Comportamientos inteligentes básicos

Sistemas Inteligentes. 2021-22

Grado en Ingeniería Informática

Especialidad: Computación

“Niveles” de IA en agentes

- *Básico*
  - Técnicas de tomas de decisiones, principalmente reactivas.
- *Avanzado*
  - Técnicas de razonamiento y planificación
- *En el futuro*
  - Temas de investigación.

En este tema...

- Vamos a ver comportamientos básicos:
  - Motores de reglas
  - Árboles de decision
  - Máquinas de estado (Jerárquicas)

IA avanzada

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

Tipos Fundamentales de Algoritmos

- Sin búsqueda:
  - Se puede predecir el coste computacional
    - Reglas, 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

Tipos Fundamentales de Algoritmos

- ¿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 se comporte major!

Sistemas basados en reglas

Sistemas basados en reglas

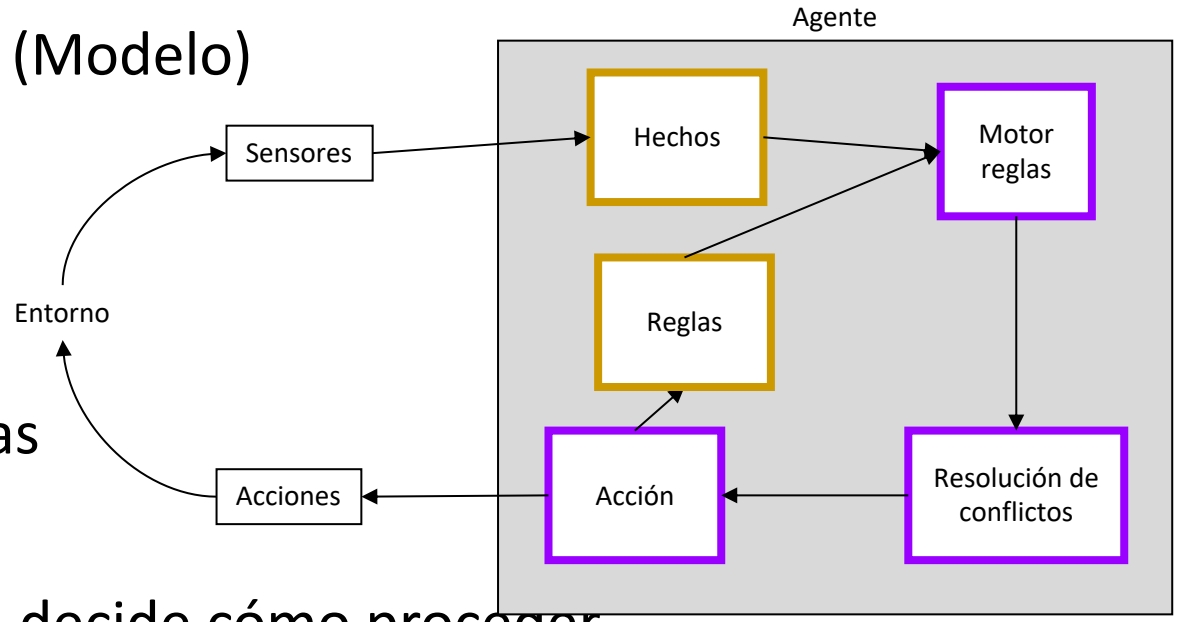
- También se conocen como "sistemas de producción" o "sistemas expertos".
- Los sistemas basados en reglas son uno de los paradigmas de IA más exitosos
  - Se originaron en los años 70 y 80.

Sistemas basados en reglas

- Datos de problemas almacenados como hechos y reglas.
- "Razona" utilizando reglas del tipo: IF...THEN...ELSE
  - `IF humedad is OK  AND  temperature > 26`
    `THEN Encender_Aire`
- Puede "razonar”:
  - deductivamente (encadenamiento hacia delante) o
  - inductivamente (encadenamiento hacia atrás).

Componentes

- Memoria de trabajo
  - donde se guardan los hechos ciertos (Modelo)
- Reglas de producción
  - reglas del funcionamiento
- Motor de inferencia
  - programa para calcular consecuencias
- Resolución de conflictos
  - si se puede aplicar más de una regla, decide cómo proceder



Razonamiento

Tipos de razonamientos

- Existen dos enfoques de inferencia en los sistemas basados en reglas:
  - **Encadenamiento hacia delante** (o basado en datos):
    Partiendo de un conjunto de antecedentes utilizamos la deducción para llegar a una conclusión.
  - **Encadenamiento hacia atrás** (o impulsado por objetivos):
    Partiendo de una conclusión intentamos demostrarla siguiendo un camino lógico hacia atrás desde la conclusión hasta un conjunto de antecedentes que están en la base de datos de hechos.

Tipos de razonamientos

- Se dice que una regla **se dispara** si sus antecedentes coinciden con algunos de los hechos de la base de datos de hechos.
- Y el **disparo** de la regla hace que su parte de **conclusión** se añada a la base de datos de hechos o se realiza su acción consecuente.

Razonamiento hacia adelante

- También llamado “Forward Chaining”

Hacerlo hasta que se resuelva el problema o no coincidan los antecedentes
Recoger las reglas cuyos antecedentes se encuentran en el Modelo.
Si más de una regla coincide
utilizar la estrategia de resolución de conflictos
eliminar todas menos una
Realizar las acciones indicadas en la regla "disparada"

Razonamiento hacia atrás

- También llamado “Backward Chaining”

```
Dado el objetivo g como entrada
        encontrar el conjunto de reglas S que determinan g
        si el conjunto de reglas no es igual al conjunto vacío entonces
                elegir una regla R del conjunto
                hacer que el antecedente de R sea el nuevo objetivo (ng)
                si el nuevo objetivo es desconocido, entonces
                        backward_chain(ng)   ## Recursivo
                si no
                        aplicar la regla R
                hasta que g se resuelva o S sea igual al conjunto vacío
        si no
                fallop
```

Ejemplo

- Tenemos un sistema experto con la siguiente memoria y el siguiente conjunto de reglas.
  - **QUEREMOS SABER SI G ES CIERTO**

Hechos:

Hecho 1: A

Hecho 2: B

Hecho 3: F

Regla 1: IF A^B THEN C

Regla 2: IF A THEN D

Regla 3: IF C^D THEN E

Regla 4: IF B^E^F THEN H

Regla 5: IF A^E THEN G

Regla 6: IF D^E^G THEN I

Ejemplo: Razonamiento hacia delante

Hechos:

Hecho 1: A

Hecho 2: B

Hecho 3: F

Regla 1: IF A^B THEN C

Regla 2: IF A THEN D

Regla 3: IF C^D THEN E

Regla 4: IF B^E^F THEN H

Regla 5: IF A^E THEN G

Regla 6: IF D^E^G THEN I

Hechos (Modelo del mundo)	Reglas activadas	Regla disparada
A,B,F	1,2	1
A,B,C,F	2	2
A,B,C,D,F	3	3
A,B,C,D,E,F	4,5	4
A,B,C,D,E,F,H	5	5
A,B,C,D,E,F,H,**G**	6	STOP

Ejemplo: Razonamiento hacia atrás

Hechos:

Hecho 1: A

Hecho 2: B

Hecho 3: F

Regla 1: IF A^B THEN C

Regla 2: IF A THEN D

Regla 3: IF C^D THEN E

Regla 4: IF B^E^F THEN H

Regla 5: IF A^E THEN G

Regla 6: IF D^E^G THEN I

Facts	Goals	Matching rules
A,B,F	G	5
A,B,F	E	3
A,B,F	C,D	1
A,B,C, F	D	2
A,B,C,D,E, F,G		Stop

Resolución de conflictos

Resolución de conflictos

- Ordenar físicamente las reglas
  - Es difícil añadir reglas a estos sistemas
- Ordenación de datos
  - ordenar los elementos del problema en la cola de prioridad
  - utilizar la regla que se ocupa de los elementos más prioritarios
- Especificidad o máxima especificidad
  - en función del número de antecedentes que coinciden
  - elegir el que tenga más coincidencias
- Selección aleatoria
- Disparar todas las reglas de aplicación
- Búsqueda de la regla más adecuada**

Ventajas y desventajas

Ventajas

- Ventajas
  - Son un mecanismo computacional universal y fácilmente exportables entre lenguajes de programación
  - Corresponde a la forma en que la gente suele pensar en el conocimiento
  - Muy expresivo
  - Conocimiento modular
  - Legibilidad
  - …

