Comput Game J
DOI 10.1007/s40869-017-0040-9

Behavior Trees for Modelling Artificial Intelligence in Games: A Tutorial

Ryan Marcotte[1] · **Howard J. Hamilton**[1]

Received: 14 February 2017 / Accepted: 30 May 2017

Abstract We provide a tutorial introduction to behavior trees, which are a useful way of structuring artificial intelligence in games. A behavior tree is a model of plan execution that is graphically represented as a tree. A node in a tree either encapsulates an action to be performed or acts as a control flow component that directs traversal over the tree. Behavior trees are appropriate for specifying the behavior of non-player characters and other entities because of their maintainability, scalability, reusability, and extensibility. We describe the main features of behavior trees, show an example of how to create a behavior tree, and briefly survey existing packages for editing behavior trees. We recommend that behavior trees be used when some game designers are not programmers, the conditions governing the behavior are complex, and the NPCs have aspects of behavior in common.

Keywords Behavior tree · Game artificial intelligence · Behavior tree editor · Non-player character · Tutorial · Computer game design · Behavior tree example

1 Introduction

A *behavior tree* (BT) is a model for plan execution that is graphically represented as a tree, such as that shown in Fig. 1. A *tree* is a way of structuring information hierarchically that is widely used in Computer Science (Weiss 2013). Figure 1 shows a BT that describes the overall behavior of a "Simple attacker" (warrior) entity that evades enemies, finds first aid, attacks, or wanders. This BT is explained in detail in Sect. 3.

✉ Howard J. Hamilton
hamilton@cs.uregina.ca

[1] Department of Computer Science, University of Regina, Regina, SK, Canada

Published online: 25 July 2017

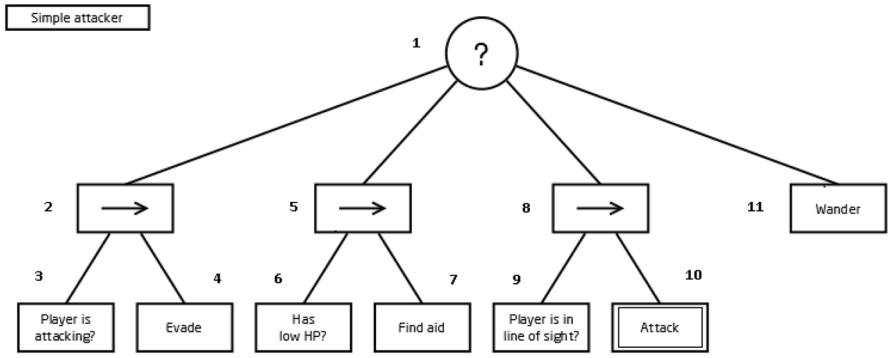

Fig. 1 A behavior tree labelled with the identifier "Simple attacker"

BTs have emerged as a popular tool for modelling artificial intelligence (AI) in games since their use in the *Halo 2* (Bungie Studios 2004) game (Isla 2005) (Isla 2008). The idea for BTs grew out of earlier work on designing behaviors for story characters (Bates 1994) (Mateas 2002) (Mateas and Stern 2005). As with the example given in Fig. 1, BTs are often used to represent the decision making of a non-player character (NPC) (Khoo 2006). They are also gaining popularity as an effective method for controlling robots (Colledanchise and Ogren 2014) (Marzinotto, Colledanchise, Smith and Ogren 2014). Four methods other than BTs of describing the behavior of an entity are: (1) procedures written in an ordinary programming language (Rabin 2010) (2) finite state machines, which describe the behavior in terms of states and transitions between states (Rabin 2010) (3) hierarchical task networks, which require a type of AI software called a planner to create a plan for an entity's behavior that is similar to a behavior tree (Humphreys 2014), and (4) scripts, which represent the behavior in a scripting or behavior language (a high-level language for describing AI behaviors) (Pousman, Mateas and Wolff 2009) (Schenk et al. 2013).

