# Practical performance comparison of all-pairs shortest path algorithms

Andrej Brodnik
University of Primorska
University of Ljubljana

Marko Grgurovič
University of Primorska

# Problem

- Directed graphs.
- Non-negative edge lengths.
- Find shortest paths between every pair of vertices (APSP).

# Algorithms: Floyd-Warshall

- Standard dynamic programming formulation.

```
FOR k=1 to n
    FOR i=1 to n
        FOR j=1 to n
            W[i,j] = MIN(W[i,j], W[i,k]+W[k,j])
        ENDFOR
    ENDFOR
ENDFOR
```

# Algorithms: Floyd-Warshall

- Standard dynamic programming formulation.

```
FOR k=1 to n
    FOR i=1 to n
        IF (W[i,k] == ∞)
            continue;

        FOR j=1 to n
            W[i,j] = MIN(W[i,j], W[i,k]+W[k,j]);
        ENDFOR
    ENDFOR
ENDFOR
```

# Algorithms: Dijkstra

- A single-source algorithm.
- Visits vertices in increasing distance from source.
- Solves APSP as separate single-source problems.
- Use priority queues (PQ) for best result.

# Algorithms: Dijkstra

- Let a candidate (shortest) path be any path satisfying some condition *C*.
- Dijkstra-like algorithms will push candidate paths into a PQ and pop to retrieve the next shortest path.
- E.g. (Dijkstra) extend a path $\pi$ with *(u,v)* if:
  - $\pi$ is empty
  - $\pi$ is a known shortest path

# Algorithms: Hidden Paths

[Karger et al., '94]

- Modifies Dijkstra to solve APSP.
- Use a single large PQ and discover paths in increasing distance from *any* source.
- Key idea: extend a path $\pi$ with $(u,v)$ if:
  - $\pi$ is empty
  - $\pi$ and $\{(u,v)\}$ are known shortest paths.

# Algorithms: Hidden Paths

- Running time is $O(m^* n + n^2 \lg n)$
- $m^*$ is the number of *essential* edges.
  - Any non-essential edge can be removed from *G*, and the APSP solution will be the same.
  - $m^* = O(n \lg n)$ *in expectation and whp* in complete graphs with random weights. [Hassin & Zemel, ’85]

# Algorithms: Uniform Paths
[Demetrescu et al., '04]

- Very similar to Hidden Paths.
- Stricter condition: extend a path $\pi$ with *(u,v)* if:
  - $\pi$ is empty
  - Every proper subpath of $\pi$ + *(u,v)* is a shortest path.
- |UP| = number of paths whose proper subpaths are shortest paths.
- Runs in $O(|UP| + n^2 \lg n)$
- |UP| = $O(n^2)$ *in expectation and whp* in complete graphs with random weights. [Peres et al., '10]

# Algorithms: Propagation

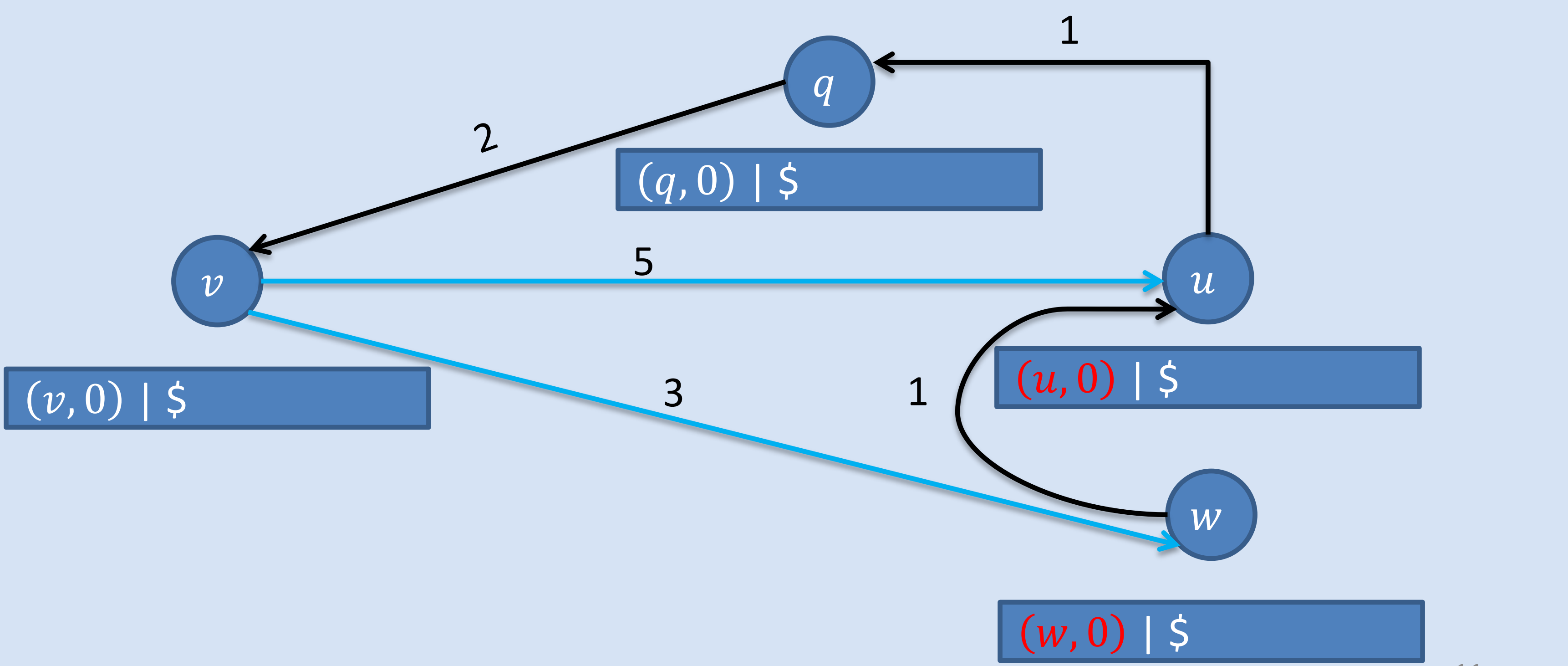
[Brodnik & G., ’12]

- General idea: each vertex is allowed to examine the (sorted by distance) shortest path lists of its neighbors, but nothing else!

