COMP 476

Advanced Game Development

Session 4
Decision Making

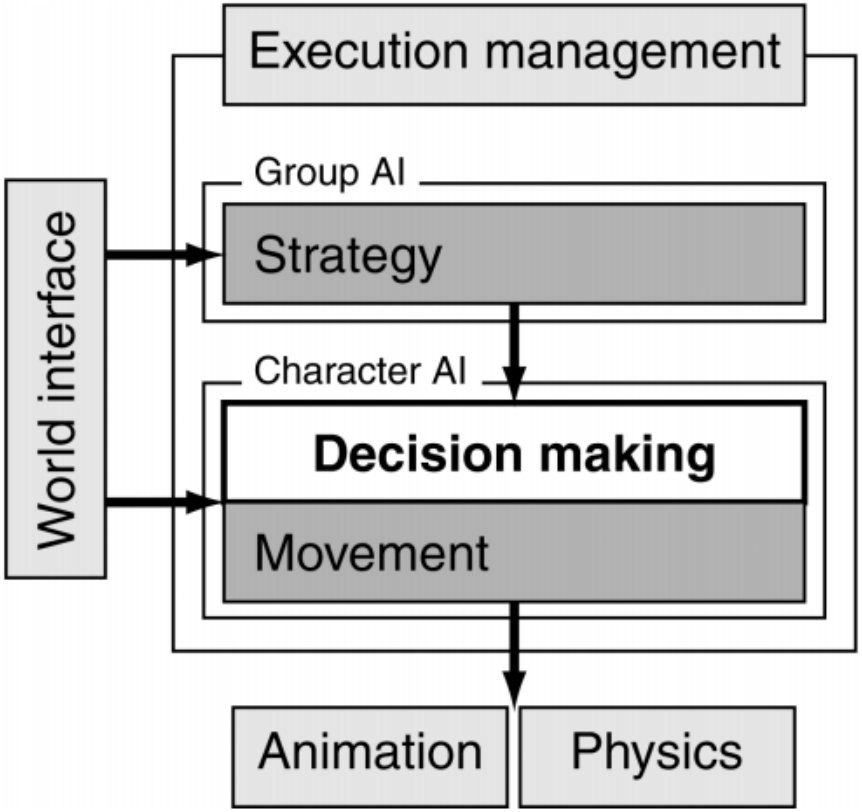
Pathfinding AI (Reading: AI for G, Millington § 5.1-5.5)

Lecture Overview

- Decision Trees
- State Machines
- Behaviour Trees
- Fuzzy Logic
- Goal Oriented Action Planning

Decision Making AI

- **Ability of a game character to decide what to do**
- **Decision Making in Millington's Model**

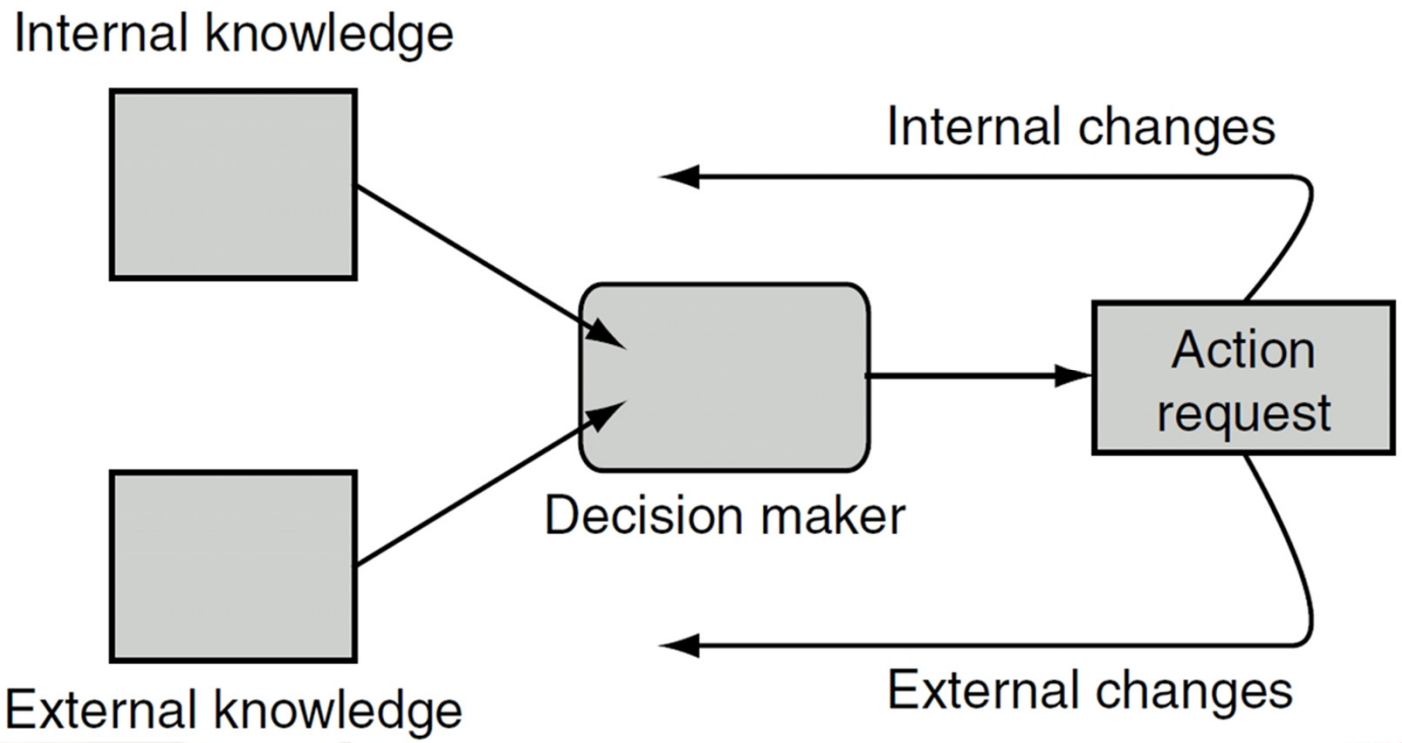


Decision Making AI

- Decision Trees (§ 5.2)
- Finite State Machines (FSM) (§ 5.3)
- Behaviour Trees (§ 5.4)
- Rule-based Systems (§ 5.8)
- Fuzzy Logic & Neural Networks (§ 5.5.2)
- Blackboard Architecture (§ 5.9)
- Utility-based Decision-Making
- Goal-Oriented Behavior
- Others…

Decision Making AI

- **Although there are many different decision making techniques, we can look at them all as *acting in the same way*.**



Decision Making AI

- Typically, the same external knowledge can drive *any of the algorithms* in this section
- But the algorithms themselves *control what kinds of internal knowledge can be used* (although they don't constrain what that knowledge represents, in game terms).
- Actions can have two components:
  - Those that will change the external state of the character or
  - Those that only affect the internal state (*perhaps more significant* in some decision making algorithms)

Decision Trees

- **Fast, easy to understand**
- **Simplest technique to implement, but extensions to the basic algorithm *can be sophisticated***
- **Typically used *to control characters*, animation or other in-game decision making (right up to complex strategic and tactical AI)**
- **Can be learned, and learning is relatively fast (compared to fuzzy logic/neural networks)**
  - **Resulting learned tree is *relatively easy to understand***

Decision Trees

Problem Statement

- Given a set of knowledge, we need to *generate a corresponding action* from a set of possible actions
- **Map between input and output** — typically, same action is used for many different sets of input
  - ***Mapping may be quite complex***
- Need *a method to easily group lots of inputs together under one particular action*, allowing the input values that are significant to control the output

Decision Trees

Problem Statement

- **Example: Grouping a set of inputs under an action**

Enemy is visible Enemy is now < 10m away	Attack
Enemy is visible Enemy is still far (> 10m), but not at flank	Attack
Enemy is visible Enemy is still far (> 10m), at flank	Move
Enemy is not visible, but audible	Creep

