# Facilitating Large-Scale Graph Search Algorithms with Lock-Free Concurrent Pair Heaps

Jeremy Mayeres
jeremym@knights.ucf.edu

Charles Newton
newton@knights.ucf.edu

Peter Tonner
ptonner@knights.ucf.edu

## ABSTRACT

This paper introduces a lock-free version of a Pairing heap. Dijkstra's algorithm is a search algorithm to solve the single-source shortest path problem. The efficiency of Dijkstra's algorithm is asymptoticly improved from $\mathcal{O}(|V|^2)$ to $\mathcal{O}(|E| + |V|\log(|V|))$ when an operation is available to decrease the recorded distance of a vertex to the target source in constant time (`decreaseKey`.) The performance of Dijkstra's algorithm also improves when threads can also perform work concurrently (in particular, when `decreaseKey` calls occur concurrently.) However, current implementations of `decreaseKey` on popular backing data structures such as Pairing heaps and Fibonacci heaps severely limit concurrency. Lock-free techniques can improve the concurrency of search structures such as heaps. In this paper we introduce `decreaseKey` and `insert` operators for Pairing heaps that provide lock-free guarantees while still running in constant time. We compare our work against a novel `decreaseKey` operator on Skiplists. Techniques for parallelizing Dijkstra's algorithm are additionally discussed.

## Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Programming Techniques; E.1 [**Lists, stacks, and queues**]: Data Structures; E.1 [**Trees**]: Data Structures

## General Terms

Performance, Algorithms, Parallel Algorithms

## Keywords

Lock-free data structures, Pairing heap, Heap, Skiplist, Skip queue, Lock-free heap, Lock-free Pairing heap

## 1. OVERVIEW OF PROGRESS

This section gives a short overview of the current state of our class project. We have developed two lock-free operators on Pairing Heaps: `decreaseKey` and `insert`. Although the remaining Pairing Heap functions do not support concurrency, our lock-free Pairing heap is complete enough to support a parallelization of Dijkstra's algorithm without compromising the constant-time performance of the `decreaseKey` operation (which allows Dijkstra's algorithm to run in $\mathcal{O}(|E| + |V|\log(|V|))$ time.) We have additionally created an optimization for Skiplists that, in some circumstances, allow calls to `decreaseKey` to execute faster than successive `delete` and `insert` calls.

## 2. INTRODUCTION

Dijkstra's algorithm [5] is a search algorithm that solves the (single-source) shortest path problem for directed graphs with non-negative weights. It has wide applications in Internet routing (see, e.g., the shortest-path calculation in the OSPF routing protocol [3]) and other scheduling algorithms that depend on finding optimal paths.

Using a naïve data structure (such as a standard binary heap) results in Dijkstra's algorithm having runtime of $\mathcal{O}(|V|^2)$, where $|V|$ is the number of vertices in the graph. Fredman and Tarjan have introduced [10] a heap variant, called the Fibonacci heap, where the `decreaseKey` operation takes $\mathcal{O}(1)$ time (amortized.) This allows Dijkstra's algorithm to run in $\mathcal{O}(|E| + |V|\log(|V|))$ time, where $|E|$ is the number of edges in the graph. However, in practice, Fibonacci heap have large constants that cause it to be slower than standard heap-backed priority queues on many practical graphs.

This deficiency led Fredman and Tarjan to develop the Pairing heap [9], which is simpler than Fibonacci heaps and has better performance in practice. Precise run-time bounds are, unfortunately, currently unknown. Pettie [22] has shown the `decreaseKey` operation takes between $\Omega(\log\log n)$ and $\mathcal{O}(2^{2\sqrt{\log\log n}})$ (amortized) time based on previous work by Fredman [8].

Current attempts to parallelize Dijkstra's algorithm rely on `decreaseKey` operations that have worse asymptotic performance than Pairing heaps provide. For example, to back Dijkstra's algorithm with Shavit and Lotan's concurrent priority queue [26] requires the user implement `decreaseKey` by deleting the target from the heap and reinserting it with a decreased value. Both of these operations occur run in $\mathcal{O}(\log n)$.

To our knowledge, there have been no attempts to construct highly concurrent lock-free heap data structures with lock-free guarantees with an efficient `decreaseKey` implementa-

```
Input: A weighted graph G = (V, E, W) and target node x
Output: The shortest distance from x to every vertex
        v ∈ V
begin
    foreach v ∈ V do
        v.priority ← ∞;
        v.previous ← NULL;
    end
    x.priority = 0;
    PQ.insert(x);
    while !PQ.empty do
        u ← deleteMin(PQ);
        foreach v where (v, u) ∈ E do
            newDist ← u.priority + weight(v,u);
            if newDist < v.priority then
                oldDist ← v.priority;
                v.priority ← newDist;
                v.previous ← u;
                decreaseKey (oldDist − newDist, v, PQ);
            end
        end
    end
end
```

**Algorithm 1:** Dijkstra's algorithm. The main loop comprises a critical section which presents difficulties when parallelizing.

tion. In this paper we introduce a lock-free variant of the Pairing heap that allows the `decreaseKey` and `removeMin` operations to be executed in parallel without the use of locks. Ignoring contention between threads, our lock-free `decreaseKey` operation has an asymptotic time complexity equal to Fredman and Tarjan's original `decreaseKey` implementation.

