Using Data-Oblivious Algorithms for Private Cloud Storage Access

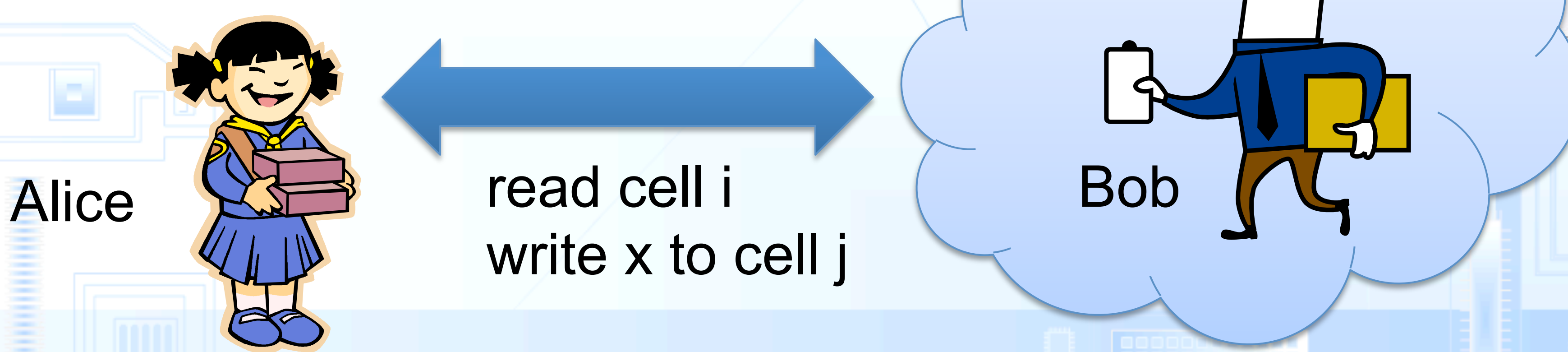
Michael T. Goodrich

Dept. of Computer Science

University of California Irvine

Privacy in the Cloud

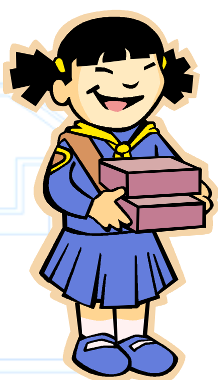
- Alice owns a large data set, which she outsources to an honest-but-curious server, Bob.
  - Alice trusts Bob to reliably maintain her data, to update it as requested, and to accurately answer queries on this data.
  - But she does not trust Bob to keep her information confidential.



Encryption is not Sufficient

- Alice certainly should use a semantically-secure encryption scheme, for each cell of her data.
- But this is not enough.

Alice



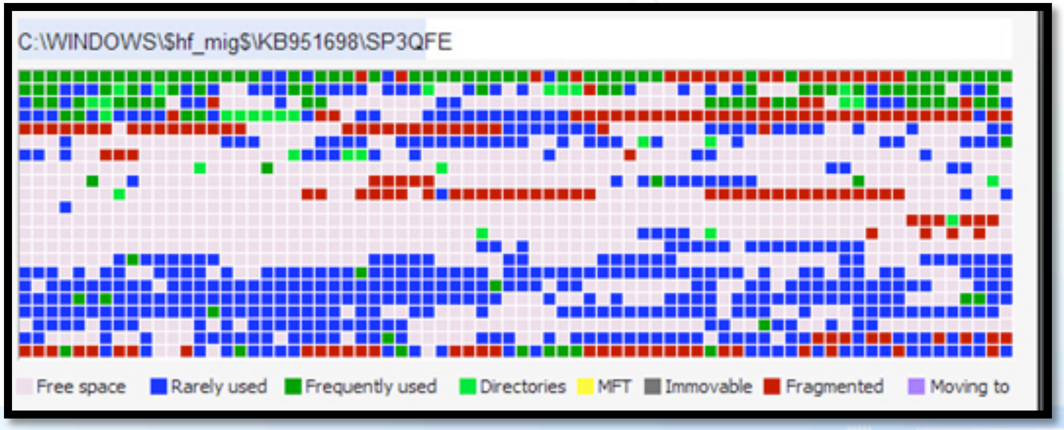
read E(x) from cell i
write E(y) to cell j

Bob

e.g., Bob can see the hot spots

Oblivious Data Storage

- Alice has a private memory, of size K, which she can use as local scratch space so that she can access her data on a untrusted server in a private fashion.
  - She wants to do this with low overhead
  - She wants to use this to hide her access patterns

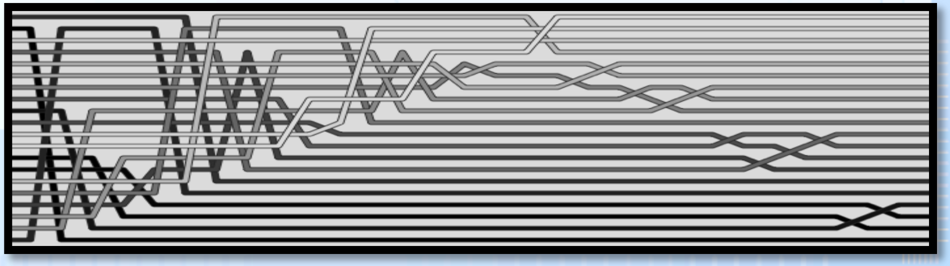


Data-oblivious Algorithms

- Alice can encrypt her data and then hide her access patterns by using data-oblivious algorithms.
  - A ***data-oblivious computation*** consists of a sequence of data accesses that **do not** depend on the input values.
  - All functions that combine data values are encapsulated into **black box** operations, with a constant number of inputs and outputs.
  - The control flow depends only on the **input size**, and, in the case of randomized algorithms, the values of **random variables**.

