Proj3 report

Daniel Silva-Inclan

8/20/2020

Traffic simulator

Describe in detail your system and the problem it is trying to solve.

My project attempts to model traffic congestion by simulating a city where cars traverse from one point in the city to another in a semi-optimized manner. The graph reflects edges as roads, vertices as intersections, and edge weights as time it take to traverse down a road. At each round, each car finds a min path with dijkstra's algorithm as a propose path. Note that dijkstra is implement with some randomly as each adjecent vertex is traverse is a random order. They traverse down that path but encounter traffic (other cars also traversing down that path). In the next round, each car incorporates the previous traffic to re-optimize their path. This is done on the graph as a change in the edge weight. As more rounds pass, the cars may converge to a steady state were there is no better outcome for any car. Depending on the structure of the graph and other parameters, the steady state of the system may be one path, multiple paths, a cycle of paths, etc. I will analyze the variance of edge weights with different parameters to better understand the system.

There are three cities type defined in the simulation: complete, city, and random.

1. complete – uses complete graph, i.e. each vertex has n^2 edges
2. city – each vertex (v_i) has a random number, $x_i \in X \sim U(2, 6)$, of edges to other vertices
3. random – each vertex (v_i) is connect to a random number, $x_i \in X \sim U(0, n^2)$, of edges to other vertices

For the experiment set defined in process.sbatch, for each city type, I search over 3 different car amounts, 3 different sizes of graph and 5 different thread amounts. Each experiment has 100 rounds and the final time is the time it takes to run 20 experiments.

A description of how you implemented your parallel solution.

The system is defined by many experiments which a graph with some parameters is generated. For each graph, some number cars run dijkstra concurrently and then afterwards update the edge weights sequential for each round. Once each experiment is done, the program writes the final edge weight to a json file.

The project parallelizes each experiment with a fanIn/fanOut implementation similar to proj2. Each goroutine in the experiment is (not quite) queued so that if it even encounters a lock, the program can work on another independent goroutine. This is implement with channels. Since each experiment has parts that are very concurrent and parts that are sequential, the fanin/out infrastructure allows other experiments to run while an experiment runs its somewhat heavy sequential write.

In terms of the data structure, the graph is defined by a RWlock, and several maps storing information about edges, vertices, and edgeweight. For most operations, the RWlock just Rlocks so that many readers can operate at once. This includes running several dijkstras concurrently (dijkstra itself is sequentially implemented). However, graph Locks when the edgeweights are being update at the end of each round.

Describe the challenges you faced while implementing the system. What aspects of the system might make it difficult to parallelize? In other words, what to you hope to learn by doing this assignment?

The most difficult challenge was balancing fast sequential write for the edge weights against fast concurrent access and space efficiency. Edge weights are usually store in one of 3 forms: adjacency matrix, adjacency list, or a nested hash table. Each one have different space and read/write time complexity. For example, in terms of space complexity, adjacency matrices are O(n^2) while both adjacency list and nested hash tables can be of size n to n^2. For look ups, adjacency matrices and nested hash tables are generally faster (O(1)) than an adjacency list (O(n)). Graphically representations of cities will be quite sparse (mean deg ~5 with low variance) and so nested hash tables or adjacency lists would be preferable in terms of space complexity.

I decided to use nested hash tables (maps in go) given that there were going to be many lookups relative to writes. In each round, each car runs a dijkstra (O((V+E)(log(V)))), and then writes on all edges E (O(E)). As such, it became more important to make sure that cars could run dijkstra concurrently than if the writes were done concurrently. However, in go maps are not concurrent with write calls so each write call must be synchronized with either a lock or CAS. My current implementation may be able to handle space complexity better, however, this is generally not a modern constraint. Perhaps a better implementation would have a adjacency matrices and use CAS on write calls. That way, dijkstra would remain relatively fast and concurrent while the writes would also be done concurrently. However, its unclear how much speed up this would allow as it would reduce cut into the benefits of using fanin/out. Future studies would compare how different adjacency representation would fair concurrently.