Desventajas

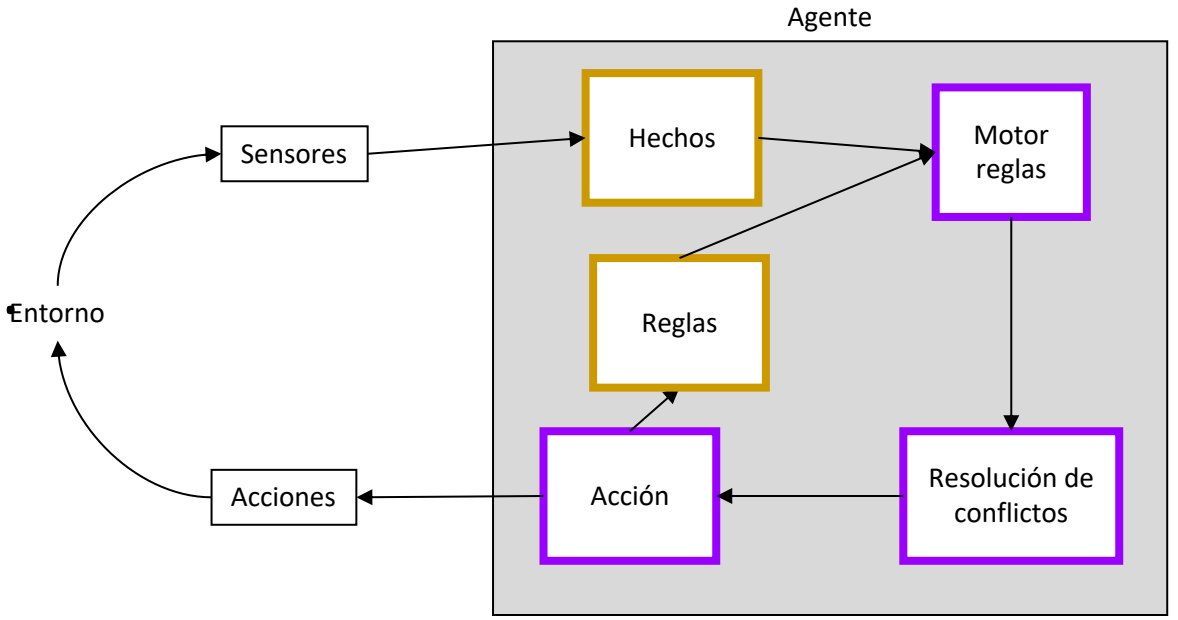
- Desventajas
  - No todo el conocimiento del dominio se ajusta al formato de las reglas
  - Debe existir un consenso de expertos
  - Puede requerir mucha memoria
  - Puede ser intensivo desde el punto de vista computacional
  - A veces es difícil de depurar
  - Requiere un conocimiento muy detallado
  - Olvidar los hechos malos es difícil

Implementación

Implementación

- Recuperando la estructura del sistema, vemos que necesitamos:
  - Construir reglas
  - Mecanismos de inferencia
  - Resolución de conflictos

- VAMOS A USAR OO
  y con la mayor generalidad posible.



Reglas

- Las reglas de nuestros sistemas van a ser de tipo:
- IF <lista_de_condiciones> THEN <acción>
- Por tanto, necesitamos implementar la CONDICIÓN y la ACCIÓN

```
public interface Condicion {
        public boolean seCumple(Cerebro cerebro);
}
```

Reglas

- Una regla la podemos construir como una lista de Condiciones (Antecedentes) y una acción (ACTIONS)

```
public class Regla {

        private List<Condicion> Antecedentes;
        private ACTIONS accion;

        public Regla(List<Condicion> _ant, ACTIONS _accion) {
                this.Antecedentes = _ant;
                this.accion = _accion;
        }
```

Reglas

- Para el razonamiento deberemos de incluir un método para saber si se cumple y la obtención de los consecuentes (Forward) y antecedentes (Backward)

```
public boolean seCumple(Cerebro cerebro) {
        … si se cumple para todas las condiciones…
        return resultado;
}

public List<Condicion> getAntecedentes() {
        return Antecedentes;
}

public ACTIONS getAccion() {
        return accion;
}
```

Motor de inferencia

- El motor de inferencia chequea la lista de reglas y selecciona aquellas que se cumplen.
- Dentro de este motor se podría incluir el mecanismo de resolución de conflictos y devolver sólo 1 acción.
  - En nuestro caso, vamos a seleccionar la **primera** que se cumpla.

Motor de inferencia

```
public class MotorReglas {

        private List<Regla> Reglas;

        public MotorReglas(List<Regla> _reglas) {
                this.Reglas = _reglas;
        }

        public Regla disparo() {
                ....
                return regla_seleccionada;
        }
}
```

En nuestro agente...

```
public class Practica_03_01 extends AbstractPlayer {
        // Atributos persistentes del agente
        MotorReglas motor;
        Cerebro cerebro;

    public Practica_03_01(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {
        // Lista de condiciones
        Condicion condi1 = new estoyenPeligro();
        …
        // Listado de los antecedentes de las reglas
        List<Condicion> Ante1 = new LinkedList<Condicion>();
        Ante1.add(condi1);
        …
        // Construcción de las reglas
        Regla regla1 = new Regla(Ante1,ACTIONS.ACTION_DOWN);
        …
        // Lista de reglas
        List<Regla> ListaReglas = new LinkedList<Regla>();
        ListaReglas.add(regla1);
        …
        // Creación del motor
        motor = new MotorReglas(ListaReglas,cerebro);
    }
```

En nuestro agente...

- En el ”act”, actualizamos nuestra visión del mundo, elegimos regla, y la disparamos.

```
public Types.ACTIONS act(StateObservation stateObs, ElapsedCpuTimer elapsedTimer) {

        cerebro.analizarMundo(stateObs);
        Regla r = motor.disparo();
        return r.getAccion();
}
```

Árboles de Decisión

Codificación de IA básica

- Usar el paradigma Orientado a Objetos

  en lugar de...

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

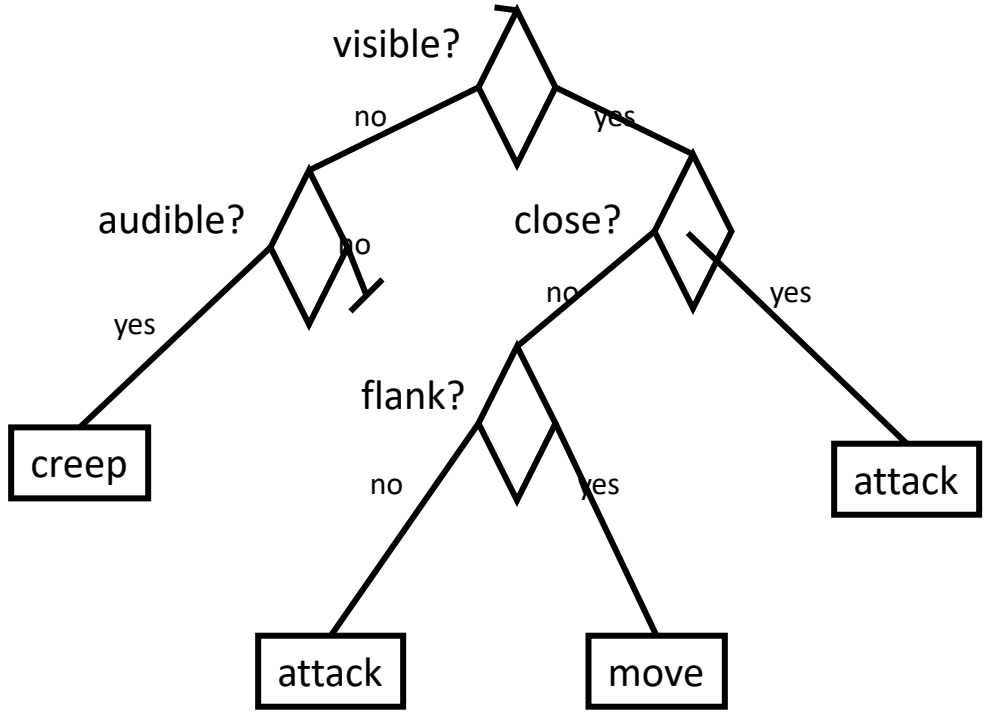
Árboles de Decisión

- 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

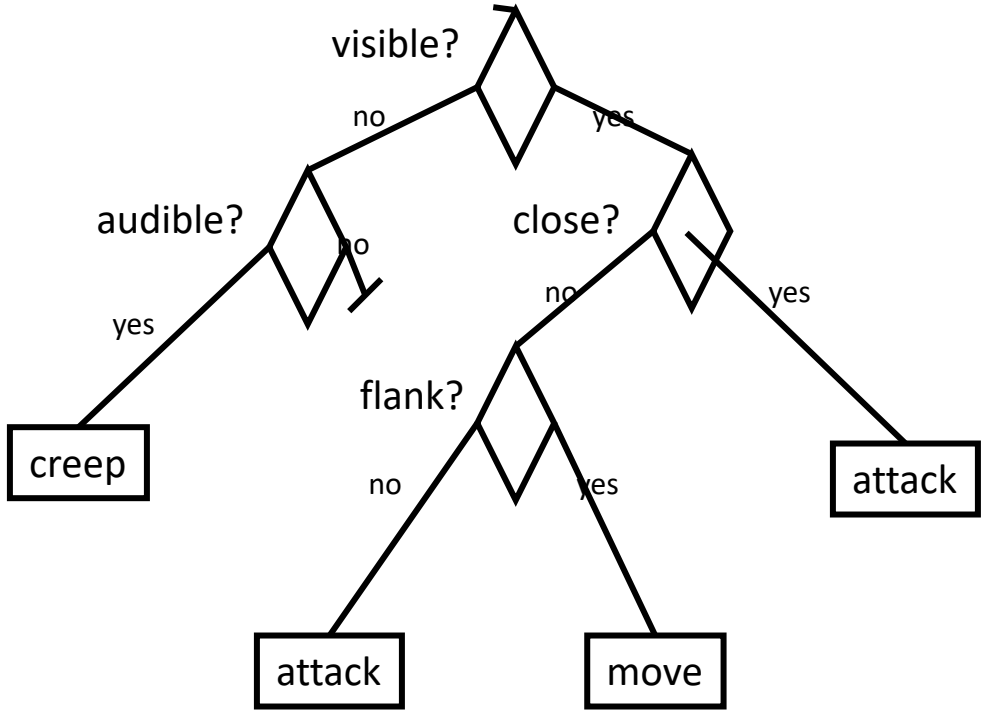


Normalmente **binarios**
(si tiene varias opciones, se puede convertir en binario)

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!

