Genetic-8-Queens

Genetic Algorithm for 8-Queens

Zhangyi Pan(zp2223), Shuhong Zhang(sz2949)

December 21, 2021

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 parallelizaiton 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.

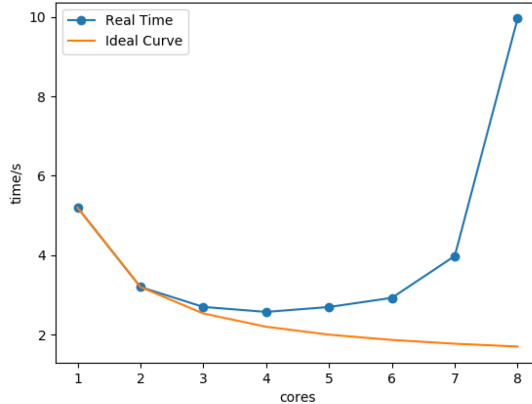

Figure 1: Parallel results and Ideal Curve

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

Cores	Time(s)	Speedup
1	5.198	1
2	3.197	1.63
3	2.694	1.93
4	2.569	2.02
5	2.692	1.93
6	2.923	1.78
7	3.978	1.31
8	9.996	0.52

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.

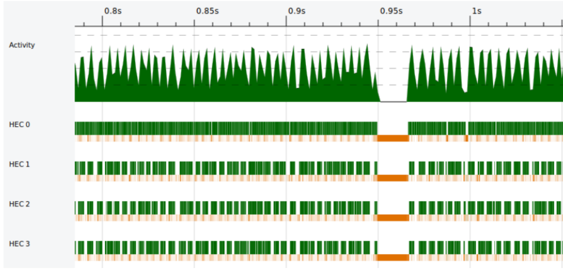

Figure 2: Threadscope result for N4

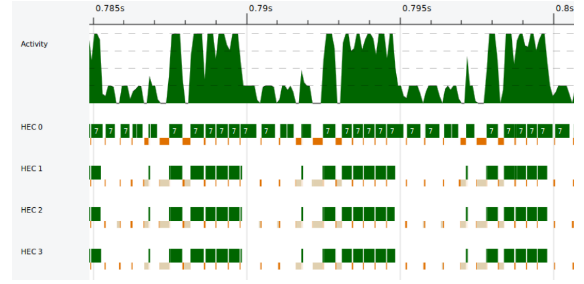

Figure 3: Same result but in more finer scope

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.

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

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 process 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

```
{-# 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.
```

```haskell
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
    | e -> s, e -> d, e -> p, e -> 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
```

```haskell
            -> 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,
-- overriden 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 entites. [optional]
--
-- Default implementation returns Nothing,
-- and triggers indivual 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.
--
-- 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

