UMBC

AN HONORS UNIVERSITY IN MARYLAND

Computational Methods in IS Research

Graph Algorithms
Shortest Path

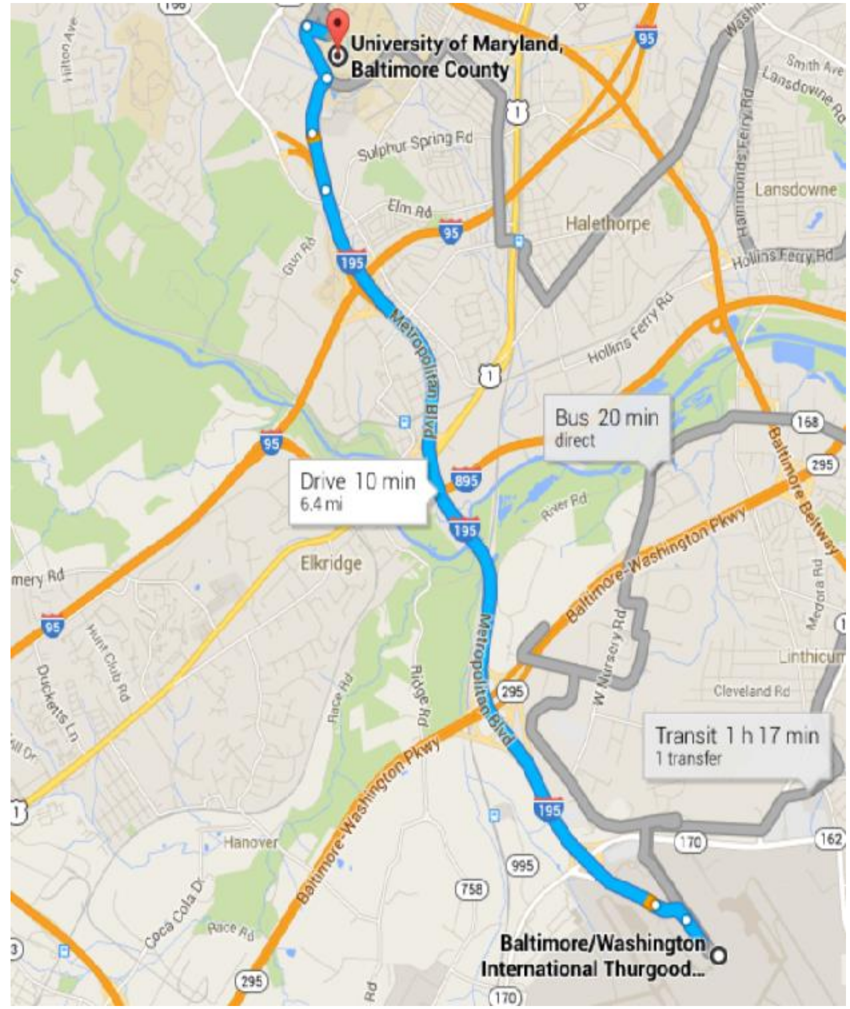
Nirmalya Roy

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

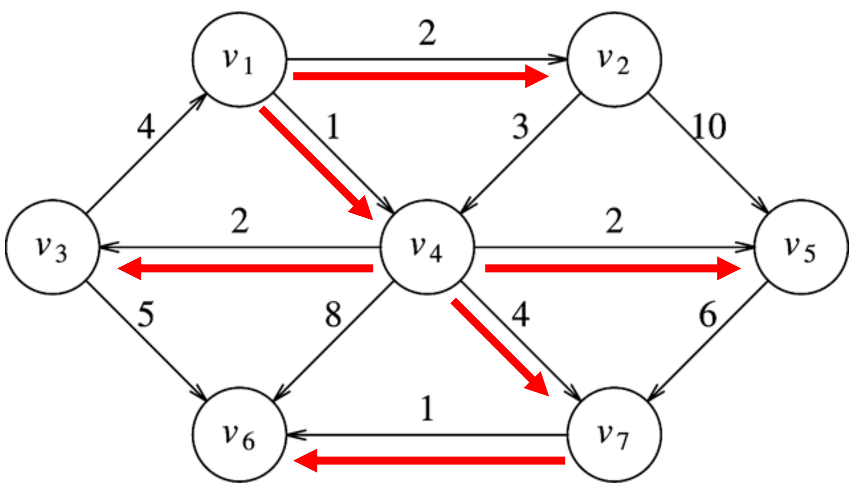
- Input is a weighted graph where each edge (v_i, v_j) has cost $c_{i,j}$ to traverse the edge

