9

An Introduction to Utility Theory

David "Rez" Graham

9.1 Introduction
9.2 Utility
9.3 Principle of Maximum Expected Utility
9.4 Decision Factors
9.5 Calculating Utility
9.6 Picking an Action
9.7 Inertia
9.8 Demo
9.9 Conclusion

9.1 Introduction

Decision making forms the core of any AI system. There are many different approaches to decision making, several of which are discussed in other chapters in this book. One of the most robust and powerful systems we've encountered is a utility-based system. The general concept of a utility-based system is that every possible action is scored at once and one of the top scoring actions is chosen. By itself, this is a very simple and straightforward approach. In this article, we'll talk about common techniques, best practices, pitfalls to avoid, and how you can best apply utility theory to your AI.

9.2 Utility

Utility theory is a concept that's been around long before games or even computers. It has been used in game theory, economics, and numerous other fields. The core idea behind utility theory is that every possible action or state within a given model can be described with a single, uniform value. This value, usually referred to as *utility*, describes the usefulness of that action within the given context. For example, let's say you need a new toy for your cat; so you go online and find the perfect one. One website has it for $4.99 while another website sells the exact same toy for $2.99. Assuming delivery times are the same, you will likely choose the toy for $2.99. That option typically has a higher utility than the toy for $4.99 because, in the end, you are left with more money.

This process gets more difficult when you need to compare the value of two things that aren't directly comparable. For instance, in the previous example let's assume that the two

websites have different delivery times. The toy for $4.99 will arrive at your house in two days while the toy for $2.99 will arrive in five days. In this case, the choice is no longer a simple matter of comparing the two price values. Some conversion between time and money has to be made in order to measure the overall worth of an action. We could say that each day is worth $1, which means that the total cost of the $4.99 toy is $6.99 while the total cost of the $2.99 toy is $7.99. The $4.99 toy is the winner in this case (because it costs you less in combined money + time). You can also weigh things such as website loyalty, recommendations from friends, history, customer reviews, and anything else that you might consider a relevant factor. All of these factors have utility scores of their own, which you can then combine to create the total *expected utility* of for the decision.

It's important to note that *utility* is not the same as *value*. Value is a measurable quantity (such as the prices above). Utility measures how much we desire something. This can change based on personality or the context of the situation. If you were a billionaire, you would likely choose the cat toy for $4.99 because you might value time more than money. The $2.00 you save is a negligible amount. On the other hand, if you were very poor, you would likely choose the cheaper toy and wait the extra time because that extra $2 is really important to you. That money has the exactly the same *value*, but the *utility* of the money is variable, based on the context in which it is being considered. This can change from moment to moment. Right now, the utility of having a bandage with you might be pretty small. If you were to accidentally cut yourself, the utility of having a bandage would climb.

9.2.1 Consistent Utility Scores

When calculating utility scores, it's important to be consistent. Because utility scores are compared to each other to come up with a final decision, they must all be on the same scale across the entire system. As you'll see later in this article, scores are often combined in meaningful ways to produce other scores. Therefore, using *normalized scores* (values that go from 0–1) provide a reasonable starting point. Normalized scores combine very easily through averaging, can be easily calculated given any value within a set range of numbers, and are easily comparable since they are on the same scale. It's important to note that any value range will work, as long as there is consistency across the different variables. If an AI agent scores an action with a value of 15, you should know immediately what that means in the context of the whole system. For instance, does that 15 mean 15 out of 25 or 15%?

9.3 Principle of Maximum Expected Utility

The key to decision making using utility-based AI is to calculate a utility score (sometimes called a weight) for every action the AI agent can take and then choose the action with the highest score. Of course, most game worlds are nondeterministic so calculating the exact utility is not usually possible. It's hard to know if an action will be preferable if you can't determine the results of performing that action. This is the heart of utility theory and where it is most useful. For example, if we had the processing power to compute the entire game tree for a chess game, scoring of moves wouldn't be necessary—we would simply determine if sequences of moves resulted in a win, loss, or tie. We currently don't have that ability, so we score each move based on how strong we think the move is. Provided a reasonable scoring system, utility-based AI is very good at making a "best guess" based on incomplete information.