```

```haskell
166   -- |Determine whether evolution should continue or not,
167   --  based on lists of archive fitnesses of previous generations.
168   --
169   --  Note: most recent archives are at the head of the list.
170   --
171   --  Default implementation always returns False.
172   hasConverged :: [Archive e s] -- ^ archives so far
173                -> Bool -- ^ whether or not convergence was detected
174   hasConverged _ = False
175 
176 -- |Initialize: generate initial population.
177 initPop :: (Entity e s d p m) => p -- ^ pool for generating random entities
178                             -> Int -- ^ population size
179                             -> Int -- ^ random seed
180                             -> m [e] -- ^ initialized population
181 initPop pool n seed = do
182                          let g = mkStdGen seed
183                              seeds = take n $ randoms g
184                          entities <- mapM (genRandom pool) seeds
185                          return entities
186 
187 -- |Binary tournament selection operator.
188 tournamentSelection :: (Ord s) => [ScoredEntity e s] -- ^ set of entities
189                                -> Int -- ^ random seed
190                                -> e -- ^ selected entity
191 tournamentSelection xs seed = if s1 < s2 then x1 else x2
192   where
193     len = length xs
194     g = mkStdGen seed
195     is = take 2 $ map (flip mod len) $ randoms g
196     [(s1,x1),(s2,x2)] = map ((!!) xs) is
197 
198 -- |Apply crossover to obtain new entites.
199 performCrossover :: (Entity e s d p m) => Float -- ^ crossover parameter
200                                      -> Int -- ^ number of entities
201                                      -> Int -- ^ random seed
202                                      -> p -- ^ pool for combining entities
203                                      -> [ScoredEntity e s] -- ^ entities
204                                      -> m [e] -- combined entities
205 performCrossover p n seed pool es = do
206     let g = mkStdGen seed
207         (selSeeds,seeds) = takeAndDrop (2*2*n) $ randoms g
208         (crossSeeds,_) = takeAndDrop (2*n) seeds
209         tuples = currify $ map (tournamentSelection es) selSeeds
210     resEntities <- sequenceA (zipWith ($)
211                      (map (uncurry . (crossover pool p)) crossSeeds `using`
    parList rpar)
212                      tuples `using` parList rpar)
213     return $ take n $ catMaybes $ resEntities
214 
215 -- |Apply mutation to obtain new entites.
216 performMutation :: (Entity e s d p m) => Float -- ^ mutation parameter
217                                     -> Int -- ^ number of entities
218                                     -> Int -- ^ random seed
219                                     -> p -- ^ pool for mutating entities
220                                     -> [ScoredEntity e s] -- ^ entities
221                                     -> m [e] -- mutated entities
222 performMutation p n seed pool es = do
223     let g = mkStdGen seed
```

```
224         (selSeeds,seeds) = takeAndDrop (2*n) $ randoms g
225         (mutSeeds,_) = takeAndDrop (2*n) seeds
226     resEntities <- sequenceA (zipWith ($)
227                       (map (mutation pool p) mutSeeds `using` parList rpar)
228                       (map (tournamentSelection es) selSeeds `using` parList rpar)
    `using` parList rpar)
229     return $ take n $ catMaybes $ resEntities
230
231 -- |Score a list of entities.
232 scoreAll :: (Entity e s d p m) => d -- ^ dataset for scoring entities
233                                -> [e] -- ^ universe of known entities
234                                -> [e] -- ^ set of entities to score
235                                -> m [Maybe s]
236 scoreAll dataset univEnts ents = do
237   scores <- scorePop dataset univEnts ents
238   case scores of
239     (Just ss) -> return ss
240     -- score one by one if scorePop failed
241     Nothing   -> return (map (score' dataset) ents `using` parList rpar)
242
243 -- |Function to perform a single evolution step:
244 --
245 -- * score all entities in the population
246 --
247 -- * combine with best entities so far (archive)
248 --
249 -- * sort by fitness
250 --
251 -- * create new population using crossover/mutation
252 --
253 -- * retain best scoring entities in the archive
254 evolutionStep :: (Entity e s d p m) => p -- ^ pool for crossover/mutation
255                                     -> d -- ^ dataset for scoring entities
256                                     -> (Int,Int,Int) -- ^ # of c/m/a entities
257                                     -> (Float,Float) -- ^ c/m parameters