- 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

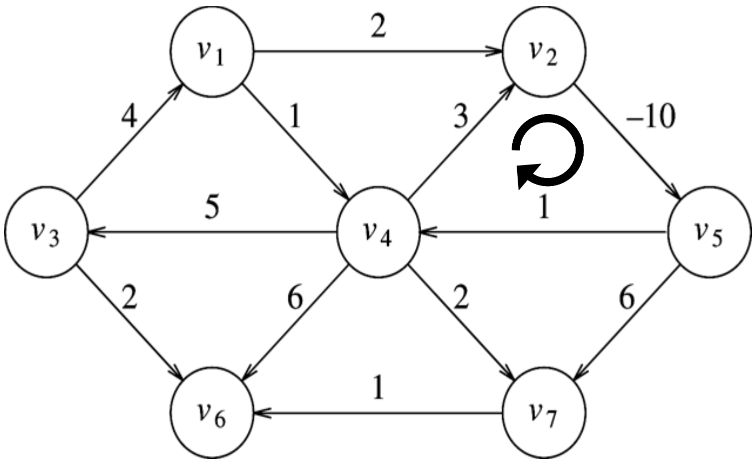
Shortest-Path Problems (cont'd)

- 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 $v_1 v_4 v_7 v_6$



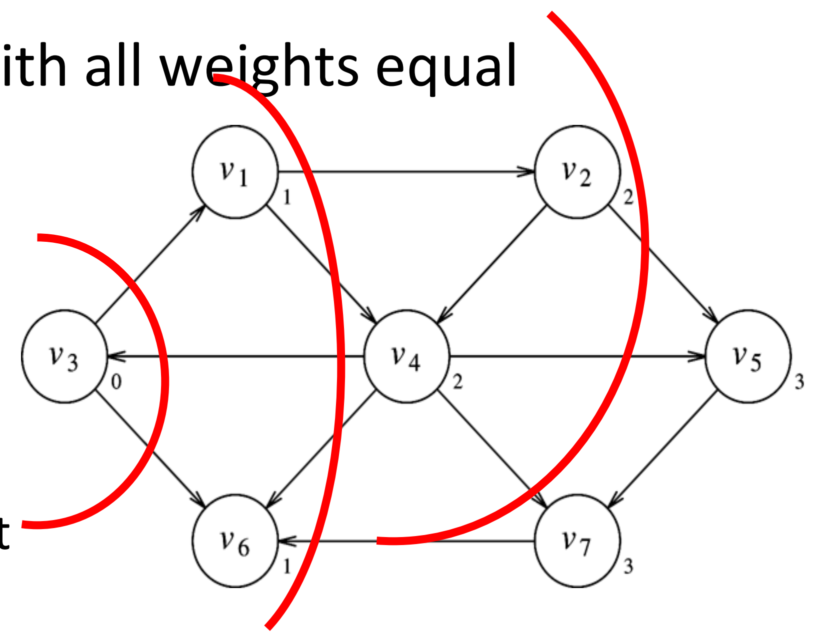
Negative Weights

- 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



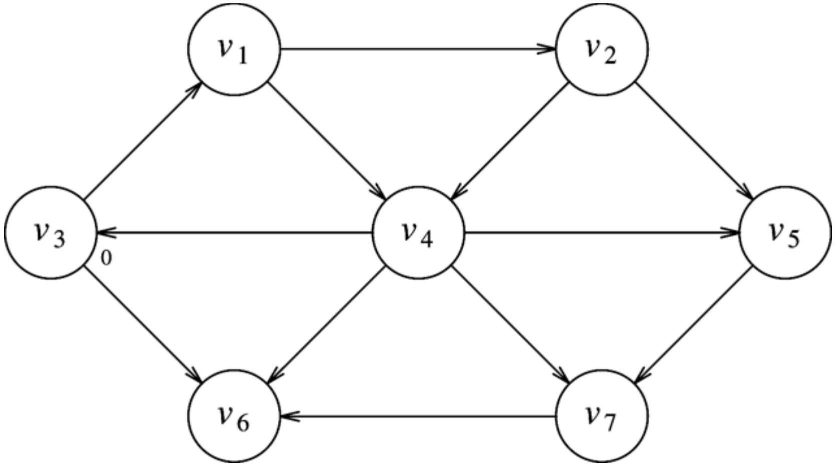
Unweighted Shortest Paths

- 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_v)
  - Its predecessor vertex along the shortest path from the start vertex (p_v)