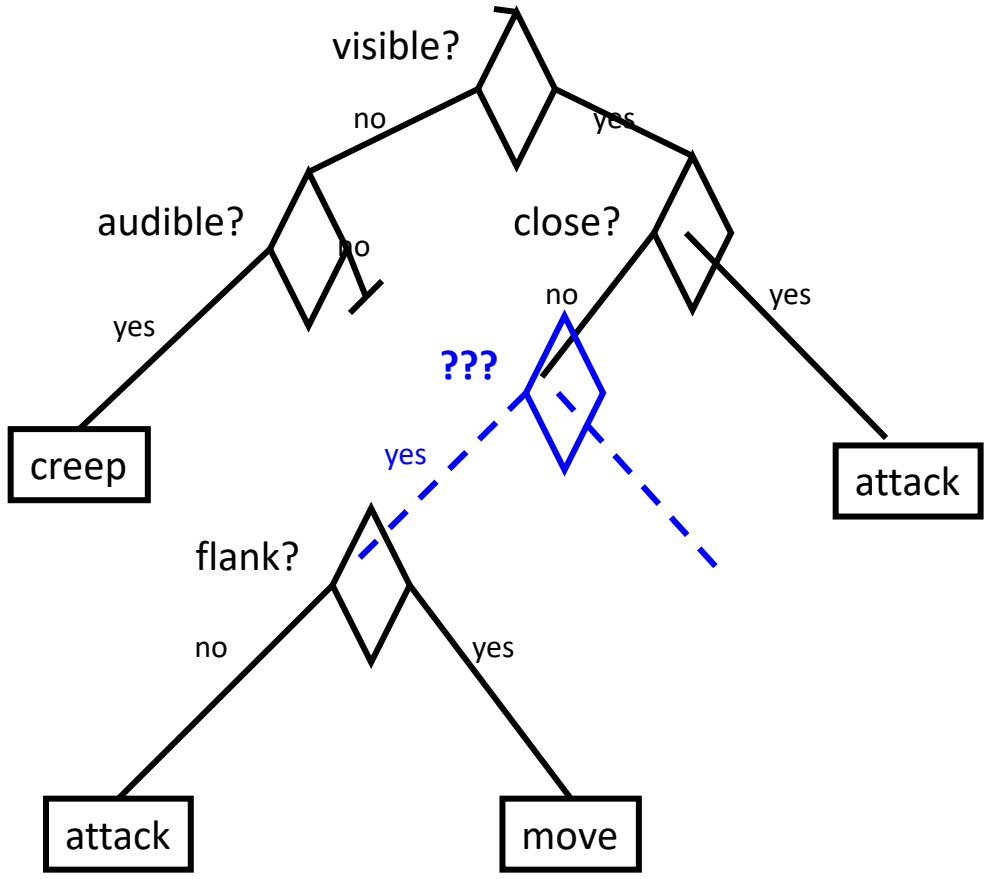
¿Cómo modificarlo?
e.g., if *close*, only flank if ally near

???

Modificación del comportamiento

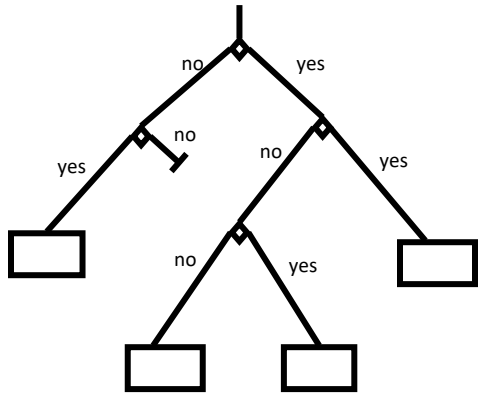
```
if visible? { // level 0
   if close? { // level 1
      attack;
   } else if flank? { // level
1&2
      move;
   } else {
      attack;
   }
} else if audible? { // level
0&1
  creep;
}
```

???



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

Árboles de Decisión OO (Pseudo-Código)



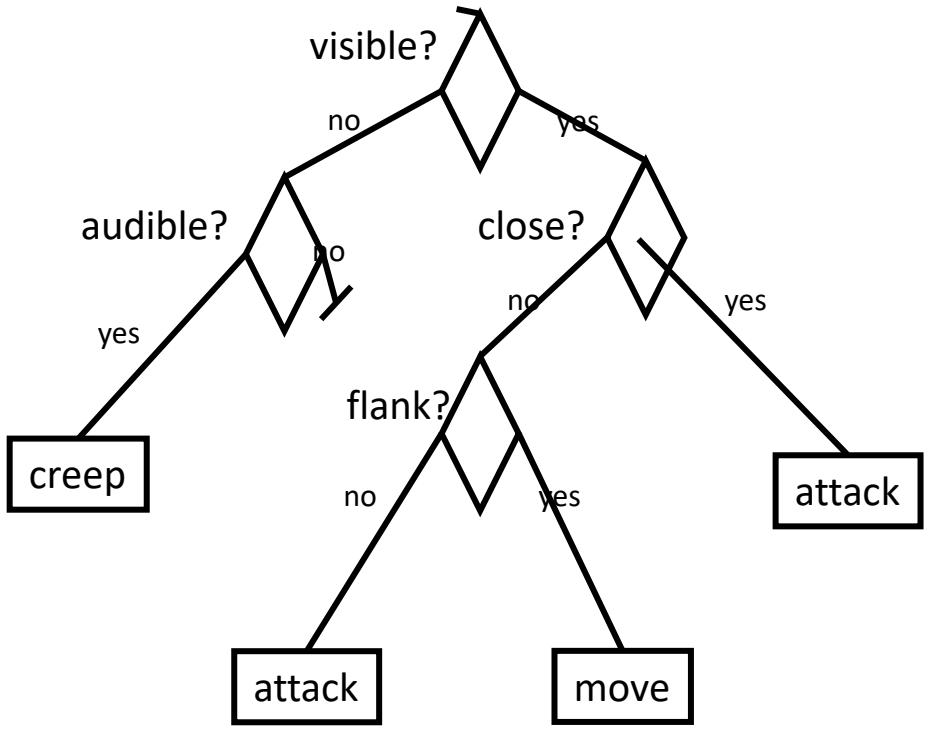

```
class Node
  def decide() // return action/decision


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

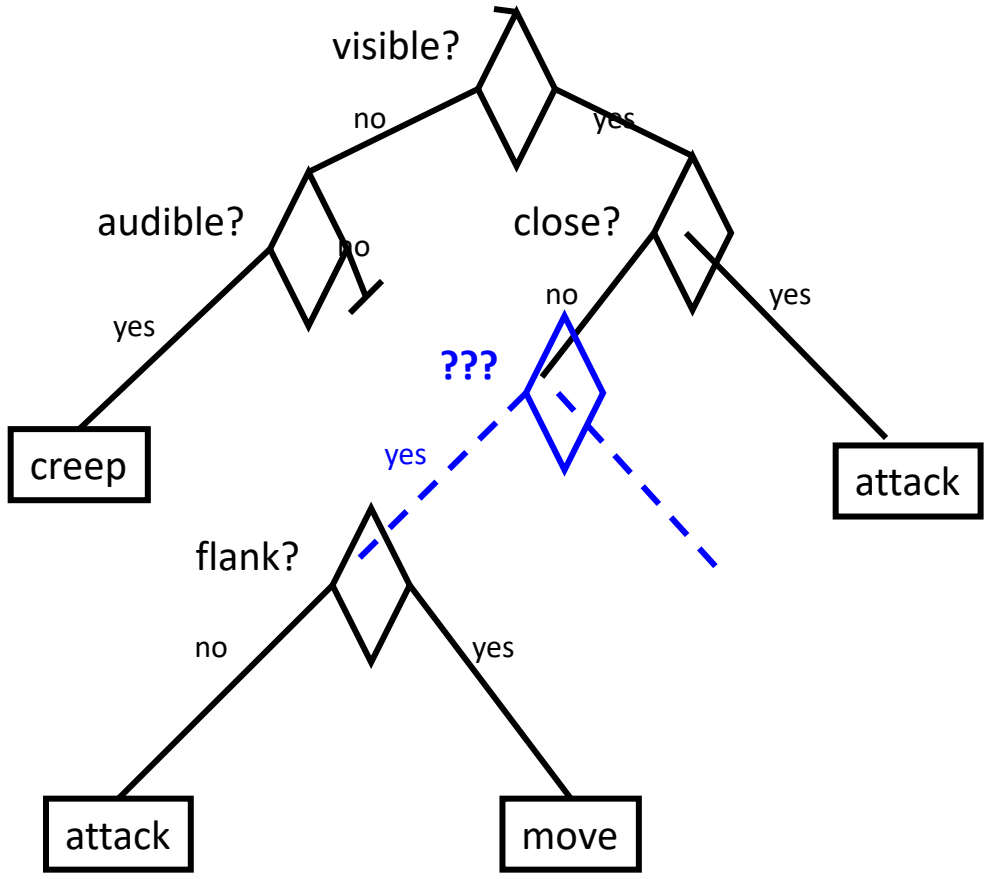
```
visible = new Boolean...
audible = new Boolean...
close = new MinMax...
flank = new Boolean...