Two Approaches

- Design general methods to efficiently simulate an arbitrary RAM algorithm, A, in a data-oblivious fashion.
  - These methods typically have an overhead per access of O(log n), $O(\log^2 n)$, or even $O(\log^3 n)$.
- Design efficient data-oblivious algorithms for specific problems of interest.
  - These methods tend to be more efficient, but are more specialized
- We are taking a unified view, which allows for both approaches.

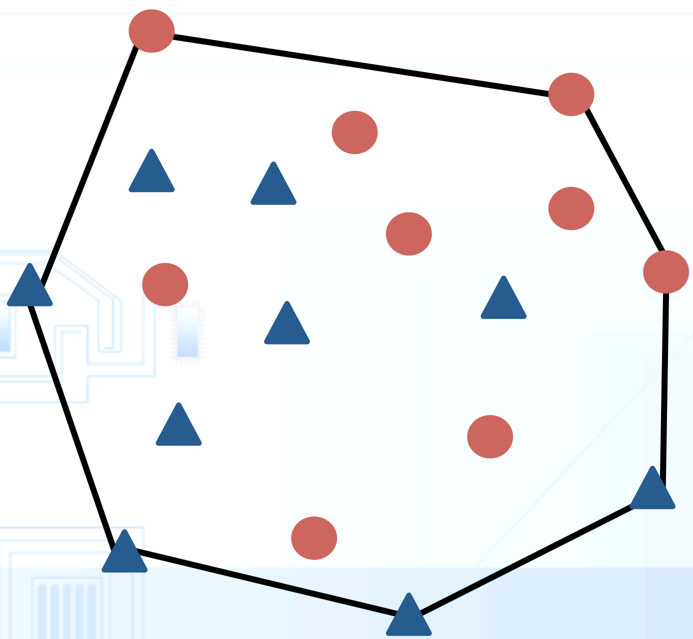


Our General Simulation Results

- We give methods for oblivious RAM simulation:
  - O(1) local memory and has $O(\log^2 n)$ overhead
  - $O(n^{\varepsilon})$ local memory and has O(log n) overhead.
  - $O(n^{\varepsilon})$ local memory and message size, and has O(1) overhead
- Our methods use the following techniques:
  - MapReduce cuckoo hashing
  - Data-oblivious external-memory sorting
  - cuckoo hashing with a shared stash

Our Results for Specific Problems

- We give data-oblivious algorithms for
  - Planar convex hull construction,
  - Minimum spanning trees,
  - Graph drawing problems,
  - All nearest neighbor finding



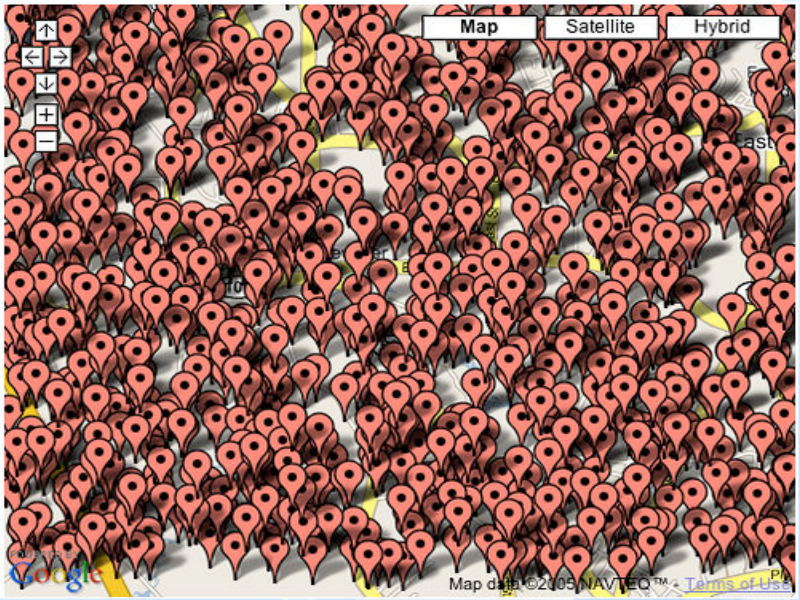
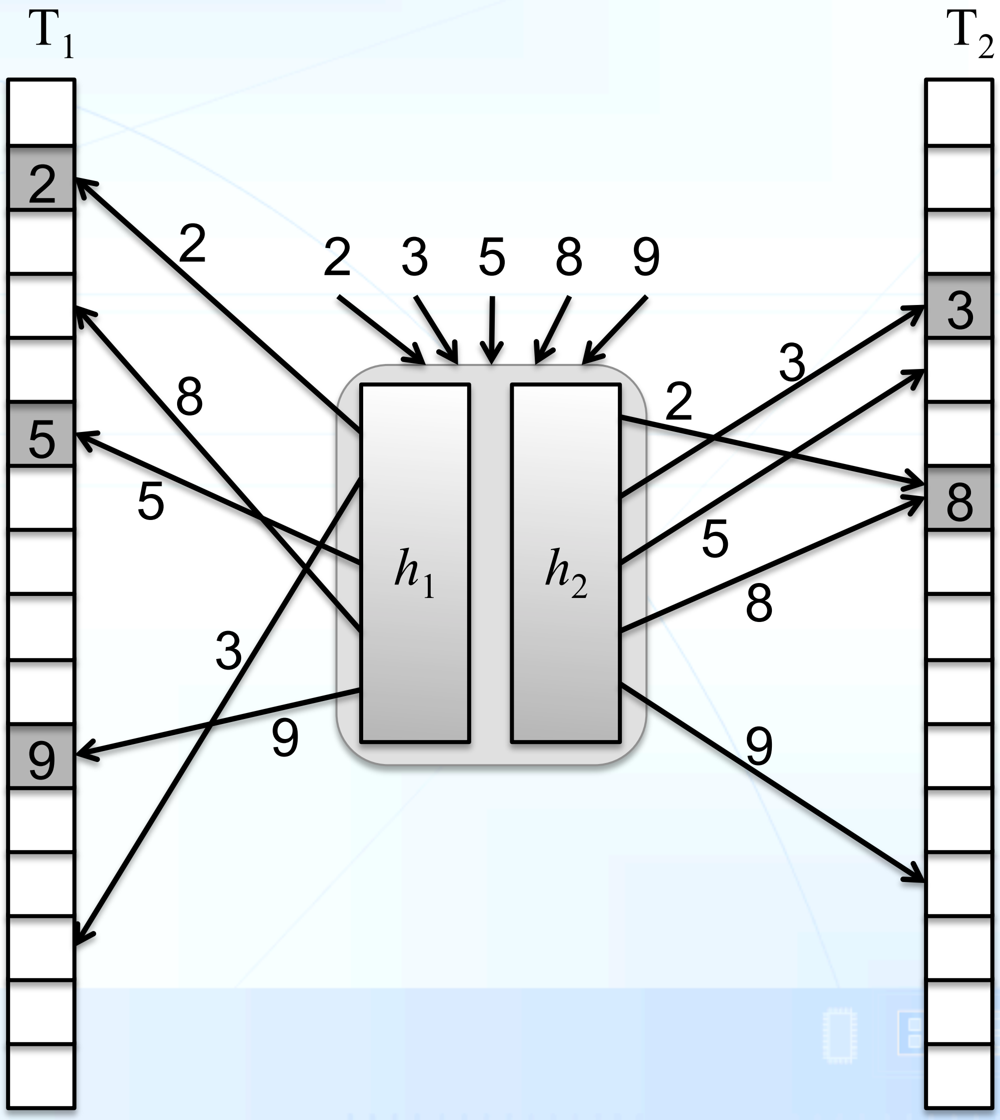


