Introducción a los comportamientos inteligentes

Referencias: Ian Millington and John Funge. *Artificial Intelligence for Games*, Morgan Kaufmann, 2009. (Capítulo 5)

¿Qué parte de un videojuego es IA?

- Todo lo que no sea gráficos (sonidos) o redes (según un profesor de IA ☺)
  - O físicas (Aunque a veces puede estar incluida)
  - Normalmente en los NPCs (non-player characters)
  - Aunque a veces puede ser más general:
    - Juegos estilo "*Civilization*" (simulaciones sofisticadas)
    - Control de la gestión del contenido.

“Niveles” de IA en Videojuegos

- *Básico*
  - Técnicas de tomas de decisions usados comúnmente en la mayoría de Videojuegos.
- *Avanzado*
  - Puestos en práctica en juegos más sofisticados.
- *En el futuro*
  - Temas de investigación.

En este tema…

- IA básica
  - Técnicas de tomas de decisions usados comúnmente en la mayoría de Videojuegos:
    - Pathfinding básico (A*) (en 2º)
    - Árboles de decision (aquí)
    - *Máquinas de estado (Jerárquicas) (aquí)*
- IA avanzada
  - Puestos en práctica en juegos más sofisticados:
    - Pathfinding avanzado (otras…)
    - Árboles de comportamiento *(aquí)*

Futuro de la IA en Videojuegos

- Lógica difusa
- Agentes Basados en Metas
- Aprendizaje Automático
- Planificación
- …

Dos tipos Fundamentales de Algoritmos para Inteligencia Artificial

- *Sin búsqueda:*
  - Se puede predecir el coste computacional
    - Árboles de decisión, Máquinas de Estados

- *Con búsqueda:*
  - El coste computacional depende del tamaño del espacio de búsqueda (normalmente grande)
    - Minimax, Planificación. A veces Pathfinding
    - Un problema para videojuegos en tiempo real (Necesitan "cortocircuitar")
    - O juegos que necesitan limitar el coste computacional

Dos tipos Fundamentales de Algoritmos para Inteligencia Artificial

- ¿Dónde está el "*Conocimiento*"?
  - *Sin búsqueda:* En la lógica del código (o tablas externas)
  - *Con búsqueda:* En la evaluación de los estados

- ¿Cuál es mejor? La que tenga mejor Conocimiento. ;-)

Codificación de IA básica

- Usar el paradigma Orientado a Objetos

  en lugar de...

  – Una maraña de sentencias *if-then-else*

Árboles de Decisión

Ian Millington and John Funge. *Artificial Intelligence for Games*, Morgan Kaufmann, 2009. (Chapter 5)

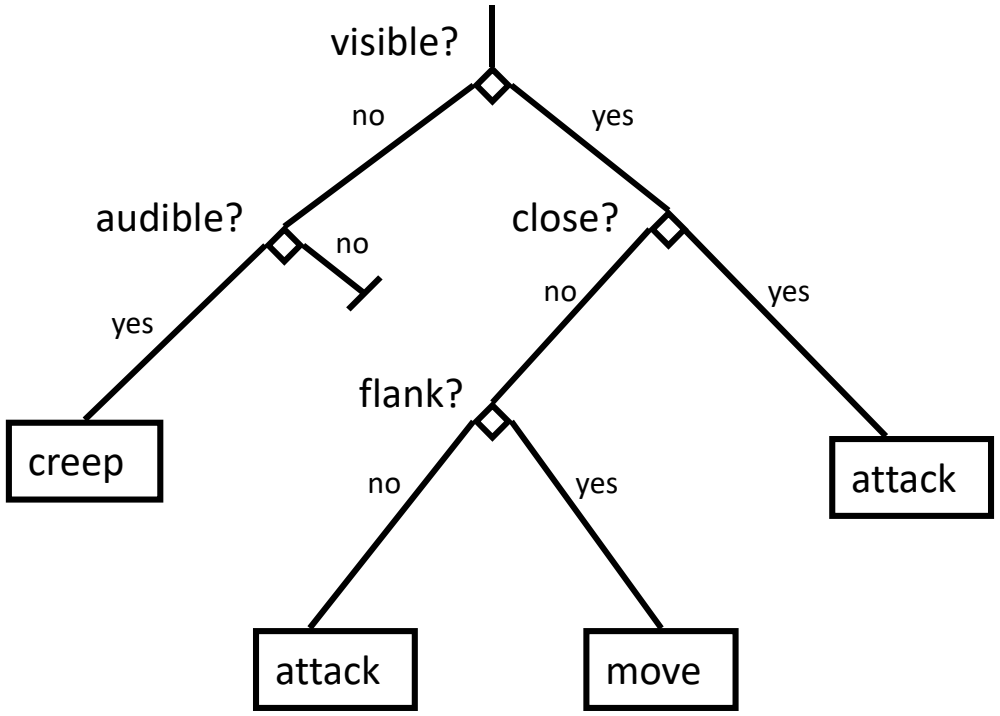
Árboles de Decisión

- Técnica de IA más básica
- Fácil de implementar
- Rápida ejecución
- Fácil de entender

Decidir cómo responder ante un Enemigo (1 of 2)

```
if visible? { // level 0
   if close? { // level 1
      attack;
   } else {     // level 1
      if flank? {  // level 2
         move;
      } else {     // level 2
         attack;
      }
   }
} else {       // level 0
   if audible? { // level 1
     creep;
   }
}
```

Hojas: Acciones
Nodos Interiores: Decisiones

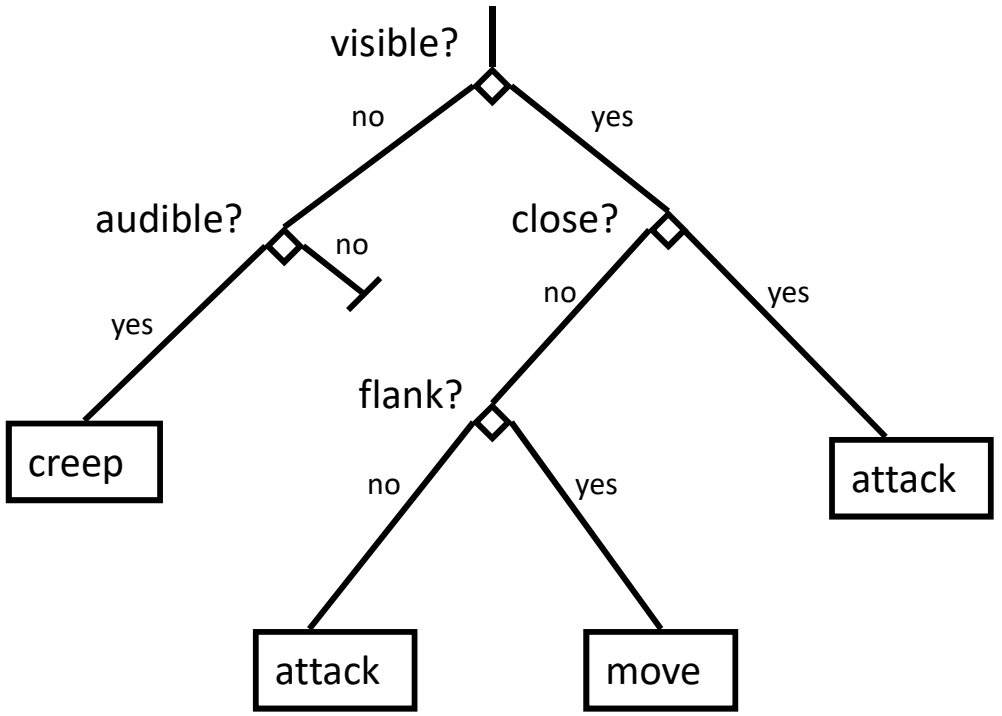


Normalmente **binarios**
(si tiene varias opciones, se puede convertir en binario)

Decidir cómo responder ante un Enemigo (2 of 2)

Forma alternativa.

```
if visible? { // level 0
   if close? { // level 1
      attack;
   } else if flank? { // level 1&2
      move;
   } else {
      attack;
   }
} else if audible? { // level 0&1
  creep;
}
```



¡Más difícil ver la profundidad!

