Using Adaptive Algorithms for Better Racing

New Mexico Adventures in
Supercomputing Challenge
Final Report
April 7, 2004

Team 96
Career Enrichment Center

Team Members:
Robert Cordwell

Teachers:
Ken Greenberg
Stephen Schum

Project Mentor:
Stephen Schum

Table of Contents

Using Adaptive Algorithms for Better Racing

Page

Table of Contents……1
Executive Summary……2
Background Section……3
Race Physics……3
Drafting……7
Passing……8
Direct / Instantaneous……8
Slingshot……9
Multi-car 21……10
Deceptive……11
Racing Psychology……11
MGG Background……14
Scientific Method……16
Purpose and Problem Statement……16
Research Plan……16
Conditions……17
Condition Analysis……18
Data……21
Data Analysis……26
Genetic Algorithm Data……28
MGG Data……29
Conclusions……30
Discussion……31
Plot Classes……31
Code Explanation……33
Future Developments……38
Main Accomplishment……39
References……40
Acknowledgements……41
Code……42
Raceplot / Fitness Calculator……42
Genetic Algorithm……61
Minimal Generation Gap GA……68
Additional Data……79
Optimizer……79
MGG Optimizer……89
MGG Continuous……93

Executive Summary

The purpose of this experiment is to determine what would occur when a simulated NASCAR race was used with a genetic algorithm. To do this, a computer algorithm was created which simulated a NASCAR race. Five representative properties for drivers were determined and then programmed into the simulation. These initial properties were: aggression, defense, deception, grudges, and adaptation. The simulation was physics based, but was focused on the game-theoretical interactions between drivers. A basic GA genetic algorithm and MGG were programmed in.

The race algorithm was then set to output the paths of the drivers onto the screen. Initially, the drivers would separate from each other and fail to interact. This was solved by the creation of a function which would occasionally give the drivers commands to cluster together in order to form more powerful draft lines, and the introduction of a loop track rather than a single straight track.

When the race algorithm was satisfactory, a genetic algorithm was run where the fitness calculation was how well the drivers did in the races and the alleles were the five traits. When 7000 iterations were run and the set of driver which had won at least one race was analyzed, it was determined that once a driver won a race, he would on average win over five races (which showed validity). The optimum driver would switch occasionally to a seemingly unrelated driver, suggesting that the race algorithm was highly nontransitive. This result is interesting because previously all (known) uses of a genetic algorithm have been to optimize to a fixed fitness function rather than a variable one.

Three other tests were run – one, a proof of concept, showed that the code could be used to optimize a driver to defeat a fixed set. The other two tests replaced the basic genetic algorithm with the MGG, but surprisingly the basic GA was more successful.

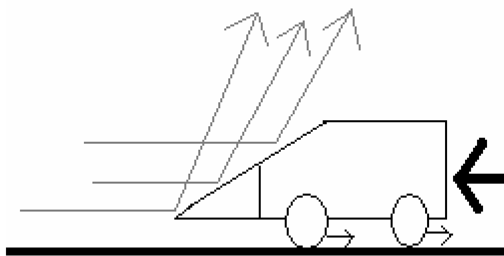
Background

Race Physics

Entire projects and papers could be done simply on the physics of cars, and millions of dollars have been poured into designing aerodynamic racing machines, to say nothing of actually building them. However, the focus of this project is on the game theoretical aspects of racing rather than the physical aspects of racing, and one major assumption is made: that the cars will perform identically under identical conditions.

Incidentally, this is extremely realistic. NASCAR race organizes know full well that a driver with a car which dramatically outperforms the other cars on the field will be able to outrace them handily, which does not make for a particularly interesting race. As such, they have created an entire rulebook with strict limits of nearly every aspect of the car. There are only a few basic "models" which a driver can use, and the regulations have been tweaked so that it there is effectively no difference between these models. Since this project is loosely based on a NASCAR race, it is reasonable enough to assume that the cars are the same.

In the most basic sense, a car is a self-powered box. There is a thrust / drive force with opposing air drag and ground friction forces.



The large black arrow represents the force pushing on the car. In actuality, this force is converted into turning the wheels of the car, but this angular momentum has the sole purpose of making the car move forward. The small black arrows pointing to the left represent friction, which is equal to:

Mass of the car · gravity · coefficient of friction between the wheels and the track + internal friction.

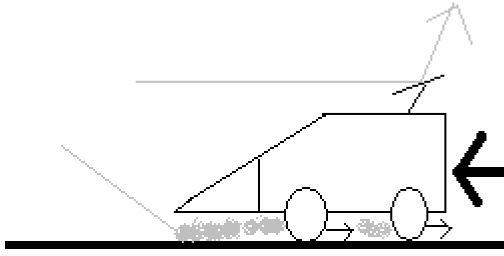
Friction is both a blessing and a curse. Without friction, the car's wheels would spin but this would not be converted into momentum. At the same time, friction causes the car to slow down. Tires in racing are designed to be balanced between the need for rapid acceleration (which requires high friction) and the need for a high maximum speed (where low friction is better).

The gray lines represent air bouncing off of the car. From the car's reference frame, it is standing still while being blasted by a high-velocity wind. Since momentum must be conserved, and each air particle (relative to the car) loses some backwards velocity as it strikes the car, the car is pushed backwards accordingly.

In the racing simulation, all forces have direct effects on the speed of the car (rather than the kinetic energy). In a given unit of time, the simulation calculates friction as being directly proportional to the velocity of the car, while drag is directly proportional to the square of the velocity of the car.

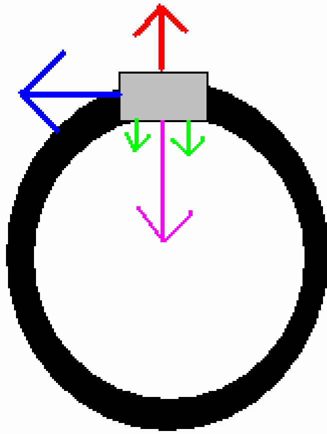
The drag force should be proportional to the amount of air impacting the car multiplied by the velocity of the vehicle. Since drag force involves a multitude of collisions, the only way to measure it is to analyze the conservation of momentum. It is realistic to assume that each air particle, on average, loses a constant fraction of its speed relative to the car (or gains a constant fraction of the car's speed, from the perspective of the air particle). If the car is going twice as fast, it will cover twice as much distance; that is, it will impact twice as many air particles on average. Each of these air particles will gain twice as much velocity, so the car will lose four times the velocity.

The friction force with the track, however, subtracts one unit of kinetic energy for each unit of track covered by the car. Thus, in one unit of time, the car will lose kinetic energy proportional to its speed, which translates as a constant reduction of speed. However, there is one other factor to consider:



All modern racecars have a spoiler mounted on the back. Since the cars are so aerodynamic, they create a powerful lift force because the air moving over the car is lower pressure than the air moving under it, producing a Bernoulli Effect. The spoiler is a wing tilted upside-down. Rather than lifting the car, the spoiler spoils the lift force and keeps the car firmly on the ground. The spoiler is so powerful that, were a suitable course designed, a car could be driver on the ceiling at full speed. Since the spoiler increases the force on the tires, this increases the force of friction by making the car seem heavier than it is. For the purposes of this simulation, at high speeds the force on the tires is proportional to the velocity of the car. Thus friction decreases kinetic energy by a factor proportional to the velocity^2. Since kinetic energy is equal to ½mass·v^2, friction has a linear effect on velocity in the simple model.

There are further complications when turning – the danger that a car will lose traction and rollover off of the road is always present. For the purposes of the simulation, the track was considered to be a super-speedway, a racecourse with well-banked, wide turns.



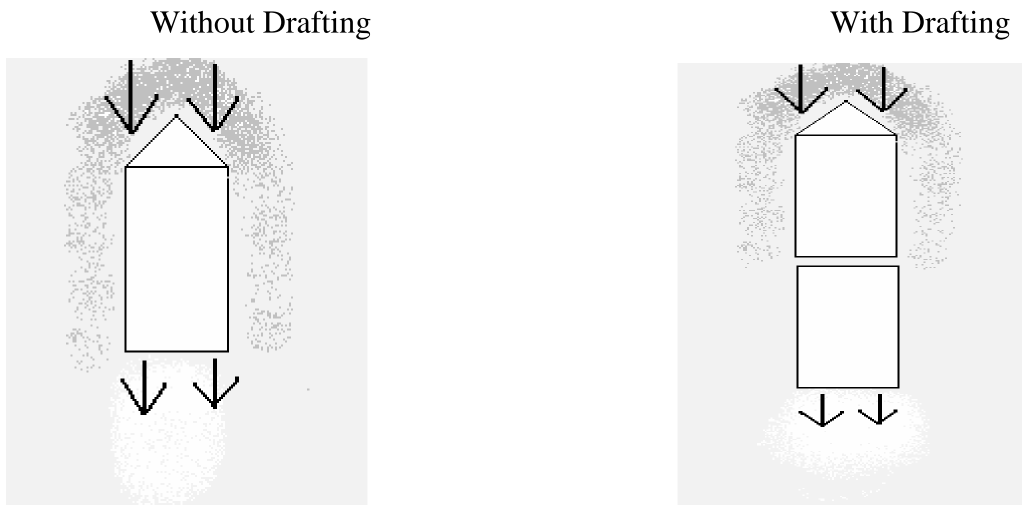
In this diagram, the car is moving around the track at constant velocity. The blue arrow represents his current velocity, while the purple arrow represents his acceleration, since he has to keep turning inward. If his acceleration were sufficient, the driver would not feel any imaginary centrifugal force pushing him to the outside of the circle. The red arrow represents the imbalance in acceleration, while the green arrows are the (static) friction between the wheels and the track which compensates for the imbalance. Since the necessary acceleration without friction is equal to $m \cdot v^2/r$, where m is the mass of the car, v is the velocity, and r is the radius of the circle, the driver may be forced to slow down to go around a curve.

There are several methods of helping the driver out and allowing him to take the curves at maximum velocity. One is to limit the car's speed – an unintended effect of the NASCAR regulations was that drivers were able to take corners at closer to maximum speed, leading to more aggressive driving and more crashes. The second is to increase the radius of the turn. Finally, the third is to bank the curve, adding an additional gravitational force into the mix which tends to pull the car towards the center in the same way that a ball on a sloped hill will roll towards the bottom. The simulation assumes that the race takes place on a super-speedway which, true to its name, has long banked curves which allow cars to go at or very near their top speed around them.

Drafting

Drafting is important primarily because it does not depend on the type of car, but rather on the driver. While the speed gains may be small, the cars are so closely matched in a NASCAR race that small speed advantages can be critical. The potential of drafting was discovered during the second-ever Daytona 500 race in 1960, by Junior Johnson. He was driving an underpowered Chevrolet compared to his rivals' Plymouths, but realized that if he stuck behind one of the more powerful cars, both he and the other car would be able to go faster than their competitors. At the end of the race, he pulled a slingshot maneuver and was able to pass the faster car long enough to win.

A car traveling by itself creates a low-pressure area behind it as it displaces the air before new air can rush in and a high-pressure area ahead of it as it compacts the air. Both of these impede its movement and tend to push it backwards:



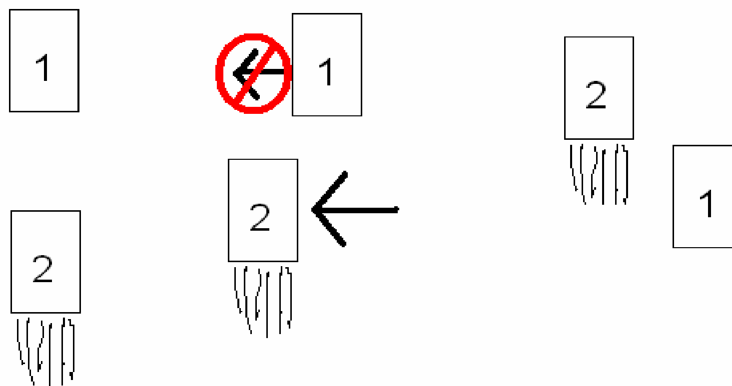
If two cars are running very close together, then they effectively share the disadvantages. Only one of them is impeded by the high-pressure head, while the other only has to deal with the low-pressure tail. This allows both cars to move significantly faster, although the car in back gains the most significant speed boost since, although he has to deal with an area of low pressure behind him, he is constantly being sucked into the foremost driver's vacuum. In general, the more cars in the line, the faster the cars go, but by far the most significant benefit is with two cars as opposed to one.

The model used in the simulation program was that two cars had to be within 10 units of each other (roughly 20 meters) for the front most car to gain any benefit from drafting. Within 100 units, though, the rear car would catch up. The bonus to power was calculated to be equal to 12/(distance + 3) for the rear car and 6/(distance + 3) for the front car. This model was chosen as a tradeoff between simplicity, realism, and its ability to promote drafting.

Passing

Passing is what makes up the game theoretical heart of a race. In a way this is fitting, since a driver may be a meter or a hundred meters from the lead but all that matters when the flag goes down is whether that driver is behind or ahead of the other drivers. Professional drivers have developed it into an art and a science, with such exotic moves as a slight tap on the opponent's car to push him forward enough so the driver can more around him and a delicate dance on the brake and gas pedals to prevent such an occurrence. The social dynamics which go into passing are equally complex, and possibly even more so. Still, despite the complexity, there are four basic types of pass in racecar driving: direct, slingshot, multi-car, and deceptive. Each method is used in a different situation.

Direct / Instantaneous

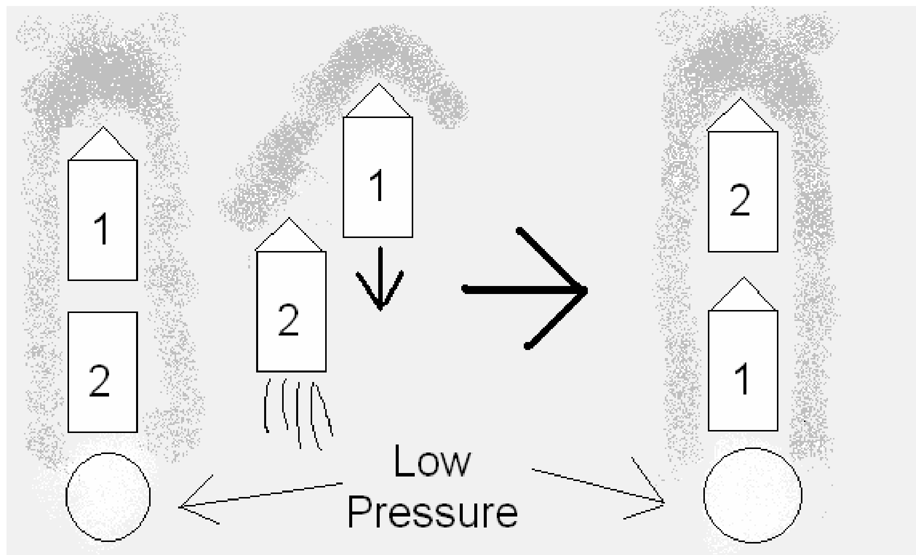


In this diagram, Driver 2 is approaching Driver 1 with high relative velocity (most likely, either Driver 2 has just received a slingshot speed boost from a previous

maneuver, or Driver 1 is damaged). Driver 2 turns to the left and Driver 1 is unable and/or unwilling to risk a collision and so allows the inevitable to happen and the faster Driver 2 passes Driver 1.

In the program, if a driver would run into another car if he were to continue going at its current speed, he is moved backwards. If, however, he is going fast enough to completely pass the slower car, then he is considered to have made a direct pass.

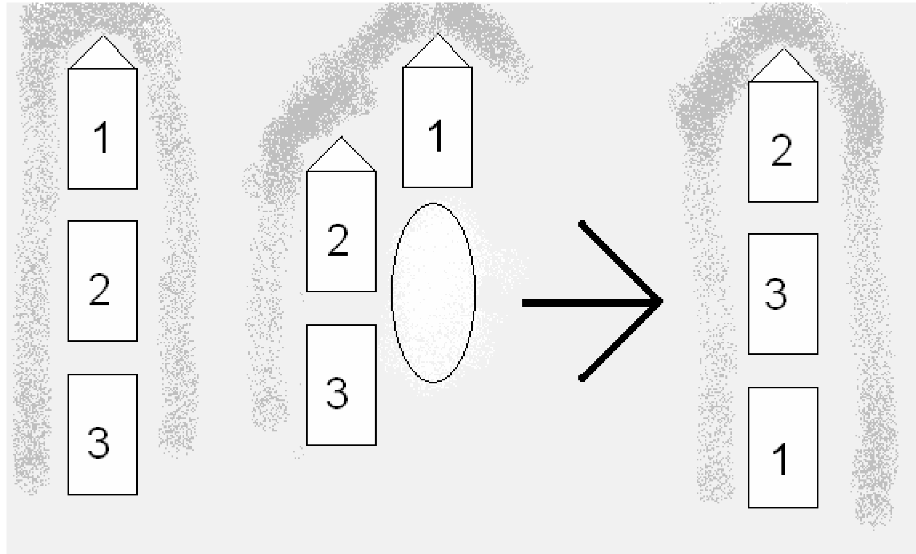
Slingshot



In this plot, the light gray area represents atmosphere, while the dark gray areas show areas of particularly high airflow / air resistance. The white areas are vacuums or areas of low air pressure, which are formed by the Driver displacing the air and then moving on before the air can fill the area.

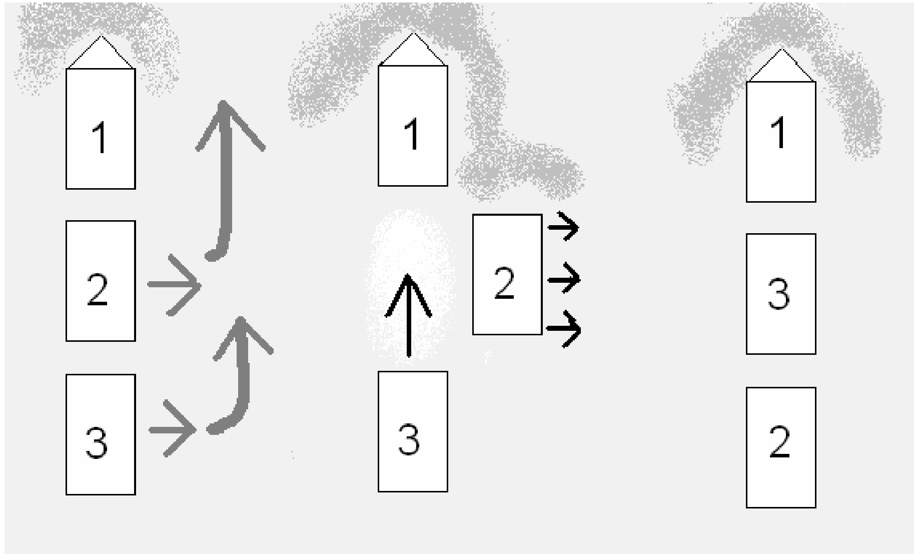
In the classic slingshot maneuver, Initially, Driver 2 is drafting off of driver 1, and there are no other drivers nearby. Driver 1 pushes the air away, so Driver 2's top speed is higher than that of driver 1. Both drivers benefit because Driver 1 would otherwise have to contend with the vacuum which forms behind his Driver and tends to drag it backwards. In this case, Driver 2 wishes to pass Driver 1. Driver 2 speeds up and swings out, creating a vacuum which drags Driver 1 back. Driver 2 then takes the lead; however, Driver 1 could pull the same trick on him shortly. In the simulations, the speed boost and speed decrease were slightly exaggerated so as to promote passing.

Multi-car



Muti-car passes are possibly the most interesting of all, since they involve cooperation between the drivers who normally are locked in a zero-sum game. To understand what makes a multi-car pass unique, suppose that Driver 2 were to attempt to pass Driver 1. In this case, Driver 3 would simply come up behind Driver 1, fill in the vacuum, and Driver 2 would be stuck in the middle of a pass and would be forced to fall behind Driver 3. Similarly, if Driver 3 were to attempt to pass Driver 2, Driver 2 would simply stick close to Driver 1, preventing Driver 3 from getting in without crashing. The only solution is cooperation. Drivers 2 and 3 move out as a unit, creating a double-slingshot effect which pulls Driver 1 backwards. Since Drivers 2 and 3 now form a drafting unit, they can both pull ahead of Driver 1 before any Drivers behind them can fill in the gap. If Driver 2 attempts to trick Driver 3 and shoves himself in front of Driver 1 without leaving a gap for Driver 3, then Driver 3 can slam on the brakes, creating a vacuum behind Driver 2 and potentially allowing Driver 1 to remain in first. Since Driver 2 has little to gain by not leaving a gap for Driver 3, most of the time he would choose to leave such a gap.

Deceptive



Driver 3 has nothing to lose. He's already last and, particularly if the race is nearly over, Drivers 1 and 2 can't do that much to him. Therefore, Driver 3 convinces Driver 2 that he will be willing to attempt a standard multi-car pass. Instead, Driver 3 accelerates forward once Driver 2 has pulled out. This leaves Driver 2 without his drafting partner and, if Driver 3 can close the gap to Driver 1 fast enough, forces Driver 2 to take last place or risk a crash. In the simulation, this method causes Driver 2 to carry a grudge against Driver 3, which causes Driver 2 to resist passing with Driver 3 and even going so far as to intentionally hit Driver 3.

Racing Psychology

The psychology of NASCAR drivers and their relations with each other resembles the classical prisoner's dilemma iterated many times. The same drivers will race with each other for many seasons, which means that all actions will be observed and will have consequences. Game-theoretical decisions occasionally arise for the drivers, and often their decisions will do more for their racing performance than their driving ability.

Racing is very nearly a zero-sum game, but there are many players so for any two players it is not necessarily so. Often two drivers will find that they are playing what is effectively a variant of the prisoner's dilemma against each other.

The prisoner's dilemma is the classical game-theory example used to show how cooperation breaks down. The plot of the game is that two cat burglars are caught in their house. The police have only enough evidence to send them to jail for three years each, so they talk to each one separately and tell each that if he will testify against his partner, they will set him free while confining the other for nine years. On the other hand, if both testify, then both will receive a jail sentence of six years. The payoff matrix is thus:

		A	
		Tell	Don't
	Tell	(1, 1)	(3, 0)
B	Don't	(0, 3)	(2, 2)

The best total sum is at (2, 2), where both don't tell. The worst total sum is at (1, 1) where both testify against the other. Suppose that the game is to be played once, and only once. Then, each person will choose the result which gives him the most benefit at that time. It is clear that, no matter what B chooses, A will always do better if he chooses to tell, and visa-versa. Thus the only stable position, the Nash Equilibrium, is at (1, 1), the worst for both.

There are several ways to "resolve" the prisoner's dilemma such that both players will choose (2, 2). Suppose that the burglars manage to steal a very expensive diamond necklace and know that the police are looking for them. They then agree to place the necklace in a safe-deposit box and break the key so that each keeps half. If one of the partners tells on the other, the one who was spited will withhold his half of the key. Neither burglar gets anything. . The matrix then looks something like this:

		A	
		Tell	Don’t
	Tell	(1, 1)	(3, 0)
B	Don’t	(0, 3)	(4, 4)

If A knows that B will tell on him, then A will tell on B and cut his losses. In this case, there is no clear stable equilibrium position. However, if both are reasonable people without any preconceptions, both will consider that, since the game is symmetric, their partner is likely to choose the same position that they are. In this case, both will choose not to tell. Together, these two games form the basis for two-driver psychology.

The latter game is representative of drafting. If two drivers agree to form a draft line, both obtain a large benefit as they are able to pass competitors who are not drafting, an example of how racing is a zero-sum game for the group at large but not when individual decisions are considered. In this case, since the equilibrium at (1, 1) is not very stable in the sense that a driver choosing to cooperate will only cost him a small amount if the other driver chooses not to and the sum of the benefits is the smallest of the four positions. On the other hand, (4, 4) is a very stable position because any deviation by one of the drivers will harm both him and his fellow player and the sum of the benefits is the largest of all four positions. Additionally, the drivers are able to communicate. A group of drivers can agree to draft with each other and kick a driver out of the group if he does not draft with others. In racing today, the result is as expected, with drafting crucial to victory.

The prisoner’s dilemma corresponds to the game where two drivers are in a draft line but each has moves which he can do to block the other driver. The driver in back can attempt to make a slingshot pass around the front driver, while the front driver can spend his energy blocking the back driver from doing so. If the back driver attempts to pass and the front driver attempts to block, then the situation is similar to the (1, 1) position in the first quadrant of the prisoner’s dilemma. Both drivers lose ground as they attempt to

out-maneuver the other and run the risk of crashing. If neither driver attempts to pass or block, then both drivers are able to form a faster draft line and potentially pass rivals (2, 2). If the back driver attempts to pass the front driver, then he gains a position (3, 0). On the other hand, if the front driver attempts to block the back driver, he guarantees his position (0, 3).

The important factor, and the ultimate solution, lies in the Tit-for-Tat algorithm. If two players play a series of prisoner's dilemma games, such as the drivers who will compete against each other in about 30 races per year, then the Tit-for-Tat algorithm has been shown to be the most effective method of obtaining a high score. The premise is simple. Start by cooperating with the other player. Then, for each successive turn, either cooperates or defect based off of what the other player did the last turn. If the other player cooperated, then agree to cooperate in turn while if the other player did not, do the same to him. In the article "Social Science at 190 MPH" by David Ronfeldt, the author says that many drivers espouse the attitude that: "I treat another driver like he treats me. If he races me hard and clean, I do the same to him. If he doesn't, then payback is okay." This is the Tit – for – Tat algorithm put into words.

