KickStarter

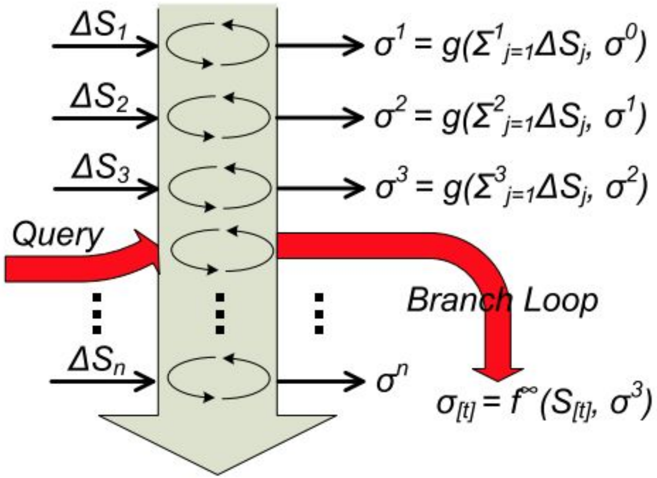
Edward Park

Background

- Streaming Graphs
  - Super common - social networks, real-time traffic info, Web graph
- Processed via incremental algorithms
  - Tornado, Kineograph, Stinger, Naiad
- Why do we need KickStarter?

Other Streaming Graph Frameworks

- Does batch updates
- Maintains intermediate approximate results
- When a query arrives, start at the most recent intermediate result and do the rest of the calculations needed (in a branch loop)
- Makes sense: the values right before the updates are a better approximation of the actual results



Problems?

- Is that true? Will values right before an update actually be a good approximation of the actual results?
- Not necessarily - especially for edge deletions
- We deal with monotonic computations
  - Calculating some data to a vertex that only ever increases / only ever decreases
  - SSSP, BFS, Clique, Label Propogation
- Edge deletions break monotonicity - invalidate intermediate values

- (Edge deletions are common in real world graphs!)

Problems?

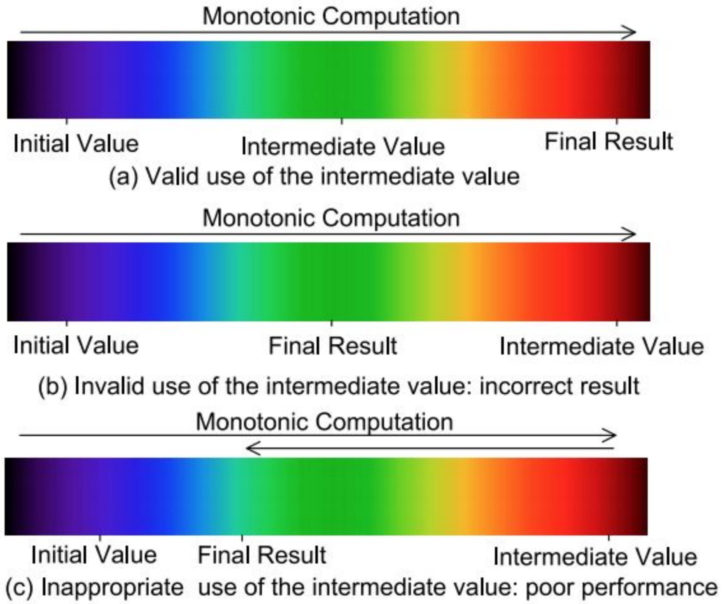
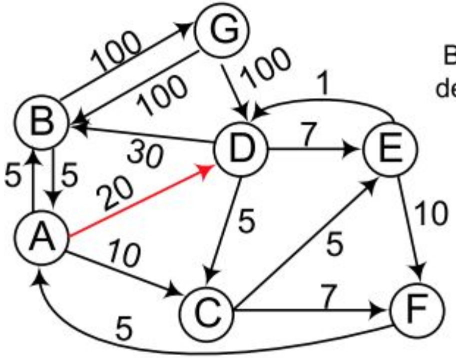


Figure 1: Three different scenarios *w.r.t.* the use of intermediate values after an edge update.

- Breaking monotonicity can have two results:
  - Incorrect results
  - Poor performance

- Let's try it out!