# 3. RELATED WORK

Currently, the most common way to implement Dijkstra's algorithm in a parallel manner is to partition the graph and apply Dijkstra's algorithm on each subgraph [4]. In this section we outline some of the novel approaches to constructing concurrent priority queues previously found in the literature.

## 3.1 Non-Lock-Free Parallel Heaps

Nageshwara et al. [21] have implemented a priority queue heap with concurrent insert and delete operations. These operations are not lock-free but represent an attempt in the literature to create a concurrent heap model. Their model only scales to $\log(n)$ processors accessing a heap of $n$ nodes. The binary heap implementation presented in their paper modifies the insert operation to work from top to bottom, rather than bottom up. This ensures that inserts can be run without deadlocks with concurrent delete operations, which also run top to bottom. While pairing heaps do not require the same method for inserting and deleting, this style of operation design, where different operations are forced to traverse the tree in the same direction, may be useful in the lock-free pairing heap solution.

Driscoll et. al [6] have introduced a variant of Fibonacci heaps, called relaxed heaps, that allows for easier parallelization. However, their approach is complicated [7] and, in practice, performs worse than Fibonacci heaps [20].

## 3.2 Automatically Generated Lock-Free Heaps

In this section we consider how some approaches that automatically transform sequential data structures into lock-free, concurrent data structures could be applied to concurrent pairing heaps.

Herlihy [12] has introduced a universal method of automatically transforming sequential objects into concurrent, lock-free objects with very few assumptions about the behavior of the sequential object. In his method, all threads share a pointer to the sequential object and a version / timestamp of the object. When a concurrent operation is called, the object is copied into a new location in memory and operated upon with a corresponding sequential operation. The shared pointer is updated if the current timestamp matches the timestamp noted at the beginning of execution. Otherwise, the operation is repeated with a new copy of the object. This could lead to the construction of a lock-free pairing heap that copies itself into memory, performs the related sequential operation, and updates the state of the object to reflect these modifications (assuming no other thread has made changes.) However, as Herlihy notes, this automatic transformation is only suitable for objects which can be efficiently copied (i.e., "small objects.") This would not be appropriate for our particular use-case since we are considering very large heaps.

Anderson and Moir [1] have improved on Herlihy's approach by automatically keeping track of locations in memory a sequential operation reads from and writes to. Only modified blocks of memory are copied. Hence, this solves some issues with performance when only small portions of a data structure are modified. For example, consider updating the head of a linked-list backed queue. With Herlihy's technique, the entire list would need to be copied, but the modification to the data structure only involves updating the head pointer. However, this method handles contention poorly if many operations access the same blocks of memory (as is the case with the `deleteMin` operation of a heap.)

## 3.3 Concurrent Priority Queues and Heaps

Shavit and Lotan have constructed [26] a concurrent priority queue, called a SkipQueue, based on Pugh's SkipList [24] data structure. Sundell and Tsigas [28] have improved on Shavit's and Lotan's method to create a fully lock-free concurrent priority queue.

Hunt et al. [14] have introduced an array-backed priority queue with very fine-grained locks. Their approach builds on previous work by Rao and Kumar [25] that showed contention on heaps can be greatly decreased by having the `deleteMin` and `insert` operations proceed in the same (top-down) directions. (If the operations proceed in opposite directions, this opens a possibility for deadlocks.) However, this increases the time complexity of an `insert` operation to $\mathcal{O}(\log n)$. The main contribution of Hun et al.'s approach is that $\mathcal{O}(1)$ performance on a bottom-up `insert` can be preserved without introducing the possibility for deadlocks by 1) ordering the acquisition of locks, 2) applying tags / timestamps to avoid the ABA problem, and 3) avoiding overlap of search paths.

Israeli and Rappoport have developed [15] a wait-free concurrent heap. However, their implementation uses atomic operations that are not widely available in today's hardware.

### 3.4 Concurrent Fibonacci and Skew Heaps

Jones [16] has constructed a concurrent Skew heap that uses fine-grained locks (as opposed to locking the entire heap structure) to allow parallel `insert` and `deleteMin` operations without violating the heap invariant. In a Skew heap, each node has three pointers: two children pointers and one sibling pointer that points to the next node at the same depth. Jones defines a bubble as the set of all pointers a Skew heap operation will read or modify before completing its task. An operation must acquire all the locks in its bubble before performing any modifications. This granular lock structure allows disjoint heap operations to complete concurrently.