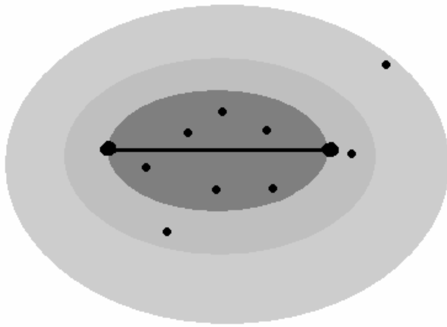
MGG Background

An MGG genetic algorithm, or Minimal Generation Gap algorithm, is a variation on a standard genetic algorithm which has been shown to be significantly superior to the standard genetic algorithm for solving complex functions. The basic procedure for implementing a MGG is a follows:

1: Create a random initial set of organisms.

2: Randomly choose two parent organisms.

3: Draw the line between the two parents and find the central point. Now, randomly create children in the area with a tendency to cluster towards the center.



4: Calculate the fitness levels of the children and parents.

5: Replace the parents with the elite organism and one randomly chosen from the set of children and parents

6: Repeat steps 2 – 5 until either the optimum has been reached or until sufficiently many values have been tested that it is clear that there is no optimum.

Scientific Method

Purpose

The purpose of this investigation is both to determine whether a program for running a simulated car race can be created to be realistic and how such a program would perform when the drivers are run through a genetic algorithm and their fitness rankings are based on how well they far in the simulated car race. The goal is twofold – both to see what happens when the genetic algorithm attempts to solve a problem where the fitness rankings of the organisms are dependant on other organisms and whether the program can be made to produce meaningful data while remaining realistic.

Problem Statement

The problem is whether a realistic simulation of the social interactions between drivers will have a global optimum driver or not. If not, then what pattern will emerge as the drivers are evolved using a genetic algorithm. Other problems include whether a MGG genetic algorithm will give improved performance and whether it is possible to evolve a driver to defeat a particular set of drivers.

Research Plan

I: Develop a basic GA (Genetic Algorithm) and test it with some simple functions to determine its efficiency. Test GA to determine optimum number of organisms to kill each round and optimum number of mutations.

II: Do research to determine what features a realistic racing algorithm would have. At this point, which alleles the drivers are going to use. Program a simple racing algorithm where the cars are all in a line and only game theory matters.

III: Add to the simple program, hopefully adding features such as distance, power, speed, a passing algorithm, and other features. Continue improving the program (or possibly start over and add in the new features). Build a graphical output so that the data

can be analyzed and the realism of the program improved. Use the GA to determine if some traits are always better in one position, regardless of the other drivers. Change or remove those traits.

IV: At this point, it is critical to increase the realism of the race algorithm. If the data is to be valid, it is important that the race program act somewhat realistic. Once the algorithm has been completed, test the hypothesis using the GA. If the result is that one organism is clearly superior, play around with the race-running program to increase the nontransitivity. If one organism continues to be superior, just move to Step V.

V: Write up results and conclusions. Future research includes things such as improving the type of GA, making the driver traits continuous rather than discreet (that is, a driver can have an aggressiveness rating of 5.67). Also, possibly improve the plot to add a mode which takes curves into account.

Conditions

Step IV of the research plan indicates that the race program must be made to be realistic. In the interest of creating a precise definition of "realistic", the following conditions were devised with the intent that they would have to be satisfied before the data gained from using a given race algorithm would be considered valid.

1: The physics must be correct. While this is not a physics-based project, preferring to focus on game theory, it is important that the physics at least be reasonable.

This aspect is not difficult to address. The physics is hard coded into the program and is not done on trial and error. Either the physics is realistic or it isn't. Simply adding basic drag, friction, power, and speed would satisfy this requirement.

2: The methods in the driver algorithms must correspond to actual driver patterns. In particular, drivers must pass in the same fashion that real drivers do and drafting must be handled in a realistic and reasonable fashion.

This is more difficult to do, but still falls in the same class as physics in the sense that it is built into the code and fulfillment of the requirement is not based off of the results of trials.

3: It must resemble actual driver behavior to a reasonable extent. The tracks of the drivers must be fairly close to those formed by real drivers.

This requirement is the most difficult to define and to satisfy. Eventually criteria for defining reasonable behavior were devised. The drivers had to be able, when falling behind the first group of drivers, to be able to regroup, form draft packs, and potentially catch up. In essence, a human who saw the plots of drivers would not be able to point to one and say "I'd do this and this and this if I were that driver."

4: In sample genetic algorithm runs, all alleles must exhibit divergence, but each must have some influence on the driver's performance. More specifically, there cannot be a value V for an allele A such that for any set of other alleles and other drivers, the given driver always performs in a superior fashion when the value of A is V as opposed to any other value. Also, if a given allele is changed, the driver's performance must change in some fashion.

Running a GA for 100 iterations and eyeballing the data is sufficient to determine major pitfalls. If in the list of race winners, one allele almost always has a certain value, then this is an indication that the divergence condition will not be satisfied. Similarly, if in a list of race winners one allele is always the one which shifts from winner to winner than it is likely that this allele is not affecting the final result.

Condition Analysis

1: Completed in the first version of the program. Initially the plan was to have a program where all of the drivers would be assumed to be in one long line and the entire focus would be on passing behaviors, and this would be completely redone and physics would be added, with a final version being primarily physics based. In reality, there was only one program and it incorporated position from the beginning since it was easier to

calculate with position than it was to do a purely game-theoretical version. This initial program was slowly added onto so that it incorporated drag, friction, drafting force, and power with relatively simple models. Traction was assumed to be sufficient so that the car would not slide off the track even at top speed, and the model for drafting was designed to work the best with the game-theoretical aspect rather than physics.

2: Completed in the first version of the program. The passing algorithm was coded before the main function was complete, again because the intention was to do a purely theoretical approach and then gradually shift to a physics-based situation. The passing algorithm is quite complicated to look at and is based on the idea that a pass follows a distinct sequence. First, there have to be cars around which a driver can pass. Secondly, the driver must be interested in passing. Thirdly, the driver must determine which type of pass he is going to attempt. Finally, he will attempt the pass and it will either fail or succeed, with different conditions for each.

3: This was by far the most difficult aspect to achieve, and only the final versions of the program have achieved this. Initially, the drivers would all cluster up at the front, with the better drivers forming a long-lasting draft pack and the other drivers simply falling behind one by one. The other drivers were unable to form draft packs, and would simply disappear off the active screen. There were a few notable exceptions, namely a particular group of organisms which showed the remarkable ability to form draft packs even without functions which promoted such packs. Another exception was when an early genetic algorithm was run and some of the race winners were placed in such a test. They were nearly identical and were able to all stay together at the front.

A function was developed which would alleviate some of these problems. It would calculate when a driver had fallen so significantly behind that his best tactic was to slow down, allow the drivers in back to catch up, and form a draft pack. The function would also calculate when it was in a driver's best interest to not pass other drivers for some time, therefore allowing more stable draft packs to be formed. The falling-behind function took a significant amount of work to design so that it would not cause drivers to lose more ground than they would eventually gain. The non-passing function, on the

other hand, proved immediately useful. Another tactic was to have drivers only attempt passes on other drivers if they were on the same lap as the other driver. Since passing a driver whom they had lapped or who had lapped them would not improve their position ranking, it was better to form a drafting pack than to attempt to pass these drivers. Similarly, other drivers would allow drivers who were not on the same lap to freely pass them rather than attempting to resist it.

When the race was changed to be lap-based rather than a single straight track, more progress was achieved. Drivers still fell behind, but at least after they were lapped, or passed in such a way that they were a full lap behind other drivers, they had a chance to join with the front drafting pack. Even more importantly, drafting packs would emerge behind the front and some of them would manage to get their drivers up to the front and some driver exchange would take place – that is, some of the drivers at the front would fall behind and some of the drivers at the back would pull ahead and stay in the front drafting pack. Draft packs of more than 5 drivers were extremely difficult to keep stable. With drivers passing each other and the corresponding speed boosts and dives, drivers would fall outside of the powerful draft created by the pack and would be unable to keep up, finally falling behind. Even in the packs behind the lead group, drivers would constantly fall behind as new drivers joined. Nevertheless, the first plots from a goodalg_1.java actually looked realistic enough to pass in this section.

4: Goodalg_1.java was the first program where data was taken for analysis, as well as the first program which satisfied condition 3. It was also the first program to resolve some major bug issues, one of which had been ruining one of the alleles. With the bugs removed, the genetic algorithm began to function extremely well, passing the eyeball test for divergence and influence. Interestingly enough, this condition was not particularly difficult to satisfy when compared to number 4. Since the alleles were all built to have some particular effect in the code (and most to have several), it was unlikely that one allele would not have an effect. The alleles were originally chosen so that there were conditions where each would be a boon and a detriment.

Data

There were actually four separate trials run: Simultaneous Evolution with basic GA, Simultaneous Evolution with MGG, optimize to fixed function with basic GA, optimize to fixed function with MGG. Of these four, the first was the most interesting and so is represented here. The data and analysis of the other three trials can be found in the final appendix.

Trial 1

4000 races
127 Elite Drivers with 10 or more consecutive wins
740 Drivers with at least one win
After a driver has one at least one race, he is likely to win 5.405 total races.

These are the drivers with 10 or more consecutive wins. It is interesting to note that they do not seem to follow any orderly pattern. Rather, most of the time each driver appears to be mostly independent of the previous. Given enough trials, there would have to be some driver repeating, but since this data has failed to be periodic so far, it is extremely unlikely that it would become periodic since there are only 11^5 possible drivers and $4 \cdot 10^3$ races have been run, each with 11 drivers.

Agg = Aggression Def = Defense Dec = Deceptive Grd = Grudges Adp = Adaptive

Agg	Def	Dec	Grd	Adp	Win
0	3	0	0	3	had: 13
8	1	4	2	9	had: 21
2	1	1	2	1	had: 13
2	3	3	8	1	had: 11
5	0	7	8	0	had: 10
9	3	8	8	0	had: 13
8	3	8	8	9	had: 12
8	3	9	8	9	had: 13
2	7	2	8	5	had: 14
1	2	5	2	1	had: 10
8	10	5	2	1	had: 11
1	1	4	1	2	had: 16
4	8	3	1	6	had: 13
8	3	0	10	8	had: 12

7	5	0	2	7	had: 10
1	5	5	4	3	had: 11
10	1	10	8	3	had: 10
4	6	4	3	2	had: 12
4	6	4	3	8	had: 15
4	5	1	3	6	had: 12
10	4	2	4	4	had: 11
0	6	1	7	4	had: 24
1	6	1	0	4	had: 20
8	5	1	0	4	had: 12
2	5	1	0	0	had: 10
10	5	9	7	4	had: 12
6	2	2	0	4	had: 10
0	10	1	10	0	had: 10
0	1	1	10	2	had: 24
6	0	7	7	0	had: 13
4	1	9	3	2	had: 16
9	1	10	4	1	had: 10
1	3	1	5	7	had: 19
5	1	10	2	7	had: 18
5	8	3	2	1	had: 11
5	8	7	2	5	had: 13
1	3	8	5	3	had: 10
0	8	8	0	8	had: 13
4	7	8	9	0	had: 11
0	6	5	10	1	had: 11
9	6	5	6	6	had: 10
10	0	1	4	1	had: 12
0	0	0	0	0	had: 17
7	6	0	0	8	had: 18
5	3	6	9	5	had: 16
4	2	5	4	1	had: 19
10	0	8	9	4	had: 15
10	1	8	7	4	had: 14
10	10	7	7	4	had: 13
2	10	6	4	3	had: 17
0	10	6	8	4	had: 10
9	10	6	3	5	had: 14
8	5	0	5	5	had: 14
8	8	6	5	5	had: 12
5	0	0	8	5	had: 10
1	0	1	7	5	had: 16
5	0	1	7	5	had: 10
5	0	1	10	5	had: 15
10	10	2	5	5	had: 12
3	10	3	0	5	had: 17

4	6	4	0	4	had: 11
7	5	2	5	5	had: 13
0	10	7	3	0	had: 21
9	9	4	8	10	had: 12
9	9	4	6	8	had: 16
3	4	5	10	6	had: 12
0	3	10	8	3	had: 15
0	3	6	0	3	had: 11
0	4	4	6	10	had: 10
0	9	3	6	10	had: 15
7	9	8	10	10	had: 10
7	5	1	0	3	had: 17
9	5	10	5	3	had: 16
9	5	10	5	0	had: 10
1	7	9	4	3	had: 21
8	3	0	5	6	had: 15
0	8	5	8	8	had: 17
10	4	6	9	2	had: 10
1	4	7	9	2	had: 12
9	0	1	7	7	had: 18
7	1	4	10	0	had: 16
9	1	7	2	2	had: 12
2	4	10	9	9	had: 10
4	4	2	0	5	had: 15
9	8	0	5	6	had: 12
0	9	10	7	9	had: 12
9	1	8	10	6	had: 10
9	9	8	3	6	had: 27
9	1	1	3	2	had: 15
4	6	4	1	9	had: 12
7	3	9	0	2	had: 12
1	5	9	7	0	had: 14
2	5	7	3	9	had: 16
5	5	1	0	9	had: 10
1	0	4	3	9	had: 11
1	0	7	6	4	had: 10
0	0	7	9	6	had: 12
2	5	6	2	9	had: 13
8	1	0	6	9	had: 18
7	1	5	5	10	had: 17
9	2	3	2	5	had: 14
2	3	1	4	2	had: 13
6	9	0	0	9	had: 11
7	10	4	6	1	had: 13
7	9	5	6	3	had: 13
4	2	0	0	9	had: 10

4	2	4	3	9	had: 15
4	6	4	7	9	had: 15
4	6	4	10	9	had: 17
9	6	5	10	9	had: 12
9	6	1	1	10	had: 12
4	6	5	8	10	had: 12
7	6	10	8	8	had: 10
3	10	0	6	4	had: 11
1	7	4	3	0	had: 10
1	4	9	0	8	had: 11
4	8	2	5	3	had: 11
9	8	2	9	0	had: 13
9	10	5	4	0	had: 10
9	6	5	7	0	had: 21
6	6	5	7	0	had: 14
7	6	5	8	0	had: 14
3	0	10	0	5	had: 13
3	9	0	2	7	had: 14
2	9	0	3	8	had: 10
2	9	0	9	8	had: 29
9	9	1	7	2	had: 14

Test 2

3000 races

84 Elite Drivers with 10 or more consecutive wins

580 Drivers with at least one win

After a driver has won one race, he is likely to win 5.1724 races total.

A similar pattern occurs here as in Test 1. Again, there is no clear pattern.

Agg = Aggression Def = Defense Dec = Deceptive Grd = Grudges Adp = Adaptive

Agg	Def	Dec	Grd	Adp	Win
0	3	7	0	9	had: 10
5	3	3	2	3	had: 13
5	2	3	9	3	had: 13
10	3	3	9	9	had: 10
0	5	2	5	5	had: 15
10	7	5	4	8	had: 12
4	10	9	6	2	had: 10

1	2	3	4	3	had: 12
6	6	1	7	8	had: 15
6	6	1	7	0	had: 13
6	6	4	4	0	had: 10
8	2	2	7	0	had: 10
1	2	2	9	0	had: 11
8	10	9	9	9	had: 12
3	6	1	5	2	had: 10
4	8	4	4	8	had: 10
4	7	2	9	1	had: 15
5	4	3	2	5	had: 16
5	4	9	0	10	had: 17
6	5	0	9	0	had: 20
3	4	0	4	3	had: 13
4	1	3	5	3	had: 10
4	1	7	5	3	had: 22
7	2	7	8	3	had: 12
7	2	7	5	3	had: 10
8	7	9	7	2	had: 24
1	10	2	2	5	had: 10
9	10	6	1	2	had: 16
4	10	1	2	3	had: 10
6	0	7	2	3	had: 18
6	4	0	2	7	had: 10
4	6	2	3	3	had: 11
6	9	7	0	1	had: 14
6	9	2	8	1	had: 20
1	4	7	4	8	had: 10
3	4	7	7	8	had: 21
6	4	7	7	8	had: 12
0	6	4	2	8	had: 13
6	6	4	8	7	had: 13
4	6	8	3	1	had: 25
9	1	6	0	1	had: 10
9	1	1	0	1	had: 15
1	2	1	8	8	had: 11
7	2	2	5	4	had: 13
7	6	4	7	0	had: 11
7	6	4	5	1	had: 10
7	6	4	5	1	had: 15
1	0	7	6	7	had: 11
1	0	0	1	2	had: 10
0	7	2	6	9	had: 12
4	7	4	6	8	had: 12
7	8	6	8	5	had: 16
4	3	6	5	5	had: 11

1	10	7	10	9	had: 18
3	0	7	10	2	had: 10
6	1	7	10	9	had: 23
6	1	7	10	3	had: 11
10	1	3	1	8	had: 10
10	1	2	1	7	had: 11
10	1	0	6	6	had: 17
0	0	6	6	8	had: 10
0	8	1	6	2	had: 11
0	2	3	10	10	had: 12
0	3	0	5	10	had: 11
10	8	4	1	3	had: 10
4	8	1	8	5	had: 13
4	10	1	1	3	had: 11
0	3	9	9	1	had: 10
3	8	5	1	9	had: 10
3	6	2	8	4	had: 12
4	6	10	4	7	had: 12
2	2	7	7	6	had: 11
6	5	7	5	6	had: 13
5	6	6	9	6	had: 16
6	4	6	9	6	had: 16
5	6	9	0	10	had: 10
2	5	2	1	10	had: 22
4	5	4	2	8	had: 18
6	7	6	3	1	had: 12
0	7	6	3	3	had: 13
10	6	8	2	10	had: 14
10	9	4	4	5	had: 10
1	10	9	4	6	had: 11
1	2	9	4	2	had: 10

Data Analysis

When the race simulation program which satisfied the realism criteria outlined in the Conditions section was used and the winners of each race analyzed, it was found that they were not periodic as predicted by the hypothesis but neither converged to a global optimum nor had a consistent stable set of drivers. Instead, the drivers seemed to be a relatively fluid set, not periodic in the sense that they repeated regularly but flowing from one optimum to another optimum superior to the previous and eventually looping back around but then going off on another direction entirely.

The genetic algorithm was tested to determine the optimum set of parameters. For the final test, the number of organisms killed was set to one out of eleven total organisms and the number of mutations was set to 3, a setting designed to increase total diversity and decrease the problem of early convergence to a local optimum.

There was the possibility that the winning driver was based more on randomness than superiority over the other drivers. To resolve this, a small program was created to analyze the raw data and it was found that the number of times the winning driver switched was 579. Since the initial positions were randomized for each new race, it would be extremely improbable that the previous winning driver gained an advantage based off of position. The number of iterations was 3,000: on average each driver who won at least one race won 5.1724 total, and since there were 11 drivers in each race, the probability that this was produced by pure chance is smaller than the probability of winning the lottery twice in a row.

When the values of the alleles elite drivers from both Test 1 and Test 2 were averaged, the following values were obtained:

Aggression: 4.828571428571428
Defense: 4.838095238095238
Deception: 4.404761904761905
Grudges: 4.9523809523809526
Adaptive: 4.733333333333333

These averages seem to indicate that the various alleles are well-balanced in the sense that no allele is always in one position, indicating that different drivers are better in different situations. Since these are the organisms which won 10 or more races, it can be deduced that these values are not products of randomization but that the organisms are actually responding to the other organisms. Additionally, eyeballing the data reveals that all of the alleles expressed 10's and 0's at various points, further indication that different values of alleles are superior for different sets of organisms.

Overall, the data shows a tremendously successful run in that the data produced was not corrupted by one of the common pitfalls of analysis.

Genetic Algorithm Data

This is data used to determine the ideal set of genetic algorithm parameters. The parameters of the test case were:

10 Alleles with range 0 through 10

10 Organisms and 1 Elite organism.

Fitness function = 100 - Σ(alleles); in this case, fitness was defined so that lower fitness values were preferable.

When the genetic algorithm produced a perfect organism (all alleles equal to 10), a break command was sent and the number of iterations used was printed. A program was devised that would automatically vary the number of organisms mutated and the number of organisms killed every round, run the genetic algorithm thousands of times, and average the number of tests. The following data was produced:

	Mutations			
	1	2	3	4
Number Killed				
1	1073.2978	1863.789	3055.948	4238.428
2	573.662	547.738	685.302	886.81
3	465.691	350.609	349.976	384.803
4	408.872	274.507	251.487	257.918

Additional runs (not shown here) produced extremely similar results, varying by about 1-2%.

The following data was produced in order to find the optimum number of mutations for given number of organisms killed.

For 1 Killed	1 Mutated
For 2 Killed	2 Mutated
For 3 Killed	2 Mutated (close between 2 and 3)
For 4 Killed	3 Mutated
For 5 Killed	4 Mutated (close between 4 and 5)

For 6 Killed	5 and 6 Mutated performed the same
For 7 Killed	7 Mutated

MGG Genetic Data

The MGG algorithm, or Minimal Generation Gap, is an alternate type of genetic algorithm which has shown to be significantly more effective in avoiding early convergence and late stagnation. While in a basic genetic algorithm varying the number of mutations and number of organisms killed per round can make a tremendous difference, in a MGG the parameters are less subtle. The only two parameters which can be varied and are significant are how close the organisms are to the center and the ratio of the orthogonal component to the component along the line between the two parents. Tightness determines how close the organisms will be to the center of the two parents, while the orthogonal component ratio determines how likely the newly generated organisms are to lie close to the line between the two parents.

Since the MGG algorithm used doubles rather than integers, a meaningful comparison of the two algorithms would be subject to error. Others have already proved that the MGG is tremendously faster in general, and there is no need to repeat their results. The data for the MGG is as follows:

	Orthogonality Ratio			
Tightness	.05	.1	.2	.4
1	49.0426	50.0118	54.7621	57.0818
2	33.9588	35.6124	37.2857	38.6945
3	33.3151	34.0768	35.3581	36.3172
4	32.9028	33.6132	34.8577	35.8018

The same fitness function as before was used, with two major differences. There were only five organisms instead of ten, and values were fixed rather than expanding

For the MGG, tightness was set to 3 and the orthogonality ratio to .1. .05 would make going outside a single line very difficult and a tightness of 4 would all-but preclude values outside of the range of the original parents.

Conclusions

1: The set of race-winning drivers generated by using a genetic algorithm and the race simulation was neither of the three expected classes. It was neither periodic nor dominated by a single driver nor is it a stable set. Instead, the winning drivers appear to shift from optimum to optimum, where each successive optimum is replaced by one which is more effective against the current set of drivers.

2: The race simulation is highly nontransitive, with many cases where driver A beats driver B, driver B beats driver C, and driver C beats driver A. The driver traits were originally chosen at least in part for their nontransitivity, so this is evidence that this attempt succeeded.

3: For the race simulation, the alleles appear to be mostly balanced, as determined by taking the average of the drivers which were able to win 10 or more consecutive runs. It is not obvious whether having high or low values for any particular allele / trait makes a driver superior or not.

4: Making the drivers behave like real drivers is extremely difficult. The passing method can be physically accurate, the traits can be realistically modeled and the physics can be valid but it is still a challenge to make the digital driver behave in the same way that human drivers would without special algorithms. In other words, simply laying down the physical rules, defining traits, and running a genetic algorithm does not necessarily cause the drivers to evolve human-like behaviors. The only consistent way of making the drivers perform similarly to human drivers is to add functions which check the current state of the driver and return a command for each driver on what to do the next round.

Discussion

Plot Classes

One main goal of the program has been to find a race-running algorithm where the drivers will behave as though they were on an actual racetrack. To this end, several algorithms have been devised, with each successive generation looking more realistic. Each of these, when run to produce a visual output, has produced a unique type of plot. Some of the classifications are as follows.

1: Christmas Tree

Nearly all of the drivers will cluster at the front, occasionally passing each other and creating a powerful draft. When one driver falls too far behind and the draft is not enough to keep him with the back, he falls to the back very quickly.

2: Patchwork Snake

The drivers will all begin clustered near the front. A few will fall behind and will immediately group up. When enough fall behind, they realize that their best chance of obtaining the lead is to form a draft line with those in back. Once there are only a few drivers left in front, the draft power of the "snake" is so great that it very quickly overtakes the drivers in front. The cycle repeats itself

3: Small Patchwork Snake

The drivers are very competitive at the front, and so are unable to produce a powerful draft. When a few drivers fall behind and activate the cooperative mechanism, they quickly form a snake which rarely goes off the screen.

4: Solid Snake

By all appearances, most of the drivers had fallen very far behind and the screen only shows the very front driver (and perhaps another). Suddenly, a whole group of the drivers in the back appear on the screen and overtake the first driver. Apparently, they

had regrouped and managed to stay in formation for thousands of iterations, long enough to overtake the first driver.

5: Triple Domination

Three drivers form a draft pack at the front, and the other drivers are unable to organize and so slowly but inevitably fall behind.

6: Double Domination

Two very strong drivers are able to remain in the front in a very tight draft group, dominating the remainder of the race. This is what happened when the race algorithm that produced Triple Domination was used with a genetic algorithm to select for the optimum set of drivers, then an evolved set of drivers was tested.

7: Ideal Plot

All drivers are potentially able to catch up to first. There is a draft group in the front, and another one or two in the back which are trying to catch up to it. Some of the newer algorithms, run with evolved sets of drivers, are able to achieve this goal.

