BFS Example

Shortest Paths

Breadth-First Search

\begin{enumerate}
\item \textsc{BFS}(G, s)
\item \textbf{for} each vertex \( u \in V[G] - \{s\} \)
\item \textbf{do} \( \text{color}[u] \leftarrow \text{WHITE} \)
\item \( d[u] \leftarrow \infty \)
\item \( \text{color}[s] \leftarrow \text{GRAY} \)
\item \( d[s] \leftarrow 0 \)
\item \textit{Q} \leftarrow \{s\}
\item \textbf{while} \textit{Q} \neq \emptyset
\item \textbf{do} \( u \leftarrow \text{head}(\textit{Q}) \)
\item \textbf{for} each \( v \in \text{Adj}[u] \)
\item \textbf{do if} \( \text{color}[v] = \text{WHITE} \)
\item \textbf{then} \( \text{color}[v] \leftarrow \text{GRAY} \)
\item \( d[v] \leftarrow d[u] + 1 \)
\item \textsc{ENQUEUE}($\textit{Q}, v$)
\item \textsc{DEQUEUE}($\textit{Q}$)
\item \( \text{color}[u] \leftarrow \text{BLACK} \)
\end{enumerate}

Digraph \( G = (V, E) \) with weight function \( W : E \rightarrow \mathbb{R} \)

Weight of path \( p = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_k \) is:

\[
w(p) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})
\]

"Shortest" path = path of minimum weight.

Applications
\begin{itemize}
\item Static/dynamic network routing
\item Robot motion planning
\end{itemize}
Optimal Substructure

**Theorem:** Subpaths of shortest paths are shortest paths

**Proof:** Cut and paste:

![Diagram](image-url)

If some subpath were not a shortest path, could substitute the shorter subpath and create a shorter total path. $\square$

Triangle Inequality

**Definition:** $\delta(u, v) \equiv$ weight of a shortest path from $u$ to $v$.

**Theorem:** $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$

**Proof:** Shortest path $u \leadsto v$ is no longer than any other path $u \leadsto v$ — in particular, the path concatenating the shortest path $u \leadsto x$ with the shortest path $x \leadsto v$. $\square$

Bellman-Ford Algorithm

**Negative weight cycle** $\Rightarrow$ no shortest path.

**Argument:** Can shorten path by traversing cycle. $\square$

![Diagram](image-url)

Most basic “single-source” shortest-paths algorithm

- Finds shortest path weights from specified source $s$ to all $v \in V$
- Maintains estimate $d[v]$ of path length from $s$ to $v$, which is updated iteratively
- Actual paths easily reconstructed (CLR §25.3)
**Bellman-Ford Algorithm**

**Bellman-Ford**\((G, w, s)\)

1. for each \(v \in V\)
2. \(d[v] \leftarrow \infty\)
3. \(d[s] \leftarrow 0\) \(\triangleright\) **Initializesingle-source**\((G, s)\)

4. for \(i \leftarrow 1\) to \(|V| - 1\)
5. do for each edge \((u, v) \in E\) \(\triangleright\) **Relax**
6. do if \(d[v] > d[u] + w(u, v)\)
7. then \(d[v] \leftarrow d[u] + w(u, v)\)

8. for each edge \((u, v) \in E\)
9. do if \(d[v] > d[u] + w(u, v)\)
10. then no solution

**Three code sections:**

- Lines 1 – 3:
  **Initialize**: \(d[v]\), which will converge to shortest-path values \(\delta\).

- Lines 4 – 7:
  **Relax**: \(|V| - 1\) times, do relaxation step for each edge.

- Lines 8 – 10:
  **Test**: was a solution achieved (iff no negative-weight cycles)?

---

**Bellman-Ford Algorithm**

**Bellman-Ford Algorithm: Running time**

**Example:**

```
Example:  

```

![Diagram of a graph with labeled edges and vertices]

**Running Time**: \(O(V \cdot E)\)

- constants are good
- it is simple
- short code

very practical.
Bellman-Ford Algorithm

Example:

- **Initialization.** Put initial $d$ values in nodes:
  $A \leftarrow 0$, rest $\leftarrow \infty$.

- **1st relaxation pass.** Process edges in order
  $(A, B), (A, C), (B, C), (B, D), (D, B), (D, C),
  (E, D), (B, E)$.

- **2nd relaxation pass.** Process edges in same
  order. Only $D$ changes.

Bellman-Ford Algorithm: Lemma

**Lemma:** $d[v] \geq \delta(s, v)$ always.

**Proof:**

- Initially true
- Let $v$ be first vertex for which $d[v] < \delta(s, v)$, and
  let $u$ be vertex that caused $d[v]$ to change:

  \[
  d[v] = d[u] + w(u, v)
  \]

- Then

  \[
  d[v] < \delta(s, v)
  \leq \delta(s, u) + \delta(u, v) \quad \text{(Triangle inequality)}
  \leq \delta(s, u) + w(u, v) \quad \text{(shortest path \leq specific)}
  \leq d[u] + w(u, v) \quad \text{(v is first violation)}
  \]

contradicts $d[v] = d[u] + w(u, v)$ (above).

