Adapting AI Behaviors To Players in Driver San Francisco
Hinted-Execution Behavior Trees

Sergio Ocio
Ubisoft Entertainment
sergio.ocio@ubisoft.com

Abstract

The creative nature of games makes trying new ideas desirable, but these changes are sometimes very risky. We need to find ways to minimize risks while we build innovative experiences. Driver San Francisco did this by using Hinted-execution Behavior Trees; this technique allows developers to modify existing AI behaviors dynamically with very low risk, and was used to adapt Driver's getaway AI to players' skills.

AI in Driver San Francisco

Driver San Francisco is the latest installment of the Driver franchise, developed by Reflections, a Ubisoft Studio, and released in September 2011. In Driver, players take the role of detective John Tanner in his race to take down criminal lord Charles Jericho. The game was critically praised for its innovative *shift* mechanic (Grisham 2011), which allowed players to instantly jump to any other vehicle at any part of the city.

A small team of engineers developed the AI in Driver. Our mandate was to create artificial drivers that can navigate through a busy city that are as skilled as the best human player. All the technology was brand new and tailor-made to enhance Driver's experience.

Path generation

The core of the system was a three-tiered path finding solution that generated "safe" paths, so vehicles could drive without hitting other cars or static obstacles during two seconds. Before we ran out of segments in our path, we generated a new one, which was then concatenated to the remaining portion of the old path; that way, vehicles drove using a continuous and seamless path.

The process started by generating a *route*, or list of roads, which were used by vehicles to get from their current positions to their destinations. These destinations could be dynamic, for example if we were chasing another car, or static, in the case where we were trying to get to another position in the city.

After a route was established, we zoomed in into a portion of the route surrounding the vehicle's current position. The AI extracted information about the environment in that area (such as world limits and static obstacles) as well as all the dynamic obstacles around.

Once all the information was gathered, a *mid-level path* was generated. These coarse paths were built as a guide for vehicles to avoid groups of obstacles, and also to get an approximate idea of the speed limitations we would encounter at future corners.

A low-level pathfinder later optimized these mid-level paths. This optimizer generated the final smoothed path, which avoided individual obstacles and was built taking into account the actual capabilities of the car.

Finally, a path-following module received the final path and drove the vehicle using the same handling model that was used by player-controlled cars.

Behaviors

The game offered a variety of missions that required creating drivers with different personalities and goals, such as reckless racers, cops or getaway drivers. Each AI driver was assigned a goal -built using *Behavior Trees* (BTs)- which was in charge of generating and updating paths to control their cars.

Cop and getaway missions were key in Driver and thus some other systems, such as group behaviors or special maneuvers were also implemented. Getaway driver AI was especially notable; these drivers were able to adapt their behaviors based on the abilities of the player, which was used both to create a more accessible experience and to provide a fun challenge to hardcore players.

Driver used *Hinted-execution Behavior Trees* to modify its AI behaviors dynamically. In this paper, we will explain in detail what these trees are and how they were used to build the game.

Behavior Trees

Behavior Trees have become increasingly popular in the game industry in the last ten years (Isla 2005) and they have been used in many titles, such as Halo or Driver.

The key characteristic of these trees is that they are a data-driven solution. Behaviors are built by instancing and linking different types of building blocks (Champandard 2008); these have different responsibilities, from controlling the execution flow (*metanodes* and *filters*) to performing actions or querying the environment for information (*conditions*). When a node is updated, it always returns a status value -*success*, *failure* or *running*-. This value is passed to the parent node, which then decides what to do with this information. To understand this better, let us study how a simple tree, such as the one shown in figure 1, would work.

