The Intelligence Engine

Sunday, September 22, 2013

Decision Modeling and Optimization in Game Design, Part 8: Graph Search

This article is the eighth 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)*

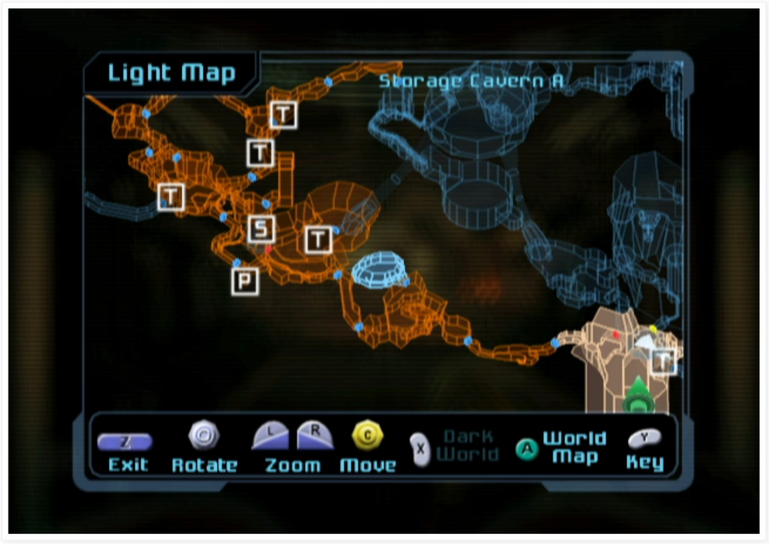
The spreadsheet for this article can be downloaded here: link

Metroid Prime World Design

If you've ever played any of the games in the *Metroid Prime* trilogy (or for that matter, any of the *Metroid* games), then you're familiar with the way a typical *Metroid* world is set up. Each map is a complex, interlocking set of rooms, with many parts of the map sealed off by requiring access to different suits, weapons, and power-ups. As the player gains access to new technologies such as "morph ball mode," "spider ball mode," and different suits and weapons, different parts of the map structure gradually unlock, forcing a large amount of re-traversal for the player to visit each new unlocked area.

As a result of this complexity, handling all of the traversal and re-traversal opportunities available to the player at each point in time is one of the major game design challenges on a *Metroid* style of game. If we move any rooms, change the connections between rooms, or change the location where the player gets a certain suit or weapon or power-up, that has the potential to completely change the player's flow and quite possibly break the game.

The in-game map looks something like this:



If you're a programmer, you'd probably look at a map structure like this and immediately realize the potential for all sorts of design tools you could use to build and manage the map structure. Most likely, you'd make a system where designers could:

- chart out a map's structure graphically,
- get instantaneous feedback on where the player could move at each stage of the game,
- view the combination of rooms the player could access with any combination of power-ups, and
- instantly determine the fastest route from any room to any other room with any possible combination of power-ups (or indicate that no such path exists).

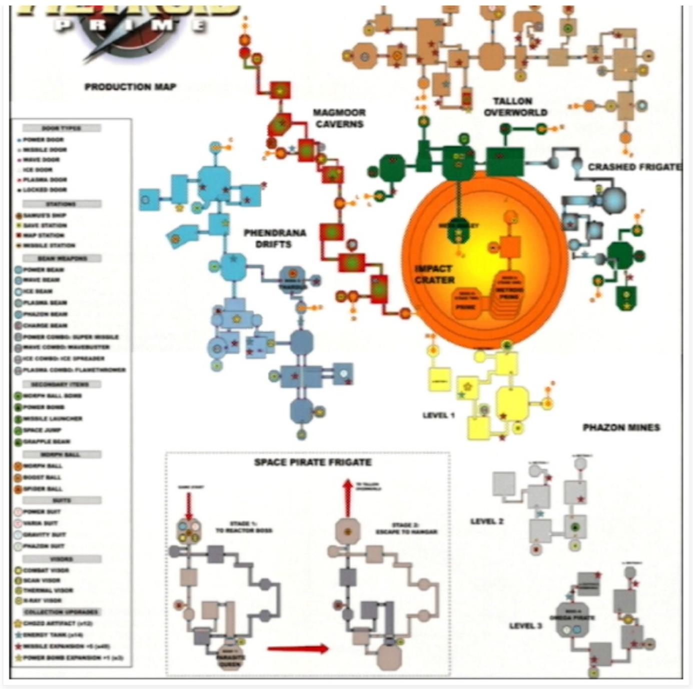
Under the hood, you'd probably use an A* search or Dijkstra's search along the graph, taking the player's movement capabilities into account while crossing each graph edge.