Commonly, a game engine repeatedly executes a game loop, which includes steps for running sub-engines for physics, artificial intelligence, graphics, etc. (Llopis 2010). The AI sub-engine runs the code associated with a game entity, such as an NPC, from time to time. This code, which in our case examines the behavior tree for the entity, plans the behavior for the entity given its current state and its situation in the virtual world. Thus, we say the BT is "executed" during an AI processing step. During such a step, execution of the behavior tree for an entity begins at the root node and proceeds according to a pre-order traversal (Weiss 2013); in other words, first the root node is executed and then each of its child nodes, from left to right, is executed in the same fashion, which may involve executing their child nodes and so on. For example, in Fig. 1, each of the shapes represents a node. Where two shapes are connected by a line, the upper shape represents the parent node, which is executed first, and the lower shape represents the child node, which is executed under the direction of its parent. The top shape (the circle labelled 1) represents the root node, where execution begins. The nodes are executed in the order 1–11 (although not all nodes may be executed during a

particular AI step). Behavior trees are appropriate for specifying behaviors because of their maintainability, scalability, reusability, and extensibility (Champandard 2008) (Flórez-Puga, Gómez-Martín, Gómez-Martín, Díaz-Agudo and González-Calero 2009) (Johansson and Dell'Acqua 2012) (Lim, Baumgarten, and Colton, 2010) (Shoulson, Garcia, Jones, Mead and Badler 2011).

In Sect. 2, we present an overview of fundamental BT concepts. Since no single BT formalism is currently dominant, we use a generic BT formalism that combines typical features. In Sect. 3, we give an example of building a BT. In Sect. 4, we describe the features of five existing software packages for editing BTs, and in Sect. 5, we draw conclusions, mention limitations of the simple BTs described in this tutorial, and present recommendations.

2 Fundamental Behavior Tree Concepts

The BT associated with an AI entity is executed each time the entity may need to act. During execution, every component in a BT that is traversed is executed in the same manner: Given a maximum amount of processor time, it executes for up to that amount of time and then returns a status code to its parent component. The status codes are SUCCESS, FAILURE, RUNNING, and ERROR. If the task performed by the component is completed successfully, SUCCESS is returned. If the task is completed, but not successfully, then FAILURE is returned. If the task performed by the component did not complete and may require more AI steps to finish, RUNNING is returned (Johansson and Dell'Acqua 2012). ERROR indicates that an error occurred during processing, likely due to a programming error (Champandard 2008). Since the ERROR status code is for debugging purposes only, we omit further mention of it. Since the root node of any BT is a component, the BT itself returns one of the status codes.

While many complicated component types could be defined, complex BTs can be built by using only the following fundamental component types: the reference, the action, the condition, the control flow node, and the decorator. After describing these five types of components, we explain why these five types are sufficient to specify all types of behaviors.

A *reference component* is a component that refers to another BT. Reference components allow complex behaviors to be specified using a set of simple sub-behaviors. For example, a new BT can be constructed that incorporates an existing BT by putting a reference component in the new tree that links to the existing BT. A reference component returns the status code produced by executing the BT to which it is linked. A reference component is represented by a double-boxed node labelled with an identifier giving the name of another BT. For example, node 10 in Fig. 1is a reference component that is linked to the "Attack" BT, which will be presented shortly.

An *action* component alters the state of the entity by performing an *action,* which corresponds to executing some game code. Examples of actions include moving the entity in the virtual world, changing the entity's internal state, playing a sound, or executing some specialized logic such as path finding. If the action was completed

successfully, then SUCCESS is returned. If the action was completed but not successfully, FAILURE is returned. If the action requires additional processing after the current AI step, RUNNING is returned. An action component is represented by a box labelled with a statement describing the action. Node 11 in Fig. 1 is an action component that is labelled "Wander".

A c*ondition* component stores a Boolean question, i.e. a question that evaluates to either true or false. The Boolean question can be a test for proximity ("Am I near the player?"), a test on the entity's state ("Am I low on health?"), etc. If the Boolean question evaluates to true, then SUCCESS is returned as the status code; otherwise, FAILURE is returned. It is not possible for RUNNING to be returned. A condition component is represented by a box labelled with a question, such as node 3 in Fig. 1.

A *control flow* component groups a set of child components together and specifies the order in which they should be executed. The behavior of a control flow component is determined by the status codes returned by its children. Only two control flow component types are needed to express most grouping behaviors: the selector and the sequence.

A *selector* component controls flow by making a choice. When executed, a selector processes its child components from left to right and immediately returns a SUCCESS code if one of its children returns a SUCCESS code. It returns a RUNNING code if one of its children returns a RUNNING code. If a child returns FAILURE, the selector continues and processes the next component. If each of the selector's children returns a FAILURE code, then the selector returns a FAILURE code. A selector node is depicted as a circle labelled with a question mark, e.g., the top node in Fig. 2. This figure shows the "Attack" BT referenced by node 10 of the "Simple Attacker" BT shown in Fig. 1. For clarity, in Fig. 2, we also show pseudo-code for the algorithm executed by a selector node.

