Adaptive Systems Project

1. Introduction & Background:

Adaptivity is the ability of a system to change itself in order to better suit the conditions it is in, as well as any parts of itself it cannot change. This study will examine how a genetic algorithm is affected by the amount of information it has about a system. Specifically this study will look at detailed vs simplified fitness functions and breeding large groups vs breeding single objects. This will be explored in a series of experiments using a genetic algorithm and the crazy killer cowboys problem.

A genetic algorithm tasked with solving the crazy killer cowboys problem meets our definition of an adaptive system. Firstly it has parts of itself it can change, that is, the locations of the cowboys. Next it has parts of itself it cannot change, the locations of the obstacles and the rules of the problem. Finally it has a way of deciding if a solution is better suited to the locations of the obstacles, since we know a better suited solution is one in which more cowboys survive.

A genetic algorithm uses an evolutionary approach to problem solving. The genetic algorithm we will use for this project will use a semi-generational approach. The generational approach involves generally starting with a randomly created generation. This generation is judged and a new generation is bred such that the next generation is mostly populated by children of the more successful elements of the previous generation. The assumption made is that the combination of the better parts of each generation will lead to an improved next generation. This is called the building block hypothesis. (Reeves, 2003)

Each generation can be made up of anything from sets of possible solutions to individual actors. Genetic operators are used to create the next generation. There are 2 main genetic operators used in genetic algorithms: crossover and mutation. Crossover involves taking 2 parent objects and returning a child which inherits some parts of itself from each parent. Mutation on the other hand generates a child by making changes to (or mutating) a single parent object.

The genetic algorithm we use for this project will use a fixed length genetic encoding for each type of object that is to be examined. This abstraction makes it far simpler both to encode and to interpret the genes, and is the approach De Jong took when first implementing what we now call simple genetic algorithms (1975). Other approaches have attempted to use variable length strings in order to improve the function of the algorithm. This type of genetic algorithm is called a messy genetic algorithm or mGA and is first described in Goldberg, Korb and Deb's work (1989). In this project however we will stick to using sGAs since using mGAs would make it harder to ensure similarity in the ways both cowboys and solutions are bred together, making it harder to make comparisons between these.

This paper is divided up into sections. The next section deals with defining some of the key terms that will be used in this paper and explaining very briefly some of the more important implementation decisions which may have an effect on the results. After that is a section discussing the precautions taken when testing. Next comes the testing itself, divided up into experiment and each containing a brief explanation of what is being tested and why, a hypothesis as to what the results will be, the relevant data and a brief discussion about what was found and how it affects the issues that are being examined. After the testing is the conclusion section containing a discussion of the trends identified and what this means for the issues being looked at. Finally are the appendices, containing the complete testing data and all the code produced.

2. Key terms & Concepts:

The genetic algorithm works by iteratively eliminating half of a list of either cowboys or solutions, and then breeding together the remaining half until it is back up to full numbers.

A **Cowboy** is simply a coordinate with an x and y value.

A **Solution** is a collection of 10 cowboys.

A **Problem** is a specific layout of obstacles and a set of instructions on how to handle breeding.

The world contains obstacles that block line of sight. These obstacles are represented as a set of 4 points. The cowboys are not allowed to start within obstacles, and the PNPoly algorithm (W. R. Franklin) is used to check if they are inside the polygonal area of an obstacle. If the cowboy does start off inside an obstacle it is moved to a random edge of the polygon's bounding rectangle.

When the program starts up the obstacle works out where its edges are and passes this list of lines to the problem class. When checking if 2 cowboys see each other a line is drawn between them and line intersection is used to see if this line intersects any of the obstacle lines.

Checking if cowboys can see each other happens simultaneously for all cowboys in the current solution.

The genetic encoding of a solution is the set of points of the cowboys in it, the genetic encoding of a cowboy is the point it represents. To avoid these being brittle (breaking when breeding produces an illegal value) the program was designed to be able to handle any set of numbers for the genetic encoding, and is able to move the cowboy should it be in an illegal location as detailed above.

3. Experimentation:

Testing Procedure:

Control group: It is important to have a control to compare the results of the testing to. A suitable control in this case is a problem which merely generates a set of random solutions and performs no actions on them. In order to get a better representation of random solutions this problem is modified to produce a pool of 100 solutions, rather than the normal 10.

Program Termination: The optimal solution to the program is when the number of dead cowboys in a solution is equal to 0. There will be some sets of obstacles where this result is not possible. To avoid non-termination the stopping point is instead based on the number of comparisons made by the algorithm. A comparison here is an operation that judges 2 objects for the purposes of sorting them. One comparison is made no matter if we are comparing single cowboys or whole solutions. The number of comparisons at which the algorithm stops breeding new solutions is 100,000. Incrementing the comparison variable is built into every method that judges items. Since the check is only made after each run through of the breeding process it is unlikely that the maximum number of comparisons will ever be exactly met, however the overrun should never be enough as to have a dramatic effect on the results.

Repeatability: Since the tests use a randomly seeded random number generator to place the initial cowboys they are not repeatable. However it is hoped that by running the algorithm repeatedly and taking an average of the results a representative result can be found. This also makes anomalous results highly unlikely.

Control of variables: In order to avoid changing more than 1 variable when rerunning tests the layout of the obstacles when running the program is set in an xml file. This ensures that the only thing changing when rerunning the program is the random initial locations of the cowboys and the different fitness functions and breeding methods.

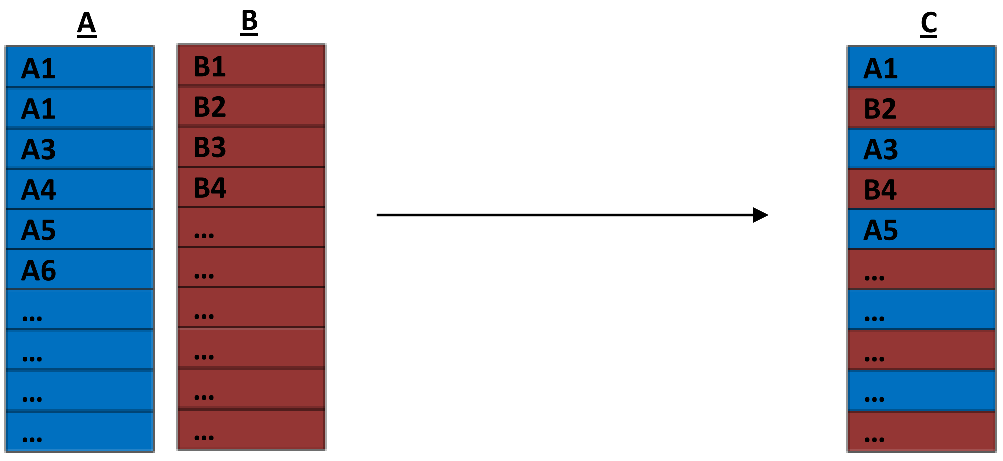
High Level Only vs Low Level Only:

Explanation:

There are 2 obvious levels of detail the algorithm can consider when trying to solve the problem. The first level, the high level approach to solving the problem, views each solution as its own object, with its own genome. The genetic encoding used for the solutions is simply the list of cowboys stored by it. This is good since it is representative of the most important information stored in the solution, while also being easy to modify.

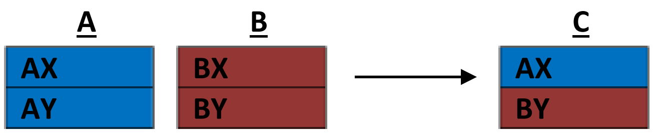
This method of solving the problem uses a gene pool of 10 possible solutions. It sorts, eliminates and breeds these solutions based on the number of cowboys alive in each. To get a new set of solutions the algorithm eliminates half the current solutions in the pool, and breeds together the remaining ones randomly until it has the desired number of solutions again. Since it doesn't breed an entirely new generation each time this approach is only partially generational. The algorithm uses the number of cowboys alive in each solution as the fitness function.

