Speeding up Dynamic Shortest Path Algorithms

Luciana S. Buriol
Ph.D. Student, UNICAMP – Brazil
Visitor, AT&T Labs Research

Joint work with
Mauricio Resende and Mikkel Thorup
AT&T Labs Research

Outline

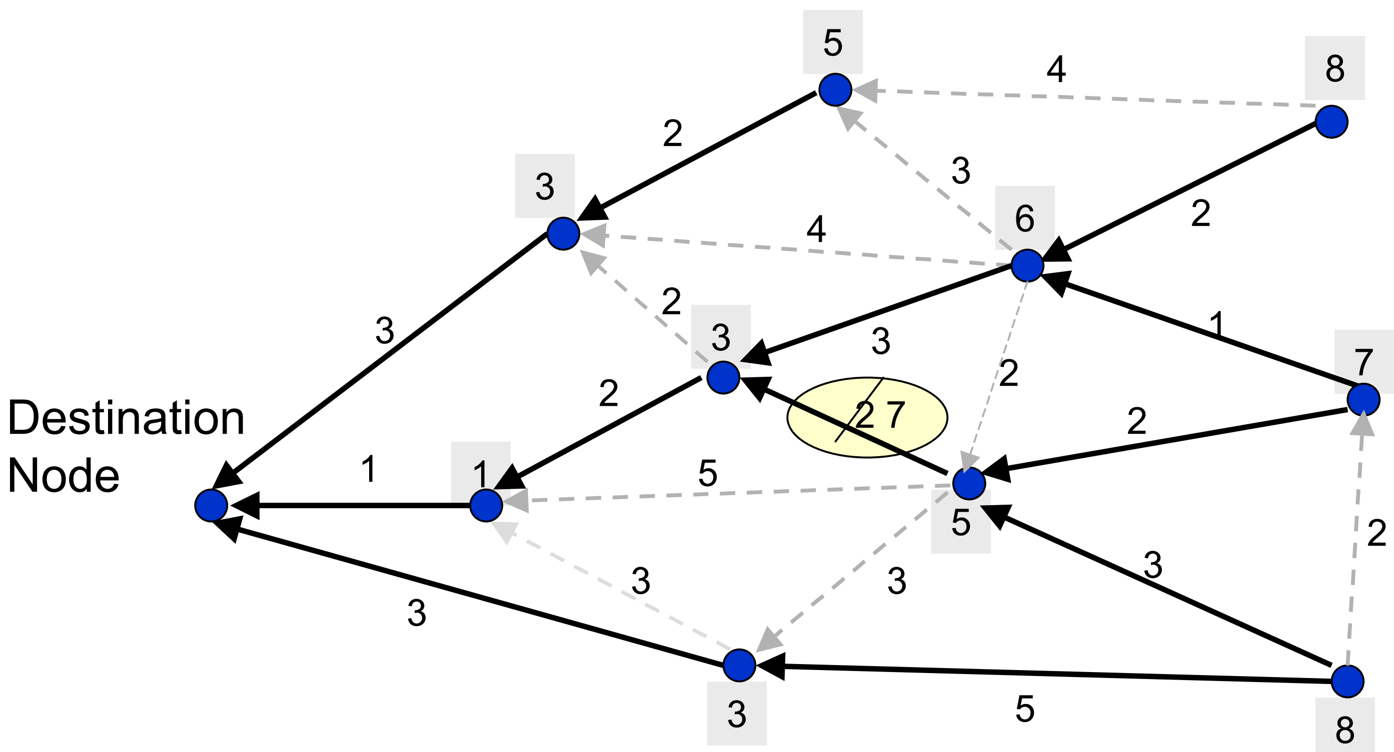
- Problem definition;
- Applications;
- Current algorithms:
- Using reduced heaps;
- Computational results;
- Conclusions.

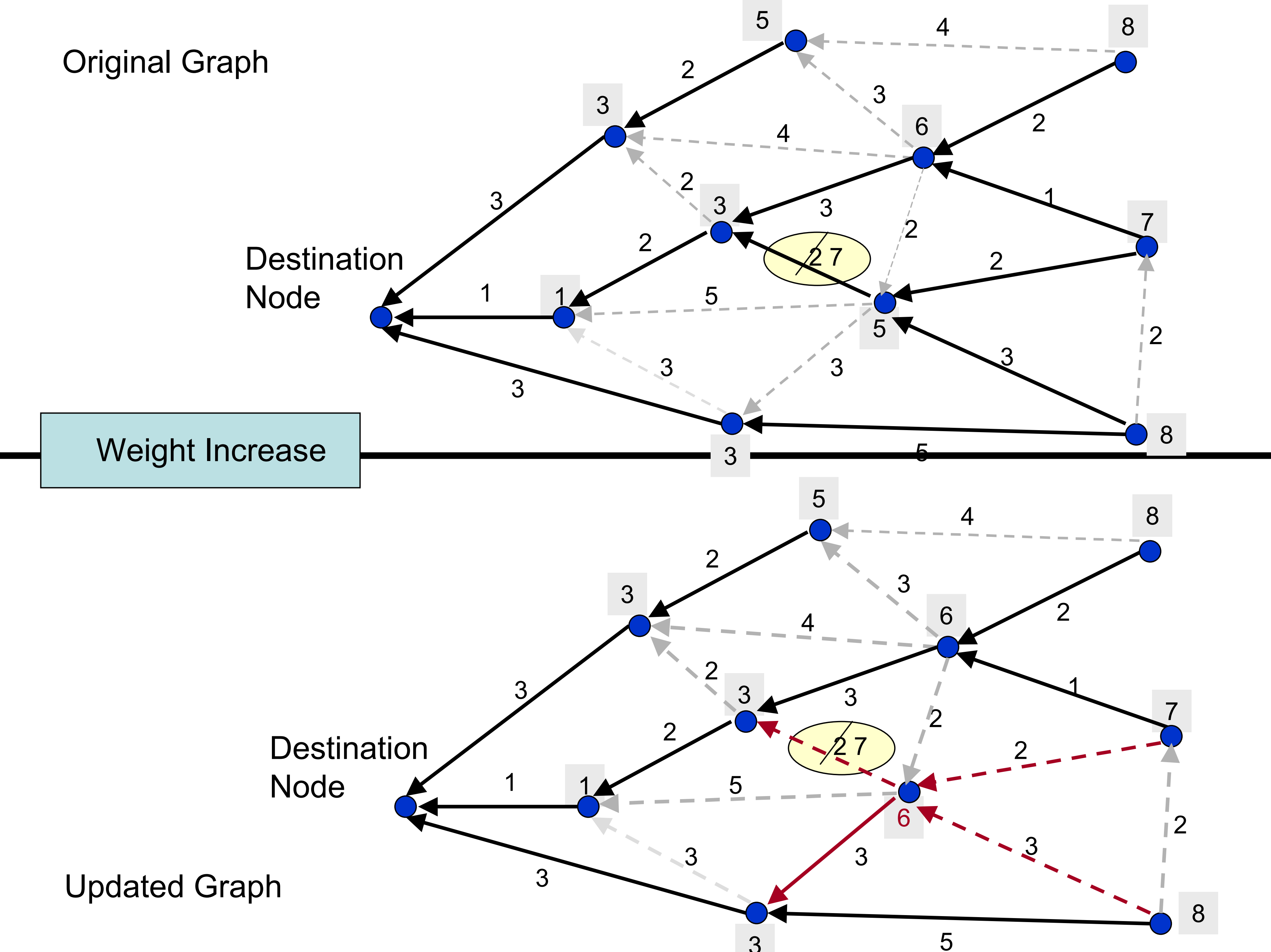
Objectives

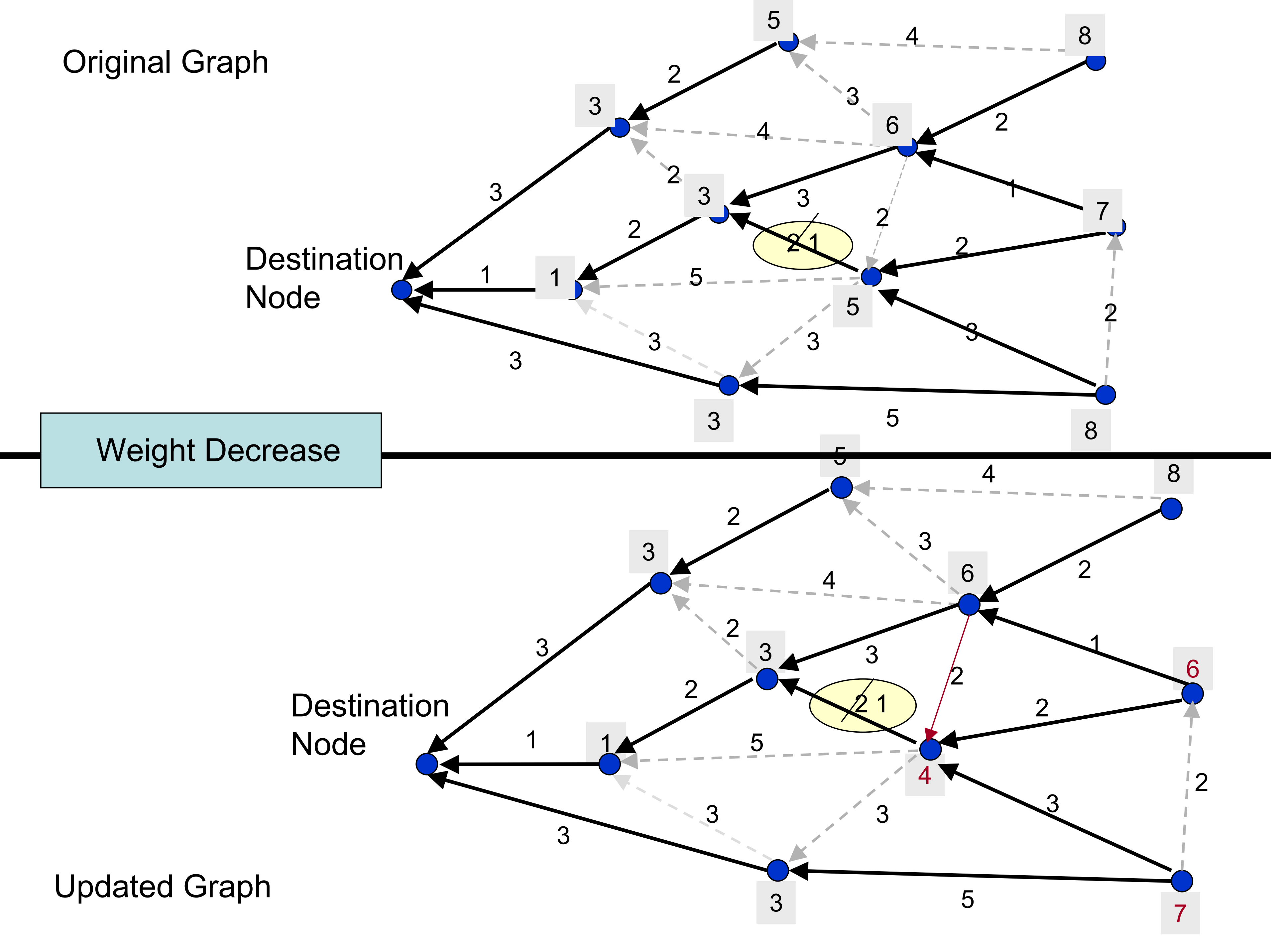
- To compare current dynamic shortest paths algorithms with respect to arc weight increase and decrease;
- To propose a new idea for reducing heap size to save computational time;

Dynamic Shortest Path problem

- Given a graph G = (V, E), a shortest path graph G_{SP} = (V, E'), and a vector W with a weight w_i associated with each link i. Update G_{SP} considering a weight change without recomputing it from scratch.




Original Graph
Destination Node
Weight Increase
Destination Node
Updated Graph


Original Graph
Destination Node
Weight Decrease
Destination Node
Updated Graph

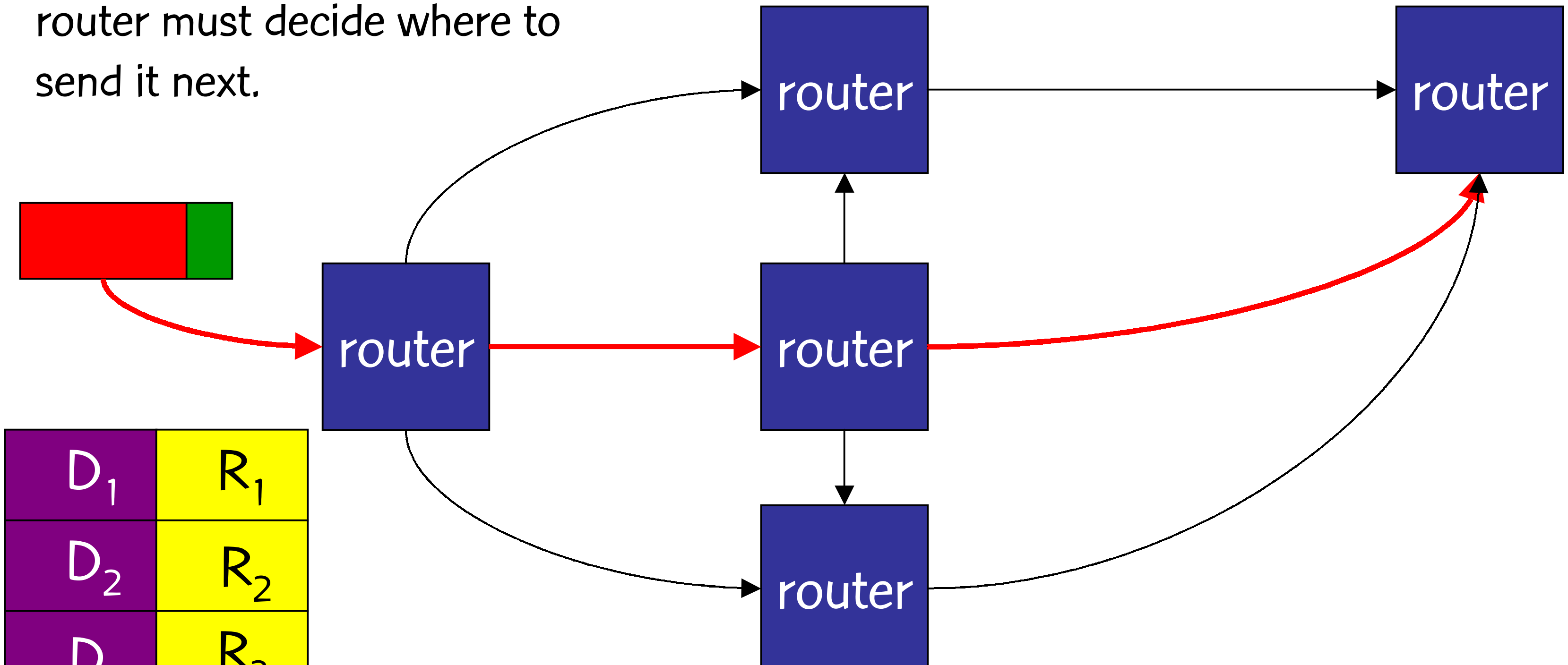
Applications

- Transportation network, when weights are associated with traffic/distance;
- Databases: maintaining distances between objects in a large data base;
- Data flow analysis and compilers;
- Document formatting;
- Local search procedure in packet routing.

Packet routing

When packet arrives at router, router must decide where to send it next.

Packet's final destination.



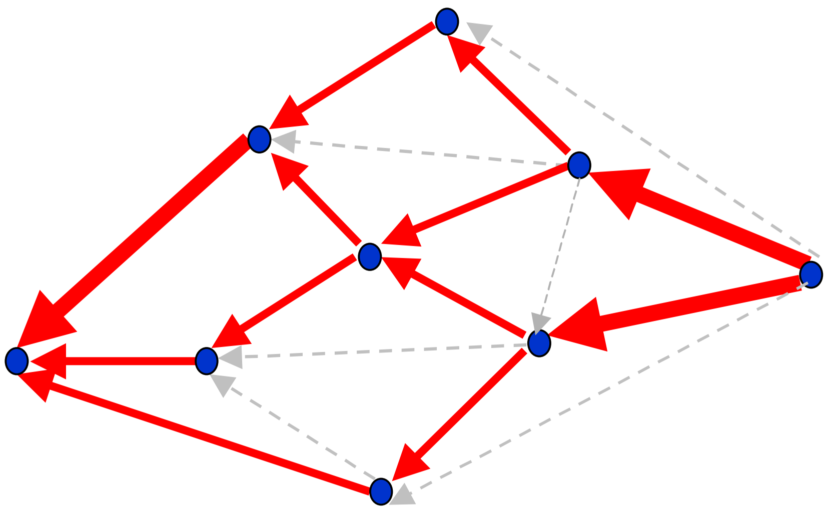
D_1	R_1
D_2	R_2
D_3	R_3
D_4	R_4

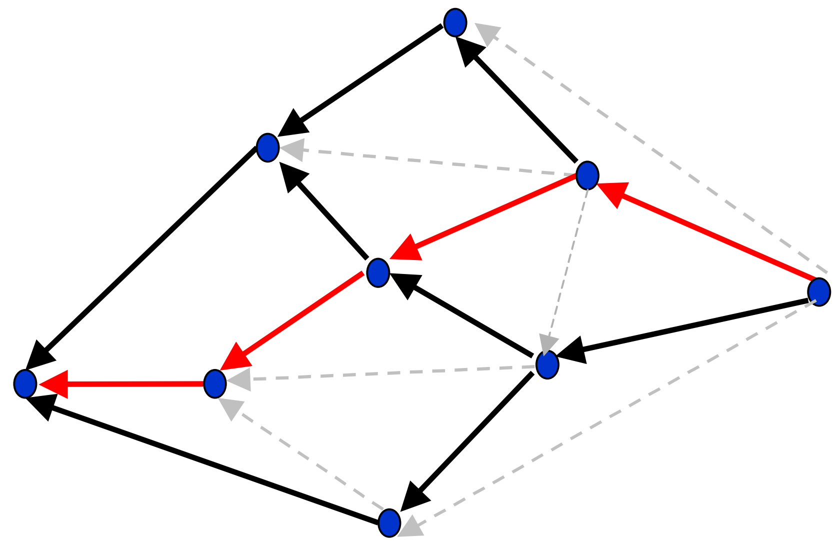
Routing table

Updating algorithms

- Specialized for weight increase and decrease;
- An arc deletion can be considered an increase of w_i to ∞;
- The shortest paths can be a tree or a graph, depending on the application;

Graph and tree representations





OSPF routing: Traffic flow is routed along shortest paths, splitting flow at nodes with more than one outgoing link.

Transportation: If the load cannot be split, only one shortest path is needed.

