Friday, July 12, 2013

Decision Modeling and Optimization in Game Design, Part 2: Optimization Basics and Unrolling a Simulation

This article is the second 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.

Decision Model Setup

Now that we've introduced decision models and explained why they're useful and discussed some of the limitations, we'd like to illustrate the basic concepts with a simple example.

Before we do that, though, we need to introduce some layout and format guidelines. Just as with code, spreadsheets can turn into an unreadable mess quickly if we're not careful.

Broadly speaking, there are four kinds of cells in our spreadsheets:

- **Decision** cells contain the variables we are trying to optimize - in other words, we are going to ask the optimizer to try to find the best possible values for these cells. We will start with 0 or some other appropriate default values in these cells, and then get the optimizer to fill in the correct values for us. In most cases, we'll also constrain them to a certain range, such as being between some minimum and maximum value, and, in some cases, being integers or binary values. For the sake of consistency and readability, decision cells will always be yellow and will be framed with a black outline.
- **"Pre-Defined"** cells are those whose value has been stated directly in the problem statement. For example, if a problem tells us that a Tootsie Pop has 17 grams and each lick removes 0.25 grams, then these two cells will be "defined." We color predefined cells in blue.
- **Calculation** cells are cells whose value is computed from other cells in the spreadsheet, and which don't fit into any of the other categories. We color them in some shade of grey.
- The **Objective** cell (or "output" cell) is the cell whose value we are trying to minimize (or maximize) when we run the optimizer. In our examples, there will only ever be a single objective cell, and it will always be colored orange and framed with a black outline. (*Note:* There are some advanced solvers that let you support multiple objectives, but that's getting too complicated for what we need to do.)

When we run the optimizer (the "Solver" tool built into Microsoft Excel), it's simply going to look at the objective cell that we identify, and then attempt to tweak the decision variables however it can (within the constraints we set up) to either minimize or maximize the value of that objective cell (whichever one we specify).

Solver doesn't know very much about what sort of calculations are going on under the hood or what the connection is between the decision cells and the output cells; it simply runs one of several available algorithms to try to minimize or maximize the value of the objective cell through a principled search of possible values of the decision cells. These algorithms ("Simplex LP," "GRG Nonlinear," "Evolutionary") are designed to be much smarter than brute force exploration of all possible choices for the decision variables, and they very often answer large problems with surprising efficiency.

For example, if we wanted to know how many licks it took to get to the center of a Tootsie Pop, we might set up a spreadsheet that looked like this:

55	Licks it takes to get to the center of a Tootsie Pop
17	Mass of a Tootsie Pop (grams)
0.25	Mass removed per lick (grams)
13.75	Total mass removed by all licks (=Licks x Mass removed per lick)
3.25	Mass remaining on Tootsie Pop (grams) = (Mass of a Tootsie Pop) - (Total mass removed by licks)

We could ask Excel's Solver to solve this, telling it to minimize the "Mass remaining on Tootsie Pop" objective cell, and it would quickly find by experimentation that the value of the yellow decision cell that does this ("Licks it takes to get to the center of a Tootsie Pop") is 68.

This is a little bit silly, of course, because it's obvious from the problem statement that the answer is 17/0.25=68. There's no need to run an optimizer tool to solve a problem you can solve with basic arithmetic instead.

In practice, though, most of the problems we face won't have simple mathematical solutions. They will have multiple decision variables that map to an objective in non-obvious ways, and the mapping between the decision variables and the output will be too complicated to solve by hand with any easily-identified set of mathematical equations (and again, complicated math is something we will studiously avoid in this series).

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

Example 1: Taxes

For our first real decision model, we'll show an example of figuring out an optimal tax rate. Nobody likes taxes, but in this case, we're receiving taxes instead of paying them, so hopefully that will lessen the pain a bit.

Imagine that we're building a "4X" strategy game similar to *Sid Meier's Civilization*. We're in the process of designing cities that have a certain level of unhappiness based on their size. "Unhappy" citizens are essentially non-cooperative, and we do not get any tax revenues from them. We can also attempt to raise money from our cities by adjusting our tax rate on a city, but the city's unhappiness level will grow exponentially as we raise our tax rates, and this might make very high tax levels counterproductive.