Specifications of the testing machine you ran your experiments on (i.e. Core Architecture (Intel/AMD), Number of cores, operating system, memory amount, etc.)

fast:

24 Cores (2x 24core Intel Xeon Silver 4116 CPU @ 2.10GHz), 48 threads

128gb RAM

OS: 2x 240GB Intel SSD in RAID1

/local: 2x 960GB Intel SSD RAID0

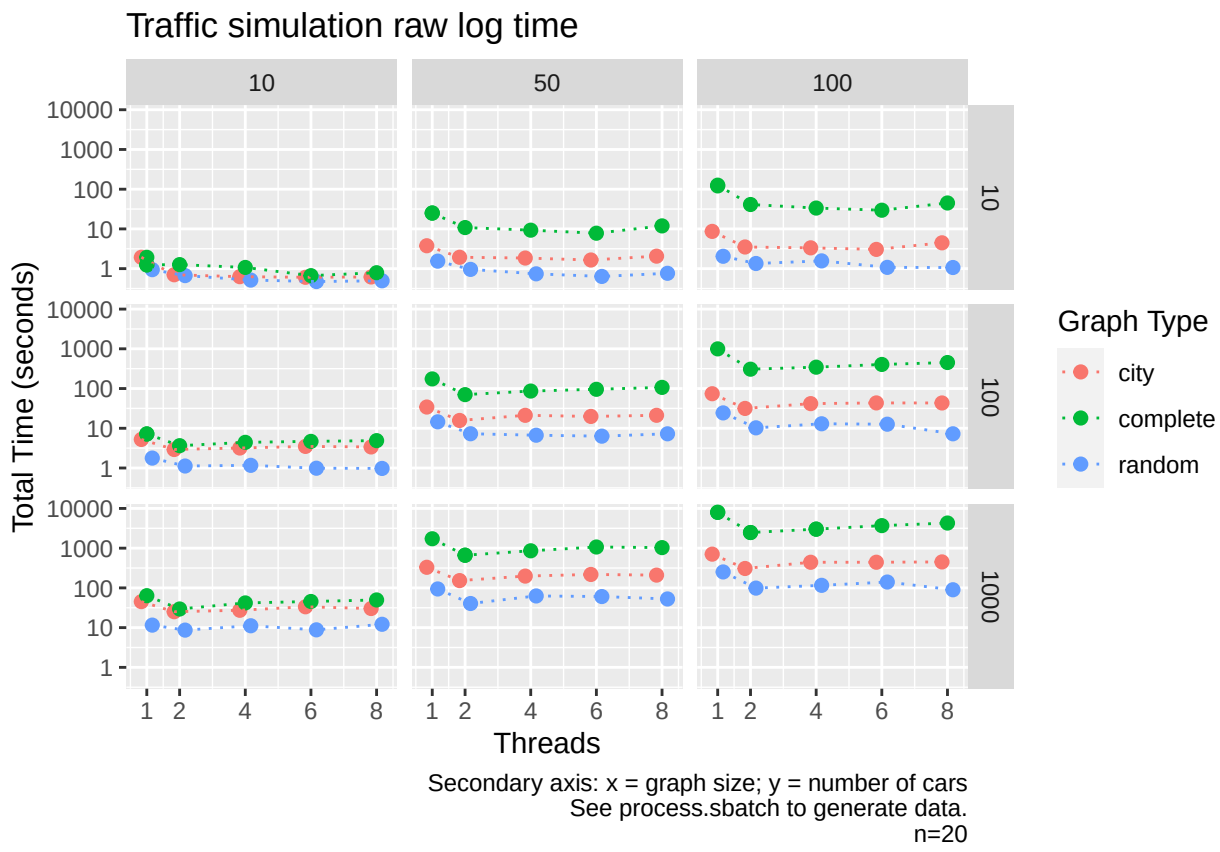
with only 8 cores and 8gbs of ram used in the experiment.