Huang and Weihl [13] have constructed a concurrent Fibonacci heap that allows threads to concurrently perform `insert`, `decreaseKey`, and `deleteMin` operations. The data structure is backed by a circular doubly-linked list. An issue with Huang and Weihl's approach is that `deleteMin` is not linearizable: threads may access the minimum element of the Fibonacci heap out of order, (i.e., `deleteMin` does not necessarily return the minimum element in the heap at the point in time it is called), although the element is guaranteed to be reasonably small ("promising") compared to the other elements in the heap. However, this presents problems with Dijkstra's algorithm and other graph algorithms that require a strict definition of `deleteMin`.

### 3.5 Other Highly Concurrent Attempts

Michael and Scott's [18] lock-free FIFO queue allows enqueue and dequeue operations to proceed concurrently on a single link-list backed queue. Moir et al. have developed [19] a variant of the Michael-Scott queue that applies an elimination back-off technique to eliminate contention for the head / tail regions of the linked list. Previously, this technique was used by Shavit and Touitou [27] to create concurrent counters and Hendler et al. [11] to create concurrent LIFO stacks. Moir's technique is a natural generalization of the elimination approach used in LIFO stacks: have the backed-off enqueue operations eliminate against a backed-off dequeue operation once all the elements prior to the back-off point have been dequeued.

## 4. SKIPLISTS

Skiplists were introduced by William Pugh [24] as a way to avoid expensive rebalancing in tree structures while still retaining desirable properties of balanced trees such as $\mathcal{O}(\log n)$ search. The basic idea behind Skiplists is to allow the tree structure to be controlled by a pseudorandom number generator (PRNG). Thus, a Skiplist only gives probabilistic guarantees that it will produce a balanced tree structure. However, pathologically bad structures that cause the data structure to degrade to a search complexity of $\mathcal{O}(n)$ are very unlikely. Also, since Skiplists do not rebalance themselves, input keys are typically uniformly mapped into another set (e.g., by using a hash function) to avoid pathological inputs.

A Skiplist is a set of linked lists with ordered elements. Each linked list is assigned a level $0 \leq i \leq M$ which describes the density of the linked list at that level. The first level (level 0) contains all the elements in the Skiplist. We adopt the notation that $L(i)$ denotes the list at level $i$. $L(0)$ (the bottom-most level) is guaranteed to contain all the elements inside the Skiplist. For each higher level $i > 0$, $L(i)$ is expected to contain $\frac{1}{2}|L(i-1)|$ elements. In Fig. 1, a Skiplist containing $L(0) = \{1, 3, 9, 11, 15\}$ is presented.

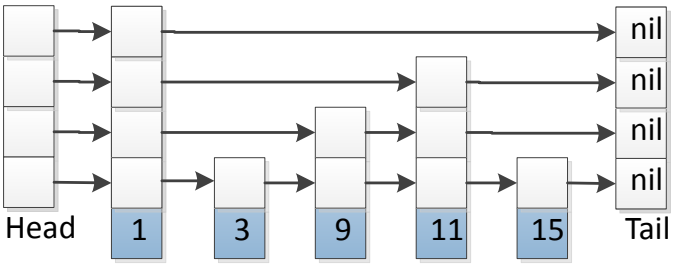


**Figure 1: A Skiplist representation of $\{1, 3, 9, 11, 15\}$. The levels of each node, respectively, are 3, 0, 1, 2, and 0.**

### 4.1 Core Operations

We now outline a few operations on Skiplists. In general, these are quite similar to their linked-list counterparts. In fact, each Skiplist operation can be decomposed into a sequence of linked-list operations that each affect one level of the Skiplist.

Suppose we want to search for element $e$. First, we look at the list at $L(M)$ and traverse it linearly until we encounter an element greater or equal to $e$. If we encounter $e$, then we stop. If we found an element greater than $e$, we repeat this procedure from this new starting point. If we only encountered the tail of the list, then we repeat the procedure starting from the first element in $L(M-1)$. This procedure repeats until we successfully locate $e$. If we encounter either an element greater than $e$ or end of the list at $L(0)$ we conclude $e$ is not contained in the Skiplist. See Figs. 2 and 3 for sample search runs on the Skiplist presented in Fig. 1.

To insert an element $e$ into a Skiplist, we first perform a search for the element as previously described. However, we also keep track of additional information. The insertion point is determined by where we stop searching at $L(0)$. Throughout the search process, an array `previous`$[i]$ is maintained that contains all the right-most pointers (at level $i$) to elements left of the insertion point. An array `next`$[i]$ is given the previous values of each pointer in `previous`$[i]$. To insert $e$, we assign it a random level $l$, set all the right-most pointers in `previous`$[i]$ for $0 \leq i \leq l$ to $e$ and set the pointers emanating from $e$ to `next`$[i]$ for $0 \leq i \leq l$. Insertion has the visual effect of simply splicing $e$ into the Skiplist (see Fig. 4.)