Image from http://cdn.venturebeat.com/wp-content/uploads/2009/03/28811286_e1671e30a9.jpg

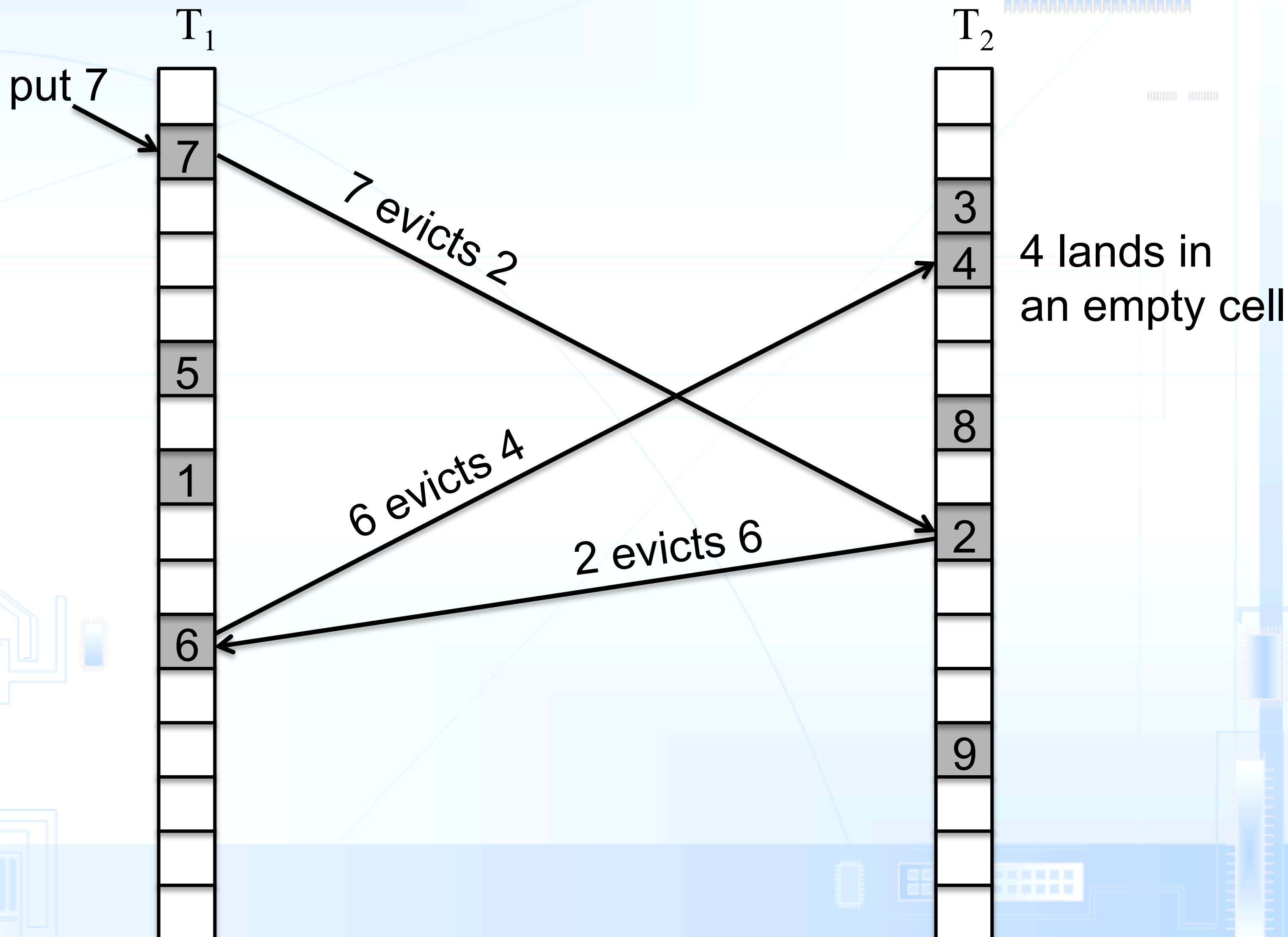
Cuckoo Hashing

- Uses two lookup tables T_0 and T_1 and two pseudo-random hash functions, f_0 and f_1.
- Each item x is stored either in $T_0[h_0(x)]$ or $T_1[h_1(x)]$.
  - When an item x is added, we put it in $T_0[h_0(x)]$.
    - If there was already an item y there, we put it in $T_1[h_1(y)]$.
      - If there was already an item z there, we put it in $T_0[h_0(z)]$.
        - » …
- Will add a new item in O(log n) time w/ probability 1-1/n.

