CS3000: Algorithms & Data
Jonathan Ullman

Lecture 14:
• Finish Dijkstra’s Algorithm
• Bellman-Ford

Oct 26, 2018

Announcements
• HW5 due tonight
• HW6 out now, due 11/2
Shortest Paths

Given: \( G = (V, E, \delta(v)) \)
- source node \( s \in V 

Find shortest path from \( s \) to all \( u \in V 

source node

blue edges: shortest path tree
Weighted Graphs

• **Definition:** A weighted graph $G = (V, E, \{w(e)\})$
  - $V$ is the set of vertices
  - $E \subseteq V \times V$ is the set of edges
  - $w_e \in \mathbb{R}$ are edge weights/lengths/capacities
  - Can be directed or undirected

• **Today:**
  - Directed graphs (one-way streets)
  - Strongly connected (there is always some path)
  - Non-negative edge lengths ($\ell(e) \geq 0$)
Shortest Paths

- The length of a path $P = v_1 - v_2 - \cdots - v_k$ is the sum of the edge lengths.

- The distance $d(s, t)$ is the length of the shortest path from $s$ to $t$.

- **Shortest Path**: given nodes $s, t \in V$, find the shortest path from $s$ to $t$.

- **Single-Source Shortest Paths**: given a node $s \in V$, find the shortest paths from $s$ to every $t \in V$. 
Dijkstra’s Algorithm

• **Dijkstra’s Shortest Path Algorithm** is a modification of BFS for non-negatively weighted graphs

• **Informal Version:**
  - Maintain a set $S$ of explored nodes \( \text{Initially empty} \)
  - Maintain an upper bound on distance \( \text{Initially } d(s)=0, \ d(u)=\infty \)
    - If $u$ is explored, then we know $d(u)$ (Key Invariant)
    - If $u$ is explored, and $(u, v)$ is an edge, then we know $d(v) \leq d(u) + \ell(u, v)$

• Explore the “closest” unexplored node
• Repeat until we’re done
Dijkstra’s Algorithm: Demo
Dijkstra’s Algorithm: Demo

Initialize

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0(u)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
</tbody>
</table>

$S = \{\}$
Dijkstra’s Algorithm: Demo

Explore A

\[ \begin{array}{c|ccccc} \hline & A & B & C & D & E \\ \hline d_0(u) & 0 & \infty & \infty & \infty & \infty \\ d_1(u) & 0 & 10 & 3 & \infty & \infty \\ \hline \end{array} \]

\[ S = \{A\} \]
Dijkstra’s Algorithm: Demo

Explore C

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0(u)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d_1(u)$</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d_2(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

$S = \{A, C\}$
Dijkstra’s Algorithm: Demo

Explore E

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0(u)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d_1(u)$</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d_2(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>$d_3(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
</tbody>
</table>

$S = \{A, C, E\}$
Dijkstra’s Algorithm: Demo

Explore B

\[ S = \{A, C, E, B\} \]
Dijkstra’s Algorithm: Demo

Don’t need to explore D

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0(u)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d_1(u)$</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$d_2(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>$d_3(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>$d_4(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>

$S = \{A, C, E, B, D\}$
Dijkstra’s Algorithm: Demo

Maintain parent pointers so we can find the shortest paths

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0(u)$</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>$d_1(u)$</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>$d_2(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>$d_3(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>11</td>
<td>5</td>
</tr>
<tr>
<td>$d_4(u)$</td>
<td>0</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>5</td>
</tr>
</tbody>
</table>
Correctness of Dijkstra

• **Warmup 0:** initially, $d_0(s)$ is the correct distance

  Quite trivial

• **Warmup 1:** after exploring the node $\nu$, $d_1(\nu)$ is the correct distance

![Diagram](image)
Correctness of Dijkstra

• **Invariant:** after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$

• We just argued the invariant holds after we’ve explored the 1\textsuperscript{st} and 2\textsuperscript{nd} nodes
Correctness of Dijkstra