A tool like that would be relatively easy to create; it would probably only take a talented programmer a few days to write, and it would likely save designers several weeks over the life of the project.

Surprisingly, though, the creators of the *Metroid Prime* series never had a tool like that. When I worked at Retro Studios / Nintendo between 2003 and 2008, we did it the old-fashioned way, and our then-Assistant Producer meticulously maintained large printed maps of all the worlds of each *Metroid Prime* game, hanging all of them around the company meeting room. Each time one of the worlds changed, he would edit the associated map graph and re-print all the affected world maps. These maps looked a lot like this:

Contributors

- Paul Tozour
- Sensai Pose
- Unknown



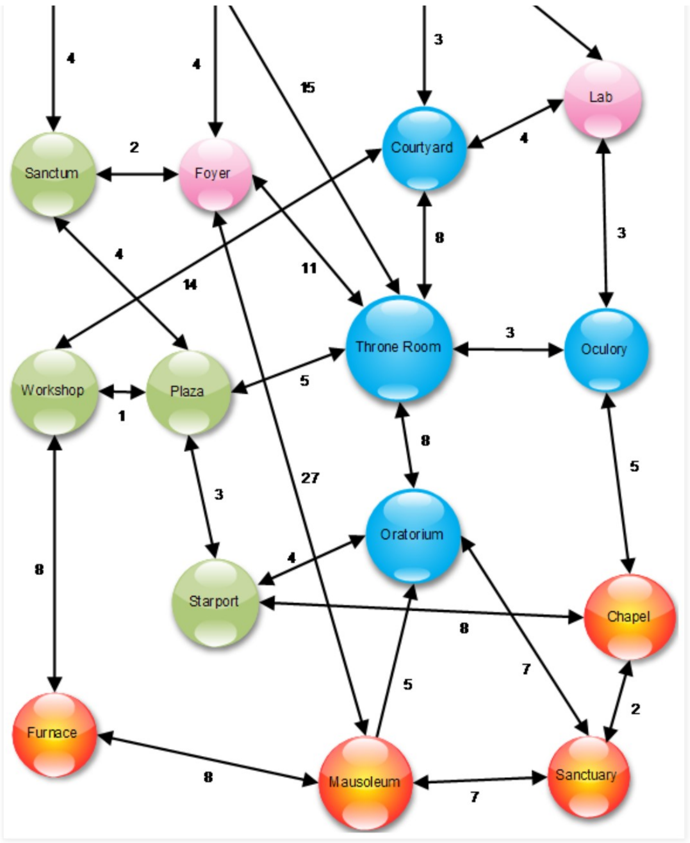
As much as we'd like to actually create a "map design tool" like this, we promised in Part 1 that we wouldn't write any code in this series. So that kind of a full-featured map design tool is outside the scope of this article.

What we'll do instead is show how we can handle a problem like this directly inside Excel. So even if we're designers who can't get any programmer support to build this kind of a tool, we can still create something similar without having to rely on outside help. And once we're done, we'll have a good template in place that programmers can take and run with to build exactly that kind of a map design tool.