Cuckoo Hashing Technique



Cuckoo Insertion



Using a Stash

- [Kirsch et al., 09] introduce the idea of using a stash with a cuckoo table.
  - A small cache where we store items that cannot be added to the cuckoo table without causing an infinite loop.
- A stash of size c improves the failure probability to be $1/n^c$.
  - Unfortunately, this is too large a failure bound for us…

Using a Big Stash

- We show that a stash of size O(log n) reduces the failure probability to be negligible.
  - But now lookups will no longer be O(1) time.
- Still, in some cases, like in ORAM simulation, we may have several cuckoo tables that share the same big stash.
- Ok, but there is still the issue of constructing a cuckoo table obliviously…

MapReduce

- A framework for designing computations for large clusters of computers.
- Decouples **location** from data and computation

Image from taken from Yahoo! Hadoop Presentation: Part 2, OSCON 2007.

Map-Shuffle-Reduce

- **Map**:
  - $(k,v) \rightarrow [(k_1,v_1),(k_2,v_2),\ldots]$
  - must depend only on this one pair, (k,v)
- **Shuffle**:
  - For each key k used in the first coordinate of a pair, collect all pairs with k as first coordinate
    - $[(k,v_1),(k,v_2),\ldots]$
- **Reduce**:
  - For each list, $[(k,v_1),(k,v_2),\ldots]$:
    - Perform a sequential computation to produce a set of pairs, $[(k'_1,v'_1),(k'_2,v'_2),\ldots]$
  - Pairs from this reduce step can be output or used in another map-shuffle-reduce cycle.

Image from http://www.wvculture.org/shpo/es/graphics/6-sorting%20shell.jpg

MapReduce Cuckoo Hashing

- We give a MapReduce Algorithm for constructing a cuckoo table.
- It performs O(n) parallel steps of item insertions
- With very high probability, this reduces the number of remaining uninserted items to be n/c, for some constant c.
  - Recursively add these items
- Total work is O(n).
- But now we need an oblivious way to simulate a MapReduce algorithm…

Oblivious Deterministic Sorting

- For internal-memory: AKS is the only deterministic oblivious method running in O(n log n) time.
- Randomized Shellsort [Goodrich '10] runs in O(n log n) time and sorts with high probability, but this isn't good enough here.
- We show how to design an oblivious external-memory sorting method that uses $O((N/B)\log^2_{M/B}(N/B))$ I/Os.

Generalized Odd-Even Sort

- We divide A into $k = (M/B)^{1/3}$ subarrays of size N/k and recursively sort each subarray.
- Let us therefore focus on merging k sorted arrays of size n = N/k each.
- If nk < M, then we copy all the lists into internal memory, merge them, and copy them back.
- Otherwise, let A[i, j] denote the jth element in the ith array. We form a set of m new subproblems, where the pth subproblem involves merging the k sorted subarrays defined by A[i, j] elements such that j mod m = p, for m = (M/B)1/3.
- Let D[i, j] denote the jth element in the output of the ith subproblem. That is, we can view
- D as a two-dimensional array, with each row corresponding to the solution to a recursive merge.

Lemma: Each row and column of D is in sorted order and all the elements in column j are less than or equal to every element in column j + k.

Proof: The lemma follows from Theorem 1 of Lee and Batcher [32].

- To complete the k-way merge, then, we imagine that we slide an m x k rectangle across D, from left to right. When it finishes, A will be sorted (obliviously)
- Runs in O((N/B)log2M/B (N/B)) I/Os.
- Note that this is O(N)-time sorting if B=1 and $M=O(N^{\varepsilon})$.

Our Simulation

- Construct O(log n) cuckoo tables in a hierarchy, H_0, H_1, H_2, …
- Each table is twice the size of the previous
- They all share a single stash of size O(log n)
- Store all the items (i,v) in these tables
- Initially, they are all empty except for the largest.

For each Access to i

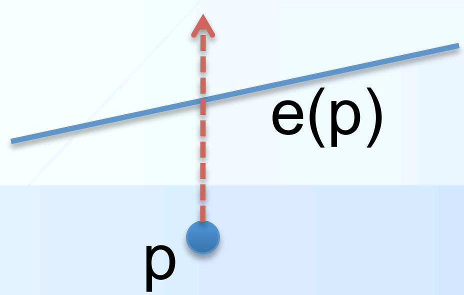
- First look in H_0 (which is just a list)
- Then look in H_1, H_2, …, doing a cuckoo lookup for i
- As soon as you find it, say in H_6, store it
- But to be oblivious, continue doing cuckoo lookups in H_7, H_8, …, for a random (previously unused) dummy index
- When we are done, but the updated value of (i,v) in H_0

Cascading

- Each time a table H_i fills up, we dump its contents in H_{i+1}, using the oblivious MapReduce construction
  - (…a few more details – please see the paper)
- We can do ORAM simulation with $O(\log^2 n)$ overhead with $O(1)$ local memory or $O(\log n)$ overhead with $O(n^{\varepsilon})$ local memory

