Coexisting Species: Drive Priorities

Matthew Gong
403489585
mattgong@cs.ucla.edu

Abstract
In this project I attempt to explore learning action selection in a multi-motivational system by applying a specific multi-motivational framework to a scenario of warfare, survival, and mating. Learning in this scenario is challenging since the death of creatures makes it necessary to use vicarious learning through observation. Furthermore, action selection in a war environment can be crucial to a species' survival, especially when its motivations are conflicting.

Summary of Motivational Framework and Inspiration
Konidaris and Barto [1] introduce a motivational system design framework for handling a situation when an agent has multiple, possibly conflicting motivations. The framework consists of many drives, each representing an individual goal of the agent. Each drive contains a satiation level and a priority parameter (which maps to a specific priority curve). To obtain the reward of a drive after an action, one would take the difference in satiation level and multiply it by the satiation level's mapping onto the priority curve. The reward of the action across multiple drives is the summation of individual rewards. The agent's goal at any given time is to maximize this total reward through its actions.

I was particularly interested in the framework's robustness, since the drive levels are numerically comparable and the drive priorities are highly precise. Konidaris and Barto's experiment, giving a single agent drives for two types of resources, inspired me to create a more complex environment with two types of species with identical drives. I wanted to give the species drives that were not traditional "consumption" goals, instead giving them goals that made modeling the satiation effects more challenging.

Hypothesis
I intended to apply Konidaris and Barto's framework to two combating species, a type of spider with superior fighting skills/strength and a type of ant with superior speed and agility. I intended to show that using the framework, specifically the adjustment of drive priority curves, would allow the two species to coexist.

More specifically, I engineered the spiders to be slower but stronger than the ants, and the ants to be faster but weaker than the spiders. I engineered both species to have the same independent motivations: increasing their population and destroying the other population. I did not want to bias their priorities initially, instead relying on learned adjustments of drive priorities to reach equilibrium. I predicted that with a proper system to update priorities, the spiders would prioritize destroying ants, while the ants would prioritize increasing population.

Implementation

This program was created in C++ using OpenGL (graphics) and GLUT (GL Utility Toolkit) for windowing. Refer to Appendix for code.

World
100x100 unit 2-D board with Cartesian coordinates and boundaries.

Species
There are two types of species in the world:

- Spider
  - 3 times the strength of an ant. More details in the Actions section.
- Ant
  - 2 times the speed of a spider, giving them a slight edge in mating.

It should be noted that these species were picked arbitrarily and do not represent real spiders or ants. These creatures have no physical size in the world (each creature contains a single location).

Drives
Each creature (from both species) was given two drives:

- Increase Population (the desire to mate)
- Destroy the Other Population (the desire to fight)

Each drive contains a satiation level and a priority parameter according to motivational framework.

Actions
The only two actions of both species are "fighting" and "mating". For both actions, they begin by choosing a random location on the grid as a target and then walk in a straight line towards the target. In the case of fighting, if a member of the opposite species is encountered (within a distance threshold of 3 units) who is also looking to fight, they will engage in fighting. Similarly, if two of the same species are looking to mate and cross paths, they will mate (no sex assignment). If an opponent or mate is not encountered before reaching the target, a new random target is selected on the grid.

Only two participants are allowed to fight at one time. The spiders start their lives with full health and an ant will inflict 1/3 damage on its health during a fight. If an ant engages a fight with a spider at 1/3 health, the spider will die and the ant will be unharmed. Otherwise, the ant will die and the spider will have its health decrease by 1/3.

Mating will produce another creature instantly. It was necessary for me to add a period of time where a creature could not mate if it had just mated or was just born. This would allow the creatures to walk away without mating again.

Perception
Each creature has a perception range of 30x30 units (9% of the board). In this range they can perceive the following items:

- Size of own population
- Size of other population
- Witness birth of same/other species
- Witness death of same/other species

The first two items are observed on the spot, whereas the latter two are counted during the creature's learning intervals. When the creature's learning interval is complete, the counts are reset.

Drive Process – Updating Satiation
I update the satiation levels of the creatures based on the following:

- Whenever a creature mates, its increase population satiation rises by:
  - (α / same population size)
- Whenever a creature witnesses a death of its own species, its increase population satiation drops by:
  - (β / same population size)
- Whenever a creature kills another creature, its destroy satiation rises by:
  - (α / other population size)
- Whenever a creature witnesses a foreign birth, its destroy satiation drops by:
  - (β / other population size)

α and β are constants and $\alpha >> \beta$. Satiation adjustments are based on population sizes to promote mating and fighting when it would make the largest effect on the population sizes. For instance, a creature that mates when its population size is 3 should be much happier than when its population size is 100.

Action Selection
A creature will always choose the action (mating or fighting) that will generate the max reward at the given time. This reward is calculated using the framework's reward function and depends on the drive levels and observed population sizes.

Priority Process – Learning priority parameters
All creatures periodically learn through slight priority parameter adjustments over time. The following formulas were used for each drive:

- Increase Population Drive
  - γ (Deaths^2 – Births^2)
- Destroy Drive
  - γ (ForeignBirths^2 – Kills^2)

γ is a constant. The quantities above are all observed in the creature's perception range and are squared to produce larger adjustments for learning periods where more data was sampled. In the case of more samples, the data is a better representation of the target priorities the creature is aiming for.

This project uses a Lamarckian evolution system where each child inherits the drive priority parameters from the oldest parent (who would have the priority curve which best reflects the environment).

Experiments (I/O and Results)
Here are the input parameters to the program:

- Number Of Ants
- Number Of Spiders
- Action Selection Type (REWARD, FIFTY_FIFTY)
- Personal Mating/Fighting Satiation Constant (α)
- Observed Mating/Fighting Satiation Constant (β)
- Learning Type (LAMARCKIAN, NONE)
- Learning Time (If Lamarckian, time for a learning adjustment)
- Priority Adjustment Constant (γ)
- Teaming-Up (OFF, ON)

For all experiments I used the following fixed parameters over a period of 50,000 time units (1 time unit = 1 animation cycle):

- 100 ants and 100 spiders randomly configured on the board
- $\alpha = 3.0$
- $\beta = 0.2$
- $\gamma = 0.0005$
- Learning Time = 250 (time units)

I ran three different types of experiments:
(Note: for all graphs [in Appendix], the satiations and priority parameters were taken from the average of the populations)

1) 50/50% action selection
This was my base experiment where the creatures did not use the drives and selected mating or fighting 50% of the time. This demonstrated the advantage the spiders had over the ants initially. Figure 1 (Appendix) shows that the spiders immediately dominate over the ants in population size.

2) Fixed Drive Priorities
In this experiment, the drives were used (including priorities), but the priorities were kept fixed. In this case, the drive with the lowest satiation level will attempt to be satiated, due to the use of the priority curves. Figure 2a shows the population sizes over time where the ants have survived better than before, but are still destroyed by the spiders over a length of time. Figure 2b shows the satiation levels of the two species. The ants' satiation level for mating is much higher than its satiation level for destroying, since the ants have a smaller population size and are generally "happier" with each mate. However, in response to the lower destroy satiation level, it will choose to fight more than mate. As it turns out, this is not the best strategy for the ants' survival.

3) Adjusting Drive Priorities

The final experiment used periodically adjusting drive priorities. Figure 3a shows that after a slight oscillation in population sizes, the ants have learned to survive with the spiders. The equilibrium of population sizes of both species is roughly proportional to the physical dominance the spiders have over the ants. Figure 3b shows similar satiation level patterns as in the previous experiment. Figure 3c shows the priority parameters adjusting over time. Note that the ants have now prioritized mating much higher than destroying, since it was recognized as a harder drive to satiate.

Sensitivity of Parameters/Formulas

The α, β, and γ constants (input parameters to the experiment) were highly sensitive and greatly affected the outcome of the experiment. The most challenging aspect when selecting the parameters was to keep the populations from growing out of control. Using many different permutations of these constants, the ending result would always be ants and spiders growing in size exponentially. For instance, when the γ constant was too high, the priority parameters adjustments were too great and caused the system to "break down", triggering exponential growth. Similarly, if α and β were not designed to keep the satiation levels relatively stable, the experiment would "break down" yet again. I found the best combination of the three parameters was: $\alpha = 3.0$, $\beta = 0.2$, $\gamma = 0.0005$.

However, the formulas for adjusting the satiation levels and priority parameters were the most sensitive of all. I underestimated the importance of these formulas at first and found that in many cases, no constants would keep the population under control. After trying several formulas and arriving at the final versions for the drive and priority processes, the rates of deaths and births had stabilized.

Teaming Up

Based on a suggestion, I incorporated a new experiment in addition to the ones above. I added "teaming up" behavior in the creatures where they would observe when a member of their species was fighting and attempt to help them. Whenever a creature had been killed in battle, the neighboring creatures of the victim (within the perception range) who were looking to fight would redirect their walking towards the enemy who had just killed the victim. This add-on was designed specifically for the ants since they would benefit the most from attacking the spiders that were wounded from battle.

This produced some interesting behavior, as shown in Figure 4. Naturally, the ants and spiders that were looking to fight began to pack together and fight in large groups. However, the population sizes of both creatures would oscillate for a moderate amount of time, but mysteriously grow exponentially as the system appears to "break down" as it did before. Figure 4c shows the wild swings in priority parameter adjustments during this exponential growth phase. It is difficult to determine what caused this wild swing, but I reasoned that the ants had now reduced their disadvantage enough to overtake the spiders. It was interesting that a seemingly minor change in behavior would affect the results so dramatically.

Conclusion

I wanted to show two things in the experiment - that the spiders would prioritize destroying ants, and that the ants would prioritize increasing in population. The experiment only showed the latter to be true. In fact, the spiders only decreased their priority parameters for the destroy drive, which was unexpected.

Another surprising aspect of this project was that it was so difficult to keep the populations from growing exponentially. At first I reasoned that it is generally easier for a mate to occur than a fight, but after several experiments I realized that the most important factor in the growth patterns was the adjustments to the priorities. When adjusting the priority parameters, I squared the values of births and deaths for two reasons. One, it would place emphasis on multiple samples extracted as a more reliable representation of the environment. Secondly, it helped to control the population because as the population grew, the priority parameter adjustments became larger and helped to prevent the system from a "break down". I concluded that the larger the population sizes, the larger the priority adjustments needed to be for a stable system.

Finally, I concluded that the addition of the "team-up" behavior reduced the ants' disadvantage enough to allow them to compete with the spiders for equal population sizes.

