1. Hey everyone, my name is Taylor Benner and my project for CIS470 was to build a Simulated Evolutionary program.
2. I have a strong background in programming and systems management, and am currently working as a Web Applications Developer and Junior Systems Administrator for a Marketing Company. My background in web development helped prepare me for this project, but I wanted to do something a bit outside of my established knowledge base.
3. So, what is a simulated evolutionary environment? In this case, it is a program that employs neural networks, genetic algorithms, and a game loop to evolve creatures over time to be more effective food seekers.
4. The primary goals of this project were mainly an effort to learn concepts I didn’t know going into it. The project had to be able to render graphics to screen, create and manage a neural network for data classification, use concepts from genetic algorithms to move towards unknown optimization, perform functions to calculate movement and collisions, effectively run large amounts of calculations and concurrent processing, store member data in a NOSQL and dictionary based file, and finally perform reporting on how creatures were optimizing over time.
5. Each major system of the project was broken down into packages that defined their function. Requirements were then selected based on the original goals of the project and divided into the primary classes that compose the system.  
     
   The environment class needed to handle the bulk of the genetic algorithm logic as it is the only class that holds instances of Members and Targets. As such, population management, fitness scoring, parent selection, breeding and crossover, mutation, and collision detection all must happen at the environment level.  
     
   The member class defines the creatures that make up the population. This class had to hold a large amount of properties, but the primary responsibilities were that the member had to perform a series of updating calculating to ensure its state, position, and vision were constantly updating with the simulation. Based on these attributes the member had to render itself to a display object provided.
6. Each member had to have a brain, or a neural network controller. The network needed to be constructed when the class was initialized based on a predetermined structure. The class also had to act as an abreaction layer to activating network in order process inputs and receive an output. The class also needed to be able to add neuron and connections in a way that didn’t break the function of the network.   
     
   The target class was pretty simple in nature. The only major requirements of this class were that the coordinates generated for the position didn’t intersect other targets, and that the draw function rendered it to screen.
7. The Helper class was required to hold most of the reusable, ambiguous processing functions. As it turned out this class mostly held equations and generators.  
     
   The Database class is required to store and retrieve data. Due to the lightweight nature of the program, it was necessary to make the database a dictionary file that stores key value pairs outside of other relationships. As such, retrieving by key partials, or value partials are requirements of the class. Report generation was also required of this class.
8. Finally, the main controller for the program holds the game loop logic, and keeps the program running. As such it is required to process initialization, execution, updates, rendering, events, and finally cleanup.
9. The design of the system was handled in a very top down approach. Because a lot of the systems were unknown to me, I started off with the rendering system and primary game loop. From here I moved down into the subsequent classes, revising the work I had done on the previous classes as I move along. This meant that each class went through many different design and development iterations as I learned how to build and implement each one. Due to developing in a modular and class oriented way, it was easy to go back and change primary functions or the coupling amount later as the class was refined.
10. The flow of data was an important distinction in this program, as the updates happen constantly and very quickly. As such, ensuring each sub-routine performed the tasks needed was one of the main considerations. The four main sections can be broken into Initialization, Primary Loop, Update, and Render. These sections are covered more in the Game Loop construction section, but this graph shows the rough flow of data used in the program.
11. Because each major system is constructed as a class object in Python, coupling and encapsulation were important facets to use to facilitate iterative development. Once the primary requirements of each class were identified, I created connections from one class to another in as little places as possible. As we can see, the main class interfaces directly with the environment and the database class for running processing and data storage and retrieval functions. The main class never needs to impact targets, members, or the brains. The environment is a key central class as it handles storing instances of members and targets for easy processing accessibility. Similarly the member class held an instance of a brain, and the member is the only class that interacts with its brain. This was in an effort to isolate changes to the classes the change happened in, and not have to fix references all over the program as development progressed.
12. Due to the number of class files there would end up being, the directory structure needed to differentiate between classes, tests, and documentation. Python supports relative loading of classes with the use of an initialization file. In this way I would be able to access all classes and tests from a central location or project file which contained the main program class and logic. This also made it easier to manage development using an online repository as the packages were easily classifiable.
13. The development and construction of this project began in week one. As mentioned earlier there were a lot of concepts and systems that needed to be taken into account that I initially didn’t know how to build. As such, I chose an iterative development methodology in which I focused on major deliverables per week in terms of system. In order to test the system to ensure that progress was being made, I wanted a way to have visual feedback. I decided to develop the rendering system first so that subsequent changes could easily be compared. Also, I used an online versioning system known as GitHub. This allowed distributed coding over multiple environments, as well as a central storage location for the project files, and a versioning system that allowed me to track changes on a per week basis. Finally, in order to run the system more effectively, I used Sublime Text 2’s built in Build procedure to compile the python code to C when I saved major changes. This ensured that the project was always built to the most recent version and operated faster than using the python interpreter.
14. The major portions of this project are the execute loop, the neural network, and the genetic algorithm. The game loop was relatively simple to get up and running first. There were plenty of good guides online that discussed the update / render system and how calculation and drawing should be kept separate. I developed the rest of the system under this paradigm trying to keep the processing portioned away.
15. The game loop is the primary driver of the system. The initialization routine handled importing library dependencies and creating the initial instances for pygame and the environment class. All program wide global variables were initialized as an attribute of the main class.  
      
