Graph Algorithms

CptS 223 – Advanced Data Structures

Larry Holder
School of Electrical Engineering and Computer Science
Washington State University
Shortest-Path Algorithms

- Find the “shortest” path from point A to point B
- “Shortest” in time, distance, cost, …
- Numerous applications
  - Map navigation
  - Flight itineraries
  - Circuit wiring
  - Network routing
Shortest Path Problems

- Input is a weighted graph where each edge \((v_i, v_j)\) has cost \(c_{i,j}\) to traverse the edge.
- Cost of a path \(v_1v_2...v_N\) is \(\sum_{i=1}^{N-1} c_{i,i+1}\)
  - Weighted path cost
- Unweighted path length is \(N-1\), number of edges on path
Shortest Path Problems

- Single-source shortest path problem
  - Given a weighted graph \( G = (V, E) \), and a start vertex \( s \), find the minimum weighted path from \( s \) to every other vertex in \( G \)
Negative Weights

- Graphs can have negative weights
- E.g., arbitrage
  - Shortest positive-weight path is a net gain
  - Path may include individual losses
- Problem: Negative weight cycles
  - Allow arbitrarily-low path costs
- Solution
  - Detect presence of negative-weight cycles
Shortest Path Problems

- Unweighted shortest-path problem: $O(|E| + |V|)$
- Weighted shortest-path problem
  - No negative edges: $O(|E| \log |V|)$
  - Negative edges: $O(|E| \cdot |V|)$
- Acyclic graphs: $O(|E| + |V|)$
- No asymptotically faster algorithm for single-source/single-destination shortest path problem
Unweighted Shortest Paths

- No weights on edges
- Find shortest length paths
- Same as weighted shortest path with all weights equal
- Breadth-first search
Unweighted Shortest Paths

- For each vertex, keep track of
  - Whether we have visited it \((\text{known})\)
  - Its distance from the start vertex \((d_v)\)
  - Its predecessor vertex along the shortest path from the start vertex \((p_v)\)
Unweighted Shortest Paths

```cpp
void Graph::unweighted( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( int currDist = 0; currDist < NUM_VERTICES; currDist++ )
        for each Vertex v
            if( !v.known && v.dist == currDist )
                {
                    v.known = true;
                    for each Vertex w adjacent to v
                        if( w.dist == INFINITY )
                            {
                                w.dist = currDist + 1;
                                w.path = v;
                            }
                }
}
```

Solution 1: Repeatedly iterate through vertices, looking for unvisited vertices at current distance from start vertex s.

Running time: $O(|V|^2)$
Unweighted Shortest Paths

```cpp
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty() )
    {
        Vertex v = q.dequeue();

        for each Vertex w adjacent to v
            if( w.dist == INFINITY )
            {
                w.dist = v.dist + 1;
                w.path = v;
                q.enqueue( w );
            }
    }
}
```

Solution 2: Ignore vertices that have already been visited by keeping only unvisited vertices (distance = $\infty$) on the queue.

Running time: $O(|E|+|V|)$
**Unweighted Shortest Paths**

<table>
<thead>
<tr>
<th>v</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>2</td>
<td>( v_1 )</td>
<td>F</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>F</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>2</td>
<td>( v_1 )</td>
<td>F</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>1</td>
<td>( v_3 )</td>
<td>F</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
</tr>
</tbody>
</table>

Q: \( v_3, v_1, v_6 \) \( v_6, v_2, v_4 \) \( v_2, v_4 \)

