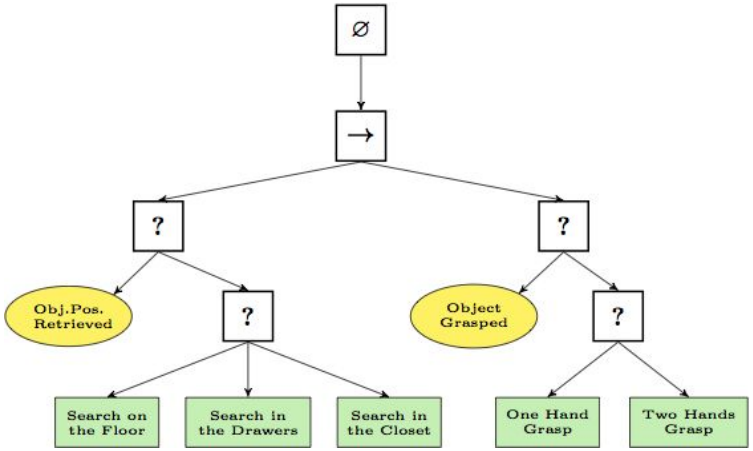
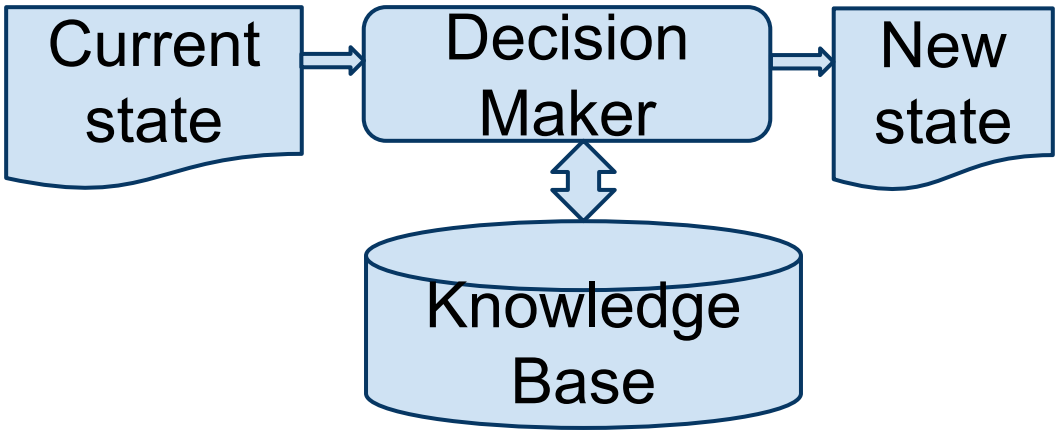
CS380
Artificial Intelligence for Games

Behavior Tree



Decision Making

- Is regarded as a process resulting in the selection of a course of **action** among several possible alternative options
- Is a reasoning process based on assumptions of values, preferences and beliefs of the **decision maker (DM)**
- Every decision making process produces a final choice, which may or may not prompt action
- DM is a part of games. Some game engine include a game independent DM with a dependant Knowledge Base



Knowledge for Decision Making

- Procedural
  - Using Scripting
- Decision Tree
- State Tree
  - Finite-State Machine
- Behavior Tree
  - As a combination of all above
- And more
  - Rule based, ...

Decision Making With Procedure

- Example:

```
if (isVisible(evemy))
    if (close(evemy))
        attack(evemy);
    else if (flank(evemy))
        moveto(evemy);
    else
        attack(evemy);
else
    if (audible(evemy))
        creep(evemy);
```

Decision Making With Procedure (contd)

- More example:
  - Path Searching
- Procedures can be
  - hard-coded into the game program,
  - dynamically linked as plug-in or add-on
    - extensions that extends the usability of the program
  - scripted to be parsed and executed in the game at run-time