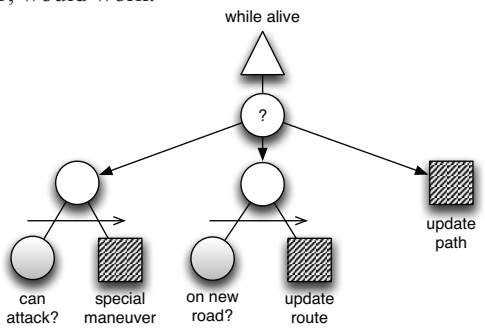

Figure 1. A simplified version of one of the BTs used in Driver

A traditional Behavior Tree is always run starting at its root. In this case, our root is a *filter*. These nodes work as decorators, i.e. they modify the behavior of another node. This particular filter is a *conditional loop*; if its child succeeds, the node will restart it, as long as the condition is met. The condition is checked every step, so as soon as it fails, the whole tree will be stopped. This means our tree will run for as long as our AI is alive.

The filter is decorating a *selector*. Selectors are a type of *metanode*, which are in charge of decision-making. BTs are considered *reactive planners*: AIs controlled by these trees have a full plan to accomplish their goals and they can backtrack and choose the most appropriate action, based on their perceptions about the world, allowing them to react to changes (Georgeff & Lansky 1987). Selector are key in this process. A selector tries to run its children, one at a time, and succeeds as soon as one of its children does; if a branch fails, the next one is tested, and the process continues until there is a success, or it runs out of children. The latter will cause the selector to fail. Children priorities can be calculated in many different ways, but we have chosen to use the order in which they are represented in the tree, left to right. In our example, the branch leading to "*special maneuver*" has the highest priority, and "*update path*", the lowest.

The first of the branches in our selector starts with a *sequence* metanode; similar to selectors, these nodes run their children one at a time, but in this case all the nodes are executed until we have run out of them, which indicates a success, or any one has failed, which makes the sequence fail and bail out. In the example, the node will perform a one-off check, "*can attack*" and if it can, the AI will execute a "*special maneuver*". If we were not allowed to attack, the failure would be passed to the parent selector, which will then choose the second branch and so on.

Getaway AI

Making players able to feel the rush of taking part in a high-speed cop chase was a very important part of Driver, so we needed to offer a very polished experience.

Every car in the game needs a route. Early in development, getaways ran on predefined routes. This could look good in some missions, but did not show intelligence and could be boring after a few playthroughs. Additionally, Driver allowed players to start "chase" and "getaway" situations at any time and location in the city, so we could not always rely on these routes. We then decided to move to a dynamic route generator for getaways.

Our path finding used information from the vehicle's route so it knows which roads to use. Paths are relatively short, only valid for a couple of seconds; with a maximum speed of 70 m/s, that meant our cars could traverse up to 140 meters in that time. This led us to decide we only needed routes to be approximately four roads long.

Building a new dynamic route could be as easy as, starting from the vehicle's current road, choosing a new road at the next junction and repeating the process until we have four connected roads. We could generate any route, from those that preferred to always turn right, to random routes, and we had in fact been using that method to have AI cars just wander around the city. However, now we were modeling aggressive getaway drivers, so we could not just choose any road; we had to make an intelligent choice. To do this, we decided to reuse our BT system and implemented the new route generator as a new tree, shown in figure 2.

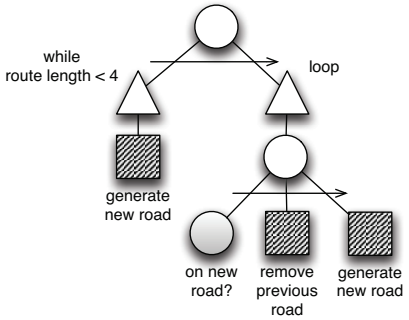

Figure 2. Getaway route generation was controlled using a BT

The behavior had two main branches: the first one was in charge of generating a brand new route, whereas the second branch kept updating the existing route properly every time a car changed roads.

The most interesting node in the tree is "*generate new road*". This was actually a sub-tree, and it was used to decide which road was our best option at a junction. It worked using a set of components, each of which was able to select a road based on some specific parameters and preconditions, and a selector that chose the most appropriate one. We used four of these components, which by themselves were quite basic; the way they were combined, however, produced very interesting getaway routes. As figure 3 shows, the components we used were "*zig-zag*", which alternate right and left turns, "*maintain type*", which tried to keep the car on a special type of road if it was on one (e.g. dirt roads, alleyways, highways...), "*to special type*", which tried to choose a road that will lead us to another road of a specific type as soon as possible, and "*straight ahead*".

