COSC343 Assignment 2 - Report

Reuben Crimp

May 14, 2014

Genetic Algorithm

My Genetic Algorithm is fairly simple, there are a few nice optimization, but it's essentially just a bog standard regular Genetic Algorithm.

There are several values which dictate the overall performance, each will initialise with a default value if none are given, which is given below in brackets.

- Board Size - (8) size of the chess board.
- Population Size - (50) size of the population.
- Max Generations - (60) number of generations before a restart.
- Max Restarts - (10) number of restarts before failure.
- Keep Ratio - (0.25) percent of the population to preserve for the next generation.
- Mutation P - (0.001) probability that a generated children will mutate.

The settings can be changed without compiling by calling passing the values as args, the program will use default values if no args are given.

Initial Population

My method ensures that each element in the initial population has no vertical or horizontal collisions whilst maintaing uniform variance. This means that the GA can find a solution very fast as each element starts off very fit.

Vertical collsions are impossble when using the recomended data structure. board permutations with horizontal collisions are omitted by ensuring every number from 1 to 8 appears in the data structure, i.e. every column is occupied by 1 and only 1 queen.

Population Stagnation

If every elements in the population converges to the same element, we essentially start breeding clones only to get more clones. The decreasing variance in the population means deriving a solution may never happen.

My method combats the likely decrease in varaiance in 4 ways.

1 Mutatation

For each child that we add to our populatin there is a chance that the any of the values in the array will "mutate" to any random value between 1-8. This increasing variance in individual elements.

2 Random Immigrants

We can at any point add new elements to the population, since new elements are created with uniform variance, adding will almost always increase the variance of the population. Exactly when and where we add them is described below.

3 Don't Breed Clones

Before breeding parents, check how similar they are, if they are identical then we replace one with a random immigrant. We can now breed the two parents with a greatly decreased chance of introducing a clone to the population.

4 Restart

If after a certain number of generations a solution has yet to be found, reset the entire population. So if any of the above methods fail to prevent stagnation, then we can start again.

This is more of a fail safe and does not usually get executed.

Crossover Method

The final method uses uniform crossover method to breed parents.

Each of the 8 queens in the child are independently chosen from either of the given parents with a random probability.

Whilst other methods were just as good in most test cases, this method continued to work when the population size was drasitcally reduced.

Parent Selection Method

The final selection method first ranks all the elements by fitness, then random parents are chosen from the top portion (the fitter elements), where the top portions size is calculated fromt the "keep rate".

Experiments

I experimented heavily on individual units, as well as tweaking the settings given above.

Crossover Methods

I tested several crossover methods, the tests were conducted with a ranked round robin parent selection. The test was deemed a failure if the generations exceeded 1,000,000 without finding a solution.

parents	method	failure rate
2	single random crossover	0.45028
2	single fitness weighted crossover	0.28652
2	uniform crossover (p = 0.5)	0.35567
2	uniform crossover (p = random)	0.24504
3	2 random crossover points	0.34903
8	each parent contributes only once	0.54120

Parent Selection Methods

I tried several methods of parent selection

For most methods I fitst sorted the population by fitness, this allowed the parent selection method to choose parents base on their relative fitness with ease, like a mock tournament.

purely stochastic, any random parent

round robin, i.e. every single unorderd pair then replacing the lower half of the sorted population with the best children. succesive pairs, chosen from the sorted population which would breed pairs with exact fitness. interleaving pairs, i.e choosing only odd parents from the sorted list which would breed pairs with similar fitness. choosing pairs which are seperated by half of the populaton in the sorted list which ensures one parent far more fit than the other.

successive pairs almost always failed, especialy without population stagnation protection, I didn't bother officially testing it.

