Designing a Theseus-Minotaur game using Genetic Algorithms

Suchao Chaisilprungrueng

(chaisilp@usc.edu)

Hadi Moradi

(moradi@usc.edu)

Abstract: Designing a game, considering different parameters involved in the difficulty of the game, is an ad hoc procedure and may take a long time w/o considering all possible cases. On the other hand, using simple brute force methods to search for the desired game with a certain difficulty level is very time consuming. In this report, we study the use of genetic algorithm to design a maze, Theseus and Minotaur maze, with the highest difficulty level. We study different fitness function to simplify the difficulty evaluation of a given maze design. We have implemented the algorithm to show the effectiveness of this approach.

Keywords: game design, genetic algorithms, fitness function, AI, Intelligent games.

Introduction

It is known that majority of classic and today games employ Finite *State Machine (FSM)* as a core technique when building AI engine. Since it is fairly easy to implement and very helpful. As an instance, FSN can be designed for forcing all in game objects to follow game's rule. It can also be designed for controlling how game state should progress. (Setup, Running, Pausing, Game clear, Game over, and etc) Fundamentally, how FSM work is it determines the current state of object and define set of states it can jump into together with conditions that must be satisfied before advancing the state.

Up until now, only small number of games in the market employ advance AI techniques such as Machine Learning, Neural Network, and Genetic Algorithm. However, the trends are moving toward utilizing capability of AI in these areas. Given that each and every year, we have more powerful computer hardware support, more processing power, more memory capacity. Soon we will reach the point where we had extra resources for Advance AI.

In this study, we use the Theseus and the Minotaur puzzle invented by Robert Abbott (http://www.logicmazes.com/theseus.html). We have built 3D PuzzleMaze Game Engine, Map-Solver Agent and the Map-Generator tools. In this study, we have only focused on describing the most interesting one, regarding how the Genetic Algorithm (GA) technique have been used in building game's map generator tools. For simplicity, all the examples were made in term of map size 6x6 (small), the larger map size applies the same concept & algorithm.

Genetic Algorithm and Map designing process

The first question that may come across a person's mind is, why bother using Genetic Algorithm? Can not we design it by hand? The answer is yes and no. You may choose to design smaller map size by hand. However, it is not an effective and it is very time consuming process. And for larger games, it might even be impractical to do so.

In brief, designing the map involves two important steps. First, configure the layout setting of all game objects. Second, determine if the map is solvable? Only solvable maps may put into play. To make this point more obvious, consider following example. As shown in figure 1-A, in a 6x6 map we can have

- 36 possible way of placing player position
- 36 possible way of placing enemy position
- 20 possible way of placing exit position
- 2 possible states for placing each of 30 horizontal walls
- 2 possible states for placing each of 30 vertical walls

Total possible configure layout setting = $36*36*20*2^{30}*2^{30}$
= **29,883,725,399,409,473,617,920**

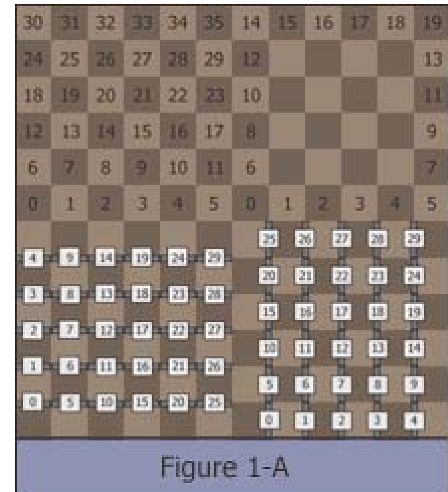

Figure 1-A

In straightforward, we will have trillions combination of possible objects placement when designing the maps. Logically, if we have time and capable of trying all possible combinations, we could design the best map. But to be realistic, trillions possible combinations are way too many. And this is only an example of a small map; larger maps will dramatically result in increase in the number of possible maps.

However, by applying Genetic algorithm in designing the maps, we become more effective, and able to design tons of maps within short period of time. This is all because we let computer's Map-Solver Agent solve the map for us, remove any unsolvable maps from the pie, and use GA to develop better maps from solvable ones. Generally, speed of solving the map depends on the total

number of "visited states" agent had encountered after all branching process. The More difficult maps usually had more visited state than less difficult one. Consequently, as the maps evolve, our agent will spend more time on solving them.

Assume that Map-Solver Agent has average solve time of 100,000 visited states per second. Maps have visited state range from (0 – 300). For simplicity sake, let say all the map has 200 visited states. Thus each second we can solve 100,000/200 = 500 maps.

The agent had to solve 154 maps per generation. Thus it can solve approximate 3.2 generation per sec. Apply this setting, it will only take us less than a minute to simulate 300 generations (Typically 300 generations are enough to get a very good design maps). As you can see, applying GA is undoubtedly more effective way of designing than do it ourselves by hand.

Note that all the number is just approximate number to illustrate how we could benefit from using genetic algorithm.

Overview of Genetic Algorithm

"In the same way that creatures evolve over many generations to become more successful at the tasks of survival and reproduction, genetic algorithms grow and evolve over time to converge upon a solution, or solutions to particular types of problems" *[Mat Buckland, AI Technique for game programming]*

First of all, let's review the theory behind Genetic Algorithm. It is known that every living organism had some basic blue print of itself called genes. These genes are forced to evolve over generations by its nature. Become more successful than its parent or die out to its predators. The evolving mechanism occurs when two organisms mate and produces its offspring (crossover). The obvious outcome, babies inherited genes from both side of parent. This may mean the babies inherited mainly the good genes and become more successful than its parent or it inherited mainly the bad genes and become unsuccessful, not even able survive and reproduce. In addition, during mating process, some of the baby's original genes may be altered (mutate). Frequently, mutations are either bad or rarely affect their fitness. Still, it is an essential element. Because mutations generate diversity effect and sometime even give birth to new feature that make the gene become more successful.

