OUTER LOOP
CONSULTING

White Paper: Performance effects of heap structure choice for Path Finding – A Traffic Modelling Perspective

Introduction 2
Profiling 2
Data structures and why it they matter 2
What is a heap? 3
How are they implemented 3
Tree-based heaps 3
Array-based heaps 3
Variety of Implementations 4
Theorised Performance 5
CPU Cache 5
Benchmark Methodology 6
Results 7
Performance across project size 7
Further Analysis 8
Cachegrind results 9
Differences in k-heap implementations 11
Effects and considerations of multithreading 13
Conclusion 14

Introduction

As part of a wider effort to improve the overall performance of the AequilibraE transport modelling package, we have recently[1] identified that the current performance of the traffic assignment module lags behind other general purpose path finding libraries.

In this paper we will discuss why the choice of heap structure was the primary reason for the performance gap, how we identified this and consequently selected an alternative that is better suited to the types of networks commonly found in transport models.

Profiling

Any runtime optimisation exercise that aims for success starts by empirically evaluating the existing performance bottlenecks that need to be relieved. For our initial profiling runs we used utilised yappi (Yet Another Python Profiler) and snakeviz to profile a typical networking skimming use case.

Network skimming involves finding the shortest path for every origin of interest to every destination, referred to as all-to-all pathfinding. Skimming then walks these paths to determine the cost of taking the given path. This is achieved through Dijkstra's algorithm, which computes the minimum spanning tree – the set of shortest paths from an origin to all destination. We profiled the entire runtime of AequilibraE's skimming functionality, which included our implementation of Dijkstra's algorithm and cascade skimming.

From this initial profiling it became apparent that, as expected from the literature and as theorised in our previous work, AequilibraE's path building is bottlenecked on the heap method `remove_min` which was responsible for 87% of the total runtime. It was definitive that this was where our optimisations efforts should be focused.

Data structures and why it they matter

The process of traffic modelling requires repeated computation of the shortest path to get from an origin to a destination to assign traffic in the first case. This is normally achieved by using Dijkstra's algorithm to build paths for every origin to every destination each iteration of traffic assignment.

To implement Dijkstra efficiently we need a fast way to store the cost to reach each node and quickly retrieve the smallest one. This use case is well served by the abstract data structure class of priority queues. Naïve implementations of priority queues as arrays have linear time complexities for insertion and removal depending on whether it is sorted or not, whereas heap-based implementations are able to execute both operations in logarithmic time. There are other specialised implementations such as a bucket queue, which operate on integer keys. We decided to examine the heap implementations due to their superior time complexities and flexibility given the weights on traffic networks tend to be floating points.

[1] https://www.linkedin.com/pulse/faster-path-building-transport-networks-outer-loop-consulting

OUTER LOOP
CONSULTING

What is a heap?

A heap is an implementation of a priority queue which makes use of a tree structure with two rules to order nodes in a given priority order. A node stores references to its parent, children, and a value representing its priority. A heap which contains the largest value at the root is called a maximum heap, whereas a minimum element corresponds to a minimum heap. We will refer to minimum heaps for the remainder of the post since that is the form most useful in Dijkstra's algorithm.

Each node maintains a reference to its children and to its parent. The two rules of a basic heap implementation are that its order cannot be violated – a child to a node must always be greater than or equal to its parent, and the heap satisfies complete and proper tree properties. A complete tree means each layer is filled before moving to the next level, and a proper tree means the nodes on the bottom level are filled from the left to right. Each time a node is removed from the heap, the heap corrects its order to maintain these rules, thereby guaranteeing the next smallest element becomes the new root. For those who are interested, this is a good website to visualise the actions of different heaps (https://www.cs.usfca.edu/~galles/visualization/Heap.html)

How are they implemented

There are two implementations of heaps which work well for different problems: tree-based and array-based implementations. Tree-based heaps use pointers to store the connections between each node, whereas an array-based heap maintains the structure through an ordered array.

Tree-based heaps

Tree-based heaps maintain the tree structure through pointers to a node's parent and children alongside the data of the node. These heaps are simpler to implement and work well for more complicated heap structures with abstracted logic. Tree-based heaps don't require contiguous chunks of memory to store its structure.