Once $d[v]$ reaches $\delta(s, v)$, can’t change. Why?

Bellman-Ford Algorithm: Correctness

**Claim:** Bellman-Ford correct (i.e.,
after $|V| - 1$ passes, all the $d$ values are correct)

**Proof:** Let $v$ be a vertex, and consider shortest path
from $s$ to $v$ (assuming no neg-weight cycles):

$s \rightarrow v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v$

- Initially, $d[s] = 0$ is correct
  (and doesn’t change thereafter —
  code never increases $d$)
Bellman-Ford Algorithm: Correctness

Proof: (continued)

- After 1 pass thru edges, \( d[v_1] \) is correct (and doesn’t change...)

\( (d[s] \) is correct, and by optimal substructure, shortest distance is \( w(s, v_1) \).
1st pass sets \( d[v_1] = d[s] + w(s, v_1) \), which is right answer.

- After 2 passes through edges, \( d[v_2] \) is correct (and doesn’t change...)

Bellman-Ford Algorithm: Correctness

Dijkstra’s Algorithm

Dijkstra’s Algorithm:

- Non-negative edge weights \( \Rightarrow \) shortest paths always exist
  (If no weights negative, can beat Bellman-Ford)

- Like breadth-first-search
  (If all weights = 1, use BFS, otherwise Dijkstra.)

- Use \( Q \), priority queue keyed by \( d[v] \).
  Greedy, like Prim’s algorithm for MST
  BFS used FIFO queue

So... Bellman-Ford can be used to check for negative-weight cycles.
Dijkstra’s Algorithm: Pseudocode

Dijkstra(G, w, s)
1 for each v ∈ V
2 do d[v] ← ∞
3 d[s] ← 0
4 S ← Ø
5 Q ← V
6 while Q ≠ Ø
7 do u ← Extract-Min(Q)
8 S ← S ∪ {u}
9 for each v ∈ Adj[u]
10 do if d[v] > d[u] + w(u, v)
11 then d[v] ← d[u] + w(u, v)

What is line 7 doing?
What is line 11 doing?

Dijkstra’s Algorithm: Notes

Observe:

• relaxation step
• setting d[v] updates Q (Decrease-Key)
• similar to Prim’s minimum-spanning-tree algorithm

Dijkstra’s Algorithm: Analysis

Run Time Analysis:

• Extract-Min executed |V| times
• Decrease-Key executed |E| times

Time = |V| · T_{Extract-Min} + |E| · T_{Decrease-Key}
Dijkstra’s Algorithm: Analysis

\[ \text{Time} = |V| \cdot T_{\text{EXTRACT-MIN}} + |E| \cdot T_{\text{DECREASE-KEY}} \]

Analysis: Look at different \( Q \) implementations.

<table>
<thead>
<tr>
<th>( Q )</th>
<th>( T_{\text{EXTRACT-MIN}} )</th>
<th>( T_{\text{DECREASE-KEY}} )</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>( O(V) )</td>
<td>( O(1) )</td>
<td>( O(V) )</td>
</tr>
<tr>
<td>binary heap</td>
<td>( O(\lg V) )</td>
<td>( O(\lg V) )</td>
<td>( O(\lg V) )</td>
</tr>
<tr>
<td>( Q = \text{Fibonacci heap} )</td>
<td>( O(\lg V) )</td>
<td>( O(1) )</td>
<td>( O(V \lg V + E) )</td>
</tr>
</tbody>
</table>

\( Q = \text{unsorted array:} \)

- scan to find minimum
- just index and update to change key

\( Q = \text{Fibonacci heap} \)

Note advantage of amortized analysis:

Can use amortized Fibonacci heap bounds in analysis, as if they were worst-case bounds, and get (real) worst-case bounds on aggregate running time.

---

Dijkstra’s Algorithm: Correctness

**Correctness:** Prove that whenever \( u \) is added to \( S \),

\[ d[u] = \delta(s,u) \]

**Proof:**

- Note that \( \forall v \ \delta(v) \geq \delta(s,v) \)
- Let \( u \) be first vertex picked such that \( \exists \) shorter path than \( d[u] \)
  \[ \Rightarrow d[u] > \delta(s,u) \]

\( (Proof \ continued) \)

- Let \( y \) be first vertex \( \in V - S \) on actual shortest path from \( s \) to \( u \)
  \[ \Rightarrow d[y] = \delta(s,y) \]
  Because:
  - \( d[x] \) is set correctly for \( y \)'s predecessor \( x \in S \) on the shortest path (by choice of \( u \) as first choice for which that’s not true)
  - when put \( x \) into \( S \), relaxed \( (x,y) \), giving \( d[y] \) correct value
Dijkstra’s Algorithm: Correctness

(Proof continued)

\[ d[u] > \delta(s, u) \]
\[ = \delta(s, y) + \delta(y, u) \quad \text{(optimal substructure)} \]
\[ = d[y] + \delta(y, u) \]
\[ \geq d[y] \quad \text{(no negative weights)} \]

- But \( d[u] > d[y] \) \( \Rightarrow \) algorithm would have chosen \( y \) to process next, not \( u \). \textit{Contradiction.}