C++ Cluster AI

Ethan J. Eldridge

May 8, 2011

1 Abstract

Brief Overview of Parts

Cluster Basic is a simple program, given a square object $\mathcal{B}$ to move, a cluster $\mathcal{C}$ will push $\mathcal{B}$ in the most efficient way to a goal point $\mathcal{G}$. $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{G}$ exist in a two-dimensional space, where movement is defined in integers with no floating point precision.

Cluster with Obstacles builds on Cluster Basic by adding in obstacles that may or may not stand in the way of reaching $\mathcal{G}$. In addition, $\mathcal{C}$ will consist of two data structures, the Cluster $\mathcal{C}$ and it's Members $\bar{m}$ These members will be assigned to push $\mathcal{B}$ by an AI within $\mathcal{C}$, and will be selectively bred for strength.

Cluster Goal Driven will build off of both previous models to reach multiple goal locations $\bar{\mathcal{G}}$ with efficiency as well as priority of goals to be reached. In addition, $\mathcal{C}$'s population will not only increase but decrease as well, resulting in a more dynamic behavior.

2 Cluster Basic

2.1 Specifications

Within Cluster Basic, a *cluster* is defined as a single entity that keeps track of it's total population and how much force said population can exert on objects. These clusters are simple organisms whose only goal is to breed new members and to push an object to a location. $\mathcal{B}$ is a simple square that has four corners to be pushed against by $\mathcal{C}$. The state of $\mathcal{B}$ can be given by two lists of information, these are specified in §2.2. A goal $\mathcal{G}$ is a simple 2-Tuple specifying a point in space that $\mathcal{B}$ needs to be moved to. A member of $\mathcal{C}$ does not consist of it's own structure, but is merely stored as a population value in the cluster data structure.

2.2 Data Structures

[List of operations and specifics of $\mathcal{B}$,$\mathcal{G}$, and $\mathcal{C}$]

$\mathcal{B}$ Member Fields

$\mathcal{B}$ has only two variables, *boxState* and *cornerState*, both of which are arrays. The first, *boxState*, can be represented easily by $|x|y|dx|dy|m|$, this is easily understood as the ordered pair for $\mathcal{B}$'s position as elements one and two, the next change to occur when moved by the next pair, and the last element corresponds to the force neccesary to move $\mathcal{B}$ to another point in the world. The second array contains only boolean values, which map onto the corners of $\mathcal{B}$ in the following way:

Upper left	→	One
Upper Right	→	Two
Lower left	→	Three
Lower right	→	Four

The purpose of *cornerState* is to allow $\mathcal{C}$ to know which corners are already being pushed against and to place it's members accordingly. This construct is not used explicitely within Cluster Basic, but the author expresses that the third and fourth elements in *boxState* could be replaced by this construct given some more complexity within the movement code. It would be an interesting exercise to modify the program to incorporate such an idea, but that is not found within the scope of Cluster Basic. Further uses of *cornerState* can be found within §3.

$\mathcal{B}$ Operations

$\mathcal{B}$'s only operations are an initializer that sets all fields to default values, a move command, and a toConsole function.

- *init* instantiates *boxState* to a state we represent as | 0 | 0 | 0 | 0 | 1.0 | and every value of *cornerState* to false.
- The *move* function simply adds elements three and four of *boxState* to the first two indices.[1]
- *toConsole* does exactly what it's name suggests, it sends the information of *boxState* and *cornerState* to the console.

$\mathcal{G}$ Structure

$\mathcal{G}$ is a simple 2-Tuple, containg two variables of type double f and s. The operations of $\mathcal{G}$ therefore are simple *fst* and *snd*, and like our other data structure, a *toConsole* function. *fst* returns the first element of the tuple, and *snd* returns the second. *toConsole* prints out (fst, snd).

$\mathcal{C}$ Member Fields

$\mathcal{C}$ contains four variables, two of which pertain to population, another to force. and the last is a variable to keep track of the next move to apply to $\mathcal{B}$

- *numMembers* is $\mathcal{C}$'s population expressed as an integer value.
- *forcePerMember* is a double value that expresses the amount of force that a single member within $\mathcal{C}$ can apply to $\mathcal{B}$.
- *force* is the combined total force of all members of $\mathcal{C}$. This value is computed as *numMembers*·*forcePerMember* and to prevent a loss of precision, is stored as a double value.
- *move* is a Tuple containing the best move to be applied to $\mathcal{B}$. The values of *fst* and *snd* of this Tuple range from -1 to 1

[1] For Clarity: The elements are added three to one, and four to two.

$\mathcal{C}$ Operations

$\mathcal{C}$'s main operation is to decide to increase it's popultation using the *breed* function, or to push $\mathcal{B}$. However to accomplish this, $\mathcal{C}$ enlists the following operations.

- *toConsole* as usual sends out data about the member fields to the console
- *checkForce* calculates the total force possible by $\mathcal{C}$ and stores it into *force*.
- *breed* increases *numMembers* by some breeding factor. In Cluster Basic, this is a simple addition of $\frac{numMembers}{2}$
- *assignMembers* modifies the *cornerState* of $\mathcal{B}$ to reflect which corners are being pushed on by $\mathcal{C}$
- *determineMove* is the main work horse of $\mathcal{C}$ and determines the move to be made. See §2.3 for details
- *action* determines what $\mathcal{C}$ will do, *breed* or *determineMove* and call *assignMembers*.

2.3 The brain of the Operation

As stated above, the primary work horse of $\mathcal{C}$'s thinking is the function *determineMove*. This function can be divided into a few phases.

1. generate possible moves of $\mathcal{B}$
2. compute the manhatten distance of each position generated
3. choose the move with the lowest manhatten distance
4. update $\mathcal{B}$'s *boxState* and $\mathcal{C}$'s *move* fields to reflect these changes.

Each of these subroutines are easily followed, and make sense within the context of the program.

Generating the possible moves is easily done by subtraction or addition of the current *boxState* (elements one and two) and some predefined dx, dy, in Cluster Basic's case dx, dy are computed by two for loops running through $-1 \rightarrow 1$ for each variable. If the reader is visual minded, they may draw out a tic-tac-toe board, where the current position is the center, and the possible moves are the squares surrounding it.

A manhatten distance is computed as $dx + dy$, and since we have already stored the moves possible, we can compare these to $\mathcal{G}$ and compute our distance. This is more efficient than finding the euclidian distance from one point to the other because we don't need to take a square root. This will save us time complexity later on.

Choosing the correct move is easily done by sorting the possible moves with an insertion sort. While the reader, if versed in algorithms may immediatly question this choice, (specifically over using quicksort) the author reminds them that on small data sets insertion sort outperforms quicksort. Because we know that the number of moves to be sorted will only ever be nine, we can safely use insertion sort without fear of reprimand. Once the items are sorted into ascending order, we simply store the first element in the list as our next move. This is a quick and effective solution to choosing the best move. The function itself has a void return type, and only operates as a side-effecting operation acting upon the states of $\mathcal{C}$ and $\mathcal{B}$.

2.4 Runtime Analysis of $\mathcal{C}$'s *action* Function

$\mathcal{C}$'s *action* function is a complex study, and the runtime can be expressed in multiple ways depending on it's context. At it's fastest runtime, the function checks itself, and then

calls *breed* only. This effectively takes $3\Theta(1) + 2M(n)$[2], as the operation is nothing more than a boolean check[3] and then a call to *breed*, which as we know, consists of an addition and a multiplication operation. However, this case where *action* takes this brief amount of time only occurs while *force* is less than *boxState* element 5. And eventually, depending on *forcePerMember*, this case will not rise again.