We'll also assume that we can set the tax rate at 10% increments anywhere between a 0% and a 50% tax rate. Here's a screenshot showing a similar system from the classic "4X" strategy game *Master of Orion 2*:

As designers, we want to ask a simple question: what's the optimal tax rate in the general case?

This should be a simple problem, since there are only 6 possible tax settings. We could just test each of the 6 settings by hand, find the one that gives us the highest revenues, and be done with it!

(In fact, we could probably find a mathematical equation to solve this, as with the Tootsie Pop example above, but that would be counterproductive since we're setting this up to grow into a more complicated model that can't be reasonably solved with equations. And we're all about avoiding math in this series, anyway.)

Let's start by setting up the problem this way:

- We have a city of size 12 (representing 12 million people). Those people are represented as 12 discrete "citizens."
- Each citizen either be Happy or Unhappy at any given moment.
- Happy citizens pay (tax rate x 10) in taxes (so that, for example, a 20% tax rate gives 2 currency units in tax revenue per Happy citizen).
- Unhappy citizens do not pay taxes.
- The city has 3 Unhappy citizens who will remain Unhappy regardless of the tax rate.
- An additional number of citizens will become Unhappy based on the following formula: (Population) x ((Tax Rate) x (Tax Rate)) x 3.5, rounded down to the nearest integer. For our size-12 city, this will give us 0 additional Unhappy citizens at 0% and 10% taxes, 1 additional Unhappy citizen at a 20% taxes, 3 additional Unhappy citizens at 30% taxes, 6 at 40%, and 10 at 50%.

Simple, right?

We set this up in the attached spreadsheet as follows:

Tax Level (0-5)	2					
Tax Rate	20%		Tax Revenue	16		
Population (millions)	**Base Unhappiness**	**Tax Rate**	**Unhappy Citizens Due To Taxes**	**Unhappy Citizens**	**Happy Citizens**	**Taxes (From Happy Citizens)**
12	3	20%	1	4	8	16

You may have noticed that we set up the yellow decision cell ("Tax Level (0-5)") as an indirect way of specifying the tax rate. Rather than specifying the tax rate directly in the decision cell, the "Tax Rate" calculation cell instead takes the Tax Level number from the decision cell and multiplies it by 10%. There is a good reason for doing it indirectly in this way, as we'll see in a moment.

Now, we can experiment, and plug in all six possible values for the tax level. We can just type in each of the digits from 0 to 5 in the "Tax Level" cell, and we get the following:

Tax Level	Tax Revenue
0%	0
10%	9
20%	16
30%	18
40%	12
50%	0

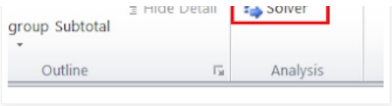
As you can see, there is an optimal setting for the tax rate: 30%, which maximizes our tax revenue at 18 units.

Let's Automate This!

That was nice, but what if there were more than just six possibilities? What if there were hundreds of possible tax rate settings, or if we had other decision variables we needed to tweak, too? Things would get too complicated to test all the values by hand.