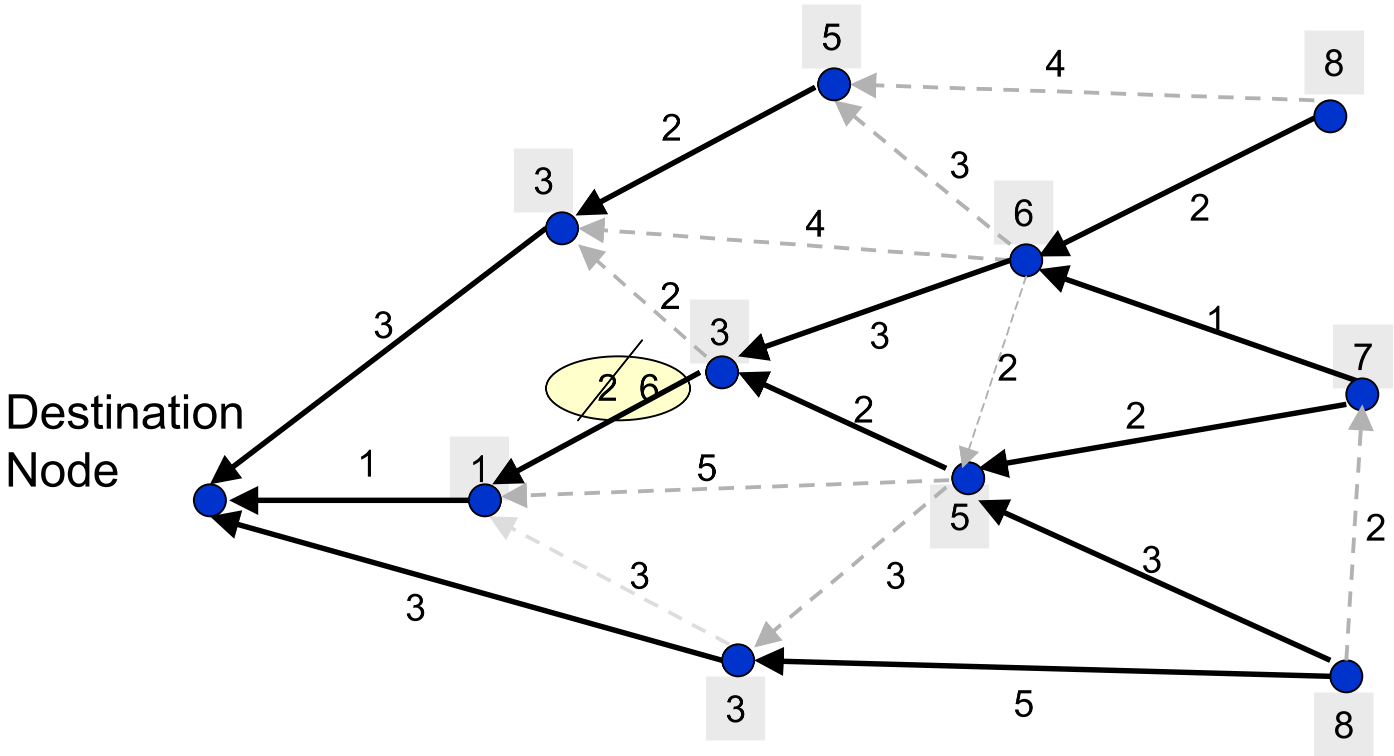
Algorithms

- Trees:
  - King & Thorup increase;
  - Demetrescu increase;
  - Frigioni et al. decrease;
- Graphs:
  - R&R increase;
  - R&R decrease;

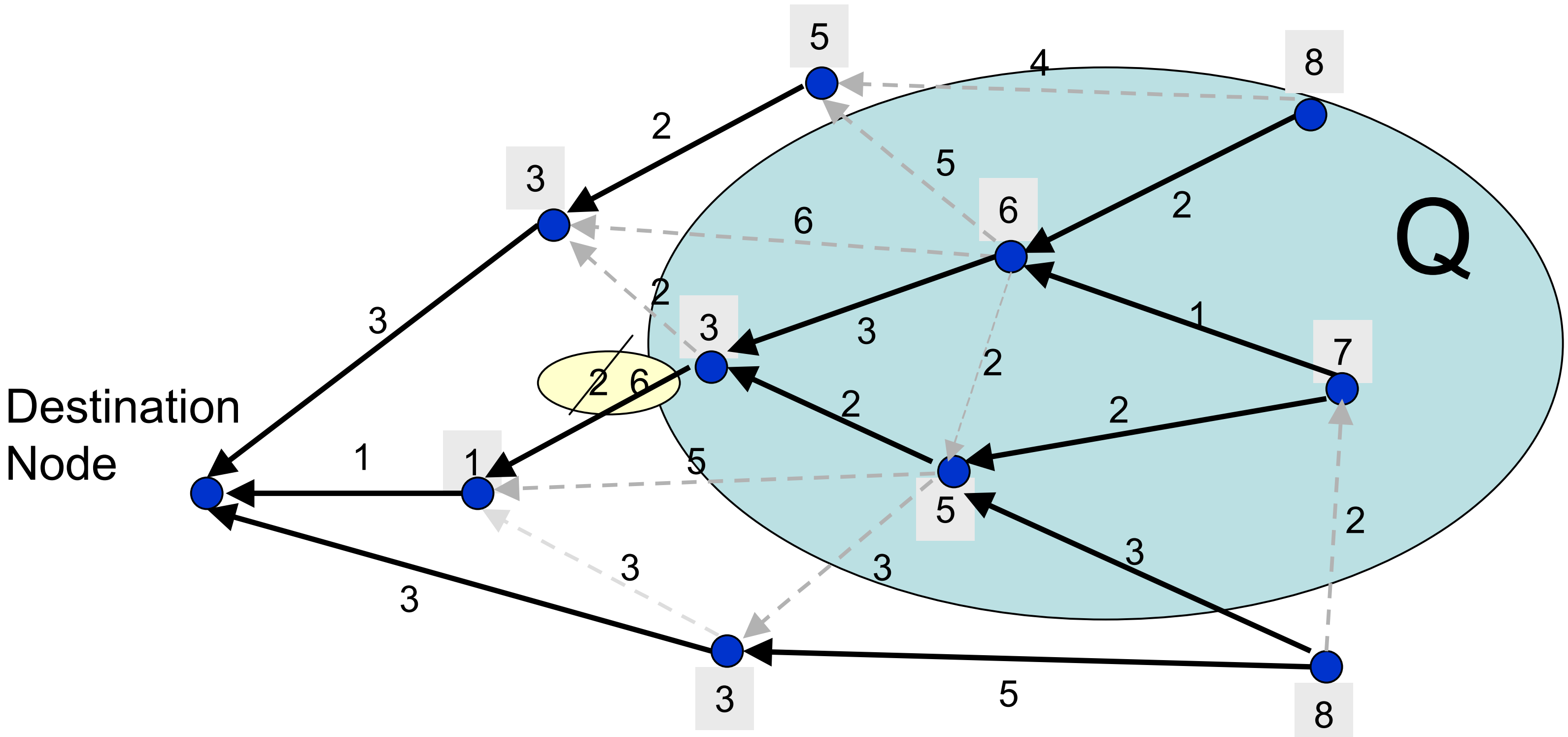
The above algorithms have two versions: the standard implementation and one avoiding use of heaps.

- Dijkstra's algorithm: recomputes shortest path graph from scratch only when at least one node distance changes. Otherwise, update the local change without Dijkstra.

Ramalingan & Reps arc weight increase

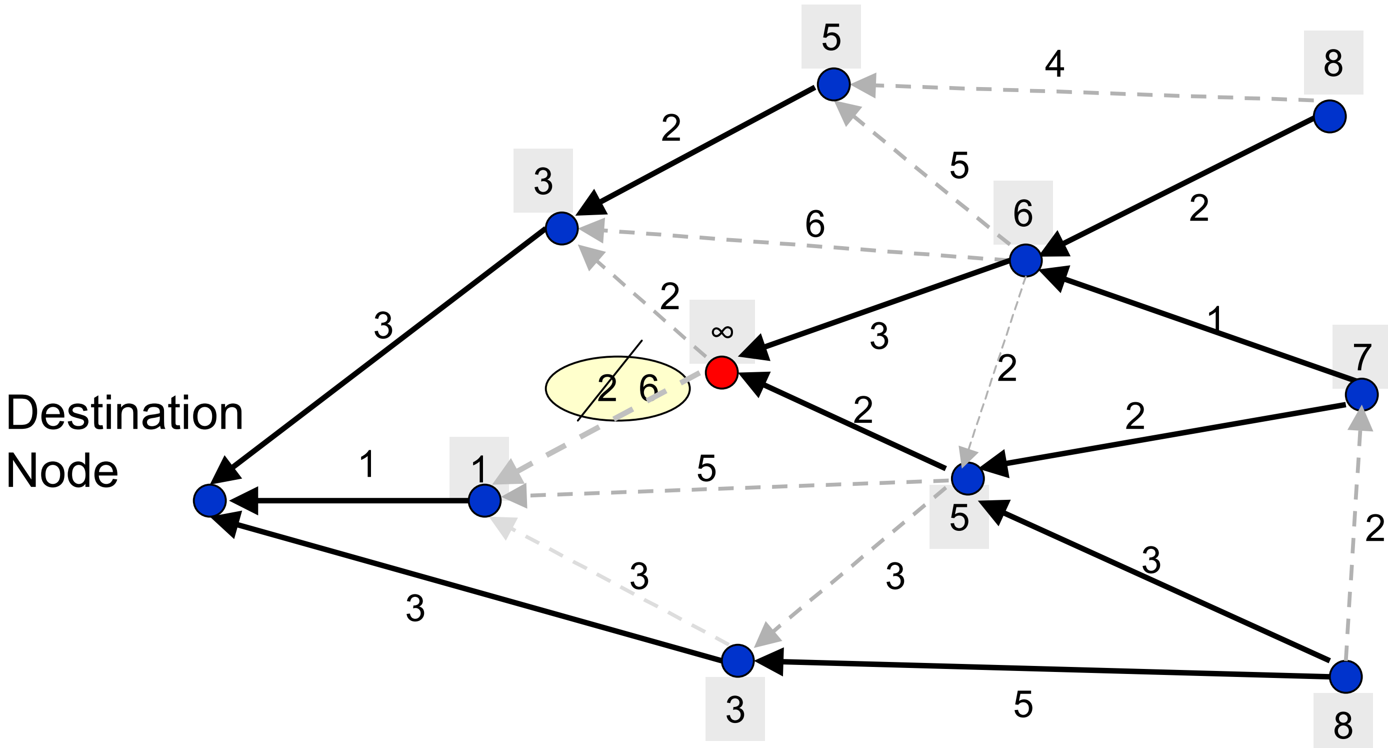


Ramalingan & Reps arc weigh increase



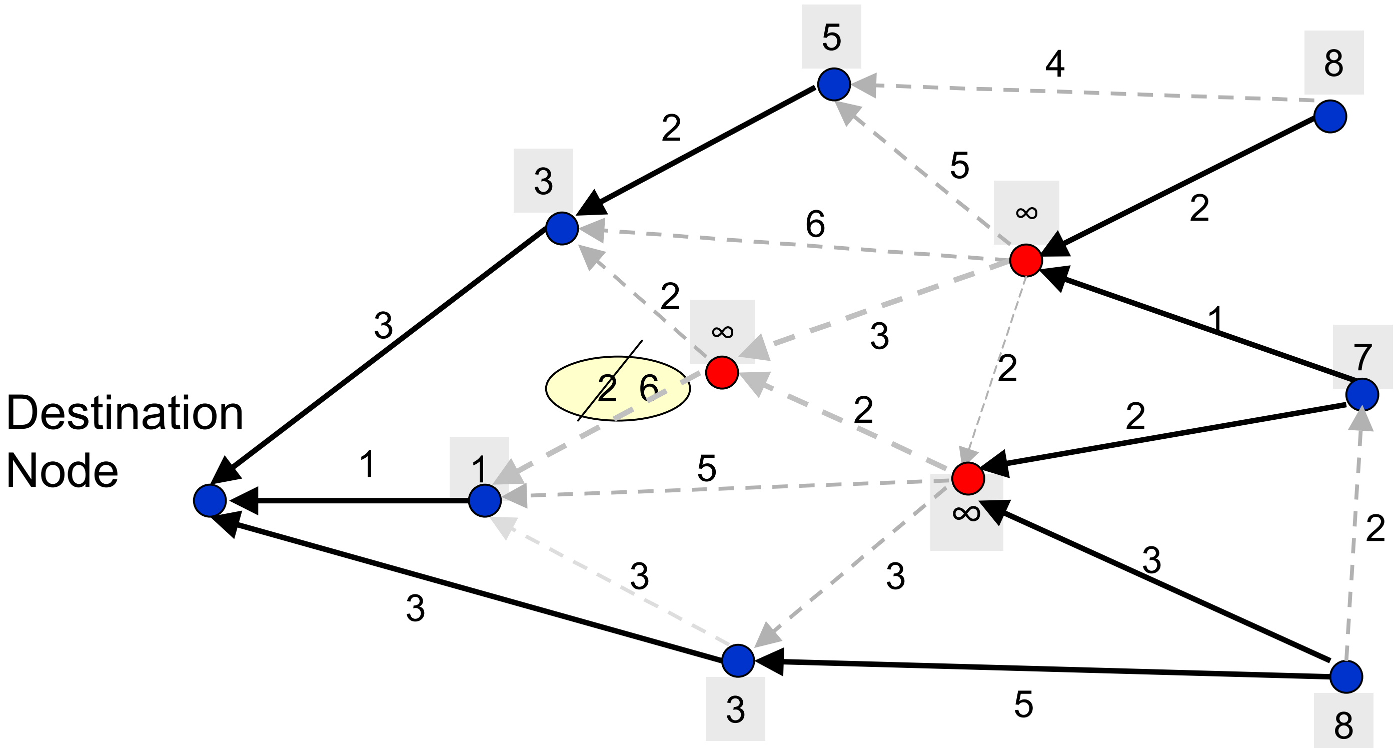
Find set Q of affected nodes. Q will contain all nodes which have all shortest paths traversing the changed arc.

Determining set Q



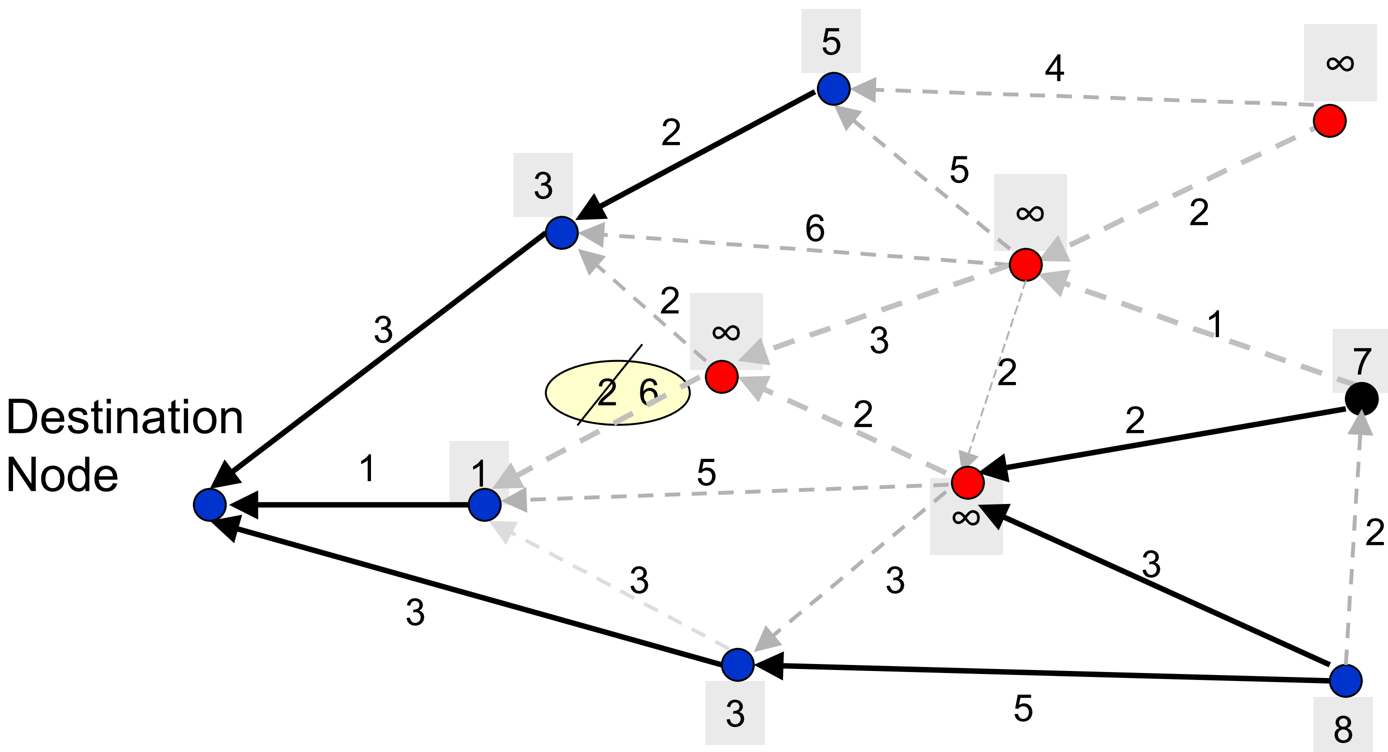
Find set Q of affected nodes. Set Q will contain all nodes which have all shortest paths crossing the changed arc.

Determining set Q



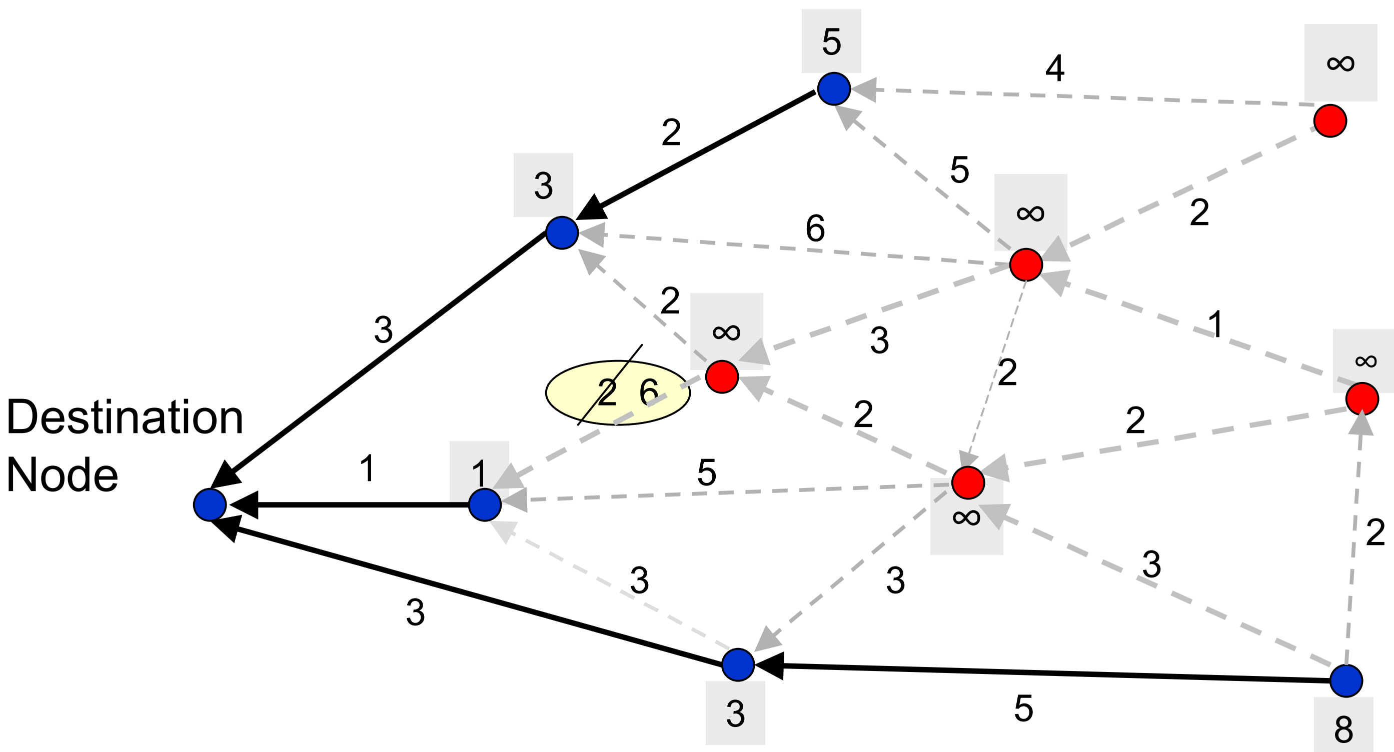
Find set Q of affected nodes. Set Q will contain all nodes which have all shortest paths crossing the changed arc.

