Δ-stepping: a parallelizable shortest path algorithm

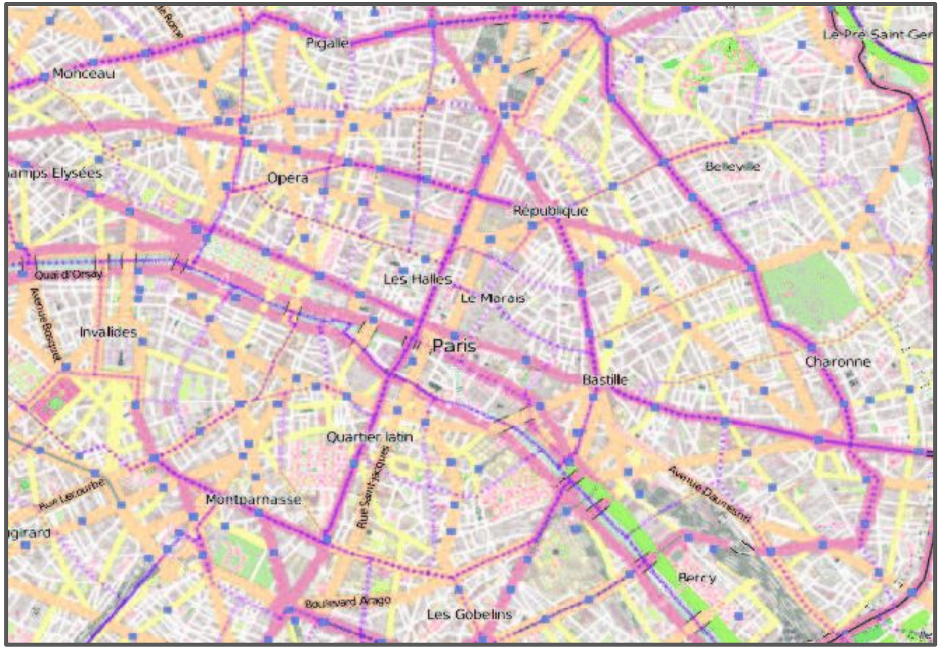
Ulrich Meyer
& Peter Sanders
Presented by Lucas Bautista

Ulrich Meyer
Goethe University Frankfurt

Peter Sanders
Karlsruhe Institute of
Technology

Parallel Single Source Shortest Path (SSSP)

- Processing large graphs quickly is important
- SSSP is a common subroutine
- Betweenness Centrality, Route Finding, generally anything that needs to use the shortest distance between two nodes



SSSP

- Dealing with nonnegative directed graphs
- Label Setting Algorithms
- Label Correcting Algorithms
- Label setting has better worst-case bounds, but label-correcting is often better in practice

Label Setting Algorithms

- The algorithm finalizes the label for one node at a time
- Once a node’s shortest distance is known, it never needs to change.
- The algorithm processes nodes in non-decreasing order of distance.

Example: Dijkstra's algorithm

Dijkstra's algorithm

Start with d[s]=0, For all v != s, d[v] = ∞

Maintain a priority queue of nodes keyed by tentative distance.

- Extract the node u with smallest d[u]
- Its label is now set (and final)
- Relax all outgoing edges (u,v):
  if d[v]>d[u]+w(u,v) then update d[v]
- Repeat until queue is empty.

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

Label Correcting Algorithms

- Nodes’ labels can be revised multiple times
- The algorithm keeps correcting them until no shorter paths exist
- There’s no strict order in which nodes are processed
- We can relax edges arbitrarily and repeatedly.

Example: Bellman Ford Algorithm

Bellman Ford

Initialize d[s]=0, all others d[v]=∞

Repeat n−1 times:

- For every edge (u,v):
  if d[v]>d[u]+w(u,v) then d[v]=d[u]+w(u,v)

Complexity: O(|E|*|V|)

Each iteration “corrects” labels based on previously found paths.