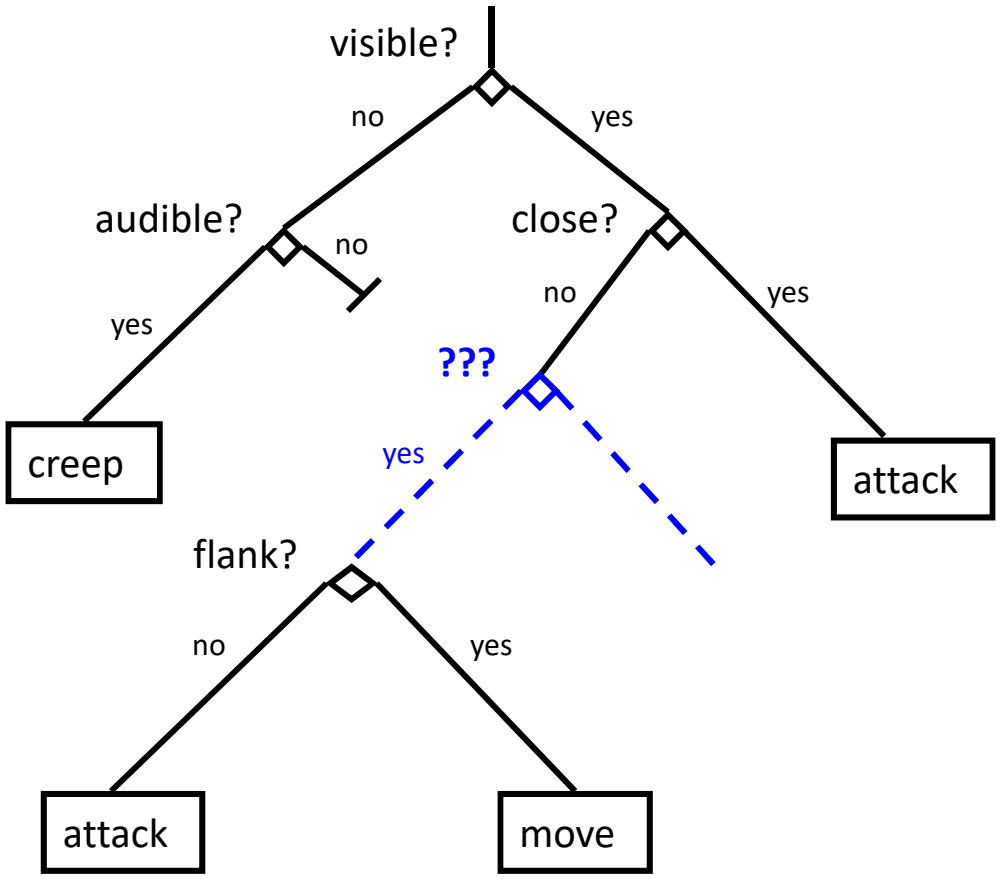
¿Cómo modificarlo?
e.g., if *close*, only flank if ally near

???

Modificación del comportamiento

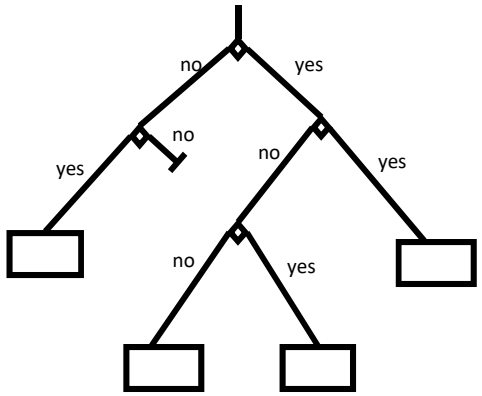
```
if visible? { // level 0
   if close? { // level 1
      attack;
   } else if flank? { // level
1&2
      move;
   } else {
      attack;
   }
} else if audible? { // level
0&1
  creep;
}
```

???



¡Las modificaciones reestructuran todo el código! Es frágil.
Solución → Programación Orientada a Objetos

Árboles de Decisión OO (Pseudo-Código)



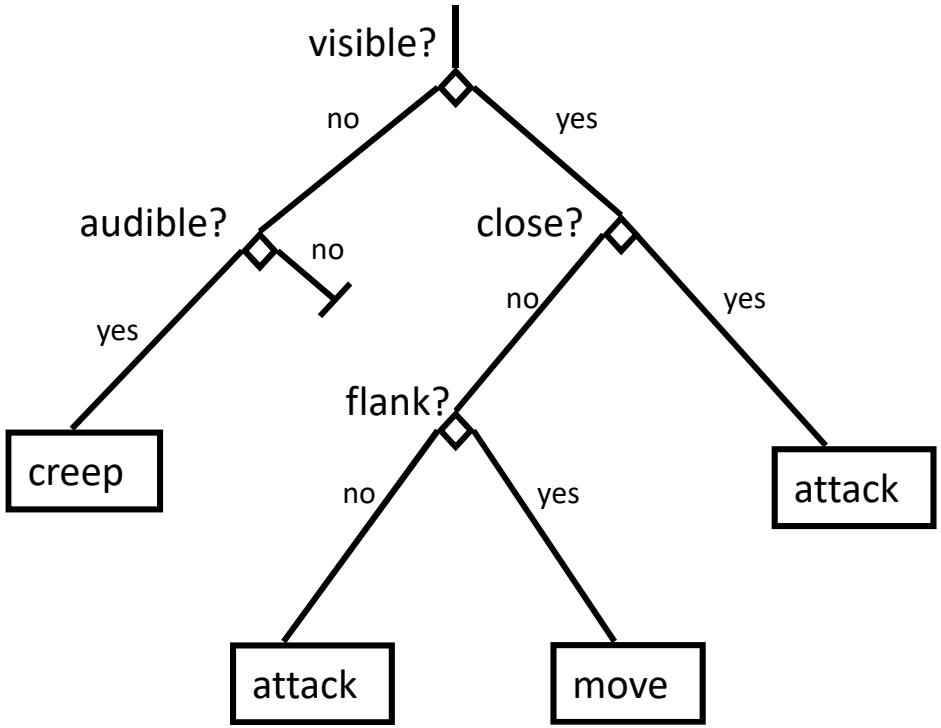

```
class Node
  def decide() // return action/decision


class Decision : Node  // interior
  def getBranch() // return a node
  def decide()
   return getBranch().decide()

class Action : Node  // leaf
  def decide() return this
```

```
class Boolean : Decision  // if yes/no
  yesNode
  noNode

class MinMax : Boolean  // if range
  minValue
  maxValue
  testValue

def getBranch()
   if maxValue >= testValue >= minValue
      return yesNode
   else
      return noNode
```

```
// Define root as start of tree
Node *root

// Calls recursively until action
Action * action = root → decide()
action → doAction()
```

Creando un Árbol de Decisión OO

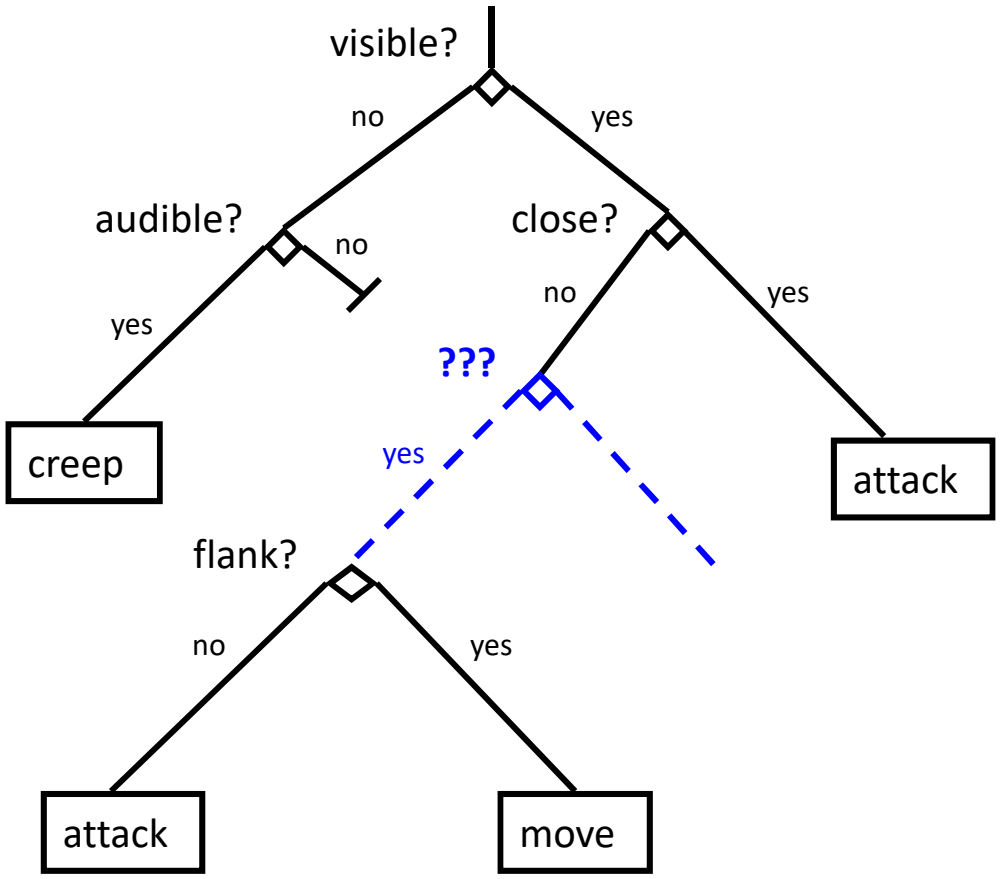
```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...

attack = new Attack...
move = new Move...
creep = new Creep...

visible.yesNode = close
visible.noNode = audible

audible.yesNode = creep

close.yesNode = attack
close.noNode = flank

flank.yesNode = move
flank.noNode = attack
...
```



