Smart Terrain Using Multiple Needs

By

James O’Rell

Submitted in Partial Fulfillment of the Requirements
for the Degree of
Master of Computing and Information Systems

Youngstown State University
May, 2012
Smart Terrain Using Multiple Needs

James O’Rell

I hereby release this thesis to the public. I understand that this thesis will be made available from the OhioLINK ETD Center and the Maag Library Circulation Desk for public access. I also authorize the University or other individuals to make copies of this thesis as needed for scholarly research.

Signature:

James O’Rell, Student
Date

Approvals:

Dr. John Sullins, Thesis Advisor
Date

Dr. Bonita Sharif, Committee Member
Date

Suzan Harper, Committee Member
Date

Peter J. Kasvinsky, Dean of School of Graduate Studies & Research
Date
Abstract

Gaming artificial intelligence must look intelligent, real intelligence is a near impossible goal to achieve because of the number of CPU cycles it requires. Using Multiple Need Smart Terrain, AI can look intelligent without requiring the large number of CPU cycles it requires for such intelligence. It can be used to manage a character’s needs and direct them to which objectives are most profitable for them. Unlike normal smart terrain that only pays attention to one need at a time, this algorithm can look at any number of needs and check which objective would be best to meet that need. This way, the AI can have little actual intelligence, the terrain tells it where to go. With this technology, AI can appear intelligent and keep the cycles required to a minimum.
Table of Contents

Introduction .................................................................................................................................................. 1

Research and Past Works .......................................................................................................................... 2
Games ....................................................................................................................................................... 4
Planning .................................................................................................................................................... 7
The Multiple Need Algorithm ....................................................................................................................... 8
The Testing Environment .......................................................................................................................... 8
Algorithm Parameters............................................................................................................................... 9
  Maximum Signal Strength ..................................................................................................................... 9
  Degrading Need Modifier ................................................................................................................... 10
  Signal Propagation .............................................................................................................................. 11
Empirical Testing ......................................................................................................................................... 12
Parameters Used For Testing .................................................................................................................. 12
Scenario 1: Identical Parameter Values .................................................................................................. 13
  Scenario 1a.......................................................................................................................................... 14
  Scenario 1b.......................................................................................................................................... 14
Scenario 2: Need vs. Proximity ............................................................................................................... 15
Scenario 3: Identical Options .................................................................................................................. 16
Scenario 4: Opportunity.......................................................................................................................... 16
Scenario 5: Introducing Degrading Needs .............................................................................................. 17
  Scenario 5a.......................................................................................................................................... 18
Scenario 6: Need Degradation Rates ...................................................................................................... 18
  Scenario 6a.......................................................................................................................................... 20
Interpreting the Results .............................................................................................................................. 21
Strengths ................................................................................................................................................. 21
  Opportunistic Behavior ....................................................................................................................... 21
  Versatility ............................................................................................................................................ 21
  Customization ..................................................................................................................................... 21
  Scalability ............................................................................................................................................ 22
Weaknesses ............................................................................................................................................ 23
  Pathing Around Objects .................................................................................................................... 23
  Scalability ............................................................................................................................................ 24
Introduction

Smart Terrain is a gaming artificial intelligence (AI) concept where the environment guides the AI to its objectives. The concept is that objects that can meet a specific character need radiate signals across the map and the character then follows the signal to the desired item. These mappings are usually created on a grid, filled with values. The higher the value, the closer the character is to the item it wants.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NeedA</td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>13</td>
<td>13</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>12</td>
<td>12</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 1: An example of radiated need map. Need A is the objective radiating the signal.

Many games make use of this Smart Terrain concept, including games like The Sims and S.T.A.L.K.E.R. In these games, the AI is not just a bunch of scripted actions, it’s important to the core of the game play mechanics. Although, if done correctly, advanced AI like Smart Terrain can complement scripting in traditional game play situations, like those found in action and adventure games.

This idea works well with one type of need that the non-player character (NPC) wants to sate. Yet, what if our NPC has multiple needs that it must fulfill? How do we handle such a task on one grid? That is where my concept of Multiple Need Smart Terrain comes in. With this method, we can have a NPC with multiple needs and multiple objectives that meet these needs on the same grid. This way, the
NPC character can truly be ‘dumb’. It doesn’t need to even know what objective it is heading for. It just travels where the signal is strongest.

This smart terrain method is more than just moving to the need objective with the highest amount of urgency. It is a system that weighs both the AI’s distance from the objectives and the AI’s need for the objectives and places them on all the same radiated map. This way the AI can make informed choices about objects it needs and what objects it can pass up, something that current AI methods handle is a far more simplistic manner by checking which object is in need the most.

At the end of the day, using multiple needs on one smart terrain grid is a form of planning. The grid itself is one massive plan of possible routes. The great part about this type of planning is that the AI unit is completely uninvolved with its creation, with the exception of the need levels derived from the unit itself. It can remain blissfully unaware of the future and be guided by the terrain itself. The world truly creates its destiny. Since the AI unit operates in real time and the plan it follows doesn’t have to, we can have an AI character that appears intelligent without actually without using large amounts of CPU cycles to make it happen.

**Research and Past Works**

The idea of creating smart terrain derives from the idea of creating ‘stupid’ agents, but intelligent environments. The concept behind this is an intelligent agent may take up a very large amount of CPU cycles, but a stupid agent making decisions based on a ‘smart’ environment, takes up far less cycles because the environmental layout is mostly pre-computed (Doyle).

**Annotation**
Not only does this concept lead to ‘smarter’ and more efficient agents, but it also leads to more believable ones. The idea was introduced by Peter Doyle and his symposium on “annotated environments”. His concept is to create such an environment that has five different types of annotations: emotional, response, problems-solving, role and game playing annotations. Using these annotations allows our character to react differently to each environmental piece and, most importantly, react believably (Doyle).

For example, we could have an NPC farmer who lives in a farm house that catches on fire. In the house, we have a picture of the farmer’s dearly departed wife. This picture is annotated with a high value emotional annotation. The farmer, living in his house his whole life, knows the annotations within the house without being in the environment and observes the high emotional annotation. This causes the NPC to ignore any need for self-preservation and head into the house after the picture (Doyle).

The best part about this setup is that we do not need to even script the action of the farmer. The object is so valuable to the NPC they will act that way accordingly because the environment (and the object is part of that environment) tells them to. The emotional annotation will over-power the other annotations in the environment, including that of the fire that could potentially kill the NPC.

Using this concept of annotation, we develop the idea of smart terrain. Our terrain has pre-established, but possibly dynamic values that dictate our NPC’s decisions. In one model (Sullins), smart terrain uses values dictated by a character’s needs. The character has a need it must prefill. An object radiates a signal, which the NPC follows to an object that meets their needs.

**Dynamic Probabilistic Smart Terrain**

We run into the issue of how does our NPC know if an object can meet their needs? For example, there are multiple refrigerators in an environment and it is hungry. We use probability to
alter our smart terrain signals. This way, our NPC has the best chance of finding the object that could most likely meet their needs (Sullins).

Let’s go back to the refrigerator example. Our NPC knows there is a 70% chance there is food in Refrigerator A and a 50% chance in Refrigerator B. The signals from A will be stronger, but depending on the NPC’s position, it might be more profitable to check B first in some cases. In other words, there may be parts of the grid where B’s signal is stronger, especially closer to where the B refrigerator is located.

