Ryan Patton
06/18/2021

Implementation Summary:

I found a potential honeypot source on Github.com and followed through with its implementation trying to think of ways to improve the results. The results from the initial run without modifications may be seen in the Results section in Table 1.

The code utilizes a greedy algorithm that assigns the beams to least-covered users first. The data structures used in the algorithm are scenario_t, vector<SatBeamEntry>, and vector<UserVisibilityEntry>. Scenario_t stores ids and positions of all users, satellites, and interferers. Vector<SatBeamEntry> uses a list of all the color beams of each satellite (size = colors / sat * # sats) to check for self-interference and max-beams constraints. Vector<UserVisibilityEntry> contains a list of all users and their respective visible satellites. To be considered “visible”, the satellite must be within visible range of the user while honoring the non-starlink interference constraint. These data structures make it easy to iterate through a list of user visibility to number visible satellites and assign beams while honoring the self-interference and max-beams constraints.

The code was ran on a Macbook 2020 with M1 chip, using Homebrew and installing build-essential to support compilation. The solution uses g++10 and needs to use C++11 or later because of the 3D vectors. The total time for the initial run was 125 seconds. Not great timing but not horrible either, the final test case took 111 seconds. It could be parallelized for a couple of the sized array iterations in the code to improve runtime if really desired. Each test passed the checks for “all users’ assigned satellites are visible”, “no satellite self-interferers”, and “no satellite interferers with a non-Starlink satellite!”.

Results:

Test Case	Percentage of Total Users Covered (%)
00_example.txt	100
01_simplest_possible.txt	100
02_two_users.txt	100
03_five_users.txt	80
04_one_interferer.txt	0
05_equatorial_plane.txt	100
06_partially_fullfillable.txt	76.8
07_eighteen_planes.txt	98.52
08_eighteen_planes_northern.txt	79.12
09_ten_thousand_users.txt	93.18
10_ten_thousand_users_geo_belt.txt	83.6
11_one_hundred_thousand_users.txt	29.271
*Test Cases 3 and 4 would not be possible to reach 100% without violating a constraint	
Total Time: ~125 s (test case 11 was 111 seconds)	

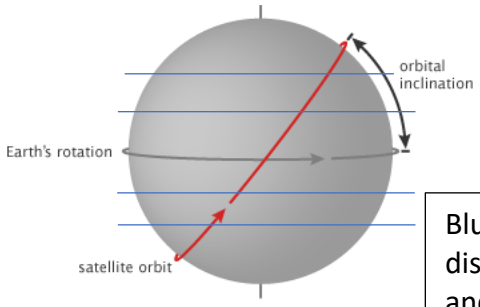
Table 1. Run Results

Since it's hard to know best test coverage possible without really analyzing the test case data, it's hard to know how accurate the data really is. Test cases 1-5 can be verified with 100% accuracy (see Improved Suggestions talking point 1) for explanation of 3 and 4).

Test Case 6 can be verified to be 100% accurate given the constraints of the problem:

(60 sats * 32 users_per_sat) / 2500 users = 1920 / 2500 = 76.8%

Test Case 7 has enough satellites (360) to cover the 2500 users if they're spread out properly (360 * 32 = 11,520). The inclination of 18 planes of 20 satellites at 53 degrees can be seen in the image, with some of those planes closer and further from Earth. I was picturing the problem seen below as the example:



Blue lines = users distribution in 49 and 53 degree latitude bands

With the above image in mind, and looking at the test case data, the results are close to 100% (98.52%) but not exactly at 100% for coverage because of a couple edge cases where the it may be impossible to form a 45 degree angle between a user and starlink satellite. Scanning the data for sats shows the data is following a pattern where every ten sat ids the x, y, and z values may just changes to negatives as another point/plane for the satellite placement. For the different planes and their angles between the satellites and users, since the latitutde bands with users scattered in them (49 – 53 degrees) are so close to the satellite inclination of 53 degrees, I think it's reasonable to assume that the cases where users are at 53 degree latitude band, and the inclination is 53 degrees causes the small amount of test case coverage failure. The problem states users are scattered through 49 and 53 degree latitude bands so I wouldn't expect too many users to be right at the 53 degree latitude band (~1.5%).

Test Case 8 should cover the 10% of users in the southern hemisphere. With 90% of the user population now in the northern hemisphere, the reduction in test case coverage to 79.12% seems reasonable as a ~20% drop in coverage. The drop in coverage might stem from the inability to create 10 degrees of space between users in the Northern Hemisphere. Assuming a 50/50 split in test case 7 between the Northern and Southern Hemispheres, in this test case 80% of users previously in the Southern Hemisphere have been moved to Northern, so the new dispersion would look like 2250 users in Northern Hemisphere and 250 users in Southern Hemisphere. This math could probably be further worked to directly prove it stems from the 10 degrees needed between each user.

Establishing the validity of Test Case 7 makes the user test coverage for Test Case 9 easy to digest at 93.18%. The users are distributed in a slightly broader range of latitude bands (40 and 55). There's double the planes of 20 satellites in this problem, with broader degree latitude bands. While the broader latitude of bands makes it easier to avoid the 53 inclination edge case errors, the added planes of satellites is not proportional enough to overcome an addition of 7500 users. The satellites have been doubled, and the users have been quadrupled. One-fourth of original users divided by doubling the satellites and planes yields 6.25%. Subtracting from 100% should give 93.75%. This math is a best guess and I'm not sure if the highest coverage

would be 93.75% but it seems approximately right given the increases in planes, difference in latitude bands, doubling the satellites, and quadrupling the users.

