CMSC 425: Lecture 21
Artificial Intelligence for Games: Decision Making

Reading: This material is discussed in Chapter 5 of the book by Millington and Funge ("AI for Games"). The material on Behavior Trees has been taken from an online lecture by Alex Champandard, "Behavior Trees for Next-Gen Game AI," which appears on `aigamedev.com` (visit: `http://aigamedev.com/insider/presentations/behavior-trees/`).

Decision Making: Designing general-purpose AI systems that are capable of modeling interesting behaviors is a challenging task. On the one hand, we would like our AI system to be general enough to provide a game designer the ability to specify the subtle nuances that make a character interesting. On the other hand we would like the system to be easy to use and powerful enough that relatively simple behaviors can be designed with ease. It would be nice to have a library of different behaviors and different mechanisms for combining these behaviors in interesting ways.

Today we will discuss a number of different methods, ranging from fairly limited to more complex. In particular, we will focus on three different approaches, ranging from simple to more sophisticated.

- Decision trees (and variants)
- Finite state machines (and variants)
- Behavior trees

At an extreme level of abstraction, you might wonder what the big deal is. After all, a decision making process quite simply is a (possibly randomized) algorithm that maps a set of input conditions into a set of actions. Thus, it could be implemented using any programming language. Indeed, scripting languages are often used for encoding decision-making systems.

The various techniques that we are going to present could all be implemented via direct implementation in any programming or scripting language. The problem with expressing complex behaviors through programs is that, due to their extremely general nature, programs can be difficult to understand and difficult to reason about. While the aforementioned techniques are limited in their capabilities, they do provide game designers a clean, visually-based structure in which to describe and reason about the behaviors they represent.

Decision Trees: As a starting point, let's consider perhaps the simplest structuring device for implementing a decision-making procedure, the *decision trees*. Decision trees are fast, easy to implement, and simple to understand. A decision tree is a rooted (typically) binary tree, where each node, or *decision point*, is labeled with a test, and the children of this node correspond to the possible outcomes of this test. An example of a decision tree for an combatant NPC is shown in Fig. 1.

In the example the decisions were all binary, but this does not need to be the case. For example, as with `switch` statements in Java, it would be possible to have a node with multiple children, where each child corresponds to one of the possible value types.

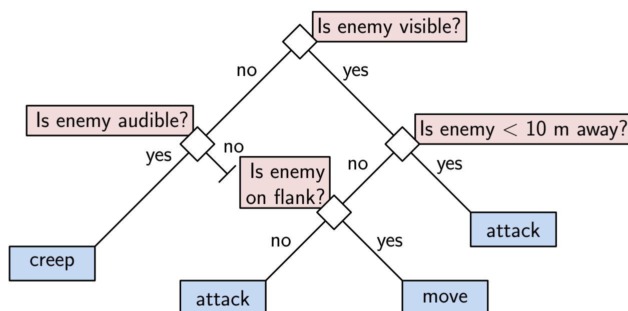


Fig. 1: A decision-tree for a combatant agent.

Variations on a Theme: While our example showed just simple boolean conditions, you might wonder whether it is possible to express more complex conditions using decision trees. The short answer is yes, but it takes a little work. In fact, any decision making algorithm based on a finite sequence of discrete conditions can be expressed in this manner. For example, suppose you have two boolean tests A and B, and you want Action 1 to be performed if both A *and* B are satisfied, and otherwise you want Action 2 to be performed. This could be encoded using the decision tree shown in Fig. 1(a). (Note that encoding a boolean *or* condition is equally easy. Try it.)

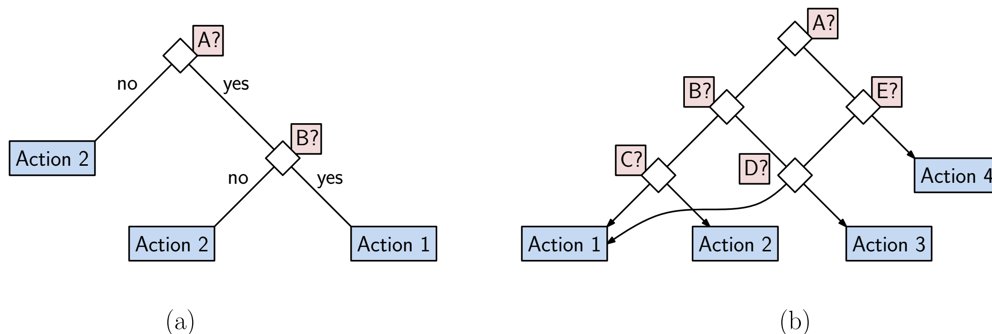


Fig. 2: (a) Complex boolean conditions and (b) subtree sharing.

