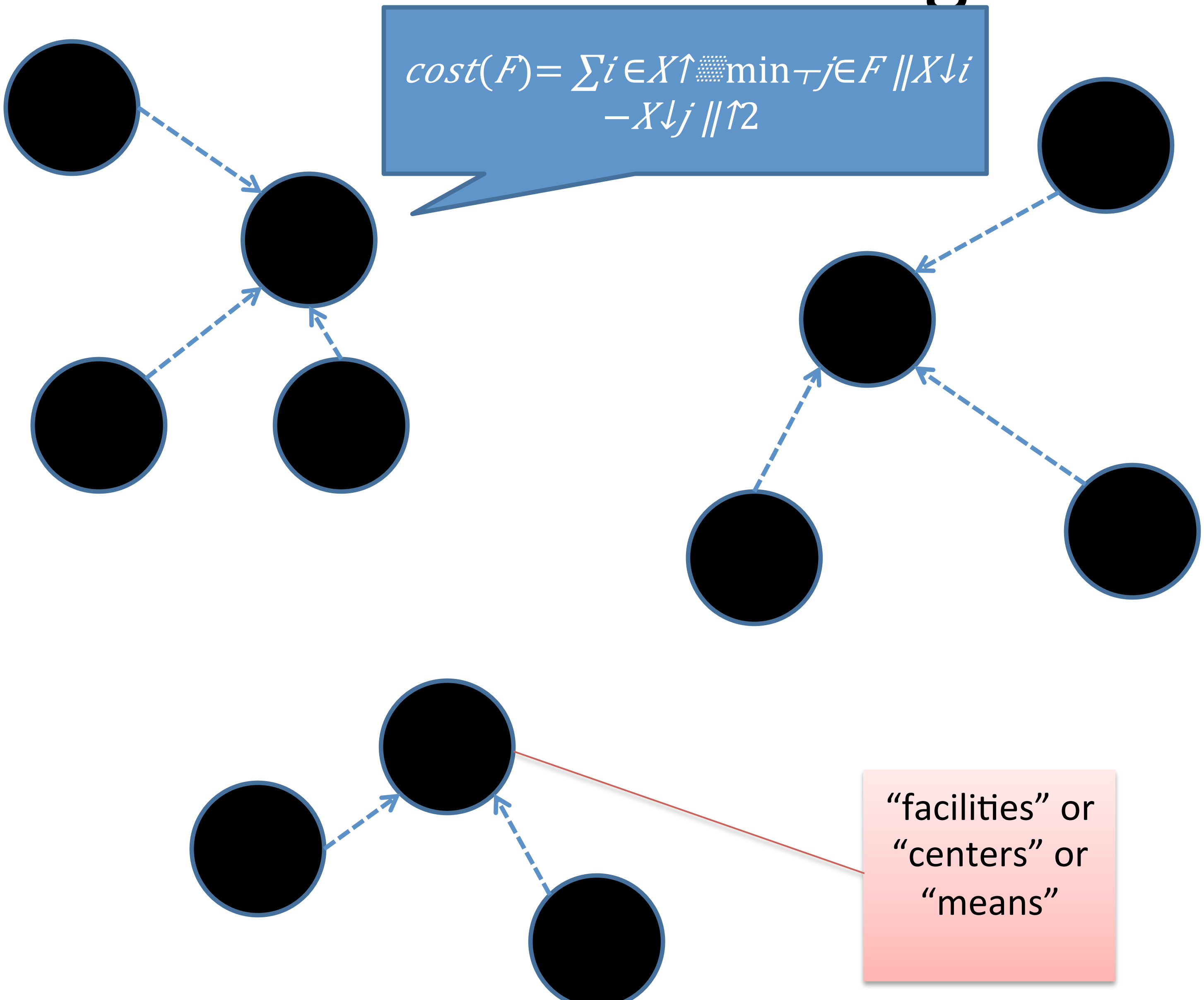
Fast and Accurate *k*-means for Large Data Sets

