Game AI for Turn Based Strategy Games

Asier Bilbao

1 Introduction

Developing game AI, that is, algorithms that controls Non-Player Characters (NPCs) is a difficult job. Sometimes the AI turns out to be very smart and the game is unfair for the player, and other times the AI is so basic that the player does not suffer any hardship. This research will focus in different methods to create a balanced AI, smart but fair, specifically for Turn Based Strategy Role Playing games (TBSRPGs).

The reason I specify this subgenre of the Turn Based strategy games, is that while both of them are Turn Based, TBSRPGs tend to put a bigger focus on the units, and give less priority to anything else. For this reason, I believe making a good AI for these kind of games can be very beneficial.

In these type of games, two opponents, one of them being the player, and the other one being an AI, takes turns controlling their own units on different maps. Controlling these units involves using commands to instruct them to move somewhere, hold the position, use an item, or attack. Most of the time, the win condition is the total destruction of the opponent.

This research paper will talk about different AI techniques that can be used to implement AI on TBSRPGs or even TBSGs.

2 Game Definition

For this research I will use my own game definition based on Nintendo's Fire Emblem game series, which is one of the most popular TBSRPGs of the world. The version made by me will be called BE (Basic Emblem).

The demo will take place on a two dimensional, tile based map. A tile has its own predefined type, such as Grass, Forest, Mountain or River. Each type has its own set of parameters, which define the characteristics of this tile, such as the number of moves required to traverse it, or the beneficial bonuses the unit receives when standing on it.
The movement of the units is locked to be only vertical or horizontal, never diagonal. The reason for this is that if diagonal movement was possible, is very difficult to balance, so that moving diagonally is not always the most optimal route.

Like tiles, units also come in different type. Each type has different values on its parameters, but they all have the same type of parameters: An amount of health points, number of moves they can make, amount of damage the deal and the actions it can perform.

The three types of units in this game are Saber, Berserker and Lancer. The difference between this units are just numbers, but they may or may not behave the same way.

Each turn the owner of the units can perform a Move or Attack action for each until under their control. A unit can Move and Attack, but not Attack and Move. A tile can only contain a single unit. Additionally, a moving unit is able to pass through a tile occupied by a friendly unit, but no through a tile occupied by an enemy unit.

3 AI Techniques

Now I will talk about the AI of the enemy. I will use a two level system: The high-level, where strategic decisions are made (Which is the best course of action?), and the low-level, where tactical decisions are made (How do I calculate the best course of action?).

With a two level system, it's easier to divide the work. We don't want to each unit to act only thinking of himself, but we also don't want a "master" to control all of them, since that can turn out unpredictable. Of course, this also makes everything more complex, since I have to create and test two different AIs at the same time.

The high level would use a Utility Score System that would get as input the current state of the game, and the analysis, and get a decision. Then I would end up with a strategic decision, which I break it down into multiple steps that each unit would have to perform, such as if the strategic decision is to attack the nearby Archer, the first step for each agent is to find a path to the tile and attack it. This would be the low-level, which would also have its own Utility Based System, to know which would be the best path and attack to do. The path-finding of the agents would be made with Dijkstra, since I also need to calculate all possible paths for the Unit.

The Utility Score System will work with a Choice system. It will always choose the one with the highest score, but I will also make a sub system to take a random one from the top three. Both of these will require testing to see which one is better, if there is one that can truly be considered better. To calculate the score of the Choice, there are some parameters that it will take into account to calculate it.

I will also perform Terrain Analysis and create different influence maps based on different attributes (allies, enemies, defensive points) to use together with the Utility Score System. Some of them would be constants, such as tiles with benefits, while other would change each turn, such as the allies and enemies influence maps.

4 Foundations

Before I start talking about how I implemented the actual AI, I want to explain first what kind of system I am using for the Tiles and Units, since this is used for the AI. Because of this, I will only talk about the parameters that the AI actually uses (for example, Tile has a sf::Sprite variable used to draw, but it's not used for the AI).

4.1 Tile

The map in which the demo will occur is very important, even though is no technically related to AI, the AI will traverse these tiles, and so I have to also take care of it and create a simple map. As said before, the map is tile based, and so I'll go with 11x11 size, big enough for a small skirmish. This is what a Tile would look like:

Table 1 Parameters of the Tile Class

Parameter Name	Type	Usage
m_tiletype	TileType	The type of tile (From best to worst: Moutain,Forest,Grass,River)
m_idx	sf::Vector2i	The position on the 11x11 grid
m_bonus	float	The bonus in combat the tile gives
m_occupant	Unit *	Which unit is currently on it, based on the type of tile.