The most common technique is to multiply the utility score by the probability of each possible outcome and sum up these weighted scores. This will give you the *expected utility* of the action. This can be expressed mathematically with Equation 9.1.

$$EU = \sum_{i=1}^{n} D_i P_i \tag{9.1}$$

In this case, D is the desire for that outcome (i.e., the utility), and P is the probability that the outcome will occur. This probability is normalized so that the sum of all the probabilities is 1. This is applied to every possible action that can be chosen, and the action with the highest expected utility is chosen. This is called the principle of *maximum expected utility* [Russell et al. 09].

For example, an enemy AI in an RPG attacking the player has two possible outcomes—either the AI hits the player or it misses. If the AI has an 85% chance to hit the player, and successfully hitting the player has a calculated utility score of 0.6, the adjusted utility would be 0.85 × 0.6 = 0.51. (Note that, in this case, missing the player has a utility of zero, so there's no need to factor it in.) Taking this further, if this attack were to be compared to attacking with a different weapon, for example, with a 60% chance of hitting but a utility score of 0.9 if successful, the adjusted utility would be 0.60 × 0.9 = 0.54. Despite having a lesser chance of hitting, the second option provides a greater overall *expected utility*.

9.4 Decision Factors

It is rare that any given decision will only rely on a single piece of data. A decision factor can be thought of as a single point of consideration for a decision. For example, when deciding which website to purchase the cat toy from, we don't usually just consider price, but also consider brand loyalty, shipping times, customer reviews, etc. Each of these data points are factors that we weigh into the calculation of the final utility score for whether or not to buy from that website. Factors can also be further modified by weights that determine how much the AI cares about that particular factor, which emulates personality.

One way to achieve this result is to apply the expected utility calculation in Equation 9.1 to each decision factor to come up with the utility of that factor. Assuming those scores are all normalized, you can average them together to come up with a final utility score for that decision. This lets us define a decision as nothing more than some combination of decision factors.

This principle is best illustrated with an example. Let's say we have an ant simulation game where the AI must determine whether to expand the colony or whether to breed. There are three different factors we want to consider for these decisions. The first is the overall crowdedness of the colony. If there are too many ants, we need to expand to make room for more. The second is the health of the colony, which we'll say is based on how full the food stores are. Ant eggs need to be kept at a specific temperature; so there are specially built nurseries that house the eggs where they are taken care of. The amount of room in these nurseries is the third decision factor. These decision factors are based on game statistics that determine the score for each factor. The population and max population determine how many ants are in the colony and how many can exist based on the current

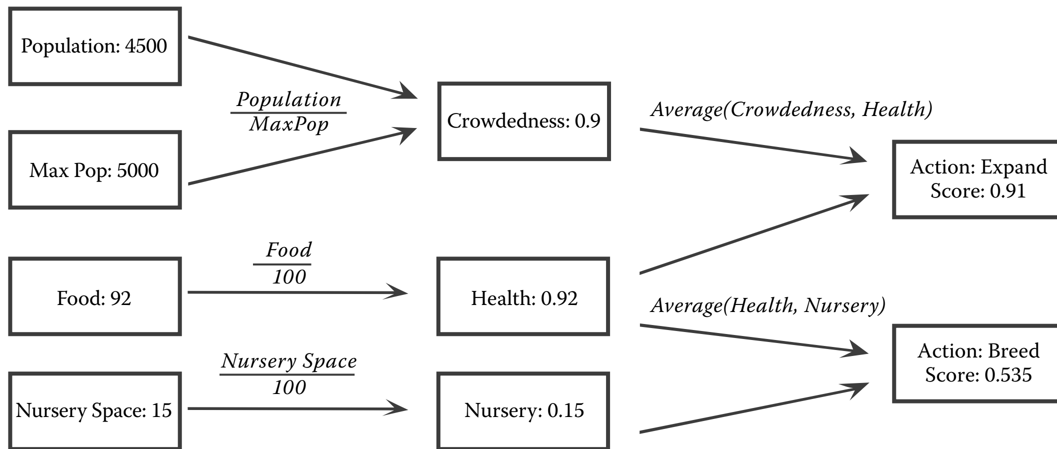

Figure 9.1

An example of combining utility scores from different decision factors to arrive at a final score.

colony size. The food stat represents how full the food stores are and is measured as a number from 0 to 100. The nursery space stat is also measured from 0 to 100 and represents how much space there is in the nursery. You can think of the last stats as percentages.

