Distinction assignment lab 3

Jens Madsen, Victor Jarlow
Group 23

To be able to benchmark the code that we produced, we performed some different tweaks to the program:

Parent continue/wait:
This tweak describes how the parent should behave after forking its children. If for example the parent has two possible paths it can continue along, it will in the case when the parent continues only fork 1 child along one of the paths, and continue itself along the other. In the case of the ParentWait scenario, the parent will fork 2 children, one along each path and then wait from them to join.

ForkAfter steps/neighbors discovered:
We used two different methods to decide when to fork, either using the amount of steps that a thread as travelled, or the amount of neighbors it has discovered.

Thread pool count:
The thread pool count describes how many threads the threadpool used to run the program has at its disposal. We used two different levels of thread pool counts: 2 and 20.
In our tests unless pool size is explicitly stated, we used the commonPool method to decide the size of the pool. All tests were carried out on a processor with 4 physical cores and 8 logical cores of the brand and make “Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz”.

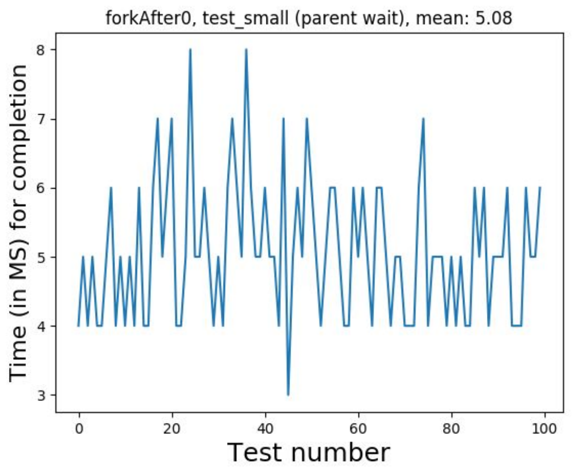
The tweaks were tested on 3 separate maps, small map, medium map and large map. The large map was created by us and is an extended version of the medium map. The small and medium maps are the same maps that were supplied to us in the skeleton-code bundle.
The map-sizes in nodes are as follows:

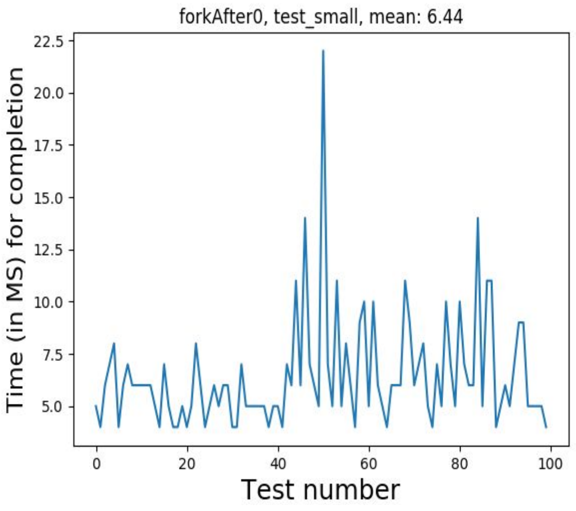
Small:	9X9
Medium:	31X11
Large:	31X21
Very Large:	119X572

