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

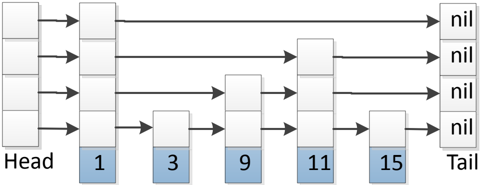
Single-Source Shortest Path Problem

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.

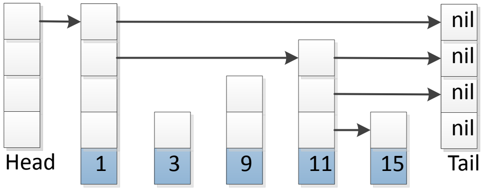
Skiplists

- 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.



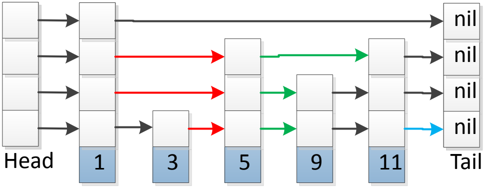
Skiplists (Search)

- 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.



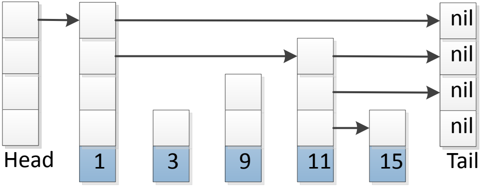
Next and Previous Windows

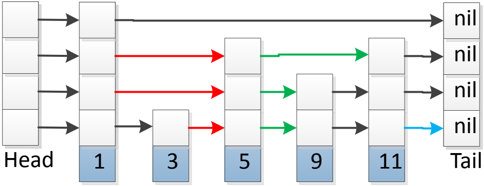
- A useful abstraction to make is the set of all pointers related to a given node. (Aggregate all the pointers related to one node.)
- $\texttt{prev}[i] \rightarrow$ the node in level i pointing to the target node.
- $\texttt{next}[i] \rightarrow$ the node the target node points to at level i.
- Use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



Skiplists (Insertion)

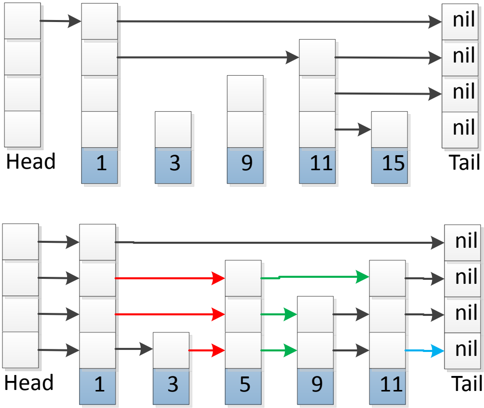
- Insertion: Find the location the target should be at. Set all of the node's next pointers to next$[i]$ and set the next pointers at each prev$[i]$ to the new node.





Skiplists (Deletion)

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



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 `prev`$[i]$ and `next`$[i]$ sets.
- Set all of the node's next pointers to `next`$[i]$.
- If we can CAS the node into the bottom level, continue. Otherwise, something changed, and restart (reconstruct `prev`$[i]$ and `next`$[i]$).
- Next, CAS all `prev`$[i]$ to the new node. If a CAS fails, reconstruct `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 Tarjan[1] 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).

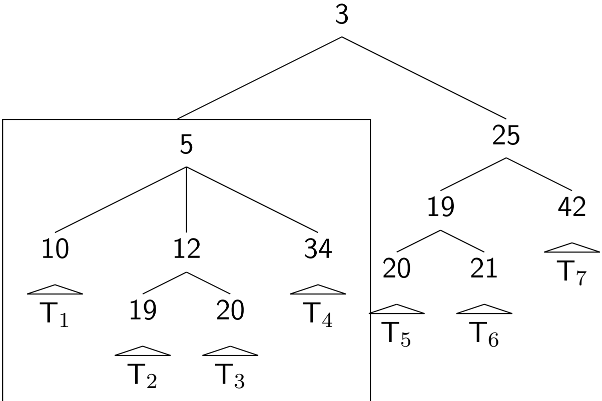
[2] "On the Efficiency of Pairing Heaps and Related Data Structures," Michael Fredman, Journal of the ACM, 1999.