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
...
```

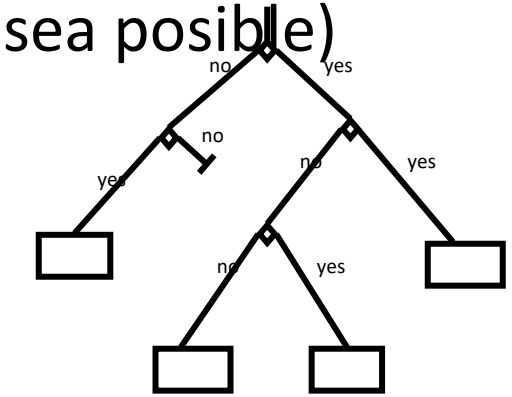


Funcionamiento de un Árbol de Decisión

- 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)

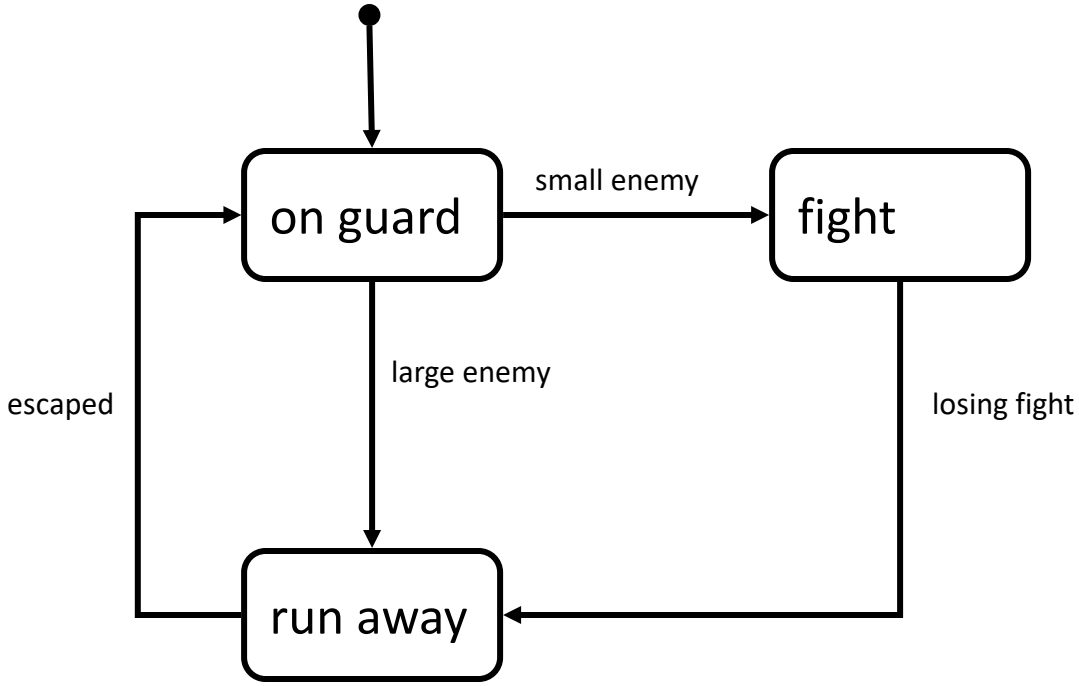
Máquinas Finitas de Estados (FSM)

- 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)

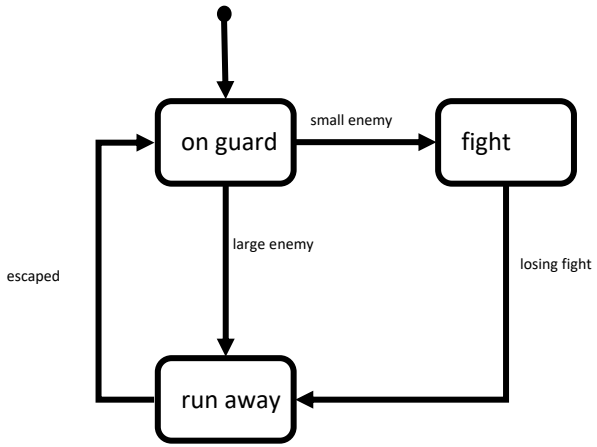
Máquina Finita de Estados

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



Implementación

```
class Soldier

   enum State
      ON_GUARD
      FIGHT
      RUN_AWAY

   currentState
```



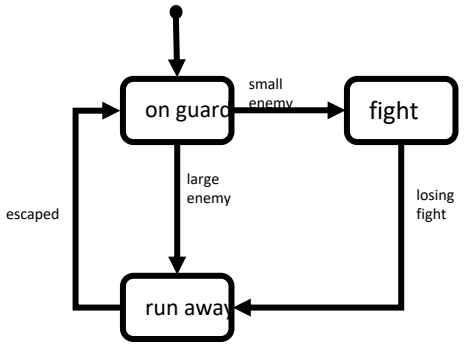

```
def update()
   if currentState == ON_GUARD {
      if small enemy {
         currentState = FIGHT
         start Fighting
      } else if big enemy {
         currentState = RUN_AWAY
         start RunningAway
      }
   } else if currentState == FIGHT {
      if losing fight {
         currentState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

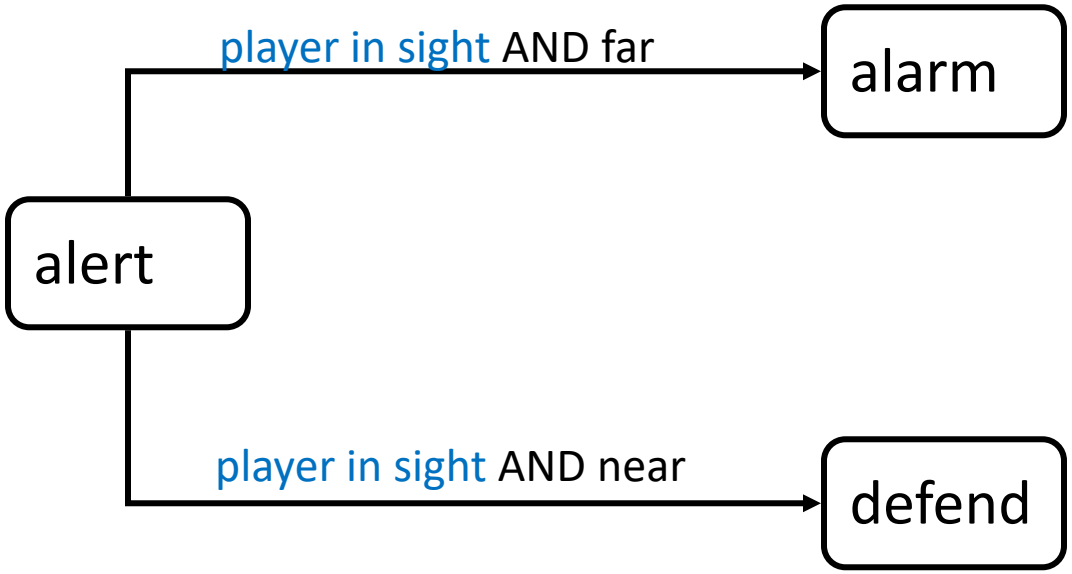
```
class State
   def getAction()
   def getEntryAction()
   def getExitAction()
   def getTransitions()

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
```