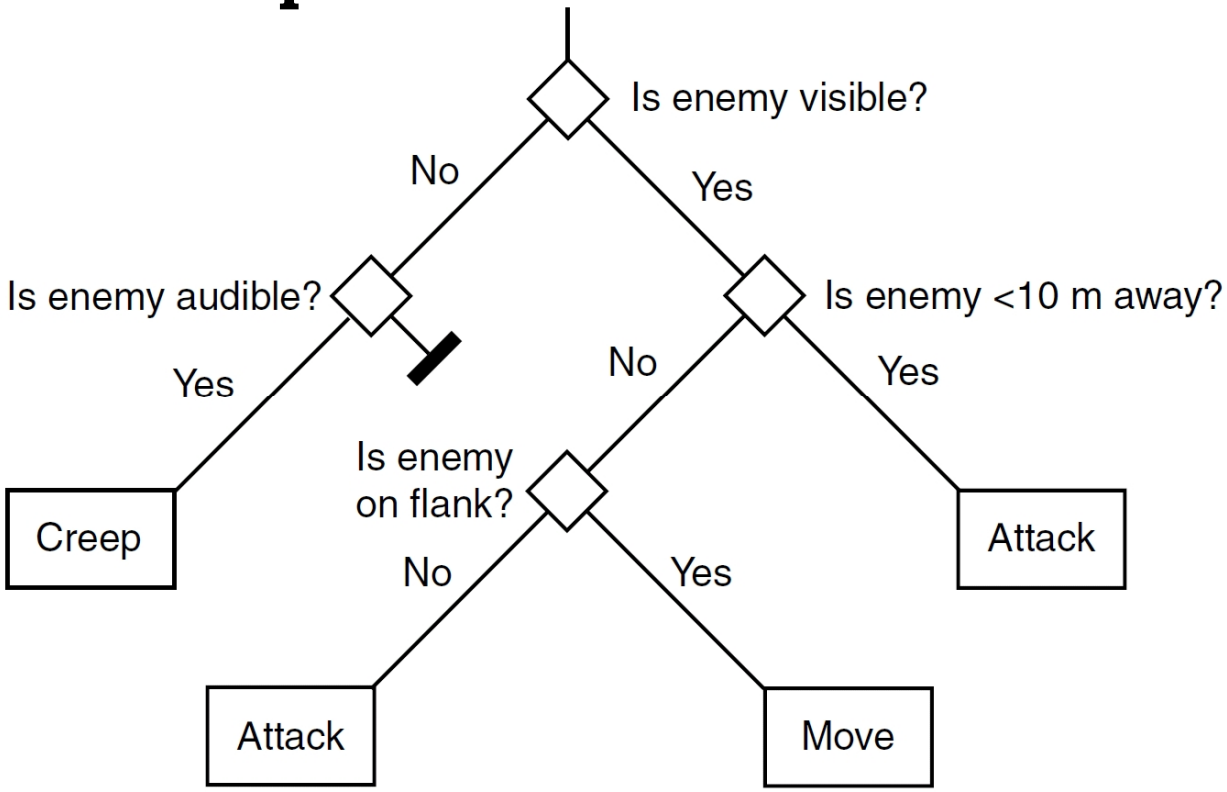
Decision Trees

Algorithm Overview

- Decision Tree: comprises of ***connected decision points***
- Tree has starting decision, its root
- For each decision, starting from the root, one of a set of ongoing options is chosen.
- ***Choice is made based on character's knowledge*** (internal/external) → aim to be simple and fast
- Continues along the tree, making choices at each decision node ***until no more decisions to consider***
- At each leaf of the tree, ***an action is attached***
- This action is carried out immediately

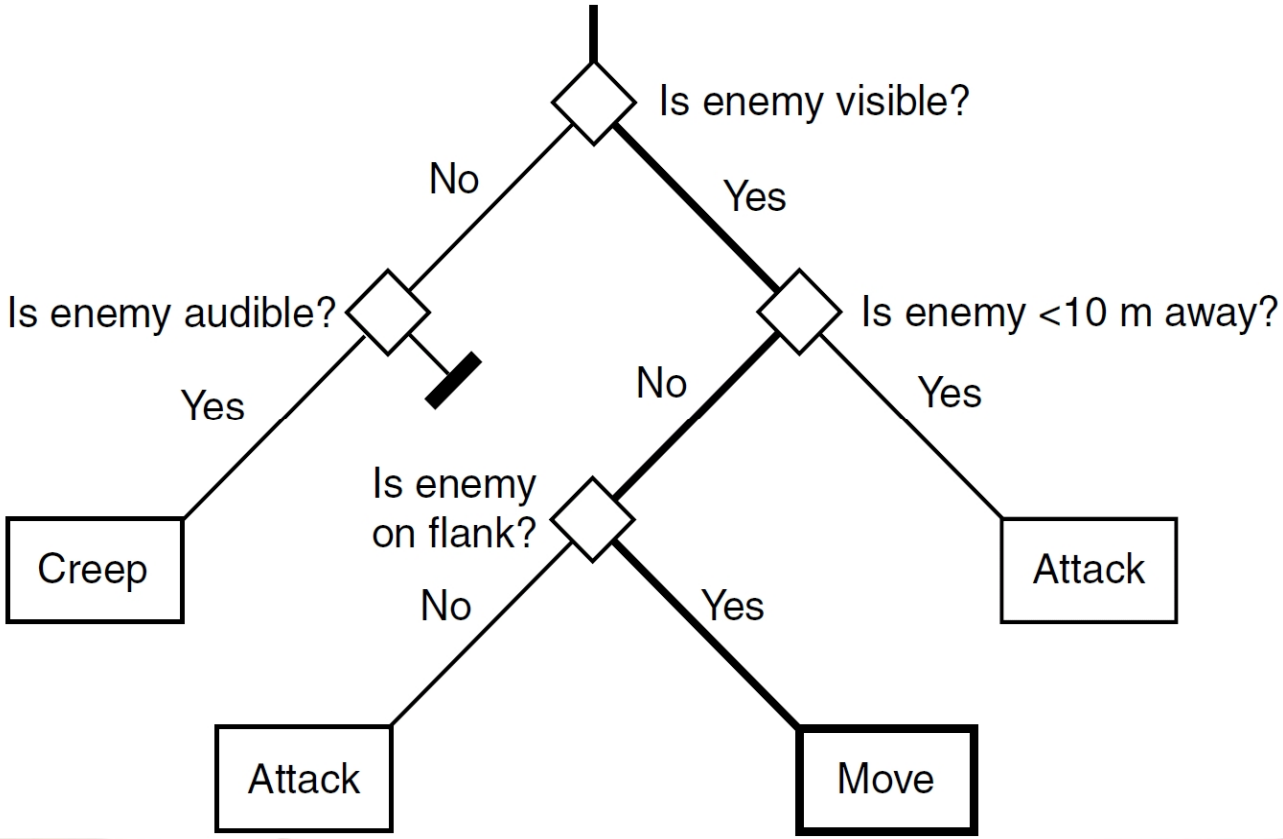
Decision Trees

- **Most decision tree internal nodes make *very simple decisions*, typically with only two (binary) possible responses.**



Decision Trees

- **Below shows the same *decision tree* with a decision having been made.**



Decisions

- **Decisions**: Check a single value and don't contain any Boolean logic (AND, OR)
- A representative set (for example)

Data Type	Decisions
Boolean	Value is true
Enumeration	Matches one of the given set of values
Numeric value	Value is within given range
3D Vector	Vector has a length within given range

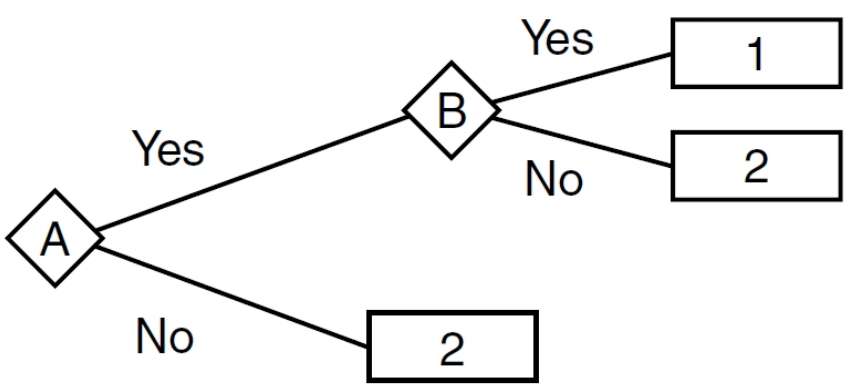
- **Examples**?