References

1. G. Konidaris and A. Barto. (2006). An Adaptive Robot Motivational System. In From Animals to Animats 9: The Ninth International Conference on the SIMULATION OF ADAPTIVE BEHAVIOR (SAB'06)
URL: http://www-anw.cs.umass.edu/pubs/2006/konidaris_b_SAB06.pdf

Appendix

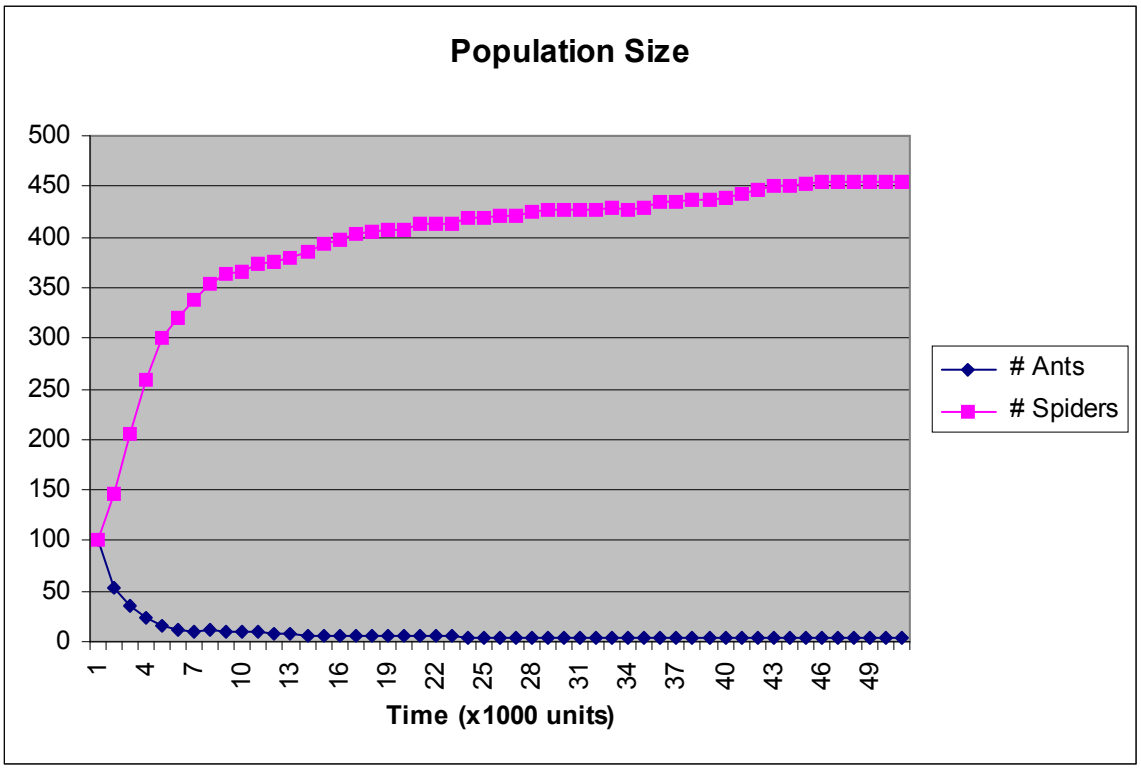
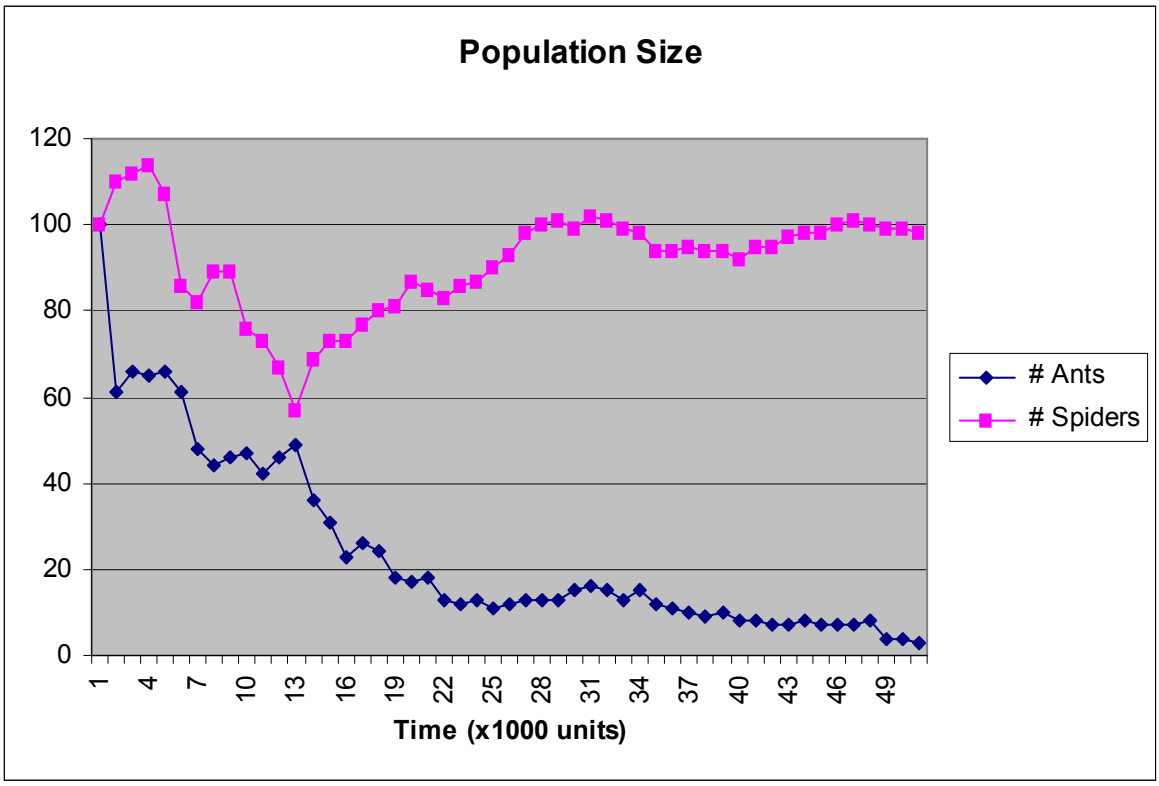
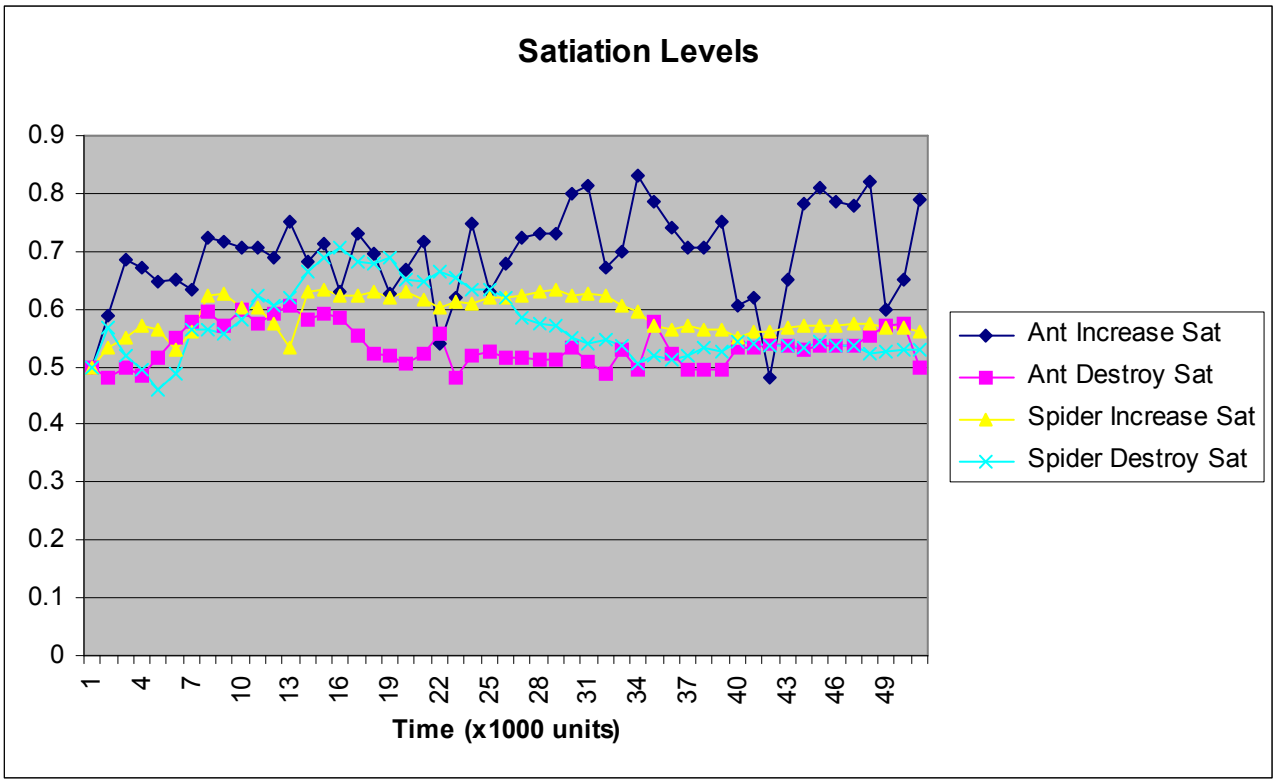


Fig 1. Population sizes of ants and spiders over time when selecting mating/fighting 50% of the time. The spiders quickly dominate over the ants.

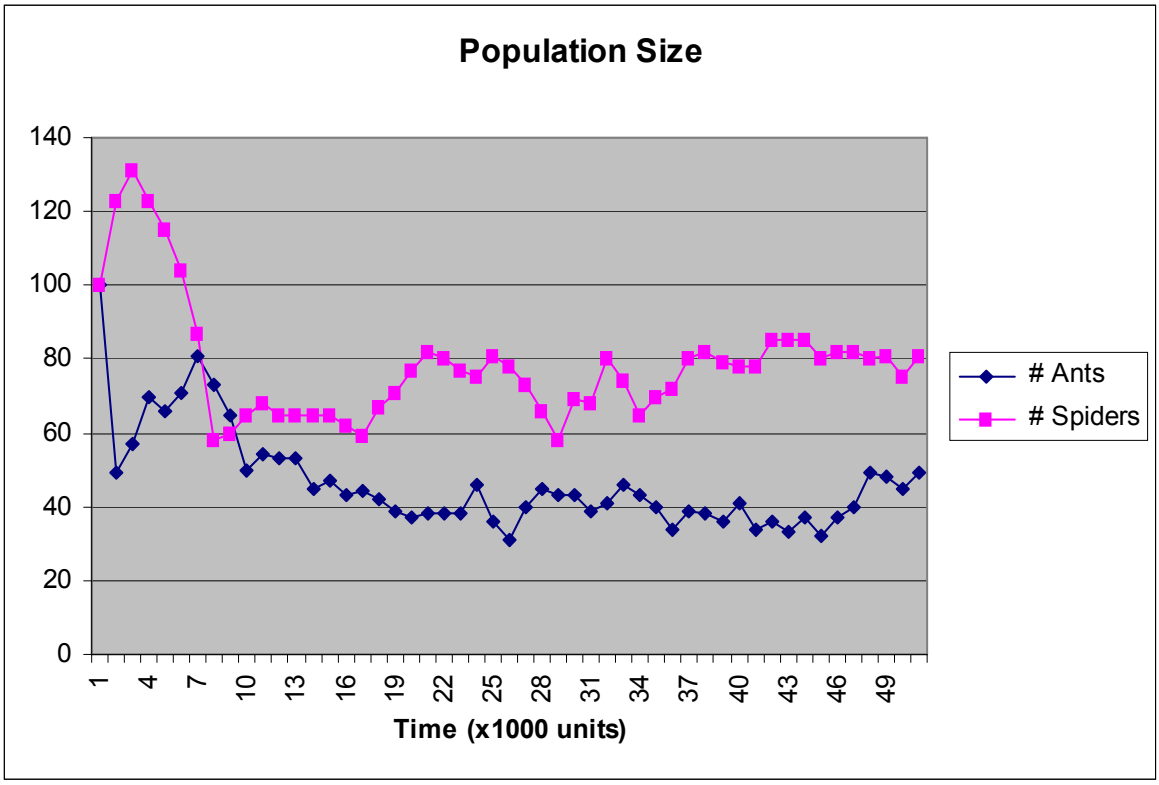

Population Size
120
100
80
60
40
20
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
Time (x1000 units)
Ants
Spiders

