Alleviating Escape Panic with Evolutionary Intelligence

Except where reference is made to the work of others, the work described in this project is my own or was done in collaboration with my advisory committee. This project does not include proprietary or classified information.

Shelby Solomon Darnell

Certificate of Approval:

Juan Gilbert
Associate Professor
Department of Computer Science and Software Engineering

Gerry Dozier, Chair
Associate Professor
Department of Computer Science and Software Engineering

John A. Hamilton
Associate Professor
Department of Computer Science and Software Engineering

Stephen L. McFarland
Acting Dean, Graduate School

Alleviating Escape Panic with Evolutionary Intelligence

Shelby Solomon Darnell

A Project

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Engineering

Auburn, Alabama
December 5, 2005

Alleviating Escape Panic with Evolutionary Intelligence

Shelby Solomon Darnell

Permission is granted to Auburn University to make copies of this thesis at its discretion, upon the request of individuals or institutions and at their expense. The author reserves all publication rights.

Signature of Author

Date

Copy sent to:

Name	Date

Vita

Shelby Solomon Darnell was born in a small country cabin in Jasper, Alabama that he helped his father build. Shelby Darnell began life humbly, went to a magnet high school and participated in a pre-engineering program. Shelby went to college at University of Alabama in Huntsville for one year, and then switched over to Auburn University in Auburn, Alabama and obtained a Bachelors of Science in Computer Engineering. After obtaining a Bachelors, Mr. Darnell decided to continue his education and is presenting his work in hopes of attaining a Masters degree in Software Engineering.

Project Report

Alleviating Escape Panic with Evolutionary Intelligence

Shelby Solomon Darnell

Master of Engineering, December 5, 2005
(B.S., Auburn University, 2003)

133 Typed Pages

Directed by Gerry Dozier

Reducing damage, danger, and panic by evolving room designs is possible with artificial intelligence. Escape panic, brought on by groups of people being in a life-threatening situation, increases the fatality rate and level of property damage incurred during unfortunate disasters. Currently buildings are designed to a code that tells how many exits a room should possess, but the code doesn't specify where to place the doors and exactly how many doors there should be in room designs to help alleviate damages to people and property. A genetic algorithm and particle swarm optimizer will be used to find a room design to help alleviate this problem.

Acknowledgments

I acknowledge GOD. I love my Momma. Yay!

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliography follows van Leunen's *A Handbook for Scholars.*

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`. All of the graphs in this work were made using Gnuplot.

Table of Contents

List of Figures x

1 Introduction 1
- 1.1 A Brief Explanation of the Social Force Model 2
- 1.2 A Brief Explanation of Escape Panic 2
- 1.3 Pedestrian Simulator (Pedsim) 3
- 1.4 Evolutionary Computation with Pedestrian Simulation 3

2 Evolutionary Computation (EC) 5
- 2.1 Genetic Algorithms(GAs) 6
  - 2.1.1 Genetic Operators and Terminology 7
  - 2.1.2 A Simple Genetic Algorithm 9
- 2.2 Particle Swarm Optimization (PSO) 10
  - 2.2.1 Swarm Operators and Terminology 11
  - 2.2.2 The Canonical Particle Swarm Optimizer 13
  - 2.2.3 A Simple Particle Swarm Optimizer 14

3 Literature Review 15
- 3.1 Facilities: Layout and Design 15
- 3.2 Crowd Flow and Dynamics 16
- 3.3 Social Force Model 17
- 3.4 Escape Panic Definition 19
- 3.5 Pedestrian Simulation 20
  - 3.5.1 Models 20
  - 3.5.2 Implementations 21

4 Problem Definition 22
- 4.1 Escape Panic 22
- 4.2 Optimizing Room Design 23

5 Evolving Exit Positions 25
- 5.1 Pedsim Review 25
- 5.2 The Room Builder 27
- 5.3 Evolutionary Computations Evolve Room Exits 29

6 Experiments 31
6.1 Base Room Design 31
6.2 Experimental Setup 32
6.2.1 Round A 33
6.2.2 Round B 34
6.2.3 Round C 35
6.3 Fitness Evaluation 36

7 Results Analysis 38
7.1 Round A 38
7.2 Round B 42
7.2.1 Analysis 49
7.3 Round C 49

8 Conclusions and Future Work 53

Bibliography 55

Appendices 59

A Escape Panic Figures 60

B Source Code 71

List of Figures

7.1 Round A – GA Best Geometry, Population Size 10, Fitness 0.153103 40

7.2 Round A – GA Best Geometry, Population Size of 20, Fitness 0.143241 40

7.3 Round A – GA Best Geometry, Population Size 30, Fitness 0.140714 41

7.4 Round A – GA Best Geometry, Population Size 40, Fitness 0.124682 41

7.5 Round A – GA Best Geometry, Population Size 50, Fitness 0.146423 42

7.6 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 1 . 43

7.7 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 2 . 44

7.8 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 3 . 45

7.9 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 4 . 45

7.10 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 5 . 46

7.11 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 6 . 47

7.12 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 7 . 47

7.13 Round B Graphs: Fitness Curves for GA/PSO with Exit Length 8 . 48

7.14 Round C Graphs: Fitness Curves for GA and PSO 51

7.15 Round C – Best GA Geometry 51

7.16 Round C – Best PSO Geometry 52

A.1 Escape Panic Arc 61

A.2 Pedsim Pic 01 62

A.3 Round B – Best GA Geometry with Exit Length of 1 63

A.4 Round B – Best GA Geometry with Exit Length of 2 63

A.5 Round B – Best GA Geometry with Exit Length of 3 64

A.6 Round B – Best GA Geometry with Exit Length of 4 64

A.7 Round B – Best GA Geometry with Exit Length of 5 65

A.8 Round B – Best GA Geometry with Exit Length of 6 65

A.9 Round B – Best GA Geometry with Exit Length of 7 66

A.10 Round B – Best GA Geometry with Exit Length of 8 66

A.11 Round B – Best PSO Geometry with Exit Length of 1 67

A.12 Round B – Best PSO Geometry with Exit Length of 2 67

A.13 Round B – Best PSO Geometry with Exit Length of 3 68

A.14 Round B – Best PSO Geometry with Exit Length of 4 68

A.15 Round B – Best PSO Geometry with Exit Length of 5 69

A.16 Round B – Best PSO Geometry with Exit Length of 6 69

A.17 Round B – Best PSO Geometry with Exit Length of 7 70

A.18 Round B – Best PSO Geometry with Exit Length of 8 70

Chapter 1

Introduction

Imagine being in a burning building. Having knowledge of this fact is enough to scare most people. Furthermore, imagine that you are in a room full to its legal capacity with other people who all know the building is on fire. Now this room has a single exit, the building is on fire, and you are crammed in with everyone else. Unfortunately, this is quite an unwelcome event, but we've not yet reached the point where we can keep this from happening ever again. As an interim solution to this problem, as well as other similar situations, we've developed a tool using evolutionary intelligence and simulation software to help alleviate panic felt by humans in situations similar to the one explained above.

This master's project is centered around a theme that is meant to help people when under situations that cause great panic and duress. Escape panic is a problem that occurs due to life-threatening situations and it sometimes occurs for seemingly no reason at all [14]. Because it is much easier to change the way we build structures than it is to change human nature we have decided to endeavor designing a room so that the design helps alleviate the amount of panic experienced by people in situations that counteract our natural rationale.

1.1 A Brief Explanation of the Social Force Model

Correctly simulating how people act in life-threatening situations has been thoroughly researched [3, 13, 14, 15, 16, 17, 23, 24, 32]. In order to write a simulation for escape panic a model must be used. Such a model has already been developed, and this model is called the social force model [13]. The social force model was developed by Helbing and Molnar in 1998. The social force model was developed citing the fact that "for relatively simple situations stochastic behavioral models may be developed if one restricts oneself to the behavioral probabilities that can be found in a huge population of individuals" [13]. The social force model tracks simulated pedestrians with respect to time and uses several equations related to fluid and gas dynamics that have been changed to relate to pedestrian movement [13].

1.2 A Brief Explanation of Escape Panic

Escape panic is simulated by solving a set of coupled differential equations or by using the social force model for pedestrian dynamics [13] , where the movement of panicked individuals is tracked with time [14]. Evolutionary techniques, a genetic algorithm and a particle swarm optimizer, are used in this work to find room geometries that will reduce the amount of time panic is felt, minimize damage done, and substantially decrease casualties accumulated by individuals attempting to escape from a confined space in life-threatening situations.

1.3 Pedestrian Simulator (Pedsim)

An open-source implementation of Helbing and Molnar's social force model, pedsim (short for pedestrian simulator), used to simulate escape panic was developed by Torsten Werner, an adviser for IT strategy at the German Foreign Office whose former primary research interest was pedestrian simulation and panics, using programming language C++ and a development framework called Qt [28], a platform with which open-source software may be developed for free.

Helbing, Farkas, and Vicsek also developed software to run continuous simulation models of pedestrian motion, and this work is available at their website [16]. Their implementation wasn't used in this work because the other implementation we found was developed to be more extensible.

1.4 Evolutionary Computation with Pedestrian Simulation

In order to design better rooms to help mollify the devastation caused by escape panic, we have used a genetic algorithm and particle swarm optimizer in conjunction with the pedsim software. A genetic algorithm is a search method that simulates evolution in order to find optimal or near optimal solutions to problems with a high amount of complexity and extremely large solution spaces []. A particle swarm optimizer is a search method that simulates social behavior of groups of animals and insects that is helpful for evolution. A particle swarm optimizer is an evolutionary computation very similar to genetic algorithms but it works on a slightly different paradigm. The genetic algorithm or particle swarm

submits different room designs to the pedsim software and causes a simulation to run and obtains the calculation of how quickly pedestrians can escape the room when panicked.

Chapter 2

Evolutionary Computation (EC)

According to Dozier in [8] “Evolutionary computation is the field of study devoted to the design, development, and analysis of problem solvers based on natural selection”. This field of study was motivated by the realization that evolution or natural selection “produce increasingly fit organisms in environments which are highly uncertain for individual organisms” [19].

Evolutionary computations (ECs) are examples of simulated evolution or evolutionary computation that were first proposed thirty years ago. ECs are characterized by using search methods that are biologically inspired. ECs are optimizers that are used to search for optimal or good enough solutions in exhaustively large search spaces without having or attempting to evaluate all possible points within a search space.

As EC is based on natural selection, the algorithms developed based on EC use methods that mimic natural selection. The most notable processes in natural selection which are mimicked in evolutionary algorithms are: selection, crossing over (crossover), mutation, and replacement.

In nature, selection is when members or organisms in a group decide to reproduce together. Crossing over is when organisms swap genetic information during reproduction. Mutation is a quality or trait found in offspring which helps add variety to the population. Replacement qualifies when the population is at its upper

bound, in terms of organisms allowed in a particular group, and a decision must be made concerning whether or not offspring are allowed to stay in the population or not. Replacement allows for organisms to keep balance with their environment. For example, from time to time the male lion in a pride is replaced by a stronger male. During this replacement process the weaker male is forced to leave the pride [25].

Several algorithms, referred to as ECs have been developed in the past thirty years. This work implements and will explain in more detail two such algorithms: genetic algorithms and particle swarm optimizers.

2.1 Genetic Algorithms(GAs)

GAs are optimization algorithms that randomly generate points within a solution space and use biologically inspired techniques like crossover and mutation to generate new points in the solution space. A GA uses a finite set of points in the solution space that it continually modifies to find an optimal solution. A GA is a first generation evolutionary technique originally developed in the late 1950s [7].

GAs are applied to problems whose solution spaces are too large or too costly to appropriately search using other, usually exhaustive, searching methods. For example, the problem being researched in this paper has a solution space with 2^{26} possible solutions. Attempting to evaluate all possible solutions in a solution space this large is very time consuming if it is possible, and using a GA to optimize

such a problem allows one to realize allowably good solutions by evaluating only a fraction of the possible solutions.

2.1.1 Genetic Operators and Terminology

When discussing GAs, there are several terms with which one should be familiar: population, representation, candidate solution, individual, selection, fitness, crossover, mutation, mutation rate, procreation, replacement, and function evaluations or generations.

A population is a set of k ($k > 0$) states which are randomly initialized set at the beginning of the procedure [30]. A representation is a correlation between a GA and the problem it is optimizing. A representation is how the GA views a possible solution to a problem or a form of the problem with which the algorithm can understand and work. A candidate solution or an individual is a single state from the k randomly generated states in the population.

Selection is the process that determines which individuals in the population will be chosen to create new candidate solutions. Selection chooses individuals based on their utility or ability to move toward an optimal value. There are several different types of selection, two of which are random and tournament. Random selection is randomly picking two individuals from the population and using them to create new candidate solutions. Tournament selection uses random selection to pick a number of individuals from the population and then compares the fitness of the chosen individuals to determine which of the chosen individuals participate

in creation of new candidate solutions. Fitness is a value given to each state in a GA that ranks or is indicative of its current utility. Fitness is usually a value that allows for linear ranking of states in the population.

Crossover and mutation are ways to create individuals for the population during the search. Crossing over is the act of selecting at least two distinct individuals from the population, and taking some of the information from each individual to create a candidate solution. Mutation takes this candidate solution and apply random changes to it. Mutation isn't an operation that is applied to each candidate solution, but crossover is usually applied to make each candidate solution. Mutation is applied to new individuals only a given fraction of the time, which is referred to as the mutation rate. If mutation rate is 1% then mutation is applied to one out of every 100 new candidate solutions.

Procreation is an umbrella term under which crossover, and mutation fall. Procreation refers to all three of the afore-mentioned terms as a single operation. Replacement is when the algorithm replaces an individual(s) in the population with a candidate solution(s) developed during procreation. GAs can replace anywhere from one member of the population up to every member of the population with candidate solutions created during procreation. When a GAs replacement procedure replaces either one or two members in the population, the GA is referred to as a steady-state GA. When a GAs replacement procedure replaces the entire population, the GA is referred to as a generational GA. A function evaluation is assigning fitness to a single state in the GA. A generation is a period that

involves procreation and replacement. In a generational GA, a generation would be counted every time newly created candidate solutions replace the entire population. In a steady-state GA a generation would be counted every time a member in the population is replaced by a new candidate solution.

Crossover involves taking more than one individual and combining parts of their representations into a new candidate solution. Mutation involves using a newly created candidate solution and changing its representation in a random manner. Mutation rate denotes how often a new candidate solution is mutated. Procreation is the combined processes of crossover and mutation. Replacement is a process that substitutes newly created candidate solutions for older members or individuals in the population.

2.1.2 A Simple Genetic Algorithm

Listing 2.1 displays a simple GA with pseudo-code from [9]. In listing 2.1 the variable t stands for time. Hence, the two operations "Initialize Population(t)" and "Evaluate Population(t)" occur when time is zero. In a GA the population size is always a number n where $n > 0$.

After creating and ranking the initial states in the GA, selection occurs, where several states from the population are chosen to progress the algorithms search for an optimal solution. The states that are selected to create new states are referred to as "Parents", and all states derived from other states in the algorithm are called "offspring" or "children". Every state in the algorithm whether parent of child may

Listing 2.1: Genetic Algorithm Pseudocode

```
Begin GA
t = 0 ;
Initialize Population(t) ;
Evaluate Population(t) ;
While Optimization is Incomplete
{
  Parents(t) = Select_Parents( P(t) ) ;
  Offspring(t) = Procreate( Parents(t) ) ;
  Evaluate( Offspring(t) ) ;
  Population(t+1) = Select_Survivors( Population(t), Offspring(t) ) ;
  t = t + 1 ;
}
End GA
```

be referred to as candidate solutions or individuals. The offspring or children are then ranked according to the problems evaluation function. After being ranked a new population is selected from the offspring and the parents, which is the selection of survivors. The GA continues this cycle of selecting parents, creating and ranking offspring, and selecting survivors continues until some stopping criteria is met. Some stopping criteria are an upper limit of evaluations, a preset time limit, or stopping after the states in the population have very similar ranking values or fitness's after evaluation.

2.2 Particle Swarm Optimization (PSO)

PSOs optimize "continuous nonlinear functions" which were found by simulating "a simplified social model" [20]. The particle swarm optimizer was born

from the simulation of flocking birds or schools of fish [20]. The ability of flocks of birds to fly synchronously, dissipate for a short interval, and then regroup and re-synchronize into a tight formation is the social model from which PSO was developed. Using this ability, flocks of birds are able to find food sources very quickly in a dynamically changing environment [20]. This social model is shared by other groups of animals including humans. The crux of this model that ties in PSOs with ECs is that "sharing of information among conspeciates offers an evolutionary advantage" [20].

2.2.1 Swarm Operators and Terminology

The candidate solutions for a PSO are called particles instead of individuals, like they are referred to for GAs. In GAs individuals can die or be replaced by more fit individuals or children created during procreation. In PSO particles never die, as in a particle is never completely replaced like an individual in a GA. A particle in PSO is made up of 3 major parts: x-vector, p-vector, and v-vector. The x-vector is the current position of the particle in D-dimensional space ($D > 1$). The p-vector is the best position recorded by the particle, or a copy of the best x-vector it has found. The v-vector is the velocity of PSO, it is responsible for the movement of the x vector. Instead of replacing an entire candidate solution PSOs change the x and v vectors during each search update, and change the p vector when necessary. The x and p vectors also have their own fitness values.

The p-vector and fitness values are updated when the fitness value of the x-vector is ranked as being more fit than the p-vector.

In PSO, the particle position (x-vector) is changed by the velocity (v -vector), and the velocities change is based on learning rates. The following equation is used to change the position of a particle in a swarm using its velocity vector: $x_{i+1} = x_i + v_i$. The learning rates change the velocity of a particle with respect to φ_1 and φ_2, which are respectively cognitive and social components of the swarm. The cognitive component or learning rate φ_1 affects particle movement with respect to its p vector. The social component or learning rate φ_2 affects particle movement with respect to the best p vector of all particles in the neighborhood. There is also a random component to the movement of particles, represented in variables $r_1 and r_2$. The values of these random numbers are drawn from a Gaussian distribution from 0 a to 1. The following equation shows how the velocity of a particle in a swarm is updated: $v_{i+1} = v_i + \varphi_1 r_1(p_i - x_i) + \varphi_2 r_2(p_g - x_i)$. The social behavior of flocking birds is partially based on the flock having somewhere to roost (destination), by basing the movement of the particles on their best and the populations best known particle positions gives the swarm a searching direction.

Updating the velocity of each particle occurs every time a particle moves. Since the velocity is always adding or subtracting from itself it tends to become large and cause the swarm to “lack local search precision” [4]. Eberhart and Kennedy recognized this problem and began limiting the value of the velocity to a preset maximum [22]. Maurice Clerc’s research has produced another method

to constrain the rate a swarms velocities increase with his constriction factor K [5]. Clerc's constriction factor uses the sum of the cognitive and social component values to calculate a value to use when changing a particles velocity. Clerc's equation for his constriction coefficient and the new velocity equation is seen below: $\varphi = \varphi_1 + \varphi_2, K = \frac{2}{|2-\varphi-\sqrt{\varphi^2-4\varphi}|}, v_{i+1} = K(v_i + \varphi_1(p_i - x_i) + \varphi_2(p_g - x_i))$ [4].

Changing the position of the x vectors in PSO to search a solution space is called updating. In PSO during each update single particles check to see whether their current position and fitness is the best they've ever found. If it is the particle then updates its p vector to its current position. When using a global neighborhood, the entire swarm keeps track of the best position of all of the particles in the swarm, or the best p vector in the swarm. There are two ways to update the best p vector in the swarm: synchronously and asynchronously. Synchronous updating of the best p vector involves updating all of the particles in the swarm before assigning a new global best. Asynchronous updating will assign a new global best after any single particle update.

2.2.2 The Canonical Particle Swarm Optimizer

The canonical PSO [4] uses Clerc's velocity constriction factor, K, a cognitive component of 2.8 and social component of 1.3, a population size of 30, asynchronous updating, and a global population. These values were described and researched in [4] as general settings and values for a reasonably good PSO.

2.2.3 A Simple Particle Swarm Optimizer

Listing 2.2 displays a simple PSO with pseudo-code. In listing 2.2 the variable t stands for time or the number of function evaluations the PSO has already completed. The *currentParticle* takes the modulus of the number of function evaluations with respect to the swarm size to get which particle should be updated at time t. Listing 2.2 shows a PSO with asynchronous updating and a global topology.