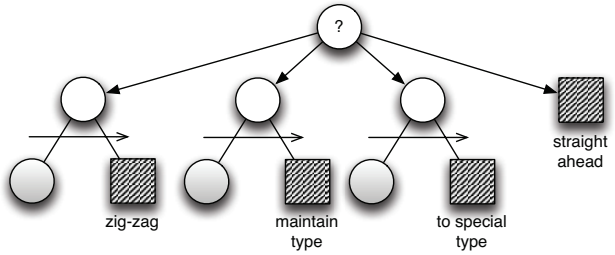

Figure 3. Getaways added new nodes to their routes using four different components

Preconditions in the tree determined whether the component could be used. For example, they allow us to determine how long a component could be run for or stop components from being chosen too frequently. These were tweaked until we had a generator that created interesting and fun routes for our getaway drivers.

Adaptation

With our new system ready, we started thinking about how to improve it. One of the ideas that came from design was to make the game more accessible to new players while keeping it fun for hardcore players. So, basically, we wanted to adapt the game to the abilities of the player.

The gameplay team had developed a *player analysis* system, which measured how well a player was doing in the game. This produced a floating point value in the range [0..1]; the higher the value, the better the player. The score came from a calculation that involved different factors, such as whether the player was crashing into other vehicles or obstacles, the speed he or she was traveling at, etc.

We wanted to adapt our getaway routes to this continuous value, which required a dynamic modification of the generator's behavior. At the same time, we did not want to change the generator itself, as it was working very well. These constraints made us choose *Hinted-execution Behavior Trees* (Ocio 2010).

Hinted-execution Behavior Trees

Today's AAA games are massive software engineering projects, built by hundreds of people. Teams are multi-disciplinary, and professionals with very different backgrounds collaborate in the process. This makes games both expensive and risky enterprises.

Due to this time and cost/risk constraints, we will most likely not be able to build many new systems from the ground up, but we will need to **extend** what we already have and improve it in such a way that we can generate brand new situations without changing the foundations of our engine.

Driver San Francisco's AI was based on Behavior Trees, so when we decided to implement new features, we needed to maintain the system that was already working and had been tested, and add a high-level layer to avoid breaking it.

Hinted-execution Behavior Trees (HeBTs) are an extension to regular Behavior Trees that give developers an extra layer of control over their trees and allow them to create and test new features in a plug-in fashion.

Even though some other data-driven approaches are available, HeBTs were the best suited solution for our game.

Hints

In a HeBT system, agents will maintain their autonomy, but will now take suggestions into consideration. We have called those suggestions *hints*.

A hint is a piece of information that lets the AI know what it **should be doing**, but it is up to the system to decide whether it is possible to accept the suggestion or not. This allows the AI to be able to react adequately to the state of the environment. For example, we could be telling the AI to try and "*take cover*" when there is none available, and the agent should be intelligent enough to ignore our hint and behave as expected (e.g. attack an enemy).

In a Behavior Tree, the execution flow is controlled by some of its nodes, specifically metanodes and conditions. So, if we want to modify a behavior, we need to change these nodes to accept our hints.

HeBT selectors

Selectors are the nodes in charge of making decisions in the tree, i.e. choosing which branch will be explored first. Each child is given a priority, and the node makes the selection based on this value that, in a basic selector, comes from the order in which the branches were added to

the tree. The priorities in the selector shown in figure 4 would make it test branches in the following order: "*take cover*", "*attack*", "*inspect*" then "*idle*".

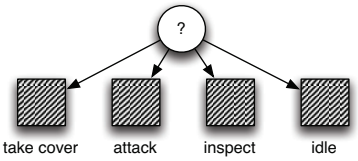

Figure 4. In a basic selector, priorities come from the ordering of their branches, left to right

Now, let us say our designers want to test something new: they would like some NPCs to be more aggressive under certain circumstances, and avoid taking cover if they can attack. Normally, this would require a change in our tree, which is always risky. However, if our selector could accept *hints*, we could very easily obtain the desired behavior and, even better, we could go back to the old, more tested one, dynamically, just by stopping the sending of the new hint to our selector. Figure 5 shows how the selector would react to an "*attack*" hint.