It follows to study what happens when this best case scenario does not arrise. The boolean operation to check the force is still carried out, however this time, it evaluates to true and we step within the body of the If expression. We then have two more operations to perform. Recall here that these two operations are *determineMove* and *assignMembers*. First, *assignMembers*'s runtime depends on the move chosen by *determineMove* within the previous run of *action* (why we study *assignMembers* first within this section will become clear soon). At it's worse, we'll compare the first item of the tuple twice before falling into the else clause. And since within *assignMembers* we must check both the first and second element of *move* this worse case can occur twice. This adds $5\Theta(1)$ to our complexity in the worst case[4]. In the best case, we make only a single comparison on each element before falling into the correct statement, in which the runtime $3\Theta(1)$ occurs.[5]

The *determineMood* method, as the main work-horse of *action* is where the largest amount of time is spent, and it's best and worst cases differ by about 60%. A good deal of the algorithm can be seen to be constantly the same, for example, certain boolean checks will always be done, certain loops will always execute the same number of times, and so on. Consider the complexity of these constant operations, let us keep a running total as we go through the algorithm. First, a boolean check will always be done at the beginning of *action* to determine whether to breed or not. This check itself takes $\Theta(1)$ plus the amount to call *checkForce*. As stated in footnote 3, *checkForce* takes $M(n) + \Theta(1)$ time. The total time so far is $M(n) + 2\Theta(1)$,

Next, we move onto *determineMove*. Ignoring variable length operations, we can observe that the loop to generate moves will always run through all nine possible moves. This sets our multiplication factor of the operations inside the loop to nine. Within the loop, there are two additions. Effectively, this means that the first loop takes $9 \cdot 2\Theta(1)$ however, we can be more specific and include the boolean checks and iterations stored within the for loop contrust itself, this bumps the runtime of the loop to $9 \cdot 7\Theta(1)$ or simply, $63\Theta(1)$. Bringing the total so far up to $65\Theta(1) + M(n)$. Continuing to the next loop within *determineMove* we compute the manhatten distance of each move, ignoring the probability of calling an absolute value function, we have four additions and a boolean to add. This brings the total to $70\Theta(1) + M(n)$. Because we are always going to check to see if the number passed to the absolute value function is less than zero, we can factor this into the constant runtime, but we cannot factor in the multiplication by negative one if the check returns true. Given that dx, dy requires the function to be called on each item, and that the loop will parse all nine moves, we can factor in another $18\Theta(1)$, increasing the total to $88\Theta(1) + M(n)$.

After *determineMove* we'll proceed into *assignMembers* where we can add $\Theta(1)$ to our total, however this is the only constant operation within that function, and since this is the last subroutine called within *action*, we have our total constant time as $89\Theta(1) + M(n)$.

The variable time, as I will call it, results from the uses of the absolute value fucntion *abs*, *insertionSort*, and *assignMembers*. Let us study each of these in detail: $\mathbf{O}(abs)$ is $M(n) + \Theta(1)$, this occurs when the number passed to *abs* is less than zero, because we must

[2] $M(n)$ is the runtime of whatever multiplication technique the compiler uses

[3] This boolean check consists of the check itself and the call to *checkForce* which takes $M(n) + \Theta(1)$ time.

[4] the 5th comparison coms from a check on *numMembers* to make sure there are more than 2 at the time. The function returns false if this is the case, in which *action* will call breed, however, within our program we initialize *numMembers* to start at two, so this never occurs and thus has been omitted from the analysis.

[5] Once again, the extra $\Theta(1)$ occurs as a result discussed in the previous footnote.

multiply the argument by negative one in order to flip it's sign. At best, we only have a single constant boolean check and then we exit. We've already factored this check into our constant time, so all that remains is to determine how many times we'll call on the internal structure of *abs* given some arbitrary $\mathcal{G}$. Consider the diagram below:

abs calls

+ +	+ +	- +
++	++	- +
+ -	+ -	- -

moves generated

$-1, 1$	$0, 1$	$1, 1$
$-1, 0$	x, y	$1, 0$
$-1, -1$	$0, -1$	$1, -1$

If you know the direction that $\mathcal{G}$ lies in, relative to $\mathcal{B}$, then we can check the left diagram to see how many calls to *abs* there will be. If we expand our graph from 3X3 to a 5X5 then we can count up the number of possible calls to *abs* and include this as a probability given some $\mathcal{G}$. Within our first graph, the probability will be $\frac{1}{3}$. This can be deduced by counting the number of $-$'s in the left diagram. Within our 5X5, this probability increases to $\frac{2}{5}$. If we keep increasing the side length keeping our current location at the center of the graph, we can generate a series. Starting from a side length of 1:

$$(6, 14, 22, 30, 38, 46, 54, 62, 70, 78, 86, \cdots)$$

This sequence of numbers is the additional number of calls to *abs* to the previous number, we can then create a sequence of the total calls as:

$$(0, 6, 20, 42, 72, 110, 156, 210, 272, 342, 420, \cdots)$$

As these numbers represent the total possible calls to *abs*, we still need to know how many possible calls there are. This can be easily computed as the side length s squared. Since s increases by two each time (once again, in order to keep $\mathcal{B}$ in the center) we can represent s as $2n - 1$ and s^2 as $(2n - 1)^2$. This however, is not the denominator of our fraction. Why? Because each square in our grid structure has an x and a y component, and therefore *abs* may be called up to two times per square. This means that the final bottom of our probability can be defined as $2 \cdot (2n - 1)^2$. The numerator can be defined as the formula that gives us the second sequence listed above, which is $2(n - 1)(2n - 1)$. Therefore, our probability equation is the following:

$$\mathbf{P}(n) = \frac{2(n-1)(2n-1)}{2 \cdot (2n-1)^2} \text{ which simplifies to } \frac{n-1}{2n-1}$$

where n is the number of increases to the side length The derivation of the sequence is easiest shown graphically, consider the diagram below:

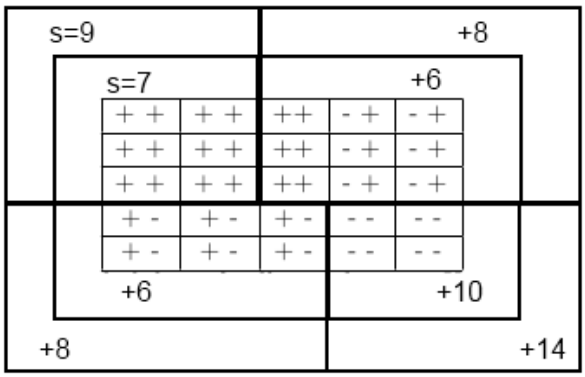
+ +	+ +	++	- +	- +
+ +	+ +	++	- +	- +
+ +	+ +	++	- +	- +
+ -	+ -	+ -	- -	- -
+ -	+ -	+ -	- -	- -

The probability of calls to *abs* in a 5X5 graph

Compare this to the 3X3 graph we've created before, and you may notice a pattern. Specifically around the corners of the graph. Before continuing on, the reader, if mathematically inclined, might try to come up with the relation between side length and increases themselves. We can see that as s increased from three to five, each corner increased by a fixed amount. The top left corner, which we can count as all the squares with ++ as their state, increased by seven. The top right corner increased by six blocks, since a single call to *abs* is present in each of these squares, we can see that the total probability for *abs* has increased. The bottom left corner is merely a reflection of the top right, and we can count this as the same case. The bottom right corner increased by five blocks, but the calls increased by ten. If we continue increasing s by two, and counting we'll come up with the following sequence:

$$(6, 14, 22, 30, 38, 46, 54, 62, 70, 78, 86, \cdots)$$

familiar looking? The figure below may clear up any confusion of where these numbers are coming from:



The successive additions of calls to *abs*

If we sum these additions up, we can attain our second sequence of numbers

$$(0, 6, 20, 42, 72, 110, 156, 210, 272, 342, 420, \cdots)$$

and from here, it's finding the numerator of the probability equation is simple math. Next, we know that the area of a rectangle is $l \cdot w$ and since we're keeping our graph a square by only having odd number of side lengths, we can then create the sequence of side lengths as

$$(1, 3, 5, 7, 9, 11, 13, 15, 17)$$

this is easily created into a formula using the definition of odd numbers $2n-1$. Since $\mathbf{A} = s^2$, we can attain $(2n-1)^2$ and as before, each square has up to two calls to *abs*, so we create our denominator as $2 \cdot (2n-1)^2$. Our unsimplified formula is our result:

$$\mathbf{P}(n) = \frac{2(n-1)(2n-1)}{2 \cdot (2n-1)^2}$$

$$\text{cancel the } \tfrac{2}{2} \qquad \frac{(n-1)(2n-1)}{(2n-1)^2}$$

$$\text{expand denominator} \qquad \frac{(n-1)(2n-1)}{(2n-1)(2n-1)}$$

$$\text{cancel (2n-1) and } \mathbf{P}(n) = \frac{n-1}{2n-1}$$

Now that we've derived a formula for our probability, given any $\mathcal{G}$ we can find the side length neccesary to encompass the point, and then the probability. In general though, considering a large problem space, the formula will converge to the value $\frac{1}{2}$, and this is the probability we use within our average case for runtime complexity. For the first few problem sizes, the probabilities have been listed out in the figure below.

side length s	$2s^2$	calls to *abs*	n	$\mathbf{P}(n)$
1	2	0	1	0
3	18	6	2	$\frac{1}{3}$
5	50	20	3	$\frac{2}{5}$
7	98	42	4	$\frac{3}{7}$
9	162	72	5	$\frac{4}{9}$
		$\vdots$		
$2n-1$	$2(2n-1)^2$	$2(n-1)(2n-1)$	∞	$\frac{1}{2}$

We can now state the best,worst, and average number of calls to *abs* for some unknown $\mathcal{G}$ as:

Best: $0 \cdot M(n)$	**Worst:** $18 \cdot M(n)$	**Average:** $9 \cdot M(n)$

The author wishes to remind the reader at this time that this is when $\mathcal{G}$ is unknown, given a known $\mathcal{G}$ we can easily figure out the number of calls to *abs* by thinking about the direction that $\mathcal{G}$ lies in.

