CS4414 Homework 3

Released on October 22, due November 5. Three days late submissions allowed, at 5pts off per day.

Homework 3 builds on homework 2, but is based on Ken's extended version of homework 2, which is not identical to the original homework 2 assignment (it has some fancy extra stuff). Originally, our plan was to allow you to build on your own version of homework 2 but to encourage focus on the goals listed below, and make the experience more uniform across the class, we decided that in fact having everyone used a single provided solution from Ken would be a better choice.

Goals: In homework 3 we will:

1) Run gprof to understand the performance of Ken's solution
2) Experiment with some changes and see their impact on the speed of the simulation
3) In the writeup, discuss the degree of speedup you obtained.

Ken's version of homework 2 differs slightly from what you probably created. He even calls it HW3 to avoid confusion. Ken added several features, including

1) Ken found it useful to support many verbose output modes. These are controlled by –v followed by one or more single character codes to control the form of output (cars, intersections, synchronization, etc). Look at his code to see which options he supports.

2) Ken's version tests an additional "medium" traffic load.

3) His code figures out which light should be green or yellow at t=0 at each synchronized intersection, and how long it should remain that color.

4) On synchronized streets, Ken sets the synchronized lights to 60s (*not* 90s), and the cross-street green lights to have a 30s cycle so that the synchronized street gets 90s green. In fact, all street green lights corresponding to unsynchronized intersections also have a 30s cycle (instead of the counter logic we implemented in HW 1). This turns out to be important – synchronization has a benefit with this pair of changes. Lacking it, the benefit was less evident.

5) Ken includes a feature for simulation of cars speeding a little on synchronized streets (yes, this exceeds the speed limit, but it does match the reality…). This matters too.

6) Ken's version includes automatic detection of gridlock.

7) Ken's priority queue holds shared_ptr objects to Car and TrafficIntersection objects, which in turn all derive from a base class called AlertEvent. This was not so easy to do, but has the benefit that objects won't be deleted inappropriately and there are no memory leaks.

8) Ken uses a std::map instead of a std::unordered_map. While they both support key lookup as the basic feature, they have important differences that are not really relevant for our assignment. Sagar prefers an std::map, in general, unless there's a strong reason to use a hash map instead of a tree map. For a discussion of the differences, see Stackoverflow posts https://stackoverflow.com/questions/2196995/is-there-any-advantage-of-using-map-over-unordered-map-in-case-of-trivial-keys and https://stackoverflow.com/questions/3902644/choosing-between-stdmap-and-stdunordered-map.

Ken's code also is deliberately inefficient in some ways. In homework 3 you will be assessing the performance impact of those design choices by (1) measuring their costs, and (2) modifying the code and

remeasuring to see if you obtained a speedup. Here is a list of the inefficiencies Ken deliberately introduced:

1) Just like on homework 2, Ken used a std::priority_queue to track alert actions. But as people noted on Ed Discussions, we could also consider a "queue" of type std::vector<std::vector<std::shared_ptr<AlertEvent>>>, and this is potentially much faster. We will explore this in homework 3, and will see how much impact it has. By the way, in words, that type is "a vector, containing…" where containing is "vector of…", and those vectors are in fact vectors of shared pointers to AlertEvent objects – the base class for Car and TrafficIntersection objects. The idea is that the outermost vector would represent time (for example, entry 37 would be events that should occur at t=0), and then the list of events at that time would also be a vector (perhaps empty), and finally the events themselves are the objects to notify at that time. When needed, Ken's code has a cast to turn a shared_ptr<AlertEvent> into a real pointer to a Car (e.g. static_cast<Car>(theSharedPtr.get())) or TrafficIntersection. You may find it useful to make a picture for yourself to visualize this.