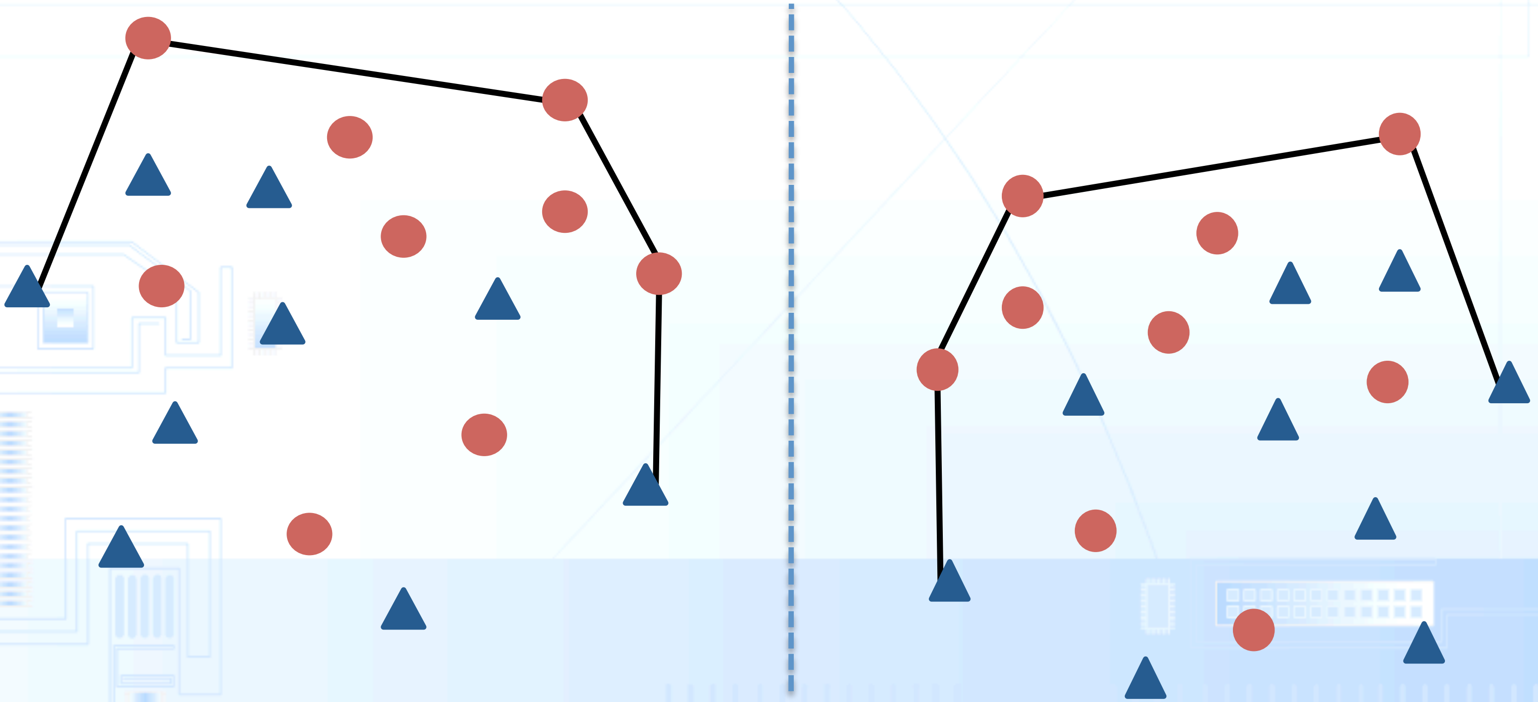
Convex Hull Representation

- We want the entire algorithm to be data-oblivious, except for low-level blackbox functions
- Given a set of points, A, ordered by their x-coordinates, we define the upper hull, UH(A), of A, to be as follows
  - For each point p in A, we label p with the edge, e(p), of the upper convex hull that is intersected by a vertical line through the point p. If p is itself on the upper hull, then we label p with the upper hull edge incident to p on the right.



