4

Behavior Selection Algorithms

An Overview

Michael Dawe, Steve Gargolinski, Luke Dicken, Troy Humphreys, and Dave Mark

4.1 Introduction

Writing artificial intelligence systems for games has become increasingly complicated as console gamers demand more from their purchases. At the same time, smaller games for mobile platforms have burst onto the scene, making it important for an AI programmer to know how to get the best behavior out of a short frame time.

Even on complicated games running on powerful machines, NPCs can range from simple animals the player might run past or hunt to full-fledged companion characters that need to stand up to hours of player interaction. While each of these example AIs may follow the Sense–Think–Act cycle, the "think" part of that cycle is ill-defined. There are a variety of algorithms to choose from, and each is appropriate for different uses. What might be the best choice to implement a human character on the latest consoles might not be suitable for creating an adversarial player for a web-based board game.

This article will present some of the most popular and proven decision-making algorithms in the industry, providing an overview of these choices and showing when each might be the best selection to use. While it is not a comprehensive resource, hopefully it will prove a good introduction to the variety of algorithmic choices available to the AI programmer.

4.2 Finite-State Machines

Finite-state machines (FSMs) are the most common behavioral modeling algorithm used in game AI programming today. FSMs are conceptually simple and quick to code, resulting in a powerful and flexible AI structure with little overhead. They are intuitive and easy to visualize, which facilitates communication with less-technical team members. Every game AI programmer should be comfortable working with FSMs and be aware of their strengths and weaknesses.

An FSM breaks down an NPC's overall AI into smaller, discrete pieces known as *states*. Each state represents a specific behavior or internal configuration, and only one state is considered "active" at a time. States are connected by *transitions*, directed links responsible for switching to a new active state whenever certain conditions are met.

One compelling feature of FSMs is that they are easy to sketch out and visualize. A rounded box represents each state, and an arrow connecting two boxes signifies a transition between states. The labels on the transition arrows are the conditions necessary for that transition to fire. The solid circle indicates the initial state, the state to be entered when the FSM is first run. As an example, suppose we are designing an FSM for an NPC to guard a castle, as in Figure 4.1.

Our guard NPC starts out in the *Patrol* state, where he follows his route and keeps an eye on his part of the castle. If he hears a noise, then he leaves *Patrol* and moves to *Investigate* the noise for a bit before returning to *Patrol*. If at any point he sees an enemy, he will move into *Attack* to confront the threat. While attacking, if his health drops too low, he'll *Flee* to hopefully live another day. If he defeats the enemy, he'll return to *Patrol*.

While there are many possible FSM implementations, it is helpful to look at an example implementation of the algorithm. First is the `FSMState` class, which each of our concrete states (*Attack*, *Patrol*, etc.) will extend:

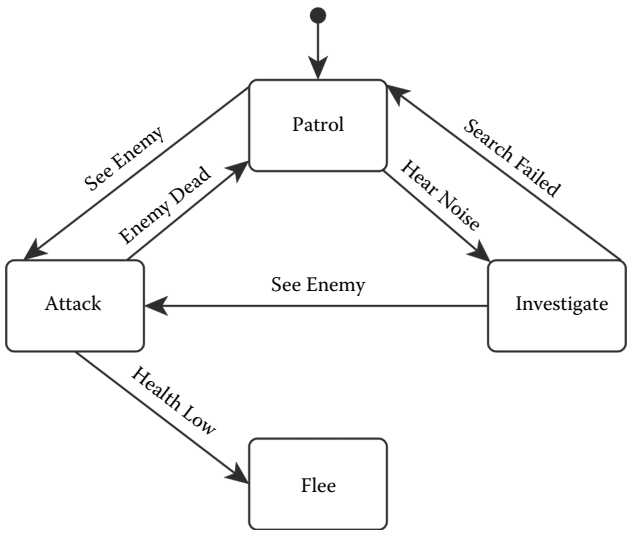

Figure 4.1

This FSM diagram represents the behavior of a guard NPC.

```
class FSMState
{
    virtual void onEnter();
    virtual void onUpdate();
    virtual void onExit();
    list<FSMTransition> transitions;
};
```