...o un editor gráfico

Modificando un Árbol de Decisión OO

```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...
??? = new Boolean...

attack = new Action...
move = new Action...
creep = new Action...

visible.yesNode = close
visible.noNode = audible

audible.yesNode = creep

close.yesNode = attack
close.noNode = ???
???.yesNode = flank

flank.yesNode = move
flank.noNode = attack
...
```

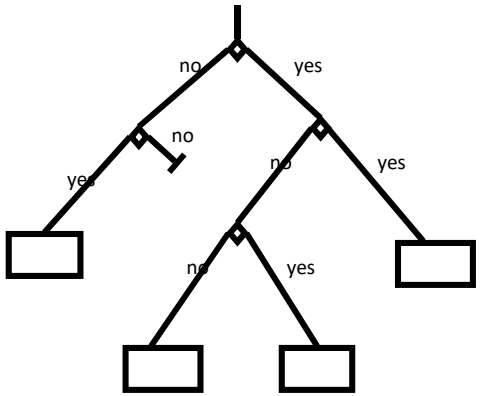


Funcionamiento de un Árbol de Decisión

- Test de nodos individuales (`getBranch`) normalmente tiempo constante (y rápido)

- El peor de los casos depende de la profundidad del árbol
  - Camino mas largo de la raíz a la hoja (acción)

- Aproximadamente árboles "balanceados" (cuando sea posible)
  - Ni mucha profundidad, ni mucha anchura
  - Hacer los caminos frecuentes cortos
  - Hacer las decisiones mas costosas al final



Máquinas Finitas de Estado (Jerárquicas)

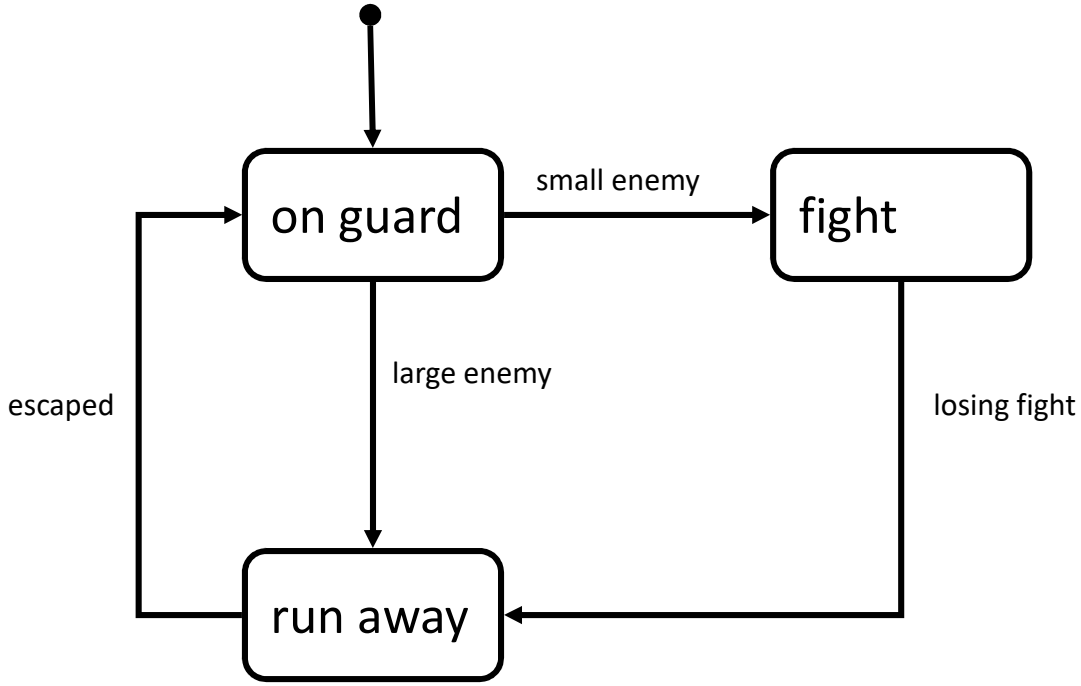
Máquinas Finitas de Estados (FSM)

- IA mediante Agentes: *percibir*, *pensar*, *actuar*
- Muchas reglas diferentes para los Agentes
  - Ex: *percibir, pensar* y *actuar* cuando *lucha*, *corre*, *explora*…
  - Puede ser difícil mantener la consistencia de las reglas
- Máquina Finita de Estados
  - Correspondencia natural entre estados y comportamientos
  - Fácil de: Representar (Diagrama), Programar y Depurar
- Formalmente:
  - Conjunto de Estados
  - Estado inicial
  - Alfabeto de entrada
  - Conjunto de Trasiciones que indican el Estado Siguiente, según el Estado Actual y la Entrada

(Ejemplo para Videojuegos en la siguiente diapositiva)

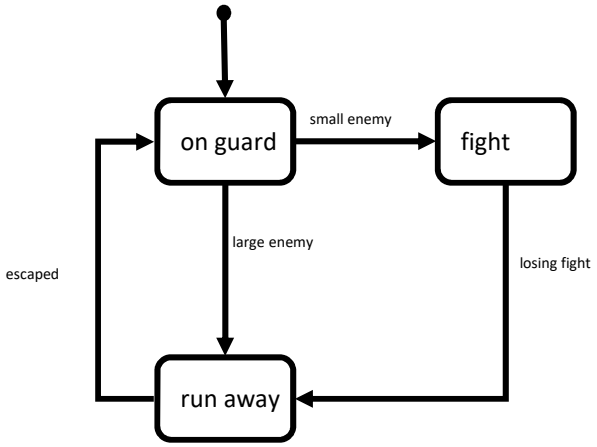
Máquina Finita de Estados

- *Estados*: Acciones
- *Condiciones*: Percepción
- *Transiciones*: Pensar



Implementación

```
class Soldier

  enum State
    ON_GUARD
    FIGHT
    RUN_AWAY

  currentState
```



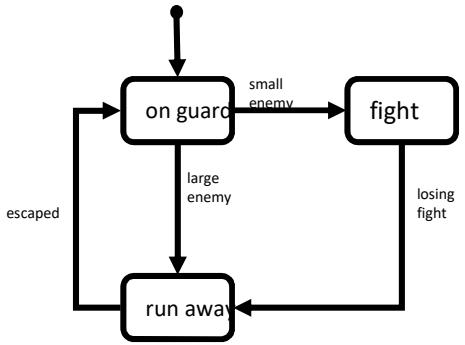

```
def update()
    if currentState == ON_GUARD {
        if small enemy {
            currentState = FIGHT
            start Fighting
        } else if big enemy {
            currentState = RUN_AWAY
            start RunningAway
        }
    } else if currentState == FIGHT {
        if losing fight {
            currentState = RUN_AWAY
            start RunningAway
        }
    } else if currentState == RUN_AWAY {
        if escaped {
            currentState = ON_GUARD
            start Guarding
        }
    }
```

Implementación

- Fácil de codificar (Al principio)
- Muy eficiente
- Muy difícil de mantener (modificar y depurar)