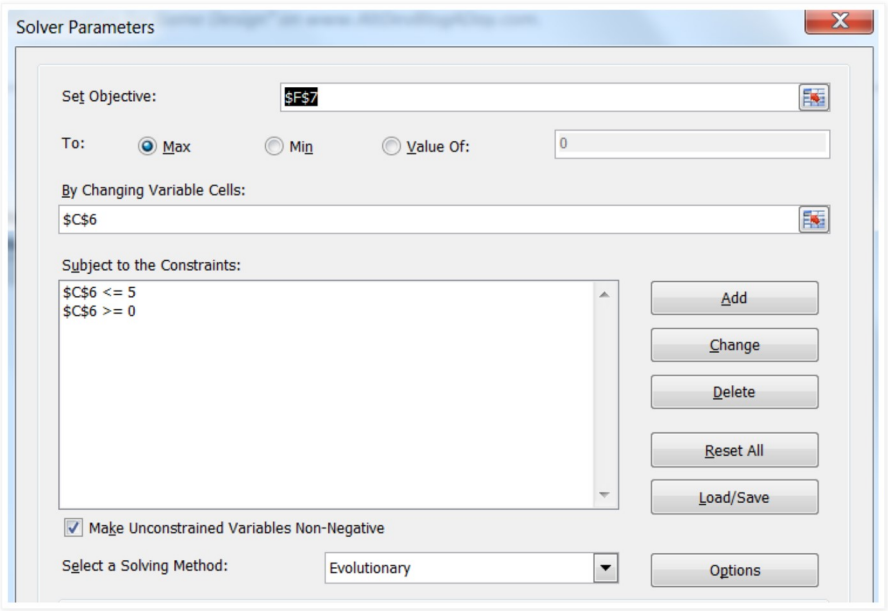
As we'll see, that's exactly why we're using Solver.

First, we'll reset our "Tax Level" cell to zero. Then go to the "Data" tab in Excel and you should see a button called "Solver" on the right side of



If you don't see it, go to Excel Options, select the Add-Ins category, ensure that the "Manage" drop box is set to "Excel Add-Ins," hit "Go..." and ensure that "Solver Add-in" is checked.

When you hit the Solver button, you should see a dialog like the one shown below.



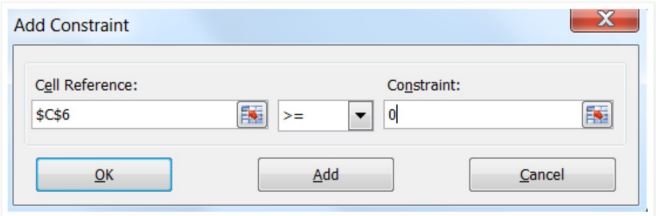
Let's go through the steps to set up the Solver dialog now.

In the "Set Objective" field, we set up the cell that represents what we're trying to optimize. In this case, we're trying to get as much tax revenue as possible, so we select the orange objective cell that represents our tax revenue, and then select "To: Max" in the radio button list just below it.

In "By Changing Variable Cells," we select the cell(s) we want Solver to figure out for us. We want it to determine the optimal tax rate for us, so we select the yellow decision cell ("Tax Level (0-5)"). If all goes as planned, it should end up setting this cell to '3' for a 30% tax rate, which we already determined was optimal by tweaking it by hand.

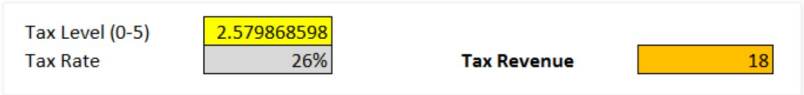
Finally, we need to add a few *constraints*. Constraints essentially act as conditions on any of the cells in our decision model, and Excel Solver will ensure that it focuses solely on solutions that respect the constraints that you specify. These can include limiting specific cells (usually decision cells and calculation cells) to be above some minimum value and/or below some maximum value, and/or forcing Solver to treat them as integers or binary variables (0 or 1). Constraints are enormously useful for helping ensure that you have a valid model and that Solver only searches through

Solver needs at least a few constraints at a minimum to help it figure out what the boundaries are on the decision cells - in other words, the minimum and maximum values for each cell. In order to add a constraint, we hit the "Add" button on the right, which brings up a dialog like the following:



We add two constraints, one to ensure that the "Tax Level" decision cell is constrained to be >=0, and another to ensure that the decision cell is <= 5. Then, we ensure that the "Solving Method" combo box is set to "Evolutionary" and hit "Solve."

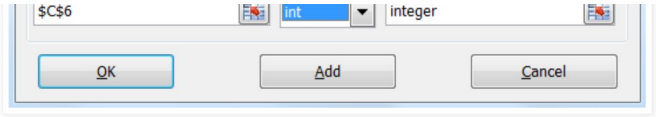
After Solver runs for 30 seconds or so, it will give us an answer like this:



Uh-oh ... This is a problem! Solver got the correct amount of revenue, but the tax level is wrong. The player can only set taxes at 10% increments, so Solver is clearly setting fractional tax rates, which is something that the player can't do.