<table>
<thead>
<tr>
<th>v</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
<th>( \text{known} )</th>
<th>( d_v )</th>
<th>( p_v )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( v_1 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>( v_2 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_3 )</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( v_4 )</td>
<td>F</td>
<td>2</td>
<td>( v_1 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
<td>T</td>
<td>2</td>
<td>( v_1 )</td>
</tr>
<tr>
<td>( v_5 )</td>
<td>F</td>
<td>3</td>
<td>( v_2 )</td>
<td>F</td>
<td>3</td>
<td>( v_2 )</td>
<td>T</td>
<td>3</td>
<td>( v_2 )</td>
<td>T</td>
<td>3</td>
<td>( v_2 )</td>
</tr>
<tr>
<td>( v_6 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
<td>T</td>
<td>1</td>
<td>( v_3 )</td>
</tr>
<tr>
<td>( v_7 )</td>
<td>F</td>
<td>( \infty )</td>
<td>0</td>
<td>F</td>
<td>3</td>
<td>( v_4 )</td>
<td>F</td>
<td>3</td>
<td>( v_4 )</td>
<td>T</td>
<td>3</td>
<td>( v_4 )</td>
</tr>
</tbody>
</table>

Q: \( v_4, v_5 \) \( v_5, v_7 \) \( v_7 \) empty
Weighted Shortest Paths

- Dijkstra’s algorithm
  - Use priority queue to store unvisited vertices by distance from \( s \)
  - After deleteMin \( v \), update distances of remaining vertices adjacent to \( v \) using decreaseKey
  - Does not work with negative weights
Dijkstra’s Algorithm

/**
 * PSEUDOCODE sketch of the Vertex structure.
 * In real C++, path would be of type Vertex *,
 * and many of the code fragments that we describe
 * require either a dereferencing * or use the
 * -> operator instead of the . operator.
 * Needless to say, this obscures the basic algorithmic ideas.
 */

struct Vertex
{
    List   adj;    // Adjacency list
    bool   known;
    DistType dist;  // DistType is probably int
    Vertex path;   // Probably Vertex *, as mentioned above
                  // Other data and member functions as needed
};
void Graph::dijkstra( Vertex s )
{
    for each Vertex v
    {
        v.dist = INFINITY;
        v.known = false;
    }

    s.dist = 0;

    for( ; ; )
    {
        Vertex v = smallest unknown distance vertex;
        if( v == NOT_A_VERTEX )
            break;
        v.known = true;

        for each Vertex w adjacent to v
            if( !w.known )
                if( v.dist + cvw < w.dist )
                    {
                        // Update w
                        decrease( w.dist to v.dist + cvw );
                        w.path = v;
                    }
    }
}

BuildHeap: O(|V|)
DeleteMin: O(|V| log |V|)
DecreaseKey: O(|E| log |V|)
Total running time: O(|E| log |V|)
Dijkstra
Why Dijkstra Works

- Hypothesis
  - A least-cost path from X to Y contains least-cost paths from X to every city on the path
  - E.g., if X→C1→C2→C3→Y is the least-cost path from X to Y, then
    - X→C1→C2→C3 is the least-cost path from X to C3
    - X→C1→C2 is the least-cost path from X to C2
    - X→C1 is the least-cost path from X to C1
Why Dijkstra Works

- Assume hypothesis is false
  - I.e., Given a least-cost path P from X to Y that goes through C, there is a better path P' from X to C than the one in P

- Show a contradiction
  - But we could replace the subpath from X to C in P with this lesser-cost path P'
  - The path cost from C to Y is the same
  - Thus we now have a better path from X to Y
  - But this violates the assumption that P is the least-cost path from X to Y

- Therefore, the original hypothesis must be true
Printing Shortest Paths

/**
 * Print shortest path to v after dijkstra has run.
 * Assume that the path exists.
 */
void Graph::printPath( Vertex v )
{
    if( v.path != NOT_A_VERTEX )
    {
        printPath( v.path );
        cout << " to ";
    }
    cout << v;
}
Negative Edge Costs

```cpp
void Graph::weightedNegative( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}

Running time: O(|E|·|V|)

Negative weight cycles?
Shortest Path Algorithms

- Important graph problem with numerous applications
- Unweighted graph: $O(|E| + |V|)$
- Weighted graph
  - Dijkstra: $O(|E| \log |V|)$
  - Negative weights: $O(|E| \cdot |V|)$
- All-pairs shortest paths
  - Dijkstra: $O(|V| \cdot |E| \log |V|) = O(|V|^3 \log |V|)$
  - Floyd-Warshall: $O(|V|^3)$