1 Introduction

In this project, we are interested in solving 8-Queens puzzle, which is a popular board game in Artificial Intelligence.

1.1 8-queens puzzle

In an 8-queens puzzle, we have a 8*8 board and 8 queens to be placed onto the board. However, queens will attack each other if they are placed in the same column, row and diagonal line. As a peace lover, we want to minimize the number of attacks and ideally, we want no combats between queens. That is, in ideal case, we want exactly 1 queen for each row and column, and at most 1 queen in each diagonal line. The problem can also be generalized to n-queens, in which we have a n*n board and n queens to be placed. In our project, we also tested our algorithm on 10-Queens and 16-Queens.

1.2 Genetic Algorithm

Genetic Algorithm is a process inspired by natural selection. In genetic algorithm, we start with some randomly generated states, which we will call as population. To reach an optimal or ideal state, we want the population to evolve. To evaluate population, we will need a fitness function that measures how good an entity is. We will randomly select pairs of individuals to generate a new populations. The parent individuals are usually selected with respect to some probabilities based on the fitness function. To generate the offspring, we randomly choose a crossover point and cross the parents at the crossover point. Each element in the offspring (e.g each character in the string) will also be subject to some mutation with a small probability. Because we generally select parents with better fitness, we will expect our new population to be better than the parent population. The algorithm terminates when we reach a goal or after some number of iterations.

2 Implementation

In this section, we will be introducing our implementation for Parallel Genetic Algorithm to solve the 8-queens problem.

2.1 Problem Formulation

To solve the problem, we have to first formulate the setting for the game. Since we know we have n queens and a n*n board, we can represent our board as a list of queen positions, and each position
is represented by a tuple of integers in $[1, n]$. Our fitness function is the number of non-attacking pairs, and our goal for a 8-queen setting will give a fitness of 28. In our implementation, we score each board using $fitness(goal) - fitness(state)$, that is 0 score for a goal state, to make our code more readable.

2.2 Genetic Algorithm

First of all, the algorithm needs a pool of gene. Here we take all position of the board as gene pool to make sure the queens can show up in any position on the board.

To generate an entity, we generate a list of random numbers to simulate the shuffle of the pool. Then we pick n gene in the pool according to the list to guarantee every possible solution has exactly n queens.

The next step is to select best part of entities depending on their scores. To score them, we calculate the number of queens on every row, column and diagonal and calculate the sum of max0, number - 1. In this way, penalty is added by the conflicting pairs of queens.

In order to introduce some new attempt of present entities, mutation is necessary. We simulate the mutation by allowing random chess to move one step up, down, left or right.

After finishing all process above, the algorithm goes to next iteration. Then do selection, crossover and mutation over and over again until max iteration or best answer.

2.3 Parallelization

Since we are generating a large population for each generation and each individual board is highly independent with the others, we want to use parallelization to speedup our algorithm. By observation, we can find several parts in our algorithm that can be parallelized.

For a new generation, we will need to generate a large number of boards through crossover. We use a random seed to choose the parent for a crossover, and the random seed can be drawn i.i.d with each other. As a result, we can compute the crossover in parallel. In our algorithm, we also have to keep track of the parent for each new board, thus we will zip the parent boards and offspring together, we also compute this part in parallel.

Another process we can parallelize is Mutation. Each grid for in a board is subject to a small chance to mutate, and the mutation does not require information from other process, thus we also parallelize this mutation process. Just like crossover, we need to keep track of the parent board, thus we do the zipWith in parallel.

Finding score for each board takes some time, and throughout iterations, we will do lots of redundant score calculations, thus we want to calculate the scores for each potential boards beforehand. The score for any board depends only on itself, thus we score all the boards in our pool in parallel to get rid of the redundant calculations.
3 Result

In this section, we will be presenting our experiment settings and results.

3.1 Experiment Setting

In our experiment, we set our population size to be 300 and we compare our results for 16-queens because for this setting, the computation will be expensive, and we will be able to see a clear difference between different number of cores we use. We set the number of iterations to be 500, because this will guarantee the same specific complexity for every experiment. Our experiment runs under WSL2-Ubuntu-20.04 on Windows with i7-1165G7 which has 4 cores 8 threads.