To delete an element $e$ from a Skiplist, we first search for the element (as we did for insertion). If the element is not found, we are done. Otherwise, we should map each pointer in `previous`$[i]$ to the value in `next`$[i]$ for $0 \leq i \leq l$, where $l$ is the level of $e$. See Fig. 4 for an illustration of deleting an element from a Skiplist.

## 4.2 Additional Operations

Pugh has defined [23] a number of useful operations such as merging Skiplists, splitting Skiplists, and concatenating two Skiplists together when all the keys in one Skiplist are less than or equal to the smallest key in a second Skiplist. However, an operation that decreases the value of an element in a Skiplist (besides simply deleting the old value and reinserting the decreased value) was not present and we were unable to find a description of this operation in the literature.

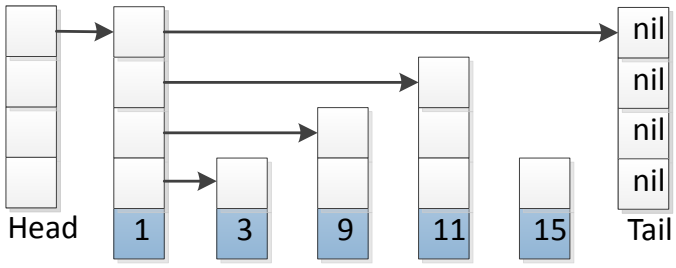


**Figure 2: Skiplist traversal for finding the element 3 in the Skiplist presented in Fig. 1**

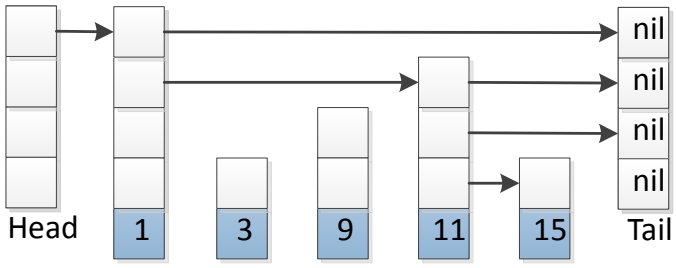


**Figure 3: Skiplist traversal for finding the element 15 in the Skiplist presented in Fig. 1**

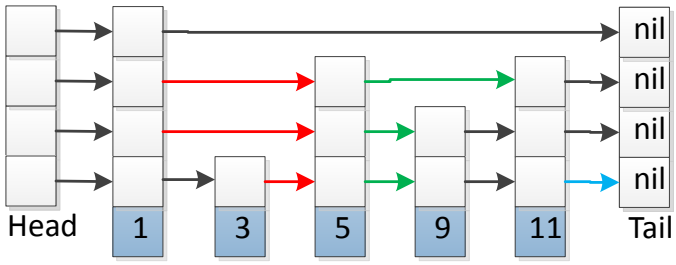


**Figure 4: Illustration of insertion and removal. 5 has been inserted into the Skiplist. Red indicates pointers contained in `previous`$[i]$, and green indicates pointers contained in `next`$[i]$. 15 has also been removed from the Skiplist and updated pointers for this operation are in blue.**

## 4.3 Lock-Free Skiplists

Shavit and Lotan's Skipqueue [26] is a modification of Pugh's Skiplist that gives its operations lock-free guarantees. We present a high-level technical summary of their work and in Section 4.4 we will build a `decreaseKey` operator for Skiplists using their approach. Following with our analogy of Skiplist operations being composed of operations on the linked lists formed at each level of the Skiplist, we see that lock-free operations on Skiplists can be composed of operations on lock-free linked lists.

One subtle issue with Shavit and Lotan's modifications is that $L(i)$ is not necessarily a subset of all $L(k)$ (for $k < l$): a value's membership is determined by it being in the lowest level of the Skiplist. In particular, this means that even if we encounter a target value in a higher level of the Skiplist, we still must verify it is linked to the structure in $L(0)$.

The search, add, and delete operations of a lock-free Skiplist follow the same form as those we defined on Pugh's Skiplist. However, here we need to handle the possibility of threads concurrently modifying the Skiplist. To detect concurrent operations, we modify our linked list's node datatype from the 2-tuple (`value`, `nextPtr[]`) to the 3-tuple (`value`, ⟨`bool`, `nextPtr`⟩`[]`), where `nextPtr` is the datatype used for pointers, and ⟨`bool`, `nextPtr`⟩ are written / read in a single atomic operation. In Java, this can be implemented with the `AtomicMarkableReference` class. In C/C++, this can be easily implemented by stealing a bit from a word-aligned pointer. The boolean value serves as a deletion bit that is set whenever a node is removed from the Skiplist but has yet to be physically removed from the linked list structure. This is important for avoiding certain interleavings of operations. Consider two threads concurrently accessing a Skiplist. The first thread reads the deletion mark on an element $a$ at level $i$, and is interrupted by a second thread which marks the element as deleted. When the first thread resumes, it will still assume that $a$ is not marked and, if inserting a new element, could create an untraversable section of the Skiplist by building on $a$'s next pointer.

