Listado 3.56: Clase PhPaperBin (Fragmento).

```
1  class PhPaperBin extends PhSprite {
2    var _idBase:Int;
3    // Create Shape ==================================================
4    override function createShape():Shape {
5      // Borde izquierdo de la papelera
6      _world.addStaticShape(
7          new phx.Polygon ([new phx.Vector(6+x,9+y),
8             new phx.Vector(14+x,62+y), new phx.Vector(20+x,62+y),
9        new phx.Vector(12+x,9+y)], new phx.Vector(-32,-32)));
10     // Borde derecho de la papelera
11     _world.addStaticShape(
12         new phx.Polygon ([new phx.Vector(52+x,9+y),
13       new phx.Vector(45+x,62+y), new phx.Vector(51+x,62+y),
14       new phx.Vector(58+x,9+y)], new phx.Vector(-32,-32)));
15     // Base de la papelera
16     var saux = new phx.Polygon ([new phx.Vector(19+x,52+y),
17       new phx.Vector(20+x,62+y), new phx.Vector(45+x,62+y),
18       new phx.Vector(46+x,52+y)],new phx.Vector(-32,-32));
19     _idBase = saux.id;
20     _world.addStaticShape(saux);
21     return null;
22   }
23   // getIdBase =====================================================
24   public function getIdBase():Int {return _idBase;}
25   // Constructor ===================================================
26   public function new(id:String, l:TileLayer, w:World, px:Float,
27               py:Float):Void {
28     super(id, l, w, px, py, false);
29   }
30 }
```

3.5. Inteligencia Artificial

3.5.1. Introducción

La Inteligencia Artificial (IA) es un elemento fundamental para dotar de realismo a un videojuego. Uno de los retos principales que se plantean a la hora de integrar comportamientos inteligentes es alcanzar un equilibrio entre la **sensación de inteligencia** y el tiempo de cómputo empleado por el subsistema de IA. Dicho equilibrio es esencial en el caso de los videojuegos, como exponente más representativo de las aplicaciones gráficas en tiempo real.

Este planteamiento gira, generalmente, en torno a la generación de soluciones que, sin ser óptimas, proporcionen una cierta sensación de inteligencia. En concreto, dichas soluciones deberían tener como meta que el jugador se enfrente a un reto que sea factible de manera que suponga un estímulo emocional y que consiga *engancharlo* al juego.

Figura 3.29: El robot-humanoide *Asimo*, creado por *Honda* en el año 2000, es uno de los exponentes más reconocidos de la aplicación de técnicas de IA sobre un prototipo físico real. Sin embargo, todavía queda mucho por recorrer hasta que los robots puedan acercarse de manera significativa a la forma de actuar de una persona, tanto desde el punto de vista físico como racional.

En los últimos años, la IA ha pasado de ser un componente secundario en el proceso de desarrollo de videojuegos a convertirse en uno de los aspectos más importantes. Actualmente, lograr un alto nivel de IA en un juego sigue siendo **uno de los retos** más emocionantes y complejos y, en ocasiones, sirve para diferenciar un juego normal de uno realmente deseado por los jugadores.

Tal es su importancia, que las grandes desarrolladores de videojuegos mantienen en su plantilla a ingenieros especializados en la parte de IA, donde los lenguajes de *scripting*, como LUA o Python, y la comunicación con el resto de programadores del juego resulta esencial.

En esta sección se ofrece una introducción general de la IA aplicada al desarrollo de videojuegos, haciendo hincapié en conceptos como el de **agente** y en herramientas como las **máquinas de estados**. Posteriormente se discuten dos casos de estudio concretos. El primero se aborda desde el punto de vista teórico y en él se estudia cómo implementar el módulo de IA del Tetris. El segundo se aborda desde una perspectiva práctica y en él se hace uso de NME para llevar a cabo una implementación del famoso *tres en raya*. En concreto, en este último caso se discute una implementación del **algoritmo minimax** para simular el comportamiento de la máquina como adversario virtual.

3.5.2. Aplicando el Test de Turing

La Inteligencia Artificial es un área fascinante y relativamente moderna de la Informática que gira en torno a la construcción de programas inteligentes. Existen diversas interpretaciones para el término *inteligente* (vea [12] para una discusión en profundidad), las cuales se diferencian en función de la similaritud con conceptos importantes como **racionalidad** y **razonamiento**.

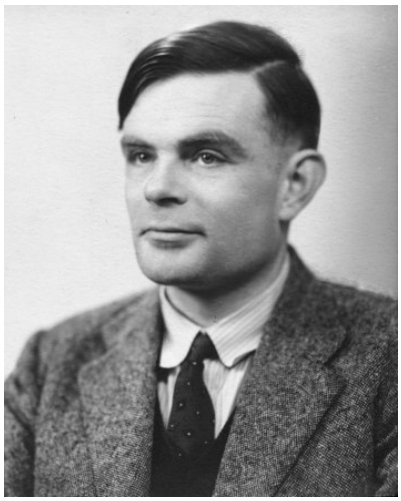

Figura 3.30: Alan Turing (1912-1954), matemático, científico, criptógrafo y filósofo inglés, es considerado uno de los Padres de la Computación y uno de los precursores de la Informática Moderna.

De cualquier modo, una constante en el campo de la IA es la relación entre un programa de ordenador y el comportamiento del ser humano. Tradicionalmente, la IA se ha entendido como la intención de crear programas que actuasen como lo haría una persona ante una situación concreta en un contexto determinado.

Hace más de medio siglo, en 1950, Alan Turing propuso la denominada **Prueba de Turing**, basada en la incapacidad de una persona de distinguir entre hombre o máquina a la hora de evaluar un programa de ordenador. En concreto, un programa pasaría el test si un evaluador humano no fuera capaz de distinguir si las respuestas a una serie de preguntas formuladas eran o no de una persona. Hoy en día, esta prueba sigue siendo un reto muy exigente ya que, para superarlo, un programa tendría que ser capaz de procesar lenguaje natural, representar el conocimiento, razonar de manera automática y aprender.

Además de todas estas funcionalidades, esta prueba implica la necesidad de interactuar con el ser humano, por lo que es prácticamente imprescindible integrar técnicas de visión por computador y de robótica para superar la *Prueba Global de Turing*. Todas estas disciplinas cubren gran parte del campo de la IA, por lo que Turing merece un gran reconocimiento por plantear un problema que hoy en día sigue siendo un reto muy importante para la comunidad científica.

En el ámbito del **desarrollo de videojuegos**, la Prueba de Turing se podría utilizar para evaluar la IA de un juego. Básicamente, sería posible aplicar esta prueba a los *Non-Player Characters* (NPCs) con el objetivo de averiguar si el jugador humano es capaz de saber si son realmente *bots* o podrían confundirse con jugadores reales.

Aunque actualmente existen juegos que tienen un grado de IA muy sofisticado, en términos generales es relativamente fácil distinguir entre

NPC y jugador real. Incluso en juegos tan trabajados desde el punto de vista computacional como el ajedrez, en ocasiones las decisiones tomadas por la máquina delatan su naturaleza.

Desafortunadamente, los desarrolladores de videojuegos están condicionados por el tiempo, es decir, los videojuegos son aplicaciones gráficas en tiempo real que han de generar una determinada tasa de *frames* o imágenes por segundo. En otras palabras, este aspecto tan crítico hace que a veces el tiempo de cómputo dedicado al sistema de IA se vea reducido. La buena noticia es que, generalmente, el módulo responsable de la IA no se tiene que actualizar con la misma frecuencia, tan exigente, que el motor de *rendering*.

