Graph Paths

CSE 373 - Data Structures
May 24, 2002
Readings and References

• Reading
  › Section 9.3, *Data Structures and Algorithm Analysis in C*, Weiss

• Other References

Some slides based on: CSE 326 by S. Wolfman, 2000
Path

- A path is a list of vertices \( \{v_1, v_2, \ldots, v_n\} \) such that \((v_i, v_{i+1})\) is in \(E\) for all \(0 \leq i < n\).

\[
p = \{Seattle, Salt Lake City, Chicago, Dallas, San Francisco\}
\]
Simple Paths and Cycles

- A *simple path* repeats no vertices
  - eg: {Seattle, Salt Lake City, San Francisco}
- A *cycle* is a path that starts and ends at the same vertex:
  - {Seattle, Salt Lake City, San Francisco, Seattle}
- A *simple cycle* is a cycle that repeats no vertices and the first vertex is also the last
- A *directed acyclic graph* (DAG) is a directed graph with no cycles
Connected

- G is *connected* if there is a path between every pair of distinct vertices in the graph.
- A graph which is not connected is the union of two or more connected subgraphs:
  - the subgraphs partition the graph G.
  - the subgraphs are the *connected components* of G.
  - note that the connected components are *not* connected to each other, but are themselves connected graphs.
Undirected Connected Graph
Connected Components of $G$
Path cost and Path length

- **Path cost**: the sum of the costs of each edge
- **Path length**: the number of edges in the path
  - Path length is the unweighted path cost (each edge = 1)

Seattle

San Francisco

Dallas

Salt Lake City

Chicago

length(p) = 5
cost(p) = 11.5
Shortest Path Problems

• Given a graph $G = (V, E)$ and a “source” vertex $s$ in $V$, find the minimum cost paths from $s$ to every vertex in $V$

• Many variations:
  › unweighted vs. weighted
  › cyclic vs. acyclic
  › pos. weights only vs. pos. and neg. weights
  › etc
Why study shortest path problems?

- Traveling on a budget: What is the cheapest airline schedule from Seattle to city X?
- Optimizing routing of packets on the internet:
  - Vertices are routers and edges are network links with different delays. What is the routing path with smallest total delay?
- Shipping: Find which highways and roads to take to minimize total delay due to traffic
Unweighted Shortest Path Problem

**Problem**: Given a “source” vertex $s$ in an unweighted graph $G = (V,E)$, find the shortest path from $s$ to all vertices in $G$
Breadth-First Search Solution

- **Basic Idea**: Starting at node \( s \), find vertices that can be reached using 0, 1, 2, 3, \ldots, \( N-1 \) edges (works even for cyclic graphs!)

![Graph Diagram]

\( A \)  \( B \)  \( C \)  \( D \)  \( E \)  \( F \)  \( G \)  \( H \)
Breadth-First Search Algorithm

- Uses a queue to track vertices that are “nearby”
- source vertex is \( s \)

\[
\begin{align*}
\text{Distance}[s] &= 0 \\
\text{Enqueue}(s) \\
\text{While queue is not empty} \\
\quad X &= \text{dequeue a vertex} \\
\quad \text{For each vertex } Y \text{ that is (adjacent to } X \text{ and not previously visited)} \\
\quad \quad \text{Distance}[Y] &= \text{Distance}[X] + 1 \\
\quad \quad \text{Previous}[Y] &= X \\
\quad \text{Enqueue } Y
\end{align*}
\]

- Running time (same as topological sort) = \( O(|V| + |E|) \)
What if edges have weights?

- Breadth First Search does not work anymore
  - minimum cost path may have more edges than minimum length path

Shortest path from C to A:
C → A (cost = 9)

Minimum Cost Path = C → E → D → A
(cost = 8)
Dijkstra’s Algorithm for Weighted Shortest Path

- Classic algorithm for solving shortest path in weighted graphs (without negative weights)
- A *greedy* algorithm (irrevocably makes decisions without considering future consequences)
- Each vertex has a cost for path from initial vertex
- Greedy choice – always expand to the least cost vertex
  - a vertex already visited may be updated if a better path to it is found before it is added to the distinguished set
Dijkstra’s Shortest Path Algorithm

• Initialize the cost of initial node to 0, and all the rest of the nodes to $\infty$

