A-mazed

Karl-Rehan Chiu Falck

March 2019

1 ForkJoinSolvers

We have 3 versions of ForkJoinSolver, we give the details about what they do and in what regards they differ here.

1.1 ForkJoinSolver

First version, uses the forkAfter parameter to spawn children at each unvisited neighbour after forkAfter steps, the parent process then waits for the children to return.

1.2 ForkJoinSolver2

Similar to above but the parent also works at the same time. If after forkAfter steps the parent is at a split in the road it will fork child processes to all roads but one and work on that one itself.

1.3 ForkJoinSolver3

The process forks at each intersection and ignores the value of forkAfter. Parent also works.

2 Tests

We ran each solution on the maps small, medium and Great_Chasm. Here ForkJoinSolver and ForkJoinSolver2 are run with a workerpool of 5 thread and with the values of forkAfter set to 1, 2, 3, 4. ForkJoinSolver3, however, does not depend on forkAfter so it is instead run with a workerpool of 1, 2, 3, 4 threads.

Each solver is run 5 times for each value and each map. The final value is the avrage of those runs.

The Great_Chasm map is an empty 25×25 map.

3 Results

For the two versions using the parameter forkAfter ForkJoinSolver seems to be faster at the small and medium maps for some reason, (see figures 1 and 2). It may be due to the possibility that spawning a thread and placing a player is somewhat faster than it is for the player to move a tile and so spawning a new player to the next tile is faster than moving it there. Of course since we have only run 5 tests on each forkAfter value and used the average from those results in our graphs it could simply be due to the randomness. In our huge custom map Great_Chasm (see 4) it does seem to be that for greater values on forkAfter ForkJoinSolver loses the advantage over ForkJoinSolver2, but here the difference is so small, it could be simply be due to randomness again.

Since the worker pool has a maximum size (set to 5 for our tests for solver 1 and 2) on the map it seems to be that once enough threads are running spawning time is about the same as moving the player a tile or possibly more. It may be due to multiple threads having to spawn so the time required to spawn stacks (technically ForkJoinSolver just reassigns a worker to new work and puts the "parent work" on pause). Also, once we hit the maximum number of workers Solver 2 won't even bother with spawning anymore and save some time that way. Of course, once again it may simply be the randomness but the reality is, spawning takes time and unless what consumes most time is waiting for a player to move a tile then spawning more players the way the original solver does should take more time.

With that said, ForkJoinSolver3 is the clear winner regardless almost always except in the special case of having only one worker in the pool at the bigger map Great_Chasm. This is most likely due to the fact that it only tries and will always fork at roadsplits thus minimising wasted time and unlike the other solvers will not miss opportunities to fork which results in it having a far higher chance of sending a player down the "right" path. Also, in the case of only one worker it simply ignores forks in the road and works similarly to the sequentialSolver.

ForkJoinSolver3 clearly solves the small and medium maps faster on average than the other solvers regardless of amount of workers and it seems that on average it tends to get slower the more workers it has in the pool for small maps which makes it somewhat suggesting that the slowing down may be due to the spawning time, again, might be the randomness.

Finally, it does however as mentioned above solve the Great_Chasm map slower than the other solvers if it only has one worker but it almost linearly improves the time taken for each worker it has in the pool. This is most likely due to the fact that due to the sheer size of the map and in order to search the whole map, multiple threads working does indeed help out and ForkJoinSolver3 will on this map immediately as soon as possible use the maximum amount of available workers to use (well, at least 4 as in our test) to search it. Of course, it has a maximum number of threads that can be put to work as each thread can only at most spawn 3 new threads and the map is finite so it'll hit a maximum amount of threads that can run on the map. Also as a test, we tried running the solver witha 100 workers in the pool (with the -l flag) and the time to search

the map in this case were 1874 ms which is clearly far much slower than even running 1 worker in the pool. Reason is not randomness this time but spawning time and time on the processor for each thread (there is only 4 processor cores on the computer).

