Shortest Path Algorithms

Andreas Klappenecker

[based on slides by Prof. Welch]
Single Source Shortest Path
Single Source Shortest Path

- **Given:**
  - a directed or undirected graph $G = (V,E)$
  - a source node $s$ in $V$
  - a weight function $w: E \rightarrow \mathbb{R}$. 
Single Source Shortest Path

- **Given:**
  - a directed or undirected graph $G = (V,E)$
  - a source node $s$ in $V$
  - a weight function $w: E \rightarrow R$.

- **Goal:** For each vertex $t$ in $V$, find a path from $s$ to $t$ in $G$ with minimum weight.
Single Source Shortest Path

- **Given:**
  - a directed or undirected graph $G = (V,E)$
  - a source node $s$ in $V$
  - a weight function $w: E \rightarrow R$.

- **Goal:** For each vertex $t$ in $V$, find a path from $s$ to $t$ in $G$ with minimum weight

**Warning!** Negative weight cycles are a problem:
Constant Weight Functions
Constant Weight Functions

Suppose that the weights of all edges are the same. How can you solve the single-source shortest path problem?
Constant Weight Functions

Suppose that the weights of all edges are the same. How can you solve the single-source shortest path problem?

Breadth-first search can be used to solve the single-source shortest path problem.
Constant Weight Functions

Suppose that the weights of all edges are the same. How can you solve the single-source shortest path problem?

Breadth-first search can be used to solve the single-source shortest path problem. Indeed, the tree rooted at s in the BFS forest is the solution.
Intermezzo: Priority Queues
Priority Queues

A min-priority queue is a data structure for maintaining a set $S$ of elements, each with an associated value called key.

This data structure supports the operations:

- $\text{insert}(S, x)$ which realizes $S := S \cup \{x\}$
- $\text{minimum}(S)$ which returns the element with the smallest key.
- $\text{extract-min}(S)$ which removes and returns the element with the smallest key from $S$.
- $\text{decrease-key}(S, x, k)$ which decreases the value of $x$'s
Simple Array Implementation

Suppose that the elements are numbered from 1 to $n$, and that the keys are stored in an array $\text{key}[1..n]$.

- insert and decrease-key take $O(1)$ time.
- extract-min takes $O(n)$ time, as the whole array must be searched for the minimum.
Binary min-heap Implementation

Suppose that we realize the priority queue of a set with \( n \) element with a binary min-heap.

- extract-min takes \( O(\log n) \) time.
- decrease-key takes \( O(\log n) \) time.
- insert takes \( O(\log n) \) time.

Building the heap takes \( O(n) \) time.
Fibonacci-Heap Implementation

Suppose that we realize the priority queue of a set with n elements with a Fibonacci heap. Then

• extract-min takes $O(\log n)$ amortized time.
• decrease-key takes $O(1)$ amortized time.
• insert takes $O(1)$ time.

[One can realize priority queues with worst case times as above]
Dijkstra’s Single Source Shortest Path Algorithm
Dijkstra's SSSP Algorithm

• Assumes all edge weights are nonnegative
• Similar to Prim's MST algorithm
• Start with source node s and iteratively construct a tree rooted at s
• Each node keeps track of tree node that provides cheapest path from s (not just cheapest path from any tree node)
• At each iteration, include the node whose cheapest path from s is the overall cheapest
Prim's vs. Dijkstra's

Prim's MST

Dijkstra's SSSP
Prim's vs. Dijkstra's

Prim's MST

Dijkstra's SSSP
Prim's vs. Dijkstra's

Prim's MST

Dijkstra's SSSP

Friday, October 12, 2012
Implementing Dijkstra's Alg.

• How can each node $u$ keep track of its best path from $s$?

• Keep an estimate, $d[u]$, of shortest path distance from $s$ to $u$

• Use $d$ as a key in a priority queue

• When $u$ is added to the tree, check each of $u$'s neighbors $v$ to see if $u$ provides $v$ with a cheaper path from $s$:
  • compare $d[v]$ to $d[u] + w(u,v)$
Dijkstra's Algorithm

• input: \( G = (V,E,w) \) and source node \( s \)

// initialization

• \( d[s] := 0 \)
• \( d[v] := \text{infinity for all other nodes } v \)
• initialize priority queue \( Q \) to contain all nodes using \( d \) values as keys
Dijkstra's Algorithm