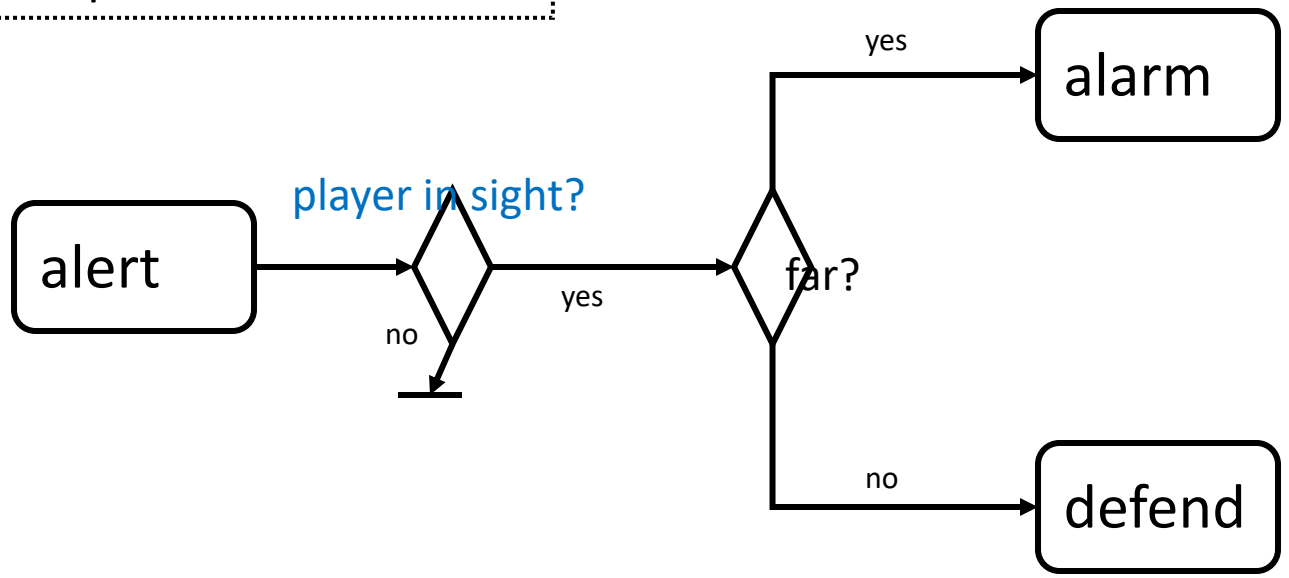
Combinar Árboles de Decisión y Máquinas de Estados (1 de 2)

- ¿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

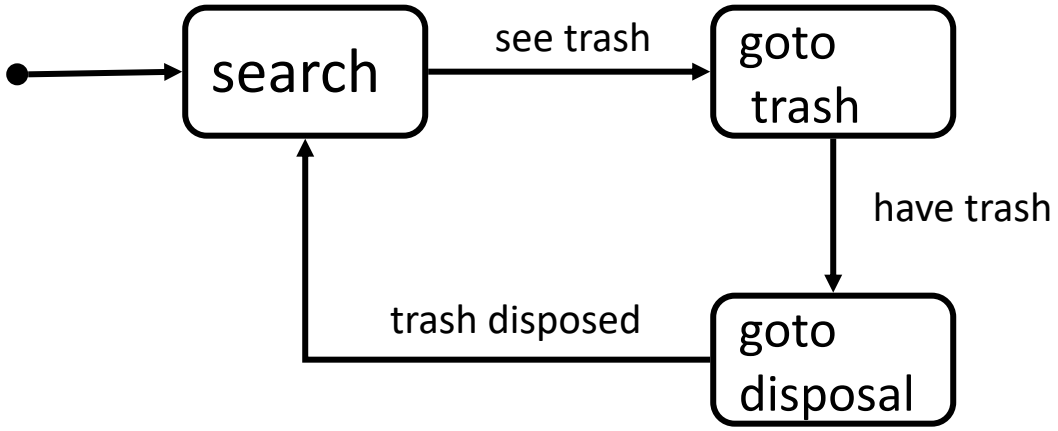


Esquema

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

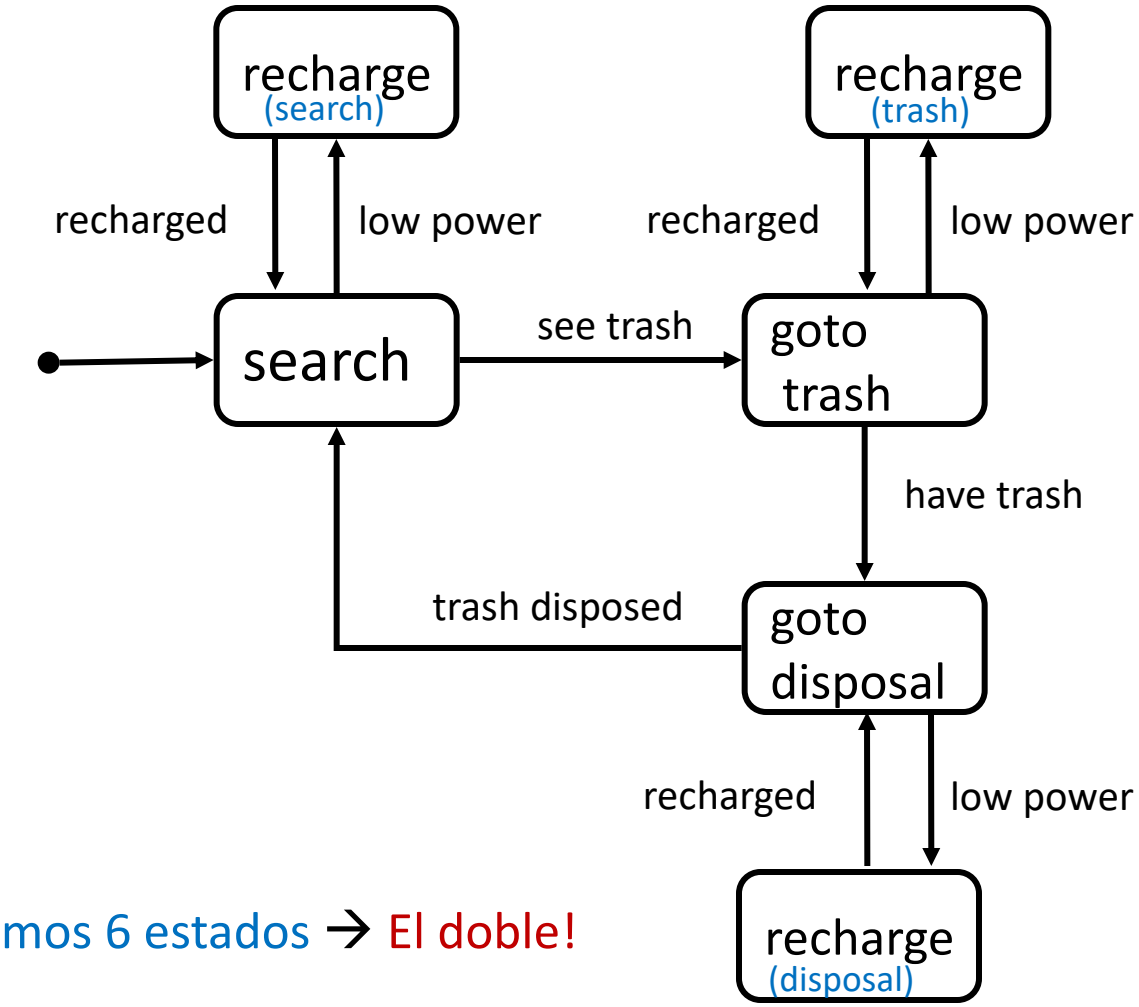
Máquinas de Estados *Jerárquicas*

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



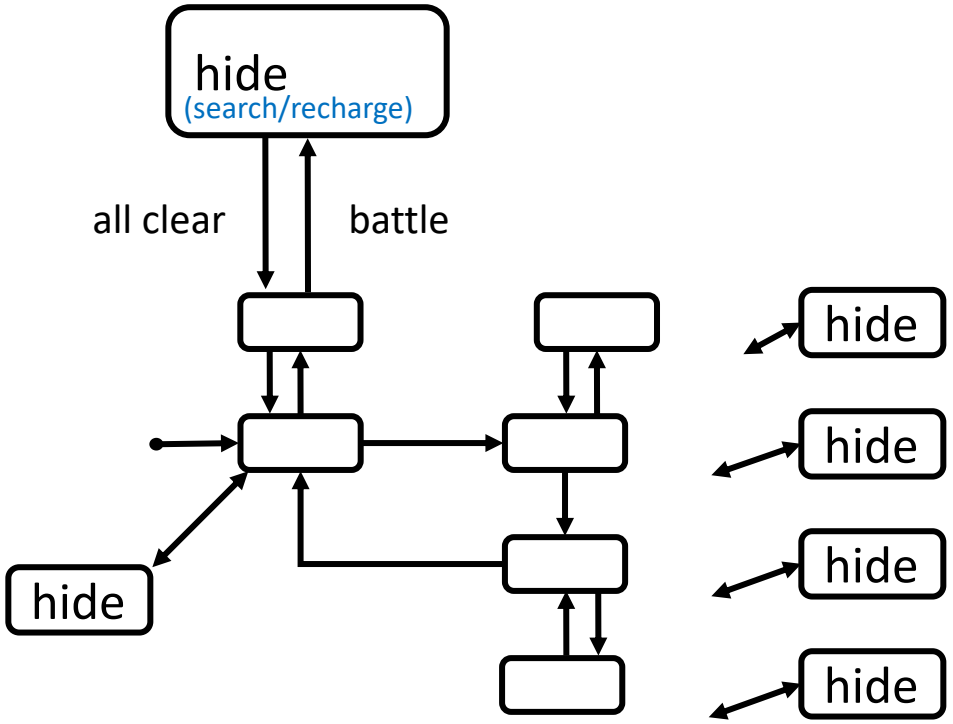
e.g., El robot puede quedarse sin batería en cualquier estado.
Necesitará recargar la batería.
Cuando haya recargado, necesita volver al Estado anterior
e.g., puede tener basura o saber dónde hay basura.

