Sunday, July 28, 2013

Decision Modeling and Optimization in Game Design, Part 4: Player-vs-Player Class Balancing

This article is the fourth in a continuing series on the use of decision modeling and optimization techniques for game design. The full list of articles includes:

- *Part 1: Introduction (Blogspot) (Gamasutra)*
- *Part 2: Optimization Basics and Unrolling a Simulation (Blogspot) (Gamasutra)*
- *Part 3: Allocation and Facility Location Problems (Blogspot) (Gamasutra)*
- *Part 4: Competitive Balancing Problems (Blogspot) (Gamasutra)*
- *Part 5: Class Assignment Problems (Blogspot) (Gamasutra)*
- *Part 6: Parametric Design Techniques (Blogspot) (Gamasutra)*
- *Part 7: Production Queues (Blogspot) (Gamasutra)*
- *Part 8: Graph Search (Blogspot) (Gamasutra)*
- *Part 9: Modular Level Design (Blogspot) (Gamasutra)*

The spreadsheet for this article can be downloaded here: link

Spreadsheets vs Simulations

In the previous three articles in this series, we introduced the concept of decision modeling and optimization and the Solver tool available in Excel, and we show how they can be used to figure out the optimal tax rates for a city in a 4X strategy game, determine the optimal placement of wormhole teleporters in a space game, and figure out the optimal weapon loadout for the SuperTank problem explained in Part 1.

A question naturally arises: what about game balancing? Is it possible to apply these sorts of techniques to the types of difficult balancing problems that you typically find in many different types of games, particularly strategy games, RPGs, and MMORPGs?

The answer is yes, of course, but with a lot of caveats. Spreadsheets in particular have a lot of limitations because in most non-trivial cases they can't accurately represent our game. This will make it difficult for us to do any robust balancing using optimization techniques; the real balancing problems in the vast majority of games will be well outside the scope of what we can model in a spreadsheet. The simulation of a game alone is usually too complex, and has too many moving parts, and is usually in real-time, which throws up obstacles to any sort of discrete simulation we might want to do.

So if we wanted to try to use these kinds of techniques to balance classes in an MMORPG like *WildStar* or a strategy game like *Planetary Annihilation*, we'd have to integrate them with the game simulation itself in order for it to be in any way accurate or useful.

Also, there is the fact that there are some aspects of balancing can't be automated; as we stated in the Disclaimers section of the first article, there's no way to automatically tune a game's "feel."

As a result, the best we can hope to do here is show a simple example to illustrate the overall approach to this sort of problem: how you would go about framing this type of balancing problem and optimizing it with a simple example in Excel. We'll show that at least for a simple combat example, Solver can do a remarkably good job of balancing several RPG classes against each other. You can then use that basic structure to serve as a basic framework to structure that type of optimization problem in a broader setting more deeply rooted in a game's simulation.

Having said that, though, we hope you'll bear with us through all the caveats and see where a simple example can get us.

"Balancing" is Undefined

There is no single, generally accepted definition for the word "balancing." It has many meanings, and the true meaning usually depends on the context of the game in question. In different settings, "balancing" can refer to setting multiple character classes to be equivalent in capabilities in a role-playing game or team-based action game, balancing some number of opposing forces against each other in a strategy game, and/or adjusting the costs of different units or resources to be equivalent to their utility (among others).

The best definition of "balancing" usually depends on the design goals of the game in question – but since those design goals could be just about anything, there's no way to determine *a priori* what balancing really means for games in general outside of a particular game or genre.

Some players tend to assume that balancing in combat means equal damage. This is particularly the case in massively-multiplayer role-playing games (MMORPGs), where players frequently complain that the damage per second (DPS) of one class is too low or too high relative to another.

Of course, classes can't be balanced on DPS alone; it's perfectly valid for one class to have higher DPS than another, and compensate for it with other factors that limit the overall utility of that class, such as lower survivability or lower long-term DPS compared to short-term burst DPS.