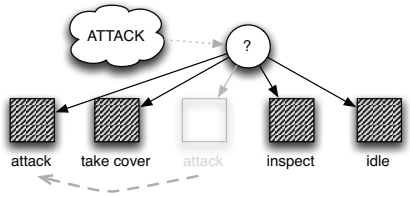

Figure 5. A selector will reorder its branches after receiving a hint

The new priorities will make the AI try to attack first. The NPC will even retain part of its autonomy and intelligence, as, if the attack fails, it will still be able to decide how to deal with the situation (e.g. taking cover). This is the key benefit of HeBTs: they provide a high-level control layer over behaviors, but free their users from the burden of micromanaging the internals of the AI.

But we may want to prevent actions. In that case, our selector would lower the priority of the branch; it will not be removed, but will be less likely to be selected. Figure 6 shows the process that takes place internally in the selector.

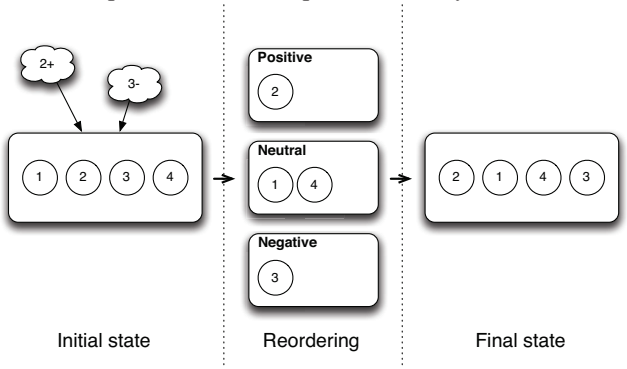

Figure 6. Hints can be either positive or negative, causing priorities to be increased or lowered

Hint conditions

Recalculating priorities in our selectors is not always enough. Let us explain this with an example, as shown in figure 7.

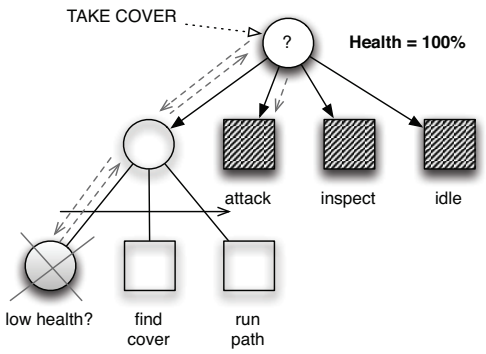

Figure 7. Sometimes, sending a hint to a selector is not enough to modify a behavior

In this case, we are sending a "*take cover*" hint to the selector we have been using in the previous examples. The branch already had the highest priority, so no reordering is required in the selector: our behavior has not changed, and our hint is basically ignored. How can we fix this?

If we take a look at the branch, we find a sequence with a "*low health?"* precondition. Because sequences bail out as soon as one of their children fails, we can see this particular one will always fail unless our NPC is low on health, which is not what we want.

We can solve this by adding a new type of condition, *hint conditions*, which allow us to listen for hint states. This way, we can make a small change in our old tree and enable the branch to be chosen, even when our agent's health is full, as shown in figure 8.

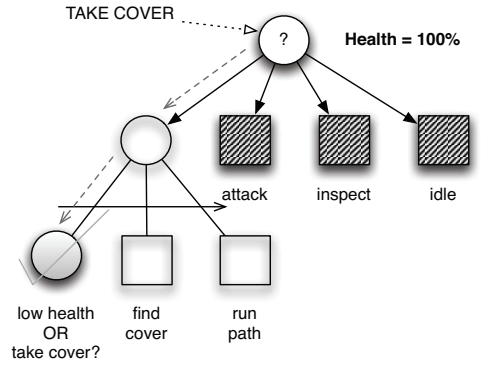

Figure 8. Hint conditions allow our BTs to use hints properly

The new condition produces the desired behavior, allowing our AI to take cover at any point.