To breed the two solutions together the algorithm uses crossover. This means it takes the two genetic encodings from the two solutions and swaps part of the encoding for one with the encoding of another. For this project this is done by alternating between each genetic encoding like this:



It is important to note that the high level approach has no idea what cowboy is in each part of each genetic encoding and will never try to view an individual cowboy. For all it knows it is selecting the worst cowboys possible for a given solution.

The low level approach on the other hand only deals with individual cowboys, not whole solutions. The low level approach starts with a single solutions and selects, eliminates and breeds together cowboys from within that solution. The breeding is handled again with crossover, however in this case the parts to be swapped between different cowboys are the x and y locations like this:



The low level approach uses the status of the cowboy, whether it is alive or dead, in order to decide fitness. An alive cowboy is better than a dead cowboy.

It is important to note that both of these breeding methods have a 1 in 10 chance of producing a completely random result. This was done to add instability to the algorithm in an attempt to avoid stagnation.

This test will compare these two approaches to problem solving, it will compare the average and best values of the solutions in the final set found by the high level approach with the solution found by the low level approach.

Hypothesis:

The high level approach will produce better results than the low level approach. This will be because in the high level approach the algorithm is implicitly examining 100 different cowboy locations and selecting the best 50 of those, while the low level approach is more explicitly selecting the best 5 out of 10 cowboys.

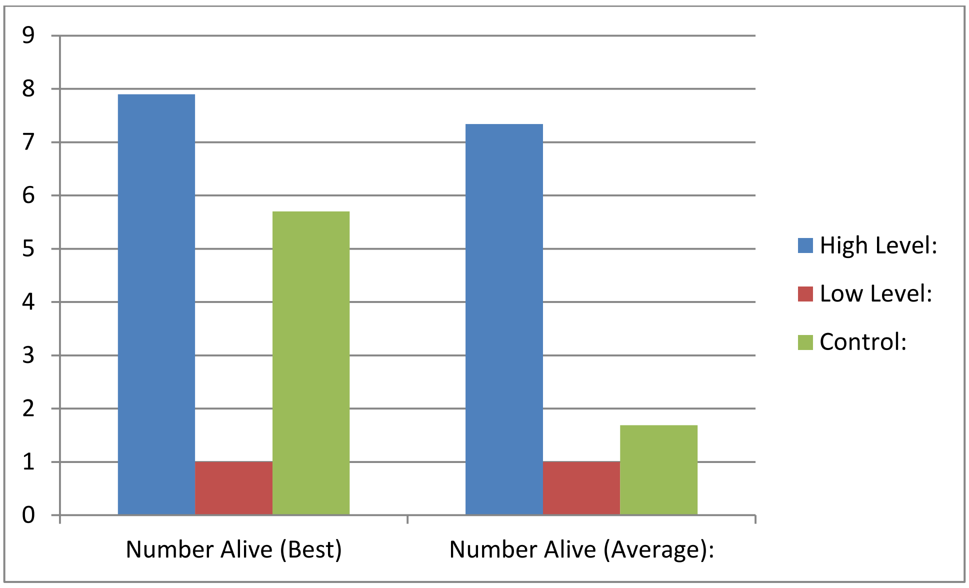
The high level approach, through this implicit selection, is able to choose cowboys that are not individually fit but which are part of a fit solution. This is important since in some cases it will be necessary to sacrifice some cowboys in order for the number alive to go up. While the high level approach is capable of this the low level approach is not, and will always seek the best for each individual cowboy, even if that damages the solution as a whole.

Furthermore, the low level approach lacks a continuous fitness function, since cowboys are either alive or dead, with nothing in between. This means that optimising cannot happen gradually and a random approach would serve just as well. On the other hand the high level approach can derive a continuous fitness function for the solutions based on the number of cowboys alive in each, meaning it will likely perform better.

Investigation:

This table shows the number of cowboys that are alive on average in both the best solution in the gene pool for each approach and the average of the entire gene pool. Since the low level approach has only 1 solution in its gene pool these values are the same, however they are included to allow easier comparisons between the different methods. Improvement is found by comparing the initial random solutions with the final solutions. Since the control does not perform any modification to its initial random solutions this measure is not applicable to it.

Approach:	**Number Alive (Best)**	**Number Alive (Average):**	**Improvement (Best):**	**Improvement (Average):**
High Level:	7.9	7.34	3.5	5.63
Low Level:	1	1	0.1	0.1
Control:	5.7	1.688	N/A	N/A



Findings:

As predicted the high level approach produced the better solutions overall. The low level approach appears to have a negative effect at first glance. Looking at the average improvement however suggests that this approach on average has very little effect on how optimised its solution becomes.

There are several possible reasons for this. One of these reasons was mentioned in the hypothesis, and that is that the fitness function is not continuous for the low level approach. Another possible reason for the failure of the low level approach is that using crossover to breed the next set of cowboys makes it too difficult to preserve structures of solutions since the new cowboys will likely not be placed nearby their parents.

<u>Visibility vs Number Alive (Low Level):</u>

Explanation:

The number of cowboys left alive is a suitable fitness function for the high level solutions. One of the problems that was identified with it for the low level approach is that if a cowboy is alive or not is a discrete piece of data rather than a continuous one, and so the algorithm cannot slowly optimise towards it. This may be the reason that the low level approach was far less effective when than the high level approach.

One solution to this is to find a new fitness function for use at the low level, one that is discrete. Visibility is such a function. Visibility is a more detailed view of whether the cowboy is dead or alive. A cowboys visibility is how many other cowboys can see it. A lower visibility is better. At a visibility of 0 the cowboy can be seen by no other cowboys, and is thus alive.

For this test the average results of the control will be compared with the final solutions found by low level approaches to this problem using both the number of cowboys alive and the visibility as fitness functions. This test will use the merge based breeding method.

Hypothesis:

The use of visibility as the fitness function will create better results than the use of the status of the cowboys. This will be because the fitness function produces a more smooth fitness landscape, and allow the algorithm a slowly increasing function that it can slowly improve, rather than having to rely upon one of the random cowboys being alive and not being able to select between the others in any way.

Investigation:

Fitness Function:	Number Alive:	Improvement (Number Alive):	Visibility	Improvement (Visibility):
Dead vs Alive	1	0.1	29.80	3.2
Visibility	0.8	-1.2	30.8	11
Control:	1.688	N/A	19.374	N/A

Findings:

The different fitness function does not appear to have a large effect on the efficiency of the low level approach. However while the number alive at the end is mostly similar it appears that in the process of getting to that final solution the algorithm has actually lost a number of cowboys.

<u>Visibility vs Number Alive (High Level):</u>

Explanation:

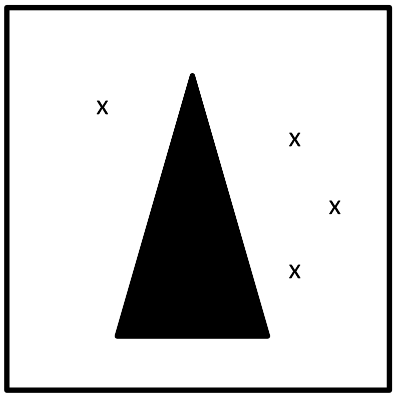
While the low level approach does not fare better with a more detailed fitness function does the same hold true for the high level approach. This test attempts to answer this question by comparing the solutions found by high level implementations of the genetic algorithm from the first set of tests using total visibility in solutions as the fitness function, and comparing this to both the high level implementation using the number of living cowboys and the control.