What are the hot spots and bottle necks in your sequential program? Were you able to parallelize the hotspots and/or remove the bottlenecks in the parallel version?

There are clear bottle necks in running dijkstra many times which was made concurrent with RWlocks and waitgroups. The number of cars in the system increases the run time faster than increasing the size of the graph. This is shown in the raw log time plot when thread=1, moving along the plots downwards increases the time more so than moving along the plots rightwards. However, when more threads are added the effect reverse and so moving along the plots rightwards increases more so than moving along the plots downwards.

The same effect could also be the result of the fanin/out infrastructure. I did not run the program without the fanin/out instructure so its unclear whether the concurrent dijkstra or the fanin/out are driving the gains.

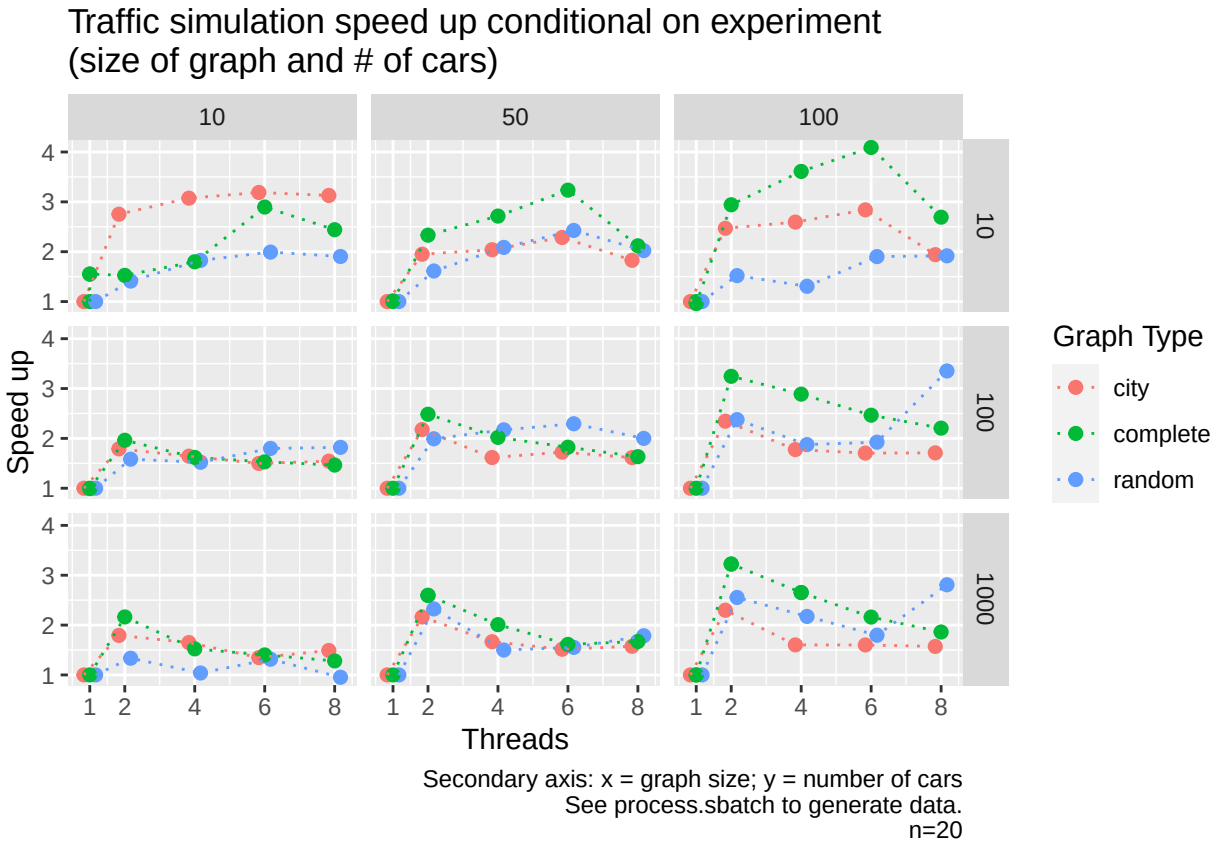
```
gg
```



What limited your speedup? Is it a lack of parallelism? (dependencies) Communication or synchronization overhead? As you try and answer these questions, we strongly prefer that you provide data and measurements to support your conclusions