To search for an element in a lock-free Skiplist we take the same approach as before, but we additionally remove any marked nodes from the linked list structure and we always traverse down to $L(0)$ in order to verify membership. To add an element into the lock-free Skiplist we construct the `previous[i]` and `next[i]` sets as before and set the next pointers of the new node to `next[i]`. We also update the pointers in `previous[i]` to point to our new node (from level 0 to level $l$). If we are unsuccessful at having `previous[0]` point to our new node, we restart the whole procedure since either a new node was inserted and the ordering of $L(0)$ may be violated if we insert or the node at `previous[0]` was marked for removal. Note that the linearization point of an add occurs when the element is inserted into $L(0)$. Once `previous[0]` is updated, we attempt to update `previous[i]` to `next[i]` in a CAS loop. To handle concurrent changes during this process (e.g., `previous[i]` being marked), we recompute the `previous` and `next` arrays whenever an atomic update fails.

Removing an element from a lock-Free Skiplist is analogous to adding an element. First, we search for the target element and compute our `previous[i]` and `next[i]` arrays. Next, we mark each `next` pointer of the target element in a top-down fashion within a CAS loop. When a CAS fails, we re-read the relevant `next[i]` as another thread could have modified it (e.g., a new node was linked to the node we're currently deleting.) When the `next` pointer has been

marked in $L(0)$, we consider the element removed (this is our linearization point.) There is no need update any pointers on a removal since subsequent searches will perform this for us.

## 4.4 A decreaseKey operator for Skipqueues

Keeping track of the `next[i]` and `previous[i]` "windows" of an element in a Skipqueue leads to a natural implementation of a `decreaseKey` operator where we re-use information from the deletion call. Suppose we have located the target element in the Skipqueue and it has a level of $j$. Let $i^*$ denote the index of the largest value in the `next` array that is smaller than the new value of the target. We can use `next`[`i`$^*$] as the starting location for determining a new location for the decreased value of our target (since every node after `next`[`i`$^*$] is greater than our new value.) Aside from the head of the list, the insert is performed the same as before. If the attempt to insert needs to be restarted, we restart as a non-optimized insertion.

The level of the decreased element is bounded above by $j$. On the first (the optimized) insertion attempt we set the level of the decreased element equal to $j$. This avoids introducing bias into the Skipqueue structure: if we sampled the new height from the discrete uniform distribution $U(0, j)$ we would force elements to gradually lose levels. If the optimized insertion attempt fails, there is no restriction placed on the level of the element.

# 5. PAIRING HEAPS

Pairing heaps were developed by Fredman et al. [9] as a simpler alternative to Fibonacci heaps that, at least empirically, do not sacrifice performance. Pairing heaps are a popular backing data structure for Dijkstra's algorithm due to its offering of an efficient `decreaseKey` operator without involving complicated tree rebalances.

## 5.1 Operations on Pairing Heaps

We provide a sketch of the salient features of pairing heaps. The main operator defined on pairing heaps is `meld`, which accepts two heaps and uses a compare-and-link approach to join its inputs together. See Fig. 5 for an example of melding two heaps together. The roots are compared and the root with the largest value (assuming we have min-heaps) is added as the left-most child of the smaller root. Insertion of a node is performed by constructing a trivial heap containing only the relevant node and melding it the main heap.

The supported pairing heap operations are as follows:

`makeHeap`$(h)$

`findMin`$(h)$

`insert`$(x, h)$

`deleteMin`$(h)$

`meld`$(h, h')$

`decreaseKey`$(\Delta, x, h)$

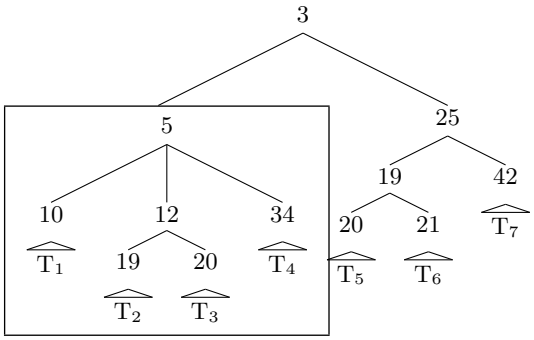


**Figure 5: Melding two min-heaps together (one with root 3 and the other with root 5) with the `meld` operator. The heap rooted at 5 becomes a subheap of the heap rooted at 3.**

`delete`$(x, h)$

where $h, h'$ are pairing heaps, $x$ is a target node (or, equivalently, a target value), and $\Delta > 0$ is the value by which the target node should be decremented.