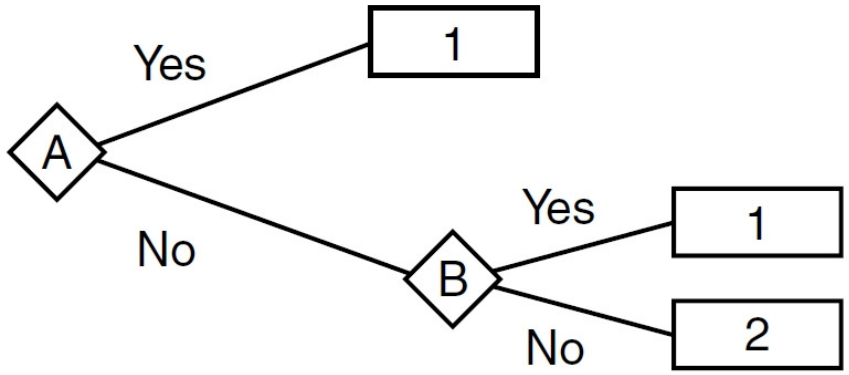
Decision Combos

- **AND** two decisions – ***place in series in the tree***
- **OR** two decisions – also use decisions in series, but the ***two actions are swapped*** over from AND

If A OR B then action 1, otherwise action 2

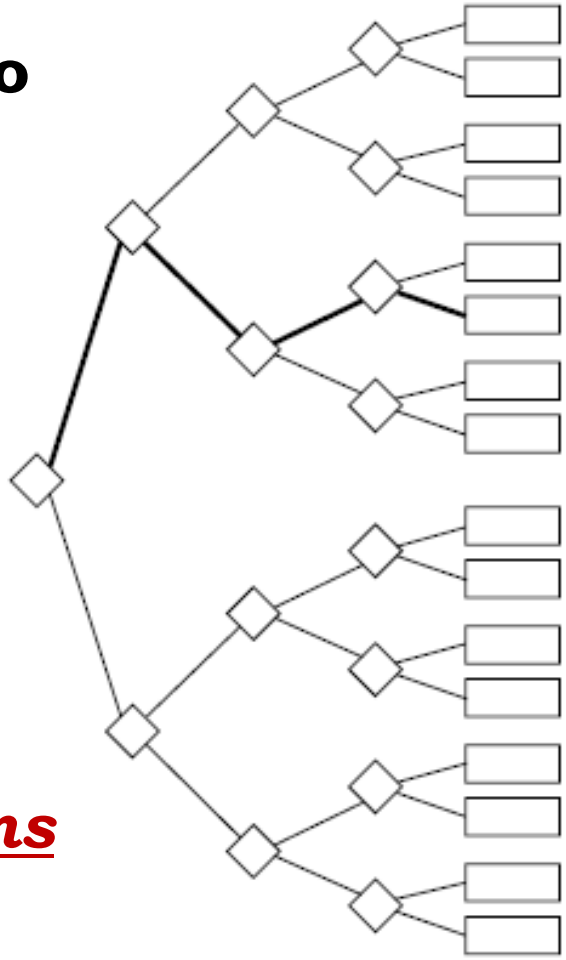
If A AND B then action 1, otherwise action 2





Decision Complexity

- **Number of decisions** that need to be considered is ***usually much smaller than number of decisions*** in the tree.
- Imagine using IF-ELSE statements to test each decision?
- Method of building DTs: ***Start with simple tree***, as AI is tested in the game, ***additional decisions can be added*** in stages to trap special cases or add new behaviors



Knowledge Representation

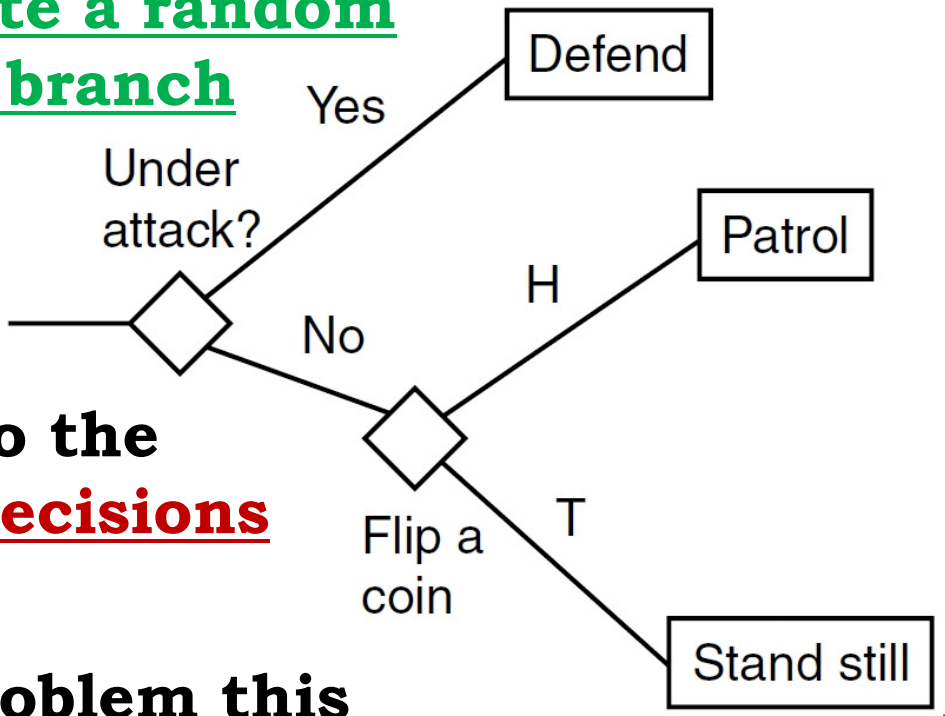
- Decision trees *work directly with primitive data* types.
- Decision trees are most commonly implemented so they *access the state of the game directly*.
- This can lead to "**broken**" decisions *if the game state changes* and the decision relies on particular structures or implementations.
- To avoid this situation, some developers *choose to insulate all access to the state of the game*, creating a world interface.

Decision Making Performance

- Takes no memory, ***performance is linear*** with number of nodes visited
- Assume each decision takes constant amount of time, and the ***binary tree*** with n decision nodes is balanced, performance: $O(\log_2 n)$
- This does not consider the execution time of the different checks required in the DT, ***which can vary a lot!***
  - E.g., line of sight checks using complex ray casting involving the level geometry

Random Decision Trees

- To provide ***some unpredictability and variation*** to making decisions in DTs
- Simplest way: **Generate a random number and choose a branch** based on its value
- DTs are normally intended to run frequently, reacting to the game state, **random decisions can be a problem**
- What is a potential problem this DT if it is run for every frame?



