>))</td>
</tr>
<tr>
<td>(</td>
<td>V</td>
<td>) ExtractMin</td>
<td>(O(</td>
</tr>
<tr>
<td>(</td>
<td>E</td>
<td>) DecreaseKey</td>
<td>(O(</td>
</tr>
<tr>
<td>Total</td>
<td>(O(</td>
<td>V</td>
<td>^2))</td>
</tr>
</tbody>
</table>

Kruskal’s: \(O(|E| \log |E|)\)

Shortest paths

What is the shortest path from A to D?

Shortest paths

How can we find this?
Shortest paths

**BFS**

1. For each $x \in V$
   1. $d(x) = \infty$
   2. $x \leftarrow x$

2. While $Q \neq \emptyset$
   1. $v \leftarrow \text{Min}(Q)$
   2. $d(v) \leftarrow 0$
   3. $Q \leftarrow \emptyset$
   4. For each neighbor $u \in \text{adj}(v)$
      1. $d(u) \leftarrow d(v) + 1$
      2. $u \leftarrow u$

What is the shortest path from a to d?

![Graph](image)

Shortest path algorithms?

What is the shortest path from a to d?

![Graph](image)
Dijkstra’s algorithm

What is dist?
What is prev?

Dijkstra’s algorithm

1. For all v ∈ V, dist[v] = ∞
2. dist[s] = 0
3. prev[s] = null
4. Q = MaxHeapWithKey(dist)
5. While MaxHeapWithKey is not empty
6. a = ExtractMax(Q)
7. For all edges (a, b) ∈ E
8. if dist[b] > dist[a] + weight(a, b)
9. dist[b] = dist[a] + weight(a, b)
10. prev[b] = a

How does it work?

prev keeps track of the shortest path

How do we get the shortest path?
Dijkstra's algorithm

```
for all v \in V
    do dist[v] \leftarrow \infty
    do pred[v] \leftarrow nil

Q \leftarrow \text{Vexes}(V)

while Q != \emptyset
    do u \leftarrow \text{Extrm}Q

    for each v \in Adj[u]
        do if dist[u] + w(u,v) < dist[v]
            then dist[v] \leftarrow dist[u] + w(u,v)
                pred[v] \leftarrow u
```

25

Dijkstra's algorithm

```
for all v \in V
    do dist[v] \leftarrow \infty
    do pred[v] \leftarrow nil

Q \leftarrow \text{Vexes}(V)

while Q != \emptyset
    do u \leftarrow \text{Extrm}Q

    for each v \in Adj[u]
        do if dist[u] + w(u,v) < dist[v]
            then dist[v] \leftarrow dist[u] + w(u,v)
                pred[v] \leftarrow u
```

26

Dijkstra's algorithm

```
for all v \in V
    do dist[v] \leftarrow \infty
    do pred[v] \leftarrow nil

Q \leftarrow \text{Vexes}(V)

while Q != \emptyset
    do u \leftarrow \text{Extrm}Q

    for each v \in Adj[u]
        do if dist[u] + w(u,v) < dist[v]
            then dist[v] \leftarrow dist[u] + w(u,v)
                pred[v] \leftarrow u
```

27

Dijkstra's algorithm

```
for all v \in V
    do dist[v] \leftarrow \infty
    do pred[v] \leftarrow nil

Q \leftarrow \text{Vexes}(V)

while Q != \emptyset
    do u \leftarrow \text{Extrm}Q

    for each v \in Adj[u]
        do if dist[u] + w(u,v) < dist[v]
            then dist[v] \leftarrow dist[u] + w(u,v)
                pred[v] \leftarrow u
```

28
3/28/24

33

34

35

36
### Algorithm Description

1. **Graph Representation:**
   - **Nodes:** A, B, C, D, E
   - **Edges:** (A, B), (A, D), (B, C), (B, E), (D, E)

2. **Heap Initialization:**
   - Use a min-heap to store nodes.

