Parallel Functional Programming Project Report

Solving the Traveling Salesman Problem with Genetic Algorithms

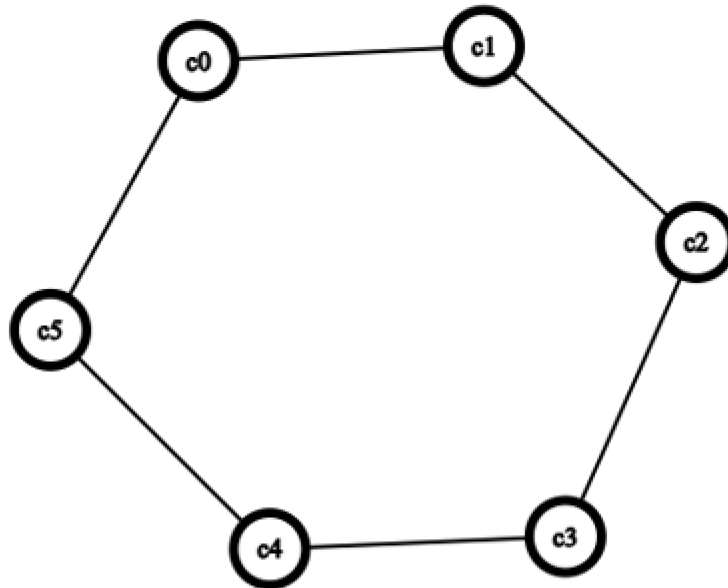
Timothy Johns
Uni: tcj2114

Project Overview

In this project, we used a Genetic Algorithm to find an approximate solution to the NP-hard Traveling Salesman Problem.

Traveling Salesman Problem

In the Traveling Salesman Problem (TSP), we're given `N` cities, and the goal is to find the shortest path that starts and ends at the same city and reaches each of the other cities exactly once along the way. For example, if the 6 cities were laid out like the following image[1], then the optimal route through the cities would look like the circular path `[c0, c1, c2, c3, c4, c5]`.



Genetic Algorithms

A Genetic Algorithm (GA) aims to solve an optimization problem through a process that mimics evolution. In a typical GA, there is a population of `N` solution Candidates. The "fitness score" for each candidate is a numeric representation of how well that candidate solves the optimization problem. The GA will then follow this pseudo code:

```
Unset
1) Create initial population of N random candidates

2) Calculate fitness for each candidate

3) If the candidate with the best fitness so far is good enough, return it

4) Create the next population as an evolution of the current population

5) Repeat steps 2 through 4 until step 3 eventually breaks out of the loop
```

There is a wide variety in the ways that GAs implement the evolution part of step 4. Typically, this will include selecting two "parent" candidates from the current population (with a preference for candidates with higher fitness) and producing a new "child" candidate as a combination of the parents' traits (this is known as "Crossover"). Then some of the traits are further randomized (which is known as "Mutation").

Reference Implementation

The Coding Train on YouTube has a series of programming tutorials that cover solutions to TSP in JavaScript, culminating in a GA-based solution. For this project, we modeled a Haskell solution to TSP on the solution presented in those videos. The original JavaScript code is not written in a functional style, so the code in this project follows the reference implementation in spirit, but is ultimately quite different.

Basic Program Structure

Here is a rough overview of the code, which is shown in an appendix. Note that this illustrates the ideas, but it isn't quite accurate. The Parallelism section below corrects some details about how we sequence the generations.

1. Generate `numCities` random 2D coordinates for the city locations.

[1] Graphs were created using https://csacademy.com/app/graph_editor/

2. Generate a population of `popSize` candidate solutions. A candidate solution is a path through the cities, represented as a permutation of the list `[0..numCities-1]` (i.e. the city indices). This initial population consists of random paths. That is, `[0..numCities-1]` is shuffled to create each candidate path.
3. The score for each candidate path is calculated by first summing up the distance of each edge in the path, and then applying a fitness function to the path lengths. While we could use the path length by itself, the fitness function helps the algorithm to more quickly hone in on optimal paths.
4. Generate a new population of `popSize` candidate solutions. This time, instead of randomly creating paths, we instead base the new population on the previous population.
    a. Select the best few candidates from the previous generation to survive into the new generation. This guarantees that the absolute best candidate ever seen will be a member of the very last generation. This also ensures that the algorithm always makes forward progress, instead of getting lost in the search space.
    b. For the remainder of the new population, randomly select two parents from the previous population, combine them via `crossOver`, and then `mutate` the result. Parent selection, `crossOver`, and `mutate` are discussed below.
5. Repeat steps 3 and 4 until we've evaluated `N` generations.
6. Display some stats about the populations that were evaluated, and show the best path that was ever found.

Parallelism

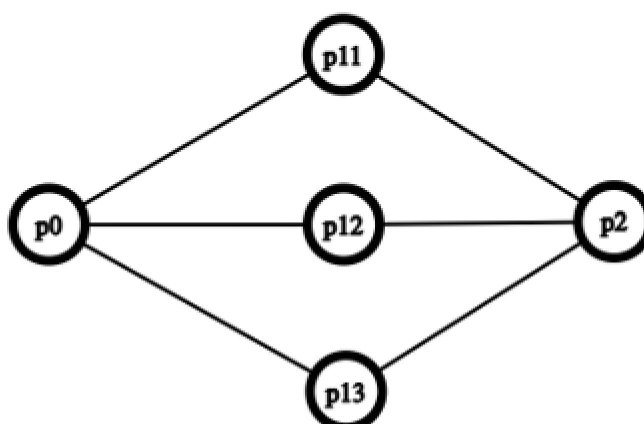
The Basic Program Structure above describes a process that is inherently serial. We need to process each population in order, since the current population (and its Fitness scores) are used to generate the next population. We can visualize this with a graph, where each node represents a population of candidates. Each generation is created from the previous generation, leading to a single sequence of populations, like this:

One way that we considered parallelizing this code was to dive into the code for a single population and:

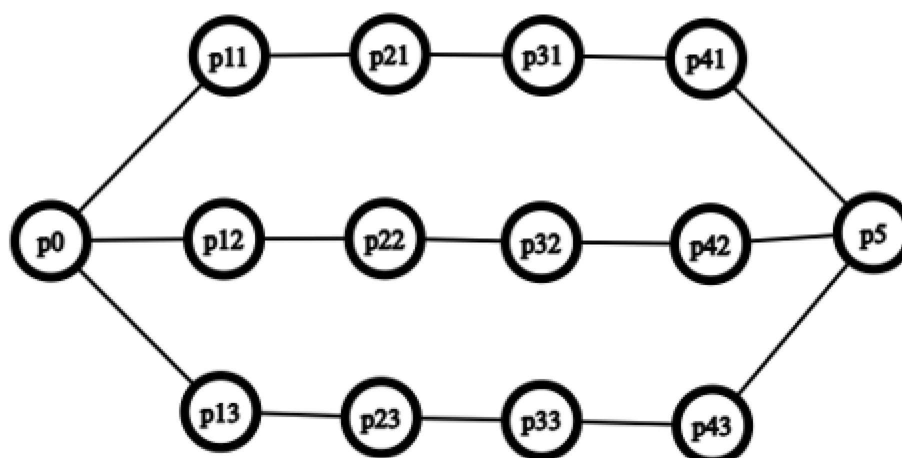
- create `N` sparks to create `N` candidates
- create `N` sparks to calculate fitness for `N` candidates

Both of these operations are quite fast for a single candidate, so the overhead of parallelism was likely to be a bottleneck. Thus, this idea was abandoned before we got to implementing it. Another idea was to partition the `N` candidates into `M` groups, and create `M` sparks that would create `N/M` candidates per spark and calculate their Fitness. This might have worked well for large populations, but we also chose not to implement this.

Instead, we could run `k` generations in parallel, and then merge the results into a single population consisting of the `popSize` best candidates from all of the input populations, like this:



The problem with this is that the overhead of merging the `K` populations is likely to limit the parallel speedup. So, what we chose to do is to run `N` generations per spark, and then merge the final generations together, like this:



We then repeat this process i.e. run `K` more sparks of `N` generations, starting with the merged result from the previous sparks). The program completes after `M` split + merge operations.