Scripting

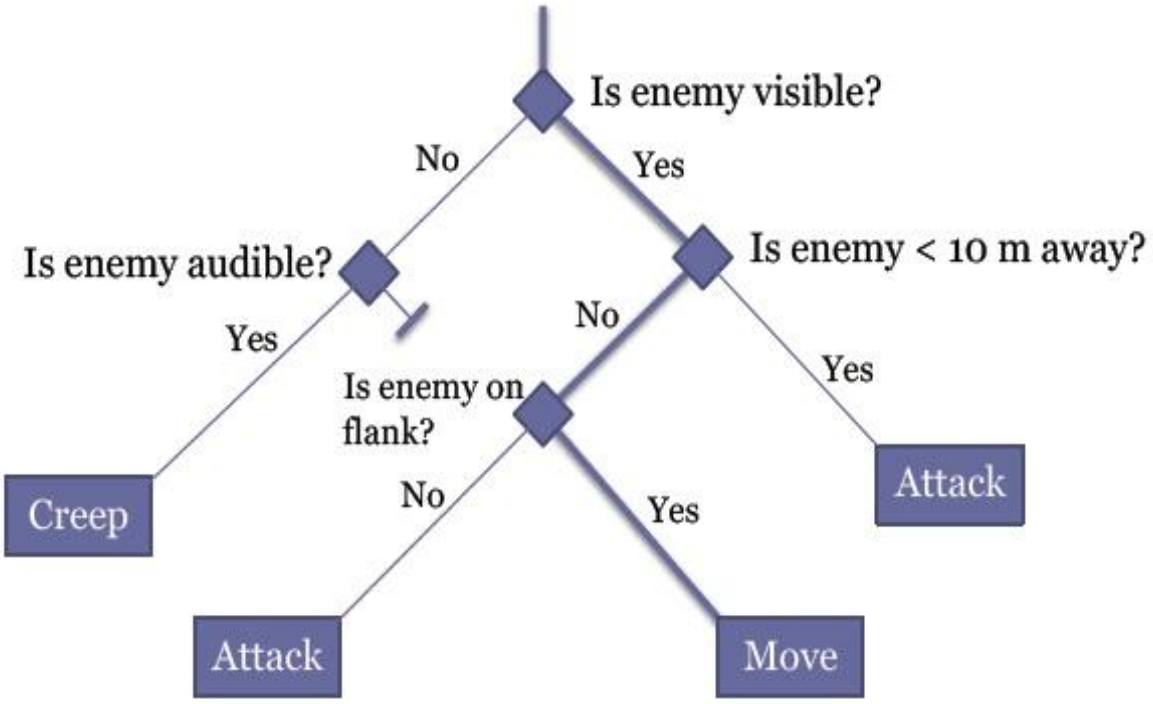
- Scripts can be embedded within the native code of the game and serve to extend its set of available functions and types after the game is created and on market
  - Scripts are often created or at least modified by the end-user
  - Ultimate use of scripts can significantly alter the game
- Most popular scripting languages:
  - Python
  - Lua
  - AngelScript
  - Perl

Making Decision Using Decision Trees

- A decision tree is the most basic technique to present knowledge base for making decision using a graphical notation
- A decision tree is a tree-like graph of decisions and their possible consequences
- Pros:
  - Easy to implement
  - Fast execution
  - Simple to understand

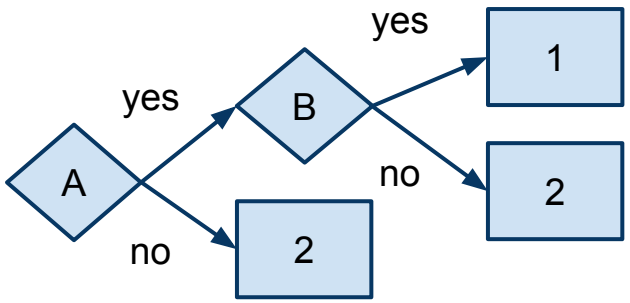
Decision Trees Example

- This decision tree is just a series of questions that help a soldier decide what to do given its current situation