After *abs*, we call upon our *insertionSort* to sort our possible choices. We can simply substitute the known runtime of insertion sort into our runtime analysis. However, we do need to consider the runtime of *assignMembers* in more detail. Let us start our analysis by viewing the source code:

```
if(f < 0){
    if(s > 0){ b->cornerState = {false,false,false,true};
    }else if(s < 0){b->cornerState = {false,true,false,false};
    }else{ b->cornerState = {false,true,false,true};  }
}else if(f > 0){
    if(s > 0){ b->cornerState = {false,false,true,false};
    }else if(s < 0){b->cornerState = {true,false,false,false} ;
    }else{ b->cornerState = {true,false,true,false};
    }
}else{
    if(s > 0){ b->cornerState = {false,false,true,true};
    }else if(s < 0){ b->cornerState = {true,true,false,false};
    }else{ b->cornerState = { false,false,false,false};
    }
}
```

Both s and f come from the *move* Tuple stored within $\mathcal{C}$ and from this we define our *cornerState* to reflect which corners need members applied to them. Within Cluster Basic, the actual assignment of members is not handled, but the basics of the function are set up. At it's best possible runtime two comparisons are made. At worst, we check both cases against f and then fall into the else statement, which we then check s and do the same. Resulting in a worst case of $4 \cdot \Theta(1)$. It follows to question what the average runtime is. As *abs* relied on where $\mathcal{G}$ was, we shall apply a similar technique in our proof of the average runtime. First, consider the diagram below:

$-1, 1$	$0, 1$	$1, 1$
$-1, 0$	$0, 0$	$1, 0$
$-1, -1$	$0, -1$	$1, -1$

diagram 1

This is almost the same diagram that appeared earlier in the explantion of *abs* runtime, and is in fact, the same moves generated only now, we're counting the middle $(0, 0)$ pair. We can modify this diagram to reflect how many comparisons will happen given the move that occupies the corresponding space in diagram 1. This comparison diagram is shown below:

l,ll	ll,ll	ll,ll
l,ll	ll,ll	ll,ll
l,l	ll,l	ll,l

Unlike *abs*, this is our entire problem space, and the probabilities of each number of comparions can be found by simple observation. For an unknown $\mathcal{G}$ the probability of the best

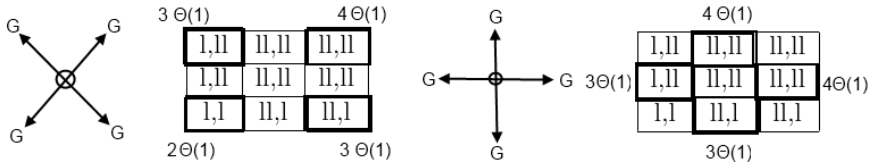
case is a mere $\frac{1}{9}$. However, if we know that the $\mathcal{G}$ is located below and to the left of our current position, this best case will be called on more more than the others. However, we are considering the general case of an unknown $\mathcal{G}$.Both the worst case, and average case occur with equal probabilitys of $\frac{4}{9}$. Using this knowledge, given some $\mathcal{G}$ we'll run with $4 \cdot \Theta(1)$ *almost* half of the time, and $4 \cdot \Theta(1)$ the other 'almost' half. This means, on average, we'll encounter the worst case $\frac{4}{9}$'s of the time. However, as we are continuously moving towards $\mathcal{G}$, and once we reach it we will always call the center square of the comparison chart. And thus, we'll tend more towards the worst running time for *assignMembers*. Also, given that the space $\mathcal{C}$ exists in is defined a non-negative integer environment, we'll in general, be more likely to move in a positive,positive direction. This leads towards the better $3 \cdot \Theta(1)$ time more often in practice. These assumptions sounds good, however, we can go one step further and prove the first trend.

First, we express the cases where the runtime is the worst as F and when the runtime is $3 \cdot \Theta(1)$ as E. Given the odds that E will occur before F, we can find the conditional probability that F will occur. This is easily done by application of Bayes Theorem.

$$\mathbf{P}(F|E) = \frac{\mathbf{P}(F \cap E)}{\mathbf{P}(F \cap E) + \mathbf{P}(\neg E \cap E)}$$

This calculation will result in a value of $\frac{1}{2}$. Because $\frac{1}{2} > \frac{4}{9}$ this proves the first trend. The second trend needs no mathematical proof; if $\mathcal{G}$ is located in the III quadrant of our graph (with the current location as the origin) *determineMove* will always choose the move resulting in the best case running time.[6]

Putting the trends aside, given that we are not yet at $\mathcal{G}$ we will have 8 possible moves to check. Let us consider two cases, one where one of $\mathcal{G}$'s elements are equal to it's corresponding element in *boxState* and when this is not the case.[7] Studying the two diagrams below we can see that $3 \cdot \Theta(1)$ occurs more often than the worst case, and the best case.



The worst time occurs three times, the best once, and the other case four times. Therefore, given some random $\mathcal{G}$, the unnamed case will be more likely than the worst case by about an eighth of the time. We can therefore say that on average, the runtime for *assignMembers* will be $3 \cdot \Theta(1)$.

With the runtime of the individual parts of *action* figured out, we can now state the worst,average, and best case of the algorithm.

Base time of $89\Theta(1) + M(n) +$		
Best Case	Average case	Worst Case
$11\Theta(1)$	$9 \cdot M(n) + \frac{87}{2} \cdot \Theta(1)$	$18 \cdot M(n) + 81 \cdot \Theta(1)$

[8]

As a quick side note, while insertion sort's average runtime is normally assumed to be $O(n^2)$, this is taken without regards to it's coefficient, which since we know the specific n we can use the more precise runtime of $\frac{1}{2}O(n^2)$[9]

[6]This best choice move is $(-1, 1)$

[7]If this is unclear, I simply mean that an x or y component of the move Tuple is 0.

[8]Average case $\Theta(1)$ coefficient is found by the base $3 + \frac{1}{2} \cdot 81\Theta(1)$

[9]This coefficient is found in an article at http://www.personal.kent.edu/ rmuhamma/Algorithms/MyAlgorithms/Sorting/insertion

2.5 Running the Program

Running the program can be done by hardcoding the desired values into the code itself and rebuilding the project, or from running the executable with different command line arguments. An example of arguments and their resulting behavior is expressed in the table below:

Input	Output
Cluster.exe	default values are loaded
Cluster.exe 5 5 13	$\mathcal{G}$ of $(5, 5)$ simulation runs set to 13 default values for $\mathcal{B}$
Cluster.exe 5 5 1 0 13	the same as above except: $\mathcal{B}$'s state is set to | 1 | 0 | 0 | 0 | 1.0 |
Cluster.exe 5 5 1 0 2.0 13	same as above except: $\mathcal{B}$'s state is set to | 1 | 0 | 0 | 0 | 2.0 |
Cluster.exe 5 5 1 0 2.0 .35 13	same as above except: *forcePerMember* is set to .35

3 Cluster with Obstacles

3.1 Primary Changes from Cluster Basic

As given in the title of the extended Cluster Basic, there are now obstacles to deal with while figuring out a path. If we are to continue to exist in an undefined space[10] then we must store these obstacles in some way, and check our moves against them. That, or we must define a more limited world for $\mathcal{C}$ to exist in. Also, instead of $\mathcal{C}$'s population existing solely as a number, each member of the population is now defined by it's own data structure. This modifies numerous functions we used in Cluster Basic, and they must be updated accordingly. Also, as a side effect of creating a new data structure for members, we breed members selectively to maximize their *force* possible. The 'mind' of the cluster can be created to maximize in the sense of the short term, or in the sense of the long term. How, and why it would do this is discussed further in this section.

3.2 Specifications

As it was in Cluster Basic, a cluster is defined as an entity encapsulating a population, that acts based on a goal $\mathcal{G}$, in this case to push an object $\mathcal{B}$ to a designated point $\mathcal{G}_p$. As stated above, the encapsulated population $\bar{m}$ now has it's own data structure and because of this numerous changes to the existing data structures must be done. These are detailed within §3.4. In addition to this, obstacles introduced into our world must be accounted for, and we must now decide whether to reduce our world from a limitless mathematical universe to a finite graph, or to continue existing within an infinite universe. The advantages and disadvantages of each are discussed within the next section.