Processing a generation consists mostly of creating the candidates and calculating their Fitness. So, the sparks should be able to process their own sequence of generations completely on their own without relying on data from the other sparks. So, we expected this to parallelize quite well. In practice, as we'll show below, regardless of how many cores we ran on, or how many sparks we created, or how many generations were processed by each spark, we were only able to achieve a speedup between 3x and 4x.

Implementation Details

Parent Selection

Parents are selected for reproduction based on their Fitness score. In particular, if a Candidate has score `X` and the total score for the whole population is `Y`, then the Candidate will be selected as the next parent with likelihood `X / Y`.

Crossover

A child Candidate is produced from parent Candidates A and B by starting with some random contiguous subset of A's path, and then filling in the rest of the cities in the order that they appear in B's path. This guarantees that the child Candidate's path still covers all of the cities exactly once and inherits some similarity from both parents.

To determine which cities in B to skip, we store the subset of cities inherited from A in a set, so that we can do fast lookups. We initially used `Data.Set` for this, which worked well. Then, we switched to `IntSet`, which was even faster.

In a standalone testbed, we tried writing the `crossOver` function using `Data.Vector` instead of lists. In the testbed, the vector-based approach was consistently about twice as fast as the list-based approach. Strangely, when this was integrated into our program, the program actually got slower.

Mutation

After creating a child Candidate via `crossOver`, the child Candidate's path can be Mutated by randomly swapping neighboring cities in the path. This was originally implemented with lists. We would loop through the list, and if a random `Double` was less than the `mutationRate`, then the current city in the list would be swapped with the next city. This was quite slow, and the profiler (i.e. `stack run --profile -- +RTS -p`) showed that the `mutate` function accounted for more than half of the program's runtime.

In a separate testbed, we wrote a version of `mutate` that creates a `Data.Vector.Unboxed.Mutable.MVector` and then swaps elements in-place. In the testbed, this was about twice as fast as the list-based implementation. However, when we brought this into the program, it resulted in a much more modest improvement (about half as much as expected).

This also had the unfortunate side effect of making it take longer for the profiler to run. So, even though the program was noticeably faster with the `MVector` approach, the profiler made it seem slower, and it made it seem like `mutate` accounted for a larger fraction of the program's runtime.

Stop Condition

As described in the Parallelism section above, the program splits into `K` sparks `M` times, and each spark runs `N` generations. So, in a given run, we will evaluate `K*M*N` populations in sparks, plus the initial random population. There are also `M` populations from merging together the populations from sparks, but these do not introduce any new candidates.

This allows us to guarantee that for a given set of `K`, `M`, `N`, `popSize`, and `numCities`, different runs will do the same amount of work, regardless of how many cores are used. This is important, as it allows us to compare apples to apples when calculating speedup by comparing the time to run with `-N1` and `-N<#>`.

Handling Randomness

This program requires randomness when generating the city locations and when generating the candidate paths. We use `System.Random.StdGen` to generate these random values.

We have the following requirements:

- From one run to the next (for the same set of inputs), we must see the exact same sequence of random values. This way, we can rule out random number generation as a factor in performance differences between runs. This also allows us to check that the final path that is returned is the same regardless of how many cores we use.
- Parallel sparks should see different sequences of random values than each other, and the sparks should not race with each other to determine these sequences.

To achieve this, we start the program with a hard-coded random seed. Then, in any sequential code that generates a random number, we keep track of the new state of the random number generator, to avoid re-generating the same random value twice. When generating sparks, we first split the main thread's random number generator into `K` generators (one per spark). This way, no two sparks see the same sequence of random values, and from one run to the next, a given spark sees the same sequence of random values. Finally, when the sparks complete and control returns to the main thread, it uses the final state of the first spark's generator, and discards the other sparks' generators.

Performance

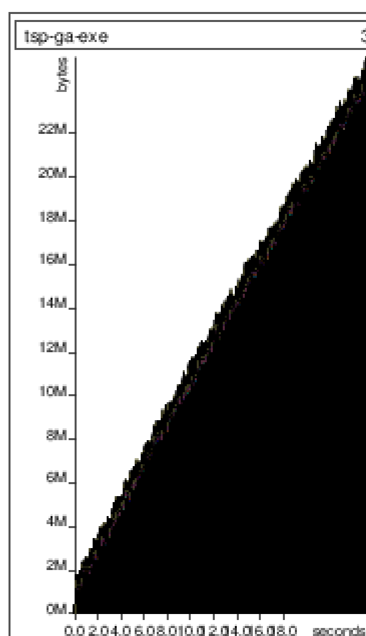
Basic Sequential Improvements

Throughout development, there were a lot of minor improvements that can be seen in the `git log`. These improvements looked like:

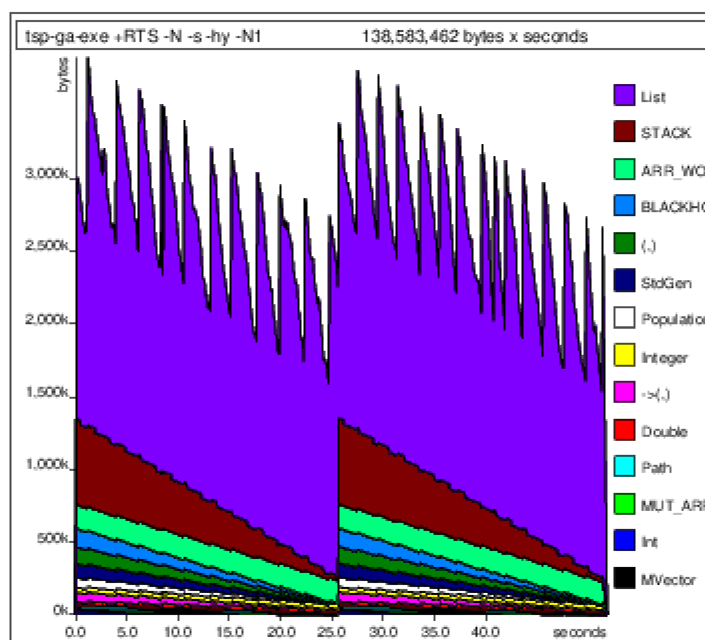
- Pre-compute a value, instead of re-computing the value repeatedly. We used this for calculating path lengths by pre-computing the distance between each pair of cities and then storing that in a `Vector` for fast access later.
- Pass around known values, like lengths of lists, instead of calculating those values as needed. This came up a couple times for well known list lengths, like `numCities` and `popSize`. This typically coincided with code organization improvements, like grouping together information about a population in a `Population` record, instead of passing around paths and scores as standalone variables.
- Using `Data.Vector` instead of `Data.List` for fast random access. We already described how this was used for storing pre-computed distances between cities and for speeding up `mutate`.

Memory Improvements

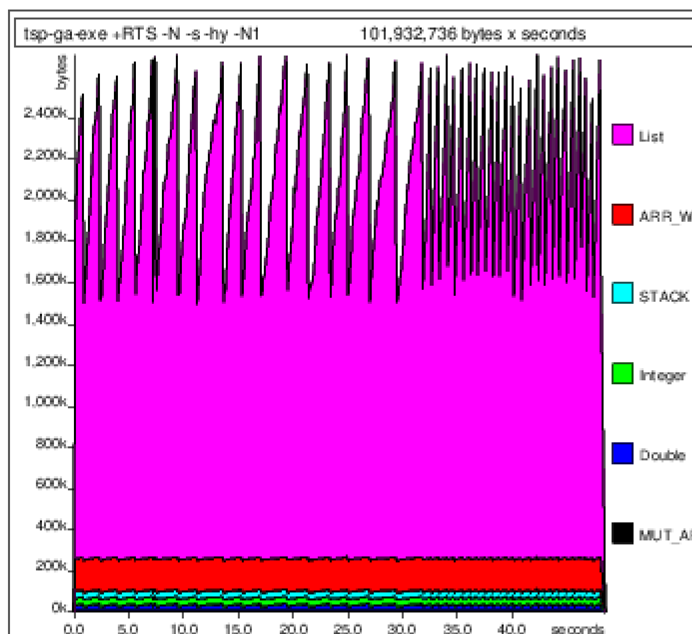
We can generate a memory profile by adding the `-h` RTS option and then convert that to a ps image with `hp2ps`, and finally view it with a tool like `evince` or `gv`. Once we had the parallel implementation working, the memory profiler showed that memory was growing throughout the execution of the program, which indicated a memory leak:



This happened because we returned a list of populations to `main` for it to calculate and print some statistics, like the minimum path length for every `N`th generation. We fixed this by returning just the stats instead of the full populations. But now the issue was that we generated a bunch of thunks at the beginning of each spark, so there was a big jump in memory followed by a gradual decline as the generations were processed:

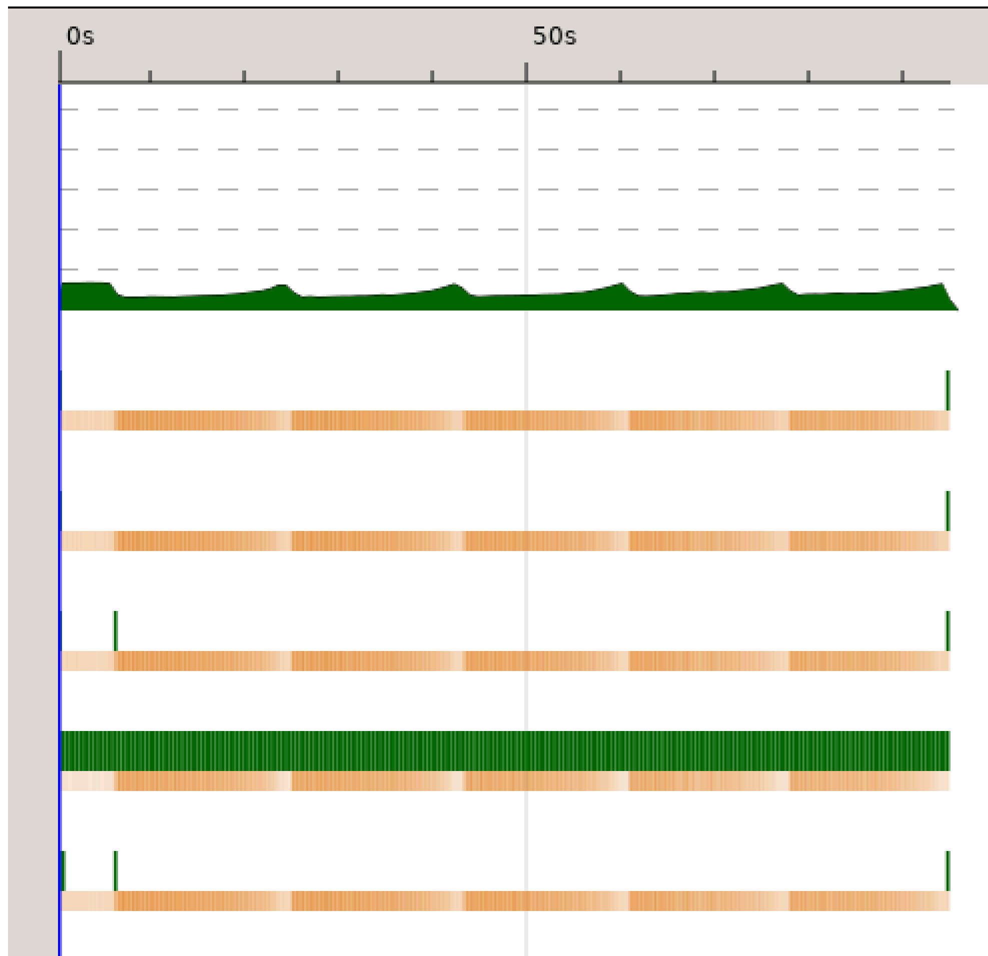


Adding `deepseq` after each generation fixed this, so now the memory usage is a flat plateau (with a spike for each generation) throughout each spark's lifetime:



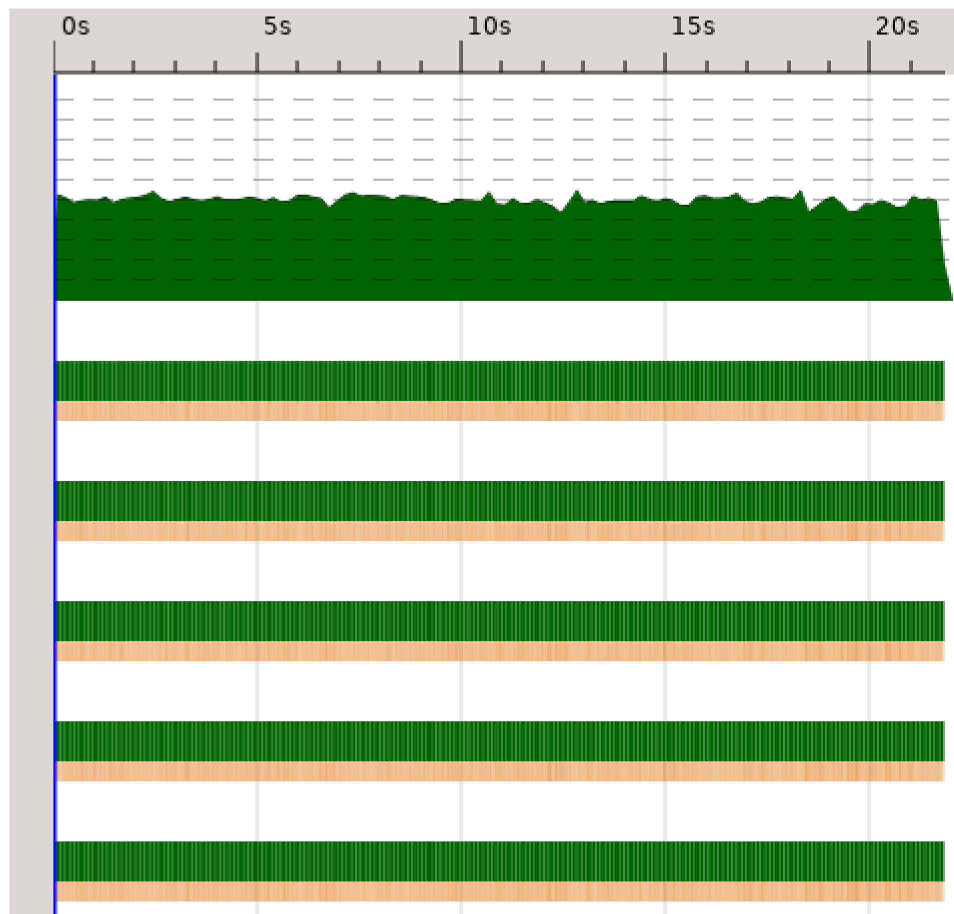
Parallel Improvements

Our first attempt to generate sparks involved using the `parList rseq` strategy. This caused the program to slow down by 2x. Looking at Threadscope, it's easy to see why:



Only 1 thread was doing any work. This was because each of the sparks would create thunks to describe the work that they planned to do and then return those thunks to the main thread, which would actually do the work.

This was solved by using `parList rdeepseq` instead, to force the sparks to evaluate all of the generations before returning. Afterwards, Threadscope showed that all of the threads were being used, and we went from a ~2x slowdown to a ~3x speedup:



Spark Stats

In all runs that we've checked, all of the sparks have converted except for `M` sparks, which fizzle:

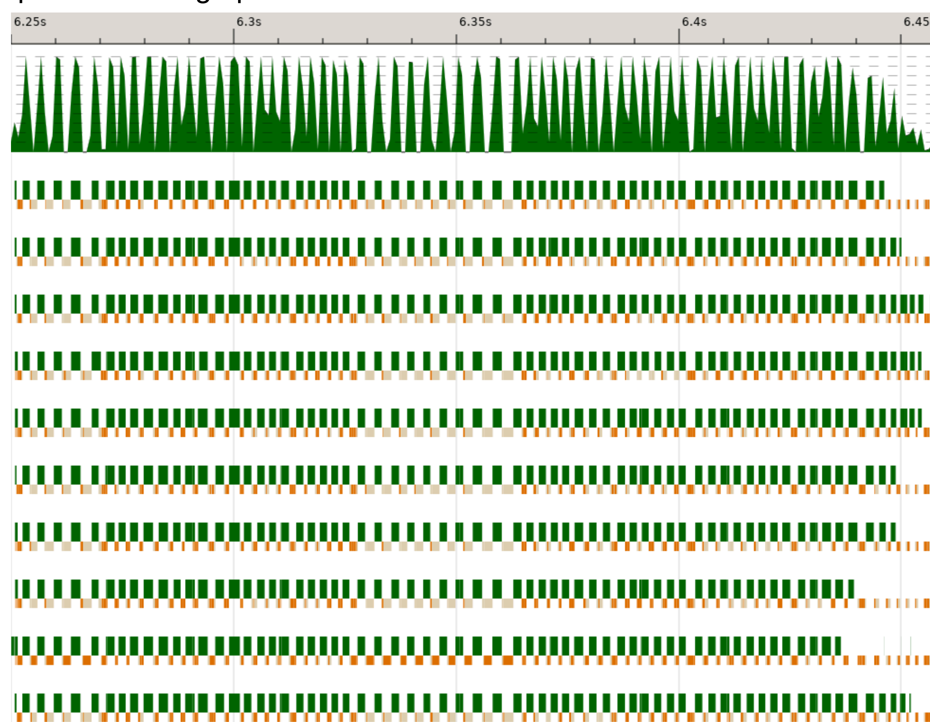
Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw e