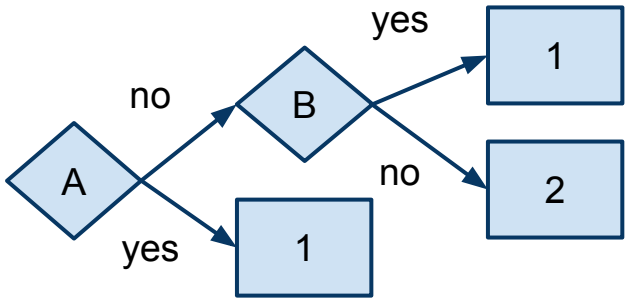


Decision Tree Construction

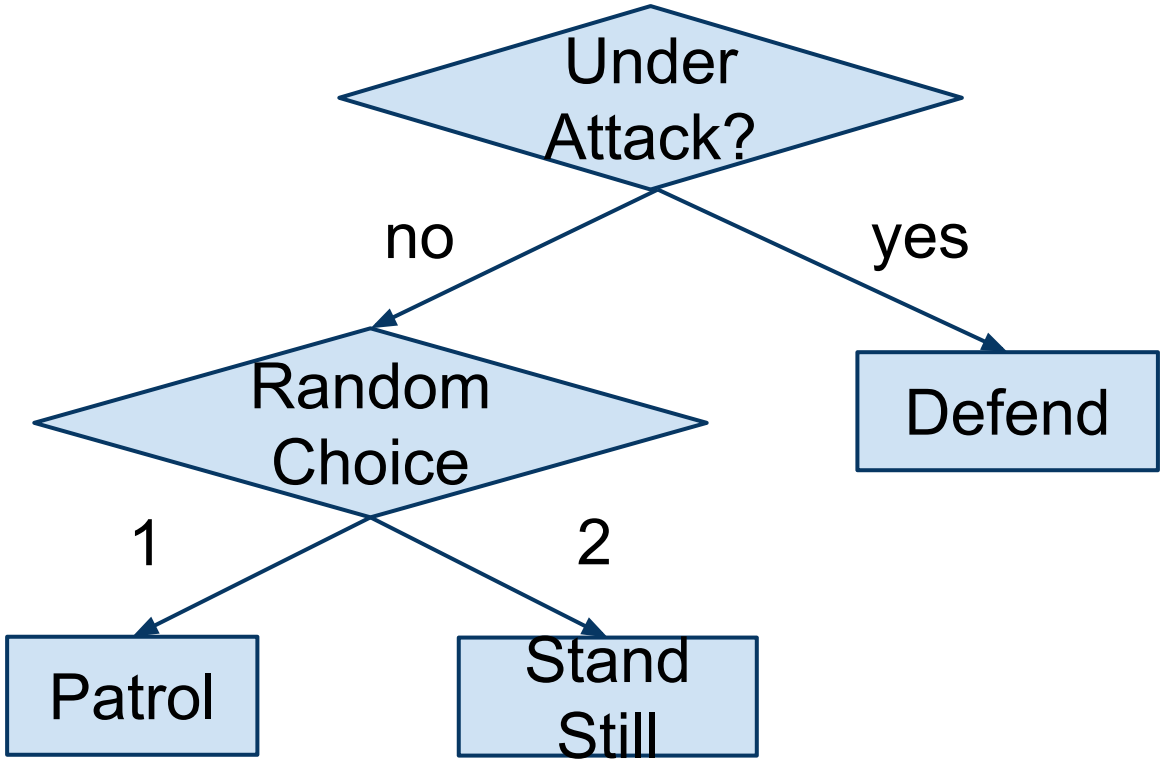
- **If** A AND B
  then
  state 1
  otherwise
  state 2