`deleteMin` removes the minimum (root) element of a pairing heap $H$. Say once the root of the tree is removed we have $n$ 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 $\{H'_1, \ldots, H'_m\}$. It is worth mentioning that this pattern of grouping into tuples (pairs) is where pairing heaps derive their namesake. If $n$ is odd, $H_n = H'_m$ is not melded. Finally, we 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 $H'$:

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

To make the ordering $H_1, H_2, \ldots, H_n$ unambiguous, we say that $i^{\text{th}}$ child of $H$ (i.e., the root of $H_i$) is the $i^{\text{th}}$ most recently melded node to the root. Note here that `deleteMin` rebalances the tree structure of the pairing heap. See Fig. 6 for an illustration of calling `deleteMin` on a Pairing heap.

`decreaseKey` is implemented as follows. Suppose a node $x$ has its value decreased and this new value violates the heap invariant (i.e., $x$ is now smaller than its parent.) The subtree rooted at $x$ is removed from the tree and melded to the parent tree as previously described.

`delete`$(x, h)$ is implemented by delinking the sub-tree $S$ rooted at $x$ from $h$, calling `deleteMin` on $S$, and melding the resultant tree back to $h$.

# 6. PARALLELIZING DIJKSTRA'S ALGORITHM

We now consider some issues pertinent to parallelizing a heap-backed implementation of Dijkstra's algorithm. If `deleteMin` does not remove and return the unvisited node in the graph with the smallest distance from the source node, Dijkstra's algorithm is not guaranteed to solve the shortest-

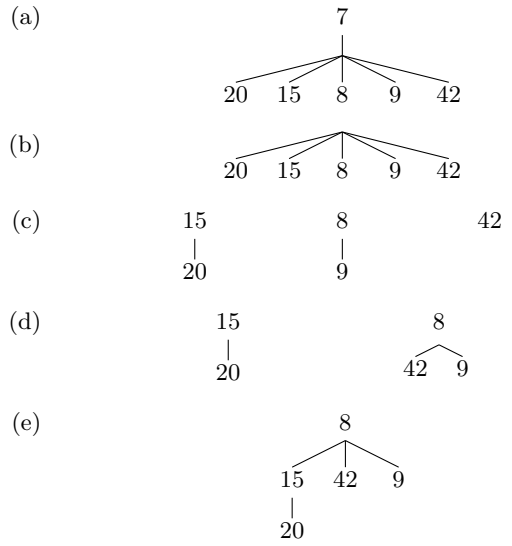


**Figure 6: Calling `deleteMin` on a Pairing heap (a). The root is removed (b) and the children are melded in pairs (c), and then the pairs themselves are melded in right-to-left order (d-e). Note that the resulting heap (e) is more balanced than (a).**

path problem. A naïve approach to running Dijkstra's algorithm in parallel would be to let calls to `deleteMin` and `decreaseKey` haphazardly interleave. However, the minimum of the heap will change on some call to `decreaseKey` if a new value of a node is less than the current value. Thus, some comparison is needed to see if calls to `decreaseKey` will produce a new minimum value in the heap. There are three basic approaches to solving this problem:

1. Optimistically continue removing minimum elements from the heap.
2. Pessimistically check if any of the new distances are less than the minimum of the heap, and continue if the minimum will remain unchanged. This has an advantage over the next method where the decision to continue can be made without any modifications to the heap.
3. Pessimistically assume that `decreaseKey` will always produce a new minimum and ensure that all calls to `decreaseKey` complete before the next iteration's call to `deleteMin` executes.

Solution 1 requires complicated interactions between concurrent states of the heap or requires independent copies of the heap to be made and later merged. Since there are two choices for every iteration of Dijkstra's algorithm (either the heap's minimum changes or does not change), a copying approach requires $\mathcal{O}(2^n)$ copies to be made, which is intractable for large graphs.

Solutions 2 and 3 are both practical. We opt for solution 3 since it is simpler, but for future work we may consider solution 2.

# 7. LOCK-FREE PAIRING HEAPS

In this section we develop lock-free `insert` and `decreaseKey` operators for Fredman et al.'s pairing heap. Our work relies on the observation that the specifics of Dijkstra's Algorithm allow us to make assumptions about how pairing heap operations interleave. We first list some assumptions that we have made about overlapping calls to operations on the heap. Then, we give a high-level overview of our lock-free approach before delving into the details of our `insert` and `decreaseKey` operators.

In section 6 we analyzed the specific performance bottlenecks and other issues with running Dijkstra's algorithm in parallel. In particular, we noted that the linearization guarantees provided by presently available lock-free heaps are insufficient for blindly executing concurrent calls to `deleteMin` and `decreaseKey`. Hence, we assume these calls will never interleave, and, further, that only concurrent calls to `decreaseKey` overlap.