**Games**

**The Sims**

The smart terrain concept was originally made popular by the game The Sims and its sequels. The Sims is a game with artificial intelligence at its very core. In this game, the player creates a virtual person who lives in an environment also created by the player. The player guides this person, also known as a Sim, through their day to day life, including interactions with other Sims. The objective of the game is really whatever the player wanted to do. If they enjoyed making their Sim happy, they could do so. If they enjoy putting their Sims in terrible situations and watch them try to cope, they can do that too.
Figure 2: This Sim’s friend is passed out on the ground, her stove is on fire, the house is a mess and the pizza guy is at the door. Which need is most urgent?

This is where smart terrain gets its concepts of needs and objectives. Each Sim has a list of needs they want to fulfill (AI Game Programmers Guild). They take their most urgent need and generally worry about that need until it’s met. It then moves on to the next most urgent need and such.

The Sims, at least in the original version of the game, has never been very good at maintaining their needs before they would get to an urgent state. This is why they would need a player to guide them, sometimes to the point of holding their hands to every small decision the character would need to make. If left to their own devices, they would walk long distances for items they need, while passing up an item they need almost as urgently.

For example, imagine if your Sim is both hungry and has to go to the bathroom, but slightly more hungry. The Sim is currently sitting on the couch, watching television. There is a bathroom to the
left of the couch. In order to get something to eat, the Sim would have to pass through the dining room and into the kitchen. In this situation, the Sim would look at its need list. It would realize that its most dire need is hunger and would get up, move through the dining room and to the kitchen, where it would prepare to snack from the refrigerator. Now, all the while, their need to the bathroom is getting worse. If the Sim takes too long, it could have an accident. It would have been far more productive to go the bathroom first and then handle the hunger issue.

As demonstrated with the example above, the big problem with this system is that they could only focus on one need at a time. The Sims AI is simply oblivious to everything else in the environment, except what they happen to need most at the time, showing that they only work with one radiated map at a time. With only focusing on one need at a time, it stops the AI from making efficient decisions and one step farther from being believable.

**S.T.A.L.K.E.R**

S.T.A.L.K.E.R was very much designed with this single-minded AI in mind. Each of its NPC characters would be given a quest and their need would be to complete this quest. The designers bypassed the idea of multiple needs altogether and created non-player characters that would work toward a specific goal.

The added wrinkle in this is that each NPC has all the characters of a player, including an inventory to manage and such. How they implemented this complex behavior is giving NPCs two modes “online and offline modes” (Game.) Online characters are usually NPCs close to the player character. They create “smooth paths” to their objectives, take care of their inventory and take steps to complete their goals. The man behind the AI system, Dmitriy lassenev, described it as a Level of Detail type
design. When the player is close, the characters behave in a more detailed manner i.e. online mode. If the player isn’t around, these NPCs are more likely to be in offline mode, where their actions are less detailed but they still move about the environment (Champandard).

Figure 3: In the radiated world of S.T.A.L.K.E.R, each AI has one quest to finish.

For what it set out to do, the S.T.A.L.K.E.R. AI appears to be far more than adequate at doing its job than the Sims, but that is because the AI character really only exists to complete its objectives, not to prefill it’s every need. This is a good example where having a single need radiated map works well, yet it would be great if the AI worried about more than its quest. It would make it far more believable if the AI took other things into consideration. For example, if they are low on health, they would seek out health before returning to their mission.

Planning

An NPC taking actions to meet multiple goals can be considered a form of planning. The AI could theoretically build a tree of all possible orders in which objectives that could be visited and choose the best one (Millington). The trouble with this is that it would be exponential in terms of the number
of objectives in the worst case scenario. Games typically cannot devote this kind of processing power to AI. Even though, planning is sometimes used in games, but it is restricted to simple STRIPS-type planning to backward chain from a goal to preconditions that can meet that goal (Orkin).

I believe that the Multiple Need Smart Terrain algorithm can provide the illusion of such planning without requiring the processing cycles. If an AI is on its way to a goal and takes it, the AI will have that need met and it will immediately head to the next goal. This appears like the AI planned the move all along, but in reality the values on the grid pointed it every step of the way.

The Multiple Need Algorithm

The concept is to create a single grid that the AI follows, allowing it make ‘correct’ choices about where it should go according to its current needs and the distance from the objective that can help quench that need. To do this, we need to describe the rules that our ‘game’ will operate by.

The Testing Environment

The rules of the game to test this AI are simple. With my sample application, there are two need objectives in the environment. Each need objective has its square grid (in this case, ten by ten), containing the radiated values from the need objective. These values are then multiplied by a need modifier, which is how much the NPC needs the objective.

Regardless of how many need objectives there are, there is always a final, master grid. This grid contains the highest values from all grids. This is the grid the NPC uses to make it decisions. It doesn’t even have to know which objective it is pursing, it just knows that the most valuable path has the highest values, thus is the most efficient and ‘correct’ choice.
The NPC goes about its grid world comparing values from all the spaces surrounding it and goes with the highest value that surrounds it. Even if two needs are tied in a space and has the same value, it matters little to the NPC. That just means that choosing that spot is one step closer to a profitable objective, regardless of which objective that may be.

Once the AI has reached an objective, it decreases the urgency of the need that objective relates to. In the sample application, for simplicity, the need of that object is set to zero. This way the values for that object turn to zero and the AI is free to go pursue the other objective that it passed up previously. In a real application where the need is not completely quenched by one objective, the recently reached objective must stop radiating for the time being so the AI will be free to go after something else.

Now that we have fully described the rules to the game, now there are multiple decisions that need to be made in order to use the algorithm.

Algorithm Parameters

There are a few major choices to make when using the algorithm. The first of which is the maximum strength of the signal. There are two major possibilities for this. For these next examples, the grid will be described as $x^2$. In other words, the $x$ is both the length and the width.

Maximum Signal Strength

When first testing the algorithm, a signal strength of $x$ was used. This left parts of the map with a signal of 0, which not only caused the AI to have nowhere to go in certain situations, but also caused to the AI to go an objective it didn’t need anywhere near as much as another objective on the map.
One way of getting around this would be strength = \( x + (x/2) \). This ensures the signal can propagate through the entire map and not be zero anywhere. This is the method used in the sample application.

Another possibility would be to use a strength equal to \( x \), but the lowest value would be 1, not 0. Again, the aim is to never have a square with a 0 in it.

Another decision to make is what the AI should do if it encounters equal numbers. In other words, if the AI encounters two values on that surround it that are the same, which one should it follow? The sample application AI chooses whatever value it sees first. There are times this leads to odd choices and in rare circumstances incorrect choices, but it works well enough for the sample application.

Another way to do this is to allow the AI to know which way it is going to the goal and make sure it picks the value that is in the right direction to the goal. In the sample application, the AI is a ‘dumb’ AI and allows the grid to tell it where to go, so this is easier said than done.

It is also possible to tell the AI which goal it is going after, so it always chooses the value that leads to the goal it is pursuing. This won’t clear up all the minor mistakes because most of these equal values occur from checking values that come from the same signal.

**Degrading Need Modifier**

Degrading need modifier values is another concept that needs some thought. If you have how much your NPC needs an object degrading in real time, it requires real time updating of the grid. In real games this could be very computationally expensive and not really an option. An option here is to allow values to degrade, but only refresh the grid when the AI reaches an objective. This could still be very expensive depending on how many AIs you have in the environment and how often they reach
objectives. As for the sample application, it was designed from the beginning to update every time the character moves, so having a need degrade was rather easy to implement.

Signal Propagation

The most important decision to make is how the signal propagates. For the sample application, it just checks the distance between a space and the objective that is radiating the signal. This works very well in a world without obstacles (i.e. the sample application) but has problems when dealing with objects in the way. The AI will still function and will still be able to work its way around objects, but it may not do it in the most efficient manner possible.