Array-based heaps

Array-based heaps make use of the heap rules to store the tree in an in-order ordered[2] array. The requirement that the heap is complete and proper guarantees that there are no null entries before the last element. Consequently, the heap inherits the cache locality advantages of a contiguous array and removes the need for storing pointers in the node, as an arithmetic relationship between node parents can be derived from the index. For example, a node at array index `i` in a binary heap can access its children by accessing the `2i+1`, `2i+2'th` index in the array, or its parent by accessing the index at the floor of `i-1/2`.

[2] https://en.wikipedia.org/wiki/Tree_traversal#Inorder_traversal

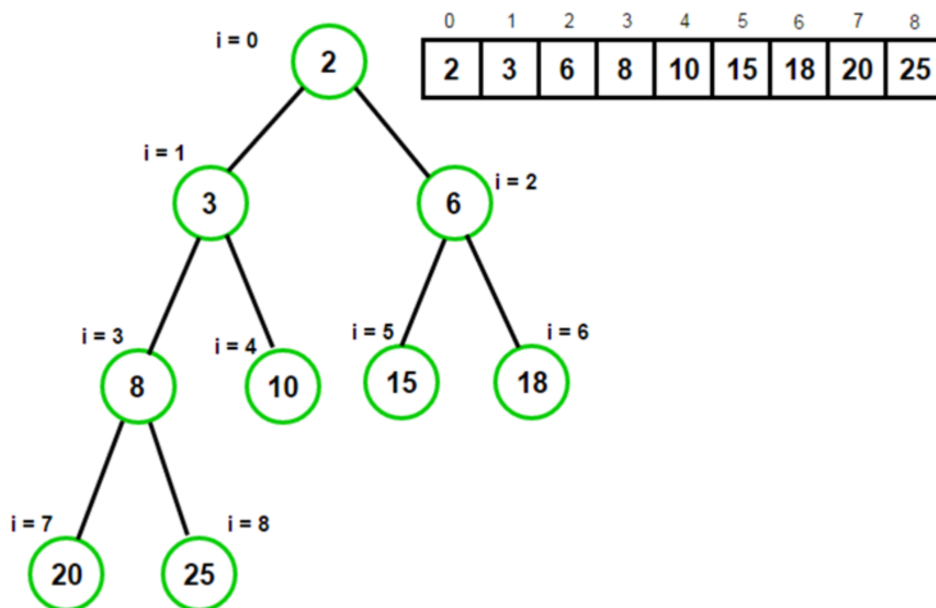


Figure 1: An example of the same binary heap in equivalent tree and array forms. Credit: https://medium.com/techie-delight/heap-practice-problems-and-interview-questions-b678ff3b694c

Variety of Implementations

Within the two implementation families identified in the previous section, there are also many more subtle implementation variations.

AequilibraE currently uses a Fibonacci heap, which is a variation on binomial trees introduced in 1984. The heap is comprised of a forest of heap ordered tree, typically a double linked list. The heap was created around optimising the amortized performance by using lazy heap operations. The techniques used to make these operations work is complex – for those who are interested this video[3] covers it in detail.

To best evaluate alternate implementations, appropriate for path building on transport networks, we implemented several variations from the family of heaps known as k-ary heaps (also referred to as d-ary heaps). Each member of the k-ary heap is an implementation where each node has up to k children. Its simplest is the well-known binary heap (k=2) which we illustrated above.

The k-ary family was chosen because of its simplicity and its consistent high performance in other path building research. In our testing, we implemented the heap for 2, 3, 4 children. Our testing showed that performance degraded above 4 children. These were implemented in Cython and compiled to C++.

In a twist of fate, it turns out our sometimes collaborator Francois Pacull was simultaneously engaged in publishing a similar investigation[4] into the efficacy of different heaps within Dijkstra's also implemented in Cython. Upon discovering this, we added in two of the heap

[3] https://www.youtube.com/watch?v=6JxvKfSV9Ns&t=1s&ab_channel=SithDev

[4] https://aetperf.github.io/2022/12/21/Dijkstra-s-algorithm-in-Cython-part-1-3.html

implementations that Francois had independently written giving us six data structures to test – the original Fibonacci heap, 3-ary heap, and two implementations each of the binary and 4-ary heap (in the following testing Francois's implementation are referred to with the `pq_` prefix).

Theorised Performance

The following table shows the amortised complexity compared to the existing Fibonacci implementation.