The solution is to ensure that the tax rate cell is constrained to be an integer. It should be either 0, 1, 2, 3, 4, or 5, with nothing in between.

Fortunately, Solver gives us an easy way to do exactly that. Open the Solver, press the Add button, select the "Tax Level" decision cell, and then select the "int" constraint from the drop-down box in the center:



Now, we re-run the Solver, and this is what we get:

Tax Level (0-5)	3		
Tax Rate	30%	**Tax Revenue**	18

Perfect! Solver got the right answer for us with a very small amount of effort. As we'll see in a moment, as we scale up to larger problems, we'll quickly discover that the amount of work that Solver can do for us dramatically outstrips the amount of effort required to tell it what to do.

A Growing City

Now let's extend it with a slightly more sophisticated model of our city.

In any "4X" strategy game, our cities (or planets, or colonies, or whatever other population units we use) will grow over time. We'll assume the city has a flat 8% growth rate per turn, beginning with 1500 thousand (1.5 million) citizens, and growing up to a size of 12 million citizens. Our spreadsheet now looks like this:

Tax Level (0-5)	5		
Tax Rate	50%	**Cumulative Tax Revenue**	
Growth Rate	1.08		65
Starting Population	1500		

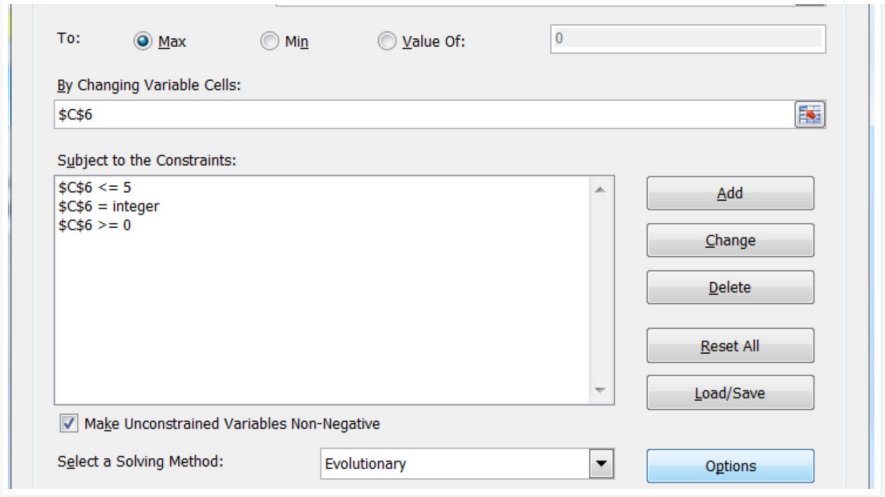
Population (thousands)	Population (millions)	Base Unhappiness	Tax Rate	Unhappy Citizens Due To Taxes	Unhappy Citizens	Happy Citizens	Taxes (From Happy Citizens)
1500	1	0	50%	0	0	1	5
1620	1	0	50%	0	0	1	5
1749	1	0	50%	0	0	1	5
1888	1	0	50%	0	0	1	5
2039	2	0	50%	1	1	1	5
2202	2	0	50%	1	1	1	5
2378	2	0	50%	1	1	1	5
2568	2	0	50%	1	1	1	5
2773	2	0	50%	1	1	1	5
2994	2	0	50%	1	1	1	5
3233	3	0	50%	2	2	1	5
3491	3	0	50%	2	2	1	5
3770	3	0	50%	2	2	1	5

Each additional subsequent row of the table represents one turn of the game.

We've also changed our calculations for the base Unhappiness level. It's now calculated as one-half of the base population level (in millions), rounded down. This keeps base unhappiness at 0 until the city gets to size 4, at which point it grows linearly with the city size.

