1. Below is proof of incorrectness:

Trace 1:

```
-------------------------
[2s] E 0.04k, U 0.00k, D 0.04k, 19/sec, active 33, run 100
[4s] E 0.38k, U 0.41k, D 0.78k, 168/sec, active 99, run 100
Thread 56: found exit: C4DA188C311D52BE, steps 925, distance 5
-------------------------
```

Trace 2:

```
-------------------------
[2s] E 0.16k, U 0.18k, D 0.34k, 79/sec, active 100, run 100
Thread 61: found exit: C4DA188C311D52BE, steps 554, distance 7
-------------------------
```

The fact that it got a distance of 7 the second run proves that it is incorrect, because it is greater than 5. If this were a true bfs, 7 would be impossible to achieve if the exit is reachable 5 steps away from the starting point, meaning that one thread got ahead.

2. Upon adjusting my hw1 code to handle batching and running it on planet 32, I discovered that after stabilizing it consistently maintains 50000-70000 rooms per second. Below is a trace of the output.

```
*** CC v2.4: starting with PID 52092 (hex CB7C)
*** CC: found 8 CPUs, 6400.51 MB of free RAM
Connecting to CC with planet 32, cave 1, robots 32...
CC says: status = 1, msg = 'opened (planet 32, cave 1) with 32 robot(s)'
Starting to search with threads using BFS...
-------------------------
{1s} [0.0 M] U 0.00 M D 0.00 M, 12 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{4s} [0.0 M] U 0.00 M D 0.00 M, 984 rooms/sec, 0*, 100% uniq [9% CPU, 0 MB]
{6s} [0.2 M] U 0.17 M D 0.37 M, 98752 rooms/sec, 0*, 100% uniq [3% CPU, 0 MB]
{8s} [0.4 M] U 1.45 M D 1.82 M, 85000 rooms/sec, 0*, 100% uniq [7% CPU, 0 MB]
{10s} [0.5 M] U 2.56 M D 3.11 M, 90000 rooms/sec, 0*, 100% uniq [8% CPU, 0 MB]
{12s} [0.7 M] U 3.74 M D 4.44 M, 75000 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{14s} [0.8 M] U 4.91 M D 5.72 M, 55000 rooms/sec, 0*, 100% uniq [12% CPU, 0 MB]
{16s} [0.9 M] U 5.99 M D 6.92 M, 60000 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{18s} [1.1 M] U 6.88 M D 7.97 M, 80000 rooms/sec, 0*, 100% uniq [13% CPU, 0 MB]
{20s} [1.2 M] U 7.67 M D 8.90 M, 70000 rooms/sec, 0*, 100% uniq [11% CPU, 0 MB]
{22s} [1.3 M] U 8.45 M D 9.73 M, 25000 rooms/sec, 0*, 100% uniq [21% CPU, 0 MB]
{24s} [1.4 M] U 9.21 M D 10.65 M, 77500 rooms/sec, 0*, 100% uniq [9% CPU, 0 MB]
{26s} [1.5 M] U 10.02 M D 11.49 M, 20000 rooms/sec, 0*, 100% uniq [12% CPU, 0 MB]
{28s} [1.7 M] U 10.78 M D 12.44 M, 95000 rooms/sec, 0*, 100% uniq [9% CPU, 0 MB]
{30s} [1.8 M] U 11.66 M D 13.48 M, 80000 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{32s} [1.9 M] U 12.50 M D 14.37 M, 20000 rooms/sec, 0*, 100% uniq [12% CPU, 0 MB]
{34s} [2.0 M] U 13.20 M D 15.23 M, 82500 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{36s} [2.1 M] U 14.17 M D 16.23 M, 15000 rooms/sec, 0*, 100% uniq [11% CPU, 0 MB]
{38s} [2.2 M] U 14.95 M D 17.18 M, 85000 rooms/sec, 0*, 100% uniq [11% CPU, 0 MB]
{40s} [2.4 M] U 15.77 M D 18.15 M, 75000 rooms/sec, 0*, 100% uniq [11% CPU, 0 MB]
{42s} [2.5 M] U 16.65 M D 19.10 M, 37500 rooms/sec, 0*, 100% uniq [11% CPU, 0 MB]
{44s} [2.6 M] U 17.38 M D 19.99 M, 77500 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{46s} [2.7 M] U 18.22 M D 20.87 M, 22500 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{48s} [2.8 M] U 18.85 M D 21.67 M, 85000 rooms/sec, 0*, 100% uniq [11% CPU, 0 MB]
{50s} [2.9 M] U 19.64 M D 22.50 M, 17500 rooms/sec, 0*, 100% uniq [13% CPU, 0 MB]
{52s} [3.0 M] U 20.32 M D 23.36 M, 90000 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{54s} [3.1 M] U 21.15 M D 24.24 M, 22500 rooms/sec, 0*, 100% uniq [10% CPU, 0 MB]
{56s} [3.2 M] U 21.85 M D 25.09 M, 75000 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{58s} [3.3 M] U 22.64 M D 25.95 M, 40000 rooms/sec, 0*, 100% uniq [9% CPU, 0 MB]
{60s} [3.5 M] U 23.36 M D 26.82 M, 72500 rooms/sec, 0*, 99% uniq [11% CPU, 0 MB]
{62s} [3.6 M] U 24.09 M D 27.68 M, 67500 rooms/sec, 0*, 99% uniq [12% CPU, 0 MB]
{64s} [3.7 M] U 24.85 M D 28.54 M, 47500 rooms/sec, 0*, 99% uniq [13% CPU, 0 MB]
{66s} [3.8 M] U 25.53 M D 29.36 M, 67500 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
```

```
{68s} [3.9 M] U 26.26 M D 30.19 M, 52500 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{70s} [4.0 M] U 26.98 M D 31.00 M, 45000 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{72s} [4.2 M] U 27.66 M D 31.81 M, 67500 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{74s} [4.3 M] U 28.38 M D 32.63 M, 50000 rooms/sec, 0*, 99% uniq [13% CPU, 0 MB]
{76s} [4.3 M] U 29.09 M D 33.43 M, 42500 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{78s} [4.4 M] U 29.76 M D 34.21 M, 52500 rooms/sec, 0*, 99% uniq [10% CPU, 0 MB]
{80s} [4.5 M] U 30.46 M D 35.01 M, 50000 rooms/sec, 0*, 99% uniq [12% CPU, 0 MB]
{82s} [4.6 M] U 31.14 M D 35.77 M, 42500 rooms/sec, 0*, 99% uniq [11% CPU, 0 MB]
{84s} [4.7 M] U 31.85 M D 36.60 M, 60000 rooms/sec, 0*, 99% uniq [11% CPU, 0 MB]
{86s} [4.9 M] U 32.46 M D 37.36 M, 75000 rooms/sec, 0*, 99% uniq [15% CPU, 0 MB]
```