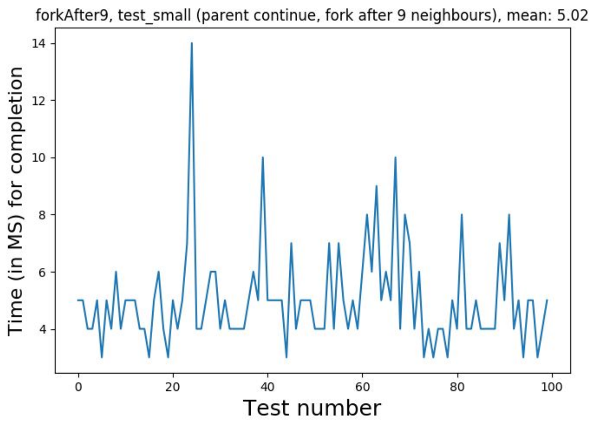
Results

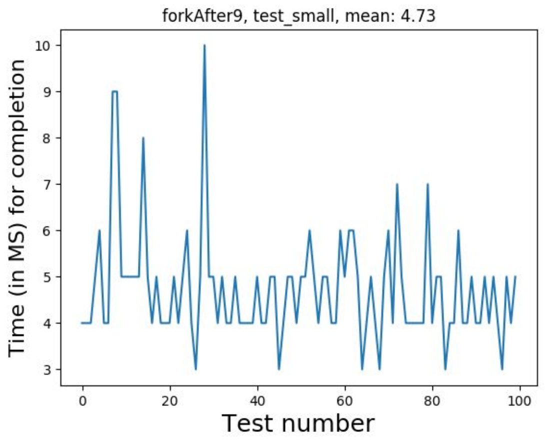
Initially we used 3 different map sizes: small, medium and large. We noticed that the maps were too small to see any significant differences between the tweaks that we applied in each case. Therefore we decided to make a significantly larger map (very large), which we hoped would highlight any differences and make them more visible.
Since the differences between tweaks in the map sizes small, medium and large were so small we decided that the results of those benchmarks were inconclusive, but we still decided to enclose them in this report.
The results of the very large map shows that the means of the different tests seem to cluster in distinct regions, with 0 step/neighbor and waiting for parent being the absolute highest, and forking after 1000000 steps (effectively never forking) is the lowest. The rest of the results end up in either the 2500-2200 ms cluster or the 1600-1400 ms cluster (see figure 2).
Figure 1 also shows that there is sometimes very high variance for each of the runs when forking after 0 steps/neighbors and waiting for parent (some runs take 30000 ms, while some take around 2000 ms).