(a)

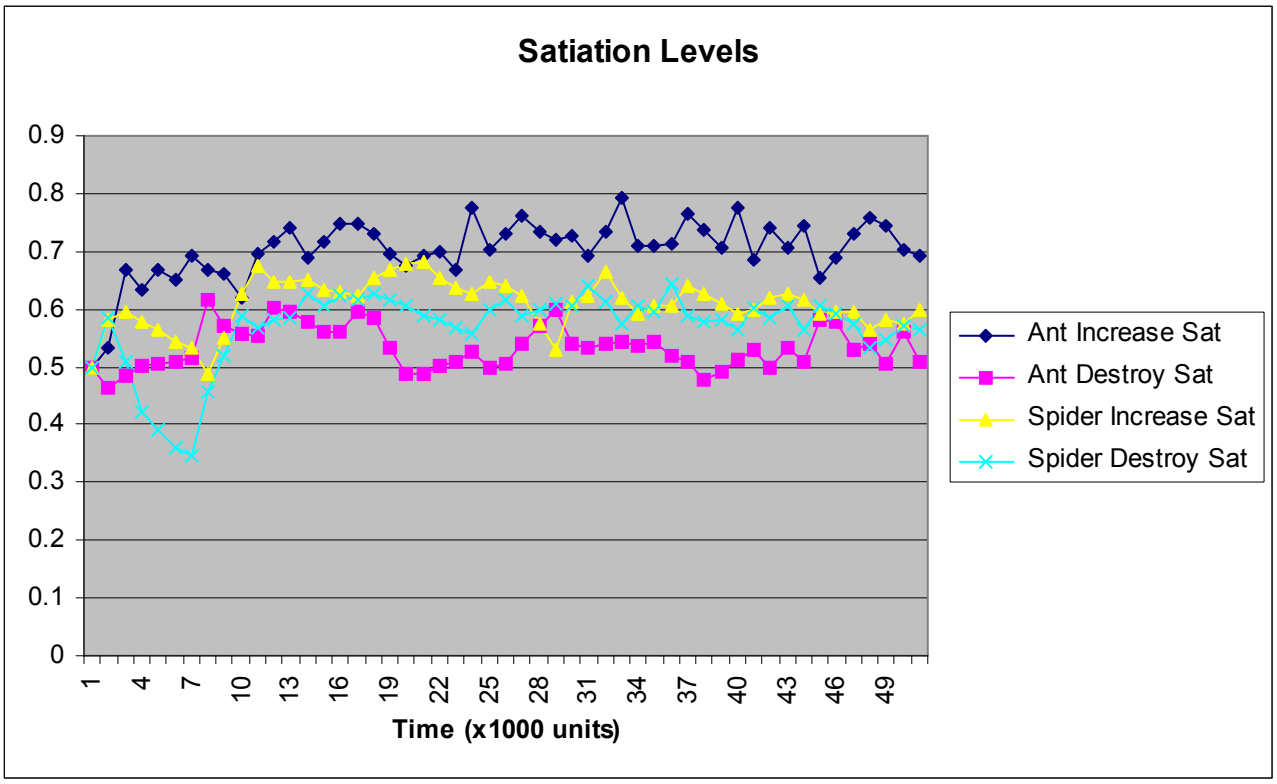

Satiation Levels
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
Time (x1000 units)
Ant Increase Sat
Ant Destroy Sat
Spider Increase Sat
Spider Destroy Sat

(b)

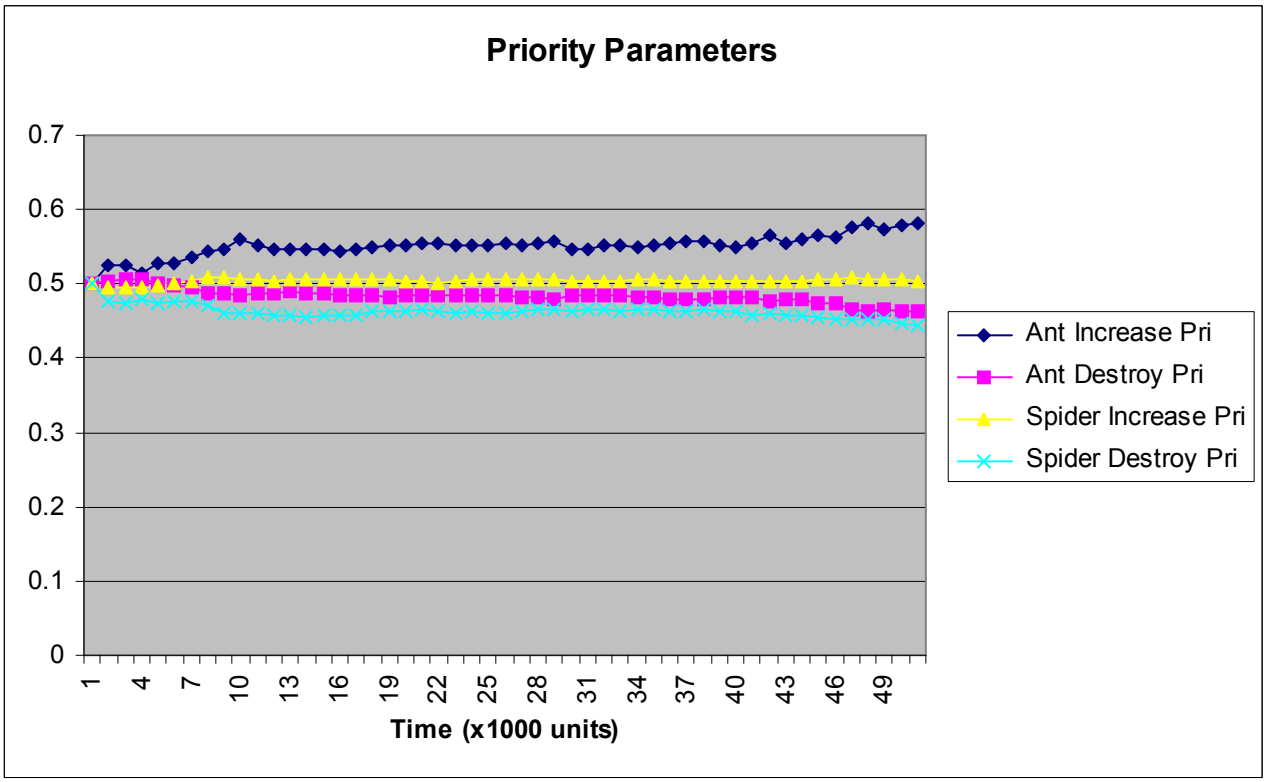
Fig 2. Experiment results for fixed priority levels. (a) shows the population size of ants and spiders and (b) shows average satiation levels over time. Even with drive levels, the spiders eventually destroy the ants.



(a)

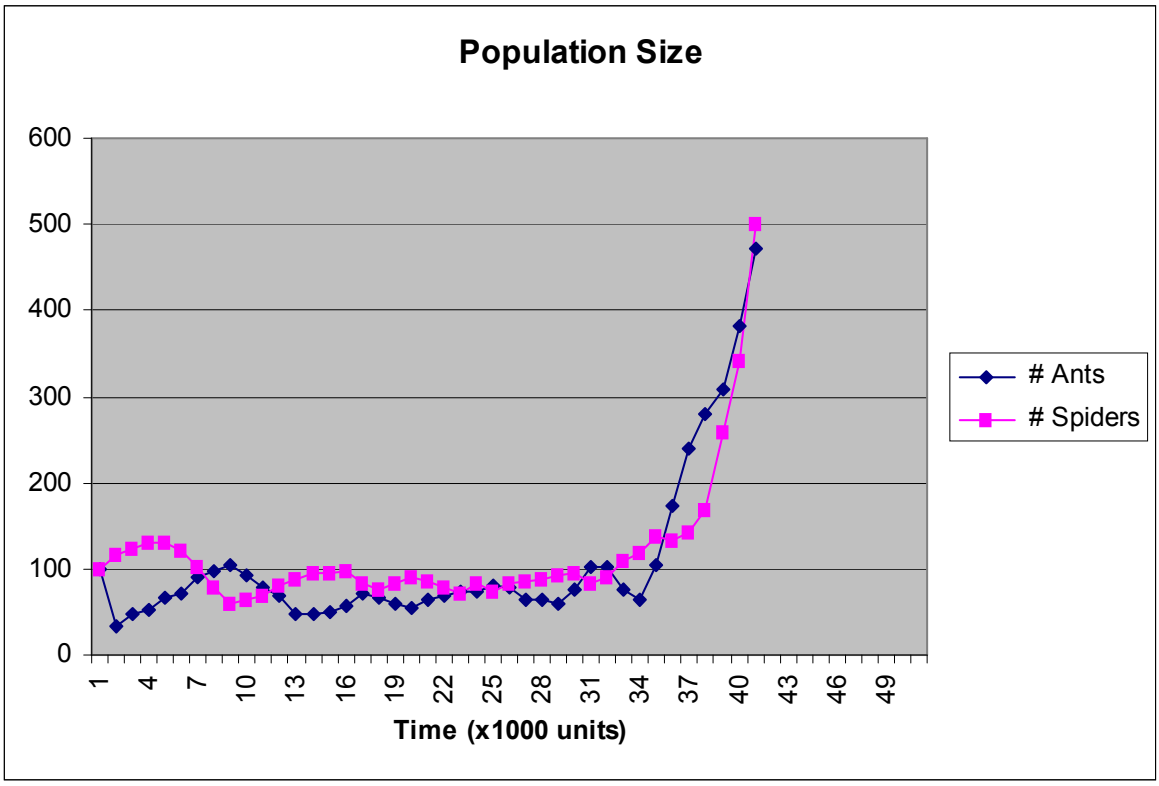

Satiation Levels
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
Time (x1000 units)
Ant Increase Sat
Ant Destroy Sat
Spider Increase Sat
Spider Destroy Sat

(b)