3.2 Result Comparison

<table>
<thead>
<tr>
<th>Cores</th>
<th>Time(s)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.198</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3.197</td>
<td>1.63</td>
</tr>
<tr>
<td>3</td>
<td>2.694</td>
<td>1.93</td>
</tr>
<tr>
<td>4</td>
<td>2.569</td>
<td>2.02</td>
</tr>
<tr>
<td>5</td>
<td>2.692</td>
<td>1.93</td>
</tr>
<tr>
<td>6</td>
<td>2.923</td>
<td>1.78</td>
</tr>
<tr>
<td>7</td>
<td>3.978</td>
<td>1.31</td>
</tr>
<tr>
<td>8</td>
<td>9.996</td>
<td>0.52</td>
</tr>
</tbody>
</table>

Table 1: Parallel result and relative speedup. N4 is the best but N8 is the worst.

In figure 1, we can see the real time with respect to number of cores used and the ideal curve computed with 1 core and 2 core. The 1-core case is the sequential genetic algorithm case, and we can see that the multi-core versions generally achieve an obvious speedup with the sequential version. We reached the optimal when we are using 4 cores, and with more than 4 cores, the curve goes up and we are not improving any more. A strange thing we observe is that for the 8-core case, the algorithm appears to run even slower than the sequential version.
3.3 Experiment analysis

It is quite interesting that the speedup is not closed to the ideal one when cores number is bigger than 3. What is more important, the speedup goes down when cores number is bigger than 4. Therefore we turned to check what is going on by Threadscope. From the statistical result showed in Table 2, we can easily find that the garbage collect time is huge when we use 8 cores. What is more, we zoom in the result from Threadscope when cores number equals to 4. It is not hard to notice that only one core is busy in most time. The others are always waiting. That may be because we only parallelize crossover, mutation and scoring seperately, but the iteration of these process is still sequential.

<table>
<thead>
<tr>
<th>Cores</th>
<th>Total Time(s)</th>
<th>GC Time(s)</th>
<th>Productivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5.13</td>
<td>1.03</td>
<td>79.9%</td>
</tr>
<tr>
<td>2</td>
<td>3.13</td>
<td>0.62</td>
<td>80.2%</td>
</tr>
<tr>
<td>3</td>
<td>2.63</td>
<td>0.58</td>
<td>78.0%</td>
</tr>
<tr>
<td>4</td>
<td>2.50</td>
<td>0.51</td>
<td>79.6%</td>
</tr>
<tr>
<td>5</td>
<td>2.62</td>
<td>0.55</td>
<td>78.9%</td>
</tr>
<tr>
<td>6</td>
<td>2.85</td>
<td>0.68</td>
<td>76.1%</td>
</tr>
<tr>
<td>7</td>
<td>3.90</td>
<td>1.11</td>
<td>71.5%</td>
</tr>
<tr>
<td>8</td>
<td>9.92</td>
<td>4.40</td>
<td>55.6%</td>
</tr>
</tbody>
</table>

Table 2: Statistical result from Threadscope
4 Future Work

As I mentioned above, the possible reason for bad performance is sequential iteration. One possible solution is to pipeline the iteration. However these processes are not strictly independent to each other. It may need more memory space to store the mediate population to make sure the correctness.