For this, we measure the distance between points on the Cartesian plane.

**Distance between two points Formula:**

Distance between two points:

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

where \((x_1, y_1)\) \((x_2, y_2)\) are any two points on the Cartesian plane (Easycalculation.com). Please note that the sample program uses a simplified version of this formula, ensuring that no square roots are necessary.

One of the better ways to handle obstacles is using a recursive algorithm for signal propagation. The algorithm checks the available paths around the object and assigns a value. It then calls itself to find the value for the next square. The recursive solution is good for navigating around objects, but if updating the grid in real time, it could require too many cycles to be efficient enough for in-game use.
Empirical Testing

For testing the algorithm, we will be using the sample application featured in Figure 4. The tests will look for how the AI behaves in different scenarios, not the efficiency of the algorithm. The efficiency of the algorithm can really only be judged while being used in coordination with a game engine in real time. These scenarios will test the AI’s appearance of intelligent behavior. These tests are to determine if the algorithm can create this ‘illusion of intelligence’ (Buckland).

Parameters Used For Testing

While evaluating these tests, it must be kept in mind that we are not looking for optimal behavior. In order to create such behavior, we would need far CPU cycles than most games are willing to set aside for artificial intelligence. What is desired here is the appearance of ‘intelligent’ behavior, from the perspective of one who is observing the AI (Buckland).

For simplicity’s sake, the number of objectives has been limited to two. This way we can easily evaluate the AI’s choices. Even with only two choices, we can still test common cases that an in-game AI would encounter. Also, this allows us to come up with simple guidelines for level designers who must work around this AI. It is important that they know the ‘do’s and don’ts’ of creating worlds with Multiple Need Smart Terrain AI agents inhabiting it.

While testing the algorithm with the sample application, there are a few options that need to be configured. There are three objects you must decide the location of on the grid: Need A objective, Need B objective and the AI’s starting location. Once those are decided, you must decide the AI’s starting level of need for each of the objectives. You have the options to have needs degrade, meaning
the need for each of the objectives will increase by a configurable amount every time the AI moves. The default for this amount is 1.

![Image](image_url)

*Figure 4: The sample application, which is built in Visual Studio 10’s Visual Basic.*

**Scenario 1: Identical Parameter Values**

The AI is placed in the middle of two needs, both equal distances away from the AI. The need levels for both objectives are the same and the levels do not degrade. In this case, the AI chooses whatever high value it finds first. Since the AI has no particular need for one object over the other, so it makes sense that would it would simple go after the first one it notices. This does help emphasize the importance of breaking ties. What happened here is the AI found a value belonging to each objective that was the highest value it came across. It simply chose the first value it saw and went along that path. In other words, the AI’s decision was decided by the tie-breaking mechanism, or in this case, the lack thereof.
Figure 5: Scenario 1: Two Need objectives with equal need levels at equal distance.

Scenario 1a

Placement is similar to scenario 1. The main difference is that one of the need objectives has a greater need than the other. Logically, the AI turns and heads straight for the objective that it has the greatest need for. This demonstrates that the AI will go after which ever objective it has the greater need for if distance is not a factor. This is the ‘correct’ behavior for the AI. Since both distances are equal, we want it to make its decision based on its needs alone.

Figure 6: Scenario 1a: Two Need objectives at equal distance with one need level greater than the other.

Scenario 1b

In this scenario, the setup is somewhat similar to scenario 1, with the exception of one of the objectives is one space along the X-axis closer to the AI’s starting position. The need levels are exactly the same. In this example, the AI takes the expected path to the closer objective. What is not expected
is that one of the other need’s values is just as high as the followed path. This means the AI used its tie-breaking behavior to decide where to go. This demonstrates that even though one objective is closer and their need levels are identical, the tie-breaking behavior could judge where the AI goes at times we might not expect it to.

![Figure 7: Scenario 1b: One need objective is slightly closer with the same need level.](image)

**Scenario 2: Need vs. Proximity**

The AI is placed next to one of the objectives but has little need for it. Another objective is as far away as possible and the AI needs it greatly. In this example, the need levels do not degrade. The AI makes the logical decision and moves for the farther away objective. It ignores the one that is placed close to it because it really doesn’t need it at the present time. This is the logical behavior we want out of the AI, since in this world, once the AI finds the object it wants, it consumes that object and the need of it is set to 0. We wouldn’t want the AI using up a resource it doesn’t need.

![Figure 8: Scenario 2: AI is near Need B objective but Need A level is far greater.](image)
Scenario 3: Identical Options

In this scenario, the need objectives are only a space apart and the AI is north of them, sitting at the top of the map in the column that sits between the objectives. Once again, needs do not degrade in this example. The AI again results to the tie-breaking mechanism with its first move (or lack thereof). Once the first move is decided, the AI continues toward the same objective. This once again emphasizes the importance of the tie-breaking mechanism. It very often decides the result of the test. In this case, there was no benefit to going after one objective or the other, so the AI took whichever path it saw first. This behavior is acceptable because it really doesn’t matter which need the AI goes after. It would be no different if a person were looking for a pencil and paper to write a note. It doesn’t really matter which object the person picks up first, since it needs both equally to complete the task.

<table>
<thead>
<tr>
<th>1a</th>
<th>1a</th>
<th>1a</th>
<th>X</th>
<th>1a</th>
<th>1a</th>
<th>1a</th>
<th>1a</th>
<th>1a</th>
<th>1a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
</tr>
<tr>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
<td>1a</td>
</tr>
<tr>
<td>1a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
</tr>
<tr>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
</tr>
<tr>
<td>1a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
</tr>
<tr>
<td>1a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
<td>2a</td>
</tr>
</tbody>
</table>

Figure 9: Scenario 3: AI has no greater need for either objective

Scenario 4: Opportunity

This scenario sets the AI on the far end of the map, with two objectives that are on the other side of the map. One of the objectives is closer to the AI, but it is also desired slightly less. The result of this case is that the AI showed opportunistic behavior and grabbed the closer need first, even though it did not need it quite as much as farther objective. This is the kind of behavior that is desired out of such an AI. If this were typical smart terrain, the agent would have passed up the closer objective and went
That behavior doesn’t seem as intelligent because the agent still needs the objective it passed up, just not quite as much. Instead, with the multiple need algorithm, it appeared that the AI grabbed the closer objective on its way to the farther, more desired objective. What appears to be a planned maneuver is really just like a pig sniffing out a snack but then noticing the valuable truffle a few yards away.

<table>
<thead>
<tr>
<th>X</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
<th>( \delta_b )</th>
</tr>
</thead>
<tbody>
<tr>
<td>10b</td>
<td>10b</td>
<td>10b</td>
<td>10b</td>
<td>10b</td>
<td>10b</td>
<td>10b</td>
<td>10b</td>
<td>10b</td>
</tr>
<tr>
<td>20b</td>
<td>20b</td>
<td>20b</td>
<td>20b</td>
<td>20b</td>
<td>20b</td>
<td>20b</td>
<td>20b</td>
<td>20b</td>
</tr>
<tr>
<td>30b</td>
<td>30b</td>
<td>30b</td>
<td>30b</td>
<td>30b</td>
<td>30b</td>
<td>30b</td>
<td>30b</td>
<td>30b</td>
</tr>
<tr>
<td>40b</td>
<td>40b</td>
<td>40b</td>
<td>40b</td>
<td>40b</td>
<td>40b</td>
<td>40b</td>
<td>40b</td>
<td>40b</td>
</tr>
<tr>
<td>50b</td>
<td>50b</td>
<td>50b</td>
<td>50b</td>
<td>50b</td>
<td>50b</td>
<td>50b</td>
<td>50b</td>
<td>50b</td>
</tr>
</tbody>
</table>

Figure 10: Scenario 4b: AI doesn’t require Need Objective B as much as A but it’s valuable enough to grab along the way.