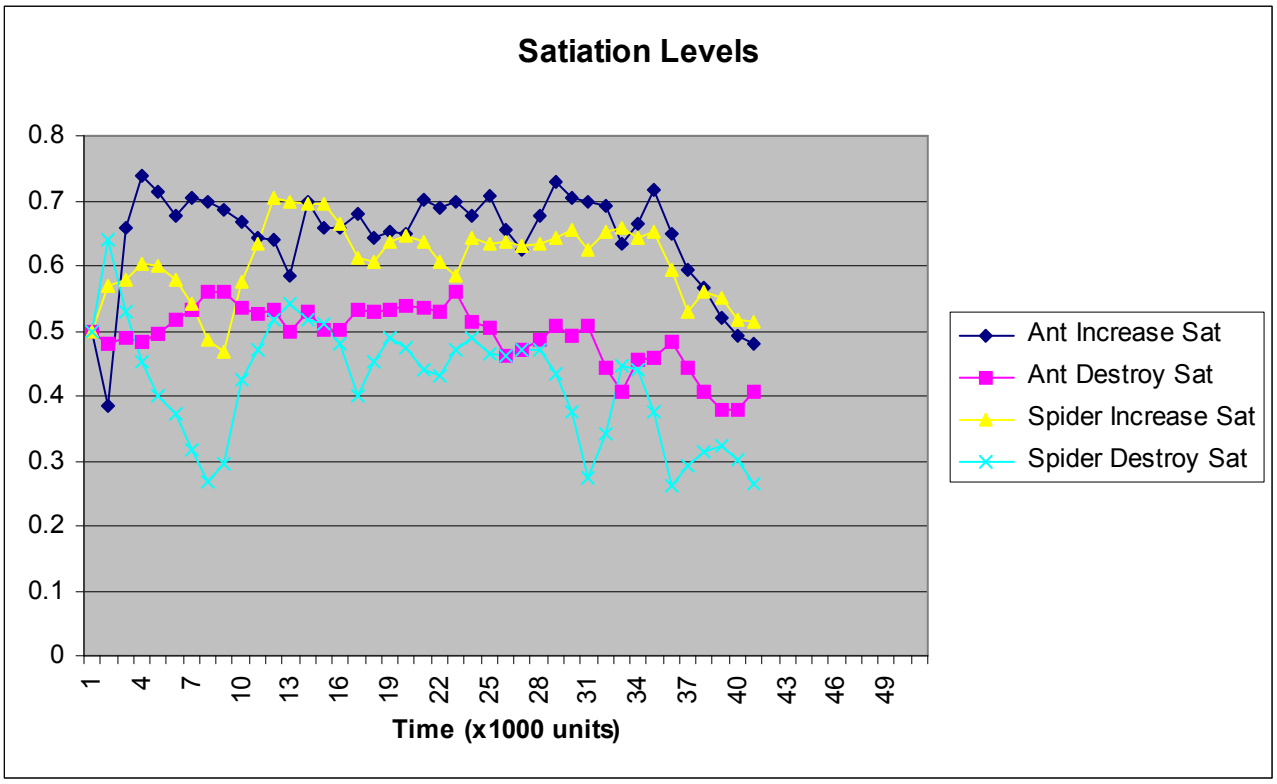

Priority Parameters
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
Time (x1000 units)
Ant Increase Pri
Ant Destroy Pri
Spider Increase Pri
Spider Destroy Pri

(c)

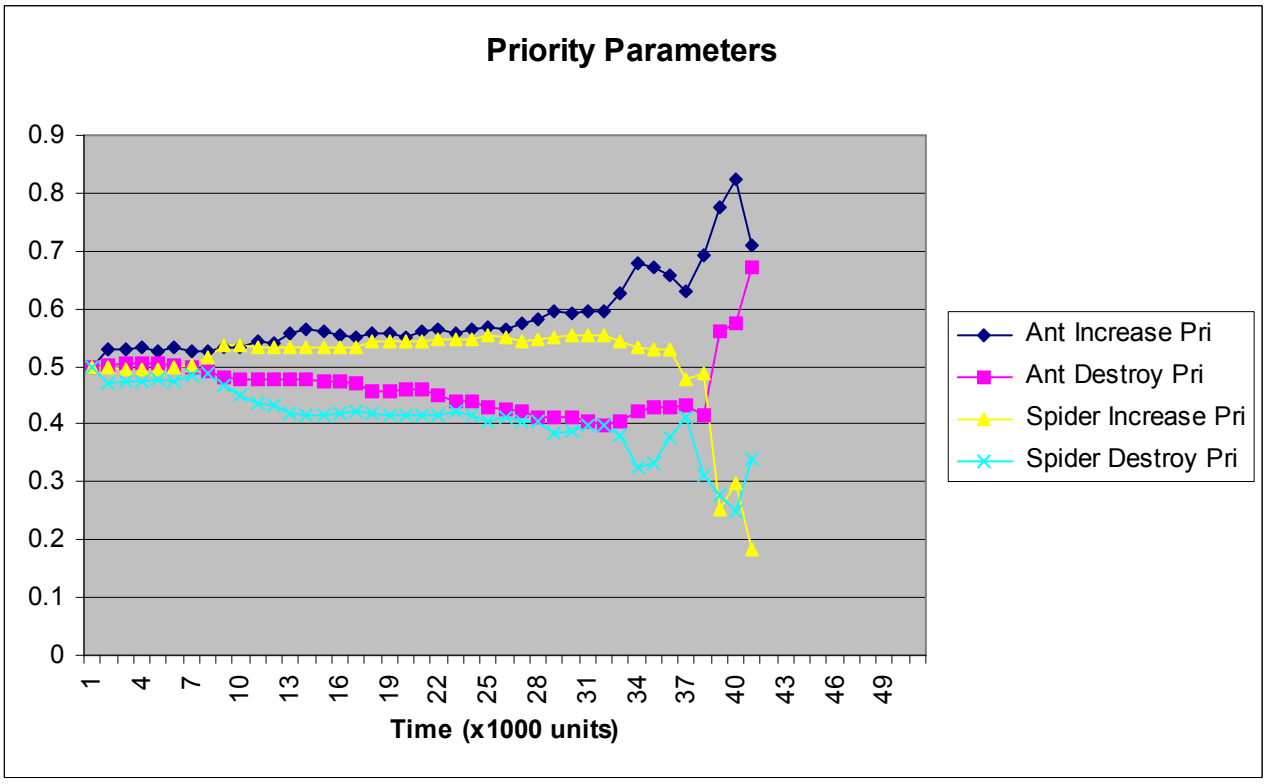
Fig 3. Experiment results for adjusting priority parameters. (a) shows the population size of the ants and spiders, (b) shows the average satiation levels, and (c) shows the average priority parameters over time. The ants can now survive with the spiders by increasing the priority of mating, recognizing it as a harder drive to satiate.



(a)


Satiation Levels
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
Time (x1000 units)
Ant Increase Sat
Ant Destroy Sat
Spider Increase Sat
Spider Destroy Sat

(b)


Priority Parameters
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1
4
7
10
13
16
19
22
25
28
31
34
37
40
43
46
49
Time (x1000 units)
Ant Increase Pri
Ant Destroy Pri
Spider Increase Pri
Spider Destroy Pri

(c)

Fig 4. Experiment results for "teaming up" behavior. (a) shows population size of ants and spiders, (b) shows average satiation levels, (c) shows average priority parameters over time. The two population sizes oscillate at first and then exponentially increase once it encounters a certain threshold.

Code

Below are the main C++ classes. I have omitted the base program, including graphics and windowing code. I have also omitted code for my tiling system (for creature perception and collision detection), the point class (for geometric computation), and the random number generator.

DRIVE.H

```
#ifndef DRIVE_H
#define DRIVE_H

/**
 ** Drive class - represents a single drive (motivation) of a creature
 **/

class Drive
{
public:

    // Constructor, destructor
    Drive();
    virtual ~Drive();

    // Get, Set satiation level
    double
    getSatiation() const;

    void
    setSatiation( double satiation );

    // Get, Set priority parameter
    double
    getPriorityParameter() const;

    void
    setPriorityParameter( double priority_parameter );

    // Calculate the reward if the satiation were to change to the
specified amount
    double
    calculateReward( double new_satiation ) const;

private:
```

```
    double m_satiation;             // Satiation Level [0, 1]
    double m_priority_parameter;    // Priority Parameter [0, 1]
(specifies priority curve)

};

#endif
```

DRIVE.CPP

```
#include "Drive.h"
#include "Common.h"
#include <cmath>

//---------------------------------------------------------------
Drive::Drive()
:   m_satiation (0.5),
    m_priority_parameter (0.5)
{
    // Constructor
}

//---------------------------------------------------------------
Drive::~Drive()
{
    // Destructor
}

//---------------------------------------------------------------
double
Drive::getSatiation() const
{
    return m_satiation;
}

//---------------------------------------------------------------
void
Drive::setSatiation( double satiation )
{
    if ( satiation < 0.0 ) {
        m_satiation = 0.0;
    }
    else if ( satiation > 1.0 ) {
        m_satiation = 1.0;
    }
    else {
        m_satiation = satiation;
    }
}

//---------------------------------------------------------------
double
Drive::getPriorityParameter() const
{
    return m_priority_parameter;
}
```

```
//---------------------------------------------------------------
void
Drive::setPriorityParameter( double priority_parameter )
{
    if ( priority_parameter < 0.0 ) {
        m_priority_parameter = 0.0;
    }
    else if ( priority_parameter > 1.0 ) {
        m_priority_parameter = 1.0;
    }
    else {
        m_priority_parameter = priority_parameter;
    }
}

//---------------------------------------------------------------
double
Drive::calculateReward( double new_satiation ) const
{
    // Calculate the reward if the satiation were to change to the
specified amount

    if ( new_satiation < 0.0 ) {
        new_satiation = 0.0;
    }
    else if ( new_satiation > 1.0 ) {
        new_satiation = 1.0;
    }

    // (From Konidaris and Barto's paper)
    // The reward is calculated by multiplying the difference in
satiation level
    // by the drive priority.  The drive priority is obtained by
mapping the
    // previous satiation level onto the priority curve.  The equation
for this is:
    // drive priority = 1 -
(new_satiation)^[tan(priority_parameter*PI/2)]

    double satiation_difference = new_satiation - m_satiation;

    double drive_priority = 1 - pow( m_satiation,
tan(m_priority_parameter*M_PI/2) );

    return satiation_difference * drive_priority;
}
```