Determining set Q



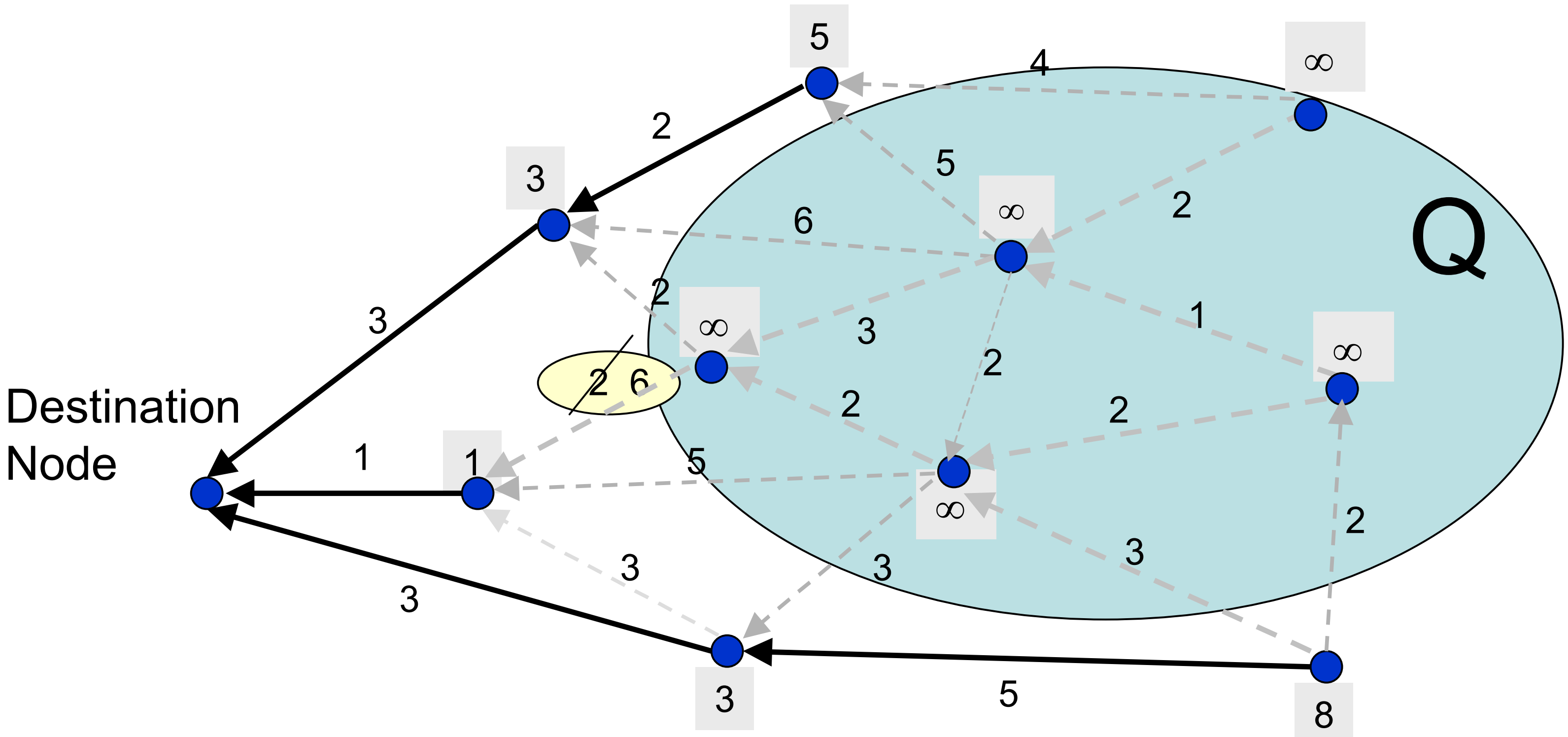
Find set Q of affected nodes. Set Q will contain all nodes which have all shortest paths crossing the changed arc.

Determining set Q



Find set Q of affected nodes. Set Q will contain all nodes which have all shortest paths crossing the changed arc.

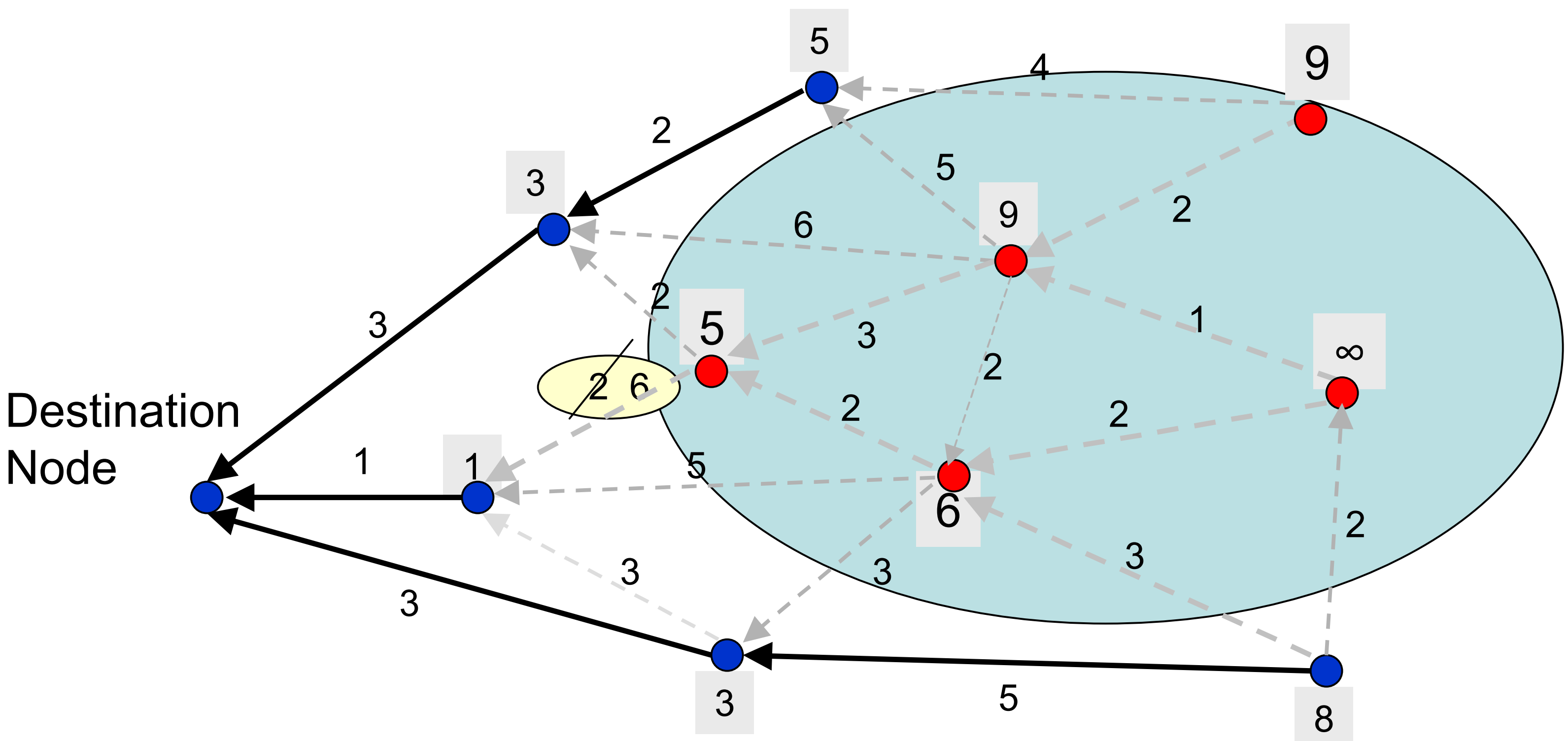
Determining set Q



All arcs $s \leftarrow u$ incoming into nodes $s \in$ Q are removed from G_{SP}. If u has no outgoing links in G_{SP}, u is an affected node. If u is an affected node, it is added to Q and $dist_u = \infty$.

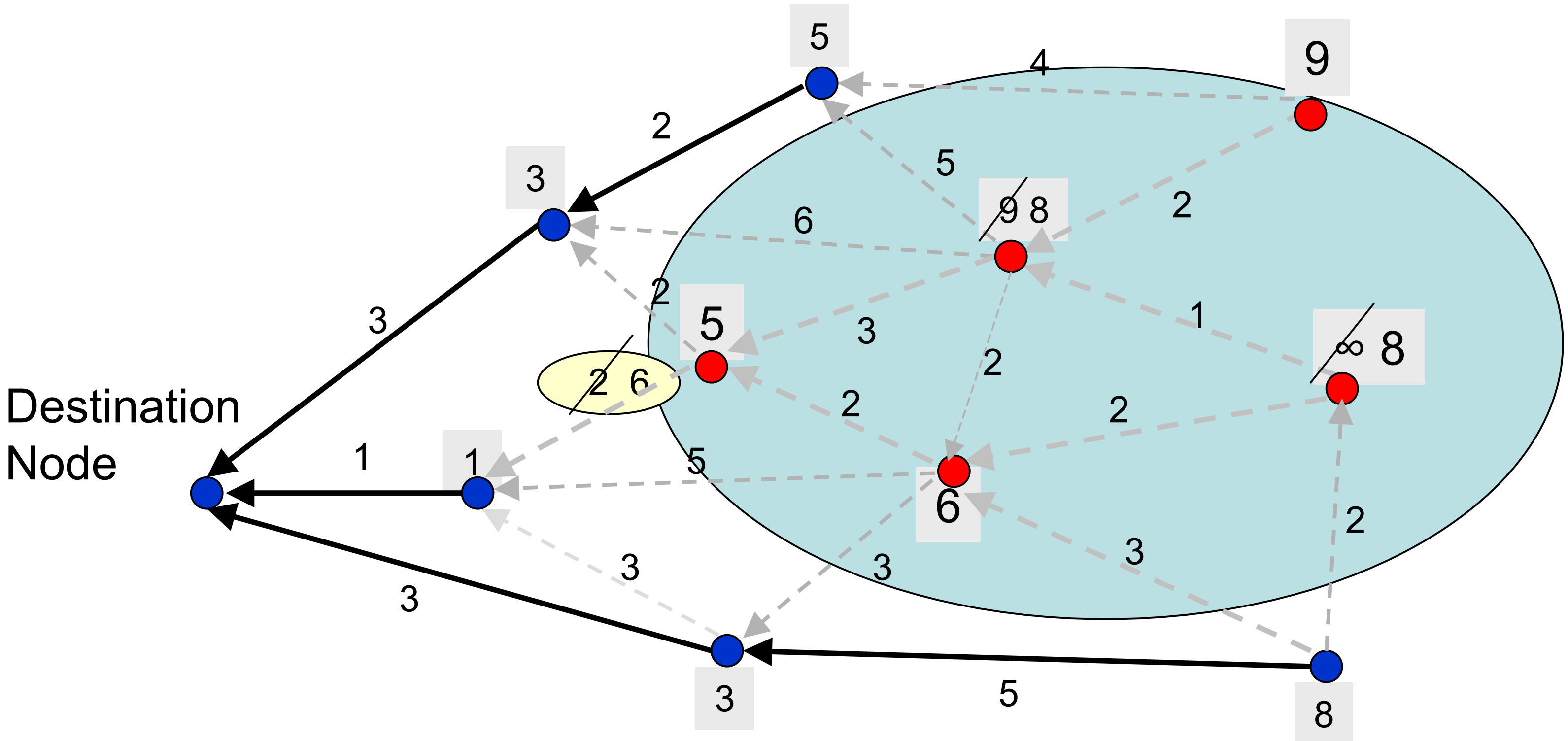
Updating Q-node distances

Update distances to nodes in Q considering arcs linking nodes outside Q.



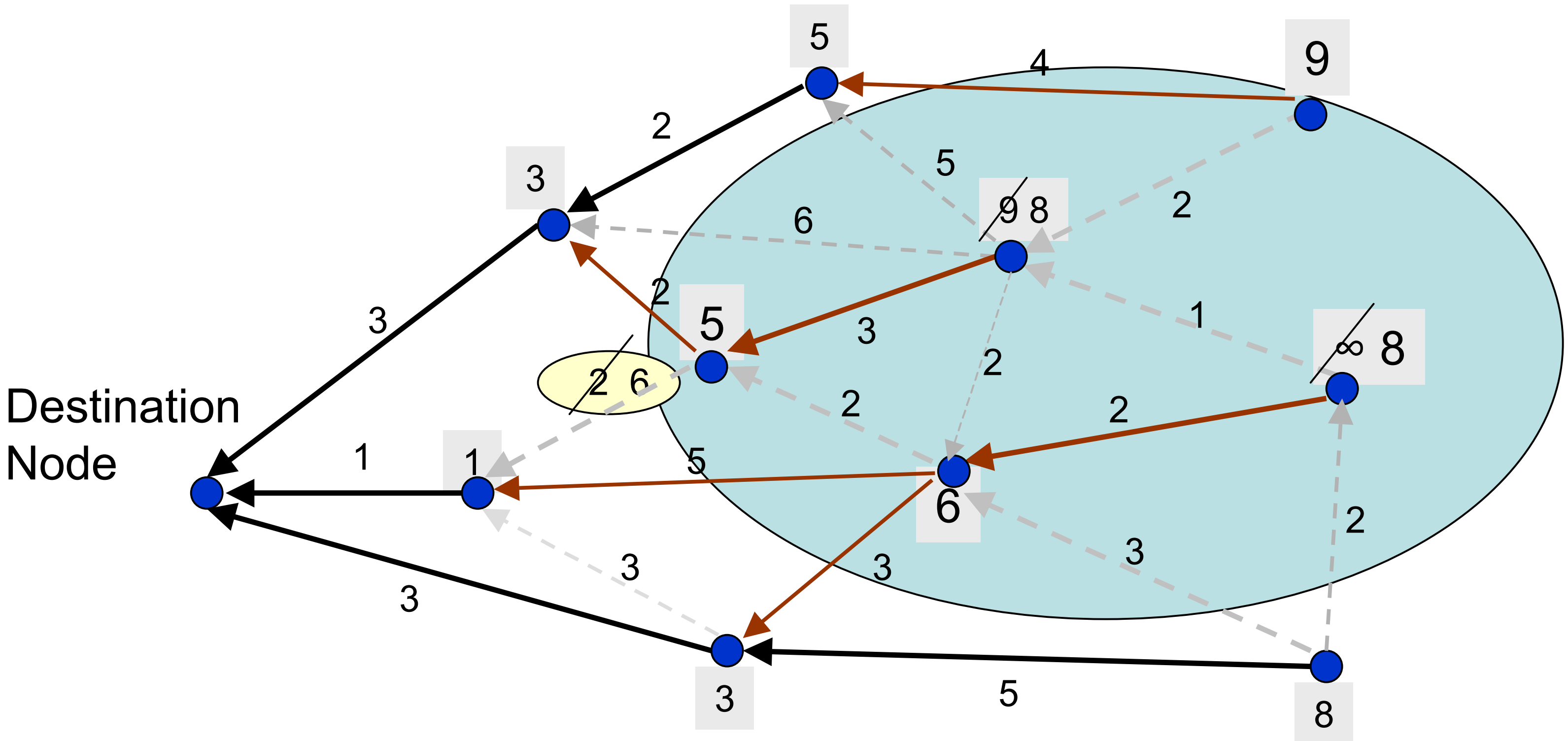
Check all outgoing links from nodes $u \in Q$ and update $dist_u$ if possible.
Insert all nodes u in a heap H considering their distances to the destination
H = {5, 6, 9, 9, ∞}

Updating Q-node distances



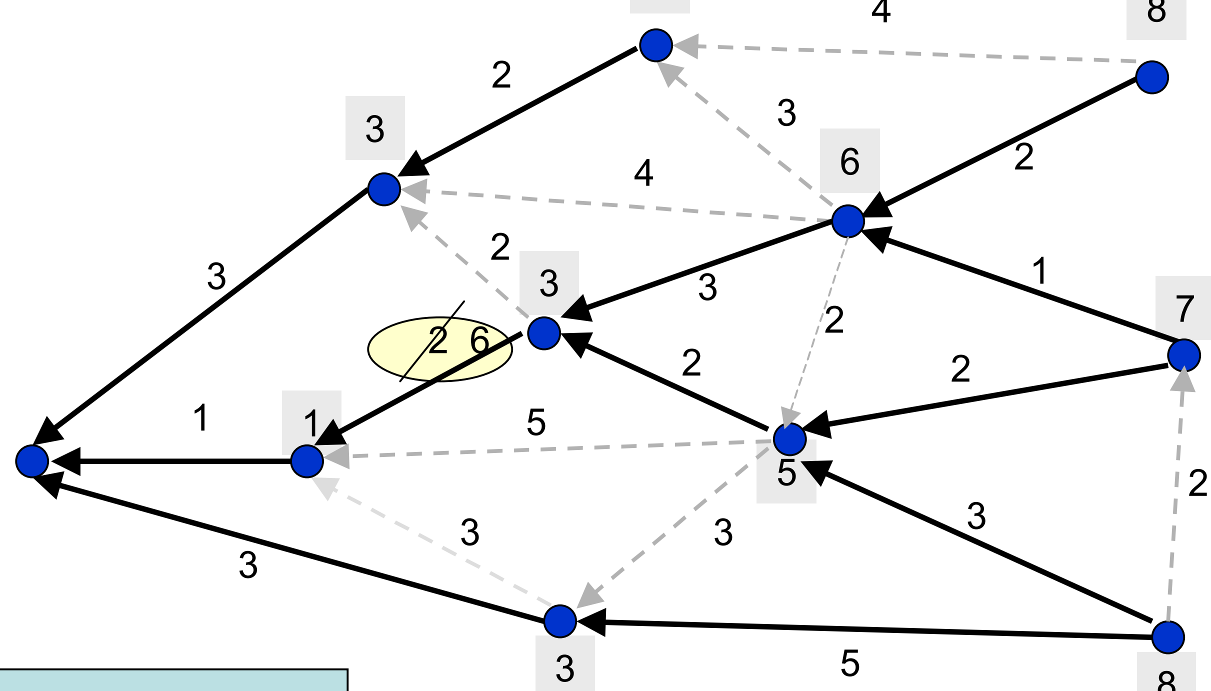
Remove nodes $u \in$ H, one by one. For each node u, traverse all incoming links $u \leftarrow s$ and update $dist_s$ if possible.