Observe that in order to achieve the more complex boolean-and condition, we needed to make two copies of the "Action 2" node. Replicating leaf nodes is not a major issue, since each such node would presumably just contain a pointer to a function that implements this action. On the other hand, if we wanted to share entire subtree structures, replicating these subtrees (especially if it is done recursively) can quickly add up to a lot of space. Furthermore, copying is an error-prone process, since any amendment to one subtree would require making the same change in all the others (assuming you want all of them to implement the same decision-making procedure.) One way to avoid the issue of copying subtrees is allow subtrees to be shared (see Fig. 2(b)). In spite of the name "decision trees," the resulting "decision acyclic directed graphs" are just as easy to search.

Another variation on decision trees is to introduce *randomization* to the decision-tree. For example, we might have a decision point that says "flip a coin" (or more generally, generate

a random integer over some finite range with a given probability distribution). Based on the result of this random choice, we could then branch among various actions. This would allow us to add more variation to the behavior of our agents.

Implementing Decision Trees: Decision trees can be implemented in a number of ways. If the tree is small (and it is a tree, as opposed to a directed-acyclic graph) you can translate the tree into an appropriate layered if-then-else statement in your favorite programming/scripting language. More generally, you can express the tree as a graph-based data structure, where internal nodes hold pointers to predicate function and leaf nodes hold pointers to action functions.

Finite State Machines: Decision trees are really too simple to be used for most interesting decision-making processing. The next step up in complexity is to add a notion of *state* to the character, and then make decisions a function of both the current conditions and the character's current state.

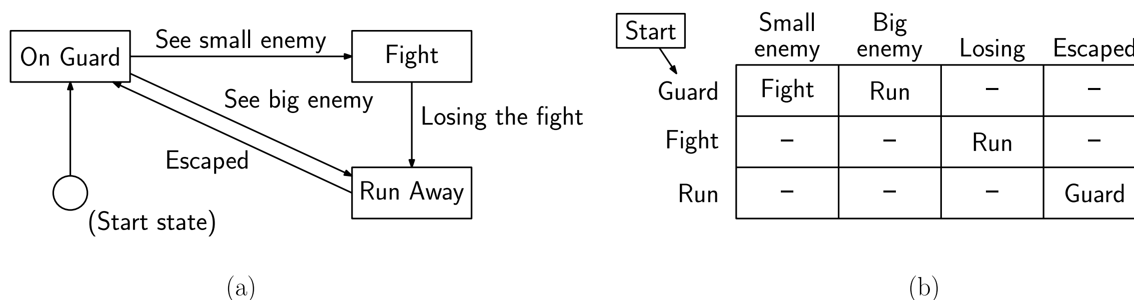
For example, a character may behave more aggressively when it is healthy and less aggressively when it is injured. As another example, a designer may wish to have a character transition between various states (patrolling, chasing, fighting, retreating) in sequence or when triggered by game events. In each state, the characters behavior may be quite different.

A *finite state machine* (FSM) can be modeled as a directed graph, where each node of the graph corresponds to a state, and each directed edge corresponds to a event, that triggers a change of state and optionally some associated action. The associated actions may include things like starting an animation, playing a sound, or modifying the current game state.

As an example, consider the programming of an warrior bot NPC in a first-person shooter. Suppose that as the designer you decide to implement the following type of behavior:

- If you don't see an enemy, stand guard
- While on guard, if you see a small enemy, fight it, but if it is too big, then flee
- If you are fighting, if you discover you are losing the fight, then flee
- While fleeing, if you have escaped to a point where there is no threat, then return to the guarding state

We can encode this behavior in the form of the FSM showed in Fig. 3(a).



	Small enemy	Big enemy	Losing	Escaped
Guard	Fight	Run	–	–
Fight	–	–	Run	–
Run	–	–	–	Guard

Fig. 3: A simple state machine for a warrior bot.

FSMs are a popular method of defining behavior in games. They are easy to implement, easy to design (if they are not too big), and they are easy to reason about. For example, based on the visual layout of the FSM, it is easy to see the conditions under which certain state transitions can occur and whether there are missing transitions (e.g., getting stuck in some state forever).