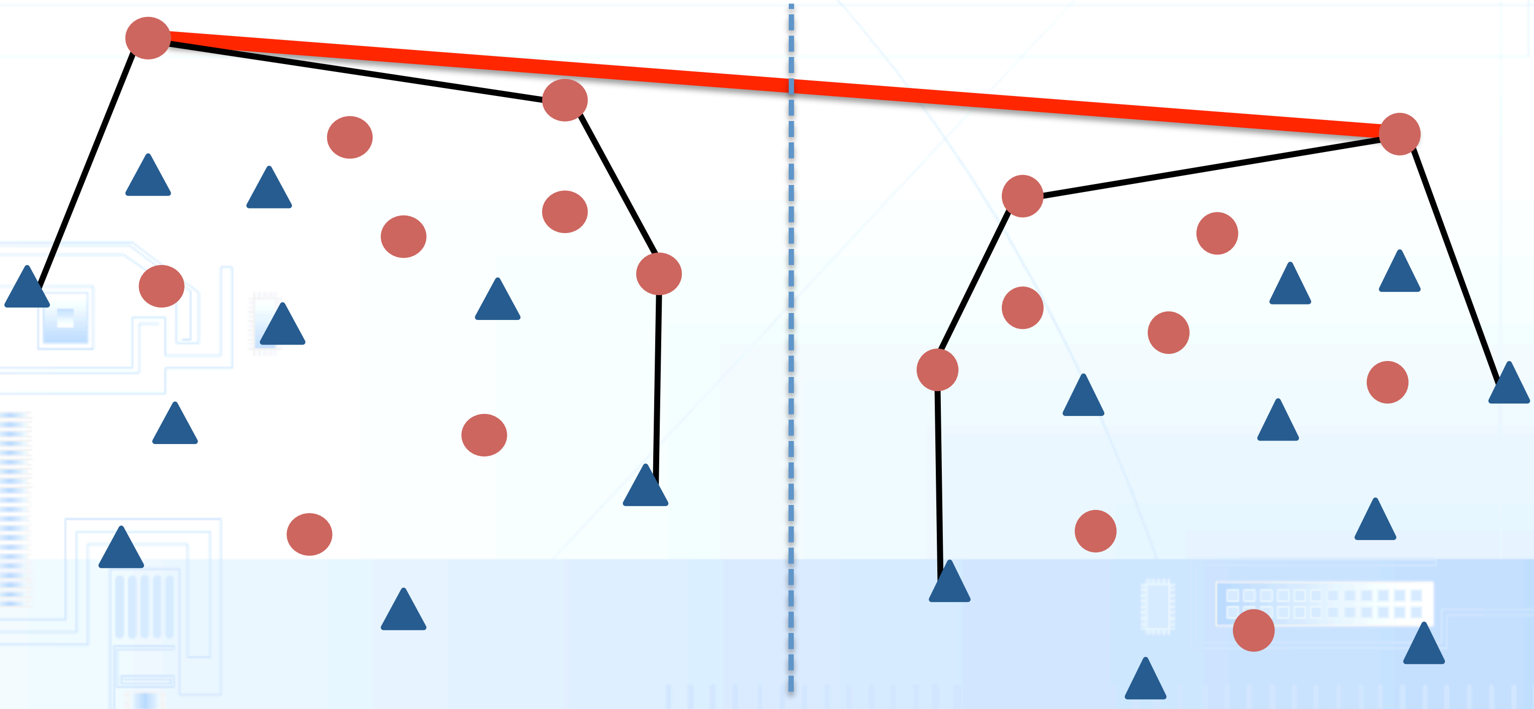
Our Approach

- Do an oblivious sort of A
- Divide A into left half and right half and recursively find UH of each side

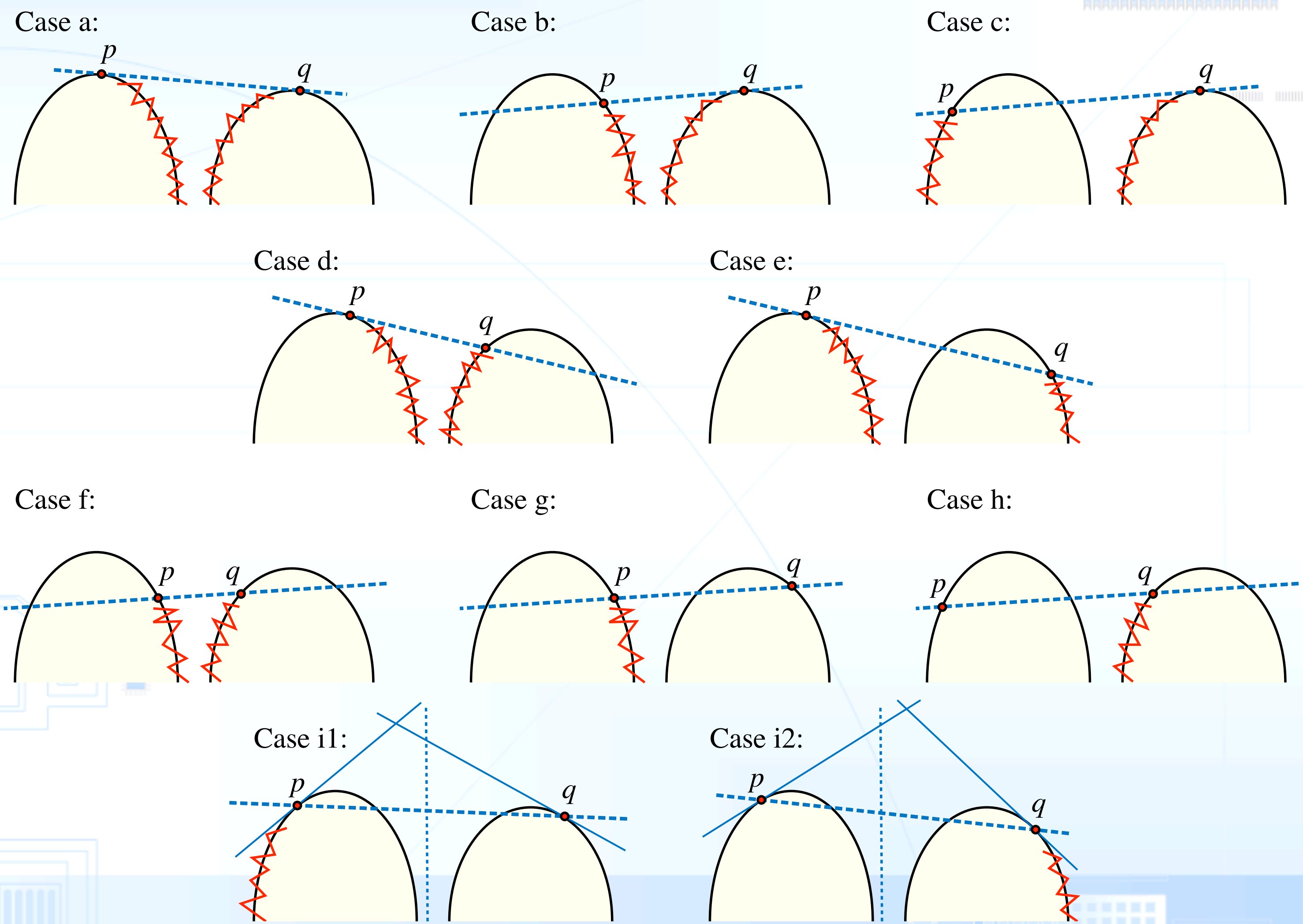


Merge Step

- Find the common upper tangent
- Relabel points under the tangent



Tangent-Finding Cases



[from Overmars & van Leeuwen]

Difficulty

- The classic binary search algorithm is **not** data-oblivious
- We need a new way to do this “search”
- We aim to assign each edge e of $UH(A_1)$ and $UH(A_2)$ one of two labels:
  - L: the tangent line of $UH(A_1 \cup A_2)$ with the same slope as e is tangent to UH(A1).
  - R: the tangent line of $UH(A_1 \cup A_2)$ with the same slope as e is tangent to UH(A2).
  - In some intermediate steps, we may be unable to determine yet whether an edge should be labeled L or R; In such cases, we temporarily label it with an X.