Michael Shindler Alex Wong Adam Meyerson

Oregon State University UCLA Google

K-means Clustering



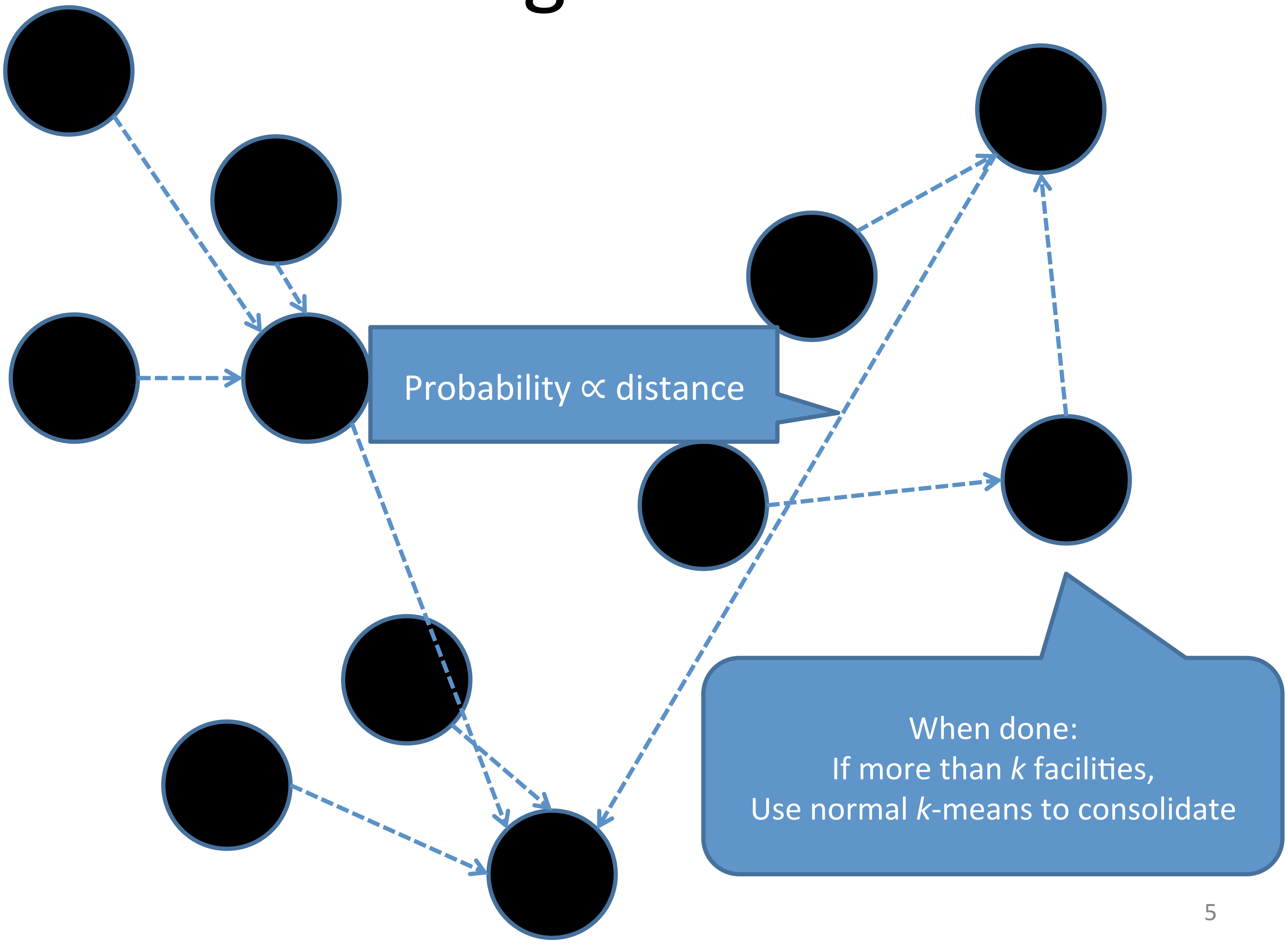
Algorithms for solving *k*-means

- Standard Algorithm (Lloyd 57)
  - Can have cost arbitrarily worse than optimal (Arthur and Vassilvitskii, 07)
  - Can take exponential time (Vattani, 11)
- Polynomial time algorithms for *k*-means
  - Bound ratio of (algorithm cost) / (optimal cost)
  - Best ratio is $9+\varepsilon$ due to (Kanungo *et al*, 02)
- These do not work for streaming setting

K-means for Large Datasets

- Want good *k*-means solution
  - Without random access to full data
  - Without using much memory
  - Without using much time

Streaming *k*-means



Improvements/Contributions

	Braverman *et al* (SODA 2011)	**This Work**