Listing 2.2: Particle Swarm Optimizer Psuedocode

```
Begin PSO
t = 0 ;
swarmSize = 30 ;
Swarm.InitializeLocations() ;
Swarm.InitializeVelocities() ;
Swarm.SetBestParticle() ;
Swarm.SetFitness() ;
While Optimization is Incomplete
{
  currentParticle = t mod swarmSize ;
  Swarm.UpdateVelocity( currentParticle ) ;
  Swarm.UpdateParticle( currentParticle ) ;
  Swarm.SetParticleFitness( currentParticle ) ;
  IF Necessary Swarm.UpdateLocalBest( currentParticle ) ;
  IF Necessary Swarm.UpdateGlobalBest() ;
  t = t + 1 ;
}
End PSO
```

Chapter 3

Literature Review

Using EC to evolve rooms that optimize escape times for pedestrians in evacuation situations is a subject for which there is currently no concrete research. In order to properly understand and implement ECs to evolve rooms to help alleviate escape panic we reviewed research about: facility layout and design, crowd flow and dynamics, the social force model for pedestrian dynamics, escape panic, and pedestrian simulation.

Crowd flow and dynamics, the social force model, escape panic, and pedestrian simulation all relate directly to the work done for this project, but facility layout and design does not. Facility layout and design is reviewed because what is optimized for its problem is very similar to what is being optimized for finding room designs to alleviate escape panic. In particular [1] uses a GA to optimize the location of input and output points for work centers in the design of a facility to minimize interaction costs between the centers. Our work uses a GA to optimize the location of exits in a room to help pedestrians escape quickly.

3.1 Facilities: Layout and Design

The facility layout problem (FLP) is a NP-hard combinatorial problem, where a planar region is divided into blocks or work centers of given area to minimize the interaction costs between blocks [1]. In [1] GAs are used to locate the best

input and output points in the design of a facility. The representation used by [1] makes changes in the active exits or inputs and outputs for a set facility design , and calculates the cost of material flow between active inputs and outputs between centers. In [11] GAs and genetic programming (another type of EC, based on GA) are used to optimize a generic FLP. The representation of a facility layout in [11] is first a specialized binary tree that evaluates into a candidate solution in the search space and is assigned a fitness measure to be used during the evolutionary process.

3.2 Crowd Flow and Dynamics

In [15] we find that ECs have already been applied to room design for several different room geometries. Room geometries refer to rooms with varying numbers of walls, internal obstacles, that represent a wide range of objects such as furniture and columns, sizes of rooms, and geometrical shape of rooms such as various ovals and polygons. The focus of applying different ECs to room geometries has so far been to evaluate where objects should be placed in a room to manipulate the flow of pedestrian traffic. Another focus of the research in [15] was to find the best and most efficient areas in rooms with different geometries to place furniture and columns.

In [15] we see that ECs have been applied to different spatial areas such as pedestrian crossings to organize crowd flow. It was found that placing objects such as columns in the middle of broad walkways helps keep pedestrian traffic organized. For example, placing columns in a large corridor like space will make

pedestrians designate sides for traveling in opposing directions. It has been noted that in such situations pedestrians will liken the divided lanes to car lanes. If you are in a country where people drive on the right side of the road, then when found in corridors divided in half length wise by some obstacle pedestrians will tend to form the same habit. It also has been found, using ECs, that having a door that is twice the size of a single door does not help guide traffic as well as intelligently placed dividers or obstacles. Current examples of placing dividers or obstacles length wise through wide corridors can be observed in such places as malls and train stations. Helbing et.al. also propose in [15] that ECs can be used to optimize the location and form of planned buildings; to arrange walkways, entrances, exits, staircases, elevators, escalators, and corridors; find the best shape of rooms, corridors, entrances, and exits.

Kirkland and Maciejewski [24] manipulate crowd dynamics in simulations using autonomous robots that strongly attract pedestrians to find efficient crowd flows. They make the robots take certain positions and actions within the crowd to see what type of influence will encourage lane formation within hallways where pedestrian traffic can flow against itself.

3.3 Social Force Model

Helbing and Molnar develop and explain the social force concept and then model in [13]. The description that follows is a summary of the social force model as explained in [13]. The general idea behind the social force model is based on

the perceived comfort of pedestrians in populated areas. People by nature desire to have a certain area of free space around themselves in order to correctly react, without danger of experiencing pain, to unforeseen circumstances. One would naturally think that this area is spherical in nature, but contrary to the most common belief, these areas of comfort are elliptical and not centered from an individuals physical equilibrium. People naturally need or feel most comfortable with more free space in front of them rather than to their sides, while they do not place high importance of how much free space is behind them given their field of vision.

People can not see behind them so their perceived level of comfort doesn't include how much free space they have to their rear [13]. A person's perceived level of comfort varies depending on the type of obstacle in their path and their environment [13]. For instance, in a crowded room where several groups of people are acquainted with each other their perceived level of comfort when around each other, in their acquainted group, is much higher than when surrounded by other unknown individuals in the same room. In panicked situations perceived levels of comfort are almost completely disregarded. In a panicked situation the perceived level of comfort between unacquainted and acquainted pedestrians becomes less apparent and the priority of the perceived level of comfort bases itself more on distance from harmful obstacles that are not other pedestrians. Say for instance in a room where one wall is on fire, the perceived level of comfort of the people in

this situation becomes more strongly affected by their distance from the burning wall rather than their distance from each other.

In other words, a person "acts as if he/she would be subject to external forces" when they slow down or speed up according to changes in their environment [13].

3.4 Escape Panic Definition

Escape panic or crowd stampede induced by panic is "one of the most disastrous forms of collective human behavior" [14]. Escape panics can be summarized by the following nine characteristics:

(a) people move or try to move considerably faster than normal,

(b) individuals start pushing, and interactions become physical in nature,

(c) moving and, in particular, passing of a bottleneck becomes uncoordinated,

(d) at exits, arching and clogging are observed,

(e) jams are building up,

(f) physical interactions in the jammed crowd produce pressures that exceed 4,450 Newtons per meter, which can bend steel barriers or tear down brick walls,

(g) escape is slowed by fallen or injured people,

(h) people show a tendency of mass behavior(do whatever they see others do, like in grade school),

(i) alternative exits are often overlooked or not efficiently used.

– [14]

3.5 Pedestrian Simulation

Pedestrian simulation is not a simple task. A lot of research has been performed to develop competent models for pedestrian behavior in order to simulate things like escape panic and crowd flow.

3.5.1 Models

Kirchner uses a bionics-inspired cellular automaton model to simulate pedestrian behavior in [23]. This model does not allow pedestrians to be directly influenced by each other, but by positions on the floor to which they may move. In this model a simulation space is divided into small cells that can be empty or not. Each cell can accommodate only one simulated pedestrian, and once in a cell a pedestrian can move to any adjacent cell that is empty, as long as the empty cell is diagonal to the rectangular shaped cell being inhabited.

Helbing et.al. use the social force model for pedestrian dynamics. This model uses mathematical equations to determine an individual pedestrians velocity with respect to its obstacles [13]. Each pedestrian has a field of comfort and tries to maintain a certain distance from other pedestrians and various other obstacles (columns, walls, fire). Under panicked situations this field of comfort decreases according to the severity of the panic [13].

3.5.2 Implementations

Pedestrian simulation has been achieved on the computer using the C [16], C++ [34] and Java programming [18] languages along with equations from the social force model [13]. A few different implementations of pedestrian simulations derived from the social force model exist. The easiest way to perceive the results of the calculations derived from equations in the social force model is to see a graphical representation. The different pedestrian simulation software packages explore different technologies to create their graphical representations. Two of the software packages that have been used to implement graphical representations of pedestrian simulation are openGL and Qt. OpenGL, [27], is a vendor-independent API for the development of 2D and 3D graphics applications. Qt, [28], is a complete C++ application development framework with tools for cross-platform development and internationalization.

The pedestrian simulator off of which this work is a result was developed using the Qt development framework. The simulation model used for this work is based off of software named pedsim developed by a Torsten Werner under the Gnu General Public License [12]. This model was chosen because the GPL (General Public License) allows for free usage, distribution, and modification of all software developed under this license. His model implementation is referred to as simply Escape Panic Simulation. The authors of both software implementations of the pedestrian simulators followed the model presented in [14].

Chapter 4

Problem Definition

Our problem is to see how well GAs and PSOs can design the exits to a certain type of room design. Within this problem we must dynamically create room designs, and before that we must find an efficient representation of a room design. We will use a GA and a PSO to evolve exits for a room in order to determine what type of exit positions help alleviate escape panic. This research will potentially decrease fatality rate due to better room design as well as help keep buildings intact by making it less likely that pedestrians create destructive forces that can undermine the structure of pedestrian facilities. Understanding how dangerous escape panics can be as well as the difficulty involved with optimizing room design will help you to better comprehend the significance of our problem.

4.1 Escape Panic

It has been observed in confined spaces, when life-threatening situations ensue, individuals tend to be extremely selfish and disregard the welfare of others so that they may relieve themselves of the adverse situation. In cases like these humans tend to push, pull, grab, and trample each other in order to exit a confined space. People act in this way because they become panicked and do whatever they can to alleviate this feeling. Groups of people becoming panicked due to being in a life-threatening situation in a confined space while trying to find an exit is called

escape panic. Escape panic has been shown to be very costly in terms of fatalities and property loss [32].

Escape panic causes people to stop thinking about others and get out of the situation that caused them to panic as soon as possible; at this moment the weaker members in the population involved in this predicament are often trampled and end up dying because of the life-threatening situation or because they are crushed by the stronger members of the population. During escape panic people have been observed forming an arc around an exit and pushing upon each other to get out. The amount of pressure caused by escape panic on the structure of a room can be more disastrous than the situation that caused the crowd to panic. Appendix A figure "Escape Panic Arc" displays exactly what pedestrians look like from above when all are clamoring to exit a room.

4.2 Optimizing Room Design

So far we've seen evolutionary algorithms used to plan facilities and the placement of obstacles in high traffic areas [1, 15]. We've seen that evolutionary algorithms can develop designs that help control pedestrian traffic and organize peoples movements. Besides evolutionary computations there are several regulations and guidelines that are used to instruct architects, city planners, and construction workers on their designs and buildings [26]. For instance there is a set of industrial planning guidelines that tell how many exits a room must have depending on its size and how many people it can hold at one time. What isn't in place are

regulations or guidelines that make people design buildings and rooms so that the number of exits and their positions in the room help to alleviate escape panic if the situation were to arise.

Developing a room design that can help more quickly relieve such a situation would greatly improve the fatality rates in already life-threatening situations, while keeping other accidents from occurring like premature collapsing of structures due to pressure on an exit. To design such a room we will use the existing pedsim software and submit to it room designs. The amount of time that the simulation takes to show the pedestrians exit the room will be the means by which we evaluate the design of the room. Pedsim will simulate pedestrians in the room who are all panicked and begin to act according to the social force model.

Chapter 5

Evolving Exit Positions

5.1 Pedsim Review

The first task for this work was to go through pedsim and figure out how to correctly interact with it. Pedsim uses configuration files to define several variables that is uses for simulation. Some of the variables that are defined by the configuration files are the length and width of the room, the configuration name, the number of pedestrians in the room, the minimum and maximum allowed size for the randomly sized pedestrians, and several other variables that are necessary for the social force model.

Depending on the room design you desire to use for simulation one can specify the length and the width of the room. Depending on the situation you want to simulate, as well as the amount of time you look to spend on a single simulation, you may vary the number of pedestrians that are created inside of the room. Pedsim is built so that the number of pedestrians that can be generated inside of any room is dependent on the room dimensions. In a room that is eight meters long and five meters wide a maximum of one hundred pedestrians can be placed into that room. Since this is the maximum and pedsim was designed to simulate escape panic the default value for the number of pedestrians placed into a room is the maximum possible for every different room design. The one hundred pedestrian limit in a room eight meters long and five meters is also based on the size limits

of the pedestrians. The values for the size limits of the pedestrians are based on typical scaled values for the shoulder widths of men and women. All of the variable values are based off of real values. The configuration name chooses the room design to which the other configuration variables will be applied.

All of the room designs for pedsim are static and placed into separate files. In other words, in order to simulate a room that is 8 x 5 meters with a single door in the middle of the left wall and another similar room with a single door in the middle of the right wall, two different configuration files must be used with different configuration names. Pedsim is an application that is designed to do one simulation per execution; there is no option within pedsim that allows one to run more than one simulation at a time. In order to use a GA to find the best room design for alleviating escape panic we had to be able to run pedsim continually with varying room designs. Two problems solved for this master's work was figuring out the best way to modify pedsim to run serial simulations on dynamic room designs.

The first concern was figuring out a way to build different geometries. Initially, to figure out the best way to build geometries, we compared the different hard coded geometries to see how similar the files were, and whether there was a methodical way to build geometries. Unfortunately, most of the geometries were static and used several different (and some unnecessary) classes and methods available in pedsim. Some of the classes and methods available to build geometries can create horizontal and vertical walls, create corners, create horizontal and vertical gates (exits or doors), as well as perform several different mirroring operations

on all other classes and methods. The hard-coded geometries use different classes and methods to do the same operations. For instance, some geometries use walls and gates to make a room, while other geometries use corners, gates, and walls with mirroring to achieve the same effect. To test whether the simulator produces different results for geometries made using mirroring and corners, as opposed to just walls that correctly connect at corners, we created new hard-coded geometries using the different classes and methods for the same design. After comparing the two geometries using the same configuration no differences, that couldn't be attributed to the random number generators, were found in the results of the hard-coded geometries. With this knowledge we decided to not use corners and mirroring to generate geometries. Instead we used horizontal and vertical walls and gates, which simplified part of our task.

5.2 The Room Builder

After making a few simple geometries for pedsim and testing them to figure out which classes and methods needed to be used for making valid rooms for the simulation of escape panic, we endeavored to make modifications that enable dynamic creation of geometries instead of having only static geometries. A room builder was developed to build the most simple geometries. A most simple geometry is a room that is rectangular and has a varied number of doors and positions for the doors. The room builder can take different dimensions for the length and width of a room as well as the length of the door. After taking these dimensions

into consideration the room builder divides the length and width of the walls by the length of a single door to get the total number of doors possible for the room size. The calculation does not include space between doors because the object is for the GA to find the best room geometry. The best geometry may call for doors that are twice or three times as wide as a single door. We believed this to be the best choice for the room builder because upon expansion, a requirement of a simulated room may be for it to have at least one set of double doors, or a slightly larger exit may aide evacuation.

When creating exits using pedsim one must be careful of the sequence in which the door positions are presented to the exit object. If presented incorrectly, instead of getting an exit, the exit that was supposed to be one meter in length becomes an obstacle and the exit field is displaced to all other walls to both sides of the exit.

The room builder calculates exits based on the length of the walls and doors. If the length of a wall is not a multiple of the exit length the room builder will use the center section of the wall that is a multiple of the door length as possible positions for exits. All of the possible exit information for a room of a given length and width is stored in a simple data structure that maintains the starting and ending positions of every possible exit in a room. This simple data structure also maintains the number of possible exits for a room with a given perimeter.

Before creating the room builder all geometries were static and different geometries were represented in separate files with corresponding configuration names.

In all of the static geometries each wall, exit, and corner were defined separately and pushed into a collection that pedsim references to build a room. Using the idea of pushing information into a collection, the room builder was able to dynamically build geometries given certain inputs by pushing every section of a room geometry onto a collection. This is possible by way of reading the position information from the special data structure and combining it with the inputs from the room builder.

The side by side positioning of the exits makes the dynamic scheme of creating geometries easy to represent with a binary string, or a sequence of ones and zeroes. The input to the room builder is a binary string that acts as a map telling the room builder how to create a room. The length of the binary string represents the number of possible doors in the entire room. The positioning, which is pre-calculated, is matched with the elements in the binary string. For the room builder, an element value of zero means that a wall is present, and an element value of one means that an exit is present. To correctly build a room the total number of possible doors and the number of elements in the binary string must be equivalent.

5.3 Evolutionary Computations Evolve Room Exits

Due to the nature of life-threatening situations that cause escape panic people discontinue compliance of normal social rules or constraints [17]. For instance, it has been observed in documented crowd stampedes that “people are obsessed by short-term personal interests” [17]. Currently there are safety standards that outline how many exits are necessary for confined spaces of certain sizes, shapes

and capacity [26]. These standards do not consider the optimal positioning and number of doors to help reduce escape panic. In order to address this oversight we will use a genetic algorithm and particle swarm optimizer to evolve room geometries and simulate escape panic in the rooms developed using our ECs to design rooms that help alleviate escape panic.

We will use a steady-state GA and binary PSO using canonical settings [4] to evolve an optimal number of exits and their positions for rooms of a specific size. The ECs will then run pedsim to simulate escape panic and use as a fitness value the amount of time it takes for all individuals to exit the room. If an individual, attempting to leave the room, ends up dying, pedsim reflects this information in the fitness value of probable room geometries because that person becomes an extra obstacle which slows the progress of other simulated pedestrians.

Chapter 6

Experiments

6.1 Base Room Design

Escape panic was simulated on a rectangular room eight meters long and five meters wide, with exits measuring a single meter in width. The base pedsim software was built with hard-coded geometries or pre-defined room definitions (the definition includes the shape, size, number of exits, and positions of exits). There was no feature in the software that facilitates building various room geometries with varying numbers of doors or other obstacles. Pedsim is the software solution that incorporates a third party application development framework, Qt, in order to visualize the escape panic simulation. For this work, that feature of the software was disabled after having tested an addition to the software that allowed for dynamic creation of room geometries for a rectangular room. The original version of pedsim was used to make sure that the room builder addition correctly created geometries. The pedsim software used in this work was also checked and tested against the software developed for Helbing et.al's work presented in [14] and found at [16].

6.2 Experimental Setup

There were three rounds of experiments done for this work. The different rounds will be referred to as: rounds A, B, and C. Simulation constants for the experiments were: the room dimensions (8 by 5 meters), the exit length (1 meter), the smallest and largest diameter for pedestrians (between .5 and .7 meters), all of the settings for the correct execution of the social force model that controls the simulation, and the number of panicked pedestrians placed in a room. Simulation values that changed during the experiments were: the number of exits in a room and the placement of exits in a room. Round A dealt only with the GA. Round A was a preliminary round of experiments, and it helped us figure out how to best set parameter values for the GA. The round A experiments also helped us come up with more efficient way to run our experiments because we began to understand what was the most important information we wanted from each experiment and the format in which to record it. In rounds A and B we optimized escape time as the fitness. In round C we modified our fitness value to optimize two objectives: escape time and fitness. Each simulation requires a random seed. This means that two simulations run using the same geometry can produce different escape times. In order to get a better escape value for each geometry we must run the simulation several times on each geometry using different random seeds. In experiment rounds A and B we run our simulation ten times on each geometry and take the average value as our escape time. In round C we run our simulation twenty times on each

geometry with different random seeds and use the average as the escape time for that geometry.

In the GA several values were changed, like the mutation rate and population size, in order to find optimal settings for use on our particular problem. For the PSO we simply used the canonical PSO settings established by Carlisle and Dozier in [4]. Rounds A dealt with finding the correct parameter settings and function evaluations for a GA. In round B, because we didn't have a set limit for the number of exits in a geometry and we were only optimizing escape time, we would set the maximum number of contiguous exits to see whether or not exit size aides evacuation time. Round C is where we begin optimizing the escape time and the number of exits. In round C we do not set the maximum number of contiguous exits because our aim in this is to see what types of geometries our ECs evolve when optimizing the escape time and the number of exits.

6.2.1 Round A

The first round of experiments was to find good parameter settings for and function evaluations for the GA. We used a steady-state GA with tournament selection that chose four members from the population and selected two to use during procreation. We ran the GA once with a population size of 10 and mutation rate of 0 for 2500 function evaluations to see how many function evaluations we would need for our first round of experiments (we chose a population size of 10 because for the experiments we planned on incrementing the population size by

10, and a population size of 10 allows for moderate selection pressure, much better than a population size of 5 or less). We plotted the best fitness values from this initial experiment and found that the curve reached a point where the fitness value stopped improving, which signifies an optimal solution, around 250 function evaluations; hence, all of the round A experiments were run using 300 function evaluations. After deciding on the number of function evaluations for each GA experiment we ran tests with mutation rates of 0, 2, 4, 6, 8, and 10 and population sizes of 10, 20, 30, 40 and 50. Each experiment was run 10 times each and the results were all averaged and compared. The fitness value came from an aggregate function that combined the escape time and number of exits for a geometry. The aggregate function calculates the number of exits as half of the fitness value and the escape time as the other half of the fitness value. The value for the number of doors is taken from the equation $FitnessValueforExits = \frac{0.5 \times (ExitNumber)}{TotalExits}$. The value for the escape time was derived in a similar way, but instead of exit number we used escape time and for total exits we used a tested value that was the upper bound on how long it took for pedestrians to exit a room with a single exit. There were fifty pedestrians in each simulation in this round.