Selectors are used to choose one branch of a BT from a set of possible branches. For example, Fig. 2 depicts a BT that specifies a warrior entity's behavior when its goal is to engage the player in an attack. In this game, the warrior can attack at long range with a bow and arrow, attack at close range with a sword, or simply taunt the player. When this BT is executed, the selector will first attempt the "Fire arrow at player" behavior; if that behavior fails, it will then attempt the "Attack player with

```
foreach child in children do
    childStatus ←
        execute(child)
    if childStatus == RUNNING
        return RUNNING
    else if childStatus ==
            SUCCESS
        return SUCCESS
end-foreach

return FAILURE
```

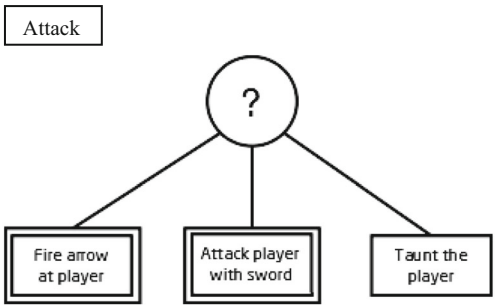

Fig. 2 Pseudo-code and example for the selector component

sword" behavior. Suppose that behavior is successful. Then processing stops and control is returned to the selector node, which returns a SUCCESS code. The "Taunt the player" action is not attempted in this case.

A *sequence* component controls flow by giving a sequence of components to execute. When executed, a sequence processes its child components from left to right and immediately returns a code if one of its children returns a FAILURE code. Similarly, it returns a RUNNING code if one of its children returns a RUNNING code. If a child returns SUCCESS, the sequence continues and processes the next component. If each of the sequence's children returns a SUCCESS code, then the sequence returns a SUCCESS code. A sequence node is depicted as a box labelled with an arrow, as shown in the top node in Fig. 3.

When the "Attack player with sword" BT in Fig. 3 is executed, the "Player in front?" condition is evaluated first. If that condition evaluates to false, FAILURE is returned to the parent sequence node and the sequence returns FAILURE. Otherwise, the "Swing sword" action is executed next. Since it is the last component in the sequence, the sequence node returns the same status code it returns.

A *decorator* affects exactly one component and modifies its processing logic; we refer to the affected component as a *wrapped component*. A decorator is depicted as an icon in the top-left corner of the shape representing a node, as shown in Fig. 4 where an inverter decorator ("!") is present in the "player visible?" node. A single component may have multiple decorators applied to it. The types of decorators of interest here are inverters, succeeders, failers, repeaters, count-based limit decorators, and time-based limit decorators. Pseudo-code and an icon for the other decorators are given in "Pseudo-Code and Icons for Decorator Components" Appendix.

An *inverter* is analogous to the NOT operator in a programming language. It negates the result of its wrapped component: SUCCESS becomes FAILURE, FAILURE becomes SUCCESS, and RUNNING is not changed. Two related component types, succeeders and failers, are provided in some BT implementations. Succeeders and failers also modify their wrapped component's return code: succeeders always return SUCCESS and failers always return FAILURE.

```
foreach child in children do
    childStatus ←
        execute(child)
    if childStatus == RUNNING
        return RUNNING
    else if childStatus ==
            FAILURE
        return FAILURE
end-foreach

return SUCCESS
```

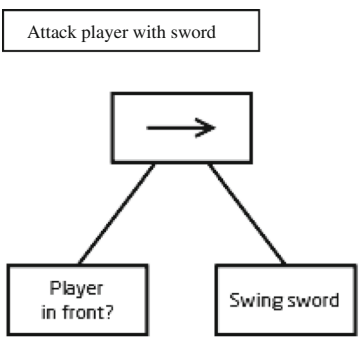

Fig. 3 Pseudo-code and example for the sequence component

```
childStatus ← execute(child)
if childStatus == SUCCESS
    return FAILURE
else if childStatus == FAILURE
    return SUCCESS

return childStatus
```

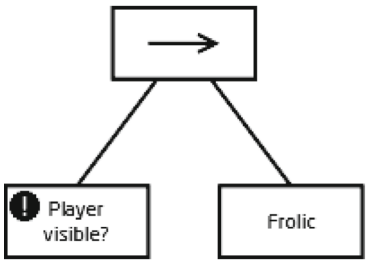

Fig. 4 Pseudo-code and example for the inverter decorator decorating a condition component

