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

Oct 26, 2018
Shortest Paths

length of shortest path to B

shortest A→D path ends with edge (B,D)

source node
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 \( d(u) \) \( \text{(Key Invariant)} \)
    - 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 \( \square \) \( \text{Maintains invariant} \)
  - 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

<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>d₀(u)</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>d₁(u)</td>
<td>0</td>
<td>10</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

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>∞</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>
</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

<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>

$S = \{A, C, E, B\}$
Don’t need to explore D

\[
S = \{ A, C, E, B, D \}
\]
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>$\infty$</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>
Correctness of Dijkstra

- **Warmup 0**: initially, \( d_0(s) \) is the correct distance

  Quite a trivial stmt

- **Warmup 1**: after exploring the node \( v \), \( d_1(v) \) is the correct distance

  \[
  \begin{align*}
  d_1(u) &= 10 \\
  d_1(v) &= 2 \\
  d_1(w) &= 8 \\
  d_1(z) &= 2 \\
  \end{align*}
  \]

  - \( d(v) \leq 2 \)
  - To prove: length of \( P' \geq 2 \)
  - \( l(P') \geq l(s \rightarrow w) \)
  - \( \geq l(s \rightarrow v) = d(v) \)

  explore \( v \)

  2\(^{nd}\)
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 1st and 2nd nodes
Correctness of Dijkstra

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

- **Proof**:
  
  - $l(P) = d_i(v)$
  
  - Consider any other path $P'$. We'll show $l(P') \geq l(P)$
  
  - $P'$ can be written as $P_{s,x} + (x,y) + P_{y,v}$
    
    might as be the shortest path
Correctness of Dijkstra

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

- **Proof**:

\[
\ell(P') = \ell(P'_{s,x}) + \ell(x \rightarrow y) + \ell(P'_y,v)
\]

\[
\geq \ell(P'_{s,x}) + \ell(x \rightarrow y)
\]

\[
\geq d(s,x) + \ell(x \rightarrow y)
\]

\[
= d_i(x) + \ell(x \rightarrow y)
\]

\[
\geq d_i(y)
\]

- \([\ell(e) \geq 0]\)
- \([\text{def of distance}]\)
- \([\text{By invariant + } x \in S]\)
- \([\text{Because } x \text{ explored + } d_i \text{ goes down}]\)
\[ \forall d_i(v) \quad \begin{array}{c}
\because \quad d_i(v) \\
\text{Because } y \in S_j, \text{ but we chose to explore } v.
\end{array}
\]

\[
\begin{align*}
l(p) &= d_i(v) \quad \text{and for every path } p' \text{ from } s \text{ to } v, \\
l(p') &\geq d_i(v) \\
\therefore P \text{ is a shortest path and } d(s,v) &= d_i(v)
\end{align*}
\]

Suppose \( x \) is the \( j \)-th node explored for \( j < i \):

\[
\begin{align*}
\cdot &\quad d_j(y) \leq d_j(x) + l(x \rightarrow y) \\
\cdot &\quad d_i(y) \leq d_j(y) \\
\cdot &\quad d_i(x) = d_j(x) \\
\therefore &\quad d_i(y) \leq d_i(x) + l(x \rightarrow v)
\end{align*}
\]
Implementing Dijkstra

Dijkstra(G = (V,E,{\ell(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 ← arg\min_{w\in 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] + \ell(u,v)):
            d[v] ← d[u] + \ell(u,v)
            parent[v] ← u

Return (d, parent)
Implementing Dijkstra Naively

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

Total Time:
\[
\sum_{u \in V} O(n) + O(\text{deg}(u)+1) = O(n^2 + m)
\]

Bottleneck is finding the minimum distance node

1. Takes $O(n)$ time to find minimum distance node
2. For each of the $\text{deg}(u)$ neighbors
   2a. $O(1)$ time to lookup
   2b. $O(1)$ time to decrease
Priority Queues / Heaps
Priority Queues

• Need a data structure Q to hold key-value pairs
  
  keys = nodes
  values = d[u]

• 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))
  u ← arg min_w d[w]

\[ d[v] ← d[u] + l(u\rightarrow v) \]
Priority Queues

• **Naïve approach:** linked lists

<table>
<thead>
<tr>
<th>Key</th>
<th>a</th>
<th>c</th>
<th>e</th>
<th>h</th>
<th>b</th>
<th>g</th>
<th>k</th>
<th>d</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value</td>
<td>11</td>
<td>12</td>
<td>2</td>
<td>36</td>
<td>4</td>
<td>20</td>
<td>42</td>
<td>10</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
Implementing ExtractMin
Implementing ExtractMin

not a heap here, and only here
Implementing ExtractMin

Fails to be a heap in one place
Implementing ExtractMin
Implementing ExtractMin

- For any triple, we can fix the heap property in $O(1)$ time.
- Any swap lowers the problem one level.

Only $\lceil \log_2 (n+1) \rceil - 1 = O(\log n)$ levels!
Implementing ExtractMin

- Three steps:
  - Pull the minimum from the root $O(1)$
  - Move the last element to the root $O(1)$
  - Repair the heap-order (heapify down) $\mathcal{O}(\log n)$
Implementing DecreaseKey

Fails to be a heap
Implementing DecreaseKey

Fails to be a heap
Implementing DecreaseKey

• Two steps:
  • Change the key  \( O(1) \)
  • Repair the heap-order (heapify up)  \( O(\log n) \)
Implementing Insert
Implementing Insert
Implementing Insert

• Two steps:
  • Put the new key in the last location $O(1)$
  • Repair the heap-order (heapify up) $O(\log n)$
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
  - $\text{LeftChild}(i) = 2i$
  - $\text{RightChild}(i) = 2i+1$
  - $\text{Parent}(i) = [i/2]$
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

\[
\text{Dijkstra}(G = (V,E,\{\ell(e)\}, s) : \\
\text{Let } Q \text{ be a new heap} \\
\text{Let } \text{parent}[u] \leftarrow \bot \text{ for every } u \\
[\text{Insert}(Q,s,0), \text{Insert}(Q,u,\infty)] \text{ for every } u \neq s \\
\text{While (Q is not empty)}: \\
\text{(u,d[u]) } \leftarrow \text{ExtractMin}(Q) \leftarrow O(\log n) \text{ time} \\
\text{For ((u,v) in E): } \leftarrow \text{Loop } \text{deg}(u) \text{ times} \\
\text{d[v] } \leftarrow \text{Lookup}(Q,v) \leftarrow O(1) \\
\text{If (d[v] > d[u] + \ell(u,v)) :} \\
\text{DecreaseKey}(Q,v,d[u] + \ell(u,v)) \leftarrow O(\log n) \\
\text{parent}[v] \leftarrow u \\
\text{Return (d, parent)} \\
O(n) + \sum_{u \in V} O(\log n) + O(\log n \cdot \text{deg}(u)) \\
= O((m+n)\log n) = O(m\log n)
\]
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: $O(m \log n)$
- For negative weight edges, Bellman-Ford takes $O(mn)$ time

Compare to BFS