	Remove Min	Increase Key	Insert
k-ary heap	`O(log(n))`	`O(log(n))`	`O(log(n))`
Fibonacci Heap	`O(log(n))`	`O(1)`	`O(1)`

Despite the excellent "theoretical performance" in comparison to the k-ary heaps, maintaining the Fibonacci heap requires a large amount of overhead which is amortised in the asymptotic analysis. Previous studies[5] of different heap implementations have shown this overhead significantly impacts performance for practical purposes. In addition, its structural complexity precludes it from using an array-based approach, which negatively impacts the cache efficiency of the data structure since pointer arrays don't tend to high spatial locality.

CPU Cache

CPU memory caching is a key principle to consider when writing high performance programs. The CPU cache is a small and incredibly fast piece of SRAM that sits in front of main memory to speed up memory access to frequent, previously accessed, or predicted memory addresses. It is typically broken down into 3 levels: L1, L2, and L3. Each level increases in size but is also slower to access.

> *"Of all the fancy data structures in the world that you can come up with. The one the hardware likes best is an array."*[6] *- Scott Meyers*

The approximate size and speed of the different caches for an i7-9xx CPU has been included below for reference.

[5] https://epubs.siam.org/doi/epdf/10.1137/1.9781611973198.7

[6] Scott Meyers, https://youtu.be/WDIkqP4JbkE?t=1542

Memory Type	Size (i7-9xx CPU)	Number of cycles to access	Cores per cache
L1 Cache	32kb	4	1
L2 Cache	256kb	11	2
L3 Cache	8mb	39	All
Main memory	-	107	All

CPU caches are further broken down into data and instruction caches.

When a memory address is accessed, not just that address is accessed. Memory is segmented into cache lines (not to be confused with pages), which are 64-byte portions of memory aligned to a boundary. When you read or write to a single byte, the CPU acts on the relevant cache line.

When the cache line is fetched, it is pulled through the various CPU caches. This can prevent having to go back to main memory for a subsequent call since if you are accessing one portion of memory, there's a high probability you will access something close by next. When the cache is full a particular cache line is evicted from that cache level[7].

Knowledge of CPU cache effects can explain some otherwise counter-intuitive performance results where simple data structures and algorithms outperform complicated ones. Fibonacci heaps nodes are often scattered or unsorted in memory and use pointers to store the children, meaning they are unable to take full advantage of cache lines.

Benchmark Methodology

In order to examine the impact of varying heap implementations on skimming performance, we used four models of increasing sizes, and executed the full skimming routine and measured the time taken. The model sizes ranged across typical sizes of real-world AequilibraE's projects. We introduced Australia as the largest model that would conceivably be used regularly to determine whether the Fibonacci heap's theoretical performance would show in the scope of practical use. The Chicago sketch model was provided by the TransportationNetworks[8] repository, which provides a set of example transportation networks.

[7] Cache lines are evicted through complicated cache replacement policies not just first in last out.

[8] https://github.com/bstabler/TransportationNetworks

OUTER LOOP
CONSULTING

	# links	# nodes
Arkansas statewide	273,964	88,446
Chicago sketch	38,733	12,694
Long An	139,966	59,329
Australia	3,074,376	1,236,497

To test the performance of the heaps, we generated a single class distance skim matrix. Skim generation requires that paths are built from all origins to all destinations without consideration of actual demand between the ODs. Thus, it serves as a good benchmarking tool for evaluating the path building performance across the entire network.

Results

Performance across project size

The first benchmark we executed confirmed our suspicions – anything is better than a Fibonacci heap. In the below plot of the minimum runtime for our benchmark (normalised to time per 1000 origins), we see that all trialled data structures have a mostly linear relationship with the network size, however the Fibonacci heap is significantly less performant than all others.

Additionally, the 3,4-ary heaps performed consistently better than the implementations of the binary heap. The 4-ary heap implemented by Francois came out on top overall as the fastest data structure consistently.