The Metroid World Example

To illustrate this problem, we will build an example "Metroid world" and show how to solve it in Excel. In order to prove that you can actually handle a problem like this from right inside Excel regardless of complexity, we're going to make this a very complex problem, with 17 rooms and non-obvious connections between them.

Our example world will look like this:



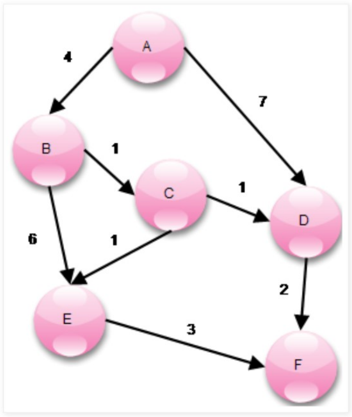
Each bubble shows a different room, and the arrows indicate the connections between pairs of rooms. The number next to each connection indicates the cost of traversing that connection in either direction. We'll assume these costs fully account for the traversal time to get between any pair of rooms, and there are no costs associated with the rooms themselves (i.e. the traversal costs are to some arbitrary point inside each room).

We'll explain the coloring of the different rooms later.

Once we've built the model, we'll be able to ask it questions such as "what's the shortest path for the player to get from the Library to the Sanctuary?"

A Simpler Problem

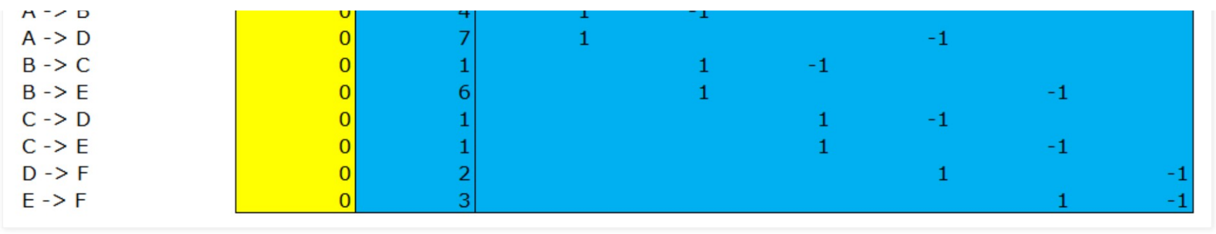
Let's start by working our way up to that with a simpler problem. Assume we have 6 rooms, 'A' through 'F', with connections as shown in the following directed graph.



We want to know: what's the fastest way from A to F? Is it ABEF, ABCEF, ABCDF, ADF ...?

You can quickly figure it out yourself by adding up the weights in all four potential paths, but let's see how we'd encode it in Excel Solver.

First, we make a column with all the possible arcs: A->B, A->D, B->C, etc. For each arc, we list the total distance of that arc (its traversal cost) in the "Total Distance" column. To the right of that, we have columns for each room, and for each arc, we put a '1' in the room where the arc originates and a '-1' in the room where the arc terminates (so, for example, "C->D" has a '1' in the 'C' column and a '-1' in the 'D' column).