8: Pseudo-Ideal

This is a rather unusual situation which occurred with an early build, using a randomly generated initial set of drivers. This set was later stored in the "uberorgs" data file. The drivers appeared, at first, to have achieved the Ideal Plot. Some drivers would fall behind, and the others would catch up to them and their combined draft would bring them back up to the front. In the end, it was determined that this was not so much a result of the drivers being good, but a result of the first few drivers' inability to form a draft pack, allowing the back drivers to do so. When a new random set of organisms was loaded in, the race followed the more familiar Triple Domination pattern.

At this point, the program was switched from a single, straight course to a looped course. This eventually led to a more realistic plot but, since the strategy function was designed for a straight course, initially caused some amusing errors.

9: Jumping Spider

The cooperative function, which told drivers to fall behind and form a draft pack with the previous driver if they were unable to get anywhere, was causing the tracks of the drivers to look like a spider web. This was more a mistake than anything else, but was very interesting to watch.

10: Rain

Stemming from the same problem as Jumping Spider, this plot occurred when some code was added in. Instead the drivers with no "safety net" of a loser to fall back on, found themselves slowing down so that they could loop around the track and get back to the front. The end result looked rather like rain on a very windy day.

11: Ideal Loop Plot

Success! When the driver strategy function was modified so that it did not produce a "2" response (driver falls back to the nearest companion), the function worked much better. It started to resemble an actual race, with drivers forming drafting groups without being commanded by the program to do so, catching up to the front, having intense competition, falling behind, forming more groups, and so on. Considering that the race in question is significantly longer than any race involving humans, it is perfectly reasonable that some drivers would be lapped several times. Their behavior is positively beautiful to watch.

Code Explanation

The following are the comments which were placed in the code as it was being written for personal reference. They give a fairly good explanation of what the code is doing.

Driver Properties, in order of number in array they appear in

0: Current Position

1: Aggressiveness - *10 = % chance to attempt pass.

2: Defense - *10 = % chance to attempt to play it safe, but each time used costs 1 power.

3: Deception - *10 = % chance too make a deceptive pass.

4: Grudges = Property that makes the drivers more likely to attack other drivers who have betrayed them. There is some feedback but not enough to self-perpetuate. If the optimum for grudge is 0, then feedback will be removed or some bonus will be added.

5: Adaptive - How much a driver forgoes his innate programming and acts according to the programming of other drivers. Note that reactivity is also non-feedback - that is, drivers do not take the other's reactivity into account when making moves. Otherwise the problem of "I know that you know that I know what you're thinking," pops up.

6: Damage - This makes the cars slower. Since this is designed to represent a road rally, damage cannot be repaired. The model is linear since I don't know for sure how to calculate progressive damage.

7: Last Round's Speed

8: Current Speed

9: Making defensive maneuvers

10: Passing lane or not

11: Formerly known as todo, tells the drivers what to do at that time

12: Formerly known as anger, how much the driver has fallen behind.

Top or normal speed is about 100 R3U / 3 Sec. Power is 300 R3U. Mass is 10 R3U. Resistance is speed*1 R3U + Speed^2 / 50. You can take damage up to 1000 points. As of now, there is no way of preventing drivers who are nearly destroyed from passing or doing things like that to gain position, but the odds are so much against them that there is no way they will win. Each unit of damage decreases power by .001 R3U, which seems like very little except for the fact that the races are very long and a car whose power is lowered by 10,000 R3U (not uncommon) will inexorably fall behind.

Drafting

Drafting force is inversely proportional to 3 + distance away from back of car (the 3 because otherwise there would be a problem at the limiting case). The cars in front of you gains a bonus equal to yours, presuming that you are at most 10 R3U away. Else, you are simply "riding his air" but not contributing to him at all.
exactly. If you are part of a pass, you gain an instant 5 R3U speed bonus.

The cars are 2 R3U long, and the reference point is at the center of the car.

Time units are about one second.
Calculation order is:

1: Run special moves, such as passes
2: Find new speed
3: Calculate new position as: (oldspeed + newspeed)/2 - 3*lane changes

The track does not consist of lanes - that is, all the cars are assumed to be in one (or two) lanes

A car makes defensive maneuvers no matter what, depending on defensive. This corresponds to looking in your mirrors to see if someone is trying to pass or attack you, taking your focus off the race.

Driversort Function

This function will sort the drivers so that the drivers[0][0] is the largest and drivers[10][0] is the smallest. To do this, it actually swaps the values for the drivers but keeps track of which driver is where in the array "driverorder". Position 0 of driverarray corresponds tells which driver is in first and so on. This function also switches the grudges around so that they correspond to the new positions of the drivers (that is, if the driver in first has a grudge against the driver in third, and the driver in third passes the driver in second, the driver in first will now have a grudge against the driver in second.) Actually switching the drivers around saves computation when checking the passes

Runrace Function

This is my function for actually running the race. In this case, the main program doesn't do that much - this one does all the work. A flowchart for what it does is available in my databook (a diary / lab notebook which will be available at the final presentation).

If you gain enough speed to cleanly pass a guy in this phase, you make an "instantaneous pass". This is when you don't even need drafting and just scoot cleanly by him.

Points system gives 0 points for leading, 2 points for second, on to 20 points for last in the lap. Plus, you get points equal to: top position - your position at the end of the race. This is a modified version of the NASCAR system, but designed to convert directly into fitness rankings (the reason for flipping the rankings is that now there is a huge proportionality difference between the top guy and number 2, whereas otherwise there would not be so much of a difference. If you are selecting an organism to kill randomly, you want to make sure that you pick the top guys rarely, but differentiating between the weaker ones does not matter as much.

Passing Function

Desire - 10*aggression = % chance to attempt any pass

Position - Is car part of 3+ car line & not in front? Yes = good, No = forget it.

If 2 passes are available, at this point calculate them both.

Pass fore = 1.5*all. Asking someone to pass when one is in back is good, because he has less chance of being tricked. All of the drivers (like real drivers) instinctively know this, no matter what stats they have. Driver's adaptive*(1 - his defense*.05) - his grudge against passing partner - his adaptive*passing partner's grudge against him is an equation used to determine whether that driver will try to pass or not. Pass acceptance for falsepass is: deception if guy is in front of you, deception/10 if guy is behind you. Pass acceptance for truepass is: aggression - grudge against other guy

If a driver is making a pass to deceive, then he must be the last guy in a chain of 3.
He must - Have failed his aggression check
Passed a deception check
Have the guy in front of him accept the offer

Once a driver makes a pass, he moves into a temporary lane. Once the driver moves out, he gains an instant 5 R3U speed bonus due to the "slingshot effect." (yes, there is such a thing and it does give a temporary speed boost) The max speed of all other drivers is recalculated. If he is moving out with another driver who is immediately behind him, the other driver does not gain the speed boost but will be sucked up into his draft. The guy whom they are passing "gains" a 5 R3U speed downgrade due to air vortices sucking him back (yes, this happens as well.)

If a driver wants to make a 2-person slingshot pass, then there must not be anyone else within 10 R3U of either his car of the cars he is trying to pass, because that driver would take advantage of the pass and manage to abort it.

These drivers are gentlemen. If a driver is already involved in a pass, they won't try to take advantage of them (until the next turn, at least). This not only makes them look good, it lowers the processing load.

Strategy Function

A signal of 0 means to do nothing. A signal of 1 means to not try to pass and instead just focus on catching up. A signal of 2 means to slow down and collect in packs.

Calculating signals: If your anger (position lost relative to the top drivers) is at least 25, you will likely do something)

If the car has either a car in front with dist <= 10 or a car in back with dist <= 10 and decreasing, then return a 1. 1 goes to racerun then to passcheck and the car is considered to already have made a pass. This organized formation will disintegrate when

the lead car is within 10 units of the front. After all, what advantages do the guys in the front gain from being nice to each other?

2 means to group. If the car is closer to somebody in back then in front, then stop. Else keep going.

Future Developments

These are possible ideas for future development of the project, ranked from least to most difficult to implement.

1: Add to the strategy function so that it returns more possible values and causes the program to behave more reasonable. Since the capacity for true human-like behavior does not seem to lie within the limits of the five traits, it is only reasonable that the drivers can be given a helping hand by the strategy function which analyzes each driver's current position and delivers a recommendation for what that driver should do.

2: Run the genetic algorithm on its parameters to find which values for the various parameters are most efficient at solving various optimization problems. Rather than having to choose parameters such as number of mutations and number of organisms killed by eyeballing performance data and choosing values which seem reasonable, running those values through a genetic algorithm would provide more certainty of using the absolute correct parameters.

3: Continue improving the MGG to see if it can be made to produce more valuable results, or whether an MGG is by nature doomed to be less effective than a standard GA for optimization problems without a fixed fitness function.

4: Potentially replace the current strategy function with an neural network which would be given the current positions and speeds of the cars and would determine what the car should do. It would be trained by having either the researcher or a real racecar driver critique its decisions and thus generate a set of sample data to train it on.

5: Create a program which would tweak the race-running algorithm by modifying various parameters and then run a genetic algorithm. Finally, it would analyze that data and determine whether it was periodic, freely-flowing, global optimum, or mixed group of drivers. The advantages would be that this program would save time by analyzing the data automatically, but would also provide a rigorous definition for the classes of possible drivers. The disadvantage would be that it would either require a supercomputer, distributed network, or a tremendous amount of time in which to run.

6: Design a new race-running program which, rather than running on a super-speedway, would be executed on a road rally type course. The drivers would, rather than being defined by traits, would be neural networks. This would also take up large amounts of processing power but such a system could be adapted to compete in competitions such as Robotic Auto Racing Simulator, or RARS.

Main Accomplishment

A question which occurs when using a genetic algorithm to optimize a function is what would happen if fitness depended on the other organisms rather than a fixed function. To the best of my knowledge, this has not been tested before. A race provides a natural ground for testing such an idea, since performance is dependant on the other drivers and there is a clear fitness function. This has been the focus of the project. Other experiments showed that using a classical GA selection could find drivers optimized to beat a particular set, which has a practical application for the professional looking for the extra edge against his competitors.

The tests which I ran showed that using competitor-dependant fitness functions was not only feasible, it could produce meaningful results. Also, an MGG type genetic algorithm did not do very well with such a fitness function, while the basic GA performed superbly.

The whole project was made much more interesting because the racing simulation was highly nontransitive. While this precluded finding a true optimum driver or even a small set of optimum drivers, it did help to explain why successful drivers today can have such a variety of different driving styles.

References

David Rondfeldt. Social Science at 190 m.p.h.
http://www.firstmonday.dk/issues/issue5_2/ronfeldt/, January 26, 2000

Marshall Brian. How Champ Cars Work.
http://entertainment.howstuffworks.com/champ-car.htm.

Brian Beckman. The Physics of Racing. http://phors.locost7.info/contents.htm.

Osamu Takahashi, Hajime Kita, and Shigenobu Kobayashi. A Distance Dependant Alteration Model on Real-coded Genetic Algorithms
http://www.fe.dis.titech.ac.jp/~osamu/paper/SMC1999pp619.pdf, 1999.

Turner Sports Interactive, How the NASCAR Point System Works
http://www.nascar.com/2004/news/headlines/cup/01/30/points_system/index.html
January 30, 2004.

Acknowledgements

I would like to first thank Mr. Greenberg, my Java teacher and mentor. He has acted as an indefatigable booster for the class' science fair projects, giving us deadlines to keep us on track and making sure that we have the supplies that we need. When I had difficult Java programming questions, he was willing to spend his own time outside of class to find the answers. So, Mr. Greenberg, thank you for all of your support.

I would next like to thank my mother for putting up with my project and volunteering to go to Hobby Lobby just to get a particular color of construction paper. She has also been willing to support me and drive me and my project wherever I need to go.

Next, I would like to thank my father for first sparking my interest in computer programming in general. Ever since he showed me how to write a "Hello World" program on a pre-Pentium machine, he has always been willing to support me in my programming and science endeavors.

Finally, I would like to thank my sisters for putting up with me and my piles of construction paper, computer printouts, and general mess while I was working.

Code Section

Raceplot

This is the program raceplot, used to run and output the progress of the race. A version of this program with the output commands removed is used to run the race. Some of the comments have been edited slightly from the original program (so that they are either clearer or so that they will fit on one line rather than most of one line and a sliver of the second).

```
import java.util.*;
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;

public class raceplot extends Canvas
  {
  public static void main(String[] args)
    {
////////////////////////////////////////Graphics Section////////////////////////////////////////
    JFrame f = new JFrame("SF Digital Driver 1000 Open");        // Make a window on the screen.
    f.addWindowListener(new WindowAdapter()             // Have the program end if the user
       {                                                // clicks the close box in the
        public void windowClosing(WindowEvent e)        // window.
           {
           System.exit(0);
           }
        });
     f.setSize(1000,500);                              // Set window size in pixels.
     f.getContentPane().add(new raceplot());           // Make an object of your class.
     f.setVisible(true);                               // Make the window visible.
     }                                                 // End of main method
  public void paint(Graphics g)
        {
   Graphics2D g2 = (Graphics2D) g;                     // Make a 2D Graphics context.

      g2.setColor(Color.black);                        // Set drawing color to black.
      g2.setStroke(new BasicStroke(1.0f));             // Set line width to 0.5.

//////////////////////////////////////////Main Section////////////////////////////////////////

        double [][] drivers = new double [11][13];     //Everything is stored in this array
        int [][] grudges = new int [11][11];           //Grudge of driver x against y = grudges[x][y]
        int [] driverorder = new int [11];             //Keeps track of which driver really is in first.
        double [] driverpoints = new double[11];       //A modified version of NASCAR scoring
```

```
        Random rn = new Random();
        //If each function declares a new random every time it's called, it's not very random at all.
        double rnd = 0;                                   //A variable to store the random number
        int temp, t2;                                     //Temporary variables

        int nd = 11;

        CSPrintWriter genout = new CSPrintWriter("userresults.txt");   //This uses different files than theoretical
        CSInputReader driverin = new CSInputReader("usertest.txt");    //just to avoid confusion

        for(temp=0;temp<11;temp++)
                {
                for(t2=1;t2<6;t2++)
                        drivers[temp][t2] = driverin.readDouble(); //Reads in the drivers.
//              System.out.println("It is reading line "+temp);   //Debug line
                }
//      System.out.println("I\'m outta\' th\' loop!!!");//Debug line

      for(temp=0;temp<11;temp++)
           {
           driverorder[temp] = temp;
                for(t2=6;t2<13;t2++)
                        drivers[temp][t2] = 0;
                }

        initialize(drivers, rn, nd);

        sortdrivers(drivers, driverorder, grudges, nd);

        runrace(drivers, driverorder, grudges, rn, driverpoints, g2, nd);

//Debug code
/*      for(temp=0;temp<11;temp++)
           System.out.println(driverorder[temp]);

        for(temp=0;temp<11;temp++)
                System.out.println(drivers[temp][0]);

        for(temp=0;temp<11;temp++)
                System.out.println(drivers[temp][6]);
*/
        for(temp=0;temp<11;temp++)
                {
//              System.out.println("Driver "+temp+" obtained "+driverpoints[temp]);
                genout.println(""+driverpoints[temp]);
                }
```

/* Keeping this in the program file provides a handy reference while I'm writing the code
Properties for the abstract simulator. * = property read in from file.

0: Current Position
* 1: Aggressiveness - *10 = % chance to attempt pass.
* 2: Defense - *10 = % chance to attempt to play it safe, but each time used costs .1 probability of losing some position.
* 3: Deception - *10 = % chance to back out of passing agreement.
* 4: Grudges = Property that makes the drivers more likely to attack other drivers who have betrayed them.
there is some feedback but not enough to self-perpetuate. If the optimum for grudge is 0, then feedback
will be removed or some bonus will be added.
* 5: Adaptive - How much a driver forgoes his innate programming and acts according to the programming of other drivers. Note that reactivity is also non-feedback - that is, drivers do not take the other's reactivity into account when making moves. Otherwise the problem of "I know that you know that I know what you're thinking," pops up.
6: Damage - This makes the cars slower. Since this is designed to represent a road rally, damage cannot be repaired. The model is linear since I don't know for sure how to calculate progressive damage.
7: Last Round's Speed
8: Current Speed
9: Making defensive maneuvers
10: Passing lane or not
11: Formerly known as todo, tells the drivers what to do at that time
12: Formerly known as anger, how much the driver has fallen behind.

Top or normal speed is about 100 R3U / 3 Sec. Power is 300 R3U. Mass is 10 R3U. Resistance is speed*1 R3U + Speed^2 / 50. You can take damage up to 1000 points. As of now, there is no way of preventing drivers who are nearly destroyed from passing or doing things like that to gain position, but the odds are so much against them that there is no way they will win. Each unit of damage decreases power by .3 R3U.

Drafting
Drafting force is inversely proportional to 3 + distance away from back of car (the 3 is because there is no zone of infinite draft). The car immediately in front of you gains a bonus equal to yours, presuming that you are at most 1 R3U away. Else, you are simply "riding his air" but not contributing to him at all.
exactly. A car moving to pass in front of you creates a vacuum which gives an instant 5 R3U bonus.

The cars are 2 R3U long. The reference point is at the center of the car.

Time units are 3 seconds. Calculation order is:
Run special moves, such as passes
Find new speed
Calculate new position as: (oldspeed + newspeed)/2 - 3*lane changes

The track does not consist of lanes - that is, all the cars are assumed to be in one (or two) lanes

You make defensive maneuvers NO MATTER WHAT, depending on defensive. This corresponds to looking in your mirrors to see if someone is trying to pass or attack you, taking your focus off the race.

```
*/
        }

////////////////////////////////////////////Initialization Function///////////////////////////////////////////
/*This function randomly initializes the drivers*/
   public static void initialize(double [][] drivers, Random rn, int nd)
        {
        int temp, t2;
        double drivercards[] = new double[11];
        double rnd;
        for(temp=0;temp<nd;temp++)
                {
                rnd = (double) rn.nextFloat();
                drivercards[temp] = rnd;

                drivers[temp][0] = 0;
                }
        for(temp=0;temp<nd;temp++)
                {
                for(t2=0;t2<temp;t2++)
                        {
                        if(drivercards[t2]>drivercards[temp])
                                drivers[t2][0] += 2;
                        else
                                drivers[temp][0] += 2;
                        }
                }
        }
/////////////////////////////////////////////////Probability Check///////////////////////////////////////////
/* A small function which I though would be useful but decided not to use.  I'm keeping it because I may want to use it
to make the reading of my code clearer. */

   public static boolean check(Random rn, double prb)
        {
        double rnd = (double) rn.nextFloat();
        if(rnd < prb)
                return true;
                else
                return false;
        }
/////////////////////////////////////////////////Driver Sorting//////////////////////////////////////
/*This function will sort the drivers so that the drivers[0][0] is the largest and drivers[10][0] is the smallest.  To
do this, it actually swaps the values for the drivers but keeps track of which driver is where in the array
"driverorder".  Position 0 of driverarray corresponds tells which driver is in first and so on.  This function also
switches the grudges around so that they correspond to the new positions of the drivers (that is, if the driver in first
has a grudge against the driver in third, and the driver in third passes the driver in second, the driver in first will
```