Implementación OO más Limpia y Flexible

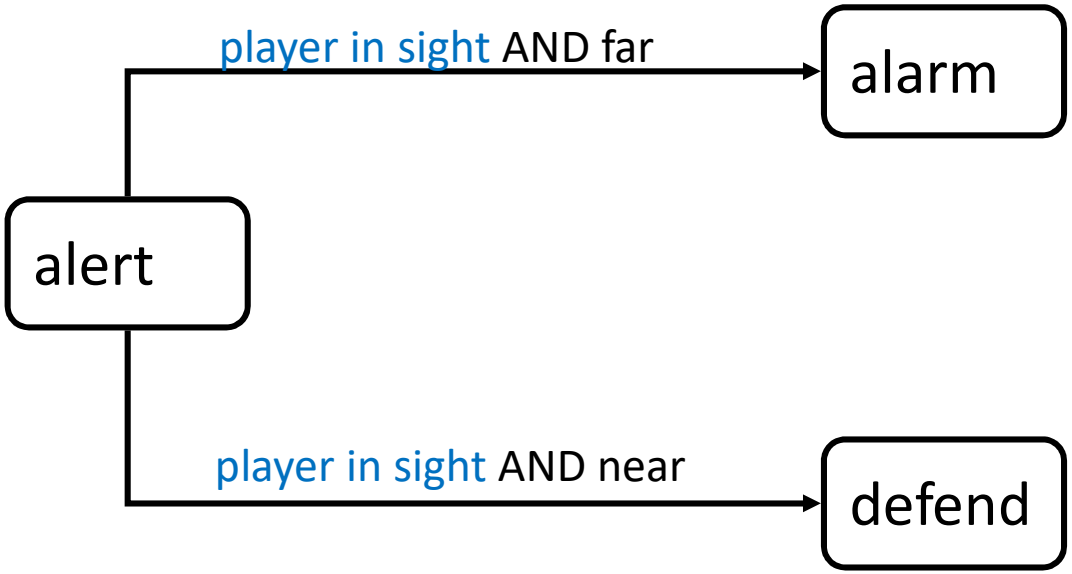
```
class State
   def getAction()
   def getEntryAction()
   def getExitAction()
   def getTransitions()

class Transition
   def isTriggered()
   def getTargetState()
```




```
class StateMachine

   states
   initialState
   currentState = initialState

   def update() // returns all actions needed this update

      triggeredTransition = null

      for transition in currentState.getTransitions() {
         if transition.isTriggered() {
            triggeredTransition = transition
            break
          }
      }
      if triggeredTransition != null {
         targetState = triggeredTransition.getTargetState()
         actions = currentState.getExitAction()
         actions += targetState.getEntryAction()
         currentState = targetState
         return actions // list of actions for transitions
      } else return currentState.getAction()  // action this state
```

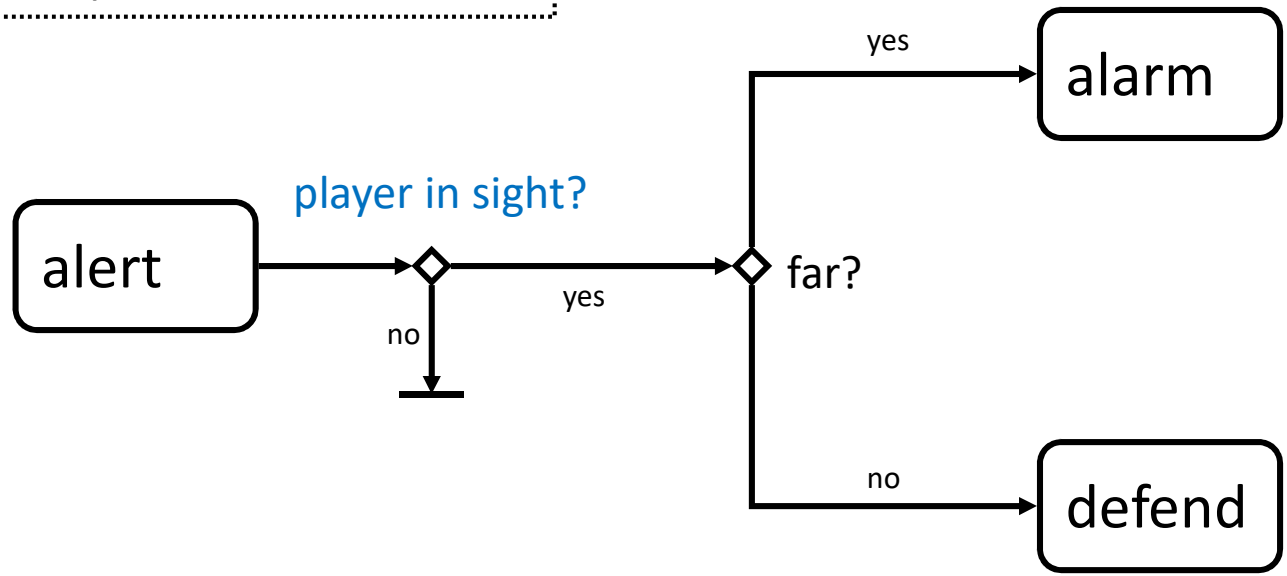
Combinar Árboles de Decisión y Máquinas de Estados (1 de 2)

- ¿Por qué?
  - Para evitar comprobaciones duplicadas (costosas) en Máquinas de Estados
  - Asumamos que "player in sight" es costoso



Combinar Árboles de Decisión y Máquinas de Estados (2 de 2)

Usar Árboles de Decisión para las transiciones de la Máquina de Estados

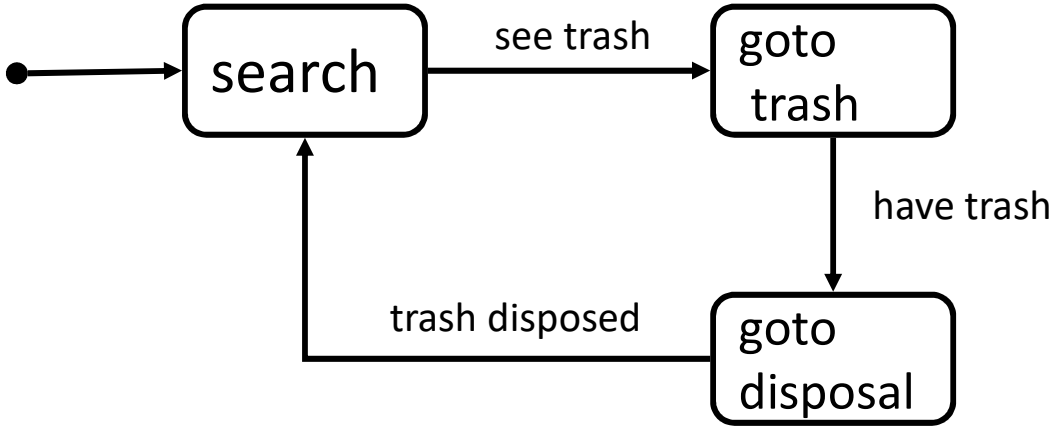


Esquema

- Introducción (Hecho)
- Árboles de Decisión (Hecho)
- Máquinas de Estado Finitas (FSM) (Hecho)
- FSM Jerárquicas (A continuación)
- Árboles de Comportamiento

