Copyright Material – Provided by Taylor & Francis

31

Behavior Decision System

Dragon Age Inquisition's *Utility Scoring Architecture*

Sebastian Hanlon and Cody Watts

31.1 Introduction

The real-time combat sequences of *Dragon Age: Inquisition* (*DA:I*) pit the player-controlled Inquisitor in a fight to the death against shambling undead, fearsome demons, and—of course—towering dragons. It is a tough job, but the Inquisitor is not alone; fighting alongside his or her are three AI-controlled allies—fellow members of the Inquisition such as The Iron Bull, a towering, axe-swinging mercenary, Varric Tethras, a smooth-talking dwarf, and Dorian Pavus, a charming and quick-witted mage.

In *Dragon Age*, combat is driven by "abilities." A typical combatant will have anywhere from 2 to 20 abilities at their disposal, ranging from the simple ("hit your foe with the weapon you are holding") to the elaborate ("call down a rain of fire upon your enemies"). Each ability has an associated cost expressed in terms of a depletable resource called "mana" or "stamina." Because mana/stamina is limited, abilities also have an implicit opportunity cost—the mana/stamina consumed by one ability will inevitably preclude the use of future abilities. This creates a challenging problem for human players and AI characters

Copyright Material – Provided by Taylor & Francis

alike: When faced with limited resources, a plurality of choices, and a constantly changing game state, how can a combatant quickly identify the course of action which will yield the greatest possible benefit? For *DA:I*, we created a utility-based AI system called the Behavior Decision System (BDS) to answer this question and to handle the complex decision-making which combatants must perform. In this chapter, we will describe the principles and architecture of the BDS, providing you with the necessary information to implement a similar system and extend it to meet the needs of your own game.

31.2 The Behavior Decision System

The architecture of the BDS is based upon the following assumptions:

1. At any given time, there is a finite set of actions which an AI character can perform.
2. An AI character can only perform one action at a time.
3. Actions have differing utility values; some actions are more useful than others.
4. It is possible to quantify the utility of every action.

When taken together, these assumptions naturally suggest a simple greedy algorithm to determine an AI character's best course of action: Start by identifying the set of actions which it can legally take. Then, evaluate each action and assign it a score based on its utility. After each action has been evaluated, the action with the highest score is the action which should be taken (Graham 2014).

There are two major challenges to this approach. First, how can an AI character enumerate all the actions which it can perform? Second, how can an AI character qualify the utility of an action? Before an AI character can answer these questions, we must first impart it with knowledge—knowledge about itself and the world in which it lives.

Consider, for example, a simple action such as "drinking a health potion." Most human players know that it is useful to drink a health potion when their health is low. Unfortunately, AI characters do not intuitively understand concepts like life, death, health, and healing potions. They do not know that being alive is "good" and being dead is "bad." They do not understand that drinking a health potion at low health is good, but drinking a health potion at full health is wasteful. And they do not understand that health potions are consumable objects, and that one cannot drink a health potion unless one owns a health potion.

At its most basic level, the BDS is a framework which allows gameplay designers to impart knowledge to AI characters. Specifically, the BDS exists to provider answers to the following questions: Which actions can an AI character perform? Under what circumstances can they perform those actions? How should those actions be prioritized relative to each other? And finally: How can those actions actually be performed?

31.3 Enumerating Potential Actions

There are more than 60 abilities in *DA:I*, but many of these abilities can be used in different ways to achieve different purposes. For example the "Charging Bull" ability allows a warrior to charge directly into combat, damaging and knocking-aside any enemy who stands in his or her way. This is its intended, obvious purpose. However, this same ability can also

Copyright Material – Provided by Taylor & Francis

be used as a way for an injured warrior to quickly retreat from combat. Though the underlying ability is the same, the motivation for the ability is completely different. Therefore, when enumerating potential actions, it is not sufficient to simply count the number of abilities at a character's disposal—we must also include the various ways in which those abilities can be performed. In order to distinguish between the various ways an ability can be used, we defined a data structure called a "behavior snippet."