Problem 1: Incorrectness with SSWP



(a) A simple graph.

	A	B	C	D	E	F	G
Before deletion →	∞	20	10	20	7	7	20
A→D deleted							
	∞	20	10	20	7	7	20
Correct Result							
	∞	5	10	5	5	7	5

(b) Computation using intermediate values after the deletion of A→D; shown in blue are the correct result.

A	B	C	D	E	F	G
∞	20	10	20	7	7	20
A→D deleted						
∞	20	10	0	7	7	20
∞	20	10	0	5	7	20
∞	20	10	20	5	7	20
∞	20	10	20	7	7	20

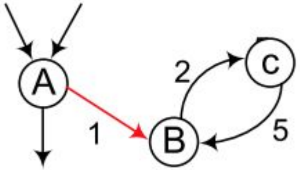
(c) Computation using the initial value for D; values changed are in red.

(b) After deleting the edge, the values are clearly incorrect

(c) After deleting the edge, try setting the value for D back to the initial value - however, still incorrect

Problem 2: Slowness with SSSP

- The deletion of the edge renders B and C disconnected from the rest of the graph
- Each iteration will bump up the values of B and C - takes forever to reach the (correct) value of MAX



(a) A simple graph.

	A	B	C
Before deletion →	5	6	8
	A→B deleted		
	5	6	8
	5	13	8
	5	13	15
	5	20	15
	...	...	...
	5	MAX	MAX

(b) Computation using the approximation after the deletion of A→B.

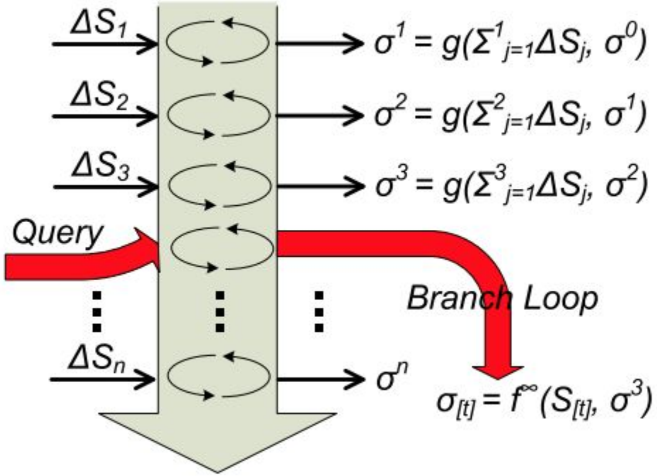
Figure 6: While using the intermediate value for vertex B yields the correct result, the computation can be very slow; the initial value at each vertex is a large number MAX.

When Incorrect? When Slow?

- Key difference is vertex update function
- In the first type, the update function only performs value selection
  - No computation is done
  - A value is propagated along a cycle
  - Incorrect result
- In the second type, the update function does some computation
  - Disallows cyclic propagation
  - Algorithm will eventually stabilize at the right value

KickStarter - How do we fix this problem?

- Edge additions are fine - edge deletions are dangerous
- Right after an execution is forked for an edge deletion, we add a trimming phase
- Unsafe vertex values are adjusted before feeding it into the forked execution
    - What is unsafe? Values that were dependent on the deleted edge



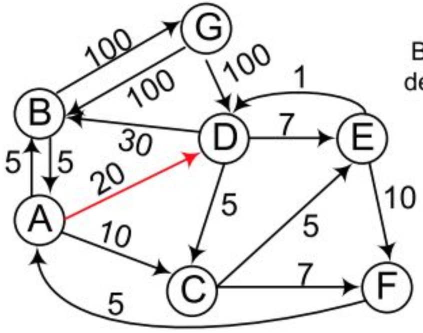
Two Trimming Methods

- Method 1 - Tagging + Resetting
  - Identifies the set of vertices possibly affected by an edge deletion
  - These vertices are resetted back to the initial value
  - Guarantees safety with conservative trimming - however, slow
- Method 2 - Active Value Dependence Tracking
  - Tracks dynamic dependencies among vertices online
  - Leads to a much smaller set of affected vertices
  - The vertices are reset to a closer (safe) approximation instead of the initial value

Method 1 - Tagging + Resetting

- Upon a deletion, the target vertex of the deleted edge is tagged using a set bit
- This tag is iteratively propagated - when an edge is processed where the source is tagged, the target is also tagged
- To reduce # of tagged vertices, rely on algorithmic insight - tag a vertex only if any of its in-neighbors that actually contributes to its current value is tagged
  - In a typical monotonic algorithm, the value of a vertex is typically only computed from a single incoming edge
- “Passive” technique - tagging is only performed upon edge deletion

Method 1 - Tagging + Resetting



(a) A simple graph.