interleaving pairs performed surprisingly well, same with

Best Solution

The final program can find a solution after 0 generations, but it usually takes on average 23.9 generations with the default values.

```
import java.util.*;
import java.lang.Math;

public class Nqueen{
  /* settings, with default values */
  public static int BOARD_SIZE = 8;
  public static int POPULATION = 50;
  public static int MAX_GENERATIONS = 60;
  public static int MAX_RESTARTS = 10;
  public static double KEEP_RATIO = 0.25;
  public static double MUTATION_P = 0.001;

  public static List<Board> boards;

  /* main */
  public static void main(String[] args){
    int c = 0;
    while(c < args.length) { //read args in pairs
      String temp = args[c++];
      if(c < args.length)
      parseArg(temp, args[c++]); //parse the pair of args
    }
    System.err.println("Begining the Genetic Algorithm with
       the following settings:" +
    "\nboard size:  " + BOARD_SIZE +
    "\npopulation:  " + POPULATION +
    "\ngenerations: " + MAX_GENERATIONS +
    "\nrestarts:    " + MAX_RESTARTS +
    "\nkeep ratio:  " + KEEP_RATIO +
    "\nmutation p:  " + MUTATION_P + "\n");

      /*************************************************/
      int generations = 0;
      int resets = 0;

      boards = new ArrayList<Board>();
      for(int i = 0; i < POPULATION; i++){
        boards.add(new Board()); // fill the list with
           random elements
      }

      while(getBest().fitness > 0){
        /* reorder the list before breeding, optional */
        sort();
        // shuffle();

        /* breed the elements */
        breedRandom();
        // breedPairs();

        /* if we exceed the generation limit restart */
        if(generations++ > MAX_GENERATIONS && resets < 10){
          resetAll();
```

```
        generations = 0;
        resets++;
      }
    }

    /* print the best board when done */
    System.err.println(getBest().toStringPretty());
    System.err.print("generations: ");
    System.out.println((resets*MAX_GENERATIONS +
       generations));
}

/* resets all elements in the population back to random
   elements */
public static void resetAll(){
  for(int i = 0; i < POPULATION; i++)
  boards.set(i, new Board());
}

/* chooses successive pairs of elements in the list to
   breed */
public static void breedPairs(){
  int numParents =
     Math.max(2,(int)(KEEP_RATIO*POPULATION));
  int numChildren = POPULATION - numParents;
  for(int i = 0; i < numChildren; i++){
    boards.set(numParents + i,
    crossover_weighted_split(i % numParents, (i+1) %
       numParents));
  }
}
/* chooses parents at random, then breeds the with random
   crossovers */
public static void breedRandom(){
  int numParents =
     Math.max(2,(int)(KEEP_RATIO*POPULATION));
  int numChildren = POPULATION - numParents;
  int r1, r2;
  for(int i = 0; i < numChildren; i++){
    r1 = (int)(Math.random()*numParents);
    r2 = (int)(Math.random()*numParents);
    while (r1 == r2)
    r2 = (int)(Math.random()*numParents);
    boards.set(numParents + i,
    crossover_uniform(r1, r2));
  }
}

/* takes the indices of two parents, and breeds them with
   a single random split point  */
public static Board crossover_random_split(int p1, int p2){
  Board parent1 = boards.get(p1);
  Board parent2 = boards.get(p2);
```

```
  //ensure we aren't breeding duplicates
  if(p1 != p2 && parent1.sameBoardAs(parent2))
  return new Board();
  int splitPoint = (int)(Math.random() * (BOARD_SIZE-1));
  return new Board(parent1, parent2, splitPoint);
}
/* takes the indices of two parents, and breeds them with
   a splitpoint derived from the parents fitness */
public static Board crossover_weighted_split(int p1, int
   p2){
  Board parent1 = boards.get(p1);
  Board parent2 = boards.get(p2);
  //ensure we aren't breeding duplicates
  if(p1 != p2 && parent1.sameBoardAs(parent2))
  return new Board();
  //int splitPoint = (int)((Math.min(parent1.fitness,
     parent2.fitness) /
  //
     (double)Math.max(parent1.fitness, parent2.fitness)) *
     (BOARD_SIZE-1));
  int splitPoint = (parent2.fitness / (parent2.fitness +
     parent2.fitness)) * (BOARD_SIZE-1);
  return new Board(parent1, parent2, splitPoint);
}
/* takes the indices of two parents, and breeds them a
   uniform crossover method */
public static Board crossover_uniform(int p1, int p2){
  Board parent1 = boards.get(p1);
  Board parent2 = boards.get(p2);
  //ensure we aren't breeding duplicates
  if(p1 != p2 && parent1.sameBoardAs(parent2))
  return new Board();
  return new Board(best, worst, Math.random());
}

/* finds the most fit elements and returns it */
public static Board getBest(){
  Board best = boards.get(0);
  for(int i = 1; i < POPULATION; i+=1)
  if(boards.get(i).fitness < best.fitness)
  best = boards.get(i);
  return best;
}

/* shuffles the list of boards, very simple shuffle*/
public static void shuffle(){
  Random random = new Random();
  for(int i = POPULATION-1; i > 0; i--){
    int index = random.nextInt(i+1);
    Board temp = boards.get(index);  //swap(index, i)
    boards.set(index, boards.get(i));
    boards.set(i, temp);
  }
```

```
}
/* sorts the list of boards */
public static void sort(){
  Collections.sort(boards, new Comparator<Board>(){
    @Override