    The Primary Loop handles the processing chain for the whole program. In this routine, references to update and render are called. As such, at this level handling events and quit flags was ideal. Ultimately the only events that were handled were the quit event and keyboard events for slowing the simulation down and exiting. This process also holds the flag that indicates if the program is running, and checks it each iteration. When the program exits, the cleanup function is called and the debug information is displayed, and the objects are destroyed.  
      
    The update system is responsible for checking the state of targets and members, and updating the system accordingly. If, after updating all objects, there are no living creatures left, then the evolution process is completed, and the requisite functions are called to create the next generation. This ended up being the most processing intensive system as all calculations are handled within this package.  
      
    Finally, the render system wipes the display screen, calls the draw method for all objects, and then renders it all to the display area. I split this system out from the others so that rendering could be controlled to potentially speed up the processing. If the system doesn’t need to render graphics, it can process more calculation per second.
16. The neural network controller was built using the Pybrain library. It abstracts the math behind summing the weights of the connections and the thresholds of the neurons. It was a good choice for the library because of the number of features supported in the framework.  
      
    I experimented with network topology throughout the development of the program, and decided on a simple structure. The input neurons are Tanh or Hyperbolic Tangent Thresholds. This means they accept input values from -1 to 1 and activate when the threshold passes 0. This means that all input data needed to be normalized from their standard values to value falling within that range.   
      
    The 3 input values passed into the network are distance, energy, and relation. Distance is the calculated distance between the member and a nearby target. Relation is the calculated relative angle between the member’s current direction and the target. Energy is the amount of energy remaining before the member dies.

The network starts out with this basic topology and randomizes the connection weights used in calculating the output. Overtime, mutations can add new logic neurons and connections, along with randomizing existing connections. In this way, the network grows in complexity over time.

1. The genetic algorithm I used was a loose optimization model. Due to there being no ideal member or network, I used a scoring model based on the members lifespan in seconds, times the number of food they ate, or 1 if 0. I did this in the hopes that the fitness function would prefer members that lived longer in the simulation, and those members would be selected as parents.  
     
   Selection is a pretty simple elitist selection method with a variable selection rate. This means that the top N percent of members would be selected for reproduction, and the rest of the members would be discarded. I elected to choose a percentage of the members to retain genetic diversity instead of limiting the population to the genome of the top members. This was in an attempt to avoid what is called local minima, and move the simulation towards more optimized members randomly.  
     
   The crossover or reproduction of the members happened asexually. I found that when combining the genomic code for two networks together, I got less refinement in terms of optimization of the network. Asexual reproduction just means that a parent produces copies of itself 2-3 times. Each time a child is made; there is random chance for that member to be completely randomized, once again, to increase genetic diversity.  
     
   The mutation function would generate a random number between 0 and 1. If that number was less than or equal to the decimal value of the mutation rate, 1 of three mutations could occur. The network of a member could add a new neuron of the sigmoid variety. These were always added as logic neurons because the input and output neurons were requirements of the program. This would hopefully create new nodes in which the network could create new pathways. The network could also add a new connection to existing neurons. This would simulate the brain’s ability to create new pathways when learning. It was important to test the network when adding new neurons or connections as an infinite loop could have been created. If that were the case, I elected to undo the action, and move on. Finally, the weights of the connections could be randomized as well. All of these actions would seek to increase the network’s complexity over time in a random attempt to produce a member that performs better than the previous generations.
2. I won’t cover the details of how I united tested the program. When setting up the unit tests, I ensured that functions performing calculations or mutating objects would produce the values I expected such as finding the difference between two coordinates.   
     