Random Decision Trees

- Alternatively, allow random decision to ***keep track*** of what it decided last time.
- When a decision is considered, a choice is made at random, and that ***choice is stored***.
- Next time the decision is considered, no more randomness, ***previous choice is maintained***, and so on for each frame **until** either
  - ***Something in the world changes***: a new random decision is made, replacing the previous one, or
  - The ***AI "times out"*** after a set time, and a random choice is to be made again.
- Gives variety and realism

State Machines

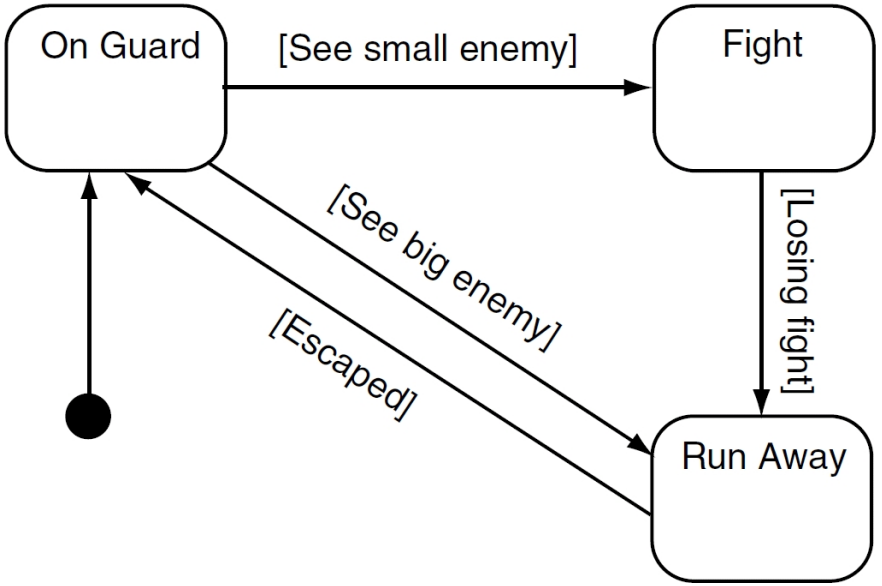
- Often, characters in a game act in ***one of a limited set of ways***
- Carry on doing the same thing until some event or influence makes them change how they act
- Can use decision trees, but is ***easier to model this*** behavior using **state machines** (or finite state machines, FSM)
- State machines consider the following:
  - the ***world around them*** and
  - their ***internal state***

A Basic State Machine

- Each AI character occupies *one state at each instance*
- Actions/behaviors are associated with each state
- So long as a character remains in that state, it will continue carrying out same action/behavior
- States are connected by transitions
- Each transition leads from one state to another, the target state, and each has a set of associated conditions
- **Changing states**: when conditions of a transition are met, transition *triggers* and when transition is followed to the new state, it has *fired*

A Simple Example

- State machine to model a soldier – 3 states
- Each state has its *own set of transitions*
- The *solid circle* (with a *transition w/o trigger* condition) points to the initial state that will be entered when the state machine is first run



State Machines Vs Decision Trees

- Now, name some *obvious differences* in making decisions using *decision trees and state machines*?
- In a decision tree,
  - the same set of decisions is always used, and
  - any action can be reached through the tree.
- In a state machine,
  - only transitions from the current state are considered, so
  - not every action can be reached.

Finite State Machines

- In game AI, a state machine with this kind of structure (as seen earlier) is usually called *a finite state machine (FSM)*
- A Game FSM has *a finite number of states and transitions*
- It has finite internal memory to store its states and transitions
- There are tens of different ways to implement a game FSM
- We'll look at representative implementation…

Hard-coded FSM

- **Hard-coded FSM –**
  - **Consists of *an enumerated value*, indicating which state is currently occupied, and a function that checks if a transition is followed**
  - **States are hard-coded, and limited to what was hard-coded**
- **In the following example, we assume *all call functions have access to the current game state***

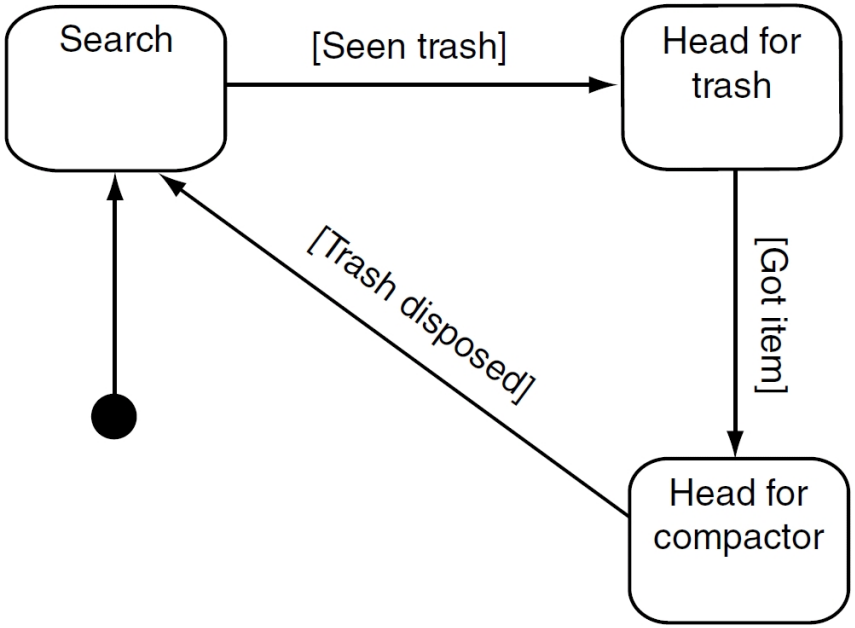
```
class MyFSM:

  # Defines the names for each state
  enum State: {PATROL, DEFEND, SLEEP}

  # Holds the current state
  myState

  def update():            # Polling state changes
  # Find the correct state
  if myState == PATROL:
    if canSeePlayer(): myState = DEFEND # Example
    if tired(): myState = SLEEP         # transitions
  elif myState == DEFEND:
    if not canSeePlayer(): myState = PATROL
  elif myState == SLEEP:
    if not tired(): myState = PATROL

  def notifyNoiseHeard(volume):   # Event driven state
    if myState == SLEEP and volume > 10:    # change
      myState = DEFEND
```

Hard-coded FSM

- **Run time: O(n + m), where n is the number of states, and m is the (average) number of transitions per state. No memory required.**
- **Pros: Easy and quick implementation, *useful for small FSMs***
- **Cons:**
  - ***Inflexible*, does not allow level designers the control over building the FSM logic**
  - ***Difficult to maintain* (alter) – Large FSMs, messy code**
  - ***Every character needs its own coded AI* behaviours…**

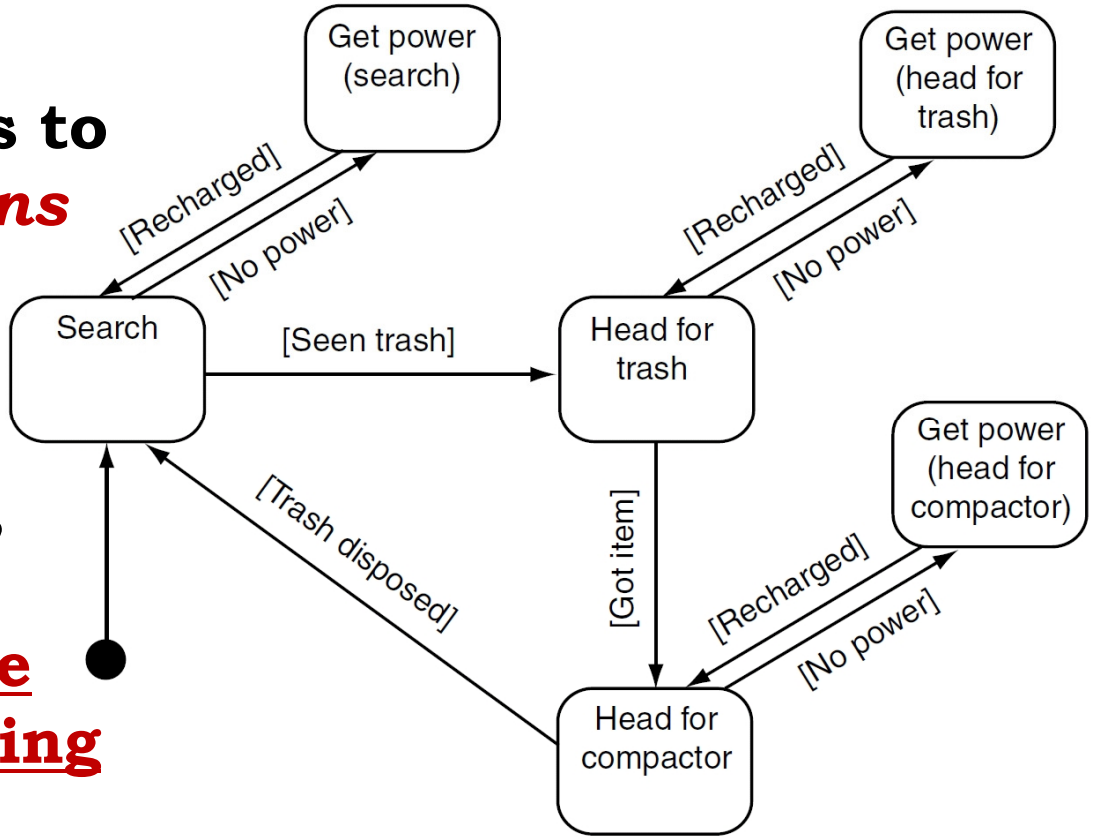
Hierarchical State Machines

- One state machine is a powerful tool, but will *still face difficulty expressing some behaviours*
- Also if you wish to model *somewhat different behaviours from more than one state machine for a single AI character*
- Example: Modeling alarm behaviours with hierarchical SM (using *a basic cleaning robot state machine*)