258                                     -> Bool -- ^ rescore archive in each step?
259                                     -> Universe e -- ^ known entities
260                                     -> Generation e s -- ^ current generation
261                                     -> Int -- ^ seed for next generation
262                                     -> m (Universe e, Generation e s)
263                                        -- ^ renewed universe, next generation
264 evolutionStep pool
265               dataset
266               (cn,mn,an)
267               (crossPar,mutPar)
268               rescoreArchive
269               universe
270               (pop,archive)
271               seed = do
272     -- score population
273     -- try to score in a single go first
274     scores <- scoreAll dataset universe pop
275     archive' <- if rescoreArchive
276       then return archive
277       else do
278         let as = map snd archive
279         scores' <- scoreAll dataset universe as
280         return $ zip scores' as
281     let scoredPop = zip scores pop
```

```
282         -- combine with archive for selection
283         combo = scoredPop ++ archive'
284         -- split seeds for crossover/mutation selection/seeds
285         g = mkStdGen seed
286         [crossSeed,mutSeed] = take 2 $ randoms g
287     -- apply crossover and mutation
288     crossEnts <- performCrossover crossPar cn crossSeed pool combo
289     mutEnts <- performMutation mutPar mn mutSeed pool combo
290     let -- new population: crossovered + mutated entities
291         newPop = crossEnts ++ mutEnts
292         -- new archive: best entities so far
293         newArchive = take an
294                      $ nubBy (\x y -> comparing snd x y == EQ)
295                      $ sortBy (comparing fst) combo
296         newUniverse = nub $ universe ++ pop
297     return (newUniverse, (newPop,newArchive))
298
299 -- |Evolution: evaluate generation and continue.
300 evolution :: (Entity e s d p m) => GAConfig -- ^ configuration for GA
301                                 -> Universe e -- ^ known entities
302                                 -> [Archive e s] -- ^ previous archives
303                                 -> Generation e s -- ^ current generation
304                                 -> (   Universe e
305                                     -> Generation e s
306                                     -> Int
307                                     -> m (Universe e, Generation e s)
308                                    ) -- ^ function that evolves a generation
309                                 -> [(Int,Int)] -- ^ gen indicies and seeds
310                                 -> m (Generation e s) -- ^evolved generation
311 evolution cfg universe pastArchives gen step ((_,seed):gss) = do
312     (universe',nextGen) <- step universe gen seed
313     let (Just fitness, e) = (head $ snd nextGen)
314         newArchive = snd nextGen
315     if hasConverged pastArchives || isPerfect (e,fitness)
316       then return nextGen
317       else evolution cfg universe' (newArchive:pastArchives) nextGen step gss
318 -- no more gen. indices/seeds => quit
319 evolution _ _ _ gen _ [] = return gen
320
321 -- |Generate file name for checkpoint.
322 chkptFileName :: GAConfig -- ^ configuration for generation algorithm
323               -> (Int,Int) -- ^ generation index and random seed
324               -> FilePath -- ^ path of checkpoint file
325 chkptFileName cfg (gi,seed) = "checkpoints/GA-"
326                            ++ cfgTxt ++ "-gen"
327                            ++ (show gi) ++ "-seed-"
328                            ++ (show seed) ++ ".chk"
329   where
330     cfgTxt = (show $ getPopSize cfg) ++ "-" ++
331              (show $ getArchiveSize cfg) ++ "-" ++
332              (show $ getCrossoverRate cfg) ++ "-" ++
333              (show $ getMutationRate cfg) ++ "-" ++
334              (show $ getCrossoverParam cfg) ++ "-" ++
335              (show $ getMutationParam cfg)
336
337 -- |Checkpoint a single generation.
338 checkpointGen :: (Entity e s d p m) => GAConfig -- ^ configuraton for GA
339                                     -> Int -- ^ generation index
340                                     -> Int -- ^ random seed for generation
```

```
341                                     -> Generation e s -- ^ current generation
342                                     -> IO() -- ^ writes to file
343 checkpointGen cfg index seed (pop,archive) = do
344     let txt = show $ (pop,archive)
345         fn = chkptFileName cfg (index,seed)
346     putStrLn $ "writing checkpoint for gen "
347             ++ (show index) ++ " to " ++ fn