- **If** A OR B
  then
  state 1
  otherwise
  state 2

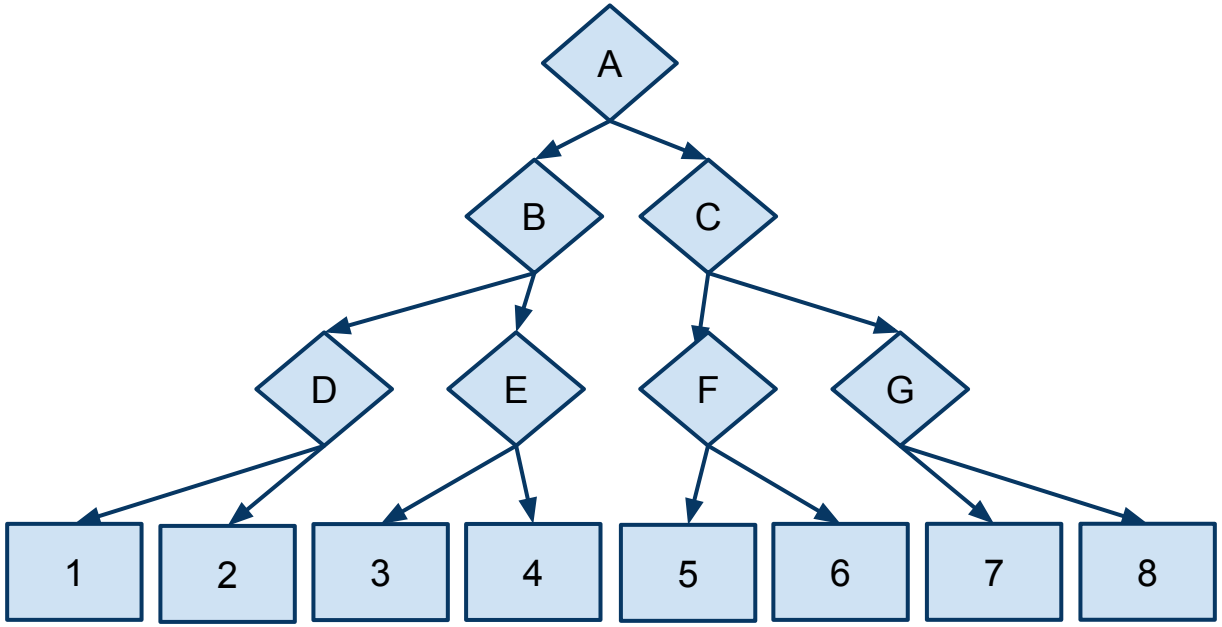


Decision Trees with Random Choice



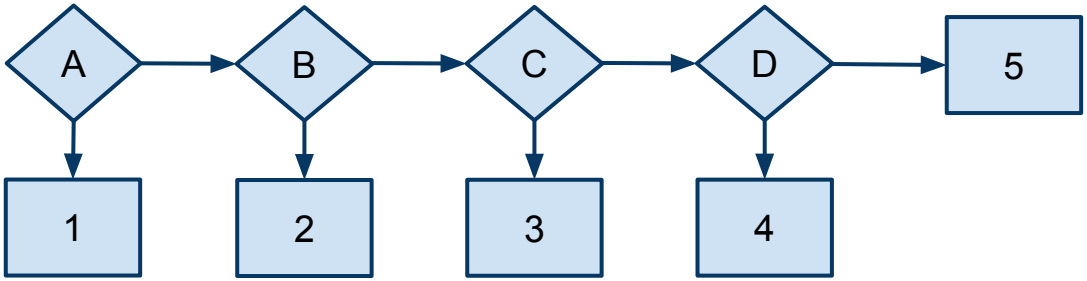
Balanced Decision Trees

- DT is balanced when no leaf (terminal node) is much farther away from the root than any other leaf
- When balanced the search time is O(lg(n))



Unbalanced Decision Trees

- Example:



  - Leaf 5 much farther away from A than 1 or 2

Decision Tree Properties

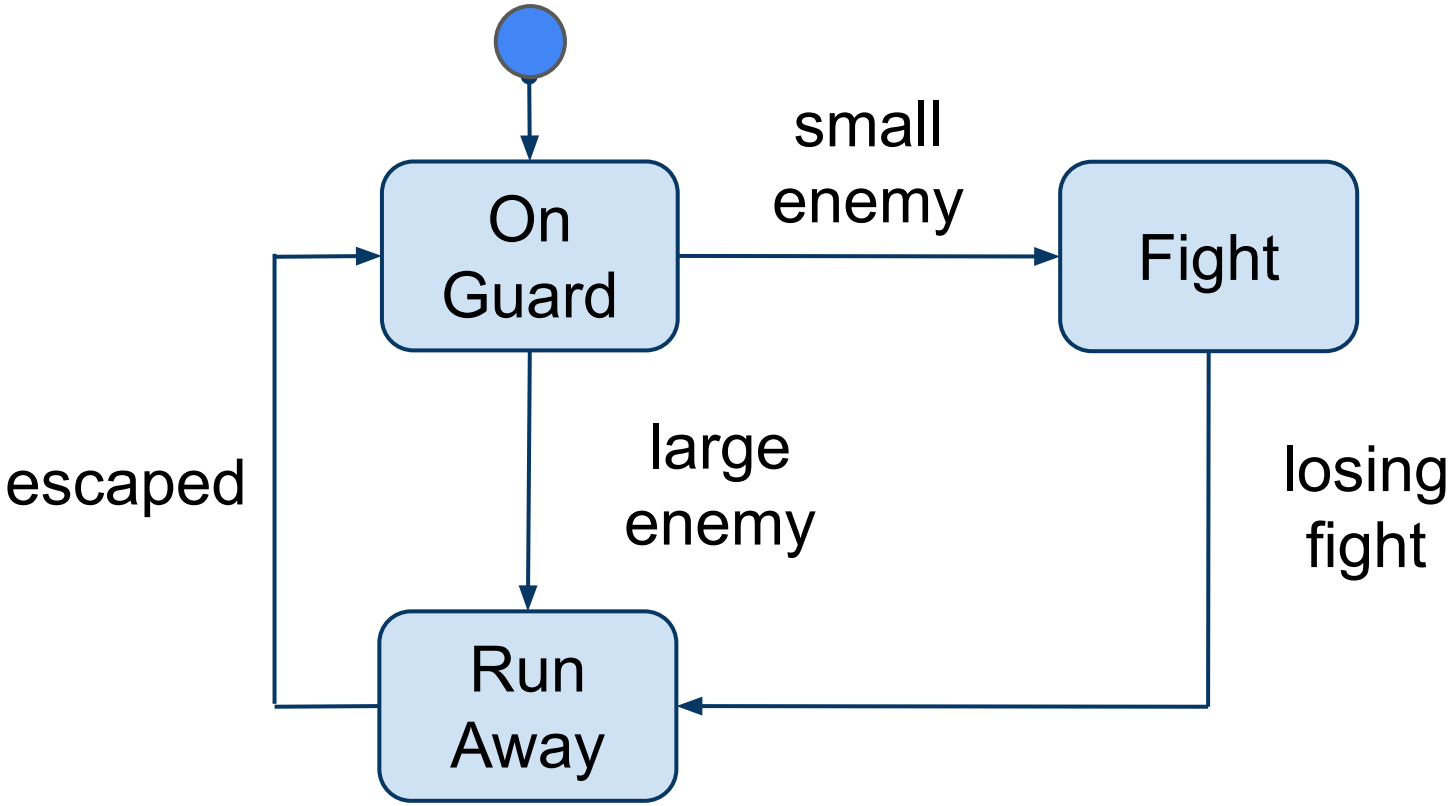
- In a balanced tree with depth d, b choices, and n states
  - $n = b^d$ states
  - A decision requires $\log_b(n) = d$ evaluations - O(d) time
  - The entire tree requires O(n) memory

- In an unbalanced tree
  - A decision could require O(n) evaluations/time

Finite-State Machine

- An abstract machine that can be in only one of a finite number of states at a time
- The state it is in at any given time is called the **current state**
- It can change from one state to another when initiated by a triggering event or condition, this is called a **transition**
- At each iteration, the state machine's **update** function is called and checks if any transition from the current state occurred
- Transitions can have **priorities** and the higher in priority is called

Finite-State Machine (contd)



Finite-State Machine Implementation

```
class Soldier {

public:

   enum State { GUARD, FIGHT, RUN_AWAY };

   State currentState;

   void Update();

}
```

Finite-State Machine Implementation (contd)