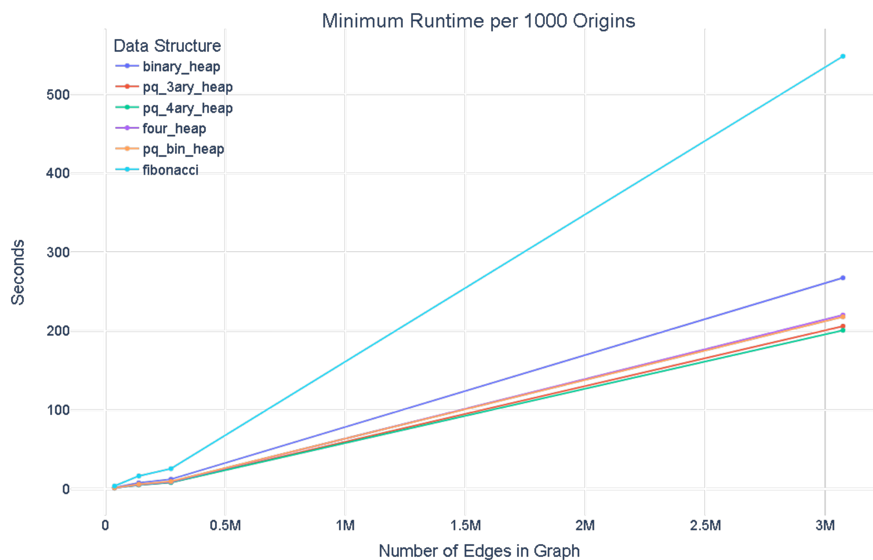


Figure 2: Minimum single threaded runtime normalised to 1000 origins

The k-heap family outperformed the Fibonacci heap by a significant margin across projects of all sizes. This corroborates with the literature, which has previously recorded the binary and 4-ary heap as the most effective heaps in practice.

The Fibonacci heap's performance has been theorised to scale better for larger network sizes, however we were not able to observe this in the scale of networks we were trialling, despite the inclusion of one of the largest traffic networks we have seen in usage anywhere. An explanation for this is that the size of the heap does not grow significantly as the size of the project does. To corroborate this, the maximum size the heap reached in each project was recorded:

Project	Max heap size (open nodes)
Chicago	312
LongAn	490
Arkansas	555
Australia	1283

This represents the "cloud" of reachable nodes currently being considered within the Dijkstra algorithm. This leads to the following key observation:

> *Due to the relative sparsity of transport network graphs, the set of open nodes does not grow fast enough for the asymptotic behaviour of the Fibonacci heap to become significant.*

A common theme we saw in studies where Fibonacci heaps performed well was densely connected graphs, such as social networks, where there were often 10s or 100s of connections per node. Road networks, however, generally have two to four connections per node and do not seem to generate enough complexity for the Fibonacci's theoretical benefits to overcome its lack of cache friendliness.

Further Analysis

At this point, we have identified a clearly superior implementation which has dramatically reduced AequilibraE's runtime for common modelling operations while requiring essentially zero trade-offs to be made. This is a slam dunk by any measure, and we could quite easily finish our analysis here. However, to gain deeper insight into exactly why the k-ary heaps outperform the Fibonacci, two further analyses were undertaken.

- The Cachegrind tool (from the Valgrind suite) was used to empirically measure the number of instructions executed and gain insight into the cache behaviour.
- The two parallelly implemented versions of the best performing 4-ary heap were compared against each other in a head-to-head fashion.

Cachegrind results

To gain some further insight into the improved performance of the 4-ary heap we used Cachegrind, a tool for simulating how a program interacts with a machines cache hierarchy. It gathers information on the number of cache reads and writes and whether the actions used the instruction and data cache or main memory. To make a fair comparison and reduce noise within the results, a simple dummy script was used which performed minimal setup. Valgrind was also invoked directly on the virtual environment's python binary to bypass any unnecessary forks/noise.

Cachegrind simulates two levels of cache, I1 D1, and LL (last-level). While most modern systems use 3 levels of cache the first two are the most impactful for performance.

Term	Meaning
Instruction Cache (I)	Cache for instruction and routines to be executed
Data cache (D)	Cache for previously fetched or predicted data
Reference (ref)	Number of times data or instructions were executed
Cache miss (miss)	A "miss" is where a requested memory address was not present within the desired cache, and it had to be fetched from a lower cache or main memory.

