East Tennessee State University
Digital Commons @ East Tennessee State University

Undergraduate Honors Theses | Student Works

5-2010

Using Ant Colonization Optimization to Control Difficulty in Video Game AI.

Joshua Courtney
East Tennessee State University

Follow this and additional works at: https://dc.etsu.edu/honors

Part of the Software Engineering Commons

Recommended Citation
Courtney, Joshua, "Using Ant Colonization Optimization to Control Difficulty in Video Game AI." (2010). *Undergraduate Honors Theses.* Paper 147. https://dc.etsu.edu/honors/147

This Honors Thesis - Open Access is brought to you for free and open access by the Student Works at Digital Commons @ East Tennessee State University. It has been accepted for inclusion in Undergraduate Honors Theses by an authorized administrator of Digital Commons @ East Tennessee State University. For more information, please contact digilib@etsu.edu.

Using Ant Colonization Optimization to Control Difficulty in Video Game AI

Thesis submitted in partial fulfillment of Honors

By

Joshua Courtney
The Honors College
University Honors Scholar Program
East Tennessee State University

April 26, 2010

____
Mr. Jeff Roach, Faculty Mentor

____
Dr. Martin Barrett, Faculty Reader

____
Dr. Robert Beeler, Faculty Reader

Contents

Introduction 5

Review of Literature 7

Game Programming Basics 8

Fig 1. Avoidance Vectors 9

Fig 2. A Simple Finite State Machine 9

AI Case Studies 10

Fig 3. Sample Behavior Diagram (Isla, 2005) 11

Game Balancing 14

Ant Colonization Optimization Overview 17

ACO Case Studies 18

Fig 4. A 4x4 BMSN (Melo, 2009) 22

Fig 5. Gene, Chromosome, and Network Relationship (Pinto & Barán, 2005) 24

Experimental Overview 25

Fig 6. Initial Simulation 25

Algorithm 1 Enemy Movement 26

Fig 7. Pheromone Lifecycle 27

Algorithm 2 Position to Pheromone Coordinate 28

Results and Analysis 30

Table 1. Test Group Totals 30

Fig 8.Total Kills and Deaths 31

Bibliography 33

Appendix A: Data Tables 35

Long Pheromone Trails....................35
Short Pheromone Trails....................36
No Pheromone Trails....................37
Appendix B: Graphical Data Summaries....................38
Data Summary for Long Trails....................38
Data Summary for Short Trails....................39
Data Summary for No Trails....................39

Introduction

Ant colony optimization (ACO) is an algorithm which simulates ant foraging behavior. When ants search for food they leave pheromone trails to tell other ants which paths to take to find food. In computer science, this has been adapted to many different problems including the traveling salesman problem. The algorithm functions by randomly sending out artificial "ants" from a hub into a search space. Each ant finds a solution and then leaves an artificial pheromone trail along its path. The amount of pheromones depends on the success of the solution, i.e., a better solution leaves a stronger pheromone trail. The next iteration of ants takes the pheromones into account when choosing a direction to travel. Pheromones also weaken over time so less-optimal paths are abandoned for more-optimal paths. The result of many iterations is finding a good solution (although not necessarily the optimal solution as finding an optimal solution is NP-hard (Coltorti & Rizzoli, 2007)).

ACO has been used to solve many different types of problems. Coltorti (Coltorti & Rizzoli, 2007) used ACO to solve vehicle routing problems. These problems involved supermarkets moving food. The algorithm determined how to minimize driver time and cost while making sure all deliveries were made on time. ACO has also been applied to *Tetris* by Chen (Chen, Wang, Wang, Shi, & Gap, 2009). Chen created a value function which gave a value to the successfulness of placing a tetromino. Each time a piece arrives, "ants" crawl the grid of the game space and determine the path for the tetronimo to reach the optimal position. By this method as many as 23,000 lines were removed in a game. França (França, Coelho, Von Zuben, & Attux, 2008) used ACO to explore continuous search spaces. In França's algorithm, ants in each new generation are concentrated around the previous best solution. They explore the area around the solution to try to improve it. The advantage is that a local optimal solution can be found quickly. However, this can cause premature convergence because only one area of the search space is examined at a time. Melo (Melo, 2009) attempted to avoid premature convergence through the use of multiple colonies. At the end of each iteration, the optimal path determined by any colony is used to

adjust the other paths. Non-optimal paths are moved closer to the current optimal path. This causes more independent searches of the optimal area.

There is no research indicating the use of ACO in video game Artificial Intelligence (AI). Pheromone trails in ACO are essentially a way for independent ants to communicate with each other. This style of communication can be incorporated into a game AI allowing separate AI agents to communicate with each other. Pheromone trail communication can be implemented by having a pheromone grid overlaying the game environment. Enemies can send out signals to other enemies via pheromone trails that the other enemies can choose to follow these trails. Enemies can send out pheromone trails when they see the player, when they die, or when some other desired condition is met. By controlling the length and the strength of the trails left, one can control inter-agent communication. The better the agents can communicate the better they will be able to accomplish their goal of defeating the player. Therefore, by controlling the effectiveness of the communication between agents, one could control the difficulty of the game.

Review of Literature

In order to avoid the extremes of frustration or boredom, a video game should challenge its players without being excessively difficult. Historically, two strategies have been used for matching game difficulty with a player's skill level. The first, in player-versus-player games, is to trust players to play against similarly skilled opponents and to match up opponents as best as possible. The second, in player-versus-environment games, is to support multiple static difficulty levels and allow the player to choose.

Creators of video games have devised two additional kinds of adjustments for managing a game's difficulty. The first involves controlling the scarcity of the environment's resources that allow the player to succeed. The second involves adjusting the quality of computer-managed opponents. Some newer games use artificial-intelligence-like strategies for dynamically adjusting a game's difficulty based on a player's skill level. The difference of dynamic difficulty control is that the environment's resources or the enemies' quality (or both) is in a constant state of flux so as the player improves the game becomes harder or as the player does poorly the game becomes easier.

A computer-generated opponent is known as an agent or an artificial intelligence (AI). Modern games have complex AIs that include methods for carrying out actions and a controller for determining which actions to perform. Controllers have been implemented as finite-state machines, hierarchical decision trees, and goal oriented action programming, which uses a variation of the A* shortest-path algorithm to find appropriate actions.

Ant Colonization Optimization (ACO) is a search algorithm that could find use in game controllers that adjust to players' levels of skill. The ACO algorithm is a type of shortest-path algorithm that finds locally optimal solutions to a graph transversal problem. Artificial ants search a graph trying to reach a certain point. When they reach their goal they release a pheromone trail which other ants have a chance to follow.

ACO pheromone trails could be used by agents to pass information to other agents in a game environment. By modifying the parameters for leaving a trail, the effectiveness of communication between agents could be controlled. Since better communications between agents would make them harder to defeat, managing the effectiveness of communication could be used to manage game difficulty.

Game Programming Basics

The task of realizing an AI for first-person shooter games—games where the player sees the world through the main characters eyes and fights enemies using a variety of weapons—can be divided into two main requirements. The first is to implement the AI's actions, including looking for an enemy, avoiding obstacles while moving, and targeting an enemy with a gun (Howland, 1999). The second is to implement the AI's decision-making processes, to choose which actions the AI should perform.

Action, the simpler of the two requirements, is typically implemented as two components. The first, a weapons controller, models shooting, including the determination of the shots' effects, like damage (based on range and accuracy) and change in ammo. The other component models motion, including searching for and pursuing enemies.

Searching, is commonly handled using A*, described by Nareyek as "an improved version of Dijkstra's shortest-path algorithm" (Nareyek, 2004). A* functions by overlaying a graph over the game map. The graph's vertices are pre-determined waypoints defined by the programmer. The programmer also defines paths between these waypoints as well as their distances. These paths are the graph's edges. When an agent wants to move it locates its nearest waypoint. From there, the A* algorithm finds the shortest path between the starting waypoint and the waypoint closest to the agent's destination. That path is passed to the mover and the mover takes the agent to the destination. Once an enemy has been sighted, pursuit is conducted by using the viewing field to keep the player in view.

Obstacle avoidance, a concern that arises in searching and pursuit, can be managed by first moving to an obstacle's edge and then along it until the agent can again move straight towards its destination. A better strategy adds an avoidance vector, which the algorithm associates with each obstacle, to an agent's current movement vector (see Fig. 1). The resulting path will curve around the obstacle, because the goal vector is constantly tracking the goal. Obstacle avoidance algorithms act as an

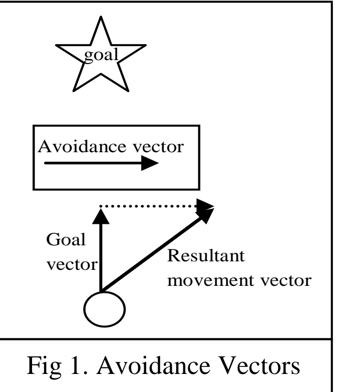


Fig 1. Avoidance Vectors

intermediary between the nodes on an A* graph. A* gives the mover a set of goals to move to, but simple obstacle avoidance algorithms handle any obstacles encountered between the nodes

The other major part of game AI, decision making, is typically implemented in one of several ways. The most basic implementation strategy, a finite state machine (FSM) (Nareyek, 2004), is a graph that characterizes every state that an AI agent can be in and all conditions for transitioning between states. An example of a very simple FSM would be one with two states and three transition conditions (See Fig. 2). FSMs are implemented using if-then statements. The problem with FSMs is that they become unmanageably complex with large and complex AIs.

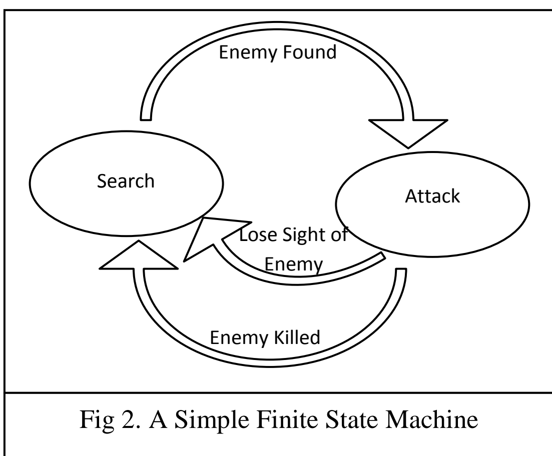


Fig 2. A Simple Finite State Machine

Another simple way to handle decision making is with decision trees. To reach a decision, the AI traverses the tree. The first level is very general with a question like "Should I work on attack or defense?" Based on the decision, the AI looks at the appropriate sub-tree. Eventually it descends to a leaf node that specifies an action. Like FSMs, decision trees can be implemented with if-then statements.

Actual game AIs combine simple and advanced mechanisms to create believable and realistic agents. Game AIs are effective if the game's agents appear to have common sense (Isla, 2005). This means knowing the right actions to do as well as how to do them and when to act. To establish the appearance of common sense, the AI needs to exhibit coherency, transparency, runnability, and understandability. Coherency is ensuring that behavior transitions are realistic and avoiding dithering, i.e., quick switches among a collection of states. Transparency is giving an agent an appearance that matches its current internal state, i.e. there should be some sort of graphical representation of an agent's state. Runnability is ensuring that the AI code can complete execution in the processor time allotted for AI decision making. Understandability is ensuring that the system is simple enough so that the developer can understand it. Also, the AI should also allow for character-to-character variety. Lastly, the AI needs variability so it can be "directed by the designers in service of the story" (Isla, 2005) through high-level scripting.

AI Case Studies

Halo 2 has a complex AI that used multiple strategies to achieve realism (Isla, 2005). *Halo 2's* AI system uses a hierarchical FSM (HFSM) combined with a decision–tree-based structure called a behavior diagram (see Fig. 3). An HFSM is an FSM whose levels are prioritized lists of states. *Halo*'s AIs service states according to their priority. A correct ordering of these states helps to ensure that an AI avoids nonsensical behaviors like trying to enter a vehicle if they are already in one.