Tree hierarchies

HeBTs are more powerful than traditional approaches to modifying behaviors (e.g. personality traits) because they allow us to generate and test new logic without changing a single line of code. So far, we have studied how a HeBT would react upon receiving a new hint. We now need to describe how we will send those hints to our trees.

HeBTs work as a hierarchy, where higher-level trees send hints to those immediately below them (figure 9).

Figure 9. Hinted-execution Behavior Trees work as a hierarchy of trees, where hints flow downwards

The lowest-level tree in this hierarchy is the one our "regular AI" uses normally, and which we want to have some control over. This tree is created and maintained by AI engineers as it can be very complex and requires a high degree of knowledge about the system.

It is also engineers who will decide which parts of the tree are exposed to higher levels. They do so by generating a list of hints the BT can accept and making some minor changes to the tree itself. The changes required are just the addition of some hint conditions to avoid the problem described in the previous subsection; these hint conditions will not affect the normal execution of a tree if no hints are being used, so the risk level is very low.

A higher-level tree is normally much simpler than the base one. These trees do not have access to the full library of building blocks, mainly because they are not allowed to use *actions*. Instead, high-level trees replace actions with *hint nodes*; they are very similar to actions, but they send a specific hint, either positive or negative, to the tree below them in the hierarchy. The process can be repeated for as many levels as we consider necessary.

Debugging

By using a good graphical representation for the different types of node, a BT can be easily read and understood. Developers can take advantage of this and build tree-authoring tools to make it easier to create or modify behaviors.

When it comes to debugging a BT, things can be quite painful if we lack good tools: several branches can run concurrently, and new nodes are started or stopped continuously, so it is hard to keep track of what is happening at any given time.

However, the nature of BTs and the fact that they can be drawn is an advantage when we try to build a debugging tool. Such a tool would be able to communicate with the game in real-time and gather information about the state of the trees; nodes could be plotted in different colors to show those that are being run, branches could be expanded or collapsed dynamically to show or hide information, breakpoints could be added to pause the execution of the game at relevant points and we could even store information to analyze the behavior offline.

Debugging HeBTs is no different than that. In this case, our hierarchy of trees is run and high-level trees will affect lower-level ones. A good tool could allow us to apply those modifications to our trees on the fly, or show them side-by-side to the original tree, so we can see how the hints are interacting with our base.

A full HeBT system needs to be complemented by a good set of tools to allow both programmers and designers to make the most of the technique.

HeBTs in Driver San Francisco

Applying the ideas explained in the previous section, adapting Driver's getaway routes to players required creating some high-level trees to control our route generator BT (figure 2). Working closely with game and level designers we decided to create a few high-level tree (we called them "presets") to control the skill of the AI.

Internally, the system matched presets to ranges of the value output by the player analysis module. The presets used in the game were "easy" [0.0-0.2), "easy-medium" [0.2, 0.4), "medium" [0.4, 0.6), "medium-hard" [0.6, 0.8) and "hard" [0.8, 1.0].

The base tree was modified to expose a hint for each component ("*zig-zag*", "*maintain type*", "*to special type*" and "*straight ahead*"), and preconditions were updated with hint conditions as required. These changes had minimum risk, as they were not going to modify the behavior without receiving hints, and also were easy to make, as they were only data-driven.

Presets could be plugged-in or removed dynamically, modifying routes as players improved their skills.

Easy-medium preset

For players who were not too skilled, but were not doing very bad either, we decided to generate routes that preferred straight exits at junctions, but occasionally chose to "zig-zag", to make the routes more interesting.

This tree made use of a third type of metanode we have not covered yet: *parallels*. These blocks run all their children concurrently (or, at least, in the same frame); if any of the branches ends, the node will bail out, returning the status of that child. In our case, we used parallels to send more than one hint at once to the base tree. The final preset is shown in figure 10.