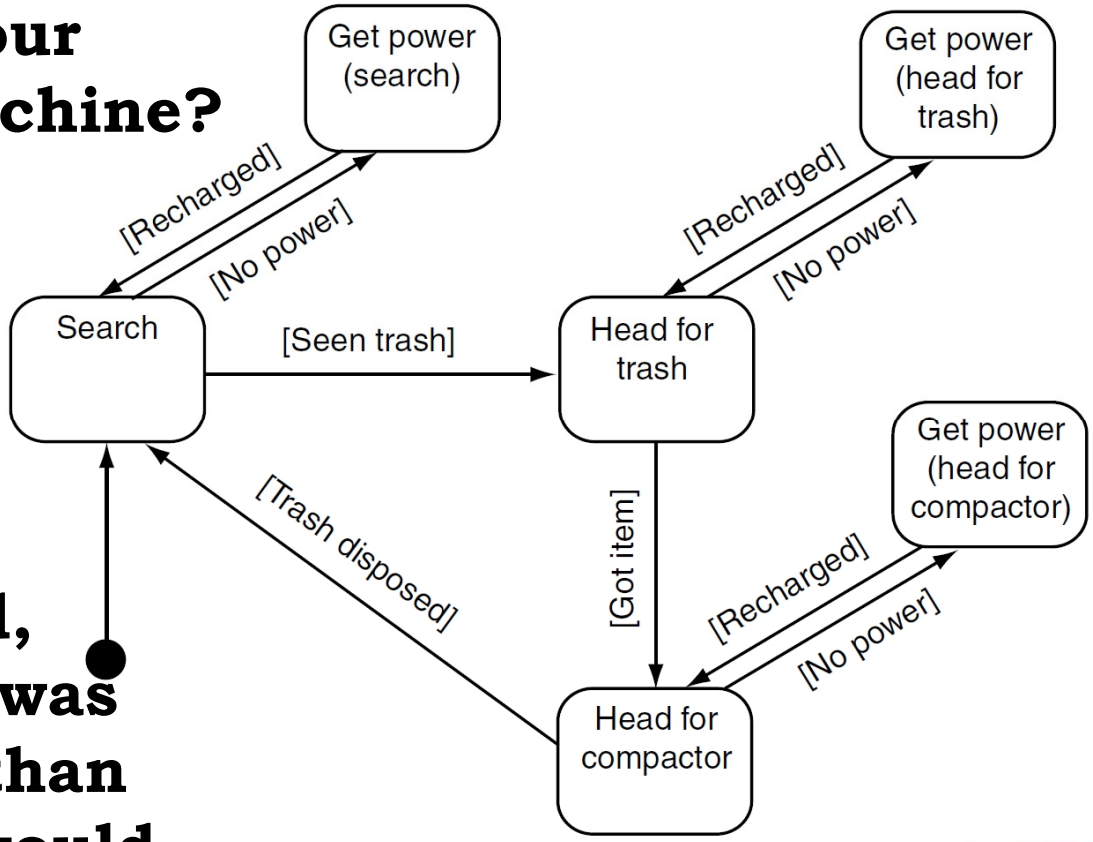
Hierarchical State Machines

- **Alarm mechanism: something that *interrupts normal behaviour* to respond to something important.**
- **If the robot needs to get power if it *runs out of power* and *resume its original duties after recharging*, these transition behaviors must be added to all existing states to ensure robot acts correctly**



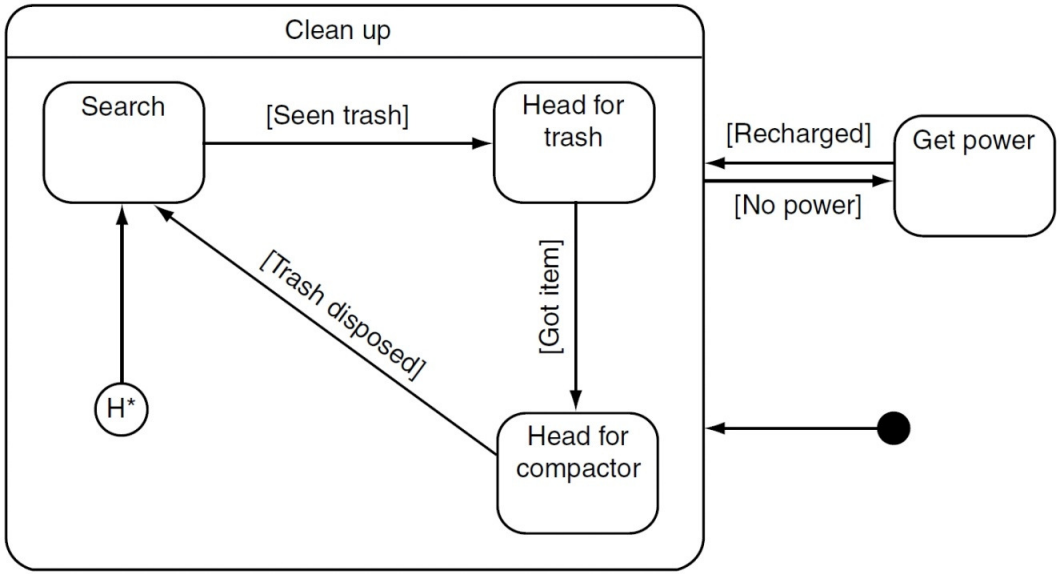
Hierarchical State Machines

- This is not exactly very efficient. Imagine if you had to add *many more (levels of) concurrent behaviors* into your primary state machine?
- What if the robot had to *hide if fighting breaks out*? And this occurs while getting recharged, such that hiding was more important than recharging? We would need 12 states



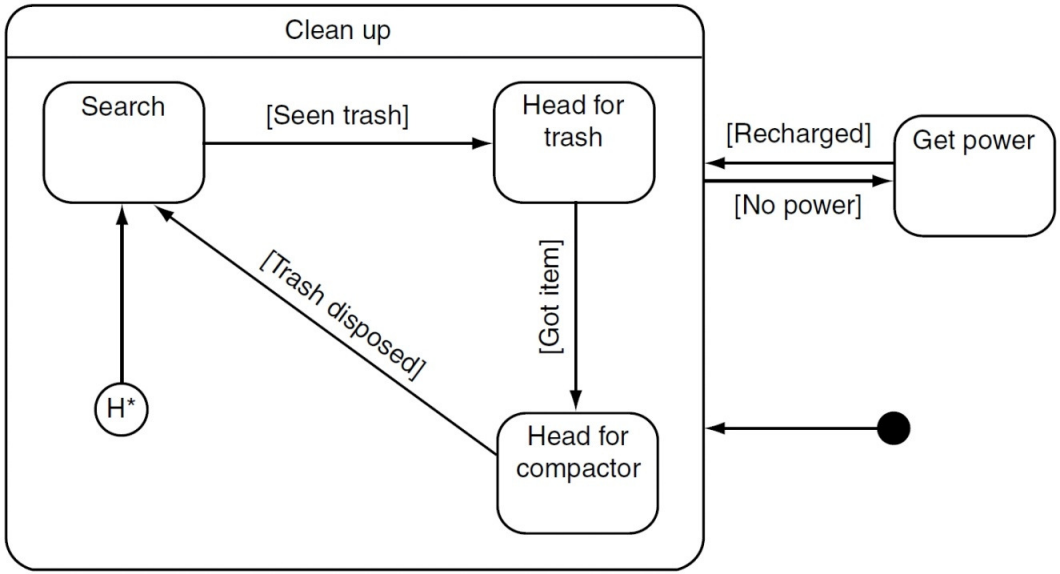
Hierarchical State Machines

- **A hierarchical state machine for a cleaning robot**
  - ***Nested states, one within another*** **– could be in more than one state at a time**
  - **States are arranged in a hierarchy ➔ next state machine down is *only considered when higher level state machine isn't responding* to its alarm**