3.3 Advantages and Disadvantages of our Representation of $\mathcal{C}$'s World

So far, the space in which $\mathcal{C}$ exists has not been formally defined except as a two-dimensional space with discrete integers, we have had no data structure, no grid, graph, or adjacency list to represent our problem space $\mathcal{W}$, this is simply because we have had no need to. But introducing obstacles requires that we keep track of them. We can do this in two different ways, one will allow us to keep out limitless space $\mathcal{W}$ and the other will reduce us down to some specified size contained within some sort of graph-like data structure. There are advantages and disadvantages to both of these, the ones which immediately pop into mind are listed below.

Limitless	Limited
Any Problem Size considerable	Limited Problem Size
Obstacles must be kept in some store	Obstacles placed into $\mathcal{W}$
Must search store for obstacles	No searching through all obstacles to check for collision

Most projects that use some type of world space, limit it down to some size, however, within this phase of the project this is not done. Why? Because, while we could create a graph, and then utilize our efficient and well known algorithms to make our problem significantly less difficult, as a human we exist within a mix of the two types of $\mathcal{W}$, we exist on a finite planet, in an infinite uinverse. But, if you observe the world around you, you know it is limited, but you'll never travel all of it, and so it seems limitless in some sense. Observable viewpoints

[10] $\mathcal{C}$'s plane of existence is not defined by an array or graph of any kind, and solely exists as points in the non-negative integers.

aside, a limitless space is considered because the author finds it to be more interesting and challenging. However, since we have now chosen this $\mathcal{W}$ we can now get into the details of how we are going to implement it.

3.4 Storing and Searching Obstacles Efficiently

Efficiency is the rule of thumb for dealing with a limitless space. We don't want to waste all our time checkings hundreds of obstacles and never actually doing work. So, we must adhere to some criteria, after all it is not how you ask the question, but what questions you ask that lead you to solutions. Obstacles $\bar{o}$ must be stored within some data structure that is easy to search and doesn't have a long searching time. There is one more criteria, and this is the deal breaker. Simply given those two points above, one might go with a simple heap, or perhaps some other heap or listlike data structure. Any obstacle $\mathcal{O}$ from $\bar{o}$, when accessed, is very likely to be accessed again during the consideration of the next move. This means that a dynamic data structure will serve our needs best, and as such Cluster with Obstacles utilizes a splay tree to store obstacles.

Of course, this leads to some more questions. How are we going to order the moves? Each $\mathcal{O}$ is stored as a Tuple, and as such has two elements. The tree could order this Tuple by it's sum, it's first element, second, or even by both. For simplicity and fast running time, we order $\bar{o}$ by the first element of the tuple.[11]

3.5 Data Structures

Members $\bar{m}$ Member Fields and Operations

Each member m contains two double type variables, *force* and *mF*. *force* is the individual member's possible force to be applied and *mF* stands for it's mutagen factor. This number represents how stable or unstable it's genetics are. An m with a high *mF* is more likely to produce a child with a differing *force* from it's own. This factors heavily into the members only function *breed*. This function operates on two $m \in \bar{m}$ and uses a function of their *force* and *mF* to create an offspring. This function can be expressed as the average of the parents *force* $+df$ where:

$$df = \frac{f_1 \cdot mf_1 + f_2 \cdot mf_2}{2mf_x} \text{ and } mf_x = \frac{1}{e^{mf_1+mf_2}}$$[12]

This function creates a realistic change based on the mutagen factor of each m, the breeding of parents and their resulting child is expressed by the table:

Parent One	Parent Two	Offspring
Low *mf*	Low *mf*	Low *df*
High *mf*	High *mf*	High *df*
High *mf*	Low *mf*	Very High *df*

Of course, this formula also depends on the *force* given by either parent. Holding these values as constants allows for the above table to be generated. However, when a low and high force are mixed, given a high mutagen factor, the table still holds strong. How $\bar{m}$ are selected to breed can be found in §3.6.

[11] If two Tuples have the same element for their first member, then the Tuples are compared based on their second.
[12] f_1 stands for *force* of one member and this notation persists throughout the equation

$\mathcal{C}$'s member fields are extended to include a pointer to the move previously done. This new variable *prevMove* is used within the modified *determineMove* function. Because each m within $\bar{m}$ has it's own *force* variable, $\mathcal{C}$'s *force* variable is no longer used the same way as it was within Cluster Basic. Instead of *numMembers* multiplied by *forcePerMember*, *force* is used to keep track of the total force, but it is also used to compute *forcePerMember* by taking the average force of all it's members. This allows $\mathcal{C}$ to assign members to $\mathcal{B}$, with reasonable confidence that they'll be able to push it. We can also use this average to decide on different strategies to pick members. If the average is above the mass of $\mathcal{B}$ then we can assign any old m. If it's below, then theres a chance that the sum total of all $\bar{m}$ will not be greater than the mass. This will be discussed in further sections. *force* is updated each time a *breed* operation occurs, adding the offsprings *force* to the total. In addition to these changes, $\mathcal{C}$ keeps track of the lowest and highest *force*[13] of $\bar{m}$. This, combined with the average and a variable *confidence*, allows for $\mathcal{C}$ to make reasonable assignments for each of it's members.

$\mathcal{B}$ is changed to accomodate individual forces being applied to the box. This is done by another array, mirroring *cornerState*, *cornerWeight* is an array of type double, which holds the mass of $\mathcal{B}$ divided evenly between it's four corners. This array adds a small amount of storage complexity, but considering to check $\bar{m}$ *force* against the weight of a corner would involve multple multiplications repeated over and over again, this saves us time overall. If the reader is confused, the author offers this brief example: If checking two numbers takes a single boolean comparison, and if one of those numbers must be computed as a division everytime that comparison is made, and you repeat this multiple times, how many multiplications will you have done? And conversly, if you store the multiplication in some variable before starting your comparisons and don't recompute it everytime, how many times will you have multiplication then? The answer to which is faster is obvious.

3.6 Breeding

They're are three main parts of the breeding process. Selection, Updating, and the side effects of the function. The last two are natural when the function is considered within the whole of the program, and the selection is the problem considered in detail here. Let us remove the two superfluous details. The side effecting operation can be understood simply by viewing the function prototype:

```
void breed( CMember &p2, CMember &child)
```

If the reader is not tipped off immediately by the void return type of *breed*, the fact that the second parameter is a reference to an instance of *CMember* named child should be enough. Updating $\mathcal{C}$ after the breeding process occurs involves comparisons between the new child, *lowerBound*, *upperBound*, and updates to *totalForce*,*averageForce*, and *confidence*.

Selection of members to breed can be handled in a few different ways. First, we can randomly select members from our container and breed them together until we've breeded all the possible pairs together, without replacement. This is easy to do, but is difficult to analyze and produces very random results. Another method is to start at either end of the container and breed until the two counting variables cross in the middle. I dubbed this *dumb* or *iterative cross breeding*. Why? Because we don't order our members and therefore this semi random breeding can be done while fully understanding the running time of it. Of

[13]These two bounds are declared *lowerBound* and *upperBound* respectfully

course, if there is a *dumb breeding* strategy, then it stands to reason that there is a *smart breeding* strategy. The reader would be correct in this assumption. The algorithm is exactly the same as *dumb* except that before the selection occurs, $\bar{m}$ are sorted in order by their *mf*. Recall tha table in §3.5, a low and a high *mf* mix will result in an extreme change in force from the offspring and the parent. *Smart breeding* exploits this fact. However, we can also create another strategy if our members are sorted, *iterative breeding*. This strategy could also be called *stable breeding* because if we breed the highest with the next highest, and so on until we reach the lowest, then we will have little change between generations and will get a similar pool of low and high *mf*'s. Another strategy we can consider is *generational breeding*, this would use an added field to keep track of what generation each m belongs to, and then only breed members of the same generation with each other. Within Cluster with Obstacles' code we use the *iterative cross breeding* strategy because it simulates the pseudo random breeding styles of organisms more closely. That is, not every birth is perfect and there are imperfections within a population. $\mathcal{C}$ is not a believer of eugenics within this implementation.

3.7 Changes to *action*'s Internal Structure

The algorithm for *action* has not changed overall, it still consists of the same operations, *checkForce*,*determineMove*,*assignMembers*, and *breed*. However, every one of those functions has been modified in some way to reflect the new way *mathcalC* handles it's problem space. Each is considered below:

3.7.1 *checkForce*

In Cluster Basic, *checkForce* returned false if *force* was greater than *bState*[4], or if the force was strong enough to push the box's mass, then the move could then be determined, otherwise it would decide to breed instead. With the inclusion of a confidence variable K inside $\mathcal{C}$ and the running average of $\bar{m}$, this function increases from a simple boolean check to include a multiplication as well. Instead of the previous runtime of $\Theta(1)$, we now have a constant runtime of $M(n) + \Theta(1)$.

