Hollow Heaps

R. Frederic Sauve-Hoover

April 21, 2020

Description

I implement two-parent hollow heaps as described in [Hansen et al., 2015] as a python package. Hollow heaps are a heap implementation with improved complexity compared to more typical priority queues (heapq in python) on par with fibonacci heaps. However, hollow heaps are significantly simpler to implement than fibonacci heaps.

The goal of this is to be able to provide an importable python library containing an implementation of a fast heap and to get some idea of how well it compares to existing python libraries.

Time Bounds

Using the two-parent hollow heap implementation, we have all usual operations: *insert*, *find_min*, *meld*, etc. taking $O(1)$ time worst-case, except for *delete* which takes $O(1)$ per hollow node losing a parent and per link being added, as well as $O(logN)$, where N is how many nodes we have in our dag prior to the *delete* or *delete_min* operation being done (*delete* and *delete_min* are trivially equivalent in time bounds since *find_min* is $O(1)$). The more exact proofs for the time bounds are in the paper and go into significantly more detail than I am in this brief overview.

For operations such as *find_min* we can trivially see that they're $O(1)$ since they're simply acessing attributes of our root. For *insert* and *meld* it's less obvious, but we're simply linking the new node to our root, which is clearly $O(1)$, with the ranking and re-ordering being left until the *delete* operation.

Resources

I follow the algorithm provided in [Hansen et al., 2015], which provides much of the basis for my code.

Instructions

To use the package, import **hollow_heap** as shown in the snippet below

```
# Element is used if you want to store additional data alongside the
# key
from hollow_heap import HollowHeap, Element

heap = HollowHeap()
e = Element([1,2,3]) # a simple element containing a list
heap.insert(4, e)

# an insertion that doesn't use an element object
heap.insert(5)

# delete_min
heap.delete_min()

# to do a delete, or decrease_key operation on an element, you must have the element initialize
```

```
heap.decrease_key(e, 1)
heap.delete(e)
```

There aren't any required libraries to install to use **hollow_heap**, pytest is however required to run **sorting**, to do so run *pip install pytest*

To run the tests, run *pytest tests.py*

To run the benchmark, run *python benchmark.py*

Assumptions

There are a few assumptions, mostly to do with the usage of the library. These are things I may improve some other time.

- inserts must be keys that are not already present in the heap
- meld will only be called on heaps with distinct items
- decrease_key is only called on items with a key ¿ k
- delete is only called on an item that exists in the heap

Files in directory

Here are the various files in the directory and a short explanation of each

- **hollow_heap.py** The hollow heap library, import the usual python way
- **tests.py** unit tests for the hollow heap library, run with *pytest tests.py*
- **dijkstras.py** Two Dijkstra's algorithm implementations using heapq and hollow heaps, to test the difference in performance on dense graphs.
  With hollow heaps, we get a $O(|E|+|V|\log|V|)$ time bound vs the usual $O(|E|\log|E|)$, so we should expect to see the hollow heap implementation perform roughly the same as the heapq implementation on sparse graphs, and improve the denser the graph
  In my test on a path graph, we see the heap implementation significantly outperform the hollow heap implementation, which is not surprising since *decrease_key* will never be used on a path, and for the random sparse graph, we see both algorithms performing roughly the same, which is also to be expected. Also as expected, but still fairly interesting to note, is that on denser graphs we see the hollow heap implementation performing significantly better (4-5x as fast), which is due to *decrease_key* being a $O(1)$ operation.
  NOTE: The graph generation can take a little while when NUM_NODES is high for the dense graphs (usually ¡1min)
- **sorting.py** Sorting benchmarks, times execution of various sorts comparing heapq, hollow heaps, and builtin sort. Interestingly enough I wasn't able to even match the speeds given by heapq and sort (sort isn't surprising), and this is likely in no small part due to implementations of standard python libraries being mostly underlying c code, and some inefficiencies likely present in my code. However, even with optimizations I wouldn't expect hollow heaps to outperform heapq by much if at all, since heapsort does one insertion for every deletion (so the asymptotic bounds don't change between hollow heaps and heapq) and doesn't rely on decrease-key which is the particularly significant improvement hollow heaps bring.
- **graph.py** The graph library from cpmut 275. Used for Dijkstra's implementation for convenience

Output of program

hollow_heap.py has no output since it's just library code
tests.py is typical pytest output, if a test fails it will output where the failure occurs, otherwise it'll output how many tests were passed and how long it took

sorting.py outputs the sorting benchmark being done, and the time taken by each algorithm
dijkstras.py outputs the type of graph dijkstras is being run on and the time taken by each algorithm

References

[Hansen et al., 2015] Hansen, T. D., Kaplan, H., Tarjan, R. E., and Zwick, U. (2015). Hollow heaps.