The scores for the decision factors are then combined to form the final score for the two actions. In this case, crowdedness and health are averaged together to form the score for the expand action while nursery space and health are averaged together to get the score for the breed action. Figure 9.1 shows this combination.

By averaging the normalized scores together, we can build an endless chain of combinations. This is a really powerful concept. Each decision factor is effectively isolated from every other decision factor. The only thing we know or care about is that the output will be a normalized score. We can easily add more game stat inputs, like the distance to an enemy ant colony. This could feed into a decision factor for deciding what kind of ants to breed. You can easily move decision factors around as well, combining them in different ways. If you wanted crowdedness to factor negatively into the decision for breeding, you could subtract crowdedness from 1.0 and average that into the score for breeding.

One of the most powerful uses of this technique is to build a tool that allows you to manipulate decision factors and game state data directly. This tool would allow designers to drag and drop boxes around and connect them with arrows, much like the layout of Figure 9.1. Each arrow would combine the decision factors in different ways. For example, you might average some decision factors together, multiply others, choose the *max* or *min* of another set, etc. Certain decision factors can also be given a weight, making those factors more or less important. There are almost endless possibilities.

9.5 Calculating Utility

So far, we've seen how to calculate the utility given a set of outcomes, and how to combine the utility of multiple decision factors to arrive at the final utility for a decision. The next

step is taking an arbitrary game value and converting it to a utility score. Calculating the initial utility for a decision factor is highly subjective; two different programmers will write two different utility functions that produce different outputs, even given the same inputs. In the ant example above, we chose to represent health as a linear ratio by dividing the current amount of food with the maximum amount of food. This probably isn't a very realistic calculation since the colony shouldn't care about food when the stores are mostly full. Some kind of quadratic curve is more of what we want.

The key to utility theory is to understand the relationship between the input and the output, and being able to describe that resulting curve [Mark 09]. This can be thought of as a conversion process, where you are converting one or more values from the game to utility. Coming up with the proper function is really more art than science and is usually where you'll spend most of your time. There are a huge number of different formulas you could use to generate reasonable utility curves, but a few of them crop up often enough that they warrant some discussion.

9.5.1 Linear

A linear curve forms a straight line with a constant slope. The utility value is simply a multiplier of the input. Equation 9.2 shows the formula for calculating a normalized utility score for a given value and Figure 9.2 shows the resulting curve.

$$U = \frac{x}{m} \tag{9.2}$$

In Equation 9.2, x is the input value and m is the maximum value for that input. This is really just a normalization function, which is all we need for a linear output.

9.5.2 Quadratic

A quadratic function is one that forms a parabolic curve, causing it to start slow and then curve upwards very quickly. The simplest way to achieve this is to add an exponent to Equation 9.2. Equation 9.3 shows an example of this.

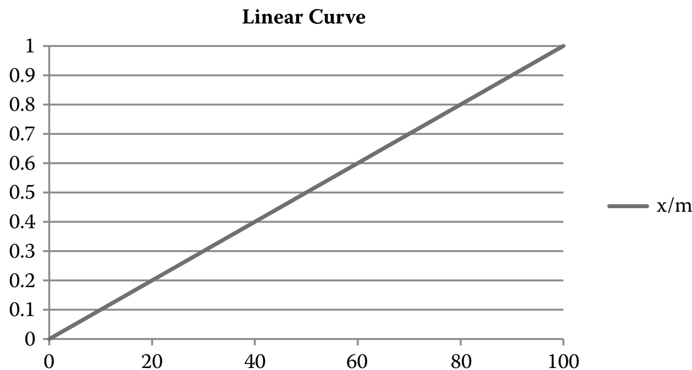

Figure 9.2

A linear curve.