We use a CAS loop to insert a new node (or a newly decreased node) at the root (or as a root's child.) A call to `meld` modifies the list of children of the new heap root, which is the root node of minimum value between two heaps being melded. The newly added node is inserted at the end of the root's child list. This is accomplished atomically through the references that heap nodes hold. Each list of subheaps is a lock-free list: this ensures that two overlapping `meld` operations will not negatively conflict with each other. We now describe the linearization point for two interleaving `meld` operations. A `meld` operation has completed when the new child node has been appended to the child list of the heap root or the root has been atomically placed with a CAS. After this, the new node must have updated references for its parent (root) and right child (null). To achieve this, a new node is created inheriting the children of the node to be inserted with new values for parent and sibling.

`insert` is the simpler of our two lock-free operations. We first take a snapshot of the address of the current root node and its timestamp. This is later used to detect if any concurrent operations have interfered with the root node. Next, we create a copy of the recorded root. The copy and the original root both share the same list of child heaps. Next, we create a trivial heap from the element we are inserting and `meld` this to the copied root. `meld` has the effect of setting the parent field of the smaller heap to be the larger heap, and appending the larger heap to the list containing the subheaps that the larger heap is a parent of. This list of subheaps is a lock-free list, so there are no issues with concurrent modifications. Next, we attempt to CAS the root node of the melded heap to the root node of the entire heap. If the CAS call succeeds, we are done. Otherwise, we retry. If the CAS fails, we remove the larger node from the smaller node's list of subheaps before retrying (this avoids accidentally duplicating nodes.)

To perform a lock-free `decreaseKey` we consider three mutually exclusive cases. Some of our decisions here rest on the assumption that no `decreaseKey` calls to the same node interleave. This is true if Dijkstra's algorithm is run on a graph where the set of edges $E$ is a set but not a multiset (so, there are no duplicated edges.) The implications of this

assumption are that calls to `decreaseKey` can freely change the parent field of the target node without worrying about contending changes.

For the first case, if the parent node of our target node is smaller than the decreased value, we immediately decrease the value of the target node without making any structural changes. This does not interfere with any interleaving calls to `decreaseKey` since this is the only call that would change the target node's parent field and we do not need to modify the root node at all. For the second case, if the target node is the root node, we record a pointer to the root node and its timestamp, create a copy of the recorded root, modify its value, and try to CAS in the new root. We also do not need any structural changes in this case, but after every failed CAS we verify that the target node is still the root node. If this is no longer the case, we handle the insertion as another case (first, if applicable, otherwise as the third.)

For the third case we have a node where its decreased value is less than its parent value. Here, the heap invariant is violated and our target node needs to be melded to the root. We immediately decrease the value stored at the target node and remove it from the parent's list of sub heaps. Siblings of the target node must have updated references to fill the gap caused by removing the target node. We then follow a similar procedure as in `insert`. We construct a copy of the root node, merge the decreased node and the root, and attempt to CAS the new root into place. To prevent duplicated nodes, if the CAS fails and the target node would have become the new root, we remove the old root from its list of subheaps. The structure of the heap is modified in this case, and there is a time when the target node does not exist in the heap after removal from the child list of the heap and before the target node has been reinserted. This is only an issue if the target node would become the new heap root and a `deleteMin` call were to delete the current root. In this case, the linearization would be that of the `deleteMin` occuring first and the `decreaseKey` occuring second. It is worth mentioning the case where a parent is decreased concurrently with its child. Although the delinking of the child may not be necessary, in this case it is performed anyway. The interleaved operations linearize as if the largest node was decreased before the smallest.

## 7.1 Future Operators

We briefly discuss the form of a parallelized `deleteMin` operator on Pairing heaps. `deleteMin` follows what is essentially a Map-Reduce pattern: pairs of heaps are constructed and then merged together to form a resultant heap. The first step has no contention between pairs and is trivial to parallelize. Our `meld` operator can be used to merge heaps in parallel.

# 8. EXPERIMENTAL RESULTS

Efficiency of the lock-free pairing heap will be examined by comparing the relative performance of Dijkstra's algorithm, backed by different heap structures, on large network data sets. Currently, we plan to test three variants of Dijkstra's algorithm. The first variant will backed by a normal Skipqueue that performs a `decreaseKey` by removing the target node and readding it with a decreased key value. The second variant will be backed with a Skipqueue that uses our optimized `decreaseKey` operator. The third variant will be backed with our lock-free pairing heap. We will use real world data sets collected from the Stanford Large Network Dataset Collection [17] and Harvard's Human Interactome Database [2]. We will additionally use two synthetic graphs: a randomly generated dense graph and a randomly generated sparse graph.