HEC	Total	Converted	Overflowed	Dud	GCed	Fizzled
Total	20000	19980	0	0	0	20
HEC 0	0	1892	0	0	0	0
HEC 1	0	2056	0	0	0	0
HEC 2	0	2066	0	0	0	0
HEC 3	1000	2041	0	0	0	1
HEC 4	0	1955	0	0	0	0
HEC 5	0	1980	0	0	0	0
HEC 6	4000	2002	0	0	0	4
HEC 7	0	1997	0	0	0	0
HEC 8	15000	1996	0	0	0	14
HEC 9	0	1995	0	0	0	1

This is great already, so we didn't need to put any effort into improving the spark stats.

GC Tuning

Threadscope shows that GC represents a large portion of the runtime:



The GC stats agree with this point. The elapsed time and amount of data copied are both high and the parallel GC work balance is low:

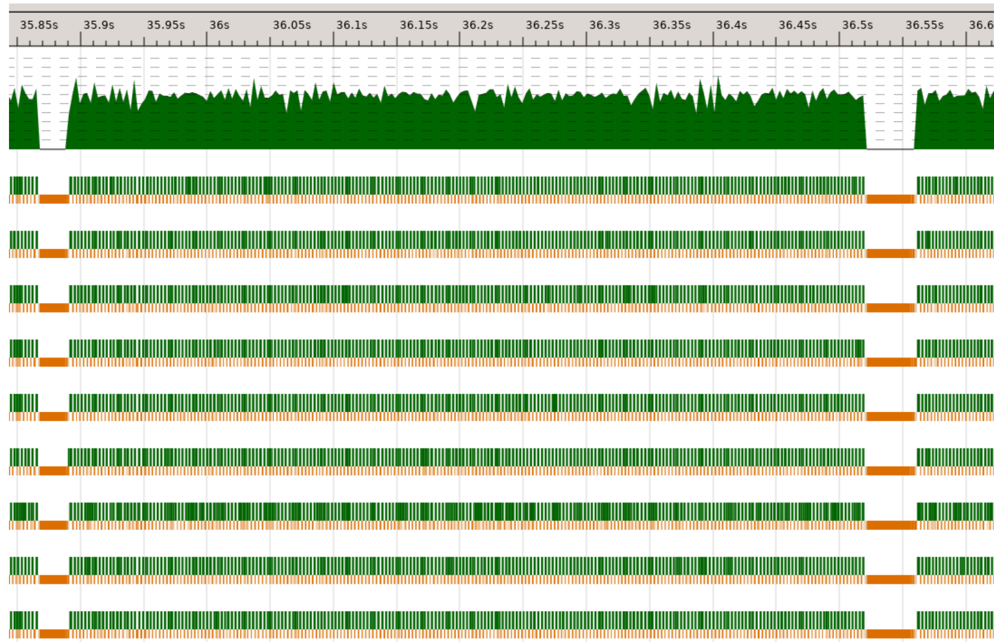
Time | Heap | GC | Spark stats | Spark sizes | Process info | Raw events

Copied during GC: 45.7 GiB 49,067,071,776 bytes
Parallel GC work balance: 38.00% (serial 0%, perfect 100%)

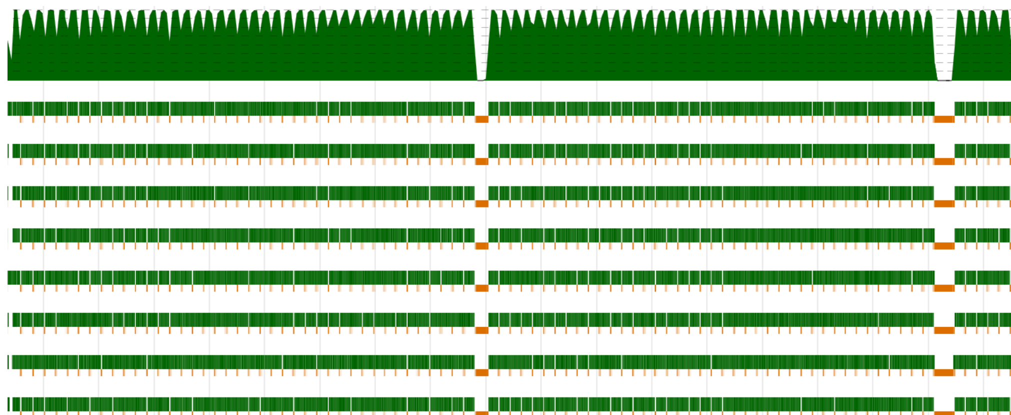
Generation	Collections	Par collections	Elapsed time	Avg pause	Max pause
GC Total	10292	8014	11.20s	0.0011s	0.0445s
Gen 0	7978	7978	7.90s	0.0010s	0.0032s
Gen 1	36	36	0.69s	0.0192s	0.0445s

I tried tuning the garbage collector in the hopes that it would decrease the overall amount of time spent doing GC. In particular, I tried some of the tips from [this thread](). All of the hints either had no effect or made the program slower.

However, I did find that one setting, `-A32m`, made the program almost 10% faster. But to understand what this did, here was the Threadscope output before using `-A32m`. Note how consistently the GC is running:



Then, here was the Threadscope output after adding `-A32m`. Note how much space is between the orange lines now:



Unfortunately, this speedup only applied for certain inputs. For one particular set of inputs, this made the program slower with `-N1` and faster with `-N10`. I don't know why it wasn't consistent in its benefits. And since it wasn't consistent, I decided not to continue using this setting.

Another GC-related idea that we tried was to decrease the size of the program args. The idea was to decrease how much work is done, and therefore how much memory is used and GC'd. When I went really small (e.g. `stack run 10 10 10 10 10`), then the program completed in less time than the granularity of the elapsed time reported by `+RTS -s`. I then gradually increased the input sizes until the elapsed time started to increase, and even then there was still too much GC as seen in Threadscope.

This program is designed to create and then discard large batches of candidates, so it's not surprising that there is a lot of GC. I don't see an obvious way to cut down on the amount of GC performed other than to decrease the program's memory footprint.

Amdahl's Law

Amdahl's Law provides a formula for the relationship between 3 things:

- `S` – speedup
- `P` – fraction of the program that can be run in parallel
- `N` – number of cores

The following formula represents the ideal speedup, `S`, as a function of `P` and `N`:

Unset
```
              1
S = -----------------
    (1 - P) + (P / N)
```

For the first screenshot in GC Tuning, Threadscope told us that about 22% of the program was spent on GC. So, with 10 cores, an upper bound for the ideal speedup would be `S = 1 / (1-0.78+0.78/10) = 3.36`.

This is interesting because it shows that the 3x-4x speedup that we're seeing is actually pretty close to ideal. If we want to speed up the program, we need to cut down the amount of time spent on GC.

Also, in the screenshot that we were just looking at, some of the threads finish their work before others. As a result, about 5-10% of the program time is spent with idle cores. With 22% spent on GC and 5% with unused cores, Amdahl drops our ideal speedup to `2.92`.

This shows an important point about the program args. If we pass in args such that there are the same number of sparks as cores at a time, then we should expect to see this pattern of unused cores. Worse yet, if the number of sparks is similar, but slightly different than the number of cores, then we'll have unused threads for even longer. One thing we can do to address this issue is to generate significantly more sparks. Doing so does actually get rid of that idle time, as seen in the second and third screenshots in GC Tuning.

