Introduction
The Travelling Salesman Problem (TSP) is one of the best known NP-hard problems, that means that no exact algorithm can solve it in polynomial time. The method that would definitely obtain the optimal solution of TSP is the method of exhaustive enumeration and evaluation. This procedure begins by generating the possibility of all the tours and evaluating according length or cost of the tour. The tour with the smallest length or cost chosen as the best, and guaranteed to be optimal. TSP is prevalent in real-world scenarios and researchers and companies are working on resolving cases of TSP and finding an optimal solution. One example is the delivery service, a courier needs to deliver goods to customers with different destinations and time is of considerable concern in delivery of goods as it relates to the reputation of the company. To reach the target requires a system capable of providing an optimal travel route so that the travel time can be minimized. In this project, we are trying to use the parallelization supported by haskell to see if parallelization can speed up the algorithm or improve it over its sequential implementation. We are using brute-force technique where we are trying out all possible orders lexicographically and Genetic algorithm approximation and analysing the performance difference between parallel and sequential implementation.

Problem formulation
The Travelling Salesman Problem (TSP) is the challenge that can be defined as follows: consider a number of cities which must be visited by a traveling salesman, only once, arriving once and departing once and starting and ending at the same city. Given the pairwise distances between cities, what is the best order in which to visit them, so as to minimize the overall distance traveled?

It is a well-known algorithmic problem in the fields of computer science. There are obviously a lot of different routes to choose, but finding the best one; the one that will require the least distance or cost is what researchers have spent decades trying to solve for.

It has commanded so much attention because it's so easy to describe it yet difficult to solve. The complexity of calculating the best route will keep on increasing when we add more destinations to the problem. That's why TSP belongs to the class of combinatorial optimization problems known as NP-complete. This implies that it is classified as NP-hard as it has no “quick” solution.

Example solution of a travelling salesman problem - the black line shows the shortest possible loop that connects every red dot:
Applications

1. **Overhauling gas turbine engines:** To guarantee a uniform gas flow through the turbines there are nozzle-guide vane assemblies located at each turbine stage. Such an assembly basically consists of a number of nozzle guide vanes affixed about its circumference. All these vanes have individual characteristics and the correct placement of the vanes can result in substantial benefits. The problem of placing the vanes in the best possible way can be modeled as a TSP with a special objective function.

2. **Order Picking problem:** This problem is associated with material handling in a warehouse. Assume that at a warehouse an order arrives for a certain subset of the items stored in the warehouse. Some vehicles have to collect all items of this order to ship them to the customer. The relation to the TSP is immediately seen. The storage locations of the items correspond to the nodes of the graph. The distance between two nodes is given by the time needed to move the vehicle from one location to the other. The problem of finding a shortest route for the vehicle with minimum pickup time can now be solved as a TSP.

3. **Vehicle Routing:** Suppose that in a city n mail boxes have to be emptied every day within a certain period of time, say 1 hour. The problem is to find the minimum number of trucks to do this and the shortest time to do the collections using this number of trucks.

Implementations

- **Brute Force sequential**
  In the brute force sequential approach we first enumerate all the possible permutations of the paths and calculate the distance of each possible path one after the other by traversing across the collected path one by one and picking the shortest one. This is an exhaustive search as we are searching over a large space. That’s this algorithm has exponential time complexity.

- **Bruteforce, calculate path distance in parallel**
  In the brute force approach, we first enumerate all the possible permutations of the paths as we did in sequential and then create sparks for each one of them to get calculated in
parallel and pick the shortest one. This is still an exhaustive search as we are searching over a large space but with a small reduction in search space as we are involving more than one core to perform this parallelization.

- **Bruteforce, calculate path distance in parallel with Chunk size**
  In the parallel brute force approach, rather than enumerate all the possible path permutations and then create sparks for each one of them we first divide them into a fixed chunk and then run these chunks in parallel and pick the shortest one.

- **Bruteforce for a batch of city groups**
  In this approach, we read an input file, replicate it a user-specified number of times, and then randomize them. Once we have got b batches we run the sequential algorithm over these b batches in a similar fashion as the naive brute force approach.

- **Bruteforce for a batch of city groups, each group in parallel**
  In this approach also, we first generate an infinite random number List between a list of empty length and the maximum number of cities as we did for the sequential batch algorithm. Once the random list is generated we pick the first b number from this random list and these b numbers for the batches from the city corpus. Now rather than running this algorithm for b batches in a sequential fashion we use Haskell parallelization to run them in parallel and choose the minimum cost path.

