Computational Methods in IS Research

Graph Algorithms
Shortest Path

Nirmalya Roy
Department of Information Systems
University of Maryland Baltimore County
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

  - Weighted path cost \(\sum_{i=1}^{N-1} c_{i,i+1}\)

- Unweighted path length is \(N - 1\), number of edges on path.
**Shortest-Path Problems (cont’d)**

- Single-source shortest path problem
  - Given a weighted graph $G = (V, E)$, and a distinguished start vertex, $s$, find the minimum weighted path from $s$ to every other vertex in $G$
  - The shortest weighted path from $v_1$ to $v_6$ has a cost of 6 and $v_1v_4v_7v_6$
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
  - Shortest path cost from $v_5$ to $v_4 = 1$?
    - $v_5 v_4 v_2 v_5 v_4 = -5$, still not shortest
  - Shortest path from $v_1$ to $v_6$ undefined
    - negative-cost cycle
- Solution
  - Detect presence of negative-weight cycles
Unweighted Shortest Paths

- Problem: Find the shortest path from some vertex $s$ to all other vertices
  - Input: $s$, the source/starting vertex
  - Output: minimum # of edges contained on the path
  - No weights on edges

- Find shortest length paths
  - Same as weighted shortest path with all weights equal
  - Start vertex is $s = v_3$
  - Shortest path from $s$ to $v_3$ is 0

- Breadth-First Search (BFS)
  - Process vertices in layers
    - Closest to the start are evaluated first
    - Then most distant vertices
Unweighted Shortest Paths (cont’d)

- For each vertex, keep track of
  - Whether we have visited it (*known*)
  - Its distance from the start vertex (*d*<sub>v</sub>)
  - Its predecessor vertex along the shortest path from the start vertex (*p*<sub>v</sub>)

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th><em>d</em>&lt;sub&gt;v&lt;/sub&gt;</th>
<th><em>p</em>&lt;sub&gt;v&lt;/sub&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₁</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₂</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₃</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>v₄</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₅</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₆</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
<tr>
<td>v₇</td>
<td>F</td>
<td>∞</td>
<td>0</td>
</tr>
</tbody>
</table>
Unweighted Shortest Paths (cont’d)

```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 + 
                        w.path = v;
                    }
            }
}
```

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

**Running time**: \(O(|V|^2)\)

<table>
<thead>
<tr>
<th></th>
<th>known</th>
<th>(d_v)</th>
<th>(p_v)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>(\infty)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>(\infty)</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>F</td>
<td>(\infty)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>F</td>
<td>(\infty)</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td>(\infty)</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>F</td>
<td>(\infty)</td>
<td>0</td>
</tr>
</tbody>
</table>
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 = ∞) on the queue

Running time: O(|E|+|V|) with adjacency lists

Two groups of vertices based on currDist and currDist+1

known data member is not used
### Unweighted Shortest Paths (cont’d)

![Graph](image)

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

**Q:**
- $v_4, v_5$
- $v_5, v_7$
- $v_7$
- empty
Weighted Shortest Paths

- Dijkstra’s algorithm
  - Proceeds in stages just like the unweighted shortest-path algorithm
  - Select a vertex $v$, which has the smallest $d_v$ among all the unknown vertices and declares the shortest path from $s$ to $v$ is known
  - Use priority queue to store unvisited vertices by distance from $s$
  - After deleteMin $v$, update distance of remaining vertices adjacent to $v$ using decreaseKey
  - Does not work with negative weights
/**
 * 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
};
Dijkstra’s Algorithm Implementation

- Priority queue such as binary heap
- Selection of a vertex \( v \) is deleteMin operation
  - Once unknown minimum vertex is found it is no longer unknown
  - Must be removed from future consideration
- Update of \( w \)'s distance (adjacent to \( v \))
  - decreaseKey operation
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’s Adjacency List

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_5$</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>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</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>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>$\infty$</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>F</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>F</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>F</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>F</td>
<td>9</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>F</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v</th>
<th>known</th>
<th>$d_v$</th>
<th>$p_v$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_1$</td>
<td>T</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$v_2$</td>
<td>T</td>
<td>2</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_3$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_4$</td>
<td>T</td>
<td>1</td>
<td>$v_1$</td>
</tr>
<tr>
<td>$v_5$</td>
<td>T</td>
<td>3</td>
<td>$v_4$</td>
</tr>
<tr>
<td>$v_6$</td>
<td>T</td>
<td>6</td>
<td>$v_7$</td>
</tr>
<tr>
<td>$v_7$</td>
<td>T</td>
<td>5</td>
<td>$v_4$</td>
</tr>
</tbody>
</table>
Dijkstra’s Algorithm
Why Dijkstra Works

- Dijkstra’s algorithm is known as **greedy algorithm**
  - Solves a problem in stages by doing what appears to be the best thing at each stage

- Prove that it 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 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
/**
* 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 but No Cycles

Running time: $O(|E| \cdot |V|)$

Negative weight cycles?
Dijkstra’s algorithm does not work

• Vertex $u$ is known but there may be a path from unknown vertex $v$ back to $u$ that is very negative
• Add a constant value to each edge cost?
• Solve this with the combination of unweighted and weighted algorithms

Does not work for above graph, as it has negative-cost cycles
Shortest-Path Problems (cont’d)

- Unweighted shortest-path problem: \(O(|E| + |V|)\)
- Weighted shortest-path problem
  - No negative edges: \(O(|E| \log |V|)\)
  - Negative edges: \(O(|E| \times |V|) \rightarrow \text{poor time bound}\)
- Acyclic graphs: \(O(|E| + |V|)\) in linear time
- No asymptotically faster algorithm for single-source/single-destination shortest path problem
  - No algorithms find the path from \(s\) to one vertex (one-to-one) any faster than finding the path from \(s\) to all vertices (one-to-many)
Shortest Path Algorithms

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