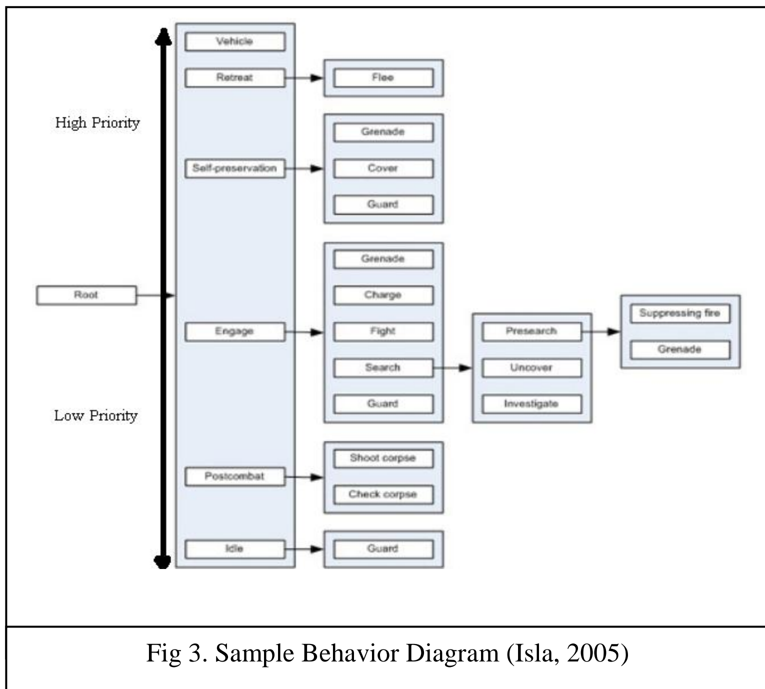


Fig 3. Sample Behavior Diagram (Isla, 2005)

The problem with HFSMs is the occasional need to dynamically raise the priority of a lower priority action. To allow a lower priority state to override a higher priority state, *Halo 2* uses behavior impulses: pointers that reference states in behavior diagrams. An example of a situation that would require a temporary priority adjustment is directing an AI to enter a vehicle if the player gets in first. Normally, fighting enemies would have a higher priority than finding a vehicle, but the "player_in_vehicle" impulse should be ranked higher than the fighting behavior. The "player_in_vehicle" impulse simply references the "enter_vehicle" code, rather than duplicating it. Impulses can be positioned at any level of a behavior diagram.

Behavior impulses can also arbitrarily execute small sections of code as well as behaviors. These codes, for example, can be used to log data, display debugging information, or play sounds. This opens up a wide range of uses for behavior impulses.

When a behavior diagram becomes large, determining which behaviors are relevant at a given time takes considerable time. To reduce the time needed to assess behavior relevancy, behaviors are tagged with the states in which they are relevant and temporarily removed from behavior diagrams when they become irrelevant. For instance, when the agent is a gunner in a vehicle the "throw grenade" behavior could be removed and the retreat option could be removed (if they are not the driver).

Another strategy for reducing the number of behaviors to check involves the use of stimulus behaviors: behaviors that are dynamically added to and removed from behavior diagrams by an event handler. An example of a stimulus behavior would be a "flee_because_leader_died" impulse. This impulse could be added to a behavior diagram by actor death event handler, and removed after an appropriate period of time or a compensatory event, like the arrival of a new leader.

Although different *Halo 2* AI's have different behavior properties, AI agents are similar enough to warrant the use of character hierarchies and inheritance to simplify implementation. For example, in *Halo 2* grunt majors take more punishment and do more damage than regular grunts, but exhibit identical behavior. So, a grunt major inherits all the characteristics of a grunt but modifies the vitality and damage statistics.

Another advanced AI system is the Goal Oriented Action Planning (GOAP) system developed by Jeff Orkin for Monolith's first person shooter, *F.E.A.R.* (2005) (Orkin, Agent Architectur, 2005) (Orkin, Three States, 2006). In place of an elaborate FSM GOAP searches for actions that meet a goal. This allows Non-Player Character (NPC) agents to handle unexpected situations.

Each agent is divided into sensors, working memory, a real-time planner, a blackboard, and subsystems that manage actions like movement and aiming. Sensors gather information about the agent's environment. Some sensors are event driven (like recognizing damage) while others poll (like finding tactical positions in the environment). The sensors store information gathered in the agent's working memory. The real-time planner watches for significant changes to working memory and responds by reevaluating the agent's goals and strategies for accomplishing those goals. If the goals are altered, the planner adjusts the relevant variables on the blackboard. Finally, the subsystems check the blackboard for changes at a set time interval and make any appropriate changes to their behavior.

The advantages of controlling agents using multiple components instead of a single FSM are threefold. First, this decouples goals from actions, making it easier to associate different strategies for achieving common goals with different units. The alternative, associating multiple strategies for achieving a common goal with a single FSM, produces extremely complex FSMs. Second, this makes it

easier to define behavior incrementally, including defining what behaviors are prerequisites for other behaviors and adding new actions late in the development cycle. Allowing the real-time planner to determine the appropriate transitions at run time eliminates the need to work new actions into an FSM. The third advantage is better support for dynamic problem solving. GOAP makes it straightforward to create agents that work through a list of prioritized strategies until they try one that succeeds. This can produce very realistic AI behavior. Orkin gives this example:

> Imagine a scenario where we have a patrolling A.I. who walks through a door, sees the player, and starts firing. If we run this scenario again, but this time the player physically holds the door shut with his body, we will see the A.I. try to open the door and fail. He then re-plans and decides to kick the door. When this fails, he re-plans again and decides to dive through the window and ends up close enough to use a melee attack! (Orkin, Three States, 2006)

Opening the door was the highest priority option, but it failed, as did the second priority option of kicking the door. The agent kept trying different methods until it found one that worked.

In *F.E.A.R.*, agents interact with their environment through the use of smart objects. A smart object is anything in the environment an agent can use to accomplish a goal. For instance, if an agent's goal is to get to a point on the other side of a closed door—a type of smart object—the agent would interact with the door to open it. Nearby smart objects are detected by an agent's sensors and placed in the agent's working memory. Since some actions may only be available when certain smart objects are present, an agent must reevaluate its goals when new objects are placed into working memory. For example, a weaponless agent that is chasing a player should pick up an assault rifle when it sees one and then continue to chase. An agent could also use a smart object to make cover for itself by flipping a table over and hiding behind it. The benefit of smart objects is that the programmer does not have to script the agent to kick over the table; the agent does so because the action helps to accomplish the agent's goal.

Action planning is done using the A* algorithm. A*, which was traditionally used for navigating a playing field, has been adapted to find the best way to accomplish a goal. To do this, each action is associated with a cost, with higher costs denoting less desirable actions. If a goal is treated as a

destination in a graph, possible actions as edges and resulting world states are intermediary nodes, A* can find the most efficient path (i.e., sequence of actions) to reach the goal. If that path fails to accomplish the goal, the edge for the inappropriate action can be removed from the search and the next best path can be found. Paths that A* finds may need to be disqualified because some paths may be unavailable at certain times. Certain actions may only be relevant if an agent is in a squad, or if the agent has no weapon.

GOAP also produces ghost behaviors or unintended behaviors that emerge in practice. One example of this was NPCs' looking at distant grenades, due to NPCs' use of noise to find players. Another ghost behavior was NPCs' finding points of cover to a player's side, which gave the appearance of NPCs working together to flank a player. Ghost behaviors emerge in a GOAP system due to unanticipated state transitions. An FSM would require a programmer to tell an agent how to act in the presence of a grenade; with GOAP, defining the grenade as a danger and a disturbance causes the agent to determine an appropriate—or, in the case of ghost behaviors, inappropriate—reaction at run time.

Another AI technique used in *F.E.A.R.* is "fake" AI, or the use of audio and visual cues to suggest agent activity. For example, when a squad of soldiers is advancing, the game may generate a cue like "I'm moving. Cover me!" Similarly, when a grenade falls near an NPC in a squad, the game may generate a cue like "Look out!" followed by the NPC's trying to escape. In these instances, the agents are reacting to their environments rather than the cues; they only appear to follow the cues because the cues are in sync with their goals. Another example of "fake" AI is a call by a squad's last member for reinforcements. While this call has no effect on calling troops, the player will encounter more troops later in the level, making it appear as though the AI responded to the NPC's request for reinforcements.

Game Balancing

Controlling game difficulty, also called game balancing, is an important gameplay issue. A player will not enjoy a game if it is too easy or too hard. Static difficulty levels (easy, medium, hard, etc.), which game developers normally use, can make parts of the game too easy and others too hard. Some game balancing systems can give players or non-player characters (NPC) an unfair advantage. A

common example of this is the “rubber band” effect found in many racing games where the last place car is rocketed forward to near the front (Hunicke, 2005). Other games use a ramping technique where the game steadily gets harder as it goes on. However, increasing the difficulty faster than a player’s learning curve can frustrate the player. These considerations have led to the study of dynamic difficulty adjustment (DDA).

Difficulty adjustments can take different forms. A game can switch between different policies for how to challenge players. For instance, a game could switch from a comfort policy that attempts to “keep players feeling challenged, but safe [by] padding their inventory” (Hunicke, 2005) to a discomfort policy that challenges players by limiting item drops. Another way to adjust difficulty is through more direct intervention. Items like weapons or health packs could be added to the playfield. The player’s hit points or attack strength could be modified. Enemies’ hit points, weapons, spawn locations, or accuracy could all be modified. A combination of these methods is normally used to adjust difficulty.

Hunicke (Hunicke, 2005) has developed a DDA system that regulates a game’s mechanics, dynamics, and aesthetics. Mechanics are player interactions with the environment. Health, ammunition, and weapons would all be part of a game’s mechanics. Dynamics are player movements between encounters. The rate at which the player finds new weapons or power-ups is part of a game’s dynamics. Aesthetics are how the mechanics and dynamics create difficulty. Increasing a game’s difficulty as it progresses is a strategy for managing a game’s aesthetics.

Hunicke integrated her system, the “Hamlet System” (Hunicke, 2005), into Valve Software’s *Half Life* game engine. The Hamlet System is divided into two parts: evaluation of a player’s performance and adjustment of a game’s settings. The system evaluates players based on the rate at which they lose health. The amount of health a player loses over a set period fits a Gaussian probability distribution. “During combat, Hamlet records the damage…each enemy does to the player” (Hunicke, 2005). From this data the Hamlet System can determine the probability the player will die in that encounter. If the player’s probability of death rises above 40%, the Hamlet System intervenes. It increases the player’s health by 15 points every 100 ticks.

Hunicke experimented to see how adjustments affected player performance, if players noticed adjustments, and if adjustments affected "the player's enjoyment, frustration, or perception of game difficulty" (Hunicke, 2005). Players playing the unadjusted and adjusted games died an average of 6.4 times and 4.0 times in the first 15 minutes, respectively. If repeated death is equated with frustration, then these adjustments should reduce player frustration. In a short survey given following gameplay, expert players rated the adjusted game more enjoyable while novice players rated them equally. The survey also found no correlation between a player's perception of game difficulty and whether the game difficulty was adjusted. This means that the game was made less frustrating and more enjoyable without the player feeling as though the game was "fixed" in their favor. Hunicke concludes that if a small change like manipulating health could improve a game, then a well designed DDA system has the potential to greatly improve a game.

Another method for handling DDA is proposed by Andrade et al. (Andrade, Ramaloh, Santana, & Corruble, Challenge-Sensative Action, 2005) (Andrade, Ramaloh, Santana, & Corruble, Automatic Computer Game Balancing, 2005). Andrade et al. integrated their DDA system into a fighting game called *Knock'Em* which is similar to Midway's *Mortal Kombat*. Like Hunicke's system, Andrade et al.'s Reinforcement Learning (RL) system uses a difficulty calculation to manipulate the game. However, fighting games differ from first-person shooters like *Half Life*: they do not provide weapons or health packs. The authors rejected two strategies for DDA. Dynamic scripting can become too complex in large systems while genetic algorithm techniques do not adapt quickly to a player's skill level.