3.

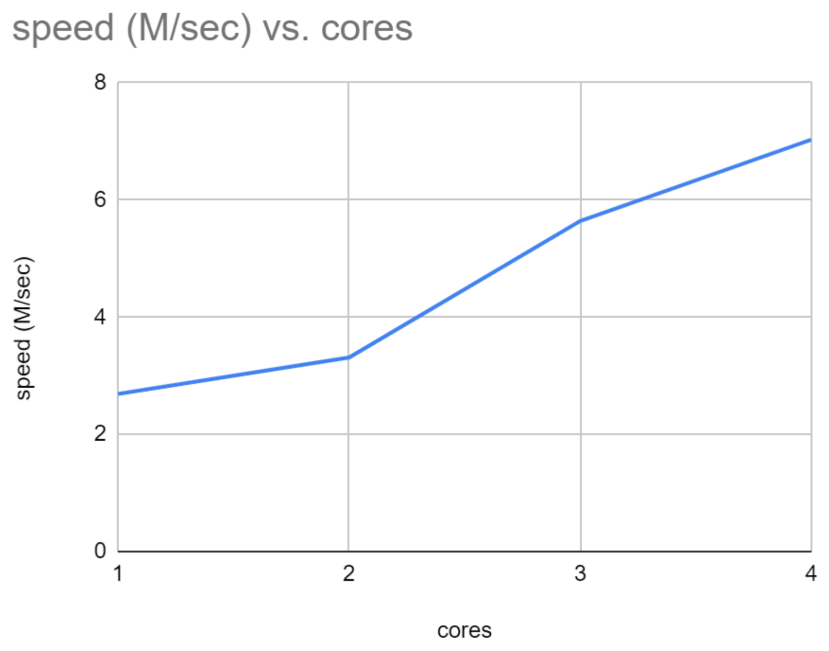
This question sucks.

4. The first strategy I began with was simply adapting my hw1 code to handle batching when popping from the queue. This is not very good however because the stl queue and set have a lot of overhead that could be made more efficient.

Next I implemented a custom queue and set. This instantly caused a huge jump in performance, but it was still being bottlenecked when it came time to push neighbors back into the queue because it was adding them one by one inside of a mutex.

My final iteration added a batch push function to the queue to solve the bottleneck issue, and uses a private heap to store neighbor data until the thread is ready to push its findings into the queue. This worked wonders and now I have an average speed of about 8 million rooms/sec.

5.



As you can see the relationship follows a fairly linear pattern.