CAB402 Programming Paradigms

Genetic Algorithms

James Galloway

n9198865

3/06/2016

Contents

1 Introduction to Genetic Algorithms

Genetic algorithms (GAs) are heuristic[1] algorithms that mimic the events of evolution by natural selection in order to provide a solution to a problem. They are used as search and optimisation tools across a range of problem domains and a member of the broader class of evolutionary algorithms (EAs). Their appeal stems from their broad applicability, ease of use, and the ingenuity of their solutions [**1**].

1.1 Brief History

The concept of EAs was first discussed in 1950 by Alan Turing in the Oxford journal *Mind* [**2**]. Since then there have been a range of different types of EAs, with genetic algorithms being the most popular. It wasn't until the early 1970s that GAs emerged, developed at the University of Michigan by Professor John Holland and his students. 1975 would see the first publication of Holland's book *Adaptation in Natural and Artificial* Systems [**3**]. It was not an instant hit. Holland later remarked about its lack of sales and a lack of interest in GAs in general [**3**]. However, the popularity of GAs saw a slow and steady increase and eventually became a serious area of study and implementation.

1.2 Problem Application

Today, the use of GAs is reasonably widespread with applications including medicine [**4**], engineering [**5**], climatology [**6**], and telecommunications[2] [**7**]. They are typically applied to problems where near-optimal solutions are sufficient or where the optimal solution either does not exist or is too computationally costly to find. That is, they can reliably generate approximate solutions but are never guaranteed to find an optimal solution for any problem [**8**]. The problem must have some criteria by which a potential solution can be evaluated.

The travelling salesman problem is often used to demonstrate the ability of GAs and serves as the ideal template for problems suited to GAs, that is: a complex problem where an approximate solution is acceptable and can reasonably be obtained through trial and error[3].

[1] 'Heuristic is defined as: "...a 'rule of thumb,' or a good guide to follow when making decisions. In computer science, a heuristic has a similar meaning, but refers specifically to algorithms ...a heuristic process may include running tests and getting results by trial and error" [**12**].

[2] NASA has put GAs to work on evolving spacecraft antennas for deep space communication, resulting in bizarre but effective designs.

[3] Heuristic algorithms can yield near-optimal solutions for complex formulations (millions of nodes) of the travelling salesman problem in reasonable time frames [**13**].

2 Genetic Algorithms

There are three processes that form the core of GAs: selection (natural selection), crossover (sexual reproduction), and mutation, each of which being the counterpart of a real-world evolutionary mechanism [**9**]. For these processes to be applied, any potential solution (individual) must be encoded into a form (very often binary code) that suitably represents a chromosome [**3**].

After an initial population (of potential solutions) is randomly generated, the iterative process of evaluation, selection, crossover, and mutation begins. Figure 1 is a flowchart depicting this process. Each individual in the population is evaluated by being tested against the problem. A 'fitness score' is assigned to the individual that reflects their level of success. Individuals are then selected for the crossover process. Those with higher fitness scores are more likely to be selected. To generate offspring, the chromosomes of two selected individuals are then combined. These new chromosomes then undergo the process of mutation, where DNA is changed at random. The offspring are then placed into the population and the process repeats.

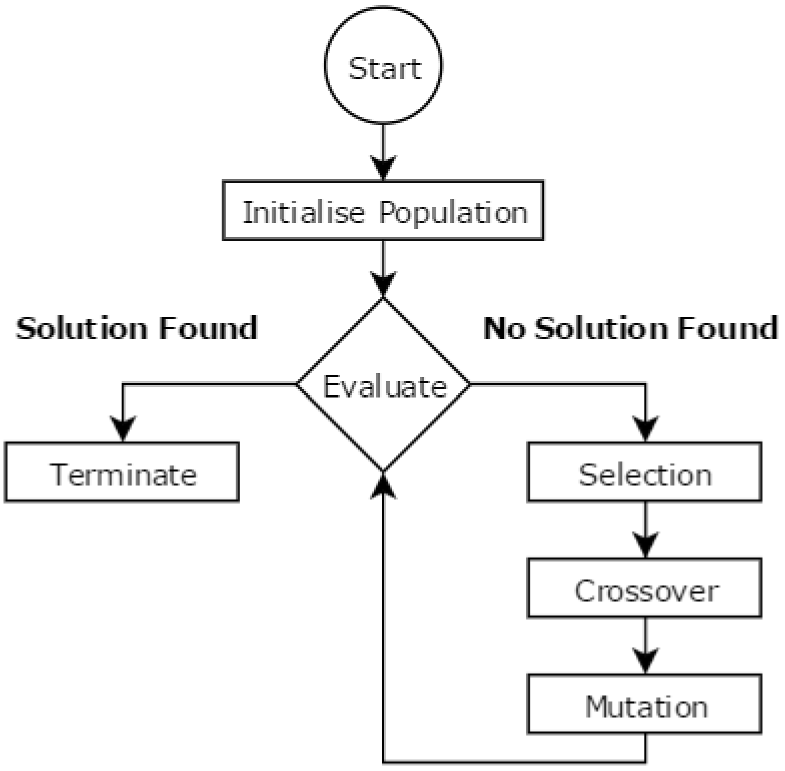


Figure 1: The process of a GA **[1]**.