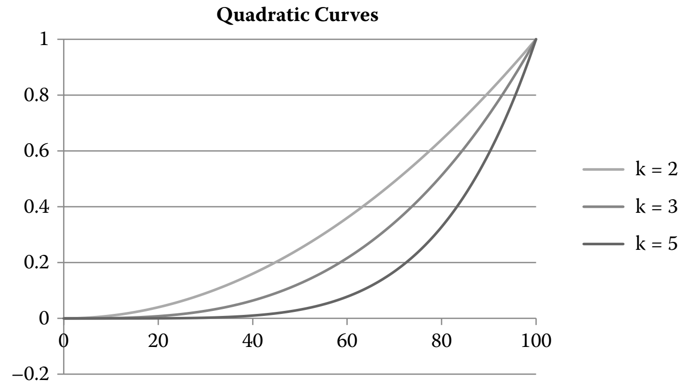

Figure 9.3

Several quadratic curves for various values of k.

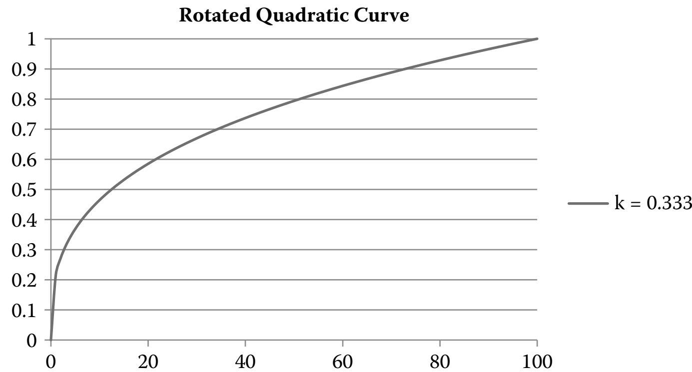

Figure 9.4

A rotated quadratic curve.

$$U = \left(\frac{x}{m} \right)^k \tag{9.3}$$

As the value of k rises, the steepness of the curve will also rise. Since the equation normalizes the output, it will always converge on 0 and 1, so a large value of k will have very little impact for low values of x. Figure 9.3 shows curves for three different values of k.

It's also possible to rotate the curve so that the effect is more urgent for low values of x rather than high values. If you use an exponent between 0 and 1, the curve is effectively rotated, as shown in Figure 9.4.

9.5.3 The Logistic Function

The logistic function is another common formula for creating utility curves. It's one of several sigmoid functions that place the largest rate of change in the center of the input

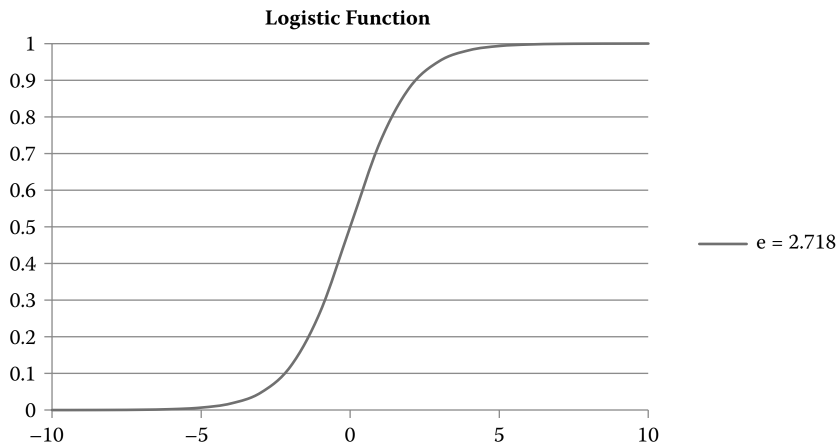

Figure 9.5

The output of a logistic function.

range, trailing off at both ends as they approach 0 and 1. The input range for the logistic function can be just about anything, but it is effectively limited to [–10 … 10]. There really isn't much point generating a curve larger than that and the range is often clamped down even further. For example, when x is 6, EU will be 0.9975.

Equation 9.4 shows the formula for the logistic function and Figure 9.5 shows the resulting curve. Note the use of the constant e. This is Euler's number—the base of the natural logarithm—which is approximately 2.718281828. This value can be adjusted to affect the shape of the curve. As the number goes up, the curve will sharpen and begin to resemble a square wave. As the number goes down, it will soften.

$$U = \frac{1}{1 + e^{-x}} \tag{9.4}$$

9.5.4 Piecewise Linear Curves

The curves we've listed so far are by no means a complete list. There are many, many different ways you can transform the input into something else. Sometimes, having a mathematical formula isn't good enough. Designers often need to fine-tune the specific outputs for various given inputs.