In the third screenshot, Threadscope claimed that the GC accounts for 11% of the program's runtime. Amdahl says that this yields an ideal speedup of 4.5x with 10 cores. Since the speedup was actually 3.7x, that means that this run had an extra ~5% of time with unused cores.

Runtime Environment

All of the data in this report was generated from a laptop running Windows 11 with WSL2. The processor is 11th Gen Intel Core i7-11800H. [According to Intel](), this has 8 cores with 2 hardware threads per core, yielding a total of 16 hardware threads.

We also tried a few runs in Windows native (using Powershell) and in a GCP VM. In both cases, the results were really similar to the WSL2 results. Windows native was a couple percent faster than WSL2 most of the time. This was a pretty small improvement, so we stuck with WSL2.

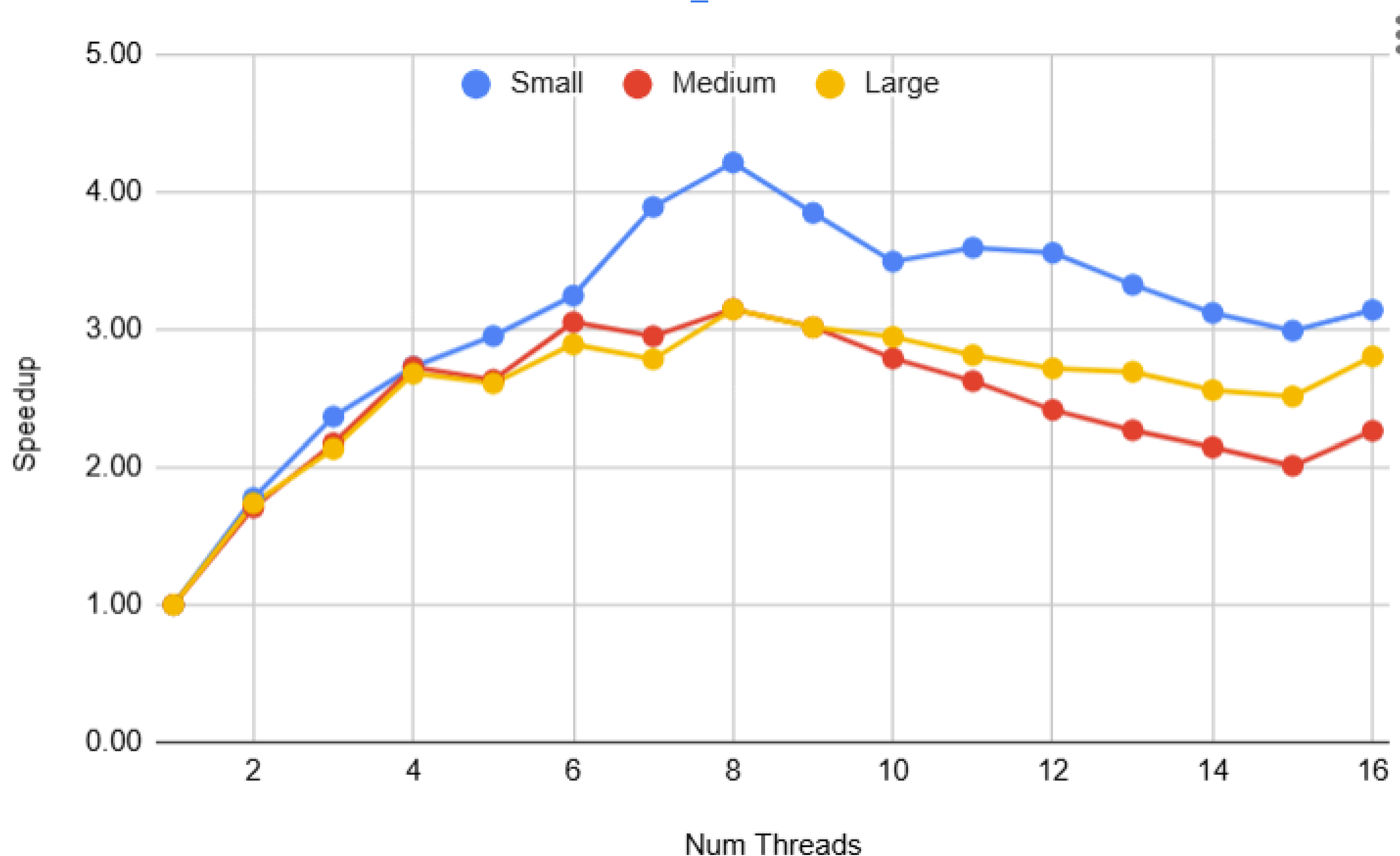
Results

Performance Analysis

We define the following 3 sets of inputs:

- `small` – cities: 10 cities, pop size: 50, K: 32, M: 100, N: 100
- `medium` – cities: 100 cities, pop size: 100, K: 16, M: 50, N: 50
- `large` – cities: 1000 cities, pop size: 50, K: 16, M: 10, N: 10

We then run each set of inputs with `-N<cores>` for 1 core up to 16 cores and plot the speedup:



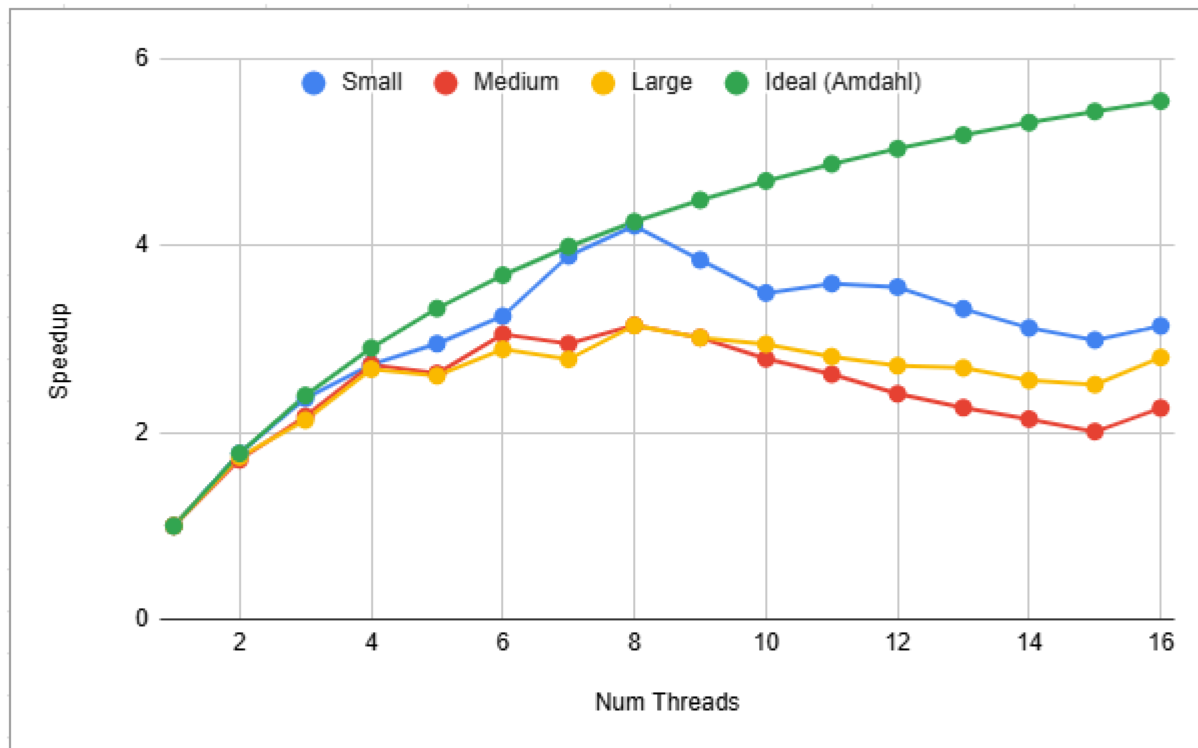
Here are some observations based on the graph:

- The speedup tends to be in the 2x to 4x range.
- The set of inputs does not have a huge effect on the shape of the speedup graph (especially up to 6 cores). Beyond 6 cores, there is a bit of variation in the amount of speedup, though not the shape of the curve.
- For all input sets, the overall best speedup is observed with 8 cores and there are local peaks in speedup at 4 and 16 cores. This is likely due to the number of sparks being multiples of 4, 8, and 16, so we observe particularly good load balancing with this many cores.