- while Q is not empty do
  - u := extract-min(Q)
  - for each neighbor v of u do
    - if \( d[u] + w(u,v) < d[v] \) then // relax
      - \( d[v] := d[u] + w(u,v) \)
      - decrease-key(Q,v,d[v])
      - parent(v) := u
# Dijkstra's Algorithm Example

![Graph](image)

**a** is source node

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>abcde</td>
<td>bcde</td>
<td>cde</td>
<td>de</td>
<td>d</td>
<td>Ø</td>
</tr>
<tr>
<td>d[a]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d[b]</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d[c]</td>
<td>∞</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d[d]</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>16</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>d[e]</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm Example

**Diagram:**
- **a** is the source node.
- Edges and distances:
  - a to b: 2
  - a to c: 12
  - a to d: 6
  - a to e: 9
  - b to c: 8
  - b to d: 4
  - c to d: 3
  - c to e: 3
  - d to c: 4
  - d to e: 4

**Table:**

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>abcde</td>
<td>bcde</td>
<td>cde</td>
<td>de</td>
<td>d</td>
<td>Ø</td>
</tr>
<tr>
<td>d[a]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d[b]</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d[c]</td>
<td>∞</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d[d]</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>16</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>d[e]</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm Example

Graph:

- Node a is the source node.
- Edges and distances:
  - a to b: 2
  - a to c: 12
  - a to d: 6
  - a to e: 9
  - b to c: 4
  - b to d: 8
  - b to e: 2
  - c to d: 3
  - c to e: 4
  - d to e: 4

Table:

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>abcde</td>
<td>bcde</td>
<td>cde</td>
<td>de</td>
<td>d</td>
<td>Ø</td>
</tr>
<tr>
<td>d[a]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d[b]</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d[c]</td>
<td>∞</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d[d]</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>16</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>d[e]</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
### Dijkstra's Algorithm Example

**Graph:**

- **a** is source node
- Edges: (a, b) = 2, (a, d) = 10, (a, c) = 12, (b, c) = 8, (b, e) = 1, (c, d) = 6, (c, e) = 4, (d, e) = 3, (d, c) = 9

**Algorithm Table:**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Q</th>
<th>d[a]</th>
<th>d[b]</th>
<th>d[c]</th>
<th>d[d]</th>
<th>d[e]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a, b, c, d, e</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>b, c, d, e</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>c, d, e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>d, e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>d, e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>

**Graph:**

- **Nodes:** a, b, c, d, e
- **Edges:** (a, b) = 2, (a, d) = 10, (a, c) = 12, (b, c) = 8, (b, e) = 1, (c, d) = 6, (c, e) = 4, (d, e) = 3, (d, c) = 9

**Algorithm Table:**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Q</th>
<th>d[a]</th>
<th>d[b]</th>
<th>d[c]</th>
<th>d[d]</th>
<th>d[e]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>a, b, c, d, e</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>b, c, d, e</td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>c, d, e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>d, e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>d, e</td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm Example

\[ \begin{array}{c}
\text{iteration} \\
0 & 1 & 2 & 3 & 4 & 5 \\
\hline
\text{Q} & abcde & bcde & cde & de & d & \emptyset \\
\text{d[a]} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{d[b]} & \infty & 2 & 2 & 2 & 2 & 2 \\
\text{d[c]} & \infty & 12 & 10 & 10 & 10 & 10 \\
\text{d[d]} & \infty & \infty & \infty & 16 & 13 & 13 \\
\text{d[e]} & \infty & \infty & 11 & 11 & 11 & 11 \\
\end{array} \]
Dijkstra's Algorithm Example

```
d is source node

\[
\begin{array}{ccccccc}
\text{iteration} & 0 & 1 & 2 & 3 & 4 & 5 \\
\text{Q} & abcde & bcde & cde & de & d & \emptyset \\
\text{d[a]} & 0 & 0 & 0 & 0 & 0 & 0 \\
\text{d[b]} & \infty & 2 & 2 & 2 & 2 & 2 \\
\text{d[c]} & \infty & 12 & 10 & 10 & 10 & 10 \\
\text{d[d]} & \infty & \infty & \infty & 16 & 13 & 13 \\
\text{d[e]} & \infty & \infty & 11 & 11 & 11 & 11 \\
\end{array}
\]
```
Dijkstra's Algorithm Example

\[ a \text{ is source node} \]

