Universidad de Valladolid

FACULTAD DE CIENCIAS

TRABAJO FIN DE GRADO

Grado en Matemáticas

INTELIGENCIA ARTIFICIAL Y TEORÍA DE JUEGOS

Autor: Ismael Alba Gallego
Tutor: Cesáreo Jesús González Fernández
Año: 2023

Índice general

Introducción **3**

1 Teoría de juegos **5**
1.1 Conceptos básicos 5
1.2 Juegos estáticos con información completa 9
1.3 Juegos estáticos con información incompleta 20
1.4 Juegos dinámicos 29
1.5 Aplicaciones de la Teoría de Juegos 39

2 Inteligencia artificial **43**
2.1 Algoritmos de búsqueda 43
2.1.1 Búsqueda en amplitud 47
2.1.2 Búsqueda en profundidad 48
2.1.3 Búsqueda en profundidad iterativa 50
2.1.4 Heurística 52
2.2 Algoritmos de aprendizaje automático 54
2.2.1 Algoritmos genéticos 55
2.2.2 Aprendizaje por refuerzo 68
2.3 Aplicaciones de la Inteligencia Artificial a la Teoría de Juegos 75
2.3.1 Minimax 77

3 Algoritmos genéticos aplicados a la Teoría de Juegos **81**
3.1 Autómatas finitos 81
3.2 Dilema del prisionero repetido con coevolución de autómatas 83
3.2.1 Metodología 85
3.2.2 Evaluación de los resultados 87
3.3 Programa sobre el oligopolio de Bertrand 96
3.3.1 Descripción del juego y algoritmo 96
3.3.2 Metodología 97
3.3.3 Evaluación de los resultados 98

4 Conclusiones **101**
4.1 Resumen del trabajo 101
4.2 Conclusiones derivadas de los resultados 101
4.3 Recomendaciones para mejoras en los métodos 102

A Algoritmos **105**

Índice de figuras **108**

Bibliografía **113**

Introducción

La Teoría de juegos es el campo matemático que analiza las situaciones en las que varios agentes racionales actúan de forma lógica para obtener el mayor beneficio posible, teniendo en cuenta los incentivos que dependen de sus elecciones. La Inteligencia Artificial es el conjunto de métodos y algoritmos que dotan a las máquinas de la capacidad de aprender y resolver problemas a partir de los datos, y de mejorar su desempeño en un dominio específico conforme disponen de más información. Estas dos áreas se relacionan de diversas formas, como por ejemplo, usando la Teoría de Juegos para crear inteligencias artificiales basadas en redes neuronales, donde cada neurona se puede ver como un jugador[1]. Otra forma de relacionarlas es usando la Inteligencia Artificial para aplicar y modelar la Teoría de Juegos en diferentes problemas complejos. Este último aspecto será el foco de nuestro estudio y el ámbito en el que nos enfocaremos.

El juego que más estudiaremos será un juego clásico de la Teoría de Juegos: el dilema del prisionero repetido. Este juego consiste en que dos jugadores se enfrentan repetidamente y deben elegir entre cooperar o traicionar a otro, obteniendo una recompensa o castigo según su decisión y la del contrincante. El objetivo es estudiar cómo evolucionan las estrategias de los jugadores a lo largo del tiempo.

Para ello, utilizaremos un modelo de aprendizaje y adaptación basado en autómatas finitos y algoritmos genéticos. Los autómatas finitos son máquinas abstractas que presentan el comportamiento de los jugadores en función de los movimientos reportados por el otro jugador. Los algoritmos genéticos son una técnica de optimización inspirada en la evolución biológica que permite explorar y mejorar el espacio de posibles autómatas.

El trabajo se estructura de la siguiente manera: En el Capítulo 1 se presenta una introducción teórica a la Teoría de Juegos y a la Inteligencia Artificial. Se comentarán las bases de la Teoría de juegos, distintos tipos de juegos, ejemplos y algunas aplicaciones de esta teoría, tanto en la vida real como en la Inteligencia Artificial. Se describirán conceptos básicos de Inteligencia Artificial, distintos algoritmos y aplicaciones.

Por otra parte, en el capítulo 2 se muestra y aplica el modelo de autómatas finitos y algoritmo genético utilizado para los dos juegos presentados: dilema del prisionero repetido y oligopolio de Bertrand repetido. Se comentará la implementación y resultados obtenidos.

Para finalizar, en el capítulo 3 se resumen las conclusiones obtenidas, se exploran algunas extensiones y aplicaciones del modelo a otros juegos y se añaden distintas recomendaciones de mejora para futuras líneas de ampliación de este trabajo.

[1]Consultar *Deep Learning meets Game Theory.* [9]

Información importante.
Los códigos utilizados en este trabajo pueden encontrarse en el Anexo A para su descarga. En caso de no tener acceso a este anexo o problemas para obtener los códigos, póngase en contacto en la siguiente dirección: ismaelalbagallego@gmail.com

Capítulo 1

Teoría de juegos

La Teoría de Juegos es una disciplina que estudia las situaciones de interacción estratégica entre agentes racionales que tienen intereses comunes o contrapuestos. Estas situaciones se denominan juegos y se caracterizan por la existencia de reglas, acciones, resultados y pagos para cada agente. La Teoría de Juegos busca analizar y predecir el comportamiento de los agentes y los resultados.

La Teoría de Juegos tiene múltiples aplicaciones en diversos campos del conocimiento, como la economía, la política, la biología, la sociología o la informática. Algunos ejemplos de juegos son el dilema del prisionero, el oligopolio de Cournot, las subastas o los juegos evolutivos. La Teoría de Juegos ha recibido un gran reconocimiento académico, al otorgarse el Premio Nobel de Economía a varios de sus creadores y practicantes.

A continuación se presentarán los conceptos básicos de la Teoría de Juegos y sus principales tipos y formas de representación. Se hará especial énfasis en los juegos estáticos y dinámicos, con información completa e incompleta, así como en los conceptos de equilibrio de Nash. Se ilustrarán los conceptos con ejemplos y aplicaciones relevantes. Nos basaremos en libro *An introduction to game theory* [6], que ofrece una introducción rigurosa y accesible a los conceptos que trataremos.

1.1 Conceptos básicos

Para analizar y comprender los juegos y sus soluciones, es necesario definir y manejar una serie de conceptos básicos que nos permitan describir las características y elementos de cada juego[1]. Comenzaremos presentando estos conceptos:

Jugador. Es uno de los participantes del juego. Es un tomador de decisiones estratégicas dentro del juego, donde el resultado depende de las decisiones de todos ellos. Para que un juego sea considerado como tal, debe haber al menos dos jugadores.

Acciones. Estas son las diferentes opciones que el jugador elige en el momento que le toque jugar, es decir, son las decisiones que tiene disponibles. El conjunto de acciones puede ser finito o infinito.

[1]Consultar *An introduction to game theory.* [6]

Resultados. Estos son los posibles desenlaces del juego, que dependen de las decisiones tomadas por los jugadores y conllevan unas consecuencias para cada jugador. Estos resultados tienen distintas formas de representación.

Pagos. En cada etapa del juego cada juagor recibe un pago. Éste lógicamente depende de las decisiones tomadas por los jugadores. Los pagos reflejan las preferencias de los jugadores sobre los posibles desenlaces del juego. Los pagos pueden ser continuos o discretos. Un juego será de suma cero cuando el benefico de un jugador es la pérdida de otro.

Estrategias. Las estrategias son las opciones que el jugador elige en un juego donde el resultado depende no solo de sus propias acciones, sino también de las acciones de sus adversarios. Un perfil de estrategias es un conjunto de estrategias, habrá un perfil por jugador, y podemos diferenciar dos casos, cuando se escogen las acciones con certeza o con cierta probabilidad.

Representación. Son las forma de mostrar un juego con sus diferentes componentes, como los jugadores, estrategias y resultados. Existen dos formas principales de representar un juego: normal o extensiva. La forma normal representa el juego en una matriz de pagos, donde cada fila corresponde a una estrategia del jugador 1 y cada columna corresponde a una estrategia del jugador 2, cada celda contiene los pagos de ambos jugadores para esa combinación de estrategia. La matriz puede ampliarse a más dimensiones para juegos con más jugadores. Esta forma es útil para analizar juegos simultáneos, donde los jugadores escogen sus estrategias al mismo tiempo. La forma extensiva representa el juego en forma de grafo, donde cada nodo corresponde a una decisión de un jugador, y cada arista una acción posible, cada nodo final posee las utilidades de ambos jugadores. Esta forma es útil para analizar juegos secuenciales, donde los jugadores eligen sus estrategias en un orden determinado.

Pongamos algún ejemplo.

Ejemplo 1. Pares y nones.

Javier y Diana son dos amigos que juegan a pares y nones. Cada uno puede elegir entre mostrar un número par o un número impar con los dedos de su mano. Ambos muestran el resultado a la vez. Si la suma del resultado es un número par, Diana pagará a Javier 5 euros, si el resultado en impar, será Javier el que pague a Diana 5 euros.

Este juego podemos resumirlo en la siguiente tabla:

		Diana	
		Par	Impar
Javier	Par	(5,-5)	(-5,5)
	Impar	(-5,5)	(5,-5)

Cuadro 1.1: Juego en forma normal.

Si Javier escoge jugar un número par y Diana escoge jugar un número par, estaremos

en la primer celda de la tabla. La suma será un número par, y el pago será $(5, -5)$, es decir, Javier tendrá un pago de 5 y Diana un pago de -5.

Este juego es representado de forma normal.

Una forma extensiva de representar este mismo juego sería la siguiente figura:

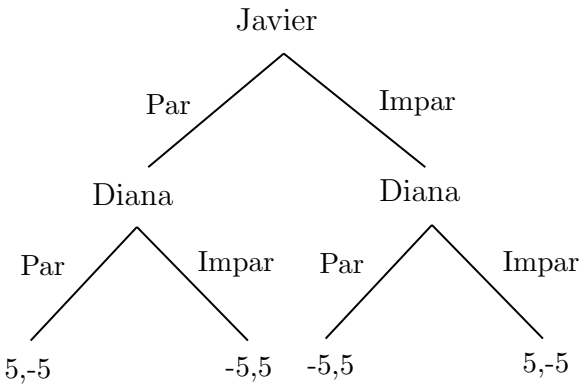


Figura 1.1: Juego en forma extensiva.

La figura se debe leer desde arriba hacia abajo, pero hay que tener en cuenta que ambos jugadores toman las decisiones al mismo tiempo. Si Javier escoge impar, seguimos la línea hasta llegar a Diana, si Diana escoge par, nos lleva al pago $(-5, 5)$, es decir, Javier pierde 5 y Diana gana 5.

Esta forma de representación es más adecuada para otro tipo de juegos, en los que la decisión de cada jugador depende del orden del movimiento de jugadores, los juegos dinámicos, de los que hablamos a continuación.

Para poder avanzar en el estudio de la Teoría de Juegos, es imprescindible que introduzcamos algunos conceptos fundamentales que nos permitirán entender mejor esta rama de la matemática aplicada y estadística.

Definición 1. *Una **lotería simple** en X es una distribución de probabilidad en X.*

Ejemplo 2. Lotería simple.

Sea $X = \{A, B, C\}$. La lotería simple $L_1 = (1/2, 1/4, 1/4)$ es la opción que consiste en obtener A con probabilidad $1/2$ y B y C con probabilidad $1/4$ cada uno.

Definición 2. *Una función $U : X \to \mathbb{R}$ es una **función de utilidad** si para todo $x, y \in X$, se tiene que x es preferido a y si y sólo si $U(x) \geq U(y)$.*

Es decir, la función de utilidad es una función que asigna un número real a cada posible resultado de un juego, de tal forma que el agente prefiere el resultado con mayor utilidad. El jugador busca la estrategia que le dé la mayor utilidad esperada, que es el promedio ponderado de las utilidades de cada resultado según su probabilidad.

Definición 3. *Si $X = \{x_1, ..., x_n\}$ es un conjunto de valores numéricos, llamamos* ***valor esperado*** *de la lotería $L = (p_1, ..., p_n)$ al valor numérico $E(L) = x_1p_1 + ... + x_np_n$.*

Ejemplo 3. Función de utilidad.

Supongamos que dos amigos, Jugador 1 y Jugador 2 lanzan una moneda. Si sale cara, Jugador 1 le paga al adversario 10 euros; si sale cruz, Jugador 2 le pagará al adversario 10 euros. Cada uno tiene una función de utilidad que depende de su riqueza final después del juego. Supongamos que la riqueza inicial de Jugador 1 es de 100 euros, mientras que la riqueza del Jugador 2 es de 50 euros. Supongamos entonces que las funciones utilidad son:

$$U_1(x) = ln(x) \text{ para Jugador 1,}$$
$$U_2(x) = 2ln(x) \text{ para Jugador 2, donde } x \text{ es la riqueza final de cada uno}$$

Si calculamos la esperanza de cada jugador, como cada uno tiene dos opciones, será:

$$E[U_1] = \tfrac{1}{2}U_1(100 + 10) + \tfrac{1}{2}U_1(100 - 10) = \tfrac{1}{2}ln(110) + \tfrac{1}{2}ln(90) \approx 4{,}6$$

$$E[U_2] = 2\tfrac{1}{2}U_2(50 + 10) + 2\tfrac{1}{2}U_2(50 - 10) = ln(60) + ln(40) \approx 7{,}78$$

Esto significa que el Jugador 2 tiene una mayor utilidad esperada, es decir, le gusta más ganar 10 euros de lo que le cuesta perderlos. El caso del Jugador 1 es al revés. Al tener un pago mayor de media, el Jugador 2 preferiría jugar, en cambio, el Jugador 1 preferiría no hacerlo.

Los juegos pueden modelizar gran cantidad de problemas, es por ello que tenemos una gran cantidad de tipos de juegos diferentes. Mostraremos varios de estos tipos y algunos ejemplos para mayor claridad.

Juegos cooperativos y no cooperativos. En los juegos cooperativos, los jugadores pueden formar acuerdos para mejorar sus resultados o incluso tener un objetivo común, mientras que en los juegos no cooperativos, cada jugador actúa de forma individual.

Juegos estáticos y dinámicos. En los juegos estáticos, los jugadores toman sus decisiones al mismo tiempo, como puede ser el ejemplo anterior. En cambio, en los juegos dinámicos, hay una secuencia temporal en las decisiones de los jugadores, es decir, hay turnos. Generalmente, la representación extensiva es la más acertada para este tipo de juegos. Veamos un ejemplo de juego dinámico.

Ejemplo 4. Juego dinámico.

El Jugador 1 puede elegir entre dos acciones A y B. El Jugador 2 puede elegir entre dos acciones I y D. Los pagos dependen de las elecciones de los jugadores. El juego se representa en forma extensiva de la siguiente manera:

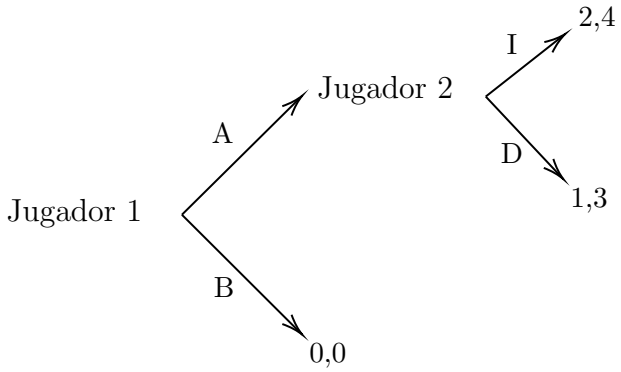


Figura 1.2: Juego dinámico.

El Jugador 1 elige primero entre A y B, y luego el Jugador 2 observa esa elección y elige entre I y D. Si el Jugador 1 elige B, se acaba el juego. Si el Jugador 1 escoge A, entonces el Jugador 2 lo sabe, y puede escoger entre I o D, dando un pago diferente en cada caso.

Juegos con información completa e incompleta. En los juegos de información completa, todos los jugadores conocen todo sobre el juego, en otras palabras, todos los jugadores conocen las reglas del juego, las acciones posibles de cada jugador y los pagos asociados a cada resultado, además, conocen que el resto de los jugadores conocen esta información, y todos los jugadores saben que todos ellos saben que conocen esta información, y así sucesivamente. En cambio, en los juegos de información incompleta, se desconoce alguna de esta información. Más adelante hablaremos de estos dos tipos de juegos.

Juegos de suma cero. Estos juegos son en los que el beneficio de un jugador es la pérdida del otro. El primer ejemplo de pares y nones refleja este tipo de juegos, ya que en el ejemplo, cada vez que un jugador gana 5 euros, el otro pierde 5 euros.

Juegos repetidos. Los juegos repetidos son aquellos que se juegan varias veces entre los mismos jugadores. En estos juegos, cada repetición se anuncia con sus resultados y cada jugador recibe su pago correspondiente. Un juego que es fácil de estudiar, al repetirse muchas veces puede volverse muy complejo, pudiendo por ejemplo utilizar otras ramas de las matemáticas, como diferentes modelizaciones de procesos estocásticos.

1.2 Juegos estáticos con información completa

Los juegos estáticos y de información completa puede decirse que son el tipo más sencillo de juegos. En ellos, los jugadores toman las decisiones simultáneamente, y conocen todo sobre el juego. Nuestro objetivo es encontrar el equilibrio o solución del juego. Para ello introduciremos conceptos como el del equilibrio de Nash, debido al matemático estadounidense John Forbes Nash, galardonado con el premio nobel de economía en 1994.

Estos juegos suelen ser representados de forma normal mediante una matriz de pago. Generalmente, se utiliza una bimatriz para dos jugadores, y una ampliación análoga a esta para más jugadores. Siguiendo la nociones introducidas anteriormente,

- El conjunto $J = 1, 2, ..., n$ será el conjunto de jugadores.

- S_i será el conjunto de estrategias del jugador $i \in J$.
- Cada n-pla $s = (s_1, s_2, ..., s_n)$ donde $s_i \in S_i$ será nuestro perfil de estrategias. Es decir, es un vector de dimensión n donde cada s_i es una estrategia para cada jugador i.
- El conjunto de todos los perfiles de estrategias será $\mathbf{S} = S_1 \times S_2 \times ... \times S_n$
- La función de pagos de cada jugador i será u_i. Esta función asigna un número a cada combinación de estrategias, es decir $u_i(s_1, ..., s_n) \in \mathbb{R}$. Este número es el pago recibido por el jugador i cuando se realizan las jugadas $(s_1, ..., s_n)$.

Con todos estos elementos, podemos denotar un juego G como:

$$G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$$

A continuación, veremos un ejemplo para interiorizar estas notaciones.

Ejemplo 5. Dilema del prisionero.

Dos jóvenes sospechosos de un crimen son arrestados y llevados a una comisaría. El fiscal les ofrece un trato a cada uno por separado: si confiesa y delata a su cómplice, quedará libre y el otro recibirá una condena de 5 años si su compañero no le delata; si no confiesa y el otro lo hace, recibirá una condena de 5 años y el otro quedará libre; si ambos confiesan, recibirán una condena de 4 años cada uno; si ninguno confiesa, recibirán una condena de 1 año cada uno por un delito menor. Los sospechosos no pueden comunicarse entre ellos y deben tomar su decisión sin saber lo que hará el otro. ¿Qué deben hacer los sospechosos?

La representación en forma matricial de este problema es la siguiente:

		Joven 2	
		Callar	Confesar
Joven 1	Callar	(-1,-1)	(-5,0)
	Confesar	(0,-5)	(-4,-4)

Cuadro 1.2: Dilema del prisionero en forma normal.

Tenemos entonces que,

- Los jugadores son $J = \{1, 2\}$.
- $S_1 = S_2 = \{Callar, Confesar\}$.
- Sumando la constante 5 a cada pago para obtener así números positivos (esto no cambia el juego), obtenermos,

$u_1(Callar, Callar) = u_2(Callar, Callar) = 4$,
$u_1(Callar, Confesar) = u_2(Confesar, Callar) = 0$,
$u_1(Confesar, Callar) = u_2(Callar, Confesar) = 5$, y
$u_1(Confesar, Confesar) = u_2(Confesar, Confesar) = 1$.

Para encontrar la solución a este tipo de juegos, necesitamos hablar de estrategias dominantes.

Definición 4. *Una **estrategia dominante** es aquella que un jugador elige siempre, independientemente de lo que realicen el resto de jugadores, porque le da un mayor o igual beneficio que cualquier otra acción posible.*

Esto se traduce en que, una estrategia s_i' es dominante si,

$$u_i(s_1, ..., s_{i-1}, s_i, s_{i+1}, ..., s_n) \leq u_i(s_1, ..., s_{i-1}, s_i', s_{i+1}, ..., s_n)$$

para toda estrategia s_i del jugador i y toda combinación de estrategias del resto de jugadores.

En el ejemplo anterior, la estrategia dominante para cada jugador es Confesar, ya que si el otro jugador escogiera confesar, tendría una pena de 4 años, en vez de 5. En cambio, si el otro jugador escoge Callar, el jugador no tendría pena en vez de 1 año de prisión. Se puede visualizar de la siguiente forma:

$$u_1(Callar, Callar) = -1 < 0 = u_1(Confesar, Callar)$$
$$u_1(Callar, Confesar) = -5 < -1 = u_1(Confesar, Callar)$$

Es claro que ambos jugadores tienen una estrategia dominante, podemos predecir el resultado del juego.

Antes de profundizar en el análisis y la resolución de este tipo de juegos, es conveniente introducir y definir los conceptos de estrategias puras y estrategias mixtas, que son fundamentales para entender la lógica y el comportamiento de los jugadores. Estas estrategias se refieren a la forma en que los jugadores eligen sus acciones en cada situación del juego, y tienen implicaciones importantes para el equilibrio y el resultado final del mismo.

En un juego $G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$, una estrategia para el jugador i es una lotería simple. Es decir, es un vector de probabilidad $(p_1, ..., p_n)$ con $0 \leq p_k \leq 1$ para cada k tal que $p_1 + p_2 + ... + p_n = 1$, donde cada p_k denota la probabilidad de que el jugador i elija la opción $s_k \in S_i$.