Amdahl's Law Revisited

Earlier, we used Amdahl's law to approximate the ideal speedup. Now, we know the actual speedup values, so let's use this to approximate how much of the program can be run in parallel. If we pick the best speedup value from the above plot for 2 cores, then we can plug it into the equation. `S = 1.78` and `N = 2`. So, `1.78 = 1 / ((1 - P) + P / 2)`. Solving for `P` yields `P = 0.874`.

Now we can add a line to our speedup plot that includes the ideal speedup based on this value of `P`.

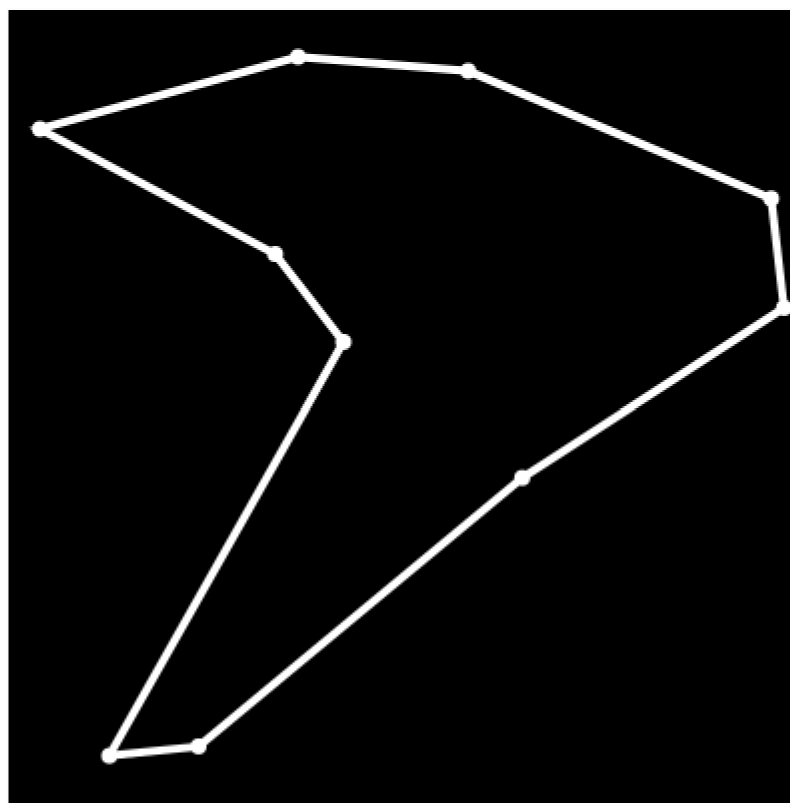


For 1-4 cores, all 3 lines stay pretty close to ideal. Beyond that, the `medium` and `large` input sets plateau, so they stop tracking the ideal speedup line. The `small` input set actually continues to do pretty well all the way up to 8 cores, even managing to match the ideal speedup for 8 cores almost perfectly. After that, the performance drops off and we see it diverge from the ideal line.

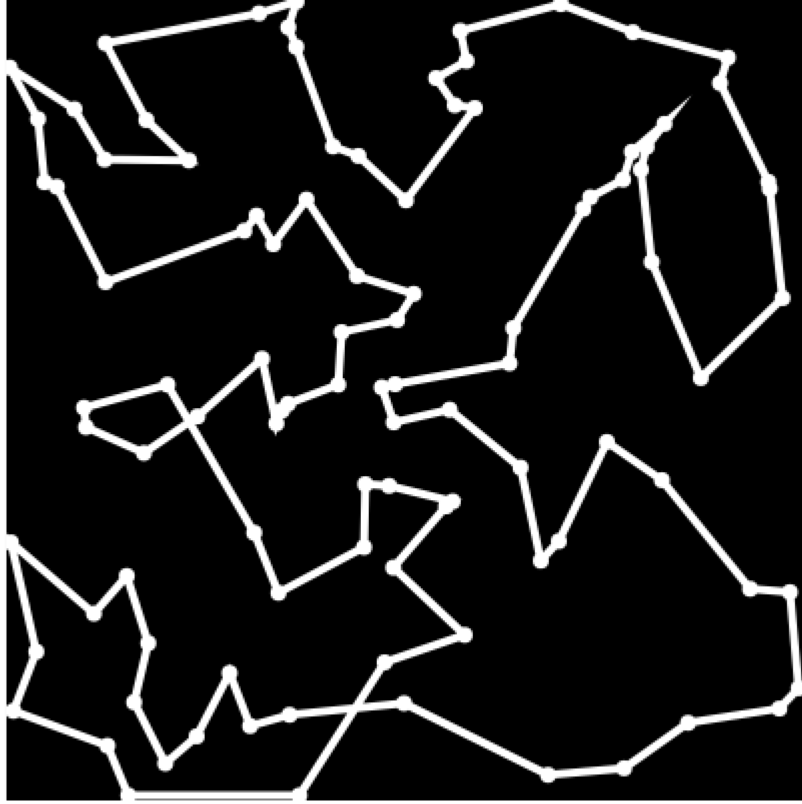
Algorithm Correctness

So far, we've just implied that the program does what it claims to do, and have focused on performance instead. But, let's take a look at the best paths that we found for each of the 3 input sets.

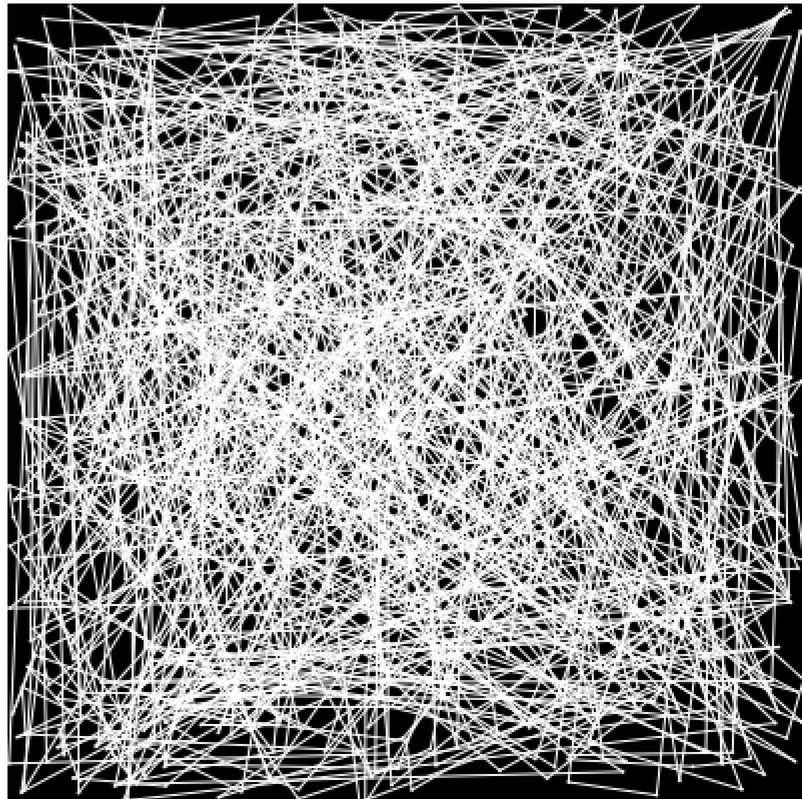
For the `small` input set, we get this boomerang-like shape, which is probably the absolute best path for this set of cities.