2) As most people did in homework 2, Ken loaded the Traffic Lights and Sync and Cars data as strings. He keeps them in this format, forever. As a result, he ends up calling stoi and stod whenever he needs something as a number. Moreover, his version of the driving time function actually parses the POINT string from the SF traffic lights data every time a distance is needed. These seem like costly and doubtful choices. Why parse or convert them again and again?

3) Ken's code has the cars rechecking a red traffic light once per second until it turns green. But in fact if a traffic light will be red for 17 seconds more, this means he is rechecking 16 times, pointlessly. That must have some cost.

4) Ken has been compiling and testing with the –g flag and without –O3, because this enables gdb to track down individual lines that caused problems. But –O3 is needed to tell the compiler to go all out on optimization. And –g has some impact on what optimizations are allowed.

On homework 3, your job will be to investigate each of these potential issues, decide if they represent a speedup opportunity, and if so, to implement the needed code change by modifying Ken's code. If not, you will tell us why the change isn't worth making. Sometimes a decision like that pays off – you don't waste time modifying and debugging code you didn't even write yourself. This is not at all unusual in terms of preparing for a job in the computing industry: most companies have tons of existing code, and it is common to ask a new employee to try and speed up some existing program that they didn't write, and that doesn't have many comments. You should make changes that have real value… and not others!

Note: Please do not post Ken's code on any kind of public site, like Chegg or Course Hero. You didn't write this code; it isn't your "IP", and it was provided only for you personally to do this assignment. This is also why Ken changed it from hw2 – if it does leak, it won't really be so easy to turn it back into an hw2 solution next year or in 2023… but we prefer that it not leak, even so!

Step 0: Get used to the given code

Your version of homework 2 probably looked somewhat different from Ken's homework 3. So it will take a little time to get used to this. Compile Ken's code and play with it. Try the various –v flags alone or in combination, and see what the output looks like. Time the program for runs of a few lengths, such as –t=1, -t=25, -t=50, -t=100, -t=200. Ken's v flags are self-documented in the code itself. You can use them combined or one by one, or say -v to get WAY MORE output than you expect!

Notice that starting with 25, each subsequent run doubled the prior one: one runs for 50 seconds, then

100, then 200. You will see that the runtime slowdown is much more than 2x: this is a "non-linear" slowdown. This is because our rule for adding cars brings them in so rapidly that the new cars end up waiting behind the current cars – we create a burst of extremely heavy traffic. This prevents us from running a more realistic simulation, such as –t=3600. For a longer, more realistic simulation, we would need a more realistic model for introducing new cars.

Step 1: Understand the performance cost of not optimizing and compiling with debugger information (-g), optimizing but also profiling, and finally optimizing but not profiling.

1) First, compile hw2 using the flag -g. Do not use -O3 or -pg. Time this version of hw2 for -t=100.

2) Now, recompile with flags -O3 and -pg (and not -g). Time this second version, again with -t=100.

3) Finally, recompile with -O3 and nothing else. Time this third version. Make note of your findings.

In what you hand in, we will look for a discussion of step 1 with each of these 3 cases listed, and for each case, a sentence or two summarizing your findings and your explanation. Does -O3 matter?

Step 2: Recompile with -O3 and -pg, run with t=100.

For this next step, run "gprof hw2" and study the gprof output page by page, using "more" to avoid having it vanish off the screen. You could alternatively put the output in a file and examine it.

Which methods are costing the most CPU time? Look back at the list of how Ken's version was designed. Can you confirm that the use of the priority_queue is turning out to be quite expensive?

At the same time, notice that stod and stoi are not listed. Yet Ken's version of hw2.cpp makes a lot of use of those function, especially in the costly methods! Explain briefly why stod and stoi are not shown. *Hint: If you research it you will find that these are both templated methods. Yet gprof examines the symbol table of the compiled executable… could this explain the issue? How and why?*

Temporarily modify hw2 to count the number of calls to each of these two functions (this isn't a change we plan to keep, we simply want to know how many calls occurred in each case). *Hint: Create "helper functions" that do the counting, then call the corresponding method.*

Now build a tiny little tester program for each of these functions and see how costly each of these actually is, by calling it 1M times, and using the "time" command to measure the amount of time spent. Use arguments similar to the ones they would be given in hw2.