3.7.2 *determineMove*

The only change to *determineMove* is to include the code neccesary to check the moves against the obstacles known. As stated in §3.4 a splay tree is used to hold the obstacles. A brief look at the code will help discussion further:

```
 //remove conflicting obstacles from choice
for(int n=0; n < 9; n++){
oTree = find(moves[(int)(manh[n].snd())],oTree);
 if(checkRoot(moves[(int)(manh[n].snd())],oTree)){
         continue;
}else{
         choice = manh[n].snd();
         break;
         }
}
```

As you can see, the splayTree is used twice, once to attempt to find a move, and second to perform the actual check to see if the move brought to the root is the move we're looking for. *moves* is the array of Tuples generated within the first part of *determineMove*, these are

unedited moves that do not take obstacles into account. *manh* is an array of Tuples that holds the manhatten distance of each Tuple from *moves* as it's f variable and the index of the move in *moves* stored in it's s member field. Since Tuples stored types of double, then in order to index into *moves* with s from the move considered in $manh[n]$ we must cast the double to an int type. If the move chosen is an obstacle stored in the splay tree *oTree* then we continue our loop and check the next best move. Once we've found a move that is not an obstacle then we will break out of the loop with the correct choice. The reader might think that loop could possibly run out of moves to check. This is impossible. $\mathcal{C}$ always has to have come from some previous move, and will choose this move as it's option if no better move is seen. At worse, the last move will be the second to last in the sorted *manh* list. We can state this with confidence because consider if $\mathcal{G}$ is located north of $\mathcal{B}$, the move with the worst manhatten distance will be behind the southernmost move. Because this move will always be south of the previous move (in this example) it will never be chosen above the previous move.

Expanding this example to include the general $\mathcal{G}$ in any direction, we can see that the option never chosen is the opposite compass direction of $\mathcal{G}$. this reduces our list of possible moves from nice to eight. Since at worse, we'll search for 8 moves within the splay tree, we can state that the searching will take in the worst case is $\mathbf{O}(8 \cdot log\ \wr\mathcal{O}\wr)$. The best case would be if the first move checked was not in the tree, and therefore only a single *find* would occur, giving us a best runtime of $\mathbf{O}(log\ \wr\mathcal{O}\wr)$. The average number of comparisons really depends on $\wr\mathcal{O}\wr$, if the obstacles are sparse then we will have a lower number of searches done before finding our best move, if the obstacles are dense, (such as in a maze), then we will only have a few moves to choose from, and the setup of the maze will determine this probability. In a sparse or dense obstacle space we will only compare, on average, three moves before finding a legal move. This number is found by empiracal testing on random test environments. Therefore, the average running time of the search is $\mathbf{O}(3 \cdot log\ \wr\mathcal{O}\wr)$. Where $\wr\mathcal{O}\wr$ is the number of obstacles within our problem space.

The function *checkRoot*'s running time is $4\Theta(1)$[14] and therefore the worst runtime is $32\Theta(1)$. Because the average number of searches determines whether a *checkRoot* operation will take place, the average of calls to *checkRoot* will be three. And therefore, the average runtime of that section of code will be $12\Theta(1)$.

The previous complexity of *determineMove* remains the same, the calls to *abs* and the runtime of the *insertionSort* remain consistent. And therefore the runtime cases of *determineMove* are:

Best:	$\mathrm{O}(log\ \wr\mathcal{O}\wr) + 13\Theta(1)$
Average:	$\mathrm{O}(3 \cdot log\ \wr\mathcal{O}\wr) + \frac{105}{2}\Theta(1) + 9 \cdot M(n)$
Worst:	$\mathrm{O}(8 \cdot log\ \wr\mathcal{O}\wr) + 18 \cdot M(n) + 113\Theta(1)$

3.7.3 *assignMembers*

In addition the boolean checks discussed in the section of §2.4 dealing with *assignMembers*, the checks that $avgForce \cdot K$ is greater than the weight of whatever corners will be pushed must be made. These checks and multiplications are fairly consistent with one exception.[15] If the Tuple chosen has an f of 0, then we must push on two corners in either case, and we must check our calculation against the sum of each corner's weight. This re-

[14] A check to see if the tree is null, and then a check against each tuples members field and then a final and operation.

[15] Two exceptions if you attempt to consider the current location of $\mathcal{B}$, if there is no work to be done, then we do not check to see if we have enough to push it because that would be unneccessary. However, this case is not considered during our analysis because only care about before reaching $\mathcal{G}$ for now.

sults in an exception case that will occur $\frac{1}{8}$ of the time[16]. This exception has a runtime that varies from the worst case by a single comparison. The number of multiplications is always consistent[17] so we can always count on a single multiplication in our runtime. The number of comparisons also increase by one, the varying element of our runtime is whether or not we are pushing a single corner. If we are not, then we must add the weights of the corners we are pushing together. The new runtime of *assignMembers* is:

Best: $3 \cdot \Theta(1) + M(n)$	**Average:** $4 \cdot \Theta(1) + M(n)[+\Theta(1)]$	**Worst:** $6\Theta(1) + M(n)$

[18]

3.7.4 *breed*

The *breed* function is probably the most changed function out of all them, as it changes from a simple multiplication and addition to a full on iteration of data structures and creation of new ones. The first thing we must consider is what happens when we actually create a new member m. Consider the construtor code of an m:

```
CMember(){
          rand();
          f = rand(); f = (int)f % 100; f = f/100;
          mf = rand(); mf = (int) mf % 100; mf = mf/100;
}
CMember(double f2, double mf2){
          f=f2; mf=mf2;
}
```

As the reader can see, an overloaded construtor exists that allows us to bypass a random f and *mf*. However, this is not the constructor used when two parents breed. As their f and mf variables are used in the creation of the new child, §3.5's formulas are used and then the child is created. The default constructor is included so that the complexity of creating the parents of our $\mathcal{C}$'s $\bar{m}$ can be examined in future sections. The creation of the parameters $f2$ and $mf2$ is our main focus, and as such, we consider another snippet of code:

```
double mfx(double m2){
      return (1/exp(mf+m2));
}
double computeDF(double f2,double mf2){
     return (f*mf +  f2*mf2)/(2*mfx(mf2));
}

void breed(CMember &p2, CMember &child){
      double df = computeDF(p2.f,p2.mf);
     double sf = (f + p2.f)/2;
     child = CMember(sf+df,df);
}
```

[16]although because the most efficient move is normally diagonal until we are aligned with $\mathcal{G}$, it actually occurs less frequently then this

[17]except for the case where we are at $\mathcal{G}$, but as stated we do not consider that move and thus, for our purposes the number of multiplications *is* consistent.

[18]The bracketed $\Theta(1)$ is from the exception being considered. As it does occur as part of the average runtimes, we include it in that scope. If this does not make sense, see §2.4 diagram 1 and the mapping of probabilities listed after it. The exception corresponds to the middle of the bottom row, and thus occurs when the number of comparisons (pre-multiplication) is three, as do the average cases. Since it falls into that space, we include it's possible addition of a comparison to the average runtime of *assignMembers*.

Now the complexity becomes much clearer. To compute the change in force, we must do three multiplications, one division and one addition, as well as the complexity of the function *mfx*. The complexity of *mfx* is $E(n) + M(n) + \Theta(1)$, where $E(n)$ is the complexity of raising the mathematical constant e to a power.[19] *computeDF*'s complexity is therefore $4 \cdot M(n) + \Theta(1) + mfx$. Substituting for mfx, we have $5 \cdot M(n) + E(n) + 2 \cdot \Theta(1)$. Returning to the breed function, we must still add this change in force to average of the parent's forces. And therefore our total complexity for a single m's breed function is $6 \cdot M(n) + E(n) + 4 \cdot \Theta(1)$, we denote this running time as $\mathcal{R}$

Using *iterative cross breeding*[20], we breed members of $\bar{m}$ with each other. Therefore, the total number of times we suffer $\mathcal{R}$, will be *numMembers* divided by 2 (ignoring any remainder). In addition to this, we must update *avgForce*,*K*,*numMembers*, and our two bounds. The recalculation of the average and confidence are only done once everytime $\mathcal{C}$'s *breed* function is called. However the total force is calculated as we add members, this saves us having to iterate through the members again after we've bred them. The number of times we will add the previous value of *totForce* to the new $m.f$ will be the same amount of times we iterate while breeding. Also, updating a lower and upper bound requres us to check our new members force is greater than the *uBound* or if it's lower than *lBound*. This adds at worst, two more comparisons, and at best a single comparison. This is the case because we use an if else if statement when checking. This means that if we do encounter a high bound we will not check to see if it is lower than the lower bound because obviously it is not. However, this does complicate the average runtime analysis. Finding the probability that the upper bound will be reset is difficult to say the least. Therefore, we turn to empirical testing results, these show that as the size of the population increases, the number of bounds is more likely. This agrees with what we are all thinking, because as we breed more, the expected result is to breed stronger and stronger members. For populations under a size of twenty, it is likely that the bound will be called once every breeding session. Inbetween fourty and twenty, two calls were made on average, and after fourty, three calls. considering that a population of size fourty will push most $\mathcal{B}$ considered in our problem space, we can confidently say that bound will only be adjusted at most three times. And therefore we can state our runtimes as the addition of our iteration variables[21] plus the addition of $\mathcal{R}$ and the likelihood of reassigning a bound and then the updating of averages and confidence. Therefore:

Best:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 4 \cdot \Theta(1)) + 2 \cdot M(n) + 4 \cdot \Theta(1)$
Average:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 5 \cdot \Theta(1)) + 2 \cdot M(n) + 4 \cdot \Theta(1)$
Worst:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 5 \cdot \Theta(1)) + 2 \cdot M(n) + 4 \cdot \Theta(1)$

[22]

For clearity, the best case involving the bound, is when the *upperBound* is calculated. Because of the if else if construct, if we do fall into our first case, we avoid having to make another comparison in the else section of the first check. This is why in the worst case we will make the same number of comparisons as the average case.