Andrade et al.'s technique of choice, Reinforcement Learning, is "characterized as the problem of learning what to do (how to map situations into actions) so as to maximize a numerical reward signal" (Andrade, Ramaloh, Santana, & Corruble, Challenge-Sensative Action, 2005). RL is based on a Markov Decision Process (MDP) involving a series of reward values $r(s, a)$, where an entity receives a reward for an action a in a state s. The RL algorithm attempts to maximize an entity's reward value by choosing the correct action based on its current state. The algorithm uses memories of past choices and the results of those choices to choose the best action to maximize reward.

The authors discuss two main difficulties with RL. The first is getting the game's AIs to play at the same level as the player at the start of the game. To do this, Andrade pre-trained AIs by having them play against themselves to develop basic character policies. Once the agents start playing against the player they refine their play style to complement the player's style and skill. The second difficulty is choosing what to do once the optimal policy has been learned. Directing agents to randomly choose actions could result in nonsense actions (like punching when the opponent is on the other side of the screen). Directing agents to choose only optimal actions would make the agent impossibly difficult. Instead, the AI agent must choose "progressively sub-optimal actions until the agent's performance is as good as the player's" (Andrade, Ramaloh, Santana, & Corruble, Challenge-Sensative Action, 2005) or more optimal actions if the game becomes too easy.

Andrade tested his RL agents against agents that randomly choose actions and agents that always choose the optimal action. He found that the fights normally ended with a small difference in health points, "meaning that both fighters had similar performance" (Andrade, Ramaloh, Santana, & Corruble, Challenge-Sensative Action, 2005)—that is, the RL AI agents closely matched their opponent's skill level. Andrade is currently in the process of testing his RL AI agents against real people to see if the results stand.

Ant Colonization Optimization Overview

Ant Colonization Optimization (ACO) is a search algorithm that simulates ant foraging behavior (Coltorti & Rizzoli, 2007). The algorithm functions by randomly sending out artificial "ants" into a search space, starting from random points in the search space or at a central hub. Each ACO ant that finds a solution mimics the behavior of a biological ant by leaving a simulated chemical (pheromone) trail on its path home. This path's strength is proportional to the solution's success; shorter solutions yield stronger trails. The next iteration of ACO ants accounts for pheromones when choosing a direction to travel. Simulated pheromones, like real pheromones, weaken over time, causing less-optimal paths to be

abandoned for better paths. More extensive searches tend to find better, if not necessarily optimal, solutions.

ACO Case Studies

ACO has been adapted to a variety of NP-hard problems. One successful use of ACO by Coltorti and Rizzoli (Coltorti & Rizzoli, 2007) used the ANTROUTE algorithm to optimize vehicle routing problems. A vehicle routing problem consists of multiple clients like grocery stores that need to be serviced by vehicles. An optimal solution serves all clients and minimizes resource consumption while respecting operational constraints. Examples of these constraints would be “the driver’s maximum working time, and minimizing the total transportation cost” (Coltorti & Rizzoli, 2007).

ANTROUTE is divided into two stages (Coltorti & Rizzoli, 2007). During the first stage, each ant independently finds a solution. During the second stage, pheromone trails are decreased due to evaporation and increased based on the ants’ paths. ANTROUTE searches for solutions using two separate colonies of ants: one that minimizes driving time and another that minimizes vehicles. By combining the solutions, paths were found that would minimize the number of vehicles needed and the total distance travelled.

Coltorti and Rizzoli used ANTROUTE to calculate a distribution strategy for a supermarket chain in Switzerland that distributes goods to over six-hundred stores. The search space consisted of a graph whose vertices modeled the chain’s stores and delivery hub and whose edges modeled transportation routes. Distances between vertices were calculated using average driving speeds determined from real world data. Coltorti and Rizzoli also modeled overheads like the time needed to hook a trailer to a truck and to unload pallets at a destination. Their main constraint was that all routes had to be competed in one day.

ANTROUTE initially proposed a strategy that cut the number of routes in use from 2056 to 1614 and increased truck space used from 77% to 98%. The solutions were found to be infeasible, due to its failure to return trucks to their starting points for next-day use. The algorithm was then modified to use

petal shaped routes, similar to those used by human planners. The modified algorithm's solutions used 1807 routes with a trucking load of 87%. ANTROUTE, moreover, took five minutes to do what took the human planners close to three hours.

Coltorti and Rizzoli used ANTROUTE to compute routes for an Italian company with multiple delivery hubs. Goods moved from manufacturers to hubs, and thence to shops. Italian laws required all pick-ups to be made before any deliveries, and forbade orders from being split between tours. Deliveries were handled by a subcontractor with trucks deployed throughout Italy. This allowed trucks to start at the first pickup point instead at a delivery hub and to be used without concern for their numbers. So, ANTROUTE was modified so it only had one colony.

Coltorti and Rizzoli determined that the algorithm did about as well as human planners if the problem complexity was low, but outperformed people for routes involving large numbers of orders and high complexity. On average, ANTROUTE reduced the number of routes needed by ten and increased the efficiency by more than 4%.

Coltorti and Rizzoli also used ANTROUTE to schedule emergency winter deliveries for a fuel oil distribution company whose customers that ran out of fuel. The challenge in this case was the need to schedule trucks that were already delivering fuel to other customers. To support new orders, the day was divided into time slices. In between these slices, filled orders were removed and new orders were added. At each time slice ANTROUTE was rerun and new routes were planned. After testing having a number of time slices between 5 and 200 it was determined that 25 time slices minimized travel time.

As with Coltorti and Rizzoli, initial ACO implementations were designed to find strategies for traversing closed graphs. ACO algorithms have since been adapted to traverse continuous search spaces. A continuous search space is a graph that has an infinite number of nodes and edges. This scenario models real-world searches for food, where there are infinitely many paths for ants to follow and directions to take. One algorithm for continuous search spaces by França et al (França, Coelho, Von Zuben, & Attux, 2008) is called Multivariate Ant Colony Algorithm for Continuous Optimization (MACACO).

França et al.'s work combines and extends two earlier ant search algorithms. The first, CACS (Continuous Ant Colony System), was proposed in 2004 by Pourtakdoust and Nobahari. In CACS, "the discrete pheromone probabilistic function is replaced by a Gaussian probability density function (PDF)" (França, Coelho, Von Zuben, & Attux, 2008). During each iteration of the search, the ants modify the distribution's mean and variance. Initially, the variance is set to three times the range of the variables. The variance is then modified at run time, using "a weighted average of the distance between each individual in the population and the best solution found so far" (França, Coelho, Von Zuben, & Attux, 2008). The method's advantages include the method's use of only one parameter—the number of ants—making successive generations of ants easier to generate. However, the algorithm converges to a single local optimum, leading to premature convergence on an optimum value.

The second search algorithm, ACOR, was proposed by Socha and Dorigo in 2006. In ACOR, solutions are "built according to an archive of the *n* best solutions found so far" (França, Coelho, Von Zuben, & Attux, 2008). Like CACS, ACOR uses a Gaussian PDF. However, ACOR has multiple PDF's: one for each of the *n* best solutions. The ants find new solutions by following the pheromone trails. The solutions are input into the archive and sorted by fitness. Solutions that are worse than the *n*th best solution so far are deleted and the process is repeated. The advantage of using multiple PDFs is there is a much lower occurrence of premature convergence. But, maintaining multiple PDFs is computationally expensive.

MACACO exploits relationships between variances in multiple variables (or dimensions) to reduce the size of the search space explored by the algorithm. In a two-dimensional space, two Gaussian distributions plotted against each other on a plane will form a circle. A multivariate distribution, on the other hand, will form an ellipse, which encompasses significantly less area than the circle. Once this ellipse's shape has been calculated, it can be translated to overlay the portion of the search space which contains the current best solution resulting in a more focused search.

To implement MACACO's search algorithm, a covariance matrix Σ is created with center μ. Then, a vector x is created for each node containing the probabilities for every variable that is a search

parameter. Next, a matrix of the normalized eigenvectors of the covariance matrix φ is created as well as a diagonal matrix Λ containing the eigenvalues. Next, a value Q is defined as $Q= \Lambda^{1/2} \varphi$. Each vector x is then replaced by a vector y where $y=Q*x+\mu$. The weights in y are used to shape the Gaussian distributions at the nodes. After each iteration, the covariance matrix is recalculated using the best 70% of the solutions.

França tested MACACO against CACS and ACOR on six benchmark problems. Even though MACACO "demands the calculation of the eigenvectors and eigenvalues of a correlation matrix at each iteration" (França, Coelho, Von Zuben, & Attux, 2008), it was, on average, no worse than about half as fast as CACS, as well as 10 times faster than ACOR. On average, MACACO also found closer to optimal solutions than CACS or ACOR on four of the six benchmarks. França concludes that MACACO improves on ACOR. He also concludes that MACAO usually improves on CACS, since it typically produces better results for a small additional computational cost.

The ACO algorithm has also been adapted to play the video game *Tetris*. *Tetris* is a good candidate for ACO because "it is NP-complete to maximize the number of rows removed while playing the given piece sequence" (Chen, Wang, Wang, Shi, & Gap, 2009). An implementation of an ACO algorithm was proposed by Chen et al (Chen, Wang, Wang, Shi, & Gap, 2009). Chen's algorithm selects the best move based on a value function which is run on "all possible subsequent game boards" (Chen, Wang, Wang, Shi, & Gap, 2009). This value function considers sixteen parameters, including the position of the highest hole, the number of blocks above the highest hole, and the number of potential lines. Chen's algorithm scans a list of Tetronimos and positions them according to their optimal values. The algorithm also uses a dynamic heuristic in conjunction with the pheromone evaporation rate to prevent premature convergence.

Chen's algorithm shows promise compared to some algorithms for playing *Tetris*. It completed an average of about 7000 lines per game with a maximum of over 17,000 lines. Its performance is however overshadowed by the Noisy Cross Entropy method, which had a maximum of almost 350,000

lines in one game, and a genetic algorithm that had a maximum of almost 600,000 lines. Chen suggests that an exponential value function instead of a linear value function could improve performance.

Melo used a multi-colony ACO to solve a node placement problem (NPP) (Melo, 2009). The goal of an NPP is to minimize the average weighted hop distance between all nodes in a graph. Melo's problem was set on a Bidirectional Manhattan Street Network (BMSN): a graph where every node is connected to four other nodes, using weighted edges that represent communication paths, together with the amount of traffic that flows along each edge (See Fig. 4). Intuitively, minimizing the average weighted hop distance corresponds to optimizing communications across the network by minimizing message traffic throughout the network.

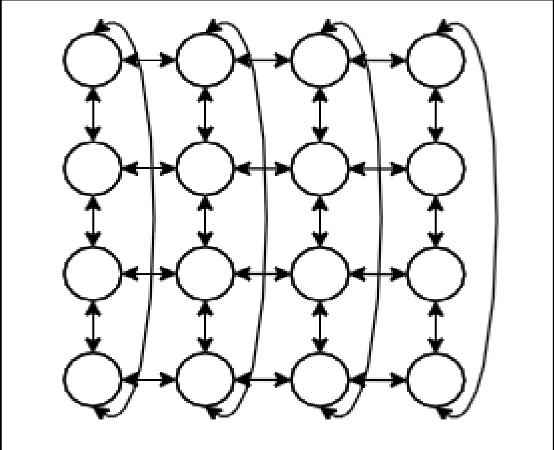


Fig 4. A 4x4 BMSN (Melo, 2009)

Melo's multi colony approach avoids premature convergence while also avoiding more computationally intensive ACO implementations. Each of Melo's colonies has the same number of ants, runs for the same number of iterations, and shares a common heuristic function. Each colony has a unique pheromone trail and records its own best performance. During each of the algorithm's iterations, each hive finds its current best solution. It then tries to improve on that solution using a local search. The algorithm then determines the best and worst colony. If the difference between the best and worst colony is significant enough then the best colony path is used to lay pheromone for the worst colony. This process is called a trail migration. This allows the colonies to explore more of the graph while focusing on the graph's more promising areas.