In your solution, we expect to see a small section for step 2, and a discussion of what you learned from gprof. Your discussion must include numbers obtained from the gprof output, and also must include numbers from the testing we just described. We expect to see a sentence or two about stod and stoi.

Step 3: Too much compute time is spent in the std::priority_queue library.

Step 2 revealed an interesting opportunity. Although a standard discrete event simulator really would use a priority queue, in our case it would make sense to switch to a vector of vectors: if type T is std::vector<std::shared_ptr<AlertEvent>> then our outer vector could be declared like this: std::vector<T> theVector(LENGTH).

Why would this work? With a standard discrete event simulation, the priority queue will need to track events at arbitrary times and perhaps on a very large time scale. But our simulation runs second by second, and even a long run never exceeds a few thousand seconds of driving time (with heavy traffic our simulator predicts about an hour and a half to cross the whole city, and sadly, anyone who has done that knows this is actually possible. With light traffic, the same drive is maybe 20 minutes). Thus our vector won't need to be very long – LENGTH can be fixed at 20,000 or so. Check to make sure that the –t argument doesn't specify a run length longer than LENGTH-10000 and you should be safe. Alternatively, you can grow the

vector as needed. No matter what, be careful to initialize all vector entries with empty event lists.

Caution: Whatever approach you use, if you use a fixed-size buffer and as a result there is a situation where you could overflow your buffer, you must check for this condition and print an error message and then abort if it were to happen. You should not hand in a program that would crash without detecting its own problems for certain inputs, like an unexpectedly large value for -t=n. This issue would never arise with a dynamic buffer that you grow as needed, so it only applies for fixed buffers that can overflow.

As for the vector of events, this will just track all the "ties" at a given second, in FIFO order.

When making this switch, it will be important to initialize the std::vector<std::shared_ptr<AlertEvent>> objects, by calling std::vector<std::shared_ptr<AlertEvent>>() – the vector constuctor for an empty vector – at a suitable place in the program. Additionally, keep in mind that the program runs as a loop, and each time it loops, we need to "reset" the vector back to its initial state. There is a "clear" method that does this efficiently.

In your writeup, have a section called step 4, and discuss the before/after impact of this change. We won't tell you what to measure, but you should use your new skills as a performance analyst to measure appropriate things and then to arrive at an insightful perspective on the effect of the change.

Step 4: Too much compute time is spent in stoi and stod.

As our simulation runs, it needs to compute distances and CNN numbers, and in Ken's version, this involves calling stoi and stod each time a conversion is required (plus, parsing the POINT string each time a distance is needed). Those are very costly and unnecessary operations.

Invent your own way to ensure that stoi, parsing the POINT string and stod are never called more than once for the same thing. Implement your solution, and measure the before/after impact of the change.

In your writeup, tell us what you did, and discuss the performance impact of the change.

Step 5: Extra credit, optional. Cars check every one second while waiting.

Step 6 will be purely an extra credit opportunity. It is not required. By studying the program, arrive at a prediction of whether "fixing" this behavior would offer a significant improvement. But then convince us that you are correct by implementing the needed logic so that a car waiting at a traffic intersection rechecks after exactly the right number of seconds. Measure to see whether your prediction was validated (was correct). In your extra credit writeup, tell us what you expected, what you did, and what the impact actually turned out to be.

Step 6: Other ideas, for extra credit, optional.

You might have other ideas for speeding up the program. For additional extra credit, you are welcome to try them – even things that involve fairly extreme changes. In your extra credit writeup for step 7, if you do this, tell us what you explored and why, and what performance impact it actually had. But be aware that in fact steps 1-5 may have used up the best opportunities!