Test Case 10 changes the latitude bands to 45 and 55 from 40 and 55, which should reduce coverage slightly and then adds in 36 interferers. I'm sure there's some math I could think of to find the difference in coverage between the latitude band change and adding in 36 interferers, but I'm in a rush to think through the scenarios and think 10% drop in coverage is reasonable.

Some basic math that might be of use:

36 interferers / 720 satellites = 5%
33% shrink in latitude band range

The 29.271% test case coverage for test case 11 initially jumps out as being poor, but taking only the beam constraint into account already narrows down the best possible coverage to 47%. The runtime seems sufficient, and were really great up until the test case 11 which took 111.69821762500001 seconds. More satellites, more users, same number of interferers, different lat bands (0 and 57 degrees), and 90% of the population in the Northern Hemisphere definitely complicates the problem. I think you could perform enough mathematical calculations building on the previous test cases to precisely determine how accurate test case 11 is, which is the culmination of the all the previous test cases most difficult aspects to analyze wrapped into one. Off the bat, I think the biggest factor is the dramatic increase in users compared to satellites. (1440 sats * 32) / 100,000 users = 46.080% (possible max coverage based on that factor alone). While 90% in Northern Hemisphere caused a 20% drop in coverage for another test case, this test case has broader latitude bands which would increase coverage, perhaps when combined with the interferers enough to cause a 16% drop in coverage.

Improvement Suggestions:

I initially thought from the numbers that the code would have drastic room for improvement. After looking at each scenario, I think the results are pretty close to optimal. A very rough estimate would be in the 90-95% range if you take into accounts the constraints placed on the problems (some of it depending on the weight of each test case too, like if test case 4 was weighted the same as test case 11 that might bring down the optimal coverage). Taking the initial run results and working to improve them, I came up with a couple suggested ways to improve the code to improve accuracy:

1) To improve the code, a separate loop only applicable for each test case that is impossible to cover 100% could be set up. These loops would make it acceptable to violate the constraints placed on the problem to give better test case coverage if desired. This would have been a good question to ask during my window for asking questions but I did not think of this during the 1.5 hour window. Is it better to maintain the constraints placed throughout each test or is it better to violate the constraints to achieve better coverage in edge cases scenarios like 3 and 4? For test case 3, since we

know that it has 5 users covered and it is impossible to reach 100% coverage, 80% coverage suggest 4/5 users are covered indicating it is operating at its max coverage potential. With only one user in test case 4, not violating the problem's constraints will simply give a test case coverage of 0% as seen. Nothing suggests that the algorithm is not working correctly just that it's not working as efficient as what it could if the emphasis is on test case coverage over consistency with constraints.

2) The code approximates pi numerically as 3.141592653589793. The code already has the necessary #includes at the top to use "M_PI" instead. This may give slightly better test case coverage for results seen in later test cases. I would guess this effect would be very minimal if not completely negligible but it couldn't hurt to try.
3) The greedy algorithm seemed to work pretty well based on my analysis of each test case, and while there might be very slight room for improvements with test case coverage, I think the logic of the code structure is more important than looking to improve upon a greedy approach. Tying into improvement 1), the logic of the code stays within the bounds of the constraints. If the code could be reformatted to include checks for the test case being run so it knows what constraint to violate than the coverage could be improved. The data structures used aren't as important for verifying the data as what I initially thought as long as the constraints are set up correctly in the code, and there are a lot of indications that it is set up correctly outside of minor edge cases accounting at most for a small percentage of test case coverages missed.
4) Different data structures could be tried to improve the runtime even though this isn't the top priority. It already makes use of a map in part of the code so I think it would be more beneficial to focus on parallelizing certain parts of the code and examine underlying hardware it might potentially be interacting with in a real life scenario.
5) To validate the test cases better, it might help to create a physical map from the test case data to help visualize where the points lie. This helps be able to see any of the edge cases visually instead of guessing where the coverage drops off.
6) This seems like a prime area for implementing ML/DL/Kalman filters if it was a real-time simulation with satellite data that was being constantly updated.

Final Thoughts:

While the code was not built from scratch, I viewed the assignment as an opportunity to take advantage of time management and come out with a solution that is understandable, and ideally optimal. I viewed it as a need to not reinvent the wheel if what's already existing can be improved and be timely implemented, as long as time has been put in to understand the problem and the underlying assumptions. I would not have tested the implementation if I thought it was a waste of time. Given a 48 hour window with other priorities like work, driving somebody to an airport an hour away, and answering some calls I knew it would be a time crunch to implement. It's not like I didn't think the problem was interesting but I view it as taking advantage of existing resources to come up with a time-effective solution. I made sure to include a section analyzing each test case's results and a section for ideas to improve the code to show my understanding of the implementation and how it could be more optimal. The main

goal of the problem in my mind is to think through tradeoffs and the impact the code will have on giving optimal results. It's interesting that you really don't have a set way of analyzing what the best possible coverage is, so it's up to you to analyze the test cases and think of the edge cases to come up with more optimal code. This is where I tried to place the emphasis of my efforts, but there's room to mathematically verify each scenario I believe to know exactly how accurate the code is. I understand if this practice is unacceptable as well, given the time constraint I thought it would be a better go than worrying about implementation details that don't matter for solving the problem. If this was a real-life scenario with real-time data, I would rely on a ML/DL approach in Python using Google Colab. I would be happy to further work on it and further analyze the test cases presented to see what additional data structures may have worked more proficiently, but at this point I think it would be more interesting to try to validate the test case coverage as close as possible.