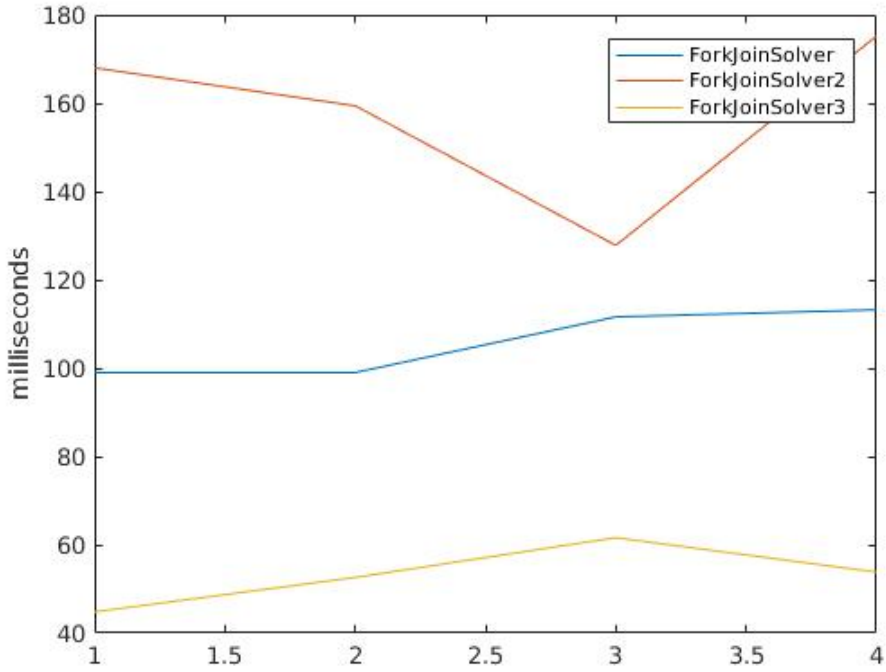

Figure 1: ForkJoinSolver, ForkJoinSolver2 and ForkJoinSolver 3 on the small map. Note that the x-axis for Solver 1 and 2 is the value of forkAfter while for Solver 3 it is the number of workers in the pool.

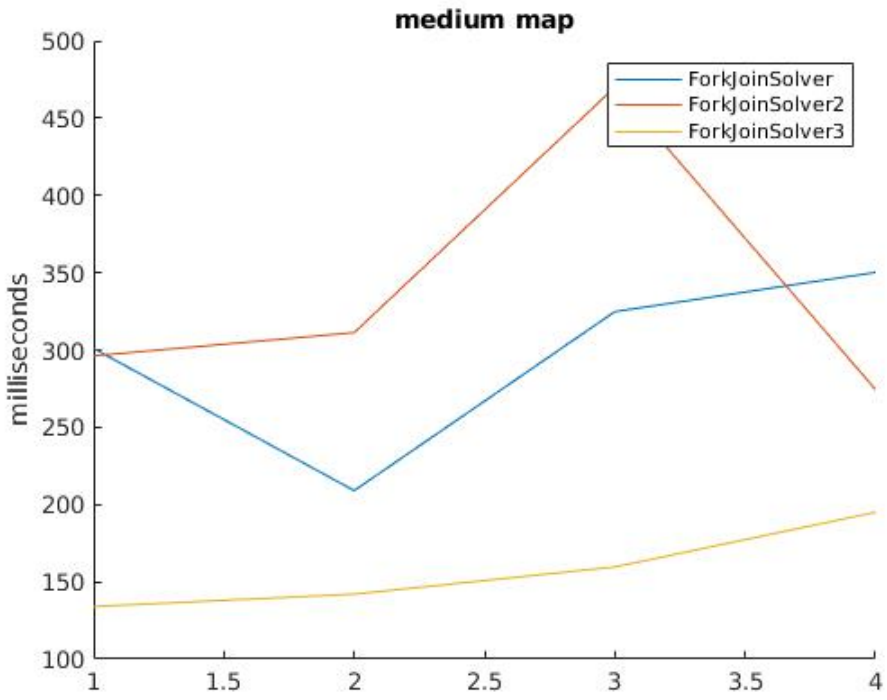

Figure 2: ForkJoinSolver, ForkJoinSolver2 and ForkJoinSolver 3 on the medium map. Note that the x-axis for Solver 1 and 2 is the value of forkAfter while for Solver 3 it is the number of workers in the pool.