5 Source code

5.1 Genetic Algorithm

```haskell
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE MultiParamTypeClasses #-}

module GA (Entity(..),
            ScoredEntity,
            Archive,
            GAConfig(..),
            evolve,
            evolveVerbose,
            randomSearch) where

import Control.Monad.Par
import Control.Parallel.Strategies
import qualified Control.Monad.Par.Combinator as C
import Control.Monad (zipWithM)
import Control.Monad.IO.Class (MonadIO, liftIO)
import Data.List (sortBy, nub, nubBy)
import Data.Maybe (catMaybes, fromJust, isJust)
import Data.Ord (comparing)
import System.Directory (createDirectoryIfMissing, doesFileExist)
import System.Random (StdGen, mkStdGen, random, randoms)

-- | Currify a list of elements into tuples.
currify :: [a] -- ^ list
        -> [(a,a)] -- ^ list of tuples
currify (x:y:xs) = (x,y): currify xs
currify [] = []
currify [_] = error "(currify) ERROR: only one element left?!?"

-- | Take and drop elements of a list in a single pass.
takeAndDrop :: Int -- ^ number of elements to take/drop
            -> [a] -- ^ list
            -> ([a],[a]) -- ^ result: taken list element and rest of list
takeAndDrop n xs
  | n > 0    = let (hs,ts) = takeAndDrop (n-1) (tail xs)
            in (head xs:hs, ts)
  | otherwise = ([],xs)

-- | A scored entity.
type ScoredEntity e s = (Maybe s, e)

-- | Archive of scored entities.
type Archive e s = [ScoredEntity e s]

-- | Scored generation (population and archive).
type Generation e s = ([e], Archive e s)

-- | Universe of entities.
```
type Universe e = [e]

-- | Configuration for genetic algorithm.
data GAConfig = GAConfig {
  -- | population size
  getPopSize :: Int,
  -- | size of archive (best entities so far)
  getArchiveSize :: Int,
  -- | maximum number of generations to evolve
  getMaxGenerations :: Int,
  -- | fraction of entities generated by crossover (tip: >= 0.80)
  getCrossoverRate :: Float,
  -- | fraction of entities generated by mutation (tip: <= 0.20)
  getMutationRate :: Float,
  -- | parameter for crossover (semantics depend on crossover operator)
  getCrossoverParam :: Float,
  -- | parameter for mutation (semantics depend on mutation operator)
  getMutationParam :: Float,
  -- | enable/disable built-in checkpointing mechanism
  getWithCheckpointing :: Bool,
  -- | rescore archive in each generation?
  getRescoreArchive :: Bool
}

-- | Type class for entities that represent a candidate solution.
--
-- Five parameters:
--
-- * data structure representing an entity (e)
-- * score type (s), e.g. Double
-- * data used to score an entity, e.g. a list of numbers (d)
-- * some kind of pool used to generate random entities,
--   e.g. a Hoogle database (p)
-- * monad to operate in (m)
--
-- Minimal implementation should include 'genRandom', 'crossover', 'mutation',
-- and either 'score', 'score' or 'scorePop'.
--
-- The 'isPerfect', 'showGeneration' and 'hasConverged' functions are optional.
--
class (Eq e, Ord e, Read e, Show e,
  Ord s, Read s, Show s,
  Monad m) => Entity e s d p m where
  -- | Generate a random entity. [required]
genRandom :: p -- ^ pool for generating random entities
  -> Int -- ^ random seed
  -> m e -- ^ random entity

  -- | Crossover operator: combine two entities into a new entity. [required]
crossover :: p -- ^ entity pool
  -> Float -- ^ crossover parameter
  -> Int -- ^ random seed
-> e -- ^ first entity
-> e -- ^ second entity
-> m (Maybe e) -- ^ entity resulting from crossover

-- | Mutation operator: mutate an entity into a new entity. [required]
mutation :: p -- ^ entity pool
  -> Float -- ^ mutation parameter
  -> Int -- ^ random seed
  -> e -- ^ entity to mutate
  -> m (Maybe e) -- ^ mutated entity

-- | Score an entity (lower is better), pure version. [optional]
--
-- Overridden if score or scorePop are implemented.
score' :: d -- ^ dataset for scoring entities
  -> e -- ^ entity to score
  -> (Maybe s) -- ^ entity score
score' _ _ = error $ "(GA) score' is not defined, "
  ++ "nor is score or scorePop!"