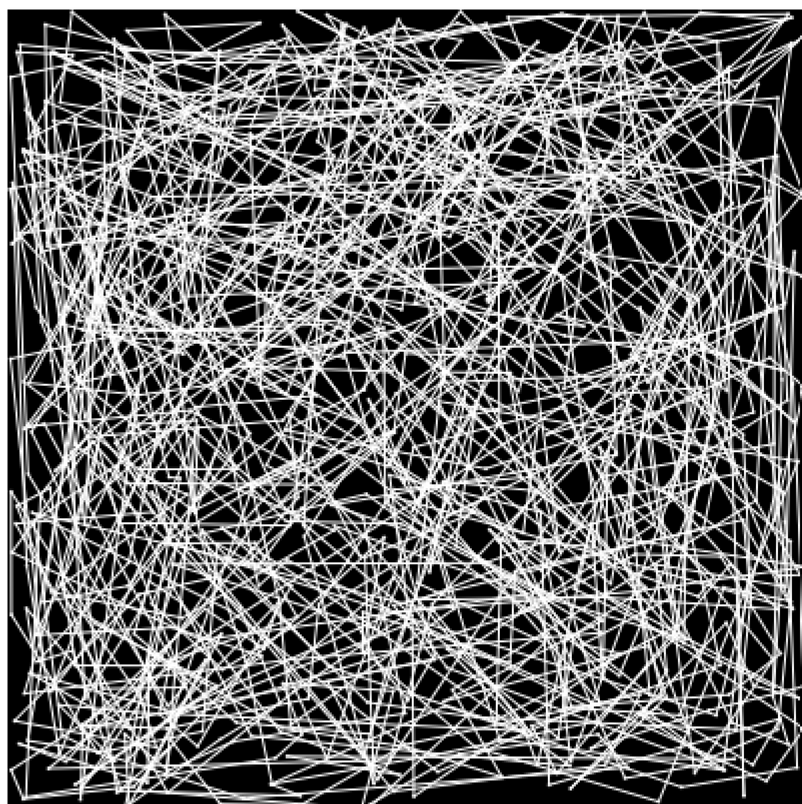
For the `medium` input set, it starts to get more complicated. The path is pretty good. There aren't any crazy edges between cities that are particularly far apart. However, there are two places where edges intersect, which could easily be improved by a person.



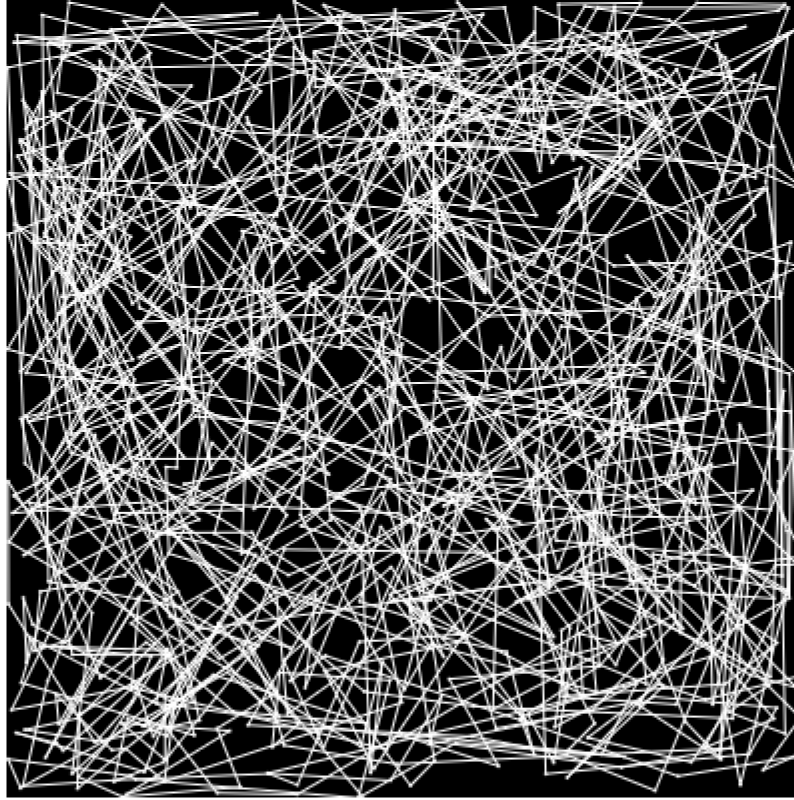
Finally, the `large` dataset is wild. The path lengths do indicate that this is better than random (~200k vs ~140k). However, there doesn't appear to be any rhyme or reason to the path. It's just a lot of intersecting lines.



If we let this run for more generations (`N=100` and `M=100`) instead, we get a slightly more organized path, which has a better path length (~85k).



And if we let it run even longer (`N=100` and `M=500`), we get this path, which has a length of ~68k. It does look slightly more organized (e.g. fewer intersections) than the previous path.



Next Steps

- Since GC seems to be the main opponent to our parallelization attempts, and since Threadscope shows that we are actually generating many GB of temporary data, we should look into ways to be more conservative with how much memory we use. This should cut down on the time spent on GC. This could also allow the program to spend more time working with data in caches instead of RAM.
- We use a lot of lists in the program, which are not compact in memory. This likely hurts us in multiple ways (e.g. repeated allocations for each list element are expensive and pointer chasing takes time to page memory into caches). We know all of the list lengths up front, so we could use vectors most places that we currently use lists to cut down on the number of allocations, to use less memory, and to improve data locality.
- Figure out why the `Vector` based version of `crossOver` didn't work well in our project, while it had a 2x improvement over the current implementation in a dedicated testbed.
- Figure out how to speed up `mutate`, since the profiler claims that we spend more than half of the program time in `mutate`.
- Try out other random number generators to see if they can speed up the program.

Appendix: Source Code

app/Main.hs

```haskell
module Main (main) where

import Lib
import System.Environment (getArgs)
import System.Exit (die)
import System.Random (mkStdGen)

main :: IO ()
main = do
  args <- getArgs
  case map read args of
    [numCities, popSize, k, n, m] -> do
      let r               = mkStdGen 12345
          (cities, r')    = createCities numCities r
          pathLength      = calcPathLength cities
          (pop, r'')      = initialPopulation cities popSize pathLength r'
          popStats        = runGA pop m k n pathLength r''

      putStrLn "Generation | Min Path Length     | Average Path Length    | Max Path Length"
      putStrLn "-----------|--------------------|----------------------|---------------------"
      mapM_ (\(i, (PopulationStats _ minLen maxLen avgLen)) ->
               putStrLn $ show i      ++ "\t    |  "
                       ++ show minLen ++ "\t| "
                       ++ show avgLen ++ "\t| "
                       ++ show maxLen
            ) $ zip [0,n..] popStats

      putStrLn $ toJsonStr (TspResult cities (last popStats))
```

```
    _ -> do
      die $ "Usage: stack run <numCities> <popSize> <k> <n> <m> -- +RTS -s -N<cores>\n\n"
         ++ "numCities - The number of cities to traverse\n"
         ++ "popSize   - The number of candidate paths that are evaluated during each generation\n"
         ++ "k         - The number of branches of generations to evaluate in parallel\n"
         ++ "n         - The number of generations to evaluate within a branch before merging with the other
branches\n"
         ++ "m         - The number of times to split into parallel branches and then merge the results\n"
         ++ "cores     - The number of cores to run the program with\n"
```

