Parallel Functional Programming Project Final Report: Genetic Algorithms (GenAlgo)

Jake Fisher (jf3148) & Pedro Santos (pb2751)

December 2021

1 Background

Genetic algorithms are a class of algorithms designed for solving search problems. Their key technique is generating a population of many potential solutions, evaluating them, and then choosing the next generation from a "crossover" of the best solutions of the previous generation. For many search problems, there are often many local minima that typical optimization algorithms can get stuck in, so an algorithm that introduces randomness can be useful for finding a global optimum. This is also a useful class of algorithms as it can find approximate solutions to NP-problems in reasonable amounts of time.

Genetic algorithms can be useful in solving constraint satisfaction problem as well because these can often have a straightforward representation and calculation of how good a solution is. This can be judged based on the number of constraints satisfied/broken.

Some example problems that can be solved efficiently with genetic algorithms are N-Queens, hyper-parameter optimization, code-breaking, and the traveling salesman problem.

2 Algorithm Structure

Our project provides a general-use genetic algorithm solver that takes in a genetic algorithm "problem" (such as N-Queens), and the parameters for that given problem (such as the number, N, of queens), and solves it using a genetic algorithm solver. Note that the problems would be provided in a separate user-defined module to be imported from the main function solver.

A genetic algorithm problem is defined as an instance of the `GAData` class, which is the following.

```
class GAData a where
  dataRange :: a -> (Int, Int)
  repLength :: a -> Int
  fitness :: a -> [Int] -> Int
```

```
maxFitness :: a -> Int
showSol :: a -> [Int] -> [Char]
```

`dataRange` and `repLength` are functions for determining the numeric range of the items in each chromosome and the length of a given chromosome, respectively. `fitness` and `maxFitness` are, respectively, the function on a given chromosome that evaluates its "quality" (and thus likelihood to move on to the next generation), and the highest possible fitness score a chromosome could get, which is both used for early termination (e.g., in a CSP where all constraints are satisfied) and as a denominator when determining the likelihood that a given chromosome will move on to the next generation. `showSol` is simply a pretty printing function (e.g., printing an N-Queens solution as an $N \times N$ chess board.

For this stage of the project, we implemented `GAData` instances of the N-Queens problem, and the Traveling Salesman Problem.

In the main module, we first parse the arguments provided which would include in the following order: number of iterations/generations to run over, the population size, the percentage of mutation, the name of the problem, the problem arguments (e.g., number of queens for the N-Queens problem).

We then loop over the number of iterations required and return the best solution from the most recent iteration. If at any point however we reach the optimal solution, we would just return that solution instead.

3 Sequential Implementation

We based our sequential implementation on a Python implementation specifically designed for solving N-Queens with a genetic algorithm [2], adapting it both for Haskell and generalizing it for all genetic algorithm problems

Our sequential implementation consists of the the following steps:

1. Population initialization
2. Fitness calculation
3. Population sorting
4. Crossover
5. Mutation

3.1 Population Initialization

This step is only run a single time, as opposed to all others, which are run in a loop. It has the following type signature (to initialize the full population:

```
initialize :: GAData a => a -> [Int] -> Int -> ([[Int]], [Int])
```

The second parameter is the list of random numbers generation by the main genetic algorithm function, which is an infinite list. Because doing all of our work in monad-space would have made for a very difficult parallelization task, we generate random numbers by intially generating two infinite lists of random numbers, one of floats between 0 and 1, to be use for calculating odds for crossover and mutation, and one of integer in the `dataRange` of the `GAData` problem, used for new values both in initialization and in mutation. Because we always know exactly how many values from the infinite list we will need for any given step, we can efficiently split up the list later to portion out to different chromosomes.

3.2 Fitness Calculation

This is the first step in the genetic algorithm loop; in our sequential implementation, it simply consists of running `map (\x -> (fitness d x, x))` over the population, creating a new list of fitness-chromosome tuples.

3.3 Population sorting

In our sequential implementation, we simply use the Standard Prelude's `sortOn` function, with `negate . fst` as the key function (in order to sort by fitness, in descending order). If the top solution has the maximum fitness, we can stop the loop and return that solution.