\[ \begin{array}{c|c|c|c|c|c|c} 
\text{iteration} & 0 & 1 & 2 & 3 & 4 & 5 \\
\hline 
\text{Q} & abcde & bcde & cde & de & d & \emptyset \\
\hline 
d[a] & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline 
d[b] & \infty & 2 & 2 & 2 & 2 & 2 \\
\hline 
d[c] & \infty & 12 & 10 & 10 & 10 & 10 \\
\hline 
d[d] & \infty & \infty & \infty & 16 & 13 & 13 \\
\hline 
d[e] & \infty & \infty & 11 & 11 & 11 & 11 \\
\end{array} \]
Dijkstra's Algorithm Example

a is source node

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>abcde</td>
<td>bcde</td>
<td>cde</td>
<td>de</td>
<td>d</td>
<td>Ø</td>
</tr>
<tr>
<td>d[a]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d[b]</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d[c]</td>
<td>∞</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d[d]</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>16</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>d[e]</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
**Dijkstra's Algorithm Example**

![Graph Diagram]

The graph has nodes labeled **a**, **b**, **c**, **d**, and **e**. Node **a** is the source node.

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>abcde</td>
<td>bcde</td>
<td>cde</td>
<td>de</td>
<td>d</td>
<td>Ø</td>
</tr>
<tr>
<td>d[a]</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>d[b]</td>
<td>∞</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>d[c]</td>
<td>∞</td>
<td>12</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>d[d]</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>16</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>d[e]</td>
<td>∞</td>
<td>∞</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
Dijkstra's Algorithm Example

a is source node

<table>
<thead>
<tr>
<th>Q</th>
<th>iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>abcde</td>
<td>0</td>
</tr>
<tr>
<td>bcde</td>
<td>2</td>
</tr>
<tr>
<td>cde</td>
<td>12</td>
</tr>
<tr>
<td>de</td>
<td>11</td>
</tr>
<tr>
<td>d</td>
<td>13</td>
</tr>
<tr>
<td>Ø</td>
<td>0</td>
</tr>
</tbody>
</table>
Correctness of Dijkstra's Alg.

Let $T_i$ be the tree constructed after $i$-th iteration of the while loop:

- The nodes in $T_i$ are not in $Q$
- The edges in $T_i$ are indicated by parent variables

Show by induction on $i$ that the path in $T_i$ from $s$ to $u$ is a shortest path and has distance $d[u]$, for all $u$ in $T_i$.

**Basis:** $i = 1$.

$s$ is the only node in $T_1$ and $d[s] = 0$. 
Correctness of Dijkstra's Alg.

- **Induction:** Assume $T_i$ is a correct shortest path tree. We need to show that $T_{i+1}$ is a correct shortest path tree as well.
- Let $u$ be the node added in iteration $i$.
- Let $x = \text{parent}(u)$.

Need to show path in $T_{i+1}$ from $s$ to $u$ is a shortest path, and has distance $d[u]$. 
Correctness of Dijkstra's Alg

\[ P, \text{ path in } T_{i+1} \text{ from } s \text{ to } u \]
Correctness of Dijkstra's Alg

$T_i \subset T_{i+1}$

$P$, path in $T_{i+1}$ from $s$ to $u$

$P'$, another path from $s$ to $u$
Correctness of Dijkstra's Alg

Let $T_i$ be the set of nodes in $T_i$, and $P$ be the path from $s$ to $u$ in $T_{i+1}$. Let $P'$ be another path from $s$ to $u$. Let $(a,b)$ be the first edge in $P'$ that leaves $T_i$. Then $(a,b)$ is the first edge in $P'$ that leaves $T_i$. The correctness of Dijkstra's algorithm follows from the fact that all edges in $T_i$ have a lower weight than the edges in $T_{i+1}$.
Correctness of Dijkstra's Alg

Let P1 be part of P' before (a,b).
Let P2 be part of P' after (a,b).

\[ w(P') = w(P1) + w(a,b) + w(P2) \]
\[ \geq w(P1) + w(a,b) \quad \text{(nonneg wts)} \]
\[ \geq \text{wt of path in } T_i \text{ from } s \text{ to } a + w(a,b) \quad \text{(inductive hypothesis)} \]
\[ \geq w(s\rightarrow x \text{ path in } T_i) + w(x,u) \quad \text{(alg chose } u \text{ in iteration } i \text{ and } d\text{-values are accurate, by inductive hypothesis)} \]
\[ = w(P). \]

So P is a shortest path, and d[u] is accurate after iteration i+1.
Running Time of Dijkstra's Alg.