now have a grudge against the driver in second.) Actually switching the drivers around saves computation when checking
the passes*/

```
    public static void sortdrivers(double [][] drivers, int [] driverorder, int [][] grudges, int nd)
            {
            int temp, t2, t3;
            int [] trueorder = new int [11];
            int [] temporder = new int [11];
            int [][] truegrudges = new int[11][11];
            double [][] tempdrivers = new double[11][13];

            for(temp=0;temp<nd;temp++)
                    temporder[temp] = 0;

            for(temp=1;temp<nd;temp++)
                    {
                    for(t2=0;t2<temp;t2++)
                            {
                            if(drivers[temp][0] > drivers[t2][0])
                                    temporder[t2] += 1;
                            else if(drivers[temp][0] == drivers[t2][0])
                                    temporder[t2] += 1;
                            else
                                    temporder[temp] += 1;
                            }
                    }

            for(temp=0;temp<nd;temp++)
                    trueorder[temporder[temp]] = driverorder[temp];

            for(temp=0;temp<11;temp++)
                    driverorder[temp] = trueorder[temp];  //trueorder is right

            for(temp=0;temp<nd;temp++)
                    {
                    for(t2=0;t2<nd;t2++)
                    truegrudges[temp][t2] = grudges[driverorder[temp]][driverorder[t2]];
                    }

            for(temp=0;temp<nd;temp++)
                    {
                    for(t2=0;t2<11;t2++)
                            grudges[temp][t2] = truegrudges[temp][t2];
                    }
            for(temp=0;temp<nd;temp++)
```

```
			{
			for(t2=0;t2<13;t2++)
				tempdrivers[temporder[temp]][t2] = drivers[temp][t2];
			}
		for(temp=0;temp<nd;temp++)
			{
			for(t2=0;t2<13;t2++)
				drivers[temp][t2] = tempdrivers[temp][t2];
			}

		}
////////////////////////////////////////////////Driver Swapping Method//////////////////////////////////////
/*Another function which I once used but then realized that it would be more efficient to just swap the drivers directly
in the "sortdrivers" function rather than calling this one.  I'm keeping it because I may try to write a more efficient
driver sorting program (that seems to be the rate-limiting factor in my program) which would take advantage of the fact
that the drivers keep some semblance of order and use the swaps.*/

	public static void driverswap(double [][] drivers, int first, int second)
		{
		double [] temparr = new double[11];
		int temp;
		for(temp=0;temp<13;temp++)
			{
			temparr[temp] = drivers[first][temp];
			drivers[first][temp] = drivers[second][temp];
			drivers[second][temp] = drivers[first][temp];
			}
		}

////////////////////////////////////////////////Race Running Function///////////////////////////////////////////

/*This is my function for actually running the race.  In this case, the main program doesn't do that much - this one
does all the work.  A flowchart for what it does is available in my databook.*/

/*If you gain enough speed to cleanly pass a guy in this phase, you make an "instantaneous pass".  This is when you
don't even need drafting and just scoot cleanly by him.  */

	public static void runrace(double [][] drivers, int [] driverorder, int [][] grudges, Random rn, double [] driverpoints,
Graphics2D g2, int nd)
		{
		int counter = 0;
		int temp, t2, unblocked = 1;
		double [] power = new double[nd];
		boolean [] roundover = new boolean[nd];
		double rnd;
		double first, p1, p2;
```

```
	for(temp=0;temp<nd;temp++)
		{
		drivers[temp][11] = 0;
		drivers[temp][12] = 0;
		}

		CSInputReader input = new CSInputReader();

	Rectangle2D e = new Rectangle2D.Double(0, 0, 1000, 500);

	for(temp=0;temp<nd;temp++)
		driverpoints[temp] = 0;

	while(drivers[0][0]<10000000)
		{
	   counter++;

		if(counter%500==0)
			{
			g2.setColor(Color.gray);
			g2.draw(e);
			g2.fill(e);
			g2.setColor(Color.black);
			}

		for(temp=0;temp<nd;temp++)
		{
			drivers[temp][7] = drivers[temp][8];
			}

		strategery(drivers, grudges, rn, nd);

		grudgecheck(drivers, grudges, rn, nd);

		for(temp=0;temp<nd;temp++)
			{
//			power[temp] = 300 - drivers[temp][7] - drivers[temp][7]*drivers[temp][7]/50 -
drivers[temp][6]*.01;
			power[temp] = 300 - drivers[temp][7] - drivers[temp][7]*drivers[temp][7]/50;
			}
		//Allows me to quickly switch between allowing and disallowing damage

		for(temp=0;temp<nd;temp++)
			{
			for(t2=0;t2<nd;t2++)
```

```
				{
				p1 = (drivers[temp][0] - drivers[t2][0])%1000;
				p1 = (p1 + 1000)%1000;
				p2 = 1000 - p1;

				if((drivers[temp][10] + drivers[t2][10])%2 == 0 && p1 <= 6)
					{
					power[temp] += 6/(p1 + 3);
					power[t2] += 12/(p1+3);
					}
				else if(p1<=100)
					power[t2] += 12/(p1 + 3);

				if((drivers[temp][10] + drivers[t2][10])%2 == 0 && p2 <= 6)
					{
					power[t2] += 6/(p2 + 3);
					power[temp] += 12/(p2+3);
					}
				else if(p2<=100)
					power[temp] += 12/(p1 + 3);

				}
			}

		for(temp=0;temp<nd;temp++)
			{
			drivers[temp][8] = drivers[temp][7] + power[temp]/10;
			drivers[temp][9] = 0;
			}

		for(temp=0;temp<nd;temp++)
			{
			if(drivers[temp][11]==1)
				{
				roundover[temp] = true;
//				drivers[temp][11] = -1;  //This line is an attempt to solve the problem.  This way I can
				}				//switch it on and off at will.
			else if(drivers[temp][11]==2 && temp != nd-1)
				{
				p1 = (drivers[temp][0] - drivers[0][0] + 100000)%1000;
				for(t2=0;t2<nd;t2++)
					{
					if((drivers[temp][0] - drivers[t2][0] + 100000)%1000<p1)
						p1 = (drivers[temp][0] - drivers[t2][0] + 100000)%1000;  //100000 is
sufficiently large to ensure that this will be positive
					}
```

```
				if(p1 > 100)
					{
					drivers[temp][0] -= 100;
					drivers[temp][12] = 0;
					}
//100 R3U is how much distance a car can lose by slowing down and then getting back up to speed in one turn.
				else
					{
					drivers[temp][0] -= p1 - 2;
					drivers[temp][12] = 0;
					}
				}
			}



/*				drivers[temp][11] = -1;
				if(drivers[temp][0] - drivers[temp+1][0] > 3)
					drivers[temp][7] -= 2*(drivers[temp][0] - drivers[temp+1][0] - 2);

				if(drivers[temp][7] < 0)
					drivers[temp][7] = 0;

//I'm keeping this code since it may yet be useful.  It was one of my earlier attempts
//to get the losing drivers to do things nicely.  It worked reasonably well, too.

				if(drivers[temp][0] - drivers[temp+1][0] > 50)
					drivers[temp][7] -= 25;
				else if(drivers[temp][0] - drivers[temp+1][0] > 20)
					drivers[temp][7] -= 10;
				else if(drivers[temp][0] - drivers[temp+1][0] > 10)
					drivers[temp][7] -= 3;
*/

		for(temp=0;temp<nd;temp++)
			{
			rnd = rn.nextFloat();
			if(rnd < drivers[temp][2]/10)
				{
				drivers[temp][9] = 1;
				drivers[temp][0] -= .1;
				}
			}
		passcheck(drivers, driverorder, grudges, rn, nd, roundover);

		//Note that power is computed before other speed-changing effects are added.
```

```
		for(temp=0;temp<nd;temp++)
			drivers[temp][0] += (drivers[temp][7] + drivers[temp][8])/2;
		for(temp=0;temp<nd-1;temp++)
			{
			if(drivers[temp][0] - drivers[temp+1][1] < 2 && drivers[temp][0] - drivers[temp+1][1] > -2 &&
drivers[temp+1][10] ==0)
				drivers[temp+1][0] = drivers[temp][0];
			}

		for(temp=0;temp<nd;temp++)
			{
			unblocked = 1;
			if(drivers[temp][10] > 1)
				drivers[temp][10] --;
			else if(drivers[temp][10] == 1)
				{
				for(t2=0;t2<nd;t2++)
					{
					if(drivers[t2][0] - drivers[temp][0] < 1 && drivers[t2][0] - drivers[temp][0] > 1)
						unblocked = 0;
					}
				if(unblocked ==1)
					drivers[temp][10] --;
				}
			}

/**/		sortdrivers(drivers, driverorder, grudges, nd);

//Debug code
/*		System.out.println("");

		for(temp=0;temp<nd;temp++)
			System.out.println(drivers[temp][0]);
		for(temp=0;temp<nd;temp++)
			System.out.println(driverorder[temp]);
*/

/*Points system gives 0 points for leading, 2 points for second, on to 20 points for last in the lap.  Plus, you get
points equal to: top position - your position at the end of the race.  This is a modified version of the NASCAR system,
but designed to convert directly into fitness rankings (the reason for flipping the rankings is that now there is a huge
proportionality difference between the top guy and number 2, whereas otherwise there would not be so much of a
difference.  If you are selecting an organism to kill randomly, you want to make sure that you pick the top guys rarely,
but differentiating between the weaker ones does not matter as much.*/

		for(temp=0;temp<nd;temp++)
```

```
                    driverpoints[driverorder[temp]] += 2*temp;

////////////////////////////////////////////////////Graphics Section//////////////////////////////////////

            if((counter%500)==499)
                    {
                    input.readInt();
//Debug code
/*                  for(temp=0;temp<nd;temp++)
                            System.out.print(drivers[temp][6] + "    ");
                    System.out.println("\n");
                    for(temp=0;temp<nd;temp++)
                            System.out.print(drivers[temp][10] + "    ");
*/
                    }

            first = drivers[0][0];

            for(temp=0;temp<nd;temp++)
                    {
                    t2 = driverorder[temp];
                    p1 = (first - drivers[temp][0]+1000000)%1000;
                    if(t2==0)
                       g2.setColor(Color.blue);
                    else if(t2==1)
                       g2.setColor(Color.red);
            else if(t2==2)
                    g2.setColor(Color.yellow);
            else if(t2==3)
                    g2.setColor(Color.green);
            else if(t2==4)
                    g2.setColor(Color.orange);
            else if(t2==5)
                    g2.setColor(Color.pink);
            else if(t2==6)
                    g2.setColor(Color.white);
            else if(t2==7)
                    g2.setColor(Color.black);
            else if(t2==8)
                    g2.setColor(Color.pink);
            else if(t2==9)
                    g2.setColor(Color.black);
            else if(t2==10)
                    g2.setColor(Color.white);

                    g2.draw(new Line2D.Double(p1, (counter%500), p1+2, (counter%500)));
```

```
                    }

                }
        rnd = drivers[0][0];
        for(temp=0;temp<nd;temp++)
                {
                driverpoints[driverorder[temp]] += rnd - drivers[temp][0];
                driverpoints[temp]++;
                }

//      return driverpoints;
        }

/////////////////////////////////////////////Passing Function//////////////////////////////////////////////////
/*Pass check:
Desire - 10*aggression = % chance to attempt any pass
Position - Is car part of 3+ car line & not in front?  Yes = good, No = forget it.
If 2 passes are available, at this point calculate them both.
Pass fore = 1.5*all. Asking someone to pass when one is in back is good, because he has less chance of being
tricked.  All of the drivers (like real drivers) instinctively know this, no matter what stats they have.
Driver's adaptive*(1 - his defense*.05) - his grudge against passing partner - his adaptive*passing partner's grudge
against him is an equation used to determine whether that driver will try to pass or not.
Pass acceptance for falsepass is: deception if guy is in front of you, deception/10 if guy is behind you.
Pass acceptance for truepass is: aggression - grudge against other guy

If a driver is making a pass to decieve, then he must be the last guy in a chain of 3.
He must - Have failed his aggression check
Passed a deception check
Have the guy in front of him accept the offer

Once a driver makes a pass, he moves into a temporary lane.  Once the driver moves out, he gain an instant 5 R3U
speed bonus due to the "slingshot effect." (yes, there is such a thing and it does give a temporary speed boost)  The
max speed of all other drivers is recalculated.  If he is moving out with another driver who is immediately behind
him, the other driver does not gain the speed boost but will be sucked up into his draft.  The guy whom they are passing
"gains" a 5 R3U speed downgrade due to air vortices sucking him back (yes, this happens as well.)

If a driver wants to make a 2-person slingshot pass, then there must not be anyone else within 10 R3U of either his
car of the cars he is trying to pass, because that driver would take advantage of the pass and manage to abort it.

These drivers are gentlemen.  If a driver is already involved in a pass, they won't try to take advantage of them
(until the next turn, at least).  This not only makes them look good, it lowers the processing load.
*/
```

```
public static void passcheck(double drivers[][], int driverorder[], int grudges[][], Random rn, int nd, boolean
roundover[])
            {
 double passfore = 0, passback = 0;
 int desiretrue = 0;
 int temp = 0, vstore, vstore2, vstore3;
 int pass;
 int passtried;
 double rnd, dtmp;
 int pdist = 6;
 int ptime = 2;

 for(temp=1;temp<nd;temp++)
  {
  passtried = 0;
  rnd = rn.nextFloat();
  if(rnd < drivers[temp][1]/10)
  {
            //If in middle of a 3+ car pack
   passfore = 0;
   passback = 0;
   if(temp!= (nd-1) && !roundover[temp] && !roundover[temp+1])
   if(drivers[temp][0] - drivers[temp + 1][0] <= pdist && drivers[temp - 1][0] - drivers[temp][0] <= pdist)
            //How much the car wants to pass with the guy on his back
     passback = (double) 10 + .1*drivers[temp][5]*(1 - drivers[temp + 1][2]*.05 - .1*grudges[temp + 1][temp]) -
.1*grudges[temp][temp-1];
            //If at the back of a 3+ car pack
   if(!roundover[temp] && !roundover[temp-1])
   if(temp != 1 && drivers[temp][0] - drivers[temp - 1][0] <= pdist && drivers[temp - 2][0] - drivers[temp - 1][0] <=
pdist)
    passfore = (double) 12 + .1*drivers[temp][5]*(1 - drivers[temp - 2][2]*.05 - .5*grudges[temp-1][temp]) -
.1*grudges[temp][temp-1];

   if(passfore>passback)
            {
            rnd = rn.nextFloat();
            dtmp = drivers[temp][1]/(drivers[temp][3]+drivers[temp][1]) ;
            vstore = 0;
            if(rnd<dtmp)
                        vstore = 1;

            rnd = rn.nextFloat();
            if(vstore == 0 && rnd < .5 + drivers[temp-1][1]/50 - drivers[temp][3]*drivers[temp-1][5]/50)
                        {
                        rnd = rn.nextFloat();
                        passtried = 1;
```

```
			roundover[temp] = true;
			roundover[temp-1] = true;
			if(rnd > .5 + drivers[temp-1][9]/6)
				{
				drivers[temp-1][10] = ptime;
				drivers[temp][8] += 5;
				grudges[temp-1][temp] += 2;
//				System.out.println("Driver " +temp+ " tricked driver "+(temp-1));
				}
			else
				{
				grudges[temp-1][temp] += 1;
//			System.out.println("Driver " +temp+ " tried to trick driver "+(temp-1));
				}
			}
		rnd = rn.nextFloat();
		if(rnd < .5 + drivers[temp-1][1]/50 - drivers[temp][3]*drivers[temp-1][5]/50)
			{
			rnd = rn.nextFloat();
			passtried = 1;
			roundover[temp] = true;
			roundover[temp-1] = true;
			if(temp > 1 && rnd > drivers[temp-2][9]/2)
				{
				drivers[temp][10] = ptime;
				drivers[temp-1][10] = ptime;
				drivers[temp][8] += 5;
				drivers[temp-1][8] += 5;
//				System.out.println("Drivers "+temp+" and "+(temp-1)+" passed driver "+(temp-2));
				}
			else if(temp > 1)
				{
				rnd = rn.nextFloat();
			grudges[temp-1][temp] += (int) rnd*2;
				drivers[temp][6] += rnd;
				rnd = rn.nextFloat();
				drivers[temp-1][6] += rnd/5;
//				System.out.println("Drivers "+temp+" and "+(temp-1)+" tried to pass driver "+(temp-1));
				}
			}
		if(passback > 0&&passtried==0)
			passback += 100;  //If no pass has been tried (that is, the driver in front rejected the offer and
					//passing to the back is possible, then this says to go ahead and try it.
		}

	if(passback>passfore && temp!=(nd-1))
```

```
        {
        rnd = rn.nextFloat();
     if(rnd < .5 + drivers[temp-1][1]/20)
            {
                roundover[temp] = true;
                roundover[temp+1] = true;
            rnd = rn.nextFloat();
                vstore2 = 0;
            if(temp > 1 && rnd > drivers[temp-2][9]/2)
                {
                drivers[temp][10] = ptime;
                drivers[temp-1][10] = ptime;
                drivers[temp][8] += 5;
                drivers[temp-1][8] += 5;
                vstore2 = 1;
//                      System.out.println("Drivers "+temp+" and drivers " + (temp+1) + " passed driver " + (temp-1));
                        }
            rnd = rn.nextFloat();
                if(temp > 1 &&rnd > .75&&vstore2 == 0)
                {
                rnd = rn.nextFloat();
                grudges[temp-1][temp] += (int) rnd*2;
                drivers[temp][6] += rnd;
                rnd = rn.nextFloat();
                drivers[temp-1][6] += rnd/5;
//                System.out.println("Drivers "+temp+" and drivers "+(temp+1)+" tried to pass driver "+(temp-1));
                        }
                }
        }
   pass = 1;
   rnd = rn.nextFloat();

   if(passtried==0 && drivers[temp-1][0] - drivers[temp][0] <= pdist && (temp==10 || drivers[temp][0] -
drivers[temp+1][0] > pdist))
        {
                if(!roundover[temp] && !roundover[temp-1])
                {
                passtried = 1;
                roundover[temp]=false;
                roundover[temp-1]=false;
                if(drivers[temp-1][9]*.5>rnd)
                        {
                        rnd = rn.nextFloat();
                        drivers[temp][6] += rnd;
                        rnd = rn.nextFloat();
                        drivers[temp-1][6] += rnd/2;
```

```
                    }
                else
                    {
                    drivers[temp][8] += 5;
                    drivers[temp-1][8] -= 5;
                    drivers[temp][10] = ptime;
                    }
                }
        }

   rnd = rn.nextFloat();
    }
    }  //These are for the two massive for / if loops which span most of the program.

 for(temp=0;temp<nd;temp++)
        roundover[temp] = false;

        }
//////////////////////////////////////////////////Grudge Checking////////////////////////////////////
   public static void grudgecheck(double drivers[][], int grudges[][], Random rn, int nd)
        {
        int temp, t2;
        double rnd, rn2;
        double dtemp;
        for(temp=0;temp<nd;temp++)
         {
         for(t2=0;t2<nd;t2++)
          {
          rnd = (double) rn.nextFloat();
          if(grudges[temp][t2]/100. > rnd && grudges[temp][t2] > 20 - drivers[temp][4] && drivers[temp][0] -
drivers[t2][0] <= 10 && drivers[temp][0] - drivers[t2][0] <= 10)
                {
                rnd = (double) rn.nextFloat();
                if(rnd > (1 - drivers[t2][9]*.25)*(.75))  //Defense gives .25 chance of blocking grudges
                    {
                    rnd = rn.nextFloat();
                    drivers[t2][6] += rnd;
                    rn2 = rn.nextFloat()/2;
                    drivers[temp][6] += rn2;
                    if(rn2/(2*rnd) > 1)
                        grudges[temp][t2] -= 2;
                    grudges[t2][temp] += 1;
                    }
                else
                    {
                    rnd = rn.nextFloat();
```

```
                        drivers[temp][6] += rnd;
                        rnd = rn.nextFloat();
                        rn2 = rn.nextFloat();
                        dtemp = rnd / rn2;
                        drivers[t2][6] += dtemp;
                        grudges[temp][t2] -= (int) dtemp;
                        grudges[t2][temp] += (int) dtemp/2;
                        }
                if(grudges[temp][t2] < 0)
                        grudges[temp][t2] = 0;
                }
         }
        }
       }
//////////////////////////////////////////////Driver Strategy Function////////////////////////////////////////
        /*A signal of 0 means to do nothing.  A signal of 1 means to not try to pass and instead just focus on catching
up.  A signal of 2 means to slow down and collect in packs.
        Calculating signals: Check of adaptive whether to do a darn thing!
If your average speed is at least .5 less than the average speed of the dudes in the front, then you do either a 1 or a 2.
        If the car has either a car in front with dist <= 10 and decreasing or a car in back with dist <= 10 and
decreasing, then return a 1.  1 goes to passcheck and considers the car to already have made a pass.  Whichever car has
less damage will go to the front.  Whichever car has the most damage will be at the back.  This organized formation
will
disintegrate when the lead car is within 10 units of the front.  After all, what advantage do the guys in the front gain
from being nice to each other?
        2 means to group.  If the car is closer to somebody in back then in front, then stop.  Else keep going.  If
they're still not catching up, look at the group as a car and maybe do output 3.
*/

public static void strategery(double [][] drivers, int [][] grudges, Random rn, int nd)
        {
        int temp, t2;
        double rnd;
        double dtemp;
        double closeup, closedown;
        for(temp=1;temp<nd;temp++)
                {
                if(drivers[temp][11]==1)
                        drivers[temp][11] = 0;
                else if(drivers[temp][11]==2)
                        drivers[temp][11] = 1;
                else if(drivers[temp][11]>2)
                        drivers[temp][11] --;
                rnd = rn.nextFloat();
                if(drivers[temp][11] == 0)
                        {
```

```
//These lines control driver frustration.  If driver no. 2 is close to the front, then they are considered one unit.
//Frustration is caused when a driver's speed is lower than the driver(s) in the front.
			if(drivers[0][0] - drivers[1][0] > 10)
				drivers[temp][12] += drivers[0][8] - drivers[temp][8];
			else
				drivers[temp][12] += (drivers[0][8] + drivers[1][8])/2 - drivers[temp][8];

			if(drivers[temp][12]<0)
				drivers[temp][12] = 0;

			if(drivers[temp][12]>50)
				drivers[temp][12] = 50;

			if(drivers[temp][12] > 25 && temp != nd-1)
				{
				closeup = (drivers[0][0] - drivers[temp][0])%1000;
				closedown = closeup;
				for(t2=1;t2<nd;t2++)
					{
					dtemp = (drivers[t2][0] - drivers[temp][0]+1000000)%1000;

					if(dtemp > 0 && dtemp < closeup)
						closeup = dtemp;
					else if(dtemp > 0 && dtemp > closedown)
						closedown = dtemp;
					}
				closedown = 1000 - closedown;

				if(closeup <= 3 || closedown <= 3)
					drivers[temp][11] = 1;

				if(closeup <= 3 || closedown <= 3)
					break;

				if(drivers[temp-1][11] != 2 && closeup > closedown)
					{
					drivers[temp][11] = 2;
					drivers[temp][12] = 0;
					}
				if(closeup > 150)
					{
					drivers[temp][11] = 2;
					drivers[temp][12] = 0;
					}
				}
			}
```

```
		if(drivers[temp][11] < 0)
			drivers[temp][11]++;

		if(drivers[temp][11]!=2 && drivers[0][0] - drivers[temp][0] > 1000)
			drivers[temp][11] = 1;  //This is, plain and simple, a patch line.  I was hoping to avoid this
						//type of thing, but I can't see any good method to get the drivers to
						//stifle their natural competition.
		}

	drivers[0][12] = 0;
	drivers[0][11] = 0;

//This little loop of code makes the preceding function return no "2"'s, which should help.
	for(temp=0;temp<nd;temp++)
		{
		if(drivers[temp][11] == 1)
			drivers[temp][11] = 0;
		}

	}
  } /////////////////////////////////////////////End of Program/////////////////////////////////////////////
```

Basic Genetic Algorithm

This is the program racealg, the genetic algorithm which uses the raceplot program to calculate the fitness of each of the drivers. This genetic algorithm is fairly simple, but worked quite well, particularly in the tests where each driver was run against its fellow drivers to determine performance.

```
import java.util.*;
import java.io.*;        //Importing the java functions

public class racealg
  {
  public static void main(String[] args)
        {
//Declare variables//

     int numorgs = 10;                                  //10 drivers (+ 1 elite driver)
     int alleles = 5;                                   //The 5 traits
     int range = 10;                                    //The properties go from 0 to 10
     range++;

     int [][] organisms = new int[numorgs+1][alleles];  //The organisms
     double [] fitness = new double[numorgs+1];
     int [] deadorg = new int[numorgs+1];               //Which organisms are to be overwritten
     int [][] temporg = new int[9][alleles];            //Temporary storage for the organisms which will
                                                        //overwrite those in deadorg.
        Random rn = new Random();                       //Using a single random is much better than
        int temp, t2;                                   //declaring it in each new function.
        int bestgot = 0, wpos = 0, count = 0;

        int mutorgs, killrnd, countsum;
        double average;

        mutorgs = 3;                                    //How many mutations
        killrnd = 1;                                    //How many killed per round

        CSPrintWriter output = new CSPrintWriter("bestorgs");   //Storage file for the best driver per round.

initialize(organisms, rn, numorgs, alleles, range);

while(true)                                             //The main loop
{
mutate(organisms, mutorgs, rn, numorgs, alleles, range);
calcfit(organisms, fitness, numorgs);
best_find(organisms, fitness, numorgs, alleles, output);
kill_orgs(fitness, deadorg, killrnd, rn, numorgs);
breed_orgs(organisms, deadorg, temporg, killrnd, rn, numorgs, alleles);
```

```
birth_orgs(organisms, deadorg, temporg, numorgs, alleles);

//calcfit(organisms, fitness, numorgs);
//best_find(organisms, fitness, numorgs, alleles, output);

/*With these two lines gone, the risk is that some truly superior organism will be produced through breeding but before
it can be stored in the "elite" position, it will be adversely affected by the mutation function.  On the other hand,
removing those two lines allows the program to run twice as many iterations in the same amount of time.*/

//Debug Code
  count++;
/*
  if(count%100==0)

        {
        System.out.println("");
        for(temp=0;temp<11;temp++)
          {
          for(t2=0;t2<10;t2++)
                {
                System.out.print(organisms[temp][t2] + " ");
                }
          System.out.println("");
          }
        }
*/

System.out.println("Counter = "+count);
if(count==10000)
        break;                                              //Stop here
}
//Debug Code
//System.out.println("The ideal solution has been found, using: " + count + " iterations.);
//              average = (double) countsum / 10000;
//              System.out.println("With "+i+" mutations and "+j+" killed every round, it averaged "+average);

        }

//////////////////////////////////////////Initialization Function/////////////////////////////////////////
/*This funciton randomly initializes the organisms*/
  public static void initialize(int [][] organisms, Random rn, int numorgs, int alleles, int range)
        {
     int temp, t2;
        for(temp=0;temp<numorgs+1;temp++)
          {
                for(t2=0;t2<alleles;t2++)
```

```
                    organisms[temp][t2] = rn.nextInt(range);
                }
        }

///////////////////////////////////////////////Organism Mutator/////////////////////////////////////////
/*This function mutates the organisms, but will not do anything to the current elite organism*/
public static void mutate(int [][] organisms, int mutations, Random rn, int numorgs, int alleles, int range)
            {
            int mut1, mut2, mut3, temp;
            for(temp=0;temp<mutations;temp++)
               {
               mut1 = rn.nextInt(numorgs);
               mut2 = rn.nextInt(alleles);
               mut3 = rn.nextInt(range);
               organisms[mut1][mut2] = mut3;
               }
            }

//////////////////////////////////////////Fitness Calculator//////////////////////////////////////////
public static void calcfit(int [][] organisms, double [] fitness, int numorgs)
        {
            int temp, t2;

            CSPrintWriter output = new CSPrintWriter("testorgs.txt");
            CSInputReader input = new CSInputReader("raceresults.txt");

            for(temp=0;temp<numorgs+1;temp++)
                    {
                    for(t2=0;t2<5;t2++)
                            output.print(organisms[temp][t2]+" ");
                    output.println("");
                    }
            output.close();

        try
                    {
                    Process theo = Runtime.getRuntime().exec("java theoretical");
                    try
                            {
                    theo.waitFor();
                            }
                    catch(InterruptedException ie)
                            {
                            }
                    }
```

```
	catch (IOException e1)
		{
	System.err.println(e1);
	System.exit(1);
		}

	for(temp=0;temp<numorgs+1;temp++)
		fitness[temp] = input.readDouble();
	}

///////////////////////////////////////////Check Exit Condition/////////////////////////////////////////
/*This checks to see if the exit condition is satisfied - used to test the validity of the GA*/
public static int check_ideal(double [] fitness)
	{
	int temp;
	for(temp=0;temp<11;temp++)
		{
		if(fitness[temp]==0)
			return 1;
		}
	return 0;
	}

/////////////////////////////////////////////Find Elite Organism////////////////////////////////////////
/*Finds the best organism and puts it in the elite position.  From there, it is not mutated*/
public static void best_find(int [][] organisms, double [] fitness, int numorgs, int alleles, CSPrintWriter output)
	{
      double a = fitness[0];
      int temp, bestval = 0;
	int different = 0;
	for(temp=1;temp<numorgs+1;temp++)
	   {
	   if(fitness[temp]<a)
		{
		bestval = temp;
		a = fitness[temp];
		}
	   }
		//Originally there was a command here to only print the best organism when it changed from the past
		// round, in order to make the output data smaller.  This has been removed.
		for(temp=0;temp<alleles;temp++)
			output.print(organisms[bestval][temp]+"\t");
		output.println("");
	}
```

```
//////////////////////////////////////////Find Worst Organism///////////////////////////////////////
/*This function finds the organism which did the worst and returns its position (in the current program it is not used,
but some GA variations automatically kill the weakest organism)*/
public static int worst_find(double [] fitness, int numorgs)
	{
	double a = fitness[0];
	int temp, curworst = 0;
	for(temp=1;temp<numorgs;temp++)
		{
		if(fitness[temp]>a)
			{
			curworst = temp;
			a = fitness[temp];
			}
		}
	return curworst;
	}

///////////////////////////////////////////////Organism Killer////////////////////////////////////
/*This function marks which organisms are to be replaced.*/
public static void kill_orgs(double [] fitness, int [] deadorg, int kills, Random rn, int numorgs)
	{
	int sumfit=0, removed = 0;
	int temp;
	double a;
      for(temp=0;temp<numorgs;temp++)
          sumfit+=fitness[temp];

	while(removed<kills)
		{
		a = (double) sumfit*rn.nextFloat();
		if(a < fitness[0]&&deadorg[0]==0)
			{
			deadorg[0] = 1;
			removed++;
			}
		else if(a < fitness[0] + fitness[1]&&deadorg[1]==0)
			{
			deadorg[1] = 1;
			removed++;
			}
		else if(a < fitness[0] + fitness[1] + fitness[2]&&deadorg[2]==0)
			{
			deadorg[2] = 1;
			removed++;
			}
```

```
        else if(a < fitness[0] + fitness[1] + fitness[2] + fitness[3]&&deadorg[3]==0)
                {
                deadorg[3] = 1;
                removed++;
                }
        else if(a < fitness[0] + fitness[1] + fitness[2] + fitness[3] + fitness[4]&&deadorg[4]==0)
                {
                deadorg[4] = 1;
                removed++;
                }
        else if(a < fitness[0] + fitness[1] + fitness[2] + fitness[3] + fitness[4] + fitness[5]&&deadorg[5]==0)
                {
                deadorg[5] = 1;
                removed++;
                }
        else if(a < fitness[0]+fitness[1]+fitness[2]+fitness[3]+fitness[4]+fitness[5]+
        fitness[6]&&deadorg[6]==0)
                {
                deadorg[6] = 1;
                removed++;
                }
        else if(a < fitness[0]+fitness[1]+fitness[2]+fitness[3]+fitness[4]+fitness[5]+
        fitness[6]+fitness[7]&&deadorg[7]==0)
                {
                deadorg[7] = 1;
                removed++;
                }
        else if(a <fitness[0]+fitness[1]+fitness[2]+fitness[3]+fitness[4]+fitness[5]+
        fitness[6]+fitness[7]+fitness[8]&&deadorg[8]==0)
                {
                deadorg[8] = 1;
                removed++;
                }
        else if(deadorg[9]==0)
                {
                deadorg[9] = 1;
                removed++;
                }
        }
}
```

```
////////////////////////////////////////////////Organism Breeder/////////////////////////////////////////////
/*This organism breeds organisms, randomly selecting alleles from two randomly selected parents.*/
public static void breed_orgs(int [][] organisms,int []deadorg, int [][] temporg, int numbreed, Random rn, int numorgs,
int alleles)
	{
	int t2, temp, bred1, bred2, bred3;
	for(t2=0;t2<numbreed;t2++)
		{
		bred1 = rn.nextInt(numorgs+1);
		bred2 = rn.nextInt(numorgs+1);
		while(deadorg[bred1]==1||deadorg[bred2]==1)
			{
			bred1 = rn.nextInt(numorgs+1);
			bred2 = rn.nextInt(numorgs+1);
			}
		for(temp=0;temp<alleles;temp++)
			{
			bred3 = rn.nextInt(2);
			if(bred3==0)
				temporg[t2][temp] = organisms[bred1][temp];
			else
				temporg[t2][temp] = organisms[bred2][temp];
			}
		}
	}

///////////////////////////////////////////////Organism Overwriter///////////////////////////////////////
/*This function overwrites the organisms marked in kill_orgs with the organisms in the array temporg*/
  public static void birth_orgs(int [][] organisms, int [] deadorg, int [][] temporg, int numorgs, int alleles)
	{
	   int a = 0, temp, t2;
	for(t2=0;t2<numorgs;t2++)
		{
		if(deadorg[t2]==1)
			{
			for(temp=0;temp<alleles;temp++)
				organisms[t2][temp] = temporg[a][temp];
			a++;
			}
		}
	for(a=0;a<numorgs;a++)
		deadorg[a]=0;
	    }
  }
```

