

Facilitating Large-Scale Graph Searches with Lock-Free Pairing Heaps

Jeremy Mayeres, Charles Newton, Peter Tonner

12 March 2012

Project Overview

- We are constructing a lock-free version of a Pairing heap (a self-balancing heap.)
- Pairing heaps have an efficient decreaseKey implementation (near-constant performance) that allow you to decrease the value in a heap without reinserting it.
- This improves the asymptotic performance of certain algorithms (e.g., Dijkstra's algorithm.)
- We are comparing our heap against Skipqueues (a priority queue backed with a Skiplist.)

Dijkstra's Algorithm

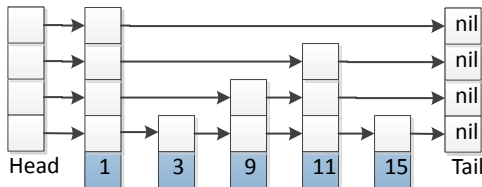
Single-Source Shortest Path Problem

In a weighted graph $G = (V, E)$, find the shortest path from a target vertex $v \in V$ to all other vertices in the graph.

- Push every node in a graph into a priority queue (PQ.) In the PQ, each node is weighted by current information we have about their distance.
- Initially, all nodes have a distance of ∞ , except the target node, which has a distance of 0.
- Dynamic programming approach: Inductively build up our shortest routes.
 - Pop off the node on the PQ with the smallest distance. (The shortest path from the source to this node is finished.)
 - Update the weights in the PQ with the distances emanating from the popped node.

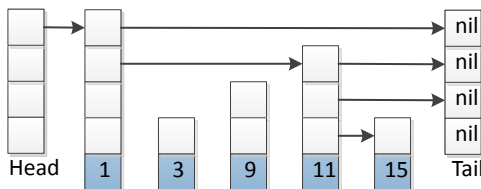
Skiplists

- Skiplists are constructed from a hierarchy of linked lists with the *Skiplist property*: the set of elements contained in level i is a subset of all the levels below it.
- Links allow you to do a binary search and “jump” around a list.
- The height of an element is randomly sampled from a power-law distribution: the probability of a node having a height of $i \geq 0$ is 2^{-i} .
- Time complexity of operations are probabilistically the same as for a binary search tree, but $\mathcal{O}(n)$ in the worst case.



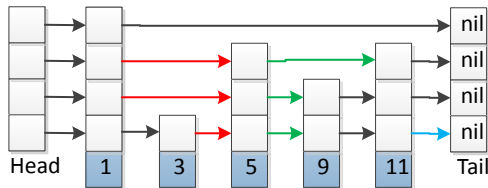
Skiplists (Search)

- For each level, move right until you run into a node greater than your target. Then, from this point, move right on the next lowest level, and repeat.
- Likely runs in $\mathcal{O}(\log n)$ time.
- Head / tail nodes have values of $-\infty$ and ∞ , respectively.



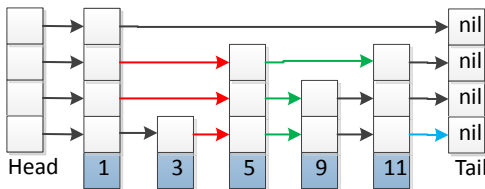
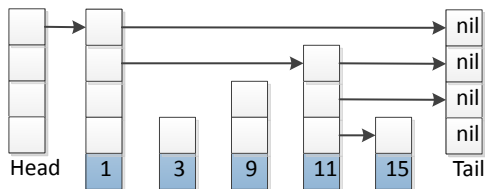
Next and Previous Windows

- A useful abstraction to make is the set of all pointers related to a given node. (Aggregate all the pointers related to one node.)
- $\text{prev}[i] \rightarrow$ the node in level i pointing to the target node.
- $\text{next}[i] \rightarrow$ the node the target node points to at level i .
- Use the search process to construct these sets.