Likewise, the maps we use within game also have blueprints, which are called as map-genome (or just genome for short). In the process of applying the theory of GA, we will try to evolve our map-genome and create the best possible maps for our PuzzleMaze game.

Classic Genetic Algorithm approach consists of seven steps as follow:

1. Encode Map Data (binary format)
2. Generate Sampling population
3. Determine fitness function and assign fitness score
4. Implement Genome Selector (Roulette Wheel Selection)
5. Mating (Generate new Babies genomes with Crossover, Mutation)
6. Repeat Step four until we generate total babies equal to number of population
7. The end of current generation, if we want to continue simulate more generation, assign new population genomes equal to babies genomes and repeat step three

STEP by STEP how to implement Genetic Algorithm

GA-STEP ONE: Encoding Map Data (binary format: zero/one)
There are lots of ways to encode the information and we us binary encoding which seems to be the most suitable way for encoding map-genome. The reason behind this is it's capability of holding all required information, easy to implement and bit operations are generally fast.

A. determine all information will be include in blueprint (map-genome)
Take a look into the game setting, and it is not difficult to come up with a list of following: Player start position, Enemy start position, Exit position, Walls, Map size.

B. determine total possible placement for each object in the map (states)
Player_start_position and **Enemy_start_position** share the same layout, the valid positions that we can place them are from 0 - 35 (total of 36 possible) as shown in figure 1-B.
Exit_position work a little bit differently, since it can be placed only on edges of the map, thus the valid positions are from 0-19 (total of 20 possible) as shown in figure 1-C.
Legend of all objects which are used through out the map figures are shown in figure 1-D

Figure 1-B

Figure 1-C

Figure 1-D

To make referencing index simpler, we categorized walls into two sub types, **Horizontal walls** (h_walls) and **Vertical walls** (v_walls). The valid positions that we can place **h_wall** are from 0 to 29 (total of 30 possible) as shown in figure 1-E. And valid position that we can place **v_wall** are from 0 to 29 (total of 30 possible) as shown in figure 1-F. Each wall has 2 possible states (present/absent)

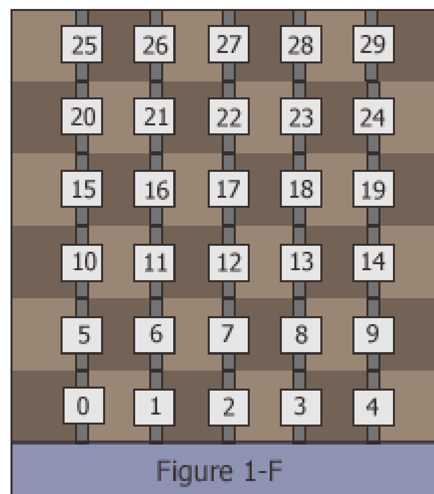

Figure 1-F

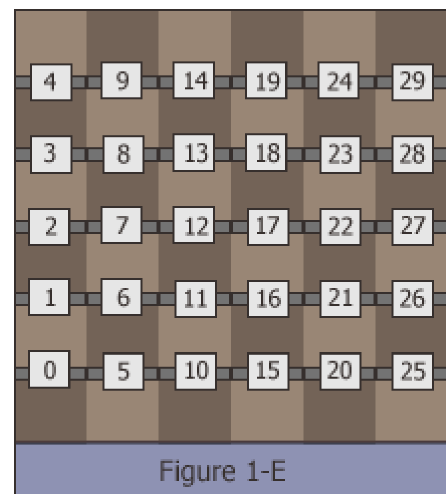

Figure 1-E

C. determine amount of bit necessary for binary encoding.

Table I, shows the relationship between numbers of bit (binary code) and number of possible states it can be represented.

Bit(s)	Possible States
1	2
2	4
3	8
4	16
5	32
6	64
7	128

To maximize efficiency, and reduce unnecessary calculation, always use least number of bits that is closest to possible states according to table I. As an example, to show an object with a total of 6 states (0-5), we use 3 bits (8 states).

Follow the rule above, we come up with Table II, which shown amount of bits required for each objects present on the map.

	Total States	Bit(s)
Player_start_position	36	6
Enemy_start_position	36	6
Exit_position	20	5
Wall	2	1

Since each wall has two possible states (present/absent) we can represent it using one bit. However we do not have one wall in the map, we can have total of 30 h_walls and 30 v_walls. Thus we will need 30 bits for representing h_walls and 30 bits for representing v_walls. All the objects combines, we will require a total of 6+6+5+30+30 = **77 bits** for representing the map-genome.

Invalid-Decoding Challenge

That's not it! As you may doubt, what happen when the first 6 bit transformed to 110001, which after decode its value equal to 49? (We only have 36 possible ways of placing player start position)

That's good question. Binary encoding always faces this problem when we do not have the total states exactly equal to number of base two. However, there are several technique we may apply to solve it.

1. Basic clamp

The basic clamping technique is to clamp any value over maximum to its maximum limit. For example we clamp Player_start_position by `If (value > 35) Then value = 35;` this way, value of (36, 37, 38, 39 … 63) all will instead represent value 35. It is easy but notices that while we eliminate illegal decoding problem, we encounter another problem. Now the odd of generating value 35 is more than the others 0-34. This mean most of the time when we do random selection or do bit operation the result will go to 35 which although it's valid, it is a huge bias.