Definición 5. *Una **estrategia pura** es una estrategia que da probabilidad 1 a una única opción y probabilidad 0 al resto. Llamamos **mixtas** a las estrategias que no son puras.*

Un posible criterio para hallar una solución de un juego consiste en identificar las estrategias dominantes para cada jugador. Cualquier perfil de estrategias dominantes formará parte de la solución del juego. Bajo el supuesto de racionalidad de los jugadores, éstos elegirán las estrategias dominantes que les reporten la mayor utilidad, sin importar la información que tengan sobre los otros jugadores. Este método es válido para algunos juegos con información incompleta, en los que el jugador puede ignorar el pago del oponente, pero aún así puede jugar una estrategia dominante.

La técnica de las estrategias dominantes nos permite encontrar una solución de un juego cuando cada jugador tiene una estrategia que le garantiza el mejor resultado posible, sin importar lo que hagan los demás. Sin embargo, esta técnica no siempre es aplicable, ya que puede haber juegos en los que ningún jugador tenga una estrategia dominante o en los que haya más de una. En estos casos, necesitamos otro criterio para determinar una solución razonable del juego, que tenga en cuenta las expectativas y las reacciones de los jugadores. Este criterio se basa en el concepto de equilibrio de Nash, que es una situación en la que ningún jugador tiene incentivos para desviarse de su estrategia actual, dado lo que hacen los demás. El equilibrio de Nash es una condición necesaria (y a veces suficiente) para que un perfil de estrategias sea la solución del juego, es decir, una predicción válida sobre el comportamiento de jugadores racionales.

Nos centraremos ahora en el equilibrio de Nash para estrategias puras.

Definición 6. *En el juego* $G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$*, decimos que el perfil de estrategias puras* $s'_1, s'_2, ..., s'_n$ *es un* ***equilibrio de Nash*** *si para cada jugador* i*,*

$$u_i(s'_1, s'_2, ..., s'_{i-1}, s_i, s'_{i+1}, s'_n) \leq u_i(s'_1, s'_2, ..., s'_{i-1}, s'_i, s'_{i+1}, s'_n)$$

para todo $s_i \in S_i$*.*

Volvamos al ejemplo del dilema del prisionero, esta vez lo solucionaremos por el equilibrio de Nash.

Ejemplo 6. Equilibrio de Nash en el dilema del prisionero.

La matriz de pagos era la siguiente:

		Joven 2	
		Callar	Confesar
Joven 1	Callar	(-1,-1)	(-5,0)
	Confesar	(0,-5)	(-4,-4)

Cuadro 1.3: Dilema del prisionero en forma normal.

Iremos analizando cada uno de los cuatro perfiles para encontrar un equilibrio de Nash.

Supongamos que el perfil (Callar, Callar) es un equilibrio de Nash. En este caso, el Joven 1 (J1) prevé que el Joven 2 (J2) jugará Callar, pero si esto fuera así, entonces J1 debería elegir Confesar, así tendría una pena menor. Por tanto, nos situamos en la celda (Confesar, Callar). Podemos realizar el mismo argumento con J2, que decidirá cambiar a la posición (Callar. Confesar).

Supongamos que el equilibrio de Nash es (Confesar, Callar), como antes, si fijamos la decisión de J2, entonces J1 no tendría porqué cambiar la decisión. Pero supongamos que fijamos la decisión de J1, entonces sabiendo que J1 elige Confesar, J2 decidirá Confesar

para tener una pena menor. Tampoco es un equilibrio de Nash. Análogamente con la posición (Callar, Confesar).

Por último, estudiamos la casilla (Confesar, Confesar). Si J1 sabe que J2 escogerá Confesar, no le interesa cambiar su decisión, pues tendría 1 año más de pena. Y viceversa, si J2 sabe que J1 escoge Confesar, no le interesaría escoger Callar, pues también tendría 1 año más de pena. Este sí es un equilibrio de Nash.

Una manera rápida de encontrar los posibles equilibrios de Nash es la siguiente: Primero fijamos la decisión de J2 en Callar, y escogemos el mayor valor para J1, será el valor 0 de la jugada (Confesar, Callar). Realizamos lo mismo fijando ahora la decisión de J2 en Confesar, obtendremos el valor -4.

Con un proceso análogo para J2, obtenemos la siguiente tabla, donde se han subrayado los mayores valores siguiendo este proceso.

		Joven 2	
		Callar	Confesar
Joven 1	Callar	(-1,-1)	(-5,$\underline{0}$)
	Confesar	($\underline{0}$,-5)	($\underline{-4}$,$\underline{-4}$)

Vemos que el equilibrio de Nash coincidide con las jugadas que tienen todos sus elementos subrayados.

Ejemplo 7. Piedra, papel o tijera.

Consideramos la siguiente matriz de pagos para el famoso juego:

		Jugador 2 (J2)		
		Piedra	Papel	Tijera
Jugador 1 (J1)	Piedra	(0,0)	(-1,1)	(1,-1)
	Papel	(1,-1)	(0,0)	(-1,1)
	Tijera	(-1,1)	(1,-1)	(0,0)

Cuadro 1.4: Juego piedra, papel o tijera en forma normal.

Realizando el mismo procedimiento que en el ejemplo anterior, vemos que no existe un equilibrio de Nash para estrategias puras, dado que en ninguna casilla de la matriz tenemos ambos elementos subrayados.

		Jugador 2 (J2)		
		Piedra	Papel	Tijera
Jugador 1 (J1)	Piedra	(0,0)	(-1,<u>1</u>)	(<u>1</u>,-1)
	Papel	(<u>1</u>,-1)	(0,0)	(-1,<u>1</u>)
	Tijera	(-1,<u>1</u>)	(<u>1</u>,-1)	(0,0)

Cuadro 1.5: Equilibrio de Nash en el juego piedra, papel o tijera.

Como hemos visto en el ejemplo del juego de piedra papel o tijera, hay juegos en los que no existe un equilibrio de Nash en estrategias puras, es decir, no hay una combinación de acciones que sea la mejor respuesta simultánea para todos los jugadores. Esto significa que los jugadores tienen incentivos para desviarse de su estrategia inicial si conocen la del oponente. Sin embargo, esto no implica que el juego no tenga equilibrio, sino que hay que buscarlo en otro tipo de estrategias: las estrategias mixtas. Una estrategia mixta, como mencionamos anteriormente, consiste en asignar una probabilidad a cada acción posible y elegir al azar según dicha probabilidad. De esta forma, los jugadores pueden introducir cierta incertidumbre en su comportamiento y evitar ser explotados por el oponente. En el siguiente apartado veremos cómo se define y se calcula el equilibrio de Nash en estrategias mixtas y cómo se aplica a diferentes juegos.

Ejemplo 8. Juego de las monedas.

		Jugador 2 (J2)	
		Cara	Cruz
Jugador 1 (J1)	Cara	(1,-1)	(-1,1)
	Cruz	(-1,1)	(1,-1)

Cuadro 1.6: Juego de las monedas en forma normal.

En este juego, los conjuntos de estrategias puras son $S_1 = S_2 = \{Cara, Cruz\}$. Tenemos dos vectores, tal como definimos anteriormente, (p_1, p_2) tal que $0 \leq p_1, p_2 \leq 1$ con $p_1 + p_2 = 1$, y (q_1, q_2) tal que $0 \leq q_1, q_2 \leq 1$ con $q_1 + q_2 = 1$. Dicho de otra forma, los dos vectores tendrán la forma $(p_1, 1 - p_1)$ y $(q_1, 1 - q_1)$. Por simplificar notación, nuestros vectores serán $\mathbf{P} = (p, 1-p)$ y $\mathbf{Q} = (q, 1-q)$ con $0 \leq p, q \leq 1$. Tendremos que $\mathbf{P}$ es nuestro vector de probabilidades para el Jugador 1, mientras que $\mathbf{Q}$ lo será para el Jugador 2.

Hagamos, por ejemplo, la estrategia $((p, 1-p), Cara)$:

$$\begin{aligned} U_1((p,1-p), Cara) &= p \cdot u_1(Cara, Cara) + (1-p) \cdot u_1(Cruz, Cara) \\ &= p \cdot 1 + (1-p) \cdot (-1) = 2p - 1 \end{aligned}$$

$$\begin{aligned} U_2((p,1-p), Cara) &= p \cdot u_2(Cara, Cara) + (1-p) \cdot u_2(Cruz, Cara) \\ &= p \cdot (-1) + (1-p) \cdot (1) = 1 - 2p \end{aligned}$$

Realizando el mismo procedimiento, se obtiene que,

$$\begin{aligned} U_1((p,1-p), Cruz) &= 1 - 2p \\ U_2((p,1-p), Cruz) &= 2p - 1 \end{aligned}$$

Ya tenemos todos los elementos para calcular la estrategia $((p, 1-p), (q, 1-q))$, utilizando los datos obtenidos arriba:

$$\begin{aligned} U_1((p,1-p),(q,1-q)) &= q \cdot U_1((p,1-p), Cara) + (1-q) \cdot U_1((p,1-p), Cruz) \\ &= q \cdot (2p-1) + (1-q) \cdot (1-2p) = 1 - 2p - 2q + 4pq \end{aligned}$$

$$\begin{aligned} U_2((p,1-p),(q,1-q)) &= p \cdot q \cdot (-1) + (1-p) \cdot q \cdot (1) \\ &\quad + p \cdot (1-q) \cdot (1) + (1-p) \cdot (1-q) \cdot (-1) \\ &= -1 + 2p + 2q - 4pq \end{aligned}$$

En este ejemplo anterior hemos visto cómo los jugadores pueden usar estrategias mixtas para jugar a un juego en el que no existe un equilibrio de Nash en estrategias puras. Esto significa que los jugadores pueden hacer indiferente al oponente entre sus acciones, ya que su beneficio esperado es el mismo para cualquier acción. Sin embargo, ¿cómo podemos saber si una estrategia mixta es óptima para un jugador? ¿Qué condiciones debe cumplir una estrategia mixta para ser un equilibrio de Nash? Para responder a estas preguntas, necesitamos definir formalmente el concepto de equilibrio de Nash en estrategias mixtas y ver cómo se puede hallar mediante un método algebraico o gráfico.

Definición 7. *En el juego $G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$, decimos que el perfil de estrategias mixtas $\sigma' = (\sigma'_1, ..., \sigma'_n)$ es un* ***equilibrio de Nash*** *si para cada jugador i:*

$$U_i(\sigma'_1, ..., \sigma'_{i-1}, \sigma_i, \sigma'_{i+1}, ..., \sigma'_n) \leq U_i(\sigma'_1, ..., \sigma'_{i-1}, \sigma'_i, \sigma'_{i+1}, ..., \sigma'_n)$$

para toda estrategia mixta.

La definición de equilibrio de Nash en estrategias mixtas nos dice que una combinación de estrategias mixtas es un equilibrio de Nash si ningún jugador puede mejorar su pago esperado cambiando unilateralmente su estrategia mixta. Sin embargo, esta definición no nos dice cómo encontrar un equilibrio de Nash en estrategias mixtas ni qué propiedades tiene. Para ello, necesitamos el siguiente teorema, que nos da una condición necesaria y suficiente para que una combinación de estrategias mixtas sea un equilibrio de Nash, y que además nos permite identificar las estrategias puras que forman parte de las estrategias mixtas de equilibrio.

Teorema 1. *En el juego* $G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$*, la combinación de estrategias mixtas* $\sigma' = (\sigma'_1, ..., \sigma'_n)$ *es un equilibrio de Nash si y sólo si para cada jugador* J_i *con estrategia mixta* $\sigma'_i = (\sigma_i^{'1}, ..., \sigma_i^{'j}, ...\sigma_i^{'k})$ *tener* $\sigma_i^{'j} > 0$ *implica que la estrategia pura* $s_i^j \in S_i$ *es una respuesta óptima a* $\sigma'_{-i} = (\sigma'_1, ..., \sigma'_{i-1}, \sigma'_{i+1}, ..., \sigma'_n)$.

Es decir, es equilibrio de Nash si y solo si para cada jugador J_i*,* σ'_i *es una solución de max* $U_i(\sigma'_1, ..., \sigma'_{i-1}, \sigma_i, \sigma'_{i+1}, ..., \sigma'_n)$ *en la variable* σ_i.

El significado del teorema es que si un jugador asigna una probabilidad positiva a una cierta estrategia pura en su estrategia mixta, entonces esa estrategia pura debe ser una respuesta óptima a las estrategias mixtas de los otros jugadores. Si no fuera así, el jugador podría aumentar su pago esperado asignando cero probabilidad a esa estrategia pura, redistribuyendo la probabilidad entre sus otras estrategias. En otras palabras, si una estrategia pura no es óptima dadas las estrategias de los otros jugadores, entonces no debería tener una probabilidad positiva en la estrategia mixta del jugador en equilibrio.

Demostración. [⇒]

Sea $\sigma' = (\sigma'_1, ..., \sigma'_n)$ un equilibro de Nash. Supongamos que dado un jugador i, su estrategia pura s_i^j pertenece al soporte de σ'_i. Veamos por reducción al absurdo que s_i^j es respuesta óptima a $\sigma'_{-i} = (\sigma'_1, ..., \sigma'_{i-1}, \sigma'_{i+1}..., \sigma'_n)$. Si no lo fuera, existiría una estrategia pura s_i^l tal que

$$U_i(\sigma'_1, ...\sigma'_{i-1}, s_i^l, \sigma'_{i+1}, ..., \sigma'_n) > U_i(\sigma'_1, ...\sigma'_{i-1}, s_i^j, \sigma'_{i+1}, ..., \sigma'_n)$$

Si σ_i^* es la estrategia mixta que es en todo igual a $\sigma'_i = (\sigma_i^{'1}, ..., \sigma_i^{'k})$ salvo que juega la estrategia pura s_i^j con probabilidad cero y la estrategia pura s_i^l con probabilidad $\sigma_i^{'j} + \sigma_i^{'l}$, dicha estrategia σ_i^* es una respuesta estrictamente mejor a σ'_{-i} que σ'_i. En efecto,

$$\begin{aligned}
U_i(\sigma'_1, ...\sigma'_{i-1}, s_i^j, \sigma'_{i+1}, ..., \sigma'_n) &= \sum_{h\neq j, h\neq l} \sigma_i^{'h} U_i(\sigma'_1, ..., \sigma'_{i-1}, s_i^h, \sigma'_{i+1}, ..., \sigma'_n) \\
&+ (\sigma_i^{'j} + \sigma_i^{'l}) U_i(\sigma'_1, ..., \sigma'_{i-1}, s_i^l, \sigma'_{i+1}, ..., \sigma'_n) \\
&> U_i(\sigma'_1, ..., \sigma'_{i-1}, \sigma'_i, \sigma'_{i+1}, ..., \sigma'_n) \\
&= \sum_{h\neq j, h\neq l} \sigma_i^{'h} U_i(\sigma'_1, ..., \sigma'_{i-1}, s_i^h, \sigma'_{i+1}, ..., \sigma'_n) \\
&+ \sigma_i^{'j} U_i(\sigma'_1, ..., \sigma'_{i-1}, s_i^j, \sigma'_{i+1}, ..., \sigma'_n) \\
&+ \sigma_i^{'l} U_i(\sigma'_1, ..., \sigma'_{i-1}, s_i^j, \sigma'_{i+1}, ..., \sigma'_n)
\end{aligned}$$

Por lo tanto, σ'_i no es respuesta óptima a σ'_{-i}, lo que contradice la hipótesis.

[⇐]

Supongamos que $\sigma' = (\sigma'_1, ..., \sigma'_n)$ cumple, que para cada jugador i que todas sus estrategias puras s_i^j soporte de σ'_i son respuesta óptima a

$$\sigma'_{-i} = (\sigma'_1, ..., \sigma'_{i-1}, \sigma'_{i+1}..., \sigma'_n)$$

Sea M_i la máxima utilidad que puede ser alcanzada por el jugador i si el resto de jugadores juegan σ'_{-i},

$$M_i = \max_{s_i \in S_i}\{U_i(\sigma'_1, ...\sigma'_{i-1}, s_i, \sigma'_{i+1}, ..., \sigma'_n)\}$$

En ese caso,

$$\begin{aligned} U_i(\sigma'_1, ...\sigma'_{i-1}, s_i^j, \sigma'_{i+1}, ..., \sigma'_n) &= \max_{s_i \in S_i}\{U_i(\sigma'_1, ...\sigma'_{i-1}, s_i, \sigma'_{i+1}, ..., \sigma'_n)\} \\ &= M_i \text{ para todo } s_i^j \text{ en el soporte de } \sigma'_i \end{aligned}$$

por lo tanto,

$$\begin{aligned} U_i(\sigma'_1, ...\sigma'_{i-1}, \sigma'_i, \sigma'_{i+1}, ..., \sigma'_n) &= \sum_{\sigma_i^{'j}>0} \sigma_i^{'j} U_i(\sigma'_1, ...\sigma'_{i-1}, s_i^j, \sigma'_{i+1}, ..., \sigma'_n) \\ &= \sum_{\sigma_i^{'j}>0} U_i(\sigma'_1, ...\sigma'_{i-1}, \sigma_i, \sigma'_{i+1}, ..., \sigma'_n) \text{ para cualquier } \sigma_i \end{aligned}$$

□

Al establecer una condición necesaria y suficiente para que una combinación de estrategias mixtas sea un equilibrio de Nash, el teorema permite determinar si una solución propuesta es un equilibrio de Nash y también ayuda a encontrar equilibrios de Nash en juegos específicos.

Volvamos al ejemplo del juego de las monedas, para calcular este equilibrio de Nash.

Ejemplo 9. Juego de las monedas.

		Jugador 2 (J2)	
		Cara	Cruz
Jugador 1 (J1)	Cara	(1,-1)	(-1,1)
	Cruz	(-1,1)	(1,-1)

Cuadro 1.7: Matriz de pagos en el juego de las monedas.

Tendremos como antes los vectores $\mathbf{P} = (p, 1-p)$ y $\mathbf{Q} = (q, 1-q)$ como estrategias mixtas para ambos jugadores. Primero debemos fijar la estrategia mixta de J2 y calculamos para J1:

$$\begin{aligned} U_1(Cara, (q, 1-q)) &= q \cdot (1) + (1-q) \cdot (-1) = 2q - 1 \\ U_1(Cruz, (q, 1-q)) &= q \cdot (-1) + (1-q) \cdot (1) = 1 - 2q \end{aligned}$$

Comparando, tenemos que:

$$\begin{aligned} U_1(Cara, (q, 1-q)) > U_1(Cruz, (q, 1-q)) &\Leftrightarrow 2q - 1 > 1 - 2q \Leftrightarrow q > \frac{1}{2} \\ U_1(Cara, (q, 1-q)) < U_1(Cruz, (q, 1-q)) &\Leftrightarrow 2q - 1 < 1 - 2q \Leftrightarrow q < \frac{1}{2} \\ U_1(Cara, (q, 1-q)) = U_1(Cruz, (q, 1-q)) &\Leftrightarrow 2q - 1 = 1 - 2q \Leftrightarrow q = \frac{1}{2} \end{aligned}$$

Por lo tanto, la respuesta óptima de J1 será:

$$R_1(q) = \begin{cases} Cruz & si \quad q > \frac{1}{2} \\ Cara & si \quad q < \frac{1}{2} \\ Cualquiera & si \quad q = \frac{1}{2} \end{cases}$$

Realizamos el mismo proceso para la respuesta óptima de J2:

$$U_2((p, 1-p), Cara) = p \cdot (-1) + (1-p) \cdot (1) = 1 - 2p$$
$$U_2((p, 1-p), Cruz) = p \cdot (1) + (1-q) \cdot (-1) = 2p - 1$$

Luego,

$$U_2((p, 1-p), Cara) > U_2((p, 1-p), Cruz) \Leftrightarrow 1 - 2p > 2p - 1 \Leftrightarrow p < \frac{1}{2}$$
$$U_2((p, 1-p), Cara) < U_2((p, 1-p), Cruz) \Leftrightarrow 1 - 2p < 2p - 1 \Leftrightarrow p > \frac{1}{2}$$
$$U_2((p, 1-p), Cara) = U_2((p, 1-p), Cruz) \Leftrightarrow 1 - 2p = 2p - 1 \Leftrightarrow p = \frac{1}{2}$$

Por lo que obtenemos que:

$$R_2(p) = \begin{cases} Cara & si \quad p > \frac{1}{2} \\ Cruz & si \quad p < \frac{1}{2} \\ Cualquiera & si \quad p = \frac{1}{2} \end{cases}$$

Por tanto, vemos que la intersección entre $R_1(q)$ y $R_2(p)$ es el punto $p = q = \frac{1}{2}$. Luego el equilibrio de Nash será en la estrategia mixta $\{(\frac{1}{2}, \frac{1}{2}), (\frac{1}{2}, \frac{1}{2})\}$. Esto significa, que el Jugador 1 jugará Cara con probabilidad $\frac{1}{2}$ y jugará Cruz con la misma probabilidad, sucede exactamente igual para el Jugador 2.

Como hemos visto en el ejemplo anterior, hay juegos que no tienen ningún equilibrio de Nash en estrategias puras, sino sólo en estrategias mixtas. El siguiente teorema nos demuestra que no siempre existe equilibrio de Nash en etrategias puras, y que esta existencia depende de las propiedades del juego. Nos dice que si el juego es finito (es decir, tiene un número finito de jugadores y de estrategias puras para cada jugador) y cumple la condición de no saturación (es decir, no hay empates entre los pagos de las distintas estrategias puras), entonces existe al menos un equilibrio de Nash en estrategias puras. Pero si el juego no cumple alguna de estas condiciones, puede ocurrir que no haya ningún equilibrio de Nash en estrategias puras, como en el ejemplo anterior.

Definición 8. *Dada una función real $f : A \to B$, con $A \subset \mathbb{R}^k$ convexo y $B \subset \mathbb{R}$, decimos que es* ***cuasicóncava*** *si $f(\lambda \mathbf{x} + (1-\lambda)\mathbf{y}) \geq min\{f(\mathbf{x}), f(\mathbf{y})\}$, $\forall \lambda \in [0,1]$, $\forall \mathbf{x}, \mathbf{y} \in A$.*

Teorema 2. *Sea el juego $G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$, tal que para todo jugador i se cumple:*

1. *S_i es un subconjunto no vacío, compacto y convexo de un espacio de $\mathbb{R}^k$.*

2. *u_i es continua en todo su dominio $S = S_1 \times ... \times S_n$, y es cuasicóncava en la variable s_i.*

Entonces, existe al menos un equilibrio de Nash en estrategias puras.