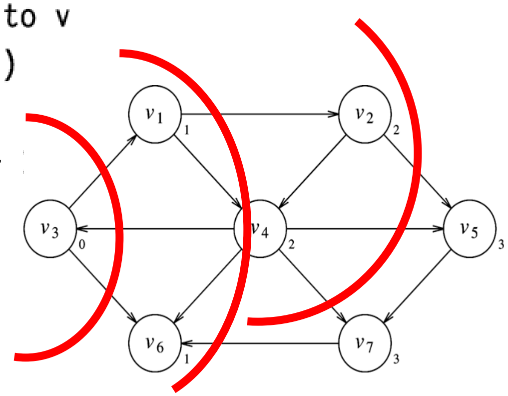
v	*known*	d_v	p_v
v_1	*F*	∞	0
v_2	*F*	∞	0
v_3	*F*	0	0
v_4	*F*	∞	0
v_5	*F*	∞	0
v_6	*F*	∞	0
v_7	*F*	∞	0

Unweighted Shortest Paths (cont'd)

```
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: $O(|V|^2)$



v	*known*	d_v	p_v
v_1	F	∞	0
v_2	F	∞	0
v_3	F	0	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

Unweighted Shortest Paths (cont'd)

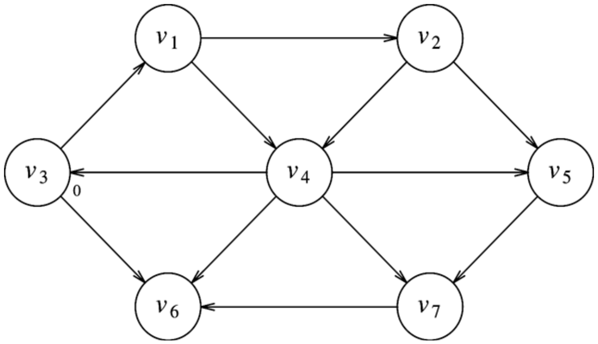
```
void Graph::unweighted( Vertex s )
{
    Queue<Vertex> q;

    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

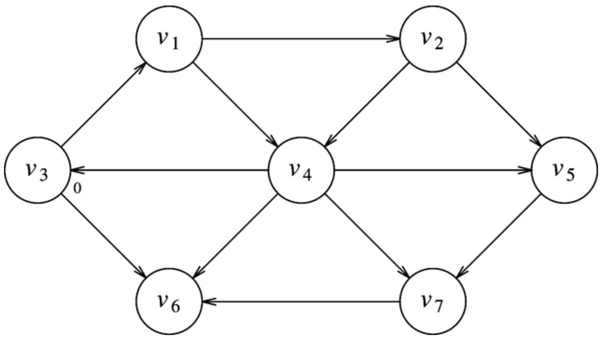
Running time: O(|E|+|V|) with adjacency lists

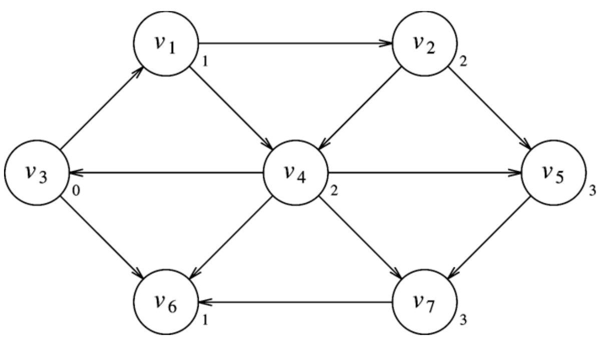
Two groups of vertices based on `currDist` and `currDist+1`

`known` data member is not used



Unweighted Shortest Paths (cont'd)