Aunque esta limitación se irá solventando con el incremento en las prestaciones hardware de las estaciones de juego, hoy en día es un gran condicionante que afecta a los recursos dedicados al módulo de IA. Una de las consecuencias de esta limitación es que dicho módulo se basa en proporcionar una ***ilusión de inteligencia***, es decir, está basado en un esquema que busca un equilibro entre garantizar la simplicidad computacional y proporcionar al jugador un verdadero reto.

3.5.3. Ilusión de inteligencia

De una manera casi inevitable, el *componente inteligente* de un juego está vinculado a la dificultad o al reto que al jugador se le plantea. Sin embargo, y debido a la naturaleza cognitiva de dicho componente, esta cuestión es totalmente subjetiva. Por lo tanto, gran parte de los desarrolladores opta por intentar que el jugador se sienta inmerso en lo que se podría denominar *ilusión de inteligencia*.

Por ejemplo, en el videojuego *Metal Gear Solid*, desarrollado por Konami y lanzado para PlayStation™en 1998, los enemigos empezaban a mirar de un lado a otro y a decir frases del tipo *¿Quién anda ahí?* si el personaje principal, Solid Snake, se dejaba ver mínimamente o hacía algún ruido en las inmediaciones de los NPCs. En este juego, el espionaje y la infiltración predominaban sobre la acción, por lo que este tipo de elementos generaban una cierta sensación de IA, aunque en realidad su implementación fuera sencilla.

Un caso más general está representado por modificar el **estado de los NPCs**, típicamente incrementando su nivel de *stamina* o vida. De este modo, el jugador puede tener la sensación de que el enemigo es más inteligente porque cuesta más abatirlo. Otra posible alternativa consiste en proporcionar más habilidades al enemigo, por ejemplo haciendo que se mueva más rápido o que dispare con mayor velocidad.

En [2], el autor discute el caso de la IA del juego *Halo*, desarrollado por *Bungie Studios* y publicado por *Microsoft Games Studio* en 2001, y

cómo los desarrolladores consiguieron *engañar* a los *testers* del juego. En concreto, los desarrolladores asociaban el nivel de IA con la altitud de los puntos de impacto sobre los NPCs. Así, los jugadores percibían un grado bajo de IA cuando este nivel no era elevado, es decir, cuando los impactos en la parte inferior del NPC eran relevantes para acabar con el mismo. Sin embargo, al incrementar dicho nivel y forzar a los jugadores a apuntar a partes más elevadas del NPC, éstos percibían que el juego tenía una IA más elevada.

3.5.4. ¿NPCs o Agentes?

En gran parte de la bibliografía del desarrollo de videojuegos, especialmente en la relativa a la IA, el concepto de *agent* (agente) se utiliza para referirse a las distintas entidades virtuales de un juego que, de alguna manera u otra, tienen asociadas un **comportamiento**. Dicho comportamiento puede ser trivial y basarse, por ejemplo, en un esquema totalmente preestablecido o realmente complejo y basarse en un esquema que gire entorno al aprendizaje. De cualquier modo, estas dos alternativas comparten la idea de mostrar algún tipo de inteligencia, aunque sea mínima.

Tradicionalmente, el **concepto de agente** en el ámbito de la IA se ha definido como cualquier entidad capaz de percibir lo que ocurre en el medio o contexto en el que habita, mediante la ayuda de sensores, y actuar en consencuancia, mediante la ayuda de actuadores, generando normalmente algún tipo de cambio en dicho medio o contexto. La figura 3.31 muestra esta idea tan general. Note cómo de nuevo la idea de la Prueba de Turing vuelve a acompañar al concepto de agente.

El concepto de agente se ha ligado a ciertas propiedades que se pueden trasladar perfectamente al ámbito del desarrollo de videojuegos y que se enumeran a continuación:

- **Autonomía**, de manera que un agente actúa sin la intervención directa de terceras partes. Por ejemplo, un personaje de un juego de rol tendrá sus propios deseos, de manera independiente al resto.
- **Habilidad social**, los agentes interactúan entre sí y se comunican para alcanzar un objetivo común. Por ejemplo, los NPCs de un *shooter* se comunicarán para cubrir el mayor número de entradas a un edificio.
- **Reactividad**, de manera que un agente actúa en función de las percepciones del entorno. Por ejemplo, un enemigo reaccionará, normalmente, atacando si es atacado.

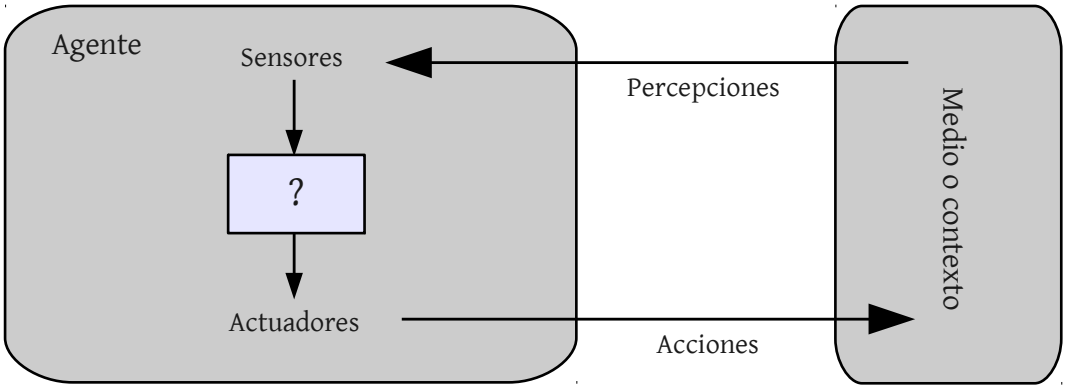

Figura 3.31: Visión abstracta del concepto de agente.

- **Proactividad**, de manera que un agente puede tomar la iniciativa en lugar de ser puramente reactivo. Por ejemplo, un enemigo feroz atacará incluso cuando no haya sido previamente atacado.

De manera adicional a estas propiedades, los conceptos de razonamiento y aprendizaje forman parte esencial del núcleo de un agente. Actualmente, existen juegos que basan parte de la IA de los NPCs en esquemas de aprendizaje y los usan para comportarse de manera similar a un jugador real. Este aprendizaje puede basar en la detección de patrones de comportamiento de dicho jugador.

Un ejemplo típico son los juegos deportivos. En este contexto, algunos juegos de fútbol pueden desarrollar patrones similares a los observados en el jugador real que los maneja. Por ejemplo, si la *máquina* detecta que el jugador real carga su juego por la parte central del terreno de juego, entonces podría contrarrestarlo atacando por las bandas, con el objetivo de desestabilizar al rival.