Demostración. Definamos para cada jugador i su correspondencia de respuesta óptima R_i, que asigna a cada perfil de estrategias puras $\mathbf{s} = (s_1, ..., s_{i-1}, s_i, s_{i+1}, ..., s_n) = (s_i, \mathbf{s_{-i}})$ el conjunto $R_i(\mathbf{s})$ de las estrategias puras de i que son respuesta óptima a $\mathbf{s_{-i}}$. Es decir, $R_i(s) = \{x_i \in S_i : x_i \text{ es solución de } \max_{y_i \in S_i} u_i(y_i, \mathbf{s_{-i}})\}$

Definamos la correspondencia global de respuesta óptima R como aquella que asigna a cada perfil $\mathbf{s}$ el producto cartesiano de los conjuntos de respuesta óptima individuales. Así pues, R es una correspondencia de $\mathbf{S}$ en $\mathbf{S}$ tal que

$$R(\mathbf{s}) = R_1(\mathbf{s}) \times \cdots \times R_n(\mathbf{s})$$

Veamos que R es hemicontinua superiormente. $\mathbf{S}$ es no vacío, compacto y convexo por ser el producto cartesiano de los S_i. Por hipótesis S_i es no vacío y compacto. Como u_i es continua en S, por el teorema de Weierstrass tenemos un máximo, por lo que $R_i(s)$ es no vacío, y por tanto $R(s)$ es no vacío. Falta ver que si las soluciones $\{s^k\}$ y $\{t^k\}$, con $s^k \in S$ y $t^k \in R(s^k)$, para cada k tienen por límite $\mathbf{s} \in S$ y $\mathbf{t} \in S$ respectivamente. Esto implica que $\mathbf{t} \in R(\mathbf{s})$. Pero

$$\lim_{k\to\infty} s^k = s \in S \Leftrightarrow \text{ Para cada jugador } i,\ lim_{k\to\infty} s_i^k = s_i \in S_i$$

$$\lim_{k\to\infty} t^k = t \in S \Leftrightarrow \text{ Para cada jugador } i,\ lim_{k\to\infty} t_i^k = t_i \in S_i$$

$$t^k \in R(s^k) \Rightarrow \text{ Para cada jugador } i,\ t_i^k \in R_i(s^k)$$
$$\Rightarrow \text{ Para cada jugador } i,\ u_i(t_i^k, s_{-i}^k) \geq u_i(z_i, s_{-i}^k) \text{ para todo } z_i \in S_i$$

Como $u_i(t_i, s_{-i})$ es continua en S_i se verifica que

$$u_i(t_i, s_{-i}) \geq u_i(z_i, s_{-i}) \text{ para todo } z_i \in S_i \Rightarrow t_i \in R_i(s) \text{ para todo } i$$

luego $t \in R(s)$.

Por otra parte, por ser las u_i cuasicóncavas en s_i sobre S_i, el conjunto $R_i(\mathbf{s})$ es convexo, por lo que $R(\mathbf{s})$ también lo es.

La correspondencia R cumple las hipótesis del teorema de Kkutani, por lo que podemos afirmar que R tiene al menos un punto fijo $\mathbf{s'}$.

Sea $\mathbf{s'} = (s'_1, ..., s'_n)$ un punto fijo de R, tenemos que $\mathbf{s'} \in R(\mathbf{s'})$, lo que implica que $s'_i \in R_i(\mathbf{s'})$ para todo i, es decir, s'_i es respuesta óptima a $\mathbf{s'_{-i}}$.

□

Debido a este teorema, podemos enunciar el siguiente, como ya hizo John Nash en 1950.

Teorema 3. *En todo juego finito* $G = \{J; S_1, S_2, ..., S_n; u_1, u_2, ..., u_n\}$*, existe al menos un equilibrio de Nash en estrategias mixtas.*

Demostración. Dado el juego G, sea $S_i = \{s_i^1, ..., s_i^{m_i}\}$ y $S = S_1 \times ... \times S_n$. Llamamos $\Delta(G)$ al juego cuyos jugadores son los mismos que los de G, cuyos conjuntos de estrategias puras para cada jugador i son los símplices $\Delta(S_i)$ constituidos por las distribuciones de probabilidad sobre S_i, y cuyos pagos son los pagos esperados,

$$U_i(\sigma_1, ..., \sigma_n) = \sum_{s \in S} \sigma_1^{j(1)} \cdots \sigma_n^{j(n)} u_i(s_1^{j(1)}, ..., s_n^{j(n)})$$

donde u_i está bien definida sobre $\Delta = \Delta(S_1) \times ... \times \Delta(S_n)$, y donde $\sigma_i^{j(i)}$ es la probabilidad con que el jugador i juega su estrategia pura $s_i^{j(i)}$ en su estrategia mixta σ_i. El juego $\Delta(G) = \{\Delta(S_1), ..., \Delta(S_n); U_1, ..., U_n\}$ cumple las hipótesis del teorema anterior, ya que para todo jugador i:

1. $\Delta(S_i)$ es un subconjunto no vacío, compacto y convexo del espacio R^{m_i}.
2. la función de pagos esperados U_i es continua en todo su dominio $\Delta = \Delta(S_1) \times ... \times \Delta(S_n)$ y es cuasicóncava en la variable σ_i (por ser función afín).

Podemos concluir que existe un equilibrio de Nash en estrategias puras del juego $\Delta(G)$, que no es otro que un equilibrio de Nash en las estrategias mixtas del juego G.

□

En esta sección hemos estudiado los juegos en los que los jugadores eligen sus acciones de forma simultánea y conocen las consecuencias de todas las posibles combinaciones de acciones. Hemos visto cómo encontrar el equilibrio de Nash en estrategias puras y en estrategias mixtas. Sin embargo, no hemos abordado otros conceptos importantes para los juegos estáticos, como el algoritmo Minimax[2], las estrategias racionalizables o los refinamientos del equilibrio de Nash.

1.3 Juegos estáticos con información incompleta

Los juegos estáticos con información incompleta son aquellos en los que al menos un jugador no conoce algún aspecto relevante de la estructura del juego, como las preferencias, las acciones o los pagos de otro jugador. Estos juegos también se conocen como juegos bayesianos, porque los jugadores tienen que formar creencias sobre la información desconocida y actualizarlas según el principio de Bayes. El concepto de solución más utilizado en estos juegos es el equilibrio bayesiano de Nash, que es una generalización del equilibrio de Nash para juegos con información asimétrica.

Algunas aplicaciones típicas de los juegos estáticos con información incompleta son los modelos de duopolio con costes inciertos, las subastas con valoraciones privadas o las contribuciones a un bien público con preferencias heterogéneas. Estos ejemplos ilustran cómo la información incompleta afecta a las decisiones estratégicas de los agentes y a los resultados del juego.

Para dar el primer paso en nuestro análisis, vamos a introducir un elemento de incertidumbre en una de las opciones que tienen los jugadores a la hora de elegir su estrategia.

[2]Debido a que este método se utiliza también en Inteligencia Artifical, será explicado en el Capítulo 2.

Para ello, vamos a utilizar la forma extensiva del juego, que nos permite representar de manera más adecuada esta situación de información incompleta, en la que no se sabe con certeza qué va a ocurrir.

Ejemplo 10. Dilema del prisionero modificado.

Ya introdujimos el juego del dilema del prisionero anteriormente, para este ejemplo, haremos unas modificaciones sobre este juego. Cuando ambos jugadores escojan Callar, los pagos vendrán dados por una probabilidad. Además, los pagos del Ejemplo 6 se han escrito de forma positiva, sumando a cada pago 5 unidades, esto no cambia nada, simplemente se realiza por comodidad. Veamos el siguiente diagrama:

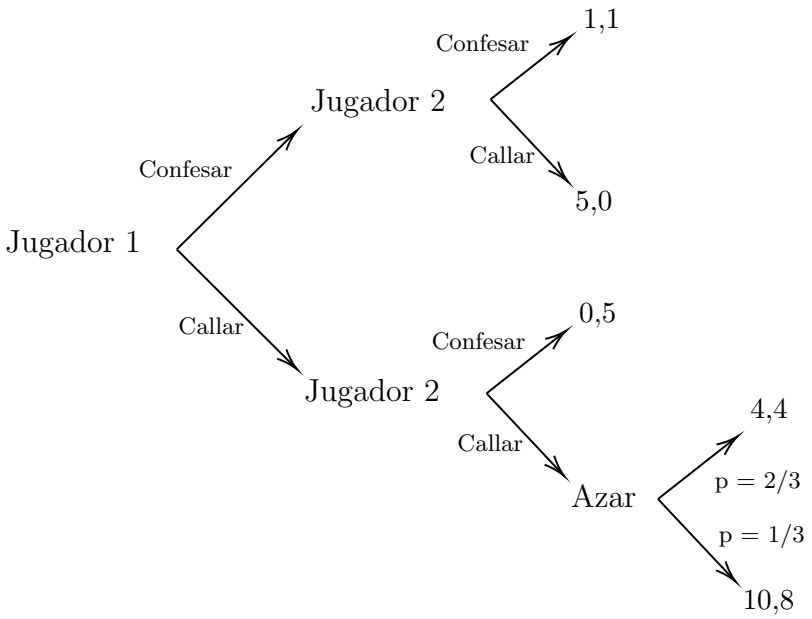


Figura 1.3: Dilema del prisionero modificado en forma extensiva.

Podemos ver, que si ambos jugadores escogen la opción Callar, nos situaríamos en el resultado (Callar,Callar), que tiene unos pagos esperados de $10\frac{1}{3} + 4\frac{2}{3} = 6$ para el Jugador 1, y $8\frac{1}{3} + 4\frac{2}{3} = \frac{16}{3}$ para el Jugador 2. Por tanto, podríamos reconvertir el gráfico en el siguiente:

Tenemos un juego como los estudiados previamente, en el que podemos aplicar algún método ya visto.

Veamos otro caso práctico, en el que se introduce una asimetría de información entre los jugadores, de tal forma que el resultado de las acciones aleatorias que se producen en el juego es conocido por alguno de ellos, pero no por todos.

Ejemplo 11. Dilema del prisionero modificado modificado.

En el ejemplo anterior, ambos jugadores conocían las probabilidades asignadas a cada pago en la situación (Calla, Calla). Supongamos ahora que las probabilidades se asignan al principio del juego, y únicamente el Jugador 1 tiene conocimiento de qué probabilidad se ha asignado. Un posible grafo es el siguiente:

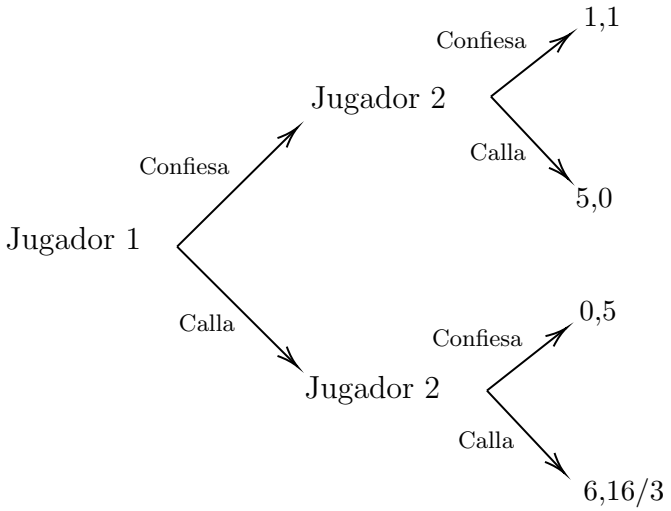


Figura 1.4: Dilema del prisionero modificado en forma extensiva con pagos calculados.

Todas los pagos son los mismos excepto en los casos señalados. El Jugador 2 a la hora de tomar la decisión desconoce cuál de los dos pagos es el correspondiente, pues la probabilidad fue asignada previamente. Este juego es un claro ejemplo de juego estático con información incompleta, pues las decisiones son simultáneas, pero uno de los jugadores desconoce la información sobre los pagos en una de las decisiones.

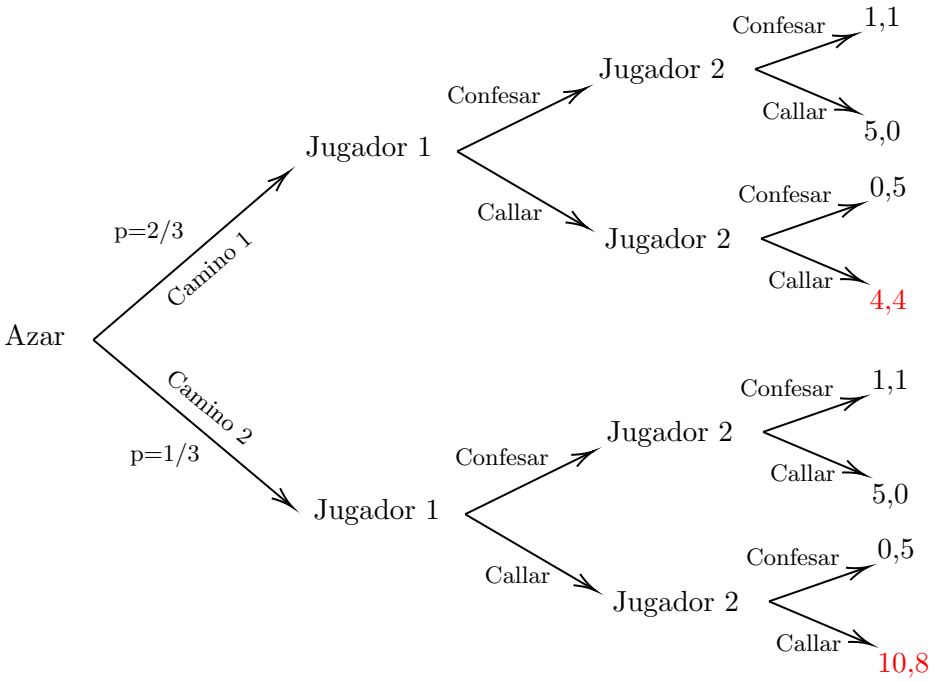


Figura 1.5: Otra modificación del dilema del prisionero.

¿Cuál es una posible solución a este juego? Como el Jugador 1 sabe cual de los caminos se ha seguido, luego sus estrategias dependerán del camino seguido, esto quiere decir que serán del tipo:

- Confesar-Confesar: J1 juega Confesar si sigue el Camino 1, juega Confesar si sigue el Camino 2.
- Confesar-Callar: J1 juega Confesar si sigue el Camino 1, juega Callar si sigue el Camino 2.
- Callar-Confesar: J1 juega Callar si sigue el Camino 1, juega Confesar si sigue el Camino 2.
- Callar-Callar: J1 juega Callar si sigue el Camino 1, juega Callar si sigue el Camino 2.

Entonces, una forma normal de representar este juego con los pagos esperados, con $p = 2/3$ y $q = 1/3$ es la siguiente:

		J2	
		Confesar	Callar
J1	Confesar-Confesar	(1p+1q,1p+1q)	(5p+5q,0p+0q)
	Confesar-Callar	(1p+0q,1p+5q)	(5p+10q,0p+8q)
	Callar-Confesar	(0p+1q,5p+1q)	(4p+5q,4p+0q)
	Callar-Callar	(0p+0q,5p+5q)	(4p+10q,4p+8q)

Calculando en cada celda, tenemos la siguiente matriz, donde ya se ha señalado el equilibrio de Nash en estrategias puras.

		J2	
		Confesar	Callar
J1	Confesar-Confesar	(<u>1</u>,<u>1</u>)	(5,0)
	Confesar-Callar	(2/3,7/3)	(<u>20/3</u>,<u>8/3</u>)
	Callar-Confesar	(1/3,<u>11/3</u>)	(13/3,8/3)
	Callar-Callar	(0,<u>5</u>)	(6,16/3)

Cuadro 1.8: Dilema del prisionero ahora en forma matricial.

Por lo tanto, el equilibrio de Nash se obtiene en (Confesar-Confesar, Confesar) y con (Confesar-Callar, Callar).

Después de ver algún ejemplo de juegos estáticos con información incompleta, vamos a introducir algunos conceptos teóricos que nos permitirán analizar este tipo de juegos de

forma rigurosa. Estos conceptos son los de juego bayesiano y equilibrio bayesiano.

Un juego bayesiano es un juego en el que los jugadores tienen información asimétrica sobre algún aspecto del juego, como los pagos, las acciones o los tipos de los otros jugadores. Esto significa que los jugadores tienen creencias o probabilidades subjetivas sobre la información que desconocen, y que pueden actualizar sus creencias según la regla de Bayes cuando observan las acciones de los otros jugadores. Un juego bayesiano se puede representar mediante la introducción de un jugador ficticio que asigna aleatoriamente un tipo a cada jugador según una distribución de probabilidad conocida por todos.

Un equilibrio bayesiano es un concepto de solución para los juegos bayesianos que generaliza el equilibrio de Nash. Un equilibrio bayesiano es un perfil de estrategias, una por cada tipo de jugador, tal que cada estrategia es una mejor respuesta a las estrategias de los otros tipos, dadas las creencias de cada jugador sobre los tipos de los otros jugadores. Un equilibrio bayesiano se puede calcular mediante el uso de las funciones de mejor respuesta, que indican qué acción maximiza el pago esperado de cada tipo de jugador para cada posible creencia.

Veremos cómo se define formalmente un juego bayesiano y cómo se encuentra un equilibrio bayesiano para algunos ejemplos concretos. También veremos cómo se puede transformar un juego bayesiano en un juego dinámico con información imperfecta y cómo se puede hallar el equilibrio bayesiano mediante el equilibrio de Nash en forma normal.

Primero, introduciremos el concepto de juego bayesiano. Un juego bayesiano es un juego que incorpora la incertidumbre sobre algunos aspectos relevantes de los jugadores, como sus preferencias, costes o pagos.

Definición 9. *Un juego **bayesiano** es un tipo de juego en el que cada jugador i posee un tipo de acciones A_i, pagos u_i, un conjunto de tipos T_i, además de una suposición p_i sobre los otros jugadores. Donde cada $t_k \in T_i$ es una variable aleatoria para todo $k \leq k_i$. Cada $t_k \in T_i$ es una variable aleatoria cuya distribución de probabilidad conjunta de los tipos viene dada por la función $(t_1, ..., t_i, ...t_n)$ y que la conjetura t_i es una probabilidad condicionada denotada $p_i(t_{-i}/t_i)$ que depende de cuál sea el tipo efectivo t_i del jugador i. Además, la función de pagos de cada jugador depende de las acciones decididas y de los tipos efectivos de todos los jugadores, denotándose por $u_i(t_1, ...t_i, ...t_n, A_1, ..., A_i, ..., A_n)$. Resumiendo,*

$$G_B = \{J; A_1, ..., A_n; T_1, ..., T_n; p_1, ..., p_n; u_1, ..., u_n\} \text{ con } J = \{1, ...n\}$$

Siguiendo el mismo procedimiento que empleamos en el apartado anterior, vamos a definir las estrategias puras de cada jugador y a utilizarlas para encontrar una solución al juego.

Definición 10. *Una **estrategia pura** del jugador i es una aplicación $s_i : T_i \to A_i$ de forma que el jugador actúa de manera consistente con su información.*

Es decir, una estrategia pura en un juego bayesiano estático es una estrategia que un jugador elige siempre, como en el apartado anterior, sin tener en cuenta lo que el jugador crea o sepa sobre el otro jugador. Una estrategia pura no varía según las opiniones o datos

que tenga el jugador sobre el jugador rival.

Definición 11. *Diremos que* $\mathbf{s}^* = (s_1^*, ..., s_n^*)$ *es un* ***equilibrio Bayesiano de Nash*** *en estrategias puras si para cada jugador* i*, la estrategia* s_i^* *es una respuesta óptima a la combinación* $(s_1^*, ..., s_{i-1}^*, s_{i+1}^*, ..., s_n^*)$

Para ilustrar el concepto de equilibrio bayesiano de Nash, vamos a considerar el ejemplo anterior. ¿Cuáles son las estrategias que deberían seguir los jugadores?

Ejemplo 12. Dilema del prisionero bayesiano.

En el ejemplo anterior, podemos identificar $J = 1, 2$ como jugadores 1 y 2, $A_1 = A_2 = \{Confesar, Callar\}$ como las diferentes acciones disponibles, $T_1 = \{$Camino 1, Camino 2$\}$ y $T_2 = \{$Único$\}$ (es decir, J_2 no tiene información privada).

Las probabilidades son $P(\text{Camino 1}) = 2/3$, $P(\text{Camino 2}) = 1/3$. Por lo que, las conjeturas, calculando las probabilidades condicionadas son,

Para el Jugador 1:

$$p_1(\text{Único/Camino 1}) = \frac{P(\text{Único} \cap \text{Camino 1})}{P(\text{Camino 1})} = 1 \ ;$$
$$p_1(\text{Único/Camino 2}) = \frac{P(\text{Único} \cap \text{Camino 2})}{P(\text{Camino 2})} = 1$$

Para el jugador 2:

$$p_2(\text{Camino 1/Único}) = \frac{P(\text{Camino 1} \cap \text{Único})}{P(\text{Único})} = 2/3 \ ;$$
$$p_2(\text{Camino 2/Único}) = \frac{P(\text{Camino 2} \cap \text{Único})}{P(\text{Único})} = 1/3$$

Supongamos que $\mathbf{s}^* = (s_1^*, s_2^*)$ es una solución del problema, es decir, un perfil del equilibrio. Supongamos que la estrategia del Jugador 2 s_2^* es Confesar, entonces si seguimos el Camino 1, el Jugador 1 escogerá Confesar, y si seguimos el Camino 2, escogerá Confesar también. Vemos que fijada la estrategia del Jugador 1 $s_1^* = (Confesar - Confesar)$, la elección óptima para el Jugador 2 es Confesar, luego hemos localizado un primero equilibrio $\mathbf{s}^* = (Confesar - Confesar, Confesar)$.