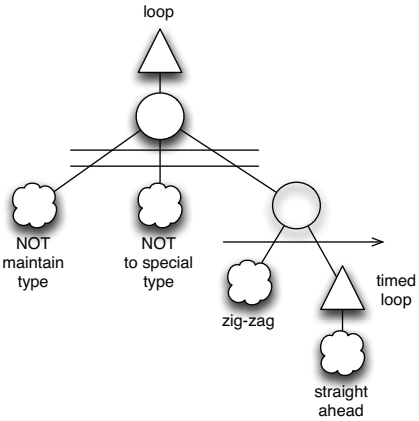

Figure 10. High-level tree used as the easy-medium preset

The tree sent negative "*maintain type*" and "*to special type*" hints. The sequence in the last branch of the parallel sent a single "*zig-zag*" hint followed by "*straight ahead*" ones that were repeated over a period of time. After plugging in this tree, our base route generation was modified as shown in figure 11.

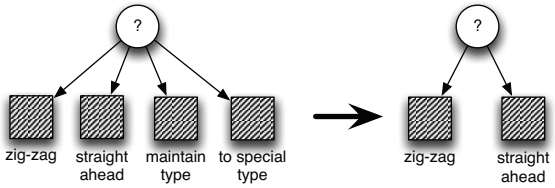

Figure 11. By using a high-level tree, the route generator changes its behavior dynamically

As "straight ahead" nodes never fail, our original tree is simplified by the easy-medium preset in such a way that it loses two of its components. The resulting tree produces the routes we were looking for.

Anti-shift preset

Driver allowed players to "shift" into any vehicle in the game at any time. This opened new options when facing a getaway, with strategies such as "shifting" into a vehicle in the oncoming traffic and hitting the getaway car head-on to try and stop it.

A good way to avoid these attacks could have been making the getaway vehicle try and stay on dirt roads and alleyways, as there was no traffic on them, which would have made it impossible for players to exploit the usage of "shift" in some situations. An anti-shift preset would have looked like the one shown in figure 12.

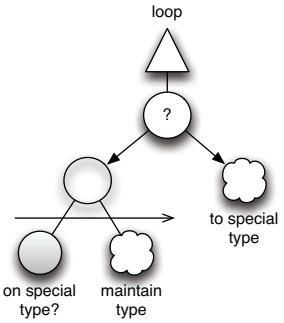

Figure 12. A preset that tries to make the car stay on special roads

Even though this preset was not included in the game, it is a good example of how easy it would have been to keep extending the base functionality of our route generator.

Conclusions

Driver San Francisco featured a robust AI, which was able to deal with a high number of dynamic obstacles and with very different road and junction configurations. The game was the first to use HeBTs, which allowed us to provide a fun and enjoyable experience for a wider range of players, no matter their skill levels, and also allowed us to:

- **Prototype** our adaptation ideas very quickly.
- **Reduce risks**, as we did not have to change any major part of the code. Most of the changes were made on data.
- Vary the behavior of the getaway AI drivers **dynamically** based on the evolving skills of players.

HeBTs where designed to encourage low-risk prototyping, allowing teams to try new ideas at late stages of development. With a few changes, traditional BTs can be tweaked and modified in a plug-in fashion: changes are not made in stone and can be applied or reverted by simply adding or removing high-level trees.

References

Champandard, A. J. 2008. *Getting started with decision making and control systems*. AI Game Programming Wisdom 4. Boston, Massachusetts: Course Technology. 257-263.

Georgeff, M. P., & Lansky, A. L. 1987. Reactive Reasoning and Planning. *In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI'87)*. American Association for Artificial Intelligence. Menlo Park, California. 677-682.

Grisham, R. 2011. Driver San Francisco Review. Gamesradar. Retriever from http://www.gamesradar.com/driver-san-francisco-review/ on April 2012.

Isla, D. 2005. Handling complexity in the Halo 2 AI. *In Proceedings of the GDC 2005.* Gamasutra.

Ocio, S. 2010. A dynamic decision-making model for videogame AI systems, adapted to players. Ph.D. diss., Department of Computer Science, University of Oviedo, Spain.