Small-sized Map

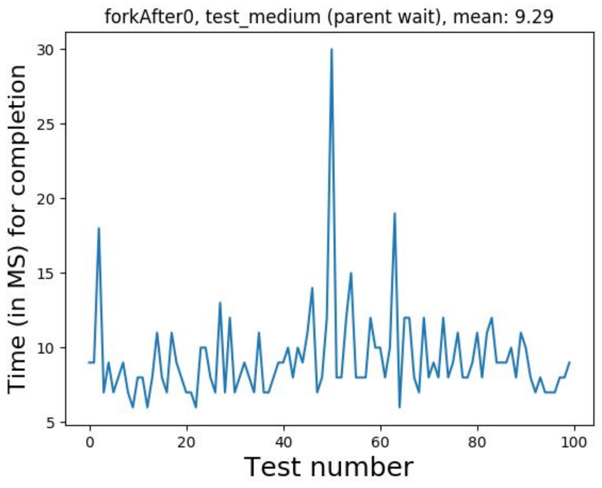


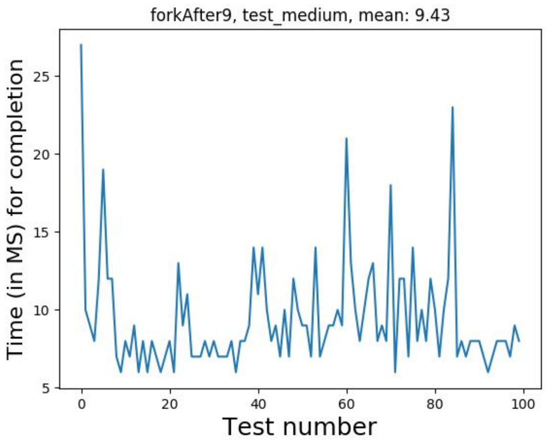


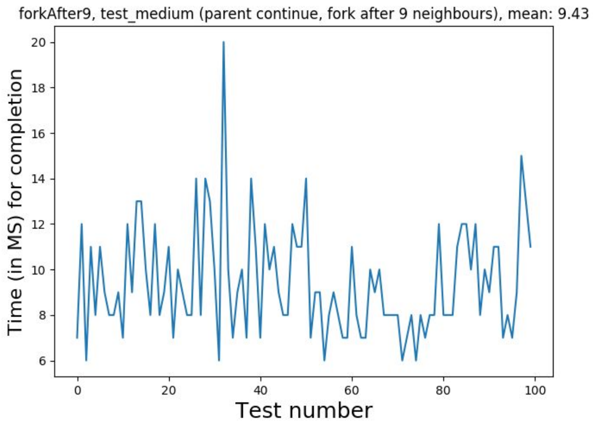


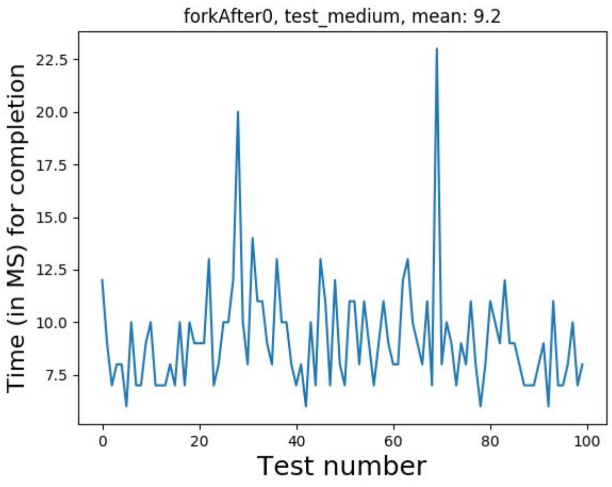


Medium-sized Map

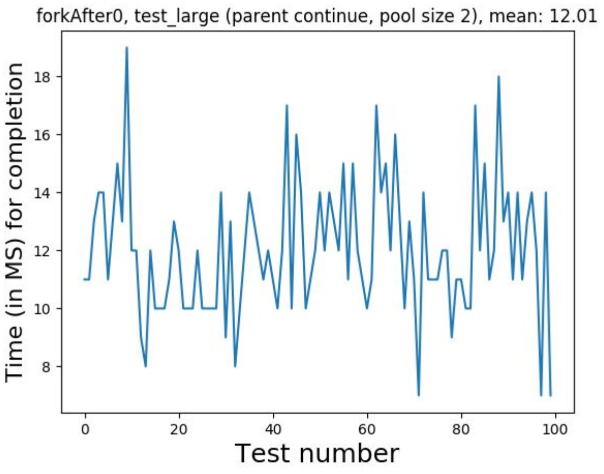


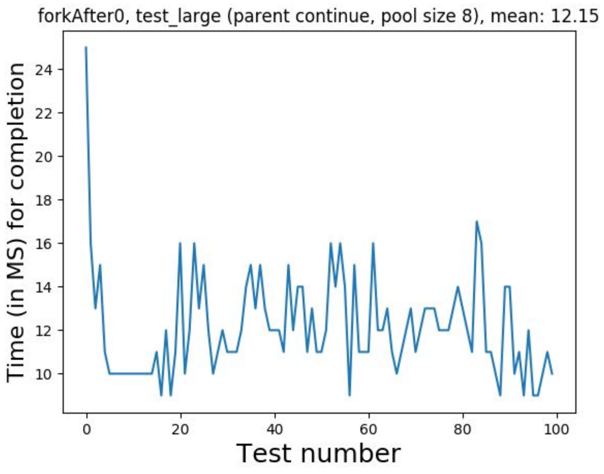


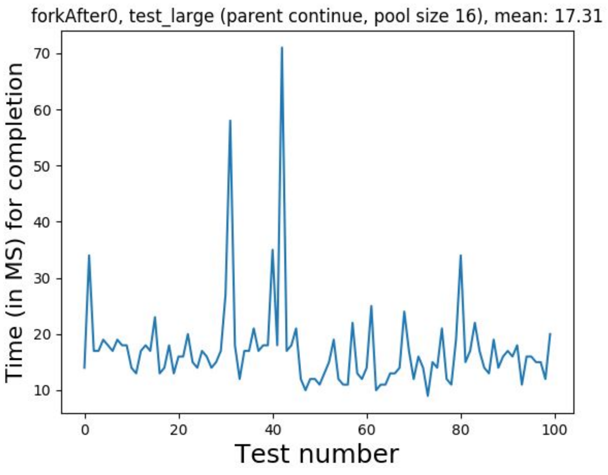