Implementing State Machines: How are FSMs implemented? A natural implementation is to use a two-dimensional array, where the row index is an encoding of the current state and the column index is an encoding of the possible events that may trigger a transition. Each entry of the array is labeled with the state to which the transition takes place (see Fig. 3(b)). The array will also contain further information, such as what actions and animations to trigger as part of the action.

As can be seen from the example shown in the figure, many state-event pairs result in no action or transition. If this is the case, then the array-based implementation can be space inefficient. A more efficient alternative would be to use the nonempty state-event pairs as keys into a hash table. Assuming a good hash-table implementation, the hash table's size would generally be proportional to the number nonempty entries.

Note that the FSM we have showed is *deterministic*, meaning that there is only a single transition that can be applied at any time. More variation can be introduced by allowing multiple transitions per event, and then using randomization to select among them (again, possibly with weights so that some transitions are more likely than others).

Hierarchical State Machines: One of the principal shortcoming with FSMs is that the number of states can *explode* as the designer dreams up more complex behavior, thus requiring more states, more events, and hence the need to consider a potentially quadratic number of mappings from all possible states to all possible events.

For example, suppose that you wanted to model multiple conditions simultaneously. A character might be *healthy/injured*, *wandering/chasing/attacking*, *aggressive/defensive/neutral*. If any combination of these qualities is possible, then we would require $2 \cdot 3 \cdot 3 = 18$ distinct states. This would also result in a number of repeated transitions. (For example, all nine of the states in which the character is "healthy" would need to provide transitions to the corresponding "injured" states if something bad happens to us. Requiring this much redundancy can lead to errors, since a designer may update some of the transitions, but not the others.)

One way to avoid the explosion of states and events is to design the FSM in a hierarchical manner. First, there are a number of high-level states, corresponding to very broad contexts of the character's behavior. Then within each high-level state, we could have many sub-states, which would be used for modeling more refined behaviors within this state. The resulting system is called a *hierarchical finite state machine* (HFSM).

For example, suppose we add an additional property to our warrior bot, namely that he/she gets hungry from time to time. When this event takes place, the bot runs to his/her favorite restaurant to eat. When the bot is full, he/she returns to the same state. Since this event can occur from within any state, we would need to add these transitions from all the existing states (see Fig. 4).

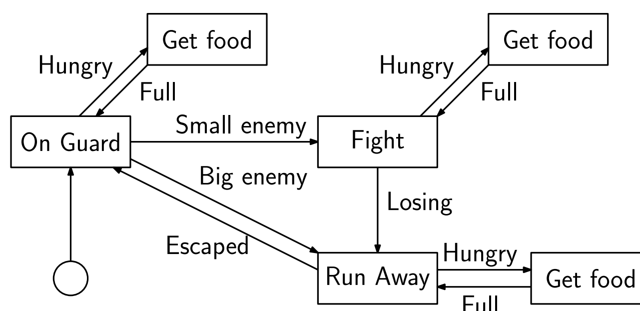


Fig. 4: State machine for a hungry warrior bot.

Of course, this would get to be very tedious if we had a very large FSM. The solution, is to encapsulate most of the guarding behaviors within one FSM, and then treat this as a single *super-state* in a hierarchical FSM (see Fig. 5). The hungry/full transitions would cause us to save the current state (e.g., by pushing it onto a stack), performing the transition. On returning to the state, we would pop the stack and then resume our behavior as before.

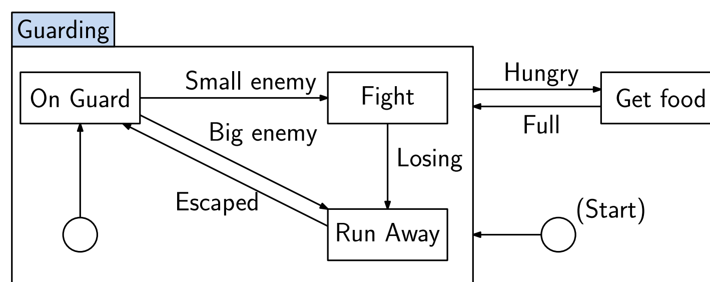


Fig. 5: Hierarchical state machine for a hungry warrior bot.