3.4 Crossover

In order to ensure that fitter chromosomes are more represented in the next generation, we pick $2N$ (where N is the population size) chromosomes randomly from the previous population, where the probability of picking a chromosome c is proportional to

$$\frac{f(c)}{\max f}$$

Where f and $\max f$ are the `fitness` and `maxFitness` functions defined for the given `GAData` problem. We then section the $2N$ chromosomes into pairs, and run the crossover function on them, which has the following type signature:

```
crossover :: GAData a => a -> ([Int], [Int]) -> [Int]
```

There are a number of potential crossover functions one can use in a genetic algorithm; we used a Uniform Crossover, where the "child" chromosome consists of the first half of the first half of one parent chromosome, and the second half of the other parent chromosome.

3.5 Mutation

The type signature for the mutation is the following:

```haskell
mutate :: GAData a => a -> Float -> [Float] -> [Int] -> [Int] ->
  ↪  [Int]
```

We take in a mutation threshold, a list of random odds, the original values, and randomly generated new values, replacing a given value only if the odds for that location are above the threshold.

4 Parallel Implementation

Given that we now have an infinite list of random numbers that is split apart, we can evaluate our expensive fitness, crossover and mutate maps in parallel. For this we used the following structure:

```haskell
import Control.Parallel.Strategies(parMap, rdeepseq)
-- ...
parMap rdeepseq f population
```

Where f represents the function that takes an element of the population, applies our fitness, crossover or mutate function and returns the modified result.

We used `parMap` as each element of the list could be evaluated in parallel when carrying out the map. Initially, we tried several combinations of `parList` and one of `rpar`, `rseq` but found that this would not do any useful work as between our key parallel functions, we needed sequential work to be done. If the parallel steps did not fully evaluate their elements, then this burden would have to be taken on by the sequential portion of our algorithm AND we would have more overhead due to spark creation. For this step we did not use `parBuffer` as the outstanding sparks created would all be consumed almost immediately and so more would have to be generated for other list members.

With this change we found that we had almost 100% spark conversion rate.

Seeing that an individual mutation was a good candidate for parallelization, as whether or not an index will be mutated is independent of the other indices, we tested an initial parallelization of that step. However, we found that it slowed down the algorithm and resulted in many fizzled sparks. We surmised that this was because a given mutation was already in a parallelized operation, so with a non-trivial population size, any cores would already be busy at that point. We thus concluded that any parallelization of other lower-level steps would not be useful, and in fact would likely be harmful.

The next change that we implemented was a parallel depth-limited sort. We found that out of the sequential portion of our algorithm, sorting was an expensive operation so we sought to improve it. While in Marlow's book he provides an attempt at parallelizing Quicksort, we attempted one of Merge Sort for simplicity. We found that a good way to parallelize it was with an implementation similar to that of Fibonacci generation seen in the class slides. This step yielded yet more speedup.

The Merge Sort code was as follows:

```haskell
merge :: [(Int, [Int])] -> [(Int, [Int])] -> [(Int, [Int])]
merge l [] = l
merge [] l = l
merge l1@(x1@(f1, _):t1) l2@(x2@(f2, _):t2)
  | f1 > f2    = x1: merge t1 l2
  | otherwise = x2: merge l1 t2

-- depth limited sorting
parMergeSort :: Int -> [(Int, [Int])] -> [(Int, [Int])]
parMergeSort _ [] = []
parMergeSort _ l@[_] = l
parMergeSort d l
  | d == 0     = sortOn (negate . fst) l
  | otherwise = mergedFirst `par` mergedSecond `pseq` merge
  ↪  mergedFirst mergedSecond
  where
    mergedFirst = parMergeSort (d-1) first
    mergedSecond = parMergeSort (d-1) second
    (first, second) = splitAt halfway l
    (halfway, _) = quotRem (length l) 2
```

Though there was fizzling, we tuned the depth limit to reduce this. We found that for the computer we were running our code on, a depth limit of 7 was a good compromise to reduce fizzling while maintaining performance.

However, we found that if we increased the population size up to 10000 or the equivalent size increase in representations, we would begin to get a large number of sparks overflowing. This was likely due to `parMap`, like `parList`, generating all the sparks for a list at the beginning. This led us to thin chunks of our population before each parallelism step.

