CSCI-GA.3033-017 Special Topic: Multicore Programming
Lab 4 Assignment (Part 3: Solving Mazes with Genetic Algorithms)

This is the fourth and final lab in which you'll be building a multithreaded system capable of using genetic algorithms to solve interesting problems. In Lab 2, you implemented a thread-pool-based solver capable of using several worker threads driven by a main thread to search for the correct coefficients to fit a line to a set of points; in Lab 3, you optimized that solver and quantified its performance. In this lab, you will attempt to solve something a bit more interesting: mazes.

Finding the path from the start to finish of a maze is a very well-studied problem, with many (and deterministic) solution methodologies developed. You'll be ignoring that body of work, and using your genetic algorithm solution framework to genetically find the path from the start of a maze to as close to the finish as possible.

- You will be given a class that, given a width and height, will give you an array containing 0 where there is empty space and 1 where there is a wall, as well as 0-indexed row and column coordinates of the start and end of the maze.
- You will write a program that spawns *mixer thread(s)* and *mutator thread(s)*, building a genetic code (genome) for the best possible solution through the maze. It will follow a more traditional GA methodology than your line-of-best-fit solver, by repeatedly mixing (mating) candidate genome and mutating the results.
- Your program will terminate when it has failed to make any forward progress for *g* generations.
- Your program will take the following arguments:
  `./lab4 <number of total threads> <threshold` *g* `for determining completion> <rows> <cols> <genome length>`
  It is your prerogative to choose the ratio of mixer to mutator threads.

Detailed Solution Algorithm

Your program shall generate an optimal genome representing a path through a maze. It will start with 4 times as many candidate genomes as you have threads, and repeatedly mix and mutate those genomes until it fails to any additional forward progress for *g* generations.

Genome

The genome your program should produce consists of five possible symbols, for convenience, numbered 0-4. Every genome will be of fixed length (see the arguments to your program above). To test a genome, a mutator thread will follow the genome from beginning to end, symbol by symbol,

and simulate the path that the genome represents through the maze. 0 represents no movement, and 1-4 respectively represent *attempted* movement up, down, left, and right.

You should start by initializing your genomes randomly, although you may want to test initializing them to 0 to seeing how that affects your solver's runtime.

Fitness Metric

The fitness of a genome shall be the sum (2 * (the *taxicab distance* from the ending point of the path encoded in a genome, and the finish point of the maze)) + (the number of times the simulated agent attempts to run into a wall). You'll attempt to minimize this value. It is so chosen that running into a wall is discouraged, but being too far from the finish point is discouraged more.

Global State

Your global state will consist of:

1. **Population:** An ordered multimap mapping from fitness values to genomes, initialized with (4 * number of threads) candidate genomes. This should be implemented as a thread-safe ordered multimap (see std::multimap), protected by a reader-writer lock. You should make a class for this, and it'll be fastest if you simple modify your ThreadSafeKVStore class. This class should have three extra thread-safe methods:
    a. **operator[n]:** Thread-safely, select and return the *n*th element in the ordered multimap, clamping the indices to the length of the multimap (i.e., if *n*>=the number of items in the multimap, return the last one). This operator need be *only* an rval, not an lval.
    b. **truncate(n):** Thread-safely truncate the multimap to the first *n* k-v pairs.
    c.
2. **Offspring:** A thread-safe queue containing offspring of mixer threads that are waiting to be mutated.
3. **Futility Counter:** A thread-safe 64-bit unsigned integer field, incremented each time a mutator thread performs a mutation, and reset to 0 every time a mutator thread notices that the fittest genome has changed.

Mixer Threads

Mixer threads are responsible for mating candidate genomes to produce a new generation. This solver uses *asynchronous* generations, rather than alternating between mixing and mutating all candidates, BSP-style. Specifically, mixer threads shall:

1. Select two random rows from the Population multimap of candidate genomes (whether this randomness is uniform or otherwise is up to you).
2. Randomly select a *splice point*: the resulting mixed genome shall consist of the first parent's genome up to the splice point, and the second parent's genome after the splice point. Watch out for useless corner cases (i.e., the splice point at the beginning or end of the genome).
3. Insert the resulting mixed genome into the Offspring queue.

Mutator Threads

Mutator threads are responsible for possibly mutating a new offspring, and then inserting it in the Population multimap. They are also responsible for checking if the fittest genome changes during that process, to determine when the system has stalled. Specifically:

1. Fetch the first row in the multimap (because this is an *ordered* multimap, this row will have the lowest cost, i.e., best fitness). Locally store the key (fitness value) of that row.
2. Fetch a genome from the Offspring queue, waiting if necessary for an offspring to be available.
3. With 40% probability, mutate the offspring, by modifying one random element to a new value, 0-4.
4. Compute the fitness metric for this offspring, and insert it into the Population multimap.
5. Truncate the Population to (4 * num threads) elements.
6. Fetch the first row in the multimap again. If the fitness of the first row has decreased, clear the Futility Counter, otherwise, increment it. Terminate the program if the Futility Counter has exceeded the command line argument-specified threshold.

Termination

The program shall terminate when a mutator thread notices that the Futility Counter has exceeded the threshold specified on the command line. You may choose to save yourself some time and abruptly abort your program, by simply displaying the maze, the best genome on the screen, and its fitness on the screen, and calling abort(). If you have the time and inclination, you may wish to make the termination more graceful.

Testing

To test this program, experiment with maze sizes between 7x7 and 21x21 and genome sizes from 20 to 200, and vary the number of threads you use from as many cores as are available to twice that number. When you have a correct solution and some intuition into the relationship between the maze size, genome size, and time to best solution, submit your program along with *one* paragraph briefly summarizing that relationship.

Administrivia

Grading: This lab will contribute at most 25% to your lab score. It will be graded on functionality, adherence to the specification, thread safety, code style, and commenting. Proper thread safety will be given an inordinate share of the grade, for obvious reasons.

Getting Help: If you're struggling, your first recourse should be the class mailing list. *Important:* you should solicit your fellow students for high-level help, such as "how can I iterate through a vector?". You should not ask for specific help with your code, such as "why doesn't the following code work?". If you're still stuck in a week, come to my office hours on Monday. See the first lecture for the policy on collaboration with other students (tl;dr: don't, outside of the mailing list). As is university policy, instances of cheating will be taken very seriously. If you believe there's an omission or error in this document, you're welcome to email me directly.

Due date: December 17, 2018, by 11:59:59pm EDT. See the first lecture for the late policy.

Submission: Push your code (and a PDF of your report) to your **private** MulticoreProgramming repository in a folder entitled "lab4". Please also send me and the grader an email once you have committed and pushed your final submission with the tag (a 7-character hexadecimal string) of that commit; I will use the timestamp GitHub records for that commit to determine whether your submission is on-time or late.