There is also an extra class called TileMap, which basically contains a 2D array of Tile, an acts as the grid that represents the map. This class also constructs the map by reading a .txt file. It also contains some helper functions and parameters used for the AI:

Table 2 Functions and Paramters of the TileMap Class

Parameter Name	Type	Usage
GetTileCost()	float	Returns the cost to travel to this tile
SetUnit()	void	Set the given unit in its tile
m_grid	Tile**	2D array of Tile
Width	int	Width of the grid, in this case 11
Height	int	Heigth of the grid, in this case 11

4.2 Unit

The unit are the pawns that the player and AI will use to move and attack. But, in the case of the AI, it has a special class, derived from Unit, called EnemyAI.

Table 3 Parameters and Functions of the EnemyAI Class

Paramter Name	Type	Usage
m_owner	OwnerType	Owner of the Unit
m_current_tile	Tile*	Tile in which the Unit is on
m_moved	bool	Know if the Unit has moved
m_attacked	bool	Know if the Unit has attacked
m_unittype	UnitType	Type of Unit
m_max_hp	int	Max HP of the Unit
m_hp	int	Current HP of the Unit
m_weapon_damage	int	Damage of the Unit
m_armor_defense	int	Defense of the Unit

m_tile_movement	int	Number of tile the Unit can move
m_tile_IM_weight	float	Weight for Tile Influence Map
m_ally_IM_weight	float	Weight for Ally Influence Map
m_enemy_IM_weight	float	Weight for Enemy Influence Map
m_nearest_weight	float	Weight for Nearest Enemy
m_fight_weight	float	Weight for Starting Fight
m_health_weight	float	Weight for Enemy Health
m_damage_weight	float	Weight for Combat Damage
m_use_global_parameters	bool	Use the Global Weight instead
CalculateChoices()	void	Calculates ALL possible Choices
m_choices	bool	Contains ALL Choices

4.3 Choice

You probably noticed that there is a parameter called Choice on the EnemyAI. Choice contains some parameters, and represent a possible movement (and/or attack) that the unit can do.

Table 4 Parameters of the Choice Class

Parameter Name	Type	Usage
m_executor	Unit*	The executor of the Choice
m_move_tile	sf::Vector2i	The tile that will be moved to
m_attack_tile	sf::Vector2i	The tile that will be attacked
m_objective	Unit*	The enemy that will be attacked
m_score	float	The score of the Choice

As I talked before in AI Techniques, I will use a Utility Score System. In this case, Choices have a score, and the Strategic AI will choose the best Choice with the best score.

5 AI Implementation

Let's start talking about what everyone wants to know about, how to implement and use the AI Techniques explained before. As I said before, I have a two level AI system, a high level AI, called Strategic AI, and a low level AI, called Enemy AI. I will explain it going from top to bottom.

The Strategic AI (SAI for short) has the job of choosing the best Choice (you can also think of it as a blackboard, Choices are posted, and the best one is chosen). When it's the enemy's turn, SAI calculates all choices that its Units have. Then, it has two options: Either it stores only the top choice of each Unit, or it chooses between a random Top Choice of each Unit and stores it. After that, it will choose between all Choices and pick the best one, or just choose a random one.

The Enemy AI (EAI for short) has the job of calculating all the possible Choices it has, which can be only move, only attack, or move and attack. In my case, I first check if I can attack (or move first and then attack) and then, if I can't attack, try to move. To calculate the score of the Choice, first I use this formula (IM stands for Influence Map):

$$Score = tile\ IM * tile\ weigh + ally\ IM * ally\ weight - enemy\ IM * enemy\ weight$$

This gives the score of the tile. The first parameter represents the value of the tile, which is extracted from the Influence Map of the tile, calculated by taking into account the type of tile (River is -1.0, Grass is 0.0, Forest is 1 and Mountain is 2).

The second parameter represents the influence of an ally in the current tile. This means that the more allies that can go to the current tile exists, the higher influence it has. It's extracted from the Ally Influence Map, which is recalculated every time a Unit moves, and takes into account the possible movement of each Unit. It has an interval of [0.0, 1.0].