**Scenario 5: Introducing Degrading Needs**

For this example, degrading needs was enabled. One objective, which the AI really doesn’t need very much, is closer by than an objective that the AI needs greatly. As the AI makes its way to the objective it needs, the needs degrade. By the time the need for the first objective becomes relevant, the AI is already too far away to notice it. The lesson here is if needs degrade at the same rate, it will hold little relevance to the AI’s decisions. It might as well be disabled, as the result would have ended up the same. Yet, this result is the logical decision for the AI to make. While it was in range of the objective that it passed up, it really didn’t need it enough until it was too late.
Figure 11: Scenario 5: Need Objective A’s level didn’t degrade fast enough for it to overpower the closer B’s signal.

Scenario 5a

The setup is the same as scenario 5, with the exception that the closer need degrades twice as fast. The result also ended up the same. The AI ignored the closer need and went after the farther away one it needed more. By the time the need of the closer objective overtook the farther away one, the AI was too close to other objective for it to matter. What is demonstrated here is even if a need degrades twice as fast as another, it will matter little if the difference in them is great at the start. By the time the second need catches up, the AI is too close to the other objective for it matter. This is the expected behavior of the AI. It makes its decisions on a turn-by-turn basis. If the importance of the objective isn’t high enough when the AI is close, it should ignore it.

Scenario 6: Need Degradation Rates

This time, both objectives are an equal distance away but the need for one degrades twice as fast. The AI’s first move did not decide which objective it chased. It took a step toward the need objective that degraded at the default rate and then turned around and headed toward the objective whose need was degrading twice as fast. This behavior is both interesting and a problem. It is really
unnatural for an entity to change its mind after it makes the first step. It simply looks unnatural, even though humans make this type of last minute change all the time. We expect the AI to head after one objective and stick to it. This type of behavior could also create AI-breaking bugs. If there were to be more than two needs that surrounded the AI, it is possible that the AI might not be able to make a decision and possibly run around in a circle, at least until one need is the clear winner. The result would be an AI that looks very stupid. This would basically be the human equivalent to pacing, simply not able to make up its mind about where to go. This is something we would like to avoid, in most cases at least.

Figure 12: Scenario 6: AI needs to use tie breaker to decide which need to go to first.

Figure 13: Scenario 6: Step 1: AI decides to go after Need Objective A.
Figure 14: Scenario 6: Step 3: AI doubles back to go after Need Objective B, whose level is degrading twice as fast as A’s.

Scenario 6a

The setup is similar to scenario 6, but this time the AI’s tie breaking mechanism will pick the objective whose need that degrades twice as fast. This is Objective B. Objective A’s need degrades at normal rates. Due to this, the AI heads toward the objective and continues normally to Objective B. Once it reaches it, that need drops to 0. It takes one step toward Objective A and that allows the need for the previous objective to be brought back up to 2. Then it turns back and grabs Objective B once again. It does this behavior one more time before the other objective’s need is high enough to overpower. This demonstrates that both the need degradation value and the tie-breaking mechanism are powerful parts of this AI. If not used with care, the AI will do things that simply don’t look natural. There are a few ways to avoid this. One way is to never allow one need to have a degradation value that’s twice as high as another. Another way is to have your objectives be consumable, or have a cool down of sorts. They stop radiating a while after they have been used. The major issue with that concept is if you have multiple AI’s, they won’t be able to use an object that they should be able, for example, a toilet.
**Interpreting the Results**

The results of the tests help display the algorithms strengths and weaknesses. The scenarios aren’t the only thing that helps us determine the algorithm’s performance, but the very choices we make in implementation gives us clues. In order to judge the algorithm’s usefulness, we must look at all of these in detail.

**Strengths**

**Opportunistic Behavior**

In most of the scenarios listed above, the AI demonstrated opportunistic behavior. The character did not pass up an objective it needed, just because it’s not the most desired item on its list. As long the objective is relatively in need, the AI will grab it on the way. Yet, since in most game worlds resources area limited, it will not grab items it does not need very much. These things are relative to the character’s need levels and the objective’s distance from the AI, but these characteristics usually hold true.

**Versatility**

The algorithm is heavily versatile. In the scenarios, it was demonstrated how we could have needs that degrade and needs that don’t. We can change how much those needs degrade at a time. The need levels themselves help us decide on the importance of objectives. We can customize any of these on the fly, only requiring an update of our combined map.

**Customization**
As discussed in the testing environment section, there are many choices that need to be made in order to make use of the algorithm. In this way, its performance is really up to the AI designer. It can be very performance friendly or a drain on the CPU cycles, depending on which choices are made.

**Scalability**

This algorithm’s scalability, in terms of how many characters can use the system, how many objectives it can handle, can vary. To answer this question, we look not at our tests but a spreadsheet created just for this task. Figure 15 shows the same grid the AI sees, except where each ‘Need’ is, the value of the space is $V = S+(S/2)^n$ where $n$ is the need level of that particular need, $S$ is the size of the rows and columns. In this case, if the need for the item is 1, $V = 15$. There are 6 different needs objectives in a 100 square area, with each need level equal to one. It can be seen that the AI’s tie-breaking mechanism is going to be used very much in this scenario, if each of the need levels remain the same. Once some of the values change, some clearer paths are noticeable. With this test it is clear that the middle areas (where Need C and Need D are located) are strong areas to put important objectives. Their values tend to dominate the map.

![Figure 15: This is the grid that the AI sees, where (in this case) the ‘Need’ objectives are equal to 15.](image-url)
As for characters that can use the system, that heavily relies upon how often maps are refreshed. The propagation only needs to be done once. Yet, each character will have to multiply the need level modifier against the map values in order to make a decision on where to go. If there are a lot of AI units, the combined map for each should be updated as infrequently as possible.

**Weaknesses**

**Pathing Around Objects**

The algorithm is perfectly capable of pathing around objects, yet in order to really do it well, it must do something akin to recursion for the propagation. As mentioned before, recursion is very CPU intensive, but if the propagation doesn’t need to be done in real time, this will not matter much.

If just using the “distance from the objective” formula for propagation, it is possible for the AI to find itself in a dead end. Note Figure 16. If the AI is coming from Need Objective E, on its way to Need Objective B, it will run into a dead end at both C10 and J10. With the way the map is set up on Figure 16, the AI will pace back and forth, lost in an infinite loop.

This can easily be avoided with a few ways. The level designer can be careful not to build such places that the AI might get stuck, or the AI designer can see the possibility ahead of time and mark those places as obstacles even though they are clear area. This has its own sort of setback, because if the AI unit is an enemy, the player now has a place to hide that the AI won’t go into. Another possible idea would be to create a contingency that if the unit gets stuck, it ignores its top priority need for a set amount of time.
Figure 16: Obstacles while using the ‘Distance From’ algorithm for propagation.

Scalability

Scalability is not only a strength for this algorithm, but in some ways it is a weakness as well. For the 100 space map used to test the algorithm, 6 objectives is likely to be too many. Just by setting Need D’s Level to 0 (like in Figure 17), it can be seen how the map clears up a bit. As seen in the scenario 7, it is possible for the AI to pace back and forth between objectives with degrading needs being enabled. Having as many as 6 objectives in 100 spaces makes this even more likely.