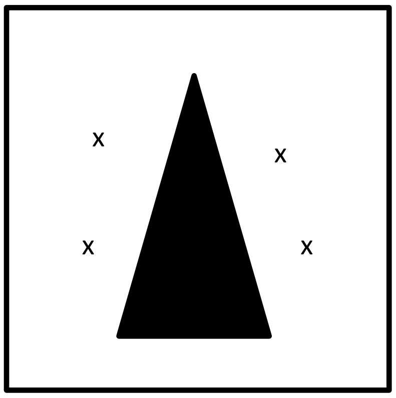
Hypothesis:

Using the visibility as the fitness function will have a worse result than using the number of cowboys alive. Firstly the high level approach already has a continuous fitness function, and so it will not see any gain from changing that.

Secondly visibility is a step removed from the actual value we are trying to maximise, though they are related. This level of removal will make changes to the visibility less easy to map to changes in the number of cowboys alive and so will make it an ineffective fitness function.

For example consider the 2 potential solutions:

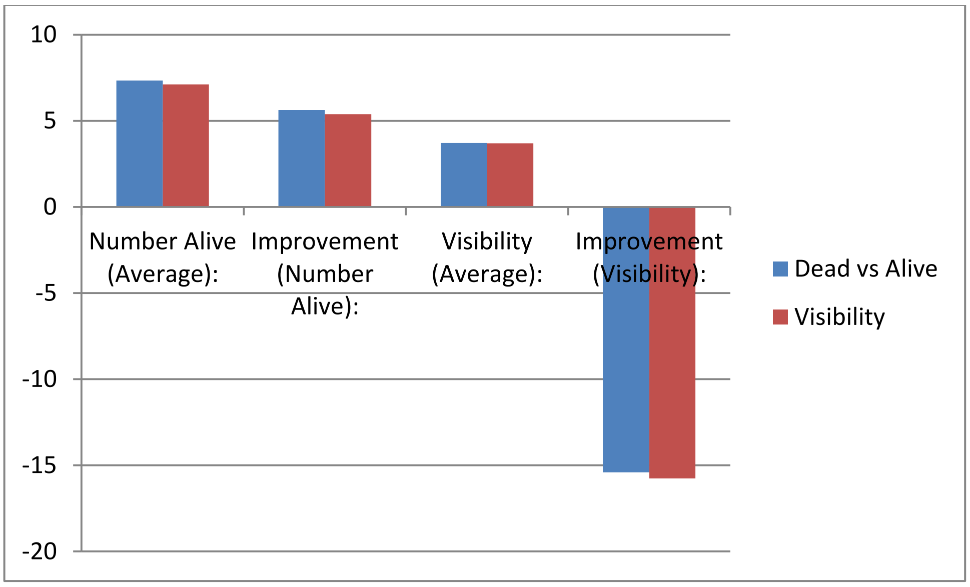




The solution on the left has visibility 6, since all the cowboys on the right can be seen by 2 other cowboys. However this solution has 1 alive cowboy. The solution on the right on the other hand has visibility 4 since all 4 cowboys can only be seen by 1 other, however in this case there are no survivors. This shows how visibility may not reflect the actual state of the solution since there are several potential ways to get to each possible visibility though of course when all cowboys survive visibility will always be 0.

Investigation:

Fitness Function:	Number Alive (Average):	Improvement (Number Alive):	Visibility (Average):	Improvement (Visibility):
Dead vs Alive	7.34	5.63	3.72	-15.4
Visibility	7.11	5.38	3.7	-15.76
Control:	1.688	N/A	19.374	N/A



Findings:

The two different fitness functions have a negligible effect on the power of the genetic algorithm using a high level approach. This suggests that the visibility, despite not being fully mappable to the actual desired result is still related enough to be effective when attempting to minimise it. Why this does not hold true for the low level is difficult to say.

4. Conclusion:

The testing performed during this project has some interesting implications. Firstly it suggests that it is better to view large groups and use data derived from these groups rather than look at individual members of the group. This is shown in the first set of tests, and is likely the result of simply being able to implicitly compare far more individuals when they are grouped than when they are on their own, as well as being more able to take advantage of random generation, when more random solutions are generated the algorithm is able to keep the few anomalous very good ones and discard all the very bad. This has a larger effect than on the low level where an anomalously good cowboy is still just one that is alive.

Secondly the testing suggests that more detailed fitness functions will not always improve the results gained from a genetic algorithm. While this could be because the visibility was not closely related to the number of cowboys surviving this is unlikely, since the last set of tests showed that there is next to no change between the results of the high level implementations.

On the other hand the low level tests showed the visibility fitness function having a seemingly negative effect. This could be because when selecting the cowboys to not eliminate the algorithm was no longer preserving the ones that were alive, however this doesn't explain the fact that the low level approach was actively harmed by this fitness function. I would guess that this is due to how the crossover breeding works for the cowboys. Since the new cowboy is placed on either the same x or the same y coordinates as the parent it seems likely that in many cases the child would have a direct line of sight on the parent, lowering the visibility and the number alive.

I am confident that the results of my study hold for this set of obstacles, however I have not had a chance to perform further testing on additional sets, since I wished to avoid changing something so important to the func tioning of the algorithm and potentially spoiling the averages. I doubt the structure of the world would have much effect on my findings about which methods were more or less effective, though they might make it more or less likely for an optimal solution to be found. Obviously this is an opportunity for further study.

Another possibility for further testing would be to explore the use of mixed level methods, such as a mostly low level approach that checked any children were better than the current solution before using them. Another mixed approach would be a mostly high level approach that examined the cowboys in each solution and removed the worst of them when breeding.

I would also be interested to study the results of using a different breeding method when breeding together cowboys. For example setting the child to the average of the parents coordinates.

Finally I feel that I study attempting to perform these same tests using mutation instead of crossover when breeding would yield interesting results.

References:

- Reeves, C. (2003). Genetic algorithms. In *Handbook of metaheuristics* (pp. 55-82). Springer US.

- De Jong, K. A. (1975). Analysis of the behavior of a class of genetic adaptive systems.

- Goldberg, D. E., Korb, B., & Deb, K. (1989). Messy genetic algorithms: Motivation, analysis, and first results. *Complex systems*, *3*(5), 493-530.
- Franklin, W. R. (2009). PNPOLY – Point inclusion in Polygon Test. [online] Available at: < http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html> [Accessed 13 May 2013].
- Ray, T. S. (1993). An evolutionary approach to synthetic biology: Zen and the art of creating life. *Artificial Life*, *1*(1_2), 179-209.
- Ibackstrom . *Geometry concepts: Line Intersection and its Applications* [online] Available at: <http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry2> [Accessed 5 May 2013].

Appendix A: Complete Testing Data:

Control:

Best:		1	2	3	4	5	6	7	8	9	10	*AVG*
	Num Alive	6	5	6	6	5	4	6	6	6	7	*5.7*
	Visibility	4	8	4	4	6	8	4	4	6	4	*5.2*
Improvement:												
	Num Alive	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Visibility	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
Average:												
	Num Alive	1.59	1.73	1.59	1.65	1.83	1.81	1.63	1.56	1.69	1.8	*1.688*
	Visibility	20.56	18.82	19.18	19.54	19.4	18.86	19.48	18.8	19.22	19.88	*19.374*
Improvement:												
	Num Alive	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
	Visibility	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

Low Level, Dead/Alive

Solution:		1	2	3	4	5	6	7	8	9	10	*AVG*
	Num Alive	0	1	2	2	1	0	0	1	1	2	*1*
	Visibility	28	34	56	24	20	42	24	20	12	38	*29.8*
Improvement:												
	Num Alive	0	-2	1	2	1	-1	0	0	-2	2	*0.1*
	Visibility	8	24	40	-10	-2	14	6	-6	0	18	*9.2*