3. **Algorithm Steps:**
   - **For each node:**
     - Initialize `dist` to infinity.
     - Initialize `parent` to null.

   - **While the heap is not empty:**
     - Extract the node with the smallest `dist` from the heap.
     - Update `dist` and `parent` for adjacent nodes.

4. **Complexity Analysis:**
   - **Time Complexity:** O((E + V) log V)
   - **Space Complexity:** O(V)

---

### Pseudocode

#### Prim's Algorithm

1. **Input:** Graph `G` with vertices `V` and edges `E`.

2. **Output:** Minimum spanning tree `MST`.

3. **Algorithm:**
   - **Initialize:**`dist` to infinity for all nodes, `parent` to null.
   - **Extract minimum:**
     - Find node `u` with minimum `dist`.
     - Update `dist` and `parent` for adjacent nodes.

4. **Repeat:**
   - **While heap is not empty:**
     - Extract node `u`.
     - Update `dist` and `parent` for adjacent nodes.

5. **Return:** Minimum spanning tree `MST`.

---

### Implementation

### Example

**Graph:**
- **Nodes:** A, B, C, D, E
- **Edges:** (A, B), (A, D), (B, C), (B, E), (D, E)

**Implementation:**
- Use a min-heap to store nodes.
- Initialize `dist` to infinity for all nodes, `parent` to null.
- **While heap is not empty:**
  - Extract node `u`.
  - Update `dist` and `parent` for adjacent nodes.

---

### Output

**Minimum Spanning Tree:**
- **Edges:** (A, B), (B, C), (B, E), (D, E)
- **Cost:** calculated

---

### Analysis

- **Time Complexity:** O((E + V) log V)
- **Space Complexity:** O(V)

---

### Notes

- **Data Structures:** Min-heap, adjacency list.
- **Applications:** Network design, clustering.
Is Dijkstra’s algorithm correct?

Invariant: For every vertex removed from the heap, \( \text{dist}(v) \) is the actual shortest distance from \( s \) to \( v \)

```
Dijkstra(G, s):
  1 for all \( v \in V \):
  2 \( \text{dist}(v) \leftarrow \infty \)
  3 \( \text{prev}(v) \leftarrow \text{null} \)
  4 \( Q \leftarrow \text{initialize}\_\text{priority}\_\text{queue}(\{(s, 0)\}) \)
  5 \( \text{while} Q \neq \emptyset \):
  6     \( (u, d) \leftarrow \text{extract}\_\text{min}(Q) \)
  7     for all \( (u, v) \in E \):
  8         if \( d + \text{wt}(u, v) < \text{dist}(v) \)
  9             \( \text{dist}(v) \leftarrow d + \text{wt}(u, v) \)
 10             \( \text{prev}(v) \leftarrow u \)
 11           \( Q \leftarrow \text{update}\_\text{priority}\_\text{queue}(Q, (v, \text{dist}(v))) \)
```

proof?

The only time a vertex gets visited is when the distance from \( s \) to that vertex is smaller than the distance to any remaining vertex.

Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path.

How do we get the actual paths?
Running time?

```plaintext
for all i ∈ V
    dist[i] = ∞
pred[i] = null
```

1 call to MakeHeap

```
while (Q.Count > 0):
    u = Q.DequeueMin()
    for v ∈ Q.Adj[u]:
        if dist[v] > dist[u] + w(u,v):
            dist[v] = dist[u] + w(u,v)
            pred[v] = u
            Q.DecreaseKey(v, dist[v])
```

[E] calls to DecreaseKey

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Bin heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MakeHeap</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>E</td>
</tr>
<tr>
<td>Total</td>
<td>O(</td>
<td>V</td>
</tr>
</tbody>
</table>

Is this an improvement?  If |E| < |V|² / log |V|

Running time?

```
for all i ∈ V
    dist[i] = ∞
pred[i] = null
```

1 call to MakeHeap

```
for all i ∈ V
    pred[i] = null
```

```
for all i ∈ V
    dist[i] = ∞
```