The Tiny MMO

Imagine we're creating a new game, a very simplified massively-multiplayer online role-playing game called "Tiny MMO." As part of our initial design, we are trying to balance four classes for player vs player ("PVP") combat such that despite being different, all four classes are relatively equally capable in combat against all others, and there is no clear "best" or "worst" class to select against the other classes.

Although "Tiny MMO" is a real-time game, each player action lasts exactly 3 seconds, so we can discretize it by representing it as a turn-based game where each "turn" is 3 seconds' worth of gameplay.

In this game, players will be able to play as any of four different character classes:

Blog Archive

▶ 2015 (5)
▶ 2014 (3)
▼ 2013 (9)
▶ December (1)
▶ September (2)
▶ August (2)
▼ July (4)
Decision Modeling and Optimization in Game Design,...
Decision Modeling and Optimization in Game Design,...
Decision Modeling and Optimization in Game Design,...
Decision Modeling and Optimization in Game Design,...

Contributors

- Paul Tozour
- Sensai Pose
- Unknown

That's all we know about these four classes, and we need to set up the initial health ("HP"), damage, healing, and attack range parameters on all four classes. We need to balance them such that each class is unique and its stats are significantly different from all of the other classes, but each class also ends up as "balanced" as possible against all three of the other classes.

In other words, we are trying to optimize the following table:

Decision variables

	HP	Damage	Healing	Range
Warrior	50	10	0	40
Mage	50	10	0	40
Healer	50	10	0	40
Barbarian	50	10	0	40

For now, we're using placeholder values, and assuming each class starts with 50 hit points, does 10 damage per turn when attacking, heals 0 points per turn, and has a range of 40 meters. Each unit moves 10 meters per turn. Since the design specifies that all four character classes can move at the same speed, we'll consider this to be fixed and we won't put movement speed into the decision variable table.

Obviously, this is a toy example with a very simplified model for damage. This is a continuous average damage-per-turn value, which ignores the burst-damage vs effective long-term damage that comes with mana or other mechanics that modify classes' attack capabilities. There is only a single damage type, which is quite unrealistic since most classes will have several if not dozens of damage types, and we'd need to implement an AI system to decide what attack to use on any given turn. Also, damage has a random element in most games, but we'll ignore this for now and assume that the variance of the damage will not be large enough to meaningfully alter the outcome of combat between any two classes.

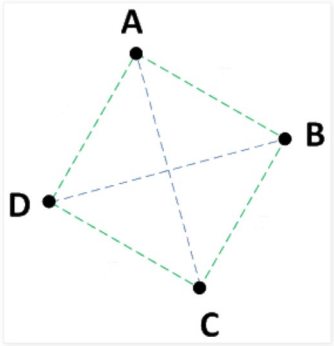
Of course, any balancing we do in Excel isn't likely to be perfect or match the game's final balancing; it's going to have to go through a lot of playtesting. But if we can take an hour or two to quickly get a *good* first pass for this simple game in Excel, at the very least it's likely to get us that much closer to solid settings for the initial balancing pass, which will get us that much closer to where the balancing ultimately needs to be.

The Victory Table

We need to balance the four classes against each other in one-to-one combat. Since we have only 4 classes (Warrior, Mage, Healer, and Barbarian), there really are only 6 possible 1-to-1 pairings of different classes:

- Warrior vs Mage
- Warrior vs Healer
- Warrior vs Barbarian
- Mage vs Healer
- Mage vs Barbarian
- Healer vs Barbarian

This kind of balancing can be remarkably difficult. Even with the fairly simple case we have of four classes, it creates six inter-class relationships, like the six lines that can be drawn between four points on a square.



Any time we want to make even a small change to one of the parameters of any of the other classes, that change will also affect the PvP balancing between that pair of classes and the *other* two classes. This exponential interconnectedness can only grow as we add more classes, and decisions based on individual PvP balancing between any pair of classes can be very dangerous when considered in a vacuum, without taking the whole of the class balancing situation into account.