Memory Requirement	1623 $k \log n$	Any $\Omega(k \log n)$ (including *1* $k \log n$)
Cost Guarantee (cost ratio against best)	O(1) 60,498	O(1) 17
If too many facilities before finishing stream?	Complicated matching	Simple re-evaluation
Optimized runtime	$O(nk \log n)$ Large lead constant	$o(nk)$ Less than $\theta(nk)$

More relevant algorithms for streaming *k*-means

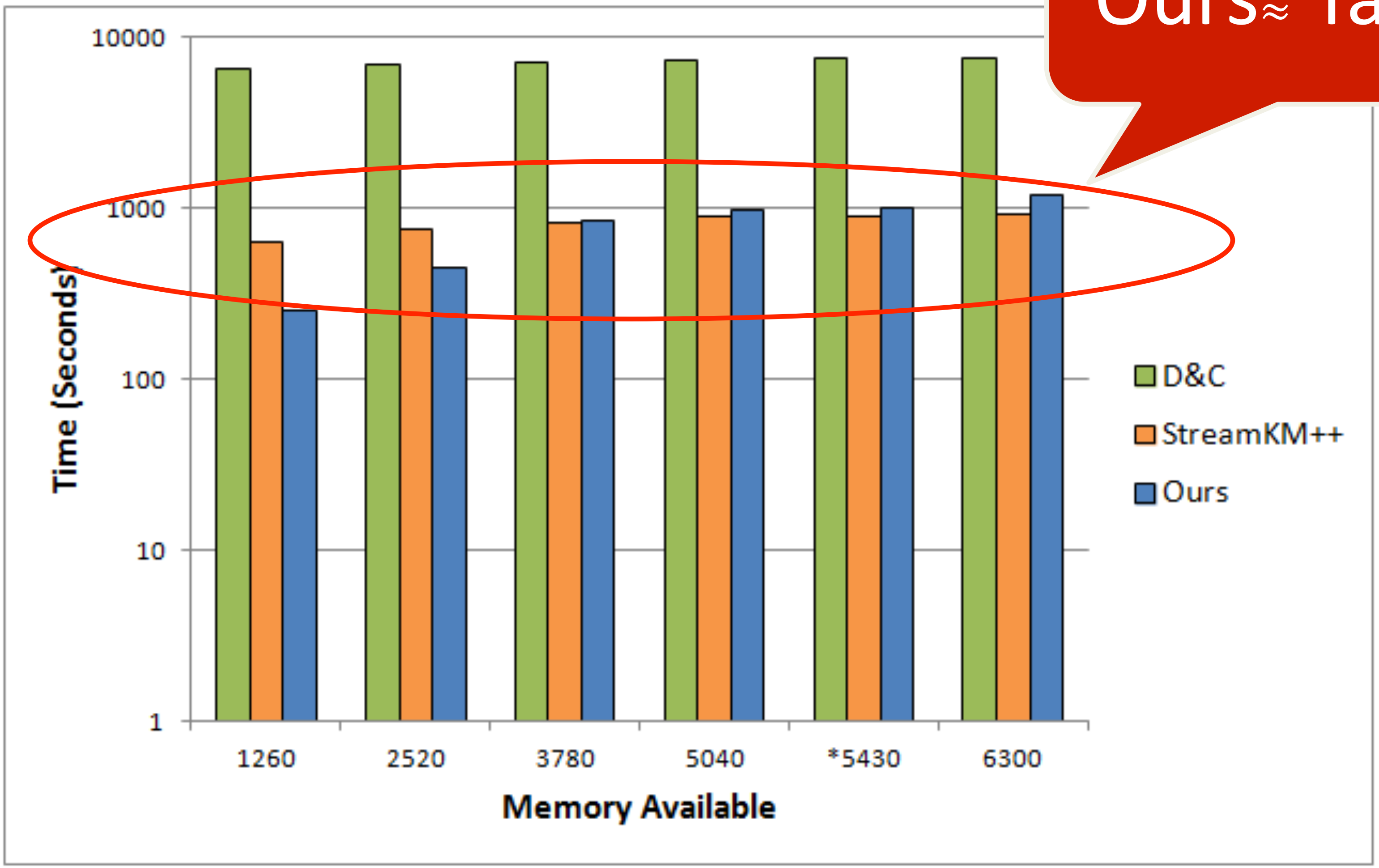
- Divide and Conquer (Ailon, Jaiswal, and Monteleoni, NIPS 09)
  - Read *M* points into memory
  - Compute and store weighted representative points
  - Repeat until stream exhausted
  - Compute *k*-means on stored representatives
- StreamKM++ (Ackermann *et al*, ALENEX 10)
  - Compute a weighted representative sample of stream
  - Solve *k*-means on sample
  - Based on *core set* paradigm
    - For current best theoretical treatment, see (Chen 09)