-- | Score an entity (lower is better), monadic version. [optional]
--
-- Default implementation hoists score' into monad,
-- overridden if scorePop is implemented.
score :: d -- ^ dataset for scoring entities
  -> e -- ^ entity to score
  -> m (Maybe s) -- ^ entity score
score d e = do
  return $ score' d e

-- | Score an entire population of entities. [optional]
--
-- Default implementation returns Nothing,
-- and triggers indidual of entities.
scorePop :: d -- ^ dataset to score entities
  -> [e] -- ^ universe of known entities
  -> [e] -- ^ population of entities to score
  -> m (Maybe [Maybe s]) -- ^ scores for population entities
scorePop _ _ _ = return Nothing

-- | Determines whether a score indicates a perfect entity. [optional]
--
-- Default implementation returns always False.
isPerfect :: (e,s) -- ^ scored entity
  -> Bool -- ^ whether or not scored entity is perfect
isPerfect _ = False

-- | Show progress made in this generation. [optional]
--
-- Default implementation shows best entity.
showGeneration :: Int -- ^ generation index
  -> Generation e s -- ^ generation (population and archive)
  -> String -- ^ string describing this generation
showGeneration gi (_,archive) = "best entity (gen. "
  ++ show gi ++ "): " ++ (show e)
  ++ " [fitness: " ++ show fitness ++ "]"
  where
    (Just fitness, e) = head archive
-- | Determine whether evolution should continue or not, based on lists of archive fitnesses of previous generations.
-- Note: most recent archives are at the head of the list.
-- Default implementation always returns False.

hasConverged :: [Archive e s] -- ^ archives so far
  -> Bool -- ^ whether or not convergence was detected
  hasConverged _ = False

-- | Initialize: generate initial population.

initPop :: (Entity e s d p m) => p -- ^ pool for generating random entities
  -> Int -- ^ population size
  -> Int -- ^ random seed
  -> m [e] -- ^ initialized population

initPop pool n seed = do
  let g = mkStdGen seed
  seeds = take n $ randoms g
  entities <- mapM (genRandom pool) seeds
  return entities

-- | Binary tournament selection operator.

tournamentSelection :: (Ord s) => [ScoredEntity e s] -- ^ set of entities
  -> Int -- ^ random seed
  -> e -- ^ selected entity

tournamentSelection xs seed = if s1 < s2 then x1 else x2
  where
    len = length xs
    g = mkStdGen seed
    is = take 2 $ map (flip mod len) $ randoms g
    [(s1,x1),(s2,x2)] = map ((!!) xs) is

-- | Apply crossover to obtain new entities.

performCrossover :: (Entity e s d p m) => Float -- ^ crossover parameter
  -> Int -- ^ number of entities
  -> Int -- ^ random seed
  -> p -- ^ pool for combining entities
  -> [ScoredEntity e s] -- ^ entities
  -> m [e] -- ^ combined entities

performCrossover p n seed pool es = do
  let g = mkStdGen seed
  (selSeeds,seeds) = takeAndDrop (2*2*n) $ randoms g
  (crossSeeds,_) = takeAndDrop (2*n) seeds
  tuples = currify $ map (tournamentSelection es) selSeeds
  resEntities <- sequenceA (zipWith ($) (map (uncurry . (crossover pool p)) crossSeeds 'using'
                           parList rpar)
                           tuples 'using' parList rpar)
  return $ take n $ catMaybes $ resEntities

-- | Apply mutation to obtain new entities.

performMutation :: (Entity e s d p m) => Float -- ^ mutation parameter
  -> Int -- ^ number of entities
  -> Int -- ^ random seed
  -> p -- ^ pool for mutating entities
  -> [ScoredEntity e s] -- ^ entities
  -> m [e] -- ^ mutated entities

performMutation p n seed pool es = do
  let g = mkStdGen seed

8
(selSeeds, seeds) = takeAndDrop (2*n) $ randoms g
(mutSeeds, _) = takeAndDrop (2*n) seeds
resEntities <- sequenceA ($)
    (map (mutation pool p) mutSeeds ‘using‘ parList rpar)
    (map (tournamentSelection es) selSeeds ‘using‘ parList rpar)
‘using‘ parList rpar)
return $ take n $ catMaybes $ resEntities