Ideally, what we'd like to do is have some sort of *victory table* like the one below. If we could model the combat between each of these 6 pairs of units in the spreadsheet, we could then generate some kind of a "score" variable for each of the 6 pairings. Since higher scores are better, we could then combine all six of those scores to generate our objective function.

Victory table

	Warrior	Mage	Healer	Barbarian
Warrior	0	0	0	0
Mage	6	0	0	0
Healer	6	13	0	0
Barbarian	14	15	16	0

The "Combat Sim"

We'll set up each pair of classes against each other at 100 meters. Each character takes 3 seconds to attack, so we can represent this as a turn-based simulation where each "turn" represents 3 seconds. Every "turn," each unit will either attack the other unit if it is in attack range, or keep walking to attempt to close the gap if it isn't.

The simulation looks like this:

Class number / name						Class number / name				
1	Mage					2	Healer			
Range	Max Range	Healing	HP	Damage	Attacks?	Max Range	Healing	HP	Damage	Attacks?
100	65	0	61	10	0	50	5	42	10	0
90	65	0	61	10	0	50	5	42	10	0
80	65	0	61	10	0	50	5	42	10	0
70	65	0	61	10	0	50	5	42	10	0
60	65	0	61	10	1	50	5	42	10	0
55	65	0	61	10	1	50	5	37	10	0
50	65	0	61	10	1	50	5	32	10	1
50	65	0	51	10	1	50	5	27	10	1

Across the top, it shows the pair of units that are squaring off: in this case, the Mage (class #1) and the Healer (class #2). Along the left side, it shows the current distance between the two simulated units.

For each unit, the columns are as follows:

- **Max Range**: This is the maximum range at which the unit can attack. It's drawn directly from the yellow decision variables in our decision variable table.
- **Healing**: This is the unit's rate of healing per turn, drawn directly from the decision variable table.
- **HP**: This is the unit's hit points each turn. It begins as the corresponding HP value in the decision variable table, but is reduced over time as it's attacked by the other unit. It's also raised every turn by the amount of healing it can apply to itself each turn.
- **Damage**: This is the amount of damage the unit applies to the enemy unit per turn when in range. It falls to 0 when the unit dies.
- **Attacks?**: This tests whether the unit is in range. If so, it will be '1,' indicating that the unit attacks this turn; if not, it will be '0,' indicating that the unit moves closer to attempt to reach the other unit instead.

In this way, both units will start moving toward each other, and then begin attacking until one or both are dead. Since each unit moves at 5 meters every 3 seconds (5 meters per "turn"), the "Range" will be reduced by 10 each turn that both units are walking toward each other, and 5 in each turn that only one unit is moving toward the other. The game itself is designed so that both units can initate a move at the same time and then the move is resolved simultaneously, so it's entirely possible for both units to end up dead simultaneously.

Next, we need to set up the scoring for this table, and generate a numeric value that indicates how "good" the combat was - in other words, how close it is to reaching our design goals.

Clearly, we want both units to be dead by the end of combat - or at least, we want both of them to be as close to dead as possible. If combat is balanced, then both of the competing classes should be able to get one another's health down as far as possible by the end of their combat.

However, that, by itself, isn't going to be enough. If we only score in that way, the optimizer is just going to ramp up the damage values as high as possible, so that both units kill each other instantly! (If you're curious, try modifying the attached spreadsheet so you can see it for yourself). Clearly, instant death is not what we want: we want both units dead or near-dead by the end of combat, but we *also* want combat to last a reasonable amount of time.

In other words, we're not only trying to ensure that all classes are relatively evenly balanced against each other; we're also trying to balance them to be *interesting*, and part of that is ensuring that fights will last the right amount of time.

In order to generate that score, we set up a few cells just to the right of each table. *Duration* indicates how long the combat lasted; it counts the number of rows in the table in which *both* units were still alive. *Total HP* counts the total combined hit points of the two surviving units - ideally, we want this to be 0, meaning that both units are dead by the time combat is over.