Si la estrategia del Jugador 2 es Callar, realizando el mismo proceso, siguiendo el Camino 1, el Jugador 1 escogerá Confesar. Siguiendo el Camino 2 la elección será Callar. Fijando la estrategia del Jugador 1 $\mathbf{s_1}* = (Confesar - Callar)$, el Jugador 2 tendrá disponibles los siguientes pagos:

$$\text{Si J2 escoge Callar} \ \rightarrow \frac{2}{3}(0) + \frac{1}{3}(8) = 8/3$$
$$\text{Si J2 escoge Confesar} \ \rightarrow \frac{2}{3}(1) + \frac{1}{3}(5) = 7/3$$

Por lo que la estrategia que debe escoger J2 es Callar. Hemos econtrado otro equilibrio $\mathbf{s}* = (Confesar - Callar, Callar)$. Podemos concluir que el conjunto de equilibrios bayesianos en estrategias puras es

$$\{(Confesar - Confesar, Confesar),\ (Confesar - Callar, Callar)\}.$$

En los juegos estáticos de información completa, los jugadores conocen todas las estrategias posibles y los pagos asociados a cada una de ellas. Sin embargo, en los juegos de información incompleta, los jugadores no tienen acceso a toda la información necesaria para tomar decisiones óptimas. En estos casos, también se pueden utilizar estrategias mixtas para maximizar el pago esperado. Como vimos en los juegos estáticos de información completa, también podemos considerar estrategias mixtas en los juegos de información incompleta. De esta manera, podemos encontrar soluciones para juegos que de otra manera serían imposibles de resolver.

En los juegos de información incompleta, las estrategias mixtas son particularmente útiles porque permiten a los jugadores tomar decisiones óptimas incluso cuando no tienen acceso a toda la información necesaria, ya que no tenemos garantizado el equilibrio bayesiano en estrategias puras en juegos bayesianos finitos (con todos los elementos finitos).

Definición 12. *En un juego* $G_B = \{J; A_1, ..., A_n; T_1, ..., T_n; p_1, ..., p_n; u_1, ..., u_n\}$*, una* ***estrategia mixta*** *del jugador* i *es una regla de decisión que especifica una lotería sobre* A_i *por cada elemento de* T_i*.*

La importancia de las estrategias mixtas radica en que nos permiten encontrar equilibrios bayesianos en juegos bayesianos finitos. En particular, el Teorema de Existencia nos garantiza que siempre existirá al menos un equilibrio bayesiano en estrategias mixtas para cualquier juego bayesiano finito. Esto es cierto incluso si el equilibrio se encuentra en estrategias mixtas, lo que hace que las estrategias mixtas sean una herramienta poderosa para analizar juegos de información incompleta.

Teorema 4. *Dado un juego bayesiano finito* $G_B = \{J; A_1, ..., A_n; T_1, ..., T_n; p_1, ..., p_n; u_1, ..., u_n\}$*, existe un perfil de estrategias, posiblemente en estrategias mixtas, que es un equilibrio bayesiano.*

Finalicemos esta sección con dos ejemplos.

Ejemplo 13. La caza de la liebre.

En un pueblo hay dos cazadores que deben elegir entre cazar una liebre o un conejo. Hay dos cotos de caza en el pueblo, pero por temas de conservación solo se permite el acceso a uno de ellos, que se determina al azar. El Coto 1 tiene más liebres que conejos, y se quiere proteger la especie, así que solo se va al Coto 1 si al tirar un dado en el ayuntamiento sale 1 o 2. El Coto 2 tiene más conejos que liebres, y se va al Coto 2 si al tirar el dado sale cualquier otro número.

El Cazador 1 tiene a su sobrino trabajando en el ayuntamiento, y éste le dice que coto será el disponible, pero esta información no la tendrá el Cazador 2. Los cazadores han llegado al acuerdo de repartirse la carne si cazan el mismo animal. Los pagos podemos representarlos de la siguiente forma:

Coto 1 (p=1/3)		Cazador 2 (C2)	
		Conejo	Liebre
Cazador 1 (C1)	Conejo	(3,3)	(0,2)
	Liebre	(2,0)	(1,1)

Coto 2 (p=2/3)		Cazador 2 (C2)	
		Conejo	Liebre
Cazador 1 (C1)	Conejo	(1,1)	(1,4)
	Liebre	(4,1)	(4,4)

Cuadro 1.9: Matrices de pago en el juego de la caza de la liebre.

Vayamos a buscar los equilibrios:

- La respuesta óptima de C1 a la estrategia Conejo de C2 es Conejo-Liebre, sin embargo, la respuesta óptima a esta estrategia por parte de C2 es Conejo si estamos en el Coto 1, pero es Liebre si estamos en el Coto 2, por tanto no es un EB.
- La respuesta óptima de C1 a la estrategia Liebre de C2 es Liebre-Liebre, la estrategia óptima en este caso para C2 es Liebre en ambos cotos, luego aquí tenemos un EB.

Ejemplo 14. Cara-Cruz mixto.

Consideramos el siguiente juego, en el que se lanza una moneda dos veces, la primera vez solo el Jugador 1 sabe el resultado, la segunda vez, ambos jugadores escogen Cara o Cruz, generando las siguientes matrices de pago:

Cara **(p=1/2)**		J2	
		Cara	Cruz
J1	Cara	(-1,1)	(1,-1)
	Cruz	(1,-1)	(-1,1)

Cruz **(p=1/2)**		J2	
		Cara	Cruz
J1	Cara	(1,-1)	(-1,1)
	Cruz	(-1,1)	(1,-1)

Cuadro 1.10: Matrices de pago en el juego de cara o cruz mixto.

Busquemos los equilibrios como en el ejemplo anterior:

- La estrategia óptima de J1 si J2 escoge Cara, es Cruz-Cara. En respuesta, la estrategia óptima de J2 a Cruz-Cara es Cruz si la primera moneda sale Cara, y es Cara si el primer lanzamiento es Cruz. Por lo tanto, no tenemos EB aquí.
- La estrategia óptima de J1 si J2 escoge Cruz, es Cara-Cruz. La estrategia óptima de J2 a Cara-Cruz es Cara si sale Cara la primera vez, y Cruz si sale Cruz. Aquí no tenemos EB tampoco.

Según el teorema que hemos visto anteriormente, podemos afirmar que en cualquier juego finito existe al menos un equilibrio de Nash. En el caso que nos ocupa, el equilibrio no puede ser en estrategias puras, ya que ninguna de ellas es una mejor respuesta a la estrategia del otro jugador. Por lo tanto, el equilibrio debe ser en estrategias mixtas, es decir, en las que cada jugador asigna una probabilidad a cada una de sus acciones posibles.

Comprobaremos que es EB el perfil $\mathbf{s^*} = (Cara - Cara, s_2^*)$ siendo s_2^* la estrategia mixta $(1/2, 1/2)$ del Jugador 2.Primero recordemos la notación; $u_i(Cara; Cara, Cruz)$ es el pago al Jugador 1 cuando el primer lanzamiento es Cara, J1 escoge Cara y J2 escoge Cruz. Vamos a buscar la estrategia óptima de J1 a la estrategia $(1/2, 1/2)$ de J2:

- Si en el primer lanzamiento sale Cara:

- J1 juega Cara:

$$\begin{aligned} U_1 &= (1/2) \cdot u_1(Cara; Cara, Cara) + (1/2) \cdot u_1(Cara; Cara, Cruz) \\ &= (1/2) \cdot (-1) + (1/2) \cdot (1) = 0 \end{aligned}$$

- J1 juega Cruz:

$$\begin{aligned} U_1 &= (1/2) \cdot u_1(Cara; Cruz, Cara) + (1/2) \cdot u_2(Cara; Cruz, Cruz) \\ &= (1/2) \cdot (-1) + (1/2) \cdot (1) = 0 \end{aligned}$$

Podemos observar que una respuesta óptima de J1 cuando sale Cara, es Cara. (También lo sería Cruz)

- Si en el primer lanzamiento sale Cruz:

- J1 juega Cara:

$$\begin{aligned} U_1 &= (1/2) \cdot u_1(Cruz; Cara, Cara) + (1/2) \cdot u_1(Cruz; Cara, Cruz) \\ &= (1/2) \cdot (1) + (1/2) \cdot (-1) = 0 \end{aligned}$$

- J1 juega Cruz:

$$\begin{aligned} U_1 &= (1/2) \cdot u_1(Cruz; Cruz, Cara) + (1/2) \cdot u_2(Cruz; Cruz, Cruz) \\ &= (1/2) \cdot (-1) + (1/2) \cdot (1) = 0 \end{aligned}$$

Vemos que en este caso una respuesta óptima de J1 es Cara.

Luego, una de las posibles respuestas óptimas de J1 a la estrategia $(1/2, 1/2)$ es Cara-Cara.

Toca realizar el paso inverso, ¿es $(1/2, 1/2)$ una estrategia óptima de la jugada Cara-Cara de J1?

- J2 juega Cara:

$$\begin{aligned} U_2 &= (1/2) \cdot u_2(Cara; Cara, Cara) + (1/2) \cdot u_2(Cruz; Cara, Cruz) \\ &= (1/2) \cdot (1) + (1/2) \cdot (-1) = 0 \end{aligned}$$

- J1 juega Cruz:

$$\begin{aligned} U_2 &= (1/2) \cdot u_2(Cara; Cara, Cruz) + (1/2) \cdot u_2(Cruz; Cara, Cruz) \\ &= (1/2) \cdot (-1) + (1/2) \cdot (1) = 0 \end{aligned}$$

Por lo tanto, el perfil (Cara-Cara, (1/2,1/2)) es un EB.

1.4 Juegos dinámicos

En este apartado nos centraremos en los juegos dinámicos, es decir, aquellos en los que las decisiones se toman de forma secuencial y los jugadores pueden observar las acciones previas de sus oponentes. Estos juegos se pueden representar mediante un árbol de juego, que muestra las alternativas posibles en cada nodo, las reglas que determinan quién mueve en cada momento y los pagos que reciben los jugadores al final del juego.

Como realizamos con los juegos estáticos, los juegos dinámicos se pueden dividir en dos tipos según el grado de información que tienen los jugadores sobre las características del juego: juegos con información completa y juegos con información incompleta. En los juegos con información completa, todos los jugadores conocen las preferencias, las estrategias y los pagos de los demás, así como la estructura del juego. En los juegos con información incompleta, alguno de estos elementos es desconocido o incierto para algún jugador.

Los juegos dinámicos con información completa se pueden resolver mediante el método de la inducción hacia atrás, que consiste en analizar el juego desde el final hacia el principio, buscando el equilibrio de Nash perfecto en subjuegos. Este concepto de equilibrio garantiza que las estrategias son óptimas y creíbles en cada etapa del juego.

Los juegos dinámicos con información incompleta requieren conceptos de equilibrio más refinados, que incorporen la incertidumbre y las creencias de los jugadores sobre los tipos o características de sus oponentes. Algunos de estos conceptos son el equilibrio bayesiano perfecto en subjuegos, el equilibrio secuencial y el equilibrio perfecto de mano temblorosa. Estos conceptos se basan en el principio de consistencia entre las creencias y las acciones de los jugadores a lo largo del juego.

Los juegos dinámicos tienen numerosas aplicaciones en economía y otras ciencias sociales, como el análisis de mercados oligopolísticos, la regulación económica, la negociación colectiva, la señalización en el mercado laboral o la teoría política. Estos ejemplos ilustran cómo la Teoría de Juegos puede ayudar a comprender y predecir el comportamiento estratégico de los agentes en situaciones complejas e interdependientes.

Es conveniente señalar aquí que los juegos estáticos analizados anteriormente no dejan de ser un caso especial de juegos dinámicos en los que las decisiones se toman simultáneamente, es decir, a la hora de escoger las jugadas ningún jugador conoce las elecciones del resto de jugadores.

Una de las características más importantes de los juegos dinámicos es el tipo y la cantidad de información que poseen los jugadores en cada momento del juego. No siempre los jugadores conocen todas las acciones que han sido realizadas por los que han jugado antes que ellos, lo que puede afectar a sus decisiones y a sus expectativas. Para representar gráficamente estas situaciones de información imperfecta, se utiliza el concepto de conjunto de información, que vamos a definir a continuación. Después veremos un ejemplo de cómo se aplican los conjuntos de información a un juego dinámico concreto.

Definición 13. *Llamaremos* ***conjuntos de información*** *a la información que tiene disponible un jugador al momento de tomar una decisión.*

Dicho de otra forma, un conjunto de información es una colección de nodos de un árbol

de juego que representan estados del juego que el jugador no puede distinguir entre sí, es una agrupación de nodos que son indistinguibles para un jugador dado.

Ejemplo 15. Dilema del prisionero repetido.

Estamos en el clásico problema del dilema del prisionero, pero en este caso, ambos jugadores juegan repetidamente. Los jugadores eligen una estrategia simultáneamente, como en el dilema del prisionero, y se revela la información sobre los pagos y las decisiones de cada uno, lo que origina una repetición. El pago final es la suma de todos los pagos. Veámoslo de forma extensiva para dos repeticiones:

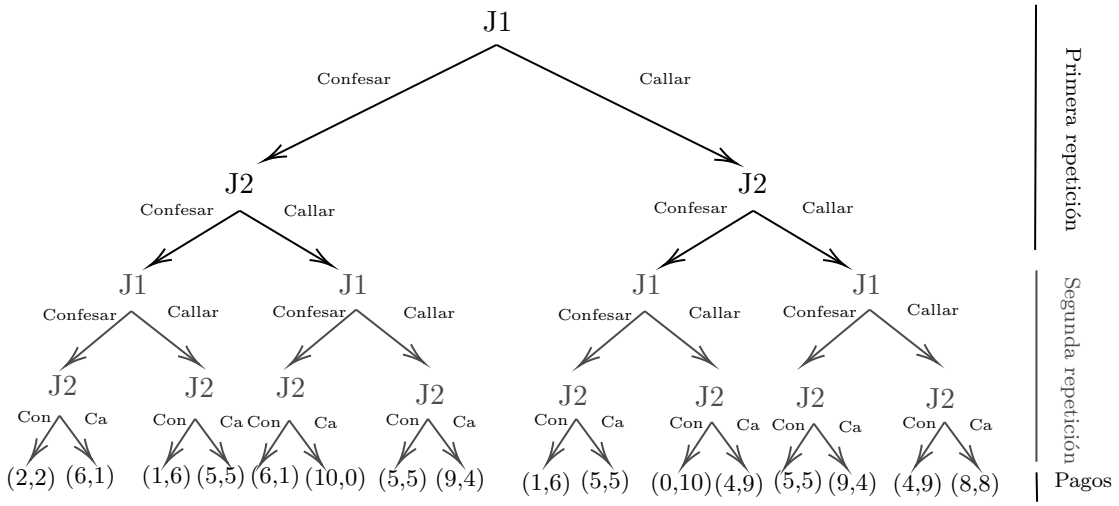


Figura 1.6: Ejemplo del dilema del prisionero repetido. Representación extensiva del juego dinámico.

En este juego contamos con 5 conjuntos de información para cada jugador. En la primera repetición, cada jugador tiene 1 conjunto de información. El conjunto de información para J1 es unitario, mientras que el del J2 es binario. En la segunda repetición, cada jugador tiene 4 conjuntos de información. Los conjuntos de información son unitarios y binarios respectivamente. En total tenemos 5 conjuntos de información para cada jugador.

En cada conjunto anterior tenemos 2 posibles jugadas: Callar y Confesar. Esto quiere decir que el juego tiene 2^5 estrategias puras.

En los juegos dinámicos, la información que tienen los jugadores sobre el juego puede variar según el contexto y la situación. Podemos distinguir entre dos tipos de información: perfecta e imperfecta. La información es perfecta cuando los jugadores conocen el orden de las decisiones, las acciones posibles y las realizadas por cada jugador y los pagos correspondientes a cada resultado. La información es imperfecta cuando los jugadores no conocen alguno de estos elementos o tienen incertidumbre sobre ellos.

Definición 14. *Dado un juego G de información completa es también de* ***información perfecta*** *si cada conjunto de información de cualquiera de sus jugadores es unitario. En cambio, si existe un jugador con algún conjunto de información no unitario, el juego será de* ***información imperfecta****.*

Un ejemplo claro de un juego de información perfecta es el ajedrez o las damas. Por otro lado, los juegos de información imperfecta, en los que los jugadores no tienen toda

la información necesaria para tomar una decisión, pueden ser los juegos de cartas, ya que los jugadores desconocen las cartas del resto de los jugadores. Otros ejemplos de juegos de información imperfecta son los juegos de azar, donde el jugador no sabe en qué nodo se encuentra a la hora de tomar una decisión.

En las secciones anteriores hemos analizado los juegos con información completa, tanto en su forma estática como en su forma dinámica. Ahora nos interesa explorar la relación que existe entre estos dos tipos de juegos. Una forma de abordar esta cuestión es considerar a los juegos estáticos como un caso particular de los juegos dinámicos, donde el jugador que debe elegir una acción no tiene información sobre las acciones previas de los otros jugadores. De esta manera, podemos interpretar este conjunto de acciones como si se hubieran tomado de forma simultánea por todos los jugadores.

Ya hemos visto en el Ejemplo 1 cómo podemos representar el mismo juego tanto en forma matricial como en forma extensiva. La forma matricial nos muestra las posibles combinaciones de acciones y pagos para cada jugador, mientras que la forma extensiva nos muestra el orden secuencial de las decisiones. Ahora podemos interpretar la forma extensiva como un juego dinámico en el que el segundo jugador (Diana en el ejemplo) no tiene información sobre la acción elegida por el primer jugador (Javier en el ejemplo).

Una primera idea que podríamos tener es aplicar el concepto de equilibrio de Nash que hemos definido para los juegos estáticos a esta nueva clase de juegos dinámicos. Sin embargo, veremos que este concepto no es adecuado para capturar las características esenciales de estos juegos, y que puede conducir a resultados poco razonables o inconsistentes. Por lo tanto, tendremos que modificar el concepto de equilibrio de Nash para adaptarlo a los juegos dinámicos, y buscar una solución más satisfactoria para estos problemas.

Empecemos por analizar el juego representado en forma extensiva. Una forma de hacerlo es convertir cada posible secuencia de decisiones en un juego en forma normal, donde los jugadores eligen simultáneamente las acciones que van a realizar. Si hacemos esto para todas las secuencias posibles, obtenemos varios juegos (que denominaremos subjuegos) donde el pago final de cada jugador es la suma de todos los pagos obtenidos en cada etapa. De esta manera, podemos aplicar el mismo método que usamos para los juegos estáticos. Un caso ilustrativo es el que hemos visto antes: el juego del dilema del prisionero repetido. Cada vez que se repite el juego, se genera una matriz de pagos distinta, que podemos estudiar de forma independiente.

Sin embargo, este método tiene una limitación importante: ignora la información dinámica del juego. Es decir, si halláramos todos los equilibrios de Nash en cada subjuego, podríamos pensar que esta solución es válida para el juego estático, pero no tendría sentido para el juego dinámico, donde las decisiones se toman de forma secuencial y dependen de las acciones anteriores.

Antes de seguir, conviene aclarar lo que entendemos por un subjuego de un juego en forma extensiva:

Definición 15. *Dado un juego G con información completa en forma extensiva, dado un nodo de decisión x de G, es decir, un punto donde un jugador tiene que elegir una acción entre varias posibles, diremos que G' es un* ***subjuego*** *de G con inicio en x si G' es una parte de G que cumple que:*

1. *Contiene al nodo x y a todos los nodos que le siguen y únicamente a éstos.*
2. *El nodo es conjunto de información unitario.*
3. *No rompe ningún conjunto de información. Si y pertenece a G', pertenecerán a G' todos los nodos del conjunto de información de y.*

Cabe destacar algunas propiedades inmdiatas:

Observación 1:

- Podemos escoger cualquier nodo para crear un subjuego, simplemente considerando este nodo como conjunto de información unitario.
- Un juego de información perfecta únicamente contiene conjuntos de información unitarios por lo que todos los nodos de un subjuego dan lugar a un subjuego.
- En los juegos estáticos representados en forma extensiva, no hay ningún subjuego diferente del juego original, porque el único nodo de decisión que existe es el inicial, que forma un conjunto de información unitario.

Ejemplo 16. Subjuegos en el dilema del prisionero repetido.

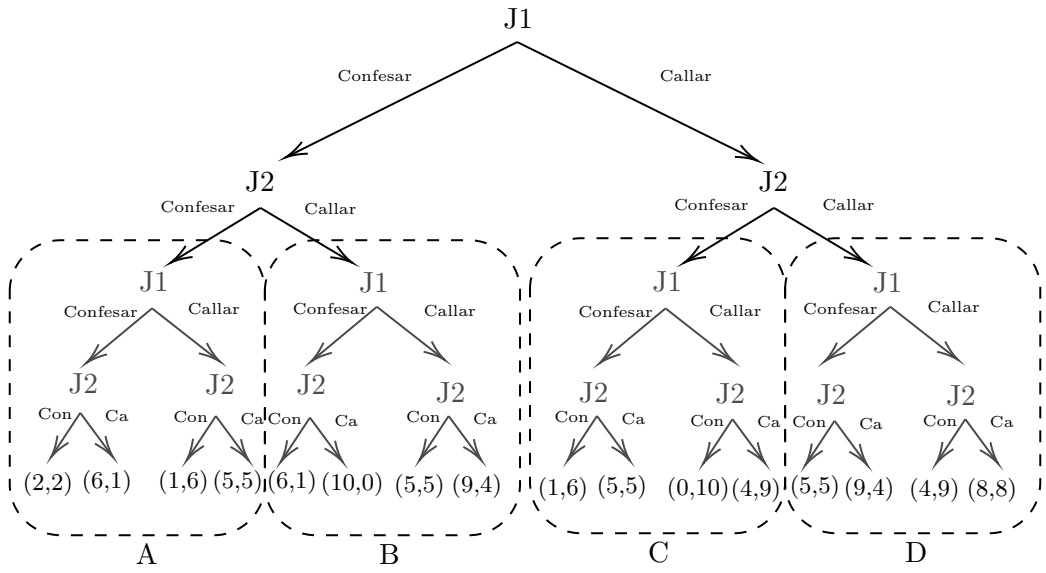


Figura 1.7: Subjuegos en el dilema del prisionero repetido.