Aunque los agentes se pueden implementar haciendo uso de una filosofía basada en el diseño orientado a objetos, informalmente se suele afirmar que *los objetos lo hacen gratis, mientras que los agentes lo hacen porque quieren hacerlo.*

Comúnmente, los agentes basan su modelo de funcionamiento interno en una **máquina de estados**, tal y como se muestra de manera gráfica en la figura 3.32. Este esquema se ha utilizado durante muchos años como herramienta principal para proporcionar esa *ilusión de inteligencia* por parte del desarrollador de IA. De hecho, aunque en algunos

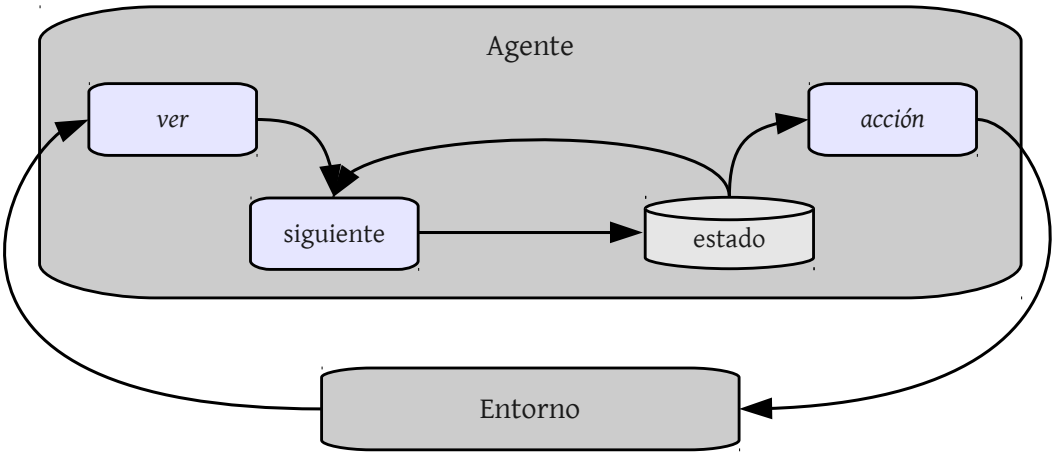

Figura 3.32: Visión abstracta del funcionamiento interno de un agente.

proyectos se planteen arquitecturas mucho más sofisticadas, en la práctica la mayor parte de ellas girarán en torno a la idea que se discute en la siguiente sección.

3.5.5. Diseño de agentes basado en estados

Una máquina de estados define el comportamiento que especifica las secuencias de estados por las que atraviesa un objeto durante su ciclo de ejecución en respuesta a una serie de eventos, junto con las respuestas a dichos eventos. En esencia, una máquina de estados permite descomponer el comportamiento general de un agente en *pedazos* o subestados más manejables. La figura 3.33 muestra un ejemplo concreto de máquinas de estados, utilizada para definir el comportamiento de un NPC en base a una serie de estados y las transiciones entre los mismos.

Como el lector ya habrá supuesto, los conceptos más importantes de una máquina de estados son dos: los estados y las transiciones. Por una parte, un **estado** define una condición o una situación durante la vida del agente, la cual satisface alguna condición o bien está vinculada a la realización de una acción o a la espera de un evento. Por otra parte, una **transición** define una relación entre dos estados, indicando lo que ha de ocurrir para pasar de un estado a otro. Los cambios de estado se producen cuando la transición se *dispara*, es decir, cuando se cumple la condición que permite pasar de un estado a otro.

Aunque la idea general de las máquinas de estado es tremendamente sencilla, su popularidad en el área de los videojuegos es enorme debido a los siguientes factores [2]:

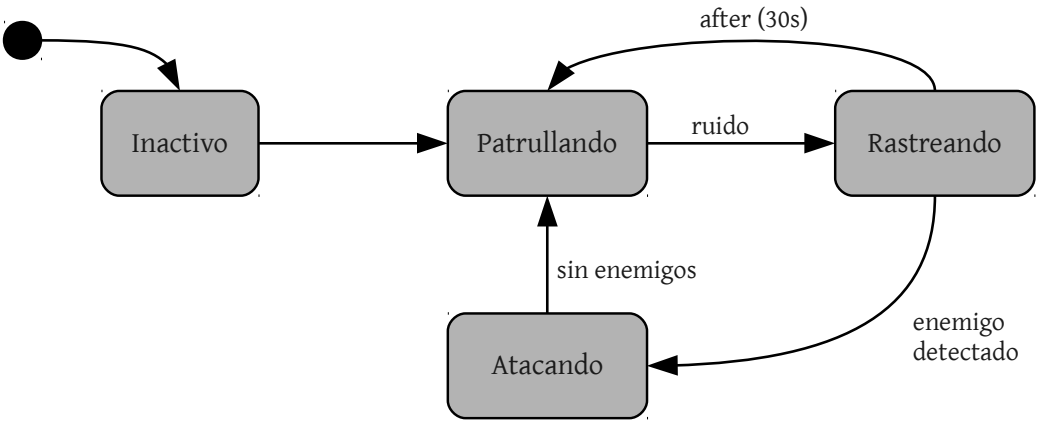

Figura 3.33: Máquina de estados que define el comportamiento de un NPC.

- Son **fáciles** de implementar y muy **rápidas**. Aunque existen diversas alternativas, todas ellas tienen una complejidad baja.
- Su **depuración** es sencilla, ya que se basan en el principio de descomposición en subestados que sean manejables.
- Tienen una **mínima sobrecarga computacional**, ya que giran en torno a un esquema *if-then-else*.
- Son muy **intuitivas**, ya que se asemejan al modelo de razonamiento del ser humano.
- Son muy **flexibles**, debido a que permiten la integración de nuevos estados sin tener un impacto significativo en el resto y posibilitan la combinación de otras técnicas clásicas de IA, como la lógica difusa o las redes neuronales.

Una máquina de estados se puede implementar utilizando distintas aproximaciones, como por ejemplo la que se estudió en la sección 3.1.4, para dar soporte al sistema de gestión de estados. En este contexto, es bastante común hacer uso del polimorfismo para manejar distintos estados que hereden de uno más general, proporcionando distintas implementaciones en función de dicha variedad de estados. En esencia, la idea consiste en mantener la interfaz pero concretando la implementación de cada estado. Normalmente, también se suele hacer uso del patrón *singleton* para manejar una única instancia de cada estado.

3.5.6. Búsqueda entre adversarios

En el ámbito de la IA muchos problemas se pueden resolver a través de una estrategia de búsqueda sobre las distintas soluciones existentes. De este modo, el razonamiento se puede reducir a llevar a cabo un proceso de búsqueda. Por ejemplo, un algoritmo de planificación se puede plantear como una búsqueda a través de los árboles y sub-árboles de objetivos para obtener un camino hasta el objetivo deseado (definido por un *test objetivo-meta*).

Para definir los árboles de búsqueda, es necesario diferenciar entre el concepto de **Estado** y **Nodo**. Un **Estado** nos representa una situación real; una fotografía de la situación del mundo en un determinado momento. El **Nodo** por su parte es una estructura de datos que es parte del árbol de búsqueda y, como parte de su información contiene:

- **Estado**. Representación interna del estado. Suele ser una referencia (puntero) a un conjunto de estados posibles. Varios nodos pueden compartir el mismo estado. Por ejemplo, en un problema de búsqueda de un camino entre dos ciudades de España, es posible que varios nodos pasen por la misma ciudad (*estado*).
- **Padre**. Referencia al nodo padre que, mediante una determinada acción, genera el nodo actual.
- **Hijos**. Referencias a los nodos hijo que se generan aplicando una determinada acción. A partir de un nodo, aplicando una acción (que tendrá asociado un coste), llegamos a un nodo hijo.
- **Profundidad**. Indica la profundidad en el árbol en la que se encuentra el nodo.
- **Costo del camino**. Indica el coste acumulado desde el nodo inicial (raíz del árbol de búsqueda) hasta el nodo actual.