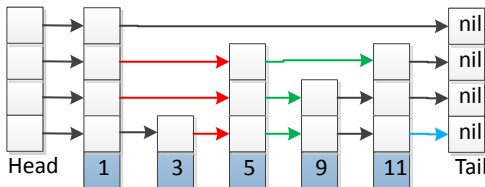
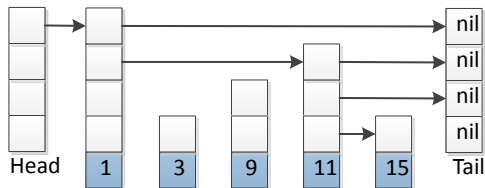
Skiplists (Insertion)

- Insertion: Find the location the target should be at. Set all of the node's next pointers to $\text{next}[i]$ and set the next pointers at each $\text{prev}[i]$ to the new node.



Skiplists (Deletion)

- Deletion: Set all of the next pointers in $\text{prev}[i]$ equal to $\text{next}[i]$.



Lock-free Skiplists

- Instead of linked lists at each level, we use lock-free linked lists.
- We lose the Skiplist property. (Levels aren't necessarily subsets of each other.) In particular, this means we need to always verify a node is in the lowest level.

Lock-free Skiplists

- Instead of linked lists at each level, we use lock-free linked lists.
- Linearization point: membership is defined at the lowest level of the Skiplist.
- We lose the Skiplist property. (Levels aren't necessarily subsets of each other.) In particular, this means we need to always verify a node is in the lowest level.

Lock-Free Skiplists (Insertion)

- Construct the $\text{prev}[i]$ and $\text{next}[i]$ sets.
- Set all of the node's next pointers to $\text{next}[i]$.
- If we can CAS the node into the bottom level, continue. Otherwise, something changed, and restart (reconstruct $\text{prev}[i]$ and $\text{next}[i]$).
- Next, CAS all $\text{prev}[i]$ to the new node. If a CAS fails, reconstruct $\text{prev}[i]$.

Lock-Free Skiplists (Deletion)

- Pointers are atomically *markable*.
 - In C / C++, steal a bit from the pointer.
 - In Java, use `AtomicMarkableReference`.
- Construct the `prev[i]` and `next[i]` sets.
- For each pointer in our target node, mark them as deleted.

Optimized decreaseKey for Lock-Free Skiplists

- Currently, to implement a decreaseKey for Skiplists, we need to do two $\mathcal{O}(\log n)$ operations (one insert and one delete.)
- We can optimize this in some cases by removing redundant work: reuse the `prev[i]` set as a starting point for the next insertion.

Pairing Heaps

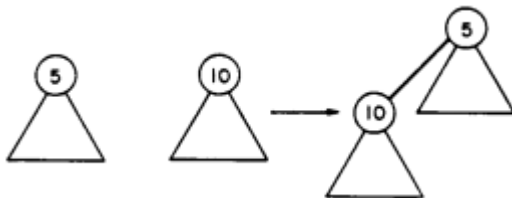
- Pairing Heaps were introduced by Fredman and Tarjan¹ as a simplification of Fibonacci heaps.
 - Fibonacci heaps have a $\mathcal{O}(1)$ decreaseKey operator, but require complicated rebalancing.
- decreaseKey runs in $2^{\mathcal{O}(\sqrt{\log \log n})}$ time.
 - In practice, constant time. E.g., for a graph with a billion edges, $2^{\sqrt{\log \log 10^9}} = 3.34$.
- Pairing heaps have better performance than Fibonacci heaps on reasonably-sized graphs.²

¹“The Pairing heap: a new form of self-adjusting heap,” Fredman, Sedgewick, Sleator, and Tarjan, Algorithmica (1986).

²“On the Efficiency of Pairing Heaps and Related Data Structures,” Michael Fredman, Journal of the ACM, 1999.

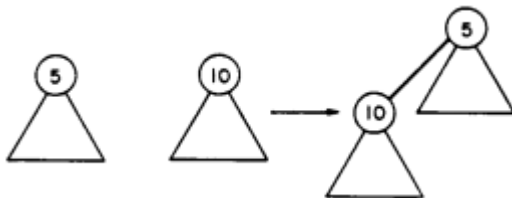
Pairing Heaps: Melding Two Heaps

- To meld two heaps, compare their roots and add the root as a child of the smaller root.
- `meld` is the core operator for Pairing heaps.



