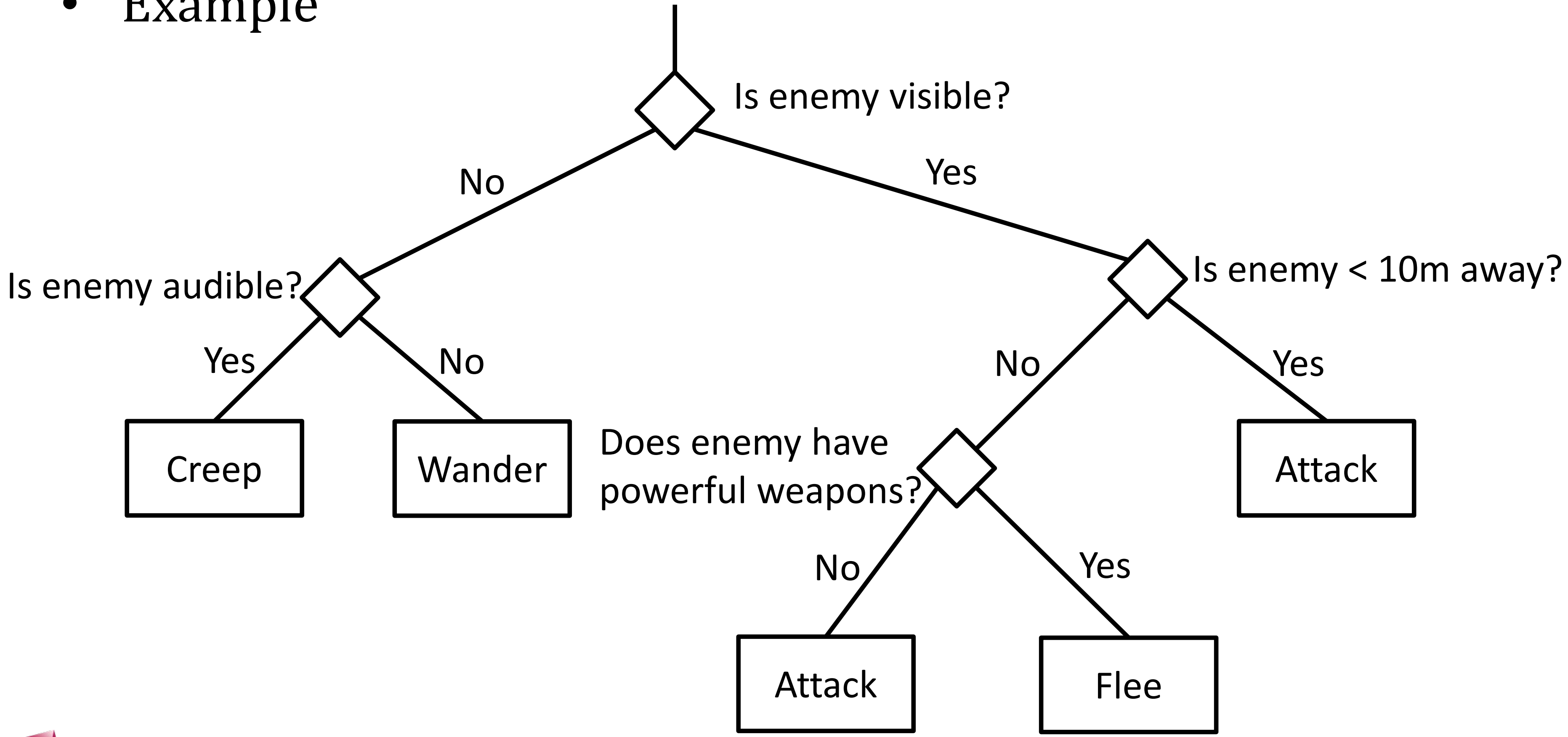
CS4386 AI Game Programming

Lecture 04
Decision Trees, Finite State Machines and Behavior Trees

Semester B, 2020-2021
Department of Computer Science
City University of Hong Kong

Decision Trees

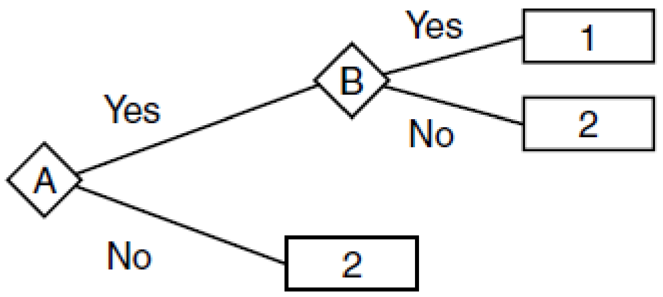
- Fast, easily implemented and simple to understand
- Example



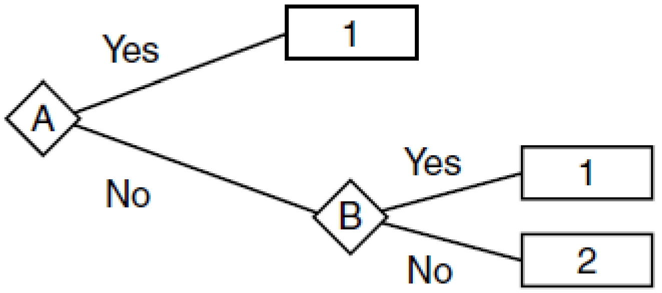
Decisions

- Decisions in a tree are simple
- They typically check a single value and do not contain any Boolean logic
- The tree structure can represent Boolean combinations of tests, e.g.,