CREATURE.H

```
#ifndef CREATURE_H
#define CREATURE_H

#include "Common.h"
#include "Point3D.h"
#include "Drive.h"
```

```
#define TARGET_THRESHOLD 0.5
#define HIT_TOLERANCE 1.5
#define PAUSE_TIME 40
#define JUST_MATED_TIME 100


enum creature_type
{
    ANT_TYPE = 0,
    SPIDER_TYPE
};


class TileCache;
class MTRand;


/**
 ** Creature class - represents one creature
 **/

class Creature
{
public:

    // Constructor, destructor
    Creature();
    virtual ~Creature();

    // Get the type of creature
    virtual creature_type
    getType() const = 0;

    // Get, Set location
    const Point3D&
    getLocation() const;

    void
    setLocation( const Point3D& location );

    // Get, Set action
    action_type
    getAction() const;

    void
    setAction( action_type action );

    // Get, Set target
    const Point3D&
    getTarget() const;

    void
    setTarget( const Point3D& target );

    // Get, Set target set
    bool
    getTargetSet() const;
```

```
void
setTargetSet( bool target_set );

// Get, Set timer
int
getTimer() const;

void
setTimer( int timer );

// Get, Set health
double
getHealth() const;

void
setHealth( double health );

// Get, Set secondary creature
bool
getSecondaryCreature() const;

void
setSecondaryCreature( bool secondary_creature );

// Get, Set other creature
Creature*
getOtherCreature() const;

void
setOtherCreature( Creature* other_creature );

// Get, Set just mated timer
int
getJustMatedTimer() const;

void
setJustMatedTimer( int just_mated_timer );

// Get, Set the drives
const Drive&
getIncreasePopulationDrive() const;

void
setIncreasePopulationDrive( const Drive& drive );

const Drive&
getDestroyDrive() const;

void
setDestroyDrive( const Drive& drive );

// Get, Set # of births
int
getBirths() const;

void
```

```
    setBirths( int births );

    // Get, Set # of deaths
    int
    getDeaths() const;

    void
    setDeaths( int deaths );

    // Get, Set # of foreign births
    int
    getForeignBirths() const;

    void
    setForeignBirths( int foreign_births );

    // Get, Set # of kills
    int
    getKills() const;

    void
    setKills( int kills );

    // Get walking speed
    virtual double
    getSpeed() const = 0;

    // Animate this creature
    virtual void
    animate(
        MTRand& random_generator,
        const TileCache& tile_cache,
        CreatureList& pending_creation,
        CreatureList& pending_deletion ) = 0;

    // Determine how the given action (MATING or FIGHTING) will affect
the
    // corresponding drive (increase population or destroy)
    // The value returned will be the positive difference in satiation
(delta)
    virtual double
    projectedSatiationEffect(
        action_type action,
        int number_of_ants,
        int number_of_spiders ) = 0;

    // This creature just mated - update drive satiation levels, etc.
of both parents
    // and others that surround it.
    virtual void
    updatesAfterMate(
        Creature& other_creature,
        const TileList& surrounding_tiles,
        int number_of_ants,
        int number_of_spiders ) = 0;
```

```
    // This creature just killed another creature - update drive
satiation levels, etc. of
    // this creature and others that surround it.
    virtual void
    updatesAfterKill(
        const TileList& surrounding_tiles,
        int number_of_ants,
        int number_of_spiders ) = 0;

    // Adjust priority parameters
    virtual void
    learn() = 0;

    // Make other creatures team up against this one
    virtual void
    teamUpAgainst( const TileList& surrounding_tiles ) = 0;

    // Action Selection
    void
    actionSelection(
        int number_of_ants,
        int number_of_spiders,
        MTRand& random_generator );

    // Get a random location on the board
    static Point3D
    getRandomLocation( MTRand& random_generator );

    // Calculate the total number of ants and spiders given the
surrounding tiles
    static void
    calculateObservedPopulations(
        int& number_of_ants,
        int& number_of_spiders,
        const TileList& surrounding_tiles );

public:

    // Constant (input) parameters for the experiment

    static action_selection_type m_action_selection_type;     // Action
selection type

    static float m_personal_mating_satiation_constant;        //
Creature just mated
    static float m_personal_fighting_satiation_constant;      //
Creature just made a kill
    static float m_observed_mating_satiation_constant;        //
Creature observed the other species mate
    static float m_observed_fighting_satiation_constant;      //
Creature observed a death of its own species

    static learning_type m_learning_type;                     //
Learning Type
    static int m_learning_time;                               // Time
for a learning adjustment
```

```
    static float m_ip_priority_adjustment_constant;           //
Priority adjustment constant for increase population
    static float m_d_priority_adjustment_constant;            //
Priority adjustment constant for destroy

    static bool m_teaming_up;                                 // Do
creatures team up for fighting?

protected:

    Point3D m_location;                 // Location (x, y)
    action_type m_action;               // Current action

    Point3D m_target;                   // Target (x,y) point that it
is currently walking to
    bool m_target_set;                  // Has a target been set?

    int m_timer;                        // Timer for fighting/mating
pause
    double m_health;                    // Health
    bool m_secondary_creature;          // Is this the secondary
creature of a fight/mate?
    Creature* m_other_creature;         // Creature we are
fighting/mating with, NULL otherwise

    int m_just_mated_timer;             // Timer to let everyone walk
away from a mate

    Drive m_increase_population_drive;  // Increase Population Drive
(motivation)
    Drive m_destroy_drive;              // Destroy Drive (motivation)

    int m_births;                       // Learning: # of births during
learning time frame
    int m_deaths;                       // Learning: # of deaths during
learning time frame
    int m_foreign_births;               // Learning: # of foreign
births during learning time frame
    int m_kills;                        // Learning: # of kills during
learning time frame

public:

    int m_learning_timer;               // Learning: Timer for learning
intervals
    double m_age;                       // Lamarckian: Age for oldest
parent

};

#endif
```

CREATURE.CPP

```
#include "Creature.h"
#include "MersenneTwister.h"
#include "Tile.h"
```

```
float Creature::m_personal_mating_satiation_constant = 3.0;
float Creature::m_personal_fighting_satiation_constant = 3.0;
float Creature::m_observed_mating_satiation_constant = 0.2;
float Creature::m_observed_fighting_satiation_constant = 0.2;
action_selection_type Creature::m_action_selection_type = REWARD;
learning_type Creature::m_learning_type = LAMARCKIAN;
int Creature::m_learning_time = 250;
float Creature::m_ip_priority_adjustment_constant = 0.0005;
float Creature::m_d_priority_adjustment_constant = 0.0005;
bool Creature::m_teaming_up = false;

//---------------------------------------------------------------
Creature::Creature()
:   m_action (LOOKING_FOR_MATE),
    m_target_set (false),
    m_timer (0),
    m_health (1.0),
    m_secondary_creature (false),
    m_other_creature (NULL),
    m_just_mated_timer (0),
    m_births (0),
    m_deaths (0),
    m_foreign_births (0),
    m_kills (0),
    m_learning_timer (0),
    m_age (0)
{
    // Constructor
}

//---------------------------------------------------------------
Creature::~Creature()
{
    // Destructor
}

//---------------------------------------------------------------
const Point3D&
Creature::getLocation() const
{
    return m_location;
}

//---------------------------------------------------------------
void
Creature::setLocation( const Point3D& location )
{
    m_location = location;
}

//---------------------------------------------------------------
action_type
Creature::getAction() const
{
    return m_action;
}
```

```
//-------------------------------------------------------------
void
Creature::setAction( action_type action )
{
    m_action = action;
}

//-------------------------------------------------------------
const Point3D&
Creature::getTarget() const
{
    return m_target;
}

//-------------------------------------------------------------
void
Creature::setTarget( const Point3D& target )
{
    m_target = target;
}

//-------------------------------------------------------------
bool
Creature::getTargetSet() const
{
    return m_target_set;
}

//-------------------------------------------------------------
void
Creature::setTargetSet( bool target_set )
{
    m_target_set = target_set;
}

//-------------------------------------------------------------
int
Creature::getTimer() const
{
    return m_timer;
}

//-------------------------------------------------------------
void
Creature::setTimer( int timer )
{
    m_timer = timer;
}

//-------------------------------------------------------------
double
Creature::getHealth() const
{
    return m_health;
}
```

```
//---------------------------------------------------------
void
Creature::setHealth( double health )
{
    m_health = health;
}

//---------------------------------------------------------
bool
Creature::getSecondaryCreature() const
{
    return m_secondary_creature;
}

//---------------------------------------------------------
void
Creature::setSecondaryCreature( bool secondary_creature )
{
    m_secondary_creature = secondary_creature;
}

//---------------------------------------------------------
Creature*
Creature::getOtherCreature() const
{
    return m_other_creature;
}

//---------------------------------------------------------
void
Creature::setOtherCreature( Creature* other_creature )
{
    m_other_creature = other_creature;
}

//---------------------------------------------------------
int
Creature::getJustMatedTimer() const
{
    return m_just_mated_timer;
}

//---------------------------------------------------------
void
Creature::setJustMatedTimer( int just_mated_timer )
{
    m_just_mated_timer = just_mated_timer;
}

//---------------------------------------------------------
const Drive&
Creature::getIncreasePopulationDrive() const
{
    return m_increase_population_drive;
}

//---------------------------------------------------------
```

```
void
Creature::setIncreasePopulationDrive( const Drive& drive )
{
    m_increase_population_drive = drive;
}

//---------------------------------------------------------------
const Drive&
Creature::getDestroyDrive() const
{
    return m_destroy_drive;
}

//---------------------------------------------------------------
void
Creature::setDestroyDrive( const Drive& drive )
{
    m_destroy_drive = drive;
}

//---------------------------------------------------------------
int
Creature::getBirths() const
{
    return m_births;
}

//---------------------------------------------------------------
void
Creature::setBirths( int births )
{
    m_births = births;
}

//---------------------------------------------------------------
int
Creature::getDeaths() const
{
    return m_deaths;
}

//---------------------------------------------------------------
void
Creature::setDeaths( int deaths )
{
    m_deaths = deaths;
}

//---------------------------------------------------------------
int
Creature::getForeignBirths() const
{
    return m_foreign_births;
}

//---------------------------------------------------------------
void
```

```
Creature::setForeignBirths( int foreign_births )
{
    m_foreign_births = foreign_births;
}

//---------------------------------------------------------------
int
Creature::getKills() const
{
    return m_kills;
}

//---------------------------------------------------------------
void
Creature::setKills( int kills )
{
    m_kills = kills;
}

//---------------------------------------------------------------
void
Creature::actionSelection(
    int number_of_ants,
    int number_of_spiders,
    MTRand& random_generator )
{
    // Select the next action

    if ( Creature::m_action_selection_type == FIFTY_FIFTY )
    {
        if ( random_generator.randInt() % 2 == 0 ) {
            setAction( LOOKING_FOR_FIGHT );
        }
        else {
            setAction( LOOKING_FOR_MATE );
        }
    }
    else
    {
        // Generate action based on projected reward

        // Determine the satiation effect on the drives if we mate or
fight
        double mating_satiation_effect = projectedSatiationEffect(
MATING, number_of_ants, number_of_spiders );
        double fighting_satiation_effect = projectedSatiationEffect(
FIGHTING, number_of_ants, number_of_spiders );

        // Get the current satiation levels of the drives
        double mating_satiation_level =
m_increase_population_drive.getSatiation();
        double fighting_satiation_level =
m_destroy_drive.getSatiation();

        // Determine the new levels with the effects applied
        double new_mating_satiation_level = mating_satiation_level +
mating_satiation_effect;
```

```
        double new_fighting_satiation_level = fighting_satiation_level
+ fighting_satiation_effect;

        // Calculate the rewards of a mate or fight action
        double mating_reward =
m_increase_population_drive.calculateReward( new_mating_satiation_level
);
        double fighting_reward = m_destroy_drive.calculateReward(
new_fighting_satiation_level );

        // Do the action for whichever reward is higher
        if ( mating_reward > fighting_reward ) {
            setAction( LOOKING_FOR_MATE );
        }
        else {
            setAction( LOOKING_FOR_FIGHT );
        }
    }
}

//---------------------------------------------------------------
Point3D
Creature::getRandomLocation( MTRand& random_generator )
{
    // Get a random location on the board

    // Get grid size
    int grid_size = AXES_SIZE * 2;

    // Get two random numbers between 0 and grid_size - 1.
    MTRand::uint32 random_num1 = random_generator.randInt();
    random_num1 %= grid_size;
    MTRand::uint32 random_num2 = random_generator.randInt();
    random_num2 %= grid_size;

    // Set new target in range of [-AXES_SIZE,+AXES_SIZE]
    return Point3D( random_num1 - AXES_SIZE, random_num2 - AXES_SIZE,
0.0 );
}

//---------------------------------------------------------------
void
Creature::calculateObservedPopulations(
    int& number_of_ants,
    int& number_of_spiders,
    const TileList& surrounding_tiles )
{
    // Calculate the total number of ants and spiders given the
surrounding tiles

    // Determine visible population size of both types of creatures
    // (tiles will be from a 3x3 tile square surrounding the creature)
    number_of_ants = 0;
    number_of_spiders = 0;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
```

```
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
            continue;
        }

        number_of_ants += tile->m_number_of_ants;
        number_of_spiders += tile->m_number_of_spiders;
    }
    if ( number_of_ants <= 0 ) {
        number_of_ants = 1;
    }
    if ( number_of_spiders <= 0 ) {
        number_of_spiders = 1;
    }
}
```