Experimental Setup

- Compare to others with equal memory
- Metrics:
  - Cost of solution (squared error)
  - Time to compute solution
- Examples in this talk are from UCI “Census 1990” dataset
  - 2,458,285 points in 68 dimensions
  - Seeking $k=12$ clusters

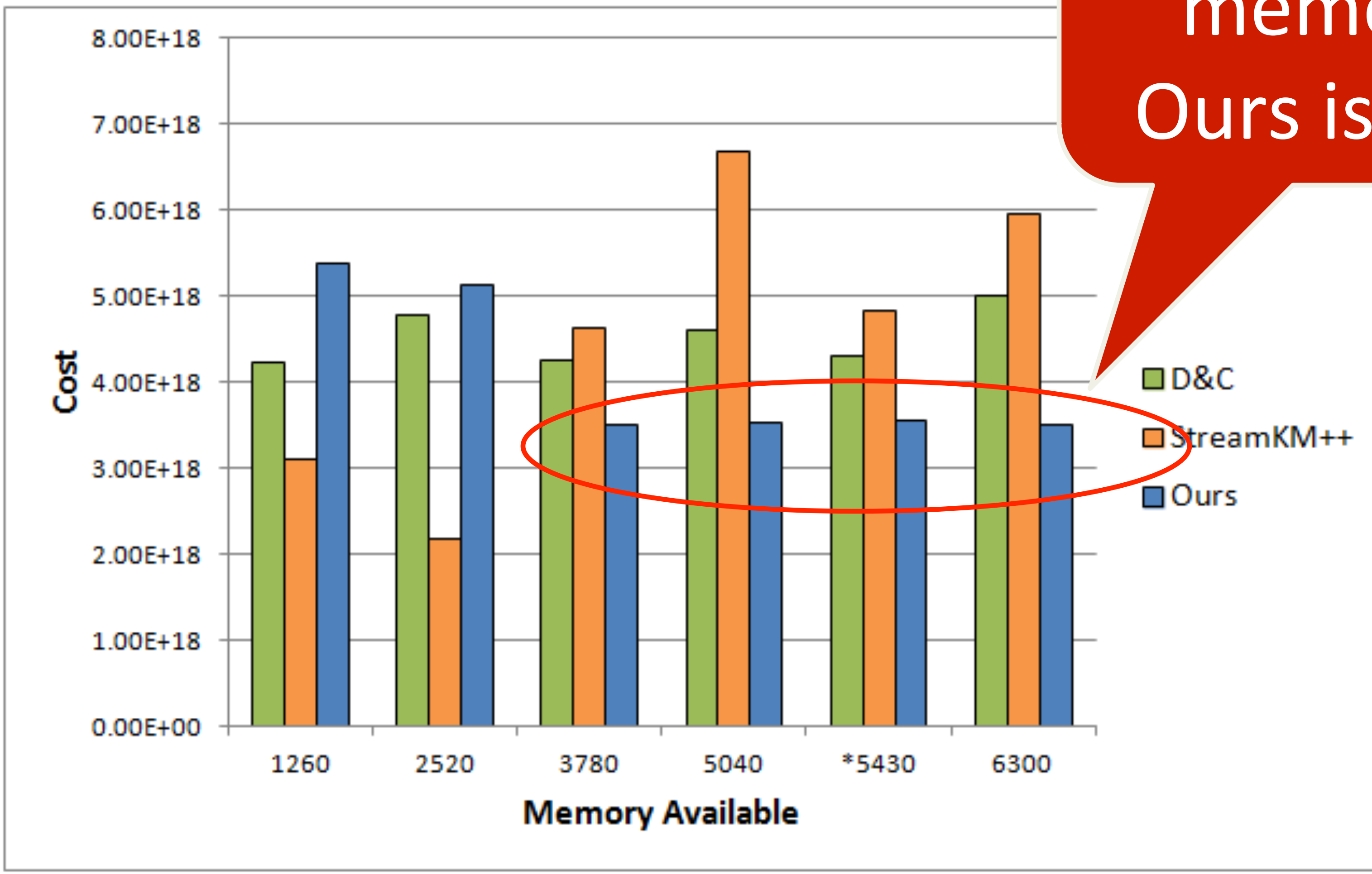
Time to Compute Solution

Ours ≈ fastest

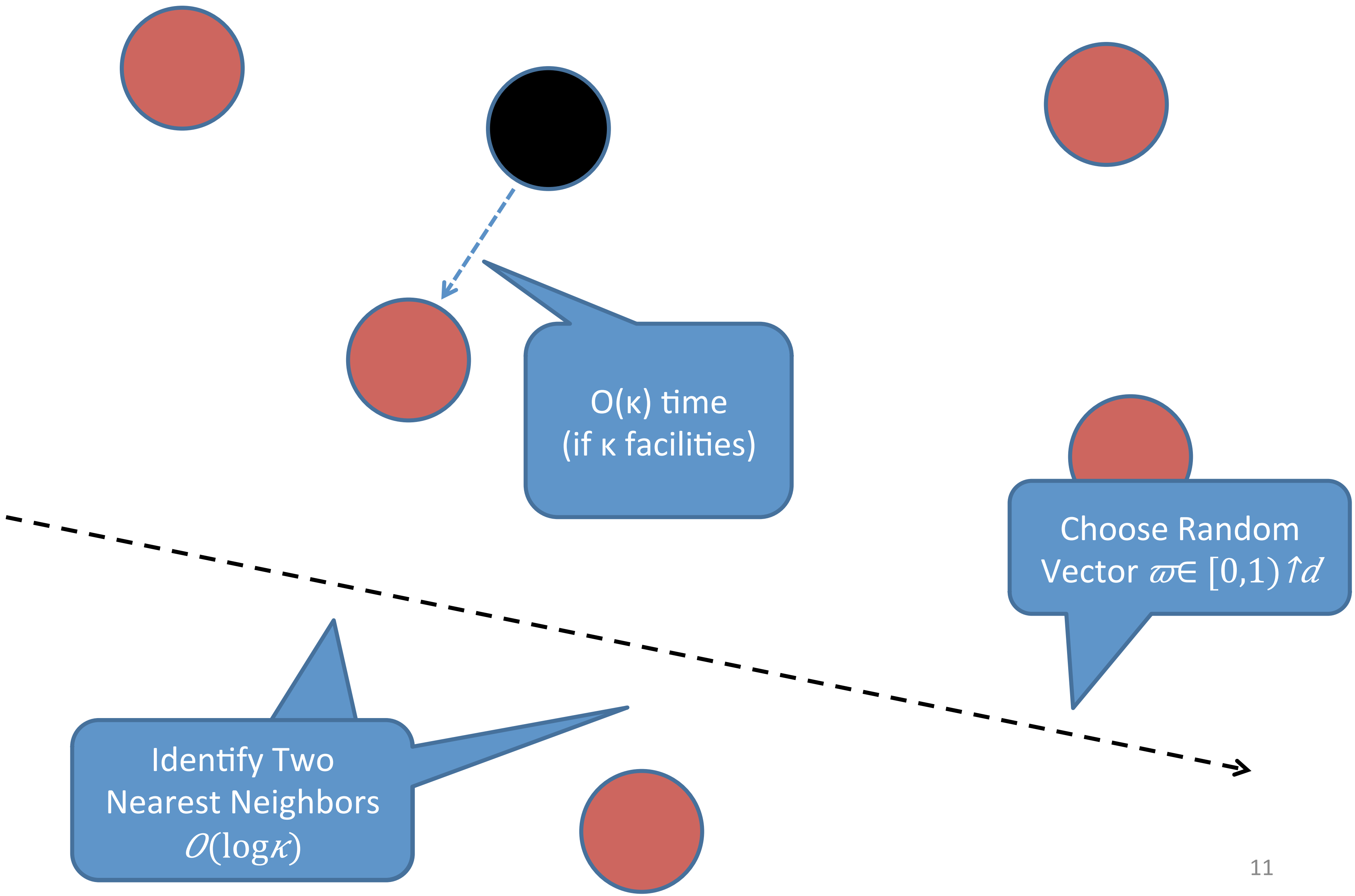


Cost (Summed Squared

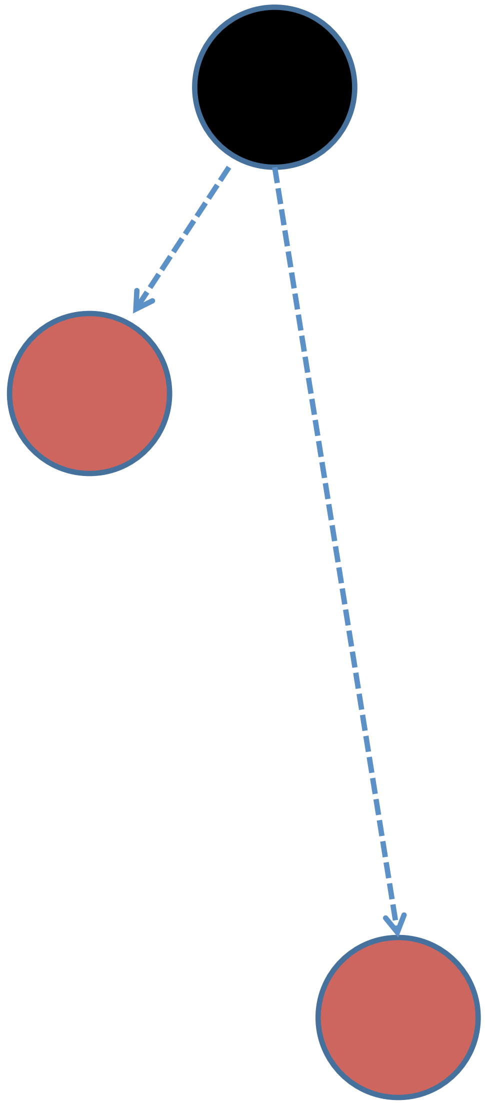
With enough memory, Ours is best



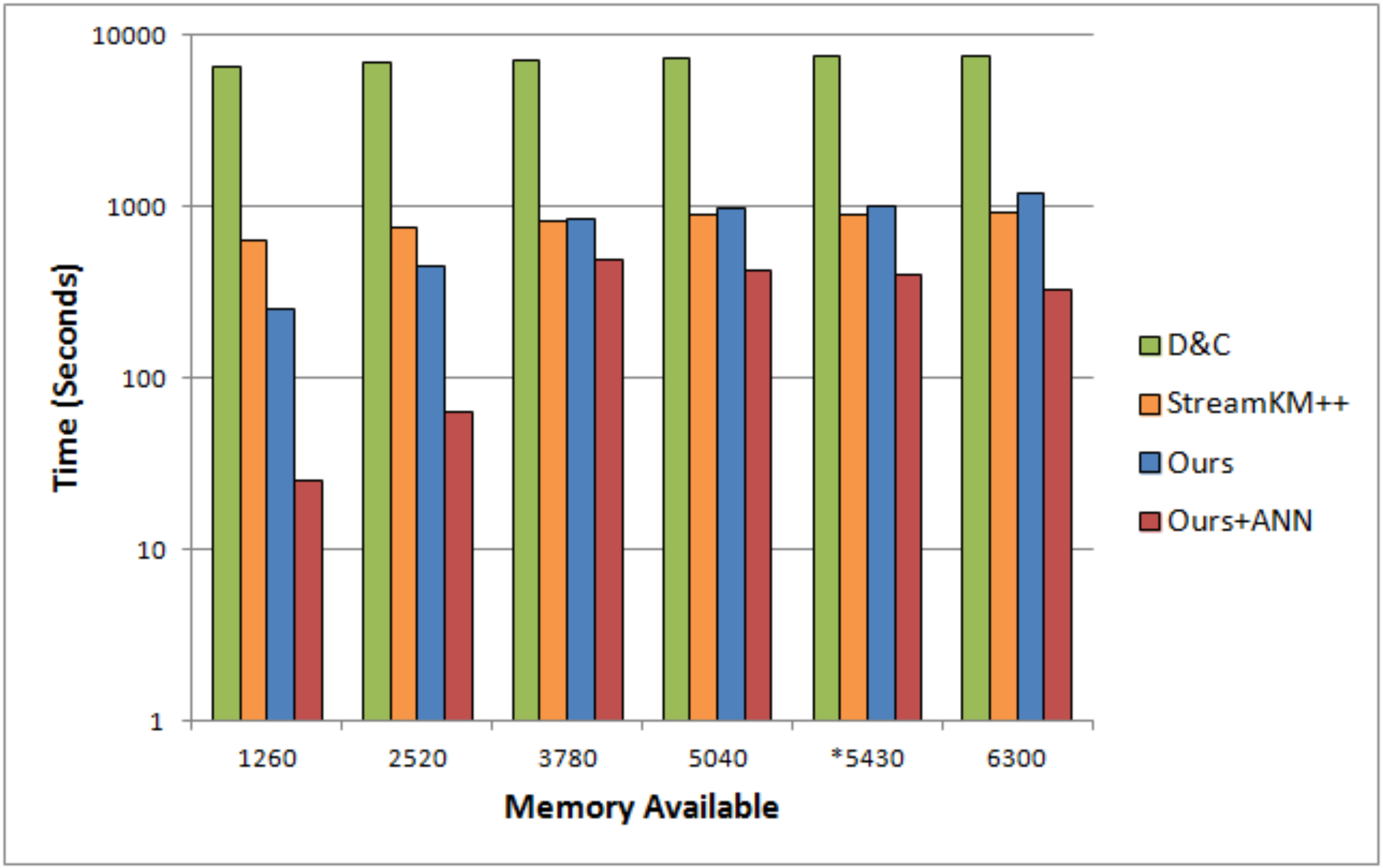
Bottleneck in Algorithm Runtime



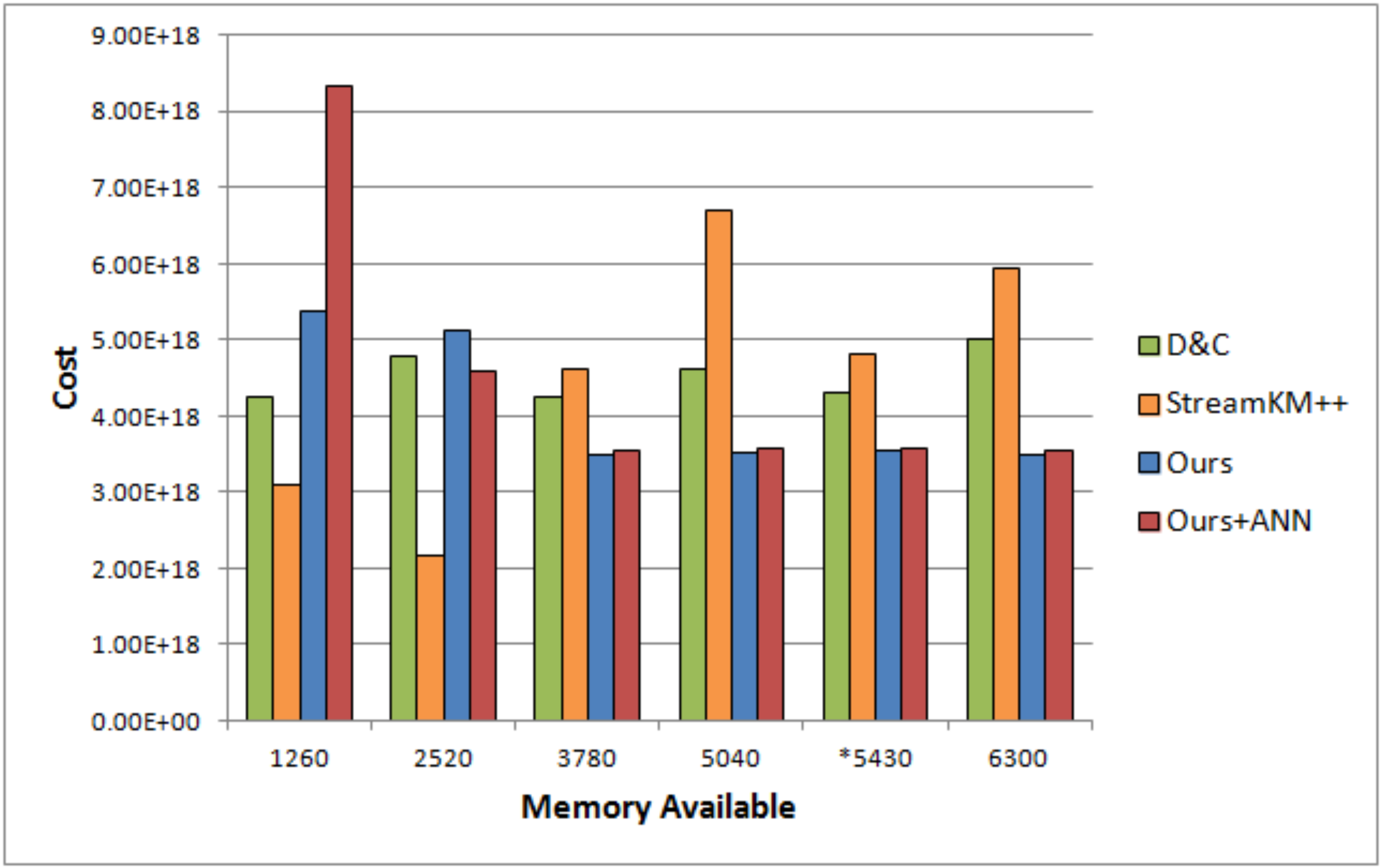
Compute Actual Distance to Those Neighbors



Substantially Faster