	Initial State			v_3 Dequeued			v_1 Dequeued			v_6 Dequeued		
v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	F	∞	0	F	1	v_3	T	1	v_3	T	1	v_3
v_2	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_3	F	0	0	T	0	0	T	0	0	T	0	0
v_4	F	∞	0	F	∞	0	F	2	v_1	F	2	v_1
v_5	F	∞	0	F	∞	0	F	∞	0	F	∞	0
v_6	F	∞	0	F	1	v_3	F	1	v_3	T	1	v_3
v_7	F	∞	0	F	∞	0	F	∞	0	F	∞	0
Q:	v_3			v_1, v_6			v_6, v_2, v_4			v_2, v_4		

	v_2 Dequeued			v_4 Dequeued			v_5 Dequeued			v_7 Dequeued		
v	known	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v	*known*	d_v	p_v
v_1	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_2	T	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_3	T	0	0	T	0	0	T	0	0	T	0	0
v_4	F	2	v_1	T	2	v_1	T	2	v_1	T	2	v_1
v_5	F	3	v_2	F	3	v_2	T	3	v_2	T	3	v_2
v_6	T	1	v_3	T	1	v_3	T	1	v_3	T	1	v_3
v_7	F	∞	0	F	3	v_4	F	3	v_4	T	3	v_4
Q:	v_4, v_5			v_5, v_7			v_7			empty		

Weighted Shortest Paths

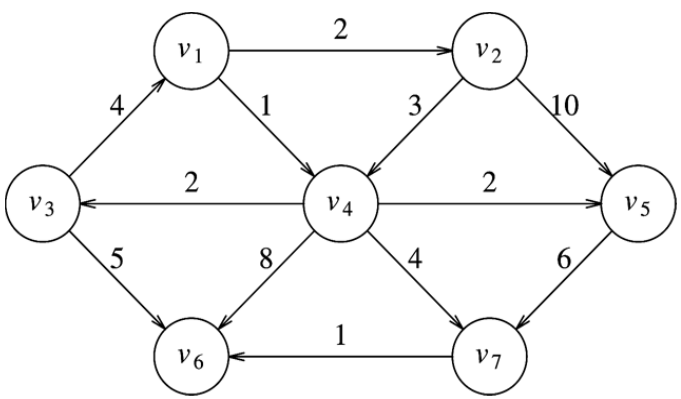
- Dijkstra's algorithm
  - Proceeds in stages just like the unweighted shortest-path algorithm
  - Select a vertex `v`, which has the smallest d_v among all the `unknown` vertices and declares the shortest path from `s` to `v` is `known`
  - Use priority queue to store unvisited vertices by distance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