ANT.H

```
#ifndef ANT_H
#define ANT_H

#include "Creature.h"

/**
 ** Ant class - represents one ant creature
 **/

class Ant : public Creature
{
public:

    // Constructor, destructor
    Ant();
    virtual ~Ant();

    // Get the type of creature
    virtual creature_type
    getType() const;

    // Get walking speed
    virtual double
    getSpeed() const;

    // Get damage this creature inflicts on spiders
    virtual double
    getDamage() const;

    // Animate this creature
    virtual void
    animate(
        MTRand& random_generator,
        const TileCache& tile_cache,
        CreatureList& pending_creation,
        CreatureList& pending_deletion );
```

```
    // Determine how the given action (MATING or FIGHTING) will affect
the
    // corresponding drive (increase population or destroy)
    // The value returned will be the positive difference in satiation
(delta)
    virtual double
    projectedSatiationEffect(
        action_type action,
        int number_of_ants,
        int number_of_spiders );

    // This creature just mated - update drive satiation levels, etc.
of both parents
    // and others that surround it.
    virtual void
    updatesAfterMate(
        Creature& other_creature,
        const TileList& surrounding_tiles,
        int number_of_ants,
        int number_of_spiders );

    // This creature just killed another creature - update drive
satiation levels, etc. of
    // this creature and others that surround it.
    virtual void
    updatesAfterKill(
        const TileList& surrounding_tiles,
        int number_of_ants,
        int number_of_spiders );

    // Adjust priority parameters
    virtual void
    learn();

    // Make other creatures team up against this one
    virtual void
    teamUpAgainst( const TileList& surrounding_tiles );

};

#endif
```

ANT.CPP

```
#include "Ant.h"
#include "TileCache.h"
#include "Tile.h"
#include "MersenneTwister.h"

//--------------------------------------------------------------
Ant::Ant()
:   Creature()
{
    // Constructor
}

//--------------------------------------------------------------
```

```
Ant::~Ant()
{
    // Destructor
}

//---------------------------------------------------------------
creature_type
Ant::getType() const
{
    // Get the type of creature

    return ANT_TYPE;
}

//---------------------------------------------------------------
double
Ant::getSpeed() const
{
    // Get walking speed

    return 0.1;
}

//---------------------------------------------------------------
double
Ant::getDamage() const
{
    // Get damage this creature inflicts on others

    return 0.4;
}

//---------------------------------------------------------------
void
Ant::animate(
    MTRand& random_generator,
    const TileCache& tile_cache,
    CreatureList& pending_creation,
    CreatureList& pending_deletion )
{
    // Animate this creature

    // If this has been labeled as a secondary creature in an
interaction, do not process
    // this creature
    if ( m_secondary_creature )
    {
        return;
    }

    // Decrement the "just mated timer" if applicable
    if ( m_just_mated_timer > 0 ) {
        m_just_mated_timer--;
    }

    // Periodically learn
    if ( Creature::m_learning_type != NONE )
```

```
    {
        // Update creature age (Lamarckian)
        m_age += 0.000001;

        // If we have reached the end of the learning interval, make
the creature learn
        if ( ++m_learning_timer % Creature::m_learning_time == 0 )
        {
            learn();

            m_learning_timer = 0;
        }
    }

    // Switch based on action
    switch ( m_action )
    {
        case LOOKING_FOR_FIGHT:
        case LOOKING_FOR_MATE:
        {
            // Walk around looking for opponent or mate

            // Only ants look for a fight (arbitrarily one direction):
            // If we are looking to fight and we found an opponent,
start fighting
            if ( m_action == LOOKING_FOR_FIGHT )
            {
                // Loop over surronding tiles around this creature
                Tile* tile = NULL;
                Creature* creature = NULL;
                TileList tiles = tile_cache.getTileList( m_location );
                TileListIterator tile_iter = tiles.begin();
                for ( ; tile_iter != tiles.end(); ++tile_iter )
                {
                    tile = *tile_iter;

                    // Loop over all creatures in the tile
                    CreatureListIterator creature_iter = tile-
>m_creatures.begin();
                    for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
                    {
                        creature = *creature_iter;

                        // If this is a spider looking to fight and is
within the distance threshold,
                        // then start fighting
                        if ( creature->getType() == SPIDER_TYPE &&
                             creature->getAction() == LOOKING_FOR_FIGHT
&&
                             m_location.isEqualTo( creature-
>getLocation(), Tolerance( HIT_TOLERANCE, HIT_TOLERANCE )) )
                        {
                            setAction( FIGHTING );
                            creature->setAction( FIGHTING );
```

```
                            // Keep track of other creature, and mark
the other creature as secondary
                            // in this interaction
                            setOtherCreature( creature );
                            creature->setSecondaryCreature( true );

                            // Set a timer for a pause
                            m_timer = PAUSE_TIME;

                            return;
                        }
                    }
                }
            }
            else if ( m_action == LOOKING_FOR_MATE &&
m_just_mated_timer <= 0 )
            {
                // If we are looking to mate and we found a mate, start
mating

                // Loop over surronding tiles around this creature
                Tile* tile = NULL;
                Creature* creature = NULL;
                TileList tiles = tile_cache.getTileList( m_location );
                TileListIterator tile_iter = tiles.begin();
                for ( ; tile_iter != tiles.end(); ++tile_iter )
                {
                    tile = *tile_iter;

                    // Loop over all creatures in the tile
                    CreatureListIterator creature_iter = tile-
>m_creatures.begin();
                    for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
                    {
                        creature = *creature_iter;

                        // Skip if this creature is the same creature!
                        if ( this == creature ) {
                            continue;
                        }

                        // If this is the same type, is within the
distance threshold, and isn't waiting
                        // because it just mated, then start mating
                        if ( creature->getType() == ANT_TYPE &&
                             creature->getAction() == LOOKING_FOR_MATE
&&
                             creature->getJustMatedTimer() <= 0 &&
                             m_location.isEqualTo( creature-
>getLocation(), Tolerance( HIT_TOLERANCE, HIT_TOLERANCE )) )
                        {
                            setAction( MATING );
                            creature->setAction( MATING );

                            // Keep track of other creature, and mark
the other creature as secondary
```

```
                            // in this interaction
                            setOtherCreature( creature );
                            creature->setSecondaryCreature( true );

                            // Set a timer for a pause
                            m_timer = PAUSE_TIME;

                            return;
                        }
                    }
                }
            }

            // If we have not set a target yet, or we have reached the
target, choose a new
            // target at random
            if ( m_target_set == false ||
                 m_location.isEqualTo( m_target,
Tolerance(TARGET_THRESHOLD, TARGET_THRESHOLD) ) )
            {
                // Set the new target as a random location on the board
                m_target = getRandomLocation( random_generator );

                m_target_set = true;
            }

            // Take a step towards target
            Point3D target_vector = m_target - m_location;
            target_vector.unitize();
            m_location += target_vector * getSpeed();

            break;
        }
        case FIGHTING:
        {
            m_target_set = false;

            // Make sure the timer has run out (for the pause)
            if ( --m_timer <= 0 )
            {
                // The ant will do damage first
                double spider_health = m_other_creature->getHealth();
                spider_health -= getDamage();
                m_other_creature->setHealth( spider_health );

                // If the spider died
                if ( spider_health <= 0.0 )
                {
                    // Add the spider to the pending destruction list
                    pending_deletion.push_back( m_other_creature );

                    // Get the surrounding tiles (3x3 grid) around the
ant
                    TileList tile_list = tile_cache.getTileList(
m_location );

                    // Determine number of ants and spiders
```

```
                    int number_of_ants = 0;
                    int number_of_spiders = 0;
                    calculateObservedPopulations( number_of_ants,
number_of_spiders, tile_list );

                    // Select a new action for this ant
                    actionSelection(
                        number_of_ants,
                        number_of_spiders,
                        random_generator );

                    // Update drive satiation level, etc. for the ant
and others spiders that surround it
                    updatesAfterKill( tile_list, number_of_ants,
number_of_spiders );

                    m_other_creature = NULL;

                    // Set a new target for this ant
                    m_target = getRandomLocation( random_generator );
                    m_target_set = true;

                    // Make other creatures team up against this one
                    if ( Creature::m_teaming_up ) {
                        teamUpAgainst( tile_list );
                    }
                }
                else
                {
                    // Otherwise, the ant dies
                    pending_deletion.push_back( this );

                    // Get the surrounding tiles (3x3 grid) around the
spider
                    TileList tile_list = tile_cache.getTileList(
m_other_creature->getLocation() );

                    // Determine number of ants and spiders
                    int number_of_ants = 0;
                    int number_of_spiders = 0;
                    calculateObservedPopulations( number_of_ants,
number_of_spiders, tile_list );

                    // Select a new action for the spider
                    m_other_creature->actionSelection(
                        number_of_ants,
                        number_of_spiders,
                        random_generator );

                    // Update drive satiation level, etc. for the
spider and others ants that surround it
                    m_other_creature->updatesAfterKill( tile_list,
number_of_ants, number_of_spiders );

                    m_other_creature->setSecondaryCreature( false );

                    // Set a new target for the spider
```

```
                    m_other_creature->setTarget( getRandomLocation(
random_generator ) );
                    m_other_creature->setTargetSet( true );

                    // Make other creatures team up against this one
                    if ( Creature::m_teaming_up ) {
                        m_other_creature->teamUpAgainst( tile_list );
                    }
                }
            }

            break;
        }
        case MATING:
        {
            m_target_set = false;

            // Make sure the timer has run out (for the pause)
            if ( --m_timer <= 0 )
            {
                // Create a new ant
                Creature* new_creature = new Ant();

                // Place the new creature in the middle of the parents
                new_creature->setLocation(
                    Point3D(
                        (m_location.m_x + m_other_creature-
>getLocation().m_x) / 2,
                        (m_location.m_y + m_other_creature-
>getLocation().m_y) / 2, 0.0 ) );

                // Get the surrounding tiles (3x3 grid) around the ant
                TileList tile_list = tile_cache.getTileList( m_location
);

                // Determine number of ants and spiders
                int number_of_ants = 0;
                int number_of_spiders = 0;
                calculateObservedPopulations( number_of_ants,
number_of_spiders, tile_list );

                // If Lamarckian, new creature inherits the priority
parameter from the
                // oldest parent
                if ( Creature::m_learning_type == LAMARCKIAN )
                {
                    Drive ip_drive = new_creature-
>getIncreasePopulationDrive();
                    Drive d_drive = new_creature->getDestroyDrive();

                    if ( m_age > m_other_creature->m_age ) {
                        ip_drive.setPriorityParameter(

getIncreasePopulationDrive().getPriorityParameter() );
                        d_drive.setPriorityParameter(
                            getDestroyDrive().getPriorityParameter() );
                    }
```

```
                    else {
                        ip_drive.setPriorityParameter(
                            m_other_creature-
>getIncreasePopulationDrive().getPriorityParameter() );
                        d_drive.setPriorityParameter(
                            m_other_creature-
>getDestroyDrive().getPriorityParameter() );
                    }

                    new_creature->setIncreasePopulationDrive( ip_drive
);
                    new_creature->setDestroyDrive( d_drive );
                }

                // Select a new action for the child and parents
                new_creature->actionSelection(
                    number_of_ants,
                    number_of_spiders,
                    random_generator );
                actionSelection(
                    number_of_ants,
                    number_of_spiders,
                    random_generator );
                m_other_creature->actionSelection(
                    number_of_ants,
                    number_of_spiders,
                    random_generator );

                // Set timers on both parents and child to let them
walk away from the mate
                // if their new action is to mate again
                if ( new_creature->getAction() == LOOKING_FOR_MATE ) {
                    new_creature->setJustMatedTimer( JUST_MATED_TIME );
                }
                if ( getAction() == LOOKING_FOR_MATE ) {
                    setJustMatedTimer( JUST_MATED_TIME );
                }
                if ( m_other_creature->getAction() == LOOKING_FOR_MATE
) {
                    m_other_creature->setJustMatedTimer(
JUST_MATED_TIME );
                }

                // Set a new target for these creatures
                new_creature->setTarget( getRandomLocation(
random_generator ) );
                new_creature->setTargetSet( true );
                m_target = getRandomLocation( random_generator );
                m_target_set = true;
                m_other_creature->setTarget( getRandomLocation(
random_generator ) );
                m_other_creature->setTargetSet( true );

                // Update drive satiation levels, etc. of both parents,
and other surrounding creatures
                updatesAfterMate( *m_other_creature, tile_list,
number_of_ants, number_of_spiders );
```

```
                // Break link between the parents
                m_other_creature->setSecondaryCreature( false );
                m_other_creature = NULL;

                // Add new creature to the pending creation list
                pending_creation.push_back( new_creature );
            }

            break;
        }
        default:
        {
            // Unknown action
            break;
        }
    };
}

//---------------------------------------------------------
double
Ant::projectedSatiationEffect(
    action_type action,
    int number_of_ants,
    int number_of_spiders )
{
    // Determine how the given action (MATING or FIGHTING) will affect
the
    // corresponding drive (increase population or destroy)
    // The value returned will be the positive difference in satiation
(delta)

    if ( action == MATING )
    {
        // The effect on the increase population drive is proportional
to the population
        // of ants
        return Creature::m_personal_mating_satiation_constant * (1.0 /
number_of_ants);
    }
    else if ( action == FIGHTING )
    {
        // The effect on the destroy drive is proportional to the
population of spiders.
        return Creature::m_personal_fighting_satiation_constant * (1.0
/ number_of_spiders);
    }
    else
    {
        // Invalid action
        return 0;
    }
}

//---------------------------------------------------------
void
Ant::updatesAfterMate(
```

```
    Creature& other_creature,
    const TileList& surrounding_tiles,
    int number_of_ants,
    int number_of_spiders )
{
    // This creature just mated - update drive satiation levels, etc.
of this creature
    // and others that surround it.

    // Return immediately if action selection type is fifty-fifty
    if ( Creature::m_action_selection_type == FIFTY_FIFTY ) {
        return;
    }

    // Determine the effect on the increase population drive for this
creature
    double satiation_effect = projectedSatiationEffect( MATING,
number_of_ants, number_of_spiders );

    // Update the increase population drive for this creature
    double satiation_level =
m_increase_population_drive.getSatiation();
    m_increase_population_drive.setSatiation( satiation_level +
satiation_effect );

    // Determine the effect on the increase population drive for the
other parent
    satiation_effect = other_creature.projectedSatiationEffect( MATING,
number_of_ants, number_of_spiders );

    // Update the increase population drive for the other parent
    Drive other_creature_increase_population_drive =
other_creature.getIncreasePopulationDrive();
    satiation_level =
other_creature_increase_population_drive.getSatiation();
    other_creature_increase_population_drive.setSatiation(
satiation_level + satiation_effect );
    other_creature.setIncreasePopulationDrive(
other_creature_increase_population_drive );

    // Loop over all surrounding tiles
    Creature* creature = NULL;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
            continue;
        }

        // Loop over creatures in this tile
        CreatureListIterator creature_iter = tile->m_creatures.begin();
        for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
        {
            creature = *creature_iter;
```

```
            // If this is a spider
            if ( creature->getType() == SPIDER_TYPE )
            {
                // Update its destroy satiation level
                Drive drive = creature->getDestroyDrive();
                drive.setSatiation(
                    drive.getSatiation() -
(Creature::m_observed_mating_satiation_constant * (1.0 /
number_of_ants)) );
                creature->setDestroyDrive( drive );

                // Update its foreign birth count
                creature->setForeignBirths( creature-
>getForeignBirths() + 1 );
            }
            else
            {
                // Ant - Update its birth count
                creature->setBirths( creature->getBirths() + 1 );
            }
        }
    }
}

//---------------------------------------------------------------
void
Ant::updatesAfterKill(
    const TileList& surrounding_tiles,
    int number_of_ants,
    int number_of_spiders )
{
    // This creature just killed another creature - update drive
satiation levels, etc. of
    // this creature and others that surround it.

    // Return immediately if action selection type is fifty-fifty
    if ( Creature::m_action_selection_type == FIFTY_FIFTY ) {
        return;
    }

    // Determine the effect on the destroy drive for this creature
    double satiation_effect = projectedSatiationEffect( FIGHTING,
number_of_ants, number_of_spiders );

    // Update the destroy drive for this creature
    double satiation_level = m_destroy_drive.getSatiation();
    m_destroy_drive.setSatiation( satiation_level + satiation_effect );

    // Loop over all surrounding tiles
    Creature* creature = NULL;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
```

```
            continue;
        }

        // Loop over creatures in this tile
        CreatureListIterator creature_iter = tile->m_creatures.begin();
        for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
        {
            creature = *creature_iter;

            // If this is a spider
            if ( creature->getType() == SPIDER_TYPE )
            {
                // Update its increase population satiation level
                Drive drive = creature->getIncreasePopulationDrive();
                drive.setSatiation(
                    drive.getSatiation() -
(Creature::m_observed_fighting_satiation_constant * (1.0 /
number_of_spiders)) );
                creature->setIncreasePopulationDrive( drive );

                // Update its death count
                creature->setDeaths( creature->getDeaths() + 1 );
            }
            else
            {
                // Ant - update its kill count
                creature->setKills( creature->getKills() + 1 );
            }
        }
    }
}

//---------------------------------------------------------------
void
Ant::learn()
{
    // Adjust priority parameters

    // Adjust priority parameter of increase population drive
    double ip_priority =
m_increase_population_drive.getPriorityParameter();
    ip_priority += Creature::m_ip_priority_adjustment_constant *
        (m_deaths - m_births) * (m_deaths + m_births);
    m_increase_population_drive.setPriorityParameter( ip_priority );

    // Adjust priority parameter of destroy drive
    double d_priority = m_destroy_drive.getPriorityParameter();
    d_priority += Creature::m_d_priority_adjustment_constant *
        (m_foreign_births - m_kills) * (m_foreign_births + m_kills);
    m_destroy_drive.setPriorityParameter( d_priority );

    // Clear the counts
    m_births = 0;
    m_deaths = 0;
    m_foreign_births = 0;
    m_kills = 0;
```

```
}

//-------------------------------------------------------------
void
Ant::teamUpAgainst( const TileList& surrounding_tiles )
{
    // Make other creatures team up against this one

    // Loop over all surrounding tiles
    Creature* creature = NULL;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
            continue;
        }

        // Loop over creatures in this tile
        CreatureListIterator creature_iter = tile->m_creatures.begin();
        for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
        {
            creature = *creature_iter;

            // If this is a spider looking to fight and this creature
wants to fight
            if ( creature->getType() == SPIDER_TYPE &&
                 creature->getAction() == LOOKING_FOR_FIGHT &&
                 getAction() == LOOKING_FOR_FIGHT )
            {
                // Set its target to this one
                creature->setTarget( getTarget() );
                creature->setTargetSet( true );
            }
        }
    }
}
```