<table>
<thead>
<tr>
<th></th>
<th>Array</th>
<th>Bin heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 MakeHeap</td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>V</td>
</tr>
<tr>
<td></td>
<td>O(</td>
<td>E</td>
</tr>
<tr>
<td>Total</td>
<td>O(</td>
<td>V</td>
</tr>
</tbody>
</table>

Is this an improvement?  If |E| < |V|^² / log |V|
Running time

<table>
<thead>
<tr>
<th>Heap Type</th>
<th>MakeHeap</th>
<th>ExtractMin</th>
<th>DecreaseKey</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
</tr>
<tr>
<td>Bin heap</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
</tr>
<tr>
<td>Fib heap</td>
<td>$O(</td>
<td>V</td>
<td>)$</td>
<td>$O(</td>
</tr>
</tbody>
</table>

Dijkstra’s vs Prim’s

Dijkstra’s algorithm only works for positive edge weights.
Is Dijkstra’s algorithm correct?

Invariant: For every vertex removed from the heap, dist[v] is the actual shortest distance from s to v.

- The only time a vertex gets visited is when the distance from s to that vertex is smaller than the distance to any remaining vertex.
- Therefore, there cannot be any other path that hasn’t been visited already that would result in a shorter path.

We relied on having positive edge weights for correctness!

Bounding the distance

Another invariant: For each vertex v, dist[v] is an upper bound on the actual shortest distance.

start off at \( \infty \)

only update the value if we find a shorter distance

An update procedure: for an edge \((u,v)\)

\[
\text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u,v) \}
\]
\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

Consider the shortest path from s to v

\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

What happens if we update all of the vertices with the above update?

\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

What happens if we update all of the vertices with the above update?

\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

What happens if we update all of the vertices with the above update?

\[ \text{dist}[v] = \min \{\text{dist}[v], \text{dist}[u] + w(u, v)\} \]

dist[v] will be right if u is along the shortest path to v and dist[u] is correct

What happens if we update all of the vertices with the above update?
dist[v] will be right if u is along the shortest path to v and dist[u] is correct

Does the order that we update the vertices matter?

How many times do we have to do this for vertex p to have the correct shortest path from s?

- 1 times
\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u, v) \} \]

\text{dist}[v] will be right if \( u \) is along the shortest path to \( v \) and \( \text{dist}[u] \) is correct

How many times do we have to do this for vertex \( p \) to have the correct shortest path from \( s \)?

- \( i \) times

\[ \text{dist}[v] = \min \{ \text{dist}[v], \text{dist}[u] + w(u, v) \} \]

\text{dist}[v] will be right if \( u \) is along the shortest path to \( v \) and \( \text{dist}[u] \) is correct

How many times do we have to do this for vertex \( p \) to have the correct shortest path from \( s \)?

- \( i \) times

**Bellman-Ford algorithm**

```
Bellman-Ford(G, \( s \))
1. \text{for all } v \in V,
2. \text{dist}[v] \leftarrow \infty
3. \text{prev}[v] \leftarrow \text{null}
4. \text{dist}[s] \leftarrow 0
5. \text{for } i \leftarrow 1 \text{ to } |V| - 1,
6. \text{for all edges } (u, v) \in E,
7. \text{if } \text{dist}[u] < \infty \text{ and } w(u, v)
8. \text{dist}[v] \leftarrow \text{dist}[u] + w(u, v)
9. \text{prev}[v] \leftarrow u
10. \text{for all edges } (u, v) \in E,
11. \text{if dist}[u] < \infty \text{ and } w(u, v)
12. \text{return / else}
```
**Bellman-Ford algorithm**

Initialize all the distances.

1. For all \( v \in V \),
   1. set \( d[v] \rightarrow \infty \).
   2. set \( prev[v] \rightarrow null \).
   3. \( d[s] \rightarrow 0 \).