Low Level, Vis

Solution:		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	***AVG***
	Num Alive	2	0	0	0	0	2	1	2	0	1	*0.8*
	Visibility	24	26	58	36	28	12	22	32	58	12	*30.8*
Improvement:												
	Num Alive	-2	-1	-1	-1	-4	0	-2	0	0	-1	*-1.2*
	Visibility	12	8	38	12	16	-8	-4	16	28	-8	*11*

High Level, Dead/Alive

Best:		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	***AVG***
	Num Alive	8	7	8	8	8	8	8	8	8	8	*7.9*
	Visibility	2	6	2	2	2	2	2	2	2	2	*2.4*
Improvement:												
	Num Alive	4	2	5	4	4	4	4	2	3	3	*3.5*
	Visibility	-10	0	-10	-8	-6	-8	-6	-8	-6	-8	*-7*
Average:												
	Num Alive	7.3	5.6	6.9	7.6	7.4	8	7.3	8	8	7.3	*7.34*
	Visibility	4.6	6.4	5.2	3.4	4	2	3.6	2	2	4	*3.72*
Improvement:												
	Num Alive	5.5	4.4	5.6	5.7	5.2	6.4	5.7	6.1	6.2	5.5	*5.63*
	Visibility	-15.2	-10.8	-15.8	-19.2	-12.6	-20.6	-13	-19.8	-14.2	-12.8	*-15.4*

High Level, Vis

Best:		**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	***AVG***
	Num Alive	8	8	8	10	7	8	6	8	7	6	*7.6*
	Visibility	2	2	2	0	4	2	4	2	4	4	*2.6*
Improvement:												
	Num Alive	6	4	3	6	3	4	2	3	4	1	*3.6*
	Visibility	-14	-10	-6	-12	-4	-10	-6	-6	-6	-2	*-7.6*
Average:												
	Num Alive	6.7	5.2	8	9.2	7	8	6	8	7	6	*7.11*
	Visibility	5.8	7	2	2.2	4	2	4	2	4	4	*3.7*
Improvement:												
	Num Alive	5.8	3.4	6	7.8	5	5.9	4.3	6	5.4	4.2	*5.38*
	Visibility	-17.4	-11.4	-14.4	-19.4	-14.4	-22.2	-14.6	-15.4	-13.4	-15	*-15.76*

Appendix B: Complete Code:

Language: C#

References: System, System.Data, System.Xml, System.Xml.Linq

Lines indicate the end of one file and the beginning of the next.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cowboys
{
    /// <summary>
    /// A Cowboy is a point in space, it can keep track of how many other cowboys can see it and from
    /// this work out if it is alive or dead
    /// </summary>
    class Cowboy
    {
        private decimal[] location; //location as a 2 element array
        private Problem problem; //problem associated with this cowboy
        private int numSeen; //the number of other cowboys that can see this cowboy

        /// <summary>
        /// Returns the location of the cowboy as an array, uses decimal primitive to avoid floating
point errors.
        /// </summary>
        public decimal[] Location
        {
            get { return location; }
        }
        /// <summary>
        /// Checks how many other cowboys the cowboy can see
        /// </summary>
        public int NumSeen
        {
            get { return numSeen; }
        }
        /// <summary>
        /// Checks if the cowboy is dead, that is, if the number of other cowboys that have seen it is
greater than 0
        /// </summary>
        public bool isDead
        {
            get { return NumSeen > 0; }
        }

        /// <summary>
        /// Sets up the cowboy with a random location
        /// </summary>
        /// <param name="problem">The problem the cowboy is a potential solution to</param>
        public Cowboy(Problem problem)
        {
            this.problem = problem;
            Random r = problem.Random;
            location = new decimal[] { (decimal)(r.Next(Problem.SizeX) + r.NextDouble()),
(decimal)(r.Next(Problem.SizeY) + r.NextDouble()) };
            checkLocation();
        }

        /// <summary>
        /// An overloaded constructor for the cowboy that allows its location to be specified.
        /// </summary>
        /// <param name="problem">The problem the cowboy is a potential solution to</param>
        /// <param name="location">The desired location for the cowboy</param>
        public Cowboy(Problem problem, decimal[] location)
        {
            this.problem = problem;
            this.location = location;
            checkLocation();
        }

        /// <summary>
        /// Checks the cowboy is not inside an obstacle, and if it is moves it to the edge of that
obstacles bounding square
        /// uses the PNPoly algorithm found at:
http://www.ecse.rpi.edu/~wrf/Research/Short_Notes/pnpoly.html#The%20C%20Code.
        /// </summary>
        private void checkLocation()
        {
            Obstacle[] obstacles = problem.Obstacles;
            foreach (Obstacle o in obstacles)
            {
                if(location[0] < o.right() && location[0] > o.left() && location[1] < o.bottom() &&
location[1] > o.top())
                {
                    int i, j = 0;
                    bool c = false;
                    int nvert = 4;
                    decimal[] vertx = new decimal[4];
                    decimal[] verty = new decimal[4];
                    decimal[,] points = o.Points;
                    for (int ptToExamine = 0; ptToExamine < 4; ptToExamine++)
                    {
                        vertx[ptToExamine] = points[ptToExamine, 0];
                        verty[ptToExamine] = points[ptToExamine, 1];
                    }
                    decimal testx = location[0];
                    decimal testy = location[1];
                    for (i = 0, j = nvert - 1; i < nvert; j = i++)
                    {
                        if (((verty[i] > testy) != (verty[j] > testy)) &&
                            (testx < (vertx[j] - vertx[i]) * (testy - verty[i]) / (verty[j] -
verty[i]) + vertx[i]))
                        {
                            c = !c;
                        }
                    }
                    if (c) //then move the cowboy to a location not inside an obstacle
                    {
                        Random r = problem.Random;
                        switch (r.Next(4))
                        {
                            case 0: location[0] = o.left() - 1; break;
                            case 1: location[0] = o.right() + 1; break;
                            case 2: location[1] = o.top() - 1; break;
                            case 3: location[1] = o.bottom() + 1; break;
                        }
                    }
                }
            }
        }

        /// <summary>
        /// Informs the cowboy it can see another, and therefore increments the numseen counter
        /// </summary>
        public void seen()
        {
            numSeen++;
        }

        /// <summary>
        /// Resets the number of times the cowboys has been seen to 0
        /// </summary>
        public void resetSeen()
        {
            numSeen = 0;
        }

        /// <summary>
        /// Returns the better cowboy based on the visibility of the cowboy, with better
        /// cowboys having lower visibility.
        /// </summary>
        /// <param name="posCowboy">The cowboy that if better will get the return value 1</param>
        /// <param name="negCowboy">The cowboy that if better will get the return value -1</param>
        /// <returns>1 if posCowboy is better, -1 if negCowboy is better and 0 if they are
equal</returns>
        public static int sortByNumSeen(Cowboy negCowboy, Cowboy posCowboy)
        {
            posCowboy.problem.compare();
            if (posCowboy.NumSeen < negCowboy.NumSeen)
            {
                return 1;
            }
            else if (posCowboy.NumSeen > negCowboy.NumSeen)
            {
                return -1;
            }
            else //if posCowboy.NumSeen == negCowboy.NumSeen
            {
                return 0;
            }
        }

        /// <summary>
        /// Sorts the cowboys by location, with top left being placed first
        /// </summary>
        /// <param name="negCowboy">The cowboy which will elicit a -1 response if it is found to be
more NW than the other</param>
        /// <param name="posCowboy">The cowboy which will elicit a 0 response if it is found to be
more NW than the other</param>
        /// <returns>-1 or 1 depending on which cowboy is the furthest NW or 0 if they are
equal</returns>
        public static int sortByLocation(Cowboy negCowboy, Cowboy posCowboy)
        {
            negCowboy.problem.compare();
            decimal[] negLoc = negCowboy.Location;
            decimal[] posLoc = posCowboy.Location;
            if (negLoc[1] < posLoc[1])
            {
                return -1;
            }
            else if (negLoc[1] > posLoc[1])
            {
                return 1;
            }
            else
            {
                if (negLoc[0] < posLoc[0])
                {
                    return -1;
                }
                else if (negLoc[0] > posLoc[0])
                {
                    return 1;
                }
                else
                {
                    return 0;
                }
            }
        }

