Australasian Transport Research Forum 2023 Proceedings
29 November – 1 December, Perth, Australia
Publication website: http://www.atrf.info

Performance effects of heap structure choice for path finding: A traffic modelling perspective

Jake Moss[1,2], Christopher O'Brien[2], Jamie Cook[1], Pedro Camargo[1, *]

[1] Outer Loop Consulting Ltd Pty
[2] University of Queensland School of Mathematics and Physics
Email for correspondence (presenting author *): pedro@outerloop.io

Abstract

Open-Source software is widely used, yet very little investment is made into their development, particularly in the transportation modelling field. In this environment, AequilibraE has been the sole open-source alternative to traditional commercial modelling software and has garnered a large user base around the world.

As the software gains momentum with new users, developers and funding sources, a wide effort of adding new features and improving the performance of key existing features is underway. In this paper we present recent improvements that have reduced the traffic assignment and network skimming processing times by a factor of up to 3.4x, while expanding on the existing research into the performance of different data structures to support the traditional Dijkstra algorithm in the context of real-world road transportation networks.

1. Introduction

AequilibraE (Camargo 2022) is a general-purpose transport modelling program designed to be an alternative to popular (and expensive) commercial software, and it is possibly the most successful in finding adoption around the world, as evidenced by its adoption by the World Bank in projects in Vietnam and Armenia, by engineering groups in Indonesia, Sri Lanka, Malta and Brazil, as well as research groups such as the Volpe institute in the USA, as well as have been selected for funding by EGIS France, one of the largest engineering conglomerates in the world.

Having identified that AequilibraE's path finding algorithm lagged behind dedicated graph-manipulation libraries available for Python (Cook 2022) by a factor of 2x, revisiting the software's core path-finding algorithms became a clear opportunity for a substantial improvement to the software.

This type of research, however, leaves the realm of transport modelling into very technical computer science aspects with considerations of both software and hardware architectures.

1.1. 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 set of shortest paths from an origin to all destinations in the network. We profiled the entire runtime of AequilibraE's skimming functionality, which included our implementation of Dijkstra's algorithm.

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%[1] of the total runtime. It was definitive that this was where our optimisations efforts should be focused.

2. Data structures and why they matter

At its most basic level, the network traffic assignment requires repeated computation of shortest path trees on the target graph. This is commonly achieved by using Dijkstra's algorithm to build paths from each centroid (origin) to every other centroid (destinations) 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.

2.1. 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 used in the implementation of Dijkstra's algorithm.

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, there are good online resources for the visualisation of these actions for different types of heaps, such as https://www.cs.usfca.edu/~galles/visualization/Heap.html.

2.2. Heap implementation

[1] *remove_min was responsible for 87% of the runtime when profiling, as profiling instrumentation can influence the performance characteristics of a program this figure may not representative of the true runtime proportion*

There are two basic implementations of heaps which work well for different problems: tree-based and array-based implementations [2]. Tree-based heaps use pointers to store the connections between each node, whereas an array-based heap maintains the structure through an ordered array.