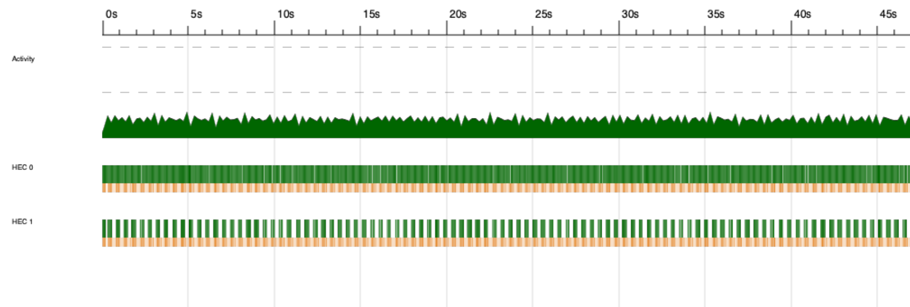
One issue we noticed with particularly large population sizes ($> 5,000$) was spark overflowing, which would be expected. A possible solution we have seen for this type of problem is through chunking, but our problem does not suit itself especially well to that strategy, as there are steps that rely on cross-chromosome interaction, such as sorting and crossover, meaning that we would have to de-chunk and re-chunk multiple times in each iteration. So, while we did not implement chunking at this stage, it is still a worthwhile avenue to consider for this type of problem, though the repeated chunking problem would need to be ameliorated in some fashion.

5 Results

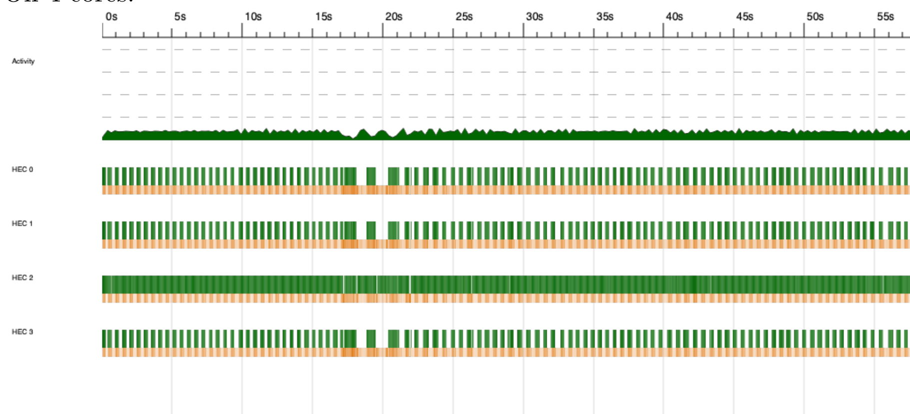
To run the code and reproduce results, first clone from GitHub and go to the project directory. Then the commands to run would be `stack build` followed by `time stack run -- <program-args> [+RTS <RTS-args>]`. Note that the largest number of cores for one of our computers was 4 hence that will be the limit for our tables and graphs.

5.1 ThreadScope Results

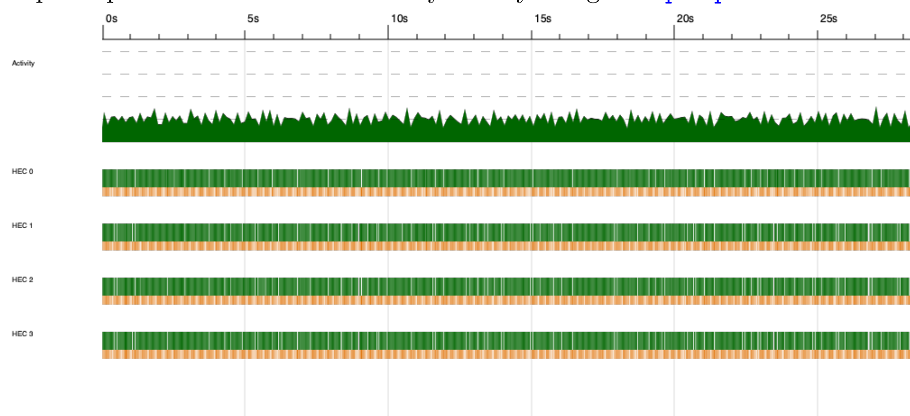
The event log of our initial attempt, using `rSeq`, on two cores:



On 4 cores:



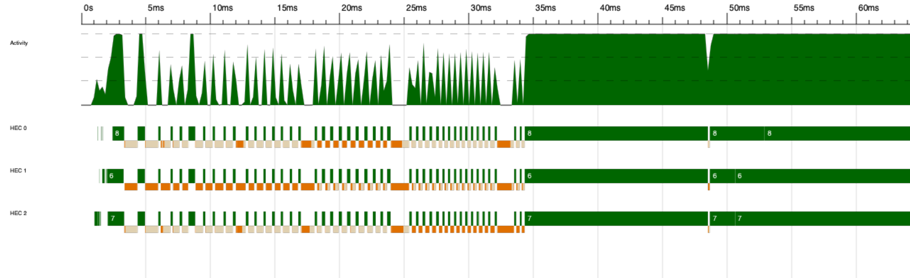
As we can see, the load balancing is very poor, as `rSeq` isn't actuallu evaluating any sizable portion of the work, which then means that the primary core needs to pick up the remainder. We remedy this by using `rdeepseq` instead:



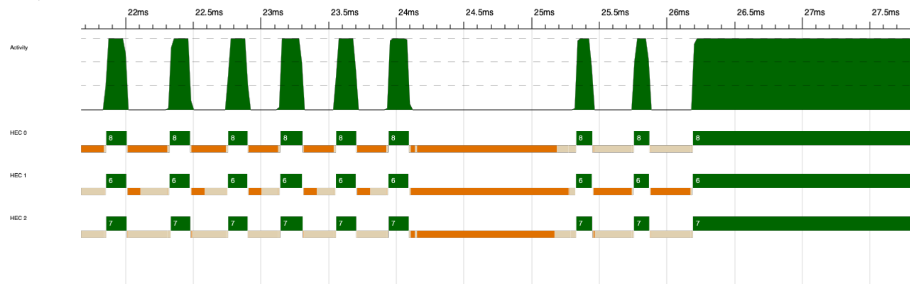
Now, we can see far improved load balancing (note the smaller gaps on the non-primary cores).

The following is a zoomed-in portion of the sorting section our ThreadScope

log with no maximum depth on sorts:



With a maximum depth of 7 (note the differing timescale from the previous log):



5.2 Timing Results

Here are presented run times and speedups for N-Queens and TSP respectively with corresponding line graphs against ideal speedup with differing population sizes and representation lengths.

Note that despite diminishing returns, as the number of cores went up, the number of sparks GCed and fizzled began to decrease.

The parameters for the following NQueens experiments were for 10 iterations, varying population size, 0.1 mutation percentage and 128 queens.

Here the population was fixed at 1000. Parameters other than the number of queens are the same as in the previous problem.

The following represent the first experiment for TSP over 20 iterations with 100 cities:

6 Conclusion

We find that we are able to efficiently parallelize the Genetic Algorithms problem, getting significant speedups, though we note that the Genetic Algorithms problem is not without its issues in parallelization. We could explore further work on reducing overflow without introducing expensive recurrent chunking/dechunking steps at each iteration, and expanding to other Genetic Algorithms applications.

Population	Cores	Time	Speedup
500	1	9.301	1
500	2	5.220	1.78
500	3	4.123	2.26
500	4	3.766	2.47
1000	1	20.556	1
1000	2	11.106	1.85
1000	3	9.133	2.25
1000	4	7.735	2.66
2000	1	41.014	1
2000	2	25.261	1.62
2000	3	19.622	2.09
2000	4	17.509	2.34

Table 1: NQueens popsize [500, 1000, 2000]

Num Queens	Cores	Time	Speedup
64	1	5.238	1
64	2	3.036	1.73
64	3	2.507	2.09
64	4	2.347	2.23
128	1	20.556	1
128	2	11.106	1.85
128	3	9.133	2.25
128	4	7.735	2.66
256	1	79.35	1
256	2	45.370	1.75
256	3	38.142	2.08
256	4	28.613	2.77

Table 2: NQueens popsize [500, 1000, 2000]

Another avenue we could look at would be maintaining several different random generators, rather than having two infinite lists we split off and divide up for each chromosome.