In experimentation, Melo found that more using colonies produced better solutions. He tested his algorithm on 4x4, 8x8, and 16x16 BMSNs. As the BMSN got larger, the ACO solutions diverged from the optimal solutions. Melo also noted that trail migrations are more frequent during the algorithm's earlier iterations because the pheromone trails are not yet well established.

Pinto and Barán used ACO algorithms to solve multiobjective multicast routing problems (Pinto & Barán, 2005). A multicast "consists of simultaneous data transmission from a source node to a subset of designation nodes in a computer network" (Pinto & Barán, 2005). Multicast is used to implement services like TV transmissions and teleconferences, both of which are offered with quality of service (QoS) guarantees. When issuing a multicast a network must consider QoS along with load balancing and network resource utilization. So, any path optimized over a network must account for all of these considerations.

Because multicast routing problems attempt to simultaneously optimize different, conflicting parameters, they have no single solution. To assure converge to a single optimal solution, these parameters must be totally ordered.

Pinto and Barán adapted two algorithms to solve the multicast routing problem. The first, the Multi-Objective Ant Colony Optimization Algorithm (MOACS), places all ants at one source node. From there the ants randomly traverse the grid until they find a solution. Each ant that finds a solution leaves a pheromone trail to that solution. The algorithm defines λ values to represent the relative importance of the different parameters. These λ values help determine which path is the best path and what strength of pheromone to leave on the solutions. Stronger pheromones are left on superior solutions and weaker pheromones are left on inferior solutions.

The second algorithm, Max-Min Ant System (MMAS), uses each iteration's best solution to update the pheromone trail. Pheromone trails are initialized with a high value to ensure high exploration at the start of the algorithm. As in MOACS, λ values determine the relative influences of the parameters.

The authors contrasted MOACS and MMAS with the Multiobjective Multicast Algorithm (MMA), an evolutionary algorithm that has previously been shown to solve a multiobjective multicast routing problem. Genes are built from random paths and chromosomes are built from the genes (See Fig. 5). Chromosomes are compared based on a fitness algorithm and new genes are built from combinations of the most fit genes along with some random mutation.

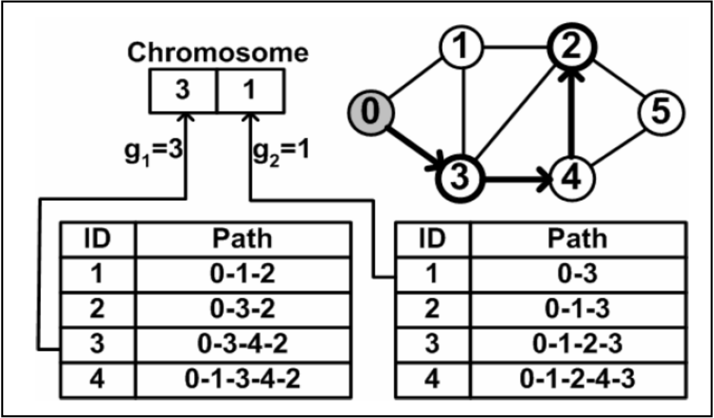


Fig 5. Gene, Chromosome, and Network Relationship (Pinto & Barán, 2005)

A comparison of the three algorithms on multiple network topologies determined that MOACS and MMAS found more solutions (30 and 12 respectively) whose parameters were within acceptable range than MMA (six solutions). However, the MMA solutions dominate several of the solutions of MOACS and MMAS, meaning MMA finds very good solutions. On average, MOACS had superior performance because it averaged 65% undominated solutions compared to MMAS which averaged 13.5% undominated solutions and MMA which only averaged 10% undominated solutions.

Based on their results, Pinto and Barán concluded that MOACS and MMAS are viable algorithms for solving multiobjective multicast routing problems with MOACS being the better of the two.

Experimental Overview

The first task in the process of integrating ACO into games was to show proof of concept, i.e., show that pheromone trails could be placed on a grid and "ants" could be made to follow them. This was accomplished through the use of a simple simulation. A grid was set up with one square marked as a goal and a different square marked as a source. One hundred "ants" started at the source and each moved in a random direction. When an "ant" moved onto the goal square it was removed from the simulation and trails were added around the goal pointing towards it. In Figure 6, the green squares represent the ants which are searching for the goal, marked in pink. When one of the ants reaches the goal it releases a pheromone trail (blue) which other ants could follow. Over time the strength of the trail would decrease

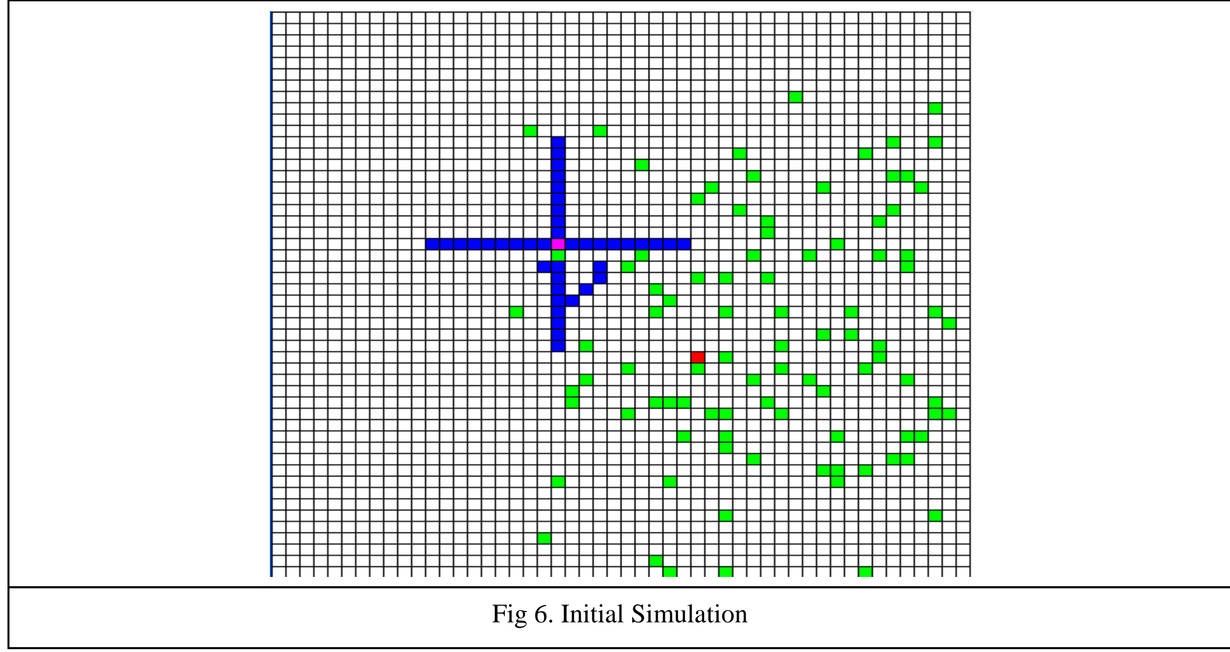


Fig 6. Initial Simulation

until it would fade away completely. After several trials, it was apparent that "ants" would follow the pheromone trails when they were active. Because these initial tests were successful, the next step was to integrate pheromone trails into a simple game.

Ant Hunter was developed to demonstrate the utility of pheromone trails in game AI. *Ant Hunter* was written in C++. The logical structure consists of six classes, two structs, several free functions, and a main driver function. It has 1772 lines of code including white space and comments. Graphics are

handled using OpenGL with the glut32.dll library. OpenGL is also used to manage keyboard I/O and run the AI tick at set time increments.

In *Ant Hunter* the player is an orange square who attempts to kill the enemy purple squares. Enemies in *Ant Hunter* have a very simple two-state AI. When an enemy is close to a player it is in the attack state. While in the attack state an enemy will move directly towards the player. If the player gets too far away, an enemy will switch to the search state. In the search state, enemies randomly move around the screen unless they are moving over a pheromone trail. Enemies that are moving over a pheromone trail have a chance of following it depending on its strength (see Algorithm 1). When an enemy gets close enough to the player it will switch to the attack state.

```
r = random num between 1-10
if (r < pheromone strength at enemy
position)
{
      move in direction of pheromone
}
else
{
      move in random direction
}
```

Algorithm 1 Enemy Movement

When an enemy is killed it releases a pheromone trail. Each pheromone has a strength value and a direction value. Strength can range from 0-10 inclusive and the direction can be up, down, left, right, and half way between each of those. A killed enemy releases a trail of strength 9 in the 8 different directions all pointing towards the point where the enemy was killed in an effort to draw in other enemies to attack the player (see Fig 7). The more recent a trail is the darker its color and the more likely an enemy is to follow it. Over time, pheromone trails fade unless they are reinforced by the deaths of other enemies. Figure 7 demonstrates the lifecycle of a pheromone trail.

Enemies also release pheromone trails when they switch from the search state to the attack state. While an enemy moves around during the attack state it keeps track of its position history. When an enemy switches from search to attack, the stored movements are turned into a pheromone trail.

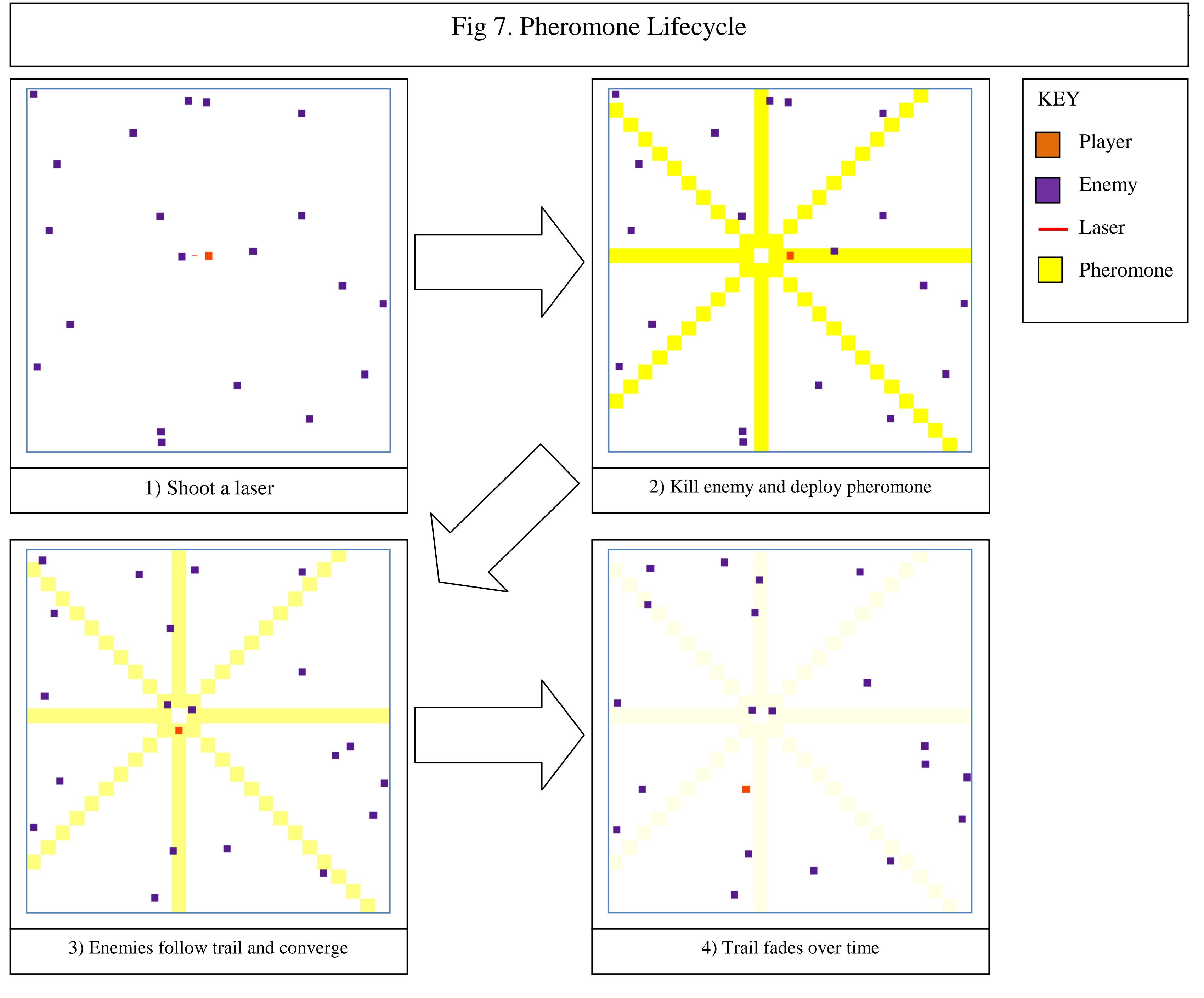


Fig 7. Pheromone Lifecycle
KEY
Player
Enemy
Laser
Pheromone
1) Shoot a laser
2) Kill enemy and deploy pheromone
3) Enemies follow trail and converge
4) Trail fades over time