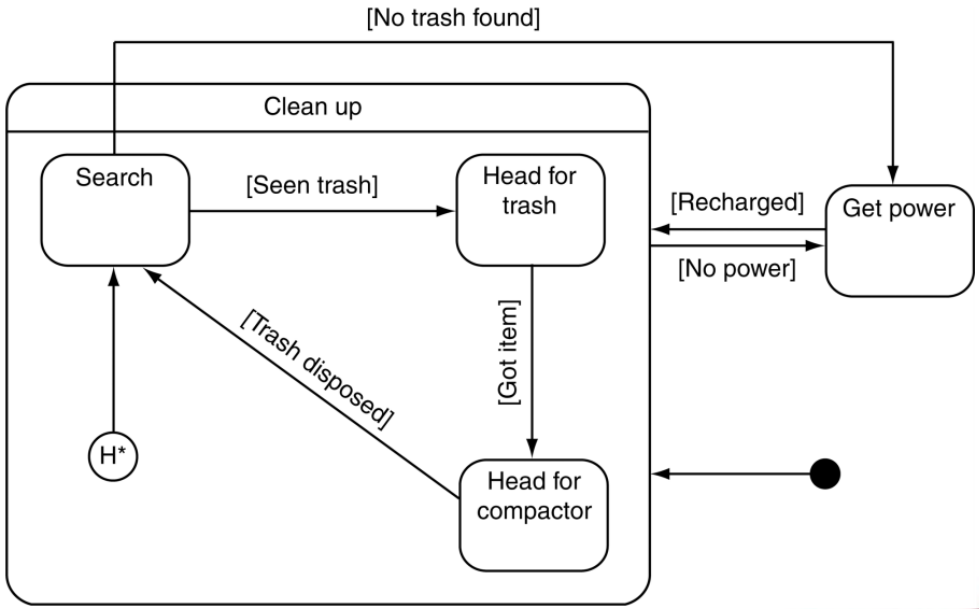
Hierarchical State Machines

- **H* - "history state" node**
  - **When the composite state (lower hierarchy) is first entered, the *H* node indicates which sub-state should be entered***
  - **If composite state was already entered, then previous sub-state is restored using the H* node**



Hierarchical State Machines

- **Hierarchical state machine *with cross hierarchy transition***
  - **Most hierarchical state machines support *transitions between levels of the hierarchy***
  - **Let's say we want the robot to go back to refuel when it does not find any more trash to collect…**

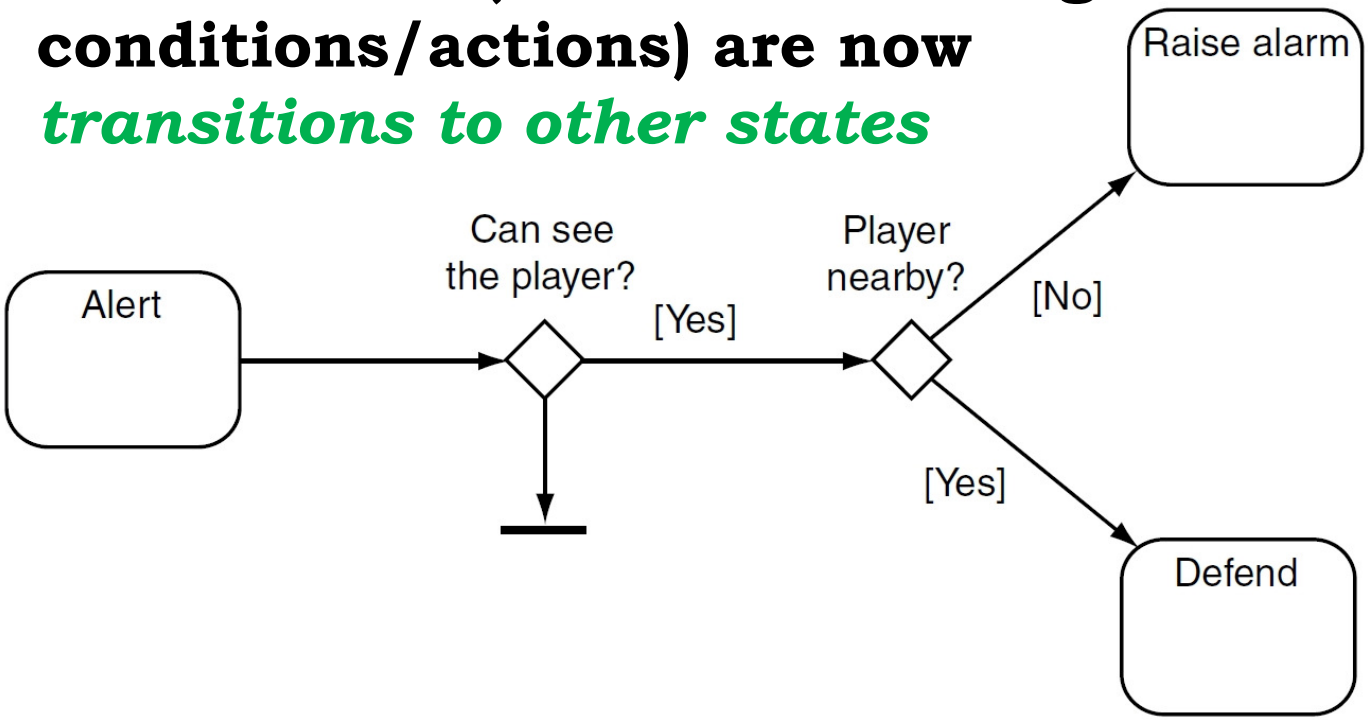


Hierarchical State Machines

- Refer to textbook for more details on its implementation
- **Performance**:
  - O(*n*) in memory (*n* is number of layers in hierarchy)
  - O(*nt*) in time, where *t* is the (average) number of transitions per state