Some other limitations include the fact that while we try to generalize, this means that it is possible that we lose some potential speedups as the representation length (not population size) gets larger. For very large representation lengths it would be important to parallelize at a lower level (within the mutate function itself, for example), though these very large representation lengths would also likely require large population sizes for good results.

Also trying to generalize means that we may be missing more optimal functions for different sorts of problems. However, this would require more onus put on another user/developer to write their own functions e.g.: crossover and we preferred to explore a simpler modular approach.

PopSize	Cores	Time	Speedup
1000	1	3.095	1
1000	2	2.115	1.46
1000	3	1.956	1.58
1000	4	1.972	1.57
2000	1	5.881	1
2000	2	3.974	1.48
2000	3	3.488	1.68
2000	4	3.479	1.69
5000	1	16.108	1
5000	2	10.897	1.48
5000	3	9.714	1.66
5000	4	9.989	1.61

Table 3: TSP popsize [1000, 2000, 5000]

A Code Listing

A.1 Main Module

```
1  module Main where
2
3
4  import System.Environment(getArgs, getProgName)
5  import System.Exit(exitFailure, exitSuccess)
6  import Data.List(sortOn)
7  import Data.List.Grouping(splitEvery)
8  import System.Random(getStdGen, randomRs)
9  import Control.Monad(when)
10 import Control.Parallel(par, pseq)
11 import Control.Parallel.Strategies(using, parList, rdeepseq)
12 import qualified Data.Map as M
13
14 import GA
15 import NQueens
16 import TSP
17
18 main :: IO ()
19 main =  do
20   args <- getArgs
21   case args of
22     iters:popSize:mutationPct:pArgs -> do
23       -- do some argparsing here
24       when (read popSize <= (0::Integer)) (do
25           putStrLn "Population size must be larger than 0."
```

```
26            exitFailure)
27
28        case pArgs of
29          ["nqueens", n] -> do
30            let d = NQueens $ read n
31            geneticAlgorithm d (read iters) (read popSize) (read
              ↪  mutationPct)
32          ["tsp", n'] -> do
33            let n = read n'
34                cm = M.fromList [(i, M.fromList [(j, i+j+1) | j <-
                  ↪  [0..(n-1)]])| i <- [0..(n-1)]]
35                d = TSP n cm
36            geneticAlgorithm d (read iters) (read popSize) (read
              ↪  mutationPct)
37          _ -> do
38            putStrLn "Unimplemented problem-string or poorly
              ↪  written problem-args"
39            exitFailure
40
41      _ -> do
42        prog <- getProgName
43        putStrLn $ "Usage: "++prog++" <num-iter> <pop-size>
          ↪  <mutation-pct> <problem-string> ...<problem-args>"
44        exitFailure
45
46
47  merge :: [(Int, [Int])] -> [(Int, [Int])] -> [(Int, [Int])]
48  merge l [] = l
49  merge [] l = l
50  merge l1@(x1@(f1, _):t1) l2@(x2@(f2, _):t2)
51    | f1 > f2    = x1: merge t1 l2
52    | otherwise = x2: merge l1 t2
53
54  -- depth limited sorting
55  parMergeSort :: Int -> [(Int, [Int])] -> [(Int, [Int])]
56  parMergeSort _ [] = []
57  parMergeSort _ l@[_] = l
58  parMergeSort d l
59    | d == 0    = sortOn (negate . fst) l
60    | otherwise = mergedFirst `par` mergedSecond `pseq` merge
      ↪  mergedFirst mergedSecond
61    where
62      mergedFirst = parMergeSort (d-1) first
63      mergedSecond = parMergeSort (d-1) second
64      (first, second) = splitAt halfway l
65      (halfway, _) = quotRem (length l) 2
```

```
66
67
68  geneticAlgorithm :: GAData a => a -> Int -> Int -> Float -> IO ()
69  geneticAlgorithm d iters popSize mutationPct = do
70    g <- getStdGen