Pairing Heaps

- Pairing Heaps were introduced by Fredman and Tarjan.
- Each node is a sub-heap
- The parent of any node has key smaller than the child node
- Each node has a left pointer to its first child, and a right pointer to its sibling
- Like `decreaseKey`, other operations run quickly as well
  - `meld`: $\mathcal{O}(1)$
  - `findMin`: $\mathcal{O}(1)$
  - `insert`: $\mathcal{O}(1)$
- `delete` and `deleteMin` run in $\mathcal{O}(\log n)$ amortized time

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.
- Example: Melding two min-heaps together (root 3 and root 5)



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: `decreaseKey`

- Change the value of a node in the pair heap (e.g. updating the minimum distance to a node in Dijkstra's algorithm)
- If new value is still greater than parent node value, change in place
- Alternatively, remove node from heap, `insert` node with new value into heap

Pairing Heaps: `deleteMin`

- `deleteMin` removes the root node of a Pair Heap (e.g. pop off node with shortest distance in Dijkstra's algorithm)
- The root of the heap is first removed
- The heap property must then be restored by finding a new root
- Build pairs of heaps from the children of the root
- `meld` each resultant heap from right-to-left.

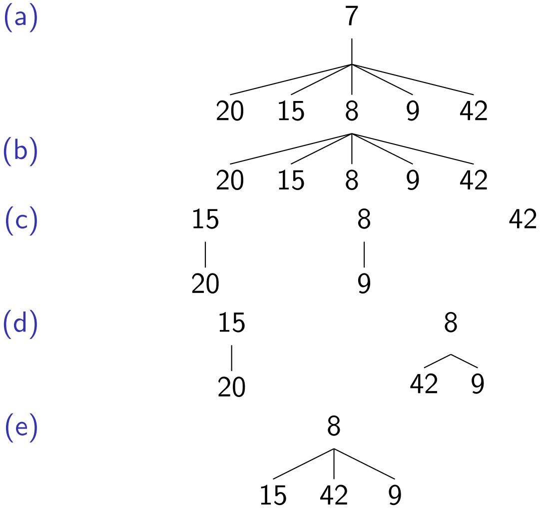
Pairing Heaps: `deleteMin` Pairing Operation

- The root of the tree has n sub-trees $H_1, H_2, \ldots, H_n$, each with their own individual roots.
- For each tuple in the set $\{(H_1, H_2), (H_3, H_4), \ldots, (H_{n-1}, H_n)\}$, we meld the two heaps together to form $\{H'_1, \ldots, H'_m\}$.
- Perform a right-iteration (similar to foldr) of `meld` over these $m = \frac{n}{2}$ (or $m = \frac{n-1}{2} + 1$) heaps to construct a newly rebalanced pairing heap
- H':

$$H' = \texttt{meld}(H'_1, \texttt{meld}(H'_2, \ldots \texttt{meld}(H'_{m-1}, H'_m) \ldots))$$

Pairing Heaps: `deleteMin` Example

- Example of the `deleteMin` operation



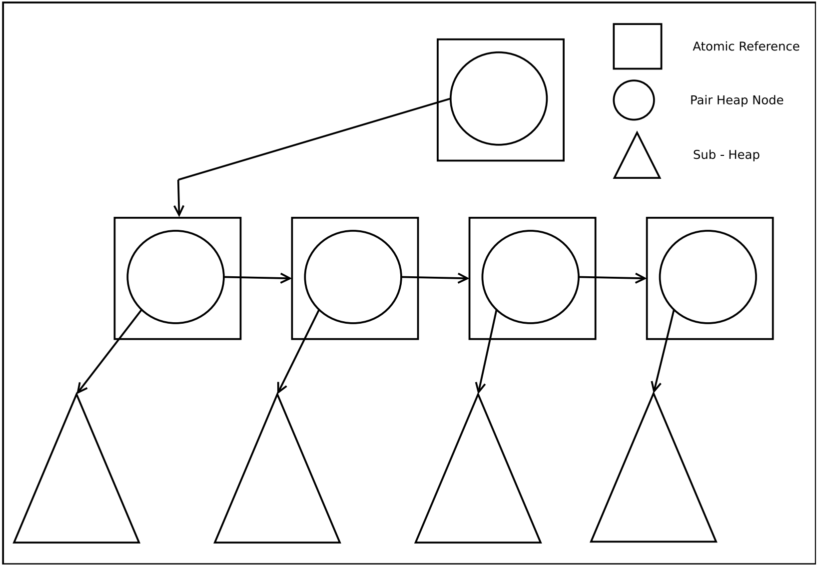
Parallelizing Dijkstra's Algorithm

- 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.
- Each node reference is updateable through a CAS operation
- In Java: `AtomicReference`
- Operations requiring linking nodes together use CAS loops
- Operations requiring node removal additionally use time stamps
- Build more complex lock-free operations from simpler ones

Lock-Free Pairing Heaps



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`.

Lock-Free `deleteMin`

- Requires the removal of root node and restructuring of heap architecture
- Root reference and timestamp will be used to update the heap

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.