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
};
```

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|)

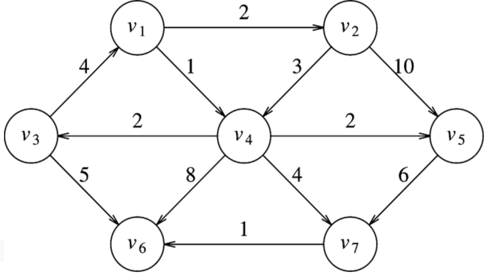
DeleteMin: O(|V| log |V|)

- *In unweighted case we set* $d_w = d_v + 1$ *if* d_w *= infinity*
- *Here we lower the value of* d_w *if vertex v offered a shorter path*
- $d_w = d_v + c_{v,w}$ *if the new value* d_w *is an improvement*

DecreaseKey: O(|E| log |V|)

Total running time: O(|E| log |V|)

Dijkstra's Adjacency List



v	*known*	d_v	p_v
v_1	F	0	0
v_2	F	∞	0
v_3	F	∞	0
v_4	F	∞	0
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	∞	0
v_4	F	1	v_1
v_5	F	∞	0
v_6	F	∞	0
v_7	F	∞	0

v	*known*	d_v	p_v
v_1	T	0	0
v_2	F	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

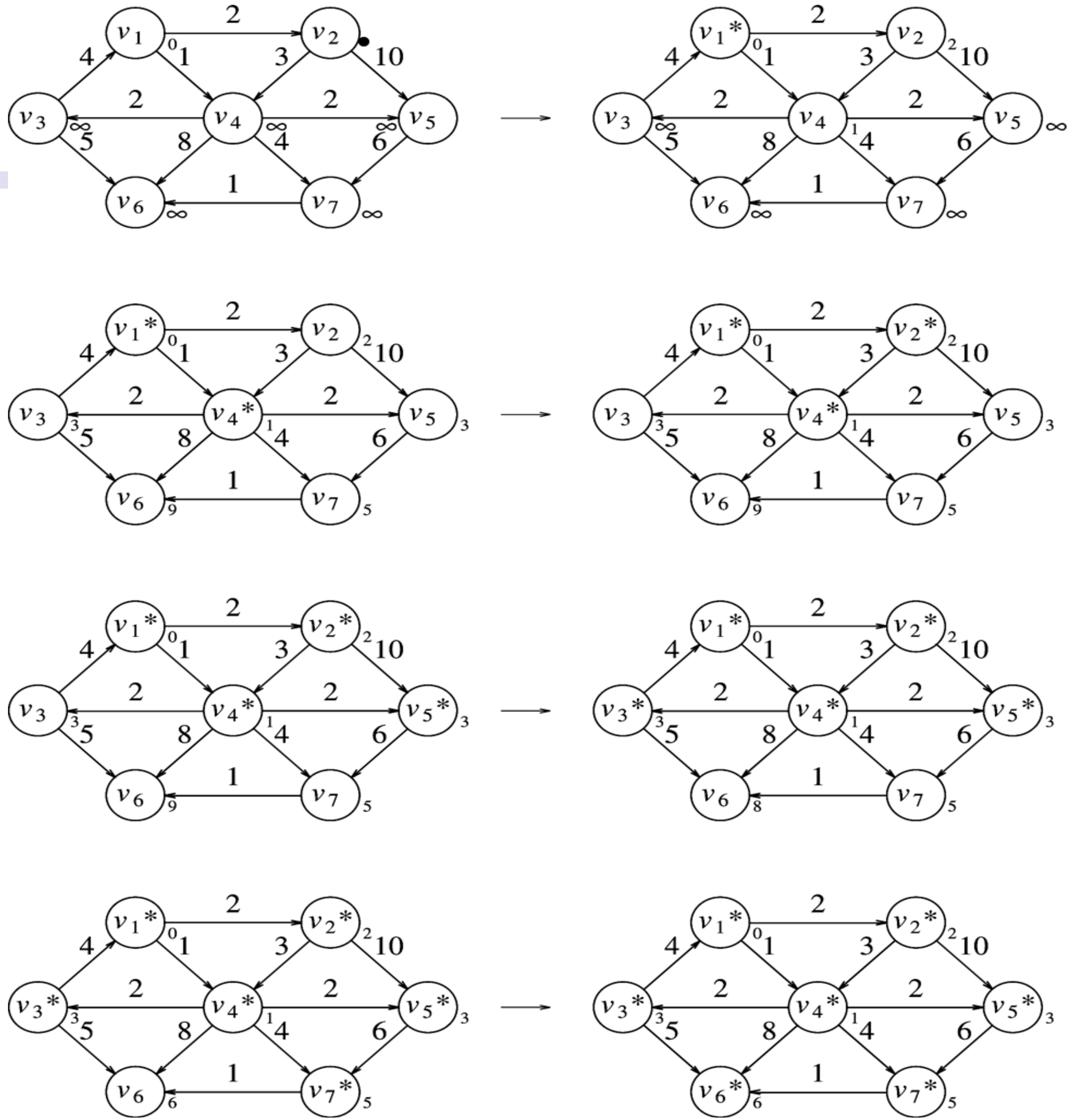
v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	F	3	v_4
v_4	T	1	v_1
v_5	F	3	v_4
v_6	F	9	v_4
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	8	v_3
v_7	F	5	v_4

v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	F	6	v_7
v_7	T	5	v_4

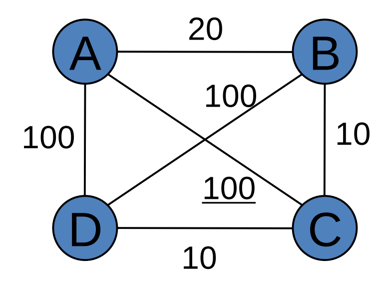
v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

Dijkstra's Algorithm

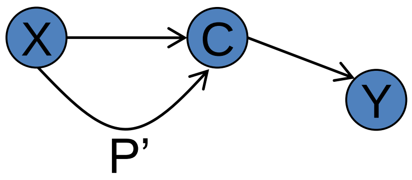


Why Dijkstra Works

- 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



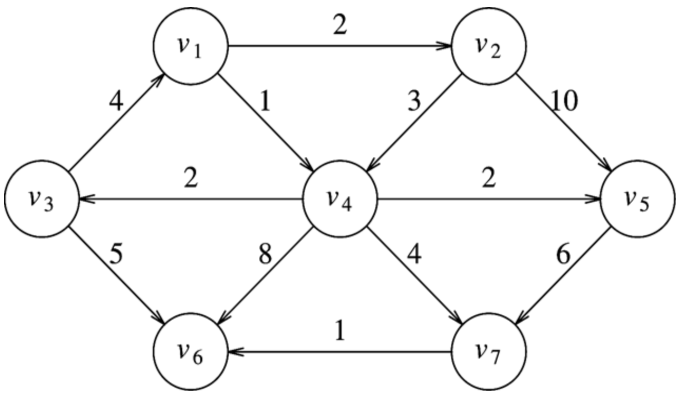
Why Dijkstra Works



- Assume hypothesis is false
  - i.e., Given a least-cost path P from X to Y that go 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;
}
```