Each `FSMState` has the opportunity to execute logic at three different times: when the state is entered, when it is exited, and on each tick when the state is active and no transitions are firing. Each state is also responsible for storing a list of `FSMTransition` objects, which represent all potential transitions out of that state.

```
class FSMTransition
{
    virtual bool isValid();
    virtual FSMState* getNextState();
    virtual void onTransition();
}
```

Each transition in our graph extends from `FSMTransition`. The `isValid()` function evaluates to true when this transition's conditions are met, and `getNextState()` returns which state to transition to when valid. The `onTransition()` function is an opportunity to execute any necessary behavioral logic when a transition fires, similar to `onEnter()` in `FSMState`.

Finally, the `FiniteStateMachine` class:

```
class FiniteStateMachine
{
    void update();
    list<FSMState> states;
    FSMState* initialState;
    FSMState* activeState;
}
```

The `FiniteStateMachine` class contains a list of all states in our FSM, as well as the initial state and the current active state. It also contains the central `update()` function, which is called each tick and is responsible for running our behavioral algorithm as follows:

- Call `isValid()` on each transition in `activeState.transtitions` until `isValid()` returns true or there are no more transitions.
- If a valid transition is found, then:
  - Call `activeState.onExit()`
  - Set `activeState` to `validTransition.getNextState()`
  - Call `activeState.onEnter()`
- If a valid transition is not found, then call `activeState.onUpdate()`

With this structure in place, it's a matter of setting up transitions and filling out the `onEnter()`, `onUpdate()`, `onExit()`, and `onTransition()` functions to produce the desired AI behavior. These specific implementations are entirely design dependent. For example, say our Attack state triggers some dialogue, "There he is, get him!" in

`onEnter()` and uses `onUpdate()` to periodically choose tactical positions, move to cover, fire on the enemy, and so on. The transition between *Attack* and *Patrol* can trigger some additional dialogue: "Threat eliminated!" in `onTransition()`.

Before starting to code your FSM, it can be helpful to sketch a few diagrams like the one in Figure 4.1 to help define the logic of the behaviors and how they interconnect. Start writing the code once the different states and transitions are understood. FSMs are flexible and powerful, but they only work as well as the thought that goes into developing the underlying logic.

4.3 Hierarchical Finite-State Machines

FSMs are a useful tool, but they do have weaknesses. Adding the second, third, or fourth state to an NPC's FSM is usually structurally trivial, as all that's needed is to hook up transitions to the few existing required states. However, if you're nearing the end of development and your FSM is already complicated with 10, 20, or 30 existing states, then fitting your new state into the existing structure can be extremely difficult and error-prone.

There are also some common patterns that FSMs are not well-equipped to handle, such as situational behavior reuse. To show an example of this, Figure 4.2 shows a night watchman NPC responsible for guarding a safe in a building.

This NPC will simply patrol between the front door and the safe forever. Suppose a new state called *Conversation* is to be added that allows our night watchman to respond to a cell phone call, pause to have a brief conversation, and return to his patrol. If the watchman is in *Patrol to Door* when the call comes in, then we want him to resume patrolling to the door when the conversation is complete. Likewise, if he is in *Patrol to Safe* when the phone rings, he should return to *Patrol to Safe* when transitioning out of *Conversation*.

Since we need to know which state to transition back to after the call, we're forced to create a new *Conversation* state each time we want to reuse the behavior, as shown in Figure 4.3.

In this simple example we require two *Conversation* behaviors to achieve the desired result, and in a more complicated FSM we might require many more. Adding additional states in this manner every time we want to reuse a behavior is not ideal or elegant. It leads to an explosion of states and graph complexity, making the existing FSM harder to understand and new states ever more difficult and error-prone to add.

Thankfully, there is a technique that will alleviate some of these structural issues: the Hierarchical Finite-State Machine (HFSM). In an HFSM, each individual state can be an entire state machine itself. This technique effectively separates one state machine into multiple state machines arranged in a hierarchy.

