Datastructures for Dynamic Graphs

Ian Chen

University of Illinois Urbana-Champaign

July 16, 2026

Graph API

- State accessing:
  - `hasNode`, `hasEdge`
- Iterators:
  - `iterNeighbors`, `iterRange`, `iterNodes`, `iterEdges`
  - `parIterNeighbors`, `parIterRange`, `parIterNodes`, `parIterEdges`
- Updates:
  - `insertNode`, `insertEdge`
  - `insertNodeBatch`, `insertEdgeBatch`

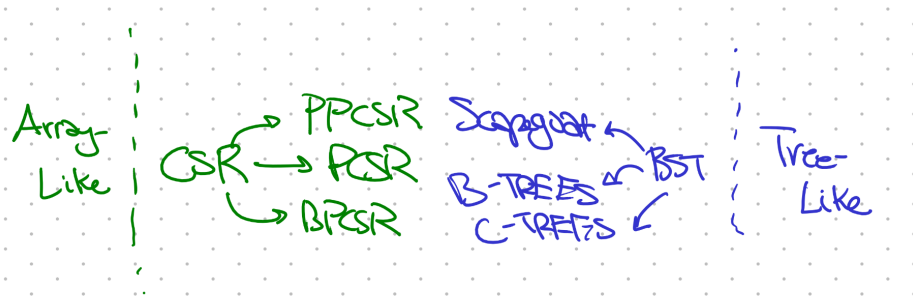
Common Solutions

- Adjacency matrix
  - Fast point queries
  - Slow iteration and updates
- Sparse adjacency matrix (sorted):
  - Fast queries, and fast iterators
  - Updates require complete rebuilds
- Blocked adjacency matrix (partial rebuilds)
- Dictionary of adjacency sets:
  - Fast queries, fast sequential iteration, slow parallel iteration
  - Many updates degrade with memory fragmentation

Two approaches

- Make a tree-like dictionary more array-like (ASPEN)
  - Represent graph's vertices and adjacency as a BST
  - Purely functional data structure $\implies$ lock-free parallelism
  - Rebuild centric means handle large batches efficiently
- Make an array-like matrix more tree-like (PPCSR)
  - Represent graph's vertices and adjacency as a sorted array
  - Hierarchical lock striping leads to fine grain shared-memory updates
  - Large rebuilds get embarrassingly parallel rebuilds

Two approaches (cont.)



Brainteaser

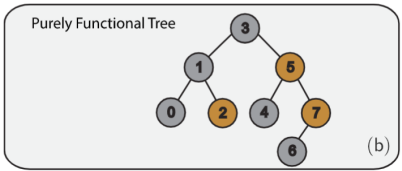
- For a (multi)-set X, $\texttt{rank}(x) = |\{\, y \in X \mid y < x \,\}|$
- For $\epsilon > 0$, handle approximate-median queries:
  - $\texttt{insert}(x)$ adds an element x into the X
  - $\texttt{remove}(\&x)$ removes $x \in X$ from X
  - $\texttt{query}()$ returns an element $x \in X$ with rank $(\frac{1}{2} \pm \epsilon)|X|$
- All operations can be done in constant *amortized* time!

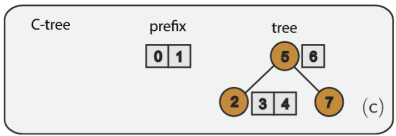
Scapegoat Trees

- Maintain binary search tree; fix $\alpha \in (1/2, 1)$
- Invariant: $|\texttt{subtree}(v.\textit{child})| \leq \alpha |\texttt{subtree}(v)|$
- Tombstone deletions, naive insertions, with *partial* rebuilding
  - $\Omega((\alpha - \frac{1}{2})|\texttt{subtree}(v)|)$ updates to go from balanced to unbalanced.
  - $O(\log_{1/\alpha}(n))$ parents $\implies$ $O(\log_{1/\alpha}(n))$ amortized cost per update.
- Storing as a flat array (naively): $O(2^{\log_{1/\alpha}(n)}) = O(n^{1/\log_2(1/\alpha)})$ space
  - $\alpha = 0.51$, the $O(n^{1.03})$ space, for $O(\log n)$ operations.
- Using *van* Emde Boas layout, bring to $O(n)$ space and $O(\log^2 n)$ updates.