De este modo, el pseudo-código de construcción de un árbol de búsqueda, se muestra en el listado 1. La única diferencia entre los diferentes métodos de búsqueda se basa en la estrategia de expansión del árbol. En realidad, el árbol de búsqueda es el mismo, pero se construirá en un determinado orden (podando ciertas ramas) según la estrategia utilizada.

Dependiendo de cada problema concreto, se creará el nodo raíz del árbol con el *Estado Inicial* del problema. Si el problema es la búsqueda del camino óptimo para viajar desde Madrid hasta Córdoba pasando por las capitales de provincia, la definición parcial del árbol de búsqueda puede verse en la Figura 3.34. En esa figura, el Nodo raíz contiene el *Estado Inicial* (Madrid). A partir de ahí se expanden los nodos hijo de

Algorithm 1 Pseudocódigo del Árbol de Búsqueda

```
Función ÁrbolBusqueda(Problema,Estrategia):return Solución o Fallo
Inicia el árbol con Estado Inicial del Problema
while (Hay Candidatos para Expandir) do
    Elegir una hoja para expandir según Estrategia
    if Nodo.Estado cumple Test-Objetivo then return Solución
    elseExpande Nodo y añade hijos al árbol de búsqueda
    end if
end while
return Fallo // No quedan Candidatos a Expandir
```

primer nivel de profundidad. El camino hasta ese nodo tendrá asociado un coste (por ejemplo, el número de kilómetros entre ciudades). Vemos que el estado en diferentes nodos puede estar repetido (como en el caso de *Ciudad Real* o *Cuenca*).

Como hemos comentado anteriormente, la *Estrategia* define el orden de expansión de los nodos del árbol. Diferentes estrategias ofrecen diferentes niveles aproximaciones a la construcción del árbol de búsqueda y puede dar lugar a soluciones que no sean óptimas (como vemos, el algortimo general estudiado en el listado 1, devuelve como solución el camino hasta el primer nodo cuyo estado cumple el *Test-Objetivo*). A continuación veremos algunas de los estrategias clásicas empleadas en la creación del árbol.

Una **solución** en problemas de búsqueda puede definirse como una **secuencia de acciones** que dirigen al sistema desde un **estado inicial** hasta un estado que cumpla un determinado **test objetivo**.

Una **Estrategia** queda definida por el orden de expansión de los nodos hoja del árbol de búsqueda (denominada *Frontera*, que está formada por los nodos que aún no han sido expandidos). Cada Estrategia se puede evaluar en base a cuatro parámetros:

1. **Completitud**. Si existe solución, ¿la *estrategia* la encuentra?
2. **Optimalidad**. De entre las soluciones posibles, ¿la *estrategia* garantiza que encuentra la mejor solución (o solución óptima en base a la función de coste definida)?
3. **Complejidad Temporal**. ¿Cuál es el orden de complejidad temporal (número de nodos a analizar para construir la solución)?.

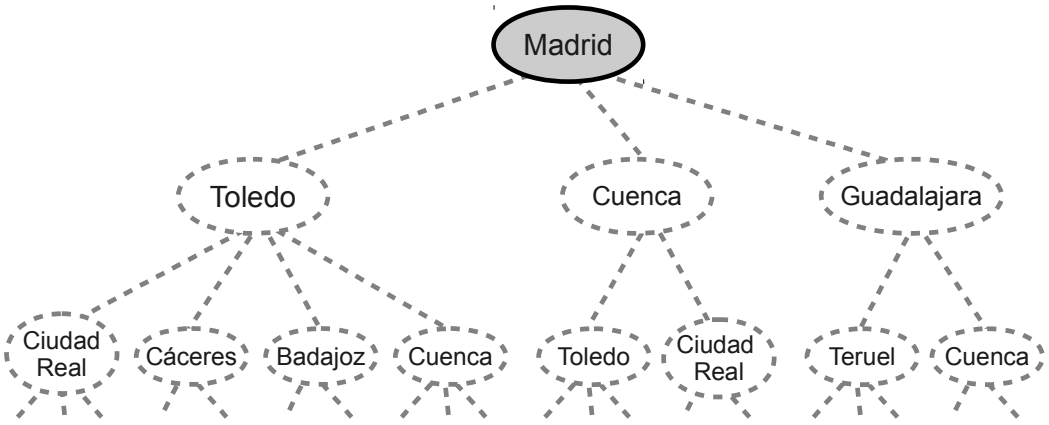

Figura 3.34: Definición parcial del árbol de búsqueda en el problema de viajar de Madrid a Córdoba pasando por capitales de provincia.

4. **Complejidad Espacial**. ¿Cuántos nodos tienen que mantenerse en memoria para conseguir construir la solución?.

Las estrategias de expansión pueden ser categorizadas en base a la información empleada en la construcción del árbol de búsqueda. En una primera taxonomía, podemos distinguir dos grandes familias de técnicas:

- **Estrategias Uniformes**. También denominadas técnicas de *Búsqueda a Ciegas*, donde no se dispone de información adicional; únicamente contamos con la información expresada en la propia definición del problema. De entre las estrategias de búsqueda a ciegas más empleadas se encuentran la búsqueda *en Anchura* y la búsqueda *en Profundidad*.
- **Búsqueda Informada**. Utilizan información adicional para dar una valoración sobre la *apariencia* de cada nodo, aportando información extra que puede utilizarse para *guiar* la construcción del árbol de búsqueda. Algunas de las estrategias más empleadas de búsqueda informada se encuentran los algoritmos *Voraces* y el *A**.

La **Búsqueda en Anchura** (ver Figura 3.35) expande el nodo más antiguo de la *Frontera*. Así, la *Frontera* se maneja como una cola FIFO. Esta estrategia es completa, no da una solución óptima y tiene una gran complejidad espacial (mantiene todos los nodos en memoria).

La **Búsqueda en Profundidad** (ver Figura 3.36) expande el nodo más moderno de la *Frontera*, manteniendo una estrutura tipo Pila. Aunque

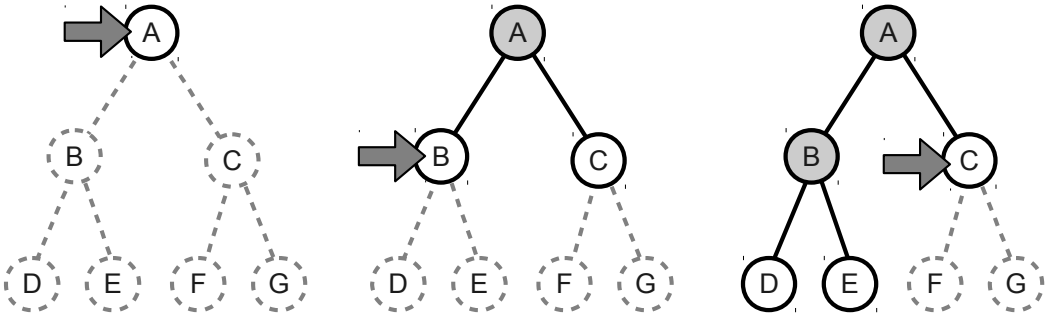

Figura 3.35: Ejemplo de Estrategia de Expansión del árbol en Anchura.