	A	B	C	D	E	F	G
Before deletion	∞	20	10	20	7	7	20
A→D deleted							
	∞	<u>20</u>	10	<u>20</u>	<u>7</u>	7	<u>20</u>
Correct Result							
	∞	5	10	5	5	7	5

(b) Computation using intermediate values after the deletion of A→D; shown in blue are the correct result.

- Tagging conservatively tags every single vertex
- Tagging only the edges that actually contribute no longer tags A and C

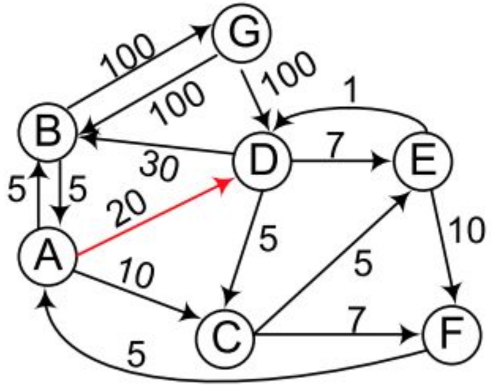
Method 2 - Active Value Dependence Tracking

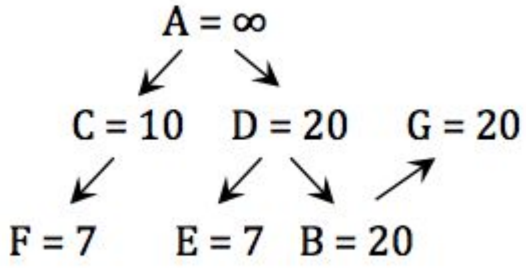
- Define a transitive, non-cyclic relation → that captures value dependencies
  - Say that u → v if there is an edge from u to v and u actually contributes to v
  - Transitive: if u → v and v → w, then u → w
  - Non-cyclic: if u → v, then v does NOT → u
- Why non-cyclic? Need to guarantee safety - if we need a safe approximate value for v, we can’t rely on any neighbor that was dependent on a past value of v
- This contributes-to relation needs to be defined by the developer (simple in practice)

Method 2 - Active Value Dependence Tracking

- Create a dependence tree
  - Acyclic
  - Every vertex has at most one incoming edge
- Non-accumulative - if a new value for a vertex is computed, that dependence replaces the old one

Now, KickStarter needs to compute new approximate values for the vertices affected by the deletion.