B-Trees and C-Trees

- B-Trees are *B*-ary trees
  - Improved cache locality; reduced height per traversal
- C-Trees store *heads* in the tree only
  - Other nodes get hooked onto the tree-elements (or prefix)
- C-trees are immutable data structures, so easy parallelism





These are included for completeness, although I am not an expert here.

Packed Memory Array (PMA)

- `ListLabelingProblem`
  - Assign labels $[\![m]\!]$ to n ordered elements
  - Labels consistent with order; allow insert/deletions
- If $m = n$, then operations take $\Omega(n)$ worst-case and amortized
- If $m = \Theta(n)$, PMA achieves $O(\log^2 n)$ amortized time (worst-case)
  - $O(\log n)$ whp in permutation model

Packed Memory Array (cont.)

	Adjacency Matrix	Adjacency list (linked list)	Blocked Adjacency List	Compressed Sparse Row	**Packed CSR** (amortized)
Storage cost / scanning whole graph	$O(n^2/B)$	$O(n/B+m)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$
Add edge	$O(1)$	$O(1)$	$O(1)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Update or delete edge from vertex v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O((m+n)/B)$	$O(\lg^2(m+n)/B)$
Add node	$O(n^2/B)$	$O(1)$	$O(1)$	$O(1)^*$	$O(\lg^2(m+n)/B)$
Finding all neighbors of a vertex v	$O(n/B)$	$O(deg(v))$	$O(deg(v)/B)$	$O(deg(v)/B)$	$O(deg(v)/B)$
Finding if w is a neighbor of v	$O(1)$	$O(deg(v))$	$O(deg(v)/B)$	$O(\frac{\lg(deg(v))}{\lg(B)})$	$O(\frac{\lg(deg(v))}{\lg(B)})$
Sparse matrix-vector multiplication	$O(n^2/B)$	$O(n/B+m+n)$	$O((m+n)/B)$	$O((m+n)/B)$	$O((m+n)/B)$

Packed Memory Array (cont.)

- Have $m = \Theta(n)$ sized array for n elements (labels is index)
- Binary sub-interval v has maximum proportion[1] of filled indices τ_v
  - Leaf intervals are of size $O(\log n)$
  - Leaves always are packed left and always non-full
  - Insert into the subinterval takes $O(\log n)$ time
- Let $\Delta = \tau_{v.child} - \tau_v = O(1/\log n)$, so that
  - Each redistribution takes linear time
  - Requires $\Omega((\Delta)|\texttt{subtree}(v)|)$ updates to make unbalanced
  - There are $O(\log n)$ parents
  - Amortized $O(\log^2 n)$ time per insertion

[1] Maintaining a lower proportion similarly allows for $O(\log^2 n)$ time deletions.

Parallel Packed Memory Array

- Two rankings of locking: updates $<$ redistribution
- To support an update:
  1. Lock the leaf; make the update; unlock the leaf
  2. Acquire any locks needed for redistribution in *lock_order* for deadlock-free
- If all locks get a unique priority, then deadlock-free
- If all binary *intervals* have a unique minimum priority, then deadlock-free
- `popcount`(v), number of right turns on path from root to v
  - Minimum is always the leftmost-leaf
  - Has $O(\log n)$ distinct priorities, so guarantee low-latency