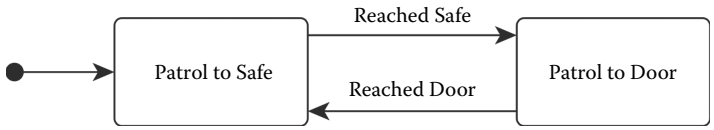

Figure 4.2

This FSM diagram represents the behavior of a night watchman NPC.

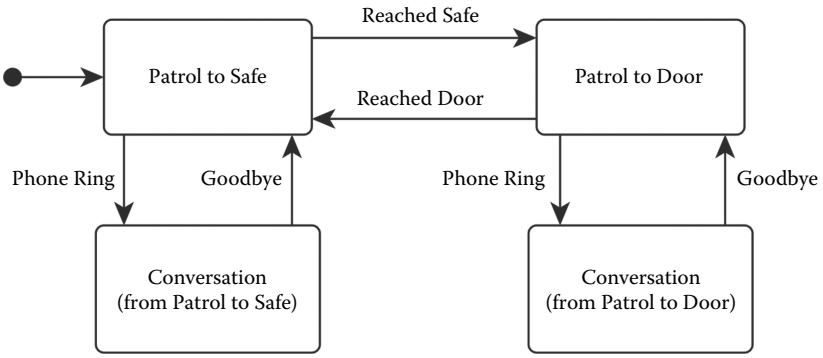

Figure 4.3

Our night watchman FSM requires multiple instances of the Conversation state.

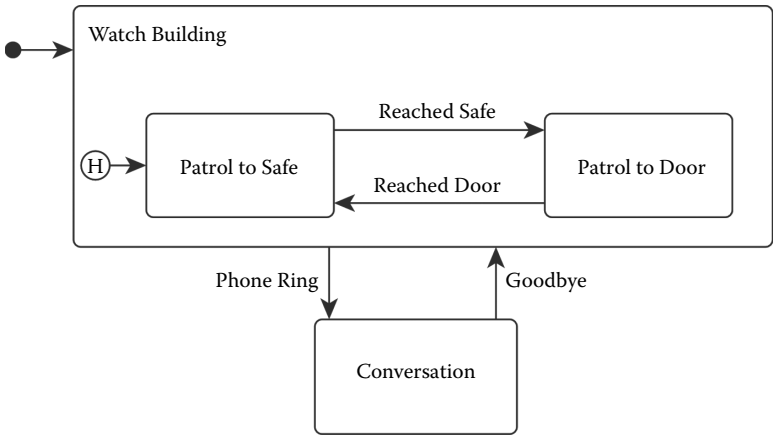

Figure 4.4

An HFSM solves the problem of duplicate Conversation states.

Returning to the night watchman example, if we nest our two *Patrol* states into a state machine called *Watch Building*, then we can get by with just one *Conversation* state, as shown in Figure 4.4.

The reason this works is that the HFSM structure adds additional hysteresis that isn't present in an FSM. With a standard FSM, we can always assume that the state machine starts off in its initial state, but this is not the case with a nested state machine in an HFSM. Note the circled "H" in Figure 4.4, which points to the "history state." The first time we enter the nested *Watch Building* state machine, the history state indicates the initial state, but from then on it indicates the most recent active state of that state machine.

Our example HFSM starts out in *Watch Building* (indicated by the solid circle and arrow as before), which chooses *Patrol to Safe* as the initial state. If our NPC reaches the safe and transitions into *Patrol to Door*, then the history state switches to *Patrol to Door*. If the NPC's phone rings at this point, then our HFSM exits *Patrol to Door* and *Watch*

Building, transitioning to the *Conversation* state. After *Conversation* ends, the HFSM will transition back to *Watch Building* which resumes in *Patrol to Door* (the history state), not *Patrol to Safe* (the initial state).

As you can see, this setup achieves our design goal without requiring duplication of any states. Generally, HFSMs provide much more structural control over the layout of states, allowing larger, complex behaviors to be broken down into smaller, simpler pieces.

The algorithm for updating an HFSM is similar to updating an FSM, with added recursive complexity due to the nested state machines. Pseudocode implementation is fairly complicated, and beyond the scope of this overview article. For a solid detailed implementation, check out Section 5.3.9 in the book *Artificial Intelligence for Games* by Ian Millington and John Funge [Millington and Funge 09].