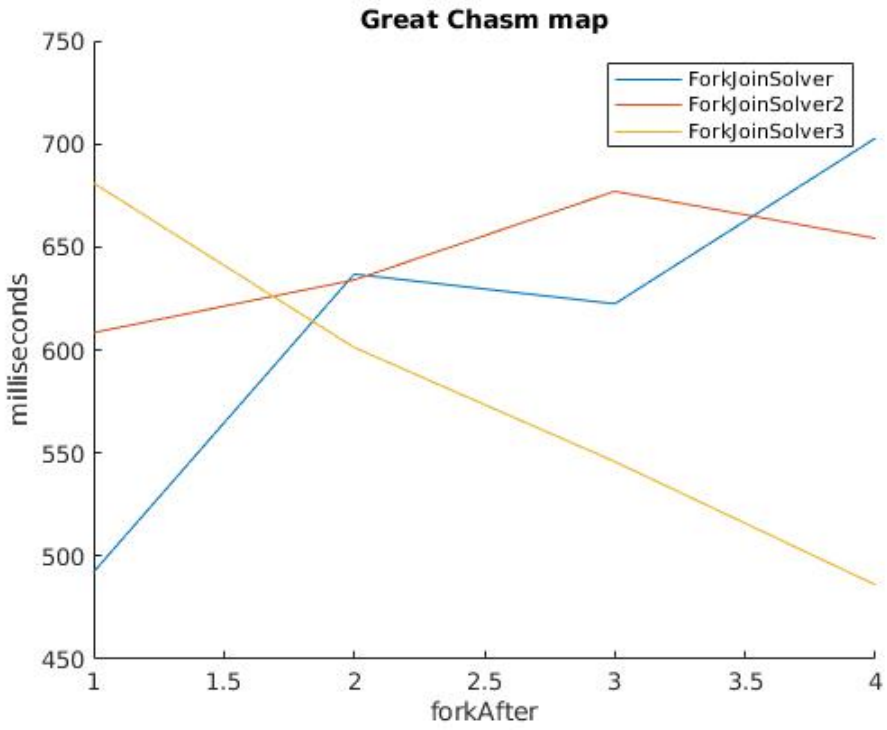

Figure 3: ForkJoinSolver, ForkJoinSolver2 and ForkJoinSolver 3 on the custom map Great_Chasm. Note that the x-axis for Solver 1 and 2 is the value of forkAfter while for Solver 3 it is the number of workers in the pool.

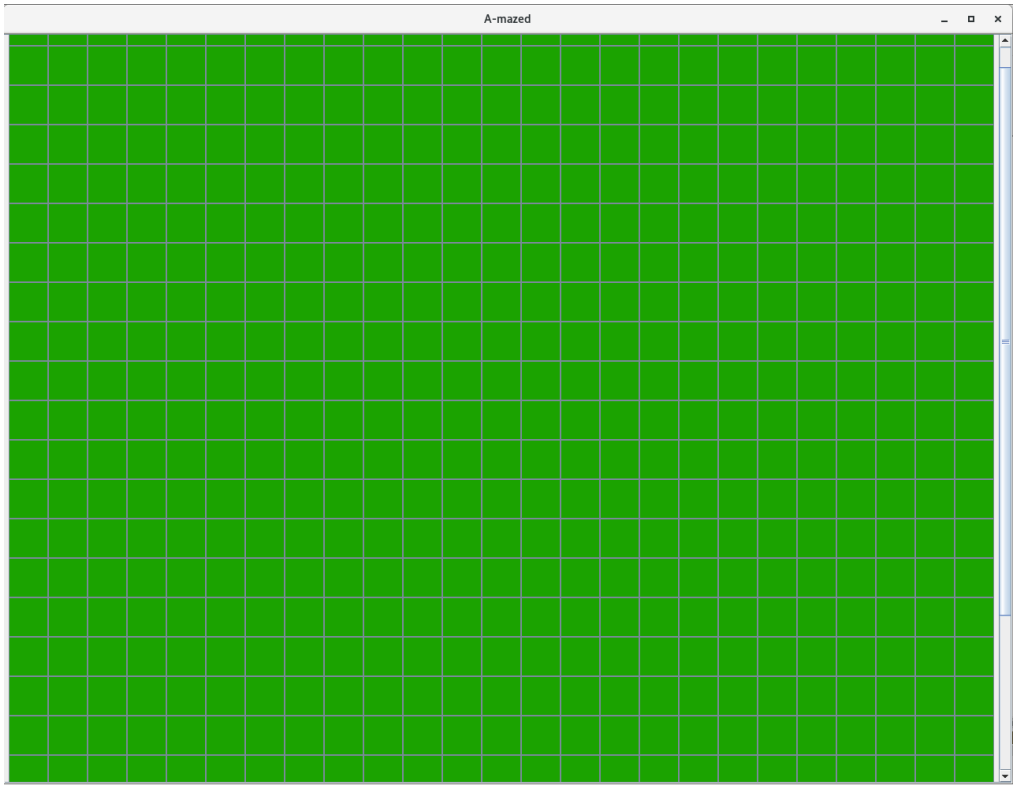

Figure 4: Map of the Great_Chasm map. Has no goal within it and is made to purely see what happens with the ForkJoinSolvers if the map is somewhat bigger than the ones given to us.