If A AND B then action 1, otherwise action 2

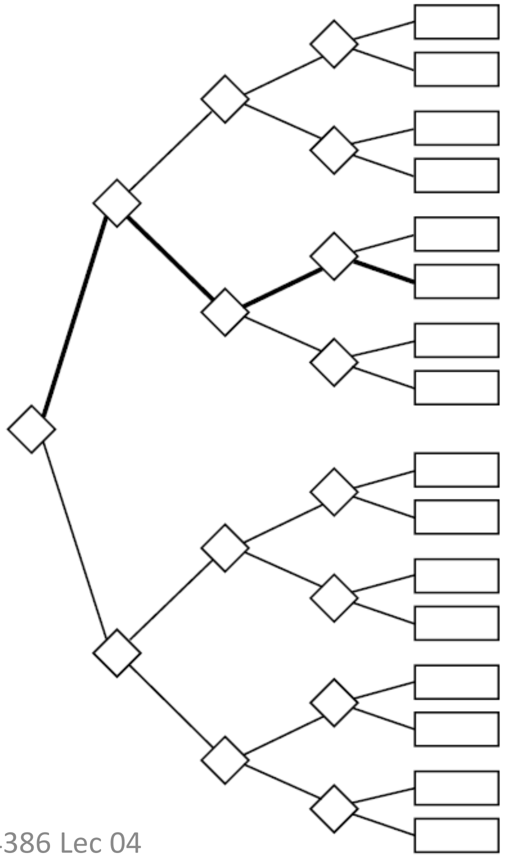


If A OR B then action 1, otherwise action 2



Decision Complexity

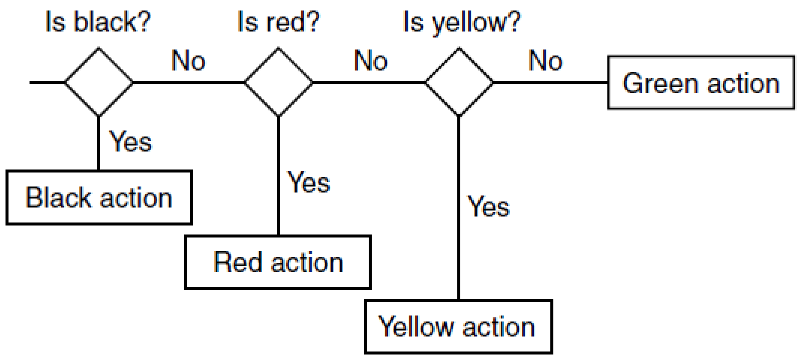
- The number of decisions that need to be considered is usually much smaller than the number of decisions in the tree, e.g.,

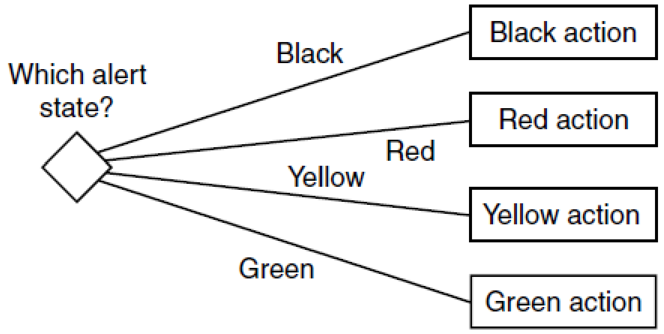


A decision tree with 15 decisions and 16 possible actions, but only 4 decisions are ever made

Branching

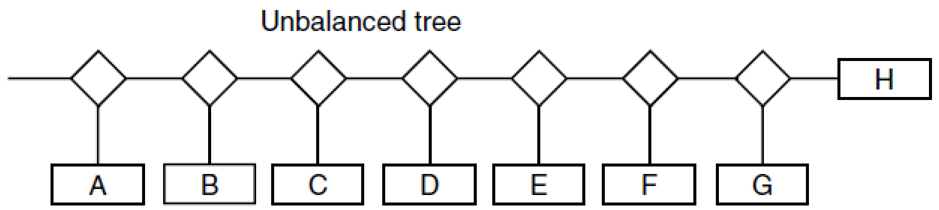
- Binary decision tree: 2 options at each branch
- Decision trees can have more than 2 branches at each decision point, giving a flatter structure
- It is more common to see decision trees using only binary decisions because
  - The underlying code for multiple branches usually simplifies down to a series of binary tests
  - It is easier to optimize binary decision trees and some learning algorithms only work with binary decision trees

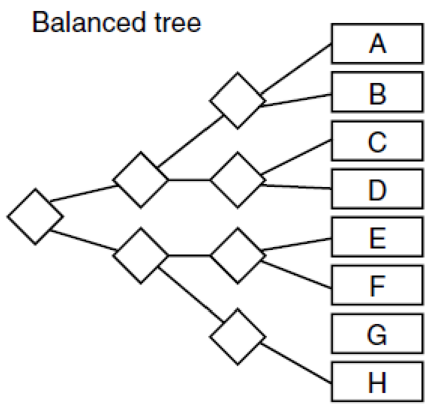




Balancing the Tree

- Decision trees are intended to run fast and are fastest when the tree is balanced
- A balanced tree has about the same number of leaves on each branch





What is the average number of decisions in each tree structure if all behaviors are equally likely?