Method 2 - Active Value Dependence Tracking

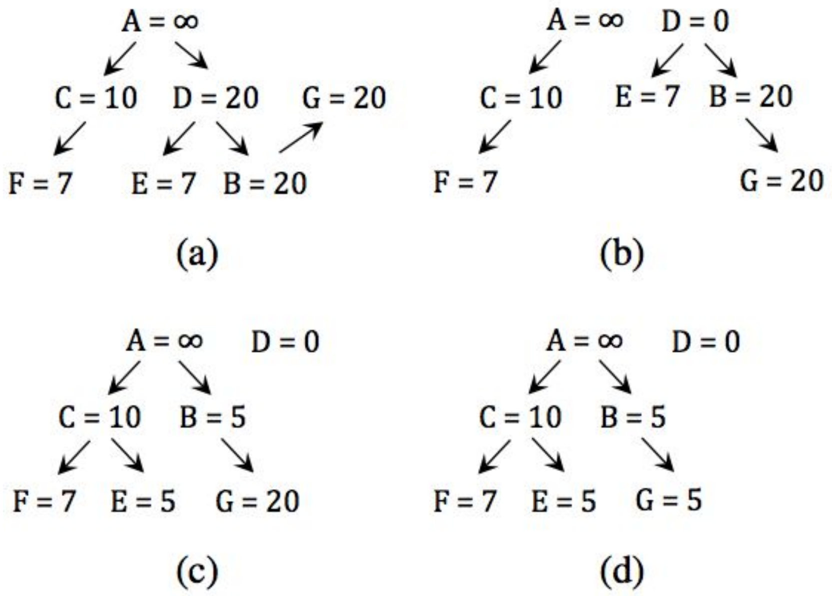
1) Identify the set of vertices affected
   a) This is the subtree rooted at the target vertex of the deleted edge
   b) Ignore all other vertices
2) For each affected vertex v, compute a safe alternative value
   a) Resetting to the initial value also OK but slower
   b) Re-execute the update function on v - however, CANNOT use any edge that was reliant on past values of v (i.e. any vertex lower than v in the dependence tree)
   c) How to quickly determine this? Use the level (depth) information in the dependence tree. Only consider edges that come from vertices whose level is <= level of v
3) Keep trimming if this safe alternative value disrupts monotonicity
   a) What direction is the monotonicity? Depends on the algorithm - in SSWP, values are monotonically increasing, in SSSP they are monotonically decreasing
   b) If the alternative value disrupts monotonicity, there might be something wrong with the children - have to trim those too

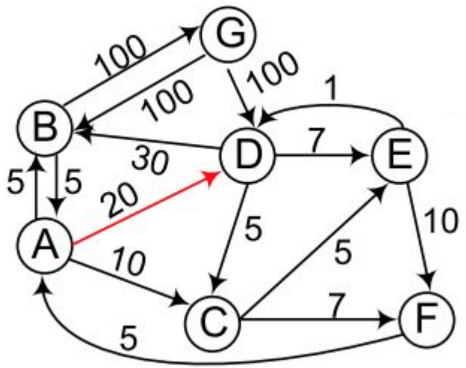
Method 2 - Active Value Dependence Tracking

(b) There are no safe incoming edges into D - thus, new value is 0. Monotonicity is disrupted.

(c) E gets incoming edges from C and D. B gets incoming edges from A and D

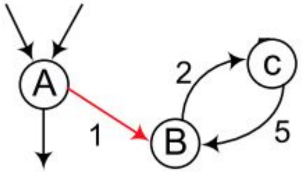
(d) Trimming continues to G





Method 2 - Active Value Dependence Tracking

- Good for performance too
- Look at previous SSSP problem - after edge is deleted, B’s value is immediately set to MAX (and C soon follows)
- Easy to parallelize too - computations are confined to a vertex and its neighbors



(a) A simple graph.

	A	B	C
Before deletion →	5	6	8
	A→B deleted		
	5	6	8
	5	13	8
	5	13	15
	5	20	15
	...	...	...
	5	MAX	MAX

(b) Computation using the approximation after the deletion of A→B.

Experimental Results

Graphs	#Edges	#Vertices
Friendster (FT) [13]	2.5B	68.3M
Twitter (TT) [6]	2.0B	52.6M
Twitter (TTW) [20]	1.5B	41.7M
UKDomain (UK) [5]	1.0B	39.5M
LiveJournal (LJ) [3]	69M	4.8M

Table 2: Real world input graphs.

- TAG = tagging + resetting
- VAD = value dependence trimming
- TOR = Tornado. Is not always correct.
- RST = no trimming, only resetting. Is always correct.
- 10% of edge updates are deletions

Algorithm	Issue	VERTEXFUNCTION	SHOULDPROPAGATE
SSWP	Correctness	$v.path \leftarrow \max_{e \in \text{inEdges}(v)} (\min(e.source.path, e.weight))$	$newValue < oldValue$
CC	Correctness	$v.component \leftarrow \min(v.component, \min_{e \in \text{edges}(v)} (e.other.component))$	$newValue > oldValue$
BFS	Performance	$v.dist \leftarrow \min_{e \in \text{inEdges}(v)} (e.source.dist + 1)$	$newValue > oldValue$
SSSP	Performance	$v.path \leftarrow \min_{e \in \text{inEdges}(v)} (e.weight + e.source.path)$	$newValue > oldValue$

Table 3: Various vertex-centric graph algorithms.