- **Genetic Algorithm with Population Size and Number of Generations**
  In the algorithm, we treat cities as genes, a single path that gets generated using these characters or problem constraints known as chromosomes, and a fitness score which is inversely proportional to the squared path length. The smaller the path length gene is, the fitter it is. The fittest of all the genes in the gene pool survive the population test and move to the next iteration. The number of iterations depends upon the value of a cooling variable. The cooling variable value keeps decreasing with each iteration and it reaches a threshold after a fixed number of iterations.

- **Genetic Algorithm for a batch of city groups**
  Here we replicate and randomize an input file to generate a batch of problems just like done before for the sequential batch processing. Once we have got b batches, we run the genetic algorithm defined earlier over these b batches in a sequential fashion and find the minimum cost path.

- **Genetic Algorithm for a batch of city groups, each group in parallel**
  This algorithm performs the same initial step as its sequential implementation defined earlier but it executes the genetic algorithm in batches parallelly similar to the brute force approach.
Performance Analysis

<table>
<thead>
<tr>
<th>Approach</th>
<th>Number of Core</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1</td>
<td>10.546</td>
</tr>
<tr>
<td>Parallel</td>
<td>1</td>
<td>86.59</td>
</tr>
<tr>
<td>Parallel</td>
<td>4</td>
<td>58.413</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>28.431</td>
</tr>
</tbody>
</table>

The sequential algorithm seems to be doing better than the parallel implementation. Therefore we look at threadscope and the number of sparks to see what the issue is:

4 core

8 Core
27,287,373,552 bytes allocated in the heap
8,660,023,944 bytes copied during GC
1,218,827,432 bytes maximum residency (27 sample(s))
1,622,514,520 bytes maximum slop
   4915 MiB total memory in use (0 MB lost due to fragmentation)

Tot time (elapsed) Avg pause  Max pause
Gen  0     14488 colls, 14488 par   29.908s  13.102s     0.0009s    0.0034s
Gen  1        27 colls,    26 par   12.141s   4.546s     0.1684s    1.1028s

Parallel GC work balance: 1.21% (serial 0%, perfect 100%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 39916800 (39872098 converted, 44702 overflowed, 0 dud, 0 GC'd, 0 fizzled)

INIT    time    0.000s  (  0.003s elapsed)
MUT     time   85.721s  ( 11.171s elapsed)
GC      time   42.049s  ( 17.649s elapsed)
EXIT    time    0.000s  (  0.006s elapsed)
Total   time  127.771s  ( 28.829s elapsed)

Alloc rate    318,327,269 bytes per MUT second
Productivity  67.1% of total user, 38.7% of total elapsed

The run is spending a lot of time doing garbage collection which is overpowering any gain got by
the parallelization. To overcome this, we divide the parallelization into chunks so as to not
overwhelm the processors with too many sparks at once. This seems to improve performance
over the normal parallel implementation.

**Dividing path calculation in chunks:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Core</th>
<th>Chunk</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parallel</td>
<td>1</td>
<td>1024</td>
<td>73.40</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
<td>------</td>
<td>-------</td>
</tr>
<tr>
<td>Parallel</td>
<td>4</td>
<td>1024</td>
<td>21.427</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>1024</td>
<td>15.323</td>
</tr>
</tbody>
</table>

4 Core

8 Core and 1024 chunk

34,964,115,760 bytes allocated in the heap
14,978,086,504 bytes copied during GC
913,065,696 bytes maximum residency (37 sample(s))
10,393,888 bytes maximum slop
2416 MiB total memory in use (0 MB lost due to fragmentation)

Tot time (elapsed)  Avg pause  Max pause
Gen 0 21932 colls, 21932 par 39.448s 11.825s 0.0005s 0.0090s
Gen 1 37 colls, 36 par 11.952s 1.834s 0.0496s 0.2569s

Parallel GC work balance: 62.58% (serial 0%, perfect 100%)

TASKS: 18 (1 bound, 17 peak workers (17 total), using -N8)

SPARKS: 38982 (38982 converted, 0 overflowed, 0 dud, 0 GC'd, 0 fizzled)
INIT time 0.001s (0.003s elapsed)
MUT time 15.361s (4.596s elapsed)
GC time 51.400s (13.658s elapsed)
EXIT time 0.000s (0.009s elapsed)
Total time 66.761s (18.266s elapsed)

Alloc rate 2,276,218,926 bytes per MUT second

Productivity 23.0% of total user, 25.2% of total elapsed

<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Core</th>
<th>Chunk</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parallel</td>
<td>8</td>
<td>1</td>
<td>35.772</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>4</td>
<td>23.727</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>8</td>
<td>21.946</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>16</td>
<td>27.513</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>32</td>
<td>22.304</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>64</td>
<td>24.741</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>128</td>
<td>14.188</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>256</td>
<td>17.628</td>
</tr>
</tbody>
</table>
The best result is achieved for a chunk size of 512 on 8 cores.