The third parameter represents the influence of the enemy in the current tile. This means that the more enemies that can attack the current tile exists, the higher influence it has. Which can be translated as if a tile is dangerous to be on or not. It's extracted from the Enemy Influence Map, which is recalculated after the Player ends its turn, and takes into account the attack range (which means all possible move and attack actions) of each Unit. It has an interval of [0.0, 1.0].

First case to take into account: What if the enemies are outside the Unit's attack range? For this, we add a new value to the score:

$$Score += \frac{1}{nearest\ enemy} * nearest\ weight$$

In this case, nearest enemy represents the distance to nearest enemy. In most cases, we want to get closer to the enemy to kill it, and so this parameter is used for that.

But what if there is an enemy next to it, which means it can be attacked? For this, there is another formula that will be taken into account and added:

$$Score += fight\ weight$$

$$Score += \max(my\ damage - (enemy\ armor + tile\ bonus), 1) * damage\ weight$$

$$Score += \frac{1}{hp} * health\ weight$$

The first one is the priority to fight. Maybe we want to tell the Unit fighting is not that beneficial. The second one is basically the combat formula, and calculates the total possible damage that it will deal to the enemy. The third one calculates how low health the enemy is, which can be used to prioritize a kill.

All of this is then used to create Choices, which contains the final score, which will be used for the Strategic AI.

You probably have noticed that all of these parameters have a weight value multiplying them. All of these weights are used by the Enemy AI, but it's not the AI who sets them, it's done by the User (in this case the Player) instead.

5.1 Weight

These parameters are possibly the most important values of the entire program. Each Unit has its own values, but there is also a global version which can also be used for EAI, contained on SAI. Thanks to these guys, I can create different "personalities" for the EAI by just changing some values. By personality I mean the way it acts to different actions. Here are some examples, including weights.

Table 5 Weight for "Coward" Personality

tile bonus weight	float	1
ally weight	float	1
enemy weight	float	1
nearest weight	float	-1
fight weight	float	-1
health weight	float	-1
damage weight	float	-1

In this example, the AI will always try to avoid conflict, and get far from the enemies. This can be taken to a more extreme, and make "ally weight" also a High Negative, which means that the AI will try to get away from its allies too.

Table 6 Weight for "Defensive" Personality

tile bonus weight	float	20
ally weight	float	1
enemy weight	float	1
nearest weight	float	0.5
fight weight	float	1
health weight	float	1
damage weight	float	1

This personality will try to stay in the best Tile all time, and attack anyone that passes near. This can be useful for a Unit that you want him to stay all the time on a Tile, like a long range Unit. You can also go for a "Highly Defensive" personality by increasing the "enemy weight" or decreasing the "nearest weight".

Table 7 Weight for “Aggressive” Personality

tile bonus weight	float	0
ally weight	float	0
enemy weight	float	0
nearest weight	float	5
fight weight	float	5
health weight	float	1
damage weight	float	1

This one will completely ignore everything and just go the nearest enemy and attack it. From this, you could side grade it to a “Killer” personality, by increasing furthermore the “health weight”, making it so that killing it’s is highest priority.

6 Conclusion

The goal for this project was to find a way to create an AI that was smart but fair. By using the Utility Score System combined with Influence Maps, while also being able to change the weight parameters, I can create a lot of different “personalities”, which can also be considered different levels of difficulty.

It’s also scalable. For example, after implementing everything, I realized that there were some upgrades that could be done to make a smarter AI. Choking Points, for example, could be another thing to keep in mind when creating the Tile Bonus Influence Map.

Of course, this system has its flaws. If there is a case where you want few but complex AIs, you are probably better off using another method. While this method does create a “smart” AI, its intelligence level is relative, maybe it’s good enough for whatever project you want to use it in, or maybe you need a more specific one, in which case is better to not try generalize so much as I did.

Also, it relies a lot on trial and error. You may know what kind of AI you want, but you have to test the numbers to get what you want.

7 References

[Buro & Furtak] Michael Buro & Timothy M. Furtak. RTS Games and Real–Time AI Research. https://skatgame.net/mburo/ps/BRIMS-04.pdf
[Robertson & Watson] Glen Robertson & Ian Watson. A Review of Real-Time Strategy Game AI. https://www.cs.auckland.ac.nz/research/gameai/publications/Robertson_Watson_AIMag14.pdf
[Bergsma & Spronck] Maurice Bergsma & Pieter Spronck. Adaptive Intelligence for Turn-based Strategy Games. https://pdfs.semanticscholar.org/2a90/3c08097bc55e99e4ae805b471078df7d72d5.pdf