6.2.2 Round B

This round of experiments is were we changed methods. Instead of running each experiment ten times each, we ran each simulation five times in succession and averaged escape times to get better fitness values. In this round of experiments

we use a geometry's escape time as fitness. We also began experimenting with a PSO. Our PSO was the canonical PSO which uses a population size of 30, a φ_1 of 2.8 and φ_2 of 1.3, asynchronous updating, Clerc's constriction coefficient, and a global neighborhood. The object of this round of experiments was to constrict how many contiguous exits could be present in any geometry. Constricting the number of contiguous exits is simply limiting the exit width. For instance, it is possible for the ECs to evolve a geometry without walls. Our ECs evolved room geometries where the number of contiguous exits was limit to 8, 7, 6, 5, 4, 3, 2, and 1 doors. There were twenty-five pedestrians in each simulation in this round.

6.2.3 Round C

This round of experiments quadrupled the number of simulations we ran on each geometry from 5 to 20. The reason to increase the number of times we simulated a single geometry was to get a better averaged fitness value for each design. This round of experiments uses another aggregate function to calculate fitness values from the number of exits and escape time of a geometry. Instead of the functions used in round A to get partial fitness values from the escape time and number of exits, we implemented an easier approach. The aggregate function for this round of experiments adds the number of exits to the escape time for the fitness value. This new fitness calculation is used and is helpful because it bias' evolution towards geometries with very few exits and is very simple computationally. Round C's aggregate function is by no means optimal, but with the results from this

work we can set benchmarks when using a true multi-objective GA or PSO. We used 400 function evaluations for each experiment in round C. There were also 25 pedestrians placed in the rooms for the simulations in this round.

6.3 Fitness Evaluation

Initially, for the first two rounds of experiments, we let the escape time, or time it took for all pedestrians to exit a room, be the fitness value. This value was calculated by the simulation for each room geometry it was presented. We initially chose to rank the geometries this way in order to find the best geometry despite the number of exits in the room. The second way we decided to get fitness values was combining two aspects found in the simulation as fitness: the amount of time that it takes for all pedestrian's placed in a room to escape (escape time) and the number of exits in the geometry. The fitness function combines these two aspects into a single value. This method of combining to objectives into one fitness value is using aggregate functions for multi-objective optimization. Combining several objectives for an optimization function "was the first technique developed for the generation of non-inferior solutions for multi-objective optimization" [6]. The fitness value is the escape time added to the number of exits. This combination of values was decided upon because we desire to minimize both the number of exits and the escape time. With this combination if the geometry has a high number of exits and does not have a relatively fast escape time it will get phased out of the population; if there is a low number of exits and the escape time is too slow, the

individual representing this cause will also be phased out of the population during evolution. Using an aggregate function for multi-objective optimization fails to "generate proper Pareto optimal solutions in the presence of non-convex search spaces", but it can be used as an "initial solution for other techniques" [6].

CHAPTER 7

RESULTS ANALYSIS

7.1 Round A

Rooms with four to seven exits have been found to most efficiently allow pedestrians to escape the eight by five meter enclosure. Rooms with more exits were always penalized by the GAs fitness evaluation because the number of doors and the exit time associated with the number of doors did not produce a value low enough to counteract a smaller number of exits. The fitness values in this experiment are based equally on the number of exits and the amount of time it takes people to escape the room.

In order to get a feel for the best configuration to evolve the geometries with our GA we used 300 function evaluations and mutation rates of 0, 2, 4, 6, 8, and 10 percent with population sizes of 10, 20, 30, 40 and 50. Each trial configuration was run ten times so that the results could be averaged and we could get a better idea of which configuration is best for our problem.

At higher percentages, percentages above 6, the GA tends to find the best candidate solutions or geometries. When obtaining the average fitness, escape time, and exit count for every run of the algorithm, where a run is allowing the GA to submit 300 different geometries to the pedestrian simulation. We find that the best performing runs were those with these higher mutation rates. In particular a mutation rate of 10 yields the best results among this set of experiments. The

graphical representations of all of these geometries can be found in the appendix with the labels Figure 7.1,Figure 7.2, Figure 7.3, Figure 7.4, and Figure 7.5.

The configurations that were found in the data to allow pedestrians to escape most quickly are not the pinnacle of the room design, but they have the aspects that show great promise. In other words the figures all share aspects that lead them to allow for quick pedestrian evacuation. Notice that four out of five of the geometries have at least one double door (where a double door is two exits that are side by side), and the geometry that allows pedestrians to escape most quickly has a double door and a triple door on the opposite wall. The only geometry without a double door has exits at every corner. Also notice that the geometries found to be most efficient have exits on at least two walls. The only geometry, Figure 7.3 on Page 41, with exits on only two walls has extra large exits on both walls. From these geometries we see that an important element in alleviating escape panic is having exits in different areas of the room to keep panicked people from all converging in a single space. The aggregate function that combined the escape time and exit count into a single number n, where $(0 < n < 1)$, only evolved rooms with between 4 and 7 exits, out of a possible 26.

The primary information from round A experiments that is used in both rounds B and C are the population size of 40 and mutation rate of 10 percent for the GA.

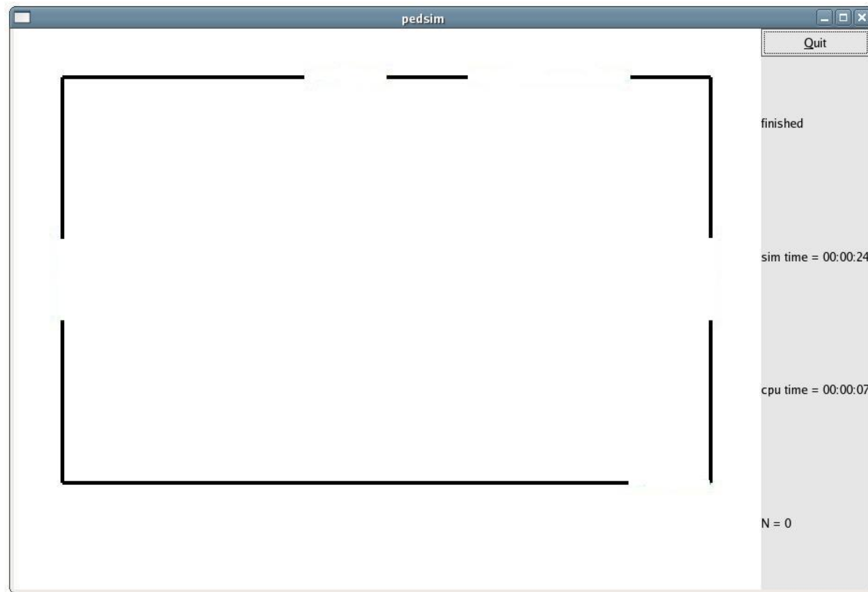

Figure 7.1: Round A – GA Best Geometry, Population Size 10, Fitness 0.153103

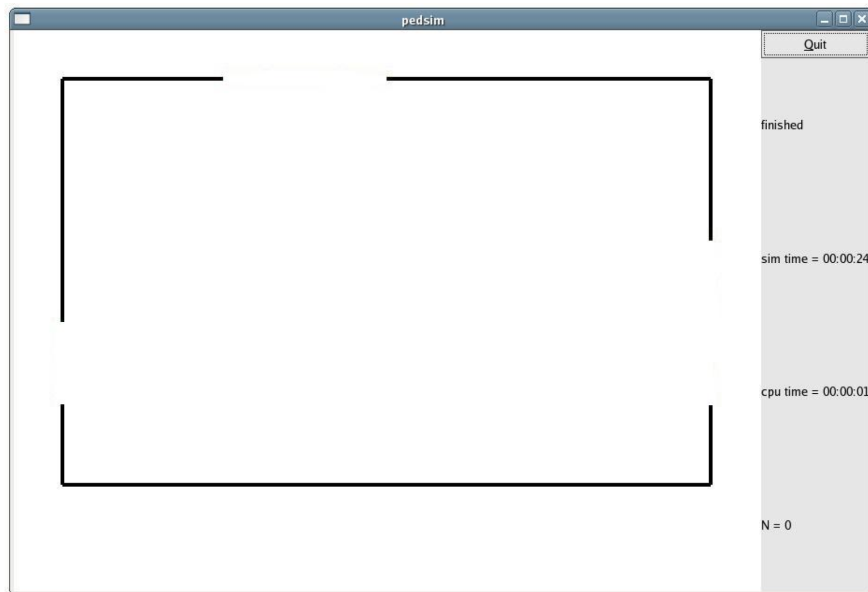

Figure 7.2: Round A – GA Best Geometry, Population Size of 20, Fitness 0.143241

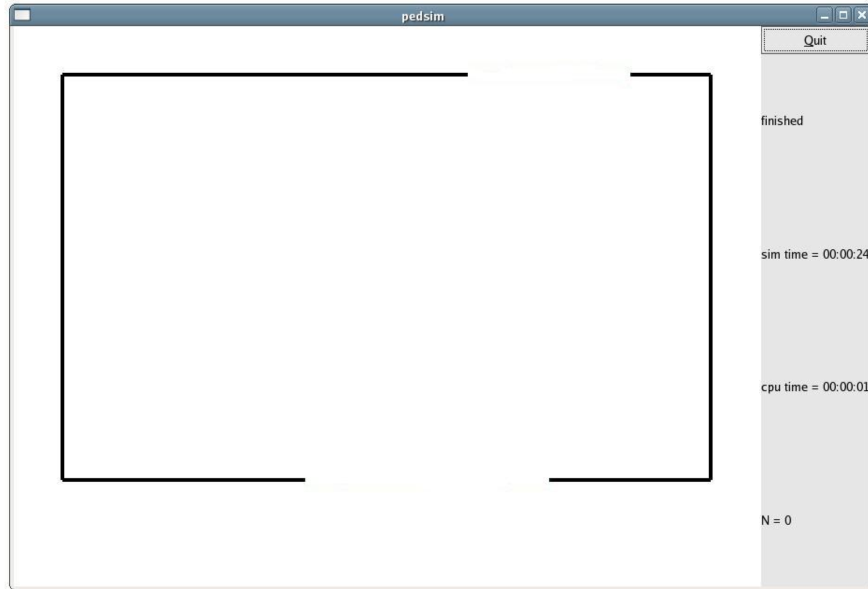

Figure 7.3: Round A – GA Best Geometry, Population Size 30, Fitness 0.140714

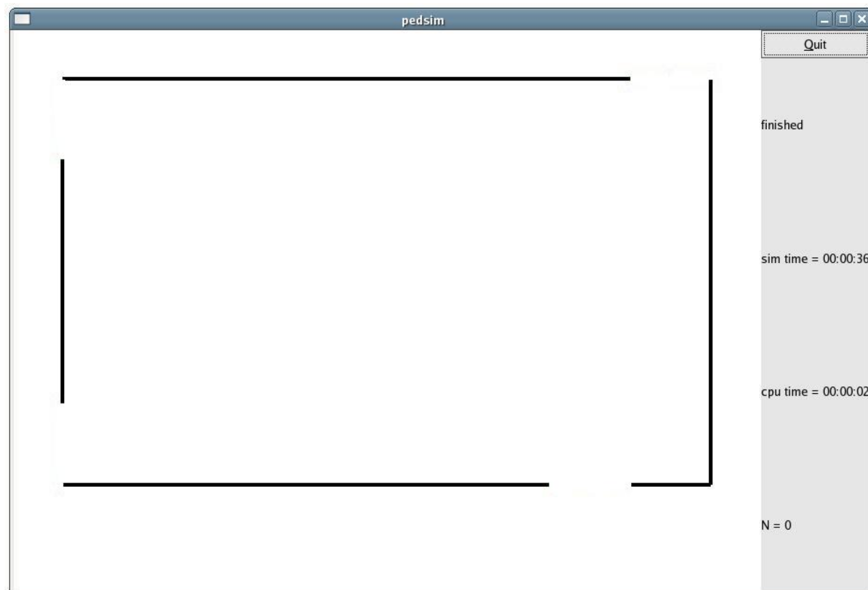

Figure 7.4: Round A – GA Best Geometry, Population Size 40, Fitness 0.124682

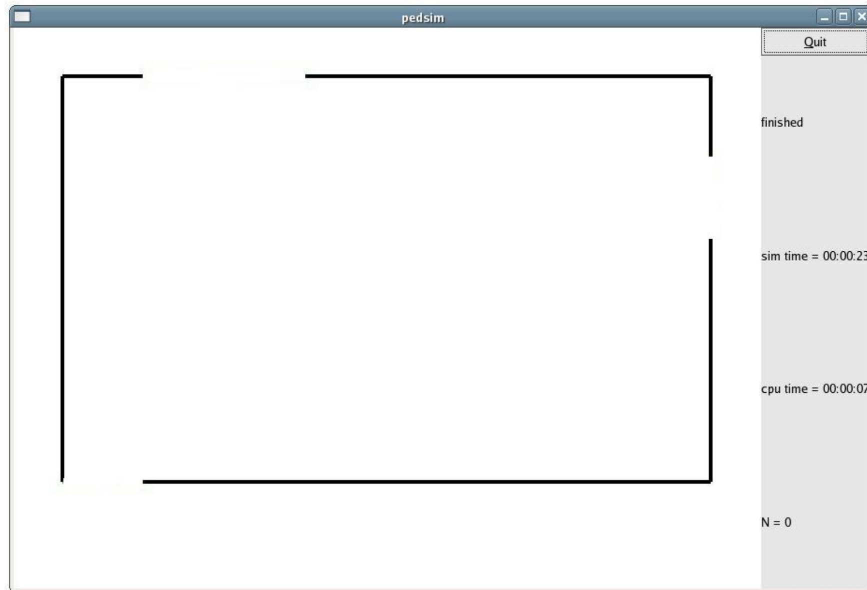

Figure 7.5: Round A – GA Best Geometry, Population Size 50, Fitness 0.146423

7.2 Round B

Round B experiments were primarily testing different exit lengths to see whether small or large exits are preferrable. During this set of experiments we use the escape time as the fitness value. We tested geometries with maximum lengths of 1, 2, 3, 4, 5, 6, 7, and 8 meters. Round B includes the use of PSOs along with GAs to optimize the geometries with varying maximum exit sizes. Each geometry in this set of experiments was simulated 5 times before getting a fitness value assigned and the number of generations was 800. Figure 7.6 displays the minimum fitness graphs for a GA and PSO with exit length 1.

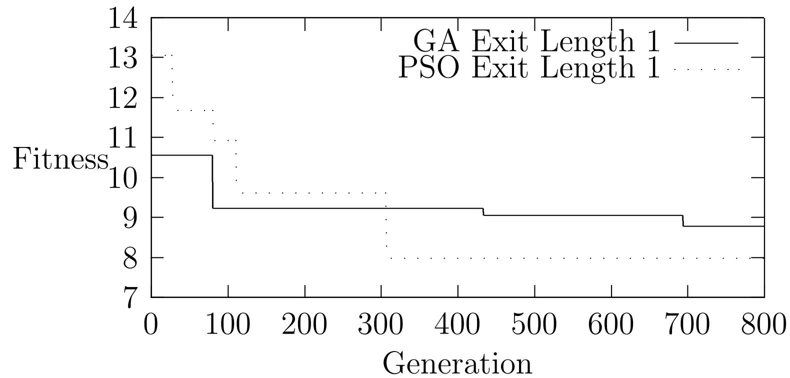

Figure 7.6: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 1

The PSO in Figure 7.6 on finds a better escape time than the GA in the allowed function evaluations. With a room 8 meters long and 5 meters wide there are 26 possible exits. When limiting the exit width to one meter only 13 exits may exist in any design. The best geometry found by our PSO and GA contain 11 out of the 13 possible exits.

The GA in Figure 7.7 on page 44 finds a better escape time than the PSO in the allowed function evaluations. The GA uses 11 exits in its evolve and the PSO uses 14 exits in its design. The maximum possible number of exits for a geometry with a maximum exit width of 2 meters is 17.

A maximum exit length of 3 with a possibility of 19 exits the GA finds a better escape time than the PSO; the graph of the GA and PSO can be seen in Figure 7.8 on page 45. The winning geometry, evolved by the GA, has 16 exits, and the losing geometry, evolved by the PSO, has 16 exits.

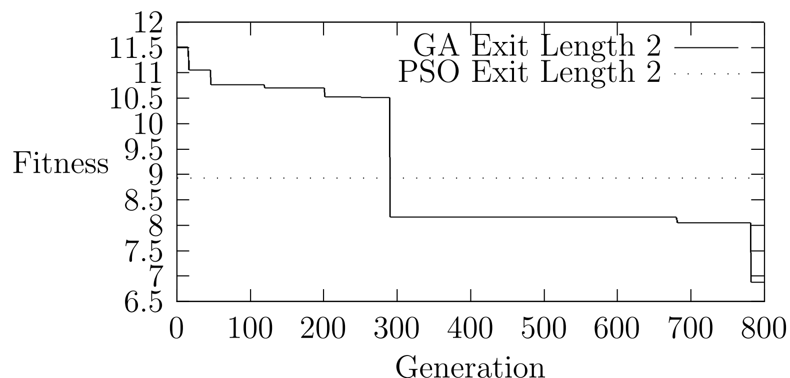

Figure 7.7: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 2

A maximum exit length of 4 with a possibility of 20 exits the PSO finds a better escape time than the GA; the graph of the GA and PSO can be seen in Figure 7.9 on page 45. The winning geometry, evolved by the PSO, has 15 exits, and the losing geometry, evolved by the GA, has 14 exits.

A maximum exit length of 5 with a possibility of 21 exits the PSO finds a better escape time than the GA; the graph of the GA and PSO can be seen in Figure 7.10 on page 46. The winning geometry, evolved by the PSO, has 17 exits, and the losing geometry, evolved by the GA, has 15 exits.

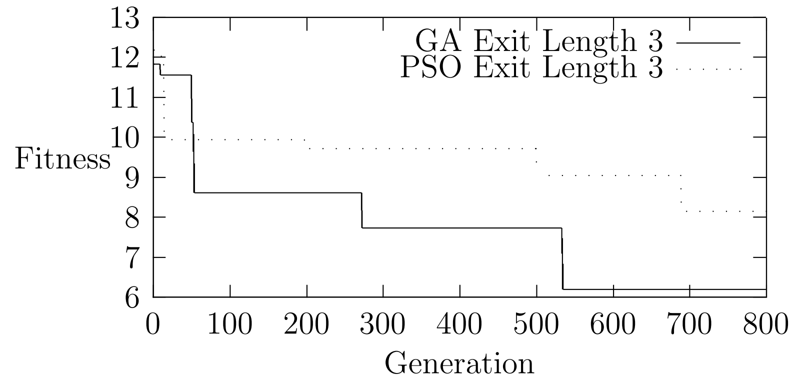

Figure 7.8: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 3

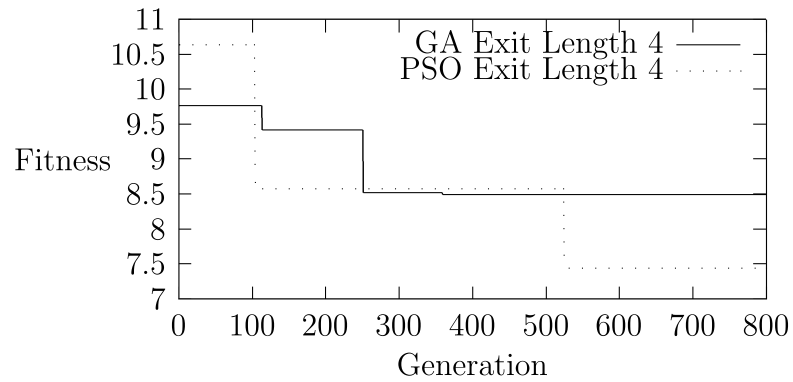

Figure 7.9: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 4

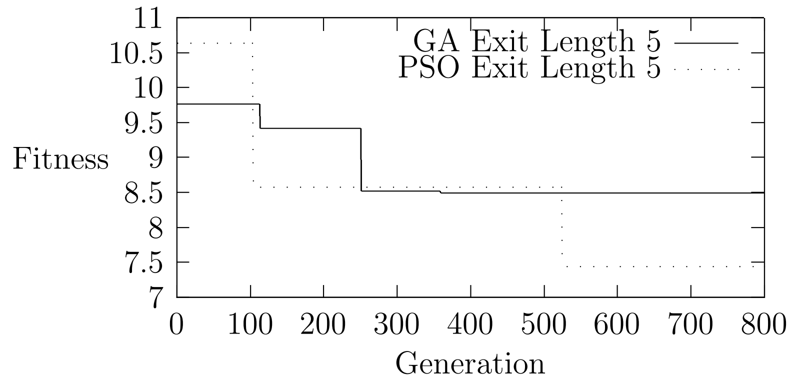

Figure 7.10: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 5

A maximum exit length of 6 with a possibility of 22 exits the PSO finds a better escape time than the GA; the graph of the GA and PSO can be seen in Figure 7.11 on page 47. The winning geometry, evolved by the PSO, has 18 exits, and the losing geometry, evolved by the GA, has 17 exits.