Results

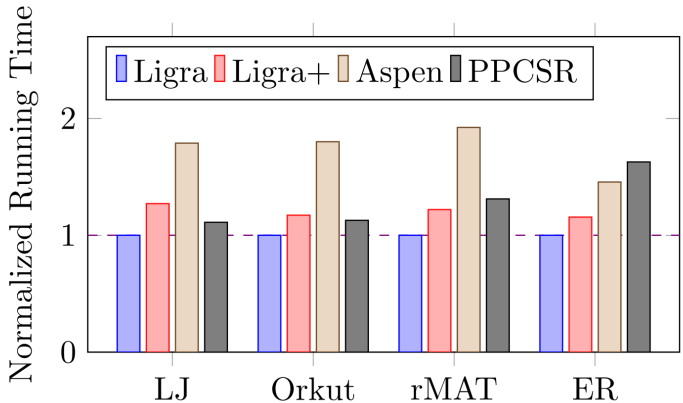


Figure: **Running time of BFS normalized to Ligra:** PPCSR is on average 1.1-1.6x slower than Ligra, a static graph structure. It is 0.9-1.6x faster than ASPEN, for supporting dynamic graphs.

Results (cont.)

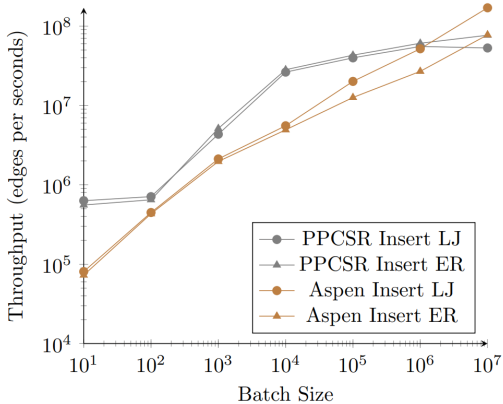


Figure: **Update throughput of PPCSR and Aspen:** For small batch sizes, PPCSR outperforms Aspen by up to 5x. For large batch sizes, Aspen outperforms PPCSR by up to 5x.

Results (cont.)

Name	PPCSR	Ligra	Ligra+	Aspen
LJ	1.3	.34 (.66)	.23 (.55)	0.66 (.98)
Orkut	4.4	.91 (1.78)	.53 (1.4)	1.04 (1.91)
rMAT	8.79	2.13 (4.23)	1.51 (3.61)	3.93 (6.03)
ER	16.2	3.76 (7.49)	2.82 (6.55)	7.06 (9.16)

Table 4: Memory footprint (in GB) of the graphs on the different systems. PPCSR was run with weights, and all other systems were run without weights. To compare weighted and unweighted, we add the ideal $4 \times |E|$ (each weight is 4 bytes) to all structures which do not store weights, shown in parentheses.

Summary

- PPCSR uses the most memory (because it introduces spacing). It is slowest on memory-bound tasks (with very large batch sizes).
- Downsides of amortized analysis (the heavy-tail latency) is mitigated by parallelism.
- Locking is better for small updates; large-scale rebuilds better for large updates.

References I

[1] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. "Low-latency graph streaming using compressed purely-functional trees". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '19. ACM, June 2019, pp. 918–934. DOI: 10.1145/3314221.3314598. URL: http://dx.doi.org/10.1145/3314221.3314598.

[2] Brian Wheatman, Randal Burns, and Helen Xu. "Batch-Parallel Compressed Sparse Row: A Locality-Optimized Dynamic-Graph Representation". In: *2024 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2024, pp. 1–8. DOI: 10.1109/hpec62836.2024.10938461. URL: http://dx.doi.org/10.1109/HPEC62836.2024.10938461.

References II

[3] Brian Wheatman and Helen Xu. “A Parallel Packed Memory Array to Store Dynamic Graphs”. In: *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*. Society for Industrial and Applied Mathematics, Jan. 2021, pp. 31–45. ISBN: 9781611976472. DOI: 10.1137/1.9781611976472.3. URL: http://dx.doi.org/10.1137/1.9781611976472.3.

[4] Brian Wheatman and Helen Xu. “Packed Compressed Sparse Row: A Dynamic Graph Representation”. In: *2018 IEEE High Performance extreme Computing Conference (HPEC)*. IEEE, 2018, pp. 1–7. DOI: 10.1109/hpec.2018.8547566. URL: http://dx.doi.org/10.1109/HPEC.2018.8547566.