Though there may be small regressions due to random elements of the algorithm, generally the population will improve after each iteration and move closer towards a viable solution. The algorithm will terminate once a suitable solution has been found.

For the following sections, assume we want to use a GA to give us two numbers that can be summed to equal the number 10[4]. So an acceptable solution would 4 and 6 while an unacceptable solution would be 2 and 3.

2.1 Chromosome

In the field of biology, a chromosome is defined as the organised structure of DNA, subdivided into genes [**10**]. In relation to the example problem, we could randomly generate a chromosome for any individual in the form of two four-bit binary strings, with each string representing is decimal conversion. Assume that we randomly generate 01011101 as a chromosome for an individual. As shown in Figure 2, the entire chromosome is 0101 1101, which is comprised of two genes (0101 and 1101), which are comprised of DNA (either a 0 or a 1).

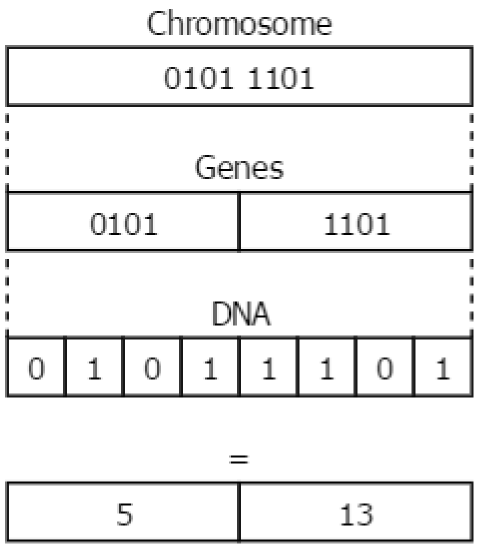


Figure 2: Example chromosome for use in a GA.

2.2 Evaluation & Fitness

We can see that the decimal conversion of these two genes is 5 and 13. It is now ready to be evaluated. To do this, there must be some representation of the problem that the solution can be tested again. In this case, it is as simple as summing the two numbers and observing the output 5 + 13 = 18.

We now have to find a suitable way to rate the fitness of this solution. This task is notoriously difficult for more complex problems that involve multiple criterions by which to judge success [**9**].

[4] This is an adaptation of slightly more complex example found at AI-Junkie.com [**14**].

Remembering that our goal is 10, we can see that this individual has missed the target by 8. We can therefore award it a fitness score of -8[5].

2.3 Selection

Darwin himself provides the following definition for natural selection: "the preservation of favourable individual differences and variations, and the destruction of those which are injurious" [**11**]. Fitter individuals survive to create new offspring and, in doing so, their favourable variations are preserved.

Enacting this concept in the world of GAs requires a guided selection process in which two or more individuals are selected for crossing. Their chance of being selected should be proportional to their fitness score so as to give preference to fitter individuals[6]. The roulette wheel selection algorithm[7] is a common method of selection and is discussed in detail in section 3.1.

[5] This is a simplification. In reality, a more suitable formula would be $f = \frac{1}{10-x}$ to normalise between results that above and below the target.
[6] It's not sufficient to simply take the fittest individuals for crossover, however, as the population can stagnate due to a lack of genetic diversity.
[7] Also known as 'fitness proportionate selection'.

2.4 Crossover

Crossover is the process of taking two parent solutions and producing from them a child [**9**]. The hope is that the offspring will perform better that the parent generation. Of course, the children that do perform better will have a higher chance of selection in the next iteration, creating a situation where the population is constantly improving. Figure 3 shows the process of an average crossover technique.