SPIDER.H

```
#ifndef SPIDER_H
#define SPIDER_H

#include "Creature.h"

/**
 ** Spider class - represents one spider creature
 **/

class Spider : public Creature
{
public:

    // Constructor, destructor
```

```
    Spider();
    virtual ~Spider();

    // Get the type of creature
    virtual creature_type
    getType() const;

    // Get walking speed
    virtual double
    getSpeed() const;

    // Animate this creature
    virtual void
    animate(
        MTRand& random_generator,
        const TileCache& tile_cache,
        CreatureList& pending_creation,
        CreatureList& pending_deletion );

    // Determine how the given action (MATING or FIGHTING) will affect
the
    // corresponding drive (increase population or destroy)
    // The value returned will be the positive difference in satiation
(delta)
    virtual double
    projectedSatiationEffect(
        action_type action,
        int number_of_ants,
        int number_of_spiders );

    // This creature just mated - update drive satiation levels, etc.
of both parents
    // and others that surround it.
    virtual void
    updatesAfterMate(
        Creature& other_creature,
        const TileList& surrounding_tiles,
        int number_of_ants,
        int number_of_spiders );

    // This creature just killed another creature - update drive
satiation levels, etc. of
    // this creature and others that surround it.
    virtual void
    updatesAfterKill(
        const TileList& surrounding_tiles,
        int number_of_ants,
        int number_of_spiders );

    // Adjust priority parameters
    virtual void
    learn();

    // Make other creatures team up against this one
    virtual void
    teamUpAgainst( const TileList& surrounding_tiles );
```

```
};

#endif
```