The environment of *Ant Hunter* consists of two grids: the position grid and the pheromone grid. The position grid uses pixels as positions. This allows for simple collision detection based on the center position of an object and its size. The pheromone grid is essentially laid over the position grid. Not every position has its own unique node on the pheromone grid. If this were the case, the effect of pheromones would be very limited. Instead, multiple positions are tied to one node of the pheromone grid.

By assigning a scale factor between the two grids and using integer division it is possible to easily transform a position to a node on the pheromone grid (see Algorithm 2). By this method, a range of positions can be linked to a single pheromone trail and thus an enemy will follow a pheromone trail if he is anywhere in that range.

```
int position x
int position y
int scale factor

pheromone grid x = position x / scale factor
pheromone grid y = position y / scale factor
```

Algorithm 2 Position to Pheromone Coordinate

The player in *Ant Hunter* is controlled via the keyboard. The player moves using the standard WASD keyset and fires using the arrow keys. The goal of the player is to kill as many enemies as possible. Enemies respawn over time after they are killed. If an enemy comes into contact with the player the player dies. When the player dies all of the pheromone trails are removed, the enemy is killed, and the player is instantly moved to a random new location.

The goal of the experiment was to determine whether or not pheromone trails could be used to affect the difficulty of the game. To test this, a simulated player (sim-player) was created to play the game. The simulated player used a simple AI algorithm. At every tick it moves in a random direction. It also shoots lasers at any enemies who were in a direct line with its position — up, down, left, or right — plus or minus five pixels. The result of this is that the sim-player has better-than-human reflexes.

The experiment was conducted with three separate test groups: one with long pheromone trails, one with medium length pheromone trails, and one with no pheromone trails. An individual simulation consisted of allowing the sim-player to play the game for five minutes at four times normal game speed.

After each minute the number of enemies killed and the number of player deaths was output to a unique log file. A total of fifty simulations were run for each test group.

Results and Analysis

After running the simulations, the results show a significant difference in the number of enemies killed and the number of player deaths between the three test groups (see Table 1).

	Kills	Deaths	K:D
Long Trails	4000	1907	2.10
Short Trails	3103	1402	2.21
No Trails	1180	587	2.01

Table 1. Test Group Totals

The first thing to note is that the sim-player was equally effective at killing enemies at all three pheromone levels. This is demonstrated in two ways. The first is that the composite kill-to-death ratio was close to 2.1 for all three test groups. The second is that the kill- to-death ratio becomes more consistent the more enemies the sim-player has to face. This is demonstrated by the decreasing variance as the number of kills and deaths increases. The variances were 1.21, 0.44, and 0.29 for no trails, short trails, and long trails respectively times (see Appendix B for complete statistical analysis). This is because the sim-player has perfect reflexes — far better than any human player could have. This means that that difficulty cannot be measured based on the performance of the sim-player.

It is clear that from the data that the sim-player will die more when pheromone trails are used as opposed to when they are not. With long pheromone trails the sim-player died an average of 38.1 times per simulation, with short trails it died an average of 28.0 times, and with no trails it died an average of 11.7. This can be attributed to the fact that the pheromone trails cause the sim-player to tend to face more enemies during a trial.

The sim-player has no logic for running away from an enemy. So, every interaction that the sim-player has with an enemy ends with either the death of the enemy or the sim-player. Therefore the sum of the kills and deaths for a trial is the number of enemies the sim-player had to interact with during a trial. Unlike with the sim-player, the more enemies a human player has to face the more likely they are to make a mistake. When a player is more likely to make a mistake, a game is harder. Therefore it is reasonable to say that the more enemies the sim-player has to face the harder the game is.

Looking at the results for the three test groups it is clear that the longer the pheromone trail was, the more enemies the player had to face (see Fig. 8). This means that the longer pheromone trails were successful in drawing more enemies to the player. As noted before, more enemy interactions are equivalent to a harder game, so, in this sense, difficulty was successfully controlled by manipulating the parameters of the pheromone trails.

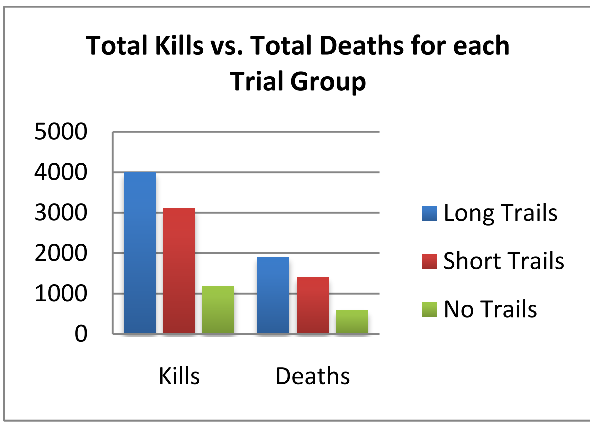


Fig 8.Total Kills and Deaths

It can also be concluded that pheromone trails are an effective form of communication between two separate AI agents. When an enemy dies it relays the current position of the player to other nearby enemies via the pheromone trails. Without this, the other enemies would continue to be ignorant of the player's position. The pheromone trails allow communication with nearby enemies using a memory-expensive method as opposed to a computationally-expensive method. An enemy simply adds pheromones to the pheromone grid instead of having to determine which other enemies are within a certain distance — a process which would require a distance calculation as well as other logic.

Pheromone trails do not need to be limited to their functionality in *Ant Hunter*. Pheromone trails could be used to relay information about the location of any important object in a gaming environment. They could also be used to issue commands from one AI agent to another (like a squad commander giving commands to his soldiers). There are many possible uses for pheromone trails and these should be explored.

Future work with *Ant Hunter* could include running trials with human players as opposed to the sim-player. This could confirm that human players have a harder time when facing more enemies and will have more deaths than when they face fewer enemies. Another option would be making the sim-player more closely model human behavior. This removes the problem of people getting better at the game over multiple trials. This could be done by adding a “stress level” to the sim-player so it would become less accurate when it has to simultaneously face multiple enemies. The use of pheromone trails in a 3D environment could also be explored. Pheromone trails in *Ant Hunter* operate in two dimensions in eight set directions. To operate in a 3D environment, the direction vector would need to be able to point in any direction and would also have to account for terrain in the game environment. Lastly, the pheromone trail system could be integrated into a more complex game to see if it is scalable to a larger gaming environment.

Bibliography

Andrade, G., Ramaloh, G., Santana, H., & Corruble, V. (2005). Automatic Computer Game Balancing: A Reinforcement Learning Approach. *AAMAS'05* (pp. 1111-1112). Utrecht, Netherlands: ACM.

Andrade, G., Ramaloh, G., Santana, H., & Corruble, V. (2005). Challenge-Sensitive Action Selection: an Application to Game Balancing. *IEEE/WIC/ACM International Conference on Intelligent Agent Technology.* Compiegne, France: IEEE.

Chen, X., Wang, H., Wang, W., Shi, Y., & Gap, Y. (2009). Apply Ant Colony Optimization to Tetris. *GECCO.* Montreal: ACM.

Coltorti, D., & Rizzoli, A. E. (2007). Ant Colony Optimization for Real-world Vehicle Routing Problems. *SIGEVOlution , 2* (2).

França, F. O., Coelho, G. P., Von Zuben, F. J., & Attux, R. d. (2008). Multivariate Ant Colony Optimization in Continuous Search Spaces. *GECCO* (pp. 9-16). Atlanta: ACM.

Howland, G. (1999, October 12). *A Practical Guide to Building a Complete Game AI: Volume II*. Retrieved October 21, 2009, from GameDev.net: http://www.gamedev.net/reference/articles/article785.asp

Hunicke, R. (2005). The Case for Dynamic Difficulty Adjustment in Games. *Advances in Computer Entertainment Technology* (pp. 429-433). Valencia, Spain: ACM.

Isla, D. (2005, March 11). *GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI*. Retrieved October 21, 2009, from Gamasutra: http://www.gamasutra.com/view/feature/2250/gdc_2005_proceeding_handling_.php

Melo, L. A. (2009). Multi-colony Ant Colony Optimization for the Node Placement Problem. *GECCO.* Montreal: ACM.

Nareyek, A. (2004, February). AI in Computer Games. *Queue* , pp. 58-65.

Orkin, J. (2005). Agent Architecture Considerations for Real-Time Planning in Games. *Artificial Intelligence and Interactive Digital Entertainment.* Marina del Rey, CA: Jeff Orkin.

Orkin, J. (2006). Three States and a Plan: The A.I. of F.E.A.R. *Game Developers Conference.* San Jose: Jeff Orkin.

Pinto, D., & Barán, B. (2005). Solving Multiobjective Multicast Routing Problem with a new Ant Colony Optimization Approach. *Latin America Networking Conference.* Calim Colombia: ACM.