Unbalanced Tree:

A: 1 decision	E: 5 decisions
B: 2 decisions	F: 6 decisions
C: 3 decisions	G: 7 decisions
D: 4 decisions	H: 7 decisions

Avg. num. of decisions
= (1+2+3+4+5+6+7+7)/8 = 4.375

Balanced Tree:

A: 3 decisions	E: 3 decisions
B: 3 decisions	F: 3 decisions
C: 3 decisions	G: 3 decisions
D: 3 decisions	H: 3 decisions

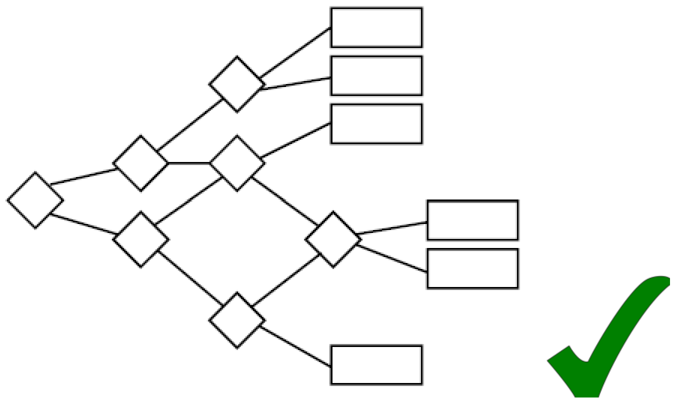
Avg. num. of decisions = 3

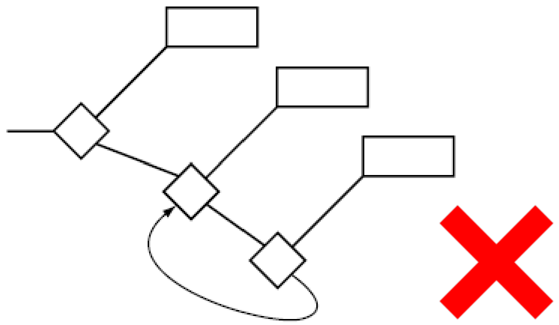
Performance

- Assuming that each decision takes a constant amount of time and that the tree is balanced
  - Performance is $O(\log_2 n)$ where n is the number of decision nodes in the tree
- With a severely unbalanced tree
  - Performance can become $O(n)$
- Although a balanced tree is theoretically optimal, in practice the fastest tree structure is more complex
  - In reality, the different results of a decision are not equally likely
  - Not all decisions are equal such that one decision may take more time to run than another decision
  - General guidelines: balance the tree, but make commonly used branches shorter than rarely used ones and put the most expensive decisions later

Beyond the Tree

- We can extend the tree to allow multiple branches into a new decision

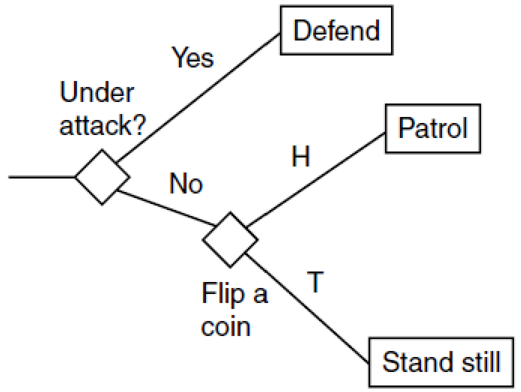




- You need to pay attention not to introduce possible loops in the tree, otherwise the decision process can loop around forever, never finding a leaf
- The valid decision structure is called a directed acyclic graph (DAG)

Random Decision Trees

- Often it is not desirable for the choice of behavior to be completely predictable
  - Some element of random behavior choice adds unpredictability, interest, and variation

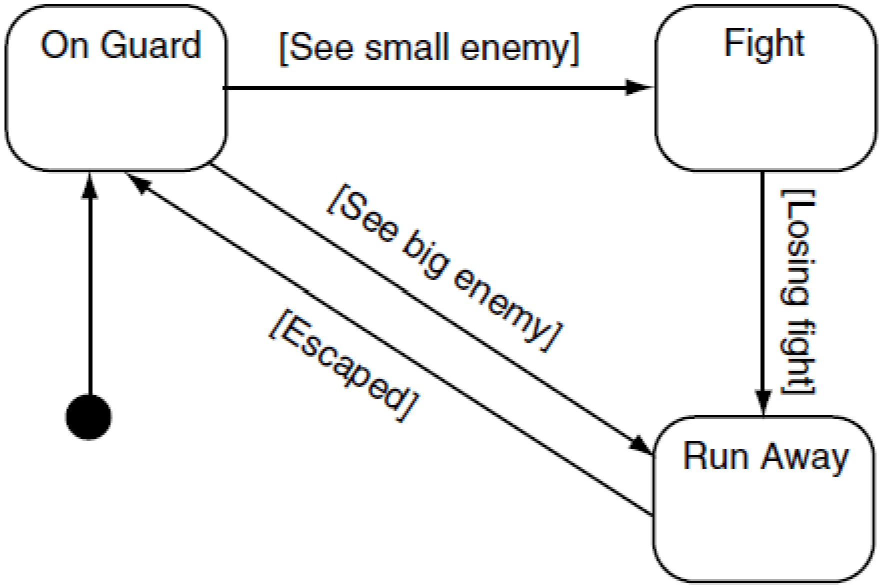


  - In the above example, as long as the agent is not under attack, the stand still and patrol behaviors will be chosen at random. If this choice is made at every frame, then the character will appear to vacillate between standing and moving, which would appear odd and unacceptable
  - To make the decision making process stable, it can keep track of what it did last time. When the decision is first considered, a choice is made at random, and the choice is stored. Next time the decision is considered, there is no randomness, and the previous choice is automatically taken. A time-out information can be further set to avoid the character repeating the same action forever