Interrupciones (e.g., Recargar)



Necesitamos 6 estados → El doble!

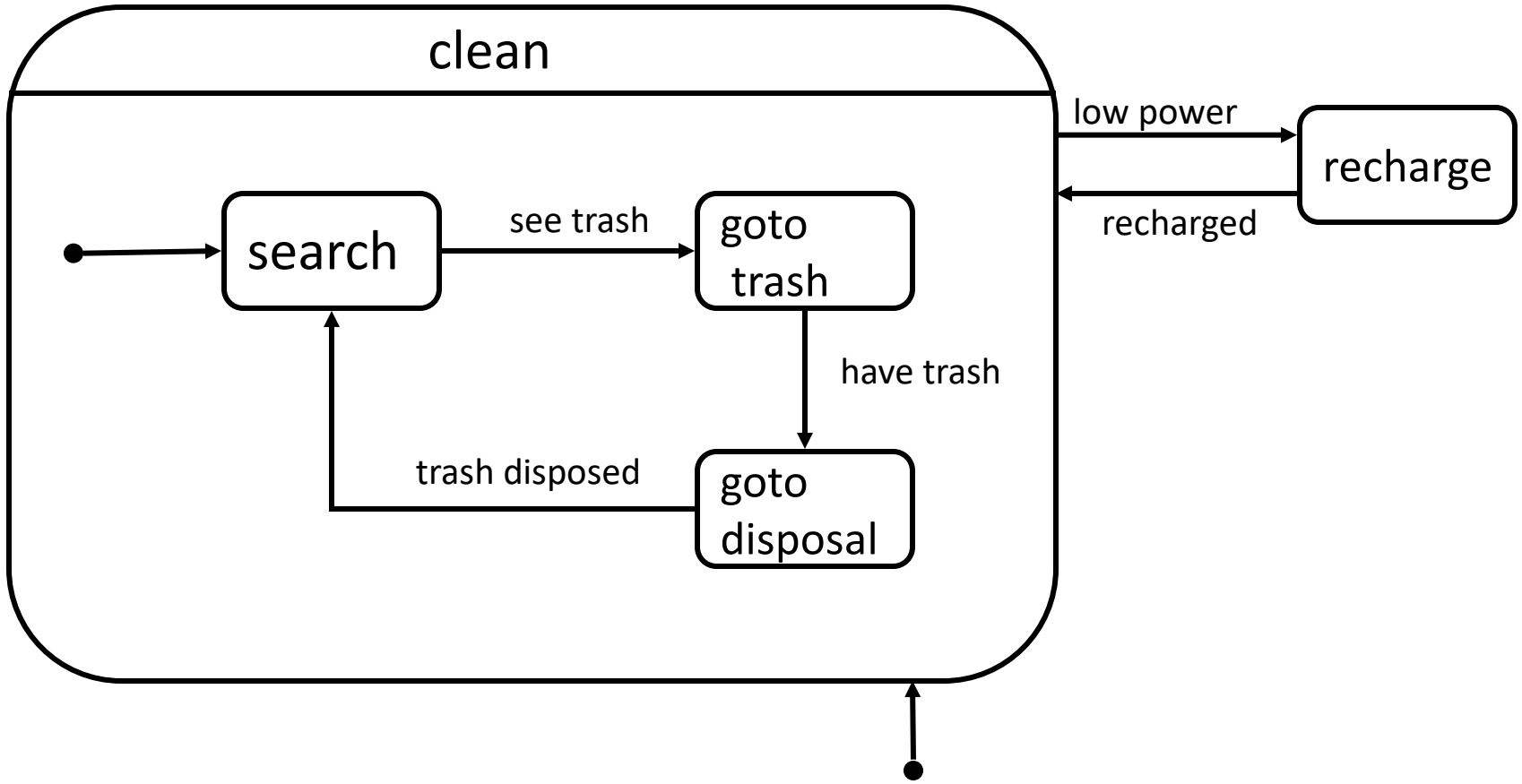
Añadir otra Interrupción (e.g., Enemigos)



Necesitamos 12 estados→ Otra vez el doble!

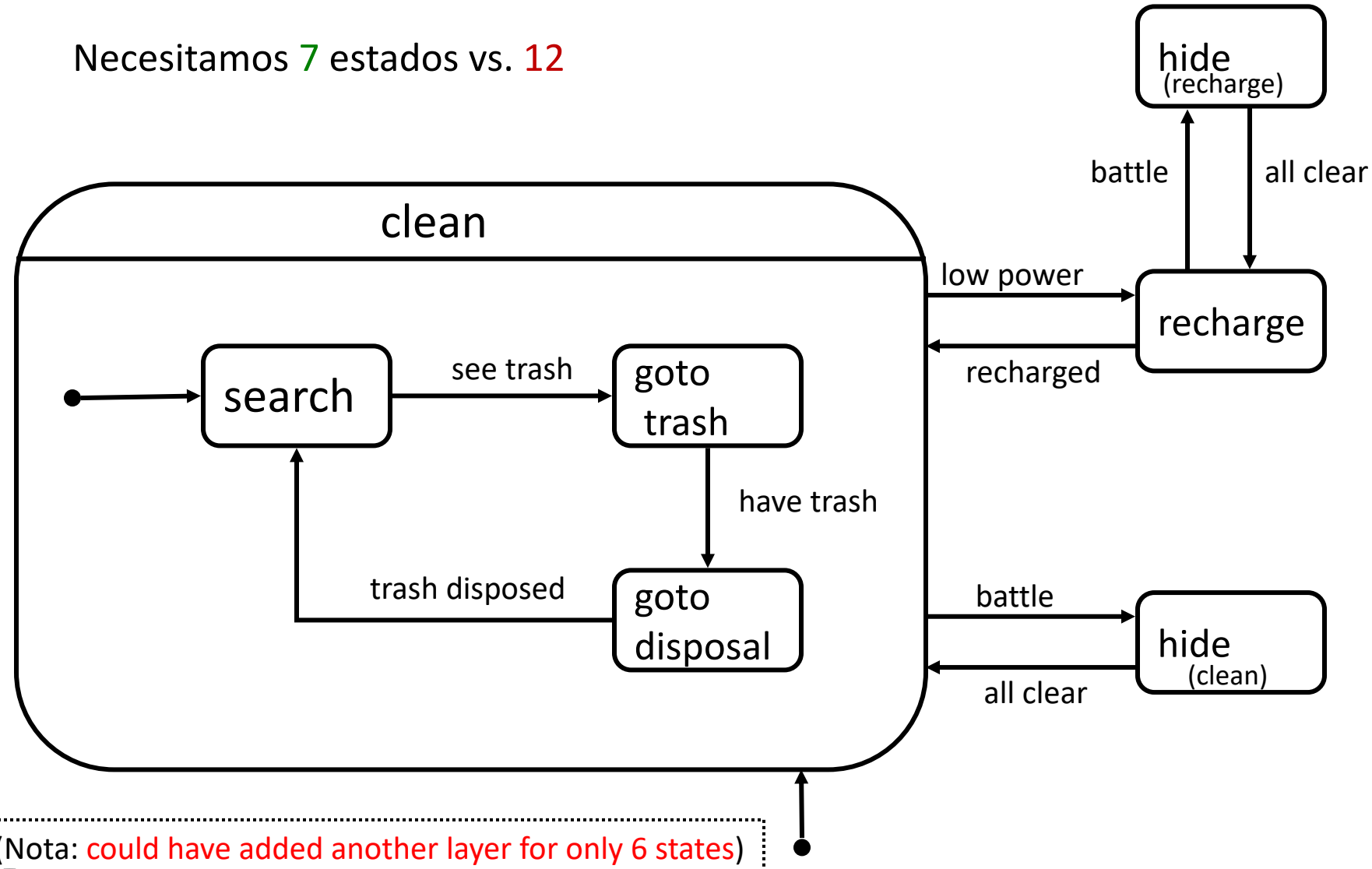
Máquina de Estados *Jerárquica*

- 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)