• Initialize set $S$ to be $\emptyset$

• While there are nodes left in the graph but not in $S$
  › Select the node $K$ with the lowest cost that is not in $S$ and identify the node as now being in $S$
  › for each node $A$ adjacent to $K$
    • if $(\text{cost}(K) + \text{cost}(K, A)) < A$’s currently known cost
      – set $\text{cost}(A) = \text{cost}(K) + \text{cost}(K, A)$
      – set $\text{previous}(A) = K$ so that we can remember the path
A weighted directed graph
Dijkstra example

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>$v_2$</td>
<td>2</td>
<td>$v_1$</td>
<td>2</td>
<td>$v_1$</td>
<td>*</td>
<td>2</td>
<td>$v_1$</td>
<td>*</td>
<td>2</td>
<td>$v_1$</td>
<td>*</td>
<td>2</td>
<td>$v_1$</td>
<td>*</td>
<td>2</td>
<td>$v_1$</td>
<td>*</td>
<td>2</td>
<td>$v_1$</td>
<td>*</td>
<td>2</td>
</tr>
<tr>
<td>$v_3$</td>
<td>8</td>
<td>3</td>
<td>$v_4$</td>
<td>3</td>
<td>$v_4$</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_4$</td>
<td>1</td>
<td>$v_1$</td>
<td>*</td>
<td>1</td>
<td>$v_1$</td>
<td>*</td>
<td>1</td>
<td>$v_1$</td>
<td>*</td>
<td>1</td>
<td>$v_1$</td>
<td>*</td>
<td>1</td>
<td>$v_1$</td>
<td>*</td>
<td>1</td>
<td>$v_1$</td>
<td>*</td>
<td>1</td>
<td>$v_1$</td>
<td></td>
</tr>
<tr>
<td>$v_5$</td>
<td>8</td>
<td>3</td>
<td>$v_4$</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td>*</td>
<td>3</td>
<td>$v_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_6$</td>
<td>8</td>
<td>9</td>
<td>$v_4$</td>
<td>9</td>
<td>$v_4$</td>
<td>9</td>
<td>$v_4$</td>
<td>8</td>
<td>$v_3$</td>
<td>6</td>
<td>$v_7$</td>
<td>*</td>
<td>6</td>
<td>$v_7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$v_7$</td>
<td>8</td>
<td>5</td>
<td>$v_4$</td>
<td>5</td>
<td>$v_4$</td>
<td>5</td>
<td>$v_4$</td>
<td>5</td>
<td>$v_4$</td>
<td>5</td>
<td>$v_4$</td>
<td>*</td>
<td>5</td>
<td>$v_4$</td>
<td>*</td>
<td>5</td>
<td>$v_4$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Analysis of Dijkstra’s Algorithm

While there are nodes left in the graph but not in S
Select the node K with the lowest cost that is not in S and identify the node as now being in S
for each node A adjacent to K
if (cost(K)+cost(K,A) < A’s currently known cost
set cost(A) = cost(K)+cost(K,A)
set previous(A) = K so that we can remember the path

Total time = $|V| (O(|V|)) + O(|E|) = O(|V|^2 + |E|)$

Dense graph: $|E| = \Theta(|V|^2) \rightarrow$ Total time = $O(|V|^2) = O(|E|)$
Sparse graph: $|E| = \Theta(|V|) \rightarrow$ Total time = $O(|V|^2) = O(|E|^2)$

Quadratic! Can we do better?
Analysis of Dijkstra’s Algorithm

Yes! Use a priority queue to store vertices with key = cost

$|V|$ times:
Select the unknown node $N$ with the lowest cost

$|E|$ times:
$A$’s cost = $N$’s cost + cost of $(N, A)$

Total run time = $O(|V| \log |V| + |E| \log |V|)$
Does It Always Work?

• Dijkstra’s algorithm is an example of a greedy algorithm
• Greedy algorithms always make choices that currently seem the best
  › Short-sighted – no consideration of long-term or global issues
  › Locally optimal does not always mean globally optimal
• In Dijkstra’s case – choose the least cost node, but what if there is another path through other vertices that is cheaper?
“Cloudy” Proof

If the path to $G$ is the next shortest path, the path to $P$ must be at least as long. Note - no negative path weights!
Therefore, any path through $P$ to $G$ cannot be shorter!
Inside the Cloud (Proof)

Everything inside the cloud has the correct shortest path
Proof is by induction on the # of nodes in the cloud:
  › Base case: Initial cloud is just the source with shortest path 0
  › Inductive hypothesis: cloud of k-1 nodes all have shortest paths
  › Inductive step: choose the least cost node G → has to be the shortest path to G (previous slide). Add k\textsuperscript{th} node G to the cloud