No order for shortest path convergence

Δ-Stepping

- Buckets of vertices grouped by their temporary distance labels
- B[i] contains vertices with labels in [i*Δ, (i+1)*Δ]
- Inner loop deals with relaxing light edges
- Outer edge deals with relaxing heavy edges

```
foreach v ∈ V do tent(v) := ∞
relax(s, 0);                                          (* Insert source node with distance 0     *)
while ¬isEmpty(B) do                                  (* A phase: Some queued nodes left (a) *)
    i := min{j ⩾ 0: B[j] ≠ ∅}                         (* Smallest nonempty bucket (b) *)
    R := ∅                                            (* No nodes deleted for bucket B[i] yet    *)
    while B[i] ≠ ∅ do                                 (* New phase (c) *)
        Req := findRequests(B[i], light)              (* Create requests for light edges (d) *)
        R := R ∪ B[i]                                 (* Remember deleted nodes (e) *)
        B[i] := ∅                                     (* Current bucket empty    *)
        relaxRequests(Req)                            (* Do relaxations, nodes may (re)enter B[i] (f) *)
    Req := findRequests(R, heavy)                     (* Create requests for heavy edges (g) *)
    relaxRequests(Req)                                (* Relaxations will not refill B[i] (h) *)

Function findRequests(V′, kind : {light, heavy}) : set of Request
    return {(w, tent(v) + c(v, w)): v ∈ V′ ∧ (v, w) ∈ E_kind}

Procedure relaxRequests(Req)
    foreach (w, x) ∈ Req do relax(w, x)

Procedure relax(w, x)                                 (* Insert or move w in B if x < tent(w) *)
    if x < tent(w) then
        B[⌊tent(w)/Δ⌋] := B[⌊tent(w)/Δ⌋] \ {w}         (* If in, remove from old bucket *)
        B[⌊x      /Δ⌋] := B[⌊x      /Δ⌋] ∪ {w}         (* Insert into new bucket *)
        tent(w) := x
```

Fig. 1. A sequential variant of Δ-stepping (with cyclical bucket reusage). The sets of light and heavy edges are denoted by E_{light} and E_{heavy}, respectively. Requests consist of a tuple (node, weight).

Bucket Processing

- Each vertex in the bucket has outgoing edges which are either “light” (weight ≤ Δ) or “heavy” (weight > Δ)
- Relaxing an edge can cause the destination vertex to be inserted into the current bucket
- Process bucket until it is empty, then relax its heavy edges

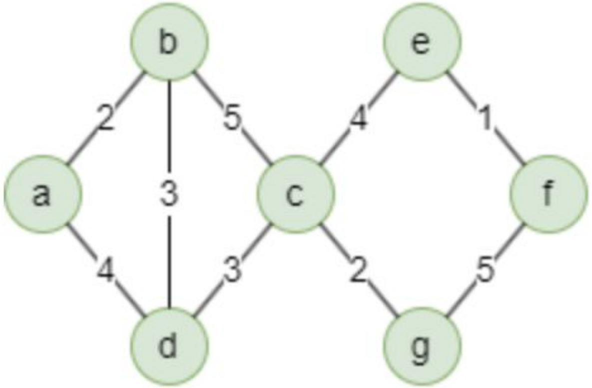
```
foreach v ∈ V do tent(v) := ∞
relax(s, 0);                                   (* Insert source node with distance 0      *)
while ¬isEmpty(B) do                           (* A phase: Some queued nodes left (a) *)
    i := min{j ⩾ 0: B[j] ≠ ∅}                  (* Smallest nonempty bucket (b) *)
    R := ∅                                     (* No nodes deleted for bucket B[i] yet     *)
    while B[i] ≠ ∅ do                          (* New phase (c) *)
        Req := findRequests(B[i], light)       (* Create requests for light edges (d) *)
        R := R ∪ B[i]                          (* Remember deleted nodes (e) *)
        B[i] := ∅                              (* Current bucket empty     *)
        relaxRequests(Req)                     (* Do relaxations, nodes may (re)enter B[i] (f) *)
    Req := findRequests(R, heavy)              (* Create requests for heavy edges (g) *)
    relaxRequests(Req)                         (* Relaxations will not refill B[i] (h) *)

Function findRequests(V′, kind : {light, heavy}) : set of Request
    return {(w, tent(v) + c(v, w)): v ∈ V′ ∧ (v, w) ∈ E_kind}

Procedure relaxRequests(Req)
    foreach (w, x) ∈ Req do relax(w, x)

Procedure relax(w, x)                          (* Insert or move w in B if x < tent(w) *)
    if x < tent(w) then
        B[⌊tent(w)/Δ⌋] := B[⌊tent(w)/Δ⌋] \ {w}   (* If in, remove from old bucket *)
        B[⌊x      /Δ⌋] := B[⌊x      /Δ⌋] ∪ {w}   (* Insert into new bucket *)
        tent(w) := x
```