A maximum exit length of 7 with a possibility of 22 exits the GA finds a better escape time than the PSO; the graph of the GA and PSO can be seen in Figure 7.12 on page 47. The winning geometry, evolved by the GA, has 15 exits, and the losing geometry, evolved by the PSO, has 18 exits.

A maximum exit length of 8 with a possibility of 23 exits the GA finds a better escape time than the PSO; the graph of the GA and PSO can be seen in Figure 7.13. The winning geometry, evolved by the GA, has 15 exits, and the losing geometry, evolved by the PSO, has 13 exits.

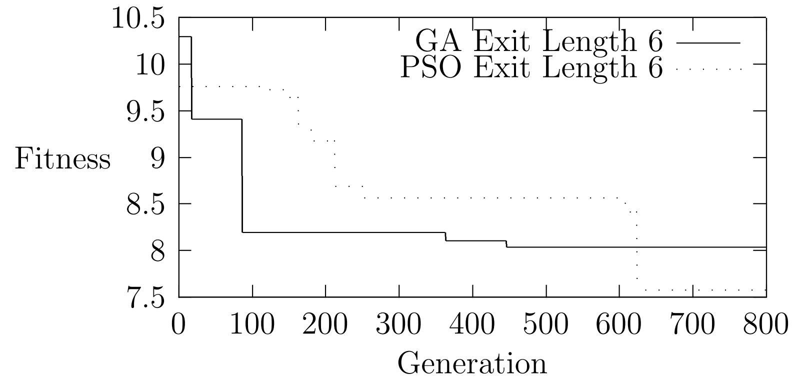

Figure 7.11: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 6

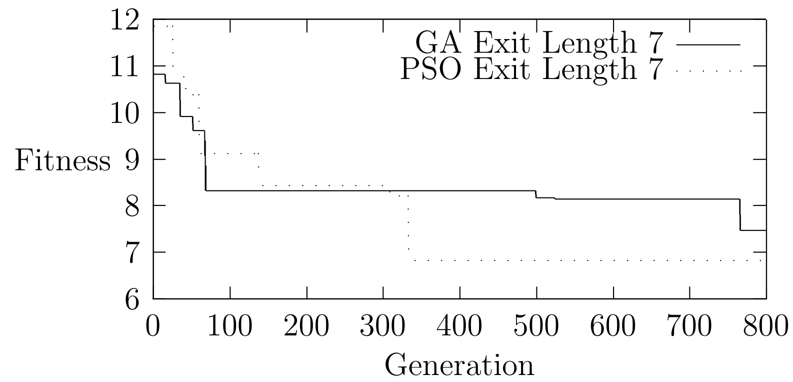

Figure 7.12: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 7

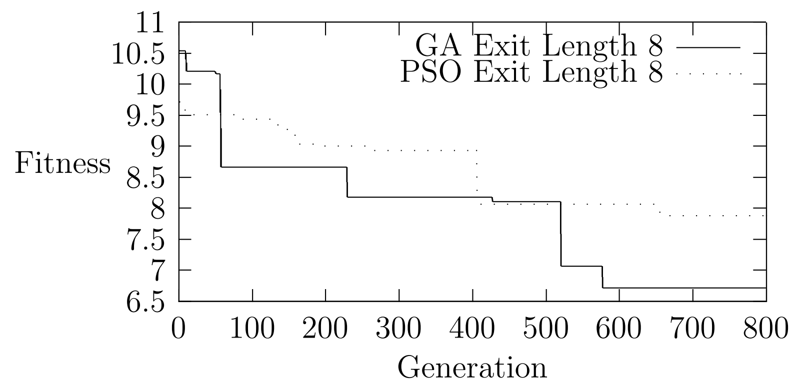

Figure 7.13: Round B Graphs: Fitness Curves for GA/PSO with Exit Length 8

7.2.1 Analysis

One of the first things we noticed when looking over these results is the large number of exits found in each geometry. Although each geometry has a large number of exits, none of the geometries in these experiments were evolved with the maximum number of exits allowed for the limits imposed on them through limiting the exit length. Futhermore, the largest number of exits used by any geometry was 18, and the smallest number of exits used by any geometry was 11. The escape times with the 25 simulated pedestrians for all of these geometries lies within 6 and 10 seconds. With this we realize that after having a certain number of exits open in a room there is a threshold where the escape time of pedestrians does not make significant improvement. After around half of the allowable doors in a geometry are open the placement of the doors has little effect on the escape time. Because of this, we decided to use an aggregate function again to make the GA and PSO evolve and find room designs that not only allow speedy escape, but also minimize the number of exits in a room.

7.3 Round C

In round C we simulated pedestrians escaping from each geometry 20 times and averaged the escape times for a single escape time for that geometry. We also used an aggregate function that combined the escape time and exit number to form a fitness value. The aggregate function simply adds the two values together to form the fitness for a geometry. The final geometries evolved by our ECs have

findings similar to what we found in round A's experiements. The number of exits evolved by the ECs in this round were between 4 and 6 instead of 4 and 7, like in round A's experiments. The design for the best geometry evolved by the PSO, found in figure 7.16, has 2 sets of double-sized doors, 2 meters wide, on opposite sides of the room. We believe this type of design in effective because it effectively divides the panicked individuals into half decreasing the possible pressure build up. The design for the best geometry evolved by the GA, found in figure 7.15, has a different design which consists of two different exits still, but on the same side of the room. This design consists of a 4 meter wide door in the center of one of the longest walls, and a 1 meter wide door a single door length away from the 4 meter door. We believe this design performed well because it has a small offset door that a small percentage of less panicked individuals can slip out of without having to engage the main group that all clamour to leave from the easily visible large 4 meter wide exit. This design performs better than the design evolved by the PSO. The geometry evolved by the GA most likely performs better because having an offset door that resides on the same wall as the primary exit means that panicked people do not have to change direction to find an alternate exit. In [17] we learn that one of the problems with panicked people is that they do not evaluate all of their options and everyone involved a situation that induces panic act together. The geometry evolved by the PSO does not handle this reality as well as the geometry evolved by the GA because its second exit is on an opposite wall.

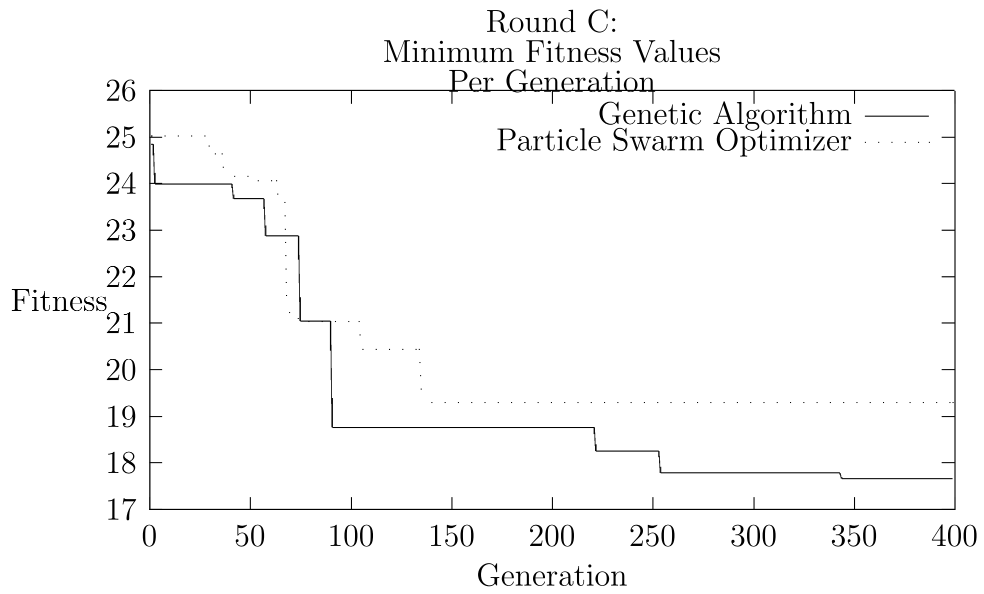

Figure 7.14: Round C Graphs: Fitness Curves for GA and PSO

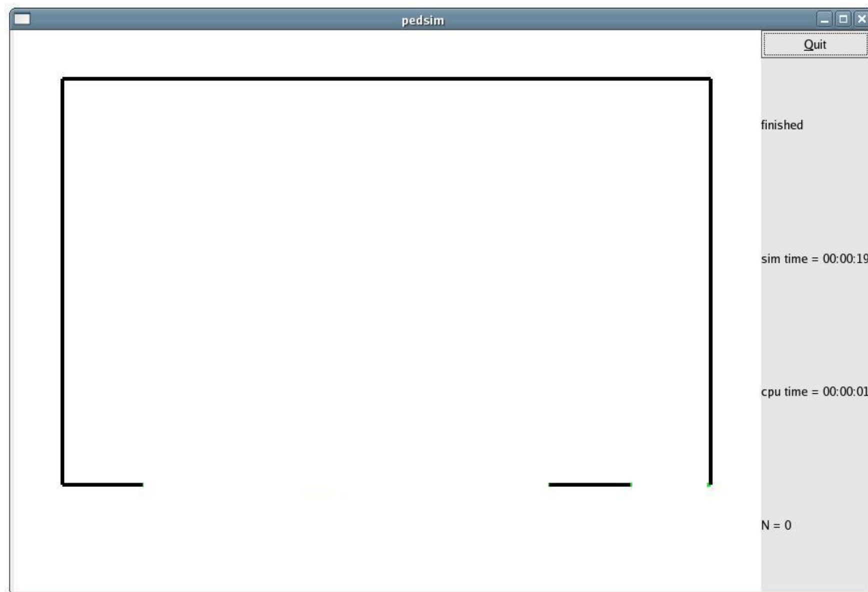

Figure 7.15: Round C – Best GA Geometry

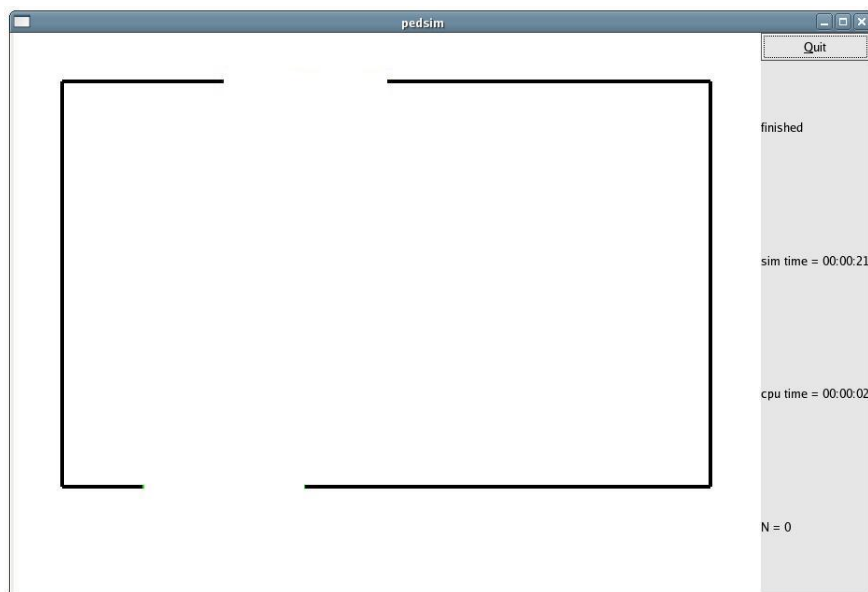

Figure 7.16: Round C – Best PSO Geometry

Chapter 8

Conclusions and Future Work

Evolutionary computations are not only useful in finding the most efficient placement of obstacles in a room, but may also be used to find the best positions and numbers of exits for rooms of varying geometries. For reactionary and evacuation purposes it is best for all rooms to be composed of a series of interlocking doors. The ECs used in this work were only tested on a single room geometry with no obstacles anywhere in the room. Now that there are ECs that can plan cities, rooms, and exit number and placement we can make more holistic plans for building and room planning. When combined with other work, like room planning, this method can be used to evolve the best placement of doors and obstacles to alleviate the amount ' of panic experienced when in a situation where a high volume of people need to exit a single space. With just finding the optimal placement and number of exits in a room we can apply this methodology to a great host of room geometries. Extra constraints can be introduced in the ECs so that the exit placement and number correctly corresponds to current guidelines for room construction.

One of the findings in experiment round B shows us that in order to evolve effective room geometries we need to optimize more than the escape time. We have already used simple aggregate functions to optimize both the exit count and escape time, but it may prove very interesting to also evolve the comfort experienced

by pedestrians when leaving an enclosure. In [17] one of the values of pedestrian simulation they recommend to evolve is the comfort experienced by panicked individuals. Comfort is measured by the number of times pedestrians must change their velocities when exiting an enclosure. In order to get better, more definitive room designs it would also be interesting to use a true multi-objective genetic algorithm (MOGA) [6] to evolve geometries using escape time, number of exits, and comfort.

Bibliography

[1] Arapoglu, R. A., Norman, B. A., Smith, A. E., "Locating input and output points in facilities design - a comparison of constructive, evolutionary, and exact methods." IEEE Transactions on Evolutionary Computation. 2001 Volume 5. pp. 192-203.

[2] Azarm, S., Reynolds, B.J., Narayanan, S. "Comparison of Two Multi-objective Optimization Techniques With and Within Genetic Algorithms", In CD-ROM Proceedings of the 25th ASME Design Automation Conference, volume Paper No. DETC99/DAC-8584, Las Vegas, Nevada, September 1999.

[3] Bierlaire, M., Antonini, G. and Weber M. "Behavioral dynamics for pedestrians", International Conference on Travel Behaviour Research. August 10, 2003.

[4] Carlisle, A., and Dozier, G. (2001). An off-the-shelf PSO. Proceedings of the Workshop on Particle Swarm Optimization. Indianapolis, IN: Purdue School of Engineering and Technology, IUPUI (in press).

[5] Clerc, M. "The Swarm and the Queen: Towards a Deterministic and Adaptive Particle Swarm Optimization". Proceedings, 1999 International Conference on Evolutionary Computation (ICEC), Washington, DC, pp 1951-1957.

[6] Coello Coello, C.A. A Comprehensive Survey of Evolutionary-Based Multiobjective Optimization Techniques, Knowledge and Information Systems, vol. 1, no. 3, pp. 269-308, Aug.1999.

[7] Dozier, G., Homaifar, A., Tunstel, E., and Battle, D., "An Introduction to Evolutionary Computation" (Chapter 17), Intelligent Control Systems Using Soft Computing Methodologies, A. Zilouchian and M. Jamshidi (Eds.), pp. 365-380, CRC press. (can be found at: <http://www.eng.auburn.edu/g̃vdozier/chapter17.doc>)

[8] Dozier, G. Introduction to Evolutionary Computation. <http://www.eng.auburn.edu/ gvdozier/Intro2EC.ppt>.

[9] Dozier, G. Genetic Algorithms. <http://www.eng.auburn.edu/ gvdozier/GAs.ppt>.

[10] Eaton, B.C., Eswaran, M. The Evolution of Competition. December 14, 2001.

[11] Garces-Perez, J., Schoenefeld, D.A., Wainwright, R.L., "Solving Facility Layout Problems Using Genetic Programming" (PS), Proceedings of the First Annual Conference Genetic Programming 1996, (GP-96), J. Koza, D. Goldberg, D. Fogel, R. Riolo, Editors, MIT Press, July 28-31, 1996, pp. 182-190

[12] "GNU General Public License". *The GNU Operating System.* Free Software Foundation, Inc. June 07, 2005. October 24, 2005. <http://www.gnu.org/licenses/gpl.html>.

[13] Helbing, D., and Molnar, P. (1998) "Social force model for pedestrian dynamics", Physical Review E 51, 4282-4286.

[14] Helbing, D., Farkas, I. and Vicsek, T. "Simulating Dynamical features of escape panic", Nature, 407, 487 - 490, doi:10.1038/35035023 (2000).

[15] Helbing, D., Molnar, P., Farkas, I.J., Bolay, K. "Self-organizing pedestrian movement", Environment and Planning B: Planning 2001, volume 28, pages 361-383.

[16] Helbing, D., Farkas, I.J., Vicsek, T., Molnar P., Bolay, K., Keltsch, P. http://pedsim.elte.hu .

[17] Helbing, D., Farkas, I.J., Molnar, P., Vicsek, T., (2002) Simulation of pedestrian crowds in normal and evacuation situations. Pages 21-58 in: M. Schreckenberg and S.D. Sharma (eds.) *Pedestrian and Evacuation Dynamics* (Springer, Berlin).

[18] "Leaving a Room". *Panic: A Quantitative Analysis.* Helbing, D., Farkas, I.J., Vicsek, T. November 06, 2005. <http://angel.elte.hu/panic/>

[19] Holland, J. H., Adaptation in natural and artificial systems, University of Michigan Press, Ann Arbor, 1975.

[20] Kennedy, J. and Eberhart, R. "Particle Swarm Optimization", Proceedings of the 1995 IEEE International Conference on Neural Networks, pp. 1942-1948, 1995.

[21] Kennedy, J. and Eberhart, R. "A Discrete Binary Version of the Particle Swarm Algorithm", IEEE International Conference on Systems, Man, and Cybernetics, 'Computational Cybernetics and Simulation'., 1997.

[22] Kennedy, J. "The Particle Swarm: Social Adaptation of Knowledge", IEEE International Conference on Evolutionary Computation., pp. 303-308, 1997.

[23] Kirchner, A., Schadschneider, A. "Simulation of evacuation processes using a bionics-inspired cellular automaton model for pedestrian dynamics". Physica A, 2002, Vol 312, Iss 1-2, pp 260-276.

[24] Kirkland, J.A., Maciejewski, A.A., A simulation of attempts to influence crowd dynamics. IEEE International Conference on Systems, Man and Cybernetics October 5, 2003. Volume: 5, pages: 4328-4333.

[25] McGrath, M., "Lions Creature Feature Fun Facts". *NationalGeographic.com Kids*. October 30, 2005. <http://www.nationalgeographic.com/kids/creature_feature/0109/lions2.html>.

[26] Cote, R., Harrington, G. E., National Fire Protection Association, Inc. Quincy, Massachusetts. 02169-7471.

[27] Silicon Graphics, "OpenGL: Developed by Silicon Graphics". *SGI United States*. 2005. October 23, 2005. <http://www.sgi.com/products/software/opengl/>

[28] "Qt Product Overview - single source C++ cross-platform application development for Windows, Linux, Mac". *Trolltech*. (2005) October 24, 2005. <http://www.trolltech.com/products/qt/index.html>

[29] Rabinovich, Y., Wigderson, A. "An Analysis of a Simple Genetic Algorithm", Proceedings of the 4th International Conference on Genetic Algorithms, pp. 215-221, Morgan Kaufmann, July 1991.

[30] Russell, S., Norvig, P. Artificial Intelligence: A Modern Approach, 2nd Edition. Prentice Hall Pearson Education, Inc., Upper Saddle River, New Jersey, 2003.

[31] Sadoun, B. "Simulation in city planning and engineering", Applied system simulation: methodologies and applications, pages 315-341, 2003.

[32] Saloma, C., Perez, G. J., Tapang, G., Lim, M. and Palmes-Saloma, C. "Self-organized queuing and scale-free behavior in real escape panic", Proceedings of the National Academy of Sciences USA published on-line, doi:10.1073/pnas.2031912100 (2003).