- At each step of the algorithm, one shortest path for each vertex is discovered (in increasing distance from source).

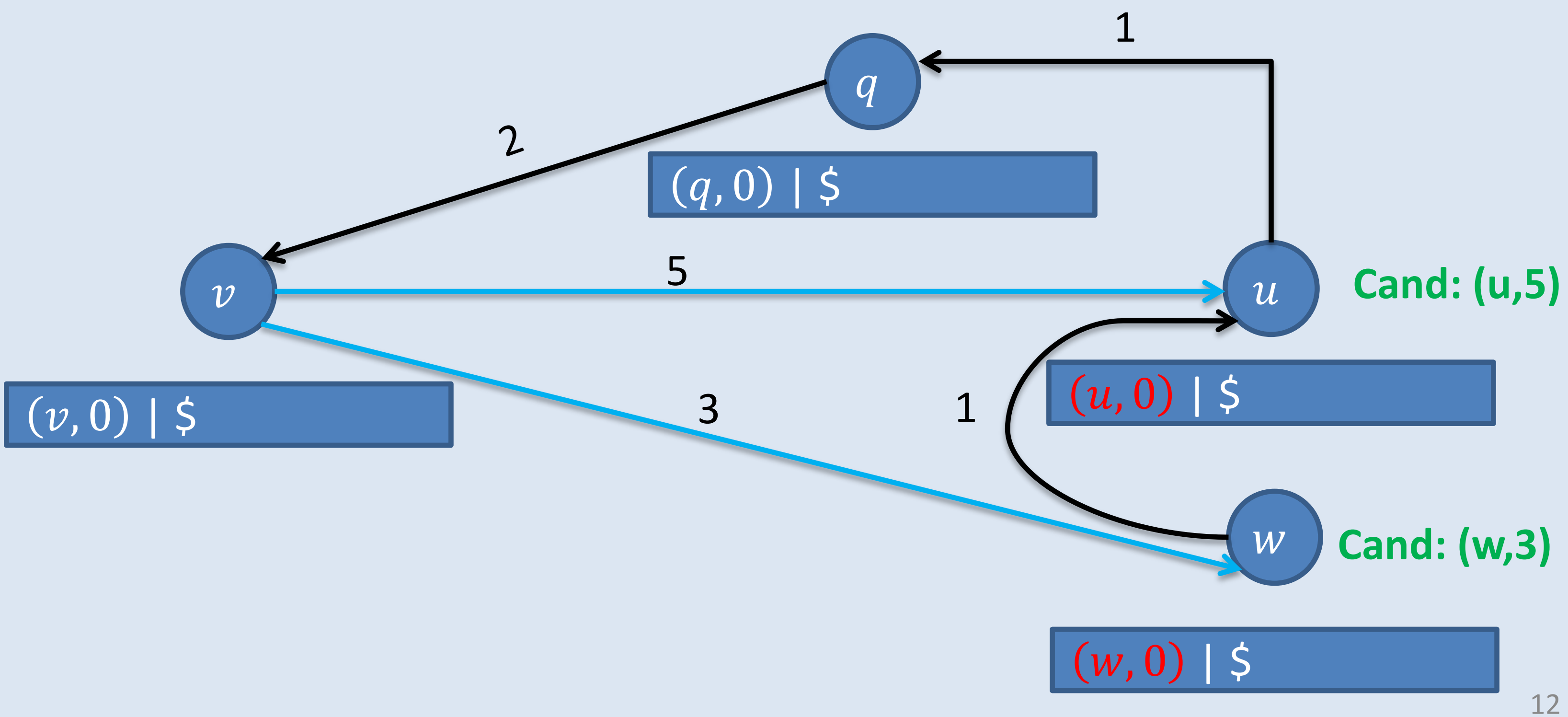
# Algorithms: Propagation

- Consider vertex $v$.
- Red = pointers (blue = out. edges for $v$)
- A pointer is moved right if the element pointed to is *not viable* (shorter path known).