Behavior snippets are the fundamental unit on which the BDS operates. Each snippet contains the information an AI character requires to evaluate and execute an ability in a particular way. In a sense, a snippet represents a fragment of knowledge—and throughout the game knowledge can be granted to characters by "registering" a snippet with to a character via the BDS. For example, a piece of weaponry may have one or more behavior snippets attached which tell AI characters how to use the weapon. When an AI character equips the weapon, these snippets will be registered to the character through the BDS. Similarly, when the weapon is unequipped, the snippets will be unregistered from that character.

Behavior snippets make it simple for the BDS to enumerate the list of actions available to a character; one merely needs to look at the set of registered snippets. The most complex AI characters in *DA:I* have over 50 behavior snippets registered simultaneously, though an average AI character will have 10–20 registered snippets.

31.4 Evaluating Behaviors

A behavior snippet contains the information an AI character requires to evaluate and execute an ability in a particular way—but what exactly does this mean? As stated previously, the BDS is based upon the assumption that it is possible to quantify the utility of every action. In order for this assumption to hold, each behavior snippet must contain within it a method to quantify the utility of the action it represents. There are many possible ways to quantify utility, but for *DA:I*, we chose to represent utility using a modified behavior tree which we call an "evaluation tree."

31.4.1 Calculating Utility

In the broadest possible terms, the purpose of an evaluation tree is simply to produce a score value for its associated behavior snippet. These scores can then be used as a basis for comparing two behavior snippets against each other in order to rank their relative utility. Score values are assigned via "scoring nodes" embedded within the evaluation tree itself. When the tree begins executing from its root node, it starts with a score of zero. As the tree progresses from one node to the next, any scoring nodes it executes will add their value to the tree's total score.

Evaluation trees are evaluated "in context"—that is to say, the nodes within the tree have access to information such as the AI character who is performing the evaluation. This allows for the creation of evaluation trees which produce different scores depending on the context in which they are evaluated (Merrill 2014). For example, Figure 31.1 shows an evaluation tree which will return a score of 5 if the evaluating character's health is less than 50%, and a score of 0 otherwise.

When constructing our scoring system, we considered various schemes for automatic normalization or scaling of scoring values. Ultimately, we chose to use (and recommend using) a designer-facing scoring convention to provide a framework for how actions

Copyright Material – Provided by Taylor & Francis

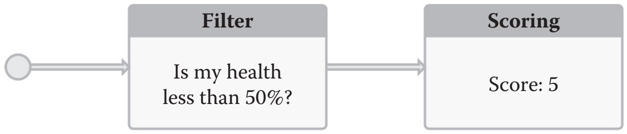

Figure 31.1

This evaluation tree returns different scores based on the health of the evaluating character.

Table 31.1 Scoring Framework Used in *Dragon Age: Inquisition*

Action Type	Point Values	Description
Basic	10	Preferable to doing nothing, and if multiple options are available, they are equivalent to each other.
Offensive	20–40	As a class, always preferable to basic actions, and compared against each other with 20 points of "urgency dynamic range" for prioritizing based on situational influences.
Support	25–45	As a class, preferable to offensive actions with the same level of "urgency," as they are either preparatory and should be used before engaging offensively, or used in reaction to emerging bad situations.
Reaction	50–70	All actions in this class have evaluation trees that respond to specific and immediate execution criteria (typically responding to an imminent threat to the AI character); if these criteria are present these actions should be executed in priority over any other (and otherwise valid) choices.

should be scored relative to each other. Note that these rules are guidance for content creators and have no explicit representation in game data. Table 31.1 shows an example of a scoring convention.

It is the responsibility of the designer constructing the evaluation trees for each snippet to conditionally allocate score so that the tree will produce a value within the appropriate dynamic range. In *DA:I*, each class of action uses a different set of scoring logic assets, built to return score values within the appropriate range. For example, the evaluation tree for a "Support" action starts by granting a baseline 25 points and conditionally adds contextual score up to a maximum of 45.