Después de haber introducido este concepto, vamos a explorar otra manera de caracterizar la noción de equilibrio que hemos desarrollado: el equilibrio de Nash perfecto en subjuegos.[3]

Recordemos que el equilibrio de Nash es un concepto de solución que se basa en la idea de que cada jugador elige la mejor respuesta a las estrategias de los demás jugadores. Sin

[3]Reinhard Selten fue un economista alemán que propuso en 1965 el concepto de equilibrio perfecto en subjuegos, un refinamiento del equilibrio de Nash para juegos dinámicos. Por sus aportaciones a la Teoría de Juegos, fue galardonado con el Premio Nobel de Economía en 1994, junto con John F. Nash y John C. Harsanyi.

embargo, el equilibrio de Nash no tiene en cuenta la estructura temporal del juego ni la posibilidad de que los jugadores hagan amenazas o promesas no creíbles.

Definición 16. *Sea G un juego en forma extensiva y $\mathbf{s}^*$ un perfil de estrategias de G que es un equilibrio de Nash, diremos que $\mathbf{s}$ es un* ***equilibrio de Nash perfecto en subjuegos*** *de G si la restricción de $\mathbf{s}^*$ a cualquier subjuego de G es un equilibrio de Nash en ese subjuego.*

Después de haber definido el nuevo concepto de equilibrios de Nash, nos interesa saber si estos equilibrios existen en los juegos que queremos analizar. Para responder a esta cuestión, presentaremos el siguiente teorema, que nos garantiza la existencia de estos equilibrios bajo ciertas condiciones.

Teorema 5. *Si G es un juego dinámico finito (todos los componentes son finitos), existe un equilibrio de Nash perfecto en subjuegos de G.*

En los juegos dinámicos, al igual que ocurre en los juegos estáticos, puede darse el caso de que no exista ningún equilibrio en estrategias puras, es decir, que no haya una combinación de acciones que sea óptima para todos los jugadores. En esa situación, se hace necesario buscar el equilibrio en estrategias mixtas.

El equilibrio de Nash perfecto en subjuegos es un concepto de solución que refina el equilibrio de Nash en juegos dinámicos, exigiendo que cada jugador actúe de forma óptima no solo al inicio del juego, sino también en cada posible punto de decisión a lo largo del juego. Sin embargo, este concepto no nos dice cómo encontrar dicho equilibrio, sino que solo nos permite descartar aquellos que no lo son.

Para hallar el equilibrio de Nash perfecto en subjuegos se utiliza un método llamado inducción hacia atrás, que consiste en analizar el juego desde el final hacia el principio, resolviendo primero los últimos movimientos y luego los anteriores, tal y como hemos realizado en el ejemplo 18. De esta forma, se determina la mejor respuesta de cada jugador en cada etapa del juego, teniendo en cuenta las expectativas racionales sobre las acciones futuras de los demás jugadores.

Definición 17. *Dado un juego G finito en forma extensiva con información completa y perfecta, llamamos* ***algoritmo de inducción hacia atrás*** *(también llamado de inducción retroactiva, algoritmo de Zermelo [4] o algortimo de Kuhn [5]) al siguiente algoritmo:*

1. *Se identifican los subjuegos que estén en último lugar. A continuación, se elimina cada subjuego salvo su nodo de comienzo, este último nodo pasará a ser el nodo final del juego global y se le atribuyen los pagos de haberse jugado la acción óptima correspondiente a este nodo. El grafo quedará recortado.*
2. *Se repite el paso 1 con el grafo resultante.*
3. *Se termina el algoritmo cuando lleguemos al nodo principal del juego global.*

[4] Matemático alemán. El algoritmo recibe su nombre por ser el primero en utilizarlo para encontrar la estrategia ganadora en juegos de suma cero.

[5] Matemático estadounidense también conocido por su colaboración con John Nash y su participación en el desarrollo del algoritmo húngaro para el problema de asignación.

Observación 2:

- Es importante el hecho de que el juego sea de información perfecta, es decir, que todos los conjuntos de información sean unitarios. Si en la forma extensiva del juego algún conjunto de información es no unitario, el algoritmo podría ser no aplicable. Esto es debido a que puede ocurrir que en algún punto sea imposible determinar cuáles son las acciones óptimas del jugador al que le toca jugar, pues puede que el jugador no sepa que acción óptima debe elegir al desconocer en que nodo se encuentra.

- Aunque el algoritmo teóricamente sea siempre aplicable, existen juegos en los que es actualmente imposible de ejecutar en la práctica. Algunos ejemplos son los juegos como las damas o el ajedrez (que tiene un número de Shannon[6] de 10^{120}). Como ambos juegos son de suma cero, que el algoritmo sea aplicable es fundamental en la búsqueda de buenas estrategias de juego, de éste se deduce que ha de ser cierta únicamente una de las siguientes afirmaciones: existe una estrategia ganadora para las piezas blancas, existe una estrategia ganadora para las piezas negras o existe una estrategia que garantiza las tablas. Se desconoce cuál de las tres afirmaciones es cierta.

¿Cuál es la relación entre este algoritmo y los equilibrios de Nash perfectos en subjuegos? El siguiente teorema de Zermelo-Kuhn muestra esta relación, además de garantizar la existencia de equilibrios para estrategias puras.

Teorema 6. *Cada juego finito con información completa y perfecta G tiene un equilibrio de Nash perfecto en subjuegos en estrategias puras que se obtiene por el método de inducción hacia atrás. Además, si ningún jugador tiene más de una acción óptima en cada nodo de decisión, tal equilibrio es único.*

Demostración. Veamos que en juegos finitos con información completa y perfecta el procedimiento de inducción hacia atrás está bien definido. El jugador que mueve en cada nodo de decisión tiene un número finito de elecciones posibles, por lo que en cada etapa del procedimiento tiene alguna acción óptima. Si un jugador es indiferente entre varias acciones, podemos escoger cualquiera.

Sea $\mathbf{s} = (s_1, ..., s_n)$ un perfil de estrategias puras obtenido por este procedimiento. Veamos, por reducción al absurdo, que se trata de un equilibrio de Nash del juego G. En efecto, si no fuera equilibrio de Nash, existiría un jugador i para el cual existe una estrategia s_i^* que verifica

$$u_i(s_i^*, \mathbf{s_{-i}}) > u_i(s_i, \mathbf{s_{-i}})$$

donde $\mathbf{s_{-i}} = (s_1, ..., s_{i-1}, s_{i+1}, ..., s_n)$.

Por inducción llegaremos a una contradicción. Diremos que el nodo de decisión x del juego G tiene una distancia m si, entre las diferentes trayectorias que le siguen hasta los nodos terminales, el máximo número de nodos de decisión que hay entre x y un nodo terminal es m. Sea M la distancia máxima posible de los nodos de decisión del juego. Como G es un juego finito, se verifica que M es un número natural finito.

Sea $s_i^*(m)$ la siguiente estrategia del jugador i:

[6] El número de Shannon es una estimación del límite inferior de la complejidad del árbol de juego del ajedrez, esto es el número real de posibles juegos, que podría ser incluso mayor. Cabe destacar que el número de átomos en el universo observable es de 10^{82}.

- $s_i^*(m)$ coincide con s_i en todos los nodos con distancia 0,1,...,m.
- $s_i^*(m)$ coincide con s_i^* en todos los nodos con distania mayor que m.

Por construcción de $\mathbf{s}$ a través del procedimiento de inducción hacia atrás, se verifica que $u_i(s_i^*(0), \mathbf{s_{-i}}) \geq u_i(s_i^*, \mathbf{s_{-i}})$, ya que el jugador i seguro que no empeora (y puede mejorar) con respecto a su estrategia s_i^* si en sus nodos de decisión finales (por tanto, con distancia cero) elige su acción óptima, manteniendo la estrategia s_i^* en los demás nodos de decisión.

Supongamos que

$$u_i(s_i^*(m-1), \mathbf{s_{-i}}) \geq u_i(s_i^*, \mathbf{s_{-i}}), \text{ (hipótesis de inducción)}$$

Veamos que esto implica que

$$u_i(s_i^*(m), \mathbf{s_{-i}}) \geq u_i(s_i^*, \mathbf{s_{-i}})$$

La única diferencia entre las estrategias $s_i^*(m-1)$ y $s_i^*(m)$ del jugador i está en las elecciones del jugador i en los nodos con distancia m. En ambas estrategias, el jugador i juega de acuerdo con la estrategia s_i en todos los nodos de decisión que siguen a los nodos con distancia m y de acuerdo con la estrategia s_i^* en todos los anteriores a ellos. Pero, puesto que todos los jugadores están jugando en concordancia con el perfil de estrategias $\mathbf{s}$ después de los nodos con distancia m, los movimientos que el procedimiento de inducción hacia atrás determina para los nodos con distancia m (aquellos que forman parte de s_i) deben ser elecciones óptimas para el jugador i en dichos nodos. Por tanto, tiene que ser

$$u_i(s_i^*(m), \mathbf{s_{-i}}) \geq u_i(s_i^*(m-1), \mathbf{s_{-i}})$$

lo que implica que

$$u_i(s_i^*(m), \mathbf{s_{-i}}) \geq u_i(s_i^*, \mathbf{s_{-i}})$$

Llegamos entonces a que

$$u_i(s_i^*(M), \mathbf{s_{-i}}) \geq u_i(s_i^*, \mathbf{s_{-i}})$$

Pero $s_i^*(M) = s_i$ y se llega a una contradicción con la primera desigualdad de la prueba ($u_i(s_i^*, \mathbf{s_{-i}}) > u_i(s_i, \mathbf{s_{-i}})$). Por tanto, $\mathbf{s}$ es un equilibrio de Nash. De igual manera que el perfil de estrategias $\mathbf{s}$ constituye un equilibrio de Nash del juego G, su restricción a cualquier subjuego es asimismo un equilibrio de Nash en el subjuego. Por último, si ningún jugador tiene más de una acción óptima en cada etapa del proceso el equilibrio es único. □

Debemos señalar que el algoritmo de inducción hacia atrás no se puede aplicar al dilema del prisionero repetido, ya que este juego tiene información imperfecta. Vamos a proponer otro juego sencillo para estudiar cómo se utilizaría este algoritmo.

Ejemplo 17. Juego del trespiés.

Inicialmente hay dos montones de monedas sobre una mesa, inicialmente cada uno tiene una moneda. Dos jugadores J1 y J2 tomarán las decisiones de forma alternada, estas decisiones pueden ser: seguir (S) o terminar (T). Comienza J1, si elige T se acaba el juego, si elige S el primer montón pasa a tener 0 monedas y 3 el segundo. Ahora J2 si elige T se

termina el juego, si juega S ambos montones pasan a tener dos monedas. Por último, J1 toma una decisión, si escoge T se termina el juego, si escoge S el primer montón pasa a tener una moneda, mientras que el segundo tendrá cuatro. El primer montón de monedas al finalizar el juego es el pago de J1, el otro montón será el pago de J2.

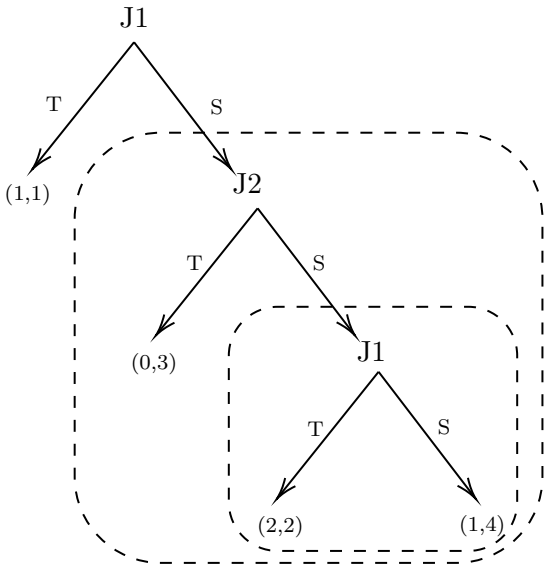


Figura 1.8: Subjuegos del juego del trespiés.

Realicemos el algoritmo.

1. Vamos al último subjuego. Aquí J1 decidirá T, obteniendo el pago (2,2). Sustituimos en el nodo este pago.

2. Ahora tenemos el árbol recortado. Es el turno de J2, que eligirá T, para obtener el pago de (0,3). Sustituimos el pago.

3. Nos queda una última decisión para J1, que eligirá T, para obtener un pago mayor.

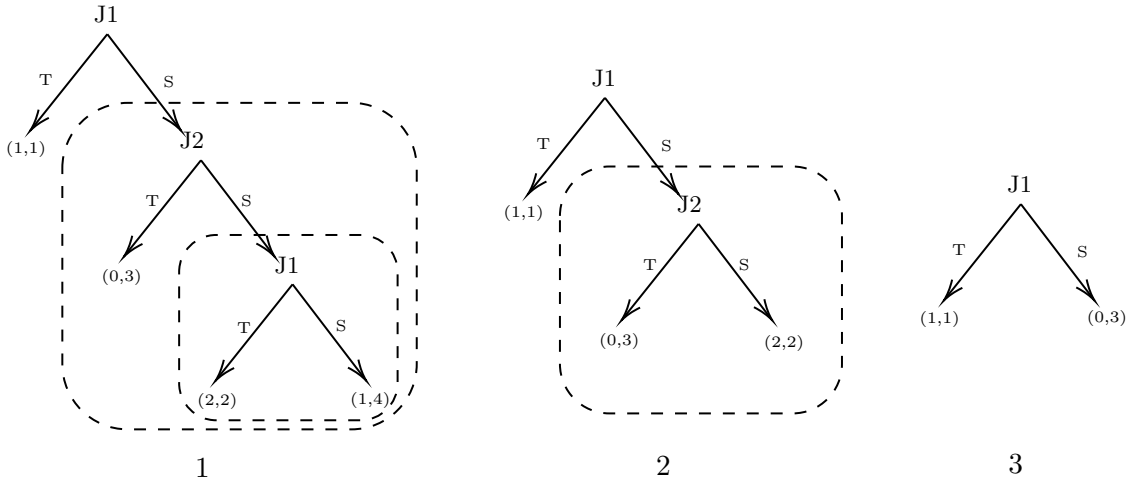


Figura 1.9: Pasos del algoritmo de inducción hacia atrás en el juego del trespiés.

Por tanto, el único equilibrio es (T-T,T) y el pago será $(1, 1)$.

Sin embargo, ¿existe alguna forma de resolver el dilema del prisionero repetido? Recordemos que en el ejemplo 16 se presentaron los subjuegos del dilema del prisionero repetido, ¿habrá algún algoritmo que nos permita hallar las soluciones? La respuesta es afirmativa. Existe una extensión del algoritmo anterior que nos ayuda a encontrar los equilibrios de Nash perfectos en subjuegos con conjuntos de información mayores que uno, es decir, cuando hay subjuegos en los que al menos un jugador tiene más de un nodo de decisión.

Antes de volver con el ejemplo, veamos la definición del algoritmo.

Definición 18. *Dado un juego G finito en forma extensiva con información completa (no necesariamente perfecta), llamamos* ***algoritmo de inducción hacia atrás generalizado*** *al siguiente algoritmo:*

1. *Se identifican los subjuegos que estén en último lugar. Se calculan los equilibrios de Nash.*

2. *Si existe un único equilibrio en estrategias puras, se elimina cada uno de los subjuegos y se atribuyen los pagos a este subjuego.*

3. *Se termina el algoritmo cuando lleguemos al nodo principal del juego global.*

Al finalizar el algoritmo, habremos asignado unos pagos al nodo inicial del juego, y habremos marcado unas ramas del árbol como parte de los equilibrios de Nash de cada subjuego. Estas ramas indican las acciones óptimas que deben tomar los jugadores en cada etapa del juego. Por lo tanto, el único resultado perfecto en subjuegos es el que se obtiene siguiendo estas ramas desde el nodo inicial hasta un nodo terminal. Los pagos asociados al nodo inicial son los que corresponden a este resultado. Asimismo, el único equilibrio de Nash perfecto en subjuegos es el que se forma con las estrategias que consisten en elegir la acción marcada en cada conjunto de información de cada jugador.

Observación 3:

- Si existieran más de un equilibrio en algún subjuego, habría que considerar todas las posibilidades. Se formarán tantos árboles como equilibrios haya en esa etapa para continuar con el algoritmo.

- También podemos considerar estrategias mixtas, que consisten en asignar probabilidades a las acciones puras. En ese caso, debemos calcular los pagos esperados que obtendremos al usar estas estrategias mixtas.

Ejemplo 18. Equilibrio perfecto de Nash en el dilema del prisionero repetido.

Calculemos el equilibrio de Nash de cada subjuego anterior. De izquierda a derecha tenemos:

A		J2	
		Con	Ca
J1	Con	(2,2)	(6,1)
	Ca	(1,6)	(5,5)

B		J2	
		Con	Ca
J1	Con	(6,1)	(10,0)
	Ca	(5,5)	(9,4)

C		J2	
		Con	Ca
J1	Con	(1,6)	(5,5)
	Ca	(0,10)	(4,9)

D		J2	
		Con	Ca
J1	Con	(5,5)	(9,4)
	Ja	(4,9)	(8,8)

Cuadro 1.11: Subjuegos del dilema del prisionero repetido en forma normal.

Los equilibrios son:

- Subjuego A: Confesar-Confesar.
- Subjuego B: Confesar-Confesar.
- Subjuego C: Confesar-Confesar.
- Subjuego D: Confesar-Confesar.

Este resultado es lógico, pues cada subjuego no deja de ser un juego del dilema del prisionero, donde el equilibrio de Nash es que ambos confiesen. Por tanto, podemos generar una tabla con los pagos obtenidos.

		J2	
		Con	Ca
J1	Con	(2,2)	(6,1)
	Ca	(1,6)	(5,5)

Cuadro 1.12: Tabla de resultados del algoritmo hacia atrás en el dilema del prisionero repetido.

Concluimos que la solución de este juego es que ambos jugadores siempre confiesen (como en el caso del dilema del prisionero sin repetición), pero con el vector de pagos $(2, 2)$. Podemos demostrar, que con la matriz de pagos del dilema del prisionero, si repetirmos n veces, el equilibrio de Nash seguirá siendo la opción en la que ambos confiesan con el vector de pagos (n, n).

Hasta ahora se han presentado los conceptos básicos de la Teoría de Juegos y hemos estudiado los juegos estáticos y dinámicos con información completa e incompleta. Hemos visto cómo se pueden representar los juegos en forma extensiva o matricial y cómo se pueden encontrar los equilibrios de Nash en estrategias puras o mixtas. También hemos analizado algunos refinamientos del equilibrio de Nash para juegos en forma normal o extensiva, así como el concepto de equilibrio bayesiano.

La Teoría de Juegos tiene una amplia gama de aplicaciones en diferentes campos del conocimiento, como la economía, la biología, la sociología o la ciencia política. En la siguiente sección veremos algunas de estas aplicaciones y cómo se pueden modelar y resolver problemas de interés práctico usando las herramientas de la Teoría de Juegos. Algunos ejemplos de aplicaciones que veremos son: el oligopolio de Cournot y Bertrand. Estas aplicaciones nos permitirán ilustrar la utilidad y el alcance de la Teoría de Juegos para comprender y predecir el comportamiento estratégico de agentes racionales en situaciones interactivas.

1.5 Aplicaciones de la Teoría de Juegos

La Teoría de Juegos es una rama de la matemática aplicada que estudia las situaciones de conflicto y cooperación entre agentes racionales que interactúan entre sí, buscando maximizar su utilidad. La Teoría de Juegos ha sido ampliamente utilizada en el campo de la economía, la biología, la sociología y la ciencia política, entre otras disciplinas, para modelar y analizar fenómenos como el oligopolio, el dilema del prisionero, la evolución de las especies o la negociación.

La Teoría de Juegos se basa en la representación formal de los elementos esenciales de un juego: los jugadores, las acciones o estrategias disponibles para cada jugador, los resultados o consecuencias de cada combinación de acciones o estrategias, y los pagos o utilidades que cada jugador obtiene de cada resultado. A partir de esta representación, se pueden definir y estudiar diferentes conceptos de solución o equilibrio para un juego, que indican qué acciones o estrategias son óptimas o racionales para cada jugador en función de sus preferencias y expectativas.

La Teoría de Juegos se puede clasificar en dos grandes enfoques: cooperativo y no cooperativo. En el enfoque cooperativo se analizan las posibilidades de que algunos o todos los jugadores lleguen a un acuerdo sobre qué decisiones va a tomar cada uno, mientras que en el enfoque no cooperativo se analiza qué decisiones tomaría cada jugador en ausencia de acuerdo previo. Dentro del enfoque no cooperativo, se pueden distinguir también dos tipos básicos de juegos: estáticos o dinámicos, y con o sin información completa. En los juegos estáticos los jugadores toman sus decisiones simultáneamente (o sin saber qué han decidido los otros), mientras que en los dinámicos puede darse el caso de que un jugador conozca ya las decisiones de otro antes de decidir. En los juegos con información completa, todos los jugadores conocen las consecuencias, para sí mismos y para los demás, del conjunto de decisiones tomadas, mientras que en los juegos con información incompleta, algún jugador desconoce alguna de esas consecuencias.

Esta teoría tiene numerosas aplicaciones en diversos campos del conocimiento, como la economía, la biología, la informática, la sociología, la ciencia política y la filosofía. En este apartado se presentan algunos ejemplos de cómo la Teoría de Juegos puede ayudar a

comprender y resolver problemas reales en estos ámbitos.