One performance improvement we implemented earlier was to transpose the output matrices during the worker thread setup stage such that the program operates on row slices instead of columns. This meant that each operation had a contiguous slice of memory and resulted in a reduction in D1 miss rate from 3.6% to 2.8% and halved the number of LLd misses. This translated to an approximate 10-15% performance improvement in multithreaded use cases. This provided no single thread performance improvement as a matrix with dimension of size 1 is already stored contiguously in memory.

```
Fibonacci heap, without transpose, quad core

I   refs:      36,498,705,383
I1  misses:        91,039,337
LLi misses:         1,862,276
I1  miss rate:           0.25%
LLi miss rate:           0.01%

D   refs:      15,857,605,173  (8,769,487,809 rd   + 7,088,117,364 wr)
D1  misses:       564,765,898  (  350,933,238 rd   +   213,832,660 wr)
LLd misses:        19,470,537  (   10,731,489 rd   +     8,739,048 wr)
D1  miss rate:            3.6% (          4.0%     +           3.0%  )
LLd miss rate:            0.1% (          0.1%     +           0.1%  )

LL refs:          655,805,235  (  441,972,575 rd   +   213,832,660 wr)
LL misses:         21,332,813  (   12,593,765 rd   +     8,739,048 wr)
LL miss rate:             0.0% (          0.0%     +           0.1%  )
```

```
Fibonacci heap, with transpose, quad core

I   refs:      36,503,276,701
```

```
I1  misses:        91,055,672
LLi misses:         1,328,315
I1  miss rate:           0.25%
LLi miss rate:           0.00%

D   refs:      15,860,038,031  (8,770,987,860 rd   + 7,089,050,171 wr)
D1  misses:       449,789,389  (  295,717,083 rd   +   154,072,306 wr)
LLd misses:        10,706,658  (    7,062,628 rd   +     3,644,030 wr)
D1  miss rate:            2.8% (          3.4%     +           2.2%  )
LLd miss rate:            0.1% (          0.1%     +           0.1%  )

LL refs:          540,845,061  (  386,772,755 rd   +   154,072,306 wr)
LL misses:         12,034,973  (    8,390,943 rd   +     3,644,030 wr)
LL miss rate:             0.0% (          0.0%     +           0.1%  )
```

The swap to a 4-ary heap resulted in reducing the instruction references by approximately 17 million (1.8x), this can be roughly considered as doing half the work and achieving the same result. In addition, the 4-ary heap uses half the data references and has 1.17x less D1 misses. Altogether, we believe the new heap is performing less work and making more efficient use of the data once it is within the cache. We believe the lack of change in the LLd misses is due to the load of data that has never been referenced before.

```
Fibonacci heap, single core

I   refs:      36,508,418,516
I1  misses:        90,781,033
LLi misses:           976,889
I1  miss rate:           0.25%
LLi miss rate:           0.00%

D   refs:      15,861,252,234  (8,773,204,189 rd   + 7,088,048,045 wr)
D1  misses:       443,906,263  (  292,277,189 rd   +   151,629,074 wr)
LLd misses:        10,421,097  (    6,850,826 rd   +     3,570,271 wr)
D1  miss rate:            2.8% (          3.3%     +           2.1%  )
LLd miss rate:            0.1% (          0.1%     +           0.1%  )

LL refs:          534,687,296  (  383,058,222 rd   +   151,629,074 wr)
LL misses:         11,397,986  (    7,827,715 rd   +     3,570,271 wr)
LL miss rate:             0.0% (          0.0%     +           0.1%  )
```

```
4-ary heap, single core

I   refs:      19,635,847,213
I1  misses:        89,308,275
LLi misses:           582,302
I1  miss rate:           0.45%
LLi miss rate:           0.00%

D   refs:       7,911,674,825  (5,176,013,613 rd   + 2,735,661,212 wr)
D1  misses:       379,788,125  (  260,426,398 rd   +   119,361,727 wr)
LLd misses:        10,478,536  (    6,715,570 rd   +     3,762,966 wr)
D1  miss rate:            4.8% (          5.0%     +           4.4%  )
LLd miss rate:            0.1% (          0.1%     +           0.1%  )

LL refs:          469,096,400  (  349,734,673 rd   +   119,361,727 wr)
LL misses:         11,060,838  (    7,297,872 rd   +     3,762,966 wr)
LL miss rate:             0.0% (          0.0%     +           0.1%  )
```

A summary table is shown below of the combined transpose and heap changes.

	Instruction refs	Data ref	I1 misses	LLi misses	D1 misses	LLd misses
Fibonacci heap, without transpose, quad core	3,649,870,538	1,585,760,517	91,039,337	1,862,276	56,4765,898	19,470,537
4-ary heap, single core	1,963,584,721	7,911,674,825	89,308,275	582,302	379,788,125	10,478,536
Ratio (before/after)	1.86	2.00	1.02	3.20	1.49	1.86

Overall, our changes have made a significant impact in reducing the number of instruction and data misses, reducing the amount of work being done and making the work being done more efficient through better use of data caches.