Duration	12		
Total HP	1	Score	6

Finally, "Score" combines duration and total hit points, as (Duration / (1 + Total HP)) -- note that we add the "+1" to the denominator since Total HP can be 0, and in this case, we would have a denominator of 0, and thus a divide-by-zero error, unless we added something to the denominator. In this way, we can ensure that we reward the optimizer for finding a maximum combat duration *and* a minimum final combined hit point value.

(Note that since there are 17 rows in each "simulation" of class vs class combat, this means we are effectively making a design choice here that combat should last roughly 17 rounds. If we want combat to be longer or shorter, we can change the number of rows, adjust the scoring formulas to match, and re-optimize.)

Finally, we take these six "Score" values (one for each table) and use them in our "Victory table" above, to show the results of combat between any pair of classes.

Adding Constraints

We're still not done yet. If we optimize the decision model with the current settings, it will probably not set up our classes correctly - in fact, it will probably just give them all identical HP, Damage, Healing, and Range values in our decision variable table.

And what we want, of course, is for each class to have its own distinct identity. We want the Warrior to do the most Damage, the Mage to have the highest Range, the Healer to have the highest Healing, and the Barbarian to have the highest HP. And we also want to ensure that these margins aren't small, either - we want these classes to be different from each other by a substantial amount.

In order to ensure this, we'll set up a small "constraints table." This table will ensure that each of our four classes has the appropriate attribute, and then scores a 0 or a 1 depending on whether the constraint is satisfied.

Constraints		*Min difference*
1	Warrior has max damage	4
1	Mage has max range	10
1	Healer has max healing	4
1	Barbarian has max HP	5

The "Min difference" table on the right specifies the minimum difference in each attribute for that class against all the other classes. In other words, the Warrior must have at least 4 more HP than any other class, the Mage has to have a range at least 10 longer, and so on.

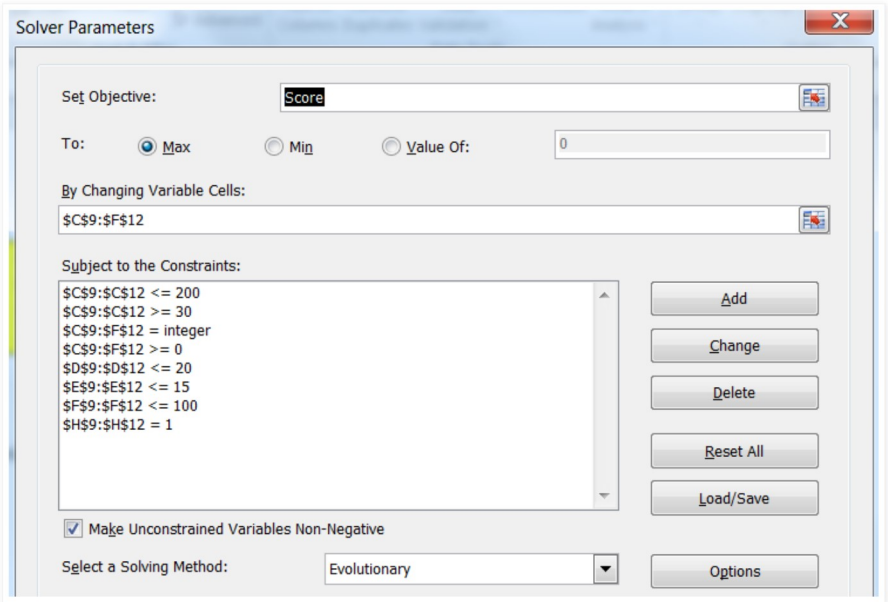
Now that we've added these special constraints, it's time to optimize!

Solving

Now we can run Excel's built-in Solver to try to optimize the initial parameters (see Part 2 for an introduction to Solver and an explanation of how to enable it in Excel). We set the "Objective" cell to be our "Score" cell that combines the results of all six of the tournaments. We set the decision variables to encompass all 16 cells in the yellow "Decision variables" table we set up at the beginning.