31.4.2 Target Selection

Most abilities in *DA:I* require a target to function. For example, an AI character cannot simply "cast Immolate"—they must cast Immolate *on* a specific foe. The chosen target of an ability can greatly affect the outcome (i.e., the utility value) of executing that ability. Consider: Casting Immolate on a target who is weak against fire will deal significant damage, whereas casting it on a target who is fire-immune will do nothing but waste mana. For that reason, target selection is a necessary part of the evaluation step; to accurately represent the value of an action, we must consider all the potential targets of that action, and then select the target which provides the greatest utility value. Therefore, in the BDS framework, evaluation trees return not only a score, but a target too.

The contextual nature of evaluation trees allows us to add target selection logic by introducing a "target selector" node. This node iterates over a list of designer-specified

Copyright Material – Provided by Taylor & Francis

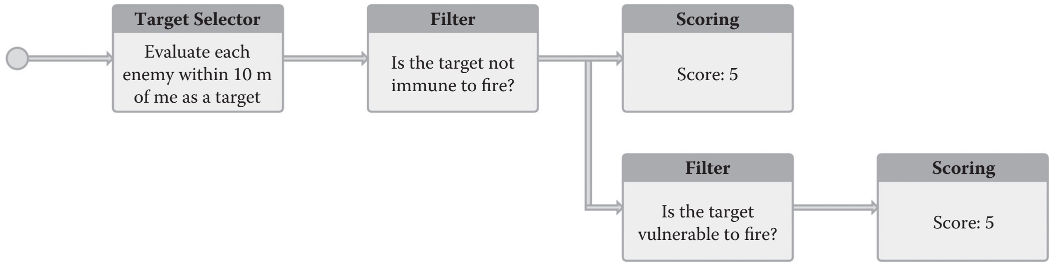
Target Selector
Evaluate each enemy within 10 m of me as a target
Filter
Is the target not immune to fire?
Scoring
Score: 5
Filter
Is the target vulnerable to fire?
Scoring
Score: 5

Figure 31.2

This evaluation tree specifically targets hostiles who are not immune to fire. Targets who are vulnerable to fire are scored higher than those who are not.

targets, and evaluates and assigns a score to each of them separately. Figure 31.2 shows an evaluation tree which makes use of the target selector node.

The target selector node maintains its own temporary evaluation state and record table, and executes according to the following algorithm:

1. Record the evaluation context's current score value as the "initial score."
2. For each targetable specified by the iterator data:
   a. Reset the context's score value to the initial score.
   b. Set the context's "behavior target iterator" slot to the current targetable.
   c. Evaluate the child node.
   d. If the child node returns true, record the context's current score value for the iterator's current targetable.
3. If at least one targetable has been recorded with an evaluation score:
   a. Set the context's Behavior Target slot to the targetable with the highest score.
   b. Set the context's score value to the score associated with that targetable.
   c. Return true.
4. If no targetables were recorded with an evaluation score, return false.

In this way, multiple targets are compared and only the target that generates the highest score for each snippet is associated with that snippet in the main BDS evaluation table.

31.4.3 Comparing Snippets

As part of the AI character's update, the evaluation tree for each registered behavior snippet is run, and the score & target produced by that tree is stored along with the snippet in a summary table. When all of the evaluation trees have been run, the snippet with the highest recorded score is selected to be executed. This cycle is typically repeated on every AI update pass, but can be performed as often as appropriate. Listing 31.1 contains a pseudocode implementation of this evaluation step.

It is not strictly necessary to retain any of the per-snippet evaluation results beyond the scope of the evaluation step; the execution step requires only a reference to the highest priority snippet and the selected targetable. In practice, though, we found that retaining the evaluation results in a debug-viewable table provides great insights when debugging and iterating on AI behavior.