[33] Shaw, J. Multi-objective Genetic Algorithms for Schedule Operation. Report on the Manufacturing Complexity Network Meeting. International Manufacturing Centre Warwick Manufacturing Group, August 1999.

[34] Werner, T., "Pedestrian Simulation 0.1". *Homepage Torsten Werner*. August 6, 2003. October 23, 2005. <http://www.twerner42.de/ped/sim/>.

Appendices

Appendix A

Escape Panic Figures

The following images were taken from pedestrian simulation visualizations.

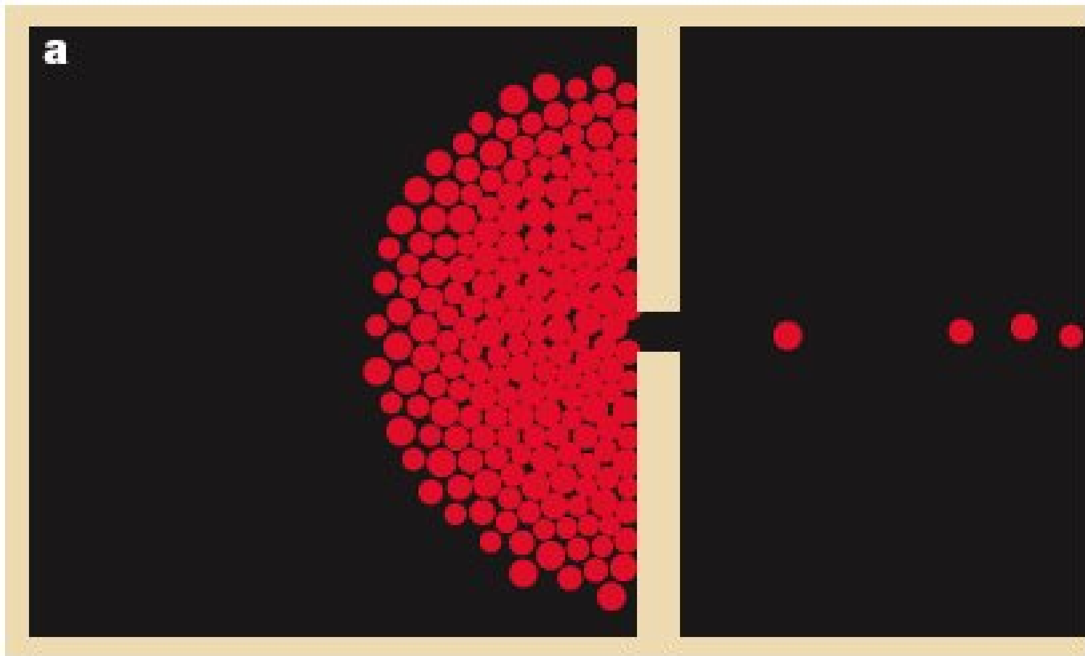

Figure A.1: Escape Panic Arc

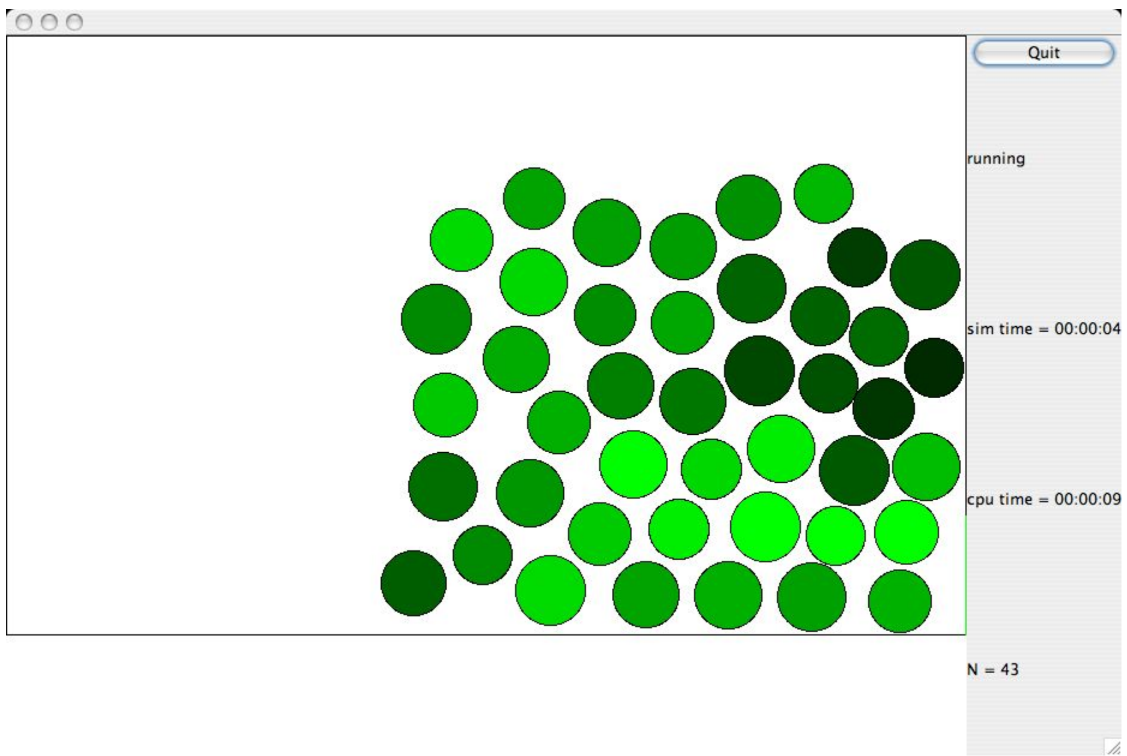

Figure A.2: Pedsim Pic 01

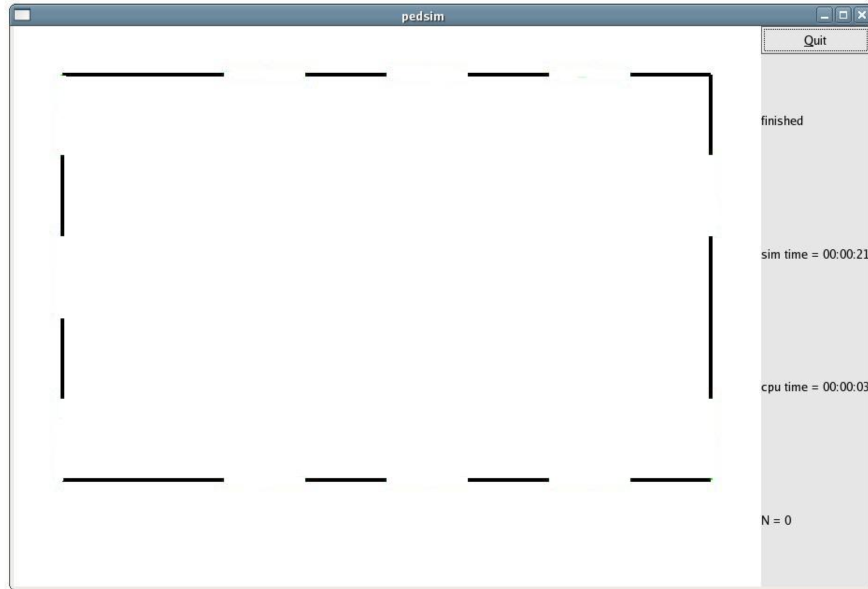

Figure A.3: Round B – Best GA Geometry with Exit Length of 1

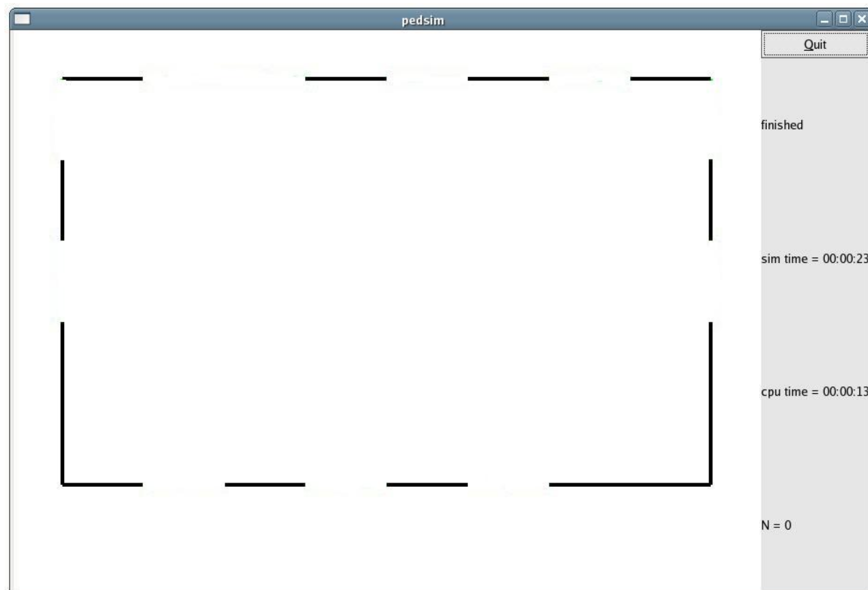

Figure A.4: Round B – Best GA Geometry with Exit Length of 2

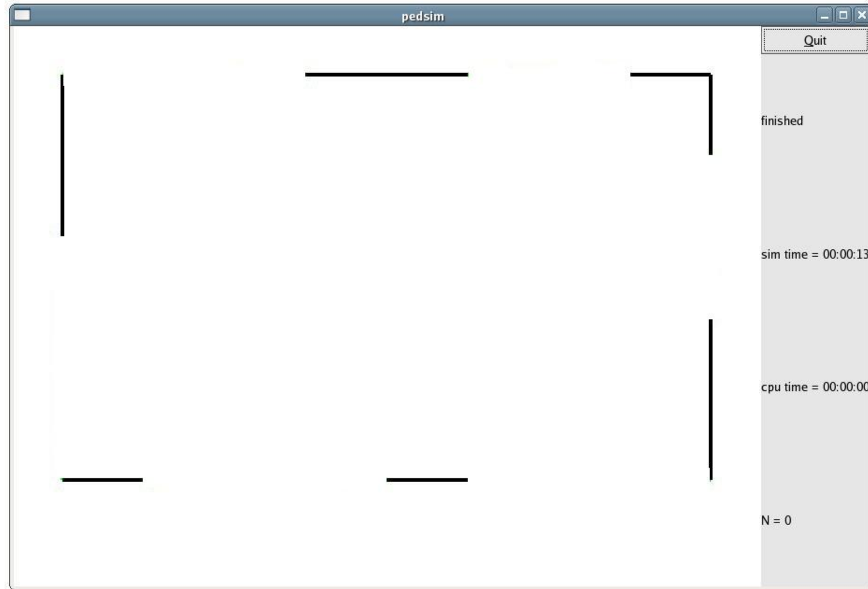

Figure A.5: Round B – Best GA Geometry with Exit Length of 3

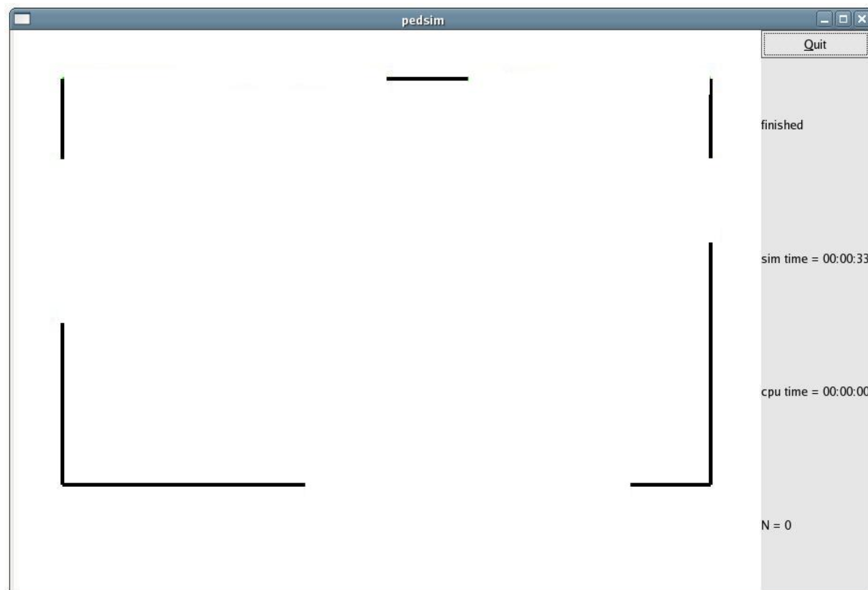

Figure A.6: Round B – Best GA Geometry with Exit Length of 4

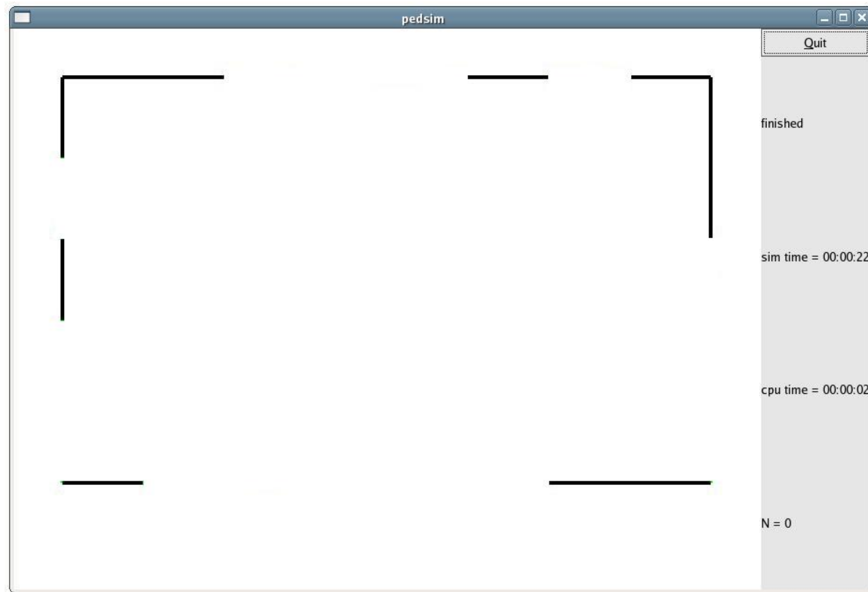

Figure A.7: Round B – Best GA Geometry with Exit Length of 5

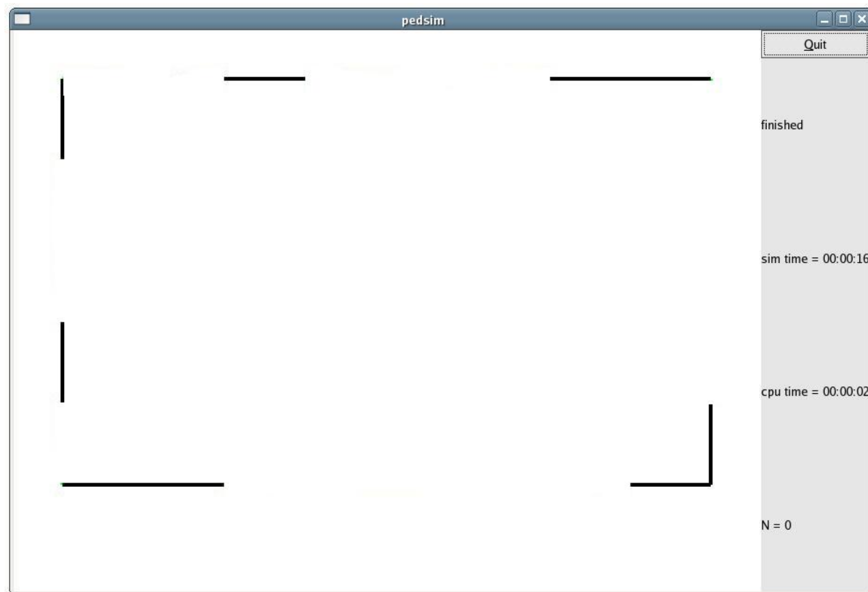

Figure A.8: Round B – Best GA Geometry with Exit Length of 6

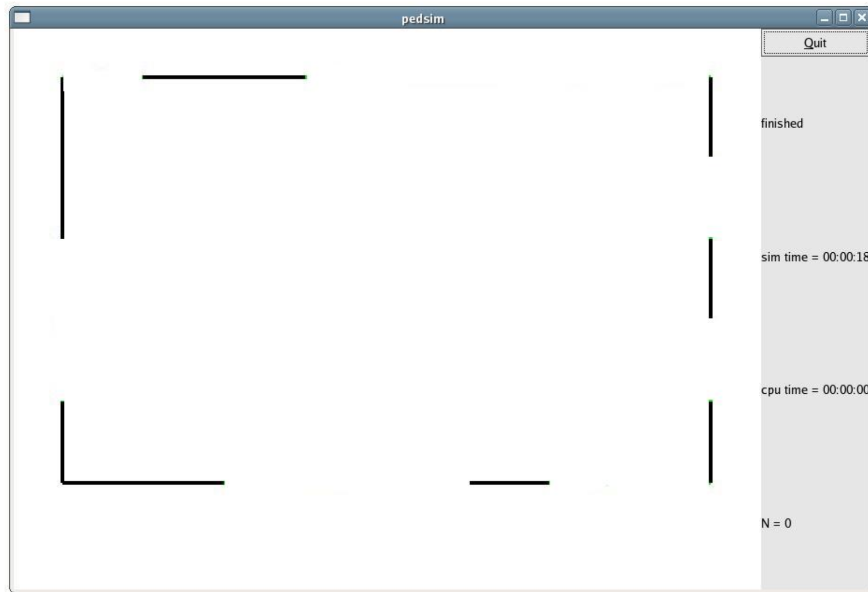

Figure A.9: Round B – Best GA Geometry with Exit Length of 7

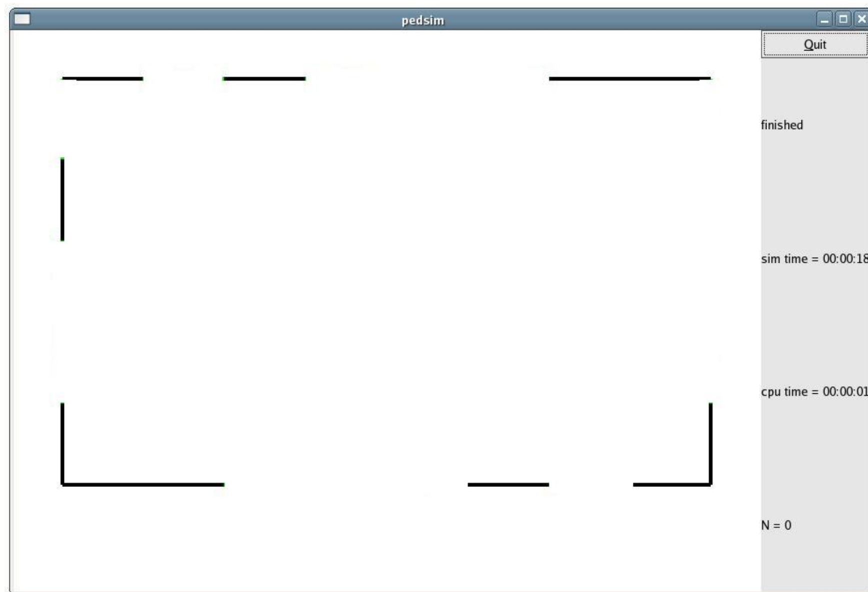

Figure A.10: Round B – Best GA Geometry with Exit Length of 8

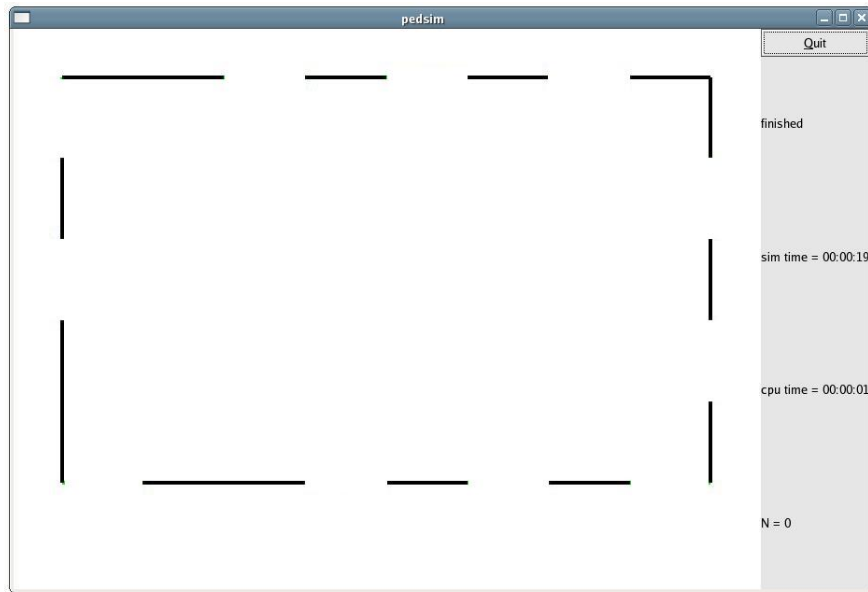

Figure A.11: Round B – Best PSO Geometry with Exit Length of 1

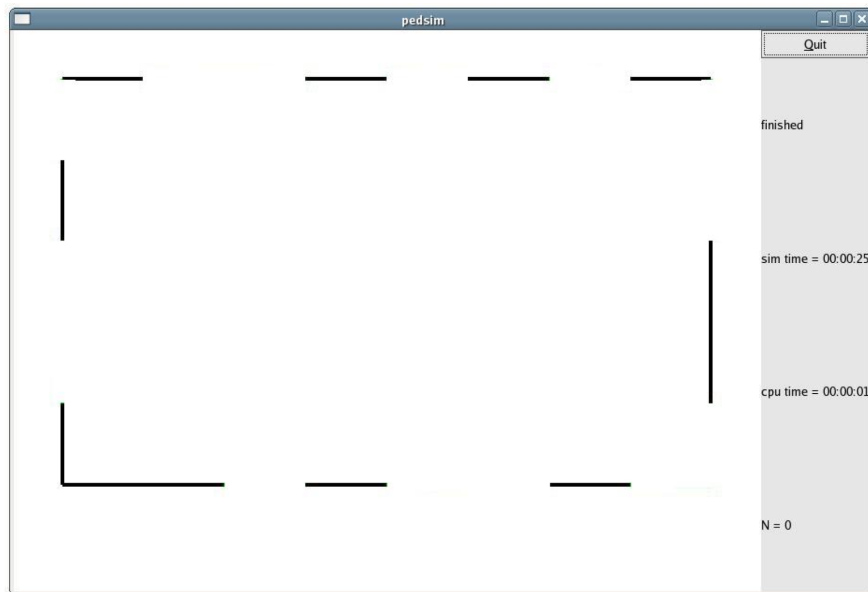

Figure A.12: Round B – Best PSO Geometry with Exit Length of 2

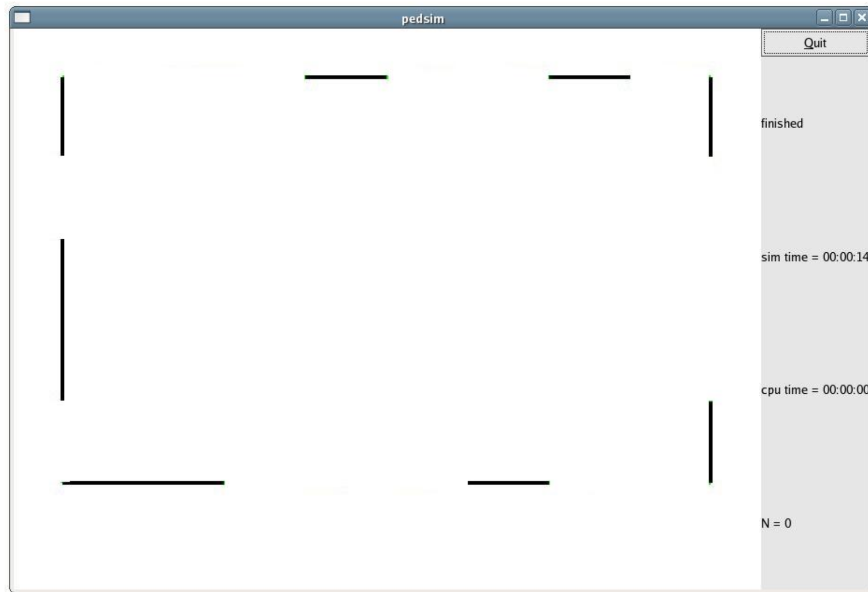

Figure A.13: Round B – Best PSO Geometry with Exit Length of 3

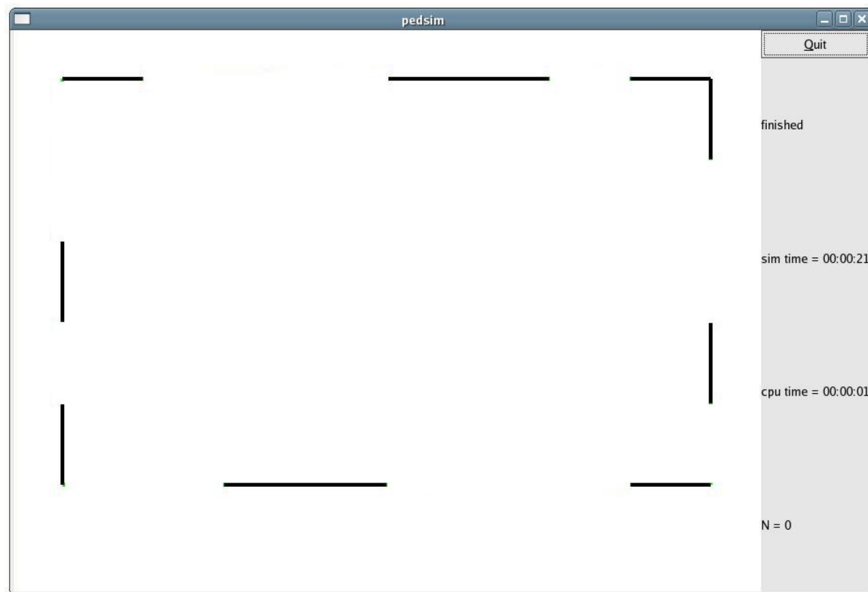

Figure A.14: Round B – Best PSO Geometry with Exit Length of 4

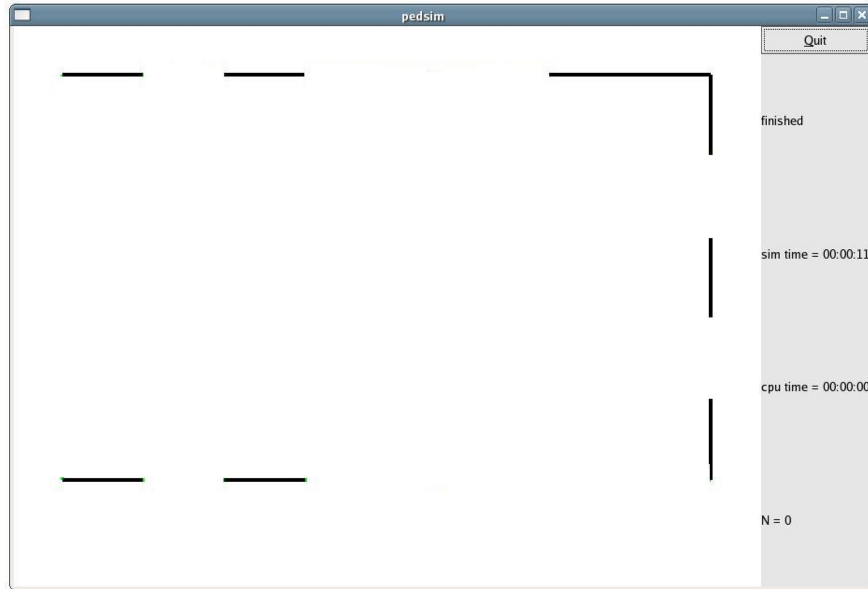

Figure A.15: Round B – Best PSO Geometry with Exit Length of 5

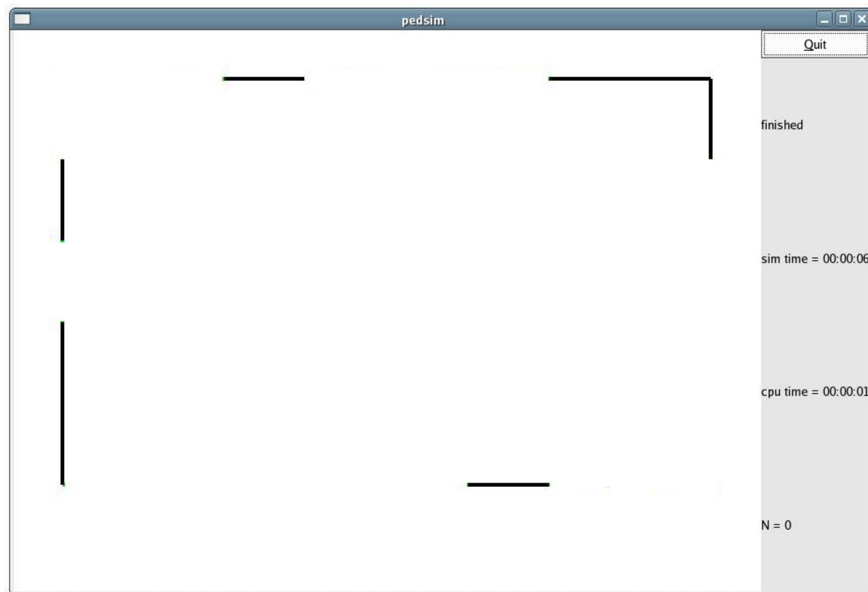

Figure A.16: Round B – Best PSO Geometry with Exit Length of 6

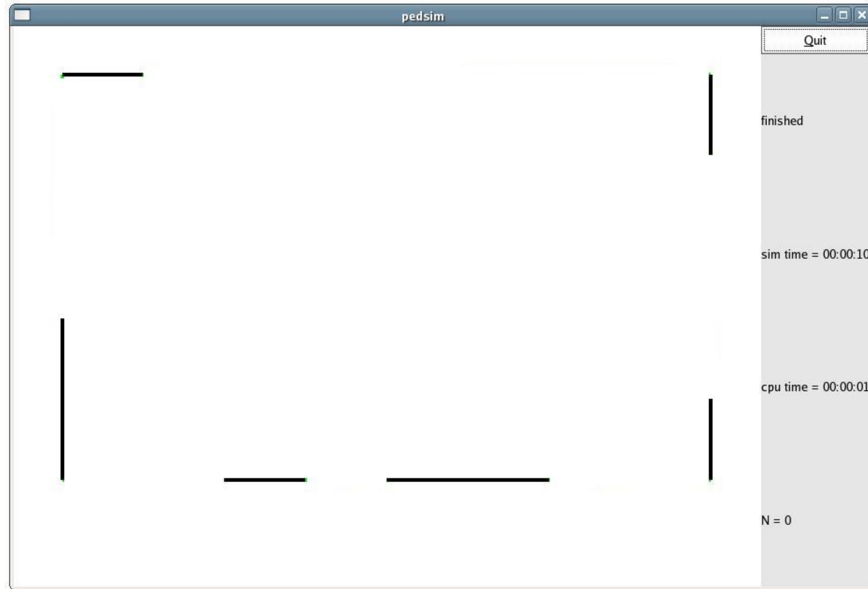

Figure A.17: Round B – Best PSO Geometry with Exit Length of 7

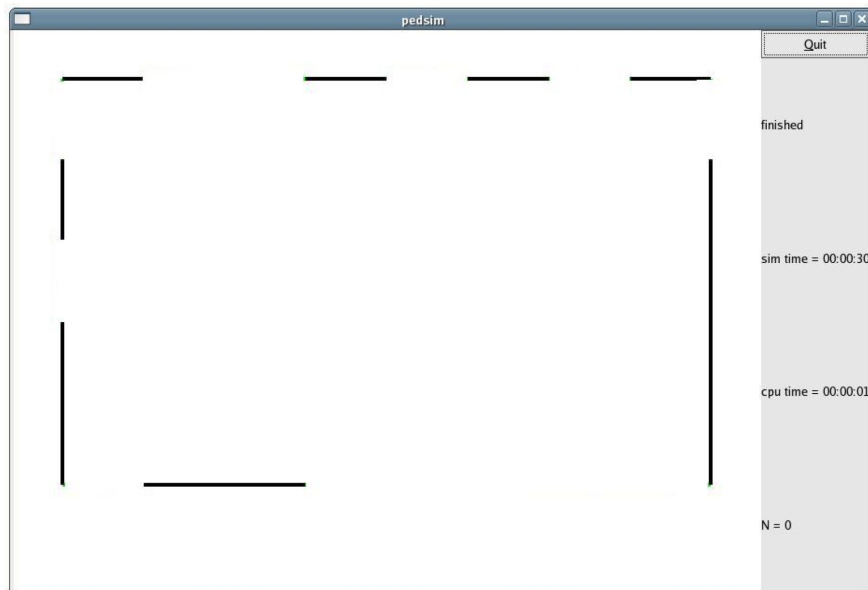

Figure A.18: Round B – Best PSO Geometry with Exit Length of 8

Appendix B

Source Code

Listing from header file `code/roombuilder.h`

Listing B.1: Multi-Page C Code for Roombuilder header

```
#ifndef ROOMBUILDER_H
#define ROOMBUILDER_H

#include "floor.h"
#include "obstacle.h"
#include "room.h"
#include "gate.h"
#include "transform.h"
#include "emitter.h"

#include "config.h"
#include <iostream>

class RoomBuilder
{
public:

// Constructor
  RoomBuilder( int _exitCount, double _xmax,
      double _ymax, double _doorWidth,
      double _lengthScale ) ;

// Deconstructor
  ~RoomBuilder() {}

// Begin Getters
  int getExitCount() { return exitCount ; }
  int getTotalNumExits()
  { return ( 2 * ( numHorizontalExits + numVerticalExits ) ) ; }
  double getXMAX() { return xmax ; }
  double getYMAX() { return ymax ; }
  double getDoorWidth() { return doorWidth ; }
  double getLengthScale() { return lengthScale ; }
// End Getters

// Begin Setters
```

```
  void setXMAX( double _xmax ) { xmax = _xmax ; }
  void setYMAX( double _ymax ) { ymax = _ymax ; }
  void setDoorWidth( double _doorWidth ) { doorWidth = _doorWidth ; }
  void setExitCount( int _exitCount ) { exitCount = _exitCount ; }
  // exitCount must be 1 or greater
  void setDP( bool * _doorPositions ) { doorPositions = _doorPositions ; }
  void setLengthScale( double _lengthScale ) { lengthScale = _lengthScale ; }
  void setMostVariables() ;
// End Setters

  void init( int _exitCount, double _xmax,
      double _ymax, double _doorWidth,
      double _lengthScale ) ;
  void geometry( int _N, Position _p ) ;
  void makeRoom( int _N, Position _p ) ;

//private:
  bool *doorPositions ;

  int exitCount ;
  int numHorizontalExits, numVerticalExits, index, index2, positionsIndex ;

  double xmax ;
  double ymax ;
  double doorWidth, lengthScale ;
  double exitSpanHorizontal, horizontalExitStart,
      horizontalExitEnd, horizontalExitEnd2 ;
  double exitSpanVertical, verticalExitStart,
      verticalExitEnd, verticalExitEnd2 ;
  double startHorizontal, startVertical ;
  double *horizontalExitStartPositions, *verticalExitStartPositions ;
  double *horizontalExitEndPositions, *verticalExitEndPositions ;
} ;

#endif
```

Listing from source file `code/roombuilder.cc`

Listing B.2: Multi-Page C Code for Roombuilder source

```
#include "floor.h"
#include "obstacle.h"
#include "room.h"
#include "config.h"
#include "gate.h"
#include "transform.h"
#include "emitter.h"
```

```
/* begin added */
#include "other.h"
#include "roombuilder.h"
#include <vector>
#include <stdio.h>
#include <iostream>
#include <iomanip>
/* end added */

RoomBuilder::RoomBuilder(int _exitCount, double _xmax,
    double _ymax, double _doorWidth, double _lengthScale)
{
  init( _exitCount, _xmax, _ymax, _doorWidth, _lengthScale) ;
    setMostVariables() ;
}

void RoomBuilder::init( int _exitCount, double _xmax,
    double _ymax, double _doorWidth, double _lengthScale )
{
    setExitCount( _exitCount ) ;
  setXMAX( _xmax ) ;
  setYMAX( _ymax ) ;