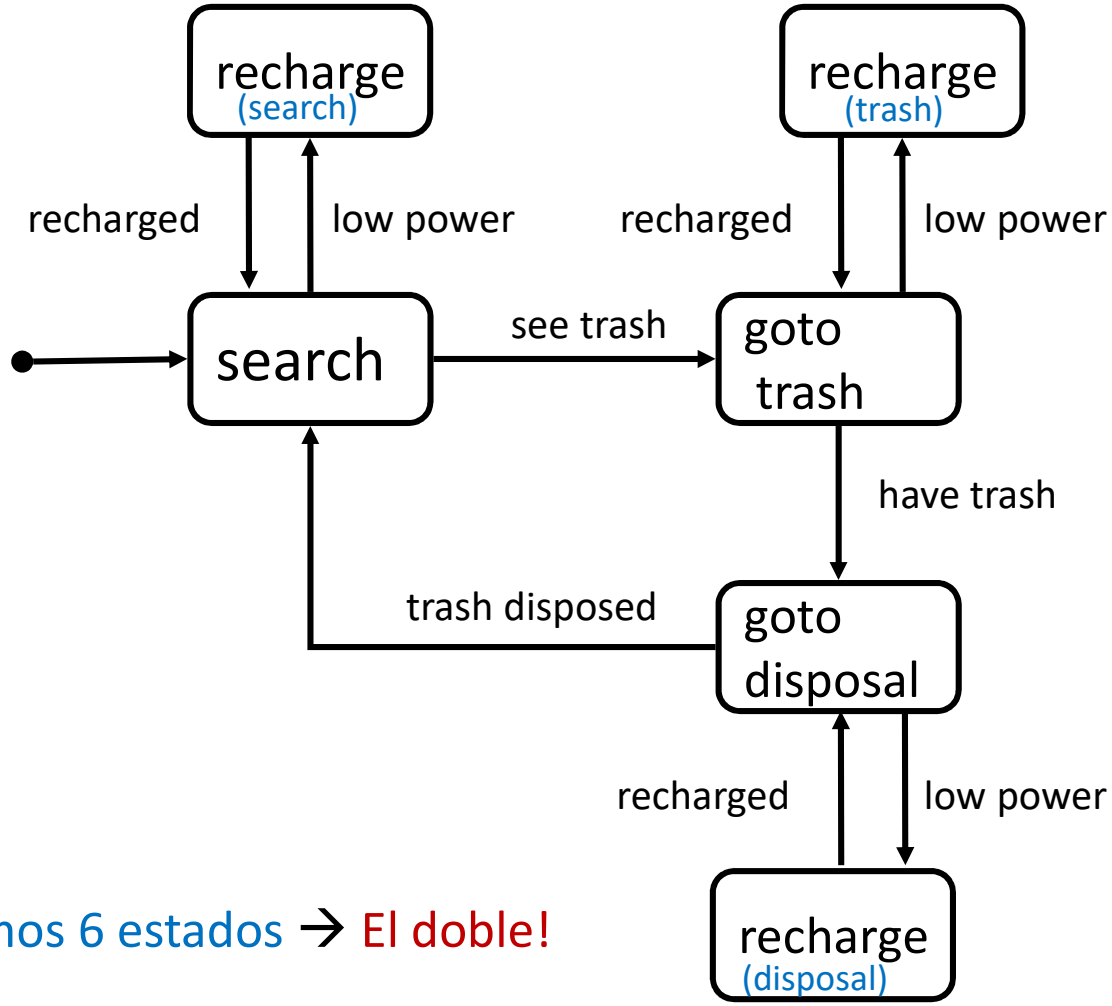
Máquinas de Estados *Jerárquicas*

- ¿Por qué? → Pueden haber interrupciones, pero no querremos retroceder al principio



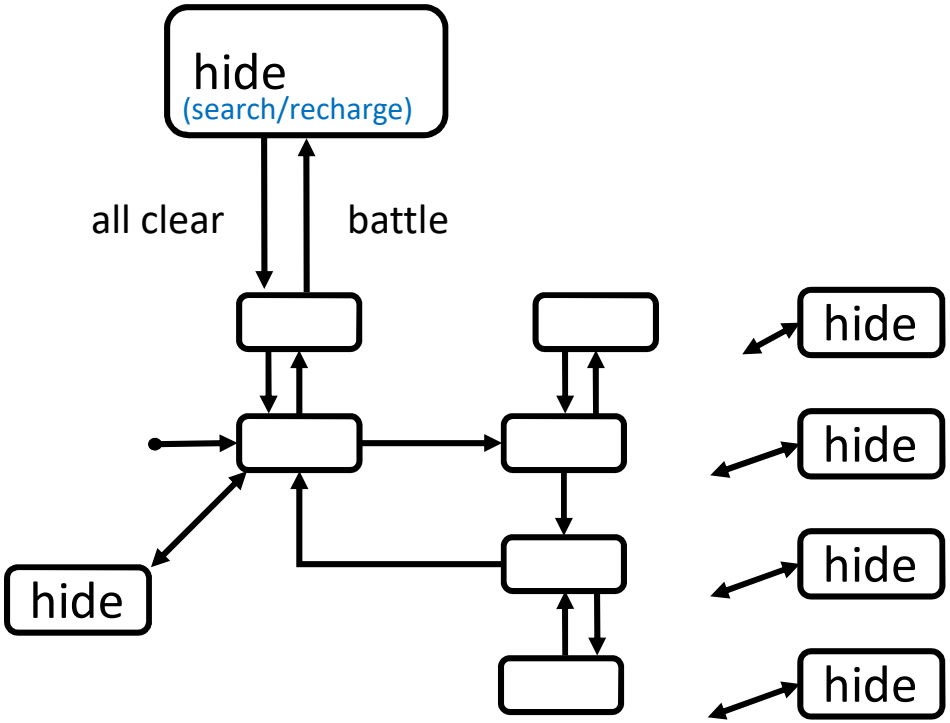
e.g., El robot puede quedarse sin batería en cualquier estado.
Necesitará recargar la batería.
Cuando haya recargado, necesita volver al Estado anterior
e.g., puede tener basura o saber dónde hay basura.

Interrupciones (e.g., Recargar)



Necesitamos 6 estados → El doble!

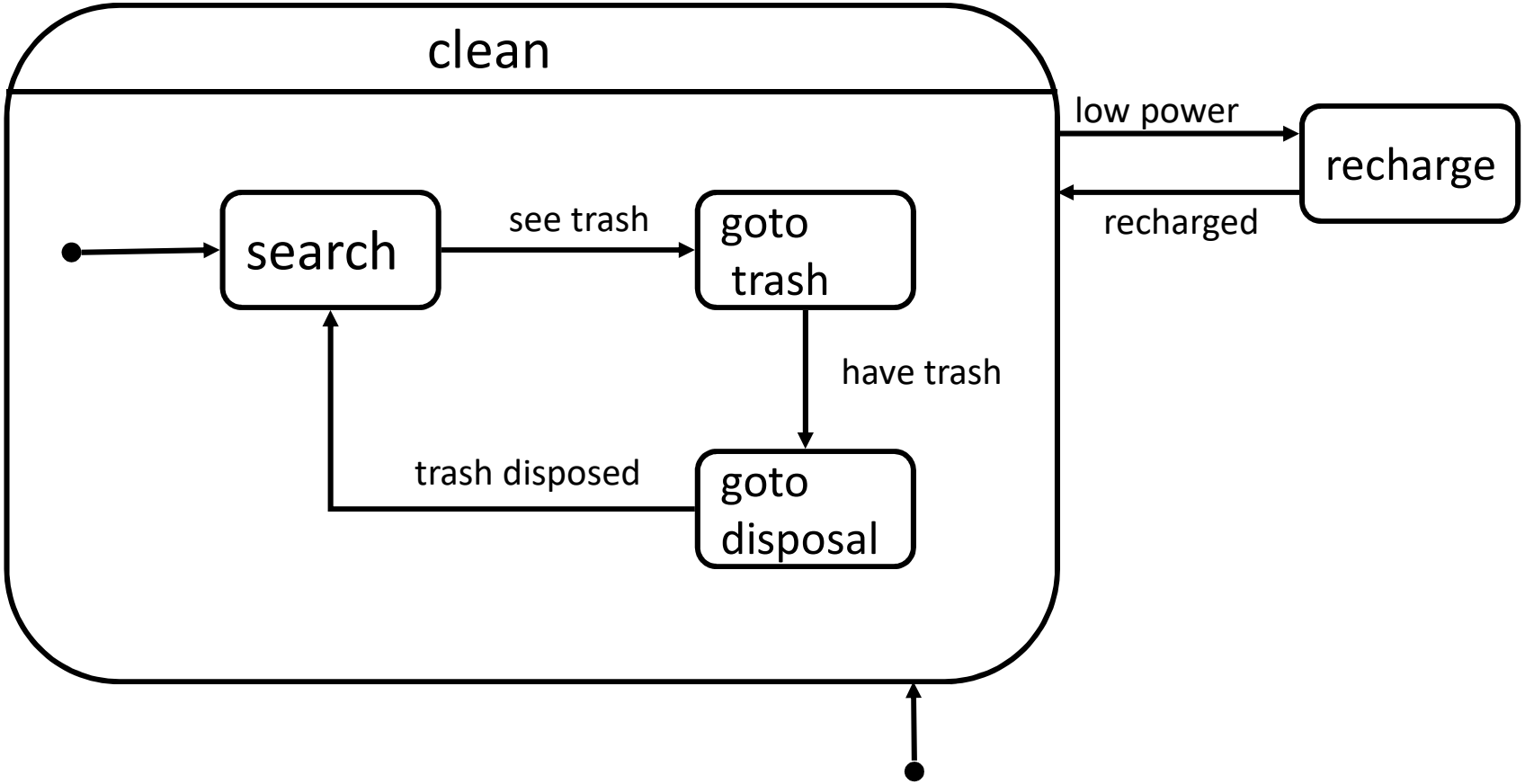
Añadir otra Interrupción (e.g., Enemigos)



Necesitamos 12 estados→ Otra vez el doble!