348     createDirectoryIfMissing True "checkpoints"
349     writeFile fn txt
350 
351 -- |Evolution: evaluate generation, (maybe) checkpoint, continue.
352 evolutionVerbose :: (Entity e s d p m,
353                    MonadIO m) => GAConfig -- ^ configuration for GA
354                               -> Universe e -- ^ universe of known entities
355                               -> [Archive e s] -- ^ previous archives
356                               -> Generation e s -- ^ current generation
357                               -> (   Universe e
358                                   -> Generation e s
359                                   -> Int
360                                   -> m (Universe e, Generation e s)
361                                  ) -- ^ function that evolves a generation
362                               -> [(Int,Int)] -- ^ gen indicies and seeds
363                               -> m (Generation e s) -- ^ evolved generation
364 evolutionVerbose cfg universe pastArchives gen step ((gi,seed):gss) = do
365     (universe',newPa@(_,archive')) <- step universe gen seed
366     let (Just fitness, e) = head archive'
367     -- checkpoint generation if desired
368     liftIO $ if (getWithCheckpointing cfg)
369       then checkpointGen cfg gi seed newPa
370       else return () -- skip checkpoint
371     liftIO $ putStrLn $ showGeneration gi newPa
372     -- check for perfect entity
373     if hasConverged pastArchives || isPerfect (e,fitness)
374        then do
375                liftIO $ putStrLn $ if isPerfect (e,fitness)
376                                      then    "perfect entity found, "
377                                           ++ "finished after " ++ show gi
378                                           ++ " generations!"
379                                      else    "no progress for 3 generations, "
380                                           ++ "stopping after " ++ show gi
381                                           ++ " generations!"
382                return newPa
383        else evolutionVerbose cfg universe' (archive':pastArchives) newPa step gss
384 
385 -- no more gen. indices/seeds => quit
386 evolutionVerbose _ _ _ gen _ [] = do
387     liftIO $ putStrLn $ "done evolving!"
388     return gen
389 
390 -- |Initialize.
391 initGA :: (Entity e s d p m) => StdGen   -- ^ random generator
392                             -> GAConfig -- ^ configuration for GA
393                             -> p -- ^ pool for generating random entities
394                             -> m ([e],Int,Int,Int,
395                                   Float,Float,[(Int,Int)]
396                                  ) -- ^ initialization result
397 initGA g cfg pool = do
398     -- generate list of random integers
399     let (seed:rs) = randoms g :: [Int]
```

```
400         ps = getPopSize cfg
401     -- initial population
402     pop <- initPop pool ps seed
403     let -- number of entities generated by crossover/mutation
404         cCnt = round $ (getCrossoverRate cfg) * (fromIntegral ps)
405         mCnt = round $ (getMutationRate cfg) * (fromIntegral ps)
406         -- archive size
407         aSize = getArchiveSize cfg
408         -- crossover/mutation parameters
409         crossPar = getCrossoverParam cfg
410         mutPar = getMutationParam cfg
411         --  seeds for evolution
412         seeds = take (getMaxGenerations cfg) rs
413         -- seeds per generation
414         genSeeds = zip [0..] seeds
415     return (pop, cCnt, mCnt, aSize, crossPar, mutPar, genSeeds)
416
417 -- |Do the evolution!
418 evolve :: (Entity e s d p m) => StdGen -- ^ random generator
419                              -> GAConfig -- ^ configuration for GA
420                              -> p -- ^ random entities pool
421                              -> d -- ^ dataset required to score entities
422                              -> m (Archive e s) -- ^ best entities
423 evolve g cfg pool dataset = do
424     -- initialize
425     (pop, cCnt, mCnt, aSize,
426      crossPar, mutPar, genSeeds) <- if not (getWithCheckpointing cfg)
427        then initGA g cfg pool
428        else error $ "(evolve) No checkpointing support "
429                  ++ "(requires liftIO); see evolveVerbose."
430     -- do the evolution
431     let rescoreArchive = getRescoreArchive cfg
432     (_,resArchive) <- evolution
433                       cfg [] [] (pop,[])
434                       (evolutionStep pool dataset
435                                      (cCnt,mCnt,aSize)
436                                      (crossPar,mutPar)
437                                      rescoreArchive   )
438                       genSeeds