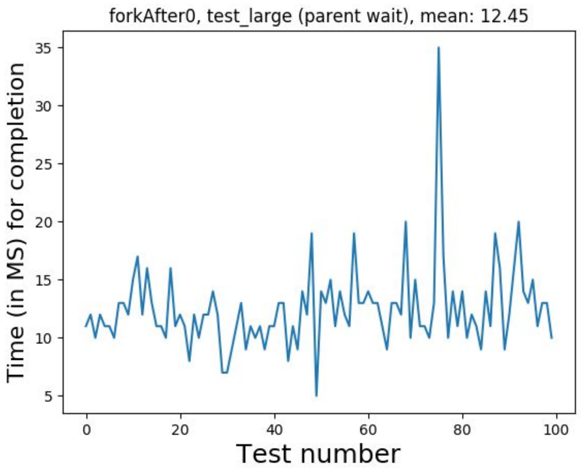


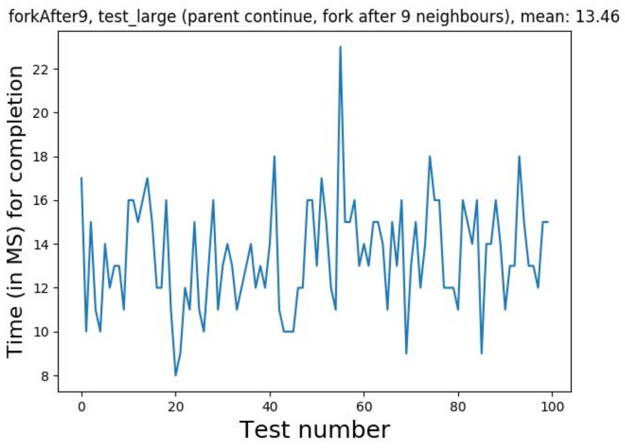
Large-sized Map

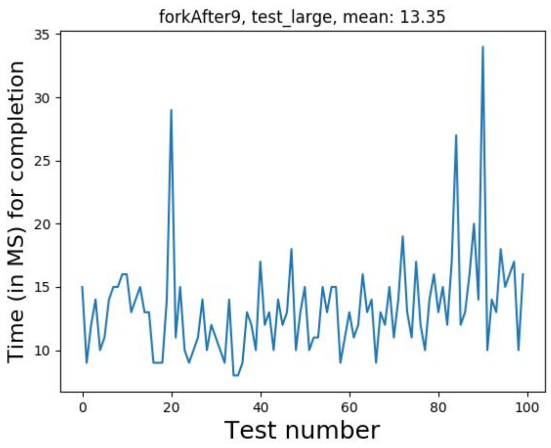


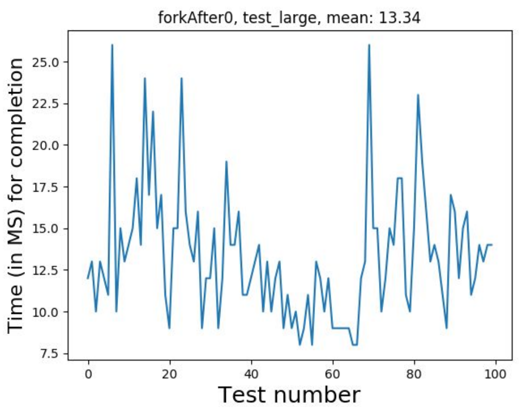












Very Large-sized Map

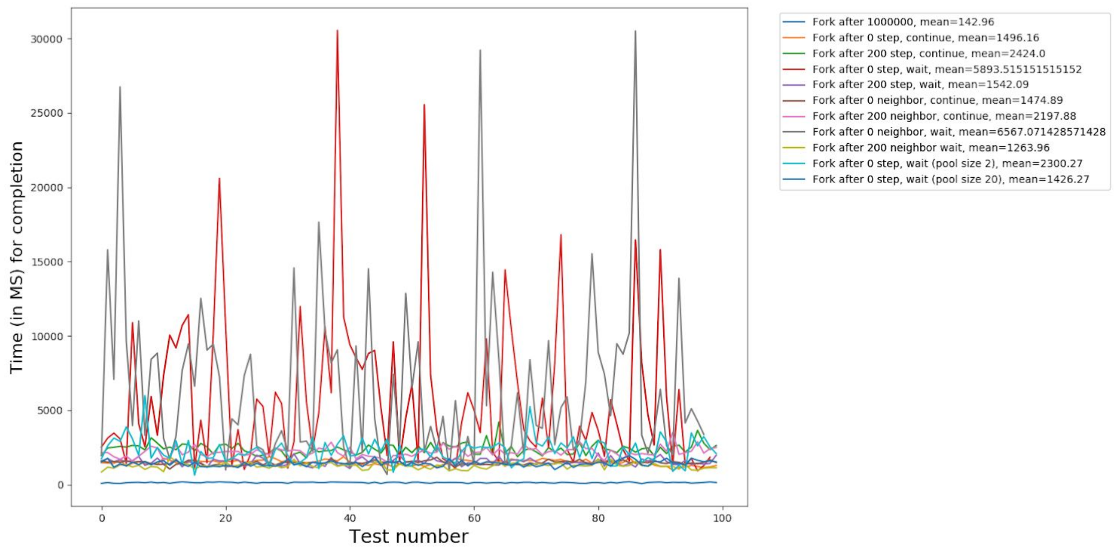


Figure 1. The total completion times for all tests on the very large map.