  setDoorWidth( _doorWidth ) ;
  setLengthScale( _lengthScale ) ;
}

void RoomBuilder::setMostVariables()
{
  setXMAX( getXMAX() / getLengthScale() ) ;
  setYMAX( getYMAX() / getLengthScale() ) ;
  setDoorWidth( getDoorWidth() / getLengthScale() ) ;

  numHorizontalExits = (int)floor( getXMAX() / getDoorWidth() ) ;
    numVerticalExits = (int)floor( getYMAX() / getDoorWidth() ) ;

    startHorizontal = numHorizontalExits * getDoorWidth() ;
    startVertical = numVerticalExits * getDoorWidth() ;

    horizontalExitStart = ( getXMAX() - startHorizontal ) * 0.66666667 ;
    verticalExitStart = ( getYMAX() - startVertical ) * 0.66666667 ;
    horizontalExitEnd = horizontalExitStart + getDoorWidth() ;
```

```
    verticalExitEnd = verticalExitStart + getDoorWidth() ;

    horizontalExitStartPositions = new double[numHorizontalExits] ;
    horizontalExitEndPositions = new double[numHorizontalExits] ;
    verticalExitStartPositions = new double[numVerticalExits] ;
    verticalExitEndPositions = new double[numVerticalExits] ;
}

void RoomBuilder::makeRoom( int N, Position p  )
{
  Floor& floor = FloorVector::create(1, 1) ;

    Room* room = floor.createRoom();

    Collection entireRoom ;

    bool doorPositions2[p.getLength()] ;

  // Get the variable room_rep and
  // make your door positions
    index = 0 ;
    while ( index < p.getLength() )
    {
        if ( room_rep.at(index) )
    {
          doorPositions2[index] = 1 ;
    }
      else
    {
          doorPositions2[index] = 0 ;
    }
      index++ ;
    }

    // This loops pushes all of the walls onto the collection
    index = 0 ;
    while ( index < p.getLength() )
    {
      if ( doorPositions2[index] ) {}
      else
      {
         if ( p.getType(index) == 'h' )
```

```
        {
          entireRoom.push_back
      ( room->newObstacle<HWall>
        ( p.getP(index),
        p.getBegin(index),
        p.getEnd(index),
        R ) ) ;
        }
        else
        {
          entireRoom.push_back
      ( room->newObstacle<VWall>
        ( p.getP(index),
        p.getBegin(index),
        p.getEnd(index),
        R ) ) ;
        }
    }
    index++ ;
}

// This loops pushes all of the exits onto the collection
index = index2 = 0 ;
while ( index < p.getLength() )
{
    if ( doorPositions2[index] )
    {
        if ( p.getType(index) == 'h' )
        {
      //if ( index % 2 != 0 )
      if ( p.getP(index) != 0 )
      {
        entireRoom.push_back
              ( room->newGate<HGate>
                ( p.getP(index),
              p.getBegin(index),
              p.getEnd(index),
              0 ) ) ;
      }
      else
      {
        entireRoom.push_back
```

```
                ( room->newGate<HGate>
                  ( p.getP(index),
                    p.getEnd(index),
                    p.getBegin(index),
                0 ) ) ;
        }
          }
          else
          {
        //if ( index % 2 != 0 )
        if ( p.getP(index) != 0 )
        {
          entireRoom.push_back
            ( room->newGate<VGate>
            ( p.getP(index),
              p.getBegin(index),
              p.getEnd(index),
              0 ) ) ;
        }
        else
        {
          entireRoom.push_back
            ( room->newGate<VGate>
            ( p.getP(index),
              p.getEnd(index),
              p.getBegin(index),
              0 ) ) ;
        }
          }
       }
       index++ ;
    }

    LatticeEmitter* emitter =
    new LatticeEmitter(0, 0, getXMAX(), getYMAX(), N);
    room->newSource<SingleSource>(emitter);
}
/**/

void RoomBuilder::geometry( int _N, Position p )
{
  int N = _N ;
```

```
  positionsIndex = 0 ;

// Set the positions of the horizontal exits and walls

    // Top Wall
    index = 0 ;
    while ( index < numHorizontalExits )
    {
    horizontalExitStartPositions[index] = horizontalExitStart ;
    horizontalExitEndPositions[index] =
      horizontalExitStartPositions[index] + doorWidth ;
    horizontalExitStart += doorWidth ;

    p.setGroupWithType( positionsIndex , 0,
        horizontalExitStartPositions[index],
        horizontalExitEndPositions[index], 'h' ) ;
    positionsIndex++ ;
    index++ ;
    }

    // Right Wall
    index = 0 ;
    while ( index < numVerticalExits )
    {
    verticalExitStartPositions[index] = verticalExitStart ;
    verticalExitEndPositions[index] =
      verticalExitStartPositions[index] + doorWidth ;
    verticalExitStart += doorWidth ;

        p.setGroupWithType( positionsIndex , xmax,
          verticalExitStartPositions[index],
          verticalExitEndPositions[index], 'v' ) ;
        positionsIndex++ ;

        index++ ;
    }

    // Bottom Wall
    index = numHorizontalExits - 1 ;
    while ( index > -1 )
    {
    p.setGroupWithType( positionsIndex , ymax,
```

```
                horizontalExitStartPositions[index],
                horizontalExitEndPositions[index], 'h' ) ;
    positionsIndex++ ;

    index-- ;
    }

    // Left Wall
    index = numVerticalExits - 1 ;
    while ( index > -1 )
    {
        p.setGroupWithType( positionsIndex , 0,
                    verticalExitStartPositions[index],
                    verticalExitEndPositions[index], 'v' ) ;
        positionsIndex++ ;

        index-- ;
    }

    makeRoom(N, p);
}
```