So, my recommendation is not to have more objectives than equal to O, where O = (O/10)/2, where A equals the total number of squares on your map or 5 objectives for every 100 squares. That, again, is a limit and a downside, but a reasonable one.
Tie-breaking Mechanism

As demonstrated in very many of the scenarios, many times the AI’s behavior is decided not by the smart terrain map, but by its tie-breaking behavior. In other words, sometimes your AI is only as intelligent looking as its tie-breaking mechanism is good at making proper decisions. In the sample application, the AI simply went with the highest value it saw first, sometimes leading to less than optimal decisions. Then there are scenarios like Scenario 1b, where the AI makes the right decision out of sheer luck. Had it went with the last high value it saw instead, it would have traveled to the farther of the two objectives. Either way, the tie-breaking mechanism must be carefully designed in order to try to make as few mistakes as possible.

Future Work

Much work is needed to be done to make this algorithm as optimal as possible. Its two largest problems are the tie-breaking mechanism and the propagation around objects.
The tie-breaking mechanism is challenging because in order to make it smarter, the AI character has to have more information about the world. For example, one possible way of fixing its odd movement choices is to have the AI keep going the way it was going the previous turn. That way, the AI has to have some sense of direction. For example, if the AI headed south the last turn, and one of the tied numbers is also south from it, it should take that route. The flaw in this is that if the objective is south-east from the AI, the AI just took the long way.

Another possible idea is to let the AI know which objective it is heading towards, but this is more complicated than it sounds. That requires our numerical map to have some sort of ‘tell’ that allows the AI to know which value on the map came from which need objective map. This could have a possible bad side effect, of course. The AI could lose some of its optimistic behavior. If two values are tied and AI decides to continue on the path it was one, it might pass up an easy objective that needs, just not quite as much as its highest priority objective.

Propagation

The propagation around object is mainly a performance versus quality issue. Using recursion is CPU intensive but accurate. Using recursion in real-time could be very slow, especially if the map updates frequently.

Finding the distance a square is from an objective is lightweight but can be fairly inaccurate when it comes to obstacles. The only way to do obstacles with this method is to black out parts of the map after the propagation has occurred. This can lead the AI down dead ends, requiring some sort of contingency place that probably will require a map update. Have a few too many units run into dead ends at the same time and the game could suffer from slow down.
Both methods have their flaws and neither is perfect. A proposed solution would be a combination of the two. Most times the propagation uses the distance formula, but when it runs into an object, it would use a recursive method to path around it. If the recursive method runs into a dead end, it can be set up to give those areas low values unless an objective is at the end of the dead end.

**Conclusion**

Today’s games are plagued with weak AI. Sometimes, the only way developers can give the player an interesting opponent is to include multi-player in the game. Lately, this has becoming a primary focus and the single player game has been taking a back seat. Sometimes, the only way developers can challenge the player is allowing the AI to ‘cheat’ instead of making it smarter or send overwhelming odds at the player. Boss battles suffer from this particularly. Players are used to seeing a boss that is bigger and more powerful than they are. Tactics that work on every other enemy in the game doesn’t work on the bosses, sometimes for the only reason behind it is that they are “immune”. Any of these tactics are ‘cheap’ and usually require scripting only. Once the player figures out the pattern, it is usually very easy to defeat the enemy.

This is where developers need to take notice. It is very rare these days where the player feels like they out-played the computer. They simply learned the patterns that are scripted and beat it that way. Using smart terrain, the enemy doesn’t have patterns and is likely to change its based on its current needs, which could very well be dynamic and semi-random.

I believe using smart terrain with multiple needs brings the algorithm one step closer to being usable in these other types of games. A smart AI makes for a vastly replayable game, giving the player a possibly new challenge every time they play through the game. Not only that, but if it is configured and implemented correctly, it can create seemingly complex characters with little use of the CPU.
It not only allows for a more believable non-playable character but it also can create a more challenging, smarter opponent, one that plays by the rules of the game and doesn’t ‘cheat’. It also can prevent players from getting bored with the game because they simply memorize the AI’s patterns and predict their every movement. The AI will likely function differently because the player changes the world and thus the world will give the AI different instructions to follow to carry out their missions.

I believe this system could benefit games use the ‘trial and error’ method of difficulty. In this scenario, the player is brutally destroyed by a scripted AI. It takes a number of retries to finish the scenario, as the player learns enemy placement and scripted behaviors. Once the player catches on, the enemy is thrown off of its scripted behavior and forced to behave on its default behavior, which is normally just the same old “rush the player and hope to kill them.” Even if we like the idea of having this scripted puzzle-like scenario, once the player breaks the AI’s routine, it’s a smile matter of dispatching it. If we use a concept like smart terrain to take over for the enemy’s default behavior, it would create a more challenging experience for the player once they learn the scripted patterns.

This is just the tip of what smart terrain is capable of and allowing the character to look for multiple objectives at the same time brings it to a whole new level of capability and more believable AI characters. With these characters, games can have a more engaging single player or co-operational experience, instead of the current norm where the single player feels tacked on just to sell a game focused on multiplayer. What multiplayer will always lack is immersion. It will always be obvious that the soldier you are fighting is someone playing a game, not a living breathing entity that is fighting for his life, country or what they believe in. AI can bring us this immersion, if given the tools to do so.
Referenced Works


Doyle, Patrick. Virtual Intelligence from Artificial Reality: Building Stupid Agents in Smart Environments.


Appendix A: Source Code

Public Class Form1

'This project source. Project was created in Visual Studio 2010 with Visual Basic.

'Project variables
Dim Str1Array(10, 10) As Integer
Dim Str2Array(10, 10) As Integer
Dim Str3Array(10, 10) As String
Dim Str4Array(10, 10) As Integer
Dim Need2Array(10, 10) As Integer
Dim objective1x As Integer
Dim objective1y As Integer
Dim objective2x As Integer
Dim objective2y As Integer
Dim Need1Var As Integer
Dim Need2Var As Integer
Dim characterx As Integer
Dim charactery As Integer
Public Counter As Integer
Dim rowsize As Integer = 10
Dim Square(9) As Integer
Dim SquareX(9) As Integer
Dim SquareY(9) As Integer
Dim FinalX As Integer
Dim FinalY As Integer
Dim Testval As Integer
Dim topleft As Integer
Dim topright As Integer
Dim bottomleft As Integer
Dim bottomright As Integer
Dim leftwall As Integer
Dim rightwall As Integer
Dim topwall As Integer
Dim bottomwall As Integer
Dim testnumber As Integer
Dim degradingneeds As Integer = 0
Dim Need1DegVar As Integer
Dim Need2DegVar As Integer

'This function fills the grid with numbers generated by the propagation formula. It is called every time the Populate button is pushed and when the Next button is clicked.
Private Sub Populate()
    objective1x = Val(NeedBoxX.Text)
    objective1y = Val(NeedBoxY.Text)
    objective2x = Val(Need2X.Text)
    objective2y = Val(Need2Y.Text)
    characterx = Val(CharX.Text)
    charactery = Val(CharY.Text)
    Need1Var = Need1.Value
Need2Var = Need2.Value
Need1DegVar = Val(Need1Deg.Text)
Need2DegVar = Val(Need2Deg.Text)

' Check if degrading needs is checked true.
If CheckBox1.Checked = True Then
    degradingneeds = 1
Else
    degradingneeds = 0
End If

' These loops call the Flood1 function, which does the actual mathematics.
For x = 1 To 10
    For y = 1 To 10
        Str1Array(x, y) = Need1Var * (Flood1(x, y, objective1x, objective1y))
    Next
Next

For x = 1 To 10
    For y = 1 To 10
        Str2Array(x, y) = Need2Var * (Flood1(x, y, objective2x, objective2y))
    Next
Next