A *repeater* can also be used to execute a wrapped component behavior repeatedly via iteration. A *basic repeater* will process its wrapped component a specified number of times, regardless of whether the wrapped component returns SUCCESS or FAILURE. After the number of repetitions has been completed, it returns SUCCESS. Two other repeaters are repeat-until-SUCCESS and repeat-until-FAILURE. These decorators repeatedly process their wrapped component until they receive the corresponding return code; the decorator will then return SUCCESS. Any repeat-until decorator can optionally also specify the maximum number of times its wrapped component is repeated. If the maximum number of repetitions is reached, without its wrapped component returning SUCCESS, the repeat-until decorator returns FAILURE.

A *count-based limit decorator* imposes a limit on the number of times that its wrapped component can be repeated during the complete execution of the BT. In other words, if the decorator is processed after its wrapped component has been executed the maximum number of times, it simply returns FAILURE. Otherwise, the decorator returns the status code returned by its wrapped component.

A *timer-based limit decorator* forces a certain amount of game time to pass between executions of its wrapped component. When the decorator is processed during an AI step, its timer is incremented. The unit of measurement for game time is game-dependent. For example, a turn-based game could increment the timer by one to signify a new turn cycle, while a real-time strategy game could increment the timer by the amount of real time in milliseconds that has elapsed since the previous AI step. If, after incrementing, the appropriate amount of time has elapsed, the decorator executes its wrapped component and returns that component's result. Otherwise, it returns RUNNING.

The type of a component determines where it may appear in a BT. Conditions, actions, and references can only occur in leaf nodes in a BT. Control flow components are placed at non-leaf nodes. Decorators can be added to leaf or non-leaf nodes.

Recall our assertion that references, actions, conditions, decorators, and control flow nodes are sufficient for specifying all types of behaviors. Following (Marcotte 2017), we use the Böhm-Jacopini theorem (Bohm and Jacopini 1966) as the basis of an informal proof of this claim. The theorem states that we can compute any computable function by combining subprograms using only three specific methods: (1) executing one subprogram and then another subprogram in sequence; (2) executing one of two subprograms selected according to the value of a Boolean

expression; and (3) repeating the execution of a subprogram until a Boolean expression is true. In our case, a single action is analogous to a statement in a subprogram and a BT is analogous to a subprogram. We can execute one BT and then another by using a sequence control-flow component, which is analogous to method (1). Similarly, we can execute one BT from a set of two BTs by using a selector control-flow component, which is analogous to method (2). Finally, we can use a repeat-until-SUCCESS decorator to execute a BT until a Boolean expression is true, which is analogous to method (3). Since the three methods listed by the theorem can be represented using BT components, BTs can be used to specify any computable behavior.

3 Example of Building a Behavior Tree

Let's use a BT to model the behavior of a warrior entity that can battle against the player in an action-adventure game. The desired behavior corresponds to the following description from the first-person point of view of the warrior: "If I am being attacked, I will move away from the enemy attacking me until I'm at a safe distance. If I have a low HP, then I will find the nearest first aid kit and pick it up. If an enemy entity is nearby and not attacking, then I will attack it. To attack, I will use an arrow if the enemy is in range to the front, attack with my sword if the enemy is in front outside arrow range, or else taunt the enemy. Otherwise, I will wander around aimlessly." One possible BT for this behavior has already been shown as Fig. 1.

We can build this BT for the warrior entity using the component types described in Sect. 2. Each of the four main tasks (FIND AID, EVADE, ATTACK, and WANDER) can be encapsulated in a BT that is used as a subtree of the main BT.

The EVADE behavior corresponds to "If I am being attacked, I will move away from the enemy attacking me until I'm at a safe distance." This behavior is specified as a sequence. It evaluates a condition that checks for attacks from the enemy; if the result is true, control passes to the Evade action.

The FIND AID behavior corresponds to "If I have low HP, then I will find the nearest first aid kit and pick it up." This behavior is represented as a sequence: a condition that, if true, is followed by an action.

The ATTACK behavior corresponds to "If an enemy entity is nearby and not attacking, then I will attack." This behavior is represented by a sequence: a condition that, if true, is followed by an action. The details of the attack are given in the "Attack" BT given in Fig. 2.

The WANDER behavior corresponds to "I will wander around aimlessly." This behavior is represented by a sequence of two actions: the first picks a random location within a short range of the swordsman and the second walks toward it. The "Otherwise" aspect of "Otherwise, I will wander around aimlessly" is specified by placing the WANDER behavior as the rightmost subtree.