Appendix A: Data Tables

Long Pheromone Trails

Trial	Number of Kills	Number of Deaths	K:D		Trial	Number of Kills	Number of Deaths	K:D
1	42	14	3.000		26	69	42	1.643
2	42	21	2.000		27	80	35	2.286
3	41	22	1.864		28	61	41	1.488
4	55	28	1.964		29	86	53	1.623
5	112	40	2.800		30	123	47	2.617
6	41	23	1.783		31	114	52	2.192
7	67	36	1.861		32	67	41	1.634
8	50	26	1.923		33	77	32	2.406
9	42	19	2.211		34	70	38	1.842
10	54	39	1.385		35	118	57	2.070
11	98	53	1.849		36	97	53	1.830
12	74	40	1.850		37	125	48	2.604
13	95	36	2.639		38	99	47	2.106
14	85	39	2.179		39	108	40	2.700
15	24	16	1.500		40	103	41	2.512
16	70	27	2.593		41	59	38	1.553
17	78	46	1.696		42	80	49	1.633
18	53	16	3.313		43	101	47	2.149
19	68	35	1.943		44	118	58	2.034
20	65	32	2.031		45	78	27	2.889
21	83	43	1.930		46	142	64	2.219
22	66	30	2.200		47	65	41	1.585
23	73	32	2.281		48	81	21	3.857
24	90	25	3.600		49	82	46	1.783
25	132	57	2.316		50	97	54	1.796

TOTAL	4000	1907	2.098

Short Pheromone Trails

Trial	Number of Kills	Number of Deaths	K:D		Trial	Number of Kills	Number of Deaths	K:D
1	39	14	2.786		26	75	31	2.419
2	144	46	3.130		27	34	22	1.545
3	51	21	2.429		28	71	36	1.972
4	85	42	2.024		29	61	45	1.356
5	83	32	2.594		30	117	41	2.854
6	87	27	3.222		31	12	7	1.714
7	53	27	1.963		32	59	34	1.735
8	73	25	2.920		33	115	59	1.949
9	21	10	2.100		34	50	10	5.000
10	97	43	2.256		35	86	31	2.774
11	74	34	2.176		36	53	26	2.038
12	49	26	1.885		37	71	45	1.578
13	22	17	1.294		38	75	39	1.923
14	68	34	2.000		39	69	32	2.156
15	75	28	2.679		40	51	45	1.133
16	42	18	2.333		41	43	27	1.593
17	68	27	2.519		42	89	42	2.119
18	82	27	3.037		43	70	37	1.892
19	39	13	3.000		44	32	15	2.133
20	31	13	2.385		45	46	20	2.300
21	65	28	2.321		46	71	28	2.536
22	20	18	1.111		47	10	7	1.429
23	25	13	1.923		48	45	18	2.500
24	36	22	1.636		49	68	24	2.833
25	99	41	2.415		50	102	35	2.914

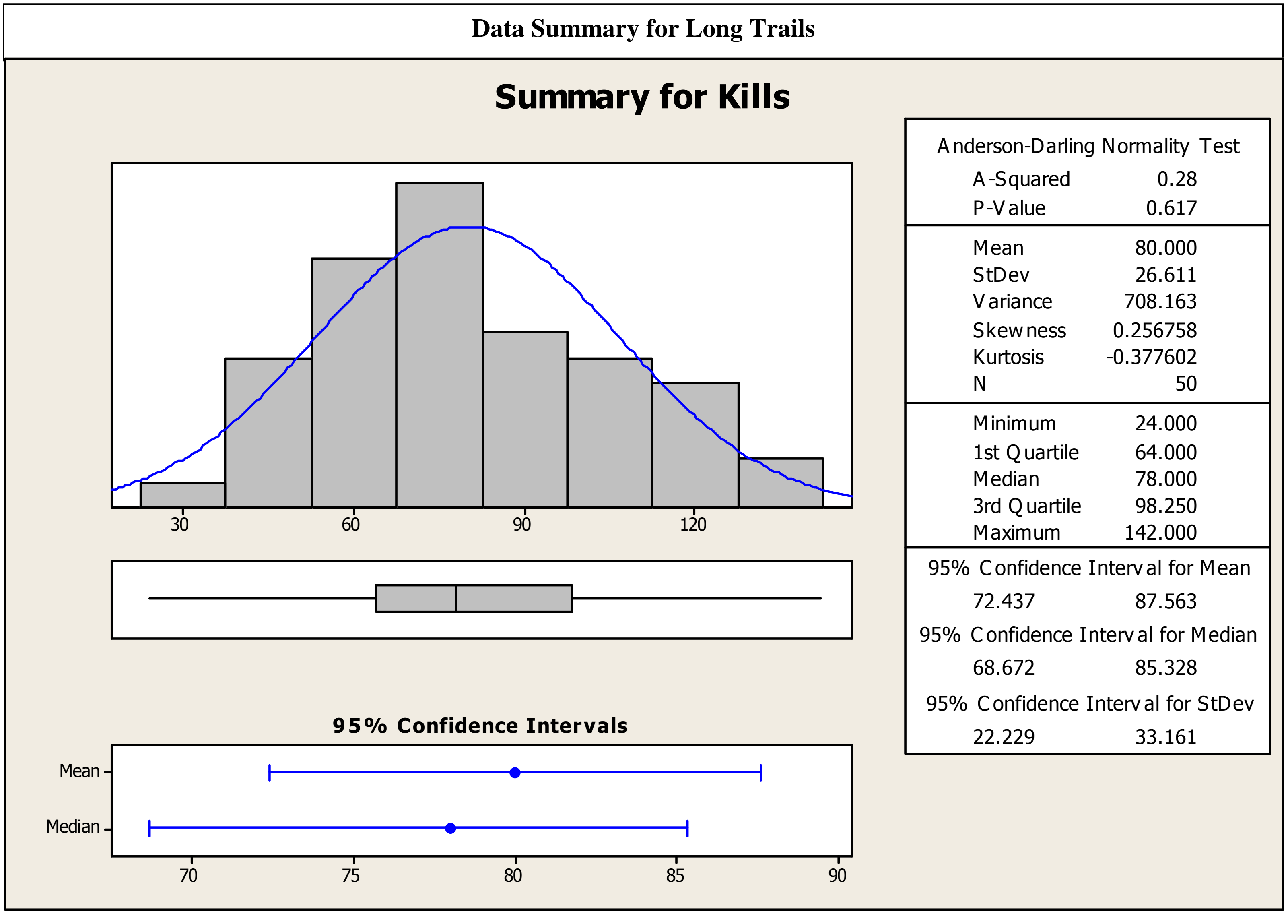
TOTAL	3103	1402	2.213

No Pheromone Trails

Trial	Number of Kills	Number of Deaths	K:D		Trial	Number of Kills	Number of Deaths	K:D
1	21	15	1.400		26	15	8	1.875
2	9	3	3.000		27	41	24	1.708
3	18	16	1.125		28	16	6	2.667
4	25	10	2.500		29	15	10	1.500
5	18	17	1.059		30	23	9	2.556
6	22	11	2.000		31	26	14	1.857
7	20	11	1.818		32	24	16	1.500
8	30	10	3.000		33	34	11	3.091
9	21	12	1.750		34	13	6	2.167
10	16	18	0.889		35	31	7	4.429
11	27	10	2.700		36	17	7	2.429
12	31	10	3.100		37	32	12	2.667
13	17	7	2.429		38	34	22	1.545
14	16	24	0.667		39	16	9	1.778
15	24	11	2.182		40	23	17	1.353
16	22	4	5.500		41	11	8	1.375
17	20	8	2.500		42	29	18	1.611
18	26	16	1.625		43	15	7	2.143
19	24	14	1.714		44	15	3	5.000
20	19	9	2.111		45	30	15	2.000
21	33	17	1.941		46	26	10	2.600
22	41	21	1.952		47	39	6	6.500
23	22	9	2.444		48	23	10	2.300
24	20	14	1.429		49	35	16	2.188
25	31	11	2.818		50	24	8	3.000

