10

Building Utility Decisions into Your Existing Behavior Tree

Bill Merrill

- 10.1 Introduction
- 10.2 Why Behavior Trees?
- 10.3 It's Not All Candy Canes and Gum Drops
- 10.4 What Is Utility Theory?
- 10.5 Applying Utility in Decision Making

- 10.6 The Utility Selector
- 10.7 Propagating Utility
- 10.8 A Twist on Behavior Trees: Evaluation versus Execution
- 10.9 Conclusion

10.1 Introduction

While there is no "silver bullet" approach to authoring AI behavior, behavior trees tend to strike a strong overall balance across ease of implementation, ease of visualization, and adoptability for new team members. Building supporting tools is straightforward, and a rapid workflow can be established in relatively short order. On the other hand, behavior trees have a fundamental limitation. They are poor at modeling analog concepts such as uncertainty over multiple valid options. Game characters are simply fun and engaging machines for players to interact with and even exploit, but requirements still often demand more than strictly Boolean selection logic. Because hand-coding analog selection logic everywhere that it is required gets messy quickly, a better solution is needed.

Utility-based decision-making addresses this problem head-on. Rather than creating variation through randomness or forcing agents to arbitrarily take one valid option over another, we can apply existing, well-documented techniques to deal with "gray area" decisions in an elegant manner. Most satisfyingly, we can do all of this without uprooting an existing implementation by imposing structural changes and we don't have to give up any of the most desirable traits of behavior trees.

This article proposes a few simple components that enable the integration of utility considerations into a behavior tree's normal selection process. The express goal of this integration is to overcome much of the behavior tree architecture's biggest weaknesses without sacrificing its strengths.

10.2 Why Behavior Trees?

Behavior trees have been growing steadily in popularity, and for good reason. Put simply, they offer a very pragmatic approach to making decisions. It is their simplicity I find most valuable, and in a world of increasingly complex software, simplicity should not go undervalued. The aggregation of systems comprising AI in modern games is becoming vast, and isn't shrinking anytime soon. It's important to find what works for the team and try not to force new learning curves unnecessarily.

Behavior trees have become somewhat of a standard in the industry. Many AAA studios make use of behavior tree technology, including Bungie with the Halo series [Bungie 07], and Crytek with Crysis 2 [Crytek 11]. A wealth of information on behavior trees is available online and existing toolkits are available for developers looking to get started quickly [Champandard 08, Brainiac 09]. This implementation included with this chapter is basic, but also relatively complete and free for any use. To see more on how developers are constantly improving on the traditional behavior tree, Alex Champandard's behavior tree toolkit provides tips on how to implement your behavior tree to optimize performance and memory access patterns on systems such as consoles that demand the extra attention [Champandard 12].

For my team in particular, the accessibility and scalability of behavior trees has us using them as our primary mechanism for decision making. Productivity depends largely on designers, scripters, and animators gaining a clear understanding of how a given character intends to behave and react to change. We never fully attained this when using a STRIPS-based planner or our finite state machine (FSM) prior to that. In both cases, so much of what was occurring "under the hood" was largely opaque to anyone other than the AI programmer(s). In the case of the planner, it was too organic and mysterious for the designers' comfort. Additionally, emergent behavior is still not a desirable feature most of the time. Planners also make it difficult to string together specific sequences of actions in a defined order when needed. As for FSMs, the lack of true modularity, complex state logic, and a tendency to get messy negatively affected both the designers and programmers alike. This was especially true as NPC characters developed over time, requiring more complex transition logic. Special cases seemed to become the norm, generating many uninvited surprises along the way as we attempted to share functionality with past projects and other game teams.

With behavior trees, our designers can effectively visualize what's happening, in real-time, and can intuitively apply changes or additions with a clear picture of what to expect. Programmers can also easily oversee the changes. Tree structures are a familiar and relatively easy concept to digest, with many designers industry-wide already using them in some form for major aspects of their workflow. This enables us to provide the designers with easily-adoptable debugging and authoring tools. This is invaluable during development when creating complex tools and chasing bugs can steal precious resources from iteration and content creation.