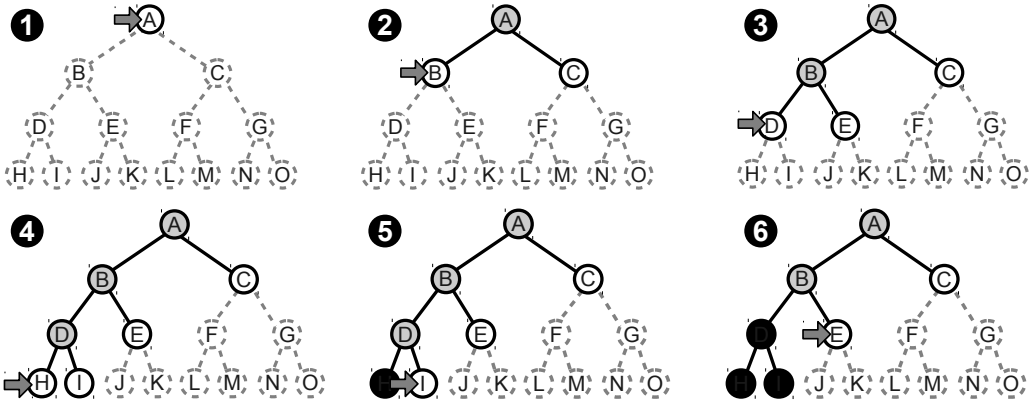

Figura 3.36: Algunas etapas de expansión en la Búsqueda en Profundidad. Los nodos marcados en color negro (por ejemplo, en el paso 5 y 6) son liberados de la memoria, por lo que la complejidad espacial de esta estrategia es muy baja.

tiene una baja complejidad espacial, cuenta con una gran complejidad temporal, es igualmente una estrategia completa aunque no óptima.

Las **Técnicas de Búsqueda Informada** se basan en la idea de utilizar una función de evaluación de cada nodo $f(n)$. Esta función nos resume la *apariencia* de cada nodo de ser mejor que el resto, de modo que expandimos los nodos de la frontera que parecen ser más prometedores de llevar a una solución.

Los algoritmos **Voraces** ordenan los nodos de la frontera por orden decreciente de la valoración $f(n)$. En el ejemplo de realizar una búsqueda del mejor camino entre Madrid y Córdoba pasando por las capitales de provincia española, podríamos utilizar como función f la distancia en línea recta desde cada capital al destino. Así, en cada nodo tendríamos una estimación de *cómo de bueno* parece ese nodo. La Figura 3.37

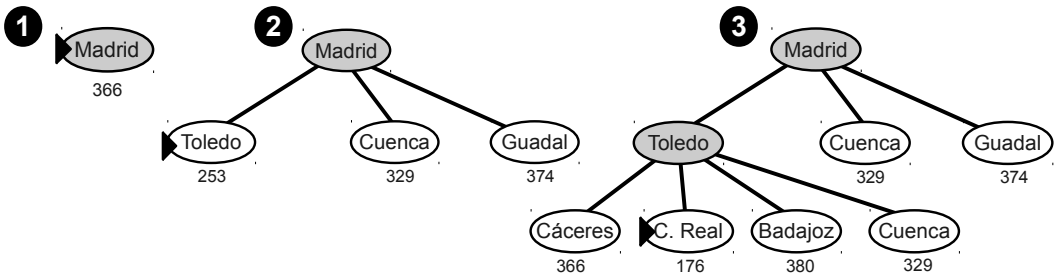

Figura 3.37: Ejemplo de Expansión mediante estrategia Voraz (las distancias entre capitales de provincia son ficticias).

muestra un ejemplo de expansión con valores de distancia ficticios. Vemos que siempre se elige como nodo a expandir aquel de la frontera que tenga menor valor de la función f, expandiendo siempre el nodo que parece estar más cerca del objetivo. Este tipo de búsqueda no es completa, puede quedarse *atascada* en bucles, y puede no dar la solución óptima. Sin embargo, eligiendo una buena función f puede ser una aproximación rápida a ciertos problemas de búsqueda.

Un método de búsqueda ampliamente utilizado en el mundo de los videojuegos es el **algoritmo A*** (que se suele leer como *A Asterisco* o *A Estrella*). La idea base de este método es tratar de evitar expandir nodos que son caros hasta el momento actual. Define una función de evaluación para cada nodo que suma dos términos $f(n)+g(n)$. La función $f(n)$ que indica el coste estimado total del camino que llega al objetivo pasando por n. Por su parte, $g(n)$ mide el coste del camino para llegar a n. Si en la definición de la función $f(n)$ aseguramos que siempre damos un valor menor o igual que el coste real (es decir, la función es *optimista*), se puede demostrar formalmente que A* siempre dará la solución óptima.

En las Figuras 3.38 y 3.39 se describe un ejemplo de uso de A*. En la Figura 3.38 definimos un mapa con ciertas poblaciones por donde batalló Don Quijote. El grafo muestra el coste (ficticio) en Kilómetros entre las poblaciones. Vamos a suponer que queremos viajar desde Ciudad Real al bello paraje de *Las Pedroñeras*. Para ello, definimos como función $f(n)$ el coste en línea recta desde cada estado al destino (recogido en la tabla de la derecha).

La Figura 3.39 resume la aplicación del algoritmo A* en diferentes etapas. Con la definición de mapa anterior, podemos comprobar que el resultado objenido en el paso 6 es óptimo. En cada nodo representaremos el coste total como la suma del coste acumulado $g(n)$ más el coste estimado (en línea recta) hasta el destino $f(n)$.

Así, la evaluación del nodo inicial (paso 1), el coste total es $366 = g(n)0 + f(n)366$. Elegimos ese nodo (el único en la frontera) y expandimos

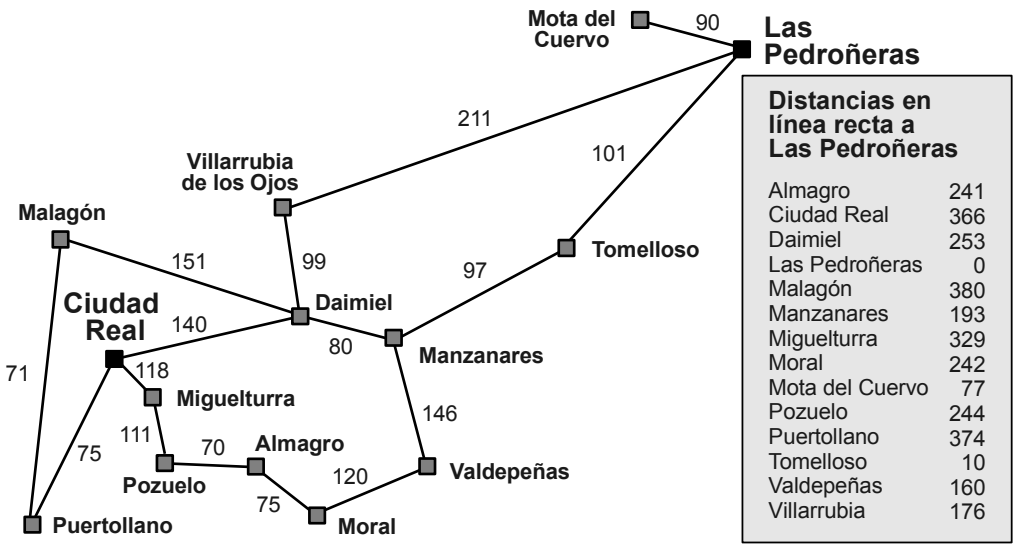

Figura 3.38: Descripción de costes de viajar por la tierra de Don Quijote (distancias ficticias). Ejemplo basado en caso de estudio de [12].

todos sus hijos. Como *Ciudad Real* está conectado con tres poblaciones, tendremos 3 hijos (paso 2). Calculamos el valor de $g(n) + f(n)$ para cada uno de ellos, y los añadimos de forma ordenada (según ese valor) en la frontera. El nodo que tiene menor valor es Daimiel (140 de coste de ir desde Ciudad Real más 253 de coste estimado - en línea recta - hasta llegar a Las Pedroñeras). Como Daimiel está conectado con cuatro poblaciones, al expandir (en 3), añadiremos cuatro nuevos nodos a la frontera de forma ordenada. Ahora la frontera tendría en total 6 nodos (con estados representados en color blanco: C. Real, Villarubia, Malagón, Manzanares, Miguelturra y Puertollano). Al ordenarlos según el valor de la suma de ambas funciones, elegimos *Manzanares* como candidato a expandir. Siguiendo el mismo procedimiento, llegamos en el paso 6 a la solución óptima de *Ciudad Real*, *Daimiel*, *Manzanares*, *Tomelloso* y *Las Pedroñeras*. Basta con recorrer el árbol desde la solución al nodo raíz para componer la ruta óptima. Cabe destacar el mínimo número de nodos *extra* que hemos tenido que expandir hasta llegar a la solución óptima.