Finite State Machines (FSM)

- Each character occupies one state where actions or behaviors are normally associated. The character will continue carrying out the same action as long as the character remains in that state
- States are connected by transitions. Each transition has a set of associated conditions for changing states when triggered

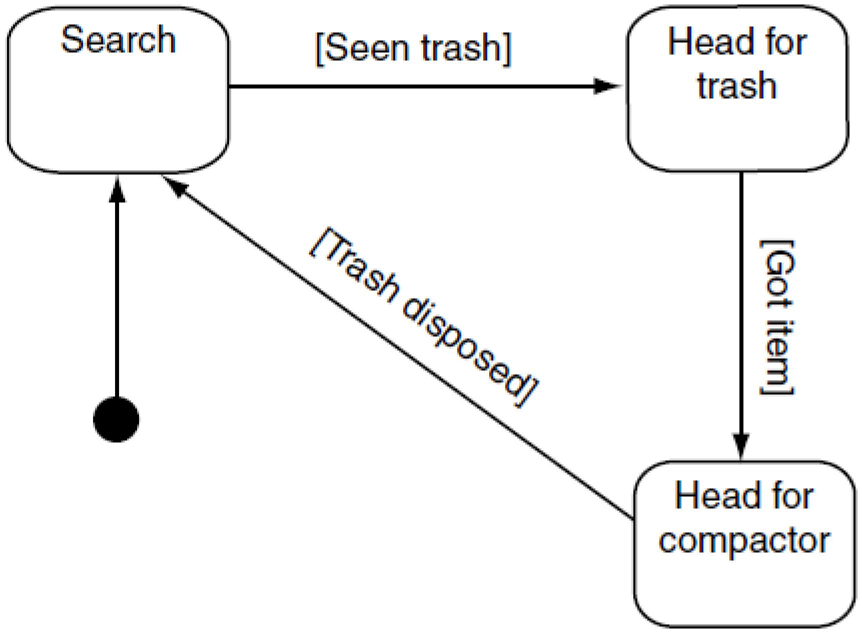
The transition has no trigger condition and points to the initial state that will be entered when the state machine is run



Conditions are contained in square brackets []

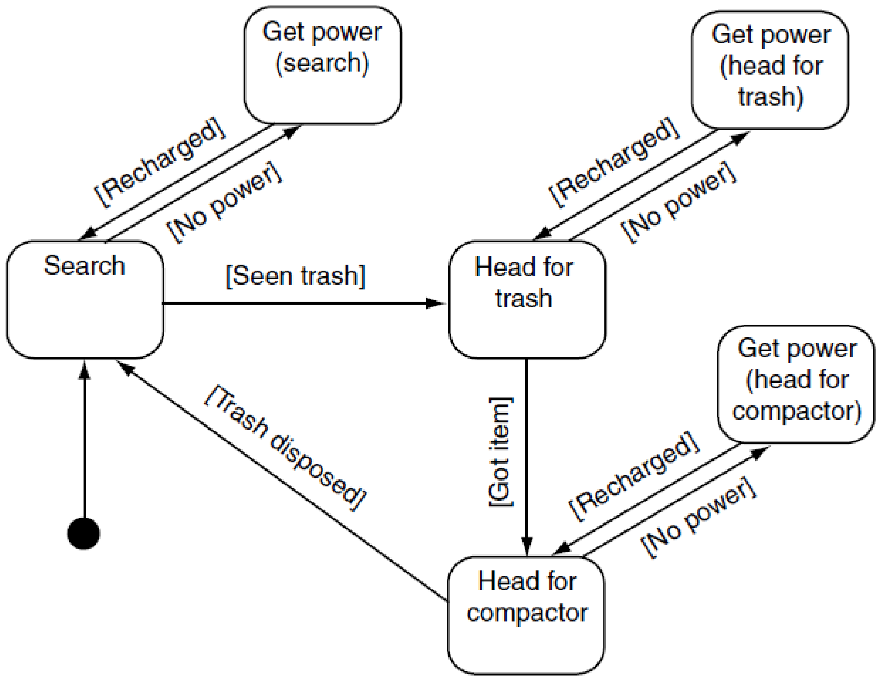
FSM Example 1

- Example 1: Implement the following scenario with a FSM
  - A service robot moves around a facility cleaning the floors. It might search around for objects that have been dropped, pick one up when it finds it, and carry it off to the trash compactor