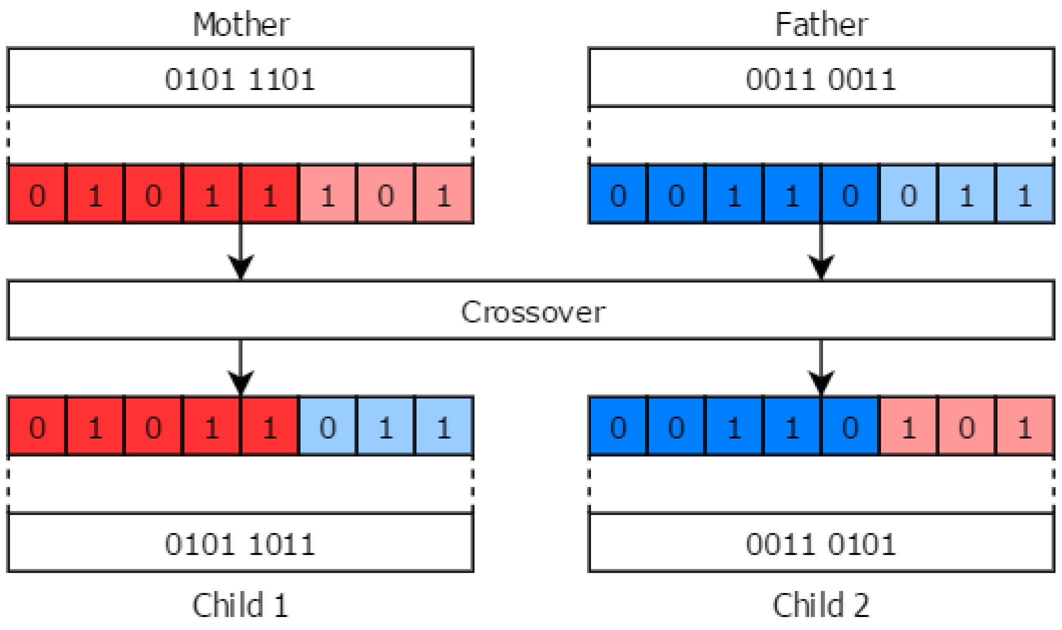


Figure 3: Example of the crossover process between two chromosomes.

This is an example of single-point crossover; a technique that produces two children from two parents by using a single cross point[8]. A 'mother' and 'father' chromosome are taken from the crossover pool that was generated by the selection process. A random crossover point is selected across the length of the chromosomes (5 in this case). Each of the parent chromosomes is then split into two pieces at this point (red to light red and blue to light blue). These pieces are then recombined to create two children. We can see that the chromosomes denoted 'child 1' and 'child 2' are the combination of one piece from each parent.

2.5 Mutation

Before the children can be introduced to the population, they must undergo the mutation process. In nature, mutation is the alteration of an organism's genetic sequence, as it is in GAs. Mutation

[8] Other crossover techniques exist with varying numbers of children and parents though the principle is the same [**15**].

plays an important role in preventing the population from being trapped in a local minimum by encouraging genetic diversity [**9**].

Figure 4 shows an example of a simple form of mutation, known as flipping[9]. In essence, the DNA of the chromosome is iterated over with the chance that any piece of DNA could be mutated. Mutation should occur fairly infrequently. If the mutation rate is too high, there is a risk of destroying favourable inherited traits; too low and there is a risk of genetic stagnation. For the flipping method of mutation, if a mutation is triggered then that bit is flipped.

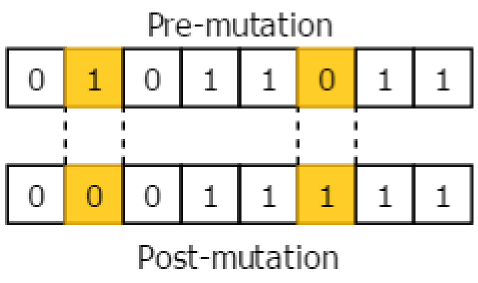


Figure 4: The states of a chromosome before and after mutation.

Once each child chromosome has been mutated, they can be introduced to the population for the next cycle of evaluation, selection, crossover, and mutation.

[9] Again, there are various different ways of achieving mutation. Sivanandam and Deepa provide a comprehensive list of both crossover and mutation techniques in *Introduction to Genetic Algorithms*, pages 50 to 59.

3 Practical Implementation

The following implementation mimics the classic mouse maze experiment wherein a mouse is tasked with locating a piece of cheese in a maze. All code was written in the F# programming language with a liberal approach to mutability. Three very simple mazes (Figure 5) were constructed in which to test the algorithm. In each maze, a piece of cheese is placed at the end. The essence of the problem is finding a route from the starting point to this piece of cheese. Hence, a solution to this problem would be any path that runs from the starting point to the cheese. We also want a fairly efficient path: to this end, mice that reach the cheese using fewer steps receive a fitness bonus. For the purposes of this example, no termination clause has been implemented. For completeness, code defining the parameters of each maze has been included in Appendix A.

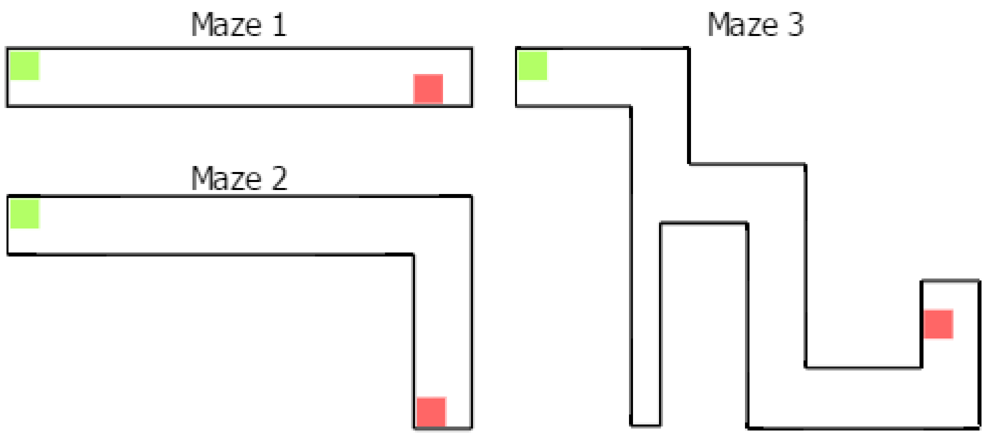


Figure 5: The three mazes to test the GA on. The green square is the starting position of the mice. The red square is the position of the cheese.