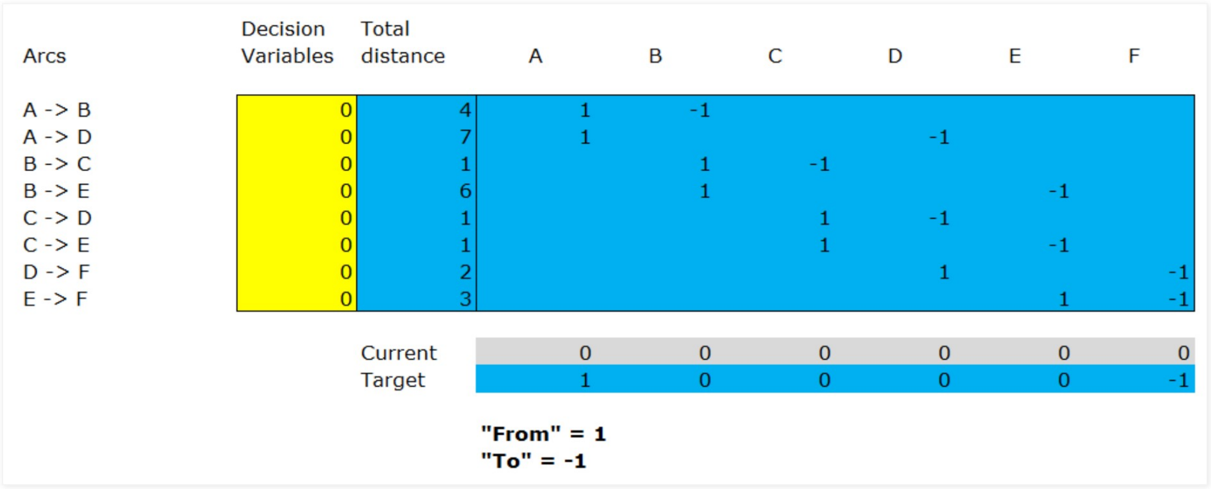


A -> B	0	4	1	-1				
A -> D	0	7	1			-1		
B -> C	0	1		1	-1			
B -> E	0	6		1			-1	
C -> D	0	1			1	-1		
C -> E	0	1			1		-1	
D -> F	0	2				1		-1
E -> F	0	3					1	-1

We also have a "Decision Variables" column in yellow -- this indicates which arcs the decision model will actually use. Right now, we'll set all of those to 0 to indicate that they're not currently used (i.e. the problem hasn't yet been solved).

Because we're following the formatting conventions listed in Part 2, the decision variables are yellow and the predefined values stated as part of the problem description are in blue.

Now, we add two rows beneath the right part of the matrix to sum up the current target values: this is the "Current" row shown below.



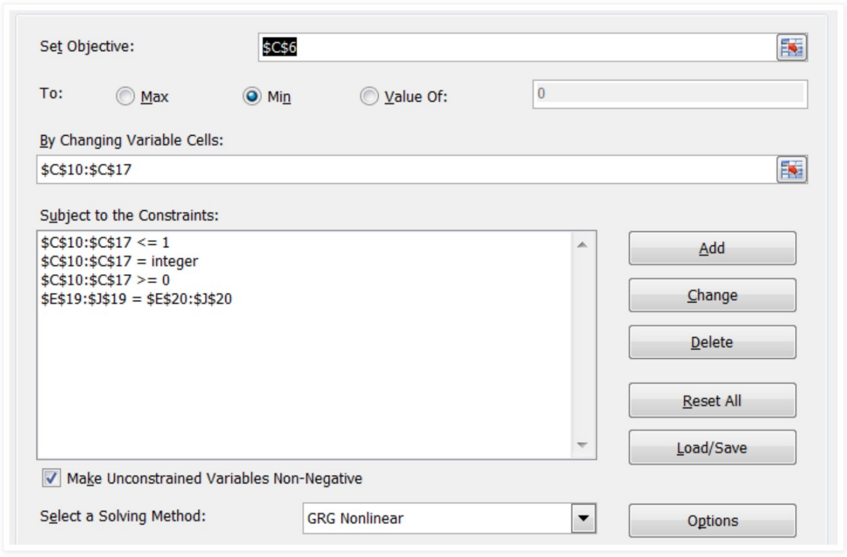
Arcs	Decision Variables	Total distance	A	B	C	D	E	F
A -> B	0	4	1	-1				
A -> D	0	7	1			-1		
B -> C	0	1		1	-1			
B -> E	0	6		1			-1	
C -> D	0	1			1	-1		
C -> E	0	1			1		-1	
D -> F	0	2				1		-1
E -> F	0	3					1	-1
		Current	0	0	0	0	0	0
		Target	1	0	0	0	0	-1

"From" = 1
"To" = -1

The "Target" row indicates what we're attempting to achieve. We simply put in a '1' for the origin, a '-1' for the destination, and '0' for all the other values. We will use these when we set up the constraints later in the exercise in order to make sure that any potential solution begins and ends where we want it to.