71    let randOdds = randomRs (0.0 :: Float, 1.0) g
72        randVals = randomRs (dataRange d) g
73        (initialPop, newRandomVals) = initialize d randVals popSize
74
75    -- we want to loop with this initial population until done
76    gaLoop initialPop randOdds newRandomVals iters
77        where
78            gaLoop pop rs rvs curIters = do
79
80              -- 1) Get the fitness of whole population in order.
                 ↪  DeepSeq for nonsequential eval
81              let fitPop = map (\x -> (fitness d x, x)) pop `using`
                 ↪  parList rdeepseq
82                  sortedFitPop = parMergeSort 7 fitPop  -- this
                     ↪  could be a parameter, but limited by our
                     ↪  machines' cores
83                  (topFitness, topSolution):_ = sortedFitPop
84
85              -- Exit if optimal
86              when (topFitness == maxFitness d) (do
87                  putStrLn $ "Optimal solution found:\n" ++ showSol
                     ↪  d topSolution
88                  exitSuccess)
89
90              -- 2) Get the proportional probabilities for each
                 ↪  chromosome
91              let probs = map (\(f, _) -> (fromIntegral f :: Float)
                 ↪  / fromIntegral (maxFitness d)) sortedFitPop
92                  totalProb = sum probs
93
94                  -- 3) take the number of probabilities needed to
                     ↪  select elements for crossover
95                  (crossRs, rs1) = splitAt (length pop * 2) rs
96                  -- this is probably a bottleneck
97                  pairList = map (\r -> pickrandom (r*totalProb)
                     ↪  sortedFitPop probs 0) crossRs
98                  pairs  = map (\[x, y] -> (x,y)) (splitEvery 2
                     ↪  pairList)
99                  crossed = map (crossover d) pairs `using` parList
                     ↪  rdeepseq
100
```

```
101                 numRs = length pop * repLength d
102
103                 spltAtEvery l = (splitEvery (repLength d) old ,
                    ↪  new)
104                   where (old, new) = splitAt numRs l
105
106                 -- 4) take odds and elements to switch to when
                    ↪  mutating randomly
107                 (splitRs, rs2) = spltAtEvery rs1
108                 (splitRvs, nextRvs) = spltAtEvery rvs
109                 zippedList = zip3 splitRs splitRvs crossed
110
111                 mutated = map (\(rands, rvals, sol) -> mutate d
                    ↪  mutationPct rands rvals sol) zippedList
                    ↪  `using` parList rdeepseq
112
113             -- VERBOSE
114             -- putStrLn £ "The top rated solution for this
                ↪  generation is:\n" ++ showSol t topSolution ++
                ↪  "\nwith fitness " ++ show topFitness
115             -- putStrLn £ "The top fitness is " ++ show
                ↪  topFitness
116             if curIters > 1
117                then gaLoop mutated rs2 nextRvs (curIters-1)
118                else do
119                  putStrLn $ "The top rated solution after all
                     ↪  iterations is:\n" ++ showSol d topSolution
                     ↪  ++ "\nwith fitness " ++ show topFitness
120
121             where
122                 -- take first element that exceeds a cumulative
                    ↪  prob threshold
123                 pickrandom _ [(_,c)] _ _ = c
124                 pickrandom r ((_,c):ct) (p:pt) upto
125                   | upto + p >= r = c
126                   | otherwise = pickrandom r ct pt (upto + p)
127                 pickrandom _ _ _ _ = []
```