3.1 Generating a Population

To generate a population, we need to be able to encode the potential solutions to a chromosome that is compatible with the genetic operators. Any solution to this problem must have regard to the movement of the mouse. Figure 6 shows the structure of the chromosomes used in this implementation.

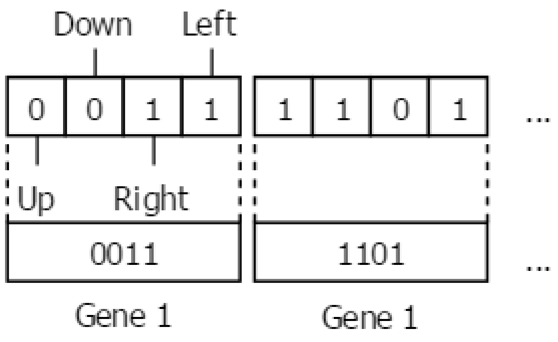


Figure 6: Example chromosome for the practical implementation.

A gene consists of four bits, each representing a direction of movement. Each bit is a decision as to whether to move or not, 0 being to stay still and 1 being to move. The position of the bit in the gene denotes what direction the decision applies to. We can expect the two example genes in Figure 6 to move the mouse one step to the left (though it takes 8 decisions to do so). A mouse has 150 genes, making for a total of 600 bits, which is to say that a mouse makes 600 decisions in its lifetime.

The following code shows the record types that were written to represent a chromosome.

```
type Gene =
      {
          Up: int
          Down: int
          Right: int
          Left: int
      }

 type Chromosome =
      {
          StepSeq : Gene list
          mutable Fitness : float
          mutable YPos : int
          mutable XPos : int
          mutable StepsTaken : int
      }
```

Note that a gene contains the four directions of movement, but that the chromosome contains the *StepSeq*, which a list of genes. As for the other parameters of the chromosome record, *Fitness* records the fitness of a chromosome after it has been evaluated while *YPos and XPos* are used to track the position of the mouse and *StepsTaken* are used to count the number of steps it has taken[10].

The following code is the *Populate* function, which is used to generate the initial population:

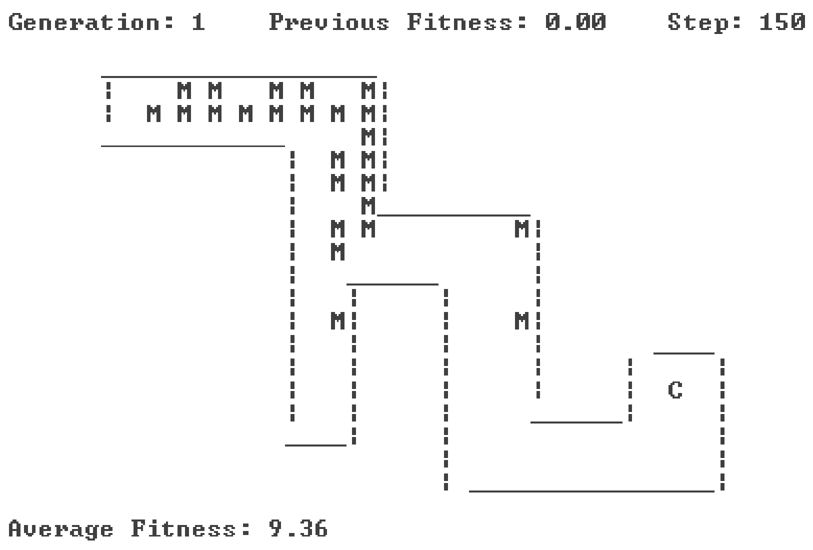
```
01   let Populate popSize =
02      Seq.fold (fun popAcc i ->
03         let genes = List.init NumGenes (fun _ ->
04            {
05               Up = seed.Next(2);
06               Down = seed.Next(2);
07               Left = seed.Next(2);
08               Right = seed.Next(2);
09            })
10
11         let chromosome =
12            {
13               StepSeq = genes;
14               Fitness = 0.0;
15               YPos = 1; XPos = 1;
16               StepsTaken = 0;
17            }
18
19         Map.add i chromosome popAcc) Map.empty (seq { 0 .. popSize - 1 })
```

Note lines 5 through 8 in which a randomised gene is created, and lines 13 through 16 in which the randomised genes are added and the chromosome is initialised. A total of 50 mice are created as the initial population.

[10] These parameters are not technically part of the chromosome. They have been included in the chromosome record for the sake of convenience.

3.2 Mouse Evaluation & Fitness

An entire generation of mice are tested in the maze at once, running through all 150 genes[11]. The code that is responsible for drawing each step has been included in Appendix B. Figure 7 shows the state of the initial population at the end of the first evaluation[12].



```
Generation: 1    Previous Fitness: 0.00    Step: 150

Average Fitness: 9.36
```

Figure 7: The final positions of the first generation after evaluation.

Based on their final positions relative to the cheese, we can calculate a fitness score for each member of the population.

