## Choosing data structure for scheduling

Having a scalable and efficient simulation environment is very dependent on the data structure we use to maintain the events and the agents for scheduling. MUSE has a two tier scheduling system. The very top tier is the scheduler and it maintains the agents and knows which agent to process at any given time. The second tier is in the agent. All incoming events to a given agent must be stored and correctly scheduled in increasing fashion according to the time of the delivery. The heap data structure seemed a great fit for both tiers. The heap data structures under consideration are the Fibonacci heap (Fredman and Tarjan) and the Binary Heap. Binary heaps are heaps that are implemented with binary trees (Wikipedia). Fibonacci heaps have very impressive runtime results, however these results are amortized. The following table shows the runtimes of both binary and Fibonacci heaps.

|  |  |  |
| --- | --- | --- |
| Standard Operations | Fibonacci Heap | Binary Heap |
| Insert | O(1) | O(log\*n) |
| Get Min | O(1) | O(1) |
| Delete Min | O(log\*n) <amortized> | O(log\*n) |
| Decrease Key | O(1) <amortized> | O(log\*n) |
| Delete | O(log\*n) <amortized> | O(n) |
| Merge | O(1) | O(m log(n+m)) |

Fibonacci heap showed impress runtimes, but we wanted to know just how much we have to amortize before we realize the gains. Binary heap on the other hand has good runtimes and no amortized costs. The two tiers make more use of different operations. Hence, there is a good chance that we would end up using a combination of the two heaps in MUSE. The first task we have done is identifying which operations were frequent in each tier. The first tier, once we add the agents we should never remove until the end of simulation. Therefore the only operation we want to compare is the *decrease key* operation. Decreasing the key in short is just an operation to reorder an element in the heap. We can draw an early conclusion here and say that Fibonacci heap should be used, but it is better to let the numbers speak. In the second tier, we frequently made use of the *insert, get min, and delete min*, whenever there was a rollback we also used the *delete* operation.

For binary heap implementation we will be using the *priority\_queue* from the C++ STL containers. Fibonacci heap we have found a nice C++ implementation. To get the source for the fibonacci heap implementation follow this reference (Kühl).

### 1.1.1 Fibonacci vs. Binary testing procedure

We have to find a good heap for both tiers and we already discussed the heavily used operations for both tiers. With the first tier we want to test the key decreasing. To get a good idea we have a couple of controlled variables. We have fixed the time steps to 1000. The basic idea is to keep increasing the number of agents, starting from 100 and ending at 1,000,000 agents. Then for each time step we will iterate over the number of agents and randomly (P = .5) increase or decrease the value of the agent’s key and call the *decrease* operation on the key. We will keep track of the time it took to execute and take the average of five runs for each increase in the number of agents. Fibonacci heap implementation we have has a *change(element, key)* method which we can use. However, the priority queue does not implement a way to change the key, so the solution is to pop the top element and then update the value and push it back into the heap. This makes the runtime from *O(log\*n)* to *O(log\*n+log\*n)* we simply added the runtimes for delete min and insert to get the updated runtime.

The second tier deals with events. To actually see something meaningful we fix our time steps to 10,000 iterations. We will slowly increase the number of events in the heap starting from 100 events per time step all the way to 1,000,000 events per time steps. There are to case to test in the second tier. First case is just going to be a test to see how long it takes to insert *X* number of events and then delete min until the heap is empty again. The second case is testing how long it takes to delete elements from the heap until it is empty. We will use the Iterators and just keep calling the *delete* operation and see which has the best time. Like the first tier we will run each five times and get the average time. The big deciding factor for the second tier will be the first case, as this is the most frequent operations. However, we will include the second case when we calculate the execution time.

### 1.1.2 Fibonacci vs. Binary data collection, results, and discussion

The table below is the collection data when we compared the two heaps for tier one. Keep in mind that that the execution times are the average of five runs.

|  |  |  |  |
| --- | --- | --- | --- |
| Agents | Time steps | Fibonacci execution time | Binary execution time |
| 100 | 1000 |  |  |
| 1000 | 1000 |  |  |
| 10000 | 1000 |  |  |
| 100000 | 1000 |  |  |
| 1000000 | 1000 |  |  |

The table below is the collection data when we compared the two heaps for tier two. Here is the execution times of case one and case two combined as discussed earlier.

|  |  |  |  |
| --- | --- | --- | --- |
| Events | Time steps | Fibonacci execution time | Binary execution time |
| 100 | 10000 |  |  |
| 1000 | 10000 |  |  |
| 10000 | 10000 |  |  |
| 100000 | 10000 |  |  |
| 1000000 | 10000 |  |  |