Cost change is (usually) minor



Conclusion

- Fast streaming *k*-means algorithm
  - Substantial Speedup
- Provides good quality clustering
  - Best *O(1)* cost guarantee among poly-time streaming algorithms
- Source Code available from

  http://web.engr.oregonstate.edu/~shindler/

Acknowledgments

- Meyerson and Shindler were partially supported by NSF CIF Grant CCF-1016540
- Shindler is supported by DARPA under Contract W911NF-11-C-0088.
  - Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the DARPA, the Army Research Office, or the US government.

Additional Slides

Room for Improvement

- [BMO+11] should be fast and straightforward
- However:
  - Actual memory requirements are high
    - O($k \log n$) memory great in limit
    - Facility cap of $\kappa = 1623\, k \log n$
  - Constant approximation bound is high
    - Constant is tens of thousands
  - End-of-phase conditions are complicated

End-of-phase conditions

- End-of-phase in [BMO+11]
  - "Phase" is reading data until *f* too low
  - When done, need to re-evaluate facilities and increase *f*
  - Performed maximal matching as part of this
  - Guaranteed no more than $n/k \log n$ phases
- Simpler phase transitions
  - Transition only on facility count
  - Increase *f*
  - Push facilities (weighted) back to stream
  - Continue reading stream, starting at those
  - Faster, no guarantee of phase count