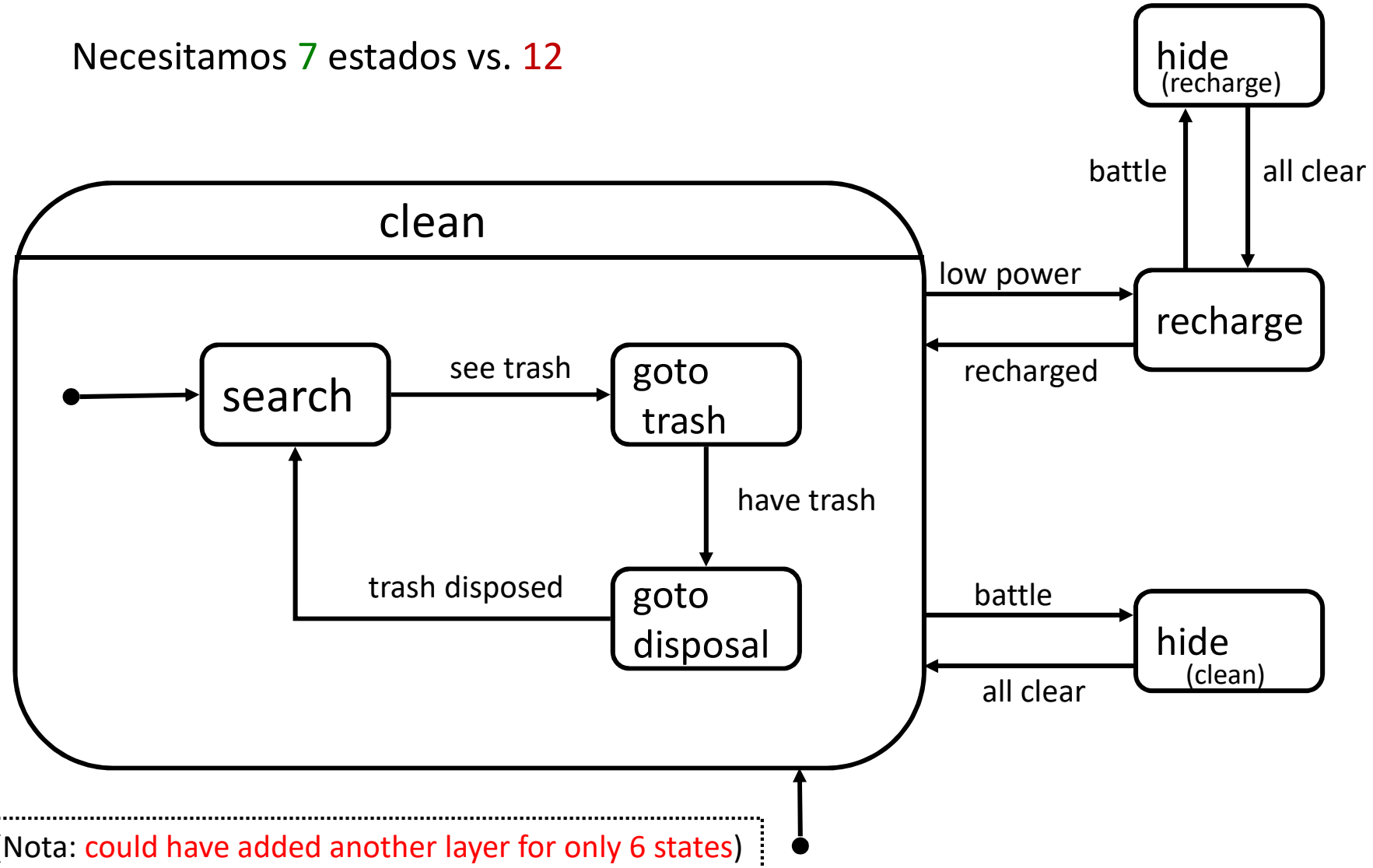
Máquina de Estados *Jerárquica*

- Deja cualquier estado del estado “Limpiar” cuando “batería baja”
- El estado “Limpiar” recuerda el estado interno y continua cuando vuelve de “Recargar”’



Añadir otra Interrupción (e.g., Enemigos)

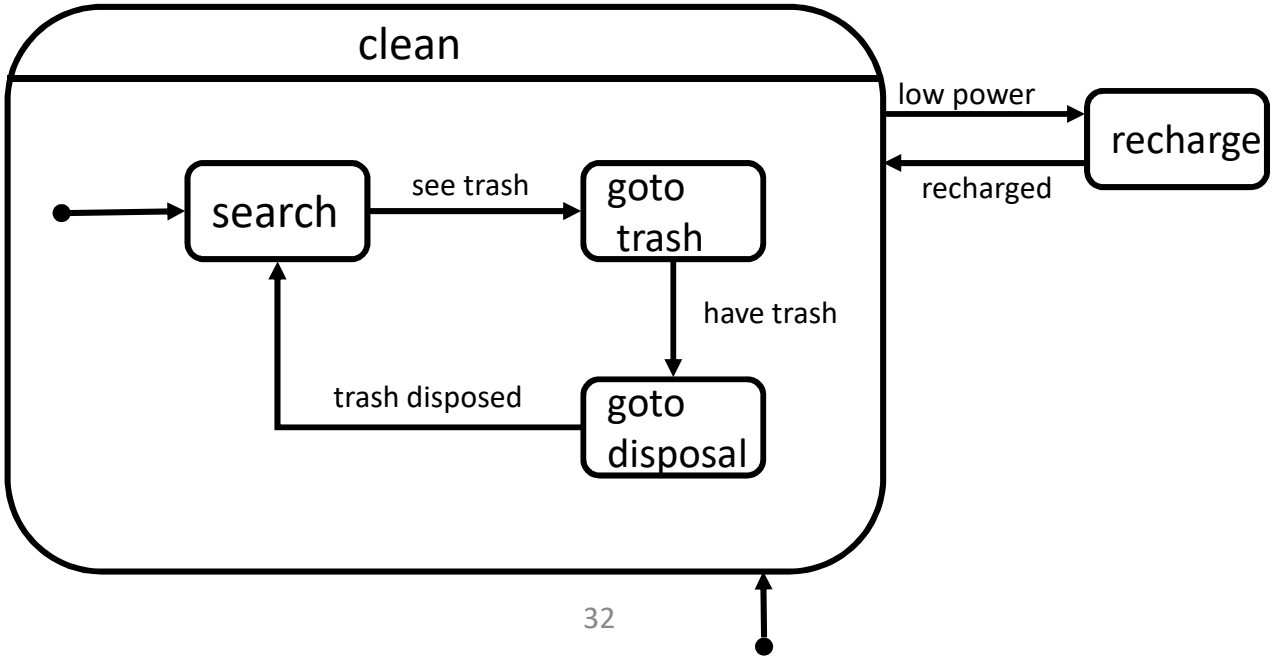
Necesitamos 7 estados vs. 12



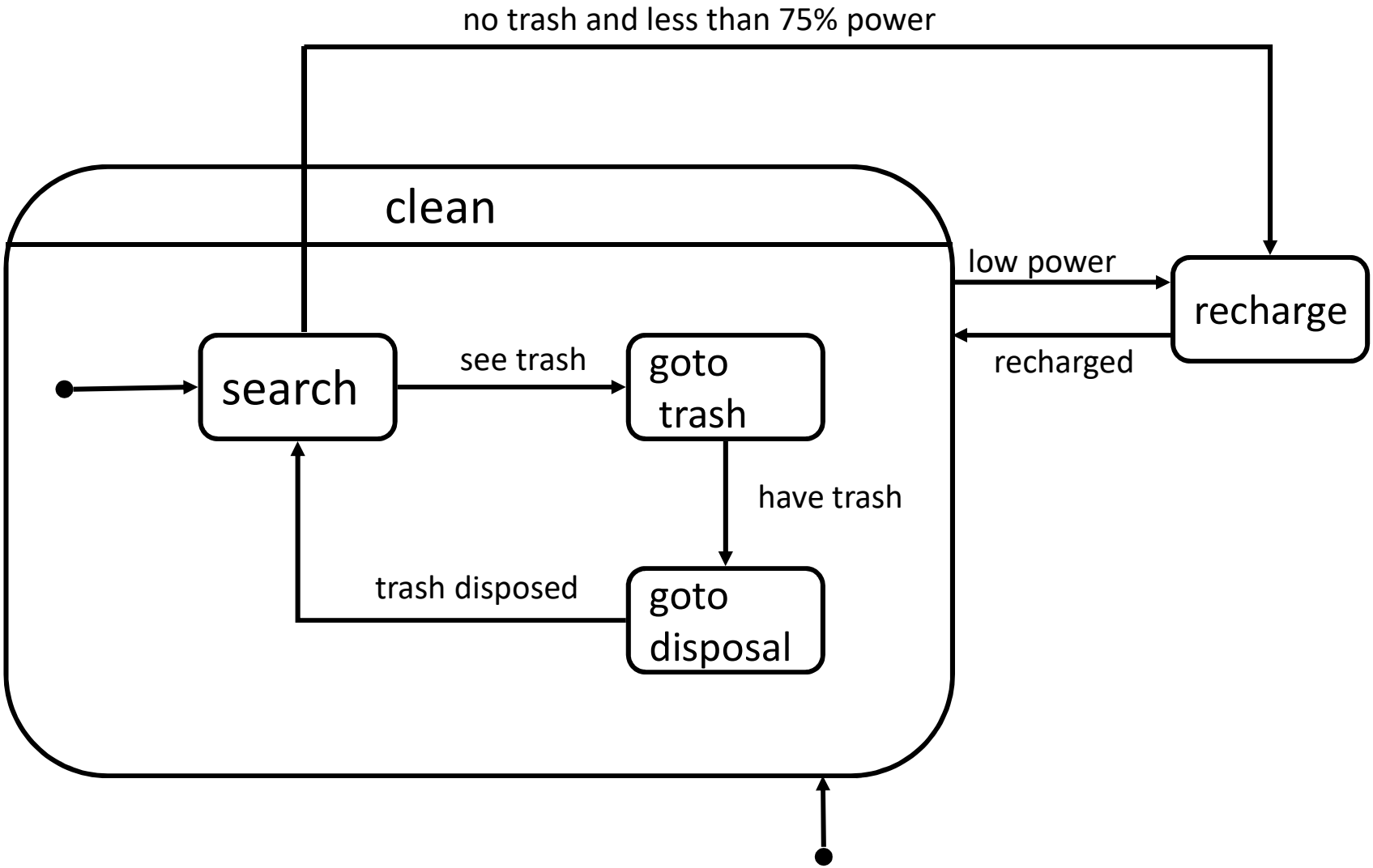
(Nota: could have added another layer for only 6 states)