New Approach

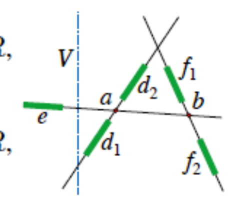
- Divide $UH(A_1)$ and $UH(A_2)$ at every $n^{1/2}$ edges
- Do brute-force comparisons
- See if we can reduce one of A_1 or A_2 to a region of size $n^{1/2}$
- Repeat until we have found the tangent
  - This sounds non-oblivious, but we can make it oblivious by trying all O(1) possible reductions in turn (one of them will work).

New Case Analysis

- For edge e in H_1, let d be the edge in H_2 with smallest slope greater than e and let f be the edge in H_2 with largest slope less than e

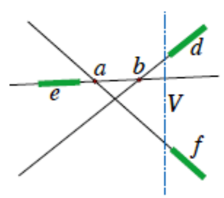
Case (ii): $V < a, b$.
$e \mapsto R$ or $d, f \mapsto L, R$

(e, d_1): Case h $e \mapsto X$, $d \mapsto L$.
(e, d_2): Case f $e \mapsto R$, $d \mapsto L$.
(e, f_1): Case g $e \mapsto R$, $f \mapsto X$.
(e, f_2): Case i2 $e \mapsto X$, $f \mapsto R$.

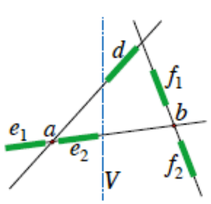


Case (i): $a, b < V$.

(impossible)

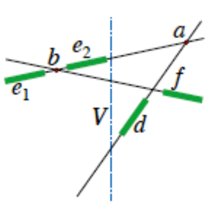


Case (iii): $a < V < b$.
$e \mapsto R$



(e_1, d): Case f $e \mapsto R$, $d \mapsto L$.
(e_2, d): Case g $e \mapsto R$, $d \mapsto X$.

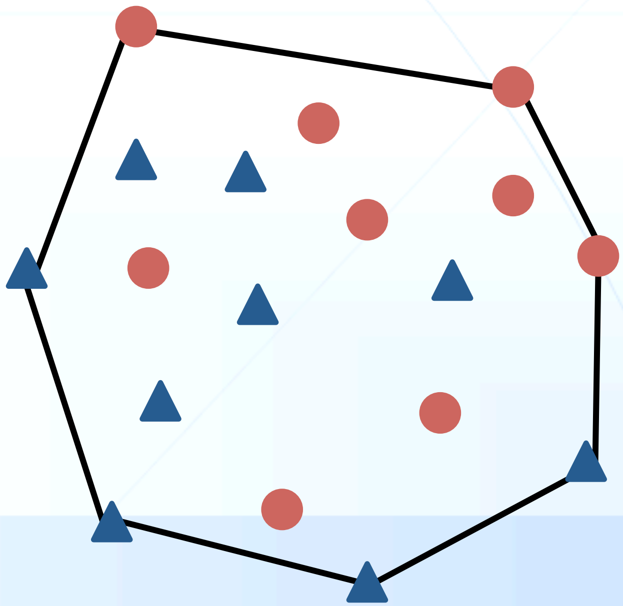
Case (iv): $b < V < a$.
$e \mapsto L$



(e_1, f): Case i1 $e \mapsto L$, $f \mapsto X$.
(e_2, f): Case h $e \mapsto L$, $f \mapsto L$.

Result

- This gives us an oblivious linear-time method for finding the common upper tangent
- This, in turn, results in a data-oblivious convex hull algorithm running in O(n log n) time.



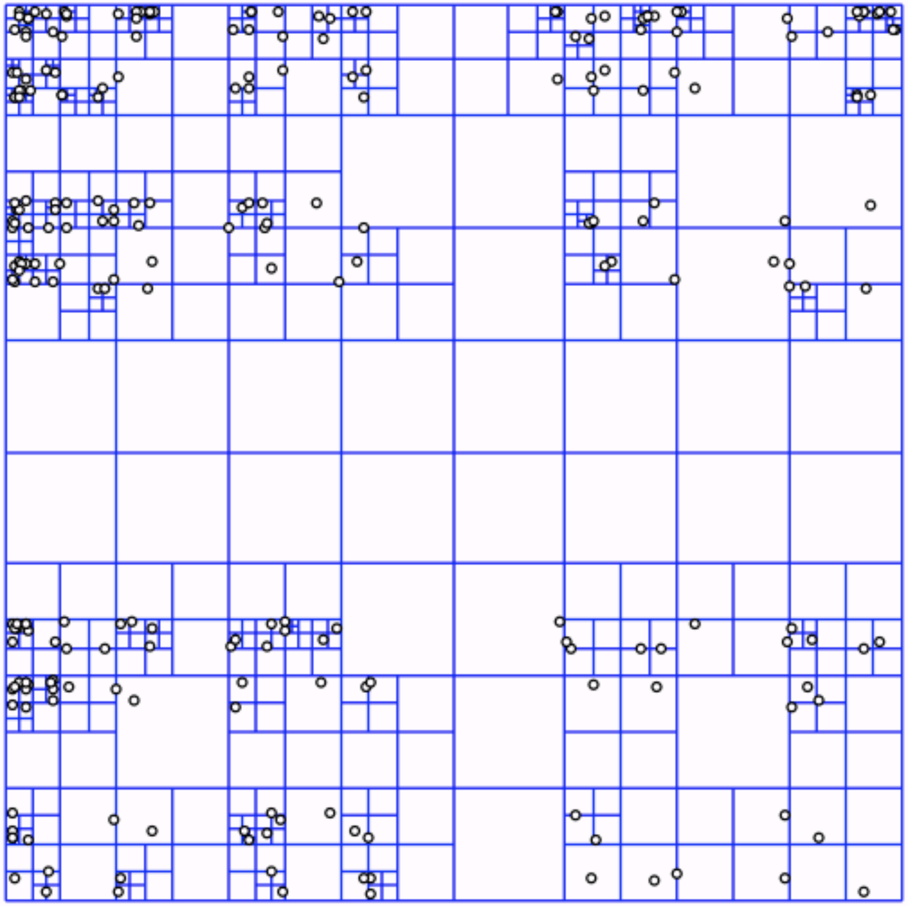
Data-Oblivious Nearest Neighbors

- Based primarily on two new oblivious algorithms
  - compressed quadtree construction
  - well-separated pair decomposition