439     -- return best entity
440     return resArchive
441
442 -- |Try to restore from checkpoint.
443 --
444 -- First checkpoint for which a checkpoint file is found is restored.
445 restoreFromChkpt :: (Entity e s d p m) => GAConfig -- ^ configuration for GA
446                                        -> [(Int,Int)] -- ^ gen indices/seeds
447                                        -> IO (Maybe (Int,Generation e s))
448                                           -- ^ restored generation (if any)
449 restoreFromChkpt cfg ((gi,seed):genSeeds) = do
450     chkptFound <- doesFileExist fn
451     if chkptFound
452       then do
453         txt <- readFile fn
454         return $ Just (gi, read txt)
455       else restoreFromChkpt cfg genSeeds
456   where
457     fn = chkptFileName cfg (gi,seed)
458 restoreFromChkpt _ [] = return Nothing
```

```
459
460 -- |Do the evolution, verbosely.
461 --
462 -- Prints progress to stdout, and supports checkpointing.
463 --
464 -- Note: requires support for liftIO in monad used.
465 evolveVerbose :: (Entity e s d p m, MonadIO m)
466                              => StdGen -- ^ random generator
467                              -> GAConfig -- ^ configuration for GA
468                              -> p -- ^ random entities pool
469                              -> d -- ^ dataset required to score entities
470                              -> m (Archive e s) -- ^ best entities
471 evolveVerbose g cfg pool dataset = do
472     -- initialize
473     (pop, cCnt, mCnt, aSize,
474      crossPar, mutPar, genSeeds) <- initGA g cfg pool
475     let checkpointing = getWithCheckpointing cfg
476     -- (maybe) restore from checkpoint
477     restored <- liftIO $ if checkpointing
478       then restoreFromChkpt cfg (reverse genSeeds)
479       else return Nothing
480     let (gi,gen) = if isJust restored
481            -- restored pop/archive from checkpoint
482            then fromJust restored
483            -- restore failed, new population and empty archive
484            else (-1, (pop, []))
485         -- filter out seeds from past generations
486         genSeeds' = filter ((>gi) . fst) genSeeds
487         rescoreArchive = getRescoreArchive cfg
488     -- do the evolution
489     (_,resArchive) <- evolutionVerbose
490                         cfg [] [] gen
491                         (evolutionStep pool dataset
492                                        (cCnt,mCnt,aSize)
493                                        (crossPar,mutPar)
494                                        rescoreArchive)
495                                        genSeeds'
496     -- return best entity
497     return resArchive
498
499 -- |Random searching.
500 --
501 -- Useful to compare with results from genetic algorithm.
502 randomSearch :: (Entity e s d p m) => StdGen -- ^ random generator
503                                    -> Int -- ^ number of random entities
504                                    -> p -- ^ random entity pool
505                                    -> d -- ^ scoring dataset
506                                    -> m (Archive e s) -- ^ scored entities (sorted
    )
507 randomSearch g n pool dataset = do
508     let seed = fst $ random g :: Int
509     es <- initPop pool n seed
510     scores <- scoreAll dataset [] es
511     return $ nubBy (\x y -> comparing snd x y == EQ)
512            $ sortBy (comparing fst)
513            $ zip scores es
```

5.2 8-queens

```haskell
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE TypeSynonymInstances #-}

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) is
        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)

```

```
59     isPerfect (_,s) = s == 0
60
61 main :: IO()
62 main = do
63         let cfg = GAConfig
64                     200 -- population size
65                     30 -- archive size (best entities to keep track of)
66                     100 -- maximum number of generations
67                     0.8 -- crossover rate (% of entities by crossover)
68                     0.4 -- mutation rate (% of entities by mutation)
69                     0.0 -- parameter for crossover (not used here)
70                     0.4 -- parameter for mutation (% of replaced letters)
71                     False -- whether or not to use checkpointing
72                     False -- don't rescore archive in each generation
73
74             g = mkStdGen 0 -- random generator
75
76             chessPool = [(a,b)| a <- [1..queensNum], b <- [1..queensNum]]
77         -- Do the evolution
78         es <- evolveVerbose g cfg chessPool ()
79         let e = snd $ head es :: Board
80
81         putStrLn $ "best entity: " ++ (show e)
```