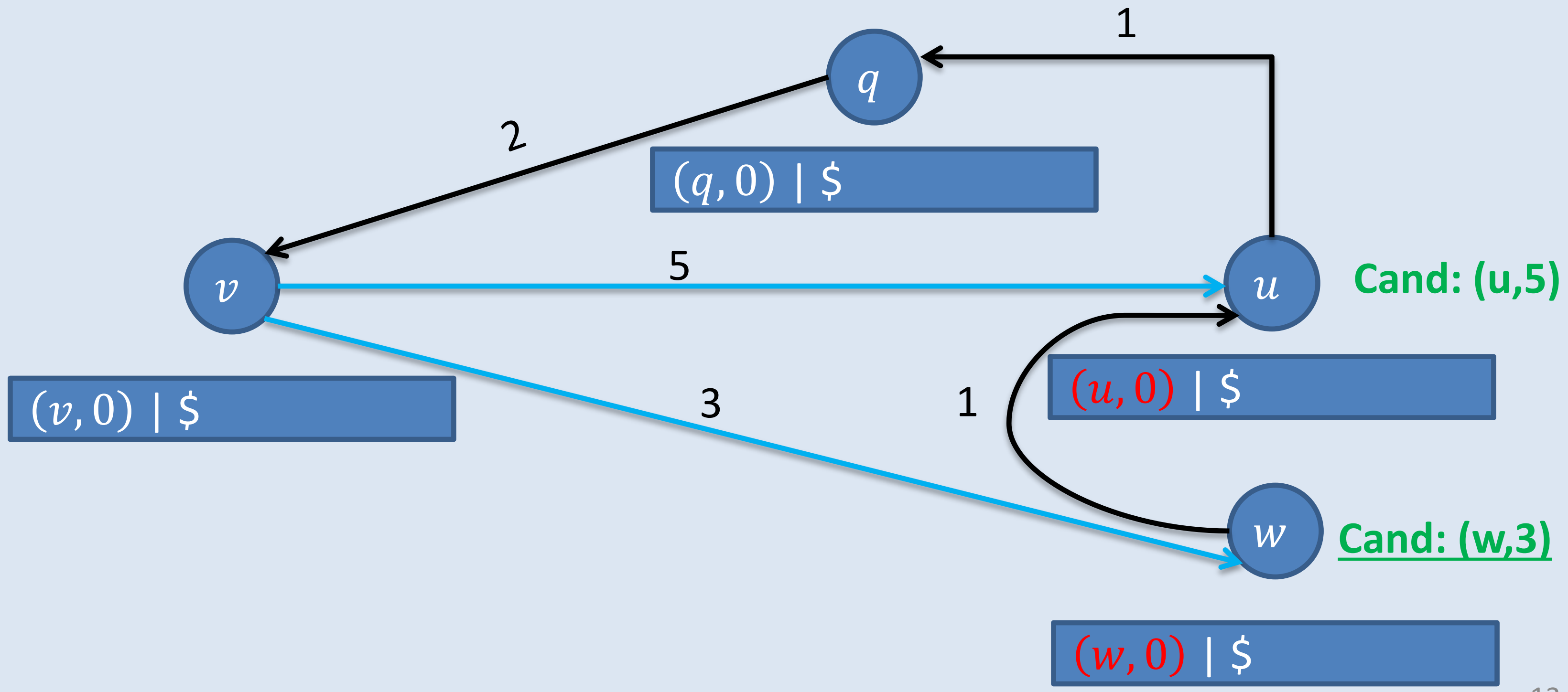
# Algorithms: Propagation

- i=1, running



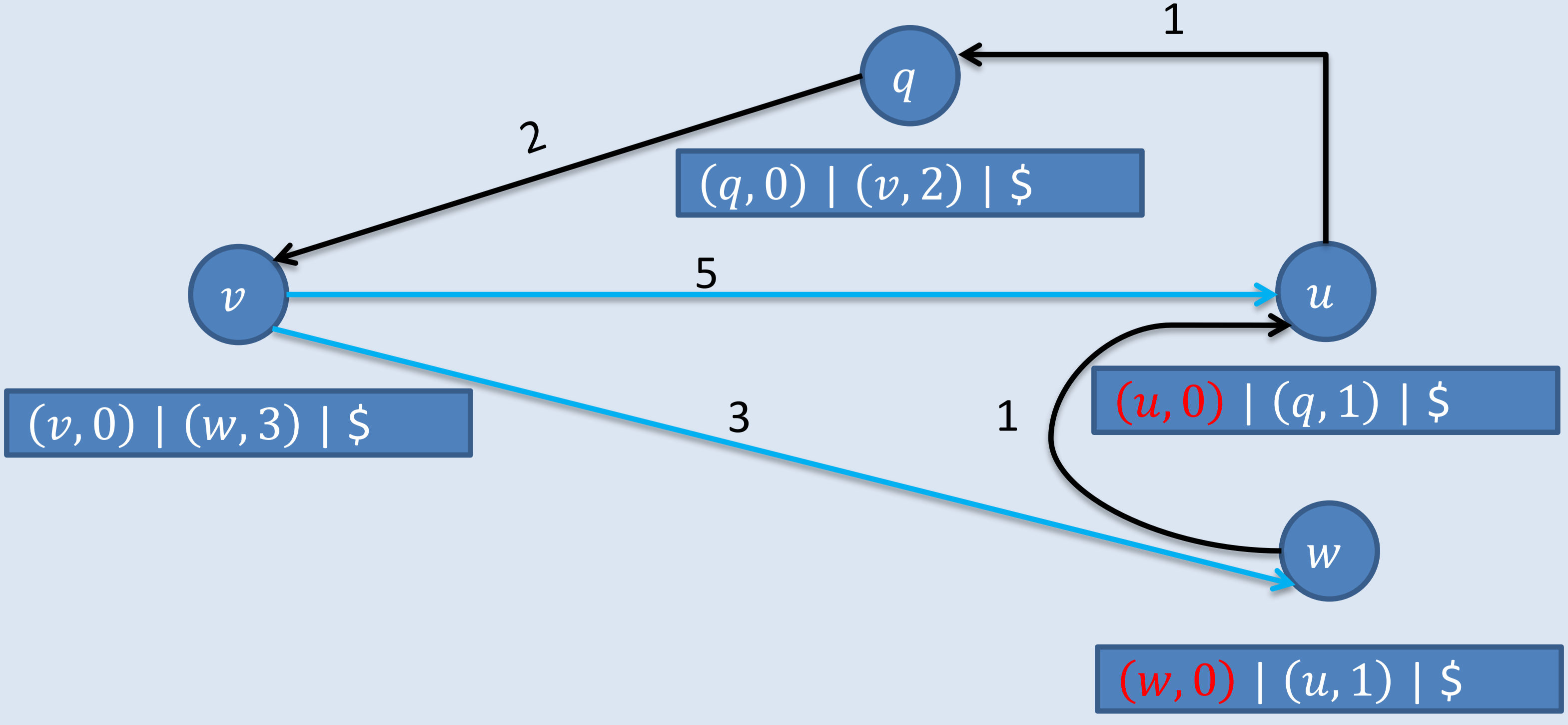
# Algorithms: Propagation

- i=1, running



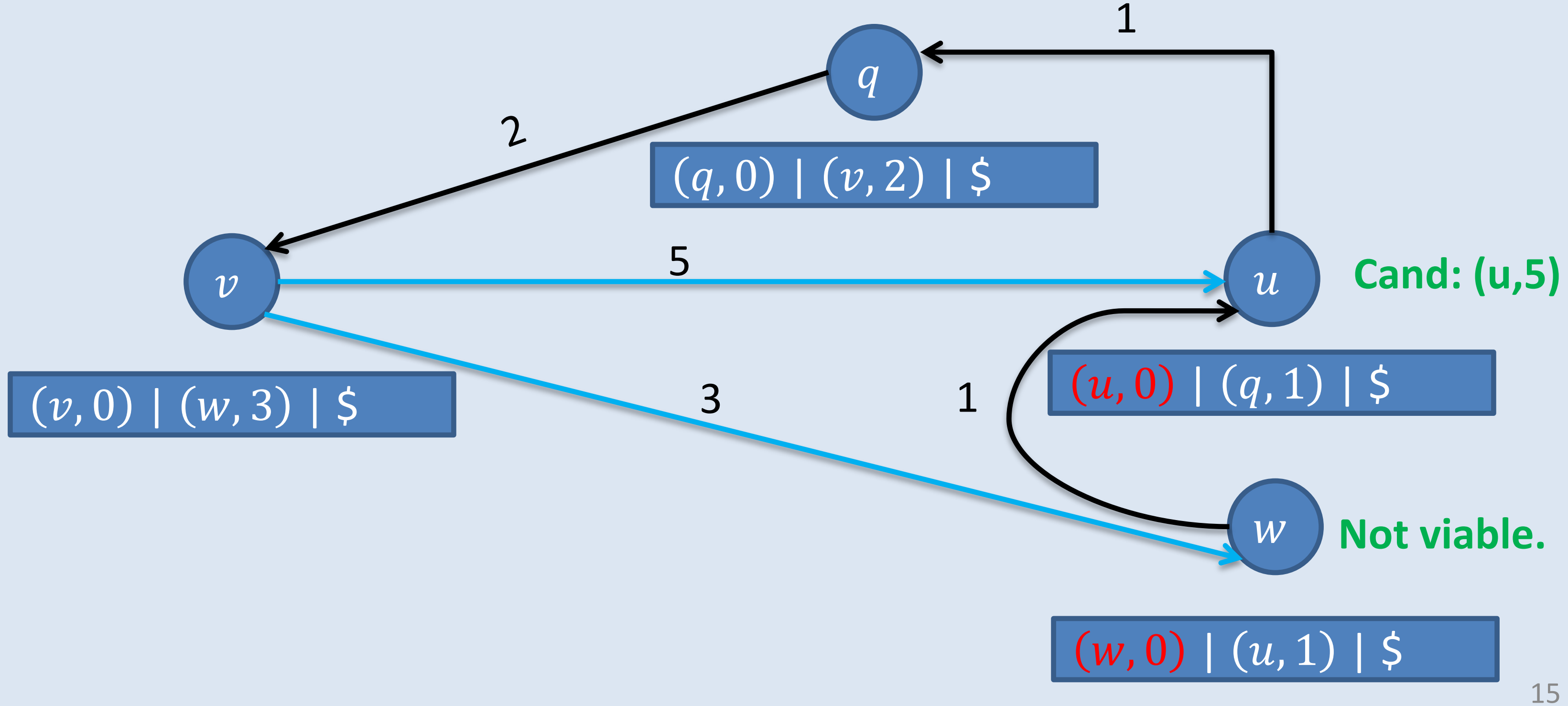