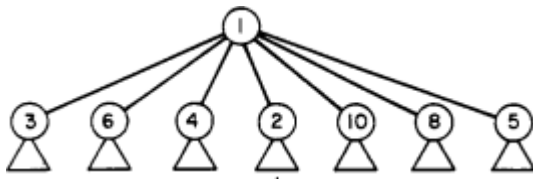
Pairing Heaps: insert

- To insert a new element into a pairing heap, construct a trivial heap containing only that element and `meld` this to the root.
- (The root of the heap will change if the new element is lower.)



Pairing Heaps: deleteMin

- Pairing heaps have a two-step deleteMin operator.



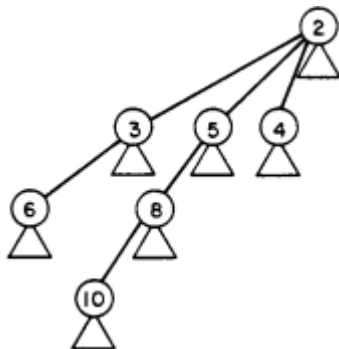
Pairing Heaps: deleteMin

- After removing the root, meld each pair of heaps.



Pairing Heaps: deleteMin

- meld each resultant heap from right-to-left.



Parallelizing Dijkstra's Algorithm

- To simplify our Pairing heap implementation, we will only consider function interleavings that can occur when the heap is used for Dijkstra's algorithm.
- For Dijkstra's algorithm, we pop the minimum off the PQ (`deleteMin`) and call `decreaseKey` on the node's neighbors.
- If `decreaseKey` produces new heap minima, this will affect future calls to `deleteMin`
- For this reason, we assume that calls to `decreaseKey` and `deleteMin` never intersect.

Lock-Free Pairing Heaps: Overview

- Modify the list of sub-heaps to be a lock-free list of subheaps.
- We have identified an easy linearization point for `insert` and `deleteMin`: a node is considered in-place when the root is updated (CAS'd into place.)
 - But, what if the new node is inserted as a subheap of the current root?
 - We create a copy of the root to avoid this issue.

Lock-Free Pairing Heaps: insert an element

- Recall the insertion procedure: create a trivial heap containing only that element and meld it to the root.
- Lock-free insert:
 - Create a copy of the root, but the root and its copy share the same list of subheaps (this avoids an $\mathcal{O}(n)$ copy.)
 - Meld the new heap with the copy of the root.
 - Try to CAS the new root as the root of the heap.
 - If CAS succeeds, we're done.
 - If CAS fails, delete the subheap list changes and retry.

Lock-Free Pairing Heaps: decreaseKey

- Main observation: If Dijkstra's algorithm is run on a graph with exactly 0 or 1 edge between each node, calls to decreaseKey will be unique (per iteration.)
- **Case 1:** If the parent is still smaller than the node, just decrease the value.
 - We know that this node won't replace the root, since the parent is larger. An interleaved call to change the parent can only decrease its value.
- **Case 2:** If we're trying to decrease the value of the root, copy the root (as before), decrease its value, and try to CAS the new root into place. After a CAS fails, check if we are still trying to decrease the root and repeat if so.
- **Case 3:** The node is now smaller than its parent, violating the heap invariant. Immediately remove the node from the parent's list of subheaps. Next, follow the same procedure as insert.

Empirical Considerations

- Skiplists are extremely concurrent, but require two $\mathcal{O}(\log n)$ operations to implement `decreaseKey`. Also, operations require many pointer updates.
- Pairing heaps have a near-constant `decreaseKey` implementation and fewer pointer updates per call, but every operation is contending for the same location in memory (every thread contends for the root.)

Experimentation

- Run Dijkstra's algorithm on various large graphs (both randomly generated and real-world, e.g, from Harvard's Human Interactome Database.)
- Compare the time Dijkstra's algorithm takes for a Skiplist-backed implementation vs. a lock-free Pairing heap.
- Run experiments with varying numbers of threads.