10.3 It's Not All Candy Canes and Gum Drops

The features required by a game change constantly, putting strain on nearly every system in your codebase. AI is particularly susceptible to this problem because it's driven directly by design, and changes more rapidly over the course of a project than other systems of similar breadth and complexity. It's our responsibility as programmers to question the fitness of our solutions in addressing the problems at hand. In terms of decision making, I found myself regularly questioning the fitness of behavior trees while implementing behaviors that didn't have easily quantifiable static priority, or didn't intuitively distill down to simple yes/no criteria.

In a standard behavior tree, priority is static. It is baked right into the tree. The simplicity is welcome, but in practice it can be frustratingly limiting. The same behavior may require different relative priorities, depending on the context. Ensuring our Monster Hunter's primary weapon has a full clip should always be a consideration, even if we're casually patrolling the jungle. But if we're engaged with a savage monster, it's absolutely necessary that we continue to deal damage. Behavior tree authors often deal with this conundrum by duplicating sections of the tree at different branches, with different conditions and/or priorities. Even with slick sub-tree instancing or referencing, this still becomes inefficient, verbose, and potentially fragile.

Even more troublesome cases surface when a simple yes versus no determination isn't easily established. If our *Combat* selector is evaluating its options, should it choose to have us seek a rendezvous with our medic and his space-age healing tech, or should we put everything we have into quickly dropping the giant alien beast threatening to eat us all? This sort of decision is best made only after considering a potentially broad combination of inputs.

Decisions are rarely binary, and many behaviors simply do not have priorities we can comfortably establish offline. Let's start with a simple example behavior tree (Figure 10.1). Having no ability to shoot is a precondition for the *Seek Medic* behavior, forcing us to duplicate the behavior, as seen in Figure 10.2. We could start by giving *Seek Medic* stricter conditions and prioritizing it over *Shoot*, but this will likely create the opposite problem where the Monster Hunter immediately takes the *Seek Medic* action the instant conditions pass. This is the sort of fundamental problem we want to address with the integration of utility.



Figure 10.1

Here is a simple, minimal behavior tree for the Monster Hunter.



Figure 10.2

In order to implement *Seek Medic* with two different priorities depending on runtime conditions, we're limited to duplication within the tree.

10.4 What Is Utility Theory?

As it applies to game AI behavior, utility theory is simply the process of measuring the relative suitability of a particular action [Mark 09]. To make good decisions, we need to quantify how *worthwhile* an option is, given all the relevant facts, rather than make a determination on validity alone. Industry veterans who advocate the use of utility theory like to remind us that there is rarely just one correct decision to make. So the question is: why do we still favor decision-making architectures that fail to address this problem elegantly?

In reality, an agent of moderate complexity may have dozens of potential options on the table at once. There may even be several perfectly sensible options. Utility theory recognizes that decisions are seldom black and white, and attempts to formally address the complexities of combining various pieces of analog information together to make a final determination. Figuring out how to identify and compare the information in a logical manner is much of the challenge. The most important goal is to ensure that the overall computation is reliable given any combination of inputs, and always results in a reasonable choice.

10.5 Applying Utility in Decision Making

Game agents are *approximations* of autonomous entities within the limited scope of a game's specific design. For this reason, it's not worth the effort in most games to deeply analyze mountains of data for the purpose of AI decision making. Going too broad with the inputs effectively dilutes their meaning, resulting in muddy, or even illogical, formulas. All that should concern us is building an experience that feels believable and engaging to the player within the context of the game.