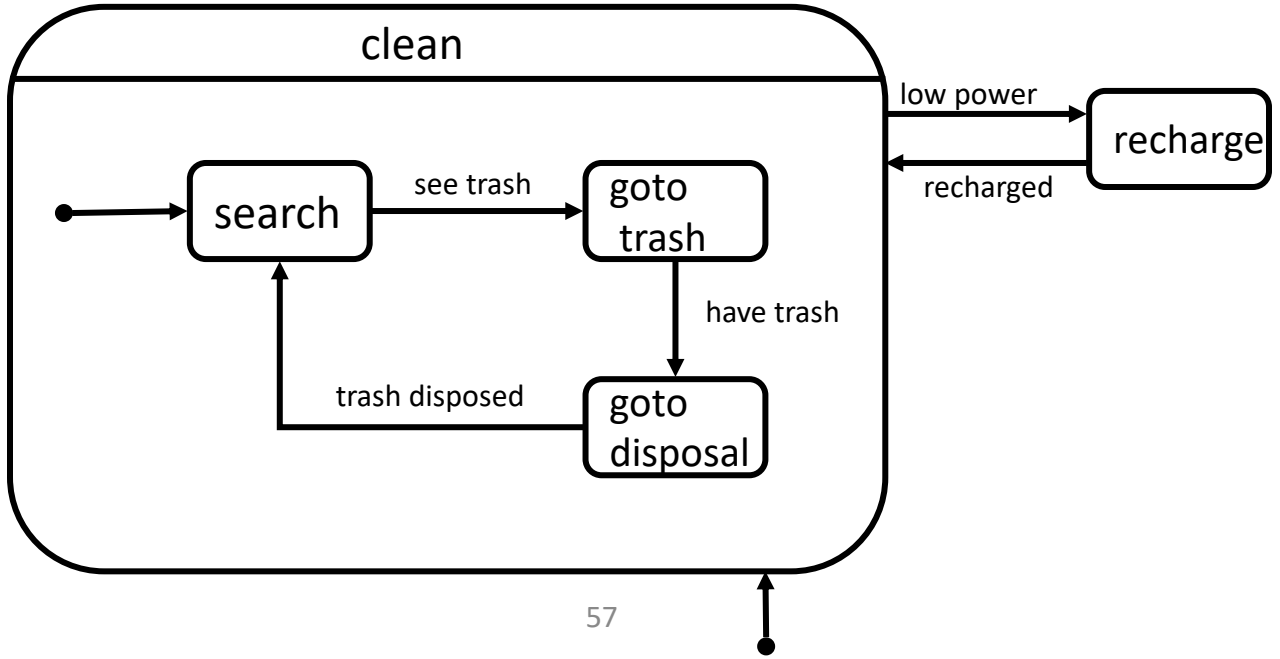
Necesitamos 7 estados vs. 12



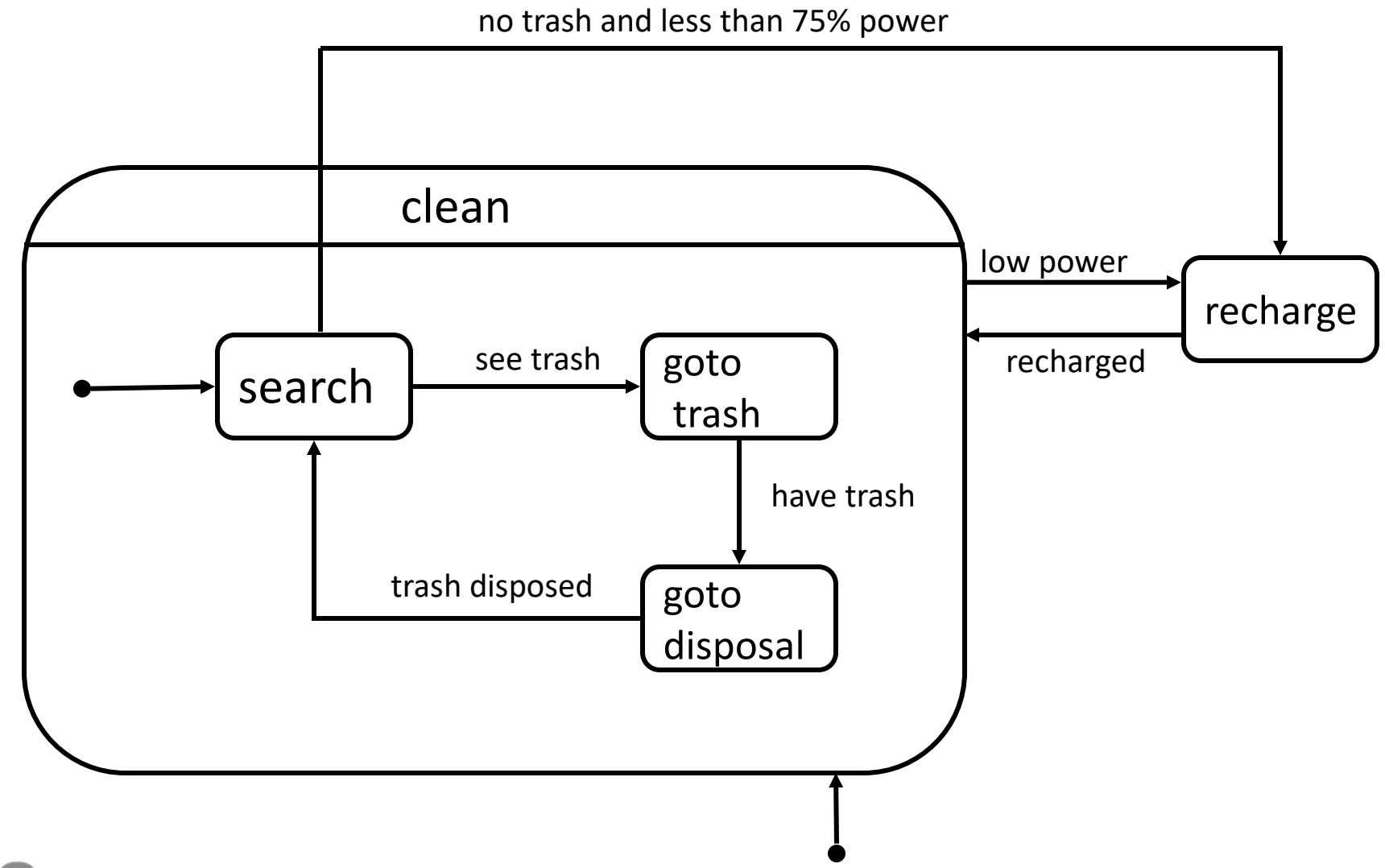
(Nota: could have added another layer for only 6 states)