Copyright Material – Provided by Taylor & Francis

Listing 31.1. Pseudocode for evaluating registered behavior snippets.

```
struct SnippetEvaluation
{
    BehaviorSnippet snippet;
    Boolean result;
    Integer score;
    Character target;
};

// This function evaluates registered behaviors
// and returns the one with the highest utility.
Optional<SnippetEvaluation> EvaluateSnippets()
{
    list<SnippetEvaluation> evaluatedSnippets;
    for (BehaviorSnippet snippet: registeredBehaviors)
    {
        SnippetEvaluation evaluation = snippet.evaluate();
        if (evaluation.result == true)
            evaluatedSnippets.push(evaluation);
    }

    sortByDescendingScore(evaluatedSnippets);
    if (evaluatedSnippets.empty() == false)
        return evaluatedSnippets.first();
    else
        return None;
}
```

31.5 Execution Step

Having identified the highest scoring snippet, and a target for its associated ability, the final step is to execute that snippet. Just as each snippet contains an evaluation tree to show how the behavior should be evaluated, it also contains a behavior tree (termed the “execution tree”) to show how the behavior should be executed. The execution tree is responsible for including any preparation or positioning required before performing the animation and game logic for the effective part of the action: the “active execution” of the action.

Like evaluation trees, execution trees have access to contextual information when being executed. Specifically, the BDS exposes information about the target which was selected during the evaluation step and the ability that the snippet is associated with. In order to simplify our execution trees, we defined a task node called “Execute Ability” which simply triggers the AI character to use the contextually-specified ability against the contextually-specified target. Figure 31.3 shows a typical “move-into-range-and-strike” execution tree.

Storing information about the ability and the target in the context (rather than explicitly referencing them in the execution tree) allows execution trees to remain generic, thus enabling their reuse across several different snippets. For example, the execution tree shown in Figure 31.3 could be applied to a punching attack, a stabbing attack, or a biting attack—just as long as the behavior occurs at melee range.

The execution tree belonging to the behavior snippet selected during the previous BDS evaluation pass will be executed once every AI update pass until the “Execute Ability”

Copyright Material – Provided by Taylor & Francis

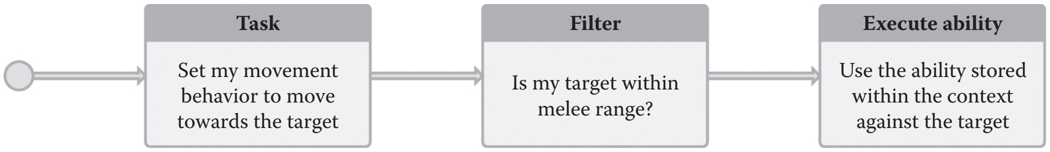
Task
Set my movement behavior to move towards the target
Filter
Is my target within melee range?
Execute ability
Use the ability stored within the context against the target

Figure 31.3

This execution tree handles moving to achieve range and line-of-sight before executing a ranged attack and stopping movement.

node is fired, signaling that the execution of the behavior is complete. However, even while an AI character is executing a particular snippet, it is still desirable for the BDS to continue performing the evaluation step. Reevaluating in this way allows AI characters to execute new snippets in response to changing circumstances rather than mindlessly carrying out a previously selected snippet which has since become suboptimal. In fact, the contract between evaluation and execution logic is that the evaluation tree is responsible for identifying and guarding against any conditions which would make it impossible to fulfill the directives contained within the execution tree. In circumstances where an execution tree has become impossible to complete (e.g., if the target of the execution dies before the ability can be used) then reevaluation ensures that the now-invalid snippet will be replaced with a valid one. Having said that, once an AI character triggers the "Execute Ability" node, it is reasonable to suspend AI updates until the ability finishes executing; this minimizes wasted AI decisions that cannot be fulfilled while the character is occupied.

31.6 Movement and Passive Behaviors

Although the BDS was originally designed to prioritize, prepare, and execute discrete actions, in the course of developing *DA:I*, we discovered that the BDS evaluation-execution framework is also useful for regulating ongoing or "passive" behaviors.

For example, in *DA:I* if the player's AI-controlled allies have nothing else to do, they will simply follow the player, wherever he or she goes. This was incorporated into the BDS by registering a snippet whose evaluation tree simply returned a constant score lower than any action (e.g., a score of 0 in the context of the scoring system in Table 31.1) and whose execution tree does nothing but trigger the "follow the leader" movement behavior. This snippet is automatically invoked by the BDS when it becomes the character's highest priority (i.e., when no other snippets with scores of greater than 0 are viable.)