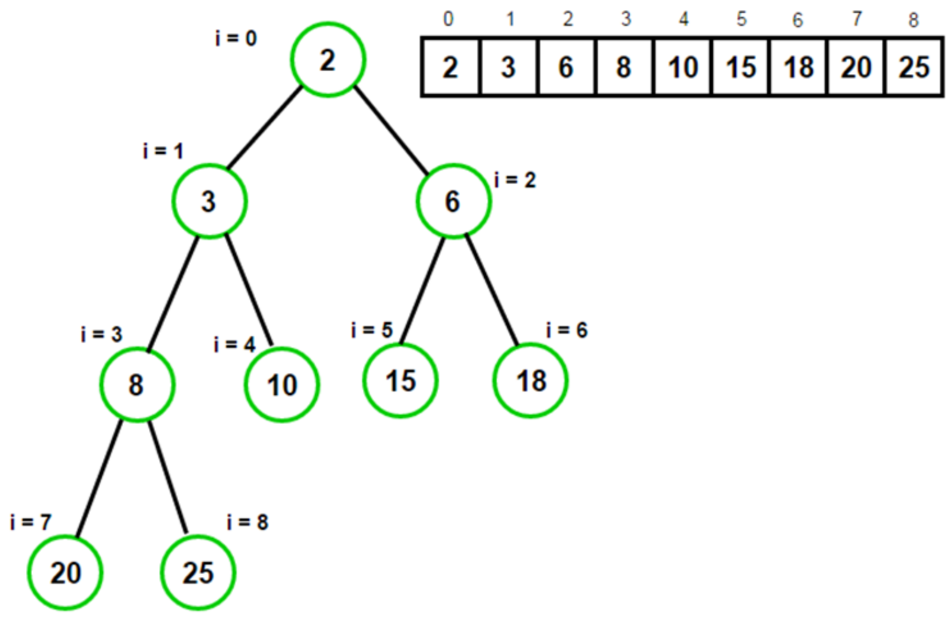
2.2.1. Tree-based heaps

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.

2.2.2. Array-based heaps

Array-based heaps make use of the heap rules to store the tree in an in-order ordered array. The requirement that a heap is to be 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`.

Figure 1: An example of the same binary heap in equivalent tree and array forms.



Source: https://medium.com/techie-delight/heap-practice-problems-and-interview-questions-b678ff3b694c

2.3. Alternative 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 trees, 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 are complex and outside the

[2] this differs from using an array naïvely to implement a priority queue which can be done as a sorted array

scope of this paper, but accessible documentation is plentiful online (*Fibonacci Heaps or 'How to invent an extremely clever data structure'* 2022).

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 is illustrated on Figure 1.

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 and 4 children, as our research revealed that performance degraded above 4 children. These were implemented in Cython (Cython 2022) and compiled to C++.

In a twist of fate, it turns out that a research collaborator was simultaneously engaged in publishing a similar investigation (Pacull 2022) into the efficacy of different heaps within Dijkstra's also implemented in Cython. Upon discovering this, we added in two of the heap implementations independently developed in this parallel research effort, 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 the implementations obtained from this parallel research by François Pacull (FP) are referred to with the `pq_` prefix).

2.3.1. Theorised performance

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

Table 1: Algorithm theoretical complexity per operation

	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 (Larkin et al. 2014) 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 have high spatial locality.

2.4. 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.

The approximate size and speed of the different caches for an i7-9xx CPU has been included on Table 2 for reference.

Table 2: Test CPU configuration

Memory Type	Size (i7-9xx CPU)	Number of cycles to access	Cores per cache
L1 Cache	32KiB	4	1
L2 Cache	256KiB	11	2
L3 Cache	8MiB	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[3].

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.

2.5 Benchmarking methodology

In order to examine the impact of varying heap implementations on skimming performance, we used four models of different 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, while the Chicago sketch model (Stabler 2022) is the only standard testing instance used.

Table 3: Test model characteristics

Model region/name	# 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.

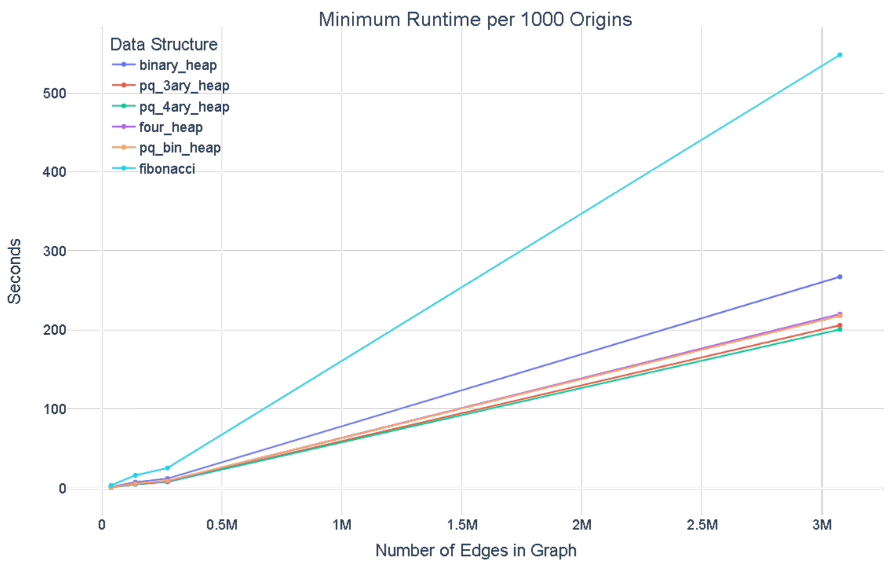
[3] Cache lines are evicted through complicated cache replacement policies not just first in last out.