For example, consider a problem faced by all Sims games, which is making a Sim eat when they are hungry. All Sims have a *Hunger* stat which measures how full they are. The lower this *Hunger* stat, the more hungry a Sim is. A naïve scoring implementation might be to model hunger with a rotated quadratic curve like the one in Figure 9.4, or perhaps one with a smaller value for k. That would make Sims get really hungry when their *Hunger* stat got low. The problem is that there would still be a chance they would choose to eat, even when their hunger stat was mostly filled up. The chance would be small, but it would eventually get chosen. Designers want a finer degree of control. They want the ability to have a Sim completely ignore hunger until it reaches a threshold, then get a little hungry, then

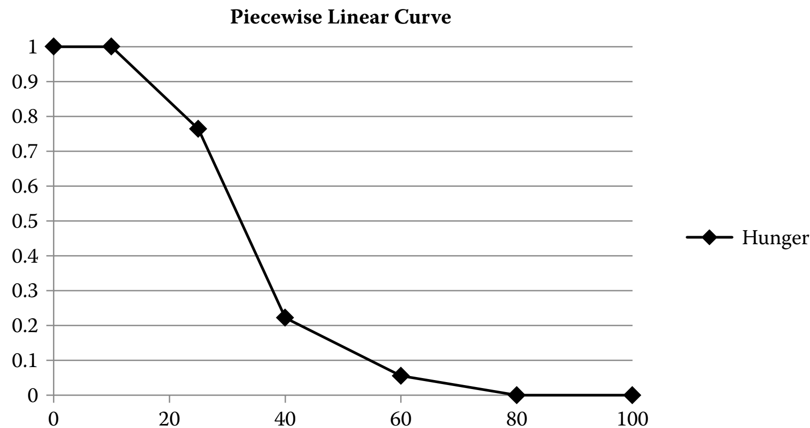

Figure 9.6

A reasonable piecewise linear curve for Hunger.

suddenly get *very* hungry. There's no way to build and tune that specific curve with a simple mathematical formula, so the solution is to create a custom curve. One example of a custom curve is a piecewise linear curve.

A piecewise linear curve is just a custom-built curve. The idea is that you hand-tune a bunch of 2D points that represent the thresholds you want. When the curve is asked for the y value given an x value, it finds the two closest points to that x value and linearly interpolates between them to arrive at the answer. This allows you to create any shaped curve you want and is exactly what *The Sims* uses. Figure 9.6 shows a simple response curve that might be used for hunger.

There are many other types of custom curves. For example, the curve in Figure 9.6 could be changed so that the values from 15 to 60 are calculated with a quadratic curve, while the rest are linear. There's no limit to the number of combinations you can have.

9.6 Picking an Action

Once the utility has been calculated for each action, the next step is to choose one of those actions. There are a number of ways you can do this. The simplest is to just choose the highest scoring option. For some games, this may be exactly what you want. A chess AI should definitely choose the highest scoring move. A strategy game might do the same.

For some games (like *The Sims*), choosing the absolute best action can feel very robotic due to the likelihood that the action will *always* be selected in that situation. Another solution is to use the utility scores as weight, and randomly choose one of the actions based on the weights. This can be accomplished by dividing each score with the sum of all scores to get the percentage chance that the action will be chosen. Then you generate a random number and select the action that number corresponds to. This tends to have the opposite problem, however. Your AI agents will behave reasonably well most of the time, but every now and then, they'll choose something utterly stupid.

You can get the best of both worlds by taking a subset of the highest scoring actions and choosing one of those with a weighted random. This can either be a tuned value, such as choosing from among the top five scoring actions, or it can be percentile based where you take the highest score and also consider things that scored within, say, 10% of it.

This will generally solve the problem at hand, but there could also be times when some set of actions are just completely inappropriate. You may not even want to score them. For example, say you're making an FPS and have a guard AI. You might have some set of actions for him to consider, like getting some coffee, chatting with his fellow guard, checking for lint, etc. If the player shoots at him, he shouldn't even consider any of those actions and only try to score actions that involve combat. In a similar example from *The Sims*, if a Sim is starving to death, she shouldn't bother scoring actions that result in her satisfying her *Fun* motive.