Determine the new SP graph

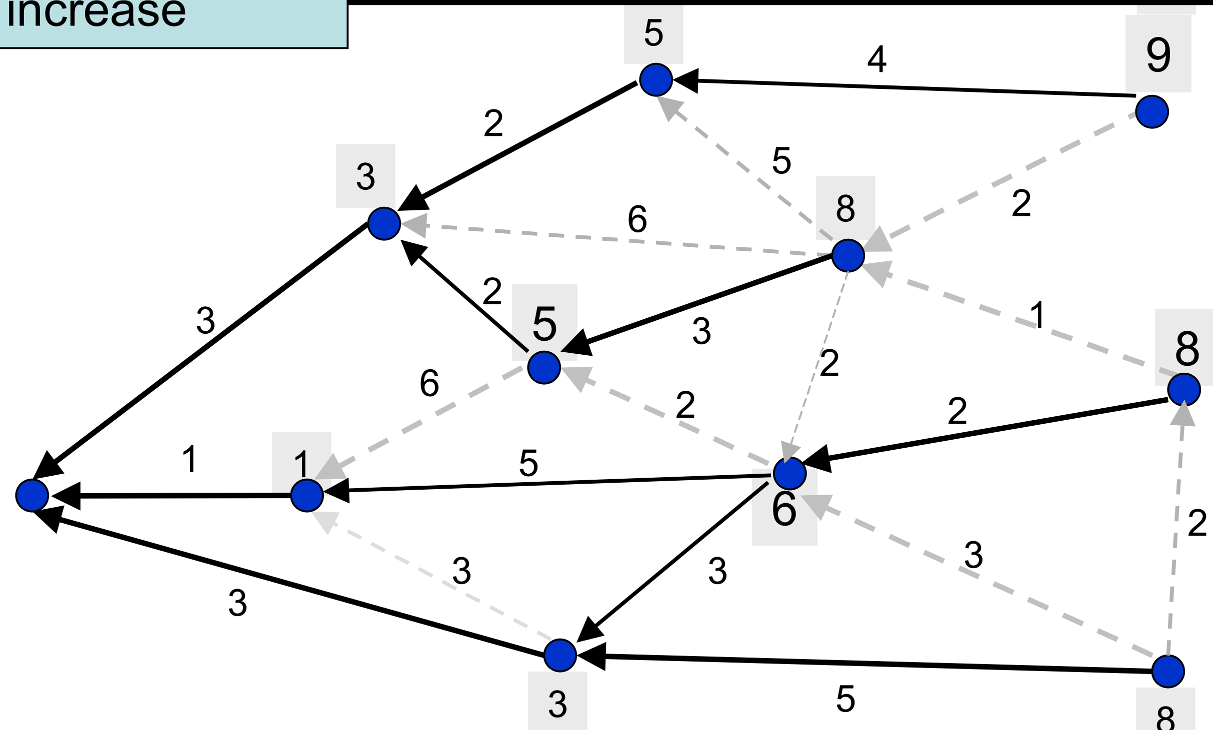


Traverse each outgoing link $e = u \rightarrow v$ from nodes $u \in Q$. If $dist_u = dist_v + w_e$ then arc $e \in G_{SP}$.

Original Graph

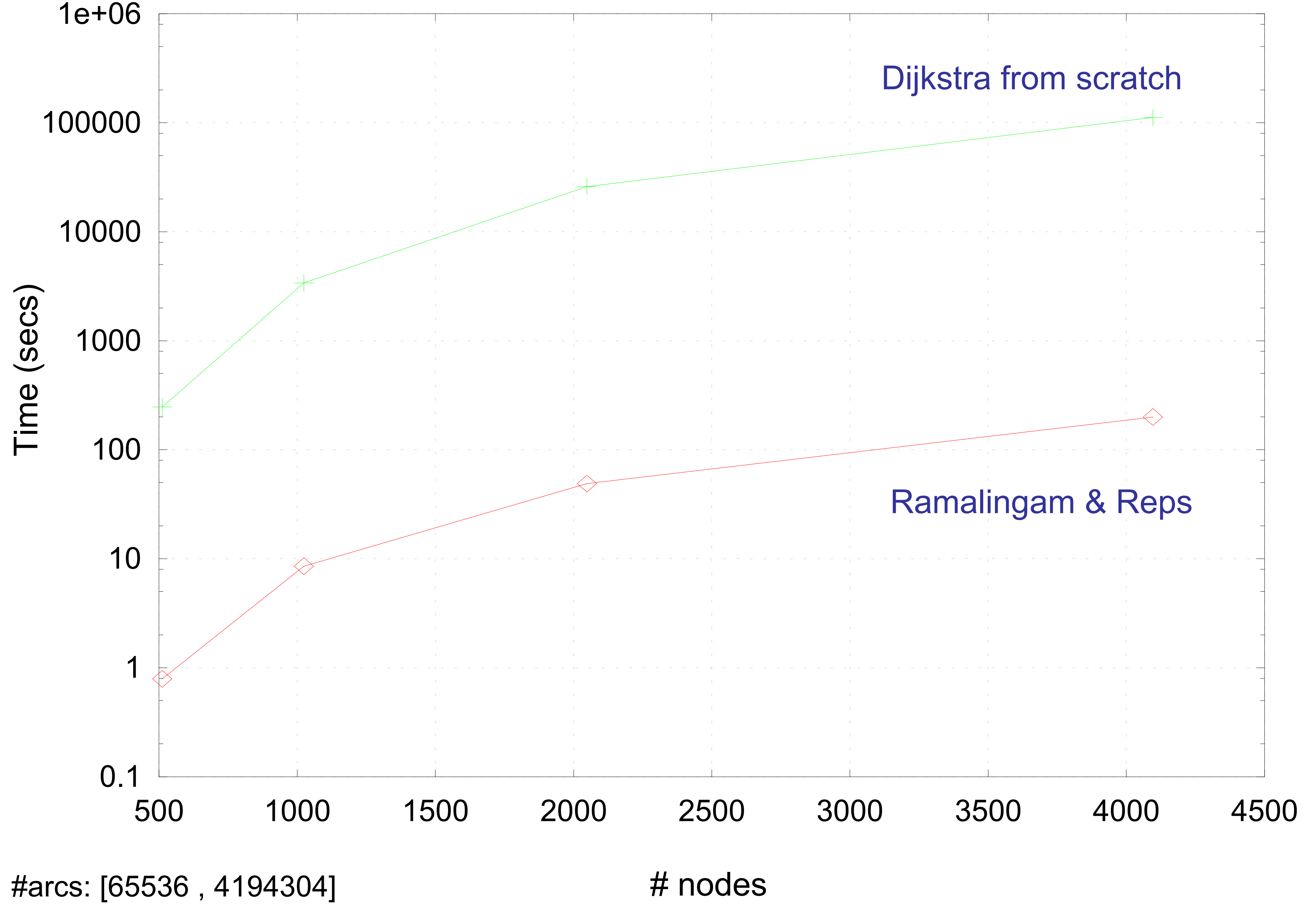


R&R weight increase

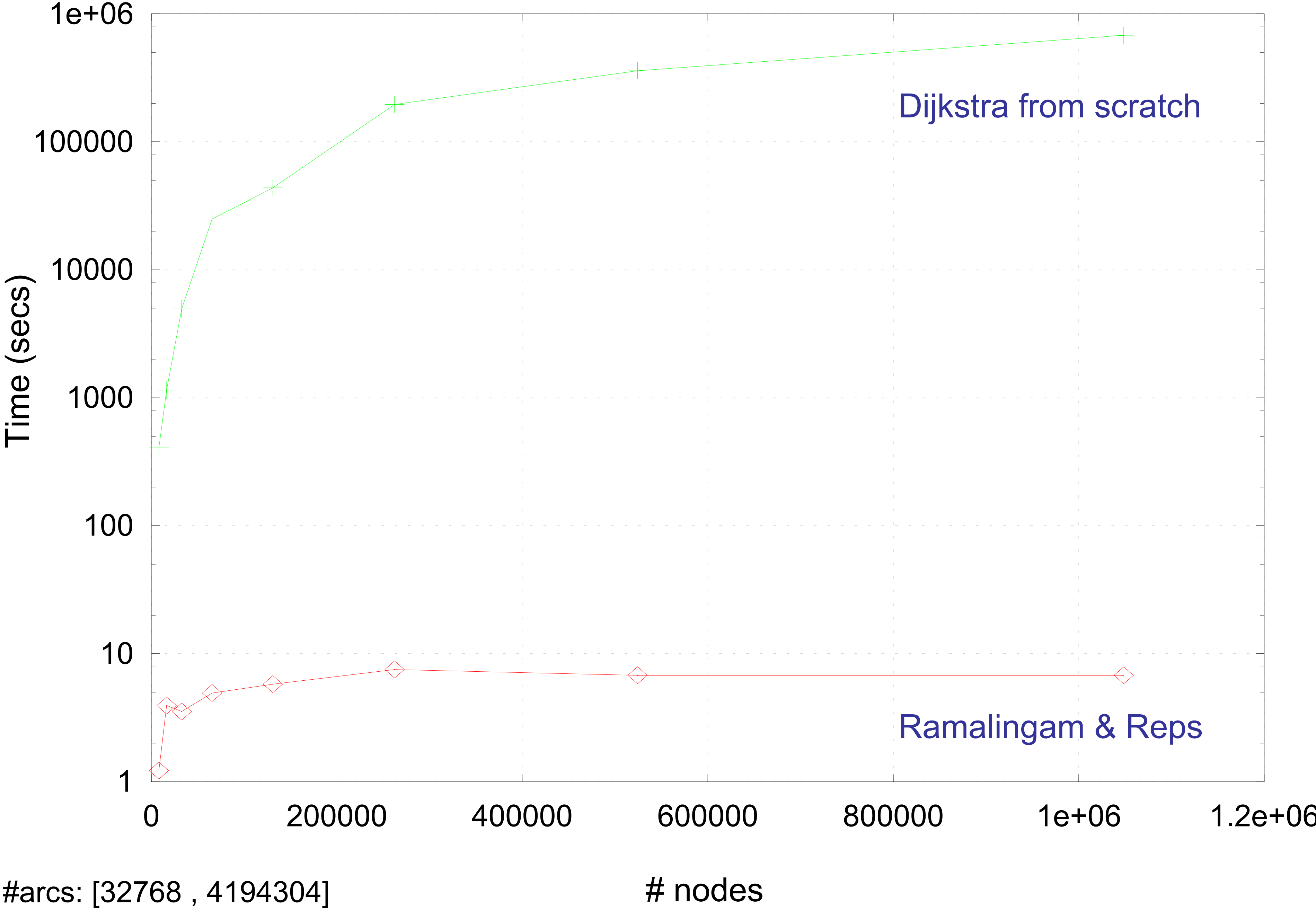


Updated Graph

Ramalingam & Reps vs Dijkstra on dense graphs

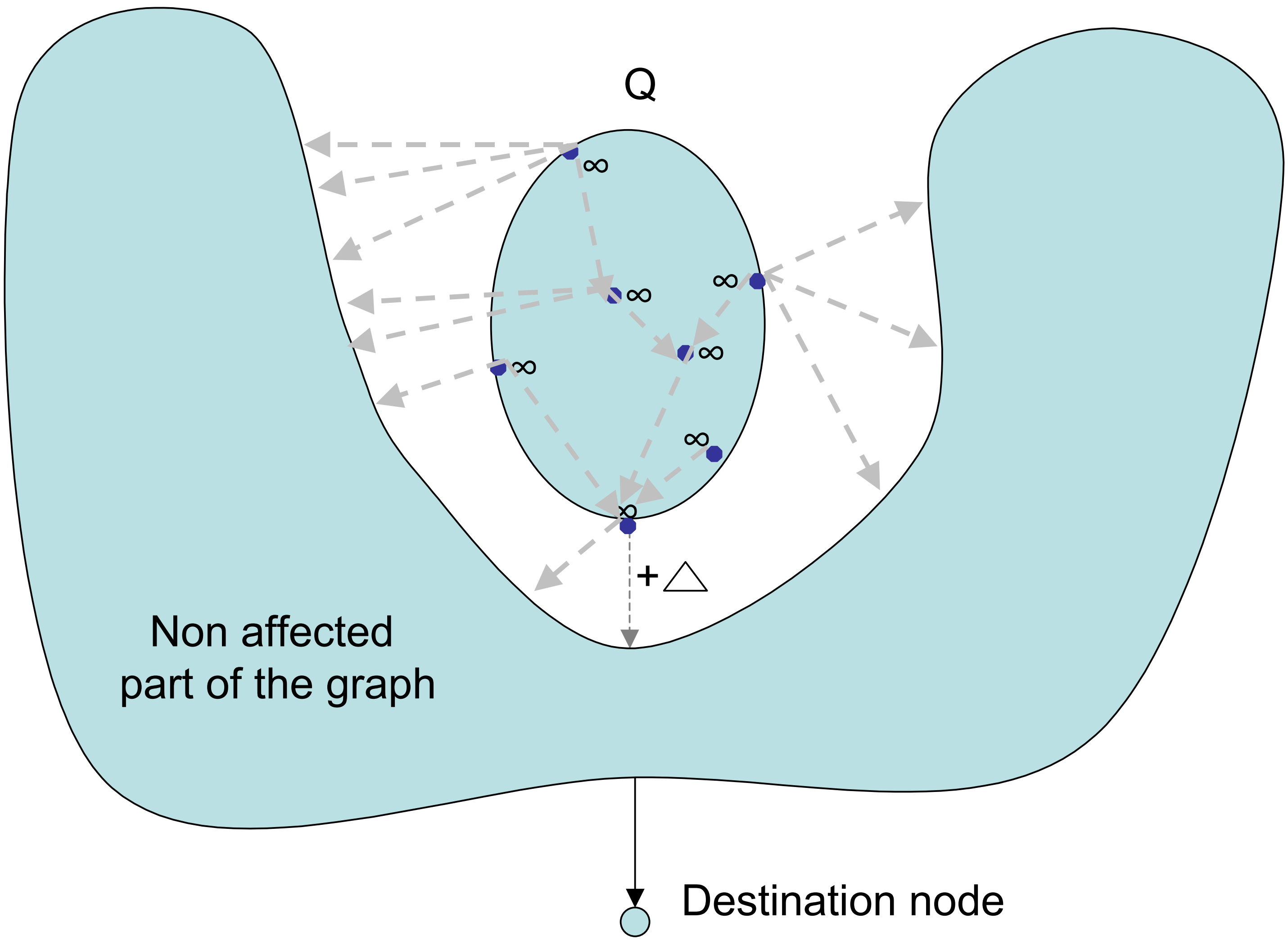


Ramalingam & Reps vs Dijkstra on sparse graphs



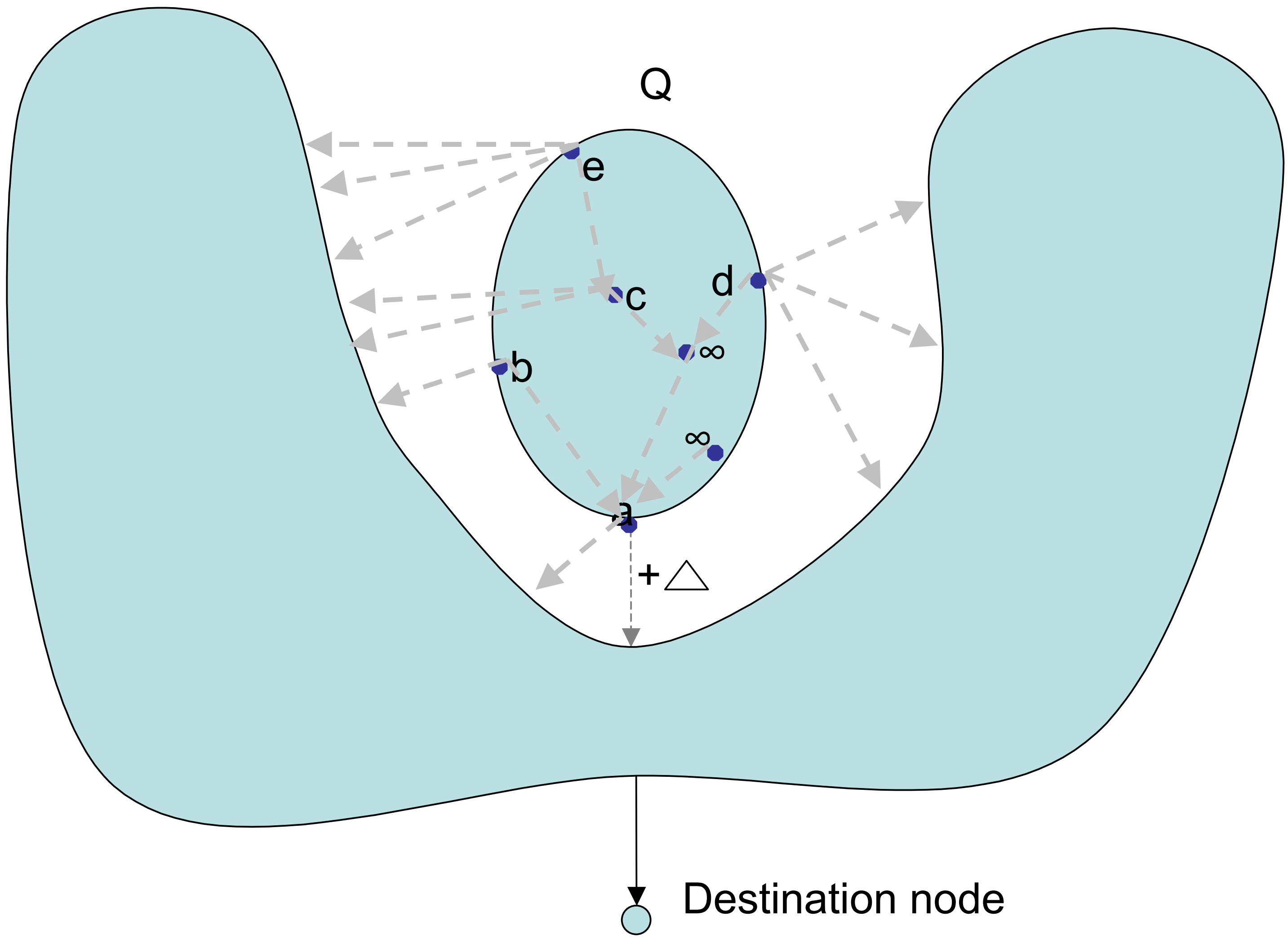
#arcs: [32768 , 4194304]

R&R: determining set Q



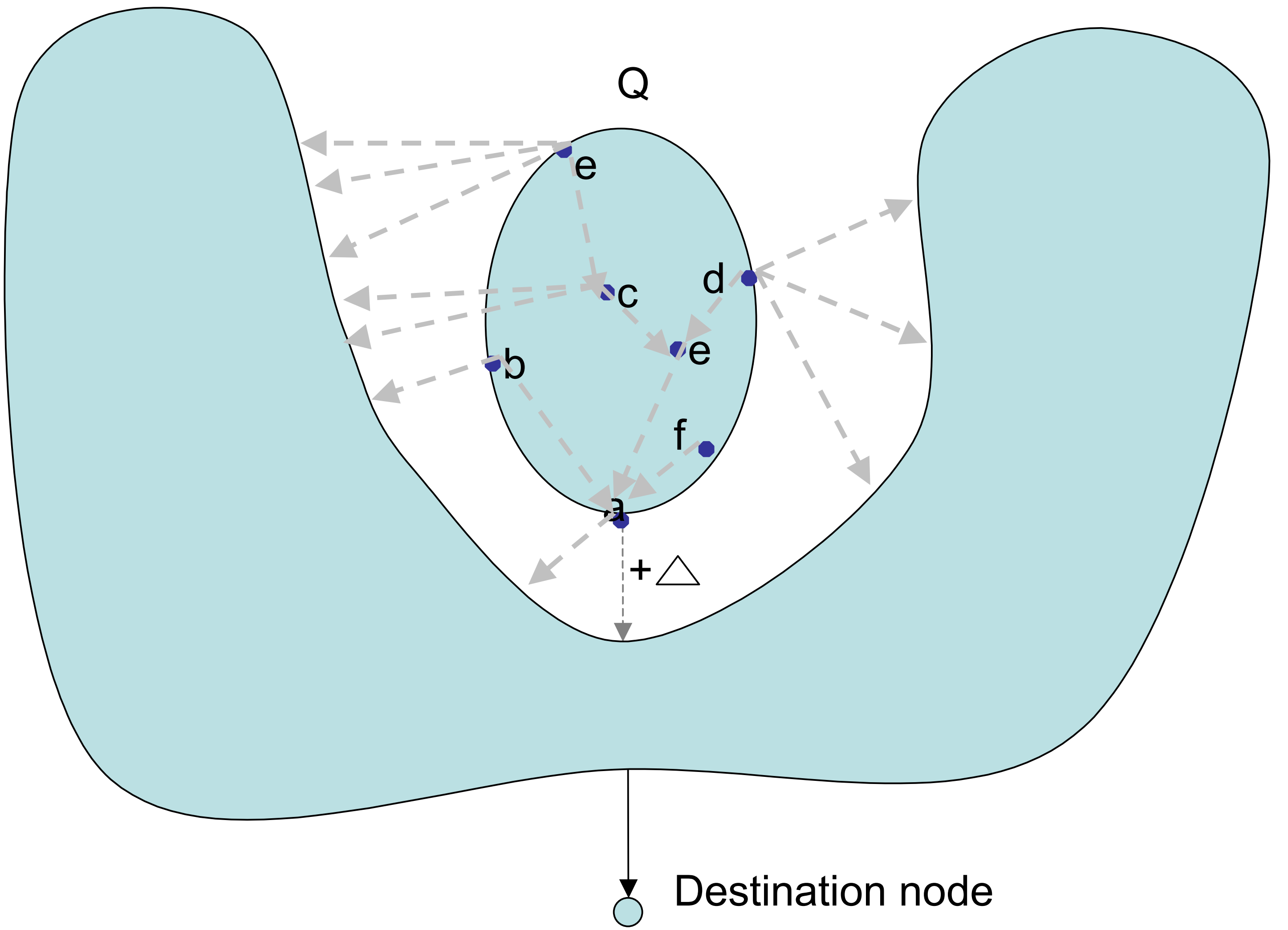
1 - Find set Q; remove all links from nodes $u \in Q$ and set $dist_u = \infty$.