The Geography Lesson (Portrait of Monsieur Gaudry and His Daughter), oil on canvas painting by Louis-Léopold Boilly, 1812, Kimbell Art Museum

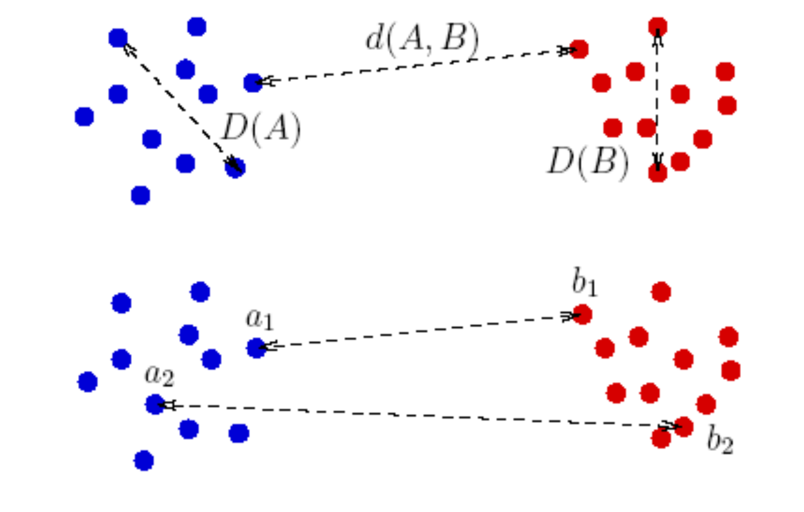
Quadtree Construction

- Use sorting and bit-interleaving trick (e.g., see Samet) to construct a compressed quadtree in a data-oblivious manner



Well-Separated Pairs

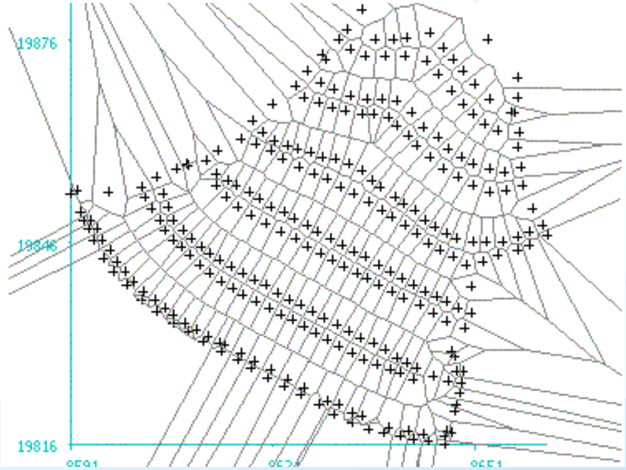
- Given a parameter s construct a set of pairs, (A_1,B_1), (A_2,B_2), …, (A_k,B_k), such that every pair of points p and q are represented by a pair (A_i,B_i) such that p is in A_i and q is in B_i, and such that there are balls of radius r containing A_i and B_i so that these balls are of distance at least sr apart.



Images from http://graphics.stanford.edu/~jgao/researchstatement/jie_research.html

Conclusion and Open Problems

- We have shown how to solve several geometric problems efficiently with data-oblivious algorithms
- These methods lead to efficient SMC protocols for privacy-preserving location-based methods
- Open: Is there a data-oblivious method for building a representation of the Voronoi diagram (or Delaunay triangulation) of a set of n points in O(n log n) time?



http://vbgraphic.altervista.org/terrain4.htm

Relevant Publications

1. M.T. Goodrich, “Randomized Shellsort: A Simple Data-Oblivious Sorting Algorithm," Journal of the ACM, 58(6), Article No. 27, 2011.
2. D. Eppstein, M.T. Goodrich, R. Tamassia, “Privacy-Preserving Data-Oblivious Geometric Algorithms for Geographic Data," Proc. 18th ACM GIS, 2010, 13-22.
3. M.T. Goodrich, “Spin-the-bottle Sort and Annealing Sort: Oblivious Sorting via Round-robin Random Comparisons," 8th ANALCO, 2011.
4. M.T. Goodrich, Data-Oblivious “External-Memory Algorithms for the Compaction, Selection, and Sorting of Outsourced Data," 23rd ACM SPAA, 2011, 379-388.
5. M.T. Goodrich and M. Mitzenmacher, “Privacy-Preserving Access of Outsourced Data via Oblivious RAM Simulation," 38th ICALP, vol. 6756, 2011, 576-587.
6. M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia, “Oblivious RAM Simulation with Efficient Worst-Case Access Overhead," ACM Cloud Computing Security Workshop (CCSW), 95-100, 2011.
7. M.T. Goodrich, O. Ohrimenko, M. Mitzenmacher, and R. Tamassia, “Privacy-Preserving Group Data Access via Stateless Oblivious RAM Simulation," 23rd SODA, 157-167, 2012.
8. M.T. Goodrich, O. Ohrimenko, M. Mitzenmacher, and R. Tamassia, “Practical Oblivious Storage," 2nd ACM CODASPY, 13-24, 2012.
9. M.T. Goodrich and M. Mitzenmacher, “Anonymous Card Shuffling and its Applications to Parallel Mixnets," 39th ICALP, Springer, LNCS, vol. 6756, 576-587, 2012.
10. M.T. Goodrich, O. Ohrimenko, and R. Tamassia, “Graph Drawing in the Cloud: Privately Visualizing Relational Data using Small Working Storage," 20th Graph Drawing 2012.