-- |Score a list of entities.
scoreAll :: (Entity e s d p m) => d -- ^ dataset for scoring entities
        -> [e] -- ^ universe of known entities
        -> [e] -- ^ set of entities to score
        -> m [Maybe s]
scoreAll dataset univEnts ents = do
    scores <- scorePop dataset univEnts ents
    case scores of
        (Just ss) -> return ss
        Nothing -> return (map (score ' dataset) ents 'using‘ parList rpar)

-- |Function to perform a single evolution step:
-- * score all entities in the population
-- * combine with best entities so far (archive)
-- * sort by fitness
-- * create new population using crossover/mutation
-- retain best scoring entities in the archive
evolutionStep :: (Entity e s d p m) => p -- ^ pool for crossover/mutation
        -> d -- ^ dataset for scoring entities
        -> (Int, Int, Int) -- ^ # of c/m/a entities
        -> (Float, Float) -- ^ c/m parameters
        -> Bool -- ^ rescore archive in each step?
        -> Universe e -- ^ known entities
        -> Generation e s -- ^ current generation
        -> Int -- ^ seed for next generation
        -> m (Universe e, Generation e s)
        -- ^ renewed universe, next generation
evolutionStep pool
dataset
(crossPar, mutPar)
rescoreArchive
universe
(pop, archive)
seed = do
    -- score population
    -- try to score in a single go first
    scores <- scoreAll dataset universe pop
    archive’ <- if rescoreArchive
    then return archive
    else do
        let as = map snd archive
        scores’ <- scoreAll dataset universe as
        return $ zip scores’ as
    let scoredPop = zip scores pop
-- combine with archive for selection
combo = scoredPop ++ archive

-- split seeds for crossover/mutation selection/seeds

-- apply crossover and mutation

let -- new population: crossed + mutated entities
newPop = crossEnts ++ mutEnts

-- new archive: best entities so far
newArchive = take an

newUniverse = nub $ universe ++ pop

return (newUniverse, (newPop, newArchive))

-- | Evolution: evaluate generation and continue.

-- | Generate file name for checkpoint.

-- Checkpoint a single generation.
checkpointGen cfg index seed (pop,archive) = do
  let txt = show $(pop,archive)
  fn = chkptFileName cfg (index,seed)
  putStrLn $ "writing checkpoint for gen "
    ++ (show index) ++ " to " ++ fn
  createDirectoryIfMissing True "checkpoints"
  writeFile fn txt

-- | Evolution: evaluate generation, (maybe) checkpoint, continue.
evolutionVerbose :: (Entity e s d p m, MonadIO m) => GAConfig -- ^ configuration for GA
                      -> Universe e -- ^ universe of known entities
                      -> [ Archive e s ] -- ^ previous archives
                      -> Generation e s -- ^ current generation
                      -> ( Universe e
                      -> Generation e s
                      -> Int ) -- ^ function that evolves a generation
                      -> [(Int,Int)] -- ^ gen indicies and seeds
                      -> m (Generation e s) -- ^ evolved generation
                      -> [(Int,Int)]
                      -> m (Generation e s)
                      -> m (Generation e s)
evolutionVerbose cfg universe pastArchives gen step ((gi , seed ): gss ) = do
  ( universe', newPa@ (_, archive ')) <- step universe gen seed
  -- checkpoint generation if desired
  liftIO $ if (getWithCheckpointing cfg )
  then checkpointGen cfg gi seed newPa
  else return () -- skip checkpoint
  liftIO $ putStrLn $ showGeneration gi newPa

  -- check for perfect entity
  if hasConverged pastArchives || isPerfect (e,fitness)
    then do
      liftIO $ putStrLn $ if isPerfect (e,fitness)
      then "perfect entity found, "
        ++ "finished after " ++ show gi
        ++ " generations!"
      else "no progress for 3 generations, "
        ++ "stopping after " ++ show gi
        ++ " generations!"
      return newPa
    else evolutionVerbose cfg universe' (archive':pastArchives ) newPa step gss
  -- no more gen. indices/seeds => quit
  evolutionVerbose _ _ _ gen _ [] = do
    liftIO $ putStrLn $ "done evolving!"
    return gen