src/Lib.hs

```
module Lib
    ( PopulationStats(..)
    , TspResult(..)
    , toJsonStr
    , createCities
    , initialPopulation
    , runGA
    , calcPathLength
    ) where

import Control.DeepSeq (deepseq, NFData, rnf)
import Control.Monad.ST
import Control.Parallel.Strategies (using, parList, rdeepseq)
import Data.List (sortOn)
import qualified Data.IntSet as Set (fromList, notMember)
import Data.Vector.Unboxed (unsafeFreeze, unsafeThaw, (!))
import qualified Data.Vector.Unboxed as V (fromList, toList)
import Data.Vector.Unboxed.Mutable (MVector, swap)
import List.Shuffle (shuffle)
import System.Random (randomR, split, StdGen)

type Index = Int
type Path = [Index]
type Location = (Double, Double)

data Cities = Cities
  { cCount :: Int
  , cCoords :: [Location]
  , cDistMap :: (Index -> Index -> Double)
  }

data Population = Population
  { pPopSize :: Int
  , pPathSize :: Int
  , pPaths :: [Path]
  , pPathLengths :: [Double]
  , pScores :: [Double]
  , pTotalScore :: Double
  }

mkPopulation :: Int -> Int -> [Path] -> [Double] -> Population
mkPopulation popSize pathSize paths pathLengths = Population popSize pathSize paths pathLengths scores (sum
scores)
  where scores = map fitness pathLengths

instance NFData Population where
  rnf (Population popSize pathSize paths pathLengths scores totalScore) =
    rnf popSize `seq`
    rnf pathSize `seq`
    rnf paths `seq`
    rnf pathLengths `seq`
    rnf scores `seq`
```

```haskell
    rnf totalScore

data PopulationStats = PopulationStats
  { sBestPath :: Path
  , sMinPathLength :: Double
  , sMaxPathLength :: Double
  , sAvgPathLength :: Double
  }

mkPopulationStats :: Population -> PopulationStats
mkPopulationStats pop = PopulationStats bestPath minPathLength maxPathLength avgPathLength
  where minPathLength = minimum $ pPathLengths pop
        (maxPathLength, bestPath) = maximum $ zip (pPathLengths pop) (pPaths pop)
        avgPathLength = (sum $ pPathLengths pop) / (fromIntegral $ pPopSize pop)

data TspResult = TspResult
  { tspCities :: Cities
  , tspFinalStats :: PopulationStats
  }

toJsonStr :: TspResult -> String
toJsonStr (TspResult cities (PopulationStats bestPath _ bestPathLength _)) =
    "{\n  \"cities\": " ++ show [[x,y] | (x,y) <- cCoords cities] ++
    ",\n  \"pathIndices\": " ++ show bestPath ++
    ",\n  \"score\": " ++ show bestPathLength ++
    "\n}"

createCities :: Int -> StdGen -> (Cities, StdGen)
createCities n r = (Cities n coords distMap, last (r:rs))
  where (coords, rs) = unzip $ take n $ drop 1 $ iterate randomCoord ((-1,-1), r)
        randomCoord (_, r') = randomR ((0,0), (400,400)) r'
        distMap = mkDistanceMap n coords

mkDistanceMap :: Int -> [Location] -> (Index -> Index -> Double)
mkDistanceMap n coords = distanceMap
  where distances = V.fromList [
                      sqrt $ (x1-x0)*(x1-x0) + (y1-y0)*(y1-y0) |
                      (x0,y0) <- coords, (x1,y1) <- coords
                    ]
        distanceMap i j = distances ! (i*n + j)

initialPopulation :: Cities -> Int -> (Path -> Double) -> StdGen -> (Population, StdGen)
initialPopulation _ 0 _ r = (mkPopulation 0 0 [] [], r)
initialPopulation cities popSize pathLength r = (mkPopulation popSize pathSize paths pathLengths, last rs)
  where pathSize    = cCount cities
        (paths, rs) = unzip $ take popSize $ drop 1 $ iterate (uncurry shuffle) ([0..pathSize-1], r)
        pathLengths = map pathLength paths

nextGeneration :: Population -> (Path -> Double) -> StdGen -> (Population, StdGen)
nextGeneration pop pathLength r = (combinedPop, r')
  where popSize = pPopSize pop
        (newPop, r') = nextGeneration' pop (popSize-20) pathLength r
        bestPop = bestPaths pop 20
        paths = pPaths newPop ++ pPaths bestPop
        pathLengths = pPathLengths newPop ++ pPathLengths bestPop
        scores = pScores newPop ++ pScores bestPop
        totalScore = pTotalScore newPop + pTotalScore bestPop
        combinedPop = pop { pPaths = paths, pPathLengths = pathLengths, pScores = scores, pTotalScore =
totalScore }

nextGeneration' :: Population -> Int -> (Path -> Double) -> StdGen -> (Population, StdGen)
nextGeneration' _ 0 _ r = (mkPopulation 0 0 [] [], r)
nextGeneration' pop n pathLength r = (mkPopulation n (pPathSize pop) paths pathLengths, last rs)
  where (paths, rs) = unzip $ take n $ drop 1 $ iterate (\(_, r') -> spawnPath pop r') ([], r)
```

```haskell
        pathLengths = map pathLength paths

runNGenerations :: Population -> Int -> (Path -> Double) -> StdGen -> (Population, StdGen)
runNGenerations pop 0 _ r = (pop, r)
runNGenerations pop n pathLength r = pop' `deepseq` runNGenerations pop' (n-1) pathLength r'
  where (pop', r') = nextGeneration pop pathLength r

runKEvolutionaryBranches :: Population -> Int -> Int -> (Path -> Double) -> StdGen -> (Population, StdGen)
runKEvolutionaryBranches pop 0 _ _ r = (pop, r)
runKEvolutionaryBranches pop k n pathLength r = (pop', head rs')
  where rs = take k $ iterate (fst . split) r
        (pops, rs') = unzip (map (runNGenerations pop n pathLength) rs `using` parList rdeepseq)
        pop' = mergePopulations pops

runGA :: Population -> Int -> Int -> Int -> (Path -> Double) -> StdGen -> [PopulationStats]
runGA pop m k n pathLength r = popStats
  where popStats = map mkPopulationStats
                 $ fst $ unzip $ take (m + 1)
                 $ iterate (\(pop', r') -> runKEvolutionaryBranches pop' k n pathLength r') (pop, r)

mergePopulations :: [Population] -> Population
mergePopulations pops = bestPaths (Population (-1) pathSize paths pathLengths scores (-1)) popSize
  where popSize = pPopSize $ head pops
        pathSize = pPathSize $ head pops
        paths = concatMap pPaths pops
        pathLengths = concatMap pPathLengths pops
        scores = concatMap pScores pops

bestPaths :: Population -> Int -> Population
bestPaths (Population _ pathSize paths pathLengths scores _) n = Population n pathSize paths' pathLengths'
scores' (sum scores')
  where (paths', pathLengths', scores') = unzip3 $ take n $ sortOn (\(_, _, score) -> negate score)
                                        $ zip3 paths pathLengths scores

spawnPath :: Population -> StdGen -> (Path, StdGen)
spawnPath pop r = mutate child pathSize r'''
  where pathSize = pPathSize pop
        (parent1, r') = pickOne pop r
        (parent2, r'') = pickOne pop r'
        (child, r''') = crossOver parent1 parent2 pathSize r''

fitness :: Double -> Double
fitness pathLen = 1 / (1 + (pathLen ** 16))

calcPathLength :: Cities -> Path -> Double
calcPathLength _ [] = 0
calcPathLength _ [_] = 0
calcPathLength cities path = pathLength' 0 path
  where distanceMap = cDistMap cities
        pathLength' acc [] = acc
        pathLength' acc [i] = acc `seq` acc + distanceMap i (head path)
        pathLength' acc (i:j:path') = acc' `seq` pathLength' acc' (j:path')
          where acc' = acc + distanceMap i j

pickOne :: Population -> StdGen -> (Path, StdGen)
pickOne (Population _ _ paths _ scores totalScore) r = (pickOne' paths scores 0, r')
  where (randVal, r') = randomR (0, totalScore) r
        pickOne' [] _ _ = []
        pickOne' _ [] _ = []
        pickOne' (path:paths') (score:scores') accScore = if accScore + score >= randVal
                                                          then path
                                                          else pickOne' paths' scores' (accScore + score)

crossOver :: Path -> Path -> Int -> StdGen -> (Path, StdGen)
```

```haskell
crossOver p1 p2 pathSize r = (newPath, r')
  where (p1', r') = randomSublist p1 pathSize r
        p1'' = Set.fromList p1'
        p2' = filter (`Set.notMember` p1'') p2
        newPath = p1' ++ p2'

randomSublist :: Path -> Int -> StdGen -> (Path, StdGen)
randomSublist path pathSize r = (take (j-i+1) $ drop i path, r'')
  where (i, r') = randomR (0, pathSize - 1) r
        (j, r'') = randomR (i, pathSize - 1) r'

mutate :: Path -> Int -> StdGen -> (Path, StdGen)
mutate path pathSize r = runST $ do
  path' <- unsafeThaw (V.fromList path)
  r' <- mutateInPlace path' 0 pathSize r
  path'' <- unsafeFreeze path'
  return (V.toList path'', r')

mutateInPlace :: (MVector s Int) -> Int -> Int -> StdGen -> ST s StdGen
mutateInPlace _ i pathSize r | i >= pathSize - 1 = do return r
mutateInPlace p i pathSize r = do
  let (randVal, r') = randomR (0 :: Double, 1) r
      j = if randVal < 0.01 then (i+1) `mod` pathSize else i
  swap p i j
  mutateInPlace p (i+1) pathSize r'
```