Finally, we set up the objective function. This is simply the decision variables multiplied by the respective values in the "Total Distance" column (implemented with the Excel SUMPRODUCT() formula). In other words, if any arc is used, its decision variable will be 1, and we will multiply it by the respective length of that arc in the "Total Distance" column; otherwise, its decision variable will be 0, and we will get a 0. This will ensure that the objective function accurately measures the total distance of all the arcs that are actually used in whatever solution we find.

Now, we solve (see Part 2 if you need a refresher on how to find and use Excel Solver).



Set Objective: $C$6

To: Max [✓ Min] Value Of: 0

By Changing Variable Cells:
$C$10:$C$17

Subject to the Constraints:
$C$10:$C$17 <= 1
$C$10:$C$17 = integer
$C$10:$C$17 >= 0
$E$19:$J$19 = $E$20:$J$20

[✓] Make Unconstrained Variables Non-Negative

Select a Solving Method: GRG Nonlinear

In our objective ("Set Objective"), we specify the orange objective cell. Since the objective represents the total distance, and we want to minimize the total distance, we select the "Min" radio button below that. For our decision cells ("By Changing Variable Cells"), we specify the yellow "Decision Variables" column. In the constraints section, we specify that the yellow decision cells have to be integers that equal 0 or 1. We also specify that the "Current" row has to equal the "Target" row, to ensure that the path begins and ends at the correct rooms.

Before we solve, note that "Select a Solving Method" at the bottom has "GRG Nonlinear" specified instead of the usual "Evolutionary" solver. We'll of the other possible paths, which are all greater than 8.

You can find this solution in the "Sample solution" worksheet of the attached spreadsheet.

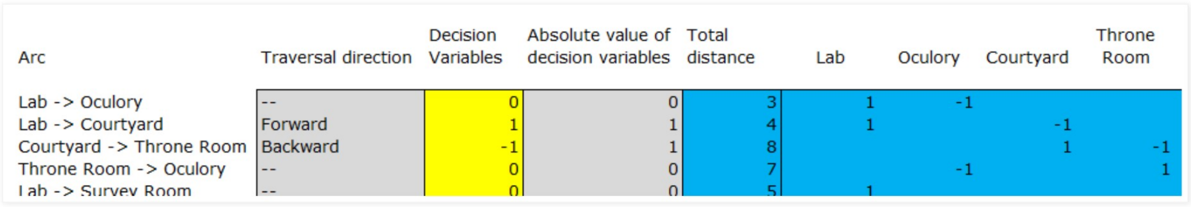
Searching in the Metroid World

With that framework in hand, let's generalize it to the large Metroid sample world above.

One big change in this version is that this example includes bidirectional links instead of directed arcs between rooms. This means our decision variables will no longer simply be '0' or '1' to indicate whether an arc forms part of the shortest path; instead, we'll also use '-1' to indicate that the path is traversed backwards.

[Note that this approach only works because all of the arcs in the Metroid example are bidirectional. If we had any directional arcs in our example, we would have to use the previous approach, and specify any bidirectional arcs as two separate rows in the spreadsheet (for example, we'd have to specify "Lab -> Oculory" and "Oculory -> Lab" on two separate rows to indicate that the player can move between the two rooms in either direction).]

Our spreadsheet looks similar to the previous one, except that we now add two columns, "Traversal direction" and "Absolute value of decision variables." "Traversal direction" is just a visual assistant to indicate which direction we'll traverse a link, with a decision variable value of '1' translated into "Forward" and a value of '-1' translated to "Backward."