Differences in k-heap implementations

The runtime between the 4-ary heap implemented by Francois and those we implemented varied significantly, which spurred us to examine the key differences in implementation and its subsequent impact on performance.

The greatest difference was the way in which the heap was stored, where Francois implemented two parallel arrays – one which stored the elements of the heap while the other maintained the tree structure through indices which accessed the element array. Our implementation used an array of pointers to the nodes of the tree, where the structure was maintained in the order of pointers.

In addition, the logic used to maintain heap structure differed in implementation, where Francois used while loops instead of tail recursion to restore heap order. The logic used to find the minimum of a given node's child also varied, where our implementation looped to find the number of children and iterated through them to find the smallest element. This was done for the brevity and scalability of the code and to allow for the implementation of any arbitrary k-ary heaps by specifying the number of children. Francois' implementation effectively unrolled this loop to make it as efficient as possible.

The pseudocode related to the logic and heap structure has been included below.

Heap Structure

```
class Heap:
    structure: array(Node*)
    size: int

class Node:
    key: float
    array_index: int # index in the
                     # structure array
    state: char # is this node in the heap
    index: int  # index in the node array
```

```
class Heap:
    structure: array(int)
    nodes: array(Node)
    size: int

class Node:
    key: float
    index: int  # index in the structure array
    state: char # is this node in the heap
```

Smallest Child Logic

```
num_children = min(q.size - 4 * a, 4)
for i in range(1, num_children + 1):
  if min_child.key > q.struct[4*a+i].key:
    min_child = q.struct[4*a+i]
    idx_min = min_child.arr_index
```

```
if c4 < pqueue.size:
  if q.nodes[q.structure[c4]].key < val_min:
    idx_min, val_min = c4, q.nodes[q.structure[c4]].key
  if q.nodes[q.structure[c3]].key < val_min:
     idx_min, val_min = c3, q.nodes[q.structure[c3]].key
  if q.nodes[q.structure[c2]].key < val_min:
     idx_min, val_min = c2, q.nodes[q.structure[c2]].key
  if q.nodes[q.structure[c1]].q < val_min:
     idx_min, val_min = c1, q.nodes[q.structure[c1]].key

elif c3 < q.size:
  ... # unrolled version for three children
elif c2 < q.size:
  ... # unrolled version for two children
elif c1 < q.size:
  ... # unrolled version for one child
```

Following this initial examination, we tested different combinations of these aspects to see what would ultimately yield the most effective heap for Dijkstra's algorithm, down to the level of comparing different data types used. Each test repeated the benchmark for all-to-all skimming - the results have been tabulated below.

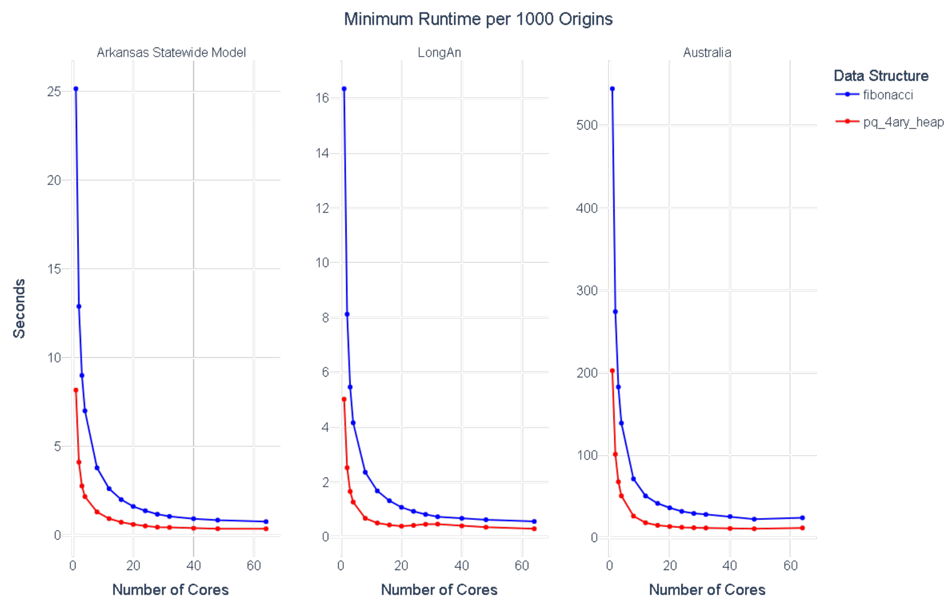
Practice	Measured improvement
Change from "pointer array" to two arrays	+10%
Unrolling hot loop	+8%
Tail recursion vs while loop	nil
Data types	nil