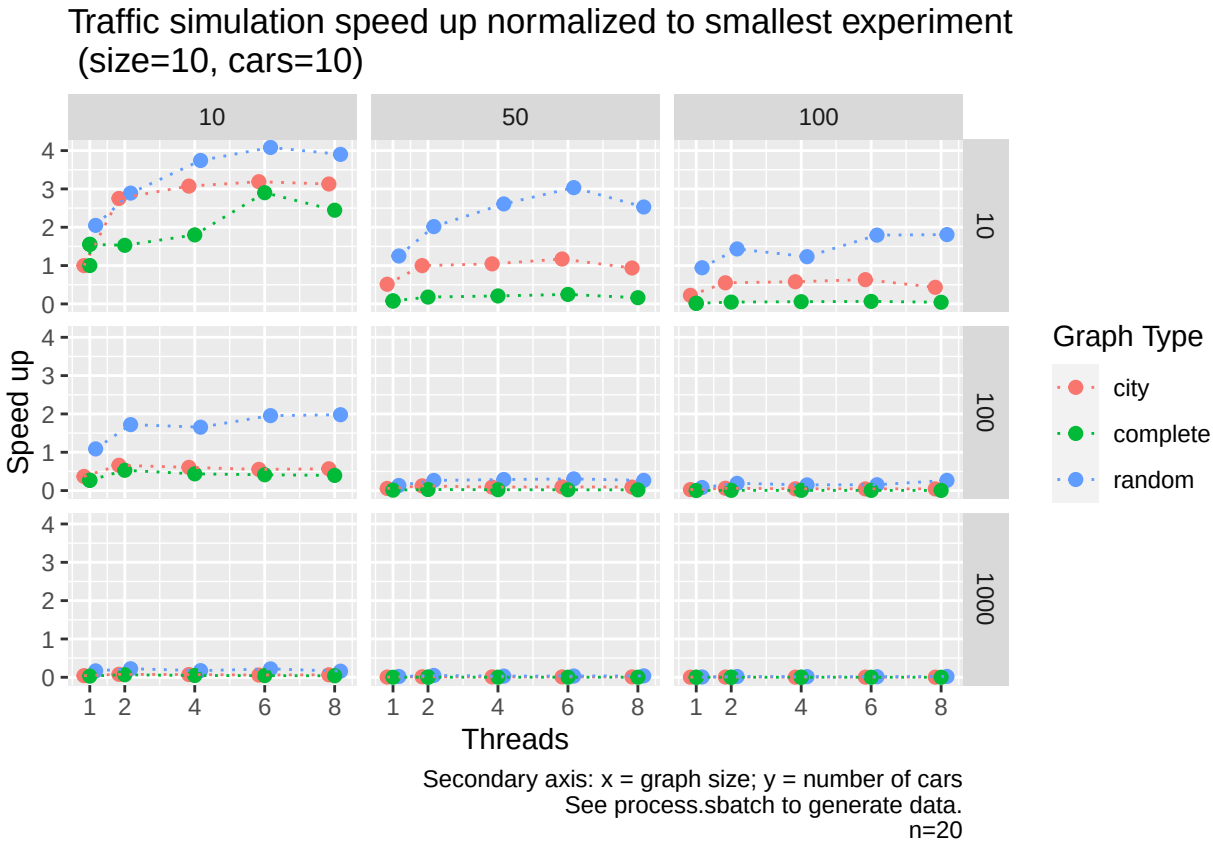
If we condition on each type of experiment, we can see the speed ups from additional threads in the plot below.

```
gg2
```