Jerarquía Cruzada: Transiciones

- ¿Por qué?
  - Supongamos que queremos un robot que "Recargue" (incluso si no tiene batería baja) cuando no vea basura



Jerarquía Cruzada: Transiciones



HFSM: Implementación

```
class State

   // stack of return states
   def getStates() return [this]

   // recursive update
   def update()

   // rest same as flat machine

class Transition

   // how deep this transition is
   def getLevel()

   // rest same as flat machine

struct UpdateResult // returned from update
   transition
   level
   actions // same as flat machine
```

```
class HierarchicalStateMachine

   // same state variables as flat
machine

   // complicated recursive
algorithm*
   def update ()


class SubMachine :
HierarchicalStateMachine,
                     State

   def getStates()
      push this onto
currentState.getStates()
```

*Pseudo-Código completo:
http://web.cs.wpi.edu/~imgd4000/d16/slides/millington-hsm.pdf

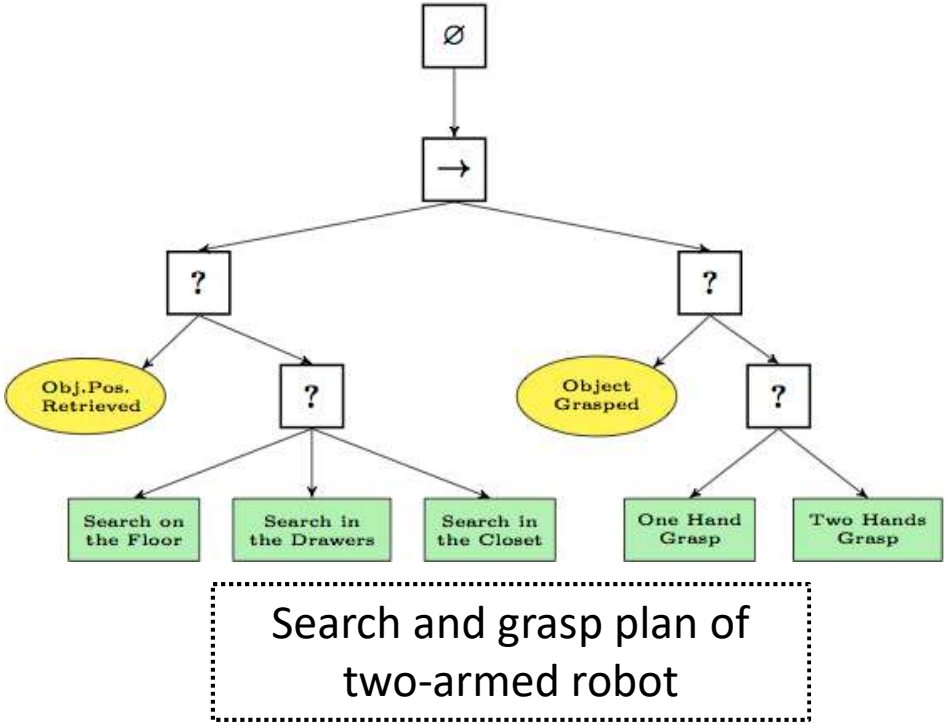
Esquema

- Introducción (Hecho)
- Árboles de Decisión (Hecho)
- Máquinas de Estado Finitas (FSM) (Hecho)
- FSM Jerárquicas (Hecho)
- Árboles de Comportamiento (A continuación)
  - In UE4

    http://www.slideshare.net/JaeWanPark2/behavior-tree-in-unreal-engine-4

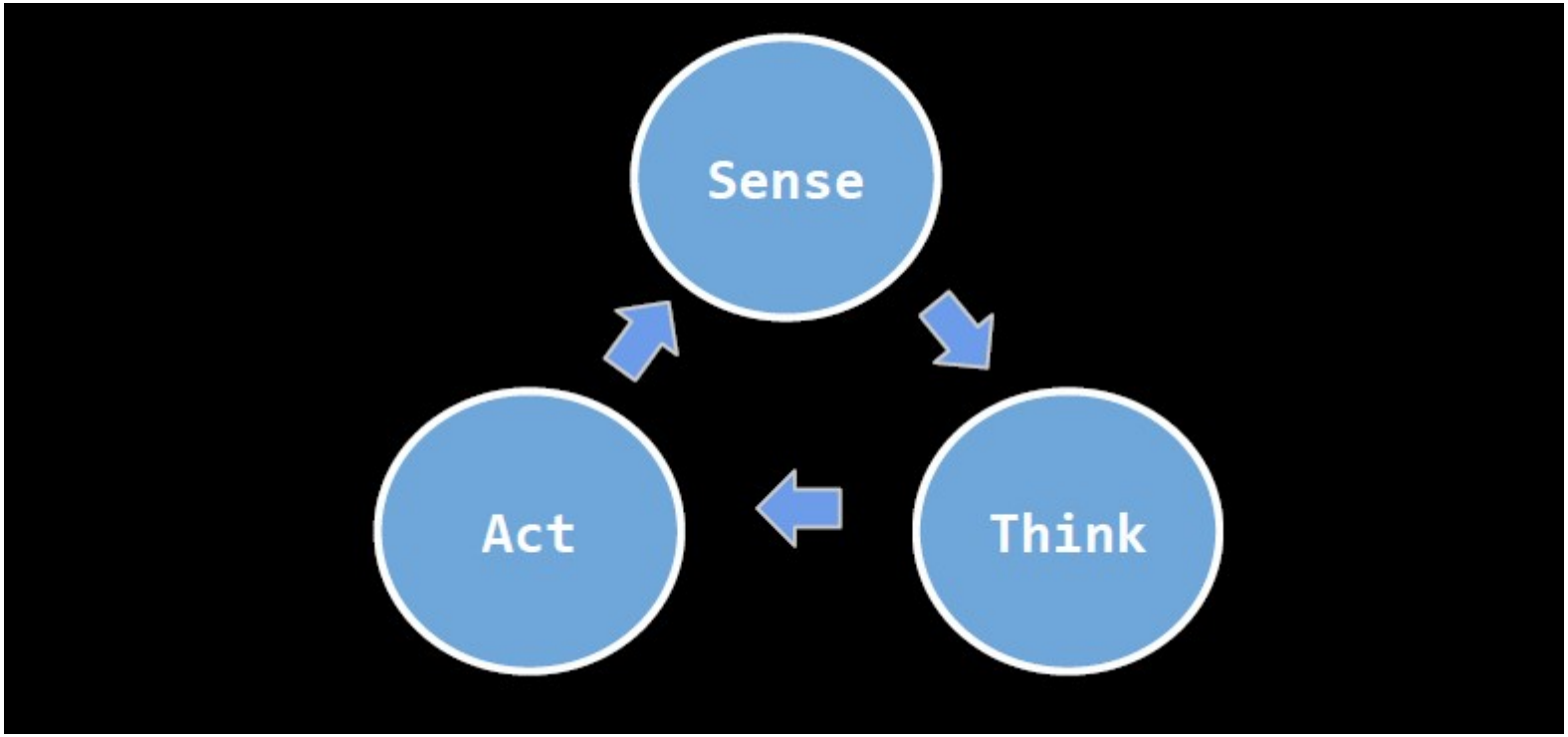
Árboles de Comportamiento

- Son un modelo de ejecución de una planificación
  - Cambia entre tareas modularmente
- Parecido a HFSM, pero los bloques son Tareas, en lugar de Estados
- Usado para NPCs (*Halo*, *Bioshock*, *Spore*)
- Árbol – Los nodos son *Raíz*, *Flujo de Control*, *Ejecución*