# Algorithms: Propagation

- i=1, finished



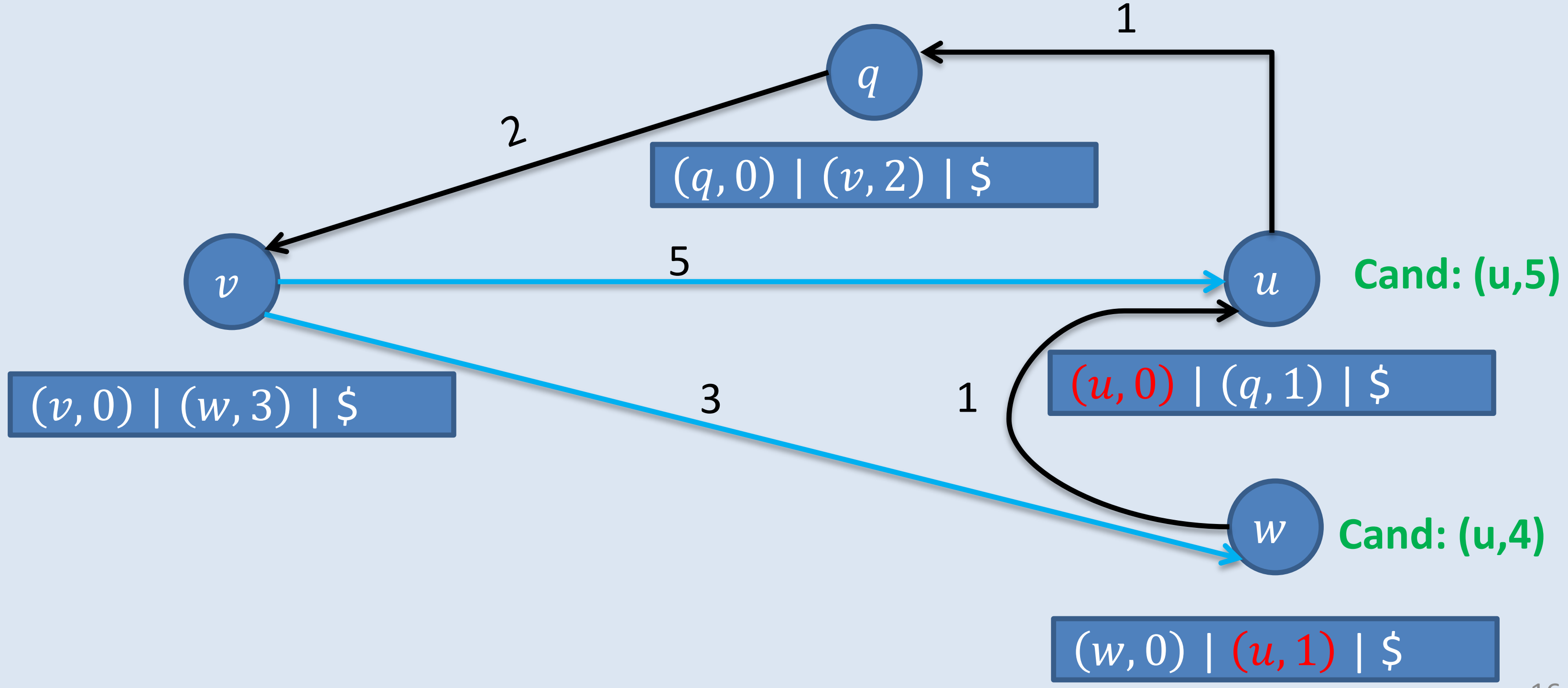
# Algorithms: Propagation

- i=2, running



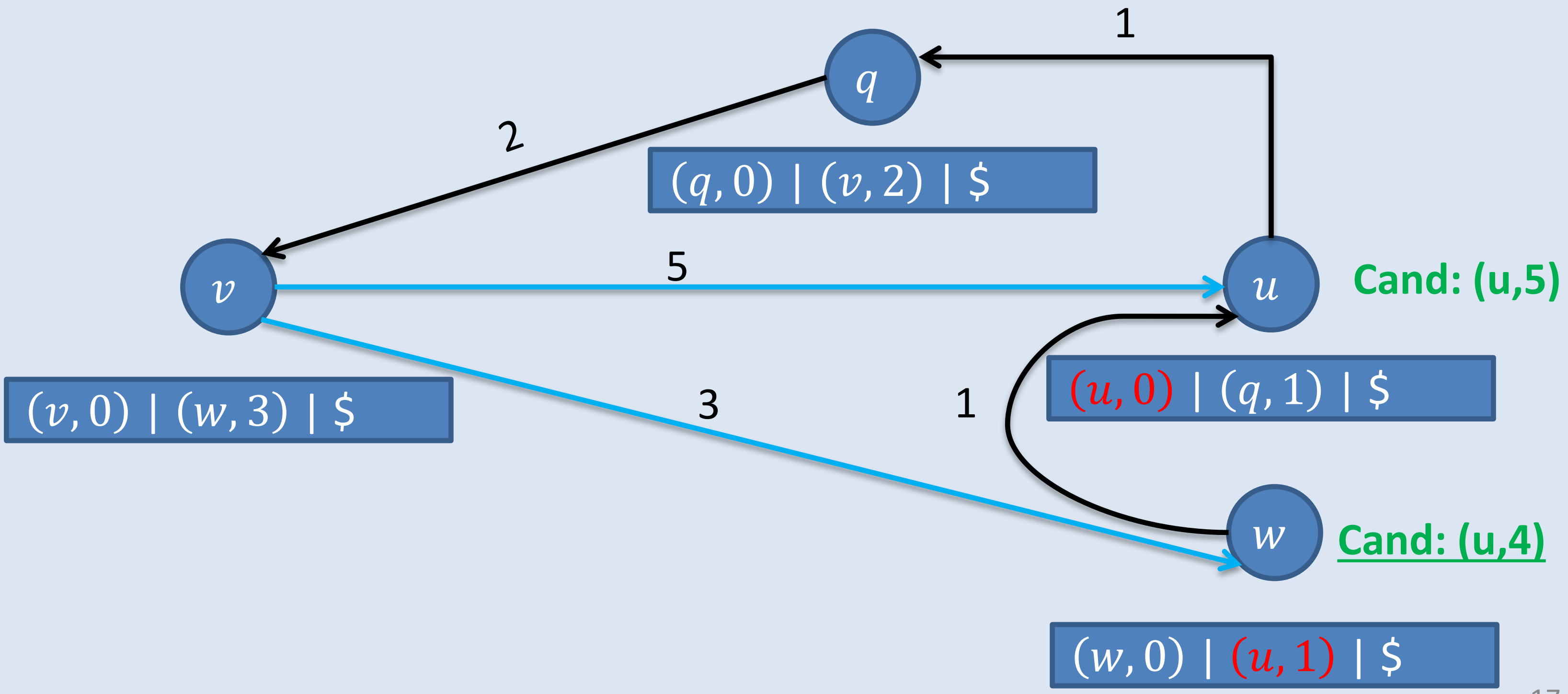
# Algorithms: Propagation

- i=2, running