-- | Initialize.
initGA :: (Entity e s d p m) => StdGen -- ^ random generator
       -> GAConfig -- ^ configuration for GA
       -> p -- ^ pool for generating random entities
       -> m ([e],Int,Int,Float,Float,[(Int,Int)])
       -- ^ initialization result
initGA g cfg pool = do
-- generate list of random integers
  let (seed:rs) = randoms g :: [Int]
ps = getPopSize cfg
-- initial population
pop <- initPop pool ps seed
let -- number of entities generated by crossover/mutation
    cCnt = round \$ (getCrossoverRate cfg) * (fromIntegral ps)
    mCnt = round \$ (getMutationRate cfg) * (fromIntegral ps)
-- archive size
    aSize = getArchiveSize cfg
-- crossover/mutation parameters
    crossPar = getCrossoverParam cfg
    mutPar = getMutationParam cfg
-- seeds for evolution
    seeds = take (getMaxGenerations cfg) rs
-- seeds per generation
    genSeeds = zip [0..] seeds
return (pop, cCnt, mCnt, aSize, crossPar, mutPar, genSeeds)

-- Do the evolution!
evolve :: (Entity e s d p m) => StdGen -- random generator
    -> GAConfig -- configuration for GA
    -> p -- random entities pool
    -> d -- dataset required to score entities
    -> m (Archive e s) -- best entities
    -> evolve g cfg pool dataset = do
        -- initialize
        (pop, cCnt, mCnt, aSize, crossPar, mutPar, genSeeds) <- if not (getWithCheckpointing cfg)
            then initGA g cfg pool
            else error "$ (evolve) No checkpointing support "$ ++ "(requires liftIO); see evolveVerbose."
        -- do the evolution
        let rescoreArchive = getRescoreArchive cfg
            (_, resArchive) <- evolution (pop,[]) (evolutionStep pool dataset (cCnt, mCnt, aSize)
                (crossPar, mutPar)
                rescoreArchive)
        -- return best entity
        return resArchive

-- Try to restore from checkpoint.
--
restoreFromChkpt :: (Entity e s d p m) => GAConfig -- configuration for GA
                    -> [(Int,Int)] -- gen indices/seeds
                    -> IO (Maybe (Int,Generation e s))
    -- restored generation (if any)
    -> restoreFromChkpt cfg ((gi,seed):genSeeds) = do
        chkptFound <- doesFileExist fn
        if chkptFound
            then do
                txt <- readFile fn
                return $ Just (gi, read txt)
            else restoreFromChkpt cfg genSeeds
        where
            fn = chkptFileName cfg (gi,seed)
        restoreFromChkpt _ [] = return Nothing
-- Do the evolution, verbosely.
-- Prints progress to stdout, and supports checkpointing.
-- Note: requires support for liftIO in monad used.
evolveVerbose :: (Entity e s d p m, MonadIO m)
  => StdGen -- ^ random generator
  -> GAConfig -- ^ configuration for GA
  -> p -- ^ random entities pool
  -> d -- ^ dataset required to score entities
  -> m (Archive e s) -- ^ best entities
evolveVerbose g cfg pool dataset = do
  -- initialize
  (pop, cCnt, mCnt, aSize,
   crossPar, mutPar, genSeeds) <- initGA g cfg pool
  let checkpointing = getWithCheckpointing cfg
  -- (maybe) restore from checkpoint
  restored <- liftIO $ if checkpointing
                 then restoreFromChkpt cfg (reverse genSeeds)
                 else return Nothing
  let (gi, gen) = if isJust restored
                  -- restored pop/archive from checkpoint
                  then fromJust restored
                  -- restore failed, new population and empty archive
                  else (-1, (pop, []))
                  -- filter out seeds from past generations
                  genSeeds' = filter (> gi) . fst) genSeeds
                  rescoreArchive = getRescoreArchive cfg
  -- do the evolution
  (_, resArchive) <- evolutionVerbose
  cfg [] [] gen
  (evolutionStep pool dataset
   (cCnt,mCnt,aSize)
   (crossPar,mutPar)
   rescoreArchive)
  genSeeds'
  -- return best entity
  return resArchive