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

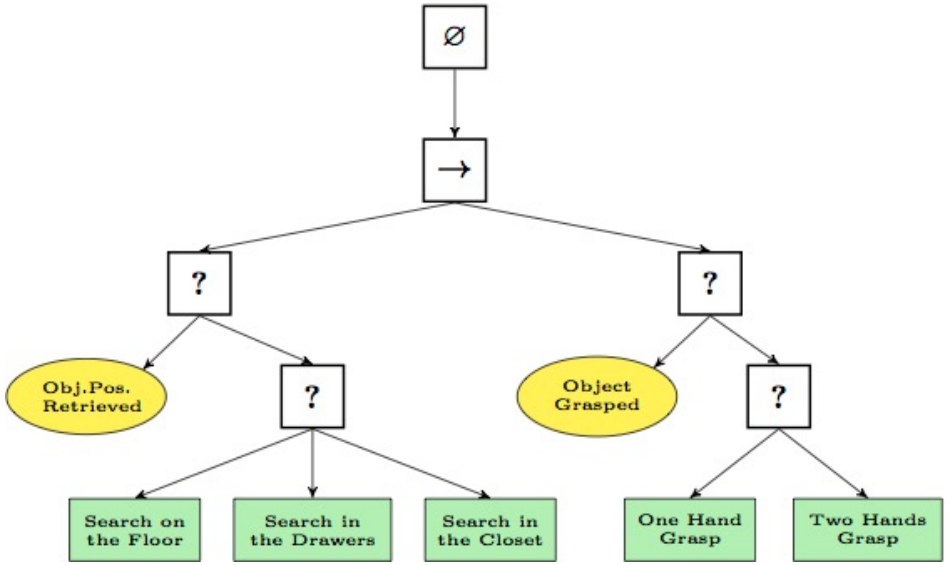
Esquema

- 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

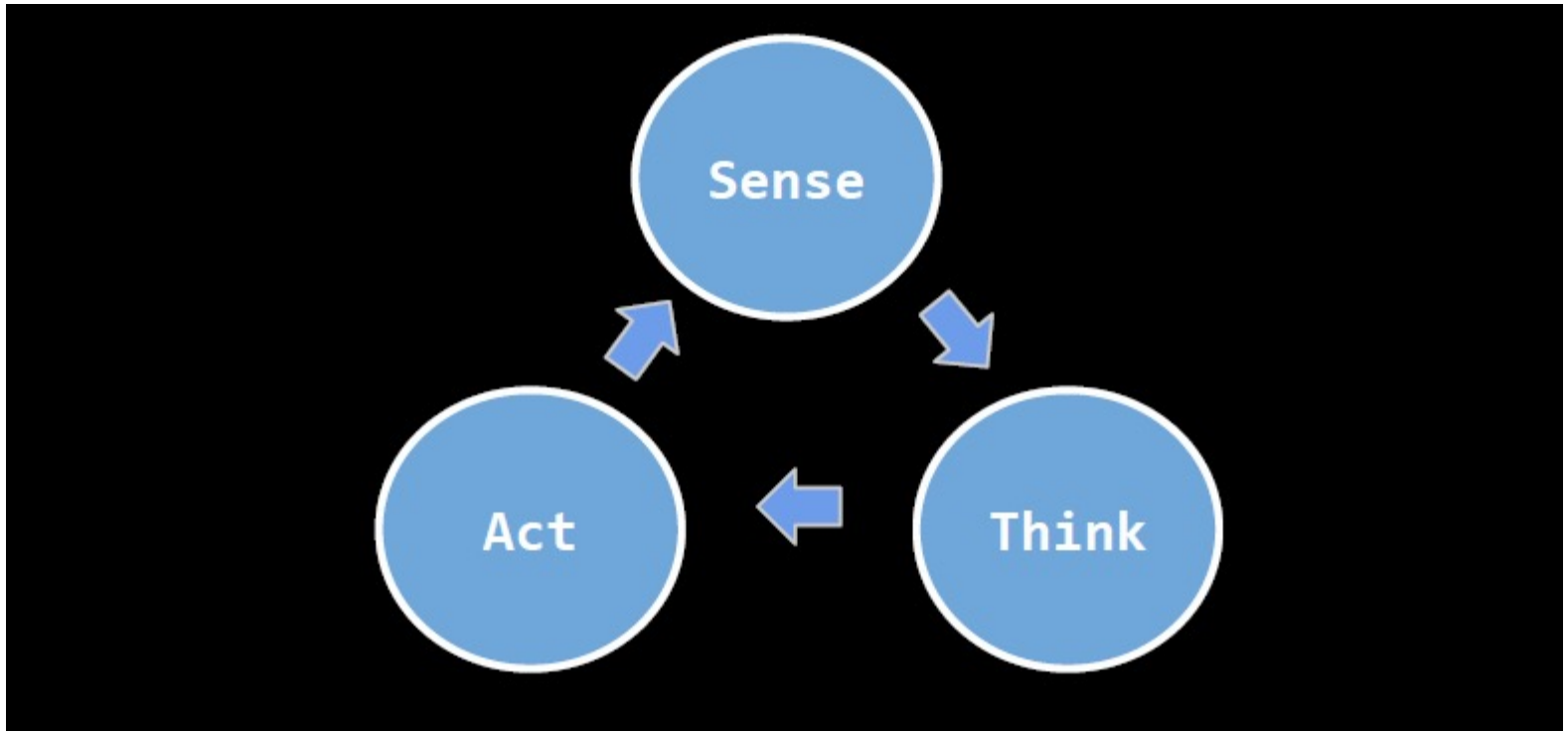
- Son un modelo de ejecución de una planificación
  - Cambia entre tareas modularmente
- Parecido a HFSM, pero los bloques son Tareas, en lugar de Estados
- Usado para NPCs (*Halo*, *Bioshock*, *Spore*)
- Árbol – Los nodos son *Raíz*, *Flujo de Control*, *Ejecución*



Search and grasp plan of two-armed robot

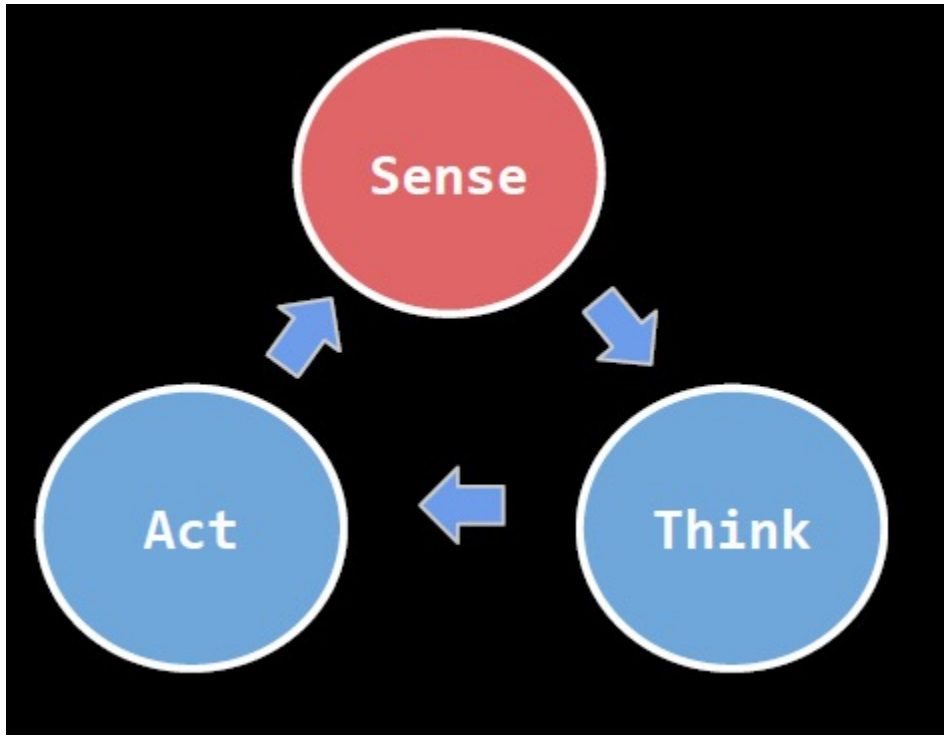
https://upload.wikimedia.org/wikipedia/commons/1/1b/BT_search_and_grasp.png

“Comportamiento”
Árboles de Comportamiento



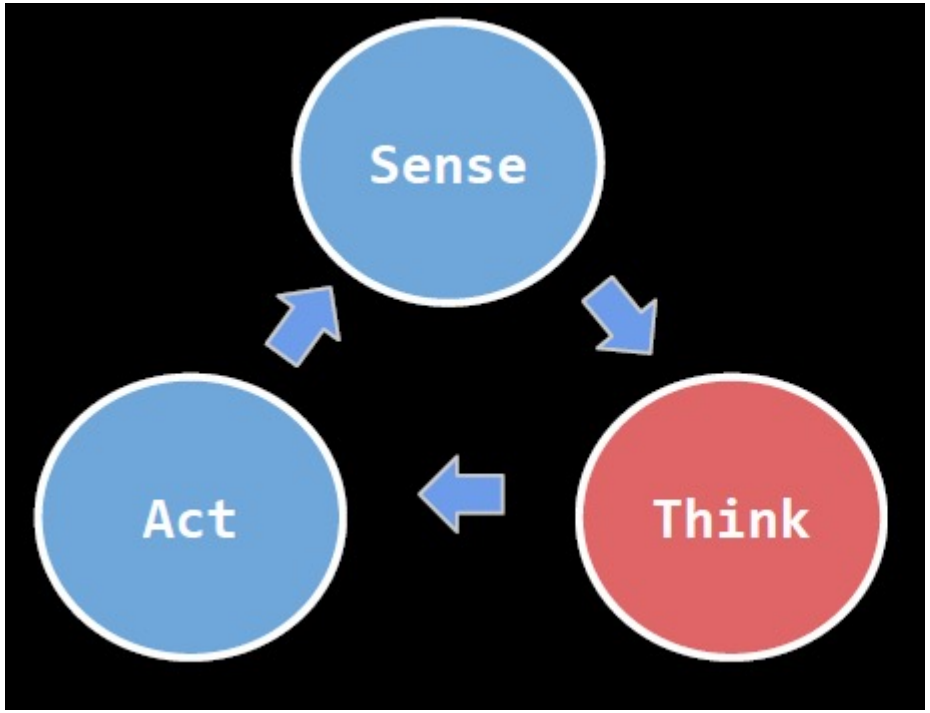
- Percibir, Pensar, Actuar
- Repetir

Percibir



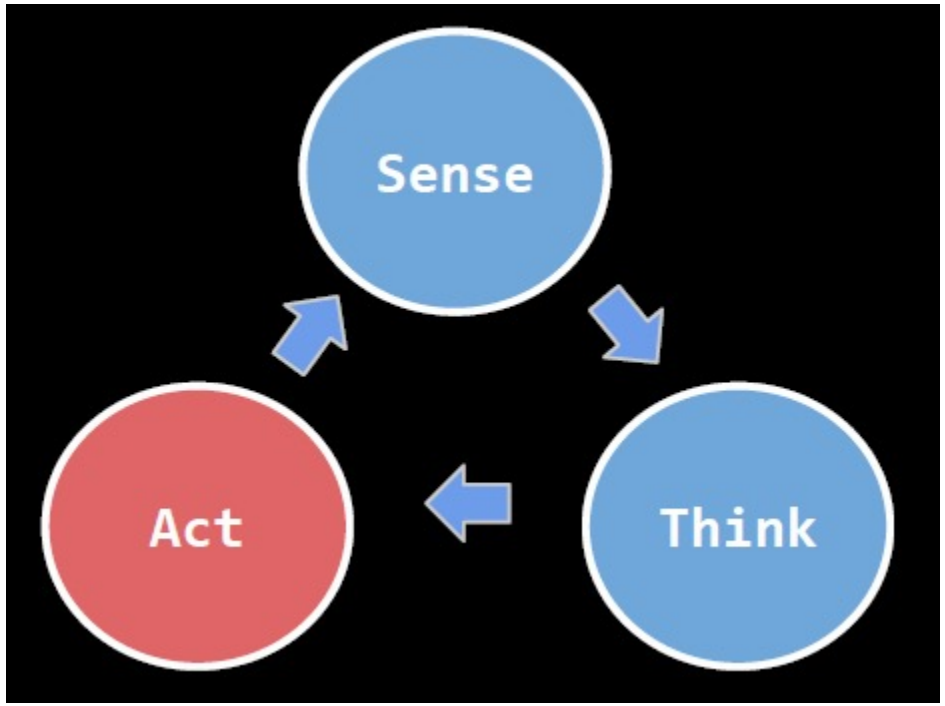
- Generalmente depende del motor de Físicas.
- Normalmente es costosa.
- No se usa en exceso.

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/