R&R: updating Q-node distances



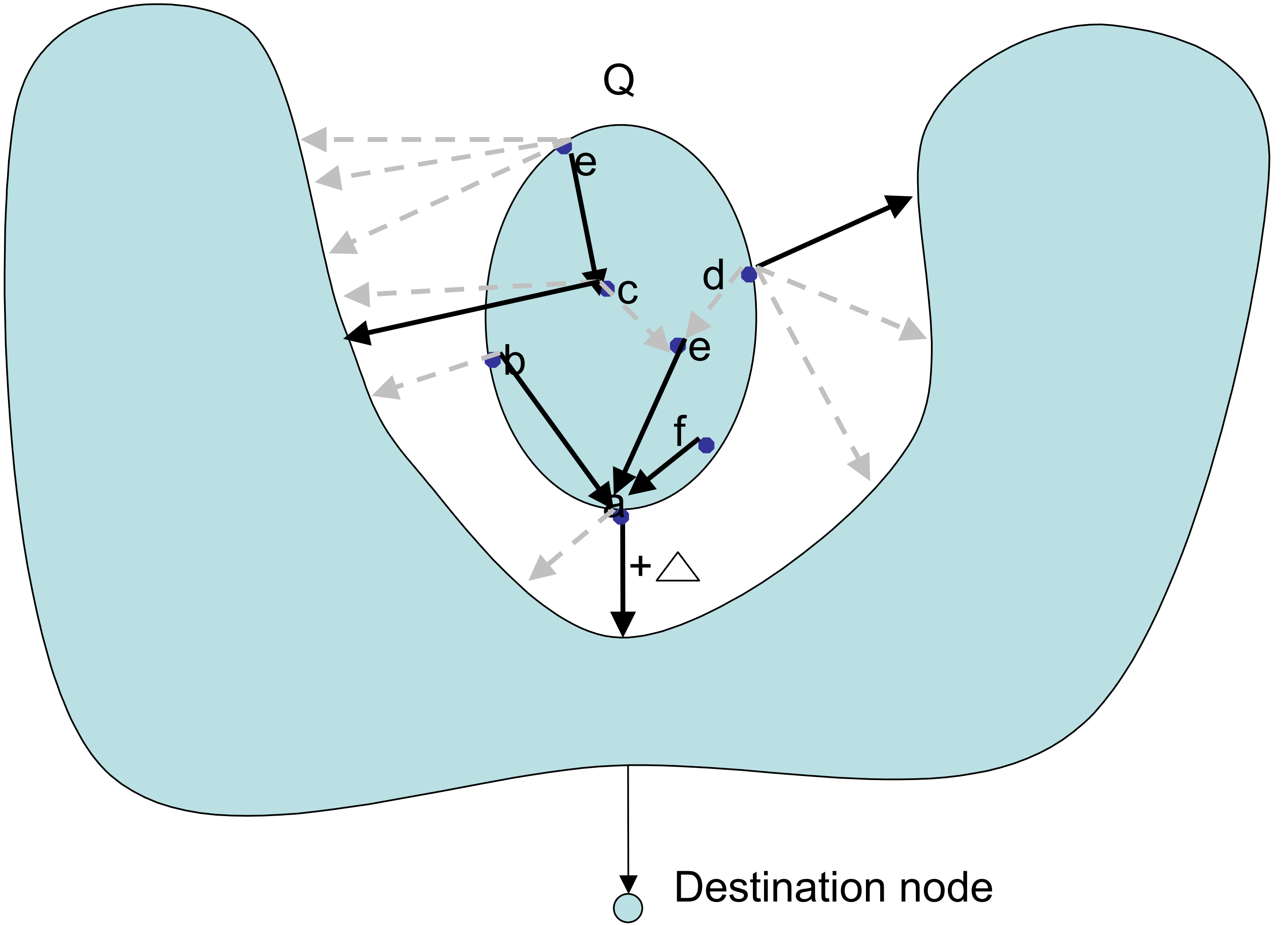
Update distances of nodes $u \in$ Q considering arcs linking nodes $\notin$ Q.

R&R: updating Q-node distances



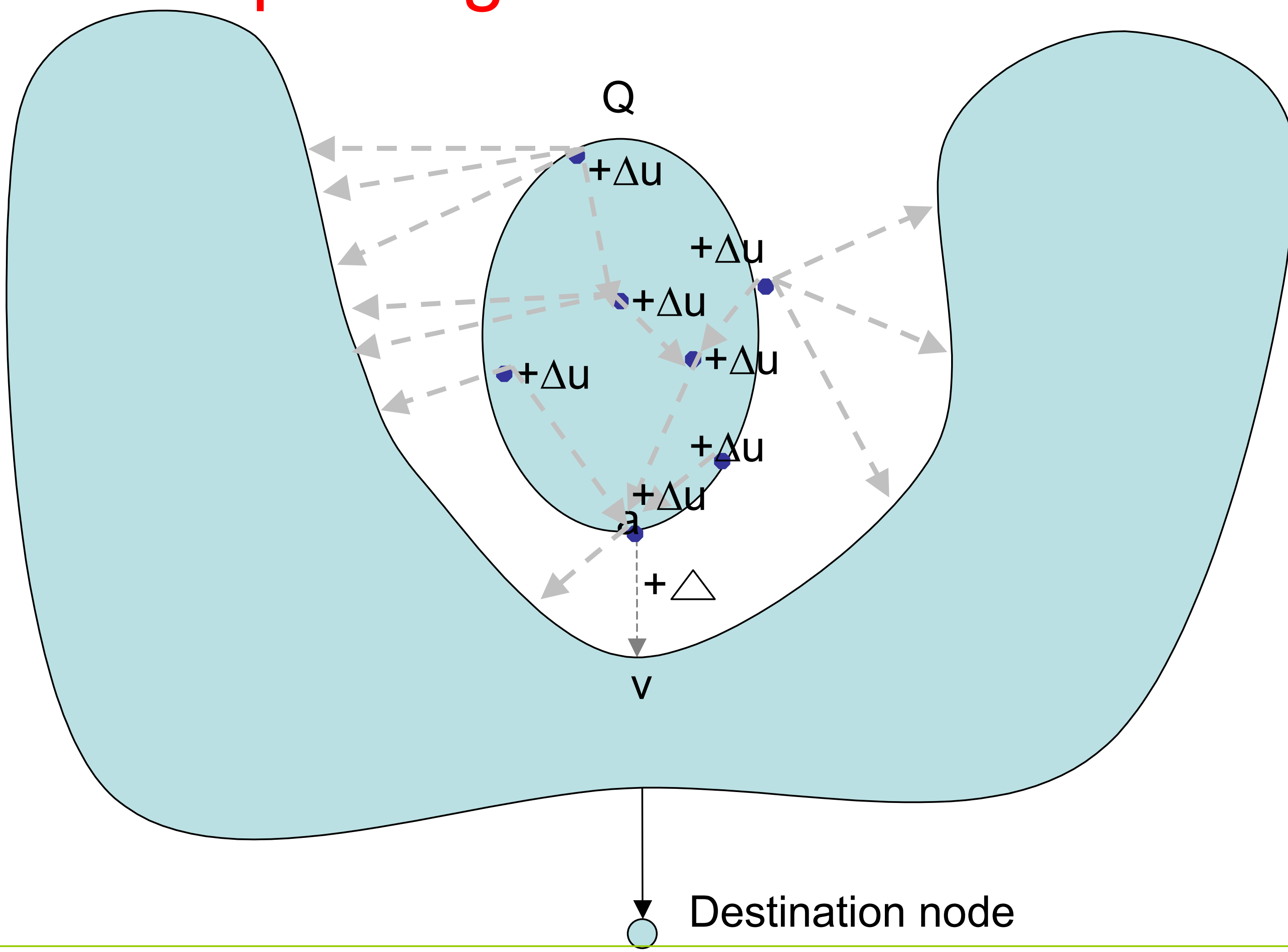
Update distances of nodes $u \in$ Q considering arcs linking nodes $\in$ Q.

R&R: determining the new G_{SP}



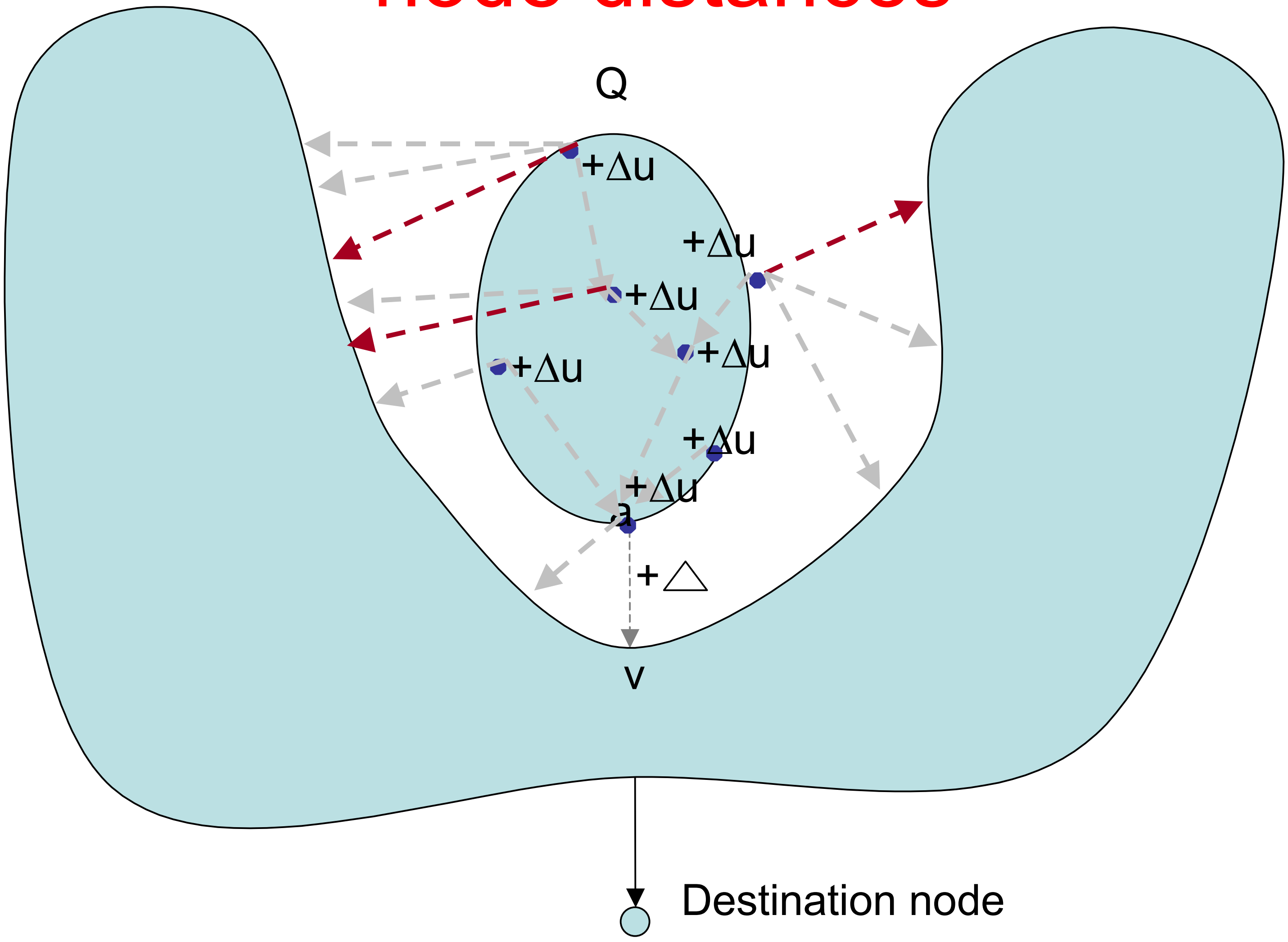
Traverse each outgoing link from nodes $u \in$ Q to compute $G_{SP.}$

Avoiding use of heaps: Determining set Q & updating Q-node distances



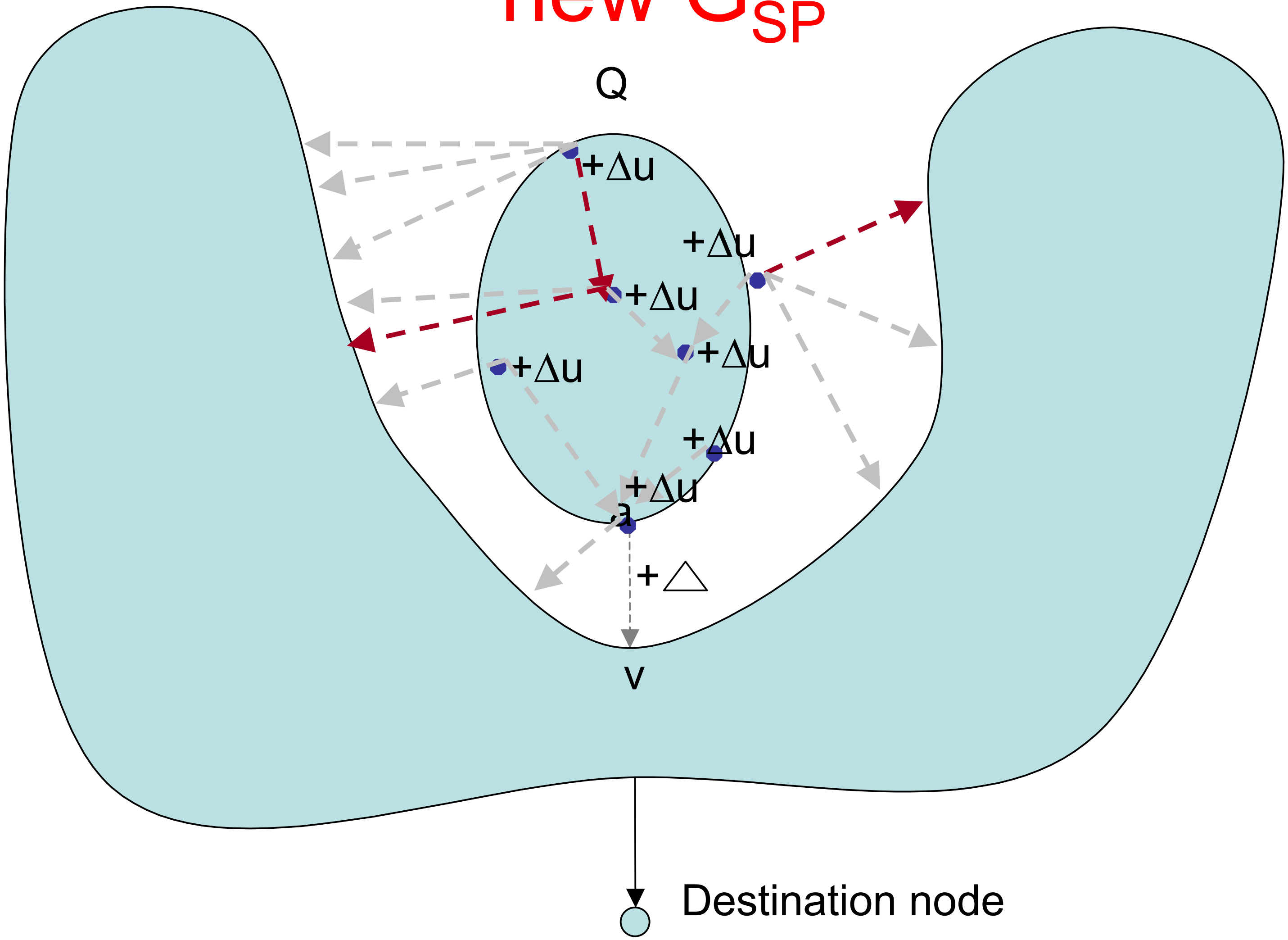
Instead of attributing ∞ to the distances of all nodes ∈ Q, add to their original distances the value Δ_u, where Δ_u is the amount that $dist_u$ will increase by considering the cheapest outgoing link from *u*.