MGG Genetic Algorithm

This is the code for the Minimal Generation Gap program, the more advanced version of the basic genetic algorithm. In general, tests were performed using both the basic genetic algorithm and the MGG algorithm.

```
import java.util.*;
import java.io.*;        //Importing the java functions

public class mgg
  {
  public static void main(String[] args)
        {
//Declare variables//

     int numorgs = 10;                                   //10 drivers (+ 1 elite driver)
     int alleles = 5;                             //The 5 traits
     int range = 10;                                     //The properties go from 0 to 10
     range++;

     double [][] organisms = new double[numorgs+1][alleles];     //The organisms
     double [] fitness = new double[numorgs+1];
     int [] deadorg = new int[numorgs+1];          //Which organisms are to be overwritten
     double [][] temporg = new double[9][alleles];       //Temporary storage for the organisms which will
                                                         //overwrite those in deadorg.
        Random rn = new Random();                        //Using a single random is much better than
        int temp, t2;                                    //declaring it in each new function.
        int bestgot = 0, wpos = 0, count = 0;

        int mutorgs, killrnd, countsum;
        double average;

        mutorgs = 1;                                     //How many mutations
        killrnd = 1;                                     //How many killed per round

        CSPrintWriter output = new CSPrintWriter("bestorgs");     //Storage file for the best driver per round.
        CSPrintWriter mggout = new CSPrintWriter("orgchange.txt");


initialize(organisms, rn, numorgs, alleles, range);

while(true)                                              //The main loop
{
mutate(organisms, mutorgs, rn, numorgs, alleles, range);
mgg_run(organisms, numorgs, alleles, rn, mggout);
```

```
count++;
System.out.println("Counter = " + count);
if(count==10000)
        break;                                          //Stop after 100 iterations.
}
//Debug Code
//System.out.println("The ideal solution has been found, using: " + count + " iterations.);
//              average = (double) countsum / 10000;
//              System.out.println("With "+i+" mutations and "+j+" killed every round, it averaged "+average);

        }

////////////////////////////////////////////Initialization Function////////////////////////////////////////
/*This funciton randomly initializes the organisms*/
  public static void initialize(double [][] organisms, Random rn, int numorgs, int alleles, int range)
        {
     int temp, t2;
        for(temp=0;temp<numorgs+1;temp++)
           {
                for(t2=0;t2<alleles;t2++)
                organisms[temp][t2] = rn.nextInt(range);
           }
     }

///////////////////////////////////////////Fitness Calculator//////////////////////////////////////////
public static void calcfit(double [][] organisms, double [] fitness, int numorgs)
     {
        int temp, t2;

        CSPrintWriter output = new CSPrintWriter("testorgs.txt");
        CSInputReader input = new CSInputReader("raceresults.txt");

        for(temp=0;temp<numorgs+1;temp++)
                {
                for(t2=0;t2<5;t2++)
                        output.print(organisms[temp][t2]+" ");
                output.println("");
                }
        output.close();

     try
                {
                Process theo = Runtime.getRuntime().exec("java theoretical");
                try
                        {
                theo.waitFor();
```

```
                        }
                catch(InterruptedException ie)
                        {
                        }
                }

        catch (IOException e1)
                {
        System.err.println(e1);
        System.exit(1);
                }

        for(temp=0;temp<numorgs+1;temp++)
                fitness[temp] = input.readDouble();

        }

/////////////////////////////////////////////Check Exit Condition/////////////////////////////////////////////
/*This checks to see if the exit condition is satisfied - used to test the validity of the GA*/
public static int check_ideal(double [] fitness)
        {
        int temp;
        for(temp=0;temp<11;temp++)
                {
                if(fitness[temp]==0)
                        return 1;
                }
        return 0;
        }

///////////////////////////////////////////////Find Elite Organism///////////////////////////////////////////
/*Finds the best organism and puts it in the elite position.  It can breed but cannot be removed*/
public static void best_find(double [][] organisms, double [] fitness, int numorgs, int alleles, CSPrintWriter output)
        {
      double a = fitness[0];
      int temp, bestval = 0;
        int different = 0;
        for(temp=1;temp<numorgs+1;temp++)
           {
           if(fitness[temp]<a)
                {
                bestval = temp;
                a = fitness[temp];
                }
           }
```

```
		//Originally there was a command here to only print the best organism when it changed from the past
round,
		//in order to save space.  This has been removed.
		for(temp=0;temp<alleles;temp++)
			output.print(organisms[bestval][temp]+"\t");
		output.println("");
	}

////////////////////////////////Find Elite Organism - Nonprint Version/////////////////////////////////////
//This functino also finds the elite organism, but does not write to file and only returns position.
public static int best_find(double [] fitness, int numorgs)
	{
	double a = fitness[0];
	int temp, bestval = 0;
	int different = 0;
	for(temp=1;temp<numorgs+1;temp++)
	   {
	   if(fitness[temp]<a)
		{
		bestval = temp;
		a = fitness[temp];
		}
	   }
	return bestval;
	}


///////////////////////////////////////////MGG Segment/////////////////////////////////////
public static void mgg_run(double [][] organisms, int numorgs, int alleles, Random rn, CSPrintWriter mggout)
	{
//Variable Declarations
	int temp, t2, t3;
	double rnd, rnd2;
	int p1, p2;
	int checkzero = 1;
	double dtmp;

	double [] axisvect = new double[numorgs];
	double [] vect1 = new double[numorgs];
	double [] vect2 = new double[numorgs];
	double [] vect3 = new double[numorgs];
	double [] vect4 = new double[numorgs];
	double [] vect5 = new double[numorgs];
	double [] tempvect = new double[numorgs];		//I found it to be easier to write out single-dimensional
	double [] tempvect2 = new double[numorgs];		//vector arrays and then combine them into a 2D array
	double [] tempvect3 = new double[numorgs];		//since it's easier to write functions for one array
```

```
	double [] tempvect4 = new double[numorgs];		//rather than having a position value and sending a large
array.

	p1 = rn.nextInt(numorgs+1);
	p2 = rn.nextInt(numorgs+1);

	while(p1==p2)
		p2 = rn.nextInt(numorgs+1);

	for(temp=0;temp<alleles;temp++)
		{
		if(organisms[p1][temp] != organisms[p2][temp])
			checkzero = 0;
		}

	while(checkzero == 1)
		{
		p2 = rn.nextInt(numorgs);

		for(temp=0;temp<alleles;temp++)
			{
			if(organisms[p1][temp] != organisms[p2][temp])
				checkzero = 0;
			}
		}

//New Variable Declaration
	double distance = 0;

	for(temp=0;temp<alleles;temp++)
		{
		axisvect[temp] = organisms[p1][temp] - organisms[p2][temp];
		distance += axisvect[temp]*axisvect[temp];
		}

	distance = Math.sqrt(distance);
/*Debug Code
	for(temp=0;temp<alleles;temp++)
		System.out.print(axisvect[temp] + "  ");

	System.out.println("");
	System.out.println("");
*/
	normalize(axisvect, alleles);

	vectmult(axisvect, vect1, -axisvect[0], alleles);
```

```
vect1[0]++;
normalize(vect1, alleles);

vectmult(vect1, vect2, -vect1[1], alleles);
vectmult(axisvect, tempvect, axisvect[1], alleles);
for(temp=0;temp<5;temp++)
        vect2[temp] -= tempvect[temp];

vect2[1]++;
normalize(vect2, alleles);

vectmult(vect1, vect3, -vect1[2], alleles);
vectmult(vect2, tempvect2, vect2[2], alleles);
vectmult(axisvect, tempvect, axisvect[2], alleles);
for(temp=0;temp<5;temp++)
        vect3[temp] -= tempvect[temp] + tempvect2[temp];

vect3[2]++;
normalize(vect3, alleles);

vectmult(vect1, vect4, -vect1[3], alleles);
vectmult(vect2, tempvect2, vect2[3], alleles);
vectmult(vect3, tempvect3, vect3[3], alleles);
vectmult(axisvect, tempvect, axisvect[3], alleles);
for(temp=0;temp<5;temp++)
   vect4[temp] -= tempvect[temp] + tempvect2[temp] + tempvect3[temp];

vect4[3]++;
normalize(vect4, alleles);

vectmult(vect1, vect5, -vect1[4], alleles);
vectmult(vect2, tempvect2, vect2[4], alleles);
vectmult(vect3, tempvect3, vect3[4], alleles);
vectmult(vect4, tempvect4, vect4[4], alleles);
vectmult(axisvect, tempvect, axisvect[4], alleles);
for(temp=0;temp<5;temp++)
        vect5[temp] -= tempvect[temp] + tempvect2[temp] + tempvect3[temp] + tempvect4[temp];

vect5[4]++;
normalize(vect5, alleles);

//This would probably be more elegant as a nested for loop, but only about the same in efficiency.  There are
//only a few values to compute, so it is just as easy, more understandable,  and only slightly longer to do it
//this way.

//New Variable Declaration
```

```
	double [][] vectors = new double[alleles][alleles];

	for(temp=0;temp<alleles;temp++)
		{
		vectors[0][temp] = axisvect[temp]*distance;
		vectors[1][temp] = vect1[temp]*distance;
		vectors[2][temp] = vect2[temp]*distance;
		vectors[3][temp] = vect3[temp]*distance;
		vectors[4][temp] = vect4[temp]*distance;
		}

	checkzero = 1;

	for(temp=0;temp<alleles;temp++)
		{
		if(vect5[temp] > .0001 || vect5[temp] < -.001)
			checkzero = 0;
		}

	t2 = 0;
	while(checkzero==0 && t2<alleles-1)
		{
		t2++;
		checkzero = 1;
		for(temp=0;temp<alleles;temp++)
			{
			if(vectors[t2][temp]>.0001 || vectors[t2][temp] < -.0001)
				checkzero = 0;
			}
		}

	if(t2 > 0 && t2!=5)
	{
	for(temp=0;temp<alleles;temp++)
		vectors[t2][temp] = vect5[temp]*distance;
	}

/* Debug Code
	for(temp=0;temp<alleles;temp++)
		{
		for(t2=0;t2<alleles;t2++)
			System.out.print(vectors[temp][t2] + "  ");
		System.out.println("");
		}

System.out.println("");
```

```
System.out.println("");

        int t3;

        for(temp=0;temp<alleles;temp++)
                {
                for(t2=0;t2<alleles;t2++)
                        {
                        for(t3=0;t3<alleles;t3++)
                                {
                                vect1[t3] = vectors[temp][t3];
                                vect2[t3] = vectors[t2][t3];
                                }
                        rnd = dotproduct(vect1, vect2, alleles);
                        System.out.print(rnd + "  ");
                        }
                System.out.println("");
                }
*/

//At this point, the array vectors holds five mutually orthogonal vectors, each of length equal to the
//distance between the two parents.

//New Variable Declarations
        double [][] spawn = new double[numorgs+1][alleles];        //Declaring some variables when they are used
        double [] center = new double[alleles];                    //rather than up front makes it easier
                                                                           //to understand the code.

        double dimratio = .33;  //Dimratio is the ratio of the primary dimension to the alternate dimensions.

        for(temp=0;temp<alleles;temp++)
                center[temp] = (organisms[p1][temp] + organisms[p2][temp])/2;

//Since orthogonal components are to be calculated independantly, to obtain an elliptical shape the
//probability of obtaining a certain value should, every time it goes 2x out, be 1/4x.  The function
// 1/2x - 1/2 is both effective and efficient.
        for(temp=0;temp<numorgs-1;temp++)
                {
                rnd = 1/(2*rn.nextFloat()) - 1/2;
                rnd2 = rn.nextFloat();
                if(rnd2>.5)
                        rnd *= -1;

                for(t2=0;t2<alleles;t2++)
                        spawn[temp][t2] = center[t2] + vectors[0][t2]*rnd;
```

```
		for(t2=1;t2<alleles;t2++)
			{
			rnd = 1/(2*rn.nextFloat()) - 1/2;
			rnd2 = rn.nextFloat();
			if(rnd2>.5)
				rnd *= -1;

			for(t3=0;t3<alleles;t3++)
				spawn[temp][t3] += rnd*vectors[t2][t3]*dimratio;
			}
		}

	for(temp=0;temp<alleles;temp++)
		{
		spawn[numorgs-1][temp] = organisms[p1][temp];
		spawn[numorgs][temp] = organisms[p2][temp];
		}

	for(temp=0;temp<numorgs;temp++)
		{
		for(t2=0;t2<alleles;t2++)
			{
			if(spawn[temp][t2] > 10)
				spawn[temp][t2] = 10;
			else if(spawn[temp][t2] < 0)
				spawn[temp][t2] = 0;
			}
		}

//New Variable Declarations
	double [] fitness = new double[11];
	int uberorg;

	calcfit(spawn, fitness, numorgs);
	uberorg = best_find(fitness, numorgs);

	t2 = rn.nextInt(numorgs+1);	//Normally I like to use temp variables only in FOR loops, but
	while(t2==uberorg)		//I've declared so many new variables
		t2 = rn.nextInt(numorgs+1);

	if(uberorg==numorgs-1)
		{
		System.out.println("Old Organism Won");
		mggout.println(1 + "\t" + p1);
		}
```

```
	else if(uberorg==numorgs)
		{
		System.out.println("Old Organism Won");
		mggout.println(1 + "\t" + p2);
		}

	else
		{
		System.out.println("New Elite Organism");
		mggout.println(0 + "\t" + new);
		}

	for(temp=0;temp<alleles;temp++)
		{
		organisms[p1][temp] = spawn[uberorg][temp];
		organisms[p2][temp] = spawn[t2][temp];
		}
	}

////////////////////////////////////////////////Vector Dot Product//////////////////////////////////
//This function calculates the dot product of two vectors
public static double dotproduct(double [] vect1, double [] vect2, int alleles)
	{
	int temp;
	double product = 0;

	for(temp=0;temp<alleles;temp++)
		product += vect1[temp]*vect2[temp];

	return product;
	}

//////////////////////////////////////////////Scalar Multiplication///////////////////////////////////
//This function multiplies a vector by a scalar
public static void vectmult(double [] vect, double [] tempvect, double scalar, int alleles)
	{
	int temp;

	for(temp=0;temp<alleles;temp++)
		tempvect[temp]= scalar*vect[temp];
	}

/////////////////////////////////////////////////Normalization///////////////////////////////////////
//This function will scale a vector so that its norm is equal to 1.  If the vector is a zero vector, this
//function will set it equal to 0.
public static void normalize(double [] vect, int alleles)
```

```
{
double nval = 0;
int temp;

for(temp=0;temp<alleles;temp++)
        nval += vect[temp]*vect[temp];

nval = Math.sqrt(nval);

for(temp=0;temp<alleles;temp++)
        vect[temp] /= nval;

if(nval<.0001)
        {
        for(temp=0;temp<alleles;temp++)
                vect[temp] = 0;
        }
}
```

Additional Data

Additional tests and data that were too long for the main report body

Optimizer

This data is from running the optimization program (which used the same raceplot function but fitness was how well a driver did against a fixed set of drivers) with the basic genetic algorithm. In this case, the fitness function was how well a driver did against a random set of other drivers. There were three drivers who did extremely well, doing the best in 6 consecutive runs:

8	0	1	2	0
9	2	0	7	8
9	5	4	7	0

The first two drivers were fairly similar, "honest" drivers with high aggression, low defense, and low deceptive, while grudges and adaptive varied. (both traits which made them pass quickly and get to the front fast, though not necessarily be able to hold the frontal position). The third driver had higher defensive and deceptive than the other two drivers, but still maintained high aggression. (possibly this combination of defensive and deceptive prevented grudge attacks from succeeding). Most likely, several types of drivers work well against the random set, with different drivers succeeding in different ways.

```
8    5    9    7    8    had: 1
9    5    4    7    0    had: 1
5    3    3    4    2    had: 1
8    5    9    7    8    had: 1
9    5    4    7    0    had: 2
4    2    4    7    0    had: 1
4    2    4    7    3    had: 1
8    5    9    7    10   had: 1
9    5    4    7    0    had: 1
6    3    6    0    3    had: 1
9    5    4    7    0    had: 1
8    5    9    5    0    had: 1
8    7    9    7    0    had: 1
8    4    9    5    0    had: 1
9    5    4    7    0    had: 3
6    3    8    6    1    had: 1
9    5    4    7    0    had: 2
8    2    4    5    0    had: 4
9    5    4    7    0    had: 1
8    5    5    7    7    had: 1
9    2    1    10   10   had: 2
8    2    4    5    0    had: 1
9    5    4    8    0    had: 1
6    0    8    6    1    had: 1
9    5    4    7    0    had: 2
6    0    5    6    2    had: 1
8    5    10   7    7    had: 1
6    2    8    7    4    had: 1
9    2    4    7    4    had: 1
9    5    4    7    0    had: 2
9    5    8    7    4    had: 1
9    5    9    1    8    had: 1
```

```
6   2   8   7   3    had: 2
9   5   4   9   4    had: 1
6   0   5   0   10   had: 1
9   5   4   7   0    had: 1
9   6   4   6   4    had: 1
8   5   9   7   4    had: 1
9   2   4   6   4    had: 1
9   4   7   2   4    had: 1
9   5   4   9   3    had: 1
8   2   9   2   6    had: 1
9   5   4   7   0    had: 1
8   5   9   6   4    had: 1
9   0   1   8   8    had: 1
9   6   4   7   0    had: 3
7   5   8   7   0    had: 1
9   5   4   7   0    had: 1
9   6   7   7   0    had: 1
9   5   4   7   0    had: 1
9   0   4   7   0    had: 1
9   5   4   7   0    had: 2
9   5   2   7   10   had: 1
7   0   8   7   4    had: 1
7   0   8   3   4    had: 2
9   1   4   7   0    had: 1
9   0   4   7   8    had: 1
10  6   1   6   10   had: 1
3   0   2   7   4    had: 1
10  6   1   6   10   had: 1
9   5   4   7   0    had: 1
9   0   4   7   4    had: 1
9   0   2   10  0    had: 2
9   3   4   7   5    had: 1
9   0   2   10  0    had: 4
9   0   5   10  0    had: 1
10  6   1   1   10   had: 1
9   0   5   10  0    had: 1
9   5   4   7   0    had: 1
9   0   2   8   0    had: 1
9   0   2   5   0    had: 2
9   2   2   10  5    had: 1
9   0   4   5   0    had: 1
9   5   4   7   0    had: 1
9   0   4   5   0    had: 1
10  2   2   5   4    had: 1
9   2   2   10  5    had: 1
10  6   1   1   10   had: 2
9   5   1   5   8    had: 1
10  6   4   7   0    had: 1
10  4   4   5   4    had: 1
9   5   1   5   8    had: 2
9   5   4   7   0    had: 1
9   5   1   5   8    had: 1
10  6   0   3   4    had: 2
10  5   4   7   0    had: 1
8   5   9   5   0    had: 1
10  5   1   3   9    had: 1
10  7   10  7   0    had: 1
10  5   4   7   0    had: 1
9   1   4   7   6    had: 1
9   5   4   7   0    had: 2
10  8   10  7   8    had: 1
9   5   1   1   1    had: 1
9   5   4   7   0    had: 1
9   5   1   1   1    had: 1
9   5   4   6   0    had: 3
9   8   4   5   0    had: 1
5   5   4   7   0    had: 1
9   8   4   5   10   had: 1
10  6   2   4   8    had: 1
9   5   4   7   0    had: 1
10  6   2   4   8    had: 1
9   5   4   7   0    had: 3
4   4   2   7   10   had: 1
9   5   4   7   0    had: 1
8   0   2   6   5    had: 1
9   5   4   7   0    had: 1
7   6   2   9   5    had: 1
8   0   2   6   5    had: 2
8   0   1   9   5    had: 1
5   0   7   4   5    had: 1
9   5   4   7   0    had: 3
7   0   2   9   8    had: 1
9   2   2   9   8    had: 1
9   5   4   7   0    had: 1
8   0   2   0   5    had: 1
8   0   1   9   5    had: 1
7   0   2   7   7    had: 1
8   1   1   9   5    had: 1
8   0   2   8   0    had: 1
8   3   1   9   5    had: 1
8   0   3   6   7    had: 1
8   1   1   5   5    had: 1
5   3   1   9   5    had: 1
8   0   10  8   6    had: 1
8   0   4   9   5    had: 1
8   0   3   6   7    had: 1
8   0   10  9   10   had: 2
6   3   1   3   4    had: 2
8   0   1   9   2    had: 1
8   3   1   5   5    had: 1
8   4   5   1   5    had: 1
8   0   1   9   2    had: 2
9   5   4   7   0    had: 1
8   0   3   6   4    had: 1
8   3   1   5   0    had: 1
8   0   3   2   4    had: 1
9   0   3   6   3    had: 1
8   3   1   5   0    had: 1
8   0   1   6   7    had: 1
8   3   1   5   0    had: 1
5   5   4   2   0    had: 1
9   3   1   5   0    had: 2
8   5   0   2   0    had: 2
9   3   3   7   0    had: 1
```

```
8    3    1    9    0    had: 1
9    5    5    7    0    had: 1
8    3    1    9    0    had: 1
8    3    4    5    0    had: 4
9    5    4    7    0    had: 1
9    8    4    7    0    had: 1
9    3    1    1    0    had: 1
9    3    3    3    8    had: 1
9    3    1    1    0    had: 1
9    2    6    1    10   had: 1
10   8    4    7    0    had: 1
9    5    4    7    0    had: 1
9    2    0    1    10   had: 1
9    5    4    7    0    had: 1
9    8    4    5    10   had: 1
10   8    4    4    0    had: 1
9    2    3    4    10   had: 1
9    3    4    7    10   had: 3
9    5    4    7    0    had: 1
9    3    4    7    10   had: 1
9    5    4    7    0    had: 1
9    2    0    2    10   had: 1
8    8    7    1    2    had: 1
9    3    1    2    2    had: 1
8    8    4    1    10   had: 1
9    3    1    2    2    had: 3
8    8    3    5    2    had: 1
6    3    1    7    2    had: 1
10   3    4    7    3    had: 1
8    3    4    9    10   had: 1
10   3    4    7    3    had: 1
9    5    4    7    0    had: 1
10   3    4    7    3    had: 2
9    3    4    5    3    had: 1
9    3    2    7    10   had: 1
10   3    4    2    0    had: 1
9    5    4    7    0    had: 1
10   3    9    7    2    had: 1
10   3    4    2    0    had: 1
10   3    9    7    2    had: 1
8    3    4    8    6    had: 1
9    2    5    1    4    had: 1
10   2    9    7    2    had: 1
10   3    9    8    3    had: 1
9    5    4    7    0    had: 1
9    1    10   7    0    had: 1
8    1    2    7    5    had: 3
8    1    2    7    0    had: 1
8    2    2    1    4    had: 1
8    3    2    7    3    had: 2
8    1    0    7    0    had: 3
9    1    0    10   0    had: 1
8    1    0    7    0    had: 1
8    1    10   2    0    had: 1
8    1    0    7    0    had: 1
9    5    4    7    0    had: 1
9    1    1    7    9    had: 1
10   4    10   8    2    had: 1
9    1    1    7    9    had: 3
9    5    4    7    0    had: 1
9    5    1    2    9    had: 1
9    4    1    10   0    had: 1
10   4    10   1    0    had: 2
9    7    1    2    6    had: 1
9    4    9    7    0    had: 1
8    0    1    7    0    had: 1
9    5    4    7    0    had: 1
10   1    10   7    4    had: 1
10   1    6    1    0    had: 1
10   4    10   3    0    had: 2
8    0    1    7    0    had: 1
9    1    4    7    0    had: 1
9    5    4    7    0    had: 1
8    4    10   3    0    had: 1
9    1    4    7    2    had: 1
10   0    10   1    4    had: 1
10   8    2    0    5    had: 1
9    4    9    7    1    had: 2
9    5    4    7    0    had: 2
10   0    10   1    4    had: 1
9    4    3    0    1    had: 2
9    0    10   7    0    had: 1
9    5    10   7    0    had: 1
10   0    4    7    0    had: 1
9    5    0    7    9    had: 1
9    0    4    7    0    had: 2
10   4    2    7    5    had: 1
9    0    4    7    0    had: 1
10   4    2    7    5    had: 1
9    0    4    7    0    had: 2
10   4    2    7    5    had: 1
9    0    4    7    9    had: 1
9    5    4    7    0    had: 1
9    0    4    7    9    had: 2
8    5    10   7    0    had: 1
9    0    4    7    9    had: 1
10   5    2    7    0    had: 2
10   10   4    7    10   had: 1
10   0    4    7    0    had: 1
10   2    2    7    0    had: 1
10   0    5    7    8    had: 1
10   5    2    7    0    had: 1
10   2    2    7    0    had: 2
9    5    4    7    0    had: 2
9    4    2    7    0    had: 1
3    2    8    6    0    had: 1
9    4    2    7    9    had: 1
9    5    4    7    0    had: 1
10   1    4    3    10   had: 2
6    10   4    9    0    had: 1
10   2    1    7    0    had: 1
10   1    4    3    10   had: 2
9    5    4    7    0    had: 1
10   1    4    3    10   had: 1
```