En el contexto del desarrollo de videojuegos, las estrategias de búsqueda son realmente relevantes para implementar el módulo de IA de un juego. Sin embargo, resulta esencial considerar la naturaleza del juego cuya IA se desea implementar. Por ejemplo, algunos **juegos clásicos** como las damas, el tres en raya o el ajedrez son casos particulares en los que los algoritmos de búsqueda se pueden aplicar para evaluar cuál es el mejor movimiento en cada momento.

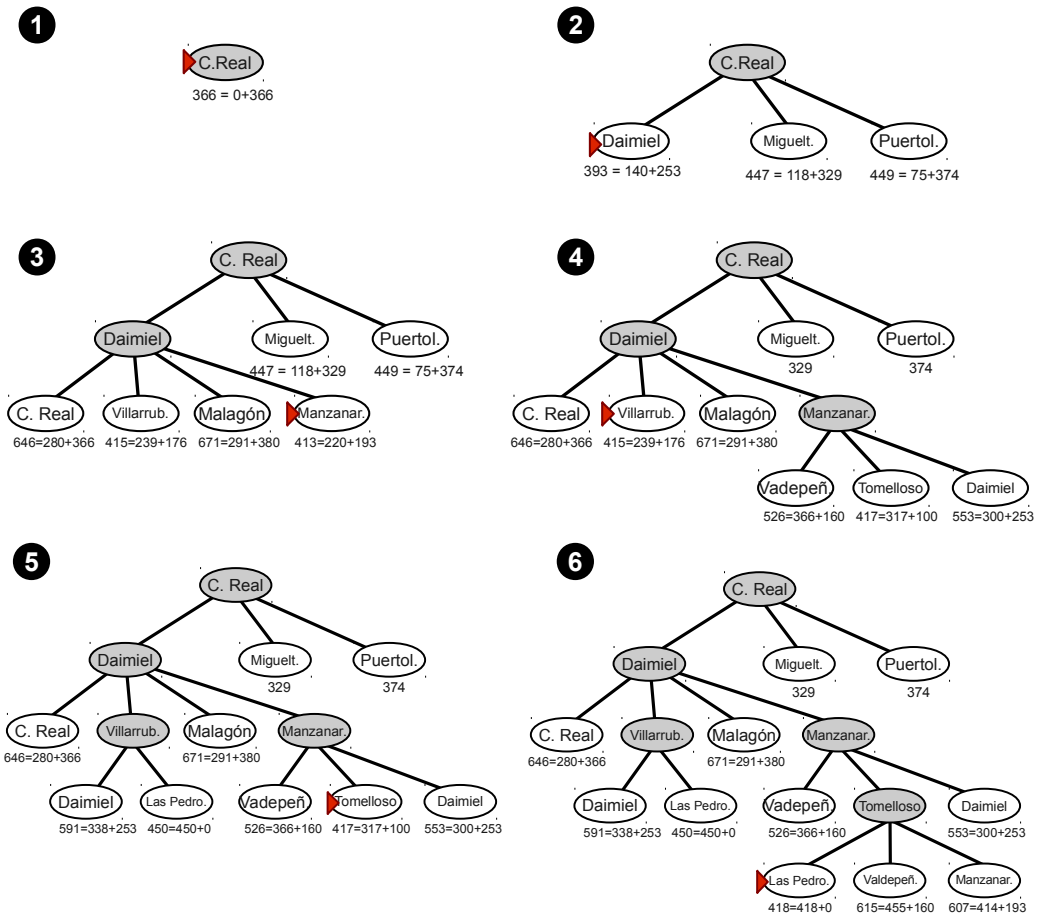

Figura 3.39: Ejemplo de Expansión mediante A* en el viaje de Ciudad Real a Las Pedroñeras.

En los juegos es necesario considerar todas las posibles acciones de réplica del contrincante. Aunque la aplicación de un algoritmo de búsqueda exhaustivo es, generalmente, muy costoso, la evolución del hardware ha posibilitado implementar soluciones de IA que derroten fácilmente a casi cualquier oponente humano. Un caso representativo es *Deep Blue*, una supercomputadora desarrollada por IBM que derrotó en una primera partida al campeón mundial de ajedrez, Gary Kaspárov, en 1997. La implementación del módulo de IA de *Deep Blue* estaba basada en la fuerza bruta, evaluando unos 200 millones de posiciones por segundo[24].

[24] En realidad, la implementación era algo más sofisticada, contando con tablas específicas de jugadas para la finalización y movimientos de inicio de partida clásicos.

En la actualidad, podemos asegurar que prácticamente cualquier juego *clásico* puede ser jugado de una forma muy satisfactoria por una máquina. En las Damas, el sistema *Chinook* ganó en 1994 al campeón mundial *Marion Tinsley* empleando una Base de Datos de jugadas perfectas cuando hay 8 piezas o menos en el tablero (unas 400.000 millones de posiciones). En el Othello (también conocido como Reversi), hace muchos años que los campeones humanos no quieren jugar contra los ordenadores porque son demasiado buenos. Por su parte, en el juego de estrategia para dos jugadores chino Go, los campeones humanos no quieren jugar contra ordenadores porque son demasiado malos[25].

Este gran número de posiciones es consecuencia de la generación del árbol de búsqueda a través del **algoritmo MiniMax**, el cual se discutirá a continuación.

La **estrategia MiniMax** es perfecta para juegos deterministas (donde no interviene el azar). La idea básica es elegir como transiciones entre nodos las acciones que maximizan el valor para el jugador *MAX* y que minimizan para el jugador *MIN*.

La consideración general se basa en tener un juego con los dos jugadores descritos anteriormente. *MAX* mueve primero y, después, mueven por turnos hasta que el juego finaliza. Cuando el juego termina, el ganador recibe una bonificación y el perdedor una penalización. Así, un juego se puede definir como una serie de problemas de búsqueda con los siguientes elementos [12]:

- El **estado inicial**, que define la posición inicial del tablero de juego y el jugador al que le toca mover.
- Una **función sucesor**, que permite obtener una serie de pares <movimiento, estado>, reflejando un movimiento legal y el estado resultante.
- Un **test terminal**, que permite conocer cuándo ha terminado el juego. Los estados en los que el test terminal devuelve un valor lógico verdadero se denominan estados terminales y representan los nodos hoja del árbol de búsqueda.
- Una **función de utilidad** o función objetivo que asigna un valor numérico a los estados terminales (nodos hoja del árbol). Por ejemplo, en el caso de las tres en raya, esta función devuelve un valor +1 a una situación ganadora, un valor de -1 a una situación perdedora y un valor de 0 a una situación de empate (ver Figura 3.40).