FSM Example 2

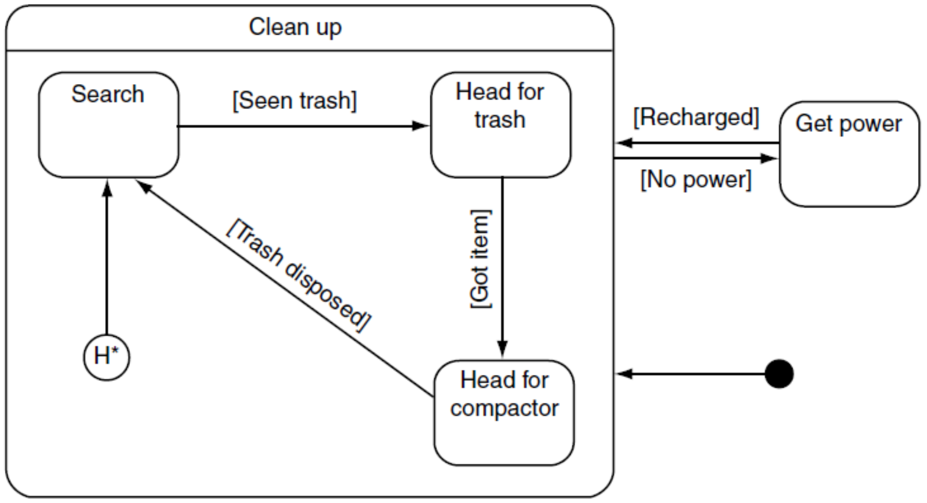
- Example 2: Implement with a FSM for the scenario in Example 1 with the following additional requirement
  - the robot can run low on power, whereupon it has to rush to the nearest electrical point and get recharged. Regardless of what it is doing at the time, it needs to stop, and when it is fully charged again it needs to pick up where it left off



The number of states is doubled when one level of alarm mechanism (interrupting normal behavior to respond to something important) is introduced
The FSM becomes even more complex when more levels of alarm are introduced!

Hierarchical State Machines

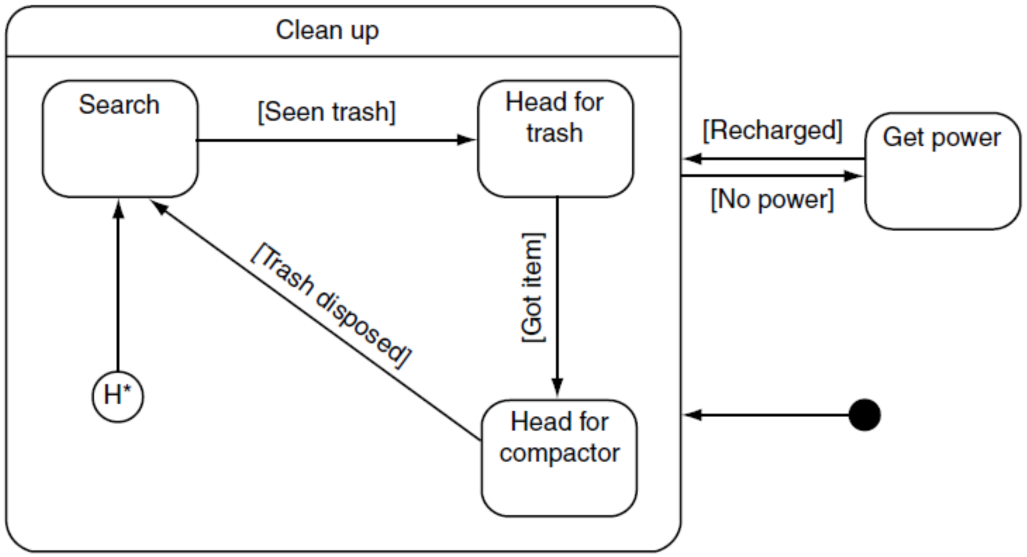
- Rather than combining all the logic into a single state machine, we can separate it into several levels of state machines
  - Each alarm mechanism has its own state machine, along with the original behavior
  - They are arranged in a hierarchy, so the next state machine down is only considered when the higher level
- Same example implemented with hierarchical state machine:



Hierarchical State Machines Explained

- A FSM can be included as a node in another FSM (thus hierarchical)
- The highest state in the hierarchy is always in control