Trimming for Correctness

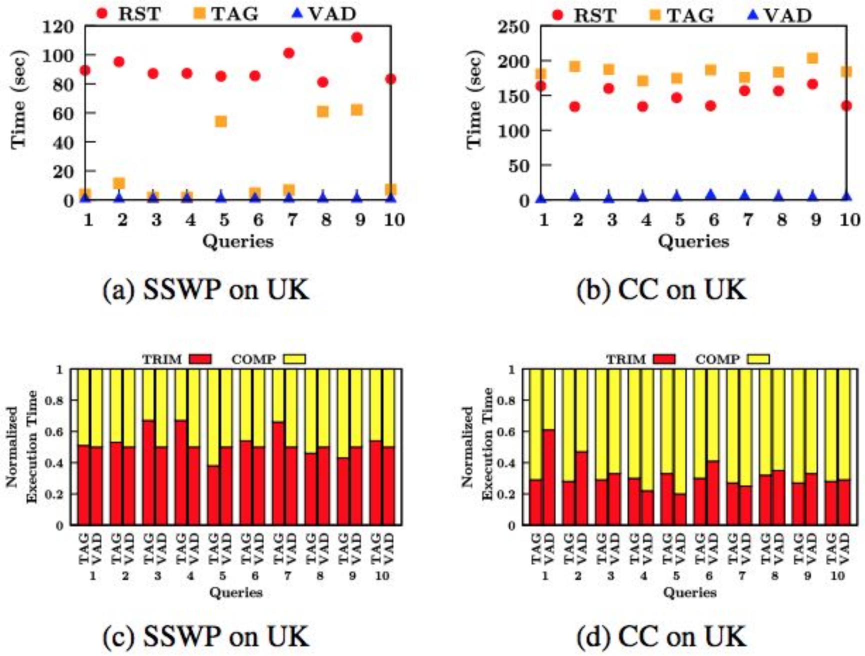
		LJ	UK	TTW	TT	FT
SSWP	RST	7.48-10.16 (8.59)	81.22-112.01 (90.75)	94.18-102.27 (99.28)	170.76-183.11 (176.87)	424.46-542.47 (487.04)
	TAG	11.57-14.71 (13.00)	1.73-62.1 (21.42)	27.38-125.91 (71.26)	262.88-278.42 (270.29)	474.64-550.25 (510.52)
	VAD	3.51-5.5 (4.48)	1.17-1.18 (1.17)	21.54-34.38 (27.55)	66.85-130.84 (75.88)	113.3-413.51 (143.72)
CC	RST	6.43-7.93 (7.19)	133.92-166.33 (148.80)	105.16-111.46 (107.54)	113.92-126.35 (126.35)	212.43-230.26 (221.05)
	TAG	10.98-12.81 (11.86)	170.91-203.54 (183.93)	176.91-201.12 (185.84)	193.77-249.93 (208.90)	331.79-386 (360.34)
	VAD	4.89-5.85 (5.30)	1.81-7.75 (4.37)	31.78-33.24 (32.54)	21.98-22.58 (22.29)	38-39.36 (38.56)

Table 4: Trimming for correctness: query processing time (in sec) for SSWP and CC, shown in the form of min-max (average).

		LJ	UK	TTW	TT	FT
SSWP	TAG	3.1M-3.2M (3.1M)	8.9K-9.7M (4.1M)	1.1K-29.5M (13.4M)	28.5M-28.6M (28.6M)	49.5M-49.5M (49.5M)
	VAD	20.1K-90.8K (60.8K)	2.9K-93.0K (33.4K)	1.0K-4.5K (2.3K)	2.4K-1.1M (106.4K)	20.7K-13.6M (1.3M)
CC	TAG	3.2M-3.2M (3.2M)	25.9M-25.9M (25.9M)	31.3M-31.3M (31.3M)	32.2M-32.2M (32.2M)	52.1M-52.1M (52.1M)
	VAD	1.1K-3.1K (1.9K)	320-1.6K (1.0K)	116-463 (212)	241-463 (344)	294-478 (374)

Table 5: Trimming for correctness: # reset vertices for SSWP and CC (the lower the better) in the form of min-max (average).