The above four BTs are grouped via a selector node, from left-to-right, in the order given above. As mentioned, the result is the BT depicted in Fig. 1.

4 Behavior Tree Editors

A *BT editor* is a software package for creating and maintaining BTs. Here we discuss Behavior Designer (Mosiman and Watson 2014), Behave (Johansen 2016), Behavior3 (Pereira 2015), Brainiac Designer (Brainiac Designer 2009), and Unreal Engine Behavior Trees (Epic Games).

A summary of features included in these editors is given in Table 1. As shown in the table, all of the editors provide the fundamental BT components described in Sect. 2. All manage BTs as a collection; that is, multiple BTs can be open for editing at the same time. A drag-and-drop system is available in all editors for constructing BTs using a graphical user interface. Under such a system, users are free to move individual components around the screen using the mouse. Of the editors, only Behavior3 and Unreal Engine Behavior Trees possess the ability to auto-arrange components into an aesthetically pleasing format. The auto-arrange feature can be especially helpful when dealing with large, complex BTs. Behavior Designer provides users with the ability to add comments to individual components to provide information about specific sub-behaviors. Only Brainiac Designer and Unreal Engine Behavior Trees allow users to extend the basic architecture by creating new component types. Only the Unreal Engine Behavior Trees allows external events to interrupt processing of a BT and cause conditional aborts and restarts. Behavior3 and Brainiac Designer are independent of the intended game platforms, while Behavior Designer, Behave, and Unreal Engine BTs are tied to specific development environments. The latter have debugging features specific to BTs.

As an example editor, we describe Behavior3, which is an open-source visual BT editor (Pereira 2015). Behavior3 provides capabilities corresponding to action, condition, sequence, and selector (called priority) components and inverter, succeeder, failer, basic repeater, repeat-until-failure, repeat-until-success, time-

Table 1 Summary of features in available BT editors

Feature	Behavior Designer	Behave	Behavior3	Brainiac Designer	Unreal Engine BT
Fundamental BT components (covered in Sect. 2)	✔	✔	✔	✔	✔
Manage collections of BTs	✔	✔	✔	✔	✔
Drag-and-drop interface	✔	✔	✔	✔	✔
Can auto-arrange components			✔		✔
Add comments to provide context for individual components	✔				
Extensible via user components				✔	✔
Event-driven BTs					✔
Conditional aborts and restarts					✔
Requires a specific development environment	✔	✔			✔
BT-specific debugger included	✔	✔			✔

based limit, and count-based limit decorators. BTs are imported and exported using JSON (JavaScript Object Notation) format. This editor does not depend on other tools, editors, or engines.

Behavior3 has a relatively simpler user interface, as pictured in Fig. 5. Condition components are denoted by circle nodes labelled with questions and actions are denoted by rounded rectangles labelled with a description of the action. Selectors, sequences, and other control flow components are identified via icons rather than text. Although this representation is compact, it sacrifices some readability since users must be familiar with the meaning of the icons. In the user interface, a list of BTs contained in the current project is displayed in the top-left corner. A list of components that can be added to the BT currently being viewed is located below the list of BTs.

The user interface of Behavior3 is customizable using the customization screen shown in Fig. 6. It allows users to configure component layout, placement-grid dimensions, and colors. The component layout can be set for either vertical or horizontal layout. A vertical layout displays a BT with its parent components above their children and the children in execution order from left to right. A horizontal layout has parent components placed to the left of their children and the children in execution order from top to bottom. A user can also resize the backing grid via Snap X and Snap Y to affect placement of individual components and can modify the color scheme of the interface.

5 Evaluations

In this article, we described the general idea of BTs, explained the main types of components, gave an example of a BT, and surveyed existing BT editors. We finish up by giving some conclusions, limitations, and recommendations.