The most straightforward way to solve this is with *bucketing*, also known as *dual utility AI* [Dill 11]. All actions are categorized into buckets and each bucket is given a weight. The higher priority buckets are always processed first. If there are any valid actions in those higher priority buckets, they are always selected before actions in lower priority buckets. In the FPS example above, all combat actions would be in a higher priority bucket than the idle actions. If there are any valid combat actions to take (i.e., they score higher than 0), the guard will always choose one of them and won't consider any of the idle actions. Only when none of the combat actions are valid will the guard choose an idle action.

Buckets can also change priority based on the situation. On *The Sims*, motives are bucketed based on their utility value. The highest scoring motives are grouped into a bucket, and none of the motives below are considered. Once the bucketing is complete, the Sim will score the individual actions on each object, but only the ones that solve for those bucketed motives. Thus, a starving Sim will never even consider watching TV unless they fail to find anything that can solve their hunger. This concept is illustrated in Figure 9.7.

In Figure 9.7, you can see that there are two buckets, one for *Hunger* and one for *Fun*. *Hunger* has scored 0.8 while *Fun* has scored 0.4. The Sim will walk through all possible actions in the *Hunger* bucket and, assuming any of those actions are valid, will choose one. The Sim will not consider anything in the *Fun* bucket, even though some of those actions

Hunger Bucket	0.8
Eat at Table	20
Drink Juice	5
Make Sushi	0

Fun Bucket	0.4
Watch TV	30
Play Video Games	28
Dance	15

Figure 9.7

The Hunger and Fun buckets, each with three actions.

are scoring higher. This is because hunger is more urgent than fun. Of course, if none of the actions in the *Hunger* bucket were valid, the Sim would move on to the next highest scoring bucket. The buckets themselves are scored based on a response curve created by designers. This causes Sims to always attempt to solve the most urgent desire and to choose one of the best actions to solve for that desire.

9.7 Inertia

One issue that's worth bringing up in any AI system is the concept of inertia. If your AI agent is attempting to decide something every frame, it's possible to run into oscillation issues, especially if you have two things that are scored similarly. For example, say you have FPS where the AI realizes it's in a bad spot. The enemy soldier starts scoring both "attack the player" and "run away" at 0.5. If the AI was making a new decision every frame, it is possible that they would start appearing very frantic. The AI might shoot the player a couple times, start to run away, then shoot again, then repeat. Oscillations in behavior such as this look very bad.

One solution is to add a weight to any action that you are already currently engaged in. This will cause the AI to tend to remain committed until something truly better comes along. Another solution is to use *cooldowns*. Once an AI agent makes a decision, they enter a cooldown stage where the weighting for remaining in that action is extremely high. This weight can revert at the end of the cooldown period, or it can gradually drop as well.

Another solution is to stall making another decision—either for a period of time or until such time as the current action is finished. This really depends on the type of game you're making and how your decision/action process works, however. On *The Sims Medieval*, a Sim would only attempt to make a decision when their interaction queue was empty. Once they chose an action, they would commit to performing that action. Once the Sim completed (or failed to complete) their action, they would choose a new action.

9.8 Demo

The demo on the book website (gameaipro.com) demonstrates many of the concepts from this article. It's a simple text-based combat program similar to the menu-based combat RPG's from the '80s, like *Dragon Warrior* (aka *Dragon Quest*) and *Final Fantasy*. You fight a single monster and each of you has the ability to attack the other, heal with a healing potion, or run away. Attacking will do a random amount of damage, healing will use up one of three healing potions to heal a random amount of hit points, and running away has a 50% chance of successfully running away. The relevant code is in AiActor.cpp, which has all of the scoring functions and is responsible for choosing the action when it's the AI actor's turn. The key function is ChooseNextAction(), which takes in the opponent actor and returns an action to perform. This function calls each of the scoring functions to calculate their scores and chooses one using a weighted random.