2. Modulate clamp

The mod clamping technique is to clamp any value over maximum by modulate it with its maximum limit. Using the same example, we clamp Player_start_position by

```
If (value > 35) Then value = value % 35;
```

This will make all the values become valid while still roughly give similar chances to all the number. By mod clamping, we have only two of 0-28 and one of 29-35. Obviously, the second technique is better than the first one and it is very easy to implement.

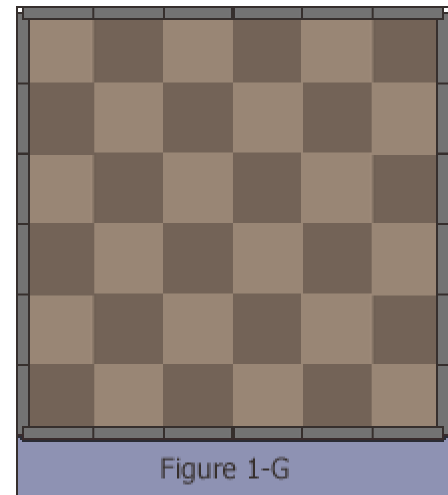
Figure 1-G

Ignoring static object

Beside possible invalid-decoding, you may notice that we did not include any of external walls and map size into the map-genome. Why? This is because externals walls are constant, they always present on the maps. Thus it does no need to encode them. After each decoding, we want them to be present anyways. The map size is also constant. We never decode 77 bits map-genome and get any other map size than the 6x6 one. All external walls present in the map are shown in figure 1-G

The glimpse of unsolvable map

Finally it should be mentioned that not all decoded and valid maps are solvable. The following example shows an unsolvable map which cannot be put it into play.

1. Player_start_position is at exact same location as Enemy_start_position or Exit_position
2. Walls layout completely block around the Player_start_position without any access path to the Exit_position
3. Enemy factor, Remember that, after player makes an action, enemy will do so too. Thus, if the map is not set correctly in such a way that it is solvable, as game progress, there is a good chance that no matter what player move, enemy always catches the player and lead to one hundred percents game over state. In other word, Player has no chance of solving the map and gets stuck at current game level. The good news is we don't have to worry about it. Genetic algorithm with a good fitness function will totally take care of the problem.

Examples of unsolvable maps are shown in figures 1-H and figure 1-I

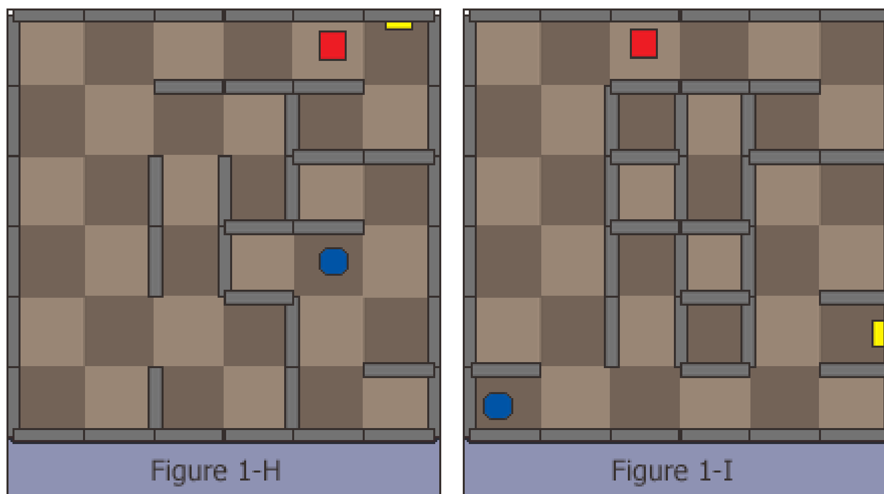
Figure 1-H Figure 1-I

In figure 1-H, the map is unsolvable, because path from Player_start_position to Exit_position is completely blocked by layout of the walls.

In figure 1-I, the map is also unsolvable, although there are possible access paths from Player_start_position to Exit_position, however when taking enemy actions into account. The map

becomes unsolvable. No matter what player make the moves. Player always ends up being catch by enemy or stall mate. As you may try some of following moves,

1. East, East, East, East, and East -> captured
2. East, East, East, East, and north -> captured
3. East, East, North, North, South, Idle, Idle -> stall mate
4. East, East, North and South -> captured
5. East, North, (…) -> captured
6. East, West, East, West, Idle, Idle, Idle, -> stall mate

Note that this map is not very hard to figure it is an unsolvable, as we discovered it within few trials. This is because the complexity of the map is low, the depth of player actions are just about 5-10 steps. However the harder unsolvable map would be very hard to be determined by ourselves. Again no need to worry, by using GA, our Map-Solver Agent will easily take care of that. We will explain more in detail about Map-Solver Agent in section of **Map-Solver Agent overview (GA-STEP THREE).**

GA-STEP TWO: Generate Sampling population

Random Sampling versus Pre-selected Sampling

Sampling population can be either random or pre-selected. Each has it own advantage and disadvantage. In Random Sampling, we have no control over what starting genomes look likes. Most of the time, we just start out from poor quality maps, or unsolvable maps. And because of that, it may take a while to develop into the maps that reach the quality we desire. In the worst case, it might not be able to generate the maps that reach our desire quality at all.

However when random sampling advantages have been considered into account, we found out that is by far outweigh disadvantages. Since this is map designing process, **variety is the most important factor**, Random give exactly what we want “The variety of layout”. Generally, we can afford to wait longer or (regenerate the map again to get better start population) for desire quality maps. Other than, generating one-hundred good maps instantly but later found out that ninety of them having the same layout, using the same trick to solve as the first Ten.