The following code is the *CalculateFitness* function, which is used to assign a fitness score to each individual post-evaluation:

```
01   let CalculateFitness genNum gen =
02      Map.iter (fun individual chromosome ->
03
04         let speedBonus = (NumGenes - chromosome.StepsTaken) * 5
05
06         let fitness = float ((fst CheesePos + snd CheesePos) + (GetDistance
07            (chromosome.YPos, chromosome.XPos))) + float speedBonus
08         chromosome.Fitness <- fitness) gen
```

The purpose of line 4 and the *speedBonus* variable is to award bonus fitness points to mice that find the cheese with steps left over. A mouse will stop moving once it has found the cheese and any steps that it has left over are multiplied and added to its fitness. In this case, each left over step is multiplied by 5, resulting in a significant jump in fitness. This is designed to give a strong preference to more efficient mice. If the mouse has not found the cheese, there is no bonus.

[11] Mice can stack on top of each other. Keep this in mind while running the application. Once the algorithm gets more efficient, the mice are in closer synchronisation so it may look like their number is fewer.

[12] The 'Average Fitness' information at the bottom-left of the figure is not strictly relevant to the GA and is not used in any calculation. Rather, it is to gauge the overall improvement of the population.

Lines 6 and 7 calculate a basic fitness score for each mouse based on its final location relative to the location of the cheese. In essence, it finds the number of remaining steps in the shortest path (ignoring walls) between the mouse and the cheese and detracts that from the current position of the mouse[13].

3.3 Roulette Wheel Selection

Parents can now be selected for crossing. The roulette wheel selection algorithm has been used in this implementation. Sivanandam and Deepa provide the following description of this selection algorithm: "The principle of roulette selection is a linear search through a roulette wheel with the slots in the wheel weighted in proportion to the individual's fitness values" [**9**].

Figure 8 shows the probability distribution for the roulette wheel selection algorithm for a hypothetical population of 5, with the following fitness scores: 2, 4, 4, 8, 10.

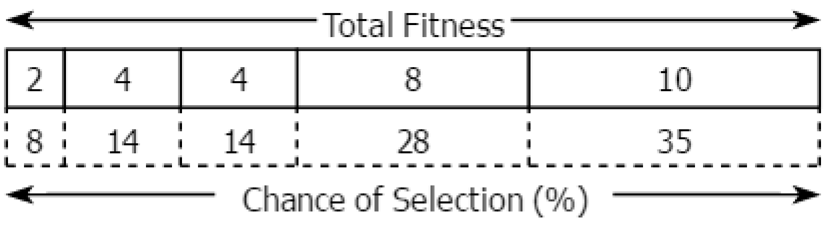


Figure 8: Example probability distribution for roulette wheel selection.

The size that each element occupies is proportionate to their chance of selection, which is denoted on the bottom row. We can see that higher values have a better of chance of selection.

[13] For example, say that a mouse at position 17, 14 is -13 squares away from the cheese. The following sum is used to calculate their fitness: $(17 + 14) + -13 = 18$.

The following code is the *Select* function, which implements the roulette wheel selection algorithm:

```
01   let Select (gen : Map<int,Chromosome>) =
02      let mutable cumulativeFitness = [gen.[0].Fitness]
03      for i in 1 .. gen.Count - 1 do
04         cumulativeFitness <- cumulativeFitness @ [(cumulativeFitness.[i - 1] +
05            gen.[i].Fitness)]
06
07      let mutable newPop = []
08      while newPop.Length < MaxPop - 2 do
09         let mutable parents = []
10         while parents.Length < 2 do
11             let randFitness = seed.NextDouble() *
12               cumulativeFitness.[cumulativeFitness.Length - 1]
13
14            let index = List.findIndex (fun fitness -> fitness >
15               randFitness) cumulativeFitness
16
17            let parent = List.exists (fun chromosome -> chromosome =
18               gen.[index]) parents
19
20            if not parent then
21               parents <- gen.[index]::parents
22
23         let children = Crossover parents
24         newPop <- newPop @ children
25      newPop
```

Lines 2 to 5 are responsible for calculating the cumulative fitness of the population. This process is equivalent to setting up the slots of the roulette table.

Line 8 sets up a *while* loop, the purpose of which is to ensure that the selection process continues until such a time as the new population nearly equals the population limit. This process terminates two individuals shy of the population limit. This is to account for a process called 'elitism', explained in section 3.5.

Line 10 is the point at which the selection begins and lines 11 to 15 are where our selection is made. This process continues until two parents are selected. A random value is selected and compared with the cumulative fitness scores of the population. The first individual with a fitness score exceeding the random value is selected. Note that lines 17 to 20 are used to prevent the same individual from being selected twice[14].

[14] This would result in a situation where an individual is crossed with themselves.