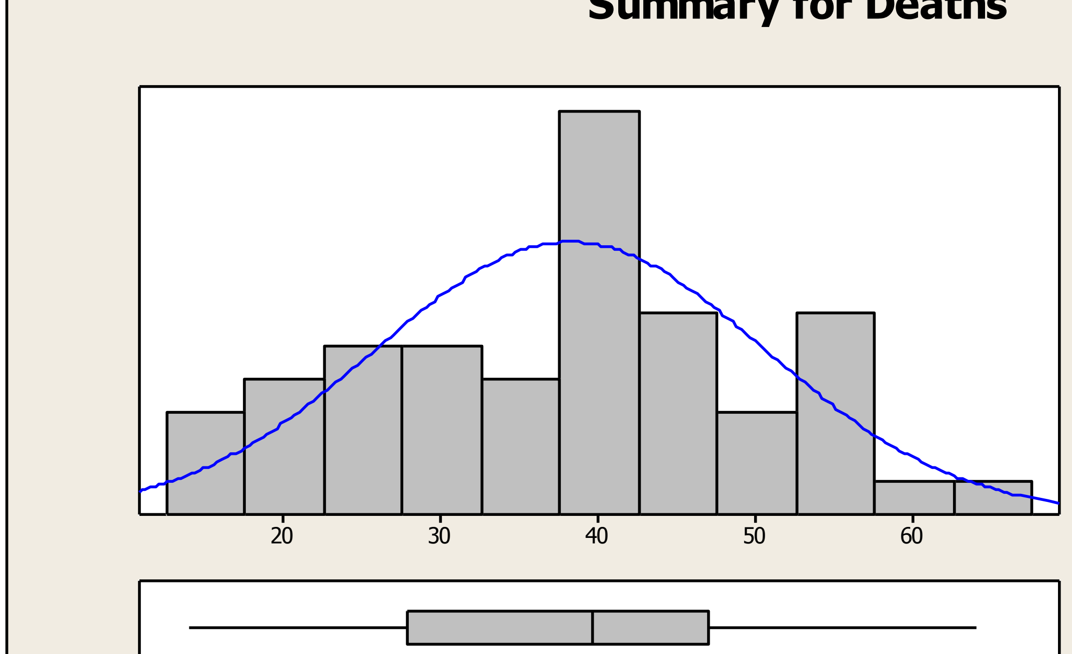
TOTAL	1180	587	2.010

Appendix B: Graphical Data Summaries



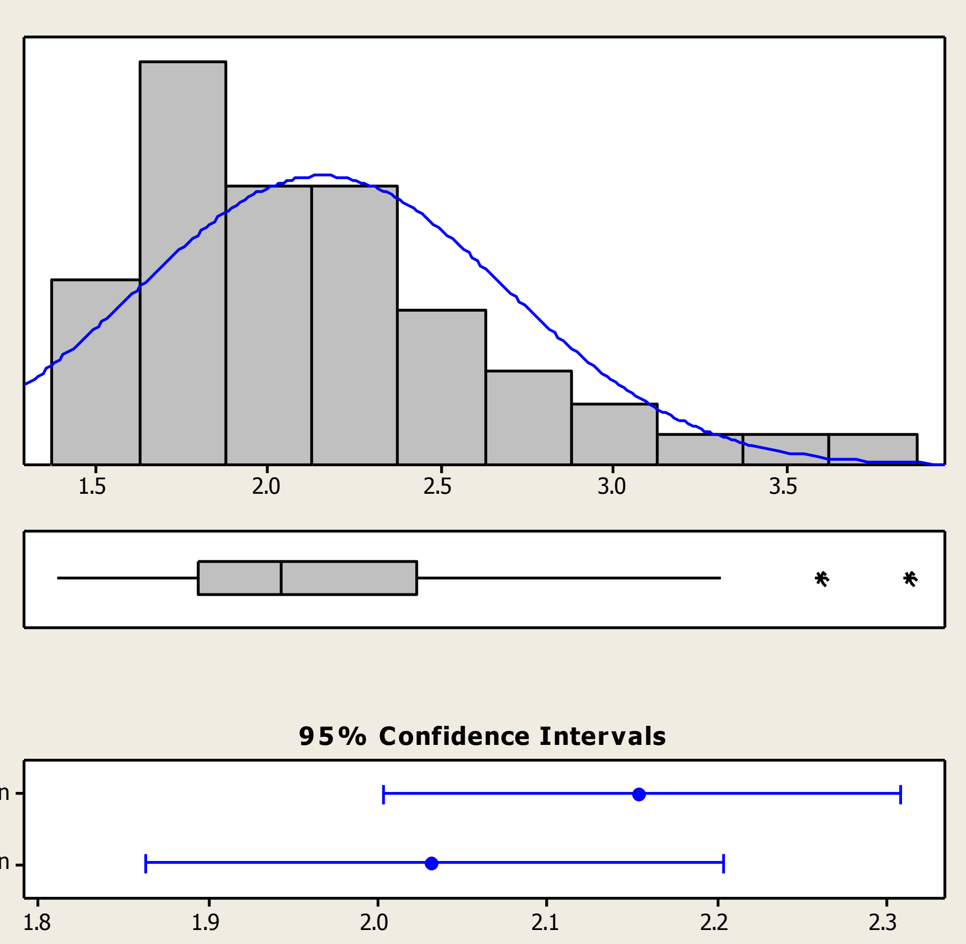
Graphs generated in Minitab 15

Summary for Deaths



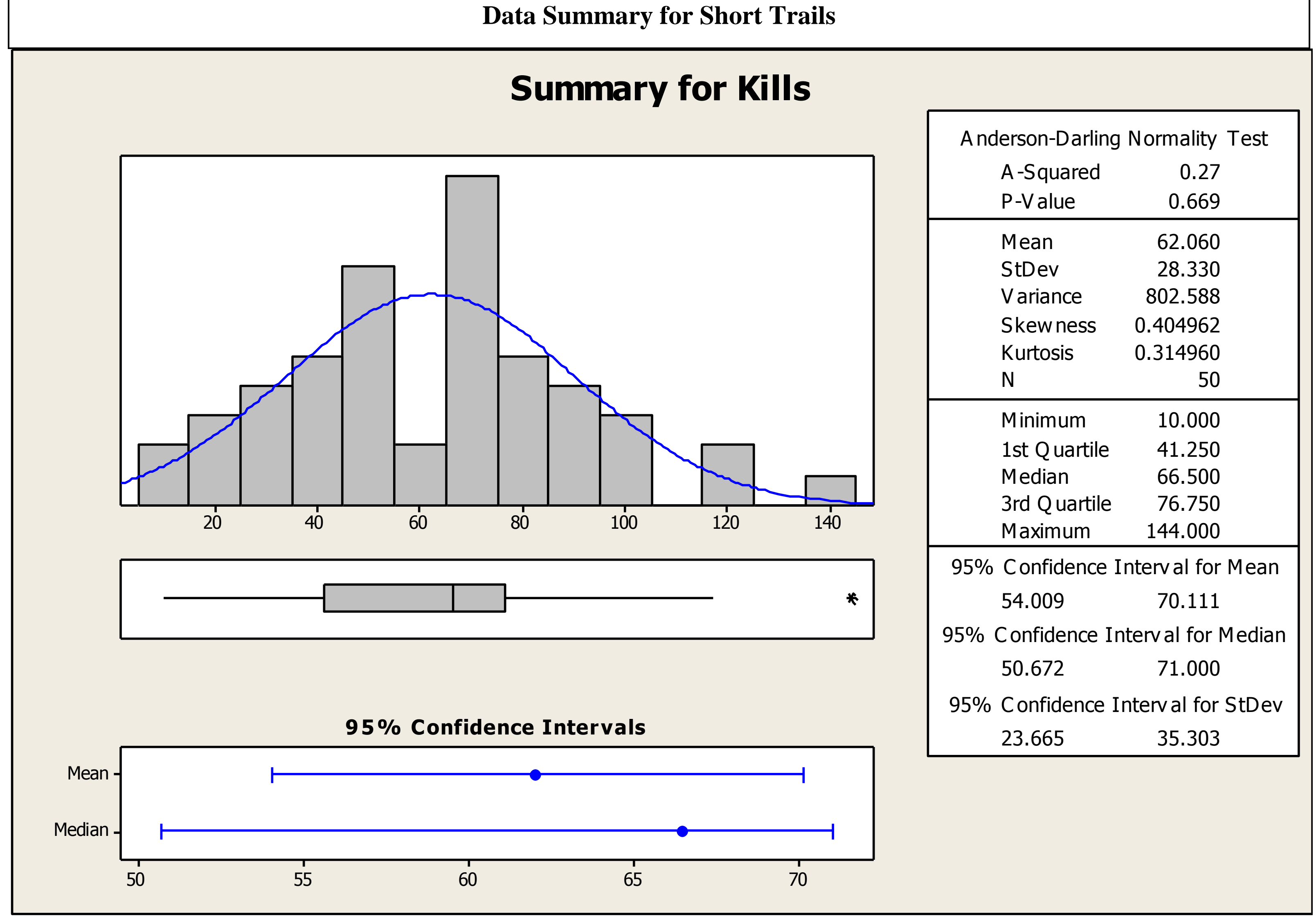
Anderson-Darling Normality Test	
A-Squared	0.29
P-Value	0.586
Mean	38.140
StDev	12.309
Variance	151.511
Skewness	-0.103586
Kurtosis	-0.678581
N	50
Minimum	14.000
1st Quartile	27.750
Median	39.500
3rd Quartile	47.000
Maximum	64.000
95% Confidence Interval for Mean	
34.642	41.638
95% Confidence Interval for Median	
35.000	41.328
95% Confidence Interval for StDev	
10.282	15.339

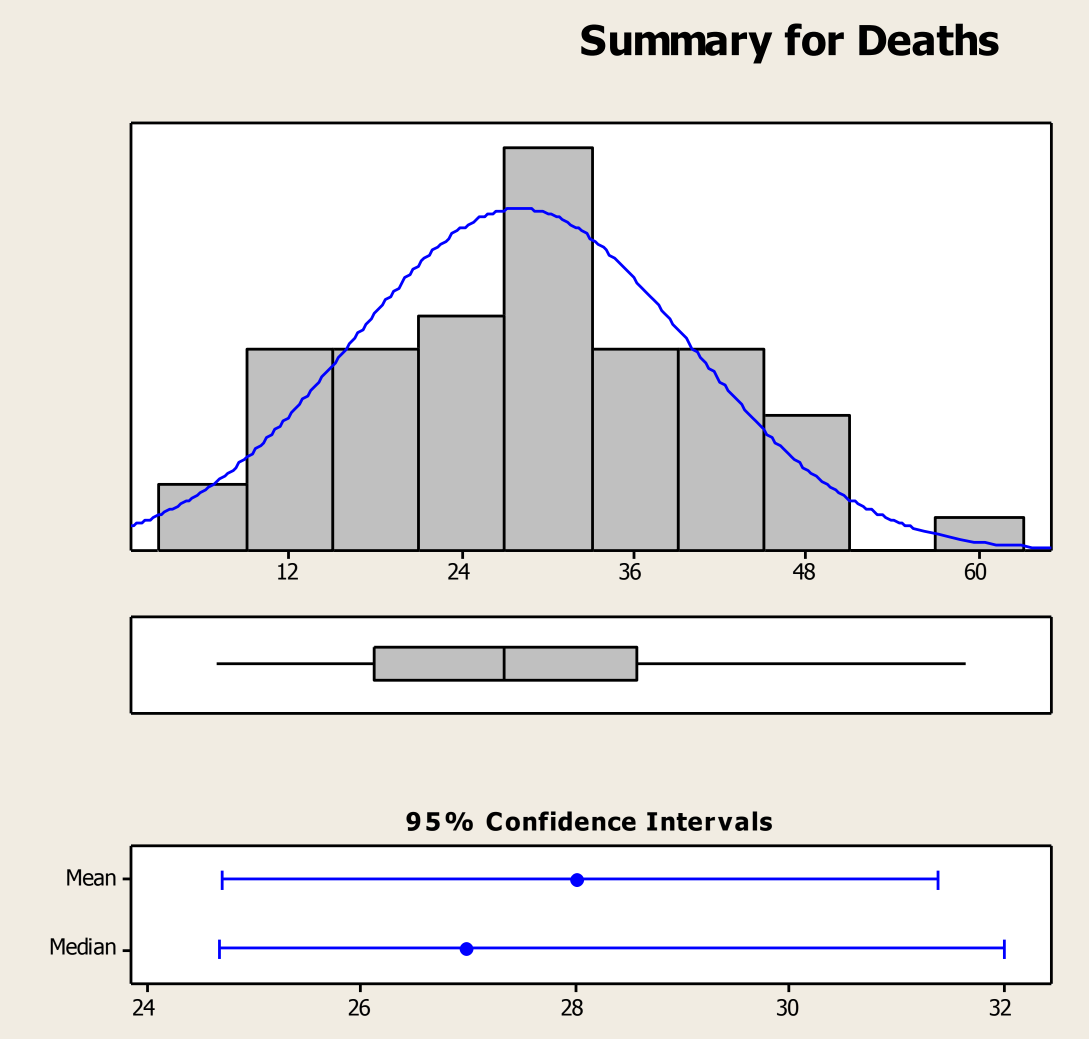
Summary for Kill to Death Ratio

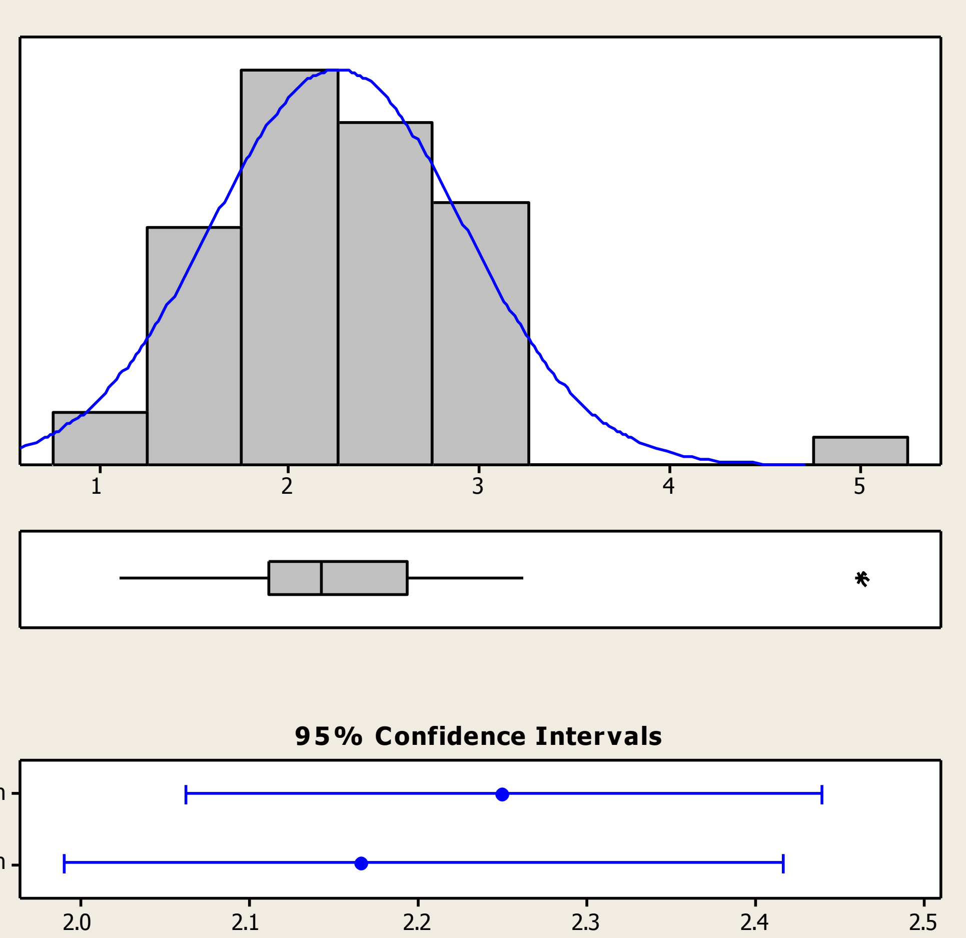


Anderson-Darling Normality Test	
A-Squared	1.25
P-Value <	0.005
Mean	2.1552
StDev	0.5358
Variance	0.2870
Skewness	1.23052
Kurtosis	1.63206
N	50
Minimum	1.3846
1st Quartile	1.7929
Median	2.0329
3rd Quartile	2.4327
Maximum	3.8571
95% Confidence Interval for Mean	
2.0030	2.3075
95% Confidence Interval for Median	
1.8628	2.2035
95% Confidence Interval for StDev	
0.4475	0.6676