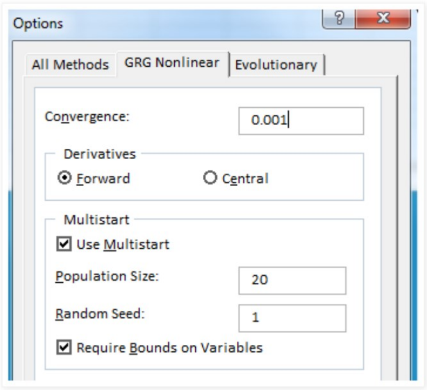
Arc	Traversal direction	Decision Variables	Absolute value of decision variables	Total distance	Lab	Oculory	Courtyard	Throne Room
Lab -> Oculory	--	0	0	3	1	-1		
Lab -> Courtyard	Forward	1	1	4	1		-1	
Courtyard -> Throne Room	Backward	-1	1	8			1	-1
Throne Room -> Oculory	--	0	0	7		-1		1
Lab -> Survey Room	--	0	0	5	1			

We also change the objective function so that it multiplies the total distance column by the "Absolute value of decision variables" column rather than the decision variables themselves (since traversing a link backwards gives us a '-1' in the "Decision Variables" column, and multiplying this by the distance value would give us a negative distance).

We also need to change our constraints so that it accepts integers between -1 and 1 in the "Decision Variables" column instead of 0 to 1.

In the "Target" row below the room source/destination matrix, we enter a '1' for the Library and a '-1' for the Sanctuary to ask Solver to figure out the shortest path from the Library to the Sanctuary.

After these changes, we solve again. Note that because this is a much more challenging problem, we now have to tweak the Solver options a bit. In the Solver dialog, we hit Options, select the GRG Nonlinear tab, and then check "Use Multistart" and set "Population Size" to 20 (see this link for more info). This will make Solver do a much more exhaustive search and will make it much more likely to find the global optimum, and the expense of taking a bit more time to solve the problem. With this option set, Solver may take 3-5 minutes to find the correct solution.



Convergence: 0.001

Derivatives: Forward [✓] Central

Multistart

☑ Use Multistart

Population Size: 20

Random Seed: 1

☑ Require Bounds on Variables

After Solver finds a solution, we scan the "Traversal direction" column to make sense of the solution that Solver came up with. We get the following as the shortest path:

Library -> Sanctum -> Plaza -> Starport -> Chapel -> Sanctuary

... for a total cost of 21.

You can find this setup in the "Metroid world solution" worksheet of the attached spreadsheet.

Adding Movement Constraints

Of course, this still doesn't quite give us what we want. The actual *Metroid* worlds have complex movement requirements that limit the player's access at any point depending on the exact combination of equipment he has acquired. What we really want is a way to limit the search to *only* those rooms that are valid with a given combination of equipment.

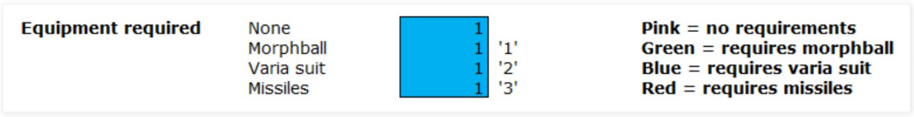
There are a lot of different ways to do this, but for the sake of simplicity, we'll just assume each room in the Metroid world graph above requires one particular type of equipment. Then we'll solve that problem and leave any more sophisticated implementations as an exercise for the reader.

We will assume the following colors in the graph have the following meanings:

- **Pink** rooms (Library, Ruins, etc) don't require any special equipment.
- **Green** rooms (Sanctum, Plaza, etc) require the player to have the **morphball** technology

The goal is to use Solver to help us figure out how the optimal path for the player changes depending on whether or not he has the morphball, Varia suit, and/or missiles.