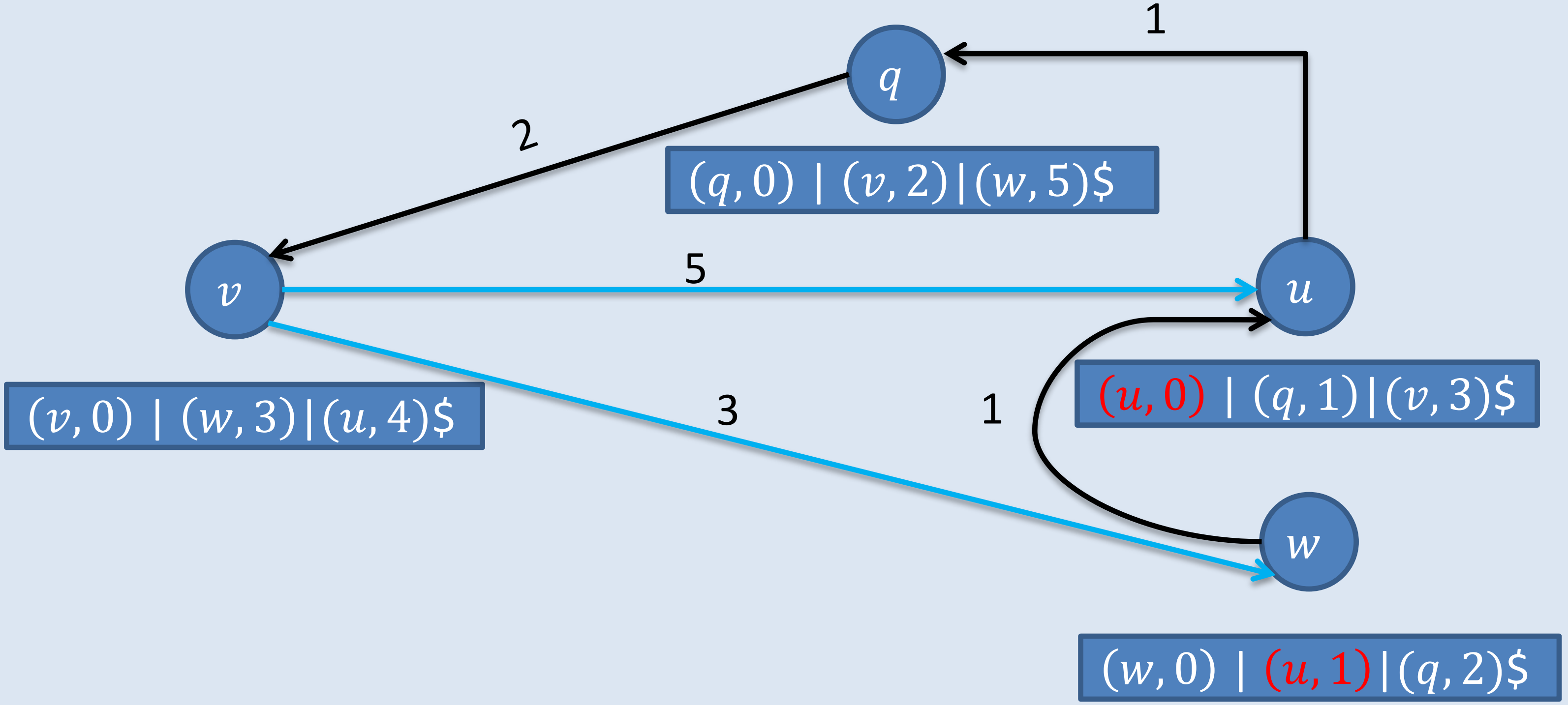
# Algorithms: Propagation

- i=2, running



# Algorithms: Propagation

- i=2, finished

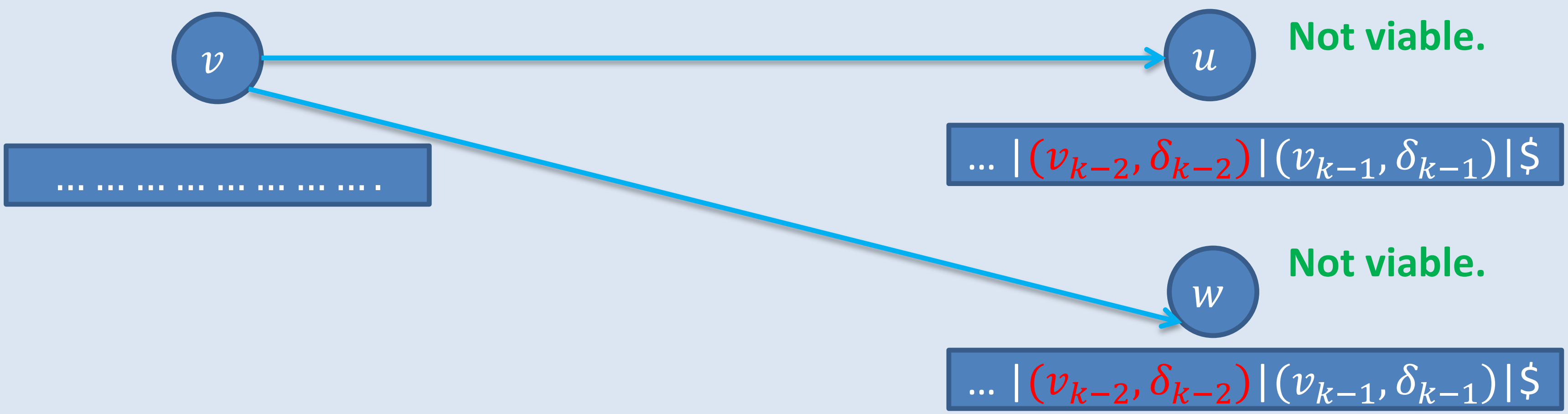


# Algorithms: Propagation

- What about when the *k*-th shortest path depends on a neighbors *k*-th shortest path?

# Algorithms: Propagation

- What about when the *k*-th shortest path depends on a neighbors *k*-th shortest path?