FSMs and HFSMs are incredibly useful algorithms for solving a wide variety of problems that game AI programmers typically face. As discussed there are many pros to using an FSM, but there are also some cons. One of the major potential downsides of FSMs is that your desired behavior might not fit into the structure elegantly. HFSMs can help alleviate this pressure in some cases, but not all. For example, if an FSM suffers from "transition overload" and hooks up every state to every other state, and if an HFSM isn't helping, other algorithms may be a better choice. Review the techniques in this article, think about your problem, and choose the best tool for the job.

4.4 Behavior Trees

A *behavior tree* describes a data structure starting from some root node and made up of *behaviors*, which are individual actions an NPC can perform. Each behavior can in turn have child behaviors, which gives the algorithm its tree-like qualities.

Every behavior defines a *precondition*, which specifies the conditions where the agent will execute this behavior, and an *action*, specifying the actual things the agent should do when performing the behavior. The algorithm starts at the root of the tree and examines the preconditions of the behaviors, deciding on each behavior in turn. At each level of the tree, only one behavior can be selected, so if a behavior executes, none of its siblings will be checked, though its children will still be examined. Conversely, if a behavior's precondition does not return true, the algorithm skips checking any of that behavior's children and instead moves onto the next sibling. Once the end of the tree is reached, the algorithm has decided on the highest-priority behaviors to run, and the actions of each are executed in turn.

The algorithm to execute a behavior tree is as follows:

- Make root node the current node
- While current node exists,
  - Run current node's precondition
  - If precondition returns true,
    - Add node to execute list
    - Make node's child current node
  - Else,
    - Make node's sibling current node
- Run all behaviors on the execute list

The real strength of a behavior tree comes from its simplicity. The base algorithm can be implemented quickly due to its straightforward nature. Since trees are stateless, the algorithm doesn't need to remember what behaviors were previously running in order to determine what behaviors should execute on a given frame. Further, behaviors can (and should) be written to be completely unaware of each other, so adding or removing behaviors from a character's behavior tree do not affect the running of the rest of the tree. This alleviates the problem common with FSMs, where every state must know the transition criteria for every other state.

Extensibility is also an advantage with behavior trees. It is easy to start from the base algorithm as described and start adding extra functionality. Common additions are behavior `on_start/on_finish` functions that are run the first time a behavior begins and when it completes. Different behavior selectors can be implemented as well. For example, a parent behavior could specify that instead of choosing one of its children to run, each of its children should be run once in turn, or that one of its children should be chosen randomly to run. Indeed, a child behavior could be run based on a utility system-type selector (see below) if desired. Preconditions can be written to fire in response to events as well, giving the tree flexibility to respond to agent stimuli. Another popular extension is to specify individual behaviors as nonexclusive, meaning that if their precondition is run, the behavior tree should keep checking siblings at that level.

A behavior tree, though simple and powerful, is not always the best choice for a selection algorithm. Since the tree must run from the root every time behaviors are selected, the running time is generally greater than that of a finite-state machine. Additionally, the naïve implementation can have a large number of conditional statements, which can be very slow, depending on your target platform. On the other hand, evaluating every possible behavior in the tree may be slow on others where processing power is the limiting factor. Either approach can be a valid implementation of the algorithm; so the programmer would have to decide what is best.

Since behaviors themselves are stateless, care must be taken when creating behaviors that appear to apply memory. For example, imagine a citizen running away from a battle. Once well away from the area, the "run away" behavior may stop executing, and the highest-priority behavior that takes over could take the citizen back into the combat area, making the citizen continually loop between two behaviors. While steps can be taken to prevent this sort of problem, traditional planners can tend to deal with the situation more easily.