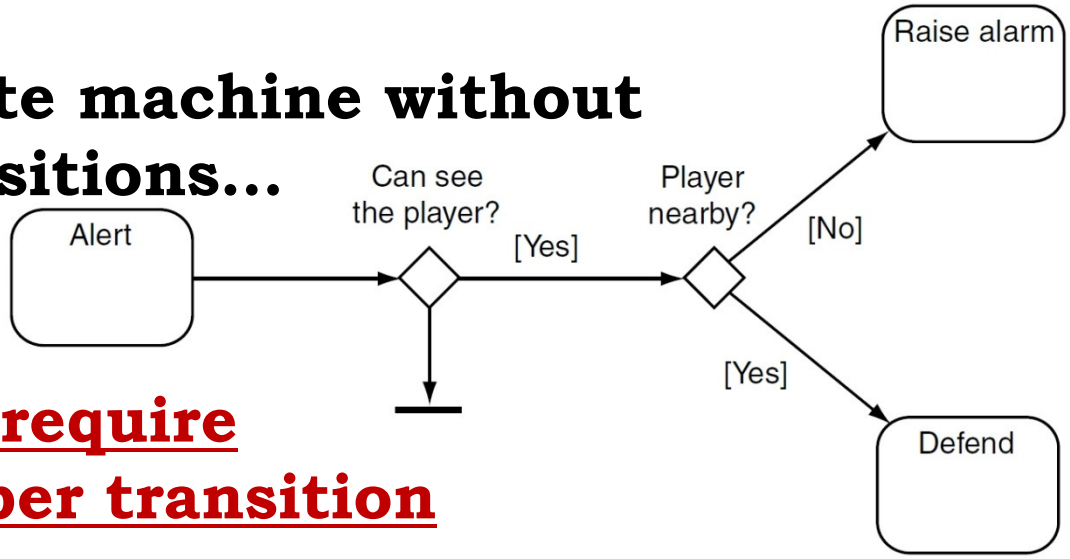
DT + SM

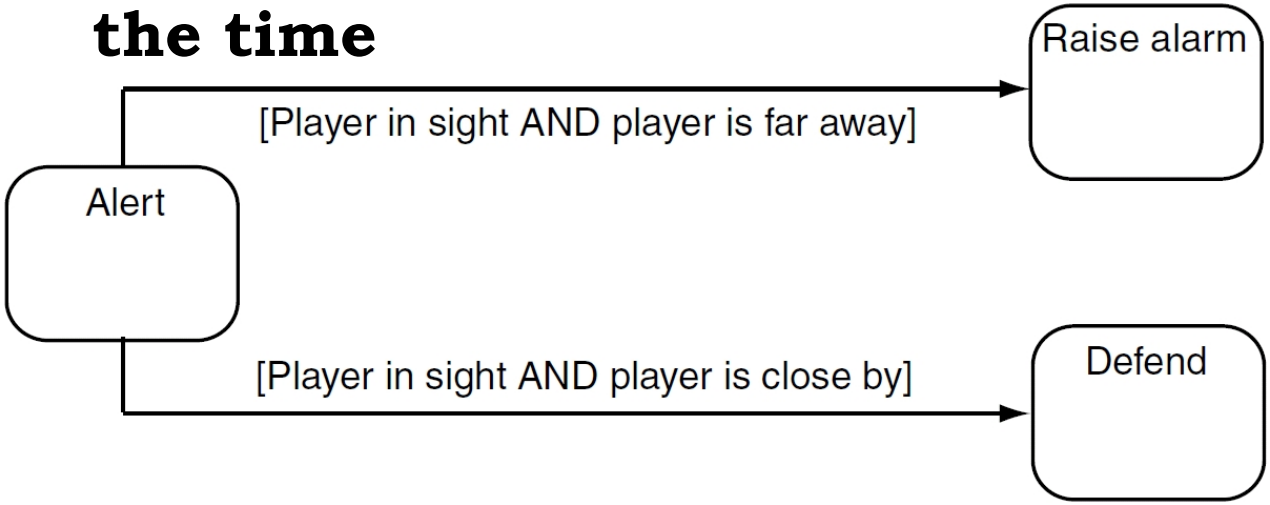
- **Combining decision trees and state machines**
  - **One approach: Replace transitions from a state *with a decision tree***
  - **Leaves of DT (rather than straightforward conditions/actions) are now *transitions to other states***



DT + SM

- **To implement state machine without decision tree transitions…**
  - **We *may need to model complex conditions* that <u>require more checking per transition</u>**
  - **May be time-consuming as need to check all the time**





Behaviour Trees

- Behaviour trees have become a popular tool for creating AI characters.
- They are a synthesis of a number of techniques that have been around in AI for a while:
  - Hierarchical State Machines, Scheduling, Planning, and Action Execution.
- Behaviour trees have a lot in common with Hierarchical SMs but, instead of a state, the main building block of a behaviour tree is a task.
- A task can be something *as simple as looking up the value* of a variable in the game state, or *executing an animation*.

Behaviour Trees

- **Tasks are composed into sub-trees to *represent more complex actions*. In turn, these complex actions can again be composed into higher level behaviours**
- **Because all tasks have a common interface and are largely self-contained, they *can be easily built up into hierarchies* (i.e., behaviour trees)**
- **Tasks in a behaviour tree all have the same basic structure. They are given some CPU time to do their thing, and when they are ready they *return with a status code* indicating either success or failure**

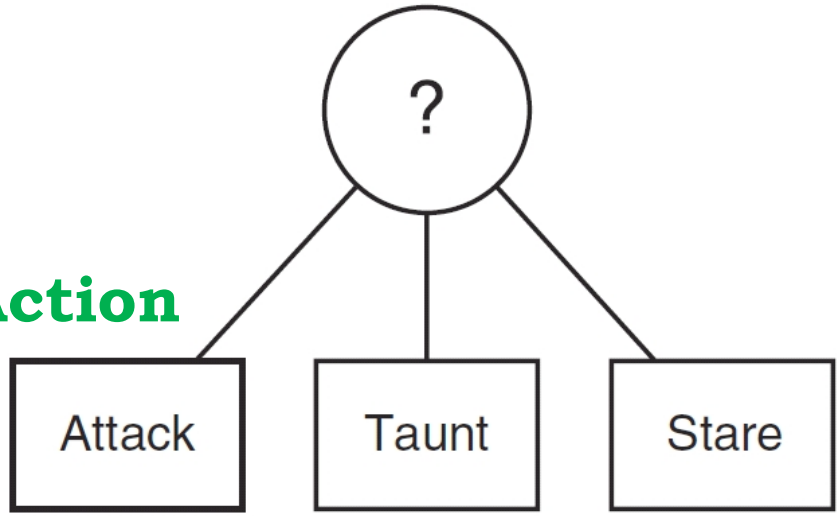
Behaviour Trees

- Our simple behaviour trees will consist of three kinds of tasks (nodes):
  - Conditions (leaf): test some property of the game.
  - Actions (leaf): alter the state of the game.
  - Composites (branch): keep track of a collection of child tasks.
- Consider two types of Composite tasks: Selector and Sequence. Both of these run each of their child behaviours in turn.

Behaviour Trees

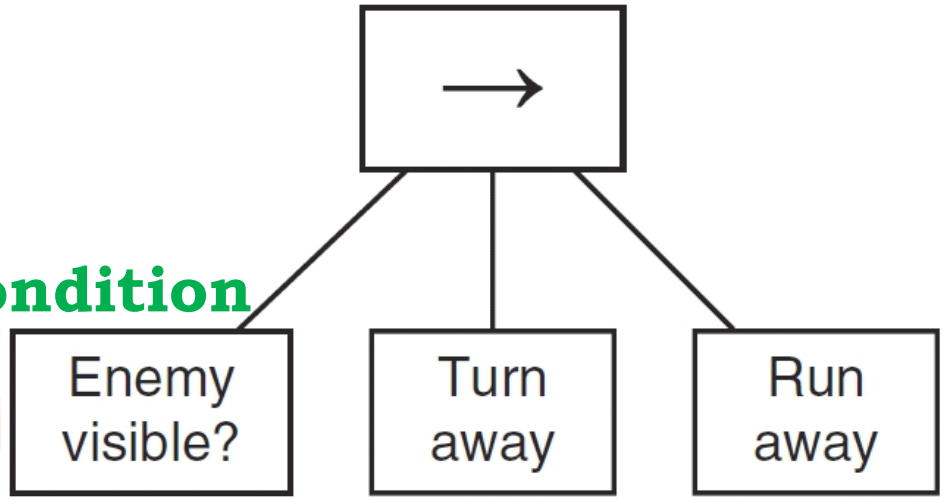
- **A Selector**
  - **returns a success status code *when one of its children runs successfully*.**
  - **As long as its children are failing, *it will keep on trying*. If it runs out of children completely, it will return *a failure status code*.**

An Action task



Behaviour Trees

- **A Sequence:**
  - **returns a failure status code *when one of its children fails*.**
  - **As long as its children are succeeding, it *will keep going*. If it runs out of children, it *will return in success*.**