        /// <summary>
        /// Sorts the cowboys by if they are alive or not
        /// </summary>
        /// <param name="negCowboy">The cowboy that gives a -1 response if it is alive and the other
isn't</param>
        /// <param name="posCowboy">The cowboy that gives a 1 response if it is alive and the other
isn't</param>
        /// <returns>-1 or 1 if one cowboy is alive and the other isnt 0 otherwise</returns>
        public static int sortAlive(Cowboy negCowboy, Cowboy posCowboy)
        {
            negCowboy.problem.compare();
            int ret = 0;
            if ((negCowboy.isDead && posCowboy.isDead) || (!negCowboy.isDead && !posCowboy.isDead))
            {
            }
            else if (negCowboy.isDead && !posCowboy.isDead)
            {
                ret = 1;
            }
            else if (posCowboy.isDead && !negCowboy.isDead)
            {
                ret = -1;
            }
            return ret;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cowboys
{
    /// <summary>
    /// A Line represents a segment of an infinite length line defined by 2 points.
    /// </summary>
    class Line
    {
        private decimal[] point1; //(x1, y1)
        private decimal[] point2; //(x2, y2)
        //Accessor methods for the different parts of the points
        private decimal X1
        {
            get { return point1[0]; }
        }
        private decimal X2
        {
            get { return point2[0]; }
        }
        private decimal Y1
        {
            get { return point1[1]; }
        }
        private decimal Y2
        {
            get { return point2[1]; }
        }
        //From previous points we transform the line into the form Ax + By = C
        public decimal A
        {
            get { return Y2 - Y1; }
        }
        public decimal B
        {
            get { return X1 - X2; }
        }
        public decimal C
        {
            get { return A * X1 + B * Y1; }
        }
        /// <summary>
        /// Initialises the Line with 2 points
        /// </summary>
        /// <param name="point1">The first point of the line</param>
        /// <param name="point2">The second point of the line</param>
        public Line(decimal[] point1, decimal[] point2)
        {
            this.point1 = point1;
            this.point2 = point2;
        }

        /// <summary>
        /// Method to check if 2 finite length lines intersect, based off of information
        /// found at:
        /// http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=geometry2 [Accessed 5/5/13]
        /// </summary>
        /// <param name="l">The line to check for intersection with</param>
        /// <returns>True if the lines intersect false otherwise</returns>
        public bool intersects(Line l)
        {
            bool intersect = false;
            decimal det = A * l.B - l.A * B;
            if (det != 0)//lines are not parallel
            {
                try
                {
                    //get where the lines intersect
                    decimal intersectX = (l.B * C - B * l.C) / det;
                    decimal intersectY = (A * l.C - l.A * C) / det;

                    //check if this point is on both lines
                    bool xOnThisLine = Math.Min(X1, X2) <= intersectX
                        && intersectX <= Math.Max(X1, X2);
                    bool xOnL = Math.Min(l.X1, l.X2) <= intersectX
                        && intersectX <= Math.Max(l.X1, l.X2);

                    bool yOnThisLine = Math.Min(Y1, Y2) <= intersectY
                        && intersectY <= Math.Max(Y1, Y2);
                    bool yOnL = Math.Min(l.Y1, l.Y2) <= intersectY
                        && intersectY <= Math.Max(l.Y1, l.Y2);

                    //Update the flag to be returned
                    intersect = xOnThisLine && xOnL && yOnThisLine && yOnL;
                }
                catch (OverflowException e)
                {
                    intersect = false;
                }
            }
            return intersect;
        }

        /// <summary>
        /// Checks if a line intersects another but doesn't touch it. A line touches another if
        /// the two lines share a point in common. Used in deciding where the edges of the obstacles
are
        /// </summary>
        /// <param name="l">The line to check if this line intersects but doesn't just touch
it</param>
        /// <returns>True if the line intersects the other, false if the lines that intersect share a
point in common</returns>
        public bool intersectsNotTouches(Line l)
        {
            if(intersects(l))
            {
                if ((this.point1[0] == l.point1[0] && this.point1[1] == l.point1[1]) ||
                    (this.point2[0] == l.point2[0] && this.point2[1] == l.point2[1]) ||
                    (this.point1[0] == l.point2[0] && this.point1[1] == l.point2[1]) ||
                    (this.point2[0] == l.point1[0] && this.point2[1] == l.point1[1]))
                {
                    return false;
                }
                else
                {
                    return true;
                }

            }
            else
            {
                return false;
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cowboys
{
    /// <summary>
    /// An obstacle contains 4 points and can transmute these points into a series of lines for
    /// line of sight checking. Obstacles cannot handle angles above 180 degrees in the angles between
    /// sides since the way they create the lines is by drawing all possible lines are removing those
that
    /// overlap.
    /// </summary>
    class Obstacle
    {
        private decimal[,] points;
        public decimal[,] Points
        {
            get { return points; }
        }
        private Problem problem;

        /// <summary>
        /// Sets up the obstacle with a semi random number of sides
        /// </summary>
        /// <param name="problem">The problem this obstacle is related to</param>
        public Obstacle(Problem problem)
        {
            this.problem = problem;
            points = new decimal[Problem.ObstSides, 2];
            //add random init code
        }

        /// <summary>
        /// An overloaded constructor that allows the specification of the points of the obstacle,
        /// as well as the problem it is related to.
        /// </summary>
        /// <param name="problem">The problem this obstacle is related to</param>
        /// <param name="points">The points that make up the vertices of this object</param>
        public Obstacle(Problem problem, decimal[,] points)
        {
            this.problem = problem;
            this.points = points;
        }

        /// <summary>
        /// Returns the obstacle as an array of lines to allow for checking of view lines
        /// </summary>
        /// <returns>The obstacle as a set of lines, with each line representing an edge of the
obstacle</returns>
        public Line[] toLines()
        {
            List<Line> lines = new List<Line>();
            for (int i = 0; i < 3; i++)
            {
                for (int j = i + 1; j < 4; j++)
                {
                    lines.Add(new Line(new decimal[] { points[i, 0], points[i, 1] }, new decimal[] {
points[j, 0], points[j, 1] }));
                }
            }
            List<Line> removeList = new List<Line>();
            for (int i = 0; i < lines.Count - 1; i++)
            {
                for (int j = i + 1; j < lines.Count; j++)
                {
                    if(lines[i].intersectsNotTouches(lines[j]))
                    {
                        removeList.Add(lines[i]);
                        removeList.Add(lines[j]);
                    }
                }
            }
            foreach (Line rem in removeList)
            {
                lines.Remove(rem);
            }
            Line[] l = lines.ToArray<Line>();
            return l;
        }

        /// <summary>
        /// Returns the bottom of the shape's bounding rectangle
        /// </summary>
        /// <returns>The location of the bottom of the shape's bounding rectangle</returns>
        public decimal bottom()
        {
            decimal[] yVals = new decimal[4];
            for (int i = 0; i < 4; i++)
            {
                yVals[i] = points[i, 1];
            }
            return Enumerable.Max(yVals);
        }

        /// <summary>
        /// Returns the top of the shape's bounding rectangle
        /// </summary>
        /// <returns>The location of the top of the shape's bounding rectangle</returns>
        public decimal top()
        {
            decimal[] yVals = new decimal[4];
            for (int i = 0; i < 4; i++)
            {
                yVals[i] = points[i, 1];
            }
            return Enumerable.Min(yVals);
        }