If Counter = 1 Then
    DataGridView1.Rows.Clear()
    DataGridView2.Rows.Clear()
    DataGridView3.Rows.Clear()
    Counter = 0
End If

With Me.DataGridView1.Rows
    .Add(Str1Array(1, 1), Str1Array(2, 1), Str1Array(3, 1), Str1Array(4, 1),
         Str1Array(5, 1), Str1Array(6, 1), Str1Array(7, 1), Str1Array(8, 1), Str1Array(9, 1),
         Str1Array(10, 1))
    .Add(Str1Array(1, 2), Str1Array(2, 2), Str1Array(3, 2), Str1Array(4, 2),
         Str1Array(5, 2), Str1Array(6, 2), Str1Array(7, 2), Str1Array(8, 2), Str1Array(9, 2),
         Str1Array(10, 2))
    .Add(Str1Array(1, 3), Str1Array(2, 3), Str1Array(3, 3), Str1Array(4, 3),
         Str1Array(5, 3), Str1Array(6, 3), Str1Array(7, 3), Str1Array(8, 3), Str1Array(9, 3),
         Str1Array(10, 3))
    .Add(Str1Array(1, 4), Str1Array(2, 4), Str1Array(3, 4), Str1Array(4, 4),
         Str1Array(5, 4), Str1Array(6, 4), Str1Array(7, 4), Str1Array(8, 4), Str1Array(9, 4),
         Str1Array(10, 4))
    .Add(Str1Array(1, 5), Str1Array(2, 5), Str1Array(3, 5), Str1Array(4, 5),
         Str1Array(5, 5), Str1Array(6, 5), Str1Array(7, 5), Str1Array(8, 5), Str1Array(9, 5),
         Str1Array(10, 5))
    .Add(Str1Array(1, 6), Str1Array(2, 6), Str1Array(3, 6), Str1Array(4, 6),
         Str1Array(5, 6), Str1Array(6, 6), Str1Array(7, 6), Str1Array(8, 6), Str1Array(9, 6),
         Str1Array(10, 6))
    .Add(Str1Array(1, 7), Str1Array(2, 7), Str1Array(3, 7), Str1Array(4, 7),
         Str1Array(5, 7), Str1Array(6, 7), Str1Array(7, 7), Str1Array(8, 7), Str1Array(9, 7),
         Str1Array(10, 7))
    .Add(Str1Array(1, 8), Str1Array(2, 8), Str1Array(3, 8), Str1Array(4, 8),
         Str1Array(5, 8), Str1Array(6, 8), Str1Array(7, 8), Str1Array(8, 8), Str1Array(9, 8),
         Str1Array(10, 8))
    .Add(Str1Array(1, 9), Str1Array(2, 9), Str1Array(3, 9), Str1Array(4, 9),
         Str1Array(5, 9), Str1Array(6, 9), Str1Array(7, 9), Str1Array(8, 9), Str1Array(9, 9),
         Str1Array(10, 9))
    .Add(Str1Array(1, 10), Str1Array(2, 10), Str1Array(3, 10), Str1Array(4, 10),
         Str1Array(5, 10), Str1Array(6, 10), Str1Array(7, 10), Str1Array(8, 10), Str1Array(9, 10),
         Str1Array(10, 10))
End With
.Add(Str1Array(1, 10), Str1Array(2, 10), Str1Array(3, 10), Str1Array(4, 10),
Str1Array(5, 10), Str1Array(6, 10), Str1Array(7, 10), Str1Array(8, 10), Str1Array(9, 10),
Str1Array(10, 10))

End With

With Me.DataGridView2.Rows
  .Add(Str2Array(1, 1), Str2Array(2, 1), Str2Array(3, 1), Str2Array(4, 1),
Str2Array(5, 1), Str2Array(6, 1), Str2Array(7, 1), Str2Array(8, 1), Str2Array(9, 1),
Str2Array(10, 1))
  .Add(Str2Array(1, 2), Str2Array(2, 2), Str2Array(3, 2), Str2Array(4, 2),
Str2Array(5, 2), Str2Array(6, 2), Str2Array(7, 2), Str2Array(8, 2), Str2Array(9, 2),
Str2Array(10, 2))
  .Add(Str2Array(1, 3), Str2Array(2, 3), Str2Array(3, 3), Str2Array(4, 3),
Str2Array(5, 3), Str2Array(6, 3), Str2Array(7, 3), Str2Array(8, 3), Str2Array(9, 3),
Str2Array(10, 3))
  .Add(Str2Array(1, 4), Str2Array(2, 4), Str2Array(3, 4), Str2Array(4, 4),
Str2Array(5, 4), Str2Array(6, 4), Str2Array(7, 4), Str2Array(8, 4), Str2Array(9, 4),
Str2Array(10, 4))
  .Add(Str2Array(1, 5), Str2Array(2, 5), Str2Array(3, 5), Str2Array(4, 5),
Str2Array(5, 5), Str2Array(6, 5), Str2Array(7, 5), Str2Array(8, 5), Str2Array(9, 5),
Str2Array(10, 5))
  .Add(Str2Array(1, 6), Str2Array(2, 6), Str2Array(3, 6), Str2Array(4, 6),
Str2Array(5, 6), Str2Array(6, 6), Str2Array(7, 6), Str2Array(8, 6), Str2Array(9, 6),
Str2Array(10, 6))
  .Add(Str2Array(1, 7), Str2Array(2, 7), Str2Array(3, 7), Str2Array(4, 7),
Str2Array(5, 7), Str2Array(6, 7), Str2Array(7, 7), Str2Array(8, 7), Str2Array(9, 7),
Str2Array(10, 7))
  .Add(Str2Array(1, 8), Str2Array(2, 8), Str2Array(3, 8), Str2Array(4, 8),
Str2Array(5, 8), Str2Array(6, 8), Str2Array(7, 8), Str2Array(8, 8), Str2Array(9, 8),
Str2Array(10, 8))
  .Add(Str2Array(1, 9), Str2Array(2, 9), Str2Array(3, 9), Str2Array(4, 9),
Str2Array(5, 9), Str2Array(6, 9), Str2Array(7, 9), Str2Array(8, 9), Str2Array(9, 9),
Str2Array(9, 10))
  .Add(Str2Array(1, 10), Str2Array(2, 10), Str2Array(3, 10), Str2Array(4, 10),
Str2Array(5, 10), Str2Array(6, 10), Str2Array(7, 10), Str2Array(8, 10), Str2Array(9, 10),
Str2Array(10, 10))
End With

' This is for the combined graph. It marks what value is the highest or if they are equal.
For x = 1 To 10
  For y = 1 To 10
    If Str1Array(x, y) > Str2Array(x, y) Then
      Str4Array(x, y) = Str1Array(x, y)
      Str3Array(x, y) = "" & Str1Array(x, y) & "a"
    End If
  End For
For y = 1 To 10
  If Str2Array(x, y) > Str1Array(x, y) Then
    Str4Array(x, y) = Str2Array(x, y)
End If
Str3Array(x, y) = "" & Str2Array(x, y) & "b"
End If

If Str2Array(x, y) = Str1Array(x, y) Then
    Str4Array(x, y) = Str2Array(x, y)
    Str3Array(x, y) = "" & Str2Array(x, y) & "=
End If
Next