It's worth first making an effort to represent the input values in a manner that enables direct comparison. This helps avoid a confusing apples-to-oranges quagmire. One easy way to accomplish this is to identify a common unit of measurement. It can be a lot like solving a system of equations. We can substitute one variable with some combination of other, better understood variables. If we're combining two inputs, it makes sense to represent them both in terms of time, health, ammo, a rate of growth/consumption, or something even more abstract. For example, if our Monster Hunter is low on health and wishes

to consider rendezvousing with the squad's medic for a health boost, we can measure the benefits of receiving treatment in health points gained. However, running frantically to a safe position is likely to gain the attention of the alien beast, putting us at a risk. If we can measure the risk by predicting the health we're likely to lose in transit, both inputs are now in terms of health points and can be combined and/or compared directly, as in Equation (10.1). We could simply take their sum, and if the net value is positive, taking this action has some benefit we can weigh against other actions.

$$RawUtility = HealthGained - HealthLost$$
(10.1)

More desirably, by attaching more weight to the amount of health we'll lose in transit, we can ensure that we only take this action if we expect to net a significant amount of health, as seen in Equation (10.2). After all, breaking even would be a waste of the time we could've otherwise spent slaying the creature. We also want a high degree of confidence that, even if our predictions were overly optimistic, we're unlikely to end up with a net loss in health and looking rather boneheaded as a result. Naturally there's more we could do, such as apply an exponential scale to *HealthLost*, which causes the utility to fall off more rapidly as the risk grows, as in Equation (10.3).

$$Value = HealthGained - (HealthLost \times 2.0)$$
(10.2)

$$Value = HealthGained - (pow(HealthLost, 1.2))$$
(10.3)

What happens if we're unable to represent our input values in such easily relatable units, and we wish to consider much more than just a net change in health? One way to combat this scenario is to combine the various influences into higher-level, more abstract values such as "Morale," "Threat," etc. The utility of running to visit our medic could also take into consideration the lost time we could've otherwise spent damaging the monster. Specifically, we could take our formula above, normalize the result, and classify it as a "Heal" factor. Next, we could generate a second formula representing this time lost, normalize it, and classify it as "Delay." We now have two normalized quantities representing higher-level valuations, which we can combine into a final utility value.

$$Utility = \frac{(Heal * HealPower - Delay * DelayPower)}{HealPower + DelayPower}$$
(10.4)

I have glossed over the concept of normalization in our example above. However, in order to logically compare apples to oranges, the normalization process is fundamentally important, as it essentially "bakes" more complex underlying computations into a single usable value. Typically this involves running a raw value (health, time, ammunition, damage, etc.) through a normalization function to generate a real number from 0 and 1. Normalization functions are most commonly linear, exponential, or sigmoidal, but can be of any form. Response curves are an elegant solution in cases where a single formula is not sufficient for representing the desired normalization, allowing the curve to be broken up into segments that can be further fine-tuned [Mark 10]. The curve you choose can dramatically impact the result, and thus are often the target of on-the-fly tuning. For this reason, I'd recommend building these formulas into components you can represent as reloadable data that you and your designers can tweak. Normalization is a deep subject, and much wisdom can be discovered in available material. Papers available on GameAI.com, the GDC Vault, and the reading material referenced herein all provide excellent background on utility-based AI and behavioral modeling.

10.6 The Utility Selector

The behavior tree structure lends itself well to extensibility. After all, it's nothing more than a tree traversal where the nodes themselves are responsible for and are able to customize the expansion of the tree. The tree already features a component for selecting which branches are taken during execution, namely the *selector*. To introduce utility-based selection, we'll simply create a new specialized type of selector that considers not just the binary validity of its children, but their relative utility as well. We'll cleverly dub the new node type the *utility selector*.

For simplicity's sake, let's consider a vanilla behavior tree implementation. Each execution pass will traverse the tree until a busy node is encountered, at which point execution will yield until the next update. When a utility selector executes, it first queries each child sub-tree for a utility value. If we gather these results first, we can apply any one of several selection methods. For one, we could simply take the child with the highest utility. Alternatively, we could sort the children into buckets and conduct a weighted random selection. Depending on the scenario, we could even apply an unweighted random selection among the children with utility values over some threshold beyond which options are considered desirable. All we're essentially doing is adding utility-gathering to a standard selector, and using the data to determine priority dynamically. With minimal effort, we've busted wide open what is arguably the biggest drawback of behavior tree-based architectures—static priorities. In fact, we can address our problem with *Seek Medic* by switching *Combat* to a utility selector, as we've done in Figure 10.3.