        /// <summary>
        /// Returns the left of the shape's bounding rectangle
        /// </summary>
        /// <returns>The location of the left of the shape's bounding rectangle</returns>
        public decimal left()
        {
            decimal[] xVals = new decimal[4];
            for (int i = 0; i < 4; i++)
            {
                xVals[i] = points[i, 0];
            }
            return Enumerable.Min(xVals);
        }

        /// <summary>
        /// Returns the left of the shape's bounding rectangle
        /// </summary>
        /// <returns>The location of the right of the shape's bounding rectangle</returns>
        public decimal right()
        {
            decimal[] xVals = new decimal[4];
            for (int i = 0; i < 4; i++)
            {
                xVals[i] = points[i, 0];
            }
            decimal right = Enumerable.Max(xVals);
            return right;
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cowboys
{
    /// <summary>
    /// The control problem is the control for this data. It contains 100 random solutions instead of 
the
    /// normal 10. When executed it sets the number of comparisons to equal to the target amount.
    /// </summary>
    class ControlProblem : Problem
    {
        /// <summary>
        /// Sets up the solutions, overrides the default Problem method
        /// </summary>
        protected override void setUpSolutions()
        {
            solutions = new List<Solution>();
            for (int i = 0; i < 100; i++)
            {
                solutions.Add(new Solution(this));
            }
        }

        /// <summary>
        /// Sets the number of comparisons to the targer amount
        /// </summary>
        public override void execute()
        {
            comparisons = targetComparisons;
        }
    }

    /// <summary>
    /// The HighLvlOnlyVisProblem breeds at the Solution level, and uses the number of cowboys still 
alive in
    /// each solution to find the fitness function.
    /// </summary>
    class HighLvlOnlyDeadProblem : Problem
    {
        /// <summary>
        /// Override of the abstract execute method in Problem. Judges the solutions by number
        /// of cowboys alive, then kills half and repopulates using the survivors
        /// </summary>
        public override void execute()
        {
            solutions.Sort(Solution.judgeByNumAlive);
            solutions.RemoveRange(numSolns / 2, numSolns / 2);
            Solution[] remaining = solutions.ToArray<Solution>();
            while (solutions.Count < numSolns)
            {
                solutions.Add(breedNewSolution(remaining));
            }
        }
    }

    /// <summary>
    /// The HighLvlOnlyVisProblem breeds at the Solution level, and uses the total visibility of the
    /// cowboys to work out the fitness function.
    /// </summary>
    class HighLvlOnlyVisProblem : Problem
    {
        /// <summary>
        /// Override of the abstract execute method in Problem. Judges the solutions total visibility
        /// then kills half and repopulates using the survivors
        /// </summary>
        public override void execute()
        {
            solutions.Sort(Solution.judgeByTotalVisibility);
            solutions.RemoveRange(numSolns / 2, numSolns / 2);
            Solution[] remaining = solutions.ToArray<Solution>();
            while (solutions.Count < numSolns)
            {
                solutions.Add(breedNewSolution(remaining));
            }
        }
    }

    /// <summary>
    /// The LowLvlOnlyProblem breeds at the Cowboy level, it uses the status of each cowboy to decide
how to sort it
    /// </summary>
    class LowLvlOnlyDeadProblem : Problem
    {
        /// <summary>
        /// Override of the set up solutions method in Problem. Used since the low level approach only
needs one solution
        /// </summary>
        protected override void setUpSolutions()
        {
            solutions = new List<Solution>();
            solutions.Add(new Solution(this));
            initialBestVisibility = solutions[0].getVisibility();
            initialBestAlive = solutions[0].getNumAlive();
        }

        /// <summary>
        /// Overrides the execute method in Problem, eliminates the worst performing cowboys and
        /// breeds new ones using whether the cowboy is dead or alive as the fitness function.
        /// </summary>
        public override void execute()
        {
            Cowboy[] cowboys = solutions[solutions.Count - 1].Cowboys;
            List<Cowboy> sortingList = new List<Cowboy>();
            sortingList.AddRange(cowboys);
            sortingList.Sort(Cowboy.sortAlive);
            sortingList.RemoveRange(NumCowboys / 2, NumCowboys / 2);
            Cowboy[] remaining = sortingList.ToArray<Cowboy>();
            while (sortingList.Count < NumCowboys)
            {
                sortingList.Add(breedNewCowboy(remaining));
            }
            solutions[0] = new Solution(this, sortingList.ToArray<Cowboy>());
        }
    }

    /// <summary>
    /// The LowLvlOnlyProblem breeds at the Cowboy level, it uses the visibility of each cowboy to
decide how to sort it
    /// </summary>
    class LowLvlOnlyVisibilityProblem : Problem
    {
        /// <summary>
        /// Override of the set up solutions method in Problem. Used since the low level approach only
needs one solution
        /// </summary>
        protected override void setUpSolutions()
        {
            solutions = new List<Solution>();
            solutions.Add(new Solution(this));
            initialBestVisibility = solutions[0].getVisibility();
            initialBestAlive = solutions[0].getNumAlive();
        }

        /// <summary>
        /// Overrides the execute method in Problem, eliminates the worst performing cowboys and
        /// breeds new ones using the visibility as a fitness function
        /// </summary>
        public override void execute()
        {
            Cowboy[] cowboys = solutions[solutions.Count - 1].Cowboys;
            List<Cowboy> sortingList = new List<Cowboy>();
            sortingList.AddRange(cowboys);
            sortingList.Sort(Cowboy.sortByNumSeen);
            sortingList.RemoveRange(NumCowboys / 2, NumCowboys / 2);
            Cowboy[] remaining = sortingList.ToArray<Cowboy>();
            while (sortingList.Count < NumCowboys)
            {

                sortingList.Add(breedNewCowboy(remaining));
            }
            solutions[0] = new Solution(this, sortingList.ToArray<Cowboy>());
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;

namespace Cowboys
{
    /// <summary>
    /// The abstract class problem deals with loading the world and contains easily overridable
methods to allow
    /// the implementation of different genetic algorithms
    /// </summary>
    abstract class Problem
    {
        protected List<Obstacle> obstacles;
        public Obstacle[] Obstacles
        {
            get { return obstacles.ToArray<Obstacle>(); }
        }
        protected List<Line> lines;
        public Line[] Lines
        {
            get { return lines.ToArray<Line>(); }
        }
        protected List<Solution> solutions;
        public List<Solution> Solutions
        {
            get { return solutions; }
        }
        protected Random rand;
        //Changing these is only partially implemented, and so they shouldn't be changed.
        protected const int numSolns = 10;
        public const int NumCowboys = 10;
        public const int ObstSides = 4;

        protected int comparisons;
        protected const int targetComparisons = 100000;

        protected int initialBestVisibility;
        public int InitialBestVisibility
        {
            get { return initialBestVisibility; }
        }
        protected int initialBestAlive;
        public int InitialBestAlive
        {
            get { return initialBestAlive; }
        }
        protected decimal initialAvgAlive;
        public decimal InitialAvgAlive
        {
            get { return initialAvgAlive; }
        }
        protected decimal initialAvgVis;
        public decimal InitialAvgVis
        {
            get { return initialAvgVis; }
        }

        public Random Random
        {
            get { return rand; }
        }
        public const int SizeX = 800;
        public const int SizeY = 600;

        //P(Mutation) = 1/mutationChance
        protected const int mutationChance = 10;
        public int MutationChance
        {
            get { return mutationChance; }
        }

        /// <summary>
        /// Initialises the problem and calls the method to set up the solutions and run the algorithm
        /// </summary>
        public Problem()
        {
            rand = new Random();
            comparisons = 0;
            parseWorld();
            //convert obstacles into a list of lines
            lines = new List<Line>();
            foreach (Obstacle o in obstacles)
            {
                lines.AddRange(o.toLines());
            }
            setUpSolutions();
            run();
        }