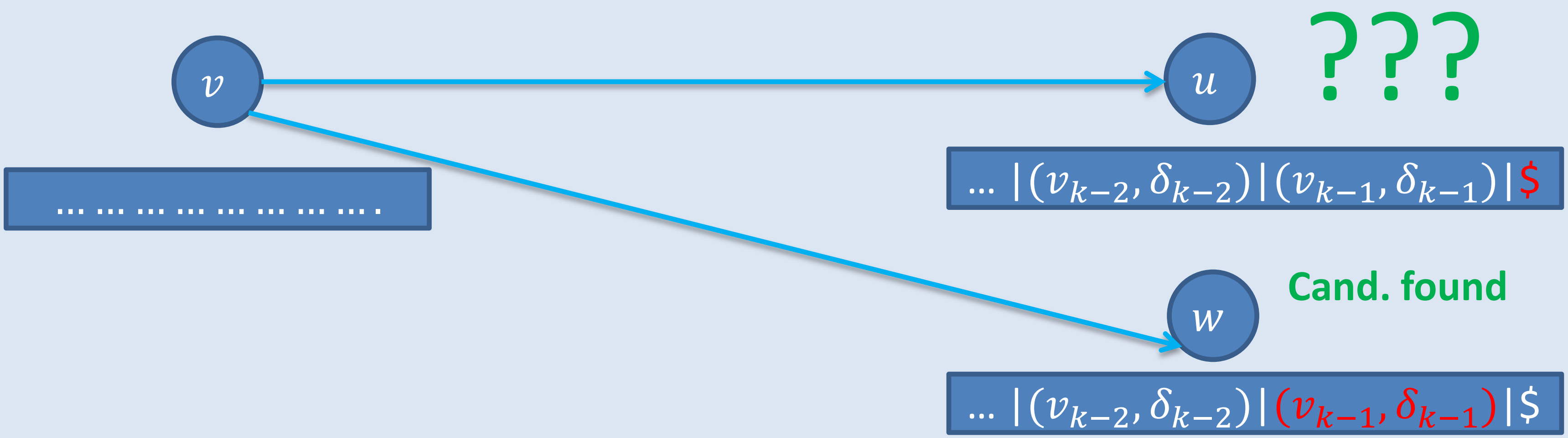


# Algorithms: Propagation

- What about when the *k*-th shortest path depends on a neighbors *k*-th shortest path?

# Algorithms: Propagation

- What about when the *k*-th shortest path depends on a neighbors *k*-th shortest path?