4.5 Utility Systems

Much of AI logic—and, for that matter, computer logic—is based on simple Boolean questions. For example, an agent may ask "Can I see the enemy?" or "Am I out of ammunition?" These are purely "yes or no" questions. The decisions that come out of Boolean questions are often just as polarized. As we saw in the prior architectures, the results of these questions are often mapped directly to a single action. For instance,

```
if (CanSeeEnemy())
{
    AttackEnemy();
}
```

```
if (OutOfAmmo())
{
    Reload();
}
```

Even when multiple criteria are combined, Boolean equations tend to lead to a very discrete result set.

```
if (OutOfAmmo() && CanSeeEnemy())
{
    Hide();
}
```

Many aspects of decision making aren't quite as tidy, however. There are numerous questions that can be asked where a "yes or no" answer is not appropriate. For example, we may want to consider how far away the enemy is, how many bullets I have left, how hungry I am, how wounded I am, or any number of continuous values. Correspondingly, these continuous values can be mapped over into *how much* I want to take an action rather than simply whether to take the action or not. A utility-based system measures, weighs, combines, rates, ranks, and sorts out many considerations in order to decide the *preferability* of potential actions. Using the above example as a guide, we could assess how *strongly* we want (or need!) to attack, reload, hide, etc.

While utility techniques can be used to supplement the transition logic of other architectures, it is very possible to build an entire decision engine based on utility. In fact, there are times when building a utility-based AI is far preferable to other methods. These might include games where there are many possible actions, and either there isn't a single "right" answer or the selection of a preferable action might be based on a large number of competing inputs. In these cases, we are going beyond simply using utility to measure or rate something. Instead, we are using it to drive the actual decision mechanism as well. Another way of stating it is that, rather than saying "This is the one action you will do," the utility-based system *suggests*, "Here are some possible options that you might want to do."

One well-documented example of this is the use of utility in *The Sims*. In these games, the agents (i.e., the actual "Sims") take information from their environment and combine it with their own internal state to arrive at a preferability score for each potential action. For example, the fact that I am "very hungry" combined with the availability of "poor food" would certainly be more attractive than if I was only "a little hungry." Additionally, the proximity of "spectacular" food might still make for a high priority even if I was only "a little hungry." Note that the descriptors "spectacular," "rather," "poor," and "a little" would actually be numbers between some set minimum and a maximum. (A typical method to use is a floating point number between 0 and 1.)

When it is time to select a new action (either because the current one is finished or through some sort of interrupt system), some method is used to select from among the candidates. For example, the scores for the potential actions could be sorted so that we can simply select the "most appropriate" action—that is, the one with the highest score. An alternate way is to use the scores to seed a weighted random selection. By casting a random number against these weighted probabilities, the most preferable actions have a higher chance of being selected. As an action's suitability goes up, its score goes up, as does its chance of being selected.

Another example of where utility-based architectures might be preferable to other architectures is RPGs. Often in these games, the options that an agent has are varied and possibly only subtly better or worse, given the situation. For instance, selecting what weapon, spell, item, or action should be taken given the type of enemy, the agent's status, the status of the player, etc., can be a complicated balancing act.

Another wheelhouse of utility architectures is any game system with an economic decision layer. The question of units or buildings to construct in a real-time strategy game, for example, is a juggling act of costs, times, and often many axes of priority (e.g., "offense" or "defense"). An architecture based on utility can often be more adaptable to changing game situations. As such, it can recover better from being disrupted than can more scripted models, which can suffer from either being hopelessly confused or can simply trundle along as if nothing ever happened.

The primary reason for this adaptability is that preferability scores are highly dynamic. As the game situation changes—either through a change in the environment or a change in state of the agent—the scores for most (if not all) of the actions will change. As the action scores change, so does their likelihood of being selected as a "reasonable" action. The resulting ebb and flow of action scores—especially when combined with a weighted random selection—often leads to very dynamic emergent behavior.

On the other hand, unlike the architectures that use Boolean-transitioned decision logic, utility systems are often somewhat unpredictable. Because the selections are based on how much the actions "make sense" in a given situation and context, however, the actions should tend to look *reasonable*. This unpredictability has benefits and drawbacks. It can improve believability because the variety of actions that could occur in a given situation can make for far more natural-looking agents rather than the predictably robotic if/then-based models. While this is desirable in many situations, if your design calls for specific behaviors at very certain moments, you must make a point to override the utility calculations with more scripted actions.