3. Results

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 pq_*4-ary* heap implementation came out on top overall as the fastest data structure across all models tested.

Figure 2: Minimum single threaded runtime normalised to 1000 origins



These results corroborate others found in literature, which have 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:

Table 4: Maximum heap size in memory

Project	Max heap size (open nodes)
Chicago	312
LongAn	490
Arkansas	555
Australia	1,283

This represents the "cloud" of reachable nodes currently being considered within the Dijkstra algorithm, and 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.

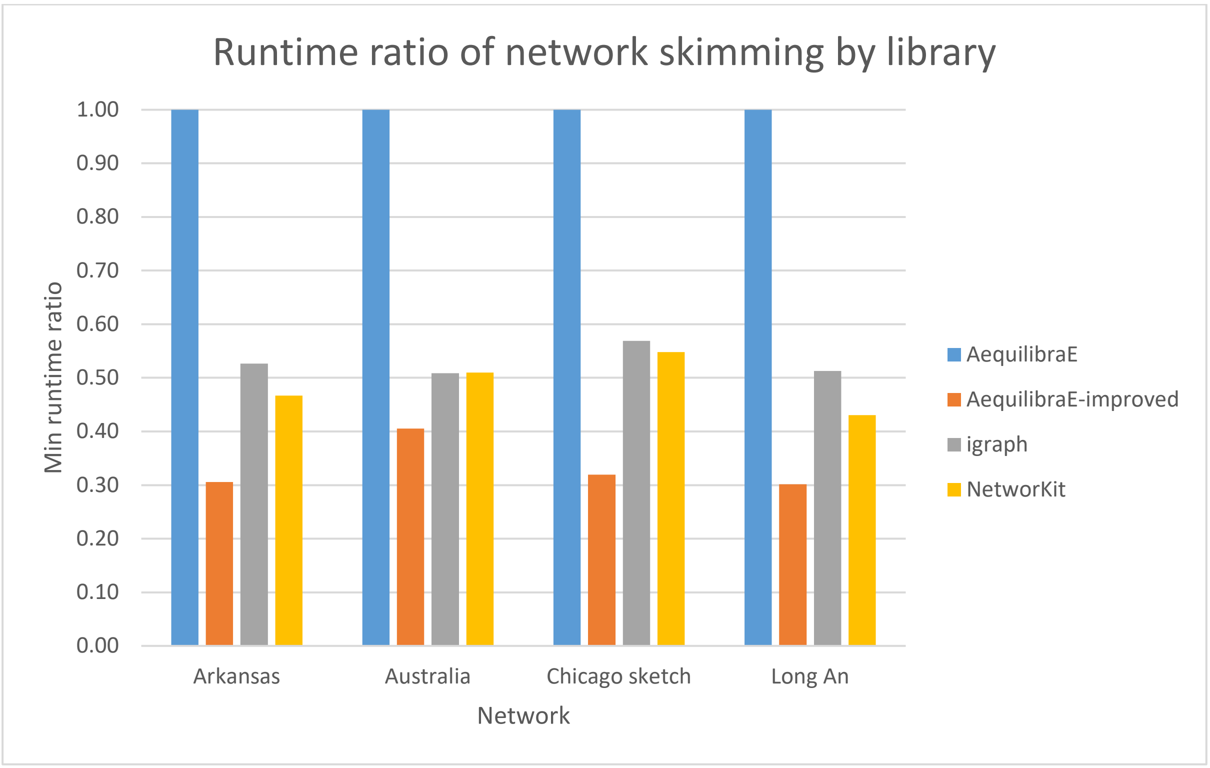
Road networks, however, generally have two to four connections per node and do not seem to generate enough complexity for the Fibonacci's theoretical benefits to overcome its lack of cache friendliness, which leads to the following key observation:

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.