SPIDER.CPP

```
#include "Spider.h"
#include "TileCache.h"
#include "Tile.h"
#include "MersenneTwister.h"

//---------------------------------------------------------------
Spider::Spider()
:   Creature()
{
    // Constructor
}

//---------------------------------------------------------------
Spider::~Spider()
{
    // Destructor
}

//---------------------------------------------------------------
creature_type
Spider::getType() const
{
    // Get the type of creature

    return SPIDER_TYPE;
}

//---------------------------------------------------------------
double
Spider::getSpeed() const
{
    // Get walking speed

    return 0.05;
}

//---------------------------------------------------------------
void
Spider::animate(
    MTRand& random_generator,
    const TileCache& tile_cache,
    CreatureList& pending_creation,
    CreatureList& pending_deletion )
{
    // Animate this creature

    // If this has been labeled as a secondary creature in an
interaction, do not process
    // this creature
    if ( m_secondary_creature )
    {
```

```
        return;
    }

    // Decrement the "just mated timer" if applicable
    if ( m_just_mated_timer > 0 ) {
        m_just_mated_timer--;
    }

    // Periodically learn
    if ( Creature::m_learning_type != NONE )
    {
        // Update creature age (Lamarckian)
        m_age += 0.000001;

        // If we have reached the end of the learning interval, make
the creature learn
        if ( ++m_learning_timer % Creature::m_learning_time == 0 )
        {
            learn();

            m_learning_timer = 0;
        }
    }

    // Switch based on action
    switch ( m_action )
    {
        case LOOKING_FOR_FIGHT:
        case LOOKING_FOR_MATE:
        {
            // Walk around looking for an opponent or mate
            // (Note: the actual link between opponents is established
by the ants)

            // If we are looking to mate and we found a mate, start
mating
            if ( m_action == LOOKING_FOR_MATE && m_just_mated_timer <=
0 )
            {
                // Loop over surronding tiles around this creature
                Tile* tile = NULL;
                Creature* creature = NULL;
                TileList tiles = tile_cache.getTileList( m_location );
                TileListIterator tile_iter = tiles.begin();
                for ( ; tile_iter != tiles.end(); ++tile_iter )
                {
                    tile = *tile_iter;

                    // Loop over all creatures in the tile
                    CreatureListIterator creature_iter = tile-
>m_creatures.begin();
                    for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
                    {
                        creature = *creature_iter;

                        // Skip if this creature is the same creature!
```

```
                        if ( this == creature ) {
                            continue;
                        }

                        // If this is the same type, is within the
distance threshold, and isn't waiting
                        // because it just mated, then start mating
                        if ( creature->getType() == SPIDER_TYPE &&
                             creature->getAction() == LOOKING_FOR_MATE
&&
                             creature->getJustMatedTimer() <= 0 &&
                             m_location.isEqualTo( creature-
>getLocation(), Tolerance( HIT_TOLERANCE, HIT_TOLERANCE )) )
                        {
                            setAction( MATING );
                            creature->setAction( MATING );

                            // Keep track of other creature, and mark
the other creature as secondary
                            // in this interaction
                            setOtherCreature( creature );
                            creature->setSecondaryCreature( true );

                            // Set a timer for a pause
                            m_timer = PAUSE_TIME;

                            return;
                        }
                    }
                }
            }

            // If we have not set a target yet, or we have reached the
target, choose a new
            // target at random
            if ( m_target_set == false ||
                 m_location.isEqualTo( m_target,
Tolerance(TARGET_THRESHOLD, TARGET_THRESHOLD) ) )
            {
                // Set the new target as a random location on the board
                m_target = getRandomLocation( random_generator );

                m_target_set = true;
            }

            // Take a step towards target
            Point3D target_vector = m_target - m_location;
            target_vector.unitize();
            m_location += target_vector * getSpeed();

            break;
        }
        case FIGHTING:
        {
            // This will never be hit - the fighting processing is done
through the ants.
```

```
            break;
        }
        case MATING:
        {
            m_target_set = false;

            // Make sure the timer has run out (for the pause)
            if ( --m_timer <= 0 )
            {
                // Create a new creature
                Creature* new_creature = new Spider();

                // Place the new creature in the middle of the parents
                new_creature->setLocation(
                    Point3D(
                        (m_location.m_x + m_other_creature-
>getLocation().m_x) / 2,
                        (m_location.m_y + m_other_creature-
>getLocation().m_y) / 2, 0.0 ) );

                // Get the surrounding tiles (3x3 grid) around the
spider
                TileList tile_list = tile_cache.getTileList( m_location
);

                // Determine number of ants and spiders
                int number_of_ants = 0;
                int number_of_spiders = 0;
                calculateObservedPopulations( number_of_ants,
number_of_spiders, tile_list );

                // If Lamarckian, new creature inherits the priority
parameter from the
                // oldest parent
                if ( Creature::m_learning_type == LAMARCKIAN )
                {
                    Drive ip_drive = new_creature-
>getIncreasePopulationDrive();
                    Drive d_drive = new_creature->getDestroyDrive();

                    if ( m_age > m_other_creature->m_age ) {
                        ip_drive.setPriorityParameter(

getIncreasePopulationDrive().getPriorityParameter() );
                        d_drive.setPriorityParameter(
                            getDestroyDrive().getPriorityParameter() );
                    }
                    else {
                        ip_drive.setPriorityParameter(
                            m_other_creature-
>getIncreasePopulationDrive().getPriorityParameter() );
                        d_drive.setPriorityParameter(
                            m_other_creature-
>getDestroyDrive().getPriorityParameter() );
                    }
```

```
                    new_creature->setIncreasePopulationDrive( ip_drive
);
                    new_creature->setDestroyDrive( d_drive );
                }

                // Select a new action for the child and parents
                new_creature->actionSelection(
                    number_of_ants,
                    number_of_spiders,
                    random_generator );
                actionSelection(
                    number_of_ants,
                    number_of_spiders,
                    random_generator );
                m_other_creature->actionSelection(
                    number_of_ants,
                    number_of_spiders,
                    random_generator );

                // Set timers on both parents and child to let them
walk away from the mate
                // if their new action is to mate again
                if ( new_creature->getAction() == LOOKING_FOR_MATE ) {
                    new_creature->setJustMatedTimer( JUST_MATED_TIME );
                }
                if ( getAction() == LOOKING_FOR_MATE ) {
                    setJustMatedTimer( JUST_MATED_TIME );
                }
                if ( m_other_creature->getAction() == LOOKING_FOR_MATE
) {
                    m_other_creature->setJustMatedTimer(
JUST_MATED_TIME );
                }

                // Set a new target for these creatures
                new_creature->setTarget( getRandomLocation(
random_generator ) );
                new_creature->setTargetSet( true );
                m_target = getRandomLocation( random_generator );
                m_target_set = true;
                m_other_creature->setTarget( getRandomLocation(
random_generator ) );
                m_other_creature->setTargetSet( true );

                // Update drive satiation levels, etc. of both parents,
and other surrounding creatures
                updatesAfterMate( *m_other_creature, tile_list,
number_of_ants, number_of_spiders );

                // Break link between the parents
                m_other_creature->setSecondaryCreature( false );
                m_other_creature = NULL;

                // Add new creature to the pending creation list
                pending_creation.push_back( new_creature );
            }
```

```
            break;
        }
        default:
        {
            // Unknown action
            break;
        }
    };
}

//---------------------------------------------------------------
double
Spider::projectedSatiationEffect(
    action_type action,
    int number_of_ants,
    int number_of_spiders )
{
    // Determine how the given action (MATING or FIGHTING) will affect
the
    // corresponding drive (increase population or destroy)
    // The value returned will be the positive difference in satiation
(delta)

    if ( action == MATING )
    {
        // The effect on the increase population drive is proportional
to the population
        // of spiders
        return Creature::m_personal_mating_satiation_constant * (1.0 /
number_of_spiders);
    }
    else if ( action == FIGHTING )
    {
        // The effect on the destroy drive is proportional to the
population of ants.
        return Creature::m_personal_fighting_satiation_constant * (1.0
/ number_of_ants);
    }
    else
    {
        // Invalid action
        return 0;
    }
}

//---------------------------------------------------------------
void
Spider::updatesAfterMate(
    Creature& other_creature,
    const TileList& surrounding_tiles,
    int number_of_ants,
    int number_of_spiders )
{
    // This creature just mated - update drive satiation levels, etc.
of this creature
    // and others that surround it.
```

```
    // Return immediately if action selection type is fifty-fifty
    if ( Creature::m_action_selection_type == FIFTY_FIFTY ) {
        return;
    }

    // Determine the effect on the increase population drive for this
creature
    double satiation_effect = projectedSatiationEffect( MATING,
number_of_ants, number_of_spiders );

    // Update the increase population drive for this creature
    double satiation_level =
m_increase_population_drive.getSatiation();
    m_increase_population_drive.setSatiation( satiation_level +
satiation_effect );

    // Determine the effect on the increase population drive for the
other parent
    satiation_effect = other_creature.projectedSatiationEffect( MATING,
number_of_ants, number_of_spiders );

    // Update the increase population drive for the other parent
    Drive other_creature_increase_population_drive =
other_creature.getIncreasePopulationDrive();
    satiation_level =
other_creature_increase_population_drive.getSatiation();
    other_creature_increase_population_drive.setSatiation(
satiation_level + satiation_effect );
    other_creature.setIncreasePopulationDrive(
other_creature_increase_population_drive );

    // Loop over all surrounding tiles
    Creature* creature = NULL;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
            continue;
        }

        // Loop over creatures in this tile
        CreatureListIterator creature_iter = tile->m_creatures.begin();
        for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
        {
            creature = *creature_iter;

            // If this is an ant
            if ( creature->getType() == ANT_TYPE )
            {
                // Update its destroy satiation level
                Drive drive = creature->getDestroyDrive();
                drive.setSatiation(
```

```
                    drive.getSatiation() -
(Creature::m_observed_mating_satiation_constant * (1.0 /
number_of_spiders)) );
                creature->setDestroyDrive( drive );

                // Update its foreign birth count
                creature->setForeignBirths( creature-
>getForeignBirths() + 1 );
            }
            else
            {
                // Spider - Update its birth count
                creature->setBirths( creature->getBirths() + 1 );
            }
        }
    }
}

//---------------------------------------------------------------
void
Spider::updatesAfterKill(
    const TileList& surrounding_tiles,
    int number_of_ants,
    int number_of_spiders )
{
    // This creature just killed another creature - update drive
satiation levels, etc. of
    // this creature and others that surround it.

    // Return immediately if action selection type is fifty-fifty
    if ( Creature::m_action_selection_type == FIFTY_FIFTY ) {
        return;
    }

    // Determine the effect on the destroy drive for this creature
    double satiation_effect = projectedSatiationEffect( FIGHTING,
number_of_ants, number_of_spiders );

    // Update the destroy drive for this creature
    double satiation_level = m_destroy_drive.getSatiation();
    m_destroy_drive.setSatiation( satiation_level + satiation_effect );

    // Loop over all surrounding tiles
    Creature* creature = NULL;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
            continue;
        }

        // Loop over creatures in this tile
        CreatureListIterator creature_iter = tile->m_creatures.begin();
        for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
```

```
        {
            creature = *creature_iter;

            // If this is an ant
            if ( creature->getType() == ANT_TYPE )
            {
                // Update its increase population satiation level
                Drive drive = creature->getIncreasePopulationDrive();
                drive.setSatiation(
                    drive.getSatiation() -
(Creature::m_observed_fighting_satiation_constant * (1.0 /
number_of_ants)) );
                creature->setIncreasePopulationDrive( drive );

                // Update its death count
                creature->setDeaths( creature->getDeaths() + 1 );
            }
            else
            {
                // Spider - update its kill count
                creature->setKills( creature->getKills() + 1 );
            }
        }
    }
}

//---------------------------------------------------------------
void
Spider::learn()
{
    // Adjust priority parameters

    // Adjust priority parameter of increase population drive
    double ip_priority =
m_increase_population_drive.getPriorityParameter();
    ip_priority += Creature::m_ip_priority_adjustment_constant *
        (m_deaths - m_births) * (m_deaths + m_births);
    m_increase_population_drive.setPriorityParameter( ip_priority );

    // Adjust priority parameter of destroy drive
    double d_priority = m_destroy_drive.getPriorityParameter();
    d_priority += Creature::m_d_priority_adjustment_constant *
        (m_foreign_births - m_kills) * (m_foreign_births + m_kills);
    m_destroy_drive.setPriorityParameter( d_priority );

    // Clear the counts
    m_births = 0;
    m_deaths = 0;
    m_foreign_births = 0;
    m_kills = 0;
}

//---------------------------------------------------------------
void
Spider::teamUpAgainst( const TileList& surrounding_tiles )
{
    // Make other creatures team up against this one
```

```
    // Loop over all surrounding tiles
    Creature* creature = NULL;
    Tile* tile = NULL;
    TileListConstIterator tile_iter = surrounding_tiles.begin();
    for ( ; tile_iter != surrounding_tiles.end(); ++tile_iter )
    {
        tile = *tile_iter;
        if ( tile == NULL ) {
            continue;
        }

        // Loop over creatures in this tile
        CreatureListIterator creature_iter = tile->m_creatures.begin();
        for ( ; creature_iter != tile->m_creatures.end();
++creature_iter )
        {
            creature = *creature_iter;

            // If this is an ant looking to fight and this creature
wants to fight
            if ( creature->getType() == ANT_TYPE &&
                 creature->getAction() == LOOKING_FOR_FIGHT &&
                 getAction() == LOOKING_FOR_FIGHT )
            {
                // Set its target to this one
                creature->setTarget( getTarget() );
                creature->setTargetSet( true );
            }
        }
    }
}
```