Note that we create a start state within the super-state. This is to handle the first time that we enter the state. After this, however, we always return to the same state that we left from.

This could be implemented, for example, by storing the state on a stack, where the highest-level state descriptor is pushed first, then successively more local states.

The process of looking up state transitions would proceed hierarchically as well. First, we would check whether the lowest level sub-state has any transition for handling the given event. If not, we could check its parent state in the stack, and so on, until we find a level of the FSM hierarchy where this event is to be handled.

The advantage of this hierarchical approach is that it naturally adds modularity to the design process. Because the number of local sub-states is likely to be fairly small, it simplifies the design of the FSM. In particular, we can store even a huge number of states because each sub-state level need only focus on the relatively few events that can cause transitions at this level.

FSM-Based Adventure Games: Very early text-based adventure games were based on finite-state automata. The earliest example was *Colossal Cave Adventure* by Will Crowther in the 1970's. It was implemented in Fortran and ran on a PDP-10 computer. The game-play

involves a series of short descriptions, after which the player could enter simple commands. The objective was to navigate through the environment to find the treasure. Here is a short example:

```
You are standing at the end of a road before a small brick building. Around you is
a forest. A small stream flows out of the building and down a gully.

> enter building

You are inside a building, a well house for a large spring. There are some keys on
the ground here. There is a shiny brass lamp nearby. There is tasty food here.
There is a bottle of water here.

> take keys

Taken

> go south

You are in a valley in the forest beside a stream tumbling along a rocky bed.
```

It is not hard to see how this could be implemented using an FSM. The player's current position is modeled as the FSM state (with auxiliary information for the player's inventory), and commands were mapped to state transitions, and each state is associated with a short description.

Behavior Trees: While FSMs are general, they are not that easy to design. We would like a system that is more general the FSMs, more structured than programs, and lighter weight than general-purpose planners. Behavior trees were developed by Geoff Dromey in the mid-2000s in the field of software engineering, which provides a modular way to define software in terms of actions and preconditions. They were first used in Halo 2 and were adopted by a number of other games such as Spore.

Let us consider the modeling of a *guard dog* in an FPS game. The guard dog's range of behaviors can be defined hierarchically. At the topmost level, the dog has behaviors for major tasks, such as *patrolling*, *investigating*, and *attacking* (see Fig. 6(a)). Each of these high-level behaviors could then be broken down further into lower-level behaviors. For example, the patrol task may include a subtask for *moving*. The investigate task might include a subtask for *looking around*, and the attack task may include a subtask for *bite* (ouch!).

The leaves of the tree are where the AI system interacts with the game state. Leaves provide a way to gather information from the system through *conditions*, and a way to affect the progress of the game through *actions*. In the case of our guard dog, conditions might involve issues such as the dog's state (is the dog hungry or injured) or geometric queries (is there another dog nearby, and is there a line of sight to this dog?). Conditions are *read-only*. Actions make changes to the world state. This might involve performing an animation, playing a sound, picking up an object, or biting someone (which would presumably alter this other object's state). Conditions can be thought of as *filters* that indicate which actions are to be performed.

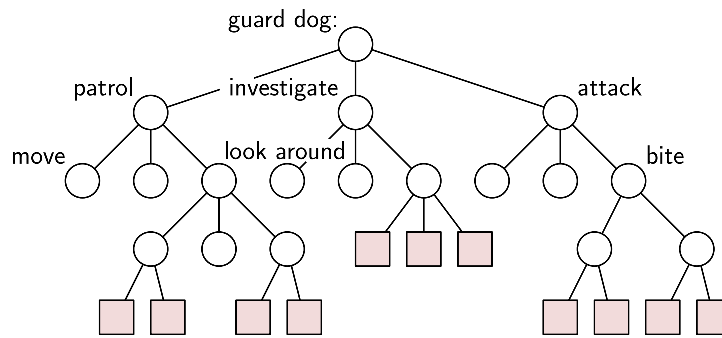


Fig. 6: Sample hierarchical structure of a guard-dog's behavior.

A *task* is a piece of code that models a latent computation. A task consists of a collection *conditions* that determine when the task is enabled and *actions*, which encode the execution of the task. Tasks can end either in *success* or *failure*.

Composing Tasks: *Composite tasks* provide a mechanism for composing a number of different tasks. There are two basic types of composite tasks, which form natural complements.