[25] No está todo inventado, afortunadamente quedan muchos retos abiertos para seguir trabajando en técnicas de Inteligencia Artificial.

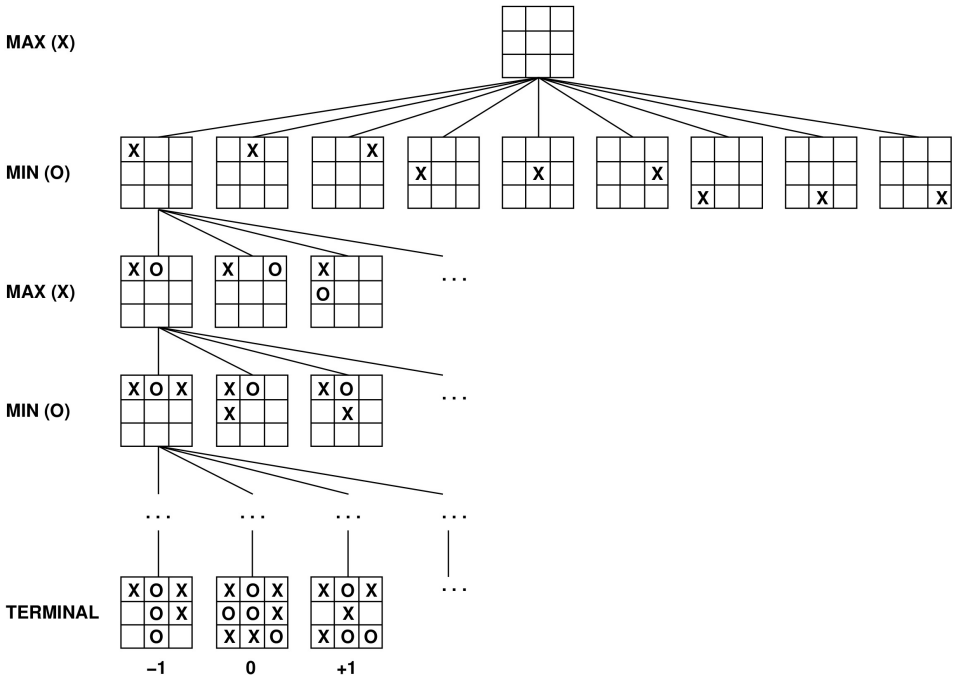

Figura 3.40: Árbol parcial de búsqueda para el juego tres en raya (tic-tac-toe). El nodo superior representa el estado inicial. MAX mueve en primer lugar, colocando una X en una de las casillas vacías. El juego continúa con movimientos alternativos de MIN y MAX hasta alcanzar nodos terminales.

En este contexto, la generación de descendientes mediante la función sucesor a partir del estado inicial permite obtener el **árbol de búsqueda**. La Figura 3.40 muestra una parte del árbol de búsqueda para el caso particular del tres en raya[26]. Note cómo el jugador MAX tiene 9 opciones posibles a partir del estado inicial. Después, el juego alterna entre la colocación de una ficha por parte de MIN (O) y MAX (X), hasta llegar a un estado terminal. En este estado, se evalúa si alguno de los dos jugadores ha ganado o si el tablero está completo. El número asociado a cada estado terminal representa la salida de la función de utilidad para cada uno de ellos desde el punto de vista de MAX. Así, los valores altos son buenos para MAX y negativos para MIN. Este planteamiento, basado en estudiar el árbol de búsqueda, permite obtener el mejor movimiento para MAX. El listado 2 muestra el pseudocódigo de la implementación general de Minimax.

[26]Puede consultar las reglas en http://es.wikipedia.org/wiki/Tres_en_línea

Algorithm 2 Pseudocódigo del algoritmo Minimax

```
función Decisión-Minimax (estado) devuelve una acción
variables de entrada: estado // estado actual del juego
v ← Max-Valor(estado)
devolver la acción de Sucesores(estado) con valor v
```

```
función Max-Valor (estado) devuelve un valor utilidad
if Test-Terminal (estado) then devolver Utilidad(estado)
v ← −∞
while (s en Sucesores (estado)) do
    v ← Máximo (v, Min-Valor(s))
end while
devolver v
```

```
función Min-Valor (estado) devuelve un valor utilidad
if Test-Terminal (estado) then devolver Utilidad(estado)
v ← ∞
while (s en Sucesores (estado)) do
    v ← Mínimo (v, Max-Valor(s))
end while
devolver v
```

La función principal *Decisión-Minimax* se encarga de devolver la acción a aplicar ante un determinado *estado* de entrada. La construcción del árbol de juego se realiza empleando dos funciones auxiliares que se llamarán alternativamente: *Max-Valor* que trata de maximizar el valor para el jugador MAX, y *Min-valor* que trata de obtener la jugada que más beneficia al jugador MIN. Comenzamos construyendo el árbol para MAX, pasando ese estado como inicial.

En cada llamada, la función *Max-Valor* primero comprueba si el estado es terminal (fin de partida). En ese caso, ejecuta la función de utilidad sobre el estado y devuelve una valoración (en el caso de las tres en raya, si gana el jugador MAX, devolverá +1). Si no es terminal, estudiamos para todos los sucesores de ese estado, cuál es la valoración más beneficiosa para el jugador MAX. Es decir, estudiamos de entre todos los hijos (expandiendo con *Min-valor*), y nos quedamos con el que nos aporta mayor beneficio.

La implementación de *Min-Valor* es similar, pero eligiendo como mejor valor el que devuelva el mínimo de entre los hijos. En el ejemplo de la Figura 3.41, vemos que el nodo raíz se quedaría con la mayor valoración de sus hijos (que son nodos MIN). A su vez, cada nodo MIN se ha quedado con el mínimo de sus hijos (que son nodos MAX). Los nodos hoja (tipo MAX en este caso) obtienen su valoración mediante la ejecución de la función de utilidad.

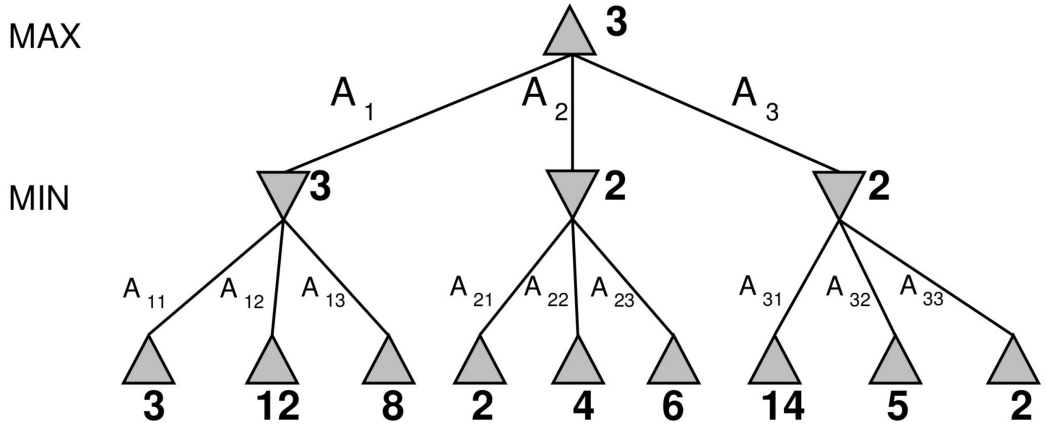

Figura 3.41: Ejemplo de árbol Minimax. La raíz es un nodo Max. Los nodos hoja incluyen el valor de la función utilidad.