Just as before, we can experiment with the tax levels manually by manually changing "Tax Level." We get 0, 102, 190, 222, 144, and 65 currency units of tax revenue, respectively, from each of the tax levels between 0% and 50%.

And just as before, we can run Solver on it; it will quickly find that the optimal tax rate is the same 30% level, giving us 222 currency units of revenue. Here's what the Solver dialog looks like:



Variable Tax Rates

But of course, the player doesn't play this way. Our simulated "city" sets a single tax rate and keeps it exactly the same every turn of the game. But the actual player can change the tax rate as often as he likes, and he'll often need to tune it as his city grows and circumstances change.

Wouldn't it be great if we could do more than figuring out a single, flax optimal tax rate, and determine the optimal tax rate every turn?

That would instantly tell us how the best possible player would adjust tax rates.

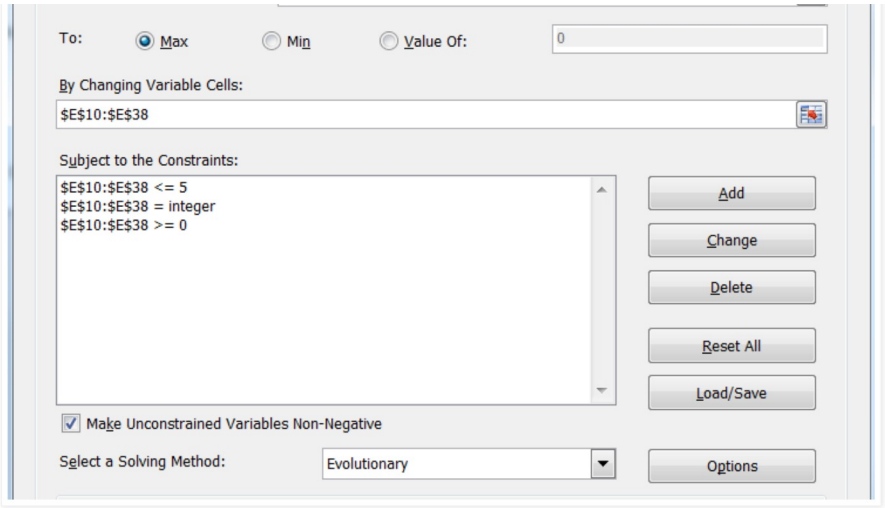
It turns out, we can! And now that we've set up the decision model in the right way, we can do it incredibly easily.

The major change is that we have to remove our decision cell - "Tax Level (0-5)" - and replace it with an entire column of tax level cells, as shown below.

Growth Rate	1.08		Cumulative Tax Revenue
Starting Population	1500		232

Population (thousands)	Population (millions)	Base Unhappiness	Tax Level	Tax Rate	Unhappy Citizens Due To Taxes	Unhappy Citizens	Happy Citizens	Taxes (From Happy Citizens)
1500	1	0	5	50%	0	0	1	5
1620	1	0	5	50%	0	0	1	5
1749	1	0	5	50%	0	0	1	5
1888	1	0	5	50%	0	0	1	5
2039	2	0	3	30%	0	0	2	6
2202	2	0	3	30%	0	0	2	6
2378	2	0	3	30%	0	0	2	6
2568	2	0	3	30%	0	0	2	6
2773	2	0	3	30%	0	0	2	6
2994	2	0	3	30%	0	0	2	6
3233	3	0	3	30%	0	0	3	9

Now, instead of telling Solver to optimize a single cell, we'll tell it to optimize the entire "Tax Level" column. Here's what the Solver dialog looks like - you'll note that it's exactly the same as before, except that the variable cells and constraints have changed to represent the entire range of cells in the "Tax Level" column instead of the single cell.



Sure enough, Solver proves that tweaking the tax rate does make a difference, giving us cumulative tax revenue of 232 currency units. This is only a 5% improvement over the 222 units we obtained with a flat tax rate, but it's still substantial enough that we know some gamers will achieve it.

Looking more closely at the solution Solver obtained, we see that it begins with a tax rate of 50%, since the size-1 city doesn't have enough population to generate Unhappiness from it at this point. As the city grows, it tweaks the tax rate between 20% and 30% each turn, depending on which one brings in more revenue.