Further application of this approach allows us to use the BDS to choose between contextually appropriate movement behaviors by conditionalizing scoring logic just as we do for combat abilities. *DA:I* uses this approach to apply variations on the follower behavior if the party is in combat, or if the player has commanded a party member to remain at a certain location; these evaluate conditionally to higher priorities than the basic party movement while still yielding to active actions.

It can also be useful to create behavior snippets which conditionally exhibit extremely high priorities, as this will suppress the execution of any other actions that might otherwise be viable. *DA:I* uses this method on characters who are expected to remain within a

Copyright Material – Provided by Taylor & Francis

certain "tethered" area. For these characters, we created a behavior snippet whose execution tree simply forces the AI character to return to the center of their assigned area. The corresponding evaluation tree returns a score higher than any other combat ability—but only when the character is positioned outside their assigned area. In this way, we ensure that if an AI character strays too far from their assigned position, they will always disengage from their current target and return home rather than allowing themselves to be drawn further and further away.

31.7 Modularity and Opportunities for Reuse

Through its use of behavior snippets, the BDS emphasizes a modular approach to AI design. A modular approach offers several benefits. When debugging, it allows developers to easily isolate the evaluation logic for a specific behavior, or to compare an AI character's relative priorities by examining the results of the evaluation step.

The modular design also allows behaviors to easily be shared or moved between AI characters and archetypes. *DA:I* leverages this functionality to allow human players to customize their AI-controlled party members. Throughout the game, the player can add, remove and modify AI characters' equipment and abilities. By tying behavior snippets to equipment and ability assets, and by following a consistent scoring system (as described in Section 31.4.1) we can ensure that AI characters will be able to make effective use of the equipment and abilities at their command—regardless of what those may be.

Although the desire for modularity was initially driven by the requirements of our AI-controlled allies, the benefits extend to hostile AI characters too. During the development of our hostile creature factions, we found that an ability or behavior which was developed for a specific AI character type could be easily shared with others, assuming that the relevant assets (e.g., animations and visual effects) also available for the new characters.

In order to support the modular design of the BDS, it is important to implement the evaluation and execution tree data structures so that they can be authored once and reused across multiple behavior snippets. In developing *DA:I*, we found that most behavior snippets could be implemented using a small pool of frequently-reused tree assets, whereas only a small number of complex actions required specific evaluation or execution logic. Modular decomposition and content-reuse can be promoted even further by separating commonly-recurring subtrees into standalone tree assets which can then be referenced from other trees. Consolidating scoring logic in this fashion can help reduce the ongoing maintenance cost of implementing a standardized scoring system.

31.8 Conclusion

In this chapter, we have presented the Behavior Decision System: a simple but powerful framework developed to support AI decision-making. At the core of the BDS are "behavior snippets"—data structures which encapsulate the information required to both evaluate and execute a discrete action. Snippets are both evaluated and executed using behavior trees; "evaluation trees" are modified behavior trees, which return both a utility score and a target, whereas execution trees contain the necessary instructions to carry out the action.

Copyright Material – Provided by Taylor & Francis

At runtime, behavior snippets can be registered to AI characters via the BDS, with each registered snippet representing a single action that the character can perform. By evaluating these snippets as part of the character's update loop and regularly executing the snippet which yields the greatest utility score, the BDS produces patterns of behavior which are directed, purposeful, and reactive.

Acknowledgments

The authors would like to thank Darren Ward who implemented the behavior tree system on which the BDS is based, along with Jenny Lee and Chris Dalton who adapted Darren's system to the Frostbite engine.

References

Graham, D. 2014. An introduction to utility theory. In *Game AI Pro: Collected Wisdom of Game AI Professionals*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 113–126.

Merrill, B. 2014. Building utility decisions into your existing behavior tree. In *Game AI Pro: Collected Wisdom of Game AI Professionals*, ed. S. Rabin. Boca Raton, FL: CRC Press, pp. 127–136.