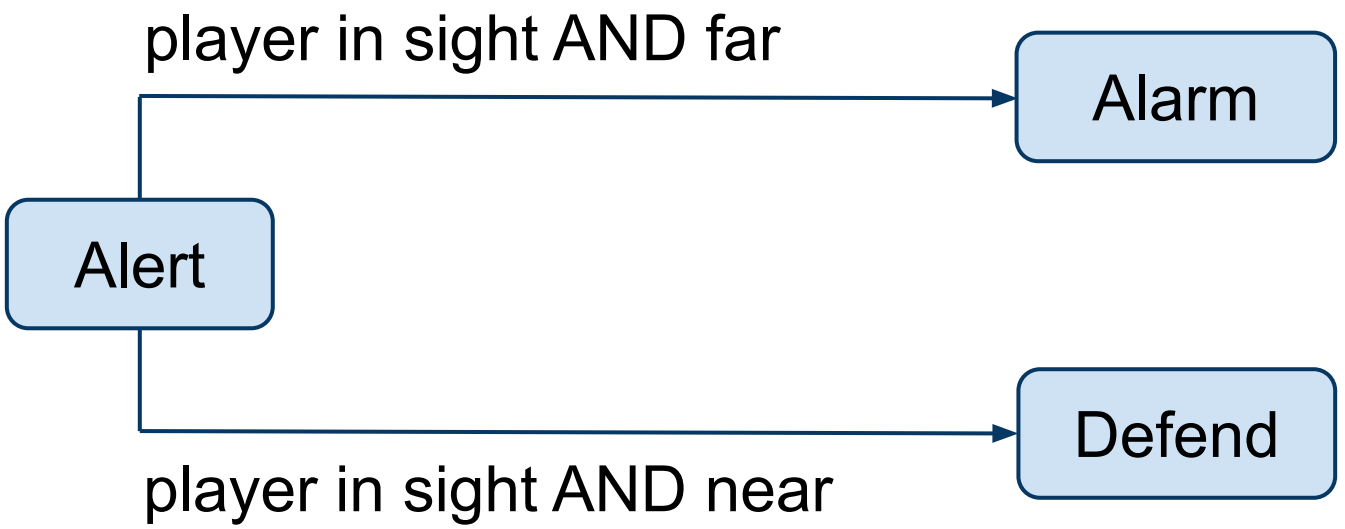
```
void Soldier::Update() {
  if (currentState == GUARD) {
    if (small_enemy()) { currentState = FIGHT;
                         startFighting(); }
    else if (big_enemy()) { currentState = RUN_AWAY;
                            startRunningAway(); }
  } else if (currentState == FIGHT) {
    if (losing_fight()) { currentState = RUN_AWAY;
                          startRunningAway(); }
  } else if (currentState == RUN_AWAY) {
    if (escaped()) { currentState = GUARD;
                     startGuarding(); }
  }
}
```

Finite-State Machine
Pros & Cons

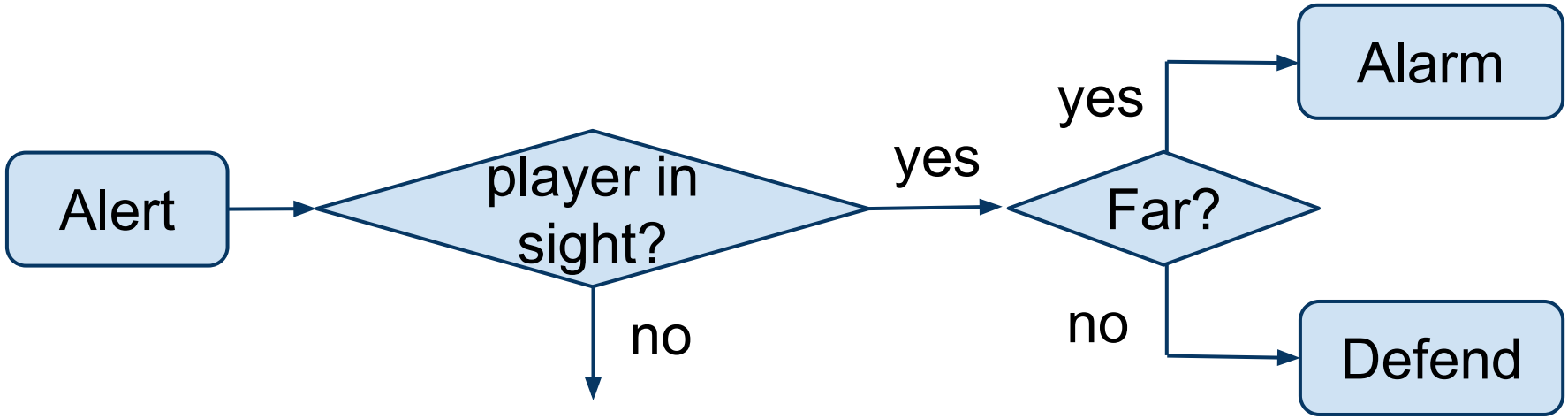
- Easy to write, but difficult to maintain
- Fast to implement for all but huge state machines

Combining Decision Trees & State Machines

- Why need combining?
  - To avoid duplicating expensive tests in state machines, like "player in sight" below



Combining Decision Trees & State Machines (contd)



Behavior Tree

- BT is a synthesis of a number of techniques
- If you have a finite set of tasks, then a BT is where you **specify how the program switches from one task to another**
- BT were initially used to develop video games **to control the behaviors** (AI) of NPC
  - Some examples of computer games that use behavior trees are:
    - Pacman
    - Halo. BT becomes important since Halo 2 (2004)
    - Spore
    - Bioshock
- Now BT also widely used in robotics