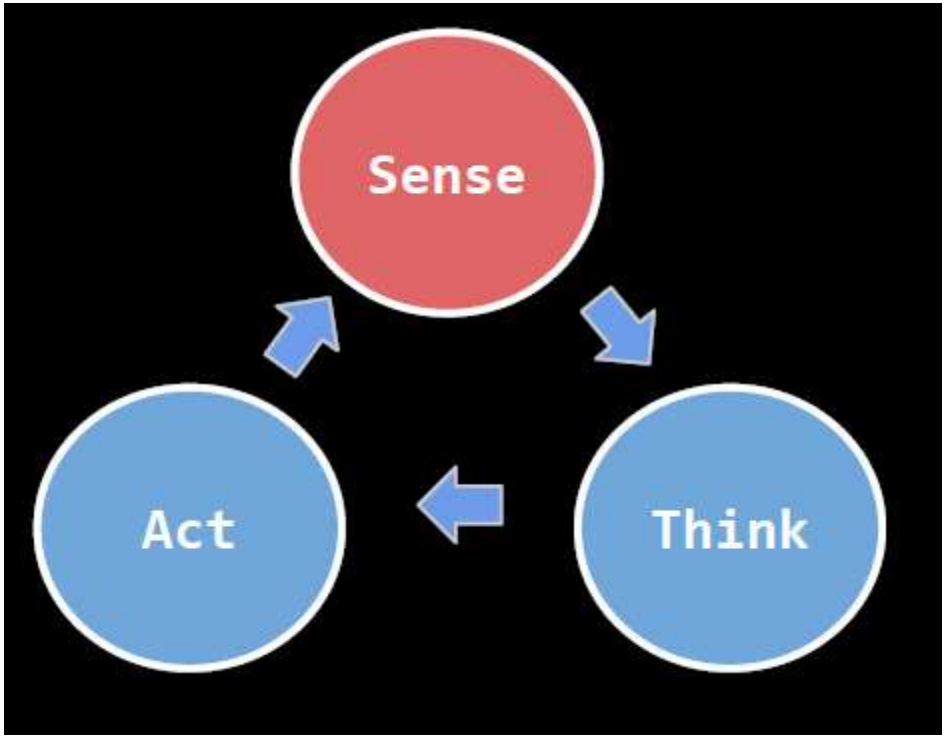
https://upload.wikimedia.org/wikipedia/commons/1/1b/BT_search_and_grasp.png

“Comportamiento”
Árboles de Comportamiento



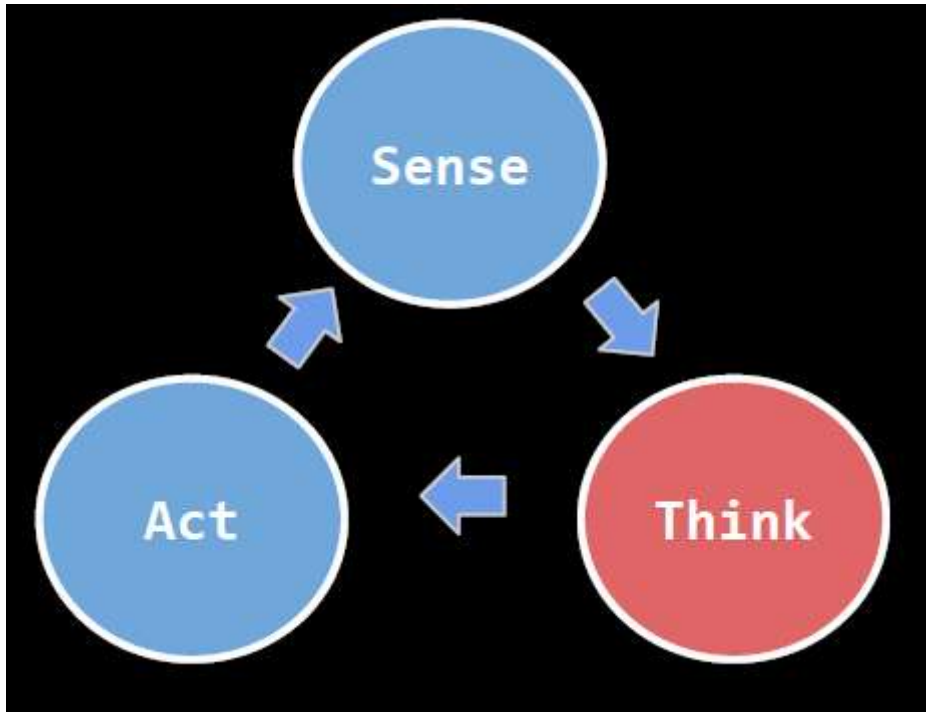
- Percibir, Pensar, Actuar
- Repetir

Percibir



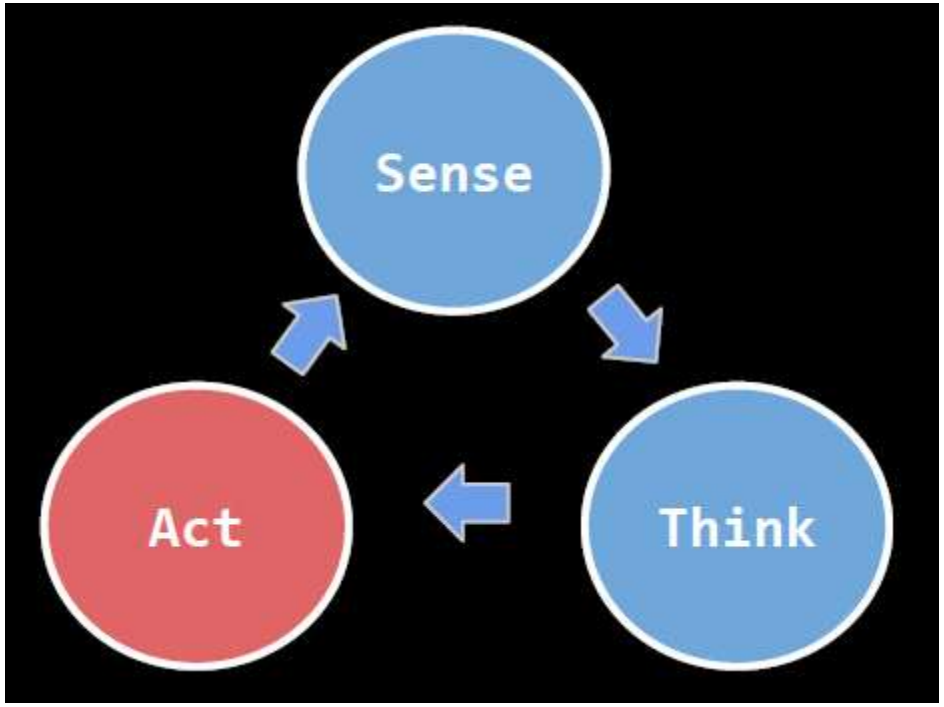
- Generalmente depende del motor de Físicas.
- Normalmente es costosa.
- No se usa en exceso.

Pensar



- Lógica de las Decisiones.
- Normalmente es simple.
- Diseño intensivo.

Actuar



- Ejecución de Acciones.
- Suele tardar ejecutándose.
- Puede no ejecutarse completamente.

Recursos

- HFSM de Millington y Funge
  http://web.cs.wpi.edu/~imgd4000/d16/slides/millington-hsm.pdf
- FSM de IMGD 3000
  - Transparencias
    http://www.cs.wpi.edu/~imgd4000/d16/slides/imgd3000-fsm.pdf
  - Archivos
    http://dragonfly.wpi.edu/include/classStateMachine.html
- Árbol de Comportamiento UE4
  - Diferencia entre AD y AC
    http://gamedev.stackexchange.com/questions/51693/decision-tree-vs-behavior-tree
  - Inicio
    https://docs.unrealengine.com/latest/INT/Engine/AI/BehaviorTrees/QuickStart/