9 1 4 7 5 had: 1
10 2 9 7 3 had: 1
5 0 0 2 9 had: 1
9 1 2 7 5 had: 1
10 0 9 2 10 had: 1
10 2 9 2 3 had: 1
9 5 4 7 0 had: 1
7 0 9 2 9 had: 1
10 0 9 2 10 had: 1
8 2 9 9 3 had: 1
9 5 4 7 0 had: 3
10 8 8 2 10 had: 1
10 7 9 2 6 had: 1
9 5 4 7 0 had: 1
5 0 2 8 8 had: 1
9 0 5 2 0 had: 2
9 5 4 7 0 had: 1
9 0 5 2 0 had: 1
7 0 5 2 1 had: 3
10 0 9 3 9 had: 1
8 0 1 2 0 had: 6
9 2 1 8 0 had: 1
8 0 2 2 0 had: 2
8 7 1 3 0 had: 1
8 0 2 2 0 had: 1
7 7 3 3 0 had: 2
8 1 5 2 10 had: 1
9 0 2 7 0 had: 1
8 9 2 2 0 had: 1
9 9 7 7 0 had: 1
7 5 1 2 0 had: 1
8 4 3 8 3 had: 1
8 1 4 2 4 had: 1
7 5 1 2 0 had: 1
7 5 4 7 0 had: 2
9 5 4 9 0 had: 2
7 5 7 7 2 had: 1
9 5 4 7 0 had: 3
9 4 1 0 3 had: 2
7 5 0 2 0 had: 1
10 6 2 6 0 had: 1
7 5 0 2 0 had: 1
8 4 7 6 7 had: 1
10 6 2 6 0 had: 1
9 1 3 6 0 had: 1
9 6 2 7 0 had: 2
8 4 7 8 8 had: 1
9 1 3 6 0 had: 1
9 1 3 6 7 had: 1
9 5 4 7 0 had: 2
9 1 3 6 9 had: 1
9 1 2 6 0 had: 1
7 1 2 6 3 had: 1
9 1 10 6 4 had: 1
9 1 3 6 2 had: 1
9 1 3 6 9 had: 1
9 4 10 0 4 had: 1
9 2 3 6 8 had: 1
8 4 5 0 4 had: 1
9 5 4 7 0 had: 2
9 3 3 6 8 had: 1
9 5 8 6 8 had: 1
9 5 0 6 3 had: 2
9 2 5 6 0 had: 1
9 2 0 7 0 had: 1
9 5 4 7 0 had: 1
9 2 0 8 0 had: 3
9 4 7 4 3 had: 1
9 2 0 7 0 had: 1
9 2 1 7 3 had: 1
9 2 0 7 0 had: 1
9 2 2 9 0 had: 1
9 4 1 9 3 had: 1
9 2 2 6 0 had: 2
9 5 4 7 0 had: 1
9 9 0 7 0 had: 1
9 2 0 7 0 had: 2
10 4 0 0 3 had: 1
9 5 4 7 0 had: 1
9 2 1 6 0 had: 1
9 2 6 9 3 had: 1
9 2 0 7 0 had: 1
9 2 6 9 2 had: 1
8 0 0 7 0 had: 1
9 5 6 7 1 had: 1
8 0 0 7 0 had: 2
9 5 4 7 0 had: 1
9 2 6 9 4 had: 1
8 0 7 7 0 had: 2
9 5 1 7 1 had: 1
9 2 6 9 4 had: 1
10 2 3 7 0 had: 1
9 5 3 8 1 had: 1
10 2 3 5 4 had: 1
10 2 3 7 0 had: 1
9 1 7 9 8 had: 1
10 2 1 5 4 had: 1
8 8 5 1 7 had: 1
9 5 0 8 1 had: 1
10 8 3 7 7 had: 1
9 5 3 8 1 had: 3
9 5 8 8 1 had: 1
9 5 4 7 0 had: 1
9 0 4 10 0 had: 1
9 5 4 7 0 had: 2
10 4 4 8 4 had: 2
9 5 4 7 0 had: 1
8 9 4 2 0 had: 1
9 5 4 7 0 had: 1
10 2 8 8 4 had: 1
9 5 4 7 0 had: 1
10 6 1 9 0 had: 2
10 2 8 8 4 had: 1
9 5 3 4 0 had: 1

```
10   6   1   9   0   had: 1
9    5   3   4   0   had: 1
10   6   1   4   0   had: 1
9    5   4   7   0   had: 1
9    5   0   4   0   had: 1
10   6   1   9   0   had: 1
9    5   4   7   0   had: 1
10   6   1   9   0   had: 1
9    5   0   4   0   had: 1
9    5   4   7   0   had: 1
6    1   1   5   0   had: 1
9    5   1   7   0   had: 1
9    0   1   7   0   had: 4
9    2   1   1   6   had: 1
9    0   1   7   0   had: 2
9    5   9   4   0   had: 1
10   1   3   5   0   had: 2
10   0   3   5   5   had: 2
9    2   5   1   2   had: 1
10   0   3   5   5   had: 1
9    5   8   1   0   had: 1
9    2   5   1   2   had: 1
9    5   4   7   0   had: 1
9    6   8   10  9   had: 1
9    5   4   7   0   had: 1
9    2   8   10  6   had: 1
9    2   3   5   10  had: 1
9    5   4   1   0   had: 1
9    2   8   10  6   had: 2
9    5   4   7   0   had: 1
8    2   5   0   7   had: 1
9    5   4   7   0   had: 1
9    2   8   3   7   had: 2
9    3   4   10  6   had: 1
9    2   4   2   0   had: 2
9    2   8   3   7   had: 1
8    3   1   0   10  had: 1
9    2   4   2   0   had: 2
9    3   4   10  0   had: 2
9    2   8   3   0   had: 3
9    5   4   7   0   had: 1
9    2   8   3   0   had: 1
9    2   8   7   0   had: 1
7    6   3   10  10  had: 1
9    2   8   7   0   had: 1
9    5   4   7   0   had: 1
9    2   8   7   0   had: 2
9    2   8   10  0   had: 3
9    5   4   7   0   had: 1
9    2   6   10  0   had: 3
9    2   3   10  10  had: 1
9    3   5   10  10  had: 1
9    2   6   1   2   had: 1
9    3   5   10  10  had: 2
6    2   4   10  2   had: 1
10   2   4   10  2   had: 1
10   9   4   10  2   had: 1
9    5   4   7   10  had: 1
9    3   7   6   10  had: 1
9    5   4   7   0   had: 1
6    5   0   5   10  had: 1
9    5   2   7   6   had: 1
9    5   0   5   10  had: 1
9    5   4   7   0   had: 1
9    1   0   5   10  had: 3
9    0   2   0   6   had: 1
9    3   0   9   9   had: 5
9    0   0   5   10  had: 3
9    0   0   5   2   had: 1
9    5   4   7   0   had: 1
10   2   4   7   0   had: 1
9    0   0   5   2   had: 1
9    3   4   7   0   had: 2
9    0   7   7   2   had: 1
9    1   4   7   10  had: 1
10   2   4   7   0   had: 1
9    3   7   7   2   had: 2
9    3   4   7   5   had: 1
9    1   7   7   10  had: 1
9    1   4   7   10  had: 1
9    3   7   9   2   had: 1
9    9   4   7   10  had: 1
9    1   4   7   10  had: 1
9    3   0   7   0   had: 1
8    3   2   7   5   had: 1
9    3   10  2   2   had: 1
9    6   4   1   0   had: 1
9    5   1   7   0   had: 1
9    2   10  7   0   had: 1
9    2   10  0   0   had: 1
9    3   0   7   0   had: 1
9    9   4   7   0   had: 1
9    2   10  0   10  had: 2
8    3   10  7   5   had: 1
9    3   8   4   0   had: 1
9    3   10  4   5   had: 1
9    3   10  4   0   had: 1
10   8   3   8   1   had: 1
9    2   2   0   8   had: 1
9    5   4   7   0   had: 1
9    6   10  4   5   had: 1
9    3   1   5   5   had: 2
9    3   3   4   7   had: 1
9    3   1   5   5   had: 1
9    3   10  4   10  had: 1
9    3   1   5   5   had: 3
9    0   10  5   5   had: 1
9    3   3   4   10  had: 1
9    5   4   4   0   had: 1
9    3   10  4   10  had: 1
9    2   6   1   3   had: 1
9    7   1   5   5   had: 1
9    5   4   7   0   had: 1
9    3   9   8   9   had: 1
```

```
9   1   4   10  0    had: 1
9   4   4   10  0    had: 1
9   5   4   7   0    had: 2
9   3   9   4   10   had: 1
9   1   4   7   0    had: 1
9   1   4   1   0    had: 1
10  5   4   5   3    had: 1
9   1   4   1   0    had: 1
9   3   4   0   0    had: 1
9   7   6   7   3    had: 1
9   3   4   0   0    had: 3
9   4   6   10  0    had: 1
9   5   4   7   0    had: 1
9   0   1   7   1    had: 1
9   4   6   10  0    had: 3
9   5   4   7   0    had: 4
9   3   4   6   0    had: 2
9   3   4   7   4    had: 1
9   5   4   7   0    had: 1
9   4   0   10  0    had: 1
9   4   4   7   0    had: 2
7   2   4   7   4    had: 1
9   2   4   7   8    had: 1
7   2   4   7   4    had: 1
9   2   4   7   8    had: 1
9   2   4   7   0    had: 1
7   2   4   7   10   had: 2
9   2   4   7   0    had: 1
9   2   6   7   1    had: 1
10  4   4   3   0    had: 1
9   6   6   7   1    had: 1
7   4   6   3   0    had: 1
5   2   4   7   0    had: 1
9   6   6   7   7    had: 1
9   5   4   7   0    had: 1
9   0   4   7   0    had: 5
9   5   4   7   0    had: 1
9   2   5   7   0    had: 3
9   0   6   7   0    had: 2
9   6   0   7   8    had: 1
9   2   6   7   1    had: 1
9   5   4   7   0    had: 2
9   2   0   5   2    had: 1
9   6   0   7   2    had: 1
10  0   6   7   0    had: 1
9   7   0   6   0    had: 2
10  0   6   7   9    had: 1
9   2   6   7   7    had: 1
10  6   0   7   0    had: 1
10  1   4   7   1    had: 3
9   2   2   10  7    had: 1
10  1   2   10  1    had: 1
10  1   6   0   1    had: 1
10  1   0   6   1    had: 1
9   1   6   6   9    had: 1
10  1   0   6   1    had: 2
10  6   0   7   9    had: 1
```

```
10  1   0   6   1    had: 2
10  4   4   9   9    had: 1
10  1   0   6   1    had: 1
9   5   4   7   0    had: 1
7   1   6   6   9    had: 1
10  8   0   6   10   had: 1
10  1   6   10  9    had: 1
9   5   4   7   0    had: 1
7   1   6   6   9    had: 1
9   5   4   7   0    had: 1
10  1   6   10  9    had: 1
10  4   8   10  9    had: 1
10  1   5   1   9    had: 1
10  8   0   4   10   had: 1
10  1   5   1   9    had: 2
10  8   0   1   0    had: 2
10  1   5   1   9    had: 1
9   5   4   7   0    had: 1
10  1   5   1   9    had: 1
10  0   0   6   1    had: 2
10  5   10  7   0    had: 1
10  8   0   6   9    had: 1
9   4   4   6   2    had: 1
10  0   4   6   1    had: 1
9   5   4   7   0    had: 1
10  5   9   7   0    had: 1
10  0   4   6   1    had: 3
10  0   4   1   1    had: 1
9   5   9   6   0    had: 1
9   5   4   7   0    had: 4
9   5   4   0   4    had: 1
7   3   5   1   0    had: 1
10  5   4   7   0    had: 1
7   0   4   8   0    had: 1
7   3   4   8   2    had: 1
7   0   4   8   0    had: 1
9   5   4   0   4    had: 1
7   0   4   8   0    had: 5
9   0   4   0   9    had: 3
9   0   4   2   2    had: 3
9   6   4   8   5    had: 1
9   0   4   8   5    had: 3
9   5   4   7   0    had: 1
10  0   6   5   2    had: 1
9   5   4   7   0    had: 2
9   6   4   8   3    had: 1
7   0   6   5   2    had: 2
7   1   9   5   2    had: 1
10  5   4   7   1    had: 1
7   1   9   5   2    had: 1
10  5   4   7   1    had: 2
5   1   8   0   1    had: 1
10  5   4   4   0    had: 1
5   1   9   0   0    had: 1
9   5   4   7   0    had: 4
10  5   4   4   0    had: 1
10  5   4   3   6    had: 3
```

```
9   5   2   7   0   had: 1
5   5   4   0   0   had: 1
10  5   4   10  6   had: 1
9   5   4   0   0   had: 2
9   5   4   7   0   had: 1
9   5   4   0   0   had: 1
7   5   2   3   2   had: 1
10  2   8   0   5   had: 1
9   5   4   7   0   had: 2
10  6   4   10  5   had: 1
9   5   4   7   0   had: 2
10  7   4   10  3   had: 2
9   4   3   0   0   had: 1
7   0   4   7   2   had: 1
10  5   4   6   6   had: 1
7   5   2   6   2   had: 1
7   0   4   7   2   had: 2
7   5   2   6   2   had: 1
9   0   4   6   0   had: 1
9   5   4   7   0   had: 1
10  1   2   6   2   had: 1
10  0   0   6   2   had: 1
10  1   2   6   2   had: 4
10  4   2   3   8   had: 1
10  0   1   6   8   had: 2
10  1   2   5   4   had: 3
8   0   1   7   8   had: 1
10  4   2   5   8   had: 1
10  1   2   0   4   had: 5
10  4   0   5   8   had: 1
8   0   1   7   1   had: 1
7   0   1   0   5   had: 1
10  5   2   10  0   had: 1
8   0   1   7   1   had: 1
10  5   2   10  0   had: 2
9   5   1   7   0   had: 1
10  5   2   9   0   had: 1
5   3   1   10  8   had: 2
9   5   1   7   0   had: 1
8   5   2   9   0   had: 1
8   0   2   0   4   had: 1
9   5   4   7   0   had: 1
6   0   6   0   2   had: 1
8   0   2   0   4   had: 1
9   5   4   7   0   had: 1
8   0   2   0   4   had: 1
8   5   1   0   0   had: 1
10  2   1   7   8   had: 1
10  2   1   7   1   had: 1
7   2   1   7   1   had: 3
9   8   1   3   0   had: 1
8   0   1   8   0   had: 3
9   5   4   7   0   had: 1
9   0   1   8   1   had: 3
9   5   4   7   0   had: 1
9   0   1   8   1   had: 1
8   2   1   5   2   had: 1
9   0   10  8   1   had: 1
9   0   6   8   1   had: 1
9   5   4   7   0   had: 1
10  2   1   4   2   had: 1
9   0   10  8   1   had: 1
9   5   4   7   0   had: 1
9   3   10  7   4   had: 1
10  2   1   4   2   had: 3
9   2   1   6   2   had: 1
10  2   1   0   0   had: 1
9   2   1   6   2   had: 1
10  5   1   6   0   had: 1
10  2   1   0   0   had: 1
8   5   7   4   0   had: 1
10  7   1   6   0   had: 2
6   3   3   0   0   had: 1
10  2   1   6   10  had: 1
10  4   2   7   0   had: 1
10  2   3   2   3   had: 1
10  2   1   6   10  had: 1
10  5   1   6   0   had: 1
9   5   4   7   0   had: 1
10  2   3   2   8   had: 1
10  5   1   6   0   had: 1
10  2   3   2   8   had: 1
10  5   1   6   0   had: 1
9   3   4   7   0   had: 2
8   5   1   6   0   had: 1
8   5   1   3   0   had: 1
8   2   3   0   0   had: 1
9   3   4   7   0   had: 1
5   2   3   0   0   had: 1
9   5   4   7   0   had: 1
7   4   1   6   4   had: 1
9   4   3   7   0   had: 1
5   2   2   0   0   had: 1
9   5   2   0   0   had: 2
9   4   3   7   0   had: 1
9   2   4   10  4   had: 1
9   5   4   7   0   had: 1
9   2   4   10  4   had: 1
9   5   4   7   0   had: 1
9   5   2   3   0   had: 1
9   2   4   10  4   had: 1
9   5   8   3   0   had: 1
9   0   0   3   0   had: 1
9   5   4   7   0   had: 1
9   2   4   7   4   had: 1
9   5   4   7   0   had: 1
9   5   5   2   0   had: 1
9   2   4   7   4   had: 1
8   4   2   6   0   had: 3
9   2   1   2   4   had: 1
9   4   2   3   1   had: 2
9   5   0   3   2   had: 1
8   2   5   6   0   had: 1
9   7   0   3   2   had: 1
```

7 2 5 6 0 had: 1
9 6 2 2 1 had: 2
9 3 10 7 0 had: 1
9 5 4 7 0 had: 1
8 2 5 6 1 had: 1
7 6 5 2 1 had: 1
9 10 2 3 0 had: 1
9 4 0 3 2 had: 2
9 5 4 7 0 had: 1
9 4 0 3 2 had: 1
9 5 1 7 1 had: 1
9 1 0 2 2 had: 1
4 2 5 6 1 had: 1
9 1 0 1 2 had: 1
9 1 1 7 1 had: 1
9 5 1 7 0 had: 1
9 1 1 7 0 had: 2
9 1 0 7 4 had: 1
9 1 1 7 1 had: 1
9 1 1 9 0 had: 1
9 1 1 7 1 had: 1
9 1 4 7 5 had: 2
9 1 1 7 1 had: 2
9 1 0 7 4 had: 1
9 5 1 3 6 had: 1
9 1 0 7 2 had: 1
9 5 1 3 6 had: 1
9 5 4 7 0 had: 1
9 1 0 7 2 had: 2
9 5 4 7 0 had: 1
9 3 0 0 4 had: 1
9 5 4 7 0 had: 1
9 1 0 7 2 had: 2
9 1 0 5 5 had: 1
9 3 2 7 0 had: 1
9 1 0 7 2 had: 1
9 1 10 0 2 had: 1
9 5 4 7 0 had: 1
9 5 1 1 0 had: 3
9 9 1 7 0 had: 1
9 5 1 1 0 had: 1
9 4 3 7 6 had: 1
9 4 3 7 2 had: 2
9 10 8 7 0 had: 1
9 4 4 7 0 had: 1
9 5 3 2 1 had: 3
6 3 1 0 3 had: 1
9 4 3 0 0 had: 1
9 1 5 0 3 had: 2
10 7 8 7 0 had: 1
9 4 6 0 0 had: 1
6 1 8 0 3 had: 1
6 4 8 7 3 had: 1
9 5 4 7 0 had: 2
9 4 5 0 0 had: 1
9 5 4 4 0 had: 1
9 5 4 7 0 had: 3
9 8 4 9 0 had: 1
9 10 1 2 0 had: 1
9 5 4 7 0 had: 1
9 10 1 2 0 had: 1
9 2 6 0 1 had: 1
9 5 4 7 0 had: 1
9 5 1 5 0 had: 2
9 5 4 7 0 had: 1
9 5 8 4 0 had: 1
9 10 1 2 0 had: 1
9 5 4 7 0 had: 1
9 4 1 5 0 had: 1
9 5 8 6 0 had: 1
9 1 1 5 0 had: 4
10 10 5 5 2 had: 1
9 1 1 5 6 had: 2
9 10 3 2 0 had: 1
9 5 4 7 0 had: 1
7 0 10 7 0 had: 2
9 1 1 5 6 had: 1
9 1 1 2 6 had: 1
10 10 8 5 0 had: 1
10 2 1 5 6 had: 1
10 2 1 5 0 had: 1
4 3 3 7 0 had: 1
10 2 9 5 0 had: 1
9 1 1 5 6 had: 3
7 1 1 2 6 had: 1
9 1 1 5 6 had: 4
10 1 8 5 6 had: 1
9 1 1 4 6 had: 1
10 1 8 5 6 had: 1
9 1 1 4 6 had: 1
10 1 3 5 6 had: 2
9 1 4 4 6 had: 1
7 5 1 2 6 had: 2
9 1 4 7 0 had: 1
4 1 1 4 3 had: 1
10 1 1 2 3 had: 3
9 0 1 0 3 had: 1
9 1 3 4 6 had: 1
9 5 4 7 0 had: 1
9 0 1 0 2 had: 1
10 1 3 4 6 had: 1
9 1 1 0 2 had: 1
10 1 3 4 6 had: 3
10 2 1 0 5 had: 1
9 1 1 0 2 had: 1
9 2 1 0 5 had: 2
9 4 1 7 2 had: 1
9 7 1 7 2 had: 1
10 1 1 0 1 had: 1
10 0 1 0 4 had: 1
10 1 1 0 1 had: 1
10 3 10 10 5 had: 1
9 9 0 7 3 had: 1
10 3 10 10 5 had: 1

```
10  1   1   2   1   had: 1
10  4   2   0   1   had: 2
10  1   1   2   1   had: 2
10  1   1   2   9   had: 3
10  7   1   2   9   had: 1
9   5   4   7   0   had: 2
10  4   2   7   8   had: 2
9   5   4   7   0   had: 1
10  7   2   3   3   had: 1
10  4   2   7   9   had: 5
10  4   0   2   9   had: 1
10  4   2   7   9   had: 1
9   4   4   7   0   had: 3
10  4   2   7   9   had: 1
9   0   4   7   0   had: 1
8   0   6   7   0   had: 1
10  4   2   7   9   had: 1
9   0   4   7   0   had: 2
9   5   4   6   0   had: 1
8   0   6   9   0   had: 1
10  4   2   7   5   had: 1
8   0   6   9   0   had: 1
9   5   4   7   0   had: 4
10  4   2   7   5   had: 1
10  8   2   2   3   had: 1
7   4   7   5   3   had: 1
9   4   4   7   0   had: 1
10  1   7   7   8   had: 2
10  4   10  0   9   had: 1
9   5   4   6   0   had: 1
10  4   2   2   3   had: 1
9   5   4   6   0   had: 1
10  2   4   2   0   had: 1
10  4   2   7   3   had: 3
10  4   9   7   3   had: 1
9   5   4   7   0   had: 3
9   4   6   7   0   had: 3
9   5   4   7   0   had: 1
8   2   1   0   1   had: 1
6   5   7   0   1   had: 1
7   5   1   6   0   had: 1
10  4   1   0   3   had: 1
8   2   2   0   6   had: 1
10  5   6   7   0   had: 1
10  4   1   2   3   had: 1
10  1   8   7   0   had: 1
10  4   1   2   3   had: 2
10  1   8   7   0   had: 1
7   5   10  0   1   had: 1
9   10  5   0   0   had: 1
9   5   10  0   0   had: 1
8   2   3   0   4   had: 1
10  1   8   7   2   had: 2
10  3   8   7   1   had: 1
10  3   3   7   1   had: 1
10  3   3   1   1   had: 1
10  4   5   0   9   had: 1
10  3   3   1   1   had: 1
10  1   8   2   1   had: 2
10  4   5   0   2   had: 1
9   5   0   10  1   had: 1
10  4   1   0   2   had: 2
10  1   8   2   1   had: 2
10  4   1   0   2   had: 1
10  3   1   7   8   had: 1
10  4   1   0   2   had: 1
10  5   6   4   5   had: 1
10  4   5   0   2   had: 2
9   7   2   7   0   had: 1
9   5   10  10  4   had: 1
9   5   4   7   0   had: 1
10  2   6   1   9   had: 1
9   4   4   1   0   had: 1
10  4   1   8   8   had: 1
10  4   1   10  4   had: 1
9   5   4   7   0   had: 1
10  4   1   10  4   had: 3
9   5   4   7   0   had: 1
10  4   4   1   7   had: 1
10  4   1   3   2   had: 2
10  5   4   10  4   had: 1
10  2   4   7   7   had: 1
9   5   4   7   0   had: 1
9   4   4   2   0   had: 1
10  5   1   10  5   had: 1
10  4   1   3   2   had: 2
9   5   4   7   0   had: 1
10  4   4   9   6   had: 1
9   1   2   2   0   had: 1
10  4   4   9   6   had: 1
9   5   4   9   3   had: 1
9   1   2   2   0   had: 1
9   5   4   2   0   had: 1
9   1   2   2   0   had: 2
9   5   4   7   0   had: 1
9   2   7   2   3   had: 1
9   5   4   8   0   had: 1
8   7   1   2   3   had: 1
9   3   2   2   0   had: 2
9   2   1   6   3   had: 2
9   5   4   7   0   had: 1
9   3   2   4   0   had: 1
9   2   1   9   3   had: 1
9   5   1   8   0   had: 1
8   2   1   2   3   had: 1
9   3   2   9   0   had: 2
9   5   4   7   0   had: 1
10  3   1   1   3   had: 2
9   5   4   7   0   had: 1
9   3   10  5   0   had: 1
9   3   6   5   0   had: 1
9   0   4   7   0   had: 1
9   3   1   8   8   had: 1
9   0   4   7   0   had: 1
```

```
10  9   4   7   0   had: 1
10  5   7   0   1   had: 1
9   0   6   7   0   had: 1
9   5   4   7   0   had: 2
9   1   10  1   3   had: 1
9   0   6   7   9   had: 1
10  0   7   0   9   had: 1
10  0   7   7   9   had: 1
10  0   7   0   9   had: 2
8   3   1   0   9   had: 1
10  4   0   0   9   had: 3
10  0   0   0   1   had: 3
9   0   4   7   0   had: 1
10  0   0   0   4   had: 1
9   0   4   7   0   had: 1
9   5   4   7   0   had: 1
10  0   0   7   7   had: 1
10  0   9   10  1   had: 1
10  0   9   10  6   had: 1
9   0   4   9   0   had: 1
10  0   9   7   1   had: 1
10  0   0   0   2   had: 1
10  0   9   10  6   had: 1
10  0   9   10  1   had: 1
10  3   1   10  2   had: 1
10  0   8   8   2   had: 1
10  0   9   10  1   had: 1
10  0   4   2   8   had: 1
10  0   9   9   1   had: 1
10  7   9   10  6   had: 1
10  3   9   4   10  had: 1
10  0   4   9   8   had: 1
10  0   4   2   8   had: 1
9   2   0   7   8   had: 6
10  0   3   7   6   had: 1
10  3   7   7   8   had: 1
10  0   4   7   1   had: 1
9   5   4   7   0   had: 1
9   7   10  6   5   had: 2
9   2   2   9   8   had: 1
7   0   4   4   10  had: 1
9   2   1   9   8   had: 1
10  8   4   3   1   had: 2
6   2   1   9   8   had: 1
5   8   4   7   1   had: 1
9   5   4   7   0   had: 2
10  8   2   3   1   had: 1
9   5   4   7   0   had: 1
10  10  9   3   0   had: 2
10  8   2   3   1   had: 1
9   5   4   7   0   had: 1
10  7   2   3   1   had: 1
7   3   1   10  10  had: 1
9   5   4   7   5   had: 2
10  7   2   3   1   had: 2
7   0   5   3   1   had: 1
9   0   7   3   2   had: 1
5   0   7   3   2   had: 1
7   0   5   7   1   had: 1
10  5   9   7   0   had: 1
10  5   1   7   0   had: 2
10  0   1   9   4   had: 1
9   5   1   7   0   had: 1
10  0   1   9   4   had: 1
9   5   4   7   0   had: 1
9   5   1   7   0   had: 1
4   5   10  7   0   had: 1
10  5   1   7   3   had: 1
10  0   1   9   3   had: 1
10  5   4   9   0   had: 1
10  0   1   9   3   had: 2
10  8   1   7   3   had: 1
10  9   4   7   7   had: 1
10  8   1   7   6   had: 1
10  8   1   4   6   had: 1
10  5   10  7   0   had: 1
9   5   4   7   0   had: 1
9   7   4   9   2   had: 2
9   5   4   7   0   had: 1
10  2   1   7   2   had: 1
10  8   8   4   0   had: 1
10  2   3   2   7   had: 1
10  1   4   7   0   had: 1
10  5   1   7   2   had: 1
10  5   10  9   0   had: 1
10  7   0   7   3   had: 1
10  5   10  9   0   had: 1
10  1   4   4   2   had: 1
9   2   3   2   7   had: 1
10  1   4   4   2   had: 1
10  5   0   6   0   had: 1
10  5   6   9   0   had: 1
10  5   0   6   5   had: 1
9   5   1   7   0   had: 1
10  7   1   6   0   had: 1
9   6   1   7   0   had: 1
10  5   1   8   5   had: 1
9   2   0   7   2   had: 1
10  5   0   7   2   had: 2
10  5   0   10  1   had: 1
10  5   0   7   2   had: 1
9   2   0   7   2   had: 1
7   0   10  6   0   had: 1
9   0   0   7   2   had: 1
9   5   4   7   0   had: 1
10  9   5   6   0   had: 1
10  0   4   1   0   had: 1
9   0   0   7   2   had: 2
10  0   4   7   0   had: 1
10  0   2   1   0   had: 1
9   0   0   7   2   had: 1
10  7   0   9   0   had: 1
10  0   2   1   0   had: 3
10  0   4   7   3   had: 1
```