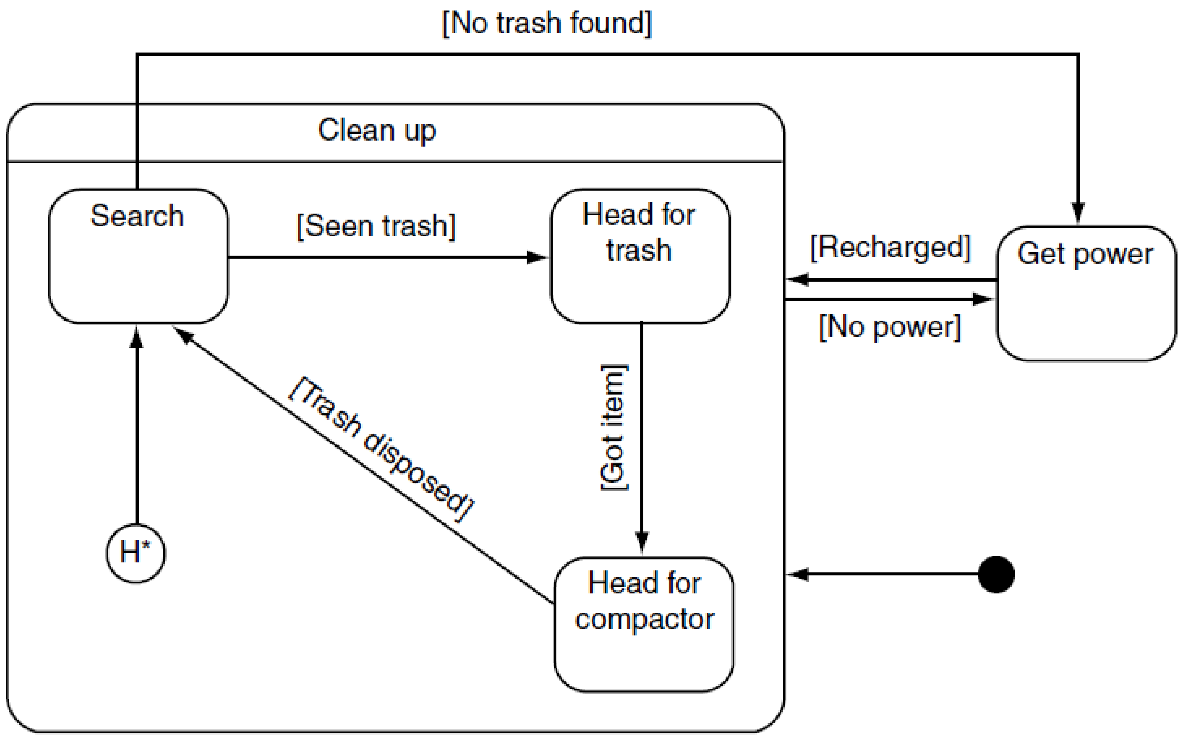
When a composite state is first entered, the circle with H* inside it indicates which sub-state should be entered.



If the composite state has already been entered, then the previous sub-state is returned to

Hierarchical State Machines Example 2

- Most hierarchical state machines, however, support transitions between levels of the hierarchy
  - let's add one more requirement such that if there are no objects to collect, the robot will use the opportunity to go and recharge, rather than standing around waiting for its battery to go flat



As we transition directly out of the state, the inner state machine no longer has any state
The robot must start the state machine again from its initial node, i.e., H* node when it is back to this inner state machine

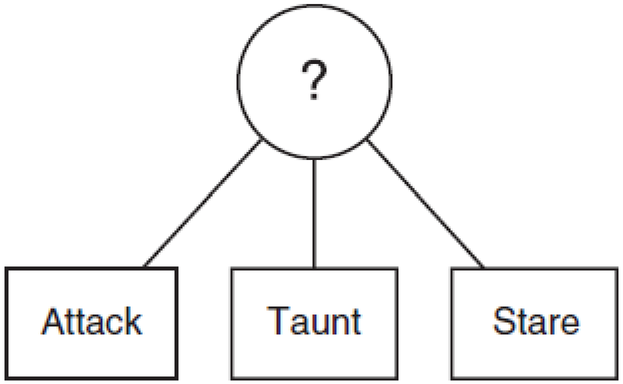
Behavior Trees

- Behavior trees have a lot in common with Hierarchical State Machines but, instead of a state, the main building block of a behavior tree is a task
- Tasks in a behavior 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
- Types of Tasks
  - Conditions: test some property of the game, e.g., tests for proximity (is the character within X units of an enemy?)
  - Actions: alter the state of the game, e.g., character movement, playing audio, etc.
  - Composite nodes: keep track of a collection of child tasks (conditions, actions, or other composites)

Both Conditions and Actions sit at the leaf nodes of the tree

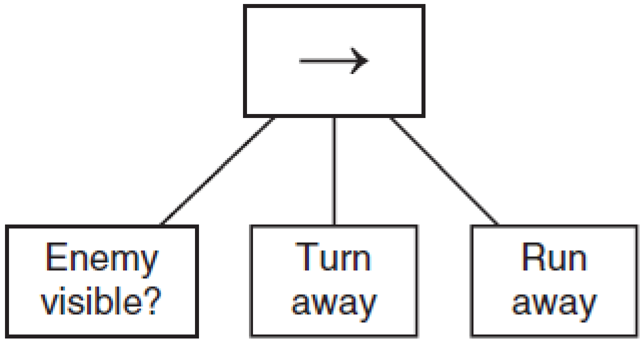
Composite Node: Selector

- A Selector will return immediately with 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
- Selectors are used to choose the first of a set of possible actions that is successful
- If we exhaust all options without success, then the Selector itself has failed