3.1. Benchmarking

As (Cook 2022) had provided the motivation for this research by establishing AequilibraE's lacking performance when compared to general path computation libraries in the Python ecosystem, it was only fitting to compare the performance of AequilibraE's new path-computation algorithm to the best competitors identified on (Cook 2022), which we present on Figure 3. We have forgone the comparison with other modelling packages as they do not allow the publication of benchmarking results when using trial licenses, and acquiring software licenses was outside the scope of this project.

Figure 3: Benchmarking general path computation Python libraries



AequilibraE's superiority is obvious across all models, as the best alternative tested takes at least 25% longer to computer a set of shortest paths, and over 50% longer in the case if the Arkansas State model.

4. Further analysis

At this point, we had 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 substantial result and the main objective of the project, However, we were interest in gaining deeper insights into exactly why the *k-ary* heaps outperformed the Fibonacci, and so 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 alternative implementations of the best performing *4-ary* heap were compared against each other in a head-to-head fashion.

4.1. 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.

Table 5: Cachegrind terminology

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.

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 loading of data that has never been referenced before.

Figure 4: Cachegrind results – Fibonacci Heap

```
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%  )
```

Figure 5: Cachegrind results – *4-ary* Heap

```
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 of the combined transpose and heap changes is shown on Table 6.

Table 6: Cachegrind results

	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.

4.2. Differences in k-heap implementations

The runtime between the two implementations of the *4-ary* heap (pq_ and the one implemented during our own research) varied significantly, which motivated a close examination the key differences in implementation and its subsequent impact on performance.

The greatest difference was the way in which the heap was stored, where the pq_ algorithms 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 own 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 pq_ algorithms 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 internal 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, while the pq_ algorithms were implemented by effectively unrolling this loop to make it as efficient as possible, as shown on Table 7.

Figure 6: Heap implementation pseudo-code

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.

Table 7: Source of performance different on k-Ary heap by algorithm component

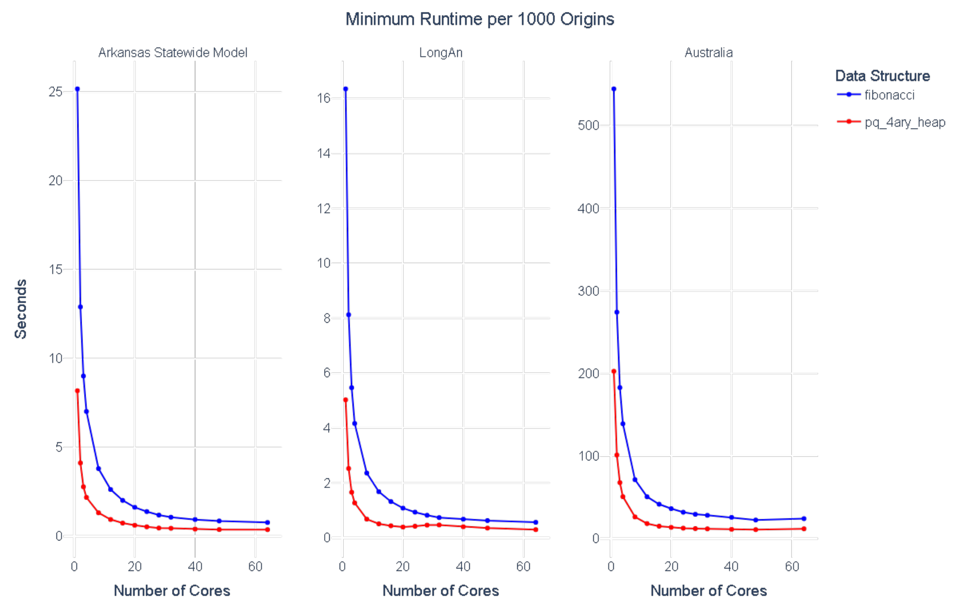