La Teoría de Juegos también se ha utilizado para analizar las implicaciones normativas de los distintos sistemas de elección social, es decir, los mecanismos que permiten agregar las preferencias individuales para llegar a una decisión colectiva. Sen [7] presenta un teorema de imposibilidad que prueba que no existe ninguna función de bienestar social que respete el criterio de Pareto y el principio de libertad mínima [8]. El autor define una situación social como una asignación de opciones a cada individuo en cada aspecto relevante de su vida, y una función de bienestar social como una relación de orden sobre el conjunto de situaciones sociales. El criterio de Pareto requiere que la función de bienestar social sea transitiva y completa, y que refleje las preferencias individuales sobre las opciones asignadas. El principio de libertad mínima requiere que la función de bienestar social respete las preferencias individuales sobre las opciones disponibles. Se construye un ejemplo con tres individuos y dos aspectos binarios, y muestra que no hay ninguna función de bienestar social que cumpla ambas condiciones. El criterio de Pareto afirma que en muchas situaciones, el 80 % de los efectos se deben al 20 % de las causas es insuficiente para evaluar las situaciones sociales, y sugiere que se deben considerar otros criterios normativos. Este criterio se puede utilizar para identificar y priorizar las acciones más importantes o influyentes en un proceso o un objetivo. Se basa en las observaciones del economista italiano Vilfredo Pareto, que en el siglo XIX descubrió que el 80 % de las tierras en Italia pertenecían al 20 % de la población.

Otro ejemplo de una situación en la que el criterio de Pareto no es suficiente para garantizar la eficiencia social es el oligopolio de Bertrand. Este es un modelo de competencia imperfecta en el que dos o más empresas producen un bien homogéneo y fijan sus precios de forma simultánea, teniendo en cuenta las reacciones de sus rivales. El oligopolio de Bertrand predice que, en equilibrio, las empresas cobrarán el precio igual al coste marginal, lo que implica que no habrá beneficios ni pérdidas. Sin embargo, este resultado no es necesariamente óptimo desde el punto de vista social, ya que puede haber otras formas de asignar los recursos que mejoren el bienestar de algunos agentes sin empeorar el de otros. Por ejemplo, si las empresas cooperan y fijan un precio más alto que el coste marginal, podrían obtener beneficios positivos y repartirlos entre ellas o entre los consumidores mediante subsidios o transferencias. Esto aumentaría la utilidad de las empresas y/o de los consumidores sin reducir la de nadie más. Por tanto, el equilibrio de Bertrand no cumple con el principio de Pareto.

El oligopolio de Bertrand es un modelo que fue propuesto por primera vez por el matemático francés Joseph Louis François Bertrand en 1831, como una crítica al modelo de Cournot, que suponía que las empresas competían en cantidades en lugar de en precios. Veámos el caso discreto en profundidad.

Supongamos que en un mercado hay sólo dos empresas E_1 y E_2, que fabrican un mismo producto y compiten en precios. Supongamos que

- Los consumidores compran a la empresa con menor precio, si tienen el mismo, los consumidores se reparten en partes iguales.

[7] Amartya Sen, economista indio profesor de Harvard, Cambridge y Oxford. Premio Nobel de Economía en 1998.

[8] The impossibility of a Paretian liberal. Journal of Political Economy, 78(1), 152-157. https://doi.org/10.1086/259614

- La función $q_i(p)$ demanda es estrictamente decreciente para precios p entre 0 y un precio fijo $p_C > 0$. Esta función será nula para $p \geq p_C$.
- Ambas empresas tienen la misma función de costes, con costes marginales constantes iguales a c.
- Se cumple que $0 < c < p_M < p_C$ donde p_M es el precio óptimo que maximizaría el beneficio de una cualquiera de las empresas si la otra se retirase.

Como estamos considerando el espacio discreto, tendremos el espacio de estrategias $A = 0, p_m, 2p_m, 3p_m, ..., kp_m, ...$ donde p_m es la unidad mínima monetaria (1 céntimo de euro por ejemplo), por lo que se cumplirá que $c, p_M, p_C \in A$ y $0 < p_m < c$.

Resumiendo, las empresas E_1 y E_2 con espacios de estrategias $S_1 = S_2 = A$ tienen funciones de ganancias:

$$u_i(p_i, p_j) = \begin{cases} 0 & si \quad p_i > p_j \\ (p_i - c)q(p_i) & si \quad p_i < p_j \\ (p_i - c)q(p_i)/2 & si \quad p_i = p_j \end{cases} \forall p_i, p_j \in A$$

Demostremos que hay dos equilibrios de Nash. El primero será cuando ambas empresas deciden un precio igual al coste marginal común: ($p_1^* = c$, $p_2^* = c$). El segundo será cuando ambas empresas deciden un precio igual al coste marginal común sumando la unidad monetaria mínima: ($p_1^* = c + p_m$, $p_2^* = c + p_m$). Consideraremos precios p tal que $c < p < p_C$.

Tenemos que estudiar los siguientes casos:

1. El precio fijado por cada empresa es diferente. Una lo fija en el coste marginal y la otra lo fija al menos dos unidades monetarias mínimas por encima.
2. El precio fijado por cada empresa es diferente. Una lo fija en el coste marginal y la otra en una unidad monetaria mínima por encima.
3. El precio fijado por cada empresa es diferente. Ambas son superiores al coste marginal.
4. Los precios son iguales. Son superiores al coste marginal en al menos dos unidades monetarias mínimas.
5. Los precios son iguales. Son superiores al coste marginal en una unidad monetaria mínima.

Vamos a estudiarlos.

1. $p_j^* \geq p_i^* + 2p_m$ y $p_i^* = c$. La empresa E_i obtiene una ganancia nula, que aumentaría si eligiera un precio $p_i' = c + p_m$, ya que

$$u_i(p_i', p_j^*) = q(p_i')(p_i' - c) = q(p_i')(p_m) > 0$$

Por lo tanto, no es un equilibrio de Nash.

2. $p_j^* = p_i^* + p_m$ y $p_i^* = c$. La empresa E_i obtiene ganancia nula, que aumentaría si elige un precio $p_i' = c + p_m$, ya que

$$u_j(p_i', p_j^*) = q(p_i')(p_i' - c)/2 = q(p_i')(p_m)/2 > 0$$

   Por lo tanto, no es un equilibrio de Nash.

3. $p_j^* > p_i^* > c$. En esta situación la empresa E_j obtiene una ganancia nula, que aumentaría si eligiera un precio $p_j' = p_i^*$, igual que al de su rival, ya que

$$u_i(p_i^*, p_j') = q(p_j')(p_j' - c)/2 > 0$$

   Por lo tanto, no es un equilibrio de Nash.

4. $p_j^* = p_i^* > c + 2p_m$. En esta situación la empresa E_i obtiene una ganancia positiva ($u_i(p_i^*, p_i^*) = q(p_i^*)(p_j^* - c)/2 \geq q(p_i^*)(2P_m)/2 > 0$), que aumentaría si eligiera un precio $p_i' = p_i^* - p_m$, ya que

$$u_i(p_i', p_i^*) = q(p_i')(p_i' - c) = q(p_i^* - p_m)(p_i^* - p_m - c)$$

   es mayor que

$$u_i(p_i^*, p_i^*) = q(p_i^*)(p_i^* - c)/2$$

   Por tanto, no es un equilibrio de Nash.

5. $p_j^* = p_i^* = c + p_m$. En esta situación ambas empresas obtienen ganancias positivas

$$u_i(p_i^*, p_i^*) = u_j(p_i^*, p_i^*) = q(p_i^*)(p_i^* - c)/2 = q(c + p_m)(p_m)/2 > 0$$

   ninguna de las empresas conseguiría aumentar su ganancia si modificase el precio, ya que si se aumentara perdería toda la demanda y tendría ganancia nula, si se disminuyera vendería a un precio igual a su coste marginal, teniendo ganancia nula. Tenemos un equilibrio de Nash.

6. $p_j^* = p_i^* = c$. En esta situación ambas empresas obtienen ganancias nulas, pero ninguna podría aumentarlas si modificara el precio. Si una empresa aumentara el precio perdería toda la demanda, y si disminuyera el precio la ganancia sería negativa. Tenemos otro equilibrio de Nash.

Este juego tiene distintas variantes, como el caso continuo, utilizando productos diferenciados o añadiendo más empresas.

Capítulo 2

Inteligencia artificial

La Inteligencia Artificial (IA) es un campo de estudio y desarrollo de tecnología que se centra en la creación de sistemas y programas capaces de realizar tareas similares a la inteligencia humana. A través de la simulación de procesos cognitivos, la IA busca construir máquinas y sistemas informáticos que puedan aprender, razonar, tomar decisiones y resolver problemas de manera autónoma.

La IA abarca una amplia gama de enfoques y técnicas. Uno de los conceptos clave en la IA es el Aprendizaje Automático (Machine Learning), que permite a las máquinas aprender de los datos sin ser programadas explícitamente. A través de algoritmos y modelos matemáticos, el aprendizaje automático permite a los sistemas de IA reconocer patrones y tendencias en los datos, y utilizar esa información para tomar decisiones o realizar tareas específicas.

En el campo de la IA, los algoritmos de búsqueda juegan un papel fundamental. Estos algoritmos se utilizan para encontrar soluciones en problemas definidos mediante un espacio de búsqueda. Por ejemplo, los algoritmos de búsqueda se aplican en la planificación de rutas, en la resolución de problemas de optimización y en la toma de decisiones en entornos complejos. Ejemplos comunes de algoritmos de búsqueda, que después comentaremos, son la búsqueda en profundidad (DFS), la búsqueda en amplitud (BFS) y el algoritmo A*.

Otros ejemplos de aplicaciones de Inteligencia Artificial pueden ser el Procesamiento del Lenguaje Natural (Natural Language Processing, NLP), que se centra en la interacción entre las computadoras y el lenguaje humano; la Visión por Computadora, donde las máquinas pueden analizar y entender imágenes y videos; la Robótica, que busca crear robots inteligentes y autónomos capaces de interactuar con el entorno de manera adaptativa y las Redes Neuronales Artificiales, donde capas de nodos interconectados (neuronas) procesan y transforman la información para realizar tareas de reconocimiento de patrones y toma de decisiones.

2.1 Algoritmos de búsqueda

La inteligencia humana nos permite resolver problemas que, aunque nos parezcan fáciles a simple vista, implican una gran complejidad en su planteamiento y solución. Por ejemplo, cuando queremos cruzar la calle, tenemos que considerar muchos factores y tomar decisiones rápidas y acertadas. Algunas de las cuestiones que nos planteamos son: ¿Cuál es el destino al que quiero llegar? ¿Existe algún semáforo que regule el tráfico? ¿Qué color

muestra el semáforo? ¿Tengo preferencia para cruzar o tengo que esperar? ¿Qué tiempo disponible tengo para cruzar sin riesgos? ¿Desde qué dirección se acercan los vehículos? ¿A qué velocidad circulan? etc. Estas preguntas no son triviales, sino que requieren de nuestra experiencia previa y de nuestra capacidad de razonar y adaptarnos a las circunstancias para darles una respuesta adecuada y lograr nuestro objetivo.

Estos problemas se pueden abordar mediante la búsqueda de la solución óptima en un espacio de búsqueda que representa todas las posibles soluciones. La complejidad del problema determina el tamaño y la estructura de este espacio de búsqueda. Por ejemplo, si tenemos tres cajas de tres colores distintos y tres bolas del mismo color que las cajas, el espacio de búsqueda para introducir cada bola en la caja que le corresponde es mucho más simple que el espacio de búsqueda para el juego del ajedrez, donde hay muchas más posibilidades y reglas que seguir.

Estos problemas se pueden clasificar como problemas de búsqueda, en los que el espacio de búsqueda se compone de un árbol de búsqueda. Este árbol de búsqueda se representa a través de un grafo que facilita la búsqueda de la solución óptima al problema.

Definamos los conceptos más formalmente[1].

Definición 19. *Un **problema de búsqueda** consiste en encontrar una forma de llegar desde una situación inicial a una situación deseada, usando las acciones que puede hacer un agente inteligente. Para eso, se necesita:*

- ***Estado**. Descripción del estado del mundo en el que se encuentra el agente de búsqueda.*
- ***Estado inicial**. Estado inicial en el que se inicia el agente de búsqueda.*
- ***Estado objetivo**. Si el agente alcanza este estado, finaliza y genera una solución.*
- ***Acciones**. Todas las acciones permitidas al agente.*
- ***Solución**. La ruta en el árbol de búsqueda desde el estado inicial hasta el estado objetivo.*
- ***Función de costo**. Asigna un valor de costo a cada acción. Necesario para encontrar una solución rentable.*
- ***Espacio de estados**. Conjunto de todos los estados.*

Es interesante observar una gran similitud entre los conceptos fundamentales de la Teoría de Juegos y esta definición de problemas.

Pongamos un ejemplo.

Ejemplo 19. Ocho reinas.

Queremos colocar ocho reinas en un tablero de ajedrez de 8x8 casillas, de manera que ninguna reina pueda atacar a la otra. Las partes de este problema según la definición anterior son:

[1] Consultar *Artificial Intelligence: A Modern Approach.* [7]

- Estado: la posición actual de las reinas en el tablero.
- Estado inicial: el tablero vacío, con algunas reinas colocadas.
- Estado objetivo: el tablero con ocho reinas que no se atacan entre sí.
- Acciones: colocar una reina en una casilla vacía (que no esté atacada por otra reina).
- Solución: la secuencia de acciones desde el estado inicial al estado objetivo.
- Función de costo: número de acciones realizadas. Se quiere minimizar este valor.
- Espacio de estados: conjunto de todas las posibles configuraciones del tablero, con cero o más reinas.

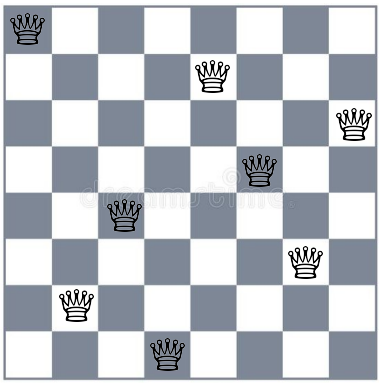


Figura 2.1: Ejemplo de un estado objetivo. Es importante no confundir con la solución.

Nota: Excepto que se indique lo contrario, trabajaremos con espacios de búsqueda finitos.

Definición 20. *El número de estados que se pueden generar a partir de un estado s se llama* ***factor de ramificación*** *$b(s)$, o b si el factor de ramificación es el mismo para todos los estados. Podemos definir también:*

- *La* ***profundidad*** *d es la medida de cuántos niveles tiene un árbol de búsqueda.*
- *El* ***factor de ramificación efectivo*** *de un árbol de profundidad d con n nodos totales es definido como el factor de ramificación que tendría un árbol con factor de ramificación constante, misma profundidad y mismo n.*
- *Un algoritmo de búsqueda se dice que es* ***completo*** *si encuentra una solución para cada problema que tiene solución. Si un algoritmo de búsqueda completo termina sin encontrar una solución, entonces el problema no tiene solución.*

Para una profundidad d y una cantidad de nodos n, el factor de ramificación efectivo se puede calcular mediante la suma de una serie geométrica de razón b:

$$n = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}.$$

Ejemplo 20. Mapa.

Supongamos que queremos hacer un viaje en autobús, pero únicamente tenemos las siguientes rutas con sus respectivas distancias:

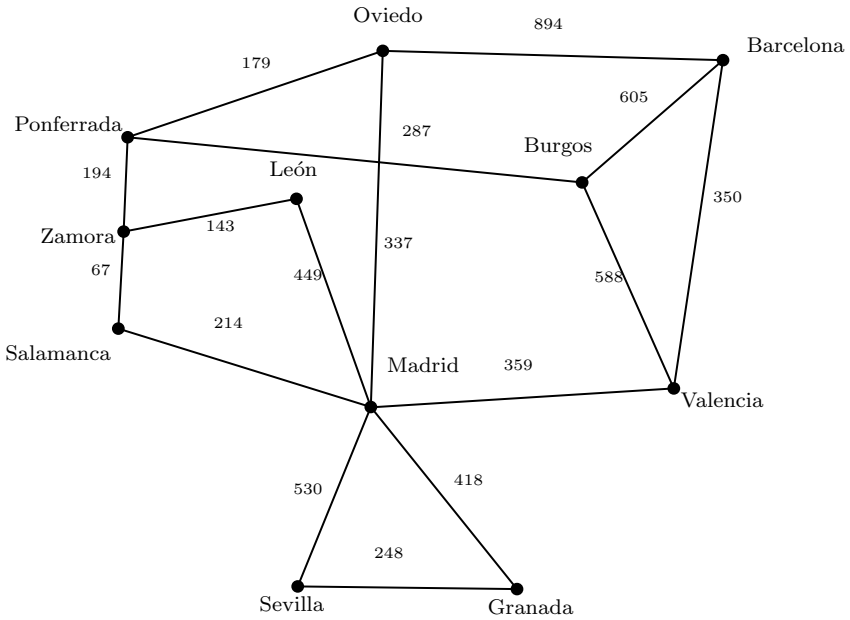


Figura 2.2: Posibles viajes representados en un grafo.

- Estado: la ciudad en la que nos encontramos.
- Estado inicial: la ciudad de origen.
- Estado objetivo: llegar a la ciudad de destino (por la ruta más corta).
- Acciones: moverse a una ciudad conectada.
- Solución: el viaje total.
- Función de costo: todos los kilómetros recorridos.
- Espacio de estados: todas las ciudades (nodos).

Después de representar el problema del viajero con un grafo, nos surge la pregunta de cómo encontrar una solución óptima que minimice el costo total del recorrido. Sin embargo, este problema no es trivial, ya que la complejidad aumenta exponencialmente con el número de nodos del grafo. Este problema tiene grandes utilidades en la vida real (¿Qué estrategias utilizan las grandes empresas para distribuir sus productos desde distintos países con el menor costo posible?). Para ello, comencemos a investigar sobre algoritmos de búsqueda.

Definición 21. *Diremos que un algoritmo de búsqueda es* ***óptimo*** *si encuentra siempre la solución óptima (siempre que exista).*

Comencemos a investigar sobre algoritmos de búsqueda.

2.1.1 Búsqueda en amplitud

La búsqueda en amplitud es un método que explora el árbol de búsqueda de forma sistemática y ordenada, desde el nivel superior hasta el inferior. El proceso consiste en examinar cada nodo de la lista de nodos para comprobar si cumple con el criterio de solución. Si se encuentra una solución, el programa finaliza. Si no, se expande el nodo actual y se añaden sus sucesores a la lista de nodos. Luego, se repite el mismo procedimiento con la nueva lista de nodos, que contiene los nodos generados en el nivel anterior. Este ciclo se repite hasta que se encuentre una solución o hasta que no queden más nodos por expandir.

La búsqueda en amplitud garantiza encontrar la solución más eficiente si existe alguna, es decir, la que requiere menos pasos o tiene menos costo. También es un método completo, que asegura hallar una solución siempre que el espacio de búsqueda sea finito. No obstante, presenta algunos inconvenientes, como el elevado uso de memoria, ya que necesita guardar todos los nodos generados en cada nivel, y el tiempo de ejecución, que puede ser muy extenso si el factor de ramificación (la cantidad media de sucesores por nodo) es alto o si la solución se encuentra muy lejos.

Veamos cómo sería su implementación, donde la función *nodosSucesores(nodo)* devuelve los nodos sucesores a un nodo pasado como argumento, la función *ObjetivoAlcanzado(nodo, objetivo)* devuelve un valor booleano que depende de si el nodo ha alcanzado el objetivo y *resto(lista)* devuelve el resto de los elementos.

Búsqueda en amplitud

```
1  BUSQUEDA_EN_AMPLITUD(listaNodos, objetivo)
2
3      lista nuevosNodos  // Definimos una lista
4
5      Para cada nodo en listaNodos
6          Si ObjetivoAlcanzado(nodo, objetivo)
7          // Si se alcanza el objetivo devolvemos solucion        ...

8              devolver("Solucion Encontrada" nodo)
9
10         nuevosNodos = nuevosNodos + nodosSucesores(nodo)
11         // En contra, agregamos los nodos que suceden con la ...
               funcion nodosSucesores
12
13     // Si seguimos teniendo nodos llamamos a la funcion de ...
           manera recursiva
14     Si nuevosNodos no vacio
15         devolver(BUSQUEDA_EN_AMPLITUD(nuevosNodos, objetivo))
16     sino
17         devolver("No hay solucion")
18         // En contra, devolvemos que no existe
```

Este algoritmo recorre los nodos del árbol en el orden que se muestra a continuación,

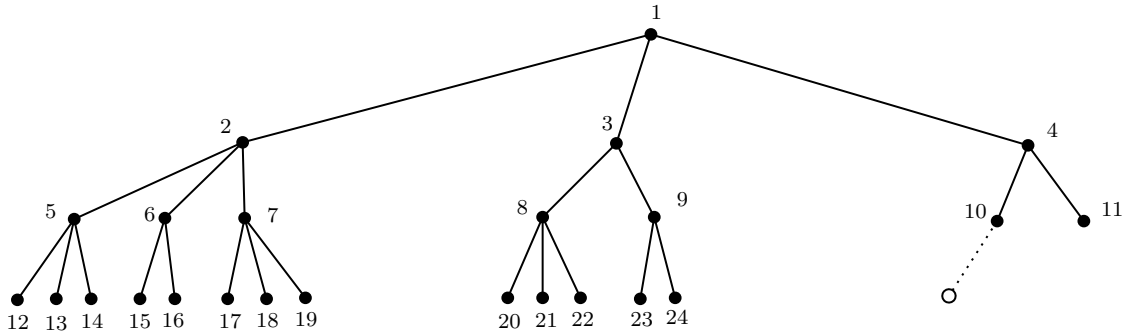


Figura 2.3: Ejemplo del recorrido del algoritmo de búsqueda en amplitud en un grafo.

el siguiente paso del algoritmo sería incluir el nodo 25 en su lista, lo que descartaría el camino 1-4-10 como solución. Si el camino 1-4-11 satisficiera el objetivo, el algoritmo se detendría y devolvería esta solución; en caso contrario, aplicaría el algoritmo de nuevo en el nodo 12.

A medida que aumenta la profundidad del árbol, el tiempo de cálculo y el espacio de memoria se multiplican de manera exponencial. Si el árbol tiene una profundidad d y un factor de ramificación b que no cambia, el tiempo será:

$$c \cdot \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d).$$

El algoritmo reutiliza la memoria para solo guardar el último nivel de nodos, pero la memoria necesaria será del orden de $O(b^d)$. Este algoritmo consume mucha memoria, además, puede resultar muy lento si el árbol es demasiado grande o el objetivo se encuentra a un nivel muy bajo del árbol.