Avoiding use of heaps: updating Q-node distances



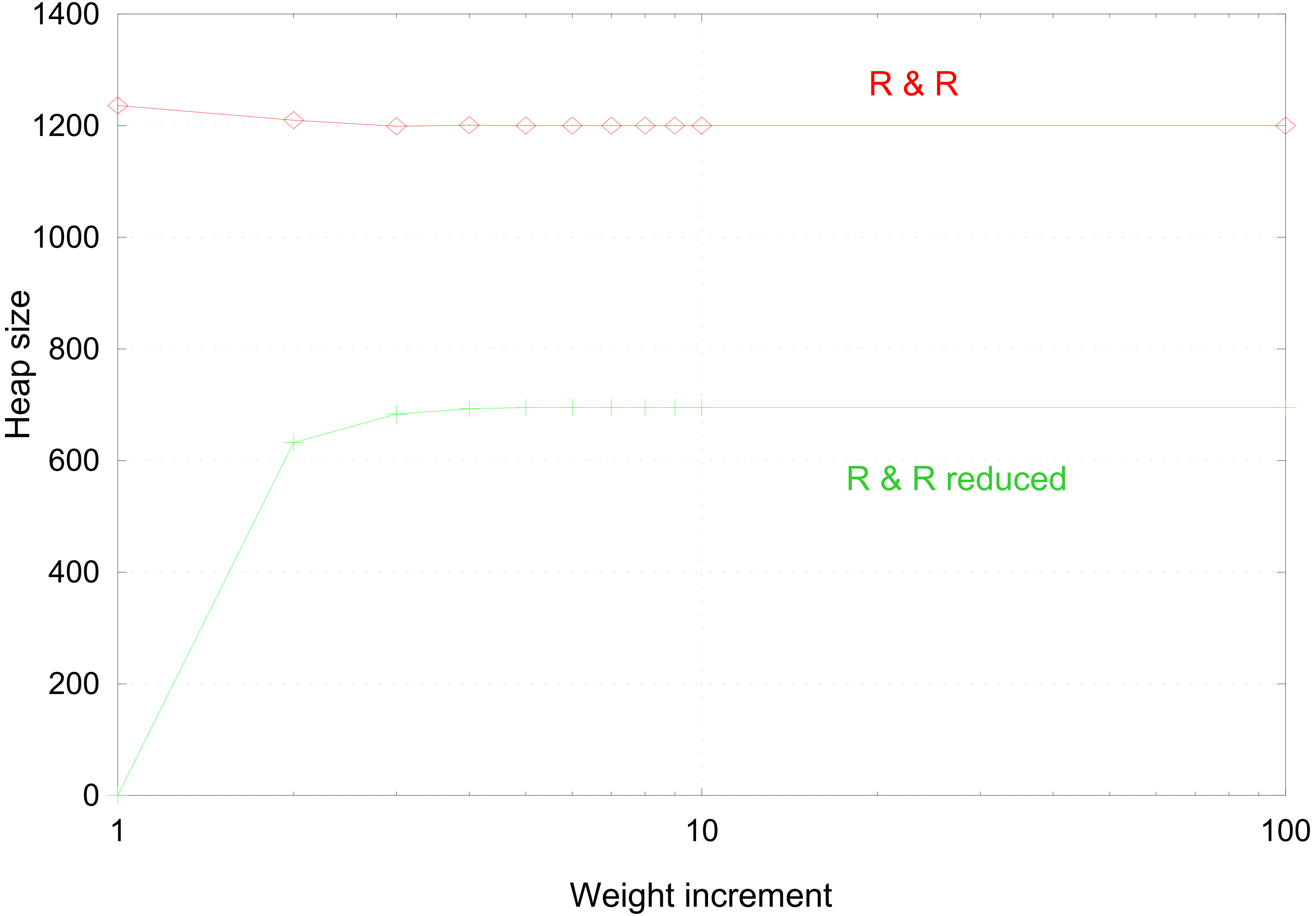
Insert in H only nodes that have an alternative cheapest path linking a node ∉ Q

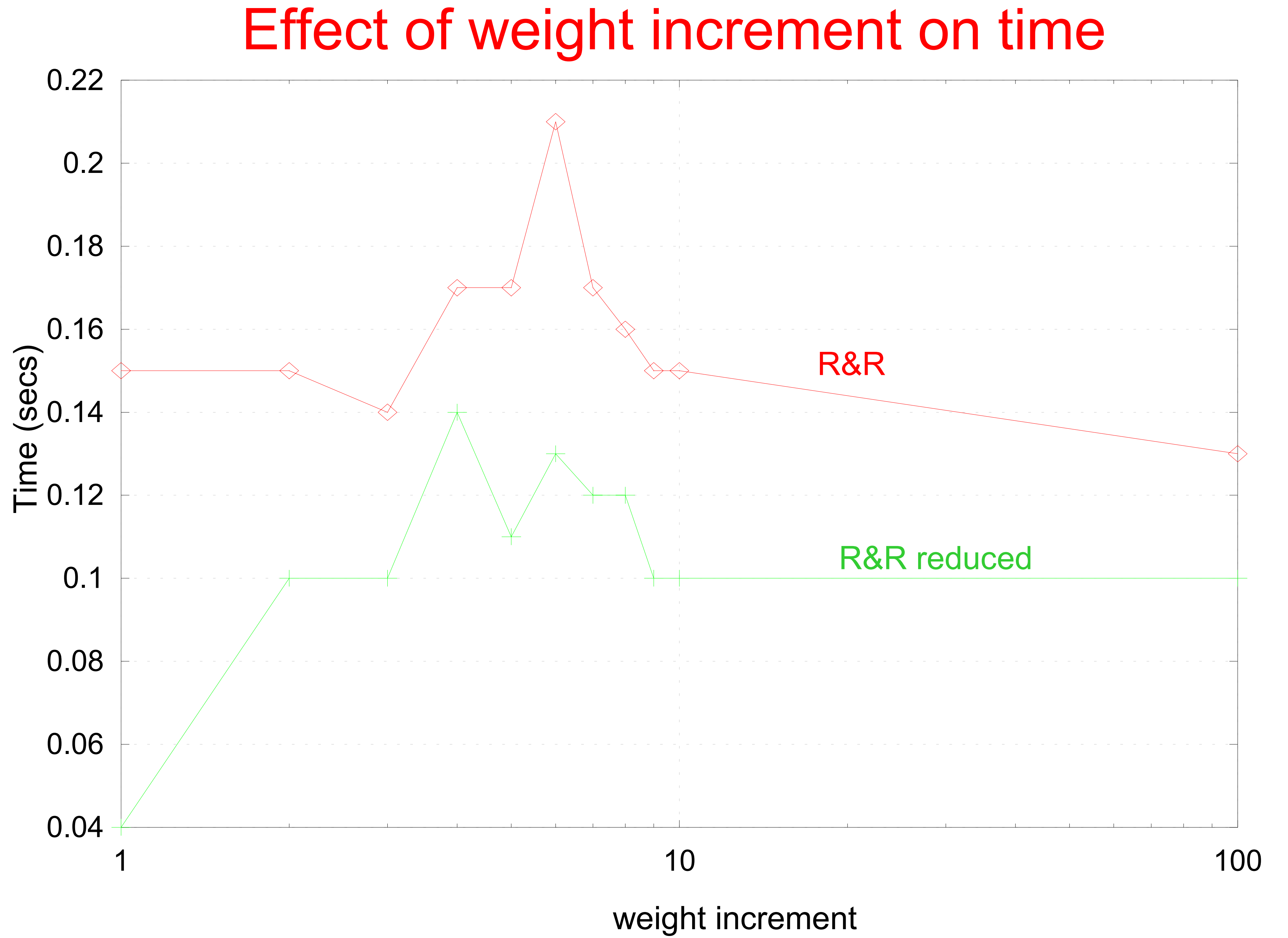
Avoiding use of heaps: determining the new G_{SP}



Remove nodes from H, one by one, and insert/update in H new nodes which can have their distances decreased.

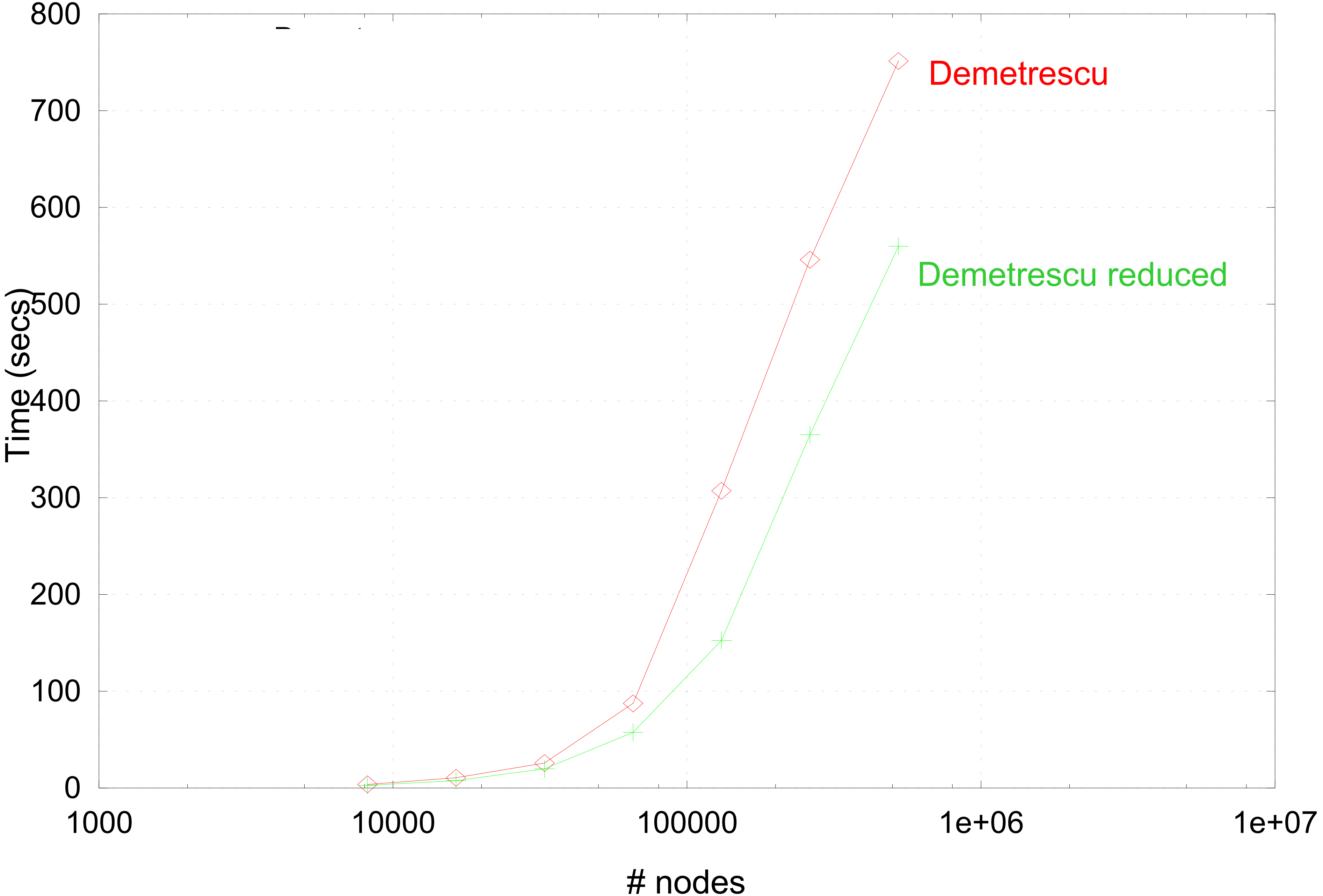
Effect of weight increment on heap size

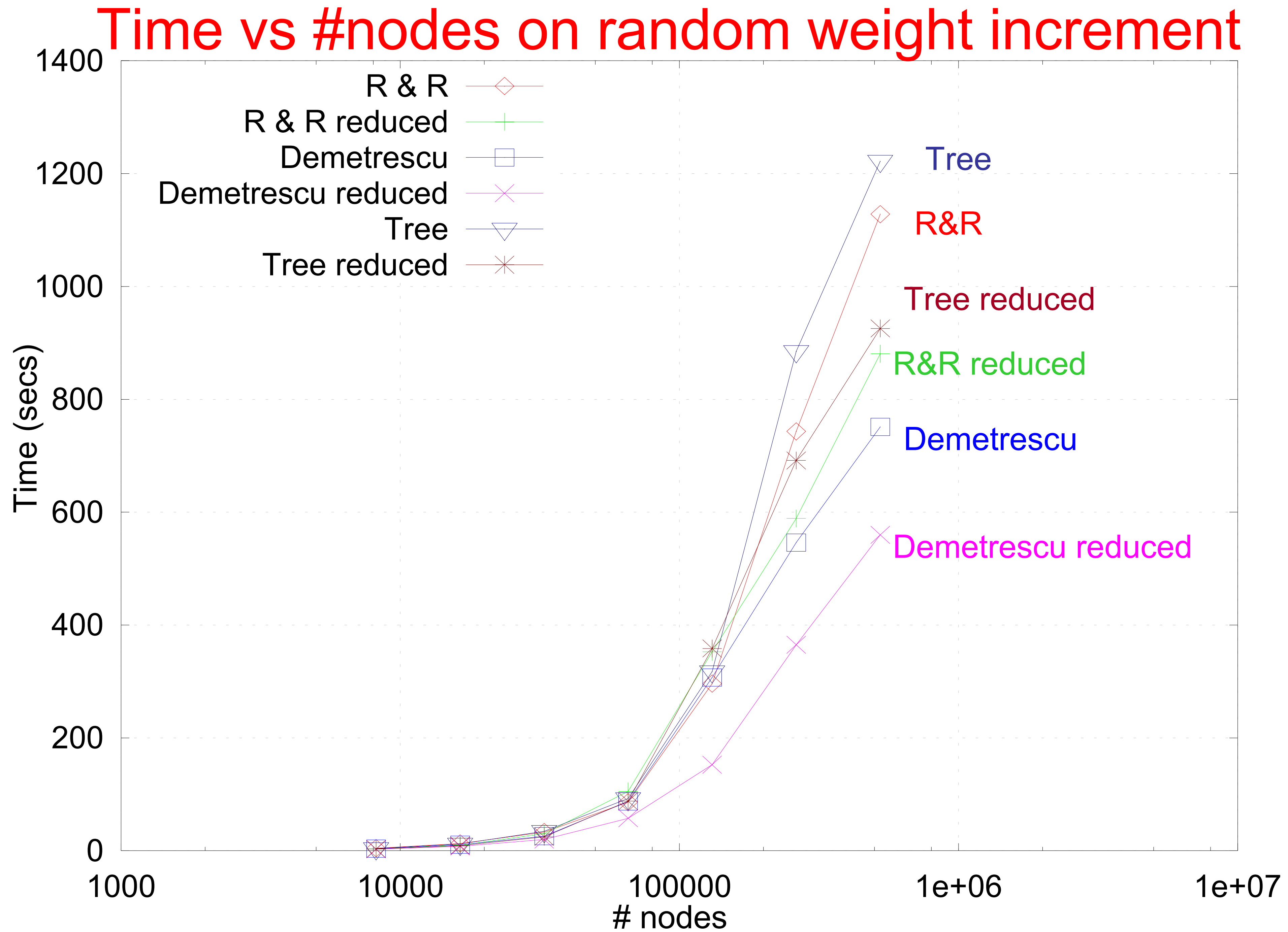




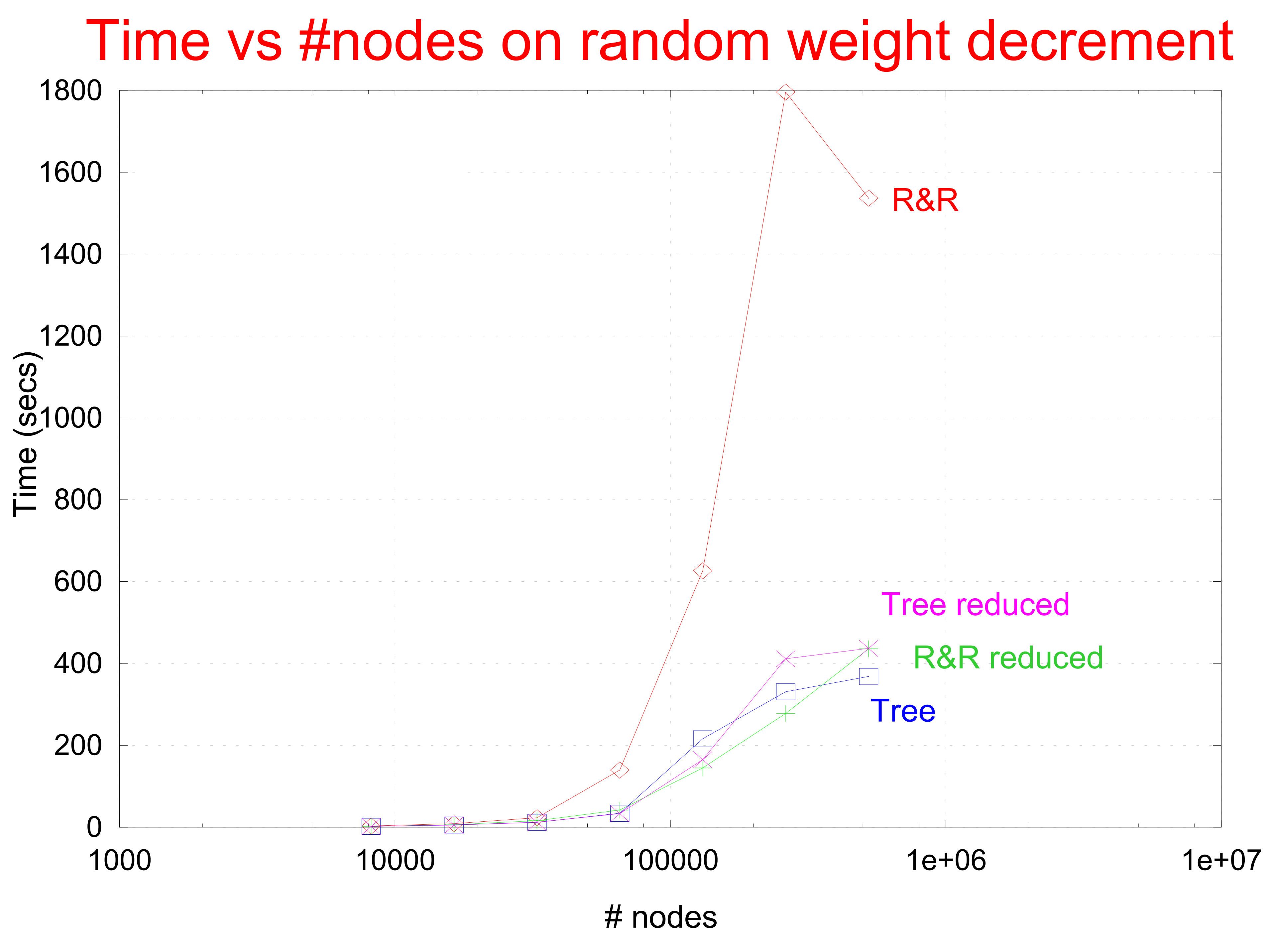
Effect of weight increment on time
0.22
0.2
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
Time (secs)
1
10
100
weight increment
R&R
R&R reduced