Being able to run the multiple dijkstras concurrently is allowing speedups 1.5x - 4x when additional threads are added. Similarly, being able to run other experiments while an experiment does a somewhat heavy sequential write is also providing the speedup lift. What is interesting is that the optimal number of threads is mostly defined by changing the number of cars (read calls) in the program. This seems to indicate that Rlocks have high diminishing margins of return. This would indicate that synchronization as oppose to communication is limiting the program. However, Rlocks dimishing margins of return may not be so high after all if we consider the next plot. There we can see that in comparison to the smallest/simplest experiment, the gains from additional threads does not overcome the additional complexity of increasing the size of the graph or the number of cars.

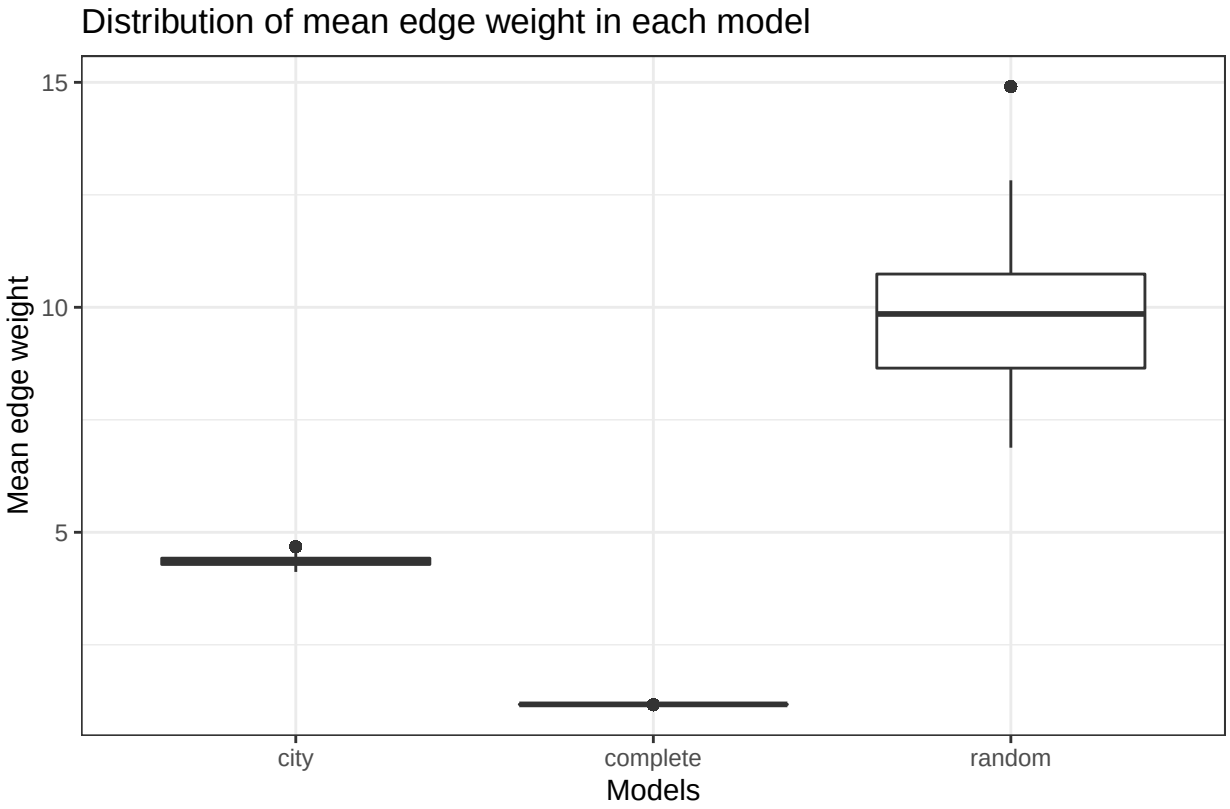
```
gg3
```



Results of experiments

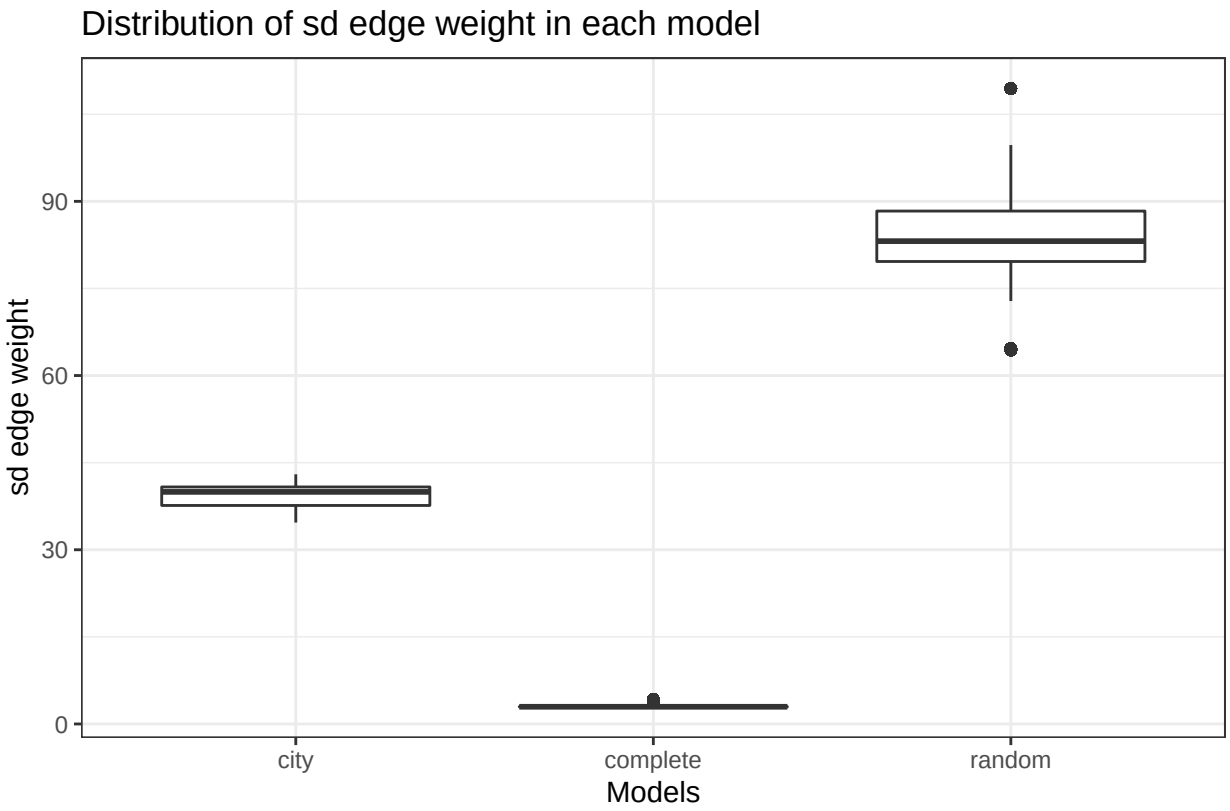
The distribution of the mean edge weights on the graphs suggest that on average, more roads will reduce traffic congestion (duh). However, randomly distribution roads will increase traffic. This make sense as random can create many small cliques where various sections of the graph are only connect with one edge. That one edge (min cut) will create traffic. The cities model, with degrees between 2 - 6, do not have min cuts but still have sections which many cars, (number of cars)/2 cars, must pass. This both increases the mean edge weight and sd edge weight in comparison to the complete graph.

```
gg5
```



See process.sbatch to generate data.

```
gg6
```



See process.sbatch to generate data.