

Árboles de decisión

Un árbol de decisión es una estructura que representa de forma jerárquica todas las posibles decisiones que puede tomar un agente (por ejemplo, un personaje controlado por IA) y las consecuencias de cada una. Cada nodo del árbol corresponde a una condición o pregunta; sus ramas son las respuestas posibles y los nodos hoja indican la acción que se ejecutará.

Ejemplo:

```
¿Ve al jugador?  
├── Sí → ¿Está cerca?  
│   ├── Sí → Huir  
│   └── No → Seguir  
└── No → Patrullar
```

¿Cómo se modela?

Podemos modelarlo utilizando la estructura de control if-then-else. Por ejemplo:

```
# Primer nivel del árbol: ¿Ve al jugador?  
if ve_al_jugador():  
    # Segundo nivel del árbol: ¿Está cerca?  
    if esta_cerca():  
        accion = "Huir"  
    else:  
        accion = "Seguir"  
else:  
    # Rama alternativa del primer nivel  
    accion = "Patrullar"
```

Salvo en NPCs con comportamientos muy simples, usar if-then-else directamente, tiene importantes inconvenientes.

- **Mantenimiento difícil:** A medida que el NPC necesita más comportamientos, el código se convierte en una maraña ilegible (el famoso “código espagueti”).
- **Rigidez:** Cualquier cambio en la lógica requiere modificar el código fuente y recompilar. Los diseñadores no pueden tocarlo.
- **Falta de reutilización:** Si quieres que otro NPC use el mismo árbol con ligeras modificaciones, tienes que copiar y pegar el bloque de código y adaptarlo.

En lugar de usar if-then-else directamente, en el programa se puede modelar el árbol como un conjunto de objetos que representan nodos (decisiones) y hojas (acciones). Esto permite:

- Modificar el árbol sin tocar el código, incluso usando herramientas gráficas sin necesidad de saber programar.
- Reutilizar la lógica para diferentes NPCs.

- Visualizar y exportar el árbol como imagen o código.

Enfoque Orientado a Objetos

En el enfoque Orientado a Objetos (Nodos en Código) se crea la estructura del árbol en memoria. Se definen clases para los nodos (ej. `NodoDecision` para las condiciones y `Hoja` para las acciones). El árbol se construye instanciando y conectando estos objetos mediante código.

Clase `NodoDecision`

Evalúa una **condición** y decide si ir por la rama Sí o No.

Está formada por:

- `condicion`: una **función** que devuelve `True` o `False`.
- `true_case`: a dónde ir si la condición es `True`.
- `false_case`: a dónde ir si es `False`.

Clase `Hoja`

Representa una acción final que se ejecuta.

Está formada por:

- `accion`: una **función** que modifica el comportamiento del NPC.

¿Cómo se construye un árbol?

Imaginemos que queremos que un NPC haga esto:

¿Ve al jugador?

└ Sí → Seguir al jugador
└ No → Patrullar borde

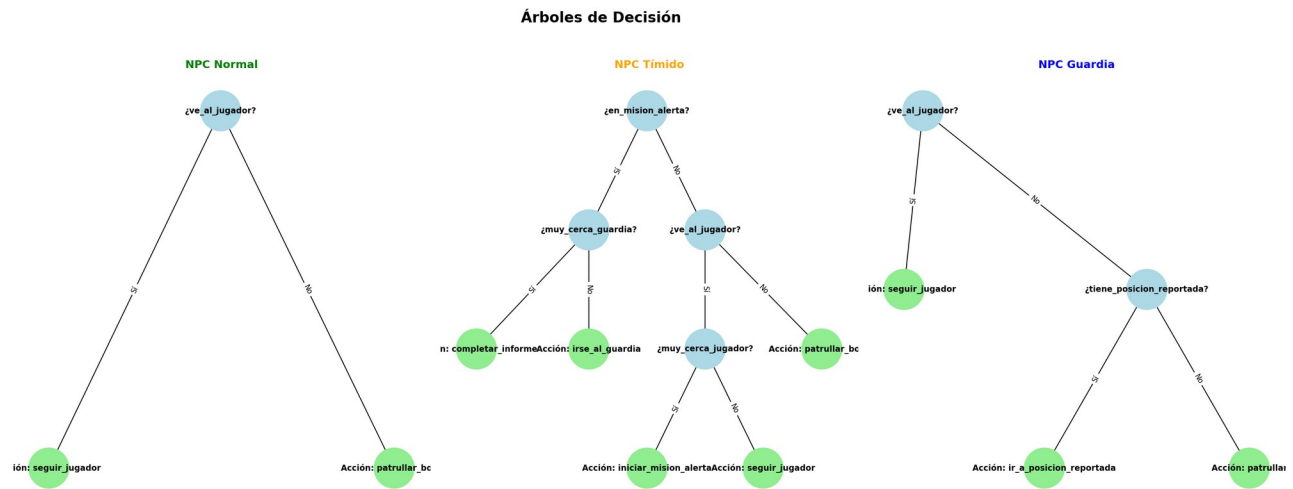
Se construye así:

```
# Creamos las acciones
# seguir_jugador es una función python que mueve el NPC hacia el jugador
accion_seguir = Hoja(seguir_jugador)
# patrullar_borde es una función python que mueve el NPC por los bordes del escenario
accion_pasear = Hoja(patrullar_borde)

# Creamos el nodo raíz
# ve_al_jugador es una función python que calcula si el NPC tiene línea de visión con el jugador.
nodo_raiz = NodoDecision(ve_al_jugador)
nodo_raiz.true_case = accion_seguir
nodo_raiz.false_case = accion_pasear
```