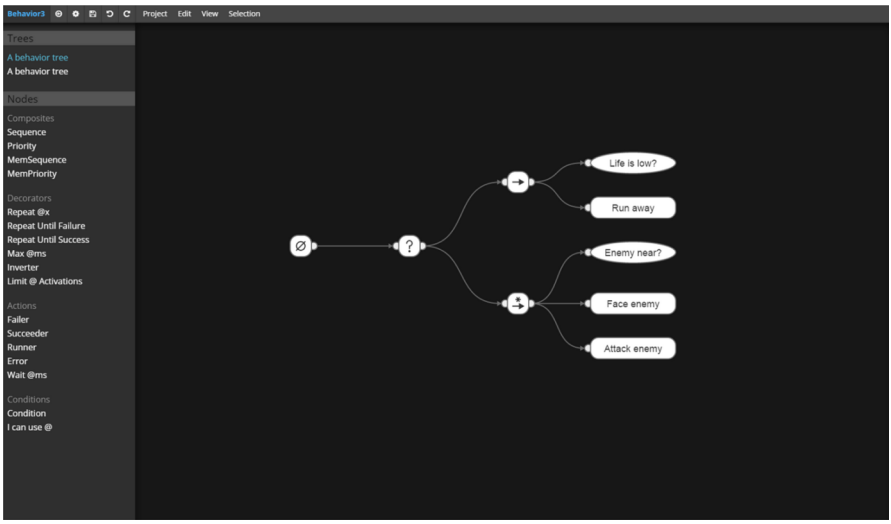

Fig. 5 The Behavior3 BT editor (screenshot from Behavior3 (Pereira, 2015))

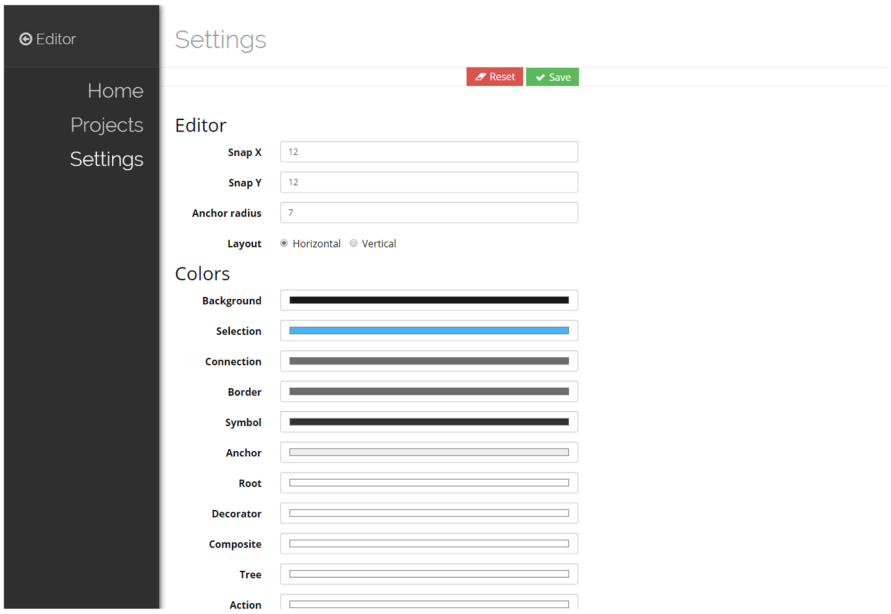

Fig. 6 The Behavior3 BT editor's configuration settings (screenshot from Behavior3 (Pereira, 2015))

5.1 Conclusions

BTs are a useful representation for specifying the behavior of non-player characters and other entities in computer games. Due to the richness of the component types provided, any desired behavior can be described by a behavior tree. Because of the hierarchical arrangement of components in a BT, adding and deleting subtrees is simple, which increases extensibility, maintainability, and scalability. BTs can be referenced by other BTs, which increases reusability. Maintenance is facilitated by using a BT editor, such as the five editors examined in this paper.

5.2 Limitations

BTs, of the simple type described in this tutorial, have several limitations. First, BTs are not able to specify all behaviors for game entities in a convenient fashion. For example, they are not well suited to modeling state-based behaviors (Millington 2009), especially when an entity needs to respond to external events, e.g. interrupting a cowardly agent's current patrol route to go into hiding when the player is detected. It is also difficult to model teamwork because doing so involves coordinating actions between behavior trees. A second limitation is that behavior trees have high memory requirements (Isla 2005), especially if each agent has a separate behavior tree. A third limitation is difficulty with storing and retrieving *history*, i.e., information related to past actions and events (Isla 2008). Difficulty in recalling past actions taken, which is equivalent to recording and storing traversals through the tree, is a problem inherent to the tree structure. Fourthly, behavior trees

are not well suited to modeling heuristic reasoning (Rasmussen 2016), especially when evaluating a heuristic function that yields numeric values. Finally, the concept of a BT has not been standardized and thus for example a BT from one editor cannot be read by another editor.