9.8.1 Decision Factors

When making decisions, the AI considers four basic factors. The first is a desire to attack, which is based on a tuned value that scales linearly as it becomes possible to kill the player

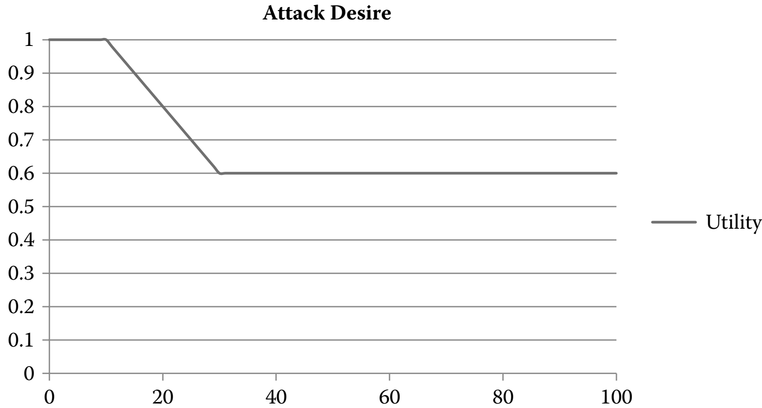

Figure 9.8

The Attack Desire curve.

in a single hit. This causes the actor to get more aggressive during the end-game and take more risks, as shown in Equation 9.5. This is a good example of a range-bound linear curve. The value of *a* in the equation is the tuned aggression of the actor, which is the default score.

$$\mathrm{U} = \max\left(\min\left(\left(1 - \frac{hp - minDmg}{maxDmg - minDmg}\right) \times (1 - a) + a, 1\right), 0\right) \tag{9.5}$$

Figure 9.8 shows the resulting curve from Equation 9.5 where *a* is set to 0.6.

The second decision factor is the threat. This is a curve that measures what percentage of the actor's current hit points will be taken away if the player hits for maximum damage. It has a shape similar to a quadratic curve and is generated with Equation 9.6.

$$U = min\left(\frac{maxDmg}{hp}, 1\right) \tag{9.6}$$

Figure 9.9 shows the resulting curve for Threat.

The third decision factor is the actor's desire for health. This uses a variation of the logistics function in Equation 9.4. As the actor's hit points are reduced, its desire to heal will rise. Equation 9.7 shows the formula for this decision factor.

$$U = 1 - \frac{1}{1 + (e \times 0.68)^{-\left(\frac{hp}{maxHp} \times 12\right) + 6}} \tag{9.7}$$

The resulting curve is a nice, smooth, sigmoid curve, which is shown in Figure 9.10. Note the addition of +6 to the exponent. This is what pushes the curve over to the positive x-axis rather than centering around 0.

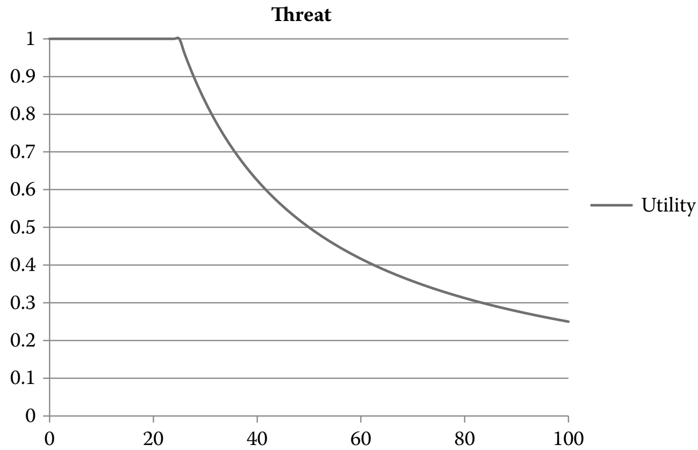

Figure 9.9

The Threat curve.

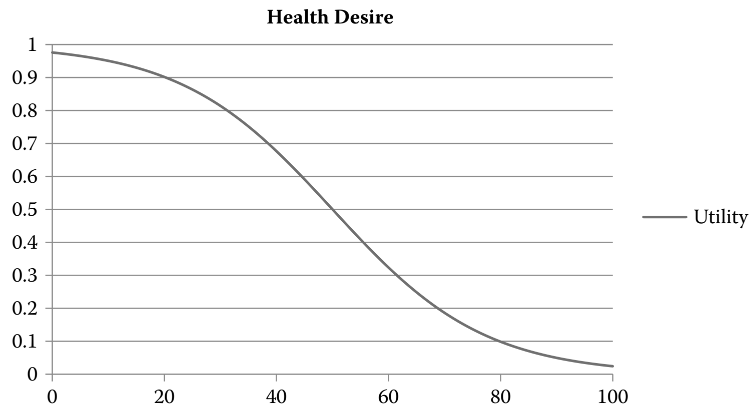

Figure 9.10

The Health Desire curve.