MGG Optimizer

This is the data which was obtained from running the optimization function using the MGG genetic algorithm. Since the alleles are by nature bounded by the values 10 and 0, I was forced to change such an allele to fit these procrustican conditions. Nevertheless, some basic trends could certainly be observed. Nearly all of these successful organisms had very high values for aggression and very low deception, a trait experienced with the previous random-test-driver optimizer function. Deceptive was most often low, but grudges and adaptive could be either high or low – both varieties were successful. The most successful drivers of the set generally had low grudges and high adaptive. This corresponds extremely well with the previous data found – high aggression and low defensive were essential, while low deceptive and high grudges were generally good and adaptive could be either high or low.

Also keep in mind that, since the MGG picks two different parent organisms each round, the number of consecutive-round wins is artificially low since to win two consecutive round an organism must be chosen as a parent both rounds.

10.0	0.0	0.0	0.0	10.0	had: 2
7.546888629587019	0.0	0.0	10.0	10.0	had: 1
10.0	0.0	10.0	9.995431376961184	0.0	had: 3
10.0	0.0	0.0	0.3773384968618228	0.0945232994519718	had: 1
9.999487693207879	0.0	9.997041723097134	10.0	0.0	had: 1
10.0	0.0	6.1346328832718955	10.0	10.0	had: 1
10.0	0.0	10.0	0.0	3.3184774586701735	had: 1
10.0	3.342141082972129	10.0	10.0	10.0	had: 2
6.787500452153182	0.0	0.0	1.3761717660644839	0.0	had: 1
9.889310683725432	0.0	10.0	10.0	10.0	had: 1
1.1893410595682028	3.7693928555147242	0.0	10.0	8.832567267595296	had: 1
10.0	0.0	9.496296051731248	9.969077037393857	0.0	had: 1
10.0	0.0	0.0	0.0	0.0	had: 1
10.0	0.0	10.0	0.0	10.0	had: 1
10.0	0.0	1.8412663350993448	10.0	10.0	had: 1
10.0	0.0	10.0	10.0	0.0	had: 1
10.0	0.0	0.0	0.0	0.0	had: 1
10.0	0.0	0.0	0.0	10.0	had: 1
10.0	0.0	0.0	0.0	0.0	had: 1
10.0	0.0	10.0	9.23504202067637	1.4897099825133044	had: 2
8.850071233953985	0.027976264411976787	10.0	3.40627643016804	10.0	had: 1

10.0 0.0 4.9082623889855785 10.0 0.0 had: 1
10.0 0.0 0.0 0.0 0.0 had: 1
10.0 0.24112858106992535 3.7099260499291793 10.0 0.0 had: 1
10.0 0.0 0.0 0.0 0.0 had: 1
10.0 0.24112858106992535 3.7099260499291793 10.0 0.0 had: 1
10.0 0.0 10.0 10.0 5.650769547880075 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 0.31242438061712685 0.0 had: 1
10.0 0.0 10.0 0.0 0.0 had: 1
10.0 0.0 10.0 9.40375915909083 0.0 had: 1
10.0 0.0 10.0 5.011608124313838 9.80297170747049 had: 1
10.0 2.093803517132182 8.316351880945971 0.19533877191232385 0.0 had: 1
10.0 0.5767011420306627 0.0 10.0 0.0 had: 1
10.0 2.733322078753231 1.4064119115117464 0.0 1.2562161153307343 had: 1
10.0 0.0 0.0 10.0 0.0 had: 2
10.0 0.0 0.0 1.3042251025978029 0.0 had: 1
10.0 0.0 0.0 0.0 2.67821257131967 had: 1
8.378018070031668 0.0 0.0 1.2570485784285335 0.0 had: 1
10.0 0.0 10.0 0.0 0.0 had: 1
10.0 0.5767011420306627 0.0 10.0 0.0 had: 1
10.0 10.0 0.0 8.599270591205322 0.0 had: 1
10.0 0.0 10.0 8.551059438230102 0.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
7.932641512823535 0.0 0.2917889904292832 0.0 0.0 had: 1
10.0 0.0 0.0 7.168601907151391 0.0 had: 1
10.0 0.0 10.0 0.0 10.0 had: 1
10.0 0.0 0.0 5.183633442335052 0.0 had: 1
10.0 0.0 7.443352923926634 0.4459249520689621 0.0 had: 1
10.0 0.0 10.0 10.0 10.0 had: 1
10.0 10.0 3.82982562665784 8.885879359883122 9.31183658665919 had: 1
9.199515731872582 0.0 0.0 8.599337544737052 7.508959699084789 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 7.960821426013512 5.025201344694233 had: 1
10.0 0.9855343506751091 0.0 0.0 10.0 had: 1
10.0 0.0 0.0 0.0 0.9224952007728503 had: 1
10.0 0.0 0.0 10.0 3.639407331231155 had: 1
10.0 0.0 10.0 0.0 0.0 had: 1
9.619726862329683 0.0 0.0 10.0 10.0 had: 1
10.0 0.0 0.0 0.0 0.9224952007728503 had: 1
10.0 0.0 0.0 9.743314641474276 1.7218813908349537 had: 1
10.0 0.0 10.0 3.8465292468973846 10.0 had: 2
10.0 0.7049261070143881 9.247486030335592 5.768739419300666 9.319176366733357 had: 1
10.0 10.0 10.0 7.764196023488478 0.0 had: 1
10.0 0.0 5.018419964574261 10.0 0.0 had: 1

9.619726862329683 0.0 0.0 10.0 10.0 had: 2
10.0 0.0 10.0 0.09880944925697538 10.0 had: 1
10.0 0.0 0.0 10.0 2.970697025231851 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
9.95068504068914 0.7695131370100794 0.0 3.7902582330509986
3.69092636826678 had: 1
7.452835383295156 0.0 1.16745212929864 10.0 0.0 had: 1
10.0 0.6471049353163307 0.6536676276361665 10.0 2.144625129912362 had: 1
10.0 0.0 9.968015844974998 10.0 0.0 had: 2
9.619726862329683 0.0 0.0 10.0 10.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 10.0 9.942295398717825 0.0 had: 1
10.0 0.0 0.0 10.0 10.0 had: 1
10.0 0.0 10.0 10.0 10.0 had: 1
9.871736504445481 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 0.0 10.0 had: 1
10.0 0.0 0.0 10.0 10.0 had: 1
10.0 0.0 0.0 0.0 10.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 3.1603096802103234 0.0 10.0 10.0 had: 1
10.0 0.0 0.0 0.0 5.864457235808147 had: 1
10.0 0.0 0.0 10.0 10.0 had: 1
10.0 0.2678333252925917 0.7656077653505169 0.0 10.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 2
10.0 0.0 10.0 9.998079973079284 0.0 had: 1
10.0 0.041445479222984555 0.0 0.0 2.6533051773258944 had: 2
10.0 0.0 10.0 0.0 0.0 had: 1
10.0 0.0 7.086369271579236 9.300727948659263 0.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 10.0 0.0 0.0 had: 1
10.0 0.0 10.0 9.998079973079284 0.0 had: 1
10.0 0.041445479222984555 0.0 0.0 2.6533051773258944 had: 2
8.85514836224938 0.0 9.364038315831044 10.0 10.0 had: 1
10.0 0.0 0.0 10.0 0.3226232954871304 had: 1
10.0 1.1852984971620277 10.0 4.890271518646003 0.0 had: 1
10.0 0.7300002355865481 10.0 9.97480499019123 0.0 had: 1
9.267574460892678 0.0 0.0 2.8100493848700827 0.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.041445479222984555 0.0 0.0 2.6533051773258944 had: 1
10.0 0.0 0.0 4.463356547418423 2.8969588606525827 had: 1
10.0 0.0 10.0 0.0 0.0 had: 1
10.0 1.1928446413430527 10.0 0.0 4.762115156355956 had: 1
10.0 0.0 10.0 4.708215339894488 0.0 had: 1
10.0 0.0 10.0 0.0 10.0 had: 1
10.0 0.41130074746403356 0.2721386657087367 0.0 2.1042808044076415
had: 1

4.392983001893211 0.24770853661087922 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 0.0 0.0 had: 1
9.63954309579806 0.0 10.0 0.2642892285716883 5.058230442772162 had: 1
10.0 0.0 10.0 0.0 0.0 had: 1
9.925290699924854 0.0 8.09775658905121 0.0 10.0 had: 1
10.0 0.0 4.778067851495239 1.8333871767404637 6.21608650384658 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 2.1923064365593117 0.0 had: 2
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 0.0 10.0 had: 1
9.29552283488711 10.0 10.0 0.972056925050062 2.9792415101490017 had: 1
10.0 0.0 0.0 0.0 10.0 had: 1
10.0 0.0 0.0 0.0 0.37263861466667236 had: 1
9.267574460892678 0.0 0.0 2.8100493848700827 0.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 8.709938244461059 0.0 0.0 had: 1
10.0 0.0 0.8568594444236391 10.0 0.0 had: 1
10.0 0.0 10.0 10.0 10.0 had: 1
10.0 0.0 10.0 7.997198944376201 10.0 had: 1
9.0674037931694 0.0 0.0 0.0 0.0 had: 4
10.0 0.0 0.0 1.6700551733973068 10.0 had: 1
10.0 0.0 0.0 10.0 10.0 had: 1
10.0 0.0 0.0 0.0 10.0 had: 1
7.834919225448732 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 0.15309296085024862 10.0 had: 1
10.0 0.0 0.0 2.1923064365593117 0.0 had: 1
10.0 0.0 0.0 0.231218991226274 9.99369980078896 had: 1
10.0 0.0 0.0 10.0 10.0 had: 1
10.0 0.15153369622938026 0.03439557374743668 0.0
7.320078842182767 had: 1
10.0 0.0 0.0 10.0 0.0 had: 1
10.0 0.010362021635464277 0 0.38448500308896644
9.589873026078825 had: 2
10.0 0.0 1.2856806125550981 0.0 8.884516630400897 had: 1
9.18292280451357 0.0 10.0 0.0 10.0 had: 1
10.0 0.0 0.0 0.5415986975777702 10.0 had: 3
10.0 2.27324198431521 0.3364539314145112 0.6644933714782687 10.0 had: 1
7.8918822334477685 0.0 0.0 0.0 0.0 had: 1
10.0 0.0 0.0 0.0 10.0 had: 1
10.0 1.7768511375045783 0.0 10.0 10.0 had: 1
10.0 2.5942346140707575 0.0 10.0 0.0 had: 1
10.0 0.0 0.0 0.0 9.354193266029608 had: 1
10.0 0.0 3.6720549695766307 3.0465351663277955 10.0 had: 1
10.0 2.4174167855084643 3.1833455961412778 10.0 10.0 had: 1
10.0 0.3000084630660754 0.0 10.0 10.0 had: 1

10.0 0.0 0.0 0.0 10.0 had: 2
9.848171888737117 0.1291904438045229 10.0 0.0 9.711771565890201 had: 1
10.0 0.5383469132022746 0.0 0.0 9.782589749340909 had: 1
10.0 0.0 10.0 10.0 0.0 had: 1
9.848171888737117 0.1291904438045229 10.0 0.0 9.711771565890201 had: 1
10.0 0.0 10.0 10.0 0.0 had: 3
9.985363853791013 0.0 9.736200586268174 0.0018497439314049916
8.55363922050764 had: 1
10.0 0.3000084630660754 0.0 10.0 10.0 had: 1
10.0 0.0 0.0 0.0 9.354193266029608 had: 1
10.0 0.0 1.3074059473084159 10.0 10.0 had: 1

MGG Continuous

This particular set of data has been edited for brevity, since it was used as a test to determine whether the MGG produced a sufficiently high-quality result to warrant further testing. The factor analyzed here was how many times the driver who had won a previous test would win again. The idea that a winner will (mostly) continue to win is extremely important in showing that the drivers do not just win by random chance. In this case, there were 8973 races and 2270 repeat winners. If the races were random, the number of repeat winners expected would be 2/11 because there are two parents which are run against 9 randomly generated drivers. The actual ratio was .253, while the random ratio would be .182. There is a measurable difference, but compared to the normal genetic algorithm's record of having the winning driver continue to win 4 races this does not even come close in terms of continuity.