Analysis of City Groups in Batches
<table>
<thead>
<tr>
<th>Method</th>
<th>Number of Core</th>
<th>Batch Size</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>1</td>
<td>1</td>
<td>9.588</td>
</tr>
<tr>
<td>Sequential</td>
<td>1</td>
<td>10</td>
<td>11.030</td>
</tr>
<tr>
<td>Sequential</td>
<td>1</td>
<td>128</td>
<td>129.58</td>
</tr>
<tr>
<td>Parallel</td>
<td>1</td>
<td>10</td>
<td>10.449</td>
</tr>
<tr>
<td>Parallel</td>
<td>4</td>
<td>10</td>
<td>12.592</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>10</td>
<td>13.848</td>
</tr>
<tr>
<td>Parallel</td>
<td>1</td>
<td>128</td>
<td>139.73</td>
</tr>
<tr>
<td>Parallel</td>
<td>4</td>
<td>128</td>
<td>48.581</td>
</tr>
<tr>
<td>Parallel</td>
<td>8</td>
<td>128</td>
<td>37.542</td>
</tr>
</tbody>
</table>

4 Core and 128 batches of city group

8 Core and 128 batch of city groups
Since calculating the entire tsp min distance path for a set of cities is a much more intensive task than calculating one euclidean path. Therefore we see that these batch computations greatly benefit by parallelization. For a batch of 128 city groups, we see a speedup of 3.45x when we move to batches of city.

**Genetic Algorithm and Parallelization Analysis:**

<table>
<thead>
<tr>
<th>Method</th>
<th>Population Size</th>
<th>Generations</th>
<th>Input Size (Number of cities)</th>
<th>Min distance found (actual = 12)</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential</td>
<td>-</td>
<td>-</td>
<td>12</td>
<td>12</td>
<td>10.476</td>
</tr>
<tr>
<td>Genetic</td>
<td>32</td>
<td>32</td>
<td>12</td>
<td>12</td>
<td>0.322</td>
</tr>
<tr>
<td>Genetic</td>
<td>16</td>
<td>1</td>
<td>12</td>
<td>15</td>
<td>0.309</td>
</tr>
<tr>
<td>Genetic</td>
<td>16</td>
<td>4</td>
<td>12</td>
<td>14</td>
<td>0.314</td>
</tr>
<tr>
<td>Genetic</td>
<td>16</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>0.305</td>
</tr>
<tr>
<td>Genetic</td>
<td>16</td>
<td>16</td>
<td>12</td>
<td>13</td>
<td>0.312</td>
</tr>
</tbody>
</table>

Here we can see that the Genetic Algorithm has a speedup of **32.53x** over the sequential algorithm while giving the same answer.
Keeping the population size constant at 16, we see that the accuracy increases as the number of generations increases and we get the optimal solution after 32 generations.

<table>
<thead>
<tr>
<th>Method</th>
<th>Population Size</th>
<th>Generations</th>
<th>Input Size (Number of cities)</th>
<th>Min distance found (actual = 12)</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genetic</td>
<td>16</td>
<td>32</td>
<td>12</td>
<td>12</td>
<td>0.308</td>
</tr>
<tr>
<td>Genetic</td>
<td>16</td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>0.312</td>
</tr>
</tbody>
</table>

Keeping the number of generations constant at 8, we see that the accuracy increases as the population size increases and we get the optimal solution with a population size of 32.

<table>
<thead>
<tr>
<th>Method</th>
<th>Population Size</th>
<th>Generations</th>
<th>Input Size (Number of cities)</th>
<th>Min distance found (actual = 102)</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genetic</td>
<td>16</td>
<td>16</td>
<td>102</td>
<td>216</td>
<td>0.339</td>
</tr>
<tr>
<td>Genetic</td>
<td>32</td>
<td>32</td>
<td>102</td>
<td>213</td>
<td>0.586</td>
</tr>
<tr>
<td>Genetic</td>
<td>64</td>
<td>64</td>
<td>102</td>
<td>161</td>
<td>3.067</td>
</tr>
</tbody>
</table>
For a very large input size (102 cities), solving it by brute force is infeasible as we would need to calculate $101!$ possible path distances. However we are able to calculate a reasonable approximation 112 to the actual min distance of 102 within reasonable time using the genetic algorithm. Above, we see that accuracy increases as population size and number of generations increases.