Generating population by pre-selected genome may guarantee that we will have good map in fast piece manner. However, it usually results in lack of layout’s variety. Since all of the map start out from the same group of parent, when we generate enough we will see that the map not really able to improve much from what the parent already have, and thus the map layout lie similar to its parent.

To study the behavior of the genetic algorithm, we wanted to see what would happen if the initial given maps are selected from some good maps. Thus, we generated and saved good maps and implemented a function to load them and run for result.

Unfortunately, it turn out that the quality of the map doesn’t work proportionally as we expected. Since, the higher map-quality, the higher chance that after genetic operation, it will transform into unsolvable map. This lead to what we called **max out** (it is not able to get much more than what its parent already had. In other word, it does not have much chance to evolve from this layout to get a better map). More over, when a pre-selected parent has been used, lack of variety becomes a big problem. Basically, we found out that it rarely generate the result map which has different layout from the pre-selected parent.

In conclusion, because of nature of the problem, Random Sampling has better performance than Pre-selected Sampling. Thus, I decide to apply Random Sampling into our map generator engine.

To implement Random sampling of population map-genome, we applied following algorithm

```
For (I = 0; I < PopSize; I++)
{
    For (bit = 0; bit < 77; bit++)
    {
        PopGenome[I].vecData[bit] = Random number between zero or one
    }
}
```

GA-STEP THREE: Determine fitness function and assign fitness score

Nature of fitness function

A good fitness function is one of most significant factors in determining the success of GA application. Generally we want it to act as a predator. It should try to eliminate all unsolvable maps, penalizes poor-quality maps, so that only good-quality maps will be survived and evolved.

Determine possible fitness functions:

1. Base on shortest-path solution
2. Base on complexity cost of the map
3. Base on enemy factor
4. Base on combine effect of all three.

An observation: most of the hardest level maps always have the longest solution, (Like 35 steps or more). Consequently, we are convinced that the length of solution should have a direct effect to its difficulty. The first trial to develop fitness function purely based on **shortest-path solution**. We built a Map-Solver Agent for calculating each map's shortest-path cost. And then assigned fitness score base on its cost. Then the performance was tested. Surprisingly, a big shortfall was discovered. The map was evolving in a way of longer solution for each generation, but it wasn't always lead to more difficulty as show in figure 3-A, figure 3-B and figure 3-C

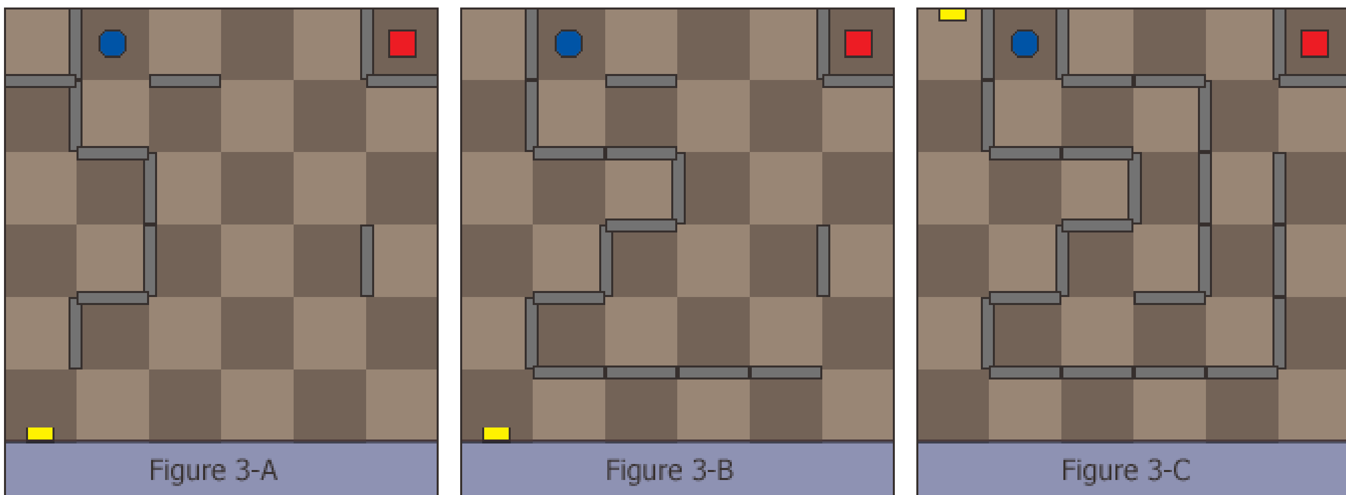

Figure 3-A Figure 3-B Figure 3-C

The first map, figure 3-A, takes minimum of 8 player actions to solve
(S, E, S, S, S, W, S, W)
The second map, figure 3-B, takes minimum of 14 player actions to solve
(E, E, E, S, E, S, S, S, S, W, W, W, W, W)
And the third map, figure 3-C takes minimum of 27 player actions to solve.
(S, E, E, S, S, W, S, E, E, N, N, N, E, S, S, S, S, W, W, W, W, W, N, N, N, N, N)

These three maps are absolutely easy to solve, it contain no trick at all. Just walk right to the exit. As you can see, our GA-fitness function can falsely determine difficulty level of the maps. Because of increasing in solution's length, our fitness function thinks the maps are harder and keep evolving

them whereas it shouldn't. Consequently, the shortest path solution was rejected and we develop fitness function base on complexity of the map itself.