Some of these limitations can be overcome by enhancements to the concept of behavior trees, such as by using globally available blackboards or agent-specific contexts to hold state and history information, by using a tactics manager (Pillosu 2009) or parameterized behavior trees(Shoulson, Garcia, Jones, Mead and Badler 2011) to coordinate a team, by using query enabled behavior trees (Flórez-Puga, Gómez-Martín, Gómez-Martín, Díaz-Agudo and González-Calero 2009) to build BTs at run time and thus reduce space requirements, and by adding utility selectors (Merrill 2013) (Marcotte 2017) to compute heuristic functions.

5.3 Recommendations

We recommend that BTs be considered for specifying and maintaining the behavior of a non-player character or a similar game entity. The following factors increase the appropriateness of behavior trees: The presence of game designers who are not programmers, highly complex conditions governing the behavior, and the need for NPCs with aspects of behavior in common. We recommend considering the features that are desired in an editor and then consulting Table 1 before choosing a BT editor. For cases where a computer game is being developed for only one platform, such as Unity or Unreal Engine, we recommend using a platform-specific editor due to the convenience of debugging, and otherwise we recommend choosing a platform-independent editor. For those who are implementing BT-based software, we recommend careful consideration of known limitations of simple formulations of BTs. In some cases, it may be appropriate to think of a BT primarily as a way of representing information about hierarchical decision making for an entity while representing other relevant data about the entity in other forms. Generally, for the field, we recommend that a standardization effort be considered, with a standardized, extensible format for encoding behavior trees in files as a possible first step.

Appendix: Pseudo-Code and Icons for Decorator Components

See Figs. 7, 8, 9, 10, 11, 12 and 13.

```
childStatus ← execute(child)
return SUCCESS
```

Fig. 7 The succeeder decorator

```
childStatus ← execute(child)

return FAILURE
```

Fig. 8 The failer decorator

```
while i < MAX_TIMES
    childStatus ← execute(child)
    if childStatus != SUCCESS and childStatus != FAILURE
        return childStatus

    i++
end-while

return SUCCESS
```

Fig. 9 The basic repeater decorator

```
repeat
    if i == MAX_TIMES
        return FAILURE

    childStatus ← execute(child)
    if childStatus != SUCCESS and childStatus != FAILURE
        return childStatus

    i++
until childStatus == SUCCESS

return SUCCESS
```

Fig. 10 The repeat-until-success decorator

```
repeat
    if i == MAX_TIMES
        return FAILURE

    childStatus ← execute(child)
    if childStatus != SUCCESS and childStatus != FAILURE
        return childStatus

    i++
until childStatus == FAILURE

return SUCCESS
```

Fig. 11 The repeat-until-failure decorator

```
if totalCalls < MAX_TIMES
    childStatus ← execute(child)
    totalCalls++

    return childStatus
end-if

return FAILURE
```

Fig. 12 The count-based limit decorator

```
totalTimeElapsed += timeElapsedSincePreviousFrame
if totalTimeElapsed >= TIME_INTERVAL
    childStatus ← execute(child)
    totalTimeElapsed -= TIME_INTERVAL

    return childStatus
end-if

return RUNNING
```

Fig. 13 The timer-based limit decorator

References

Bates, J. (1994). The role of emotion in believable agents. *Communications of the ACM, 37*(7), 122–125.

Bohm, C., & Jacopini, G. (1966). Flow diagrams, turing machines and languages with only two formation rules. *Communications of the ACM, 9*(5), 366–371.

Brainiac Designer. (2009, November 23). Retrieved June 11, 2016, from CodePlex: https://brainiac.codeplex.com/.

Bungie Studios. (2004). *Halo 2, video game*, Xbox, Microsoft.

Champandard, A. J. (2008). Getting started with decision making and control systems. In S. Rabin (Ed.), *AI Game programming wisdom* (Vol. 4, pp. 257–264). Boston: Charles River Media.

Colledanchise, M., & Ogren, P. (2014). How behavior trees modularize robustness and safety in hybrid systems. *IEEE/RSJ International Conference on Intelligent Robots and Systems* (pp. 1482–1488). Chicago: IEEE.

Epic Games. (n.d.). Behavior Trees. Retrieved June 18, 2016, from Unreal Engine Documentation: https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/index.html.

Flórez-Puga, G., Gómez-Martín, M. A., Gómez-Martín, P. P., Díaz-Agudo, B., & González-Calero, P. A. (2009). Query-enabled behavior trees. *IEEE Transactions on Computational Intelligence and AI in Games, 1*(4), 298–308.