' Showing the user where the need objectives and the AI are on the Combined map.
Str3Array(objective1x, objective1y) = "NeedA"
Str3Array(objective2x, objective2y) = "NeedB"
Str3Array(characterx, charactery) = "X"
With Me.DataGridView3.Rows
    .Add(Str3Array(1, 1), Str3Array(2, 1), Str3Array(3, 1), Str3Array(4, 1),
         Str3Array(5, 1), Str3Array(6, 1), Str3Array(7, 1), Str3Array(8, 1), Str3Array(9, 1),
         Str3Array(10, 1))
    .Add(Str3Array(1, 2), Str3Array(2, 2), Str3Array(3, 2), Str3Array(4, 2),
         Str3Array(5, 2), Str3Array(6, 2), Str3Array(7, 2), Str3Array(8, 2), Str3Array(9, 2),
         Str3Array(10, 2))
    .Add(Str3Array(1, 3), Str3Array(2, 3), Str3Array(3, 3), Str3Array(4, 3),
         Str3Array(5, 3), Str3Array(6, 3), Str3Array(7, 3), Str3Array(8, 3), Str3Array(9, 3),
         Str3Array(10, 3))
    .Add(Str3Array(1, 4), Str3Array(2, 4), Str3Array(3, 4), Str3Array(4, 4),
         Str3Array(5, 4), Str3Array(6, 4), Str3Array(7, 4), Str3Array(8, 4), Str3Array(9, 4),
         Str3Array(10, 4))
    .Add(Str3Array(1, 5), Str3Array(2, 5), Str3Array(3, 5), Str3Array(4, 5),
         Str3Array(5, 5), Str3Array(6, 5), Str3Array(7, 5), Str3Array(8, 5), Str3Array(9, 5),
         Str3Array(10, 5))
    .Add(Str3Array(1, 6), Str3Array(2, 6), Str3Array(3, 6), Str3Array(4, 6),
         Str3Array(5, 6), Str3Array(6, 6), Str3Array(7, 6), Str3Array(8, 6), Str3Array(9, 6),
         Str3Array(10, 6))
    .Add(Str3Array(1, 7), Str3Array(2, 7), Str3Array(3, 7), Str3Array(4, 7),
         Str3Array(5, 7), Str3Array(6, 7), Str3Array(7, 7), Str3Array(8, 7), Str3Array(9, 7),
         Str3Array(10, 7))
    .Add(Str3Array(1, 8), Str3Array(2, 8), Str3Array(3, 8), Str3Array(4, 8),
         Str3Array(5, 8), Str3Array(6, 8), Str3Array(7, 8), Str3Array(8, 8), Str3Array(9, 8),
         Str3Array(10, 8))
    .Add(Str3Array(1, 9), Str3Array(2, 9), Str3Array(3, 9), Str3Array(4, 9),
         Str3Array(5, 9), Str3Array(6, 9), Str3Array(7, 9), Str3Array(8, 9), Str3Array(9, 9),
         Str3Array(10, 9))
    .Add(Str3Array(1, 10), Str3Array(2, 10), Str3Array(3, 10), Str3Array(4, 10),
         Str3Array(5, 10), Str3Array(6, 10), Str3Array(7, 10), Str3Array(8, 10), Str3Array(9, 10),
         Str3Array(10, 10))
End With
Counter = 1
End Sub

Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
    Handles Button1.Click
        Populate()
        testnumber = testnumber + 1
Log.Text = Log.Text + "Test Number: " & testnumber & vbCrLf
Log.Text = Log.Text + "Need A Objective is at" & NeedBoxX.Text & " , " & 
NeedBoxY.Text & vbCrLf
Log.Text = Log.Text + "Need B Objective is at" & Need2X.Text & " , " & 
Need2Y.Text & vbCrLf
Log.Text = Log.Text + "Need A is set at" & Need1.Value & vbCrLf
Log.Text = Log.Text + "Need B is set at" & Need2.Value & vbCrLf
& vbCrLf
If CheckBox1.Checked = True Then
    Log.Text = Log.Text + "Degrading Needs is enabled." & vbCrLf
degradingsneeds = 1
Else
    Log.Text = Log.Text + "Degrading Needs is disabled." & vbCrLf
degradingsneeds = 0
End If

Private Sub MoveCharacter()
    ' This is the actual AI algorithm. It consists of a number of different checks 
    depending on where the AI is on the map

    bottomleft = 0
    bottomright = 0
    topright = 0
    topleft = 0
    rightwall = 0
    leftwall = 0
    topwall = 0
    bottomwall = 0
    ' bottom left corner check
    If charactery = rowsize And characterx = 1 Then
        Square(0) = Str4Array(characterx, charactery - 1)
        SquareX(0) = characterx
        SquareY(0) = charactery - 1
        Square(1) = Str4Array(characterx + 1, charactery - 1)
        SquareX(1) = characterx + 1
        SquareY(1) = charactery - 1
        Square(2) = Str4Array(characterx + 1, charactery)
        SquareX(2) = characterx + 1
        SquareY(2) = charactery
        Testval = Square(0)
        FinalX = SquareX(0)
        FinalY = SquareY(0)
bottomleft = 1
        Log.Text = Log.Text + "Bottom Left Corner Detected." & vbCrLf
        For x = 1 To 2
            If Testval < Square(x) Then
                Testval = Square(x)
                FinalX = SquareX(x)
                FinalY = SquareY(x)
            End If
        Next x
    End If

    End Sub
' top left corner check
If charactery = 1 And characterx = 1 Then
    Square(0) = Str4Array(characterx + 1, charactery)
    SquareX(0) = characterx + 1
    SquareY(0) = charactery
    Square(1) = Str4Array(characterx + 1, charactery + 1)
    SquareX(1) = characterx + 1
    SquareY(1) = charactery + 1
    Square(2) = Str4Array(characterx, charactery + 1)
    SquareX(2) = characterx
    SquareY(2) = charactery + 1
topleft = 1
Log.Text = Log.Text + "Top Left Corner Detected." & vbCrLf
Testval = Square(0)
FinalX = SquareX(0)
FinalY = SquareY(0)
For x = 1 To 2
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next
End If

' top right corner check
If charactery = 1 And characterx = rowsize Then
    Square(0) = Str4Array(characterx - 1, charactery)
    SquareX(0) = characterx - 1
    SquareY(0) = charactery
    Square(1) = Str4Array(characterx, charactery + 1)
    SquareX(1) = characterx
    SquareY(1) = charactery + 1
    Square(2) = Str4Array(characterx - 1, charactery + 1)
    SquareX(2) = characterx - 1
    SquareY(2) = charactery + 1
topright = 1
Log.Text = Log.Text + "Top Right Corner Detected." & vbCrLf
Testval = Square(0)
FinalX = SquareX(0)
FinalY = SquareY(0)
For x = 1 To 2
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next
End If

' bottom right corner check
If charactery = rowsize And characterx = rowsize Then
    Square(0) = Str4Array(characterx - 1, charactery - 1)
SquareX(0) = characterx - 1
SquareY(0) = charactery - 1
Square(1) = Str4Array(characterx, charactery - 1)
SquareX(1) = characterx
SquareY(1) = charactery - 1
Square(2) = Str4Array(characterx - 1, charactery)
SquareX(2) = characterx - 1
SquareY(2) = charactery
bottomright = 1
Log.Text = Log.Text + "Bottom Right Corner Detected." & vbCrLf
Testval = Square(0)
FinalX = SquareX(0)
FinalY = SquareY(0)
For x = 1 To 2
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next