3.8 Runtime Analysis of the Updated *action* Function

The analysis of *action* is the sum of it's parts. And therefore, in order to analize the function, let us consider it's pseudo-code:

[19] This complexity is $\mathbf{O}(kM(n))$, where k is a k-bit exponent.

[20] refered to as *dumb breeding* previously and within the code

[21] There are three, two to keep track of where the parents are stored and one to to keep track of where the child is to be stored

[22] The additional $\Theta(1)$ added to the constant runtime within breed comes from initalizing a counter variable as *numMembers*-1

```
if(checkForce(b)){
        determineMove(gx,gy,b,oTree);
       assignMembers(b);
}else{
       breed();
}
```

As we see, *checkForce*,*determineMove*, and *assignMembers* are called only once. *breed* has the chance to be called in two places. Once if $\mathcal{C}$ confidently knows that it needs more members, once if it's confidence was close but it over estimated itself and failed to assign it's members. We can therefore state the best and worst runtime of the algorithm as:

Runtime of Algorithm after Breeding is Complete

Base time of $89\Theta(1) + 2 \cdot M(n) +$	
Best:	$O(log\ \wr\mathcal{O}\wr) + 16 \cdot \Theta(1)$
Average:	$O(3 \cdot log\ \wr\mathcal{O}\wr) + \frac{113}{2} \cdot \Theta(1) + 9 \cdot M(n)[+\Theta(1)]$
Worst:	$O(8 \cdot log\ \wr\mathcal{O}\wr) + 119 \cdot \Theta(1) + 19 \cdot M(n)$

RunTime of Action if Breeding Occurs

Base time of $3 \cdot M(n) + 5\Theta(1) +$	
Best:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 4 \cdot \Theta(1))$
Average:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 5 \cdot \Theta(1))$
Worst:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 5 \cdot \Theta(1))$

As Expected, the only additional complexity of the model stems from storing and searching obstacles, and the additional look to check the obstacles against the moves themselves.

3.9 Running The Program

There are several ways to run the program, one is build the project after hard coding parameters into the program, the other is to use a command line prompt. The arguments and their style are in the same style as Cluster Basic's. In general, the amount of steps to run the simulation for is the last argument, except for when a tree is provided from a file, when debugging information is turned on, or when individual member data is printed at the end of the run. The table on the next page shows the overloaded command line calls:

Input	Output
"Cluster with Obsts".exe	$\mathcal{B}$ with *bstate* | 1 | 1 | 0 | 0 | 2.0 | $\mathcal{G}$ of $(3, 3)$ simulation step counter of 10 information about $\bar{m}$ not being display at the end of the simulation.
"Cluster with obsts".exe 2 3 15	$\mathcal{G}$ of $(2, 3)$ simulation step counter of 15 default values for the rest
"Cluster with obsts".exe 2 3 15 "tree.txt"	an obstacle tree is created from the file provided other variables same as above
"Cluster with obsts".exe 2 3 0 0 15	$\mathcal{G}$ of $(2, 3)$ *bstate* of | 0 | 0 | 0 | 0 | 2.0 | simulation step counter of 15 defaults for the rest
"Cluster with obsts".exe 2 3 0 0 3.0 15	same as above except *bstate* of | 0 | 0 | 0 | 0 | 3.0 |
"Cluster with obsts".exe 2 3 0 0 3.0 15 "tree.txt"	same as above but an obstacle tree is created from the file provided
"Cluster with obsts".exe 2 3 0 0 3.0 15 "tree.txt" 1	same as above except if the last parameter is a one then debug information will be printed out as well
"Cluster with obsts".exe 2 3 0 0 3.0 15 "tree.txt" 1 1	same as above except if last parameter is anything but a 0 information about $\bar{m}$ will be printed out once the simulation steps have been run

3.10 Limitations Of The Program

Because of the greedy style of move selection by which $\mathcal{C}$ moves, mazes that give an almost perfect route to a route by one straight line, but end in a dead end will cause the program to get as close as it can, and then stop moving. This is because no backtracking is allowed within the movement of $\mathcal{C}$. This could be remedied by keeping track of all the moves done by the program so far, and then back tracking sufficiently far enough and then trying a new route. However, numerous difficulties arise on choosing when to decide on a new route, should the program go back a single step and attempt the second best movement possible? If one thinks about how such a program would behave, it would certainly look patheticly similar to a toy car running into a wall at an angle, moving slowly and roughly against it's barriers before perhaps finding an openning to continue. Or perhaps a corner in which to become permanently stuck in. Corner-like obstacle shapes also tend to cause trouble within the program because, almost always, the move to go up and around an obstacle structure is less desirable distance wise than $\mathcal{C}$'s current location. A remedy would be to remove the previous move from the allowable moves, but then how would backtracking be able to be implemented? Not to mention the additional back and forth motion that would occur when $\mathcal{B}$ has been pushed to $\mathcal{G}$.

4 Cluster Goal Driven

4.1 Specifications

Cluster Goal Driven is an extension of Cluster with Obstacles, in this model the goal is not only to manuever to a point, but either to another point or back to the location that $\mathcal{C}$ started at. How this is handled is described in §4.4. The changes to $\bar{m}$ are described fully in §4.3. As it was previously, $\mathcal{C}$ still pushes a box $\mathcal{B}$ of some mass to each $\mathcal{G} \in \bar{\mathcal{G}}$.

4.2 Primary Changes from Cluster with Obstacles

As stated in the abstract, the primary change between Cluster Goal Driven and the previous models is the addition of multiple goals $\bar{\mathcal{G}}$. The population of $\mathcal{C}$ is also modified to have a finite lifespan. Because of this, numerous control features must be changed within the *action* function. The changes can be seen abstractly as: Checking to see if the population's lifespan is low, checking to see if any $m \in \bar{m}$ are dead and need to be removed, and checking to see if a new goal point is needed (as the previous goal has been reached). Also, the goals and obstacles are stored within txt files that can be read by the program, see §4.6 for formatting files to be read and on calling the program with the correct arguments.

4.3 Changes to $\bar{m}$

As stated, the modification of $\bar{m}$ to have finite lifespan increases the behavior of the program from, *breed* until the force is enough to push $\mathcal{B}$, to *breed* if the population's in danger of extinction from 'old age' or if the force is not enough to push. Because $m \in \bar{m}$ will die during the journey from some point to $\mathcal{G}$, the population will stop and breed as needed. To accomodate this new feature, we introduce two new variables *tl* and *life*. *life* is a dynamic value that changes over time and is what is ultimately used to decide if a m has died or not, and *tl* is the original value of *life* when the m was created, the equation governing *tl* is described below:

$$tl = \frac{10}{mf} + \frac{10}{f}$$

Translated into layman terms, the lifespan of a member is it's stability plus how much energy it uses. The second fraction, as one can see, if f is high the additional lifespan will be lower. This makes sense, as the f value is how much energy the m is applying to $\mathcal{G}$. An interesting exercise would be if the recipricol fractions were taken instead. The fraction governed by mf contributes the amount of lifespan affected by the mutagen value, if an m is highly unstable (has a high mf) the additional lifespan shall be low, keeping with the trend that the unstable organism is more likely to die. The numerator is an arbituary choice, and could be tweaked to allow for smaller or greater lifespans for all members.

4.4 Dealing with multiple $\mathcal{G}$

As defined, any $\mathcal{G}$ is a point located in $\mathcal{W}$ which $\mathcal{C}$ shall push $\mathcal{B}$ to. Because of this simplified idea of a goal, the best way to prioritize from goal to goal is to use their distance as the primary choosing mechanism. Therefore, when $\mathcal{C}$ recieves it's list of goals $\bar{\mathcal{G}}$ it uses a *calcWorth* function, and then stores them into a priority queue with respect to the value returned from the mentioned function. After all goals have been stored in this fashion, the *home* of $\mathcal{C}$ is added. *Home* is a simple tuple, much like all spaces in *mathcalW* can be represented, this also normally represents the starting point of $\mathcal{B}$, or at least the starting point of $\mathcal{C}$ though this does not neccesarily have to be the same.