        /// <summary>
        /// Tells the problem a comparison has been made, and increments the comparison variable.
        /// </summary>
        public void compare()
        {
            comparisons++;
        }

        /// <summary>
        /// Runs the program until a set number of comparisons have been reached. Uses the execute
        /// method which should be overriden in each subclass of Problem.
        /// </summary>
        public void run()
        {
            while (comparisons < targetComparisons)
            {
               execute();
            }
            solutions.Sort(Solution.judgeByNumAlive);
        }

        /// <summary>
        /// Set the solutions to a number of random solutions, overridden for lower level only
breeding
        /// since that doesn't need more than one solution object
        /// </summary>
        protected virtual void setUpSolutions()
        {
            solutions = new List<Solution>();
            for (int i = 0; i < numSolns; i++)
            {
                solutions.Add(new Solution(this));
            }
            initialBestVisibility = getBestVisibility().getVisibility();
            initialBestAlive = getBestNumAlive().getNumAlive();
            initialAvgVis = getAvgVisibility();
            initialAvgAlive = getAvgNumAlive();
        }

        /// <summary>
        /// Parses the world from an xml file and sets it up.
        /// </summary>
        private void parseWorld()
        {
            obstacles = new List<Obstacle>();
            var xml = XDocument.Load("C:/Users/Owen/Documents/GitHub/Cowboys/Cowboys/World1.xml");
            IEnumerable<string> obstStrings = from o in xml.Descendants("Obstacle")
                                              select o.Value;
            foreach (string s in obstStrings)
            {
                decimal[,] points = new decimal[4,2];
                int numPoints = 0;
                string[] pts = s.Split(':');
                foreach (string point in pts)
                {
                    string[] stringCoOrds = point.Split(',');
                    points[numPoints, 0] = int.Parse(stringCoOrds[0]);
                    points[numPoints, 1] = int.Parse(stringCoOrds[1]);
                    numPoints++;
                }
                obstacles.Add(new Obstacle(this, points));
            }

        }

        /// <summary>
        /// Gets the average number of cowboys alive in the current solution pool
        /// </summary>
        /// <returns>The average number of cowboys alive in the current solution pool</returns>
        public decimal getAvgNumAlive()
        {
            decimal i = 0;
            foreach (Solution s in solutions)
            {
                i += s.getNumAlive();
            }
            i /= solutions.Count;
            return i;
        }

        /// <summary>
        /// Gets the average total visibility of each solution in the current solution pool
        /// </summary>
        /// <returns>The average total visibility of each solution in the current solution 
pool</returns>
        public decimal getAvgVisibility()
        {
            decimal i = 0;
            foreach (Solution s in solutions)
            {
                i += s.getVisibility();
            }
            i /= solutions.Count;
            return i;
        }

        /// <summary>
        /// Breeds a new solution at random from a list of possible parents
        /// </summary>
        /// <param name="solutionsList">The list of potential parent</param>
        /// <returns>A solution gotten by breeding 2 of the solutions on the list</returns>
        public virtual Solution breedNewSolution(Solution[] solutionsList)
        {
            if (Random.Next(MutationChance) == 0)
            {
                return new Solution(this);
            }
            else
            {
                Solution s1 = solutionsList[rand.Next(solutionsList.Length - 1)];
                Solution s2 = solutionsList[rand.Next(solutionsList.Length - 1)];
                while (s1 == s2)
                {
                    s2 = solutionsList[rand.Next(solutionsList.Length - 1)];
                }
                List<Cowboy> completeGenes = new List<Cowboy>();
                for (int i = 0; i < NumCowboys; i++)
                {
                    if (i % 2 == 0)
                    {
                        Cowboy c = s1.Cowboys[i];
                        completeGenes.Add(new Cowboy(this, new decimal[] { c.Location[0], 
c.Location[1] }));
                    }
                    else
                    {
                        Cowboy c = s2.Cowboys[i];
                        completeGenes.Add(new Cowboy(this, new decimal[] { c.Location[0],
c.Location[1] }));
                    }
                }
                return new Solution(this, completeGenes.ToArray<Cowboy>());
            }
        }

        /// <summary>
        /// Gets the solution with the most cowboys alive in the current pool
        /// </summary>
        /// <returns>The solution with the most cowboys alive</returns>
        public Solution getBestNumAlive()
        {
            int numAlive = 0;
            Solution best = null;
            foreach (Solution s in solutions)
            {
                if (s.getNumAlive() >= numAlive)
                {
                    numAlive = s.getNumAlive();
                    best = s;
                }
            }
            return best;
        }

        /// <summary>
        /// Gets the solution with the best visibility in the current pool
        /// </summary>
        /// <returns>The solution with the lowest total visibility</returns>
        public Solution getBestVisibility()
        {
            int visibility = int.MaxValue;
            Solution best = null;
            foreach (Solution s in solutions)
            {
                if (s.getVisibility() <= visibility)
                {
                    visibility = s.getVisibility();
                    best = s;
                }
            }
            return best;
        }

        /// <summary>
        /// Breeds a new cowboy selecting the parents randomly from a specified list
        /// </summary>
        /// <param name="cowboysList">The list of cowboys that are potential parents for the new
cowboy</param>
        /// <returns>A new cowboy bred from 2 random parents in the provided list</returns>
        public Cowboy breedNewCowboy(Cowboy[] cowboysList)
        {
            if (Random.Next(MutationChance) == 0)
            {
                return new Cowboy(this);
            }
            else
            {
                Cowboy cb1 = cowboysList[Random.Next(cowboysList.Length)];
                Cowboy cb2 = cowboysList[Random.Next(cowboysList.Length)];
                return (breedNewCowboy(cb1, cb2));
            }
        }

        /// <summary>
        /// Breeds a new cowboy with the specified parents.
        /// </summary>
        /// <param name="c1">The first parent of the new cowboy</param>
        /// <param name="c2">The second parent of the new cowboy</param>
        /// <returns>The cowboy that is the offspring of the 2 parent cowboys provided</returns>
        public Cowboy breedNewCowboy(Cowboy c1, Cowboy c2)
        {
            return new Cowboy(this, new decimal[] { c1.Location[0], c2.Location[1] });
        }

        /// <summary>
        /// Execute should do some kind of breeding action
        /// </summary>
        public abstract void execute();
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cowboys
{
    /// <summary>
    /// The runner class contains the main method, it runs the different problems in order and returns
the result
    /// </summary>
    class Runner
    {
        public static void Main(string[] args)
        {
            ControlProblem control = new ControlProblem();
            write(control, "Control:");

            LowLvlOnlyDeadProblem low = new LowLvlOnlyDeadProblem();
            write(low, "Low Level Only, Dead vs Alive:");

            LowLvlOnlyVisibilityProblem lowV = new LowLvlOnlyVisibilityProblem();
            write(lowV, "Low Level Only, Visibility:");

            HighLvlOnlyDeadProblem hListAD = new HighLvlOnlyDeadProblem();
            write(hListAD, "High Level Only, Dead vs Alive:");

            HighLvlOnlyVisProblem hListV = new HighLvlOnlyVisProblem();
            write(hListV, "High Level Only, Visibility:");