Behavior Tree

- Behavior trees became popular for their development paradigm:
  - being able to create a complex behavior by only programming the NPC's actions and then designing a tree structure (usually through drag and drop) whose leaf nodes are actions and whose inner nodes determine the NPC's decision making
- Behavior trees are visually intuitive and easy to design, test, and debug, and provide more modularity, scalability, and reusability than other behavior creation methods

Key concepts

- A behavior tree is graphically represented as a directed tree in which the nodes are classified as root, control flow nodes, or execution nodes (tasks)
  - For each pair of connected nodes the outgoing node is called parent and the incoming node is called child
  - The root has no parents and exactly one child, the control flow nodes have one parent and at least one child, and the execution nodes have one parent and no children
  - Graphically, the children of a control flow node are placed below it, ordered from left to right

Execution

- The execution of a behavior tree starts from the root followed by execution of all children, some of them, or just one child
- When the execution of a node in the behavior tree is allowed, it returns to the parent a status Success if it has achieved its goal or Failure otherwise

Tasks

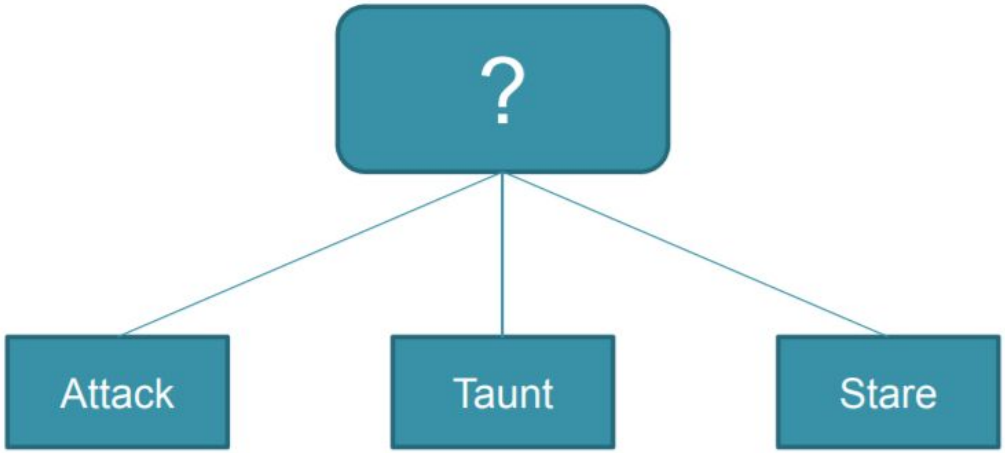
- Simple
  - Condition tests
    - Test some property of game
      - Proximity, line of sight, state of character, ...
      - Usually implemented in a parameterized task
- Complex
  - Actions
    - Alter state of game
      - Animation, character movement, change of internal state, audio, ...
- Composite
  - groups of tasks that executed in controlled manner

Control flow node

- A control flow node is used to control the subtasks of which it is composed
- A control flow node may be either a **selector** node or a **sequence** node
- They run each of their subtasks in turn
- When a subtask is completed and returns its status (Success or Failure), the control flow node decides whether to execute the next subtask or not

Selector

- Selectors are used to find and execute the first child that does not fail
- A selector will return immediately with a status code of Success when one of its children returns Success