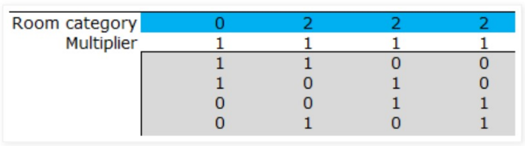
We'll start by adding an "equipment required" table near the top of the spreadsheet:



Equipment required				
	None	1		**Pink = no requirements**
	Morphball	1	'1'	**Green = requires morphball**
	Varia suit	1	'2'	**Blue = requires varia suit**
	Missiles	1	'3'	**Red = requires missiles**

This table uses a '1' for each cell to indicate that the player currently has access to that technology, or a '0' if he doesn't. So far, we've been assuming that the player can travel anywhere, so we've filled in a '1' for each technology.

Now, we add two rows directly below the arc matrix, "Room category" and "Multiplier."



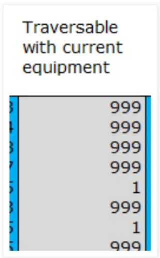
Room category	0	2	2	2
Multiplier	1	1	1	1
	1	1	0	0
	1	0	1	0
	0	0	1	1
	0	1	0	1

Room category specifies the equipment type of the room listed in the same column -- None ('0'), Morphball ('1'), Varia suit ('2'), or Missiles ('3').

The **Multiplier** row contains a formula that checks the room category in each column against the "Equipment required" table above. It produces a '1' if the player possesses the appropriate equipment, and a '0' otherwise.

The grey cells below that are a table that combines the arc reachability table above with the "Multiplier" row. It produces a '1' if the room in the same column is included in the path for the corresponding row, *and* the multiplier in the same column is 1 (in other words, it multiplies the absolute value from the arc reachability table by the multiplier in the same row).

We also add an additional column to the table just to the right of the "Total distance" column: "Traversable with current equipment." This column will produce a '1' if *both* rooms in the arc are reachable with the player's current equipment. If not, it will produce a value of '999' in order to make the arc prohibitively expensive.



Traversable with current equipment
999
999
999
999
1
999
1
999

Finally, we change our objective cell (the orange cell called "minimum total distance") to also include this column in its SUMPRODUCT() formula.

Now, we run Solver exactly as before. Make sure you have the Options set as specified above (Use Multistart = TRUE with a Population Size of at least 20), or you may not get the desired results.

First, we'll set the value for "Morphball" in the "Equipment required" table at the top of the spreadsheet to 0 and re-solve (note that this may take 3-5 minutes to solve). This is the optimal path with no morphball:

> **Library -> Ruins -> Survey Room -> Courtyard -> Lab -> Oculory -> Chapel -> Sanctuary**

... for a total cost of 25.

Next, we set "Morphall" back to 1 and set the cell below it to 0 to find out the optimal path when the player doesn't have the Varia suit:

> **Library -> Sanctum -> Plaza -> Starport -> Chapel -> Sanctuary**

You'll notice that this is exactly the same as the solution we found in the previous worksheet, before we began taking the player's equipment into account.

What happens when we set both of these to 0, so the player doesn't have the morphball or the Varia suit? Re-solving, we get:

> **Library -> Ruins -> Foyer -> Mausoleum -> Sanctuary**

for a total cost of 41. As you can see from the graph above, going down the very expensive Foyer -> Mausoleum arc (with a cost of 27) is the only option available when both the morphball (green) and Varia suit (blue) are disabled ... and this is, in fact, the shortest path that includes that link.

Problem solved!

You can find this spreadsheet in the "With equipment requirements" worksheet of the attached spreadsheet.

Conclusion

In this example, we've shown how you can use decision modeling and optimization techniques as a powerful tool to solve graph search and reachability problems. This particular *Metroid* game world example is of course only one application of this technique; there are many more. You could just as easily use it to find the shortest path through a star system in a space game with direct links between stars, for example, or the shortest path along a flight path network in an MMO such as *World of WarCraft*.

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 10:23 AM

Labels: Part 8

No comments:

Post a Comment

Enter your comment...

Comment as: Ahmad Al-Kashef ((Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)