2. For \( i \leftarrow 1 \) to \( |V| - 1 \) do:
   1. For all edges \((u, v) \in E\):
      1. if \( d[u] \rightarrow d[u] + w(u, v) \)
      2. \( prev[v] \rightarrow u \).

3. For all edges \((u, v) \in E\):
   1. if \( d[u] \rightarrow d[u] + w(u, v) \)
   2. \( prev[v] \rightarrow u \).

**Check for negative cycles**

1. For all edges \((u, v) \in E\):
   1. if \( d[u] \rightarrow d[u] + w(u, v) \)
   2. return false.

---

**Negative cycles**

What is the shortest path from a to e?

- A → B → D → E
- A → C

**Bellman-Ford algorithm**

Initialize all the distances.

1. For all \( v \in V \),
   1. set \( d[v] \rightarrow \infty \).
   2. set \( prev[v] \rightarrow null \).
   3. \( d[s] \rightarrow 0 \).

2. For \( i \leftarrow 1 \) to \( |V| - 1 \) do:
   1. For all edges \((u, v) \in E\):
      1. if \( d[u] \rightarrow d[u] + w(u, v) \)
      2. \( prev[v] \rightarrow u \).

3. For all edges \((u, v) \in E\):
   1. if \( d[u] \rightarrow d[u] + w(u, v) \)
   2. \( prev[v] \rightarrow u \).

**Check for negative cycles**

1. For all edges \((u, v) \in E\):
   1. if \( d[u] \rightarrow d[u] + w(u, v) \)
   2. return false.
How many edges is the shortest path from s to:

A: 3

B: 5
How many edges is the shortest path from s to:

A: 3
B: 5
D: 7

Bellman-Ford algorithm

Iteration: 0

Iteration: 1
 Iteration: 2
 A has the correct distance and path

 Iteration: 3
 A has the correct distance and path

 Iteration: 4

 Iteration: 5
 B has the correct distance and path
Bellman-Ford algorithm

![Iteration: 6](image1)

![Iteration: 7](image2)

Correctness of Bellman-Ford

Loop invariant: After iteration i, all vertices with shortest paths from s of length i edges or less have correct distances

```
Bellman-Ford(G, s)
1: for all v ∈ V
2: dist[v] ← ∞
3: prev[v] ← nil
4: dist[s] ← 0
5: for i = 1 to |V| - 1
6: for all edges (u, v) ∈ E
7: if dist[u] + dist[v] + w(u, v) < dist[v]
8: dist[v] ← dist[u] + dist[v] + w(u, v)
9: prev[v] ← u
10: for all edges (u, v) ∈ E
11: if dist[u] < dist[v] + w(u, v)
12: return false
```

Runtime of Bellman-Ford

```
O(|V| |E|)
```

Can you modify the algorithm to run faster (in some circumstances)?

What algorithm would we use to calculate this?
Shortest Paths

- Bellman-Ford (since the graph has negative edges)
- O(VE)

What is the shortest path from A to C?
If we already calculated A to E using Bellman-Ford do we need to do any work?
No new calculations!
Bellman-Ford calculates all shortest paths starting at A.

What is the shortest path from D to C?
If we already calculated A to E using Bellman-Ford do we need to do any work?

Different source.
Have to run Bellman-Ford again!

All pairs shortest paths: calculate the shortest paths between all vertices.
All pairs shortest paths

All pairs shortest paths: calculate the shortest paths between all vertices

Easy solution?

Running time (in terms of $E$ and $V$):

$O(V^2E)$

• Bellman-Ford: $O(VE)$
  • $V$ calls, one for each vertex

DAGs?
Handout

Kruskal's Algorithm:
- Add smallest edge that connects two sets not already connected.

Prim's Algorithm:
- Starting at 0, grow MST by adding lowest weight edge out of current MST.

Bellman-Ford Algorithm:
- Checks for negative weight cycles.