-- Random searching.
-- Useful to compare with results from genetic algorithm.
randomSearch :: (Entity e s d p m) => StdGen -- ^ random generator
  -> Int -- ^ number of random entities
  -> p -- ^ random entity pool
  -> d -- ^ scoring dataset
  -> m (Archive e s) -- ^ scored entities (sorted)
r
randomSearch g n pool dataset = do
  let seed = fst $ random g :: Int
  es <- initPop pool n seed
  scores <- scoreAll dataset [] es
  return $ nubBy (\x y -> comparing snd x y == EQ)
  $ sortBy (comparing fst)
  $ zip scores es

5.2 8-queens
import Data.Map (fromListWith, toList)
import System.Random (mkStdGen, random, randoms)
import System.IO (IOMode (..), hClose, hGetContents, openFile)

import GA (Entity (..), GAConfig (..), evolve, evolveVerbose, randomSearch)

queensNum = 8
count [] = []
count ((a,b):xs) = (b - 1) : (count xs)

bound :: Int -> Int
bound x = (x `mod` queensNum) + 1

type Location = (Int, Int)
type Board = [(Int, Int)]
type Target = [(Int, Int)]

instance Entity Board Int () [Location] IO where
  genRandom pool seed = return $ take queensNum $ map (!!!) pool
  where
    g = mkStdGen seed
    k = length pool
    is = map (flip mod k) $ randoms g
  crossover _ _ seed e1 e2 = return $ Just e
  where
    g = mkStdGen seed
    cps = zipWith \(x y -> [x,y]) e1 e2
    picks = map (flip mod 2) $ randoms g
    e = zipWith (!!!) cps picks
  mutation _ _ seed e = return $ Just $(zipWith replace tweaks e)
  where
    g = mkStdGen seed
    tweaks = randoms g :: [Int]
    replace i x = if (2 `mod` 2) == 0
      then case (i `mod` 4) of
        1 -> (bound (pred $ fst x), bound (pred $ snd x))
        2 -> (bound (pred $ fst x), bound (succ $ snd x))
        3 -> (bound (succ $ fst x), bound (succ $ snd x))
        0 -> (bound (succ $ fst x), bound (pred $ snd x))
        _ -> error "crossover: unknown case"
    else x

  score _ e = Just $ fromIntegral (row + column + diagonal)
  where
    rowfreq e = toList (fromListWith (+) [(snd(x),1) | x <- e])
    columnfreq e = toList (fromListWith (+) [(fst(x),1) | x <- e])
    freq e = toList (fromListWith (+) [(x,1) | x <- e])
    row = sum $ count $ rowfreq e
    column = sum $ count $ columnfreq e
    diagonal = (sum $ count $ freq $ map (\(a,b) -> a-b) e) + (sum $ count $ freq $ map (\(a,b) -> a+b) e)
isPerfect (_, s) = s == 0

main :: IO()
main = do
    let cfg = GAConfig
        200 -- population size
        30 -- archive size (best entities to keep track of)
        100 -- maximum number of generations
        0.8 -- crossover rate (% of entities by crossover)
        0.4 -- mutation rate (% of entities by mutation)
        0.0 -- parameter for crossover (not used here)
        0.4 -- parameter for mutation (% of replaced letters)
        False -- whether or not to use checkpointing
        False -- don’t rescore archive in each generation

    g = mkStdGen 0 -- random generator

    chessPool = [(a,b)| a <- [1..queensNum], b <- [1..queensNum]]
    -- Do the evolution
    es <- evolveVerbose g cfg chessPool ()
    let e = snd $ head es :: Board

    putStrLn $ "best entity: " ++ (show e)