Interval partitioning report

Sigurt Dinesen, Gustav Røder & Lars Yndal Sørensen

September 2, 2015

Results

The implementation can parse input files, and produce the expected output compared to the existing output files.

When the program isn't given any starting arguments, it will run all input files, and compare the resulting resource depth with that of the example outputs. The result is as follows:

```
ip-1.in is ok? true
ip-2.in is ok? true
ip-3.in is ok? true
ip-rand-100k.in is ok? true
ip-rand-10k.in is ok? true
ip-rand-1M.in is ok? true
ip-rand-1k.in is ok? true
```

Implementation details

The implementation is based on the pseudo code from page 166 in Kleinberg and Tardos, *Algorithms Design*, Addison–Wesley 2005.The content of the input file is parsed into Job objects, which are ordered by increasing start-time by use of the standard Java `java.util.PriorityQueue` Jobs are then assigned to the first available resource. This ordering is also maintained by use of the Java priority queue.

let n denote the number of jobs. The initial sorting of the jobs takes $O(n\log(n))$. We then traverse the jobs in sorted order, taking an additional $O(n\log(n))$ time, as we do one dequeue operation for each job, each operation taking $O(\log(n))$ time. For each job, we dequeue a resource from the resource queue, and add it again, taking $2 * O(\log(n))$ time for each n. Thus, the total running time comes to $4n\log(n) \in O(n\log(n))$

Notably, the jobs do not need to be ordered in a priority queue, as we do not make insertions after the first ordering. Instead, a sorted list could have been used. A list would save us $O(n\log(n))$ operations, but would not change the running time in asymptotic (big-O) notation.