4.5 Analysis of Cluster Goal Driven

4.5.1 Overview

The addition of a priority queue itself does not add much complexity besides the actual data structure. Building the object itself takes linear time with respect to the number of $\mathcal{G}$ to add, and this is added only once so the overall time is negligable. Accessing the queue itself takes no time at all. The change to implement the death of m is the most notable. As it does add a significant degree of complexity to update the numbers keeping track of the population. The upper and lower bounds for both lifespan and force must be updated when a member who is responsible for these values dies. This takes linear time with respect to the number of members. Also, because each member must be updated to decrement their lifespan variable every step of the simulation this also adds complexity. In Cluster with Obstacles, the breeding and movement phase were seperate and you could count on the breeding to occur first, and then the movement, however in this model the two are entangled.

Beginning with the simple, the *action* function has been modified as follows:

```
if there are too few members
    breed
else
    if there is enough force to push the box
        determineGoal()
        determineMove()
        assignMembers()
    else
        breed();
check deaths and update lifespans
```

as one can immediately notice, there are a few more checks and function calls then the previous implementations. A function now also measures how low the lifespan of $\mathcal{C}$ and decides whether to breed or not based on that. Also, instead of just determining a goal and assigning members, we also must now check to see if we are at our goal point and whether or not to grab the next goal from the queue. At the very end of the function, regardless of what we did, we check the deaths and update the lifespans $\forall m \in \bar{m}$.

4.5.2 Base Time

As we've done in previous sections, we can account for a base constant time that is then modified by some variable time. Unfortunately, because of the entanglement of breeding and movement, this time is not as nearly consistant as it was before. At the beginning of action we can count on a call to *checkLife* which takes $\Theta(1)$ time. However, beside the call to check deaths, this is the only constant time operation we can count on. So, let us move into the *death* function and extrapolate the constant time from this. We can break down the *death* function into the following pseudo code:

```
for all members
      if update returns true //member has died
            numMembers--
            subtract tl of member from total life, and force from total force
            if member was the lower bound to force
                  if member was a bound to lifespan
                        iterate through members and find new bounds for both
```

```
                    else
                         iterate through members and find new bound for force
               if member was an upper bound to force
                     same as lower force
               if member was a lower bound to life
                     iterate through members and find new bounds for life
               if member was an upper bound for life
                     same as lower bound
             otherwise
                     remove the member
recalcuate averages and confidence
```

The most constant time we can squeeze out from this is that we must iterate through all members and check to see if their lifespan has run up or not. The update function takes $2\Theta(1)$ time due to it's subtract to the lifespan and test on the new lifespace for negativity. Therefore we can add in $\bar{m} \cdot 2\Theta(1)$ to our constant time. Unfortunately, this concludes the base time and we must now turn to the difficulties of the varying complexity.

4.5.3 Variability of the *death* function

Let us begin by the assertion of some simple facts. No member can hold all four bounds. That is, no member can be both the upper and lower bound for force or lifespan. Why is this? Because the first check within *action* keeps a certain threshold of members alive no matter what. In our implementation, this number is three. However, a member can be both a bound for both force and lifespan. This is why the additional checks within the *death* pseudo code exist in the first two checks. The choice to check this within the same if statement is one that will be explained shortly. The "best" case for checks is when a member is a lower bound to force and an upper bound to lifespan as this results in only two checks. However, this is only the best in terms of checks. In order to update the force and lifespan variables, if a single member holds two bounds then we therefore must find not one, but two new numbers. This means that when we iterate through our members, our search must be done for both variables. This is the reason why we include the checks for life inside the checks for force as well as outside. By searching within that space we save ourselves iterating twice through $\bar{m}$. The difference is staggering when considering large population times. During a runtime with over two million members, the updating and death process took at most ten seconds (an approximation by the author, not rigourously tested). Therefore, the best case in checks is not the best case running time for the death function. As one might suspect, the best case arises when a member is not a bound of any sort. While we do perform all four checks against bounds, the overall result is only to erase the member from the population vector. When a member is a bound we must iterate through $\bar{m}$ and depending on whether a member is a double bound or single, this running time is either $(\bar{m} - 1) \cdot 4\Theta(1)$ or $(\bar{m} - 1) \cdot 3\Theta(1)$, plus whatever checks were performed to reach the iteration step. These checks may range between two to four. This is a result of simple deduction on the if statements. The worst cases iteratively adds either two checks (upper force bound and lower life bound), three checks (upper force bound and upper life bound, lower force bound and upper life bound) or four checks (lower force bound and lower life bound), the better single bound finding iteration step occurs after either three or four checks. The single bound finding is a result of else statements on the end of the inner if's within the force bound checks, or as the lifespan bound checks after the force bound checks. While describing these cases in text is slightly confusing, a simple look at the pseudo code and looking for the cases described is intuitive and is recommended by the author.

The worst case occurs when the unlikely case of a member who is an upper force bound is also the upper life bound. I say unlikely because the very equations governing these two variables contrast in such a way that a high force member is more likely to die sooner on account of their (likely) high mutation rate. This is why the checks within the program check for the more likely cases first. This worst case scenario takes $3\Theta(1) + (\bar{m} - 1) \cdot 4\Theta(1)$. Because of how unlikely this is to occur, we can consider our more average case worst case as our worst case[23]

As our members are dying, and are always going to die, it stands to reason that eventually our lower lifespan bound member will die, in fact they will die quite often[24]. This simple intuitive notion leads us to our more likely worst case, $3\Theta(1) + (\bar{m} - 1) \cdot 3\Theta(1)$. While this only differs by $\bar{m} - 1$, as $\bar{m}$ grows larger, this can be considerable. At this point, the reader may have not figured out why we use $\bar{m} - 1$ instead of $\bar{m}$, because we are deleting a member, we can ignore any checks that would have to do with it. We do this by rotating the member to be deleted to either the front or the back of the array and then adjusting our loops accordingly. That aside, this average case worst case scenario is mainly presented as a thought exercise and for interested individuals. The worst time considered in the previous paragraph is the worst time we shall factor into our total analysis.

On average, as $\bar{m}$ grows during the runtime, the worst case will crop up less and less, only occuring when the considerably long lifespans of population members have run out. Since unstable members die more quickly, the population will slowly level out to be considerably large but long lived. This pushs our average case to be the best case considered above, four checks per member who has died. Of course, these best, average and worst cases are multiplied by the number of deaths occuring. Because of this, the worst case imaginable would be a mass extinction where the runtime would be $\bar{m} \cdot (3\Theta(1) + (\bar{m} - 1) \cdot 4\Theta(1))$, or $3\bar{m}\Theta(1) + 4\bar{m}^2 - 4\bar{m}$ which we could simplify to $\bar{m}^2$ complexity. This of course would occur if, starting at the first member, that member was the lower force bound, and the lower life bound (which would make sense), we would iterate and update our numbers and remove this member, moving on to the next, it just so happens that this member is the same case as the previous, and so on and so on. In otherwords, if the members were to be ordered by both force and lifespan, this worst case scenario could occur far too frequently for our taste.[25]

This pessimistic scenario is extremely unlikely to happen, and is in part, prevented from doing so by the *checkLife* function which breeds members whenever the population dips down drastically to critically low numbers. At best, no members die within the execution of the *death* function and the complexity is simply $\bar{m} \cdot 2\Theta(1) + 4M(n) + 6\Theta(1)$.[26] On average, the runtime depends on how many members died (which is normally a fairly low number[27]) and also runs into the case where most members are not bounds. Therefore, we state our average runtime to be $\bar{m} \cdot 2\Theta(1) + 4M(n) + 6\Theta(1) + \bar{m}_{dead} \cdot 4\Theta(1)$. And our worst case is of course, the mass extinction scenario of quadratic complexity plus $4M(n) + 6\Theta(1)$ for the calls to *calcAvg* and *calcConfidence*.

4.5.4 determineGoal

Death and *checkLife* aside, the only new function added is *determineGoal*, which has a constant runtime of $2\Theta(1)$. As one might expect, there is a check to see if the goal and the previous move were the same. Why the previous move? Because we expect that the goal

[23]Clarification: This sentence simply means that our worst case does not occur very often, if ever. And therefore we consider a worst case with a higher probability of occurence to be our worst case on average.

[24]relatively speaking of course, and moreso during the beginning of the programs runtime than near the end.