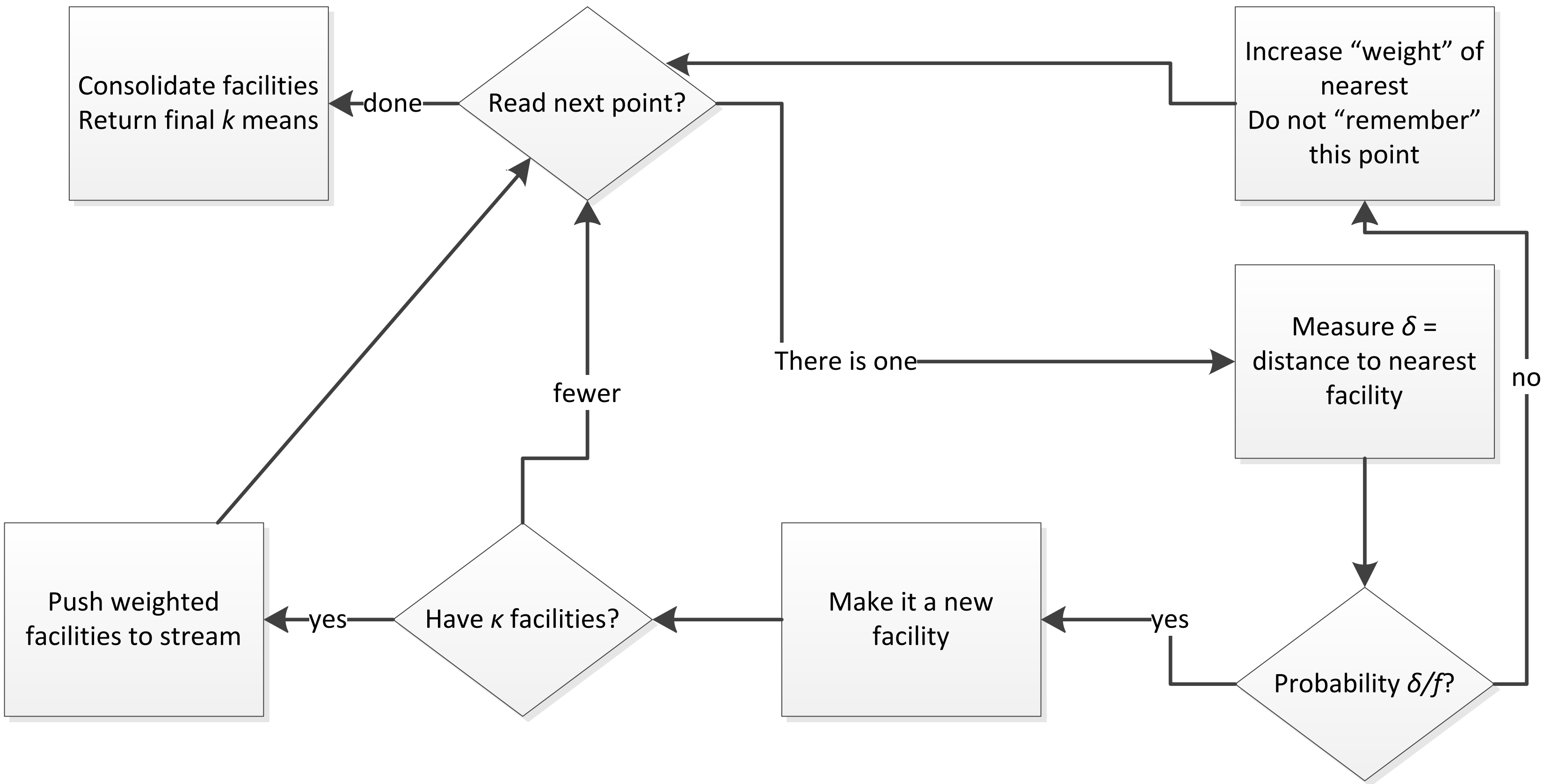
Memory Requirement

- [BMO+11] : facility cap of *1623* $k \log n$
- Great as an asymptotic bound
- Quite large in practice
- Instead, we will allow any κ facilities
- Facility count κ can be any in $\Omega(k \log n)$
- Will demonstrate that $\kappa = k \log n$ works well