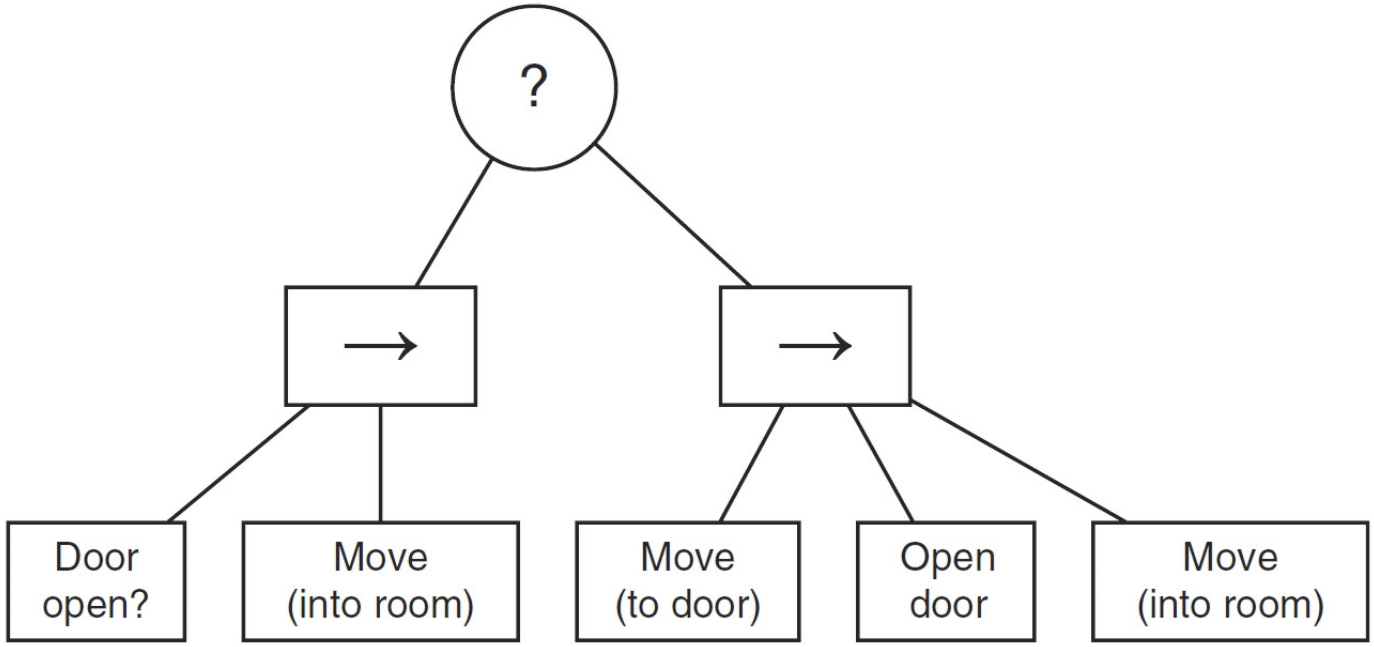


A Condition task

Behaviour Trees

Simple Example:

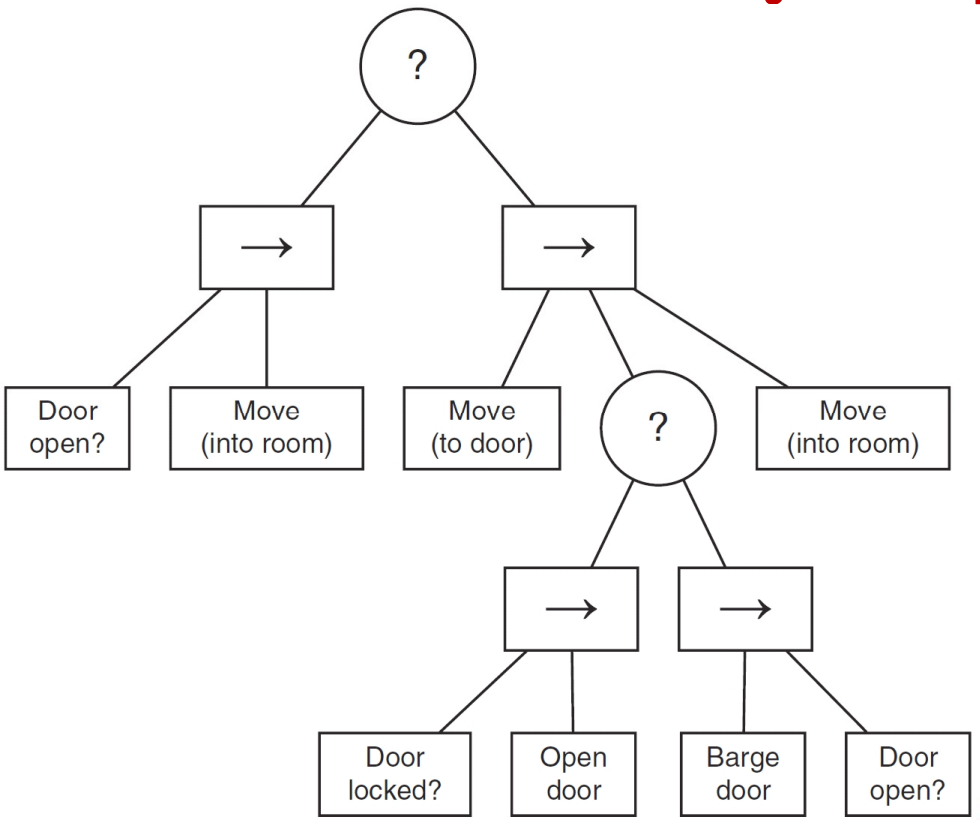
- **E.g., moving into a room.**
- **What if the door is open? Closed?**



Behaviour Trees

Another Example:

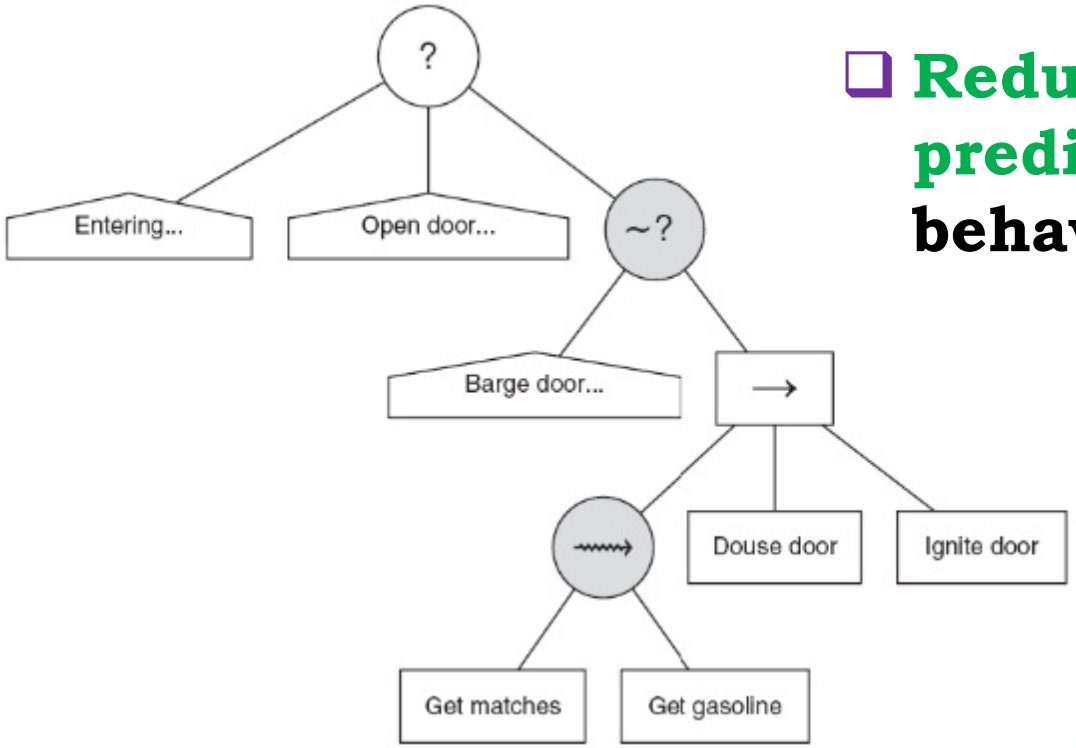
- **Behaviour tree for a minimally acceptable enemy**



Behaviour Trees

Randomness:

- Behaviour Trees with **Order Randomization** for some Sequencers and Selectors



- **Reduces the predictability of the behaviour**

Behaviour Trees/ Reactive Planning

- Behaviour trees implement a very simple form of planning, sometimes called reactive planning.
- Selectors allow the character to try things, and fall back to other behaviours if they fail. This isn't a very sophisticated form of planning.
- Nevertheless, even this rudimentary planning can give a good boost to the believability of your characters.
- Similar behaviour can be modelled using finite state machines, scripts, etc., but typically they are much harder.