[25]The worst case complexity shown in this paragraph does not add in the calls to *calcAvg* or *calcConfidence*

[26]Update checks for each member and the call to *calcAvg* and *calcConfidence*

[27]observed empirically to be around 3 or 4

point should be visited at least once. If the goal and next move were checked, then we would get one step away from our goal, and then suddenly veer away toward the next. If there are no goals within the priority queue then we simply assign our goal to be the home point of $\mathcal{C}$. This keeps the program from crashing once there are no new goals.

4.5.5 Overall Analysis of *action*

Although the function is no longer as deterministic as before, we can still count on a few things. As mentioned before, we can count on *checkLife* to execute each time *action* is called. Since the population is always initialized with two members and *checkLife* will call *breed* if the population is below four members, we can count on *breed* being called two times at the beginning of our function. Therefore for the first two steps of the simulation we can count on the runtime described in S3.8 for breeding plus $2\Theta(1)$ plus $2 \cdot (\bar{m} \cdot 2\Theta(1) + 4M(n) + 6\Theta(1))$ (Checks within *action* and two calls to *death*). We can state the best runtime for death in this case because there is no chance that the two initial members will die right away.[28]

After this initial burst of activity, we must turn towards our trends and our intuition. Because members with high unstability have shorter lifespans, it is likely that these members will die off, leaving a stable, long lived, albeit weaker population. However, the magnitude of the population size guarantees being able to accomplish goals. We therefore trade a strong small population for a larger, more stable collective group. Thus, while the beginning runtimes of the simulation may fall into the breeding areas, we tend towards the segment of code that calls *determineGoal*, *determineMove*, and *assignMembers*. This chunk of code has a runtime of:

Base time of $M(n) + 5\Theta(1)$ +		
$89\Theta(1) + 2 \cdot M(n)$ +		
Best	Average	Worst
$O(log\ \wr\mathcal{O}\wr) + 16\Theta(1)$	$O(3 \cdot log\ \wr\mathcal{O}\wr) + \frac{113}{2}\Theta(1) + 9M(n)$	$8O(8 \cdot log\ \wr\mathcal{O}\wr) + 119\Theta(1)$+18 M(n)
+ *death*		

Besides this case, the only other case are when breeding occurs. This can occur at two points within *action*, either after the single *checkLife* check, or after *checkForce*. Thus the base time for breeding described in §3.8 remains the same except for an additional $\Theta(1)$ or $2\Theta(1)$ plus the time for *death* to occur. This modified table is printed below:

RunTime of Action if Breeding Occurs

Base time of $3 \cdot M(n) + 6\Theta(1)$ or $3 \cdot M(n) + 7\Theta(1)$ +	
Best:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 4 \cdot \Theta(1))$
Average:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 5 \cdot \Theta(1))$
Worst:	$\frac{\bar{m}}{2} \cdot (\mathcal{R} + 5 \cdot \Theta(1))$
+ *death*	

As is immediately noticable on cross reference, the overall time has only changed becase of *death*. While *determineGoal* and sthe other functions added add a small degree of complexity, they are all significantly dwarfed by *death*'s possibly large running time. Also, we see that as the Cluster program increases in complexity, the parameters affecting the runtime increase as well. Within Cluster Basic, the variabilty was a simple problem of variability inherent in *determineMove*. When obstacles were added, the parameters increased to include the number of obstacles being delt with, however due to our choice of data structure and organization of the obstacles this variability is, in practice, very slight. Within this version the main source of complexity is dealing with updating variables affected by the death of any m that happens to be a bound to *force* or *lifespan*.

[28] We can state this because of the equations governing lifespan and that the random number seeds for the populations initial members are between zero and one

4.6 Limitations of the program and File Reader

The limitations described in 3.10 still hold for this version of Cluster. And to restate them would be redudant, however a brief overview never hurt anyone. As stated previously, because of the greedy nature that manhatten distance movement calculations impart to the program, it is very possible for $\mathcal{C}$ to get stuck in corners or against long walls. This occurs because the current move is computed as more desirable because the only heuristic affecting movement choice is one of distance. While this is a downfall of the program, it is done for simplicity. Cluster is not created with maze traversal in mind, graph theory as well as Dijkstra's algorithm and the variations thereof are ample solutions to such problems and Cluster is not meant to compete with them. Cluster is a simple model of some type of simple swarm organism that is controlled by a primary directive.

The file reader header file included in the project includes three simple functions. A string to number function, aptly called *StringToNumber*, a function to read obstacle files *readFileForObsts*, and a function to read goals from a file *readFileForGoals*. The key reason for seperating these functions is because *readFileForObsts* returns a *tree* pointer, and as one might expect *readFileForGoals* returns a *vector* of type *Tuple*. Both file types adhere strictly to the following format:

;first,second;sfirst,second;s ... sfirst,second;se

The file begins with a semicolon, then the tuple desired is entered, with it's first and second arguments seperated by a comma, a semicolon ends the tuple and then the s character tells the function to store the two numbers entered into a tuple, then the next tuple is read. It is important to notice that there is no semicolon between the s character and the next tuple's first argument. Once the last tuple has been entered, the e character signifies the end of the file. **If the e is not included at the end of the file, the file reader function will crash and terminate the program.**The format for goal files is the same, for example if you wanted to load the goals $(1, 1), (20, 523)$ your file would read:

;1,1;s20,523;se

as one can see, the format is followed as described, with no spaces, no semicolon between the s character and any numbers, and the file ends with e. If the user experiences something along the lines of a single goal or obstacle being created when they've entered many more, it is recommended to check to make sure the comma's and semicolons are included in the proper places as well as the s character. It is the s character that signifies the creation of a tuple of the two previous numbers, and if this is not included the described error may occur.

4.7 Running the Program

The arguments passable to the program via command line are listed below, default values for $\mathcal{B}$ are a starting location of $(1, 1)$ and a mass of 2. Default values for goals is to load from the default text file, the default value for obstacles are loaded from the default file path. These file paths may be changed as a user wishes, though rebuilding the program will be neccesary.

Arguments	Effect
"goals.txt" 15	will Load goals from goals.txt set simulation step to 15 default values for $\mathcal{B}$ and obstacles $\bar{\mathcal{O}}$ are loaded.
"goals.txt" "obst.txt" 15	same as above except obstacles are loaded from obst.txt
"goals.txt" 2 3 15	goals loaded from goals.txt $\mathcal{B}$ start point is $(2, 3)$, step value 15 defaults for the rest
"goals.txt" 2 3 2.0 15	same as above except the mass of $\mathcal{B}$ is set to 2.0
"goals.txt" 2 3 2.0 15 "obst.txt"	same as above except obstacles are read from obst.txt
"goals.txt" 2 3 2.0 15 "obst.txt" 1	same as above except debug output is displayed
"goals.txt" 2 3 2.0 15 "obst.txt" 1 1	same as above except member information is displayed at the end of the program

5 Final Thoughts

The Cluster project is not a perfect model of any sort of emergent behavior. Despite the use of random numbers, with the given seeds to the program a human computer could follow the same process given the set of rules that $\mathcal{C}$ follows. The way that $\mathcal{C}$ can become stuck in simple areas where a child could manuever around is proof of it's imperfection. The use of manhatten distance and greedy path finding in it's movement protocols accounts for this. In a more sophisticated model, $\mathcal{C}$ could scan it's environment for hazards, build a simple map, recognize obstacles, and then compute a path around them, $\mathcal{C}$ could then be made "dumber" by lowering it's look ahead value, and smarter but more timeconsuming as that value increased.

The way the population evens out is a simple show of evolutionary progress. We cannot however, attribute this evolution to the program itself or to $\mathcal{C}$, as it is dictated by the equations created to govern the species life and death rates. The fact that members are bred in favor of lifespan instead of strength is a result of this. In the first two models, members are bred for strength, and they do an excellent job of creating them. But given the choice between life or death parellel to weakness or strength, the focus is switched to long lasting members. The reason for this, though we want strength, is because members that die too often will result in $\mathcal{C}$ continually breeding and never progressing with it's goals. Just as human's take chances everyday that shorten their lives, so does the Cluster.

This is not an attempt to create any form of intelligent behavior in the form of machinery. Afterall, the computer only knows that it is crunching numbers and spitting out a result, and we can not mistake this for true semantic based intelligence. A simple microbe is more intelligent than $\mathcal{C}$ for a very simple reason. The Cluster model has no form of feedback. The only reason the members strive towards a goal of any kind is because that is the goal assigned to them. While the equations governing decision processes are created with a nod towards biology, they do not fully render any sort of cognitive function to the program, they merely set a threshold for a decision to be made. The threshold does not stay fixed, and for this I attriubte $\mathcal{C}$ with some small amount of decision making process, because the chosen number is affected by the environment of $\mathcal{C}$, namely it's members and their state. In closing, Cluster AI is a simple program that displays very simple path finding and decision making processes based upon a human selected goal.