   Integration tests would check to make sure that classes would instantiate objects correctly and that series of functions were performed in the correct order such as creating the next population.
3. Due to the nature of the system, I wasn’t really able to start analyzing results until mid-way through week 7. This was when I was confident that the program was going to operate in the way I anticipated. This means that I could do limited functional testing in terms of testing various parameters to see how they influenced the optimization potential. I decided to test high and low mutation values against increasing population sizes to see how the optimization was changed from test to test.  
     
   Through testing the parameters I was able to, I noticed that a higher mutation rate tended to produce higher scoring members. I suspect this is due to the shotgun blast approach, whereas if you try enough combinations, eventually one will perform well. Growing beyond this member however seemed to be just outside what I was able to accomplish as the scoring or selection methods weren’t rigorous enough.  
     
   Sadly I didn’t record how long it took to run 50 generations, although if I had I would have been able to calculate the potential for optimization as a function of time. This would have indicated if a higher or lower population size was in general higher performing. Ideally, I would have been able to test the simulation using a large amount of configuration options.
4. Throughout the lifetime of the project, through the various iterations and configuration changes I noticed a few things. It seems that in this simulation, high mobility was the winner. The more ground they could cover means the more chances to snag a food particle. I don’t think I had enough reinforcement to promote smart movement behavior.  
     
   It was also important to keep genetic diversity high enough to ensure that new combinations could be tested, but low enough to ensure that a good parent wasn’t wasted. This is a difficult option to nail down exactly and it ties into one of the bigger problems in hill climbing optimization insofar as a local minima may be a good selection for now, but not the best overall.  
     
   High mutation rates definitely contribute to the rigidity of the optimization, however too high of a value can produce too much noise or random data. A good parent may be selected, only to have all the children mutated in a way that prevents the species from moving forward, and a lesser performing member may be selected as the best.  
     
   Similarly too large of a population can have a similar effect of introducing too much noise into the simulation. At this saturation level, random chance plays too large of a role, and a less optimized member may accidentally come across more food than a more optimized member. Also, it’s a lot hard to process 100 members than it is to process 10!
5. Throughout the development of this project I came across some pretty large hurdles. The hardest hurdle by far, was I did not know the systems I was using to develop the project. Because I chose this project to catapult my learning forward, I had to learn these systems as I completed the rest of the project. In order to draft the requirements for instance, I had to first develop a portion of each system to know what was going to be required of it!   
     
   Working by myself was also very difficult regarding a project of this scope. Knowing that each week I had to complete coding, implementation, testing, and documentation for all aspects was a daunting prospect and I freaked myself out more than once. However, this also meant that I had control over all aspects of the project.  
     
   I had never done test driven development, so the few weeks spent on testing were pretty difficult. I learned afterwards that I had combined unit and integration testing into one package, but was unable to go back and refine it as I was running low on time. This also lead to some haphazard development practices as I didn’t have a clear understanding of the requirements / tests needed to make everything work until later.  
     
   Ultimately the most confusing problem I faced was getting the networks to optimize. There are so many variables that go into the creation of a successful network and genetic algorithm that ultimately I wasn’t able to produce something that achieved the desired output. However with the framework in place, I can go back later and try additional techniques that may lead to better results.
6. Over the lifetime of the project, I generally accomplish what I set out to. I learned a lot about the various systems at play, and have a much greater degree of confidence when it comes to preplanning a software project. I learned that test driven development can help flesh out requirements and expedite the development portion of a project. It can also lead to a greater understanding of how the system comes together, and reduce the amount of time spent on iterations.  
     
   If I could go back to week one and start again, I would have simplified the project scope and identified the requirements more in depth. Iteratively defining requirements works in certain circumstances but can be difficult to base a project around in terms of known development path and documentation.   
     
   I also learned that inputs for neural networks are best normalized. Linear, large whole numbers produce too much noise in the confines of a network and reduce the effectiveness of the changing weights. Normalizing input data to a particular scale helps the classification effort.  
     
   Genetic diversity that is too low will create a stagnant population and will lead to the local minima trap more often than not. Increasing or decreasing diversity within a genetic algorithm seems to be a trial and error approach for finding what works best for the specific parameters.  
     
   At first I started with 7 neural network inputs. I found that this ended up introducing too much noise and the neural network could not categorize that amount of input data down to 2 dimensions of output data. Reducing the number of inputs by specifying the values that were ultimately more important produced much smoother results.  
     
   Finally, I learned the value and application of time management. Due to the amount of work required for this project, combined with a full time job, I was forced to schedule my time more efficiently in order to complete the requirements. Falling back on the WBS and Gantt chart was useful, but ultimately the most useful lesson I learned was how, and when, to set aside time for development or documentation.
7. Show the application!