How do we determine the map's complexity?

Since the branching factor and the depth of the search tree for the player is big, there will be many unsuccessful paths which player would consider before determining the right solution. Take a look at two following maps as shown in figure 3-D and figure 3-E

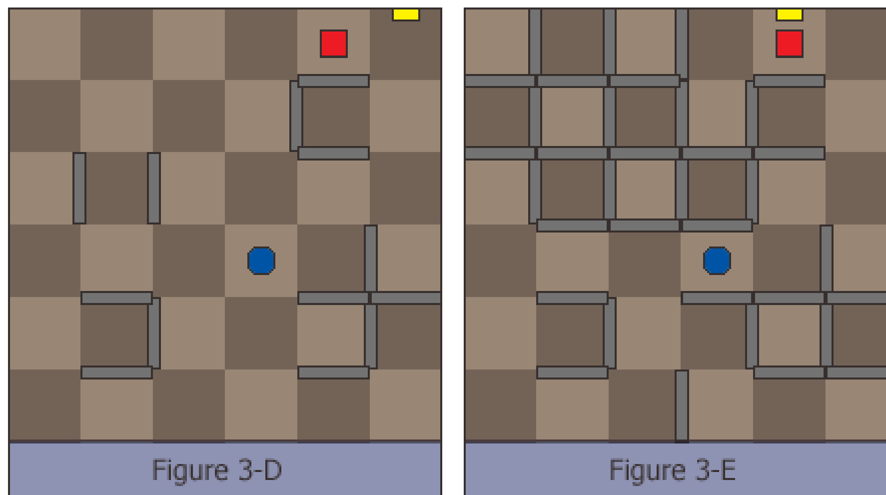
Figure 3-D Figure 3-E

Basically both map had the same trick, to solve. We must trap the enemy in order of tile (4, 4) -> (5,2) -> (1,1). But notice that the first map is harder to solve, typically need more attempts to solve it, because it gives the player many choices of actions. In comparison to the second map which filled with walls, forces the player to move along at limited actions.

In other words, in a map which fills with lots of walls, the agent would not do a lot of branching to get to the solution, thus **fewer visited state**, drained up less memory and solving the map quicker result in less complexity

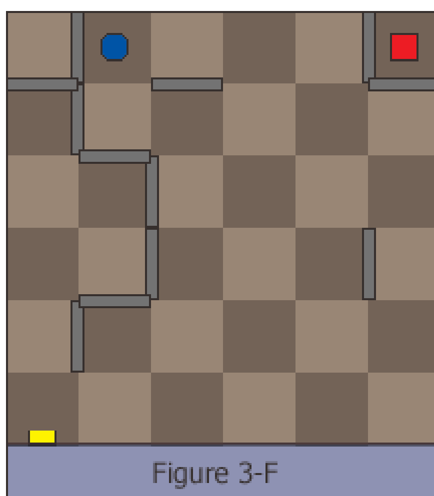
Figure 3-F

Thus another fitness function was developed based on **complexity** of the map. Then again, we went for testing performances. Another interesting result, the maps were evolving in a way of higher complexity (more visited state for each generation). However, still in many cases, the maps were solved with 20 moves or less. Although, it has many visited state, it is considered very easy map as shown in figure 3-F

Our next fitness function is based on enemy movement factor. This is probably one of the most important factors that contribute to level difficulty of the map. It is obvious that trapping the enemy in a place makes it easier for the player to win. Consequently, the question is how to develop a fitness function based on enemy movement factor to consider such factors.

How do we identify movement factor?

The first attempt was to count the number of places it has been trapped. It looks pretty obvious as we count it from visualization viewpoint. But to make agent truly understand the meaning of each "unique trap" require extremely complex algorithm. Obviously, simply counting the times that enemy stays idle more than once as single trap is not going to work. It is massively include false counting. A lot of cases have to be detected. For example, the same trap location can means two different traps. And sometime same trap can be started at different trap locations, player positions or both. A simpler alternative is to count the number of enemy move. It is observed that if the enemy gets trapped in a lot of places, he must move a lot too. Although, it is a weaker version of counting, but looks good enough for representing enemy movement factor

So we developed another fitness function based on counting the number of enemy moves in shortest-path solution and tested the performance. To our Surprise, most of the maps were not evolving at all. After awhile, we figured out that to make the map evolve one or more enemy-move further, it really require lots of layout shifting. And such shifting just destroy previous layout it already process. In other word, when it create new trap, it also lost some of the traps it already had. After all, we still believe that enemy movement factor is really important factor contribute to difficulty of the map. But to directly build fitness function based on its score, just not a good way of using it

Our final fitness function was to combine all these factors. We built a formula and tweak its parameter and run the simulation again. It is interesting to see that the maps are now evolving in a way we expected. Each generation the maps tend to become has more shortest-path solution, has more visited state, and has more enemy movement.

Fitness's formula

Total Fitness = ((#action / (6*MapSize)) * 200) +
((#visited / (24*MapSize)) * 100) +
((#enemy move / (5*MapSize)) * 200)

How did we get this combine Fitness's formula?

1. We collected and averaged the score of each parameter from the high-quality maps included action score, visited score, enemy move score. Hard 6x6 maps generally had about 30-40 actions, 135-155 visited state, 20-30 enemy moves
2The factors were weighted best ratio of 2:1:2
3. The MapSize is only purpose for supporting multi-size decoding. We need it because our PuzzleMaze game supports two MapSize (6x6 and 8x8). Solution of 30-40 actions are generally difficult to solve for 6x6 MapSize, however it is easy for 8x8 MapSize (the proper number is 80-100 actions)
4. The parameters were tuned based on trial-error until the satisfactory result is achieved.