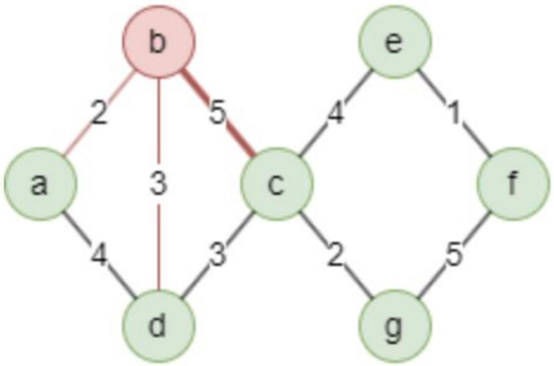
Fig. 1. A sequential variant of Δ-stepping (with cyclical bucket reusage). The sets of light and heavy edges are denoted by E_{light} and E_{heavy}, respectively. Requests consist of a tuple (node, weight).



Node	a	b	c	d	e	f	g
dist()	∞	0	∞	∞	∞	∞	∞

Bucket	Distance Range	Node(s)
B[0]	[0, 3)	b

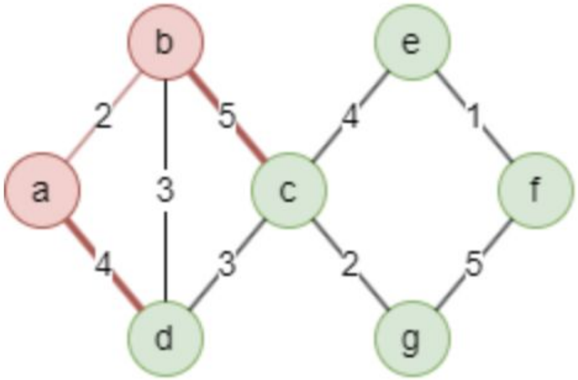
https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp



Node	a	b	c	d	e	f	g
dis()	2	0	∞	3	∞	∞	∞

Bucket	Distance Range	Node(s)
B[0]	[0, 3)	a
B[1]	[3, 6)	d

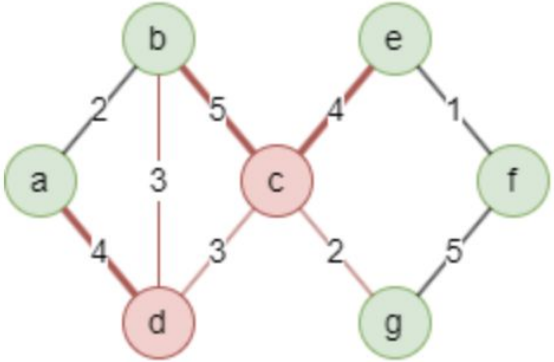
https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp



Node	a	b	c	d	e	f	g
dist()	2	0	5	3	∞	∞	∞

Bucket	Distance Range	Node(s)
B[0]	[0, 3)	
B[1]	[3, 6)	d c

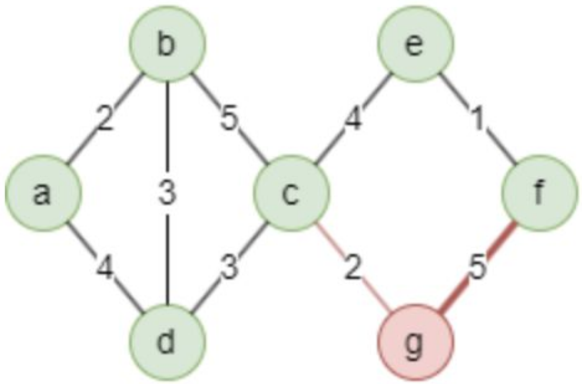
https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp



Node	a	b	c	d	e	f	g
dist()	2	0	5	3	9	∞	7

Bucket	Distance Range	Node(s)
B[0]	[0, 3)	
B[1]	[3, 6)	
B[2]	[6, 9)	g
B[3]	[9, 12)	e

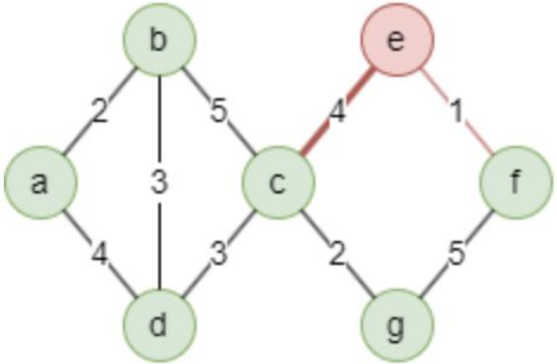
https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp



Node	a	b	c	d	e	f	g
dist()	2	0	5	3	9	12	7

Bucket	Distance Range	Contain Node(s)
B[0]	[0, 3)	
B[1]	[3, 6)	
B[2]	[6, 9)	
B[3]	[9, 12)	e
B[4]	[12, 15)	f

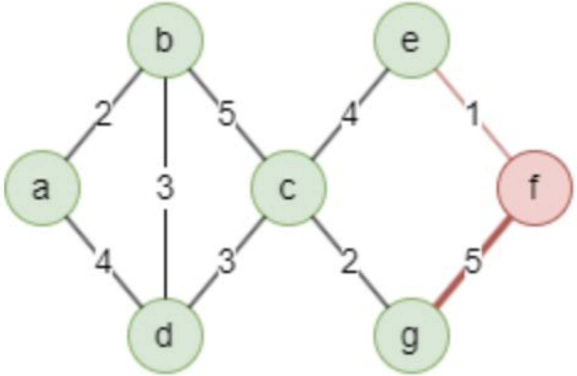
https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp



Node	a	b	c	d	e	f	g
dis()	2	0	5	3	9	10	7

Bucket	Distance Range	Contain Node(s)
B[0]	[0, 3)	
B[1]	[3, 6)	
B[2]	[6, 9)	
B[3]	[9, 12)	f
B[4]	[12, 15)	

https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp



Node	a	b	c	d	e	f	g
dist()	2	0	5	3	9	10	7

Bucket	Distance Range	Node(s)
B[0]	[0, 3)	
B[1]	[3, 6)	
B[2]	[6, 9)	
B[3]	[9, 12)	
B[4]	[12, 15)	

https://www.ultipa.com/docs/graph-analytics-algorithms/delta-stepping-sssp

Δ-Stepping in Parallel

- Each PU is assigned nodes **randomly.** Adjacency lists are organized into heavy and light edges

- Each PU maintains a bucket structure with the queued nodes that it is responsible for