Time vs #nodes on random weight increment

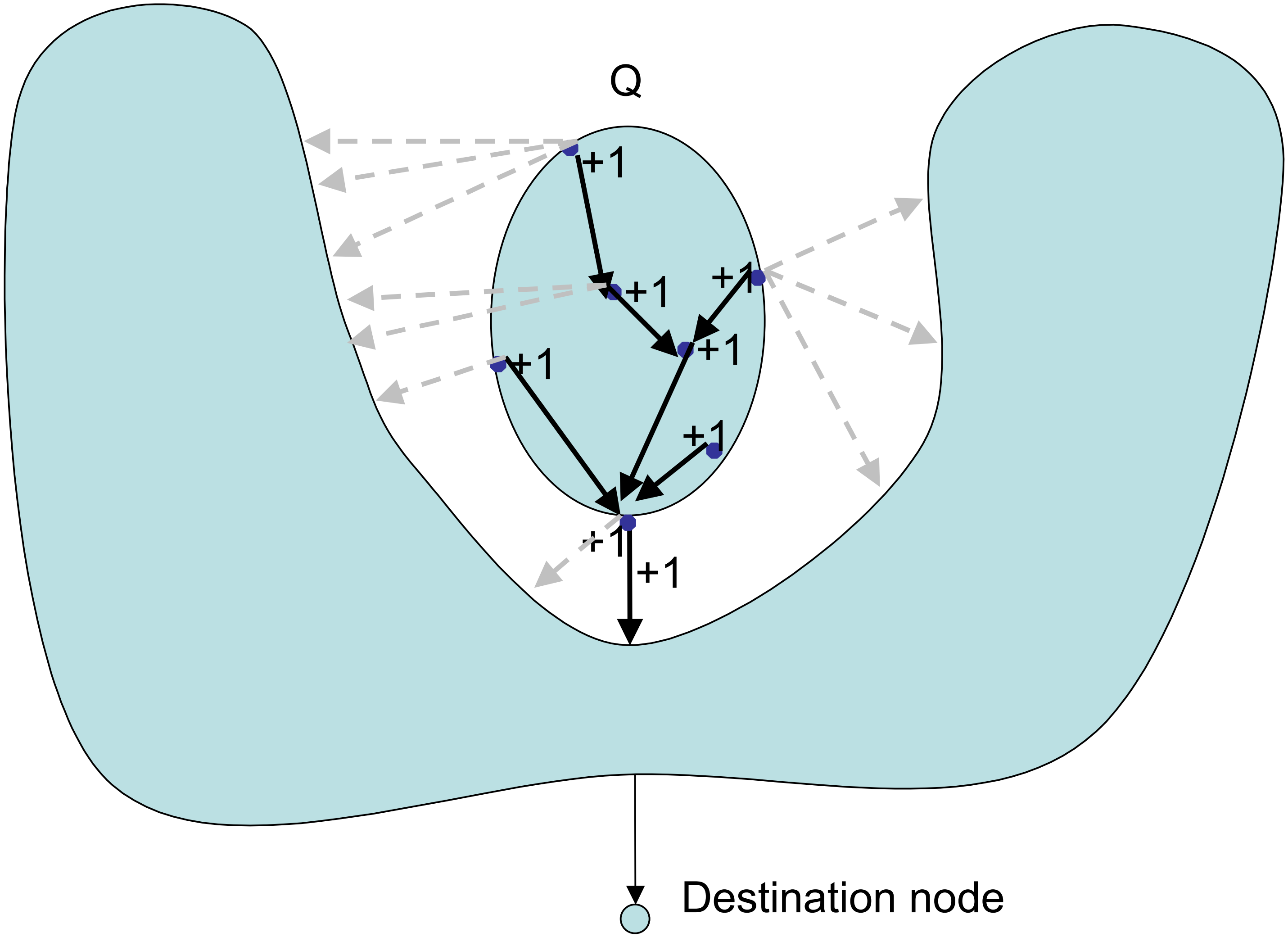




Time vs #nodes on random weight increment
R & R
R & R reduced
Demetrescu
Demetrescu reduced
Tree
Tree reduced
Tree
R&R
Tree reduced
R&R reduced
Demetrescu
Demetrescu reduced
Time (secs)
nodes
0
200
400
600
800
1000
1200
1400
1000
10000
100000
1e+06
1e+07

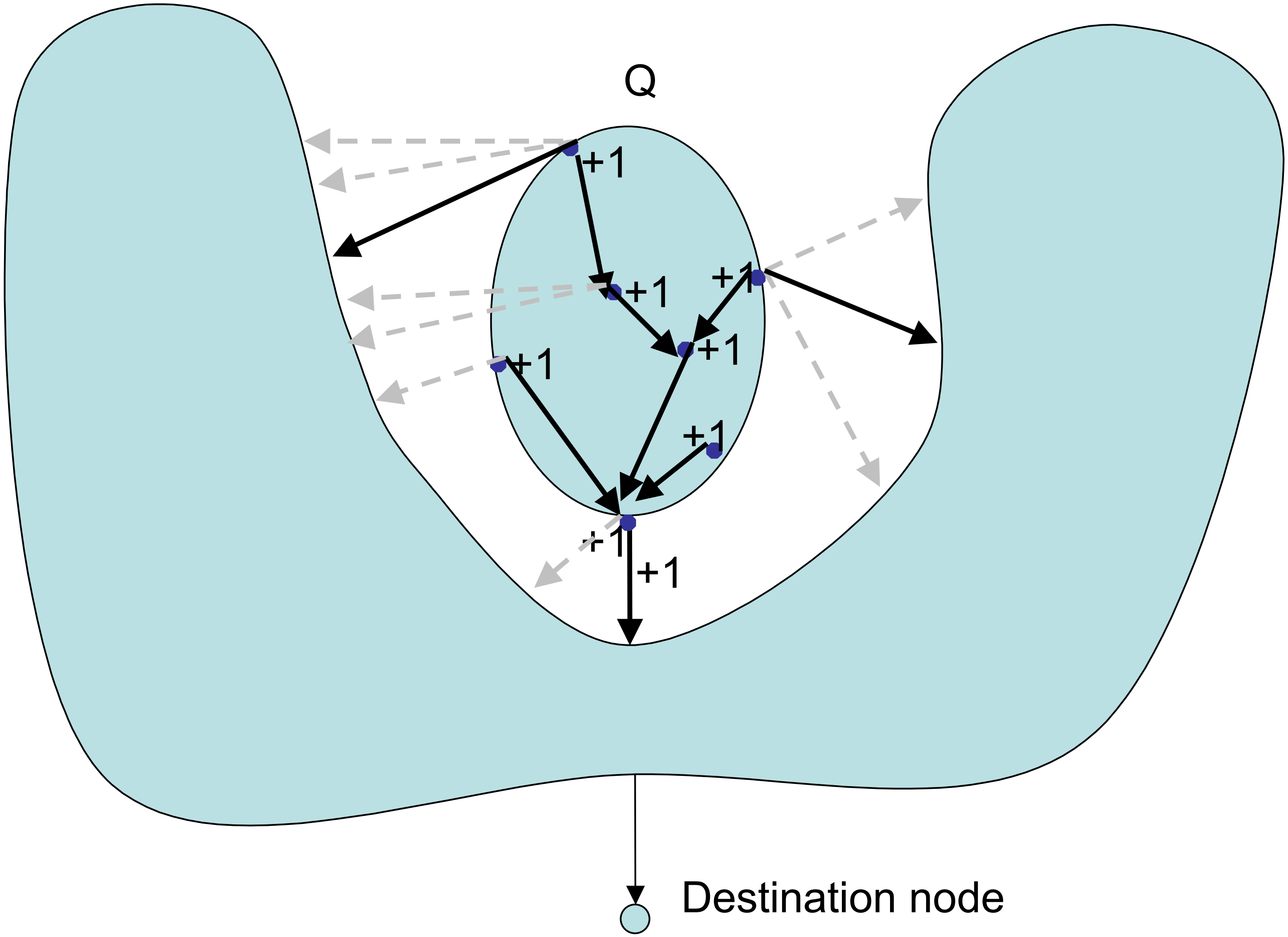

Time vs #nodes on random weight decrement
Time (secs)
1800
1600
1400
1200
1000
800
600
400
200
0
1000
10000
100000
1e+06
1e+07
nodes
R&R
Tree reduced
R&R reduced
Tree

Avoiding heaps: Unit increase



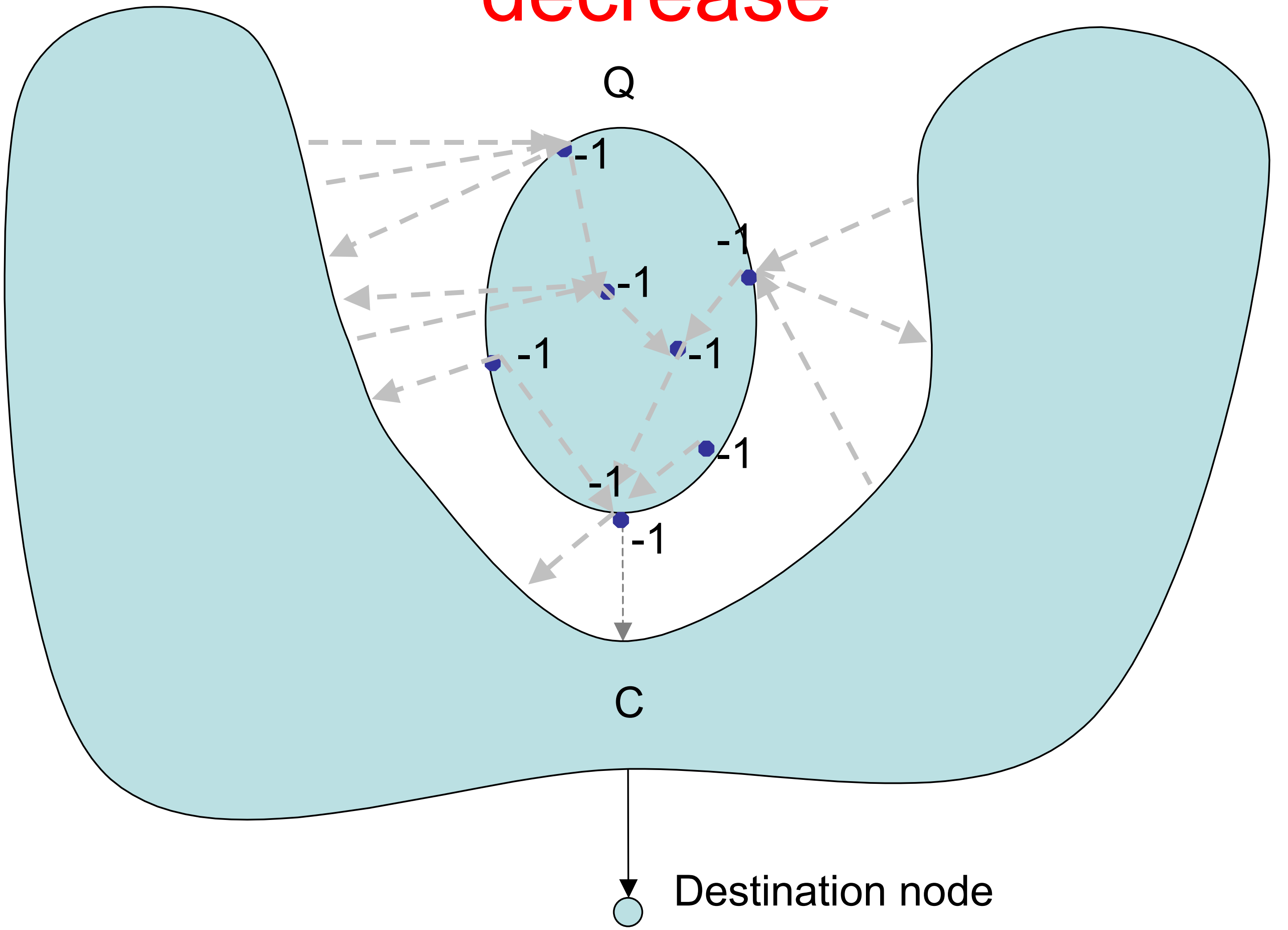
Increment by 1 all distances from nodes $u \in$ Q.

Avoiding heaps: Unit increase



Traverse each outgoing link from nodes $u \in$ Q to compute G_{SP}.

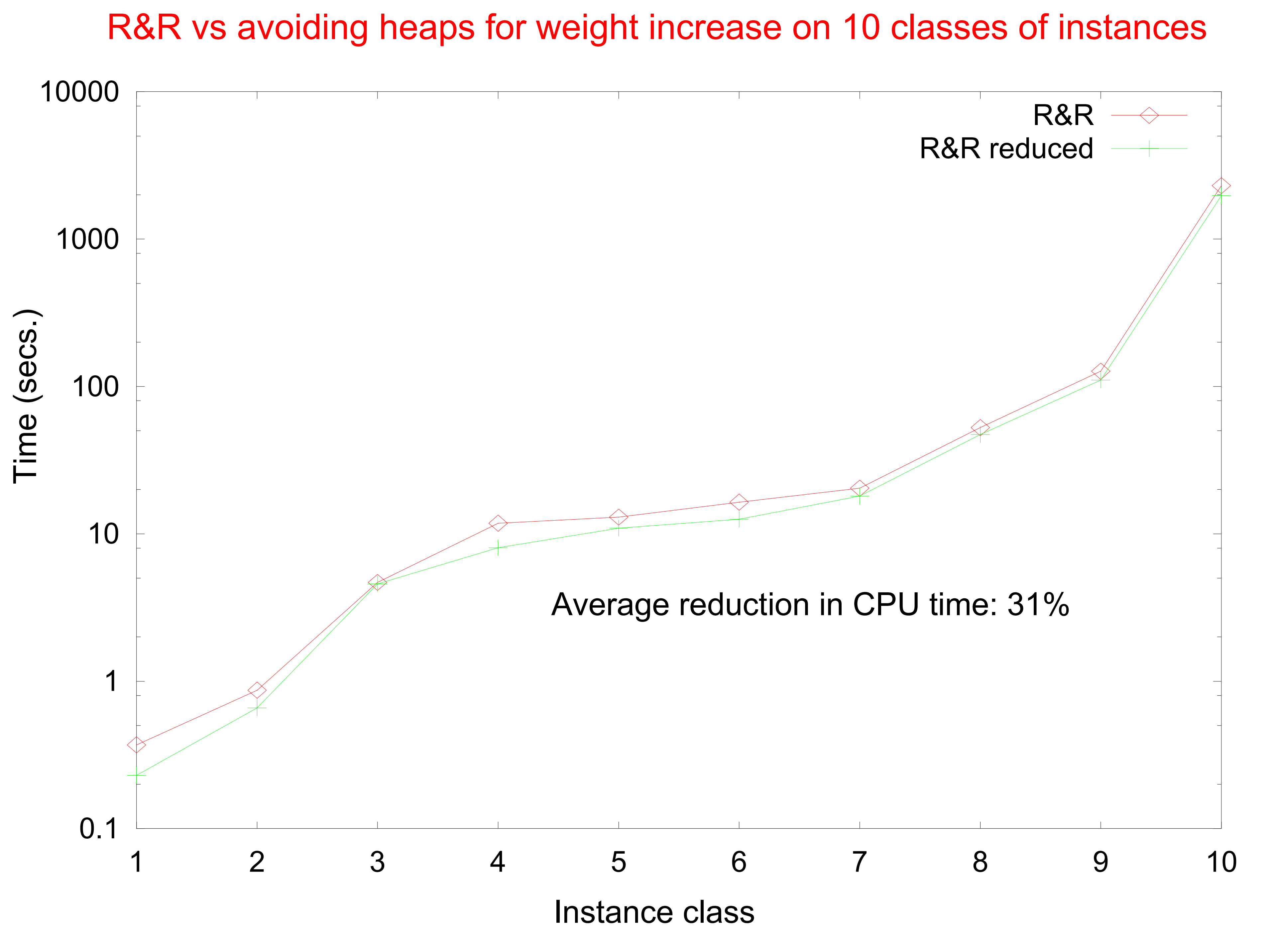
Avoiding use of heaps in unit weight decrease



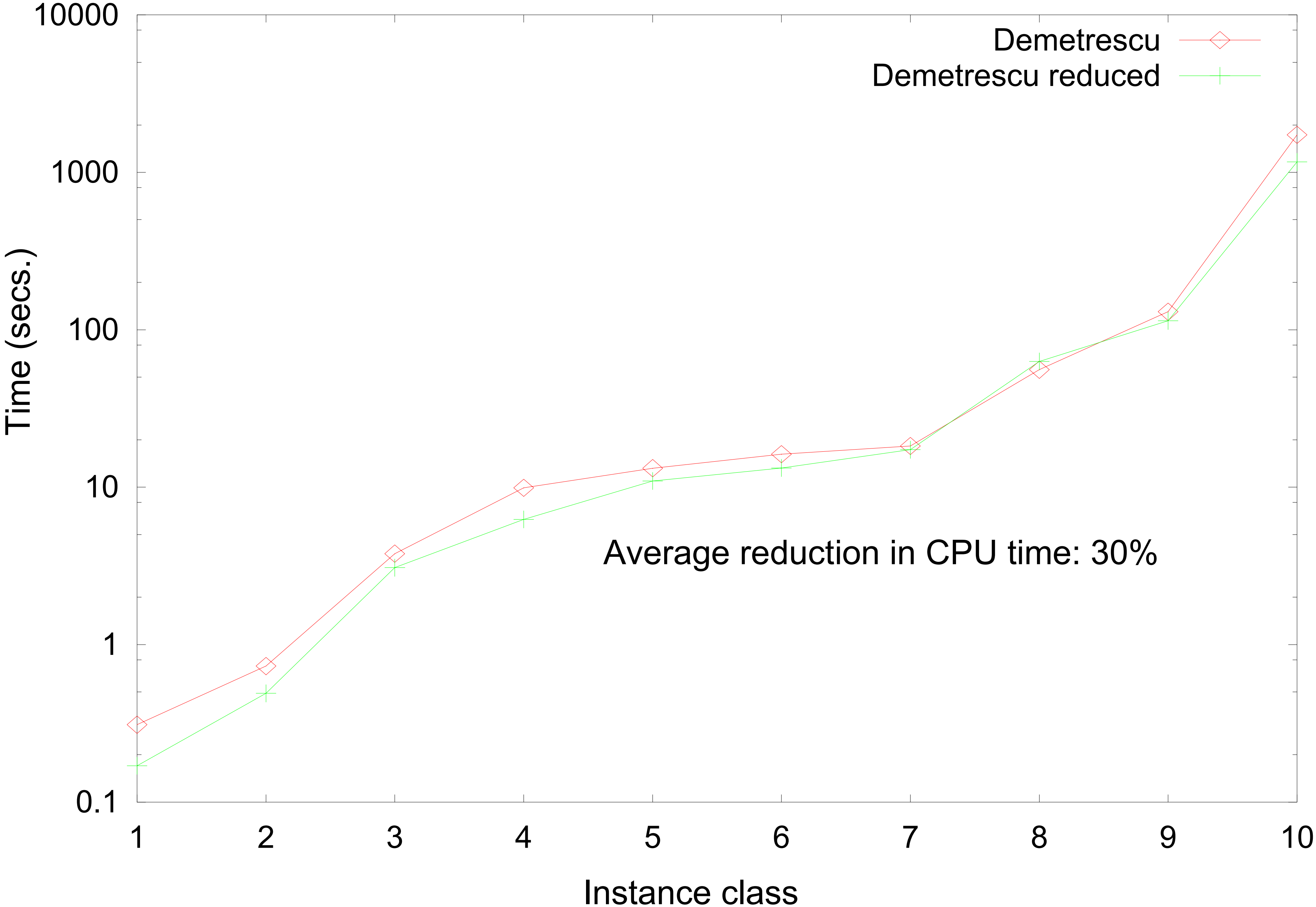
Considering unit decrement, the sets A and B are empty.