The final decision factor is the desire to run away. This is a quadratic curve with a steepness based on the number of potions the agent has. If the agent has several potions, the likelihood of running away is extremely small. If the agent has none, this desire grows much faster. Equation 9.8 shows the formula for the run desire.

$$U = 1 - \left(\frac{hp}{maxHp}\right)^{\frac{1}{(p+1)^4} \times 0.25} \tag{9.8}$$

The curve itself is dependent on the value of p, which is the number of potions the actor has left. Figure 9.11 shows various curves for various values of p.

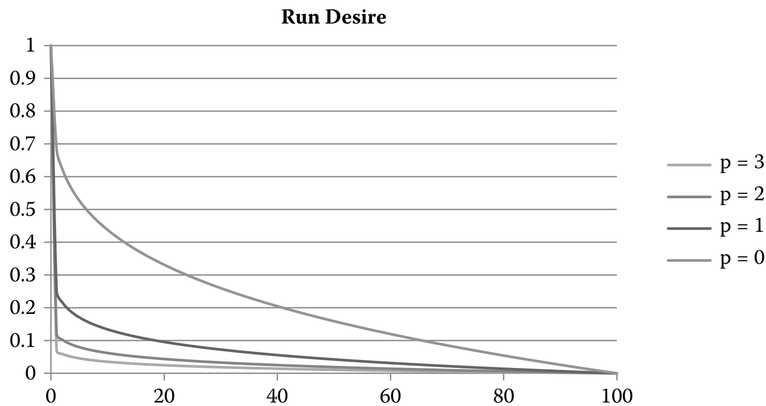

Figure 9.11

The Run Desire curve for several values of *p*.

These curves all represent the various decision factors the AI will use to choose one of the three options. The decision to attack is based entirely on the Attack Desire curve. The decision to heal is based on the Health Desire curve multiplied with the Threat curve. This mitigates the case where the actor may want to heal prematurely and changes behavior based on the maximum damage of the player. It's worth noting that we originally had this set to the average damage the player can do since that's more in the spirit of *expected* utility, but this didn't yield strict enough results. This is a good example of how tuning utility curves and formulas is more art than science. The decision to run away works the same way as the decision to heal; it multiplies in the desire to run with the threat.

9.8.2 Decision Making and Gameplay

The demo is a turn-based duel between you and a ferocious utility curve. You start first by choosing one of the available actions, and then your opponent decides how to respond. This continues until one of you is dead or has run away. The AI opponent starts by calculating the score for each decision factor by running through the utility formulas above. The score for each action is calculated by combining the decision factors. The Attack action is just the outcome of the Attack Desire decision factor. The Run action is scored by multiplying the Run Desire decision factor with the Threat. The Heal action is scored by multiplying the Heal Desire decision factor with the Threat. The final decision is chosen with a weighted random.

9.9 Conclusion

Utility theory is a very powerful way to get rich, life-like behavior out of your AI and has been used in countless games of nearly every genre. It can be extremely fast, especially if you choose simple utility functions, and it scales very well. One of the great appeals of this system is the amount of emergent behavior you can get with just a few simple values and a handful of weights to add some personality. By combining decision factors in

meaningful ways, you can build up decisions from these atomic components to provide very deep AI.

In this article, we had a whirlwind tour of utility theory and how it can be applied to games. We showed you some basic principles of decision making and dug into the math behind it. With the tools in this article and a bit of work, you can build a powerful, emergent AI system.

References

[Dill 11] K. Dill. "A game AI approach to autonomous control of virtual characters." *Interservice/Industry Training, Simulation, and Education Conference, 2011*, pp. 4–5. Available online (http://www.iitsec.org/about/PublicationsProceedings/Documents/11136_Paper.pdf).

[Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Reading, MA: Charles River Media, 2009, pp 229–240.

[Russell et al. 09] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Reading, MA: Prentice Hall, 2009, pp. 480–509.