Another caveat to using utility-based architecture is that all the subtlety and responsiveness that you gain often comes at a price. While the core architecture is often relatively simple to set up, and new behaviors can be added simply, they can be somewhat challenging to tune. Rarely does a behavior sit in isolation in a utility-based system. Instead, it is added to the pile of all the other potential behaviors with the idea that the associated mathematical models will encourage the appropriate behaviors to "bubble to the top." The trick is to juggle all the models to encourage the most reasonable behaviors to shine when it is most appropriate. This is often more art than science. As with art, however, the results that are produced are often far more engaging than those generated by using simple science alone.

For more on utility-based systems, see the article in this book, *An Introduction to Utility Theory* [Graham 13] and the book *Behavioral Mathematics for Game AI* [Mark 09].

4.6 Goal-Oriented Action Planners

Goal-Oriented Action Planning (GOAP) is a technique pioneered by Monolith's Jeff Orkin for the game *F.E.A.R.* in 2005, and has been used in a number of games since, most recently for titles such as *Just Cause 2* and *Deus Ex: Human Revolution*. GOAP is derived from the Stanford Research Institute Problem Solver (STRIPS) approach to AI which was first developed in the early 1970s. In general terms, STRIPS (and GOAP) allows an AI system

to create its own approaches to solving problems by being provided with a description of how the game world works—that is, a list of the actions that are possible, the requirements before each action can be used (called "preconditions"), and the effects of the action. The system then takes a symbolic representation of the initial state of the world and some set of objective facts that need to be achieved. In GOAP these objectives are typically chosen from a predetermined set of goals that an NPC may want to achieve, chosen by some method such as priority or state transition. The planning system can then determine a sequence of actions that will allow the agent that it is controlling to change the world from the original state into a state that contains the facts that need to be true to satisfy its current goals. In classical planning this would ideally be the critical path to the target state, and that target would be the most easily reachable state that contained all of the objective facts.

GOAP works by "*backwards chaining search,*" which is a fancy phrase which means starting with the goals you want to achieve, working out what actions are required for those to happen, then working out what needs to happen in order to achieve the preconditions of the actions you just identified and so on. You continue to work backwards in this fashion until you arrive at the state you started from. It's a fairly traditional approach, which has fallen out of favor in the scientific world, replaced by "*forwards chaining search*" which relies on heuristic search, pruning, and other tricks. Backwards search is a solid workhorse, however, and although it's less elegant, it's far easier to understand and implement than more modern techniques.

Backwards chaining search works in the following manner:

- Add the goal to the outstanding facts list
- For each outstanding fact
  - Remove this outstanding fact
  - Find the actions that have the fact as an effect
  - If the precondition of the action is satisfied,
    - Add the action to the plan,
    - Work backwards to add the now-supported action chain to the plan
  - Otherwise,
    - Add the preconditions of the action as outstanding facts

One final interesting aspect of GOAP is that it allows "context preconditions" that are ignored by the planning system, but must be satisfied at run-time in order for an action to be executed. This allows for reasoning to bypass certain aspects of the world that cannot be easily represented symbolically—such as ensuring line of sight to a target before beginning to fire—while ensuring that by accessing information not made available during planning (to ensure the search remains tractable), these constraints can be met. This allows the plan GOAP generates to be somewhat flexible, and the actions it calls for apply more at a tactical level than at the most basic level of execution. That is, the plan tells you *what* to do, but not necessarily *how* to do it. For example, detailed instructions such as how to establish a line of sight to begin shooting are omitted and can be handled more reactively.

Let's suppose we have a typical NPC soldier character, whose goal is to kill another character. We can represent this goal as `Target.Dead`. In order for the target to die, the character needs to shoot him (in a basic system). A precondition of shooting is having a weapon equipped. Assuming our character doesn't have one, we now need an action that

can give the character a weapon, perhaps by drawing one from a holster. This, of course, has its own precondition—that there is a weapon available in the character's inventory. If this is the case, we have just created a simple plan of drawing the weapon and then shooting. What if the character doesn't have a weapon? Then our search would have to find a way to get one. If that isn't possible the search can backtrack and look for alternatives to the shoot action. Perhaps there is a mounted weapon nearby that could be used to provide the `Target.Dead` effect, or even a vehicle that we can use for running over the target. In either case, it's clear that by providing a comprehensive set of action choices of what *can* be done in the world, we can leave it up to the character to decide what *should* be done, letting dynamic and interesting behaviors emerge naturally, rather than having to envisage and create them during development.