The most likely explanation is that the winning driver will have to race against a completely different set of drivers each round, destroying any advantage it had against the set it currently raced against.

```
0 0   had: 1
0 new had: 4
1 3   had: 1
0 new had: 6
1 7   had: 1
0 new had: 2
1 4   had: 1
1 1   had: 1
1 4   had: 1
0 new had: 5
1 6   had: 1
0 new had: 3
1 7   had: 1
1 6   had: 1
0 new had: 12
1 1   had: 1
1 2   had: 1
1 9   had: 1
0 new had: 1
1 4   had: 1
1 9   had: 1
0 new had: 2
1 5   had: 1
1 6   had: 1
1 2   had: 1
0 new had: 3
1 0   had: 1
0 new had: 5
1 10  had: 1
0 new had: 7
1 3   had: 1
0 new had: 3
1 5   had: 1
0 new had: 3
1 1   had: 1
0 new had: 1
1 4   had: 1
1 9   had: 1
0 new had: 1
1 4   had: 1
0 new had: 6
1 7   had: 1
0 new had: 1
1 10  had: 1
0 new had: 2
1 9   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 7   had: 1
0 new had: 1
1 8   had: 1
0 new had: 3
1 2   had: 1
0 new had: 2
1 10  had: 1
0 new had: 1
1 4   had: 1
0 new had: 13
1 1   had: 1
0 new had: 1
1 0   had: 1
1 5   had: 1
0 new had: 2
1 4   had: 1
1 1   had: 1
0 new had: 5
1 7   had: 1
0 new had: 2
1 0   had: 1
0 new had: 1
1 1   had: 1
0 new had: 2
1 4   had: 1
1 1   had: 1
1 0   had: 1
0 new had: 4
1 7   had: 1
0 new had: 1
1 0   had: 1
0 new had: 1
1 10  had: 1
0 new had: 1
1 9   had: 1
0 new had: 3
1 2   had: 1
1 7   had: 1
0 new had: 6
1 1   had: 1
0 new had: 1
1 9   had: 1
1 8   had: 1
0 new had: 2
1 6   had: 1
1 8   had: 1
1 6   had: 1
1 8   had: 1
0 new had: 6
1 8   had: 1
0 new had: 3
1 9   had: 2
0 new had: 2
1 0   had: 1
0 new had: 5
1 10  had: 1
0 new had: 6
1 3   had: 1
0 new had: 5
1 2   had: 1
0 new had: 4
1 4   had: 1
0 new had: 1
1 10  had: 1
0 new had: 4
1 5   had: 1
1 3   had: 1
1 6   had: 1
0 new had: 2
1 6   had: 1
0 new had: 3
1 10  had: 1
0 new had: 7
1 0   had: 2
1 4   had: 1
0 new had: 2
1 1   had: 1
0 new had: 1
1 10  had: 1
0 new had: 4
1 1   had: 1
0 new had: 1
1 4   had: 1
1 8   had: 1
1 3   had: 1
0 new had: 7
1 3   had: 1
0 new had: 3
1 3   had: 1
0 new had: 6
1 5   had: 2
0 new had: 3
1 5   had: 1
1 1   had: 1
1 3   had: 1
0 new had: 6
1 1   had: 2
0 new had: 3
1 0   had: 1
0 new had: 1
1 3   had: 1
0 new had: 6
1 8   had: 1
0 new had: 7
1 4   had: 1
0 new had: 1
1 0   had: 1
0 new had: 4
1 2   had: 1
0 new had: 4
1 8   had: 1
0 new had: 3
1 0   had: 1
1 3   had: 1
0 new had: 2
1 6   had: 1
0 new had: 1
1 6   had: 1
0 new had: 4
1 1   had: 1
0 new had: 1
1 2   had: 1
0 new had: 2
1 4   had: 1
1 9   had: 1
0 new had: 3
1 6   had: 1
0 new had: 1
1 10  had: 1
1 4   had: 1
0 new had: 4
1 8   had: 1
0 new had: 9
1 8   had: 1
0 new had: 5
1 7   had: 1
0 new had: 4
1 4   had: 1
0 new had: 2
1 0   had: 1
0 new had: 6
1 6   had: 1
0 new had: 3
1 1   had: 1
0 new had: 7
1 4   had: 1
1 7   had: 1
0 new had: 2
1 5   had: 2
0 new had: 4
1 6   had: 1
1 8   had: 1
0 new had: 2
1 6   had: 1
0 new had: 7
1 4   had: 1
0 new had: 1
1 5   had: 1
0 new had: 3
1 4   had: 1
0 new had: 4
1 4   had: 1
0 new had: 6
1 7   had: 1
0 new had: 6
1 4   had: 1
0 new had: 17
1 7   had: 1
0 new had: 5
1 8   had: 1
1 4   had: 1
0 new had: 4
1 10  had: 1
1 8   had: 1
0 new had: 6
1 0   had: 1
0 new had: 3
1 10  had: 1
1 1   had: 1
```

```
0 new had: 4
1 10  had: 2
0 new had: 4
1 5   had: 1
0 new had: 4
1 0   had: 1
1 7   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 1   had: 1
1 0   had: 1
1 9   had: 1
0 new had: 7
1 4   had: 1
0 new had: 8
1 1   had: 1
0 new had: 5
1 7   had: 1
1 5   had: 1
0 new had: 2
1 10  had: 1
0 new had: 1
1 3   had: 1
0 new had: 2
1 1   had: 1
1 6   had: 1
1 9   had: 1
0 new had: 2
1 5   had: 1
1 3   had: 1
0 new had: 5
1 2   had: 1
0 new had: 3
1 3   had: 1
0 new had: 1
1 3   had: 1
1 8   had: 1
0 new had: 3
1 8   had: 1
0 new had: 2
1 5   had: 1
0 new had: 10
1 10  had: 1
1 6   had: 1
0 new had: 1
1 6   had: 1
0 new had: 3
1 5   had: 1
1 3   had: 1
0 new had: 1
1 10  had: 1
0 new had: 3
1 4   had: 1
0 new had: 1
1 4   had: 1
0 new had: 8
1 3   had: 1
0 new had: 2
1 6   had: 1
0 new had: 6
1 5   had: 1
0 new had: 1
1 6   had: 1
1 2   had: 1
0 new had: 2
1 10  had: 1
0 new had: 6
1 1   had: 1
1 0   had: 1
0 new had: 6
1 10  had: 1
0 new had: 1
1 2   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 6   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 4   had: 1
0 new had: 1
1 5   had: 1
0 new had: 2
1 1   had: 1
0 new had: 13
1 4   had: 1
1 8   had: 1
0 new had: 6
1 3   had: 1
0 new had: 1
1 1   had: 1
1 5   had: 1
0 new had: 8
1 0   had: 1
0 new had: 1
1 0   had: 1
0 new had: 1
1 5   had: 1
1 2   had: 1
1 3   had: 1
0 new had: 3
1 6   had: 1
0 new had: 4
1 1   had: 2
0 new had: 1
1 0   had: 1
0 new had: 3
1 7   had: 1
0 new had: 4
1 0   had: 1
0 new had: 2
1 0   had: 1
1 8   had: 1
0 new had: 10
1 2   had: 1
0 new had: 1
1 2   had: 1
0 new had: 5
1 4   had: 1
0 new had: 1
1 0   had: 1
1 8   had: 1
0 new had: 3
1 6   had: 1
0 new had: 7
1 2   had: 1
0 new had: 9
1 10  had: 1
0 new had: 2
1 8   had: 1
0 new had: 4
1 0   had: 1
1 2   had: 1
0 new had: 1
1 4   had: 1
0 new had: 1
1 10  had: 1
0 new had: 1
1 0   had: 1
0 new had: 4
1 10  had: 1
0 new had: 9
1 10  had: 1
0 new had: 2
1 8   had: 1
0 new had: 4
1 4   had: 1
0 new had: 9
1 10  had: 1
0 new had: 3
1 0   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 0   had: 1
1 4   had: 1
1 2   had: 1
0 new had: 3
1 6   had: 1
1 0   had: 1
0 new had: 4
1 8   had: 1
0 new had: 2
1 3   had: 1
0 new had: 3
1 1   had: 1
0 new had: 9
1 10  had: 1
1 7   had: 1
0 new had: 8
1 10  had: 1
0 new had: 1
1 4   had: 1
0 new had: 2
1 7   had: 1
0 new had: 3
1 10  had: 1
1 2   had: 1
0 new had: 4
1 0   had: 1
1 3   had: 1
0 new had: 1
1 6   had: 1
0 new had: 5
1 1   had: 1
0 new had: 1
1 10  had: 1
0 new had: 3
1 10  had: 1
1 2   had: 1
0 new had: 9
1 10  had: 1
0 new had: 3
1 3   had: 1
0 new had: 1
1 0   had: 1
0 new had: 4
1 2   had: 1
0 new had: 4
1 6   had: 1
0 new had: 4
1 4   had: 1
1 10  had: 1
0 new had: 3
1 8   had: 1
0 new had: 8
1 0   had: 1
1 10  had: 1
0 new had: 6
1 4   had: 1
0 new had: 1
1 0   had: 1
0 new had: 3
1 3   had: 1
0 new had: 10
1 0   had: 1
0 new had: 7
1 5   had: 1
0 new had: 14
1 3   had: 1
0 new had: 1
1 10  had: 1
0 new had: 1
1 4   had: 1
0 new had: 2
1 8   had: 1
```

```
1 7   had: 1
0 new had: 1
1 10  had: 1
0 new had: 7
1 7   had: 1
0 new had: 4
1 0   had: 1
1 3   had: 1
1 5   had: 1
1 1   had: 1
0 new had: 1
1 8   had: 1
0 new had: 1
1 1   had: 1
0 new had: 6
1 4   had: 1
1 7   had: 1
0 new had: 1
1 1   had: 1
0 new had: 2
1 10  had: 1
0 new had: 1
1 5   had: 1
0 new had: 2
1 10  had: 1
0 new had: 10
1 7   had: 1
0 new had: 2
1 4   had: 1
1 1   had: 1
0 new had: 18
1 6   had: 1
0 new had: 2
1 10  had: 1
0 new had: 7
1 8   had: 1
0 new had: 5
1 5   had: 1
1 6   had: 1
1 3   had: 1
1 8   had: 1
0 new had: 6
1 6   had: 1
0 new had: 1
1 5   had: 1
0 new had: 3
1 6   had: 1
0 new had: 3
1 5   had: 1
1 4   had: 1
0 new had: 7
1 9   had: 1
0 new had: 1
1 6   had: 1
0 new had: 3
1 10  had: 1
1 4   had: 1
0 new had: 2
1 4   had: 1
1 3   had: 1
0 new had: 2
1 4   had: 1
0 new had: 1
1 6   had: 1
0 new had: 2
1 1   had: 1
0 new had: 2
1 0   had: 1
0 new had: 2
1 9   had: 1
0 new had: 1
1 2   had: 1
0 new had: 5
1 2   had: 1
0 new had: 5
1 4   had: 1
0 new had: 2
1 6   had: 1
0 new had: 4
1 9   had: 1
0 new had: 1
1 6   had: 1
0 new had: 3
1 0   had: 1
0 new had: 1
1 8   had: 1
0 new had: 1
1 9   had: 2
0 new had: 7
1 3   had: 1
0 new had: 1
1 10  had: 1
0 new had: 2
1 1   had: 1
0 new had: 5
1 9   had: 1
0 new had: 2
1 10  had: 1
0 new had: 2
1 5   had: 1
1 4   had: 1
0 new had: 6
1 10  had: 1
0 new had: 1
1 7   had: 1
0 new had: 1
1 4   had: 1
0 new had: 6
1 5   had: 1
0 new had: 7
1 6   had: 1
0 new had: 1
1 0   had: 1
0 new had: 5
1 5   had: 1
0 new had: 7
1 6   had: 1
0 new had: 9
1 9   had: 1
1 2   had: 1
0 new had: 5
1 8   had: 1
0 new had: 1
1 7   had: 1
0 new had: 1
1 4   had: 1
0 new had: 3
1 7   had: 1
0 new had: 4
1 9   had: 1
0 new had: 3
1 9   had: 1
1 8   had: 1
0 new had: 7
1 7   had: 1
0 new had: 1
1 8   had: 1
0 new had: 2
1 7   had: 1
0 new had: 7
1 5   had: 2
1 1   had: 1
0 new had: 1
1 2   had: 1
0 new had: 3
1 10  had: 1
0 new had: 9
1 2   had: 1
0 new had: 4
1 7   had: 1
0 new had: 4
1 5   had: 1
0 new had: 4
1 0   had: 1
1 3   had: 1
0 new had: 1
1 8   had: 2
0 new had: 1
1 1   had: 1
0 new had: 2
1 0   had: 1
0 new had: 1
1 2   had: 1
0 new had: 2
1 10  had: 1
0 new had: 8
1 6   had: 1
0 new had: 4
1 3   had: 1
0 new had: 10
1 3   had: 1
0 new had: 1
1 6   had: 1
0 new had: 14
1 6   had: 1
0 new had: 1
1 4   had: 1
0 new had: 3
1 6   had: 1
0 new had: 7
1 1   had: 1
0 new had: 2
1 3   had: 1
1 6   had: 1
0 new had: 7
1 4   had: 1
0 new had: 7
1 5   had: 1
1 2   had: 1
1 3   had: 1
0 new had: 4
1 2   had: 1
0 new had: 2
1 9   had: 1
0 new had: 1
1 6   had: 1
0 new had: 1
1 5   had: 1
0 new had: 5
1 6   had: 1
0 new had: 13
1 7   had: 1
0 new had: 14
1 10  had: 1
0 new had: 16
1 7   had: 1
0 new had: 4
1 4   had: 1
0 new had: 3
1 10  had: 1
0 new had: 3
1 4   had: 1
0 new had: 2
1 9   had: 1
0 new had: 1
1 5   had: 1
0 new had: 5
1 6   had: 1
0 new had: 8
1 7   had: 1
0 new had: 1
1 0   had: 1
0 new had: 4
1 3   had: 1
0 new had: 1
1 0   had: 1
0 new had: 6
1 7   had: 1
```

```
0 new had: 1
1 4   had: 1
0 new had: 3
1 1   had: 1
0 new had: 1
1 9   had: 1
1 3   had: 1
0 new had: 3
1 6   had: 1
1 3   had: 1
1 0   had: 1
0 new had: 3
1 0   had: 1
0 new had: 8
1 4   had: 1
0 new had: 1
1 2   had: 1
1 3   had: 1
1 10  had: 1
0 new had: 3
1 0   had: 1
0 new had: 6
1 4   had: 1
0 new had: 1
1 10  had: 1
1 3   had: 1
0 new had: 2
1 8   had: 2
0 new had: 2
1 9   had: 1
1 3   had: 1
0 new had: 14
1 10  had: 1
0 new had: 2
1 10  had: 1
0 new had: 4
1 1   had: 1
0 new had: 1
1 6   had: 1
0 new had: 8
1 6   had: 1
1 0   had: 1
0 new had: 4
1 10  had: 1
0 new had: 24
1 1   had: 1
0 new had: 11
1 4   had: 1
0 new had: 5
1 3   had: 1
0 new had: 3
1 0   had: 1
1 10  had: 1
0 new had: 2
1 6   had: 1
0 new had: 3
1 3   had: 1
0 new had: 2
1 8   had: 1
1 4   had: 1
0 new had: 5
1 6   had: 1
0 new had: 1
1 10  had: 1
0 new had: 1
1 1   had: 1
0 new had: 3
1 3   had: 1
1 5   had: 1
0 new had: 9
1 7   had: 1
1 4   had: 1
0 new had: 3
1 1   had: 1
0 new had: 4
1 9   had: 1
0 new had: 4
1 1   had: 1
0 new had: 2
1 10  had: 1
1 3   had: 1
0 new had: 4
1 3   had: 1
0 new had: 7
1 1   had: 1
1 4   had: 1
0 new had: 2
1 5   had: 1
1 6   had: 1
0 new had: 9
1 7   had: 1
0 new had: 5
1 8   had: 1
0 new had: 5
1 9   had: 1
0 new had: 1
1 4   had: 1
1 9   had: 1
0 new had: 2
1 3   had: 1
0 new had: 2
1 2   had: 2
0 new had: 1
1 7   had: 1
0 new had: 1
1 3   had: 1
1 7   had: 1
0 new had: 3
1 4   had: 1
0 new had: 2
1 2   had: 1
0 new had: 14
1 5   had: 1
0 new had: 5
1 9   had: 1
0 new had: 1
1 10  had: 1
0 new had: 7
1 1   had: 1
1 10  had: 1
0 new had: 3
1 3   had: 2
0 new had: 12
1 7   had: 1
0 new had: 9
1 8   had: 1
0 new had: 8
1 9   had: 1
0 new had: 3
1 6   had: 1
0 new had: 1
1 7   had: 1
0 new had: 2
1 10  had: 1
0 new had: 5
1 10  had: 1
0 new had: 1
1 7   had: 1
0 new had: 4
1 9   had: 1
0 new had: 1
1 7   had: 1
0 new had: 1
1 5   had: 1
1 0   had: 1
0 new had: 1
1 10  had: 1
0 new had: 4
1 0   had: 1
0 new had: 1
1 4   had: 1
0 new had: 1
1 10  had: 1
0 new had: 1
1 1   had: 1
0 new had: 3
1 7   had: 1
0 new had: 2
1 7   had: 2
0 new had: 3
1 1   had: 1
1 9   had: 1
0 new had: 1
1 4   had: 1
0 new had: 11
1 1   had: 2
0 new had: 3
1 7   had: 1
0 new had: 2
1 7   had: 1
0 new had: 5
1 9   had: 1
0 new had: 5
1 7   had: 1
1 1   had: 1
1 7   had: 1
0 new had: 9
1 4   had: 1
0 new had: 2
1 9   had: 1
1 6   had: 1
1 9   had: 1
1 1   had: 1
0 new had: 1
1 8   had: 1
0 new had: 1
1 6   had: 1
0 new had: 7
1 4   had: 1
0 new had: 1
1 10  had: 1
1 4   had: 1
0 new had: 2
1 4   had: 2
0 new had: 1
1 3   had: 1
0 new had: 5
1 8   had: 1
0 new had: 13
1 2   had: 1
1 10  had: 1
1 0   had: 1
0 new had: 4
1 1   had: 1
0 new had: 3
1 10  had: 1
0 new had: 1
1 8   had: 1
1 4   had: 1
0 new had: 4
1 1   had: 1
0 new had: 3
1 10  had: 1
0 new had: 1
1 3   had: 2
0 new had: 2
1 2   had: 1
0 new had: 1
1 1   had: 1
0 new had: 6
1 1   had: 1
0 new had: 5
1 0   had: 1
0 new had: 2
1 7   had: 1
1 0   had: 1
1 2   had: 1
0 new had: 9
```

```
1 5   had: 1
1 0   had: 2
0 new had: 2
1 6   had: 1
0 new had: 4
1 8   had: 1
0 new had: 1
1 6   had: 1
1 0   had: 1
1 5   had: 1
1 9   had: 1
0 new had: 2
1 5   had: 1
0 new had: 1
1 5   had: 1
1 4   had: 1
0 new had: 1
1 4   had: 1
0 new had: 4
1 2   had: 1
0 new had: 3
1 8   had: 1
0 new had: 4
1 3   had: 1
1 6   had: 1
0 new had: 2
1 10  had: 1
0 new had: 1
1 9   had: 1
0 new had: 2
1 6   had: 1
0 new had: 2
1 2   had: 1
0 new had: 3
1 5   had: 1
0 new had: 9
1 7   had: 1
0 new had: 4
1 0   had: 1
0 new had: 3
1 8   had: 1
0 new had: 6
1 4   had: 1
1 1   had: 1
0 new had: 3
1 10  had: 1
0 new had: 2
1 6   had: 1
0 new had: 6
1 5   had: 1
0 new had: 4
1 1   had: 1
1 3   had: 1
1 0   had: 1
0 new had: 3
1 4   had: 1
0 new had: 5
1 1   had: 1
1 2   had: 1
0 new had: 2
1 1   had: 1
0 new had: 2
1 10  had: 1
0 new had: 3
1 4   had: 1
1 6   had: 1
0 new had: 3
1 7   had: 1
0 new had: 1
1 4   had: 1
0 new had: 1
1 4   had: 1
0 new had: 17
1 2   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 6   had: 1
0 new had: 2
1 10  had: 1
0 new had: 3
1 10  had: 1
0 new had: 4
1 6   had: 1
0 new had: 7
1 9   had: 1
0 new had: 1
1 3   had: 1
1 6   had: 1
0 new had: 1
1 5   had: 1
0 new had: 2
1 4   had: 1
1 1   had: 1
0 new had: 3
1 2   had: 1
0 new had: 3
1 5   had: 1
0 new had: 5
1 8   had: 1
0 new had: 6
1 6   had: 1
0 new had: 6
1 5   had: 1
0 new had: 4
1 0   had: 1
0 new had: 2
1 3   had: 1
1 0   had: 1
0 new had: 4
1 9   had: 1
1 7   had: 1
1 6   had: 1
1 0   had: 1
0 new had: 3
1 1   had: 1
1 8   had: 1
0 new had: 8
1 0   had: 1
1 3   had: 1
1 5   had: 1
0 new had: 7
1 5   had: 1
1 6   had: 1
0 new had: 1
1 7   had: 1
1 1   had: 1
0 new had: 1
1 4   had: 1
1 2   had: 1
1 0   had: 1
0 new had: 7
1 2   had: 1
1 4   had: 1
0 new had: 1
1 2   had: 1
0 new had: 12
1 0   had: 1
0 new had: 6
1 6   had: 1
0 new had: 4
1 7   had: 1
0 new had: 2
1 5   had: 1
0 new had: 2
1 0   had: 1
0 new had: 1
1 9   had: 1
0 new had: 1
1 0   had: 1
0 new had: 1
1 4   had: 1
0 new had: 1
1 1   had: 1
1 7   had: 1
0 new had: 9
1 0   had: 1
0 new had: 4
1 4   had: 1
0 new had: 1
1 2   had: 1
0 new had: 4
1 6   had: 1
0 new had: 4
1 3   had: 1
1 5   had: 1
0 new had: 1
1 4   had: 1
0 new had: 2
1 1   had: 1
0 new had: 2
1 5   had: 1
0 new had: 7
1 1   had: 2
0 new had: 11
1 2   had: 1
0 new had: 6
1 9   had: 1
0 new had: 7
1 7   had: 1
1 2   had: 1
0 new had: 2
1 7   had: 1
0 new had: 2
1 4   had: 1
0 new had: 3
1 7   had: 1
1 1   had: 1
1 7   had: 1
0 new had: 3
1 3   had: 1
0 new had: 1
1 7   had: 1
1 6   had: 1
0 new had: 4
1 10  had: 1
1 2   had: 1
0 new had: 1
1 6   had: 1
0 new had: 1
1 6   had: 1
1 2   had: 1
0 new had: 1
1 7   had: 1
0 new had: 7
1 3   had: 1
0 new had: 3
1 9   had: 1
0 new had: 4
1 2   had: 1
1 3   had: 1
0 new had: 1
1 1   had: 1
0 new had: 2
1 3   had: 1
0 new had: 2
1 0   had: 1
0 new had: 1
1 10  had: 1
1 0   had: 1
0 new had: 2
1 2   had: 1
0 new had: 2
1 10  had: 1
1 2   had: 1
0 new had: 1
1 6   had: 1
0 new had: 9
```

```
1 4   had: 1
1 3   had: 1
0 new had: 7
1 10  had: 1
0 new had: 6
1 10  had: 1
0 new had: 9
1 6   had: 1
0 new had: 4
1 1   had: 1
0 new had: 4
1 3   had: 1
0 new had: 2
1 10  had: 1
1 5   had: 1
0 new had: 7
1 5   had: 1
0 new had: 7
1 3   had: 1
1 9   had: 1
0 new had: 12
1 5   had: 1
0 new had: 1
1 0   had: 1
1 5   had: 1
0 new had: 14
1 3   had: 1
0 new had: 1
1 7   had: 1
0 new had: 3
1 4   had: 1
0 new had: 3
1 7   had: 1
0 new had: 4
1 4   had: 1
0 new had: 2
1 9   had: 1
1 8   had: 1
0 new had: 1
1 6   had: 1
0 new had: 3
1 7   had: 1
1 0   had: 1
0 new had: 14
1 3   had: 1
0 new had: 5
1 4   had: 1
0 new had: 1
1 1   had: 1
0 new had: 1
1 3   had: 1
1 4   had: 1
0 new had: 4
1 4   had: 1
0 new had: 8
1 9   had: 1
0 new had: 7
1 2   had: 1
0 new had: 3
1 5   had: 1
1 6   had: 1
0 new had: 1
1 4   had: 1
0 new had: 5
1 8   had: 1
1 0   had: 1
0 new had: 6
1 7   had: 1
0 new had: 8
1 9   had: 1
0 new had: 1
1 10  had: 1
0 new had: 4
1 10  had: 1
0 new had: 8
1 1   had: 1
0 new had: 7
1 0   had: 1
0 new had: 6
1 9   had: 1
0 new had: 9
1 0   had: 1
1 3   had: 1
0 new had: 1
1 4   had: 1
0 new had: 8
1 10  had: 1
1 7   had: 1
0 new had: 4
1 4   had: 1
0 new had: 14
1 5   had: 1
0 new had: 1
1 0   had: 1
1 10  had: 1
1 7   had: 1
0 new had: 2
1 1   had: 1
1 3   had: 1
1 9   had: 1
0 new had: 1
1 1   had: 1
0 new had: 7
1 7   had: 1
1 2   had: 1
0 new had: 1
1 7   had: 1
0 new had: 5
1 1   had: 1
0 new had: 1
1 6   had: 1
1 5   had: 1
0 new had: 5
1 4   had: 1
0 new had: 5
1 6   had: 1
0 new had: 3
1 1   had: 1
0 new had: 7
1 0   had: 1
1 3   had: 1
0 new had: 5
1 1   had: 2
0 new had: 3
1 10  had: 1
1 4   had: 1
0 new had: 2
1 9   had: 1
0 new had: 1
1 4   had: 1
0 new had: 7
1 10  had: 1
0 new had: 5
1 9   had: 1
0 new had: 1
1 9   had: 1
0 new had: 3
1 3   had: 1
1 0   had: 1
0 new had: 1
1 10  had: 1
0 new had: 9
1 7   had: 1
0 new had: 6
1 2   had: 1
1 10  had: 1
1 6   had: 1
0 new had: 4
1 3   had: 1
1 0   had: 1
1 3   had: 1
0 new had: 4
1 0   had: 1
0 new had: 2
1 3   had: 1
0 new had: 1
1 3   had: 1
0 new had: 2
1 5   had: 1
0 new had: 4
1 0   had: 1
0 new had: 2
1 7   had: 1
1 9   had: 1
0 new had: 3
1 7   had: 1
0 new had: 11
1 3   had: 1
0 new had: 4
1 1   had: 1
0 new had: 1
1 2   had: 1
0 new had: 3
1 2   had: 1
0 new had: 11
1 0   had: 1
0 new had: 5
1 5   had: 1
0 new had: 5
1 6   had: 1
0 new had: 1
1 2   had: 1
0 new had: 2
1 10  had: 1
0 new had: 1
1 6   had: 1
1 7   had: 1
1 6   had: 1
0 new had: 1
1 9   had: 1
0 new had: 1
1 10  had: 1
0 new had: 4
1 5   had: 1
0 new had: 3
1 5   had: 1
0 new had: 1
1 10  had: 1
0 new had: 3
1 2   had: 1
1 6   had: 1
0 new had: 1
1 0   had: 1
1 2   had: 1
0 new had: 5
1 2   had: 1
0 new had: 4
1 7   had: 1
0 new had: 2
1 5   had: 1
0 new had: 6
1 4   had: 1
0 new had: 1
1 4   had: 1
0 new had: 3
1 4   had: 1
0 new had: 13
1 8   had: 1
0 new had: 2
1 10  had: 1
0 new had: 1
1 7   had: 1
0 new had: 6
1 1   had: 1
0 new had: 4
1 6   had: 1
0 new had: 1
1 7   had: 1
```

```
0 new had: 1
1 9   had: 1
1 1   had: 1
0 new had: 9
1 9   had: 1
0 new had: 1
1 7   had: 1
0 new had: 1
1 6   had: 1
0 new had: 1
1 5   had: 1
0 new had: 8
1 2   had: 1
1 9   had: 1
0 new had: 6
1 2   had: 1
0 new had: 5
1 9   had: 1
0 new had: 1
1 2   had: 1
1 1   had: 1
0 new had: 2
1 9   had: 1
0 new had: 1
1 5   had: 1
1 3   had: 1
1 5   had: 1
0 new had: 1
1 7   had: 1
0 new had: 2
1 0   had: 1
0 new had: 7
1 8   had: 1
1 3   had: 1
0 new had: 7
1 6   had: 1
0 new had: 2
1 7   had: 1
1 1   had: 1
1 4   had: 1
0 new had: 5
1 1   had: 1
0 new had: 3
1 10  had: 1
0 new had: 4
1 10  had: 1
1 5   had: 1
0 new had: 1
1 5   had: 1
0 new had: 4
1 6   had: 1
0 new had: 5
1 8   had: 1
0 new had: 14
1 4   had: 1
0 new had: 2
1 5   had: 1
0 new had: 2
1 1   had: 1
1 8   had: 1
0 new had: 3
1 10  had: 1
0 new had: 4
1 4   had: 1
0 new had: 7
1 7   had: 1
0 new had: 2
1 1   had: 2
1 3   had: 1
0 new had: 2
1 6   had: 1
1 4   had: 1
1 1   had: 1
0 new had: 5
1 1   had: 1
0 new had: 1
1 7   had: 1
0 new had: 8
1 1   had: 1
0 new had: 9
1 3   had: 1
1 2   had: 1
1 1   had: 1
1 4   had: 1
1 8   had: 1
1 10  had: 1
0 new had: 6
1 1   had: 1
0 new had: 4
1 8   had: 1
0 new had: 3
1 1   had: 1
0 new had: 1
1 7   had: 1
0 new had: 4
1 7   had: 1
0 new had: 2
1 0   had: 1
1 7   had: 1
0 new had: 3
1 7   had: 1
0 new had: 2
1 5   had: 1
0 new had: 2
1 2   had: 1
1 7   had: 1
0 new had: 10
1 5   had: 2
0 new had: 3
1 4   had: 1
0 new had: 3
1 8   had: 1
0 new had: 11
1 5   had: 1
0 new had: 15
1 1   had: 1
0 new had: 4
1 10  had: 1
1 8   had: 1
0 new had: 10
1 5   had: 1
0 new had: 7
1 1   had: 1
0 new had: 2
1 7   had: 1
0 new had: 4
1 8   had: 1
0 new had: 6
1 4   had: 1
0 new had: 3
1 10  had: 1
0 new had: 1
1 4   had: 1
0 new had: 1
1 9   had: 1
0 new had: 2
1 8   had: 1
0 new had: 11
1 4   had: 1
0 new had: 2
1 1   had: 1
0 new had: 1
1 0   had: 1
0 new had: 6
1 5   had: 1
0 new had: 2
1 4   had: 1
0 new had: 8
1 3   had: 1
0 new had: 6
1 4   had: 1
0 new had: 2
1 9   had: 1
0 new had: 4
1 6   had: 1
0 new had: 4
1 5   had: 1
1 2   had: 1
1 4   had: 1
1 10  had: 1
1 7   had: 1
1 4   had: 1
0 new had: 2
1 2   had: 1
0 new had: 5
1 9   had: 1
0 new had: 11
1 5   had: 1
0 new had: 3
1 10  had: 1
1 9   had: 1
1 10  had: 1
0 new had: 2
1 1   had: 1
0 new had: 10
1 8   had: 1
0 new had: 1
1 8   had: 1
0 new had: 5
1 1   had: 1
0 new had: 5
1 4   had: 1
0 new had: 3
1 8   had: 1
0 new had: 2
1 2   had: 1
1 0   had: 1
0 new had: 1
1 3   had: 1
0 new had: 2
1 9   had: 1
0 new had: 6
1 4   had: 1
0 new had: 1
1 6   had: 1
1 0   had: 1
1 1   had: 1
0 new had: 1
1 7   had: 1
0 new had: 1
1 1   had: 1
1 10  had: 1
0 new had: 1
1 0   had: 1
0 new had: 1
1 6   had: 1
0 new had: 3
1 3   had: 1
1 10  had: 1
1 0   had: 1
1 10  had: 1
1 7   had: 1
0 new had: 3
1 6   had: 1
0 new had: 3
1 3   had: 1
0 new had: 1
1 0   had: 1
0 new had: 5
1 1   had: 1
0 new had: 3
1 2   had: 1
1 9   had: 1
0 new had: 1
1 10  had: 1
0 new had: 5
1 4   had: 1
```