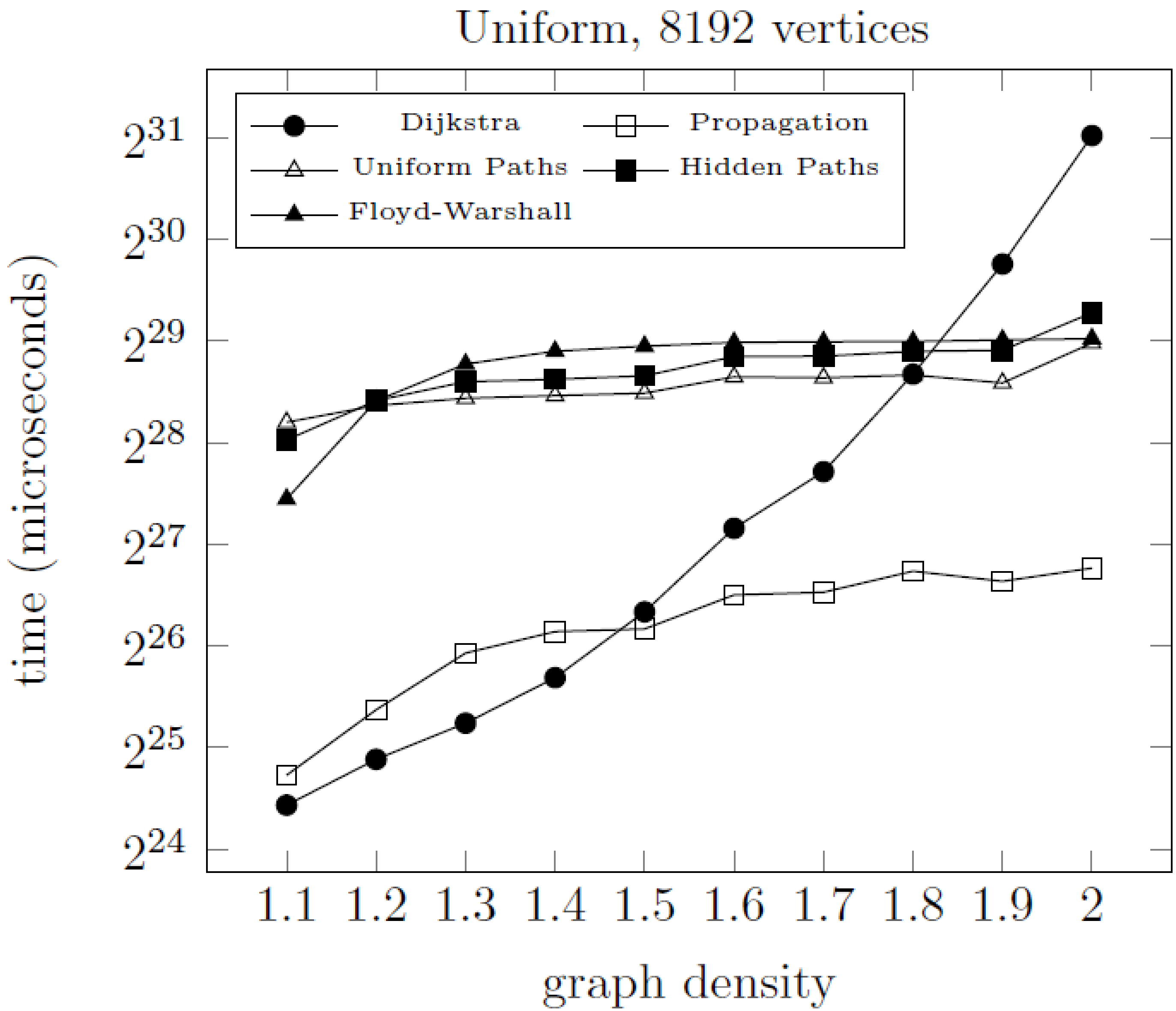


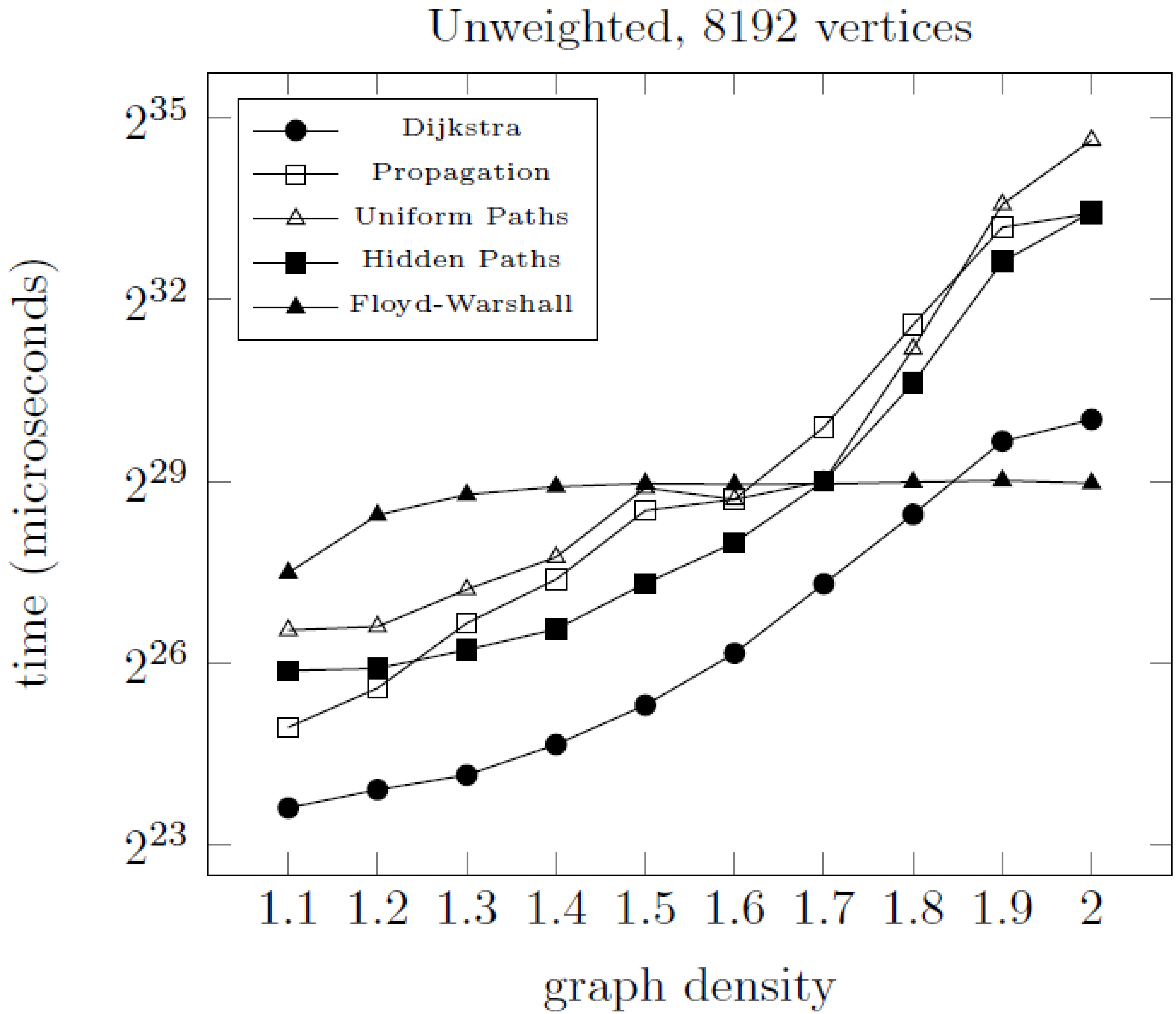
# Algorithms: Propagation

- Require (any) single-source algorithm to be provided.
- Use provided algorithm to solve the tricky cases when they occur via reduction (on a pruned graph).
- Overall, algorithm runs in $O(nT_s(m^*, n) + m \lg n)$ where $T_s(m, n)$ is the running time of the provided alg.

# Experiments

- Implemented in C++.
- Ran on an i7-3930K@3.20GHz with 64GB RAM on Ubuntu 12.04.5 LTS.
- Using pairing heaps for PQ from the Boost library.
- Generate random graphs from 512-16k vertices with varying densities and consider:
  - Unweighted
  - Uniform random weights in (0,1)


Uniform, 8192 vertices
Dijkstra
Propagation
Uniform Paths
Hidden Paths
Floyd-Warshall
time (microseconds)
$2^{31}$
$2^{30}$
$2^{29}$
$2^{28}$
$2^{27}$
$2^{26}$
$2^{25}$
$2^{24}$
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
graph density



Unweighted, 8192 vertices
Dijkstra
Propagation
Uniform Paths
Hidden Paths
Floyd-Warshall
time (microseconds)
$2^{35}$
$2^{32}$
$2^{29}$
$2^{26}$
$2^{23}$
1.1
1.2
1.3
1.4
1.5
1.6
1.7
1.8
1.9
2
graph density

Thanks for your attention!

# Questions