• **Invariant:** after we explore the i-th node, \( d_i(v) \) is correct for every \( v \in S \)

• **Proof:**

  - There is some path \( P \) of length \( l(P) = d_i(v) \)

  - Consider any other path \( P' \) from \( s \) to \( v \)
    - \( P' \) leaves \( S \) using some edge \( x \rightarrow y \)
    - The part of \( P' \) from \( s \) to \( x \) is a shortest path, has length \( d_i(x) \)
Correctness of Dijkstra

• **Invariant**: after we explore the i-th node, $d_i(v)$ is correct for every $v \in S$

• **Proof:**

  - $l(P') \geq (\text{distance to } x) + l(x \rightarrow y) \geq d_i(x) + l(x \rightarrow y) \geq d_i(y) \geq d_i(v) \geq d(s, v)$

  [Because $x$ was explored]
  [Because we explored $x$, not $y$]
  [Because $d_i$ is an upper bound]
Correctness of Dijkstra

- **Invariant**: after we explore the i-th node, \( d_i(v) \) is correct for every \( v \in S \)

- **Proof:**
  - \( l(P') > l(P) \)
  - \( \therefore P \) is a shortest path
  - \( \therefore d_i(v) = d(s,v) \)
There is no relationship between consecutive nodes that we explore.
Implementing Dijkstra

Dijkstra(G = (V,E, {ℓ(e)}, s):
    d[s] ← 0, d[u] ← ∞ for every u != s
    parent[u] ← ⊥ for every u
    Q ← V // Q holds the unexplored nodes

    While (Q is not empty):
        u ← argmin_{w∈Q} d[w] // Find closest unexplored
        Remove u from Q

        // Update the neighbors of u
        For ((u,v) in E):
            If (d[v] > d[u] + ℓ(u,v)):
                d[v] ← d[u] + ℓ(u,v)
                parent[v] ← u

    Return (d, parent)
Implementing Dijkstra Naively

- Need to explore all $n$ nodes
- Each exploration requires:
  - Finding the unexplored node $u$ with smallest distance
  - Updating the distance for each neighbor of $u$
    - Lookup current distance
    - Possibly decrease distance

interact with each other
Priority Queues / Heaps
Priority Queues

• Need a data structure Q to hold key-value pairs

\[
\begin{align*}
\text{keys} &= \text{nodes} \\
\text{values} &= \text{distances}
\end{align*}
\]

• Need to support the following operations
  • Insert(Q, k, v): add a new key-value pair
  • Lookup(Q, k): return the value of some key
  • ExtractMin(Q): identify the key with the smallest value
  • DecreaseKey(Q, k, v): reduce the value of some key

If \( d[v] > d[u] + l(u, v) \),

\[
d[v] \leftarrow d[u] + l(u, v)
\]

\[
u \leftarrow \text{argmin}_{w \in Q} d[w]
\]
Priority Queues

- **Naïve approach**: linked lists
  
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>11</td>
</tr>
<tr>
<td>c</td>
<td>12</td>
</tr>
<tr>
<td>e</td>
<td>2</td>
</tr>
<tr>
<td>h</td>
<td>36</td>
</tr>
<tr>
<td>b</td>
<td>4</td>
</tr>
<tr>
<td>g</td>
<td>20</td>
</tr>
<tr>
<td>k</td>
<td>42</td>
</tr>
<tr>
<td>d</td>
<td>10</td>
</tr>
<tr>
<td>f</td>
<td>8</td>
</tr>
</tbody>
</table>

  - Insert takes $O(1)$ time
  - ExtractMin, DecreaseKey take $O(n)$ time

- **Binary Heaps**: implement all operations in $O(\log n)$ time where $n$ is the number of keys
Heaps

• **Organize key-value pairs as a binary tree**
  • Later we’ll see how to store pairs in an array

• **Heap Order:** If a is the parent of b, then \( v(a) \leq v(b) \)