You can download the spreadsheet for this example here; it splits the three stages of this example into their own worksheets on the spreadsheet (flat tax with a non-growing city, flat tax with a growing city, and adjustable tax rate with a growing city).

Conclusions

The solution we found above reveals something interesting: the discrete nature of our game sim, which represents arbitrary groupings of millions of people as discrete "citizens" who can be in only one of two discrete states of happiness, introduces idiosyncrasies in the model. Although the actual game itself needs to do this discretization at some level for the sake of accessibility and playability, clever players and "power gamers" will be able to exploit this artificial granularity to generate advantages over players who don't want to futz around with tax levels every turn.

This gives rise to an interesting question: Is this what we want? Does this mechanic make players feel that they need to micro-manage their tax levels every turn to play the game? And do we want to enable "power gamers" to be able to game the system in that way, and is the 5% reward they receive appropriate for doing so?

These are questions I can't answer. After all, *you're* the designer in this case, which means that you set the design goals, so it's up to you to determine whether this level of exploitation is in line with your goals for the game.

Of course, this model is only a bare-bones framework. In a real 4X strategy game, you would be able to make all sorts of decisions about how to grow your city and how to create buildings and make other modifications that would affect city growth, happiness, tax revenues, and productivity further on down the line.

In a future article in this series, we'll build a similar but much more complex model of an entire planetary colony in a game similar to *Master of Orion 2*. This example will be far more sophisticated, since we will be able to make decisions every turn that will subsequently affect all of those parameters, such as growth and productivity, so that each decision has "knock-on" effects that affect subsequent decisions. However, we'll also see that Solver's evolutionary optimizer is up to the challenge.

Stay tuned for the next exciting episode, where we'll fulfill our promise of optimizing the SuperTank weapons loadout in the example we posed in the introductory article.

-Paul Tozour

Part 3 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.

The author would like to thank Jurie Horneman and Robert Zubek for their assistance in editing this article.

Posted by Paul Tozour at 8:17 AM

Labels: Part 2

8 comments:

Fhyl July 16, 2013 at 11:14 AM

Absolutely great stuff... However, the 2007 Excel solver gives some funky results depending on the starting value of the cell, and there seems to be nowhere to impose the Evolutionary method. I assume you use Excel 2010 in your example?

Some research lead me to a message board discussing my very problem : considering I might make systems in which many variables have to be taken into account, is there a way we can force the Solver to run through as many scenarios as possible before assuming something ridiculous like "0 is the optimal tax rate"?

Reply

Yes, I am using Excel 2010 for everything, so I can't speak to any issues with Excel 2007.

I am not aware of any method to do what you describe in your second paragraph. I recommend just using Excel 2010; many of the examples in the later articles require the Evolutionary solver to be used and I can't guarantee you'll get anything like the results you're looking for with a different version.

Fhyl July 16, 2013 at 11:23 AM

Thanks, I'll look into this, and rest assured that I'm eager to read following posts in this series!

Paul Tozour July 16, 2013 at 11:25 AM

Terrific! If all goes as planned I should be posting a new one every Sunday.

Reply

FreakyZoid July 16, 2013 at 12:27 PM

Fantastically useful article. I knew Excel must have the tools to solve this kind of game balancing problem, and your examples make it very easy to follow. Really looking forward to future articles in this series, and I've added your blog to my feedly :)

Reply

Anonymous July 29, 2013 at 8:16 AM

Thanks for this great tutorial.

For the first solver calculation I get a different result each time I run it. However, all are near 30% and end up being the same end-result eventually :)

Reply

Replies

Paul Tozour July 29, 2013 at 8:22 AM

Yes, the evolutionary solver is not deterministic, so it can give slightly different results if those different results do not have any effect on the value returned by the objective function. So since anything close to 30% will work in that case (before it's been constrained to an integer value), it will return any value close to 30%.

Reply

Voosco December 29, 2013 at 5:36 PM

This comment has been removed by the author.

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)