    public int compare(Board v1, Board v2) {
      return v1.compareTo(v2);
    }
  });
}

/* parses the args, stupidly simple lol */
public static void parseArg(String type, String value){
  switch (type.toLowerCase().charAt(0)){
    case 'b':
    BOARD_SIZE = Math.max(1,stringToInt(value));
    break;
    case 'p':
    POPULATION = Math.max(1,stringToInt(value));
    break;
    case 'g':
    MAX_GENERATIONS = Math.max(1,stringToInt(value));
    break;
    case 'r':
    MAX_RESTARTS = Math.max(0,stringToInt(value));
    break;
    case 'k':
    KEEP_RATIO = stringToDouble(value);
    if(KEEP_RATIO > 1) KEEP_RATIO = 1;
    break;
    case 'm':
    MUTATION_P = stringToDouble(value);
    if(MUTATION_P < 0f) MUTATION_P = 0f;
    if(MUTATION_P > 1f) MUTATION_P = 1f;
    break;
    default:
    System.err.println("ERR: unkown arg: " + type);
  }

}
public static int stringToInt(String s){
  try {
    return Integer.parseInt(s);
  } catch (NumberFormatException e) {
    System.err.println("ERR: Argument" + s + " must be an
       integer.");
    System.exit(1);
  }
  return 0;
}
public static double stringToDouble(String s){
  try {
    return Double.parseDouble(s);
```

```
    } catch (NumberFormatException e) {
      System.err.println("Argument" + s + " must be a
         double.");
      System.exit(1);
    }
    return 0;
  }
}

/* a Board object describes a particular board permutation
   using an array
*
*
*/
class Board {
  public int[] array; //the board permutation is stored this
     array
  public int fitness; //the fitness

  /* this constructor makes a random board permutation */
  public Board(){
    array = new int[Nqueen.BOARD_SIZE];
    for(int i = 0; i < Nqueen.BOARD_SIZE; i++)
    array[i] = i; //array = {1, 2, .. 8}
    shuffle();
    fitness = calcFitness();
  }
  /* this constructor creates a board from 2 other 'parent'
     boards
  uses single point crossover point */
  public Board(Board parent1, Board parent2, int splitPoint){
    array = new int[Nqueen.BOARD_SIZE];
    for(int i = 0; i < Nqueen.BOARD_SIZE; i++){
      if(i <= splitPoint)
      array[i] = parent1.array[i];
      else
      array[i] = parent2.array[i];
    }
    mutate();
    fitness = calcFitness();
  }
  /* this constructor creates a board from 2 other 'parent'
     boards
  uses a uniform distribution crossover method */
  public Board(Board best, Board worst, double p){
    array = new int[Nqueen.BOARD_SIZE];
    for(int i = 0; i < Nqueen.BOARD_SIZE; i++){
      if(Math.random() < p)
      array[i] = best.array[i];
      else
      array[i] = worst.array[i];
    }
    mutate();
```

```
  fitness = calcFitness();
}

/* mutate the board */
private void mutate(){
  for(int n = 0; n < Nqueen.BOARD_SIZE; n++){
    if(Math.random() < Nqueen.MUTATION_P){
      array[n] = (int)(Math.random() * Nqueen.BOARD_SIZE);
    }
  }
}

/* calculates the fitness as a count of un-orderd
   attacking pairs of queens
best fitness = 0
worst fitness = 8 choose 2 = 28
*/
private int calcFitness(){
  int attackingPairs = 0;
  for(int i = 0; i < Nqueen.BOARD_SIZE; i++)
  attackingPairs += countEast(i) + countSouthEast(i) +
     countNorthEast(i);
  return attackingPairs;
}
//count the queens horizontally left (east) of the current
   piece
private int countEast(int start){
  int result = 0;
  for(int i = start+1; i < Nqueen.BOARD_SIZE; i++)
  if (array[i] == array[start])
  result++;
  return result;
}
private int countNorthEast(int start){
  int result = 0;
  for(int i = start+1; i < Nqueen.BOARD_SIZE; i++)
  if (array[i] == array[start]-i+start)
  result++;
  return result;
}
private int countSouthEast(int start){
  int result = 0;
  for(int i = start+1; i < Nqueen.BOARD_SIZE; i++)
  if (array[i] == array[start]+i-start)
  result++;
  return result;
}

/* shuffles the array, simple knuth shuffle */
private void shuffle(){
  Random random = new Random();
  for(int i = Nqueen.BOARD_SIZE-1; i > 0; i--){
    int index = random.nextInt(i+1);
```

```
      int temp = array[index];
      array[index] = array[i];
      array[i] = temp;
    }
  }

  /* compareTo required for sort */
  public int compareTo(Board v2){
    double tmp = this.fitness - v2.fitness;
    return (tmp > 0) ? 1 : (tmp < 0) ? -1 : 0;
  }

  /* used to determine if this is identical to another board
     */
  public boolean sameBoardAs(Board board2){
    int count = 0;
    for(int i = 0; i < Nqueen.BOARD_SIZE; i++)
    if (array[i] == board2.array[i])
    count++;
    return (count == Nqueen.BOARD_SIZE);
  }

  /* nice string representation of the board, in a nice grid
     */
  public String toStringPretty(){
    String grid = "";
    for(int col = 0; col < Nqueen.BOARD_SIZE; col++){
      grid += "|";
      for(int row = 0; row < Nqueen.BOARD_SIZE; row++){
        if(array[row] == col)
        grid += "Q|";
        else
        grid += "_|";
      }
      if(col+1 < Nqueen.BOARD_SIZE)
      grid += "\n";
    }
    return toString() + "\n" + grid + "\nfitness: " +
       fitness;
  }

  /* simple string representation of the board */
  public String toString(){
    String s = "[";
      for(int col = 0; col < Nqueen.BOARD_SIZE; col++){
        s += (array[col]+1);
        if(col+1 < Nqueen.BOARD_SIZE)
        s += ",";
      }
      return s + "]";
  }
}
```