### Genetic Algorithm in batch analysis over big input

<table>
<thead>
<tr>
<th>Method</th>
<th>Population Size</th>
<th>Generations</th>
<th>Batch Size</th>
<th>Num of Core</th>
<th>Time Taken(secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genetic</td>
<td>128</td>
<td>128</td>
<td>1</td>
<td>1</td>
<td>19.193</td>
</tr>
<tr>
<td>Genetic</td>
<td>128</td>
<td>128</td>
<td>10</td>
<td>1</td>
<td>80.40</td>
</tr>
<tr>
<td>Genetic batch parallel processing</td>
<td>128</td>
<td>128</td>
<td>10</td>
<td>1</td>
<td>78.87</td>
</tr>
<tr>
<td>Genetic batch parallel processing</td>
<td>128</td>
<td>128</td>
<td>10</td>
<td>4</td>
<td>25.806</td>
</tr>
<tr>
<td>Genetic batch parallel processing</td>
<td>128</td>
<td>128</td>
<td>10</td>
<td>8</td>
<td>26.524</td>
</tr>
</tbody>
</table>

We parallelize the calculation of a batch of 10 groups using the genetic algorithm, each having about 102 cities. We see a speedup of 3.03x by parallelizing the batch processing.
Conclusion

Calculating the euclidean path distance for a set of coordinates is not a very heavy task by itself and therefore parallelizing the calculation of multiple paths at once does not benefit us much. Dividing the parallelization into chunks helps reduce the number of sparks and garbage collection. Since calculating the entire tsp min distance path for a set of cities is a much more intensive task than calculating one euclidean path. Therefore we see that these batch computations greatly benefit by parallelization. For a batch of 128 city groups, we see a speedup of 3.45x when we move from sequential to batch algorithms. We also see that the Genetic Algorithm has a speedup of 32.53x over the sequential algorithm while giving the same answer. The Genetic Algorithm also makes it feasible to solve very large problems which are not possible to be solved by brute force in reasonable time. We also parallelize the calculation of a batch of 10 groups using the genetic algorithm, each having about 102 cities and see a speedup of 3.03x by parallelizing the batch processing

Code Listing

Lib.hs

module Lib

{ runMain,
}

where

import Control.Parallel.Strategies
import Data.List (permutations)
import GeneticUtils
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import System.IO (readFile)
import Types
import Utils

-- Consider all permutations while keeping starting point fixed
minPathDistance :: [Point] -> Int
minPathDistance [] = -1
minPathDistance (c : cities) =
    minimum $
        map (pathDistance . (c :)) $
        permutations cities

parallelMinPathDistance :: [Point] -> Int
parallelMinPathDistance [] = -1
parallelMinPathDistance (c : cities) =
    minimum $
        parMap rseq (pathDistance . (c :)) $
        permutations cities

chunkedParallelMinPathDistance :: [Point] -> Int -> Int
chunkedParallelMinPathDistance [] _ = -1
chunkedParallelMinPathDistance (c : cities) chunkSize =
    minimum $
        withStrategy (parListChunk chunkSize rdeepseq)
        . map (pathDistance . (c :))
        $ permutations cities

batchMinPathDistance :: [[[Point]]] -> [Int]
batchMinPathDistance = map minPathDistance

batchParallelMinPathDistance :: [[[Point]]] -> [Int]
batchParallelMinPathDistance = parMap rseq minPathDistance

geneticMinPathDistance :: Int -> Int -> [Point] -> Int
 GENETIC_MINPATHDISTANCE populationSize generations cities =
    minimum $ map pathDistance finalPop

    where
        population = replicate populationSize cities
randomList = randomListInRange 0 (length cities - 1)
finalPop =
  fst $
  foldr
    (\f (p, r) -> (f p r, tail r))
    (population, randomList)
    (replicate generations nextGen)

batchGeneticMinPathDistance :: Int -> Int -> [[Point]] -> [Int]
batchGeneticMinPathDistance p g =
  map
    (geneticMinPathDistance p g)

batchParallelGeneticMinPathDistance :: Int -> Int -> [[Point]] -> [Int]
batchParallelGeneticMinPathDistance p g =
  parMap
    rseq
    (geneticMinPathDistance p g)

runMain :: IO ()
runMain = do
  args <- getArgs
  case args of
    -- bruteforce sequential
    ["-s", filename] -> do
      corpus <- readFile filename
      print $ minPathDistance $ makeCities corpus

    -- bruteforce, calculate path distance in parallel
    ["-p", filename] -> do
      corpus <- readFile filename
      print $ parallelMinPathDistance $ makeCities corpus