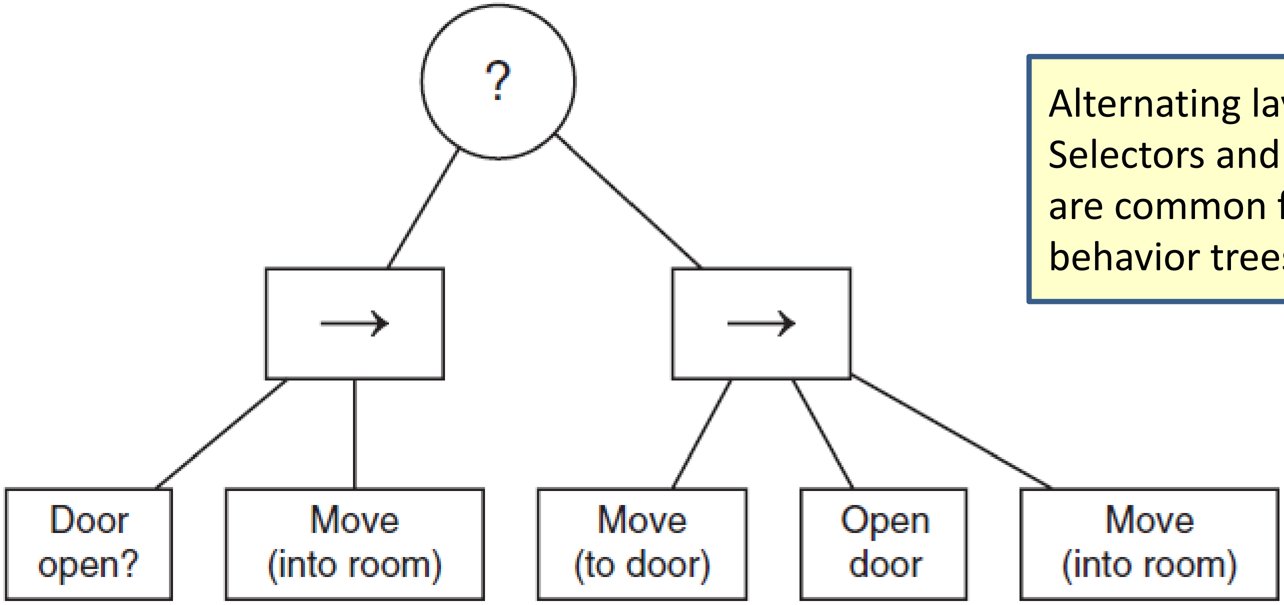
Composite Node: Sequence

- A Sequence will return immediately with 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
- Sequences represent a series of tasks that need to be undertaken
- Only if all the tasks in the Sequence are successful can we consider the Sequence as a whole to be successful



Composite Nodes Example 1

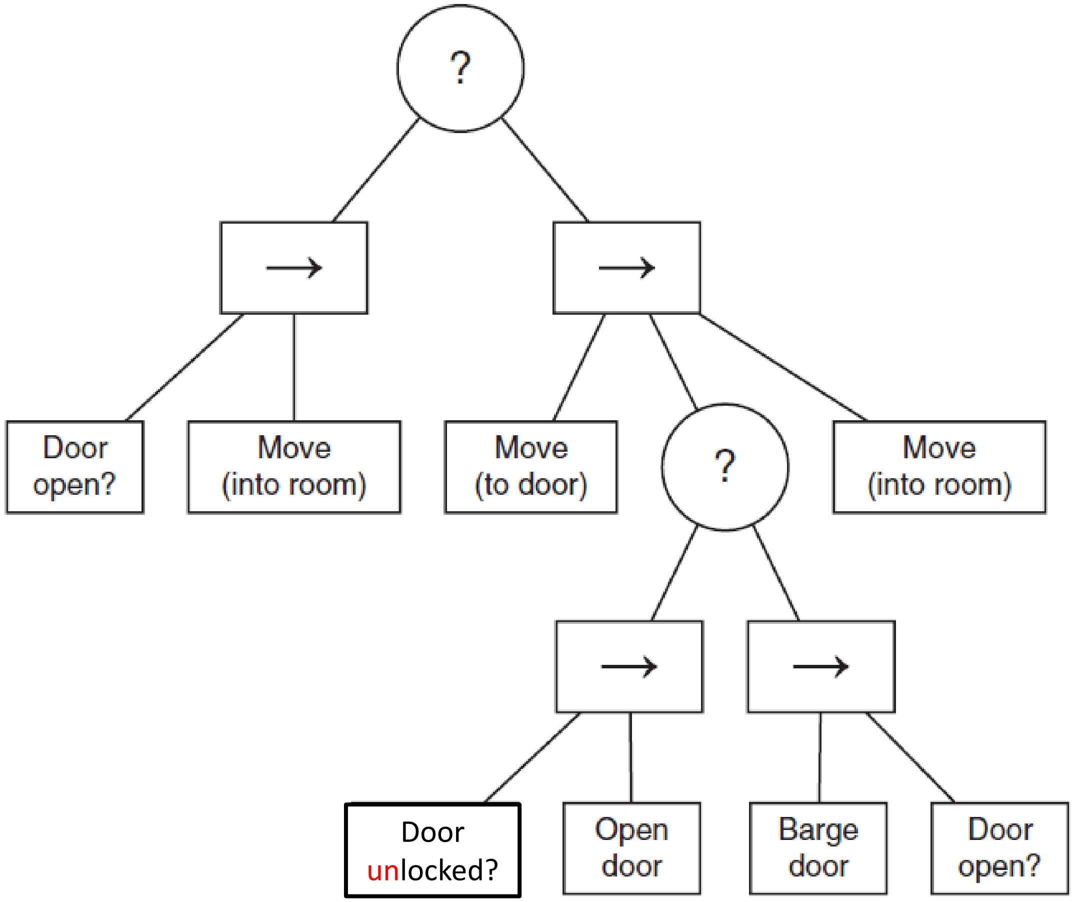
- Consider what will happen if the door is open and if the door is closed



Alternating layers of Selectors and Sequences are common features of behavior trees

Composite Nodes Example 2

- Consider what will happen if the door is open, if the door is closed and unlocked, and if the door is closed and locked