Trimming for Correctness



(a) SSWP on UK

(b) CC on UK

(c) SSWP on UK

(d) CC on UK

Figure 8: Time taken to answer queries.

Trimming for Performance

		LJ	UK	TTW	TT	FT
SSSP	**TOR**	1.27-102.88 (39.10)	2.84-119.03 (24.90)	17.62-131.9 (112.57)	42.13-584.64 (190.78)	90.59-179.83 (163.99)
	TAG	3.25-4.49 (3.97)	2.03-2.94 (2.19)	46.06-52.5 (48.96)	98.59-118.23 (105.73)	131.22-150.16 (142.60)
	VAD	2.12-3.22 (2.55)	1.33-1.5 (1.41)	28.68-32.33 (30.21)	41.35-48.65 (44.19)	93.74-101.67 (97.22)
BFS	**TOR**	1.17-77.05 (7.17)	1.24-588.09 (142.55)	23.94-1015.76 (199.23)	55-283.71 (120.45)	190.52-2032.38 (881.17)
	TAG	3.47-4.43 (3.88)	1.81-5.14 (1.97)	51.08-58.3 (54.36)	110.75-192.71 (127.54)	143.21-334.07 (166.60)
	VAD	1.96-3.37 (2.59)	1.21-3.88 (1.42)	32.02-34.86 (32.96)	69.43-91.88 (74.27)	107.4-136.73 (114.56)

Table 6: Trimming for performance: query processing times (in sec) for SSSP and BFS in the form: min-max (average).

		LJ	UK	TTW	TT	FT
SSSP	**TAG**	8.2K-59.8K (25.9K)	4.1K-193.4K (36.4K)	19.7K-183.7K (89.4K)	6.2K-196.7K (51.5K)	19.8K-31.2K (25.4K)
	VAD	1.7K-40.1K (7.0K)	2.9K-52.2K (16.6K)	2.1K-77.7K (19.6K)	836-110.9K (11.1K)	4.5K-12.5K (8.0K)
BFS	**TAG**	10.8K-354.5K (79.0K)	1.3K-483.0K (35.5K)	20.9K-1.2M (457.6K)	44.2K-8.6M (1.1M)	19.1K-4.5M (469.8K)
	VAD	5.5K-116.6K (36.4K)	3.2K-469.9K (41.2K)	860-3.1K (1.6K)	742-1.4K (1.1K)	2.7K-5.2K (3.4K)

Table 7: Trimming for performance: number of reset vertices for SSSP and BFS in the form: min-max (average).

Trimming for Performance

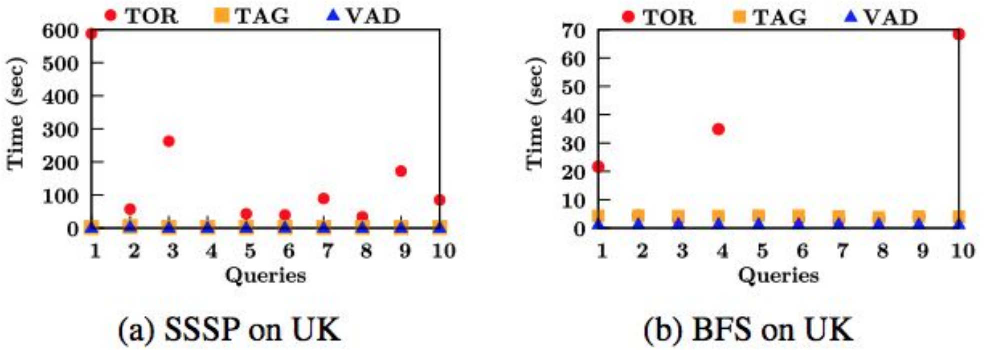


Figure 9: Trimming for performance: time taken to compute answer queries by **TAG** and **VAD**.

Experimental Results

- KickStarter always produces correct results
- Much faster - speedup of 8.5 - 23.7
- Computing new approximate values in VAD drastically reduces # of reset vertices
- TAG is typically slower than VAD because it resets more vertices
- Dependence tracking overhead is only 13%