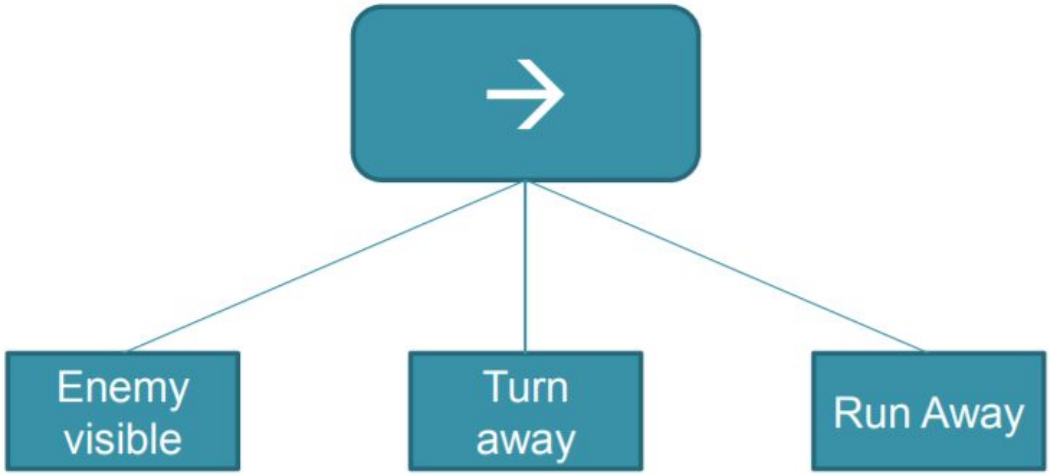


Pseudocode

```
class Selector extends Task {

  ...

  run() {

    this.state = State.Failure;

    for (var task of this.tasks) {

      this.state = task.run().state;

      if (this.state == State.Success) break;

    }

    return this;

  }

}
```

Sequence

- Sequence nodes are used to find and execute the first child that has not yet succeeded
- A sequence node will return immediately with a status code of Failure when one of its children returns Failure

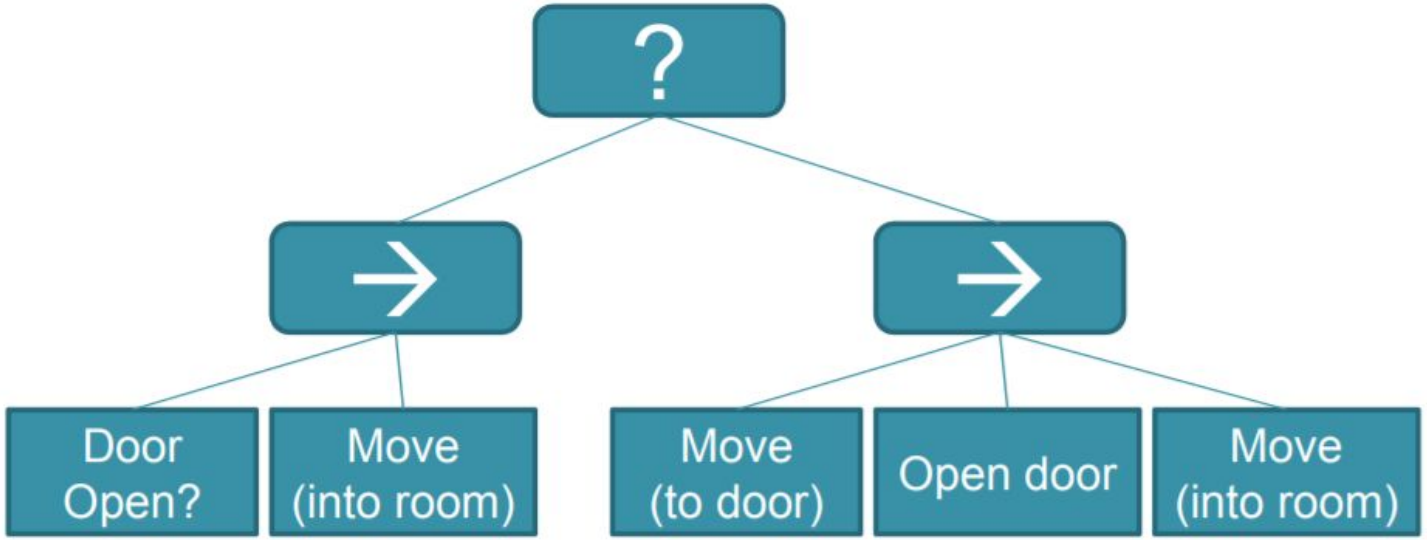


Pseudocode

```
class Sequence extends Task {

  ...

  run() {

    this.state = State.Success;

    for (var task of this.tasks) {

      this.state = task.run().state;

      if (this.state == State.Failure) break;

    }

    return this;

  }

}
```

Example

Entering a room opening a door if necessary



Parallel

A node that runs all its children at the same time and returns Success if all or some of its children returns Success

Decorator

A node that modifies the returned status of its child

Blackboard

- Tasks need to have access to global data
- Passing as parameters generates a difficult API
- Use blackboard data structure
  - Write data and messages into a common, global storage structure