Listing from header file `code/other.h`

Listing B.3: Multi-Page C Code for Other header

```
#ifndef OTHER_H
#define OTHER_H

#include <cstring>



/**
 * A class that is made to pass room creation information
 */
class Position
{
public:
    // Constructor
  Position( int _length )
  {
     setLength( _length ) ;
     type = new char[ _length ] ;
```

```
    p = new double[ _length ] ;
    begin = new double[ _length ] ;
    end = new double[ _length ] ;
  }
  ~Position() {} // Deconstructor

  // Setters
   void setLength( int _length ) { length = _length ; }
   void setP(int index, double _p) { p[index] = _p ; }
   void setBegin( int index, double _begin ) { begin[index] = _begin ; }
   void setEnd( int index, double _end ) { end[index] = _end ; }
   void setType( int index, char _type ) { type[index] = _type ; }

   // Group Setters
   void setGroup( int index, double _p, double _begin, double _end )
   {
        p[index] = _p ;
        begin[index] = _begin ;
        end[index] = _end ;
     }
   void setGroupWithType( int index, double _p,
     double _begin, double _end, char _type )
   {
        p[index] = _p ;
        begin[index] = _begin ;
        end[index] = _end ;
        type[index] = _type ;
     }

  // Getters
  int getLength() { return length ; }
  double getP( int index ) { return p[index] ; }
  double getBegin( int index ) { return begin[index] ; }
  double getEnd( int index ) { return end[index] ; }
  char getType( int index ) { return type[index] ; }

private:
  int length ;
  char *type ;
  double *p ;
  double *begin ;
  double *end ;
```

```
} ;

#endif
```

Listing from header file `code/base.h`

Listing B.4: Multi-Page C Code for Base functions header

```
/****************************************
 * Created by Shelby Darnell
 * May 16, 2005
 *
 * base.hh
 * File contains generic includes and
 * useful functions
 *
 ****************************************/

#ifndef _BASE_H_
#define _BASE_H_

#include <cmath>
#include <cstdlib>
#include <ctime>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>
#include <string>
#include <vector>

#define UB  10
#define LB -10

using namespace std ;

typedef vector< int > VecInt ;

class Base
{
public:

// String manipulation methods
```

```
  static int charStringToInt( char* word ) ;
  static int stringToInt( string word ) ;

  static double charStringToDouble( char* word ) ;
  static double stringToDouble( string word ) ;

  static string doubleToString( double number ) ;
  static string intToString( int number ) ;

// Number manipulation methods
  static int convertToInt( double n, int length ) ;
  static int decodeBinary( int length , vector<int> binaryRep ) ;

  static double convertToDouble( int n, int length ) ;

  static vector<int> convertToBinary( int length , int p ) ;

// Random number methods
  static int randInt(int max) ;

  static double randDouble() ;

// Other methods
  static double clercs( double g1, double g2 ) ;
  static double gasdev(double mult) ;
  static double getCSFit( string cs ) ;
  static double sigmoid( double x ) ;

  static string getDirectory( string source ) ;
} ;

#endif //_BASE_H_
```

Listing from header file `code/base.cc`

Listing B.5: Multi-Page C Code for Base functions source

```
#include "base.h"

using namespace std ;

/**
 * String manipulation methods
```

```
*/

//-----------------------------------------------------------------

// Converts a character string into an integer
int Base::charStringToInt( char* word )
{
  int result ;
  stringstream temp ;

  temp << word ;
  temp >> result ;

  return result ;
}

//-----------------------------------------------------------------

// Converts a string into an integer
int Base::stringToInt( string word )
{
  int result ;
  stringstream temp ;

  temp << word ;
  temp >> result ;

  return result ;
}

//-----------------------------------------------------------------

// Converts a character string into a double
double Base::charStringToDouble( char* word )
{
  double result ;
  stringstream temp ;

  temp << word ;
  temp >> result ;

  return result ;
```

```
}

//------------------------------------------------------------

double Base::stringToDouble( string word )
{
  double result ;
  stringstream temp ;

  temp << word ;
  temp >> result ;

  return result ;
}

//------------------------------------------------------------

// Converts a double into a string
string Base::doubleToString( double number )
{
  string result ;
  stringstream temp ;

  temp << number ;
  temp >> result ;

  return result ;
}

//------------------------------------------------------------

// Converts an integer into a string
string Base::intToString( int number )
{
  string result ;
  stringstream temp ;

  temp << number ;
  temp >> result ;

  return result ;
}
```

```
/**
 * Number Manipulation Methods
 */

  //------------------------------------------------------------

  int Base::convertToInt ( double n, int length )
  {
      return ( int )( ( ( n - LB )
          * ( pow( 2.0, length ) - 1 ) ) / ( UB - LB ) );
  }

  //------------------------------------------------------------

  /**
   * Method works correctly.
   *
   * It take the length of a binary representation
   * and obtains its integer value.
   */
  int Base::decodeBinary( int length, VecInt binaryRep )
  {
    int up, down, phenotype ;

    phenotype = up = 0 ;
    down = length - 1 ;
    while ( up < length )
    {
      if ( binaryRep.at( up ) == 1 )
      {
        phenotype += (int) pow( 2.0, down ) ;
      }

      up++ ;
      down-- ;
    }
    return phenotype ;
  }

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

```
double Base::convertToDouble ( int n, int length )
{
    return ( ( ( UB - LB ) * n ) / ( pow( 2.0, length ) - 1 ) + LB );
}

//----------------------------------------------------------------

VecInt Base::convertToBinary( int length, int p )
{
  int index = length - 1 ;
  int subtractor ;
  VecInt reps ;

  while ( index > -1 )
  {
    subtractor = ( int )pow( (double)2, index ) ;
    if ( p >= subtractor )
    {
      p -= subtractor ;
      reps.push_back( 1 ) ;
    }
    else
    {
      reps.push_back( 0 ) ;
    }
    index-- ;
  }
  return reps ;
}

/**
 * Random Number Methods
 */
//----------------------------------------------------------------

// Function to derive a random integer
int Base::randInt(int max)
{
    return (int)((double)max * rand() / ( RAND_MAX + 1.0 )) ;
}

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

```
// Function to derive a random doubleing point number
double Base::randDouble()
{
    return (double) (rand()/(RAND_MAX+1.0)) ;
}

/**
 * Other methods
 */

//-----------------------------------------------------------------

/**
 *  Function to help move a particle in PSO
 *     The inputs that the function takes are phi 1
 *     and phi 2 which denote social and cognitive
 *     awareness.
 */
double Base::clercs( double g1, double g2 )
{
   double k, gam, gam4, gsr, gabs;

   gam = g1 + g2;
   gam4 = gam * 4;
   gsr = sqrt( pow( gam, 2 ) );
   gabs = fabs( 2 - gam - gsr - gam4 );
   k = 2 / gabs;

   return k;
}

//-----------------------------------------------------------------

double Base::gasdev(double mult)
{
    static int iset=0;
    static double gset;
    double fac, rsq, v1, v2;//, idum;


    if ( iset == 0 )
```

```
    {                          // We don't have an extra deviate handy so,
        do {
            v1=2.0 * randDouble( ) - 1.0;
      // pick two uniform #'s in the square
            v2=2.0 * randDouble( ) - 1.0;
      // extending from -1 to +1 in each
            rsq = ( v1*v1 ) + ( v2*v2 );
      // direction, see if they are in the unit circle,
        } while( rsq >= 1.0 || rsq == 0.0 );
    // and if they are not, try again.

    fac=sqrt( -2.0*log( rsq )/rsq );
        // Now make the Box-Muller transformation to
        // get 2 normal deviates.
        // Return one and save the other for next time.
        gset=v1*fac;                           // Set flag.
        iset=1;
        return v2*fac*mult;
    }
    else
    {
                                 // We have an extra deviate handy.
        iset=0;                              // so unset the flag.
        return gset*mult;                         // and return it.
    }
}

//------------------------------------------------------------

/**
 * Runs pedsim to get the fitness of the
 * candidate solution
 */
double Base::getCSFit( string cs )
{
  string configFile, pedsimExecute, fitnessFile ;

  configFile = getDirectory( "pedsim" ) + "config2.in" ;
  //pedsimExecute = getDirectory( "pedsim" ) + "pedsim -n" ;
  pedsimExecute = getDirectory( "pedsim" ) + "user-base.pl" ;
  fitnessFile = getDirectory( "pedsim" ) + "fit.in" ;
```

```
  // Write the candidate solution to file so
  // that pedsim can read it
  double fitness ;
  ofstream out( configFile.c_str() ) ;
  out << cs << endl ;
  out.close() ;

  system( pedsimExecute.c_str() ) ;
  //cout << "Pedsim execute " << pedsimExecute << endl ;

  // Read the candidate solution fitness from the
  // file written by pedsim
  ifstream in( fitnessFile.c_str() ) ;
  in >> fitness ;
  if ( fitness == 0 ) { fitness = 100 ; }
  in.close() ;

  return fitness ;
}

//------------------------------------------------------------

double Base::sigmoid( double x )
{
  double result, temp ;
  temp = 1 - exp( -x ) ;
  result = 1 / temp ;
  return result ;
}

//------------------------------------------------------------

string Base::getDirectory( string source )
{
  string result ;
  if ( source == "pedsim" )
  {
    ifstream inFile( "data/pedsimDirectory.dat", ios::in ) ;
    inFile >> result ;
  }
  else
  {
```

```
    ifstream inFile( "data/particle_swarm.dat", ios::in ) ;
    inFile >> result ;
  }
  return result ;
}

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

Listing from header file `code/randpack.h`

Listing B.6: Multi-Page C Code for Dozier's RandPack header

```
#ifndef _RANDPACK_H_
#define _RANDPACK_H_

#include "base.h"

class RandPack
{
public:
  int seed ;

    void myRandomize() ;
    void seed_myRandomize( int ) ;

    int myRandomInt( int ) ;

    double myRandom() ;
    double myRandRange( double, double ) ;

} ;

#endif //_RANDPACK_H_
```

Listing from header file `code/randpack.cc`

Listing B.7: Multi-Page C Code for Dozier's RandPack source

```
#include "randpack.h"

using namespace std ;

//RandPack::RandPack() {}
```

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

//RandPack::~RandPack() {}

//------------------------------------------------------------

void RandPack::myRandomize()
{
  seed = rand() ; //time(NULL);
    myRandom();
}

//------------------------------------------------------------

void RandPack::seed_myRandomize( int _seed )
{
  seed = _seed;
  myRandom();
}

//------------------------------------------------------------

int RandPack::myRandomInt( int modulus )
{
  return (int) ( myRandom() * ( modulus-0.00000001 ) );
}

//------------------------------------------------------------

double RandPack::myRandom()
{
  seed = rand() ;
  int
    a = 16807,
        m = 2147483647,
        q = 127773, /* m div a */
        r = 2836,   /* m mod a */

        lo, hi, test;

  hi = seed / q;
```

```
  lo = seed % q;

  test = a * lo - r * hi;

  if (test > 0)
    seed = test;
  else
    seed = test + m;

  return (double) seed/m;
}

//---------------------------------------------------------------

double RandPack::myRandRange( double v1, double v2 )
{
  double temp;

    if (v2 < v1)
    {
        temp = v2;
        v2 = v1;
        v1 = temp;
    }

    return ((v2-v1) * myRandom() + v1);
}

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

Listing from header file `code/individual.h`

Listing B.8: Multi-Page C Code for the GA's Individual or Candidate Solution header

```
#ifndef _INDIVIDUAL_H_
#define _INDIVIDUAL_H_

#include "base.h"
#include "randpack.h"

typedef vector< int > VecInt ;
using namespace std ;
```

```
/**
 * 'rep' stands for 'representation'
 */

class Individual
{
public:
  Individual() {}
  Individual( int, int ) ;
  ~Individual() ;

  void init( int, int ) ;
  void initBinaryRep() ;
  void flipMember( int ) ;
  void mutate() ;
  void fixExitLength() ;

  bool equals( VecInt ) ;

  int repAt( int index ) { return rep.at( index ) ; }

  string repToString() ;
  string toString() ;

// Setters
  void calculateOnes() ;
  void setFitness() ;
  void setOnes( int _ones ) { ones = _ones ; }
  void setRep( VecInt, int ) ;
  void setRepSize( int _repSize ) { repSize = _repSize ; }
  void setTime( double _simTime ) { simTime = _simTime ; }
  void setMaxExitLength( int _maxExitLength )
  { maxExitLength = _maxExitLength ; }

// Getters
  int getOnes() { return ones ; }
  int getRepSize() { return repSize ; }
  int getMaxExitLength() { return maxExitLength ; }
  //int checkOddExits() ;
  //int checkEvenExits() ;
  double getFitness() { return fitness ; }
```

```
  double getTime() { return simTime ; }
  VecInt getRep() { return rep ; }

private:
  int ones,
      repSize,
      maxExitLength ;

  double simTime, fitness ;

  RandPack rnd ;

  vector< int > rep ;


} ;

#endif  //_INDIVIDUAL_H_
```

Listing from header file `code/individual.cc`

Listing B.9: Multi-Page C Code for the GA's Candidate Solution source

```
#include "base.h"
#include "individual.h"

#define THRESHOLD 30

using namespace std ;

//------------------------------------------------------------

Individual::Individual( int _repSize, int _maxExitLength )
{
  init( _repSize, _maxExitLength ) ;
}

//------------------------------------------------------------

Individual::~Individual()
{
  rep.clear() ;
}
```

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

void Individual::init( int _repSize , int _maxExitLength )
{
  rnd.myRandomize() ;

  setRepSize( _repSize ) ;
  setMaxExitLength( _maxExitLength ) ;
  initBinaryRep() ;
  fixExitLength() ;
  setFitness() ;
}

//--------------------------------------------------------------

void Individual::initBinaryRep()
{
  rep.clear() ;

  int index = 0 ;
  while ( index < getRepSize() )
  {
    if ( rnd.myRandomInt( 2 ) < 1 )
    {
      rep.push_back( 1 ) ;
    }
    else
    {
      rep.push_back( 0 ) ;
    }
    index++ ;
  }
}

//--------------------------------------------------------------

void Individual::flipMember( int index )
{
  if ( rep.at(index) == 0 )
  {
    rep[index] = 1 ;
```

```
    }
    else
    {
      rep[index] = 0 ;
    }
}

//-------------------------------------------------------------------

bool Individual::equals( vector<int> repB )
{
  bool result = true ;
  int index = 0 ;
  while ( index < getRepSize() )
  {
    result = ( rep.at(index) == repB.at(index) ) ? true : false ;
    // Exit the loop if things don't match
    if ( result == false )
    {
      index = getRepSize() ;
    }
    index++ ;
  }
  return result ;
}

//-------------------------------------------------------------------
/**
 * The string representation of the room geometry
 * must have spaces in between each wall or exit
 * representing integer, because the roombuilder
 * reads the file uses spaces.
 */
string Individual::repToString()
{
  int index = 0 ;
  string result = "" ;

  while ( index < getRepSize() )
  {
    result += Base::intToString( rep.at(index) ) + " " ;
    index++ ;
```

```
  }

  return result ;
}

//-----------------------------------------------

string Individual::toString()
{
  return ( repToString()
      + "," + Base::doubleToString( getFitness() )
      + "," + Base::doubleToString( getTime() )
      + "," + Base::intToString( getOnes() ) ) ;
}

//-----------------------------------------------

void Individual::setFitness()
{
  double fitnessSum = 0 ;
  calculateOnes() ;
  for( int i = 0 ; i < THRESHOLD ; i++ )
  {
    do
    {
      fitness = Base::getCSFit( repToString() ) ;
    } while ( fitness < 0 ) ;
    fitnessSum += fitness ;
  }
  setTime( fitnessSum / THRESHOLD ) ;
  fitness = getTime() + getOnes() ;
}

//-----------------------------------------------
/// Calculate the number of exits in the
/// geometry.
void Individual::calculateOnes()
{
  int index, sum ;

  index = sum = 0 ;
  while ( index < getRepSize() )
```

```
  {
    if ( rep.at( index ) == 1 )
    {
      sum++ ;
    }
    index++ ;
  }
  setOnes( sum ) ;
}

//-----------------------------------------------------------

void Individual::setRep( VecInt _rep , int _maxExitLength )
{
  setRepSize( _rep.size() ) ;
  setMaxExitLength( _maxExitLength ) ;
  rep = _rep ;
  fixExitLength() ;
  calculateOnes() ;
}

//-----------------------------------------------------------
/// This function goes to each integer in the
/// binary string and will flip it with a 50%
/// probability.
void Individual::mutate()
{
  int index = 0 ;
  while ( index < getRepSize() )
  {
    if ( rnd.myRandom() <= 0.5 )
    {
      flipMember( index ) ;
    }
    index++ ;
  }
}

//-----------------------------------------------------------
/*
 *
 *
```

```
 */
void Individual::fixExitLength()
{
  int j = 0,
      count = 0,
    specialCount = 0 ;
  bool flag = false ;
  while( j < getRepSize() )
  {
    if ( repAt(j) == 1 )
    {
      count++ ;
      if ( j == 0 )
      {
        flag = true ;
      }
      if ( j == getRepSize() - 1 )
      {
        specialCount += count ;
        if ( specialCount == (getMaxExitLength()+1) )
        {
          //cout << "Got him " << repToString() << endl ;
          flipMember(j) ;
          //cout << "New him " << repToString() << endl ;
        }
      }

      if ( count == (getMaxExitLength()+1) )
      {
        if ( flag == true )
        {
          specialCount = count ;
        }
        flipMember(j) ;
        count = 0 ;
        flag = false ;
      }
    }
    else
    {
      if ( flag == true )
      {
```

```
            specialCount = count ;
        }
        count = 0 ;
        flag = false ;
    }

    j++ ;
  }
}

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

Listing from header file `code/population.h`

Listing B.10: Multi-Page C Code for the GA's Population header

```
#ifndef _POPULATION_H_
#define _POPULATION_H_

#include "individual.h"

typedef vector< Individual > VecInd ;
typedef vector< int > VecInt ;

class Population
{
public:
  //---------------------------------------------
  // Empty constructor
  Population() {}

  //---------------------------------------------
  // Empty deconstructor
  ~Population() ;

  Population( int, int, int, int ) ;

  void init( int, int, int, int ) ;

  int procreate() ;
  int fitnessLevel( string ) ;
  int selectParent() ;
```

```
  double getAverageFitness() ;
  double getFitness( int ) ;

  string toString( int ) ;

//============================================
// Setters
  void setPopSize( int _popSize ) { popSize = _popSize ; }
  void setRepSize( int _repSize ) { repSize = _repSize ; }
  void setMutationRate( int _mutationRate )
  { mutationRate = _mutationRate ; }
  void setMaxExitLength( int _maxExitLength )
  { maxExitLength = _maxExitLength ; }

//============================================
// Getters

  int getPopSize() { return popSize ; }
  int getRepSize() { return repSize ; }
  int getMutationRate() { return mutationRate ; }
  int getMaxExitLength() { return maxExitLength ; }



private:
  int popSize,
      repSize,
      mutationRate,
      maxExitLength ;

  double fitness ;
  // Average fitness of all individuals in the population

  RandPack rnd ;