Este algoritmo es completo (siempre que el árbol de sea finito), y ofrece una solución óptima. El principal problema es de complejidad [27]. Ante este problema relativo al crecimiento en el número de nodos exponencial, existen aproximaciones que *podan* el árbol de juego, tomando decisiones sin explorar todos los nodos. La poda $\alpha - \beta$ es una técnica de implementación clásica empleada en el algoritmo Minimax, que no afecta a la calidad del resultado final (sigue siendo óptimo). Puede verse como una forma simple de *meta-razonamiento*, donde se estudian qué cálculos son relevantes. La eficacia de la poda depende de la elección ordenada de los sucesores.

3.5.7. Caso de estudio. Un Tetris *inteligente*

En esta sección se discute cómo afrontar el módulo de IA del Tetris en el modo ***Human VS CPU***, es decir, cómo se puede diseñar el comportamiento de la máquina cuando se enfrenta a un jugador real.

Tradicionalmente, el modo *versus* del Tetris se juega a pantalla dividida, al igual que el modo de dos jugadores. En la parte izquierda juega el jugador real y en la derecha lo hace la máquina. En esencia, el perdedor es aquél que es incapaz de colocar una ficha debido a que ha ocupado la práctica totalidad del tablero sin *limpiar* líneas.

Para llevar a cabo la gestión de la dificultad de este modo de juego o, desde otro punto de vista, modelar la habilidad de la máquina, se pueden plantear diversas alternativas. Por ejemplo, si se desea modelar

[27] Por ejemplo, para el caso del ajedrez hablamos de una complejidad temporal de $O(35^{1}00)$

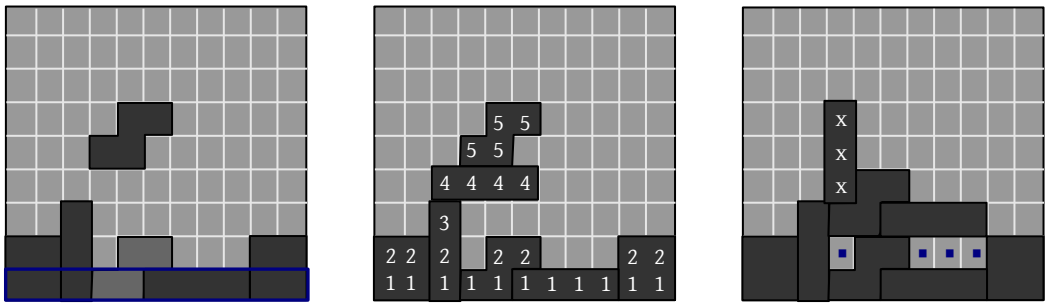

Figura 3.42: Planteando un módulo de IA para el Tetris. **a)** *Limpieza* de una línea, **b)** Cuantificando la altura del montón de fichas, **c)** Problema de huecos perdidos (los puntos representan los huecos mientras que las 'x' representan los potenciales bloqueos).

un nivel de complejidad elevado, entonces bastaría con incrementar la **velocidad** de caída de las fichas. En este caso, la máquina no se vería afectada ya que tendría tiempo más que suficiente para colocar la siguiente ficha. Sin embargo, el jugador humano sí que se vería afectado significativamente.

Otra posibilidad para ajustar el nivel de dificultad consistiría en que el jugador y la máquina recibieran distintos **tipos de fichas**, computando cuáles pueden ser más adecuadas para completar una línea y, así, reducir la altura del montón de fichas. También sería posible establecer un ***handicap***, basado en introducir deliberadamente piezas en la pantalla del jugador real.

No obstante, la implementación de todas estas alternativas es trivial. El **verdadero reto** está en modelar la IA de la máquina, es decir, en diseñar e implementar el comportamiento de la máquina a la hora de ir colocando fichas.

La solución inicial planteada en esta sección consiste en **asignar una puntuación** a cada una de las posible colocaciones de una ficha. Note que las fichas se pueden rotar y, al mismo tiempo, se pueden colocar en distintas posiciones del tablero. El objetivo perseguido por el módulo de IA será el de colocar una ficha allí donde obtenga una mejor puntuación. El siguiente paso es, por lo tanto, pensar en cómo calcular dicha puntuación.

Para ello, una opción bastante directa consiste en distinguir qué aspectos resultan fundamentales para ganar o perder una partida. En principio, el módulo de IA debería evitar formar torres de fichas de gran altura, ya que lo aproximarían a perder la partida de forma inminente,

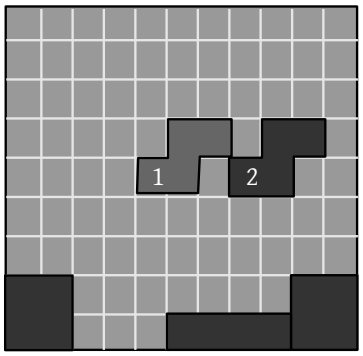

Figura 3.43: Idealmente, la posición 2 debería ser premiada en detrimento de la posición 1, ya que facilita la colocación de futuras piezas. Por lo tanto, esta primera opción debería tener una puntuación superior a la última cuando se realice el cálculo de la *utilidad* de la pieza.

tal y como muestra la figura 3.42.b. Este factor debería suponer una penalización para la puntuación asociada a colocar una ficha. Por el contrario, la máquina debería *limpiar* líneas siempre que fuera posible, con el objetivo de obtener puntos y evitar que el montón de fichas siga creciendo. Este factor representaría una bonificación.

Además, idealmente el módulo de IA debería evitar generar espacios que no se puedan aprovechar por las fichas siguientes, es decir, debería evitar que se perdieran huecos, tal y como se refleja en la figura 3.42.c. Evidentemente, la generación de huecos perdidos es, a veces, inevitable. Si se produce esta situación, el módulo de IA debería evitar la colocación de fichas sobre dichos huecos, con el objetivo de liberarlos cuanto antes y, en consecuencia, seguir *limpiando* líneas.

Estos cuatro factores se pueden ponderar para construir una **primera aproximación** de la fórmula que se podría utilizar para obtener la puntación asociada a la colocación de una ficha:

$$P = w_1 * salt + w_2 * nclears + w_3 * nh + w_4 * nb \tag{3.1}$$

donde

- $salt$ representa la suma de las alturas asociada a la pieza a colocar en una determinada posición,
- $nclears$ representa el número de líneas *limpiadas*,
- nh representa el número de huecos generados por la colocación de la ficha en una determinada posición,
- nb representa el número de bloqueos como consecuencia de la potencial colocación de la ficha.
- w_i representa el peso asociado al factor i.

Evidentemente, $nclears$ tendrá asociado un peso positivo, mientras que el resto de factores tendrán asociados pesos negativos, ya que representan penalizaciones a la hora de colocar una ficha.

Un inconveniente inmediato de esta primera aproximación es que no se contempla, de manera explícita, la construcción de **bloques consistentes** por parte del módulo de IA de la máquina. Es decir, sería necesario incluir la lógica necesaria para premiar el acoplamiento entre fichas de manera que se premiara de alguna forma la *colocación lógica* de las mismas. La figura 3.43 muestra un ejemplo gráfico de esta problemática, cuya solución resulta fundamental para dotar a la máquina de un comportamiento *inteligente*.