3.4 Single-Point Crossover & Mutation

The same methods of crossover and mutation as discussed in sections 2.4 and 2.5 have been implemented here. The following code is the *Crossover* function:

```
01   let Crossover (parents : Chromosome list) =
02       let father = parents.[0]
03       let mother = parents.[1]
04
05       let crossPoint = seed.Next(NumGenes)
06
07       let partFather = TakeBefore crossPoint father.StepSeq [] 0
08       let partMother = TakeAfter crossPoint mother.StepSeq 0
09       let fstChild = List.concat [partFather;partMother]
10
11       let partMother = TakeBefore crossPoint mother.StepSeq [] 0
12       let partFather = TakeAfter crossPoint father.StepSeq 0
13       let sndChild = List.concat [partMother;partFather]
14
15       [fstChild;sndChild]
```

Line 5 defines a random point within the length of a chromosome. Lines 7 and 8 take the father's genes before this point and the mother's genes after this point to create the first offspring in line 9. Lines 11 to 13 create the second offspring by reversing this behaviour: that is, taking the father's genes after the crossover point and the mother's genes before.

The follow code is the *Mutate* function:

```
01    let Mutate children =
02      List.fold (fun mutateAcc child ->
03        let mutatedGenes =
04          List.fold (fun genesAcc gene ->
05            if seed.NextDouble() < 0.005 then
06              let newGene =
07                {
08                  Up = seed.Next(2); Down = seed.Next(2);
09                  Left = seed.Next(2); Right = seed.Next(2);
10                }
11              newGene::genesAcc
12            else gene::genesAcc) [] child
13        (Reverse mutatedGenes)::mutateAcc) [] children
```

We iterate over each gene in the child chromosomes. If no mutation is triggered, we add the gene as it is to the accumulation (Line 12). A random double (between 0 and 1) is created and compared in line 5. 0.005 specifies the chance of mutation, which is equivalent to half a percent. This figure was arrived at through trial and error. Lines 6 to 11 are read if a mutation is triggered. They are used to create a new random gene and add it to the chromosome instead of adding the current gene to the chromosome.

3.5 Elitism

The new generation is ready to be placed into the population for the next cycle. At this stage, however, we still don't have a complete population, remembering that we are missing two from the early termination of the *Select* function. This is because of elitism: a concept that expressly preserves the two best chromosomes as they are so as to save the most successful traits from being lost in the randomness of selection, crossover or mutation [**9**].

The following code is the *AssembleNextGeneration* function, in which the new population is assembled using the child chromosomes and elite:

```
01   let AssembleNextGeneration (gen : Map<int,Chromosome>)
02      (children : Gene list list) =
03
04       let blankChrom =
05         {
06            StepSeq = []; Fitness = 0.0;
07            YPos = 1; XPos = 1; 05StepsTaken = 0;
08         }
09      let blankElite = Map.add 0 blankChrom Map.empty |> Map.add 1 blankChrom
10
11      let elite = Map.fold (fun (elite : Map<int,Chromosome>) individual
12         chromosome ->
13            if chromosome.Fitness > elite.[0].Fitness then
14               Map.add 0 chromosome elite
15            else if chromosome.Fitness > elite.[1].Fitness then
16               Map.add 1 chromosome elite
17            else elite) blankElite gen
18
19      elite.[0].YPos <- 1; elite.[0].XPos <- 1; elite.[0].StepsTaken <- 0;
20      elite.[1].YPos <- 1; elite.[1].XPos <- 1; elite.[1].StepsTaken <- 0;
21
22      let mutable i = elite.Count - 1
23
24      List.fold (fun nextGen child ->
25         i <- i + 1
26         let chromosome =
27            {
28               StepSeq = child; Fitness = 0.0;
29               YPos = 1; XPos 25= 0; StepsTaken = 0;
30            }
31         Map.add i chromosome nextGen) elite children
```

Lines 11 to 17 search the old population for the two fittest individuals. These individuals are preserved and added into the next generation in lines 24 to 31. The new population is now ready for evaluation, consisting of the unevaluated child chromosomes and the two elite chromosomes from the previous generation.

3.6 Progression

Figure 9 is a graph showing the progression of the average fitness of successive generations across the three mazes. Red markers indicate the generation in which the cheese is first found. Notice that the fitness of generations after this point increases dramatically. This is mainly because of the speed bonus; however it does also reflect the improvement of the population in general as the mouse receiving this bonus is crossed.

The maximum of 25 generations is arbitrary since no point of termination has been implemented. Typically, the population reaches a certain point of efficiency and stalls. This could be considered as a suitable point of termination and is usually a fairly efficient path. However, if left running long enough, the most efficient paths would be located.