Finally, consider a game in which weapons have a maximum range. As a context precondition, we can say that the target must be within that range. The planner won't spend time in its search trying to make this true—it can't, as it would involve reasoning about how the target might move and so on—but it either won't fire its weapon until the condition is true, or it will instead use an alternative tactic such as a different weapon with a longer range.

There's a lot to like about an approach to NPC control based on automated planning. It streamlines the development process by allowing designers to focus on creating simple components that will self-assemble into behaviors, and it also allows for "novel" solutions, which may never have been anticipated by the team, often making for excellent anecdotes that players will re-tell. GOAP itself remains the lowest hanging fruit of what automated planning can provide and, from a purely scientific point of view, the state of the art has progressed significantly since it was developed. With that said, it can still be a very powerful technique when used correctly, and provides a good, adaptable starting point for specific customization.

It is worth noting that these kinds of approaches that adopt a character-centric view of intelligence remove a lot of the authorial and directorial control from the development team. Characters that can "think" for themselves can become loose cannons within the game world, creating plans that, while valid for achieving the character's goals, do not achieve the broader goals of creating immersive and engaging experiences, and this can then potentially disrupt cinematic set-pieces if, for example, a soldier's plan doesn't take him past the conveniently placed Big Red Barrel.

While it's possible to avoid these kinds of issues by using knowledge engineering techniques and representational tricks, it isn't as straightforward as with architectures such as behavior trees, which would allow the desired behavior to be injected directly into the character's decision logic. At the same time, a GOAP approach is significantly easier to design than one based around hierarchical task networks, since in GOAP you just need to describe the mechanics of the objects within a world.

GOAP and similar techniques are not silver bullet solutions, but in the right circumstances they can prove to be very powerful in creating realistic behaviors and immersive-feeling characters that players can fully engage with.

4.7 Hierarchical Task Networks

Though GOAP is perhaps the best-known game planner, other types of planners have gained popularity as well. One such system, *hierarchical task networks* (HTN), has been used in titles such as Guerrilla Games' *KillZone 2* and High Moon Studios' *Transformers:*

Fall of Cybertron. Like other planners, HTN aims to find a plan for the NPC to execute. Where it differs is how it goes about finding that plan.

HTN works by starting with the initial world state and a root task representing the problem we are looking to solve. This high-level task is then decomposed into smaller and smaller tasks until we end up with a plan of tasks we can execute to solve our problem. Each high-level task can have multiple ways of being accomplished, so the current world state will be used to decide which set of smaller tasks the high-level task should be decomposed into. This allows for decision making at multiple levels of abstraction.

As opposed to *backward* planners like GOAP, which start with a desired world state and move backwards until it reaches the current state world state, HTN is a *forward* planner, meaning that it will start with the current world state and work towards a desired solution. The planner works with several types of primitives, starting with the *world state*. The world state represents the state of the problem space. An example in game terms might be an NPC's view of the world and him in it. This world state is broken up into multiple properties such as his health, his stamina, enemy's health, enemy's range, and the like. This knowledge representation would allow the planner to reason about what to do.

Next, we have two different types of tasks: *primitive tasks* and *compound tasks*. A primitive task is an actionable thing that can be done to solve a problem. In game terms, this could be *FireWeapon*, *Reload*, and *MoveToCover*. These tasks are able to affect the world state, such as how the *FireWeapon* task would use ammo and the *Reload* task would refill the weapon. *Compound task*s are higher level tasks that can be accomplished in different ways, described as *methods*. A method is a set of tasks that can accomplish the compound task, along with preconditions determining when a method may be used. Compound tasks allow HTN to reason about the world and decide which course of action to take.