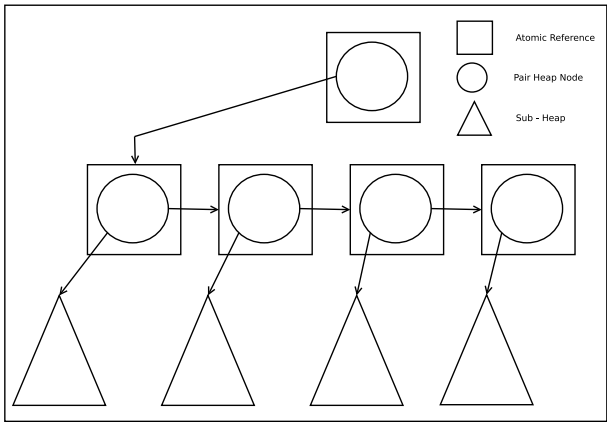


**Figure 7: Illustration of the concurrent pair heap structure. Pair heap nodes exist inside atomic references and hold atomic references to other heap nodes.**

# 9. REAL-WORLD PERFORMANCE OF LOCK-FREE PAIRING HEAPS AND SKIPQUEUES

In non-lock-free Pairing heaps, nodes are inserted (a similar procedure is used for `decreaseKey`) into the heap by either replacing the root of the heap or inserting itself as a subheap to the root. It is clear to see that parallel variants that preserve the general form of insertion operations will place un-due contention at the root of the data structure. This is the main issue we foresee with our lock-free pairing heap: always inserting at the root is a performance bottle-neck. Skipqueues, on the other hand, can insert elements in, e.g., the middle of the list in a less-contended way.

However, insertions into a Pairing heap run in asymptotically less time than a comparable Skipqueue insertion: a Pairing heap insertion runs close to $\mathcal{O}(1)$ time but a Skipqueue insertion runs in $\mathcal{O}(\log n)$ time. This is due to the differing methods used to balance the heap structure.

# 10. REFERENCES

[1] J. Anderson and M. Moir. Universal constructions for large objects. *Parallel and Distributed Systems, IEEE Transactions on*, 10(12):1317 –1332, dec 1999.

[2] Center for Cancer Systems Biology. Human interactor database.

[3] R. Coltun, D. Ferguson, J. Moy, and A. Lindem. OSPF for IPv6. RFC 5340 (Proposed Standard), July 2008.

[4] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstra's shortest path algorithm. In L. Brim, J. Gruska, and J. Zlatuška, editors, *Mathematical Foundations of Computer Science 1998*,

volume 1450 of *Lecture Notes in Computer Science*, pages 722–731. Springer Berlin / Heidelberg, 1998. 10.1007/BFb0055823.

[5] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959. 10.1007/BF01386390.

[6] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: an alternative to fibonacci heaps with applications to parallel computation. *Commun. ACM*, 31:1343–1354, November 1988.

[7] A. Elmasry. The violation heap: A relaxed fibonacci-like heap. In M. Thai and S. Sahni, editors, *Computing and Combinatorics*, volume 6196 of *Lecture Notes in Computer Science*, pages 479–488. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-14031-0_51.

[8] M. L. Fredman. On the efficiency of pairing heaps and related data structures. *J. ACM*, 46:473–501, July 1999.

[9] M. L. Fredman, R. Sedgewick, D. D. Sleator, and R. E. Tarjan. The pairing heap: a new form of self-adjusting heap. *Algorithmica*, 1:111–129, January 1986.

[10] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM*, 34:596–615, July 1987.

[11] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '04, pages 206–215, New York, NY, USA, 2004. ACM.

[12] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15:745–770, November 1993.

[13] Q. Huang and W. Weihl. An evaluation of concurrent priority queue algorithms. In *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on*, pages 518 –525, dec 1991.

[14] G. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Proc. Letters*, 60:151–157, 1996.

[15] A. Israeli and L. Rappoport. Efficient wait-free implementation of a concurrent priority queue. In A. Schiper, editor, *Distributed Algorithms*, volume 725 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 1993. 10.1007/3-540-57271-6_23.

[16] D. W. Jones. Concurrent operations on priority queues. *Commun. ACM*, 32:132–137, January 1989.

[17] J. Leskovec. Stanford large network dataset collection, 2012.

[18] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[19] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 253–262, New York, NY, USA, 2005. ACM.

[20] B. M. E. Moret and H. D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree, 1994.

[21] R. Nageshwara and V. Kumar. Concurrent access of priority queues. *Computers, IEEE Transactions on*, 37(12):1657 –1665, Dec 1988.

[22] S. Pettie. Towards a final analysis of pairing heaps. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 174 – 183, oct. 2005.

[23] W. Pugh. A skip list cookbook. Technical report, University of Maryland, 1990.

[24] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33:668–676, June 1990.

[25] V. N. Rao and V. Kumar. Concurrent access of priority queues. *IEEE Trans. Comput.*, 37:1657–1665, December 1988.

[26] N. Shavit and I. Lotan. Skiplist-based concurrent priority queues. In *Parallel and Distributed Processing Symposium, 2000. IPDPS 2000. Proceedings. 14th International*, pages 263 –268, 2000.

[27] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. *Theory of Computing Systems*, 30:645–670, 1997. 10.1007/s002240000072.

[28] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.*, 65:609–627, May 2005.