2.1.2 Búsqueda en profundidad

El algoritmo de búsqueda en amplitud consiste en explorar todos los nodos de un mismo nivel del árbol de búsqueda antes de pasar al siguiente nivel. Este algoritmo tiene la ventaja de que encuentra la solución más corta si existe, pero también tiene el inconveniente de que consume mucha memoria, ya que tiene que almacenar todos los nodos de la frontera. Por esta razón, se puede considerar otro algoritmo que siga una estrategia diferente: en lugar de buscar "a lo ancho", buscar "a lo largo" del árbol de búsqueda. Este algoritmo se denomina búsqueda en profundidad y se basa en profundizar en una rama del árbol hasta llegar a un nodo terminal o ya visitado, y luego retroceder para explorar otras ramas.

El algoritmo de búsqueda en profundidad se puede realizar de forma recursiva. En cada iteración, se selecciona un nodo de la frontera y se genera sus hijos, que se agregan a la frontera. La selección del nodo se hace siguiendo un orden[2], el nodo que se ha incorporado más recientemente a la frontera es el primero que se saca. Así, el algoritmo avanza por un camino desde el nodo inicial hasta un nodo solución o sin salida, y luego vuelve atrás para continuar por otro camino. El algoritmo finaliza cuando encuentra un nodo solución o cuando la frontera está vacía. Para evitar explorar nodos repetidos, el algoritmo mantiene

[2]LIFO por sus siglas en ingés (Last In, First Out).

un registro de los nodos ya visitados.

Este algoritmo tiene una limitación importante cuando se aplica a problemas que tienen una ramificación infinita o muy grande, ya que puede quedar atrapado en un ciclo infinito sin encontrar la solución. Aunque nosotros estamos trabajando con problemas que tienen un espacio de estados finito, este inconveniente se puede evitar añadiendo un límite máximo a la profundidad que puede alcanzar el algoritmo, de forma que no explore más allá de ese nivel.

Veamos su implementación.

Búsqueda en profundiad

```
1  BUSQUEDA_EN_PROFUNDIDAD(nodo, objetivo)
2
3      // Cuando el nodo alcance el objetivo paramos
4      Si ObjetivoAlcanzado(nodo, objetivo)
5          devolver("Solucion Encontrada"  nodo)
6
7      // Guardamos los sucesores del nodo en una lista
8      lista nuevosNodos = nodosSucesores(nodo)
9
10     // Recorremos toda la lista
11     Mientras nuevosNodos no vacio
12         resultado = BUSQUEDA_EN_PROFUNDIDAD(nodosSucesores[0], ...
               objetivo)
13         Si resultado = "Solucion Encontrada"
14             devolver("Solucion Encontrada" nodo)
15
16         nuevosNodos = resto(nodosSucesores)
17
18     // Si no encontramos solucion en la lista
19     devolver("No hay solucion")
```

Se puede visualizar en la siguiente figura:

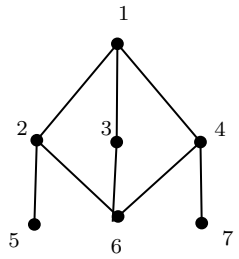


Figura 2.4: Grafo para ejemplificar el algoritmo de búsqueda en profundidad.

El recorrido que haría el algoritmo sería: 1, 1-2, 1-2-5, 1-2-6, 1-3, 1-3-6, 1-4, 1-4-6, 1-4-7.

El costo computacional de este algoritmo es mucho menor al anterior, como máximo se guardan los b nodos de cada profundidad, por lo que la memoria utilizada será de $b \cdot d$.

2.1.3 Búsqueda en profundidad iterativa

La búsqueda en profundidad iterativa combina las ventajas de la búsqueda en profundidad y la búsqueda en amplitud. Consiste en realizar sucesivas búsquedas en profundidad limitada, aumentando el límite de profundidad en cada iteración, hasta encontrar el estado objetivo o agotar el espacio de estados. Esta búsqueda es equivalente a la búsqueda en anchura en términos de completitud y optimalidad, pero usa mucha menos memoria, ya que solo almacena los nodos del nivel actual y el anterior. Además, visita los nodos del árbol de búsqueda en el mismo orden que una búsqueda en profundidad, lo que puede ser útil para algunos problemas.

Se basa en la idea de que si un estado objetivo se encuentra a una profundidad d, entonces también se puede encontrar con una búsqueda en profundidad limitada a d. Por lo tanto, se empieza con un límite de profundidad 0 e intenta encontrar el estado objetivo expandiendo solo el nodo inicial. Si no lo encuentra, incrementa el límite a 1 e intenta de nuevo expandiendo el nodo inicial y sus sucesores. Este proceso se repite aumentando el límite de profundidad en cada iteración hasta que se encuentre el estado objetivo o se alcance la profundidad máxima del árbol.

Veamos la inciación del código

Búsqueda en profundidad iterativa

```
1  ITERACION(nodo, objetivo)
2
3      limite = 0
4      Do
5          resultado = BUSQUEDA_PROFUNDIDAD_ITERATIVA(nodo, ...
               objetivo, 0, limite)
6          limite = limite + 1
7      Mientras resultado = "Solucion Encontrada"
```

Donde la función recursiva que nos interesa es la siguiente.

```
1  BUSQUEDA_EN_PROFUNDIDAD(nodo, objetivo, profundidad, limite)
2
3      Si ObjetivoAlcanzado(nodo, objetivo)
4          devolver("Solucion Encontrada", nodo)
5
6      lista nuevosNodos = nodosSucesores(nodo)
7      Mientras (nuevosNodos no vacio) Y (profundidad < limite)
8          resultado = BUSQEUDA_EN_PROFUNDIDAD(nodosSucesores[0], ...
               objetivo, profundidad + 1, limite)
9          Si resultado = "Solucion Encontrada"
10             devolver("Solucion Encontrada", nodo)
11         nuevosNodos = resto(nodosSucesores)
12
13     devolver("No hay solucion")
```

Este algoritmo es completo, es decir, garantiza encontrar siempre la solución, siempre y cuando exista y el árbol sea finito. La memoria necesaria para este algoritmo es la misma que la del algoritmo de búsqueda en amplitud. Aunque parezca que el algoritmo da mucho

más trabajo que los otros algoritmos, cuando el árbol es suficientemente grande.

Sea $N_b(d) \in \mathbb{N}$ el número de nodos con factor de ramificación b y profundidad d. Sea d_{max} la última profundidad utilizada en la búsqueda. Es decir, el último árbol que hemos investigado tiene $N_b(d_{max})$ nodos, con

$$N_b(D_{max}) = \sum_{i=0}^{d_{max}} b^i = \frac{b^{d_{max}+1} - 1}{b - 1}.$$

Hagamos la suma de todos los árboles para ver la cantidad de nodos,

$$\begin{aligned}
\sum_{i=1}^{d_{max}-1} N_b(d) = \sum_{i=1}^{d_{max}-1} \frac{b^{d_{max}+1} - 1}{b - 1} &= \frac{1}{b-1}((\sum_{i=1}^{d_{max}-1} b^{d+1}) - d_{max} + 1) \\
&= \frac{1}{b-1}((\sum_{i=2}^{d_{max}-1} b^{d}) - d_{max} + 1) \\
&= \frac{1}{b-1}(\frac{b^{d_{max}+1} - 1}{b-1} - 1 - b - d_{max} + 1) \\
&\approx \frac{1}{b-1}(\frac{b^{d_{max}+1} - 1}{b-1}) = \frac{1}{b-1} D_b(d_{max}).
\end{aligned}$$

Para cualquier factor de ramifiación $b > 2$ este número de nodos es menor que $N_b(d_{max})$ nodos en el último árbol. Para, por ejemplo, $b = 20$ en los primeros $d_{max} - 1$ árboles tienen sobre $\frac{1}{20-1} = \frac{1}{19}$ del número de nodos en el último árbol. El número de nodos explorados en las primeras iteracciones es muy pequeño en comparación con el número de nodos explorados en la última iteración, por lo que podemos ignorar todas las iteracciones excepto la última.

Ejemplo 21. Jarras.

Un ejemplo de aplicación de este algoritmo es la resolución del problema de las jarras de agua. El problema consiste en tener dos jarras con capacidades conocidas (por ejemplo, 3 y 5 litros) y un grifo ilimitado de agua. El objetivo es medir una cantidad exacta de agua (por ejemplo, 4 litros) usando solo las operaciones de llenar una jarra desde el grifo, vaciar una jarra al suelo o trasvasar agua entre las jarras. El espacio de estados se puede representar como un árbol donde cada nodo contiene el estado de las jarras (por ejemplo, $(0, 0)$ significa que ambas jarras están vacías) y cada arista representa una operación válida entre estados. El algortimo puede explorar este árbol empezando por el nodo inicial $(0, 0)$ e incrementando el límite de profundidad hasta encontrar el nodo objetivo $(4, x)$ o $(x, 4)$, donde x puede ser cualquier valor.

Por ejemplo, si el límite de profundidad es 0, solo se explora el nodo inicial y se comprueba si es solución. Si no lo es, se incrementa el límite a 1 y se exploran los nodos sucesores del inicial, que son los estados que se obtienen al llenar cada jarra desde el grifo: $(3, 0)$ y $(0, 5)$. Se comprueba si alguno de ellos es solución y si no lo son, se incrementa el límite a 2 y se exploran los nodos sucesores de estos dos nodos, que son los estados que se obtienen al vaciar cada jarra al suelo o trasvasar agua entre ellas: $(0, 0)$, $(3, 5)$, $(0, 3)$ y $(5, 0)$. Se repite este proceso hasta encontrar un estado que tenga 4 litros en alguna jarra o llegar al límite máximo de profundidad.

2.1.4 Heurística

Las heurísticas son estrategias de resolución de problemas que en muchos casos encuentran una solución muy rápido. Sin embargo, encontrarla no está garantizado. La búsqueda heurística podría requerir mucho más tiempo e incluso pueden resultar en que no se encuentre la solución. Los humanos usamos con éxito procesos heurísticos para todo tipo de cosas. Cuando elegimos un restaurante para cenar, por ejemplo, valoramos las diferentes opciones según unos pocos criterios simples como el precio, la calidad, la ubicación y las opiniones de otros clientes, y luego decidimos la mejor opción por intuición. En teoría, podría ser mejor probar todos los platos de cada restaurante antes de decidir cuál es el mejor. Por ejemplo, el restaurante podría tener algunos platos desastrosos. Si ese fuera el caso, la prueba habría sido útil. Sin embargo, no hacemos este tipo de prueba porque hay una probabilidad muy alta de que nuestra elección heurística sea acertada y nos lleve rápidamente a nuestro objetivo de disfrutar de una buena cena, en vez de dedicar todo el tiempo, esfuerzo y dinero probando todos los platos.

Las decisiones heurísticas están estrechamente relacionadas con la necesidad de tomar decisiones en tiempo real con recursos limitados. En la práctica, se prefiere una buena solución encontrada rápidamente a una solución que es óptima, pero muy costosa de obtener. Se utiliza una función de evaluación heurística $f(s)$ para estados para modelar matemáticamente una heurística. El objetivo es encontrar, con poco esfuerzo, una solución al problema de búsqueda planteado con un costo total mínimo.

A continuación, modificaremos el algoritmo de búsqueda primero en amplitud añadiendo la función de evaluación a él. Los nodos actualmente abiertos ya no se expanden de izquierda a derecha por nivel, sino según su valor heurístico. Del conjunto de nodos abiertos, el nodo con el valor mínimo siempre se expande primero. Esto se consigue evaluando los nodos cuando se generan y ordenándolos en la lista de nodos abiertos.

Veamos su implementación para seguir comentando este método.

Búsqueda en profundiad

```
1  BUSQUEDA_HEURISTICA(start, objetivo)
2      // Guardamos el nodo inicial en una lista
3      lista listaNodos = [start]
4      // Comenzamos un bucle infinito
5      Mientras Verdad
6          // Si no quedan nodos en la lista salimos y devolvemos ...
               mensaje
7          Si listaNodos es vacio
8              devolver("Sin solucion")
9
10         // Vamos a estudiar el primer nodo de la lista, y nos ...
               quedamos con el resto en la lista
11         nodo = listaNodos[0]
12         listaNodos = resto(listaNodos)
13
14         // Salimos y devolvemos la solucion si nuestro nodo es ...
               solucion
15         Si ObjetivoAlcanzado(nodo, objetivo)
16             devolver("Solucion Encontrada", nodo)
```

```
17
18          // Reordenamos los nodos que teniamos en la lista y los ...
                sucesores de nuestro nodo segun la heuristica asignada
19          listaNodos = ordenarHeuristica(nodosSucesores(nodo), ...
                listaNodos)
```

La función *ordenarHeuristica(lista1,lista2)* que ordena dos listas de nodos según unos pesos dados (la heurística). La mejor heurística sería una función que calcula los costos reales desde cada nodo hasta el objetivo. Pero para hacer eso, tendríamos que recorrer todo el espacio de búsqueda, que es justamente lo que la heurística trata de evitar. Por eso necesitamos una heurística que sea rápida y fácil de calcular.

¿Cómo encontramos esa heurística? Una idea interesante para encontrar una heurística es simplificar el problema para tener menos costo computacional para utilizar los costos del problema simplificado como una aproximación al problema real. Llamaremos h a esta función de aproximación.

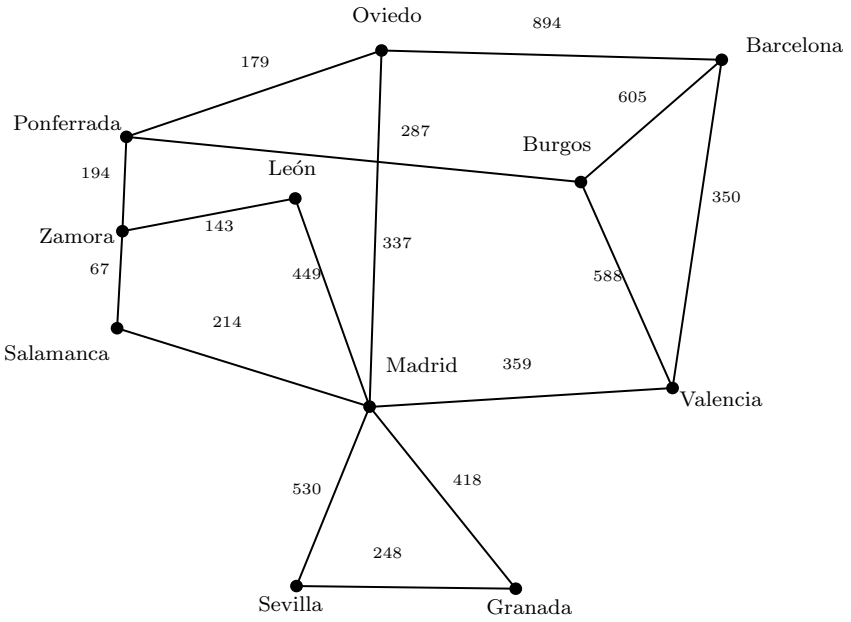
Para tener en cuenta los costos de cada nodo durante la búsqueda, añadiremos una función g a la función de aproximación. Es decir

$$f(s) = g(s) + h(s).$$

así obtenemos f como la función de evaluación heurística.

Definición 22. *El algoritmo de búsqueda heurística junto con la función de aproximación $f = g + h$ tal que h no sobrestima el costo real del nodo, se llama* ***A**** *o* ***A estrella****.*

En el Ejemplo 20 vimos un mapa de costos de diferentes viajes, calculemos la ruta óptima de León a Barcelona mediante este algoritmo



Valencia	348
Burgos	605
Oviedo	887
Madrid	601
Granada	852
Sevilla	1007
León	817
Salamanca	835
Zamora	925
Ponferrada	922

A la derecha tenemos una aproximación de la distancia de cada ciudad a Barcelona en coche, nuestros valores de la función h, por lo que el algoritmo para encontrar la ruta óptima desde León estaría representado de la siguiente forma,

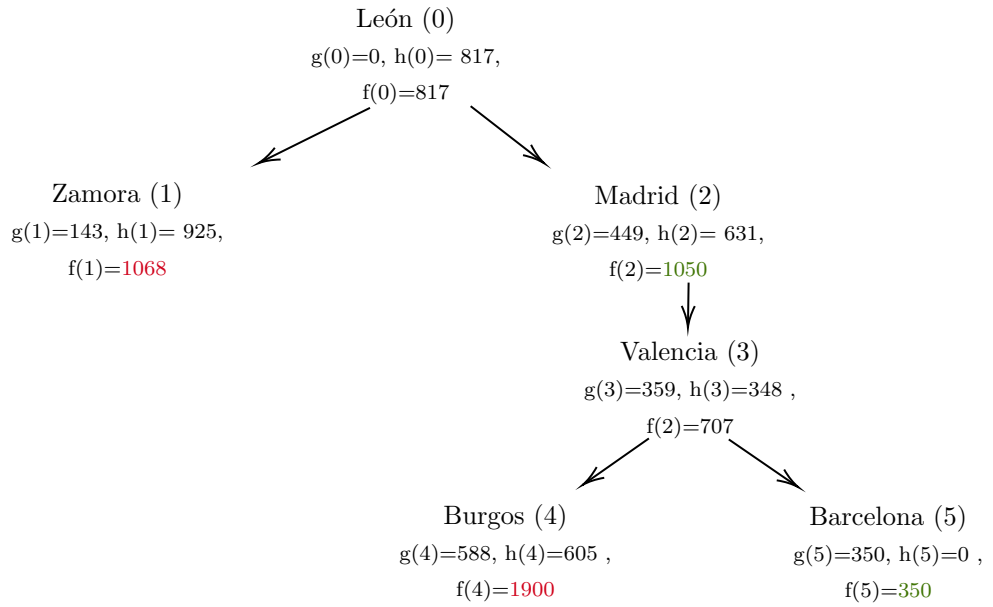


Figura 2.5: Representación de una posible solución.

El valor del nodo es el número que está entre paréntesis, es el orden en el que vamos visitándolos. Primero visitamos el nodo de Zamora, obteniendo el valor de $f(1)$, el siguiente nodo es Madrid con $f(2)$, como $f(1) > f(2)$ nos quedamos con el camino de Madrid. El algoritmo también debería visitar Granada, Sevilla y Oviedo, que hemos omitido de la explicación por comodidad. Nos vamos al nodo de Valencia, que tiene dos posibilidades: Burgos y Barcelona. Burgos nos da $f(4)$, que es menor (como es lógico) que $f(5)$. Luego ya tenemos una solución: León-Madrid-Valencia-Barcelona.

Este ejemplo es sencillo y se han simplificado algunos pasos para su comprensión, pero se puede ver que este algoritmo heurístico se puede adaptar a diferentes problemas que se quieran resolver, y que tiene mucha aplicación en el día a día (basta imaginar en el mapa de distribución de empresas internacionales como Amazon.com, Inc).

La función de aproximación determina la calidad del algoritmo A estrella. Según el problema a resolver, necesitamos más o menos información para tener una buena o mala función de aproximación. Por ejemplo, para jugar al ajedrez requerimos muchos datos de los posibles movimientos de nuestro jugador, pero para hallar el mes con menos visitantes en una ciudad, obtener los datos de las personas que entran y salen cada hora puede ser excesivo.

2.2 Algoritmos de aprendizaje automático

En el apartado anterior hemos introducido el concepto de algoritmos de búsqueda, que son una de las técnicas fundamentales de la Inteligencia Artificial[3]. Estos algoritmos permiten encontrar soluciones óptimas o aproximadas a problemas que se pueden formular como espacios de estados, es decir, como conjuntos de posibles configuraciones del mundo y las acciones que se pueden realizar para pasar de una a otra. Algunos ejemplos de problemas

[3]Consultar *Deep Learning.* [3]

que se pueden resolver mediante algoritmos de búsqueda son el problema del viajante y el problema de las ocho reinas.

Sin embargo, no todos los problemas se pueden abordar mediante algoritmos de búsqueda, o al menos no de forma eficiente o satisfactoria. Hay problemas que requieren una adaptación dinámica a los cambios del entorno o a las preferencias de los usuarios, o que implican una gran incertidumbre o complejidad. Para estos problemas, se necesitan otras técnicas de Inteligencia Artificial que sean capaces de aprender de los datos y mejorar su rendimiento sin necesidad de una programación explícita. Estas técnicas se engloban bajo el nombre de algoritmos de aprendizaje automático.

En contraste con los algoritmos de búsqueda, que se centran en encontrar soluciones mediante la exploración sistemática de un espacio de estados, los algoritmos de aprendizaje automático abordan problemas más complejos que requieren adaptación dinámica, manejo de incertidumbre y capacidad de aprendizaje a partir de los datos disponibles. Estas técnicas de Inteligencia Artificial han revolucionado numerosos campos, incluida la Teoría de Juegos.

Dentro del amplio espectro de algoritmos de aprendizaje automático, en este apartado nos enfocaremos en dos enfoques destacados: algoritmos genéticos y aprendizaje por refuerzo. Los algoritmos genéticos están inspirados en la teoría de la evolución y se basan en la idea de evolucionar soluciones a través de la selección natural y la reproducción de individuos con mejores características. Por otro lado, el aprendizaje por refuerzo se basa en el concepto de aprendizaje a través de la interacción con un entorno, donde un agente aprende a tomar decisiones para maximizar una recompensa o minimizar una penalización.

El objetivo principal de este apartado es comprender cómo estos algoritmos de aprendizaje automático pueden contribuir a la resolución de problemas complejos y cómo pueden mejorar el rendimiento y la eficiencia en diversas aplicaciones. A través de este análisis, podremos apreciar el potencial de la Inteligencia Artificial para ofrecer soluciones innovadoras y eficientes en el aprendizaje y la adaptación a partir de los datos disponibles.

2.2.1 Algoritmos genéticos

Los algoritmos genéticos se basan en la idea de simular los mecanismos de selección natural, reproducción y mutación para generar y mejorar soluciones a través de iteraciones. Estos algoritmos utilizan una población de individuos, cada uno representado por un cromosoma que codifica una posible solución al problema en cuestión.