Map-Solver Agent overview

First thing before we start building an agent, we must realize that what kind of information our agent needs in order to solve the problem? In this case, "to solve the map", it work exactly the same way as we try to solve the map ourselves. All we need to know is the settings of the map (Player_current_position, Enemy_current_position, Exit_position, and wall layout)

Fortunately, all information already packed within map-genome, so we do not need to gather anything else for our agent. Unfortunately, the agent may not use the wall data directly. We need slightly transform the wall data.

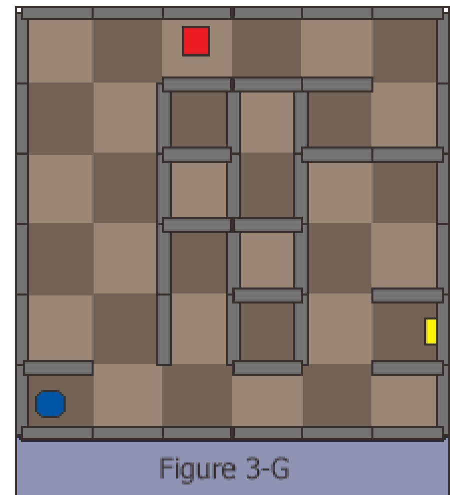
Figure 3-G

In our agent implementation, we did translate the entire wall components into a table called MoveAllowTable (MATable). This table will keep the information concern each position and all valid actions, player/enemy at that position may make (West, East, North, South). An example is shown in figure 3-G

`MATable[0] = (False, True, False, False);` Mean player/Enemy at position 0 can only move East. `MATable[1] = (True, True, True, False);` Mean player/enemy at position 1 can move West, East and North.

This MATable will help the agent easily distinguish between valid action and non-valid action. And only do branching on valid path.

Next, we need to know what kind of output we want. Obviously, we would like a series of actions that lead to solving the map. Example (North, West, North, North, Idle, North, East)

After desire output format, we need to know about game states, consider playing this game for a while; you probably notice that it is just a collection of states. Each state can jump to new state by performing an actions and the objective of the game is to find shortest series of action that will lead player from game_start_state to the game_clear_state without being captured by the enemy, in other word avoid all game_over_states.

The key to solve the problem is to know that game state is progress only by Player_current_position and Enemy_current_position this mean total possible game state (state_space) is only 36 * 36 = 1296 for 6x6 map size which is considering incredibly small.

Once we know the key, to solve state problem is not difficult at all. Basically what we need is do check with the MATable for valid branching action, and keep branching from game_start_state until we reach game_clear_state or unable to continue the branching. And simple Breath First Search (BFS), Depth First Search (DFS) branching algorithm, will do this task just fine.

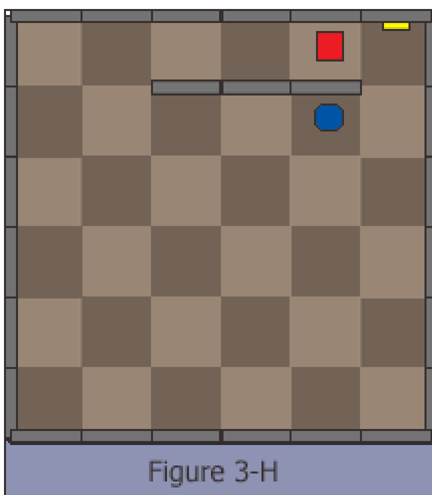
Figure 3-H

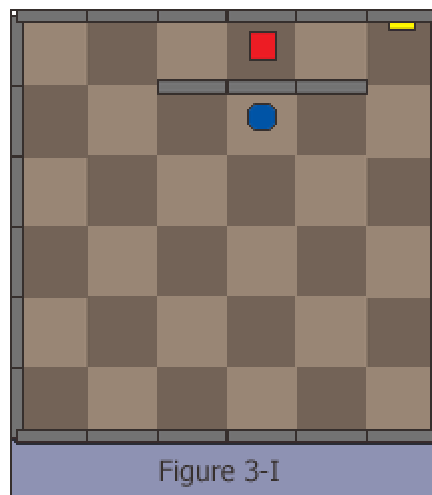
Figure 3-I

The second key, about building a Map-Solver Agent is we have to realize that brute force BFS / DFS is not going to work. Since there can be situation in which 2 states infinitely jump back and forth into each other as shown in figure 3-H and figure 3-I (Loop problem)

These will generally open ways for agent behavior, to avoid being capture from enemy by keep moving back and forth behind the wall infinitely. And the game just not turns into game_over_state or game_clear_state. In other word, neither the branching process reaches the end, nor solution found.

Once we know the basic of unique state, we could write basic structure to keep track of all visited state. When new state is branching, first thing to do is check if it is new state? If true branch it and add it into visited state list, if it is fault just cancel the branching. The problem is solved! All duplicate branches were eliminated and we will never run into the infinite branching problem again.

How does the Agent determine if a map is unsolvable?

It is easy, if he can no longer branch, all the node reach to its leaf, and still not reaches any game clear state, the map is considered "unsolvable".

To summarize it all, in this implementation, the agent utilized Breath First Search algorithm with loop detection (duplicate state avoidance checking). Thus it always finds the highest quality solution if there is one exists. We also implement DFS version of agent for comparing its performance as you can see in the game when you left click on the Help button.