v	*known*	d_v	p_v
v_1	T	0	0
v_2	T	2	v_1
v_3	T	3	v_4
v_4	T	1	v_1
v_5	T	3	v_4
v_6	T	6	v_7
v_7	T	5	v_4

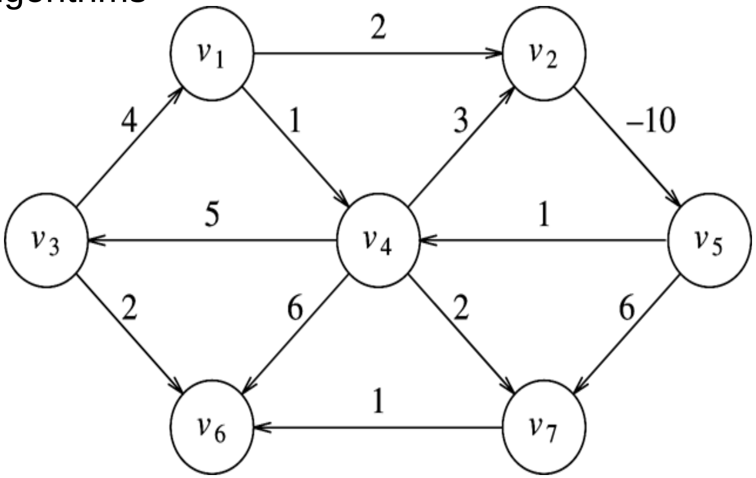
Negative Edge Costs but No Cycles

```
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 );
            }
    }
}
```

// a bit can be set for each vertex to indicate presence in the queue

Running time: O(|E|·|V|)
Negative weight cycles?
Dijkstra's algorithm does not work

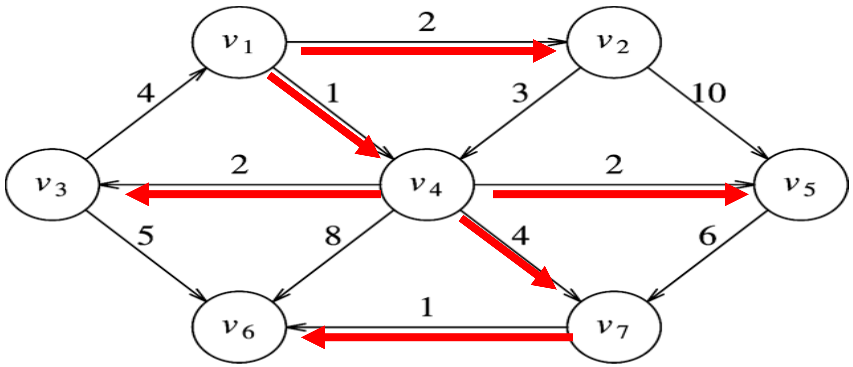
- Vertex `u` is known but there may be a path from unknown vertex `v` back to `u` that is very negative
- Add a constant value to each edge cost?
- Solve this with the combination of unweighted and weighted algorithms



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| x |V|)→ 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| x |V|)
- All-pairs shortest paths
  - Dijkstra: O(|V| x |E| log |V|) = O(|V|^3 log |V|)
  - Floyd-Warshall: O(|V|^3)