Practice	Measured improvement
Change from "pointer array" to two arrays	+10%
Unrolling hot loop	+8%
Tail recursion vs while loop	nil
Data types	nil

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 that sometimes the shortest code isn't always the most effective, as was the case with heap structure and child logic.

4.3. 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.

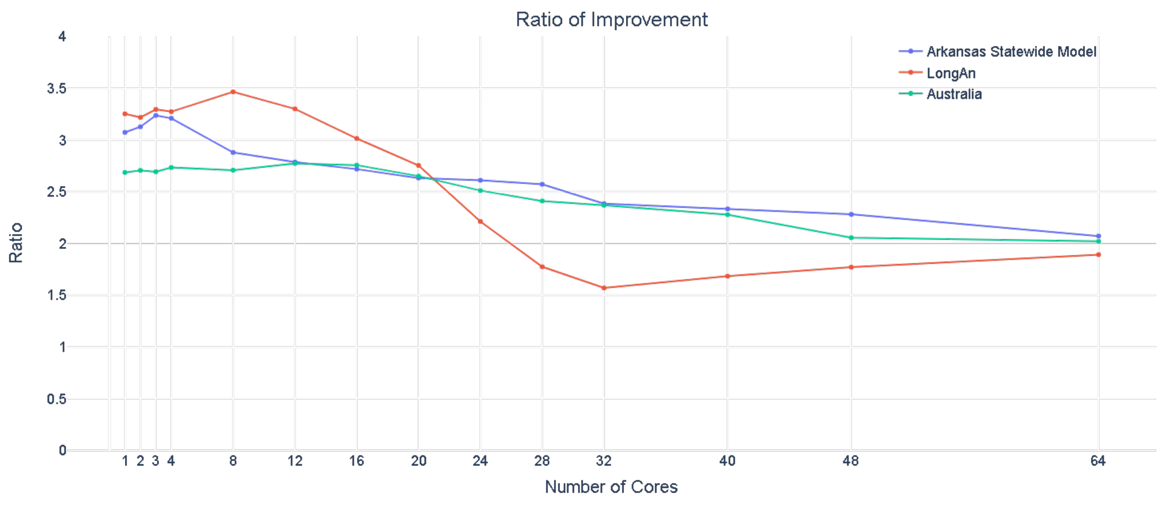
Figure 7: Effect of multi-threading on path computation time differences



As the number of cores increase, the behaviour of the two heaps is similar and consistent with Amdahl's law (Herlihy et al. 2020) 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.

Figure 8: Ratio of path computation times for different levels of multi-threading



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 which may impact the performance characteristics and was not explored.

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 topology that we could discern.

5. 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 AequilibraE. The results presented here have been uniformly positive results with no appreciable loss and as such we have replaced the existing Fibonacci implementation within Aequilibrae with a slightly modified version of the *4-ary* heap implementation provided in (Pacull 2022). This work provides improvements to the performance of with no cost or change to the user as well as provides insight and examples of performance analysis for those writing custom software with performance in mind. Beyond this, we have also added further weight to observation that the Fibonacci heap is not effective in the context of transportation modelling and have established AequilibraE as the best general path-computation library in the Python eco-system.

6. References

Camargo PV de (2022) ‘AequilibraE - An Open-Source transportation modeling package’, http://aequilibrae.com, accessed 18 July 2022.

Cook J (2022) ‘Faster Path Building for Transport Networks’, https://www.outerloop.io/blog/20221213_faster_path_building/, accessed 11 April 2023.

Cython D (2022) ‘Cython’, https://cython.org/.

Fibonacci Heaps or ‘How to invent an extremely clever data structure’ (2022), https://youtu.be/6JxvKfSV9Ns.

Herlihy M, Shavit N, Luchangco V and Spear M (2020) ‘The Art of Multiprocessor Programming’, in M Herlihy, N Shavit, V Luchangco, and M Spear (eds) *The Art of Multiprocessor Programming (Second Edition)*, Morgan Kaufmann, Boston, doi:https://doi.org/10.1016/B978-0-12-415950-1.00009-4.

Larkin DH, Sen S and Tarjan RE (2014) ‘A Back-to-Basics Empirical Study of Priority Queues’, in *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, *16th Workshop on Algorithm Engineering and Experiments*, https://epubs.siam.org/doi/epdf/10.1137/1.9781611973198.7.

Pacull F (2022) ‘Dijkstra’s algorithm in Cython’.

Stabler B (2022) ‘Transportation Networks’, https://github.com/bstabler/TransportationNetworks.