- **Initialization**: insert each node once
  - $O(V T_{\text{ins}})$
- **$O(V)$ iterations of while loop**
  - one extract-min per iteration $\Rightarrow O(V T_{\text{ex}})$
  - for loop inside while loop has variable number of iterations...
- **For loop has $O(E)$ iterations total**
  - one decrease-key per iteration $\Rightarrow O(E T_{\text{dec}})$
Running Time using Binary Heaps and Fibonacci Heaps

• \(O(V(T_{\text{ins}} + T_{\text{ex}}) + E \cdot T_{\text{dec}})\)

• If priority queue is implemented with a binary heap, then
  • \(T_{\text{ins}} = T_{\text{ex}} = T_{\text{dec}} = O(\log V)\)
  • total time is \(O(E \log V)\)

• There are fancier implementations of the priority queue, such as Fibonacci heap:
  • \(T_{\text{ins}} = O(1), T_{\text{ex}} = O(\log V), T_{\text{dec}} = O(1)\) (amortized)
  • total time is \(O(V \log V + E)\)
Using Simpler Heap

- $O(V(T_{ins} + T_{ex}) + E \cdot T_{dec})$

- If graph is dense, so that $|E| = \Theta(V^2)$, then it doesn't help to make $T_{ins}$ and $T_{ex}$ to be at most $O(V)$.

- Instead, focus on making $T_{dec}$ be small, say constant.

- Implement priority queue with an unsorted array:
  - $T_{ins} = O(1), T_{ex} = O(V), T_{dec} = O(1)$
The Bellman-Ford Algorithm
What About Negative Edge

- Dijkstra's SSSP algorithm requires all edge weights to be nonnegative. This is too restrictive, since it suffices to outlaw negative weight cycles.
- Bellman-Ford SSSP algorithm can handle negative edge weights. [It even can detect negative weight cycles if they exist.]
Bellman-Ford: The Basic Idea

- Consider each edge \((u,v)\) and see if \(u\) offers \(v\) a cheaper path from \(s\)
  - compare \(d[v]\) to \(d[u] + w(u,v)\)
- Repeat this process \(|V| - 1\) times to ensure that accurate information propagates from \(s\), no matter what order the edges are considered in
Bellman-Ford SSSP Algorithm

• input: directed or undirected graph $G = (V,E,w)$

// initialization
• initialize $d[v]$ to infinity and $\text{parent}[v]$ to nil for all $v$ in $V$ other than the source
• initialize $d[s]$ to 0 and $\text{parent}[s]$ to $s$

// main body
• for $i := 1$ to $|V| - 1$ do
  • for each $(u,v)$ in $E$ do  // consider in arbitrary order
  • if $d[u] + w(u,v) < d[v]$ then
    • $d[v] := d[u] + w(u,v)$
    • $\text{parent}[v] := u$
Bellman-Ford SSSP Algorithm

// check for negative weight cycles

  • for each (u,v) in E do
    • if d[u] + w(u,v) < d[v] then
      • output "negative weight cycle exists"
Running Time of Bellman-Ford

- $O(V)$ iterations of outer for loop
- $O(E)$ iterations of inner for loop
- $O(VE)$ time total
Correctness of Bellman-Ford

Assume no negative-weight cycles.

**Lemma:** $d[v]$ is never an underestimate of the actual shortest path distance from $s$ to $v$.

**Lemma:** If there is a shortest $s$-to-$v$ path containing at most $i$ edges, then after iteration $i$ of the outer for loop, $d[v]$ is at most the actual shortest path distance from $s$ to $v$.

**Theorem:** Bellman-Ford is correct.

This follows from the two lemmas and the fact
Bellman-Ford Example

process edges in order
(c,b)
(a,b)
(c,a)
(s,a)
(s,c)

Exercise!
Correctness of Bellman-Ford

• Suppose there is a negative weight cycle.
• Then the distance will decrease even after iteration $|V| - 1$
  • shortest path distance is negative infinity
• This is what the last part of the code checks for.
The Boost Graph Library

The BGL contains generic implementations of all the graph algorithms that we have discussed:

- Breadth-First-Search
- Depth-First-Search
- Kruskal’s MST algorithm
- Prim’s MST algorithm
- Strongly Connected Components
- Dijkstra’s SSSP algorithm
- Bellman-Ford SSSP algorithm

I recommend that you gain experience with this useful library. Recommended reading: The Boost Graph Library by J.G. Siek, L.-Q. Lee, and A. Lumsdaine, Addison-Wesley, 2002.