Ejemplo

En esta imagen se muestra el comportamiento de tres NPCs distintos modelados usando árboles de decisión. Los nodos de condición aparecen en azul mientras que los nodos de acción aparecen en verde.



Los tres árboles de decisión representados en formato de texto serían:

● NPC Normal

es el más simple: solo decide entre perseguir o patrullar según si detecta al jugador.

```
¿ve_al_jugador?  
├ Sí → Acción: seguir_jugador  
└ No → Acción: patrullar_borde
```

● NPC Tímido

Es el más complejo (3 niveles de profundidad). Su comportamiento cambia drásticamente dependiendo de si está en “misión de alerta” (buscando al guardia para reportar la posición del jugador):

1. Si **está en alerta**, prioriza su seguridad (huye hacia un guardia o completa un informe).
2. Si **no está en alerta**, se comporta de forma más agresiva (inicia la alerta si el jugador está cerca, o lo sigue si está lejos).

```
¿en_mision_alerta?  
├ Sí → ¿muy_cerca_guardia?  
│   ├── Sí → Acción: irse_al_guardia  
│   └ No → Acción: completar_informe  
└ No → ¿ve_al_jugador?  
    ├── Sí → ¿muy_cerca_jugador?  
    │   ├── Sí → Acción: iniciar_mision_alerta  
    │   └ No → Acción: seguir_jugador  
    └ No → Acción: patrullar_borde
```

□ NPC Guardia

tiene un comportamiento intermedio. Si no ve al jugador, actúa como un “investigador”: va a la última posición reportada del jugador si tiene esa información, o vuelve a patrullar si no.

```
¿ve_al_jugador?  
└ Sí → Acción: seguir_jugador  
└ No → ¿tiene_posicion_reportada?  
    └ Sí → Acción: ir_a_posicion_reportada  
    └ No → Acción: patrullar
```

Puedes ver y ejecutar el ejemplo en este [script de python](#)

Árboles dirigidos por datos (data-driven)

Es un caso muy similar al enfoque orientado a objetos, pero en este caso el árbol se define en un archivo de datos externo (como un JSON, XML o un formato binario personalizado). El juego carga este archivo en tiempo de ejecución (o en tiempo de carga) y un parser lo convierte en la estructura de Objetos mencionada en el enfoque anterior.

Por ejemplo, el NPC tímido se representaría de esta forma

```
{  
  "type": "decision",  
  "condition": "en_mision_alerta",  
  "true_branch": {  
    "type": "decision",  
    "condition": "muy_cerca_guardia",  
    "true_branch": {  
      "type": "action",  
      "action": "irse_al_guardia"  
    },  
    "false_branch": {  
      "type": "action",  
      "action": "completar_informe"  
    }  
  },  
  "false_branch": {  
    "type": "decision",  
    "condition": "ve_al_jugador",  
    "true_branch": {  
      "type": "decision",  
      "condition": "muy_cerca_jugador",  
      "true_branch": {  
        "type": "action",  
        "action": "iniciar_mision_alerta"  
      },  
      "false_branch": {  
        "type": "action",
```

```

        "action": "seguir_jugador"
    }
},
"false_branch": {
    "type": "action",
    "action": "patrullar_bc"
}
}
}

```

y el código en python para leer el json y crear el árbol sería:

```

def construir_arbol(nodo_json):
    """
    Lee un diccionario JSON y devuelve una instancia de
    NodoDecision o Hoja.
    """
    tipo = nodo_json.get("type")

    if tipo == "action":
        # Caso base: Es una hoja. Buscamos la función en el
        registro.
        nombre_accion = nodo_json["action"]
        funcion_accion = REGISTRO_ACCIONES[nombre_accion]
        return Hoja(funcion_accion)

    elif tipo == "decision":
        # Caso recursivo: Es un nodo de decisión.
        nombre_condicion = nodo_json["condition"]
        funcion_condicion = REGISTRO_CONDICIONES[nombre_condicion]

        nodo = NodoDecision(funcion_condicion)

        # Construimos las ramas recursivamente
        nodo.true_case = construir_arbol(nodo_json["true_branch"])
        nodo.false_case =
construir_arbol(nodo_json["false_branch"])

        return nodo
    else:
        raise ValueError(f"Tipo de nodo desconocido: {tipo}")

```

y se construiría con

```

def crear_arbol(self, datos_json):
    arbol = construir_arbol(datos_json)
    return arbol

```

Tracedump:

newBaseSize: 12pt

newBaseSizeInPt: 12