GA-STEP FOUR: Implement Genome Selector (Roulette Wheel Selection)

Genome Selector determines how the parent picking process occurs. We followed the natural GA model. Each mating, it picks two genomes as a parent by Roulette wheel selection method, due to it is quite simple & fair methods, every genome receive chance of being picked according to their fitness score. This mean the genome which has higher fitness score is likely to be picked as parent and continue to breed their genes to next generation.

Here is how it is implemented

1. Count total fitness score from all genomes and line up all genomes on one line
2. Then randomly selected number between (0, total fitness)
3. Determine the gene id according to number selected.

Table III shows example of roulette wheel selection picking process. We have total of six genes, gene id 0-5, with fitness score 3, 5, 7, 15, 2, 8 (total of 40). Then we randomly pick a number between 1 and 40. Suppose we picked 25, this means Gene ID "3" is selected.

Gene's ID	0	1	2	3	4	5
Fitness	3	5	7	15	2	8
Total Fitness	40					
Selected	25					

Convergent Issue

As it was previously mentioned, the higher the quality of the map, the harder it is to evolve. In addition to that, most of the time after mating we just turn very good maps, with 35-50 shortest-path solution into unsolvable maps. This imposes a huge problem, fast convergent issue.

Let's say we have 154 population map-genomes. About 50% of them are unsolvable maps; 35-40% is poor quality maps (these maps had low fitness scores, but potential for evolving into a better map) and 10-15% that is actually good quality map.

To select parent genome purely base on Roulette wheel selection, 50% of unsolvable map is gone. Thus variety of genome is half each generations. With in few generations all the genome becomes exactly the same. This result in dramatically reduce the effect of crossover operator

We tried two ways to handle the unsolvable maps. First, add some criteria to discriminate the unsolvable-map but not totally assign its fitness as zero, just enough to reduce some chance of being select to retain gene variety. Or another way is just do it as usual; the unsolvable map will be assign fitness of zero and become extinct. However each generation added variety by Random another pie of map-genomes from scratch and insert into population.

We decide to use second method in which, for each generation, first randomly create 50% of map-genome and insert directly into babies maps, increase total amount of babies. And then do a roulette wheel selection for another 50% of the babies. The results were really satisfactory.

The need of Elitism

Generally, good genetic algorithm should always include elitism of some form. How elitism work is, before any selection process, first pick a pack of genomes that has highest fitness and send it directly to the population pie of the next generation (babies). This is to ensure we do not lose the best genome we already had so far. If we do not apply elitism, sometime we just lost it, because it has not been selected. Or in another case, it is selected, but after Crossover it keeps producing unsolvable babies' map-genome.

GA-STEP FIVE: Mating (Generate new Babies genomes)

This step is the core of genetic algorithm. It is the phase we made actual production of next-generation maps. Fundamentally, we use function implemented in previous step, Genome-selector, to select two genomes at a time, we called it as Dad-genome and Mom-genome. Then we apply Crossover according to Crossover Rate, and apply Mutation according to Mutation Rate in order to generate 2 baby-genomes. (Note that Crossover Rate determine how often Crossover operation occurs, Mutation Rate determine how often Mutation occurs)

During Crossover, 2 new baby-genomes are created based on both Dad's gene and Mom's gene by follow Crossover bit patterns algorithm. Start from the first bit of the baby1-genome's data, randomly selected whether it should acquire bit's data from dad or mom. Then we assigned the other parent bit's data to baby2-genome's data and continue to the next bit. Repeat the process until finish assigning the last bit of genome-data.

Crossover Algorithm

Determine if Crossover necessary?

```
// if random number greater than crossover rate or mom equal dad --> no crossover
If ((GetRandomProb(1000) > m_dCrossoverRate) || (Mom == Dad))
{
    // just copy mom to baby1 and dad to baby2
    Baby1 = Mom; Baby2 = Dad;
}
Else
{
    // Applied crossover operator;
    For each bit of genome data, determine if the bit will crossover from dad or mom by follow
    Determine Multipoint Crossover bit pattern algorithm
}
```

Determine Multipoint Crossover bit pattern

```
For (bit = 0; bit < 77; bit++)
{
    If (random number between 0 and 1 equal zero)
    {
        Current bit of baby1 crossover from dad genome;
        Current bit of baby2 crossover from mom genome;
    }
    Else
    {
        Current bit of baby1 crossover from mom genome;
        Current bit of baby2 crossover from dad genome;
    }
}
```

The following example shows how successful Crossover might be occurred. Initially we selected 2 genomes from populations (Dad's genome & Mom's genome) and applied Crossover operator.

Mom	pPos	0	0	1	0	0	0																													
	ePos	1	0	0	0	1	0																													
	exit	1	0	0	1	0																														
	h_wall	0	0	0	0	0		1	1	1	0	0		0	0	0	0	0		0	0	0	1	0		0	1	0	1	1		0	1	0	0	0
	v_wall	0	0	0	1	0		0	1	1	0	1		0	0	0	0	1		1	1	0	0	0		0	0	0	1	1		0	0	0	0	0

Dad	pPos	0	1	0	1	1	1																													
	ePos	1	0	0	0	1	0																													
	exit	1	0	0	1	1																														
	h_wall	0	0	0	0	0		0	0	0	1	0		0	0	0	1	0		0	1	0	0	0		1	0	1	1	1		0	1	0	0	0
	v_wall	0	1	0	0	0		0	1	0	0	1		1	0	0	1	1		1	1	0	0	0		0	0	0	1	0		0	0	0	0	0

Figure 5-A and figure 5-B shows what Maps look like after decoding Mom's genome and Dad's genome accordingly.