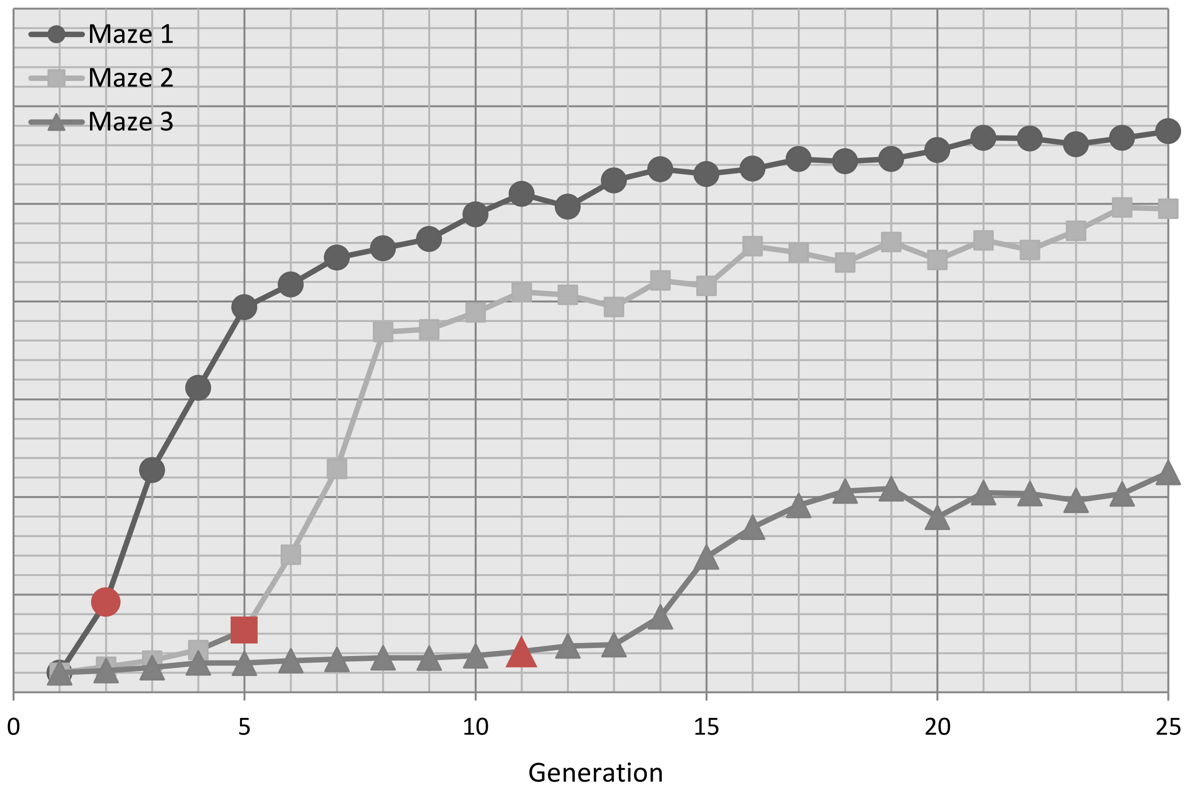


Figure 9: Average fitness of generations across the three mazes

4 Conclusion

We have seen the ways in which a basic GA works, including the four key processes of evaluation, selection, crossover mutation. These processes have been implemented and exampled in a practical example, which also included the 'elitism' variant.

Genetic algorithms excel when fitted to a suitable problem. Typically applied to search and optimisation problems, they can be to generate ingenious solutions which are informed by randomness and aren't coloured by a human way of thinking. Care should be taken in deciding to use a GA, given that they shouldn't be relied on to produce optimal solutions. The success of a GA depends upon choosing effective parameters related to the encoding of chromosomes, the balancing of success criteria and the selection, crossover and mutation operators.

5 References

[1] Deb Kalyanmoy, "An Introduction to Gentic Algorithms," *Sadhana*, vol. 24, no. 4, pp. 293-315, 1999.

[2] Alan M. Turing, "Computing Machinery and Intelligence," *Mind: A Quarterly Review of Psychology and Philosophy*, vol. 236, pp. 433- 460, October 1950.

[3] John H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*, 1st ed. United States of America: MIT Press, 1992.

[4] Krzysztof Krawiec, "Genetic Programming with Alternative Search Drivers for Detection of Retinal Blood Vessels," in *European Conference on Applications of Evolutionary Computation*, Copenhagen, 2015, pp. 554-565.

[5] K. C. Ng , D.J. Murray-Smith , G.J. Gray , K. C. Sharman Y. Li, "Genetic Algorithm Automated Approach to Design of Sliding Mode Control Systems," *International Journal of Control*, vol. 63, no. 4, pp. 721-739, 1996.

[6] Karolina Stanislawska, Krzysztof Krawiec, and Zbigniew W. Kundzewicz, "Modeling Global Temperature Changes with Genetic Programming," *Computers & Mathematics with Applications*, vol. 64, no. 12, pp. 3717-3728, December 2012.

[7] Alexandre Guillaume, "Deep Space Network Scheduling Using Evolutionary," in *Aerospace Conference*, Big Sky, 2007, pp. 1-6.

[8] Daniel W. Dyer. (2010) Evolutionary Computation in Java. [Online]. http://watchmaker.uncommons.org/manual/ch01s02.html

[9] S.N Sivanandam and S.N Deepa, *Introduction to Genetic Algorithms*, 2008th ed.: Springer, 2007.

[10] Jurgen Brosius, "The Fragmented Gene," *Annals of the New York Academy of Sciences*, vol. 1178, no. 1, pp. 186-193, Oct. 2009.

[11] Charles Darwin, *On the Origin of Species*, 6th ed. United Kingdom: John Murray, 1859.

[12] TechTerms. [Online]. http://techterms.com/definition/heuristic

[13] César Rego, Dorabela Gamboa, Fred Glover, and Colin Osterman, "Traveling salesman problem heuristics: Leading methods, implementations," *European Journal of Operational Research*, vol. 211, no. 3, pp. 427-441, June 2011.