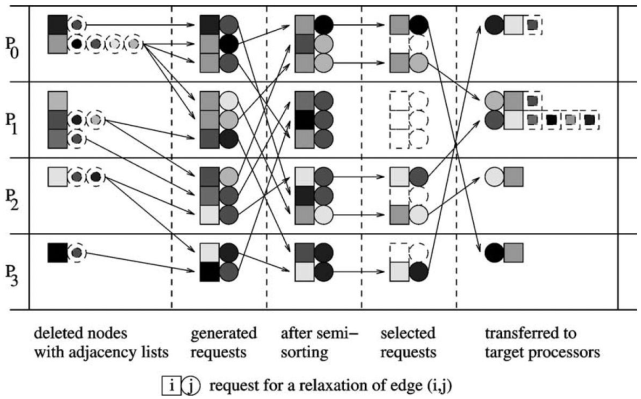
- Every Θ(log n) iterations it is checked whether any PU has found a nonempty bucket and, if so, the globally smallest index with a nonempty bucket is found, is processed.

- Each PU scans its request buffer and sequentially performs the relaxations assigned to it. Since no other PUs work on its nodes the relaxations will be atomic.

Δ-Stepping in Parallel

- Instead of one PU handling all edges of a large node, multiple PUs cooperate to generate requests for its neighbors.
- If node u has 10 million outgoing edges → split its adjacency list across 4 PUs; each generates part of the requests.
- The requests are grouped (batched) by which PU they'll be sent to (to reduce communication overhead) -> semisorted
- Before sending, we can filter out duplicates or non-improving requests locally to reduce communication overhead.

Fig. 6. Load balanced edge relaxation using semi-sorting.



Delta Stepping in a Distributed Setting

- The hash function ind(w) replaces the index array used in the PRAM algorithm.
- Processors are assigned to groups of size 2^i . these groups share adjacency lists.
- The group cooperates via collective communication (broadcasts, reductions)

Choosing Δ

- Choosing Δ = 1, makes our problem basically Dijkstra (with an array instead of a heap) (which has better runtime for sparsely bounded degrees)
- Choosing Δ >= n, makes our problem Bellman Ford
- We want to choose Δ so that we can take advantage of the parallelism but also want to reduce the amount of unnecessary work we do

Sequential Analysis

- Sequential Δ-stepping can be implemented in $O(n + m + L/\Delta + n\Delta + m\Delta)$

- On random edge weights $O(n + m + d * L)$

- $L := \max\{dist(v): dist(v) < \infty\}$

Parallel Analysis

- Simple parallelization runs in O(L/Δ * d * l_Δ * logn)

- Can take advantage of shortcut edges

$$l_\Delta = 1 + \max_{\langle u,v\rangle \in C_\Delta} \min\left\{|A|:\ A = (u, \ldots, v) \text{ is a minimum-weight } \Delta\text{-path}\right\}.$$

Performance

- The tested graphs ranged from 10^3 up to 10^6 nodes and comprised up to 3 * 10^6 edges
- Tests were run on an INTEL Paragon with 16 processors
- Parallel/Distributed: for n = 2^19 nodes and d = 3, speedup **9.2** was obtained against the sequential Δ-stepping approach
- Sequential Δ-stepping approach is **3.1** times faster than an optimized implementation of Dijkstra's algorithm
- Results on dense graphs are slightly worse [Communication costs]

Strengths, Weaknesses, Thoughts

Strengths:

- I liked the connection between Dijkstra & Bellman Ford (and how delta stepping is something in the middle)
- Load balancing requests among processors

Weaknesses:

- Slightly hard to follow proofs

Future work:

- Better load balancing strategies
- Better use of communication (lower messages)
- Maybe extending it to negative edges as well using ideas from Bellman Ford

Discussion

- The experiments show O(logn) phases even though theory guarantees only O(log^2(n)/loglogn)
- Why might the empirical performance be better than the worst-case bound?
- Is there a better way to choose Δ, depending on the graph we are processing