Sequences: A sequence task performs a series of tasks sequentially, one after the other (see Fig. 7(a)). As each child in the sequence succeeds, we proceed to the next one. Whenever a child task fails, we terminate the sequence and bail out (see Fig. 7(b)). If all succeed, the sequence returns success.

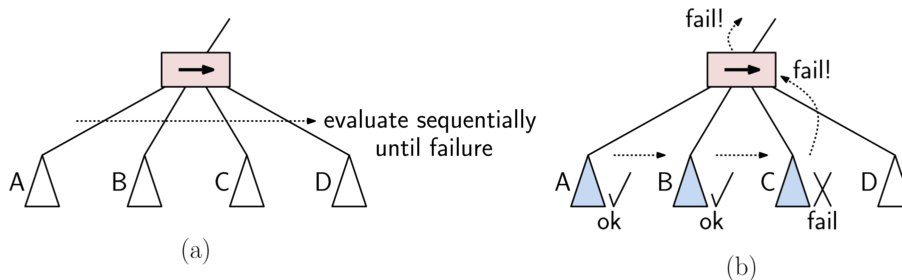


Fig. 7: Sequence: (a) structure and (b) semantics.

Selector: A selector task performs at most one of a collection of child tasks. A selector starts by selecting the first of its child tasks and attempts to execute it. If the child succeeds, then the selector terminates successfully. If the child fails, then it attempts to execute the next child, and so on, until one succeeds (see Fig. 8(b)). If none succeed, then the selector returns failure.

An example of a behavior tree is presented in Fig. 9 for an enemy trying to enter a room. If the door is open, the enemy moves directly into the room (left child of the root). Otherwise, the enemy approaches the door and tries the knob. If it is unlocked, it opens the door. If locked, it breaks the door down. After this, it enters the room.

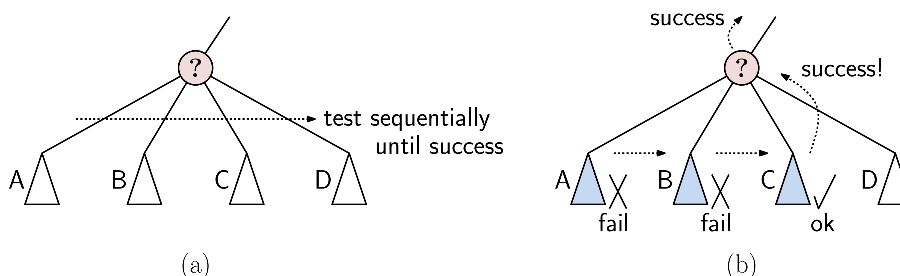


Fig. 8: Selector: (a) structure and (b) semantics.

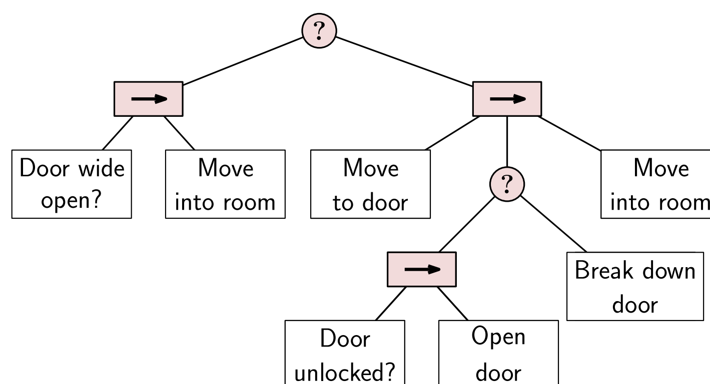


Fig. 9: Example of a behavior tree for an enemy agent trying to enter a room.

Sequences and selectors provide some of the missing elements of FSMs, but they provide the natural structural interface offered by hierarchical finite state machines. Sequences and selectors can be combined to achieve sophisticated combinations of behaviors. For example, a behavior might involve a sequence of tasks, each of which is based on making a selection from a list of possible subtasks. Thus, they provide building blocks for constructing more complex behaviors.

From a software-engineering perspective, behavior trees give a programmer a more structured context in which to design behaviors. The behavior-tree structure forces the developer to think about the handling of success and failure, rather than doing so in an ad hoc manner, as would be the case when expressing behaviors using a scripting language. Note that the nodes of the tree, conditions and tasks, are simply links to bits of code that execute the desired test or perform the desired action. The behavior tree provides the structure within which to organize these modules.