Having tested these factors on skimming performance, we determined the two-array approach had the largest impact on performance. In addition, the Cython compiler is able to optimise tail

recursion out, resulting in no performance difference in how the loop is set up. The types of data were inconsequential in testing, likely due to its relatively inconsequential size difference, for instance integer versus unsigned integer. This exercise showed the counterintuitive conclusion that sometimes the shortest code isn't always the most effective, as was the case with heap structure and child logic. In other words, Francois nailed it.

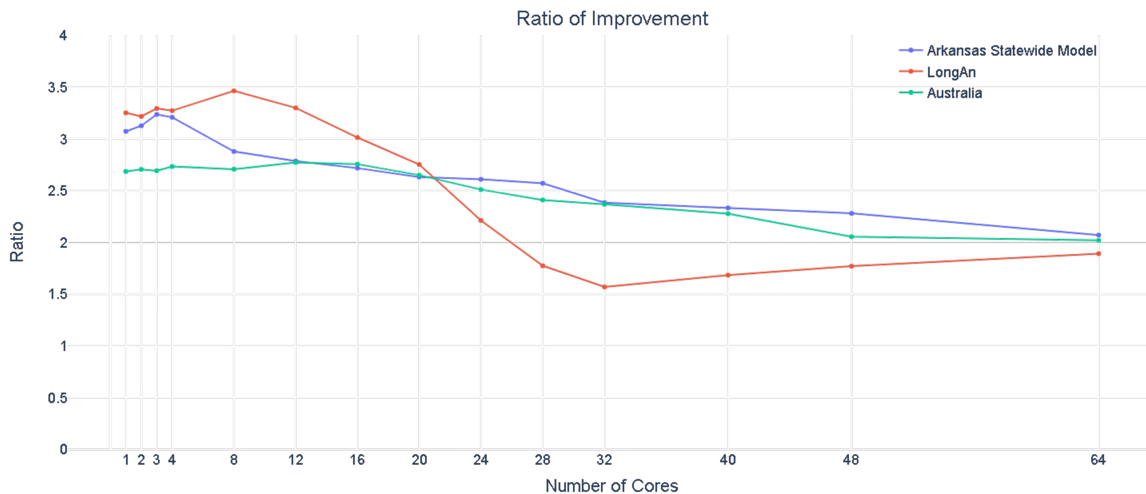
Effects and considerations of multithreading

Having now identified the best heap across project sizes, we also examined how these runtime improvements changed over a range of thread counts to account for different levels of resourcing available to users. Chicago sketch was excluded from this benchmark due to its size.



As the number of cores increase, the behaviour of the two heaps is similar and consistent with Amdahl's law and yields diminishing returns. As the core counts increase, the runtime of each data structure begins to converge, lending credence to "throwing money at the problem" as a means of improving computation time.

Given the total runtime naturally diminishes as the number of threads running increases, we examined the ratio of performance of the 4-ary heap as a factor of the Fibonacci runtime.



Across all core counts, the 4-ary heap results in a significant performance bump. This is most pronounced on the lower end of core counts which are above 2.5x up to 20 threads. The improvement in performance declines as the number of cores increases. We believe this is to do with the increasing proportion of non-parallelisable overhead and single threaded code. Beyond 32 threads, the machine used to benchmark utilises hyperthreading.

Despite spending considerable time investigating we were unable to adequately explain the dramatically different shape of the Long An model compared to the other two models. It is neither the smallest nor largest network and doesn't have significantly different network shape that we could discern.

Conclusion

Our goal during this exercise was to determine whether an alternate implementation of a priority queue could result in an improvement in the performance of the package. The results presented here have been uniformly positive results with little down-side and as such we have replaced the existing Fibonacci implementation within Aequilibrae with the 4-ary heap implementation written by Francois Pacull in this Pull Request. Beyond this, we have also added further weight to observation that the Fibonacci heap is not effective in the context of transportation modelling.