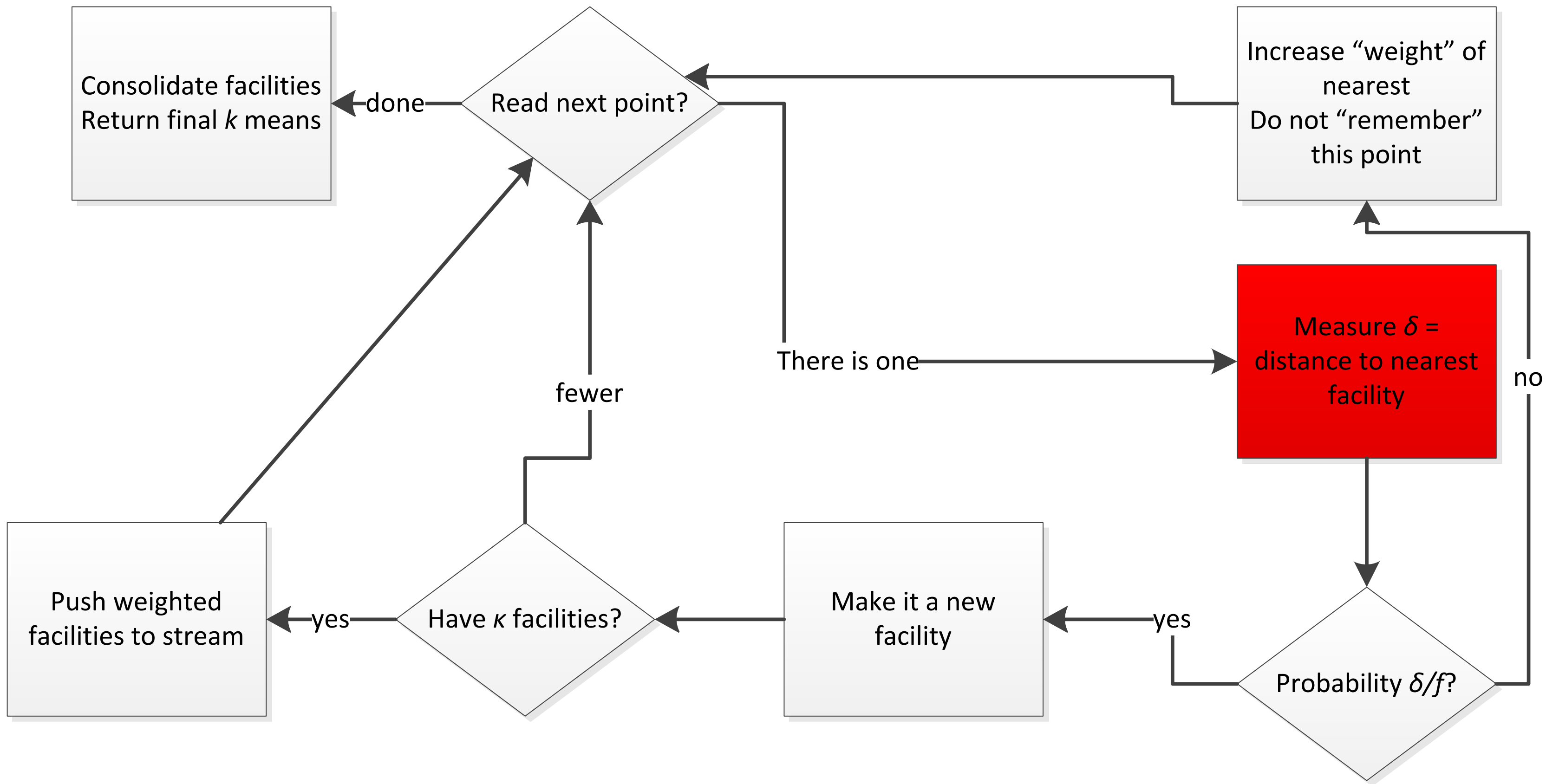
Approximation Bound

- Ratio of cost of solution vs optimal
- Approximation factor in [BMO+11] is 60,498
- We achieve a bound of 17

Algorithm: Spot the Bottleneck



Algorithm: Spot the Bottleneck



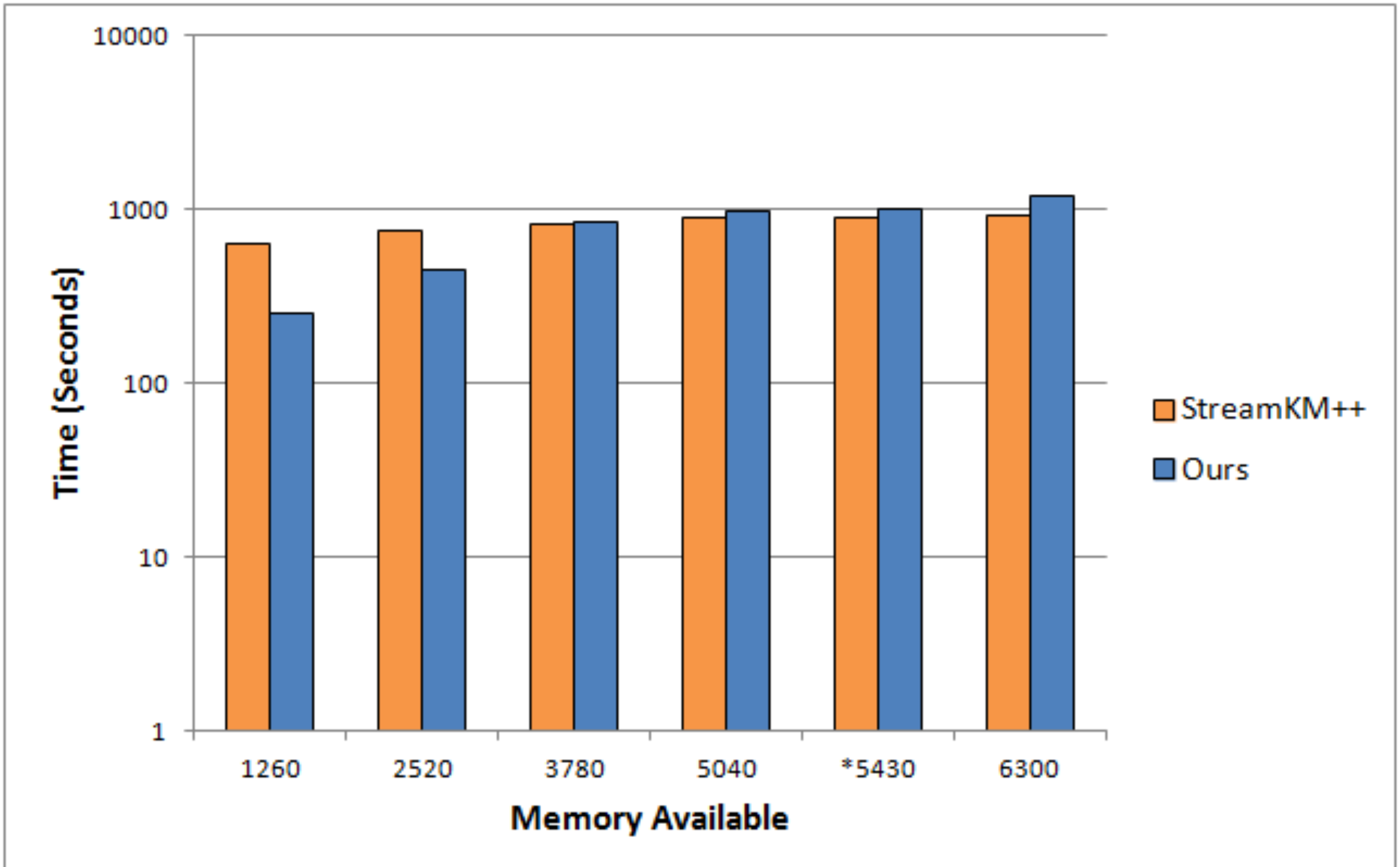
Bottleneck: Finding Nearest Facility

- Use approximate nearest neighbor algorithms
- To achieve guarantee:
  - Techniques from hashing and metric embedding
  - Look up is O(log n(log k + loglog n))
- MAIN RESULT:
  - Algorithm runtime is $o(nk)$ for most values of k
  - (Computing cost given solution takes $\theta(nk)$)

Bottleneck: Finding nearest Facility

- Fast practical implementation:
  - Select random point $\varpi \in [0,1)\uparrow d$
  - Store facilities sorted by inner product with ϖ
  - To find "nearest" facility to x:
    - Find a, b:
      - $a * \varpi \leq x * \varpi \leq b * \varpi$
    - Use closer of (a,b)

Speed close to competitors’



Bottleneck in Algorithm Runtime