            Console.Read();
        }

        /// <summary>
        /// The write class writes the important information about a problem to the console
        /// </summary>
        /// <param name="p">The problem to write to the console</param>
        /// <param name="title">The title to begin the section with</param>
        private static void write(Problem p, string title)
        {
            int i = title.Length;
            string topBottom = "";
            for (int j = 0; j < i + 6; j++)
            {
                topBottom += "X";
            }
            Console.WriteLine(topBottom);
            Console.WriteLine("XX " + title + " XX");
            Console.WriteLine(topBottom);
            Console.WriteLine();
            Solution t = p.getBestNumAlive();
            Console.WriteLine("Number alive (Best): " + t.getNumAlive());
            Console.WriteLine("Visibility (Best): " + p.getBestVisibility().getVisibility());
            Console.WriteLine("Improvement: Number Alive (Best): " + (t.getNumAlive() -
p.InitialBestAlive));
            Console.WriteLine("Improvement: Visibility (Best): " +
(p.getBestVisibility().getVisibility() - p.InitialBestVisibility));
            if (p.Solutions.Count != 1)
            {
                Console.WriteLine("_________________________________________________________");
                Console.WriteLine("Number alive (Mean): " + p.getAvgNumAlive());
                Console.WriteLine("Visibility (Mean): " + p.getAvgVisibility());
                Console.WriteLine("Improvement: Number Alive (Mean): " + (p.getAvgNumAlive() -
p.InitialAvgAlive));
                Console.WriteLine("Improvement: Visibility (Mean): " + (p.getAvgVisibility() -
p.InitialAvgVis));
               /* Console.WriteLine("_________________________________________________________");
                Console.WriteLine("Number Alive (Final Set):");
                foreach (Solution s in p.Solutions)
                {
                    Console.Write(s.getNumAlive() + ", ");
                }
                Console.WriteLine();
                Console.WriteLine("Visibility (Final Set):");
                foreach (Solution s in p.Solutions)
                {
                    Console.Write(s.getVisibility() + ", ");
                }
                Console.WriteLine();*/
            }
            Console.WriteLine("=========================================================");
            Console.WriteLine();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Cowboys
{
    /// <summary>
    /// A solution is a set of 10 cowboys, and is responsible for checking which of them can see each
other.
    /// </summary>
    class Solution
    {
        private Cowboy[] cowboys;
        public Cowboy[] Cowboys
        {
            get { return cowboys; }
        }

        //The problem associates with this solution, used to find the location of the obstacles
        //informed each time a comparison is made
        private Problem problem;

        /// <summary>
        /// Initialises the solution and associates it with a specified problem.
        /// </summary>
        /// <param name="problem">The problem to associate this solution with</param>
        public Solution(Problem problem)
        {
            this.problem = problem;
            cowboys = new Cowboy[Problem.NumCowboys];
            for (int i = 0; i < Problem.NumCowboys; i++)
            {
                cowboys[i] = new Cowboy(problem);
            }
            checkCowboys();
        }

        /// <summary>
        /// Initialises the solution providing it the list of cowboys and the problem to
        /// be associated with
        /// </summary>
        /// <param name="problem">The problem to associate this solution with</param>
        /// <param name="cowboys">The list of cowboys to use for this solution</param>
        public Solution(Problem problem, Cowboy[] cowboys)
        {
            this.problem = problem;
            this.cowboys = cowboys;
            checkCowboys();
        }

        /// <summary>
        /// Checks which cowboys can see each other
        /// </summary>
        public void checkCowboys()
        {
            foreach (Cowboy c in cowboys)
            {
                c.resetSeen();
            }
            for (int i = 0; i < cowboys.Length - 1; i++) //only check each pair once
            {
                for (int j = i + 1; j < cowboys.Length; j++)
                {
                    Cowboy cb1 = cowboys[i];
                    Cowboy cb2 = cowboys[j];
                    Line l = new Line(cb1.Location, cb2.Location);
                    int k = 0;
                    Line[] lines = problem.Lines;
                    bool intersects = false;
                    while (!intersects && k < lines.Length)
                    {
                        if(l.intersects(lines[k]))
                        {
                            intersects = true;
                        }
                        else
                        {
                            k++;
                        }
                    }
                    if(!intersects) //If the line between them hasn't intersected any obstacle lines
                    {
                        cb1.seen();
                        cb2.seen();
                    }
                }
            }
        }

        /// <summary>
        /// Gets the count of the number of cowboys in the solution still alive
        /// </summary>
        /// <returns>The number of cowboys alive in this solution</returns>
        public int getNumAlive()
        {
            int alive = 0;
            foreach (Cowboy c in cowboys)
            {
                if (!c.isDead)
                {
                    alive++;
                }
            }
            return alive;
        }

        /// <summary>
        /// Judges 2 solutions by how many cowboys they have alive in them
        /// </summary>
        /// <param name="negSoln">The solution which if better elicits a -1 response</param>
        /// <param name="posSoln">The solution which if better elicits a 1 response</param>
        /// <returns>-1 if the negative solution is better, 1 if the positive solution is better, 0 if
they are equal</returns>
        public static int judgeByNumAlive(Solution negSoln, Solution posSoln)
        {
            //tell the problem we are making a comparison
            posSoln.problem.compare();
            //first check if they are null
            if (posSoln == null && negSoln == null)
            {
                return 0;
            }
            else if (posSoln == null)
            {
                return -1;
            }
            else if (negSoln == null)
            {
                return 1;
            }
            else
            {
                //Here we want to select the one with the maximum number of living cowboys
                int posAlive = posSoln.getNumAlive();
                int negAlive = negSoln.getNumAlive();
                if (posAlive > negAlive)
                {
                    return 1;
                }
                else if (posAlive < negAlive)
                {
                    return -1;
                }
                else //if posAlive == negAlive
                {
                    return 0;
                }
            }
        }

        /// <summary>
        /// Judges 2 solutions their total visibility, with lower being better
        /// </summary>
        /// <param name="negSoln">The solution which if better elicits a -1 response</param>
        /// <param name="posSoln">The solution which if better elicits a 1 response</param>
        /// <returns>-1 if the negative solution is better, 1 if the positive solution is better, 0 if
they are equal</returns>
        public static int judgeByTotalVisibility(Solution negSoln, Solution posSoln)
        {
            posSoln.problem.compare();
            if (posSoln == null && negSoln == null)
            {
                return 0;
            }
            else if (posSoln == null)
            {
                return -1;
            }
            else if (negSoln == null)
            {
                return 1;
            }
            else
            {
                //In this case we want the lowest value for the number of cowboys that can
                //see each other.
                int numSeenPos = posSoln.getVisibility();
                int numSeenNeg = negSoln.getVisibility();
                if (numSeenPos > numSeenNeg)
                {
                    return -1;
                }
                else if (numSeenPos < numSeenNeg)
                {
                    return 1;
                }
                else //If numSeenPos == numSeenNeg
                {
                    return 0;
                }
            }
        }
        /// <summary>
        /// Gets the total visibility of this solution
        /// </summary>
        /// <returns>The sum of all the visibilities of the cowboys in this solution</returns>
        public int getVisibility()
        {
            int i = 0;
            foreach (Cowboy c in cowboys)
            {
                i += c.NumSeen;
            }
            return i;
        }
    }
}
```

NOTE: This file must be called World1.xml and be in the same folder as the rest of the classes in order to work.

```
<?xml version="1.0" encoding="utf-8" ?>
<World>
  <!-- Obstacles are a set of n points to describe a polygon. Points are seperated by
a : -->
  <!-- Points are of the form x,y -->
  <Obstacle>100,110:75,50:100,210:50,180</Obstacle>
  <Obstacle>250,50:240,30:210,303:360,222</Obstacle>
  <Obstacle>100,420:250,410:190,505:50,530</Obstacle>
  <Obstacle>250,150:310,140:300,250:350,190</Obstacle>
  <Obstacle>350,250:485,310:270,340:270,330</Obstacle>
  <Obstacle>312,480:370,550:405,370:450,430</Obstacle>
  <Obstacle>410,110:430,60:570,150:490,210</Obstacle>
  <Obstacle>580,112:700,20:770,190:640,150</Obstacle>
  <Obstacle>640,160:740,270:650,340:580,300</Obstacle>
  <Obstacle>670,440:715,540:600,540:570,500</Obstacle>
</World>
```