[14] AI-Junkie. [Online]. http://www.ai-junkie.com/ga/intro/gat1.html

[15] V Edmondson and B Gillet, "Genetic Algorithm with 3-Parent Uniform Crossover," University of Missouri, Missouri, PhD Thesis 1993.

Appendix A: Code Defining Maze Parameters

```
// Maze 1
// let CheesePos = (2, 17) // The position of the coveted cheese
// let XWalls =
//   Map.add 0 (0, 18) Map.empty |>
//   Map.add 3 (0, 18)
//
// let YWalls =
//   Map.add 0 (1, 3) Map.empty |>
//   Map.add 19 (1, 3)

// Maze 2
// let CheesePos = (14, 17) // The position of the coveted cheese
//
// let XWalls =
//   Map.add 0 (0, 18) Map.empty |>
//   Map.add 3 (0, 15) |>
//   Map.add 15 (17, 18)
//
// let YWalls =
//   Map.add 0 (1, 3) Map.empty |>
//   Map.add 16 (4, 15) |>
//   Map.add 19 (1, 15)

// Maze 3
   let CheesePos = (14, 18) // The position of the coveted cheese

   let XWalls =
     Map.add 0 (0, 8) Map.empty |>
     Map.add 3 (0, 5) |>
     Map.add 6 (9, 13) |>
     Map.add 9 (8, 10) |>
     Map.add 16 (6, 7) |>
     Map.add 15 (14, 16) |>
     Map.add 18 (12, 19) |>
     Map.add 12 (18, 19)

    let YWalls =
     Map.add 0 (1, 3) Map.empty |>
     Map.add 6 (4, 15) |>
     Map.add 8 (10, 16) |>
     Map.add 9 (1, 6) |>
     Map.add 11 (10, 18) |>
     Map.add 14 (7, 15) |>
     Map.add 17 (13, 15) |>
     Map.add 20 (13, 21)


   let X = Map.fold (fun coordAcc Y wallLen ->
     let coords = List.init (snd wallLen + 1 - fst wallLen + 1) (fun x -> (Y,
          fst wallLen + x))
     List.concat[coordAcc;coords]) [] XWalls

   let Y = Map.fold (fun coordAcc X wallLen ->
     let coords = List.init (snd wallLen + 1 - fst wallLen + 1) (fun y -> (fst
       wallLen + y, X))
     List.concat[coordAcc;coords]) [] YWalls

   let WallSpace = List.concat[X;Y]
   let mutable PrevAvg = 0.0
```

Appendix B: DrawSpace Function

```
let DrawSpace genNum stepNum gen =
   Console.Clear()

   let sb = new Text.StringBuilder(String.Format("\n Generation: {0} Previous
      Fitness: {1:0.00}Step: {2}\n\n", genNum, PrevAvg,stepNum))

   let str =
      Seq.fold (fun (sb : Text.StringBuilder) y ->
         let segment =
            Seq.fold (fun line x ->
               let isOccupied = Map.exists (fun _ chromosome -> (y,x) =
                  (chromosome.YPos, chromosome.XPos)) gen
               if isOccupied && (y,x) = CheesePos then
                  line + " !"
               else if isOccupied then
                  line + " M"
               else if (y,x) = CheesePos then
                  line + " C"

               // Draw walls
               else if XWalls.ContainsKey(y) then // X Walls
                  if x >= (fst XWalls.[y]) && x <= (snd XWalls.[y]) then
                     line + "__"
                  else if YWalls.ContainsKey(x) then
                     if y >= (fst YWalls.[x]) && y <= (snd YWalls.[x]) then
                        line + "| "
                     else line + "  "
                  else line + "  "

               else if YWalls.ContainsKey(x) then// Y Walls
                  if y >= (fst YWalls.[x]) && y <= (snd YWalls.[x]) then
                     line + "| "
                  else line + "  "
               else line + "  ") "" (seq { 0 .. SpaceXLen })

      sb.Append(segment + "\n")) sb (seq { 0 .. SpaceYLen }) |> fun x ->
         x.ToString()
   printfn "%s" str
    Thread.Sleep(75)
```

Appendix C: Relevant Helper Methods

```
  let GetDistance pos =
    let YDist = abs (fst pos - fst CheesePos)
    let XDist = abs (snd pos - snd CheesePos)
    -YDist + -XDist
```

```
  let rec TakeBefore index L acc i =
    if i = index then Reverse acc else
      match L with
      | head::tail -> TakeBefore index tail (head::acc) (i + 1)
      | [] -> Reverse acc
```

```
  let rec TakeAfter index L i =
    if i = index then L else
      match L with
      | head::tail -> TakeAfter index tail (i + 1)
      | [] -> L
```

Appendix D: Main Function

```
let main argv =
   let mutable genNum = 1
   let mutable Population = Populate MaxPop

   while true do
      Population <- Evaluate genNum Population |>
         Select |>
         Mutate |>
         AssembleNextGeneration Population
       genNum <- genNum + 1

   Console.ReadKey() |> ignore
   0
```