Humphreys, T. (2014). Exploring HTN planners through example. In S. Rabin (Ed.), *Game AI pro: Collected wisdom of game AI professionals* (pp. 149–167). Boca Raton: CRC Press.

Isla, D. (2005). Handling complexity in the Halo 2 AI. Retrieved January 15, 2014, from http://www.gamasutra.com/view/feature/130663/gdc_2005_proceeding_handling_.php.

Isla, D. (2008). Building a better battle: The Halo 3 AI objectives system. Retrieved February 8, 2017, from https://web.cs.wpi.edu/∼rich/courses/imgd4000-d09/lectures/halo3.pdf.

Johansen, E. (2016). The Behave project. Retrieved June 15, 2016, from AngryAnt: http://angryant.com/behave/.

Johansson, A., & Dell'Acqua, P. (2012). Emotional behavior trees. *IEEE Conference on Computational Intelligence and Games* (pp. 355-362). Granada, Spain: IEEE Xplore.

Khoo, A. (2006). An Introduction to behaviour-based systems for games. In S. Rabin (Ed.), *AI Game programming wisdom* (Vol. 3, pp. 351–364). Boston: Charles River Media.

Lim, C.-U., Baumgarten, R., & Colton, S. (2010). Evolving behaviour trees for the commercial game DEFCON. *Applications of Evolutionary Computation* (pp. 110–110).

Llopis, N. (2010). Game architecture. In S. Rabin (Ed.), *Introduction to Game Development* (pp. 235–270). Boston: Charles River Media.

Marcotte, R. (2017). *Modelling artificial intelligence in games using MindSet behavior trees*. Regina: Department of Computer Science, University of Regina.

Marzinotto, A., Colledanchise, M., Smith, C., & Ogren, P. (2014). Towards a unified behavior trees framework for robot control. *IEEE International Conference on Robotics and Automation* (pp. 5420–5427). Hong Kong: IEEE.

Mateas, M. (2002). Interactive drama, art, and artificial intelligence. Technical report CMU-CS-02-206, Carnegie Mellon University, School of Computer Science, Pittsburgh.

Mateas, M., & Stern, A. (2005). Structuring Content in the Facade Interactive Drama Architecture. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2005).* Marina del Rey, CA.

Merrill, B. (2013). Building utility decisions into your existing behavior tree. In S. Rabin (Ed.), *Game AI Pro* (pp. 127–136). Boston: Charles River Media.

Millington, I. (2009). *Artificial intelligence for games* (2nd ed.). San Francisco: Morgan Kaufmann.

Mosiman, J., & Watson, S. (2014, February 10). Behavior Designer—behavior trees for everyone | Unity Community. Retrieved June 17, 2016, from Unity3D Forums: http://forum.unity3d.com/threads/behavior-designer-behavior-trees-for-everyone.227497/.

Pereira, R. (2015, June 24). GitHub—behavior3/behavior3editor. Retrieved June 14, 2016, from GitHub https://github.com/behavior3/behavior3editor.

Pillosu, R. (2009). Coordinating agents with behavior trees. Paris Game AI Conference '09. Paris, France. Retrieved from https://aigamedev.com/premium/presentations/coordination-behavior-trees/.

Pousman, Z., Mateas, M., & Wolff, M. (2009, July 21). ABL (A Behavior Language), Tutorial v01. Retrieved 05 28, 2017, from http://www.cc.gatech.edu/~simpkins/research/afabl/ABL_Tutorial.pdf.

Rabin, S. (2010). Artificial intelligence: Agents, architecture, and techniques. In S. Rabin (Ed.), *Introduction to game development* (2nd ed., pp. 521–557). Boston: Charles River Media.

Rasmussen, J. (2016, April 27). Are behavior trees a thing of the past? Retrieved August 2, 2016, from Gamasutra: http://www.gamasutra.com/blogs/JakobRasmussen/20160427/271188/Are_Behavior_Trees_a_Thing_of_the_Past.php.

Schenk, K., Lari, A., Church, M., Graves, E., Duncan, J., Miller, R., & Schaeffer, J. (2013). ScriptEase II: Platform independent story creation using high-level patterns. *Proceedings of the Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)* (pp. 170–176). Boston: AIIDE.

Shoulson, A., Garcia, F., Jones, M., Mead, R., & Badler, N. (2011). Parameterizing behavior trees. *Motion in Games, 2011,* 144–155.

Weiss, M. A. (2013). *Data structures and algorithm analysis in C ++* (4th ed.). Upper Saddle River: Pearson.