Figure 5-A

Figure 5-B

Mom's Stat
Solution: ISWWE ENNNN ENE
Fitness (13, 9, 112): 210.000

Dad's Stat
Solution: WWWSS SEEEW WWNNW SWNNI SNNEE ENEE
Fitness (29, 15, 138): 356.944

Determine Multipoint Crossover bit pattern

Baby-1	Bit **0-16**	M	M	M	D	D	D	M	D	M	D	M	M	M	M	D	D	D													
	Bit **17-46**	D	M	M	M	D	M	M	D	M	D	M	M	D	M	D	M	M	D	D	M	D	M	M	D	D	D	M	M	M	D
	Bit **47-76**	D	M	M	D	M	M	M	D	M	M	M	D	D	M	D	D	M	M	M	D	M	M	M	D	D	M	D	M	D	M

Baby-2	Bit **0-16**	D	D	D	M	M	M	D	M	D	M	D	D	D	D	M	M	M													
	Bit **17-46**	M	D	D	D	M	D	D	M	D	M	D	D	M	D	M	D	D	M	M	D	M	D	D	M	M	M	D	D	D	M
	Bit **47-76**	M	D	D	M	D	D	D	M	D	D	D	M	M	D	M	M	D	D	D	M	D	D	D	M	M	D	M	D	M	D

Baby-1	pPos	0	0	1	1	1	1																													
	ePos	1	0	0	0	1	0																													
	exit	1	0	0	1	1																														
	h_wall	0	0	0	0	0		1	1	0	0	0		0	0	0	0	0		0	0	0	0	0		1	1	0	1	1		0	1	0	0	0
	v_wall	0	0	0	0	0		0	1	0	0	1		0	0	0	0	1		1	1	0	0	0		0	0	0	1	0		0	0	0	0	0

Baby-2	pPos	0	1	0	0	0	0																													
	ePos	1	0	0	0	1	0																													
	exit	1	0	0	1	0																														
	h_wall	0	0	0	0	0		0	0	1	1	0		0	0	0	1	0		0	1	0	1	0		0	0	1	1	1		0	1	0	0	0
	v_wall	0	1	0	1	0		0	1	1	0	1		1	0	0	1	1		1	1	0	0	0		0	0	0	1	1		0	0	0	0	0

Figure 5-C and figure 5-D shows what Maps look like after decoding Baby1's genome and Baby2's genome accordingly.

Figure 5-C

Figure 5-D

Baby's stats

Baby1 Genome
Solution: ENEWW SSSEE IWWWW WNNNS SSEEE NNENE NN
Fitness (32, 15, 152): 383.333

Baby2 Genome
Solution: SSEIW NWSWN NWSWN NNEEE NE
Fitness (22, 10, 97): 256.25

As you can see from baby1 and baby2's stats, they both become more successful than their parent. Not only they have higher fitness score, they are definitely harder to solve.

Next step, we applied Mutation operator. During Mutation, we start from first bit of newly created baby's genome data. Pick one random floating number between 0 and 1, if the number is less than or equal Mutation Rate, altered the bit with opposite value (ie. 0 to 1, 1 to 0). Repeat the step until the last bit genome data

Mutation Algorithm

```
// testing each bit of new born baby
For (bit = 0; bit < 77; bit++)
{
    If (random floating number between 0 and 1 less than or equal Mutation Rate)
    {
        // flip current bit data
        If (current bit data equal 0)
            Then set bit data to 1
        Else
            Set bit data to 0
    }
}
```

Determine mutation bits

Baby1 mutated bit: 21
Baby2 mutated bit: none

Baby-1																																			
pPos	0	0	1	1	1	1																													
ePos	1	0	0	0	1	0																													
exit	1	0	0	1	1																														
h_wall	0	0	0	0	**1**		1	1	0	0	0		0	0	0	0	0		0	0	0	0	0		1	1	0	1	1		0	1	0	0	0
v_wall	0	0	0	0	0		0	1	0	0	1		0	0	0	0	1		1	1	0	0	0		0	0	0	1	0		0	0	0	0	0

Figure 5-E and figure 5-F shows what Maps look like before and after Baby1's genome passing mutation process.

Figure 5-E

Figure 5-F

Baby1 Genome
Solution: ENEWW SSSEE IWWWW WNNNS SSEEE NNENE NN
Fitness (32, 15, **150**): **381.944**

As previously stated about mutation effect in Overview section, it barely cause any change to fitness of the map.

Note that the Mutation Rate, equal to 0.0001, is hard coded to the system. This means roughly one bit will be mutated from one thousand bits testing. One thousand bits are about THIRTEEN 6x6

maps or EIGHT 8x8 maps. Considering we have population of 154 and 258. This is approximately 10 mutations on 6x6 maps, and 32 mutations on 8x8 maps for each generation.

GA-STEP SIX: Repeat Step four until we generate total babies equal to number of population

GA-STEP SEVEN: The end of current generation, if wanted to continue simulate more generation, assign new population equal to babies genomes and repeat step three

Conclusion

This study shows that AI approaches such as genetic algorithms can be used to design and to play games more intelligently. We used genetic algorithm to design the hardest Thesues-Minotaur maze. In this process we analyzed 3 different fitness functions and realized that the combination of the 3 will be a better fitness function than each function alone. It is important to mention that finding a better fitness function is an open problem and we are not satisfied that our fitness function is the best of all.

For more information about the Genetic Algorithm & Game model

Web sites

Robert Abbott, http://www.logicmazes.com/theseus.html
Theseus and the Minotaur, http://www.logicmazes.com/theseus4.html

Books

AI Techniques for Game Programming (Premier Press, 2002)
Artificial Intelligence: A Modern Approach (Prentice-Hall, Inc, 1995)