We also set up the constraints (in the "Subject to the Constraints" box) as follows:

- All of the decision cells must be integers with a minimum value of 0.
- All of the cells in the "HP" column have a maximum value of 200 and a minimum of 30.
- All of the cells in the "Damage" column have a maximum value of 20.
- All of the cells in the "Healing" column have a maximum value of 15.
- All of the cells in the "Range" column have a maximum value of 100.
- Also, all four of the cells in the special "Constraints" section we set up must have a value of 1 to ensure that these special constraints are satisfied.



Finally, making sure the "Solving Method" is set to "Evolutionary," we run Solver. Note that since this is an evolutionary solver, you may be able to improve the solution you find by running the Solver a second or third time after it finishes, or tweaking the settings (via the "Options" button) for evolutionary optimization.

We should now end up with something like this:

Warrior	71	15	0	43
Mage	61	10	0	65
Healer	42	10	5	50
Barbarian	110	10	0	43

1	Warrior has max damage	4
1	Mage has max range	10
1	Healer has max healing	4
1	Barbarian has max HP	5

Victory table

	Warrior	Mage	Healer	Barbarian
Warrior	0	0	0	0
Mage	6	0	0	0
Healer	6	13	0	0
Barbarian	14	15	16	0

420	**Score**

... And as if by magic, Solver has given us a good initial balancing pass.

As you can see, the Warrior now does the most damage, the Mage has the greatest range, the Healer has the most healing, and the Barbarian has the most HP. Also, you can scroll down to the individual class-vs-class tournaments see how the classes fared against each other; as you can see, most of them are very evenly balanced, with either both classes dying by the end of combat, or one class only barely surviving. Also, all of the tournaments are reasonably long-lasting, with none of the classes "one-shotting" any of the others.

Not bad for a few hours of work, is it?

Conclusion

In this example, we've set up a simple balancing problem and shown that you can, in fact, solve it through simulation and optimization. Though obviously a toy example, it shows the power of decision modeling and optimization techniques and could also serve as an inspiration for more exhaustive balancing tools more tightly integrated with your own game simulation. If nothing else, we hope it will serve as a guide for how to frame this type of problem in practice.

In the next two articles in this series, we'll delve into assignment problems, which involve selecting the optimal assignments between two or more sets of entities. We'll show how to solve these types of problems and illustrate how we used that approach to design the towers in our iOS/Android strategy game *City Conquest*.

-Paul Tozour

Part 5 in this series is now available here.

This article was edited by Professor Christopher Thomas Ryan, Assistant Professor of Operations Management at the University of Chicago Booth School of Business.

Posted by Paul Tozour at 8:56 AM

Labels: Part 4

4 comments:

Unknown July 28, 2013 at 4:48 PM

Great Series Paul,
Any information on the three engines used in solver and where they perform best and worst?

Also curious of you ever combine these results with a two way table(s), map size, map terrain, etc.?

I have found recently in combined heat and power (CHP) project that the optimized output has "cliffs" on some of the input variables that if those inputs the optimized solution is way off.

i.e. in this example if the board size went from 50 to 49 the barbarian or warrior might become a dominate piece

Thanks,
KK

Reply

Replies

Paul Tozour July 29, 2013 at 6:13 AM

Great questions, Kevin!

I'm planning to discuss the different algorithms in Solver (Simplex LP, GRG Nonlinear, Evolutionary) in a future article, though I'm not going to be able to get into too much detail in an article intended for designers.

Re: map size/terrain: class balancing should be independent of the map in general, so I didn't consider it here. There will be an article in the future on using these techniques to optimize map and level design, though.

Reply

Anonymous August 1, 2013 at 8:14 AM

After resetting the decision variables and running the solver, I just couldn't get an appropriate range value for the mage and I believe the respective constraint formula isn't quite right.

Changing the plus into a minus did the trick in my case, so that H10=IF(F10-L10>MAX(F9,F11,F12),1,0)

Thank you, Jacek! You are correct. I've fixed that bug and uploaded a new version.

Reply

Enter your comment...

Comment as: Ahmad Al-Kashef (Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)