Figure 10.3

The *Shoot* vs. *Seek Medic* conundrum has been solved by converting *Combat* to a utility selector in the original tree.

Listing 10.1. Pseudocode for a basic selector.

```
Status Execute()
{
    if(CurrChild == null) then CurrChild = FirstChild;
    //Execute all children until we encounter a valid one.
    while(CurrChild != null)
    {
        Status s = CurrChild.Execute();
        if(s == Busy || s == Done) return s;
        CurrChild = CurrChild.Next;
    }
    return Failed;
}
```

```
Listing 10.2. Pseudocode for a basic utility selector.
Status Execute()
    if(Utility.Size() == 0) then
    ł
        //Query for child utility values.
        for(CurrChild = FirstChild; CurrChild != NULL; CurrChild =
CurrChild.Next)
           Utility[CurrChild] = CurrChild.CalculateUtility();
        //Sort from highest utility to lowest.
       SortChildrenByUtility();
       CurrChild = FirstChild;
    //Evaluate in utility order and select the first valid child.
   while(CurrChild != null)
       Status s = CurrChild.Execute();
       if (s == Busy) then return Busy;
       else if(s == Done) then
           Utility.Clear();
           return Done;
       CurrChild = CurrChild.Next;
    }
   Utility.Clear();
   return Failed;
}
```

10.7 Propagating Utility

The utility selector simply queries its children for their utility values. Typically only leaf behaviors will conduct utility calculations, but the utility selector's children may be of any node type, including composite nodes or even another utility selector. For utility information to intuitively propagate up the tree, we need to override CalculateUtility() for all composite node types.

For both selectors and sequencers, the simplest method is to return the highest utility value gathered from its own children. Consequently, in order to gather necessary utility data, a utility selector must expand all nodes in its child sub-trees, potentially conducting large quantities of utility calculations in a single pass. This may or may not be a problem depending on the scale you're working with, but with complex utility calculations in large behavior trees on platforms sensitive to random memory access patterns, it's certainly not ideal.

Thankfully, there are ways to mitigate this problem. For one, we could limit utility calculations to some interval within our leaf behaviors' implementations, and return cached values. Alternatively, we could compute utility values for all of our tree's leaf nodes within a completely separate pass, with its own load balancing, leaving only cached values to be used during calls to CalculateUtility().

10.7.1. Transforming Utility During Propagation

For additional flexibility, nodes can choose to modify utility as it works its way up the tree. *Decorators* are a fundamental concept in behavior trees, referring to single-child nodes that can be used to introduce various useful behavior features. Some common examples include repeating the child node *n* times, monitoring a runtime condition, or limiting the child's execution time, but they're a general-purpose tool with infinite potential uses. In fact, there's nothing stopping us from creating a utility decorator that applies some transformation to the utility value of its child. Perhaps it could multiply its child's utility by some factor for weighting purposes, or it could run the value through a custom function.

To provide a simple example, let's say our *Reload* behavior is a black box that internally computes a normalized utility value. Under most circumstances, we may choose to compare *Reload*'s utility directly to that of its siblings. However, we may encounter a case in our game where we wish to limit *Reload*'s utility until we're desperate for ammunition. We can accomplish this goal by adding a utility decorator above *Reload* that runs the utility value through a simple square() or cube() function, as illustrated in Figures 10.4 and 10.5.

10.8 A Twist on Behavior Trees: Evaluation versus Execution

The behavior tree coupled with this chapter differs from some traditional implementations in that it separates the idea of tree evaluation from actual execution, which I've also done in the version I use for professional work. Doing so provides opportunity for a few improvements over a typical behavior tree implementation. One of those opportunities is to more optimally integrate utility-based decisions. Most notably, the utility selector is able to evaluate its children prior to calculating utility, meaning it must update utility only for valid children. This can be seen in the accompanying source code's UtilitySelector implementation. Furthermore, leaf nodes with costly utility calculations can do the work while verifying its conditions in Evaluate(), and return a cached value in CalculateUtility().