Computational results

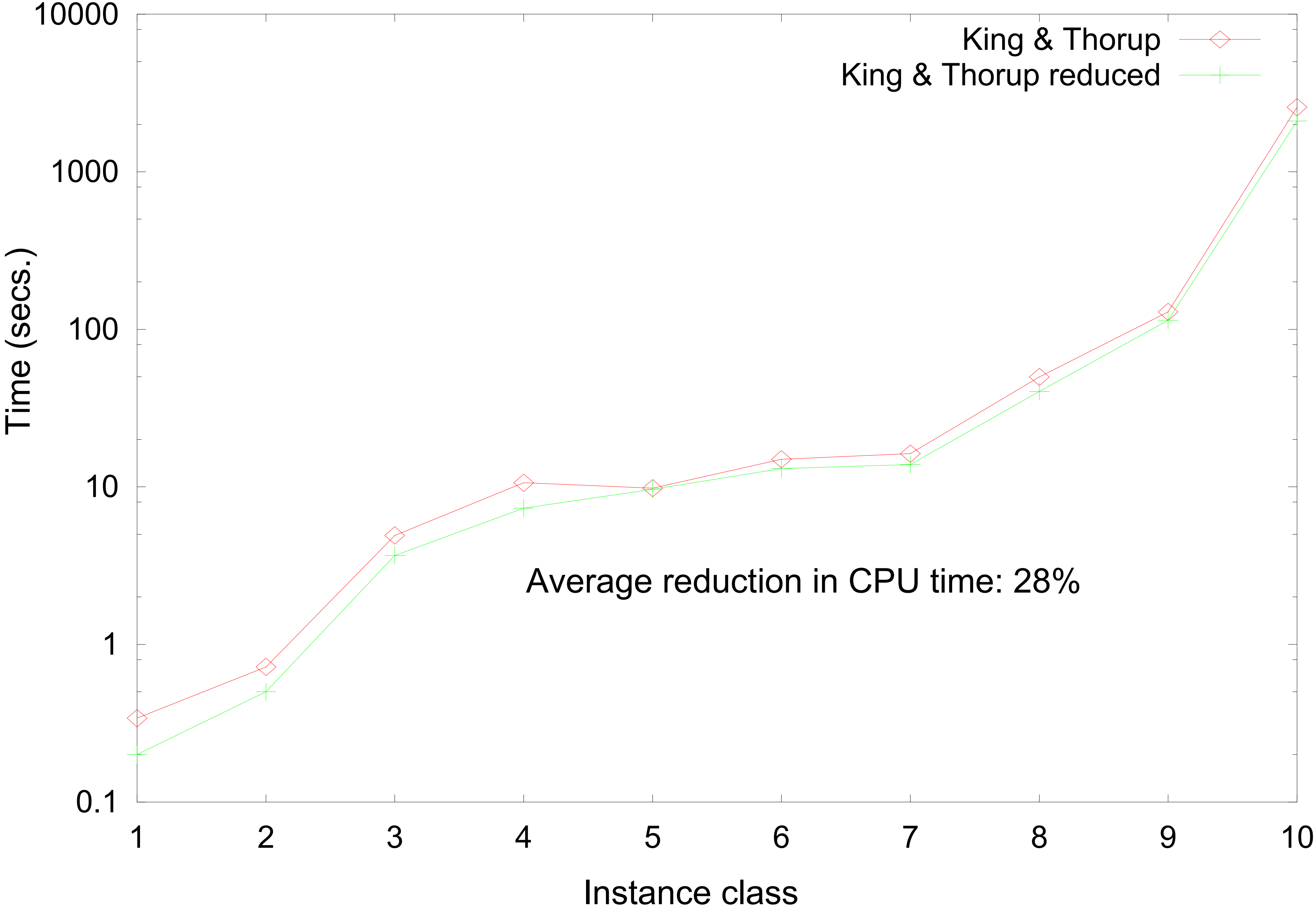
- 10 classes of graphs:
  - Real data from AT&T;
  - Small instances used in OSPF studies by Fortz & Thorup (2000);
  - Sparse graphs, dense graphs, square/long/large shape, hard graphs, etc. by A. Goldberg from DIMACS Challenge;
- Instance sizes from 50 to 3 million nodes; 200 to 5 million arcs.
- Weight setting range: [1,10000];
- For each instance, we applied 5000 weight increases and 5000 decreases. We force the changes to always alter G_{SP}.


R&R vs avoiding heaps for weight increase on 10 classes of instances
R&R
R&R reduced
Time (secs.)
10000
1000
100
10
1
0.1
1
2
3
4
5
6
7
8
9
10
Instance class
Average reduction in CPU time: 31%

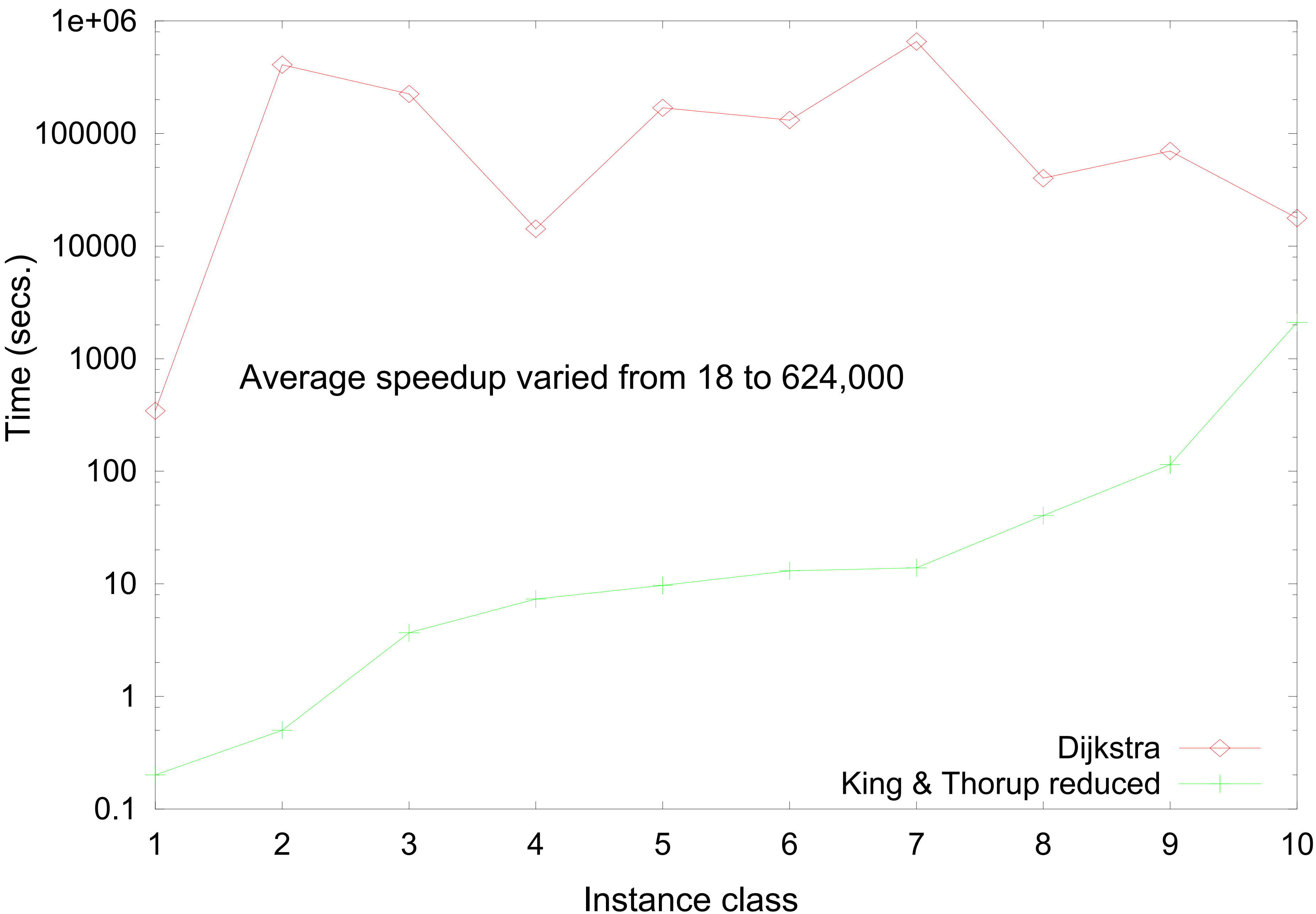
Demetrescu vs avoiding heaps for weight increase on 10 classes of instances



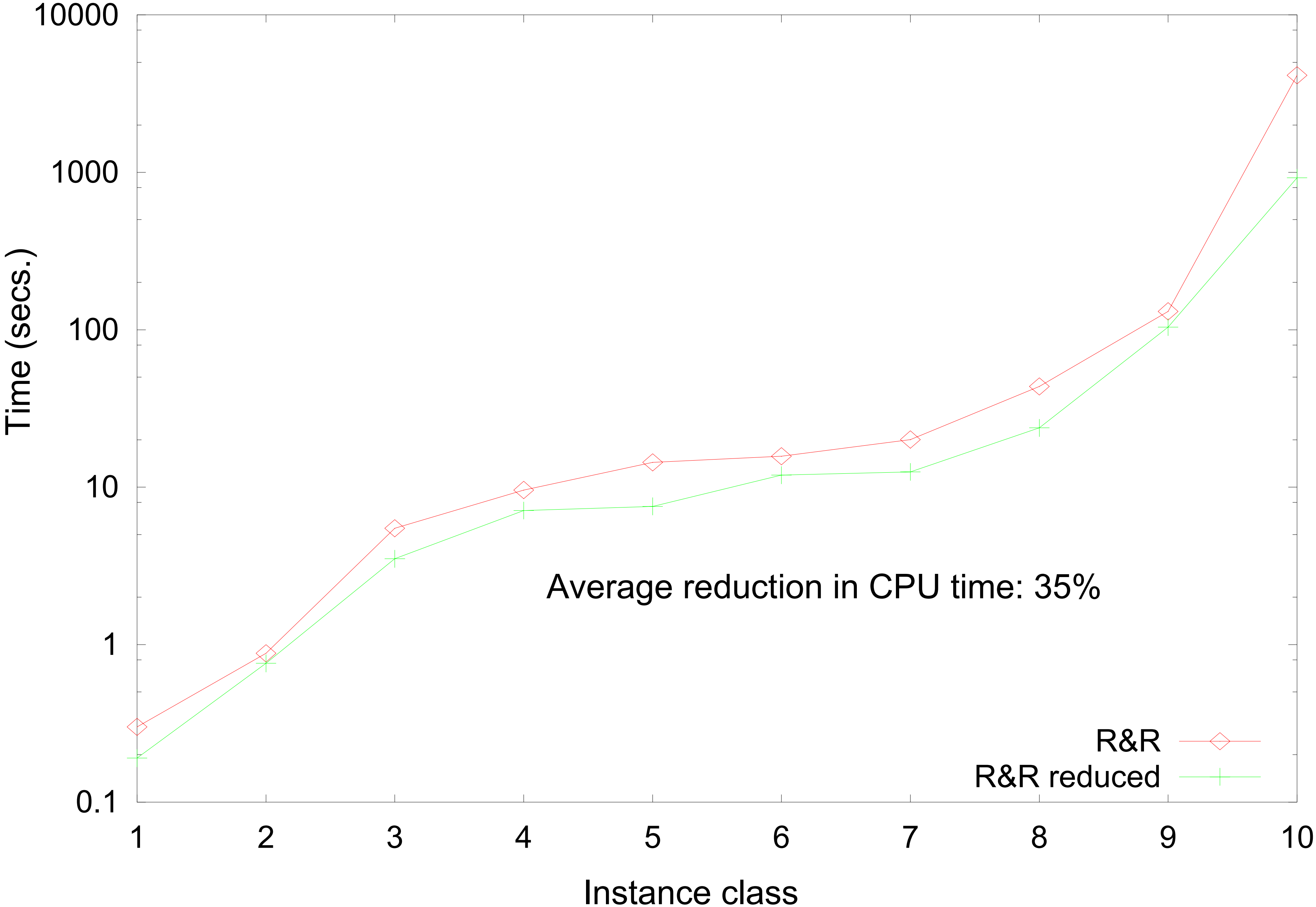
K&T vs avoiding heaps for weight increase on 10 classes of instances



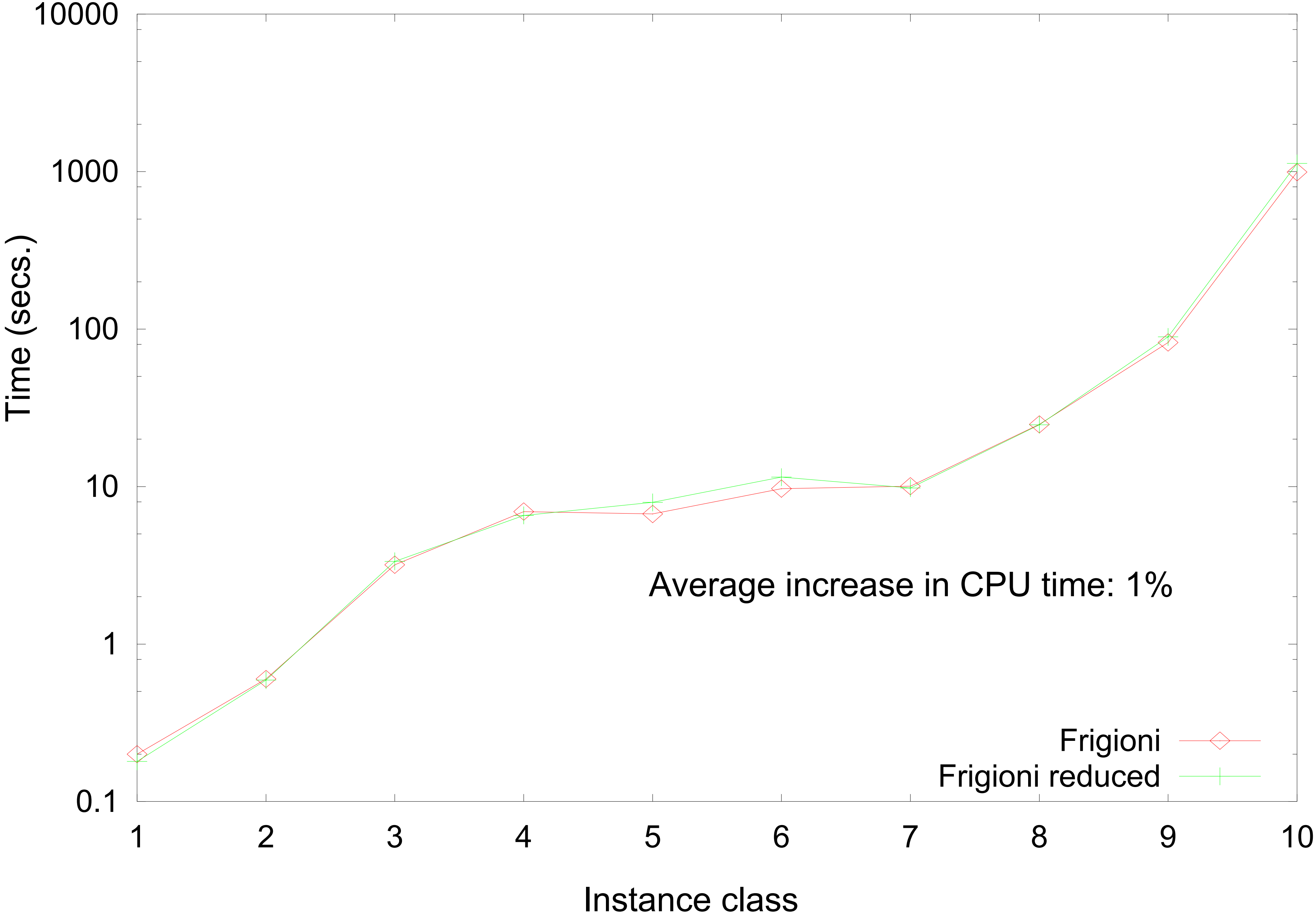
Dijkstra vs K&T avoiding heaps for weight increase on 10 classes of instances

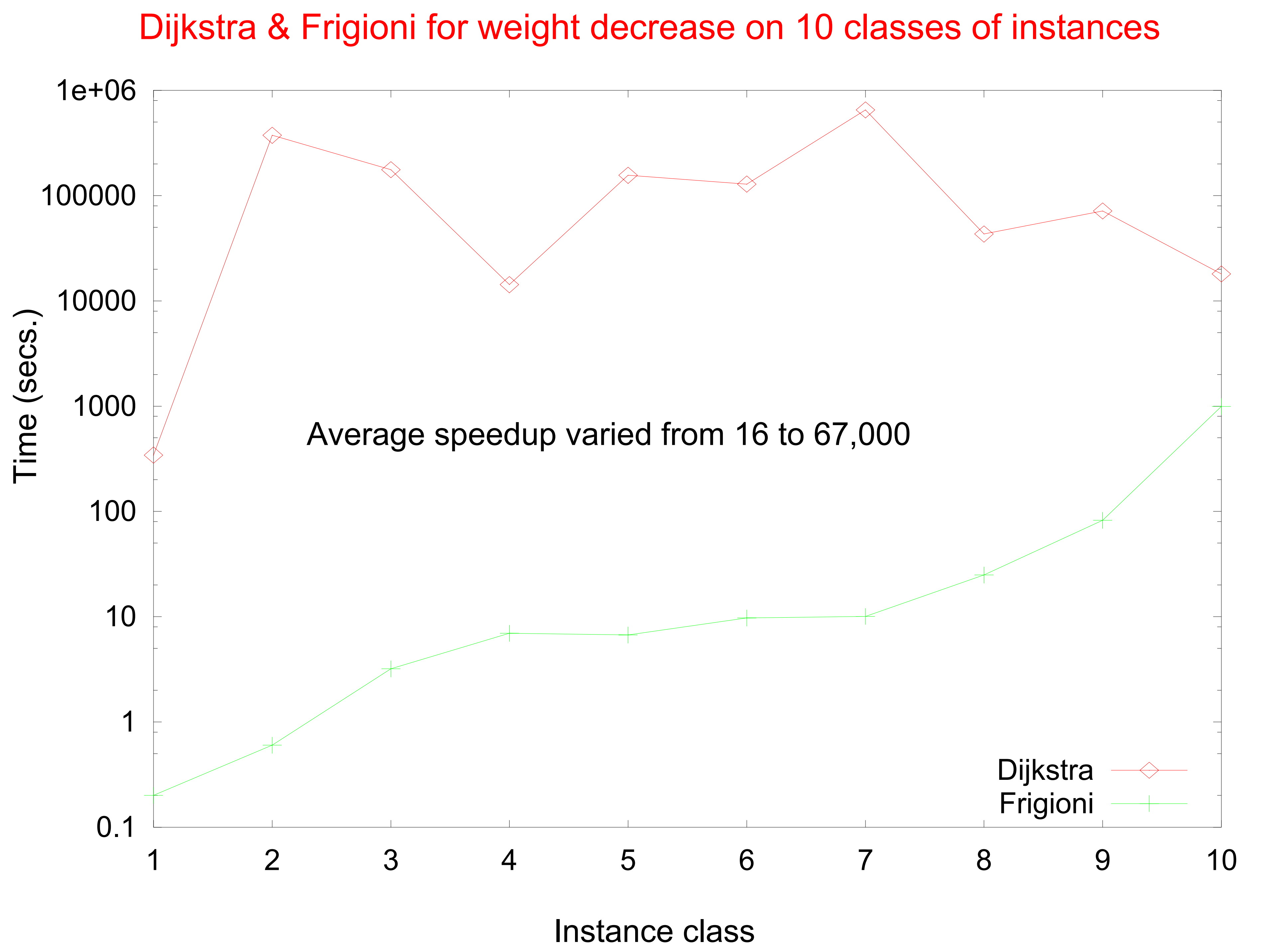


R&R and avoiding heaps for weight decrease on 10 classes of instances



Frigioni and avoiding heaps for weight decrease on 10 classes of instances




Dijkstra & Frigioni for weight decrease on 10 classes of instances
Time (secs.)
1e+06
100000
10000
1000
100
10
1
0.1
Average speedup varied from 16 to 67,000
Dijkstra
Frigioni
1
2
3
4
5
6
7
8
9
10
Instance class

Conclusions

- ☺ Ramalingan & Reps on graphs: avoiding use of heaps reduced CPU time by 31% for weight increase and by 35% for weight decrease;
- ☺ Demetrescu weight increase on trees: avoiding use of heaps reduced CPU time by 30%;
- ☺ King & Thorup weight increase on trees: avoiding use of heaps reduced CPU time by 28%;
- ☹ Frigioni et al. weight decrease on trees: avoiding use of heaps increased CPU time by 1%;

Conclusions

- ☺ Considering unit weight changes, the standard algorithms are 3 times faster if they avoid using heaps;
- The incremental algorithm is 60% faster then the decremental algorithm;
- On average, King & Thorup algorithm is 4% faster then Demetrescu algorithm;
- Updating trees is 6% faster than updating graphs for weight increase and 68% faster for weight decrease.

Local search for OSPF routing

- ☺ For unit increment/decrement the idea of avoiding heaps reduced the computational time by a factor of 3.