    -- bruteforce, calculate path distance in parallel chunks
    ["-c", ':'] : 'n' : n, filename] -> do
      corpus <- readFile filename
      print $ chunkedParallelMinPathDistance (makeCities corpus) (read n)

    -- bruteforce for batch of city groups
    ["-s", ':'] : 'b' : b, filename] -> do
      corpus <- readFile filename
      let cities = makeCities corpus

randomList = randomListInRange 0 (length cities)

in print $ batchMinPathDistance [take r cities | r <- take (read b) randomList]

-- brute force for batch of city groups, each group in parallel
["-sp", 'b' : b, filename] -> do
  corpus <- readFile filename
  let cities = makeCities corpus
      randomList = randomListInRange 0 (length cities)
  in print $ batchParallelMinPathDistance [take r cities | r <- take (read b) randomList]

-- genetic algorithm
["-g", 'p' : p, 'g' : g, filename] -> do
  corpus <- readFile filename
  print $ geneticMinPathDistance (read p) (read g) $ makeCities corpus

-- genetic algorithm for batch of city groups
["-g", 'p' : p, 'g' : g, 'b' : b, filename] -> do
  corpus <- readFile filename
  let cities = makeCities corpus
      randomList = randomListInRange 0 (length cities)
  in print $ batchGeneticMinPathDistance (read p) (read g) $ makeCities corpus

-- genetic algorithm for batch of city groups, each group in parallel
["-gp", 'p' : p, 'g' : g, 'b' : b, filename] -> do
  corpus <- readFile filename
  let cities = makeCities corpus
      randomList = randomListInRange 0 (length cities)
  in print $ batchParallelGeneticMinPathDistance (read p) (read g) $ makeCities corpus

-- invalid running params
_ -> do
  pn <- getProgName
  die $ "Usage: " ++ pn ++ " [-s|-p|-c :nN|-s :bN|-sp :bN|-g :pN :gN|-g :pN :gN|-gp :pN :gN :bN] <filename>"
**type Point** = (Int, Int)

**Utils.hs**

module Utils

{ distance,
  squaredDistance,
  makeCities,
  pathDistance,
  squaredPathDistance,
  randomListInRange,
 }

where

import System.Random (mkStdGen, randomRs)
import Types

squaredDistance :: Point -> Point -> Int
squaredDistance (x1, y1) (x2, y2) = ((x2 - x1) ^ 2) + ((y2 - y1) ^ 2)

distance :: Point -> Point -> Int
distance a b = floor . sqrt . fromIntegral $ squaredDistance a b

makeCities :: String -> [Point]
makeCities corpus = makePairs $ map read $ words corpus

where

  makePairs [] = []
  makePairs [p] = [(p, p)] -- replicate last coordinate if odd numbers
  makePairs (p : q : r) = (p, q) : makePairs r

pathDistance :: [Point] -> Int
pathDistance cities = sum $ zipWith distance path (tail path)

where

  path = last cities : cities

squaredPathDistance :: [Point] -> Int
squaredPathDistance cities = sum $ zipWith squaredDistance path (tail path)

where

  path = last cities : cities

randomListInRange :: Int -> Int -> [Int]
randomListInRange s e = randomRs (s, e) rg
where
  rg = mkStdGen 0

GeneticUtils.hs
module GeneticUtils (nextGen) where

import Data.List (sortBy)
import qualified Data.Set as S
import Types
import Utils

crossover :: [Point] -> [Point] -> Int -> Int -> [Point]
crossover parentA parentB i j = c1 ++ c2
  where
    s = min i j
    e = max i j
    c1 = [x | (x, i) <- zip parentA [0 ..], s <= i && i <= e]
    c1Set = S.fromList c1
    c2 = [x | x <- parentB, not (x `S.member` c1Set)]

nextGen :: [[[Point]]] -> [Int] -> [[[Point]]]
nextGen pop randomList =
  take (length pop) $
    map fst $
      sortBy (\p1 p2 -> compare (snd p1) (snd p2)) $ 
      map (\p -> (p, squaredPathDistance p)) $ 
        [ crossover pa pb ri rj
          | (i, pa, ri) <- zip3 [0 ..] pop randomList,
            (j, pb, rj) <- zip3 [0 ..] pop (tail randomList),
            i < j
        ]

Main.hs
module Main where

import Lib

main :: IO ()
main = runMain
Others

dependencies:
- base >= 4.7 && < 5
- parallel
- random
- containers

dhc-options:
- -O2
- -threaded
- -rtsopts
- -eventlog

References