Figure 10.4

We've added a decorator to modify Reload's standard utility value at execution time.



Figure 10.5

Reload's utility is now being cubed as it propagates up the tree, delaying the urgency to reload.

Another useful benefit is the ability to evaluate the tree independently of an agent's behavior execution. While an agent is actively executing behaviors, we can freely evaluate the tree in parallel without interfering with the executing nodes, and only interject if the results vary from the presently executing plan.

Evaluating the tree in its entirety also means we can optionally perform a limited version of look-ahead planning, since we can ensure that an entire plan is valid to the end, at least at the time of evaluation, before committing any of it to the agent. In cases where this is not desirable, nodes can still defer validation until they are executed, enabling them to behave as they would in a typical behavior tree flow.

10.9 Conclusion

Behavior trees and utility are both powerful concepts, made practical by their ease of implementation and experimentation. If you haven't done so already, I highly recommend tinkering with them as a potential solution in your professional endeavors. When combining utility behavior trees, these two otherwise disjointed techniques can help tackle the wide variety of behavioral problems found across genres and scopes.

We started with a straightforward behavior tree implementation, and without making any fundamental changes, we've introduced the ability to blend in utility-based decisions *only where desired*, preserving the tree's default behavior elsewhere. While the examples here are limited in scope for clarity, you've hopefully identified cases where this will help you solve real-world problems you've already encountered while applying behavior trees in practice. Beyond that, hopefully you can make use of utility-based decisions to improve your characters' behaviors further.

I am continuing to develop the coexistence of behavior trees and utility for my own needs in a very demanding commercial project, featuring dozens of unique NPC characters spanning a wide range of classifications. I wanted to share my discoveries thus far, as the results have been pleasantly surprising in practice. We've been able to represent characters ranging from simple wildlife to autonomous beasts with a vast repertoire of special abilities to soldiers with unique and obscure capabilities that must effectively emulate human players, all with the same behavioral foundation and toolset. For example, giant beasts can weigh different types of attacks against multiple targets dynamically, and human soldiers can evaluate and use their deep inventories to cooperatively take down targets, heal and revive teammates, and combine strategies. The integration of utility helped tremendously in mitigating complexity since characters can weigh multiple factors during decision making in a manner that's intuitive and "just makes sense." Rather than fight against the limitations of a single textbook architecture, a simple-to-implement hybrid has provided a great deal of power without sacrificing usability.

If you have questions, suggestions, or simply want to discuss something nerdy, don't hesitate to email bill.merrill at outlook.com.

References

[Brainiac 09] "Brainiac Designer." http://brainiac.codeplex.com/, 2009.

- [Bungie 07] M. Dyckhoff. "Evolving Halo's Behavior Tree AI." http://www.bungie.net/images/ Inside/publications/presentations/publicationsdes/engineering/gdc07.pdf, 2007.
- [Champandard 08] A. Champandard. "Behavior Trees for Next-Gen Game AI." http://aigamedev.com/insider/article/behavior-trees/, 2008.
- [Champandard 12] A. Champandard. "Behavior Tree Starter Kit." http://aigamedev.com/ ultimate/release/behavior-tree-starter-kit-source-release/, 2012.
- [Crytek 11] R. Pillosu. "Coordinating Agents with Behavior Trees." http://staff.science.uva.nl/ ~aldersho/GameProgramming/Papers/Coordinating_Agents_with_Behaviour_Trees. pdf, 2011.
- [Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Boston, MA: Charles River Media, 2009.
- [Mark 10] D. Mark and K. Dill. "Improving AI Decision Modeling Through Utility Theory." http://www.intrinsicalgorithm.com/media/2010GDC-DaveMark-KevinDill-Utility-Theory.pdf, 2010.