' left wall check
If characterx = 1 And bottomleft = 0 And topleft = 0 Then
    Square(0) = Str4Array(characterx, charactery - 1)
    SquareX(0) = characterx
    SquareY(0) = charactery - 1
    Square(1) = Str4Array(characterx + 1, charactery - 1)
    SquareX(1) = characterx + 1
    SquareY(1) = charactery - 1
    Square(2) = Str4Array(characterx + 1, charactery)
    SquareX(2) = characterx + 1
    SquareY(2) = charactery
    Square(3) = Str4Array(characterx, charactery + 1)
    SquareX(3) = characterx
    SquareY(3) = charactery + 1
    Square(4) = Str4Array(characterx + 1, charactery + 1)
    SquareX(4) = characterx + 1
    SquareY(4) = charactery + 1
    Testval = Square(0)
    FinalX = SquareX(0)
    FinalY = SquareY(0)
    leftwall = 1
Log.Text = Log.Text + "Left Wall Detected." & vbCrLf
For x = 1 To 8
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next

End If
'right wall check
If characterx = rowsize And bottomright = 0 And topright = 0 Then
    Square(0) = Str4Array(characterx, charactery - 1)
    SquareX(0) = characterx
    SquareY(0) = charactery - 1
    Square(1) = Str4Array(characterx - 1, charactery - 1)
    SquareX(1) = characterx - 1
    SquareY(1) = charactery - 1
    Square(2) = Str4Array(characterx - 1, charactery)
    SquareX(2) = characterx - 1
    SquareY(2) = charactery
    Square(3) = Str4Array(characterx - 1, charactery + 1)
    SquareX(3) = characterx - 1
    SquareY(3) = charactery + 1
    Square(4) = Str4Array(characterx, charactery + 1)
    SquareX(4) = characterx
    SquareY(4) = charactery + 1
    Testval = Square(0)
    FinalX = SquareX(0)
    FinalY = SquareY(0)
    rightwall = 1
Log.Text = Log.Text + "Right Wall Detected." & vbCrLf
For x = 1 To 4
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next
End If

'top wall check
If charactery = 1 And topright = 0 And topleft = 0 Then
    Square(0) = Str4Array(characterx - 1, charactery)
    SquareX(0) = characterx - 1
    SquareY(0) = charactery
    Square(1) = Str4Array(characterx - 1, charactery + 1)
    SquareX(1) = characterx - 1
    SquareY(1) = charactery + 1
    Square(2) = Str4Array(characterx, charactery + 1)
    SquareX(2) = characterx
    SquareY(2) = charactery + 1
    Square(3) = Str4Array(characterx + 1, charactery + 1)
    SquareX(3) = characterx + 1
    SquareY(3) = charactery + 1
    Square(4) = Str4Array(characterx + 1, charactery)
    SquareX(4) = characterx + 1
    SquareY(4) = charactery
    Testval = Square(0)
    FinalX = SquareX(0)
    FinalY = SquareY(0)
    topwall = 1
Log.Text = Log.Text + "Top Wall Detected." & vbCrLf
For x = 1 To 4
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
    End If
Next
FinalY = SquareY(x)

End If
Next
End If

'bottom wall check
If charactery = rowsize And bottomright = 0 And bottomleft = 0 Then
    Square(0) = Str4Array(characterx, charactery - 1)
    SquareX(0) = characterx
    SquareY(0) = charactery - 1
    Square(1) = Str4Array(characterx - 1, charactery - 1)
    SquareX(1) = characterx - 1
    SquareY(1) = charactery - 1
    Square(2) = Str4Array(characterx - 1, charactery)
    SquareX(2) = characterx - 1
    SquareY(2) = charactery
    Square(3) = Str4Array(characterx + 1, charactery)
    SquareX(3) = characterx + 1
    SquareY(3) = charactery
    Square(4) = Str4Array(characterx + 1, charactery - 1)
    SquareX(4) = characterx + 1
    SquareY(4) = charactery - 1
    Testval = Square(0)
    FinalX = SquareX(0)
    FinalY = SquareY(0)
    bottomwall = 1
    Log.Text = Log.Text + "Bottom Wall Detected." & vbCrLf
For x = 1 To 4
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next
End If

'General check
If topleft = 0 And topright = 0 And bottomleft = 0 And bottomright = 0 And bottomwall = 0 And topwall = 0 And rightwall = 0 And leftwall = 0 Then
    Square(0) = Str4Array(characterx - 1, charactery - 1)
    SquareX(0) = characterx - 1
    SquareY(0) = charactery - 1
    Square(1) = Str4Array(characterx, charactery - 1)
    SquareX(1) = characterx
    SquareY(1) = charactery - 1
    Square(2) = Str4Array(characterx + 1, charactery - 1)
    SquareX(2) = characterx + 1
    SquareY(2) = charactery - 1
    Square(3) = Str4Array(characterx + 1, charactery)
    SquareX(3) = characterx + 1
    SquareY(3) = charactery
    Square(4) = Str4Array(characterx + 1, charactery + 1)
    SquareX(4) = characterx + 1
    SquareY(4) = charactery + 1
    Square(5) = Str4Array(characterx, charactery + 1)
End If
SquareX(5) = characterx
SquareY(5) = charactery + 1
Square(6) = Str4Array(characterx - 1, charactery + 1)
SquareX(6) = characterx - 1
SquareY(6) = charactery + 1
Square(7) = Str4Array(characterx - 1, charactery)
SquareX(7) = characterx - 1
SquareY(7) = charactery
Testval = Square(0)
FinalX = SquareX(0)
FinalY = SquareY(0)
Log.Text = Log.Text + "General check done." & vbCrLf
For x = 1 To 7
    If Testval < Square(x) Then
        Testval = Square(x)
        FinalX = SquareX(x)
        FinalY = SquareY(x)
    End If
Next
End If
characterx = FinalX
charactery = FinalY
CharX.Text = FinalX
CharY.Text = FinalY
Log.Text = Log.Text + "AI has moved to " & characterx & " , " & charactery & vbCrLf

'Calculate degrading needs effects.
If degradingneeds = 1 Then
    If Need1Var < 10 Then
        Need1.Value = Need1.Value + Need1DegVar
        Need1Var = Need1Var + Need1DegVar
        If Need1Var > 10 Then Need1Var = 10
        Log.Text = Log.Text + "Need 1 has changed to " & Need1Var & vbCrLf
    End If
    If Need2Var < 10 Then
        Need2.Value = Need2.Value + Need2DegVar
        Need2Var = Need2Var + Need2DegVar
        If Need2Var > 10 Then Need2Var = 10
        Log.Text = Log.Text + "Need 2 has changed to " & Need2Var & vbCrLf
    End If
End If

If characterx = objective1x And charactery = objective1y Then
    Need1.Value = 0
    Log.Text = Log.Text + "AI has found Need A at " & characterx & " , " & charactery & vbCrLf
End If
If characterx = objective2x And charactery = objective2y Then
    Need2.Value = 0
    Log.Text = Log.Text + "AI has found Need B at " & characterx & " , " & charactery & vbCrLf
End If
Populate()
End Sub
Private Function Flood1(ByVal x1, ByVal y1, ByVal x2, ByVal y2)
    ' Propagation formula.
    Dim strength As Integer
    Dim resultx As Integer
    Dim resulty As Integer
    If x1 > x2 Then
        resultx = (x1 - x2) * (x1 - x2)
    Else
        resultx = (x2 - x1) * (x2 - x1)
    End If
    If y1 > y2 Then
        resulty = (y1 - y2) * (y1 - y2)
    Else
        resulty = (y2 - y1) * (y2 - y1)
    End If
    strength = Math.Sqrt(resulty + resultx)
    strength = (rowsize + (rowsize / 2)) - strength
    If strength < 0 Then strength = 0
    Return strength
End Function

Private Sub Button2_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button2.Click
    MoveCharacter()
End Sub

Private Sub Button3_Click(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles Button3.Click
    Log.Text = ""
End Sub

Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As System.EventArgs)
Handles MyBase.Load
End Sub
End Class