Non-Deterministic Composite Tasks

- Sometimes actions in a sequence may not need to be performed in a particular order and some actions in a selector can be tried in different orders
- To make the AI less predictable, we can use variations of Selectors and Sequences that can run their children in a random order
- These kinds of constraints are called “partial-order” constraints in the AI literature
  - Some parts may be strictly ordered, and others can be processed in any order

Simple Implementation

- The simplest would be a Selector that repeatedly tries a single child

```
class RandomSelector (Task):
    children
    def run():
        while True:
            child = random.choice(children)
            result = child.run()
            if result:
                return True
```

Problems:

1. it may try the same child more than once, even several times in a row
2. it will never give up, even if all its children repeatedly fail

Shuffling Procedure

- A better approach would be to walk through all the children in some random order for either Selectors or Sequences

```
class NonDeterministicSelector (Task):

    children

    def run():
        shuffled = random.shuffle(children)
        for child in shuffled:
            if child.run(): break
        return result
```

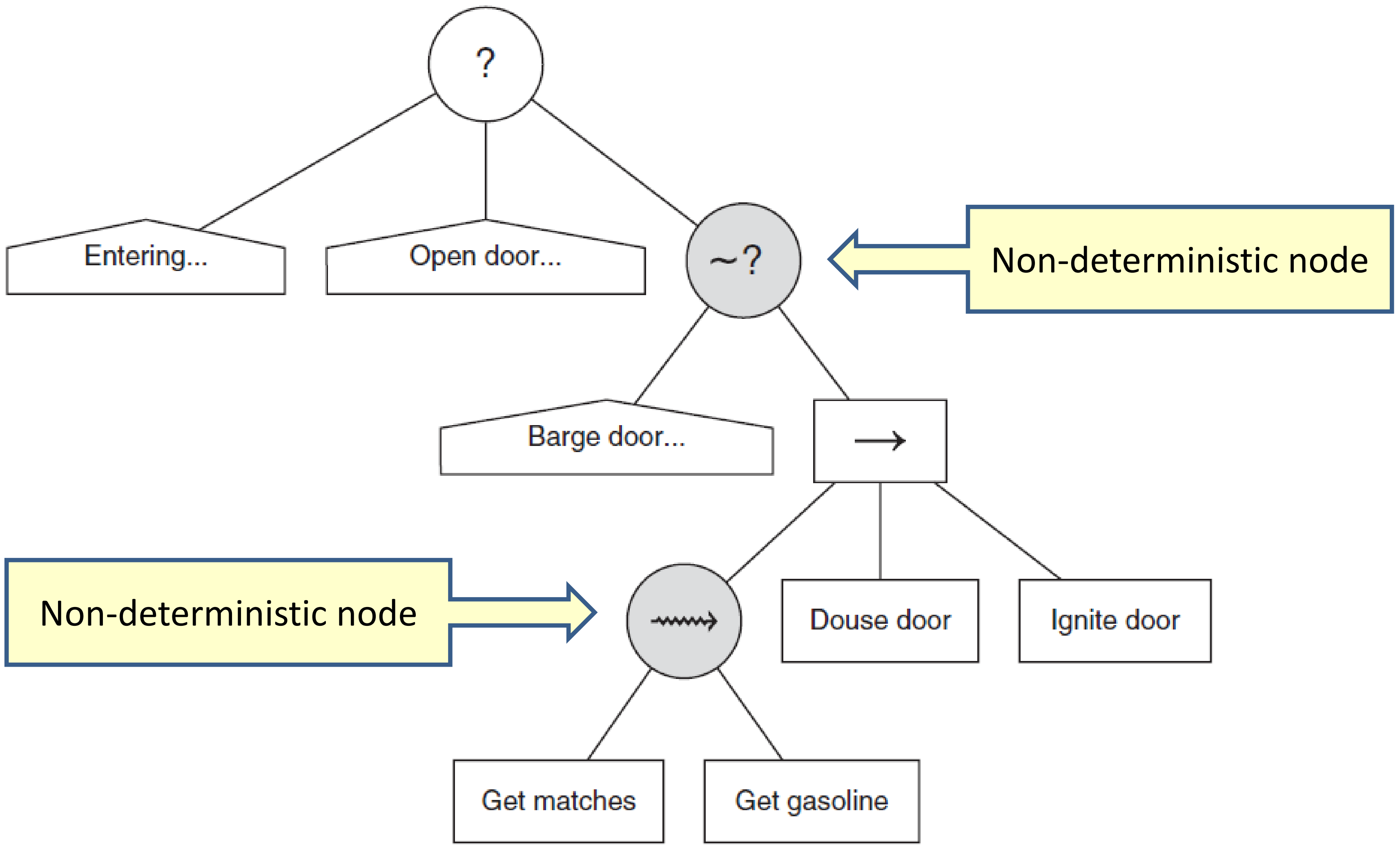
```
class NonDeterministicSequence (Task):

    children

    def run():
        shuffled = random.shuffle(children)
        for child in shuffled:
            if not child.run(): break
        return result
```

In each case, just add a shuffling step before running the children
This keeps the randomness but guarantees that all the children will be run and that the node will terminate when all the children have been exhausted

Behavior Tree Example with Partial Ordering



Reference

- Artificial Intelligence for Games
  - Chapter 5.2 - 5.4