Using compound tasks, we can now build an HTN domain. The domain is a large hierarchy of tasks that represent all the ways of solving our problem, such as how to behave as an NPC of some type. The following pseudocode shows how a plan is built.

- Add the root compound task to our decomposing list
- For each task in our decomposing list
  - Remove task
  - If task is compound
    - Find method in compound task that is satisfied by the current world state
    - If a method is found, add method's tasks to the decomposing list
    - If not, restore planner to the state before the last decomposed task
  - If task is primitive
    - Apply task's effects to the current world state
    - Add task to the final plan list

As mentioned, HTN planners start with a very high-level root task and continuously decompose it into smaller and smaller tasks. This decomposition is steered with each compound task's set of methods by comparing each method's conditions with the current world state. When we finally come across a primitive task, we add it to our final plan. Since each primitive task is an actionable step, we can apply its effects to the world state, essentially moving forward in time. Once the decomposing list is empty we will either have a valid plan or have backed out entirely leaving us with no plan.

To demonstrate how an HTN works, suppose a game has a Soldier NPC that needs its AI written. The root compound task might be named *BeSoldierTask*. Next, the soldier should behave differently if he had an enemy to attack or not. Therefore, two methods are needed to describe what to do in these cases. In the case where an enemy is present, the *BeSoldierTask* would decompose using the method that required that condition. The method's task in this case would be *AttackEnemyTask*. This task's methods define the different ways that the soldier could attack. For example, the soldier could shoot from a cover position if he has ammunition for his rifle. If he didn't have ammo for his firearm, he could charge the enemy and attack him with his combat knife. Writing these give *AttackEnemyTask* two methods to complete the task.

The more we drill down on a soldier's behavior, the more the hierarchy forms and refines. The structure of the domain fits naturally in how one might describe the behavior to another person.

Since HTNs describe behavior using a hierarchical structure, building and reasoning about characters is done in a natural way, allowing designers to more easily read through HTN domains, assisting with collaboration between programming and design. Like other planners, the work that is actually done by the AI is kept in nice modular primitive tasks, allowing a lot of reuse across different AI characters.

Since HTN is a search through a graph, the size of your graph will affect search times, but there are two ways to control search size. First, the method's conditions can be used to cull entire branches of the hierarchy. This happens naturally as behaviors are built. Second, having partial plans can defer complex calculations until plan execution. For example, consider the compound task *AttackEnemy.* One method might have the subtasks *NavigateToEnemy* followed by *MeleeEnemy*. *NavigateToEnemy* requires pathing calculations, which can not only be costly, but could be affected by the state of the world, which might change between planning and execution. To utilize partial plans, split these two tasks into two methods, rather than one method with subtasks: *NavigateToEnemy*, if the enemy is out of range, and *MeleeEnemy* when in range. This allows us to only form a partial plan of *NavigateToEnemy* when the enemy is out of range, shortening our search time.

One other note is that the user needs to build the network for HTN to work. This is a double-edged sword when comparing it to GOAP-style planners. While this allows the designer to be very expressive in the behavior they are trying to achieve, it removes the NPC's ability to build plans that the designer might not have thought of. Depending on the game you are building, this can be considered either a strength or a weakness.

4.8 Conclusion

With such a wide variety of behavior selection algorithms available, it is imperative that the AI programmer have knowledge of each tool in the toolbox in order to best apply each to a given situation. Which algorithm is best for a given NPC can depend on the game, the knowledge state of the NPC, the target platform, or more. While this is not a comprehensive treatment on every available option, knowing a bit about where to start with the options can be invaluable. By taking time to think carefully about the needs of the game, an AI system can be crafted to give the best player experience while maintaining the balance between development time and ease of creation.

References

[Graham 13] R. Graham. "An Introduction to Utility Systems." In *Game AI Pro*, edited by Steve Rabin. Boca Raton, FL: CRC Press, 2013.

[Mark 09] D. Mark. *Behavioral Mathematics for Game AI*. Boston, MA: Cengage Learning, 2009.

[Millington and Funge 09] I. Millington and J. Funge. *Artificial Intelligence for Games*. Burlington, MA: Morgan Kaufmann, 2009, pp. 318–331.