Data Summary for Short Trails

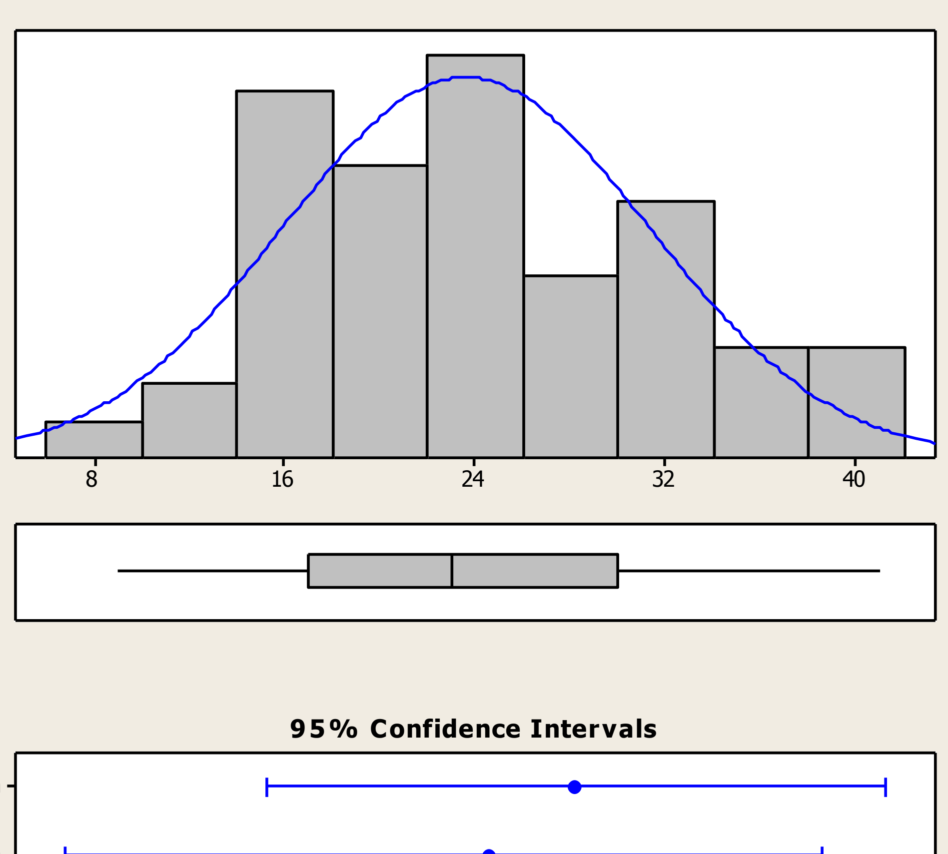



Summary for Deaths
12
24
36
48
60
95% Confidence Intervals
Mean
Median
24
26
28
30
32
Anderson-Darling Normality Test
A-Squared 0.28
P-Value 0.615
Mean 28.040
StDev 11.775
Variance 138.651
Skewness 0.202695
Kurtosis -0.354725
N 50
Minimum 7.000
1st Quartile 18.000
Median 27.000
3rd Quartile 36.250
Maximum 59.000
95% Confidence Interval for Mean
24.694 31.386
95% Confidence Interval for Median
24.672 32.000
95% Confidence Interval for StDev
9.836 14.673

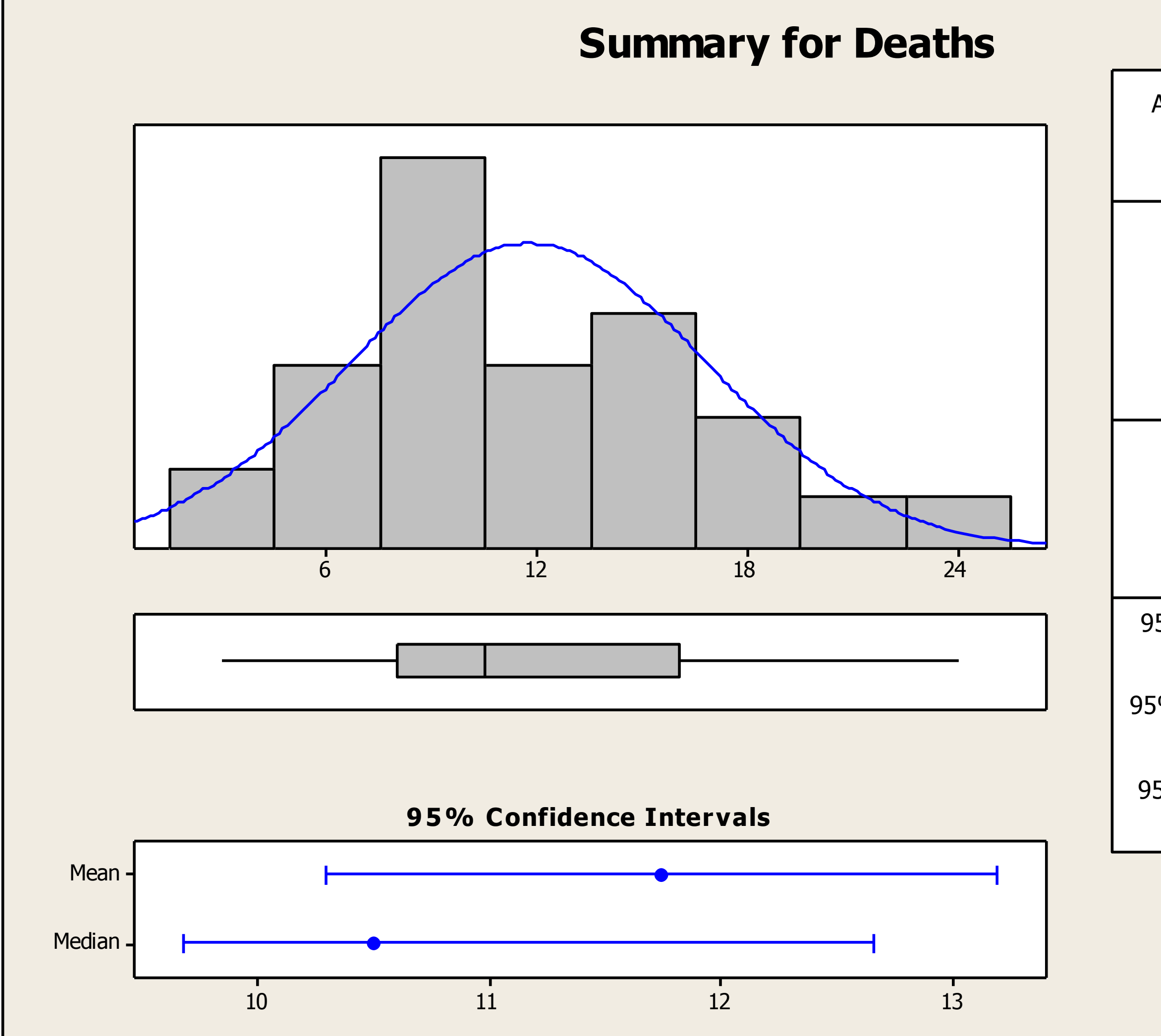

Summary for Kill to Death Ratio
Anderson-Darling Normality Test
A-Squared 0.53
P-Value 0.169
Mean 2.2507
StDev 0.6617
Variance 0.4378
Skewness 1.32214
Kurtosis 4.79731
N 50
Minimum 1.1111
1st Quartile 1.8901
Median 2.1664
3rd Quartile 2.6150
Maximum 5.0000
95% Confidence Interval for Mean
2.0626 2.4387
95% Confidence Interval for Median
1.9909 2.4162
95% Confidence Interval for StDev
0.5527 0.8245
1
2
3
4
5
95% Confidence Intervals
Mean
Median
2.0
2.1
2.2
2.3
2.4
2.5

Data Summary for No Trails

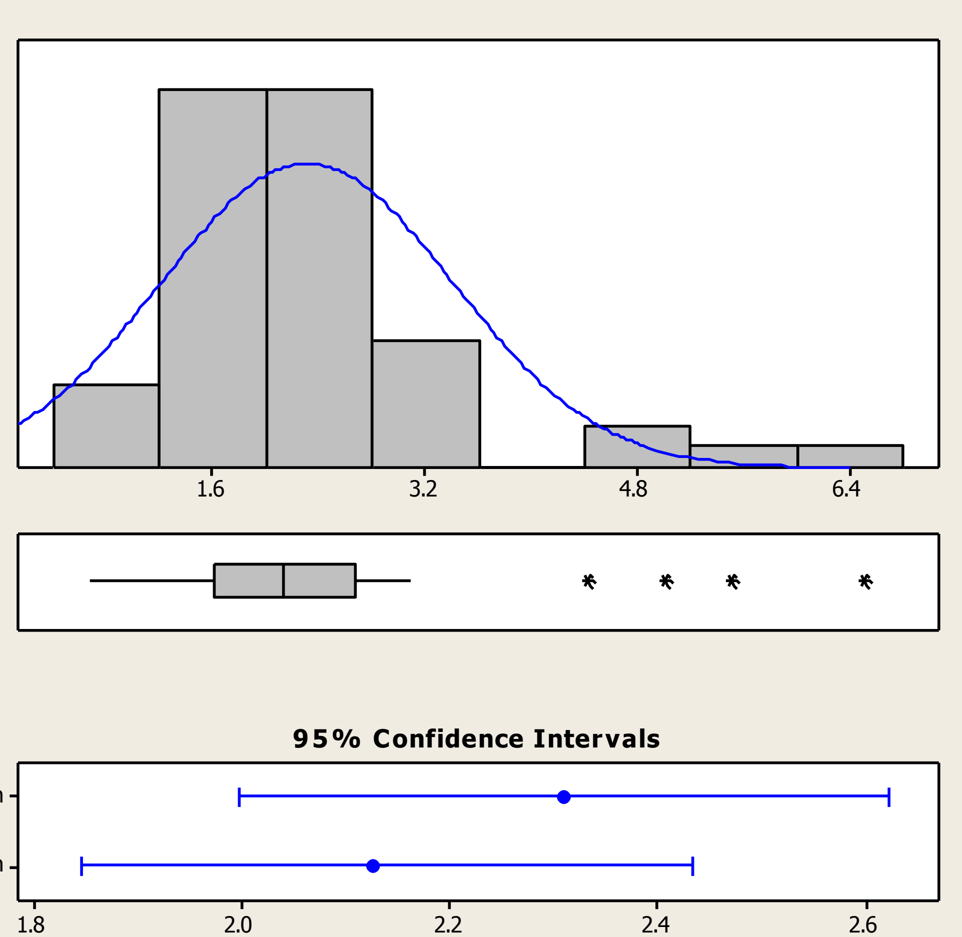
Summary for Kills



Anderson-Darling Normality Test	
A-Squared	0.48
P-Value	0.221
Mean	23.600
StDev	7.690
Variance	59.143
Skewness	0.444432
Kurtosis	-0.347826
N	50
Minimum	9.000
1st Quartile	17.000
Median	23.000
3rd Quartile	30.000
Maximum	41.000
95% Confidence Interval for Mean	
21.414	25.786
95% Confidence Interval for Median	
20.000	25.328
95% Confidence Interval for StDev	
6.424	9.583



Anderson-Darling Normality Test	
A-Squared	0.81
P-Value	0.035
Mean	11.740
StDev	5.098
Variance	25.992
Skewness	0.605967
Kurtosis	-0.044154
N	50
Minimum	3.000
1st Quartile	8.000
Median	10.500
3rd Quartile	16.000
Maximum	24.000
95% Confidence Interval for Mean	
10.291	13.189
95% Confidence Interval for Median	
9.672	12.657
95% Confidence Interval for StDev	
4.259	6.353


Summary for Kill to Death Ratio
1.6
3.2
4.8
6.4
95% Confidence Intervals
Mean
Median
1.8
2.0
2.2
2.4
2.6
Anderson-Darling Normality Test
A-Squared 2.33
P-Value < 0.005
Mean 2.3098
StDev 1.1021
Variance 1.2145
Skewness 1.90382
Kurtosis 4.73480
N 50
Minimum 0.6667
1st Quartile 1.6215
Median 2.1270
3rd Quartile 2.6667
Maximum 6.5000
95% Confidence Interval for Mean
1.9966 2.6230
95% Confidence Interval for Median
1.8444 2.4338
95% Confidence Interval for StDev
0.9206 1.3733