  VecInd pop ;
} ;

#endif  //_POPULATION_H_
```

Listing from header file `code/population.cc`

Listing B.11: Multi-Page C Code for the GA's Population source

```
#include "population.h"

using namespace std ;

Population::Population( int _popSize , int _repSize ,
    int _maxExitLength , int _mutRate )
{
  init( _popSize , _repSize , _maxExitLength , _mutRate ) ;
}

//--------------------------------------------------------------

Population::~Population()
{
  pop.clear() ;
}

//--------------------------------------------------------------

void Population::init( int _popSize , int _repSize ,
    int _maxExitLength , int _mutRate )
{
  pop.clear() ;
  rnd.myRandomize() ;
  setPopSize( _popSize ) ;
  setRepSize( _repSize ) ;
  setMaxExitLength( _maxExitLength ) ;
  setMutationRate( _mutRate ) ;

  int index = 0 ;
  while ( index < getPopSize() )
  {
    pop.push_back( Individual(getRepSize(), getMaxExitLength()) ) ;
    index++ ;
  }
}

//--------------------------------------------------------------
/// This mutation operator takes as an input
/// an integer which will determine the amount of
/// mutation by dividing it by 1000.
```

```
///
/// So sending the number 5, gives a 0.5 % mutation
/// rate.

int Population::procreate()
{
  int mom,
    dad,
    index,
    replaceMe,
    count,
    fitA,   // temp fit value in int format to help test for fitness
    fitB ;  // temp fit value in int format to help test for fitness
  // Boolean variables used to simplify the code
  bool conditionA = false,
    conditionB = false,
    conditionC = false ;
  VecInt repA, repB ;
  Individual childA, childB ;

  dad = selectParent() ;
  do
  {
    mom = selectParent() ;
  } while ( dad == mom ) ;

  // Randomly selects the gene of either
  // parent for the new child
  index = count = 0 ;
  while ( index < getRepSize() )
  {
    if ( rnd.myRandomInt( 2 ) == 1 )
    {
      repA.push_back( pop.at(dad).repAt(index) ) ;
      repB.push_back( pop.at(mom).repAt(index) ) ;
    }
    else
    {
      repA.push_back( pop.at(mom).repAt(index) ) ;
      repB.push_back( pop.at(dad).repAt(index) ) ;
    }
```

```
    index++ ;

  }

  childA.setRep( repA, getMaxExitLength() ) ;

  // Mutation
  if ( rnd.myRandomInt( 1000 ) <= getMutationRate() )
  {
    childA.mutate() ;
  }

  childA.fixExitLength() ;
  childA.setFitness() ;
  count++ ;
  replaceMe = fitnessLevel( "worst" ) ;
  fitA = int( childA.getFitness() + 0.5 ) ;
  fitB = int( pop.at(replaceMe).getFitness() + 0.5 ) ;
  conditionA = childA.getFitness() < pop.at(replaceMe).getFitness() ;
  conditionB = ( fitA == fitB ) ;
  conditionC = ( childA.getOnes() < pop.at(replaceMe).getOnes() ) ;

  if ( conditionA || ( conditionB && conditionC ) )
  {
    pop[replaceMe] = childA ;
  }

  return count ;

}

//--------------------------------------------------------------
/// Fitness level is based on minimization.
///
/// Method has been successfully tested

int Population::fitnessLevel( string level )
{
  int index, result ;
  double current, best, worst ;

  index = result = 0 ;
```

```
  best = pop.at( index ).getFitness() ;
  worst = pop.at( index ).getFitness() ;
  while ( index < getPopSize() )
  {
    current = pop.at( index ).getFitness() ;
    if ( level == "best" )
    {
      if ( current < best )
      {
        best = current ;
        result = index ;
      }
    }
    else
    {
      if ( current > worst )
      {
        worst = current ;
        result = index ;
      }
    }

    index++ ;
  }
  return result ;
}

//------------------------------------------------------------
/// Tournament Selection
/// Pick the best of 2

int Population::selectParent()
{
  int pA, pB, result ;

  pA = rnd.myRandomInt( getPopSize() ) ;
  do
  {
    pB = rnd.myRandomInt( getPopSize() ) ;
  } while ( pA == pB ) ;

  if ( pop.at(pA).getFitness()
```

```
      < pop.at(pB).getFitness() )
  {
    result = pA ;
  }
  else
  {
    result = pB ;
  }
  return result ;
}

//-----------------------------------------------------------

double Population::getAverageFitness()
{
  int index = 0 ;
  double sum = 0.0 ;
  while ( index < getPopSize() )
  {
    sum += pop.at(index).getFitness() ;
    index++ ;
  }
  return ( sum/getPopSize() ) ;
}

//-----------------------------------------------------------

double Population::getFitness( int index )
{
  return pop.at(index).getFitness() ;
}

//-----------------------------------------------------------

string Population::toString( int runNumber )
{
  string result = "" ;
  int index = 0 ;
  while ( index < getPopSize() )
  {
    result += Base::intToString( runNumber )
      + "," + Base::intToString(index+1)
```

```
      //+ "," + Base::intToString( getPopSize() )
      //+ "," + Base::intToString( getMutationRate() )
      + "," + pop.at(index).toString()
      + "\n" ;
    index++ ;
  }
  return result ;
}

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

Listing from source file `code/ga.cc`

Listing B.12: Multi-Page C Code that Runs the GA source

```
#include "base.h"
#include "population.h"

using namespace std ;

typedef vector< int > VecInt ;

int main( int argc , char ** argv )
{
  srand( time( NULL ) ) ;

  int popSize ,
      index ,
      repSize ,
      mutRate ,
      runs ,
    // function evaluation ,
    // just about akin to generations
    // for a steady-state GA
      count ,
      maxExitLength ,
      temp ;

  Population pop ;

  popSize = Base::charStringToInt( argv[1] ) ;
  repSize = Base::charStringToInt( argv[2] ) ;
```

```
  mutRate = Base::charStringToInt( argv[3] ) ;
  runs = Base::charStringToInt( argv[4] ) ;
  maxExitLength = Base::charStringToInt( argv[5] ) ;

  pop.init( popSize , repSize , maxExitLength , mutRate ) ;
  count = popSize ;

  index = 0 ;
  cout << "Runs,Individual" ;
  cout << ",Representation,Fitness" ;
  cout << ",EscapeTime,Exits" ;
  cout << endl ;
  while ( count < runs )
  {
    cout << pop.toString( index + 1 ) ;
    cout << endl ;
    count += pop.procreate() ;
    index++ ;
  }
  return 0 ;
}
```

Listing from header file `code/particle.h`

Listing B.13: Multi-Page C Code for the PSO's Particle header

```
#ifndef _PARTICLE_H_
#define _PARTICLE_H_

#include "base.h"
#include "randpack.h"

#define VMAX 6

using namespace std ;

class Particle
{
public:
  Particle() {}
  Particle( int , int ) ;
  ~Particle() ;
```

```
  void init( int, int ) ;
  void flipMember( int ) ;
  void calculateOnes() ;
  void fixExitLength() ;
  void initBinaryRep() ;
  void initVs() ;

  // PhiA, PhiB, P vector of best particle
  void move ( double, double, vector< int > ) ;

  int repAt( int index ) { return x.at( index ) ; }

  string repXToString() ;
  string repPToString() ;
  string vsToString() ;
  string toString() ;

// Setters
  void setMaxExitLength( int _maxExitLength )
  { maxExitLength = _maxExitLength ; }
  void setRepSize( int _repSize ) { repSize = _repSize ; }
  void setTime( double _simTime ) { simTime = _simTime ; }
  void setOnes( int _exits ) { exits = _exits ; }
  void setFitness() ;
  void setP()
  { p = x ; pFitness = fitness ; pExits = exits ; pSimTime = simTime ; }

// Getters
  int getMaxExitLength() { return maxExitLength ; }
  int getRepSize() { return repSize ; }
  int getOnes() { return exits ; }
  double getTime() { return simTime ; }
  double getFitness() { return fitness ; }
  vector< int > getX() { return x ; }

private:
  int maxExitLength,
    neighborA,
    neighborB,
    repSize,
    exits,
      pExits  ;
```

```
// Fitness
  double fitness ,
         pFitness ,
       pSimTime ,
         simTime ;

// Swarm Vectors
  vector< int > x ;
  vector< int > p ;
  vector< double > v ;

  RandPack rnd ;
} ;

#endif // _PARTICLE_H_
```

Listing from header file `code/particle.cc`

Listing B.14: Multi-Page C Code for the PSO's Particle source

```
#include "particle.h"
#include "base.h"

#define THRESHOLD 30

using namespace std ;

//————————————————————————————————————————

Particle::~Particle()
{
  x.clear() ;
  v.clear() ;
  p.clear() ;
}

//————————————————————————————————————————

Particle::Particle( int _repSize , int _maxExitLength )
{
  init( _repSize , _maxExitLength ) ;
}
```

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

void Particle::init( int _repSize, int _maxExitLength )
{
  x.clear() ;
  v.clear() ;
  p.clear() ;

  rnd.myRandomize() ;

  setMaxExitLength( _maxExitLength ) ;
  setRepSize( _repSize ) ;
  initBinaryRep() ;
  initVs() ;
  fixExitLength() ;
  setFitness() ;
  setP() ;
}

//------------------------------------------------------------

void Particle::initBinaryRep()
{
  int index = 0 ;
  while ( index < getRepSize() )
  {
    if ( rnd.myRandom() < 0.5 )
    {
      x.push_back( 1 ) ;
    }
    else
    {
      x.push_back( 0 ) ;
    }
    index++ ;
  }
}

//------------------------------------------------------------

void Particle::initVs()
```

```
{
  int index = 0 ;
  while ( index < getRepSize() )
  {
    v.push_back( rnd.myRandom() ) ;
    if ( rnd.myRandom() < 0.5 )
    {
      v[index] *= -1 ;
    }
    index++ ;
  }
}

//--------------------------------------------------------------

void Particle::move( double phiA, double phiB, vector< int > pg )
{
  int index,
      localBestLessCurrent,
    globalBestLessCurrent ;
  double k ;

  // Clercs likes social and cognitive components
  // that add up to 4.1, and if they are not
  // greater than 4, the Clercs defaults to 1 which
  // does not add anything to the PSO.
  k = Base::clercs( phiA, phiB );
  if ( ( phiA + phiB ) < 4 )
  {
    k = 1.0 ;
  }

  index = 0 ;
  while ( index < getRepSize() )
  {
    localBestLessCurrent = p.at(index) - x.at(index) ;
    globalBestLessCurrent = pg.at(index) - x.at(index) ;

    v[index] += ( phiA * Base::gasdev(1) * localBestLessCurrent )
        + ( phiB * Base::gasdev(1) * globalBestLessCurrent ) ;
    v[index] *= k ;
```

```
    if ( abs( v.at( index ) ) > VMAX )
    {
      v[index] = rnd.myRandom() ;
      if ( rnd.myRandom() < 0.5 ) { v[index] *= -1 ; }
    }
    x[index] = ( rnd.myRandom() < Base::sigmoid( v.at(index) ) ) ? 1 : 0 ;

    index++ ;
  }

  fixExitLength() ;
  setFitness() ;
  calculateOnes() ;

  if ( fitness < pFitness ||
    ( int(0.5+fitness) == int(0.5+pFitness) && exits < pExits ) )
  {
    setP() ;
  }
}

//-----------------------------------------------------------------

string Particle::repXToString()
{
  int index = 0 ;
  string result = "" ;

  while ( index < getRepSize() )
  {
    if ( index > 0 )
    {
      result += " " ;
    }
    result += Base::intToString( x.at(index) ) ;
    index++ ;
  }

  return result ;
}

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

```
string Particle::repPToString()
{
  int index = 0 ;
  string result = "" ;

  while ( index < getRepSize() )
  {
    if ( index > 0 )
    {
      result += "␣" ;
    }
    result += Base::intToString( p.at(index) ) ;
    index++ ;
  }

  return result ;
}

//———————————————————————————————————————————————

string Particle::vsToString()
{
  int index = 0 ;
  string result = "" ;

  while ( index < getRepSize() )
  {
    if ( index > 0 )
    {
      result += "␣" ;
    }
    result += Base::doubleToString( v.at(index) ) ;
    index++ ;
  }

  return result ;
}

//———————————————————————————————————————————————

string Particle::toString()
```

```
{
  return ( repXToString()
    + "," + Base::doubleToString( getFitness() )
    + "," + Base::doubleToString( getTime() )
    + "," + Base::intToString( getOnes() )
    + "," + repPToString()
    + "," + Base::doubleToString( pFitness )
    + "," + Base::doubleToString( pSimTime )
    + "," + Base::intToString( pExits )
    ) ;
}

//------------------------------------------------------------

void Particle::setFitness()
{
  int i = 0 ;
  double fitnessSum = 0 ;
  while ( i < THRESHOLD )
  {
    do
    {
      fitness = Base::getCSFit( repXToString() ) ;
    } while( fitness < 0 ) ;
    fitnessSum += fitness ;
    i++ ;
  }
  calculateOnes() ;
  setTime( fitnessSum / THRESHOLD ) ;
  fitness = getTime() + getOnes() ;
}

//------------------------------------------------------------

void Particle::calculateOnes()
{
  int index, sum ;

  index = sum = 0 ;
  while ( index < getRepSize() )
  {
    if ( x.at( index ) == 1 )
```

```
    {
      sum++ ;
    }
    index++ ;
  }
  setOnes( sum ) ;
}

//----------------------------------------------------------------

void Particle::fixExitLength()
{
  int j = 0,
      count = 0,
    specialCount = 0 ;
  bool flag = false ;
  while( j < getRepSize() )
  {
    if ( repAt(j) == 1 )
    {
      count++ ;
      if ( j == 0 )
      {
        flag = true ;
      }

      if ( j == ( getRepSize() ) - 1 && specialCount != 0 )
      {
        specialCount += count ;
        if ( specialCount > getMaxExitLength() )
        {
          flipMember(j) ;
        }
      }

      if ( count > getMaxExitLength() )
      {
        if ( flag == true )
        {
          specialCount = count ;
        }
        flipMember(j) ;
```

```
        count = 0 ;
        flag = false ;
      }
    }
    else
    {
      if ( flag == true )
      {
        specialCount = count ;
      }
      count = 0 ;
      flag = false ;
    }

    j++ ;
  }
}

//------------------------------------------------------------

void Particle::flipMember( int index )
{
  if ( x.at(index) == 0 )
  {
    x[index] = 1 ;
  }
  else
  {
    x[index] = 0 ;
  }
}

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

Listing from header file `code/swarm.h`

Listing B.15: Multi-Page C Code for the PSO's Swarm header

```
#ifndef _SWARM_H_
#define _SWARM_H_

#include "particle.h"

```

```
typedef vector< Particle > VecPart ;

class Swarm
{
public:
  Swarm() {}
  Swarm( int, int, int, double, double ) ;
  ~Swarm() ;

  void init( int, int, int, double, double ) ;
  void update( int ) ;

  int findGlobalBest() ;
  int update() ;

  double getBest() { return swarm.at( findGlobalBest() ).getFitness() ; }

  string toString( int ) ;

// Setters
  void setSize( int _size ) { size = _size ; }
  void setRepSize( int _repSize ) { repSize = _repSize ; }
  void setMaxExitLength( int _maxExitLength )
  { maxExitLength = _maxExitLength ; }
  void setPhiA( double _phiA ) { phiA = _phiA ; }
  void setPhiB( double _phiB ) { phiB = _phiB ; }

// Getters
  int getSize() { return size ; }
  int getRepSize() { return repSize ; }
  int getMaxExitLength() { return maxExitLength ; }
  double getPhiA() { return phiA ; }
  double getPhiB() { return phiB ; }

private:
  int size,
    repSize,
    runNumber,
      maxExitLength,
    best ;
  double phiA,
    phiB ;
```

```
  Particle g ;
  VecPart swarm ;
} ;

#endif // _SWARM_H_
```

Listing from header file `code/swarm.cc`

Listing B.16: Multi-Page C Code for the PSO's Swarm source

```
#include "swarm.h"

Swarm::~Swarm()
{
  swarm.clear() ;
}

//--------------------------------------------------------------

Swarm::Swarm( int _size, int _repSize,
      int _maxExitLength, double _phiA, double _phiB )
{
  init( _size, _repSize, _maxExitLength, _phiA, _phiB ) ;
}

//--------------------------------------------------------------

void Swarm::init( int _size, int _repSize,
        int _maxExitLength, double _phiA, double _phiB )
{
  setSize( _size ) ;
  setRepSize( _repSize ) ;
  setMaxExitLength( _maxExitLength ) ;
  setPhiA( _phiA ) ;
  setPhiB( _phiB ) ;
  //cout << "Before G Init" << endl ;
  //g.init( getRepSize(), getMaxExitLength() ) ;

  int index = 0 ;
  while( index < getSize() )
  {
    //cout << "Particle " << (index+1) << endl ;
    swarm.push_back( Particle( getRepSize(), getMaxExitLength() ) ) ;
```

```
    index++ ;
  }
  //cout << "After Pushing particles into vector" << endl ;
  best = findGlobalBest() ;
  g = swarm.at( best ) ;
}

//------------------------------------------------------------
/*
 * This update method is for synchronous updating
 */
int Swarm::update()
{
  best = findGlobalBest() ;
  g = swarm.at( best ) ;
  int index = 0 ;
  while( index < getSize() )
  {
    swarm.at( index ).move( getPhiA(), getPhiB(), g.getX() ) ;
    index++ ;
  }
  return getSize() ;
}

//------------------------------------------------------------
/*
 * This update method is for asynchronous updating
 */
void Swarm::update( int index )
{
  best = findGlobalBest() ;
  g = swarm.at( best ) ;
  swarm.at( index ).move( getPhiA(), getPhiB(), g.getX() ) ;
}


//------------------------------------------------------------

/*
 * Best is initialized in the init function so it
 * doesn't have to be initialized in this function.
 */
```

```
int Swarm::findGlobalBest()
{
  int index = 0,
      b = 0 ;

  while( index < getSize() )
  {
    if( swarm.at(index).getFitness() < swarm.at(b).getFitness() )
    {
      b = index ;
    }
    index++ ;
  }
  best = b ;
  return b ;
}

//------------------------------------------------------------

string Swarm::toString( int runNumber )
{
  int index = 0 ;
  string result = "" ;
  while( index < getSize() )
  {
    result += Base::intToString( runNumber )
      + "," + Base::intToString( index+1 )
      + "," + swarm.at( index ).toString() ;
    result += "\n" ;
    index++ ;
  }

  return result ;
}

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

Listing from source file `code/pso.cc`

Listing B.17: Multi-Page C Code that Runs the PSO source

```
#include "base.h"
```

```
#include "swarm.h"

using namespace std ;

typedef vector< int > VecInt ;
typedef vector< Particle > VecPart ;

int main( int argc, char ** argv )
{
  srand( time( NULL ) ) ;

  int populationSize,
    index,
    repSize,
    count,
    generation,
    maxExitLength,
    currentParticle,
    runs ;       // function evaluations
  double phiA,
      phiB ;
  Swarm swarm ;
  Particle part ;

  if ( !argv[1] || !argv[2] || !argv[3]
     || !argv[4] || !argv[5] || !argv[6] )
  {
    cout << "Try something like ./pso 30 26 2.8 1.3 1000 3" << endl ;
    exit(0) ;
  }

  populationSize = Base::charStringToInt( argv[1] ) ;
  repSize = Base::charStringToInt( argv[2] ) ;
  phiA = Base::charStringToDouble( argv[3] ) ;
  phiB = Base::charStringToDouble( argv[4] ) ;
  runs = Base::charStringToInt( argv[5] ) ;
  maxExitLength = Base::charStringToInt( argv[6] ) ;

  swarm.init( populationSize, repSize, maxExitLength, phiA, phiB ) ;
  cout << "generation,individual" ;
  cout << ",xrepresentation,xescapetime,xexits" ;
  cout << ",prepresentation,pescapetime,pexits" << endl ;
```

```
  count = populationSize ;
  generation = 1 ;
  while ( count <= runs )
  {
    currentParticle = count % populationSize ;
    swarm.update(currentParticle) ;
    cout << swarm.toString( generation ) << endl ;
    count++ ;
    generation++ ;
  }

  return 0 ;
}
```