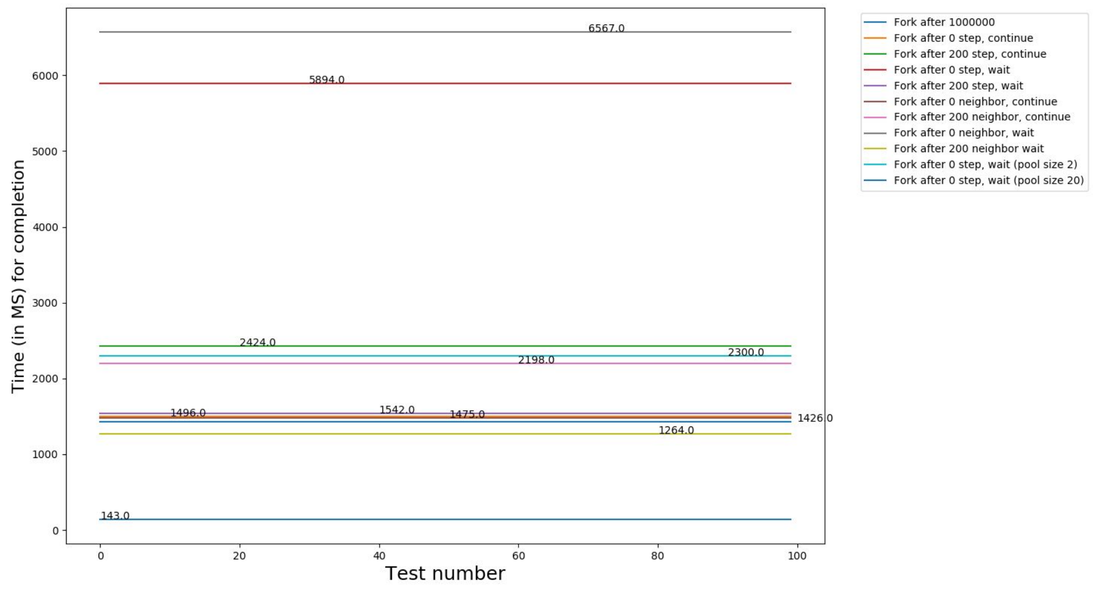


Figure 2. The mean completion time for the tests on the very large map.

	Continue (milliseconds)	Wait (milliseconds)
Fork after 1000000 step (never forks)	143	143
Fork after 0 step	1496	5893
Fork after 0 neighbor	1474	6567
Fork after 200 step	2424	1542
Fork after 200 neighbor	2198	1264
Fork after 200 pool size 2	N/A	2300
Fork after 200 pool size 20	N/A	1426

Table 1. Benchmarking times using the very large map.

Conclusions

In this chapter we will discuss what conclusions can be drawn from the benchmarking test. As previously mentioned we will be focusing on the benchmarking tests made using the very large map, since the results obtained using the other maps did not show enough variance between tests to rule out randomness.

When we made the map large (verylarge.map) we found that forkAfter a high number of iterations was much more efficient than forking after a few iterations. We think this is because the map is large and wide (and we put the goal the furthest away) which leads to a depth-first search being more effective than a breadth-first search. Using this map and goal position is the worst case for a breadth-first search. Generating new threads for each of the elements in the frontier mimics a breadth-first search which leads to forking new threads all the time and is less efficient than not forking at all (which mimics a depth-first search) for this type of map.

Limiting the pool size to be only 2 proved to give a substantial loss in performance compared to having a pool size of 20. Because of time limitations we decided not to compare all combinations of forking method (step/neighbor), pool size (2,20) and forkAfter (0,200), therefore it is hard to draw any conclusions from the data that we collected with regards to how pool size is a factor of performance in relation to the other tweaks.

We noticed that if the threads are set to fork after 0 steps/neighbors and the tweak where the parent has to wait for the children is used, the time it takes to find the goal increases dramatically.
The same is true for the inverse, when threads are set to fork after 200 steps/neighbors and the tweak that parents continue after forking children.

We don't understand why this phenomenon occurs, but perhaps we need to conduct more tests to be able to come to a conclusion.