Each node represents a key-value pair

Create an array to locate any given key
Implementing ExtractMin

- If we delete the min, we get two binary trees
- Need to get back to being a tree
Implementing ExtractMin

- Not in heap order

Everything but the root satisfies heap order
Implementing ExtractMin

Fails to be a heap in one place

If there are $n$ key-value pairs, the # of levels is $O(\log n)$
Implementing ExtractMin
Implementing ExtractMin

After 0(\log n) swaps, the tree is in heap order again!
Implementing ExtractMin

• Three steps:
  • Pull the minimum from the root
  • Move the last element to the root
  • Repair the heap-order (heapify down)

\[
\begin{align*}
\text{OCD} & \\
\text{0} & \\
\text{Oli} & \\
\end{align*}
\]

\[
\begin{align*}
\text{time} & \\
\text{per swap} & \\
\text{O(1) time per swap} & \\
\text{O(log n) swaps} & \\
\text{O(log n) time} & \\
\end{align*}
\]
Implementing DecreaseKey
Implementing DecreaseKey

Fails to be a heap
Implementing DecreaseKey

• Two steps:
  • Change the key \( O(1) \) time
  • Repair the heap-order (heapify up) \( O(1) \) time per swap
    \( \times O(\log n) \) swaps

\[ \text{Total time} = O(\log n) \]
Implementing Insert

Fails to be a heap
Implementing Insert

![Diagram of a binary heap structure before and after insert operation]

- Before insert: (a,1), (c,2), (f,5), (h,15), (r,17), (o,20), (e,9), (b,15), (y,8), (x,16), (g,10), (z,3), (q,7), (w,4)
- After insert: (a,1), (c,2), (f,5), (g,10), (r,17), (o,20), (e,9), (b,15), (y,8), (x,16), (h,15), (z,3), (q,7), (p,11), (w,4), (x,16)

Fails to be a heap
Implementing Insert

• Two steps:
  • Put the new key in the last location
  • Repair the heap-order (heapify up)
Implementation Using Arrays

- Maintain an array $V$ holding the values
- Maintain an array $K$ mapping keys to values
  - Can find the value for a given key in $O(1)$ time
- For any node $i$ in the binary tree
  - $\text{LeftChild}(i) = 2i$
  - $\text{RightChild}(i) = 2i + 1$
  - $\text{Parent}(i) = \lfloor i/2 \rfloor$
Binary Heaps

• **Heapify:**
  - O(1) time to fix a single triple
  - With n keys, might have to fix O(log n) triples
  - Total time to heapify is O(log n)

• **Lookup** takes O(1) time
• **ExtractMin** takes O(log n) time
• **DecreaseKey** takes O(log n) time
• **Insert** takes O(log n) time
Implementing Dijkstra with Heaps

Dijkstra(G = (V,E, {ℓ(e)}), s):

Let Q be a new heap
Let parent[u] ← ⊥ for every u
Insert(Q,s,0), Insert(Q,u,∞) for every u ≠ s

While (Q is not empty):
(u,d[u]) ← ExtractMin(Q)
For ((u,v) in E):
    d[v] ← Lookup(Q,v)
    If (d[v] > d[u] + ℓ(u,v)):
        DecreaseKey(Q,v,d[u] + ℓ(u,v))
        parent[v] ← u

Return (d, parent)

Total Time: \[ \sum_{u} O(\log n) + O(\deg(u) \cdot \log n) = O((m+n) \log n) = O(m \log n) \]
Dijkstra Summary:

• **Dijkstra’s Algorithm** solves **single-source shortest paths** in non-negatively weighted graphs
  • Algorithm can fail if edge weights are negative!

• **Implementation:**
  • A **priority queue** supports all necessary operations
  • Implement priority queues using **binary heaps**
  • Overall running time of Dijkstra: $O(m \log n)$
    • With negatively weighted edges: $O(mn)$ time

• **Compare to BFS**