A.2 Genetic Algorithm Solver

```
1  module GA(
2    GAData,
3    dataRange,
4    repLength,
5    fitness,
6    maxFitness,
```

```
7    showSol,
8    crossover,
9    mutate,
10   initialize
11 ) where
12
13 import Data.List.Grouping(splitEvery)
14
15 class GAData a where
16   dataRange :: a -> (Int, Int)
17   repLength :: a -> Int
18   fitness :: a -> [Int] -> Int
19   maxFitness :: a -> Int
20   showSol :: a -> [Int] -> [Char]
21
22
23 -- These functions will be common to all genetic instances
24 initialize :: GAData a => a -> [Int] -> Int -> ([[Int]], [Int])
25 initialize d l popSize = (splitEvery r vals, newVals)
26   where
27     r = repLength d
28     (vals, newVals) = splitAt (r*popSize) l
29
30
31
32 {-
33       The following will be implemented as Uniform Crossover:
34       We could generate a random number to see which int is kept
   ↪  in the child.
35       We will use splitting instead.
36       References:
37       How to generate different pcts
38
   ↪  https://mail.haskell.org/pipermail/beginners/2010-April/004058.html
39 -}
40 crossover :: GAData a => a -> ([Int], [Int]) -> [Int]
41 crossover d (a, b) = take half a ++ drop half b
42     where (half, _) = quotRem (repLength d) 2
43
44
45 mutate :: GAData a => a -> Float -> [Float] -> [Int] -> [Int] ->
   ↪  [Int]
46 mutate _ mutPct pcts rvs sol = map keep $ zip3 pcts rvs sol
47   where
48     keep (r, x', x)
49       | r > mutPct = x
```

```
50        | otherwise = x'
```

A.3 NQueens Problem Module

```haskell
1  module NQueens(
2    NQueens(NQueens)
3  ) where
4
5  import GA
6  import Data.List(intercalate, intersperse)
7
8  newtype NQueens = NQueens Int
9    deriving Show
10
11 instance GAData NQueens where
12   dataRange (NQueens n) = (0, n-1)
13   repLength (NQueens n) = n
14   fitness d@(NQueens n) queens = maxFitness d - vertCols -
     ↪ diagCols1 - diagCols2
15     where
16       calc l = sum $ map checkMult l
17       checkMult x = if x > 1 then x-1 else 0
18       posQueens = zip [0..] queens
19       cols f = calc [length $ filter (f i) posQueens | i <-
         ↪ [(-n)..(n-1)]]
20       vertCols = cols (\i (_, column) -> i == column)
21       diagCols1 = cols (\i (row, column) -> row == column + i)
22       diagCols2 = cols (\i (row, column) -> row == n - column +
         ↪ i)
23   maxFitness (NQueens n) = fst $ quotRem (n*(n-1)) 2
24   showSol (NQueens n) queens = intercalate "\n" $ map line queens
25     where line x = intersperse ' ' $ (replicate x 'x') ++ "Q" ++
       ↪ (replicate (n-x-1) 'x')
```

A.4 Travelling Salesman Problem Module

```haskell
1  module TSP(
2    TSP(TSP)
3  ) where
4
5  import GA
6  import qualified Data.Map as M
7  import Data.List.Unique(allUnique)
8  import Data.Maybe(fromMaybe)
9
10 data TSP = TSP Int CityMap
```

```
11    deriving Show
12
13  type CityMap = M.Map City (M.Map City Weight)
14  type City = Int
15  type Weight = Int
16
17  instance GAData TSP where
18    dataRange (TSP n _) = (0, n-1)
19    repLength (TSP n _) = n
20    fitness d@(TSP _ cm) order = allCities*pathExists*(maxFitness d
      ↪  - distance)
21      where
22        allCities = if allUnique order then 1 else 0
23        paths = zip order $ tail order
24        maybeDistance (s, e) = M.lookup s cm >>= M.lookup e
25        maybeDistances = map maybeDistance paths
26        distances = map (fromMaybe 0) maybeDistances
27        distance = sum distances
28        pathExists = if product distances > 0 then 1 else 0
29    maxFitness (TSP n cm) = 1 + (n-1) * M.foldr (\m acc -> max acc
      ↪  $ M.foldr max 0 m) 0 cm
30    -- we add 1 to the maxFitness so that "longest" paths get
      ↪  fitness 1, but invalid paths get fitness 0
31    showSol _ order = show order
```

References

[1] Stephen Edwards. strategies.pdf.

[2] Waqqas Iqbal. N-Queen problem using Genetic Algorithm, November 2021. original-date: 2019-01-01T06:41:39Z.

[3] Uddalok Sarkar and Sayan Nag. An adaptive genetic algorithm for solving n-queens problem. *CoRR*, abs/1802.02006, 2018.

[4] Eric Stoltz. Evolution of a salesman: A complete genetic algorithm tutorial for Python, March 2021.