Durante la ejecución de un algoritmo genético, los individuos compiten entre sí en función de su aptitud, que es evaluada mediante una función de evaluación. Los individuos con una aptitud más alta tienen más probabilidades de ser seleccionados para reproducirse y transmitir sus características a las generaciones futuras.

Los operadores genéticos, como la selección, el cruce y la mutación, son fundamentales en los algoritmos genéticos. La selección permite elegir a los individuos más aptos para la reproducción, mientras que el cruce combina características de dos individuos diferentes para generar descendencia. La mutación introduce cambios aleatorios en los cromosomas,

fomentando la exploración del espacio de soluciones.[4]

La aplicación de algoritmos genéticos abarca una amplia gama de problemas, desde la optimización de funciones matemáticas hasta la resolución de problemas de diseño y toma de decisiones en diferentes áreas. Su capacidad para buscar soluciones prometedoras en espacios de búsqueda complejos y su flexibilidad para adaptarse a diferentes dominios han sido ampliamente reconocidas.

Estos algoritmos ofrecen soluciones efectivas para una amplia gama de problemas matemáticos, desde la optimización y aproximación de funciones hasta la resolución de ecuaciones y el modelado matemático. A continuación, presentamos algunos ejemplos concretos de estas aplicaciones:

- Optimización de funciones: Los algoritmos genéticos se utilizan para encontrar valores óptimos de variables en problemas de optimización de funciones. Pueden explorar eficientemente el espacio de búsqueda de variables para encontrar el máximo o mínimo de una función, considerando restricciones y objetivos específicos. Esta capacidad de optimización los convierte en una herramienta valiosa en problemas matemáticos complejos.

- Modelado y simulación matemática: Los algoritmos genéticos se utilizan para desarrollar modelos matemáticos y realizar simulaciones en diversos campos. Pueden ajustar los parámetros del modelo para que se ajusten mejor a los datos observados y generen resultados más precisos. Esta capacidad de modelado y simulación es valiosa en áreas como la dinámica de sistemas, la optimización de redes y la predicción de fenómenos complejos.

- Aproximación de funciones: Los algoritmos genéticos pueden utilizarse para encontrar una función que se ajuste a un conjunto de datos, mediante la optimización de los parámetros de una función aproximada. Estos algoritmos pueden ajustar los parámetros para minimizar la diferencia entre los valores estimados y los datos reales, lo que resulta útil en problemas de interpolación y regresión.

Los algoritmos genéticos son una herramienta poderosa que se aplica no solo en matemáticas, sino también en diversos campos de estudio. Para comprender cómo funcionan los algoritmos genéticos, veamos un ejemplo.

Ejemplo 22. Aproximación por algoritmos genéticos.

Supongamos que queremos aproximar la función de Weierstrass [5]

$$f(x) = \sum_{i=0}^{\infty} a^i \cos(b^i \pi x), \text{ con } 0 < a < 1 \text{ y } b \text{ impar positivo,}$$

mediante un polinomio $g(x)$ de grado $n > 0$.

[4] Consultar *Genetic Algorithms in Search, Optimization and Machine Learning.* [2]

[5] El matemático alemán Karl Theodor Wilhelm Weierstrass (1815-1897) es considerado el fundador del análisis moderno. Creó la función de Weierstrass, que es un ejemplo de una función que es continua en todo su dominio pero no tiene derivada en ningún punto. Esto contradice la creencia de que una función continua solo puede ser no diferenciable en puntos aislados.

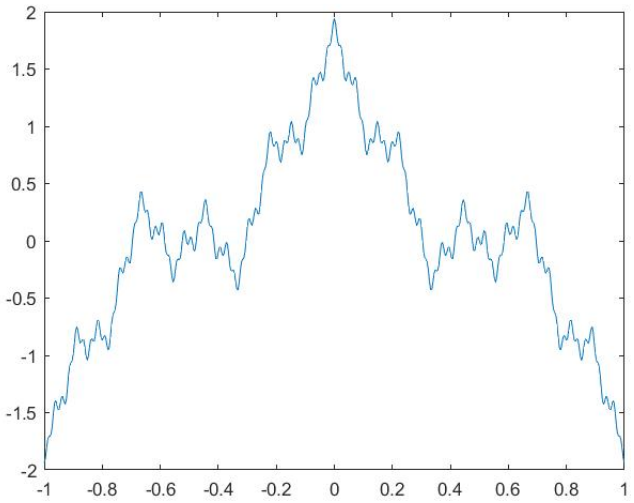


Figura 2.6: Función de Weierstrass con $a = 1/2$ y $b = 3$.

Vamos a crear una colección de polinomios de grado n al azar, es decir, dado un conjunto de índices I, para cada polinomio de la forma $g_i(x) = a_{in}x^n + ... + a_{i0}$ vamos a asignar los valores a_{ik} de forma aleatoria para cada $i \in I$ y cada $k = 0, ..., n$. Buscamos la mejor solución posible dentro de esta colección de polinomios.

A continuación, se evalúa el error de cada aproximación por cada polinomio por mínimos cuadrados calculando una aproximación a cada integral,

$$\int_{-1}^{1} (g_i(x) - f(x))^2 = \int_{-1}^{1} (a_{in}x^n + ... + a_{i0} - f(x))^2, \text{ para cada } i \in I,\ k = 0, ..., n.$$

Esta aproximación puede realizarse mediante métodos numéricos básicos, como la regla de Simpson o fórmulas de Newton-Cotes.

Después de medir los errores de cada polinomio como ajuste de los datos, se eligen los más adecuados (con menor error) para cruzarse. A estos polinomios se les da una probabilidad de ser elegidos, que depende de su error. Se toman dos polinomios al azar según esta probabilidad, y se combinan para crear un descendiente. Este proceso se realiza tantas veces como descendientes queramos, y nos permite mezclar y combinar información genética para generar variación.

Además de la reproducción, también incorporaremos el concepto de mutación. La mutación implica la introducción de cambios aleatorios en los individuos, permitiendo explorar nuevas regiones del espacio de búsqueda y evitar quedar atrapado en óptimos locales. La tasa de mutación controla la probabilidad de que un individuo sufra una mutación. En nuestro caso, podemos cambiar un elemento $a_{i_0k_0}$ por un número real aleatorio.

Luego de la reproducción y la posible mutación, sustituyendo los peores elementos por estos nuevos, se obtiene una nueva generación de individuos. Este proceso de selección, reproducción y mutación se repite durante un número determinado de generaciones o hasta que se cumpla un criterio de terminación predefinido, como que el error de nuestra

solución sea menor que una tasa fijada con anterioridad.

Los pasos a seguir en el ejemplo anterior, son los pasos generales de un algoritmo genético, con cada generación se espera que la población mejore su aptitud, ya que los individuos más aptos tienen una mayor probabilidad de ser seleccionados y transmitir sus características deseables a la siguiente generación. De esta manera, los algoritmos genéticos exploran y explotan el espacio de búsqueda, buscando converger hacia soluciones óptimas o cercanas a ellas.

Los pasos principales de una algoritmo genético son:

1. **Reproducción**: proceso en el cual los individuos más aptos de una población son seleccionados para crear una descendencia.

2. **Cruce**: proceso mediante el cual se combinan segmentos de información genética de dos individuos seleccionados de la población para crear nuevos individuos.

3. **Mutación**: proceso aleatorio que introduce cambios o alteraciones en la información genética de un individuo.

Para el proceso de cruzamiento, en nuestro ejemplo hemos usado una probabilidad que depende del error, pero hay muchos otros métodos que se pueden usar, como por ejemplo:

- Selección por torneo: se eligen dos elementos al azar y se compara su error. El que tenga menor error pasa a la siguiente ronda. Se repite este proceso hasta tener los que se van a cruzar.

- Selección por rango: se ordenan los elementos de menor a mayor y se les asigna una probabilidad de ser elegidos que depende de su posición en el orden. Los primeros tienen más probabilidad que los últimos.

- Selección por umbral: se fija un valor máximo y se descartan los elementos que lo superen. Los que queden se eligen al azar para cruzarse.

Aunque los métodos de elección sean distintos, todos ellos se basan en utilizar el concepto de aptitud para escoger los mejores candidatos.

Definición 23. *La **aptitud** de un individuo (o cadena) es una medida numérica que indica qué tan bien un individuo se adapta a un problema específico. Representa la calidad de una solución en relación con los objetivos establecidos.*

Para aplicar el operador de cruce a los elementos que hemos seleccionado de la población, necesitamos codificarlos de alguna manera para que el ordenador pueda manipularlos.

Una forma de hacerlo es utilizar cadenas binarias, formadas por ceros y unos, que representen los valores numéricos de los coeficientes de los polinomios que estamos utilizando como posibles soluciones. Para mostrar diversos ejemplos, observemos este proceso con otro problema.

Ejemplo 23. Algoritmos genéticos para la optimización.

Queremos encontrar el mínimo del polinomio $p(x) = x^6 - x^5 - x^3 - 2x^2 + x$ en el intervalo $[0, 3/2]$. Para que el ejemplo sea más visual, generaremos aleatoriamente una familia de 3 números en el intervalo como nuestra generación base G_0.

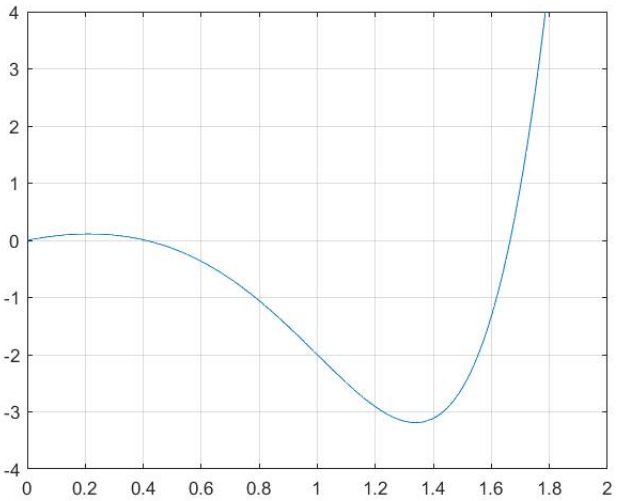


Figura 2.7: Grafo del polinomio $p(x) = x^6 - x^5 - x^3 - 2x^2 + x$ en el intervalo $[0, 2]$.

Supongamos que en la generación contiene los números: $a_{01} = 1{,}41$, $a_{02} = 0{,}86$ y $a_{03} = 0{,}51$. Para representar estos valores en forma binaria, multiplicaremos cada a_{i0} por 100 para obtener un número entero que pasaremos a binario, es decir,

- $a_{01} = 1{,}41 \to \tilde{a}_{01} = 141 \to \hat{a}_{01} = 10001101$,
- $a_{02} = 0{,}86 \to \tilde{a}_{02} = 86 \to \hat{a}_{02} = 1010110$,
- $a_{03} = 0{,}51 \to \tilde{a}_{03} = 51 \to \hat{a}_{03} = 110011$.

Para realizar la reproducción, tenemos que seleccionar qué dos progenitores vamos a utilizar, en este caso nos basaremos en el propio valor $p(a_{0i})$ de cada uno para ver qué elementos nos conviene elegir,

- $p(a_{01}) = -3{,}08$,
- $p(a_{02}) = -1{,}32$,
- $p(a_{03}) = -0{,}16$.

Como estamos buscando el valor mínimo, escogeremos los dos elementos con menor imagen. Vamos al proceso de cruce con ellos. Elegimos un número natural al azar entre el 1 y 8 (porque $\hat{a}_{01}$ tiene 8 elementos), digamos que ese número es 5, entonces, juntaremos los 5 primeros unos y ceros de $\hat{a}_{01}$ con los 3 últimos de $\hat{a}_{02}$:

$$\underline{10001}101 + 110\underline{011} = 10001011.$$

Ahora agregaremos la mutación cambiando el valor de uno de los valores de la cadena binaria escogido aleatoriamente, supongamos que hemos escogido el sexto elemento. El

resultado final por tanto es $\hat{a}_{13} = 10001111 \rightarrow \tilde{a} = 143 \rightarrow a_{13} = 1{,}43$. Así pues, nuestra nueva generación, intercambiando a_{03} por a_{13} será $G_1 = \{a_{01}, a_{02}, a_{13}\}$.

Actualizamos la notación con $a_{11} = a_{01}$ y $a_{12} = a_{02}$ y repetimos el proceso tantas veces queramos, cada una será una generación. Notamos que el nuevo valor es más cercano al mínimo, que se encuentra en 1,3385.

Un desafío importante de estos algoritmos genéticos, puede ser la forma de codificar nuestras soluciones. El ejemplo previo, contiene un error fácil de subsanar pero que da visibilidad a los posibles inconvenientes de utilizar este formato de cadenas de bits. Imaginemos que juntamos en vez de los 5 primeros dígitos de $\hat{a}_{01}$ con los 3 últimos de $\hat{a}_{02}$, hubieramos escogido los 5 primeros de $\hat{a}_{02}$ y los 3 útimos de $\hat{a}_{01}$, tendríamos la cadena: 11001101, que es el número binario 205, interpretado como en el ejercicio nos daría el número 2,05, fuera del intervalo en el que estamos estudiando la función.

En los algoritmos genéticos, las cadenas de ceros y unos (bits) que representan las posibles soluciones del problema se convierten en portadoras de una amplia gama de nociones acerca de lo que es importante para resolver el problema y lo que no lo es. Cada cadena contiene una valiosa información acumulada a lo largo de las generaciones anteriores, resultado de la selección y combinación de las soluciones más aptas. Esta información colectiva, dispersa en la población, constituye un recurso valioso para el algoritmo genético.

Cada individuo en la población alberga una combinación única de características y atributos, transmitidos a través de los procesos de selección, reproducción y mutación. Estas características encapsulan diferentes estrategias, enfoques y soluciones parciales que han demostrado cierto grado de eficacia en la resolución del problema. Al aprovechar la riqueza de información contenida en la población, el algoritmo genético tiene la capacidad de explorar una amplia variedad de soluciones potenciales, favoreciendo la convergencia hacia soluciones óptimas o de alta calidad.

La diversidad presente en la población de cadenas binarias permite al algoritmo genético explorar diferentes enfoques y combinar características prometedoras de diversas soluciones para generar descendientes mejor adaptados. A medida que las generaciones avanzan, las nociones más efectivas y las combinaciones exitosas de características se propagan y refuerzan, permitiendo que el algoritmo genético evolucione hacia soluciones más óptimas.

Aunque la selección y la reproducción se encargan de propagar las características prometedoras y mejorar gradualmente la calidad de las soluciones, la mutación juega un papel crucial al introducir variabilidad y exploración en el espacio de búsqueda.

La mutación consiste en la introducción aleatoria de cambios en los genes de las cadenas, lo que provoca pequeñas modificaciones en las soluciones. Estas alteraciones impredecibles permiten escapar de posibles estancamientos y de soluciones subóptimas en el espacio de búsqueda. Aunque la mutación es secundaria en el algoritmo, actúa como una fuerza exploratoria, rompiendo patrones preexistentes y permitiendo la aparición de nuevas características y soluciones potenciales.

La importancia de la mutación radica en su capacidad para mantener la diversidad en

la población y evitar la convergencia hacia una solución local óptima. Aunque la selección natural y la reproducción tienden a favorecer soluciones prometedoras, existe el riesgo de quedarse atrapado en un óptimo local y perder la oportunidad de descubrir soluciones aún mejores.

Podemos aplicar la mutación en los algoritmos genéticos de diferentes maneras, lo que nos brinda flexibilidad para adaptarla a las necesidades específicas del problema a resolver. Algunas de las técnicas más conocidas son las siguientes:

- Añadir: Esta técnica implica la inserción de un bit aleatorio en una posición al azar dentro de la cadena. Al agregar un nuevo bit, se introduce una posible modificación en la solución, lo que puede llevar a la exploración de diferentes configuraciones y caminos en el espacio de búsqueda.
- Eliminar: Consiste en quitar un bit de la cadena y reducir su longitud. Al eliminar un bit, se eliminan ciertos aspectos de la solución, lo que puede generar cambios significativos y abrir nuevas posibilidades en la búsqueda de soluciones óptimas.
- Invertir: Esta técnica implica cambiar el valor de un bit en la cadena, es decir, pasar de 0 a 1 o viceversa. Al invertir un bit, se introduce una modificación local en la solución, lo que puede permitir explorar diferentes configuraciones y ajustes en la búsqueda del óptimo global.
- Intercambiar: Consiste en permutar dos bits de la cadena entre sí. Al intercambiar dos bits, se modifican las relaciones y estructuras dentro de la solución, lo que puede generar cambios en la calidad y adaptabilidad de las soluciones generadas.

Es importante destacar que no existe un método de mutación superior a los demás. La elección de la técnica de mutación a aplicar dependerá del problema en cuestión, su complejidad y las características específicas. La experiencia y el conocimiento del dominio pueden ser útiles para determinar qué técnica de mutación es la más adecuada en cada caso, maximizando así las posibilidades de obtener soluciones óptimas.

A medida que avanzan las generaciones, nuestra población de posibles soluciones se acerca cada vez más a la solución óptima del problema. Es comprender las características de estas soluciones prometedoras, ya que esa información puede ser utilizada para guiar y mejorar el rendimiento de otras cadenas en futuras generaciones.

Continuaremos utilizando las cadenas de bits[6] como ejemplo para ilustrar el espacio de soluciones que los algoritmos genéticos exploran. Esta representación en forma de secuencias de bits nos permite visualizar claramente cómo se manipulan y combinan los genes para generar nuevas soluciones en cada generación del algoritmo genético.

Podemos denotar cada cadena como una combinación de elementos del siguiente conjunto $\{0, 1, *\}$, donde el símbolo $*$ representa un 0 ó un 1. Por ejemplo, la cadena $*101$ representa el conjunto de cadenas $\{0101, 1101\}$. A estos conjuntos los llamaremos **esquemas**. Podemos ver que en un algoritmo en el que las posibles soluciones contienen $n > 0$ bits, tendremos 2^n posibilidades y 3^n esquemas. Aunque esta notación parece complicar las cosas, considerar las cadenas de forma individual nos proporciona una cantidad limitada de información, al considerar esquemas, podemos estudiar similitudes entre las cadenas,

[6]Existen otros métodos como la codificación real, permutaciones, árboles o de conjuntos, por ejemplo.

recogiendo más información de la proporcionada por cadenas únicamente.

Cuando se introducen operaciones como la reproducción y el cruce en los algoritmos genéticos, los esquemas juegan un papel importante. La reproducción tiende a favorecer a los esquemas más aptos, lo que significa que se les asigna un mayor número de muestras en las generaciones posteriores. Por otro lado, el cruce puede afectar a los esquemas si los divide. Algunos esquemas de longitud corta pueden no ser interrumpidos por el cruce y, por lo tanto, se reproducen exitosamente en cada generación.

En el proceso de la mutación, cuando se realiza una mutación baja, no se interrumpe con frecuencia los esquemas. En general, los esquemas altamente aptos y de corta longitud se propagan de generación en generación. Esto muestra cómo los esquemas contribuyen a la eficacia de los algoritmos genéticos al guiar la búsqueda y mantener características beneficiosas a lo largo de las generaciones.

Los diferentes esquemas no son creados de igual forma. Algunos son más específicos que otros, por ejemplo el esquema $01110*$ está más definido que el esquema $*0**1*$. Definiremos dos parámetros para poder diferenciar numéricamente unos esquemas de otros.

Definición 24. *Denotaremos por $o(H)$ al número de posiciones fijas (que no son el símbolo $*$) de un esquema H.*

Definición 25. *Llamaremos $\delta(H)$ a la distancia etre la primera y la última posición fijada en una cadena.*

Por ejemplo, $o(0*1**1) = 3$ y $\delta(01*1*1**) = 6 - 1 = 5$.

Supongamos que en una generación $t > 0$ tenemos m cadenas en el conjunto generado por el esquema H. Escribiremos $m = m(H, t)$ ya que este número puede ir variando. En cada etapa, una cadena A_i tendrá cierta aptitud, que denotaremos por f_i, gracias a esta aptitud, podemos calcular la probabilidad de que esta cadena sea escogida, que denotaremos por $p_i = f_i / \sum_k f_k$. En una población de n cadenas en el instante t, ¿cuántos representantes del esquema H tendremos en el instante t+1? Este resultado viene dado por la ecuación:

$$m(H, t+1) = \frac{m(H,t) \cdot n \cdot f_H}{\sum_k f_k},$$

donde f_H denota la media de las aptitudes de todas las cadenas del conjunto H. Denotando la media de las aptitudes de todas las cadenas como $\hat{f} = \sum_k f_k / n$, podemos expresar la ecuación anterior como

$$m(H, t+1) = m(H,t)\frac{f_H}{\hat{f}}.$$

Podemos observar que los esquemas con valores de aptitud por encima del promedio de la población recibirán un número creciente de muestras ($\frac{f_H}{\hat{f}} > 1$), los conjuntos tendrán más elementos. En cambio, los esquemas con valores de aptitud por debajo de la media, disminuirán su número de elementos.

Si suponemos que un esquema H se mantiene por encima de la media en una cantidad constante $c\hat{f}$, donde c es constante, podemos expresar la ecuación así:

$$m(H, t+1) = m(H, t)\frac{(\hat{f} + c\hat{f})}{\hat{f}} = m(H, t)(1 + c).$$

Conociendo la generación aleatoria base G_0, podemos expresar la ecuación en función de $m(H, 0)$ de la siguiente forma,

$$m(H, t) = m(H, 0)(1 + c)^t.$$

Al ser t un número natural, es claro que tenemos una progresión geométrica de razón $(1 + c)^t$, que crecerá o disminuirá exponencialmente.[7] Significa que los esquemas con un rendimiento superior al promedio recibirán cada vez más oportunidades de reproducción en la siguiente generación, mientras que los esquemas con un rendimiento inferior al promedio recibirán menos oportunidades de reproducción.

La mutación vuelve a ser un factor clave, ya que aunque la reproducción nos permite explorar muchos esquemas, tenemos que extender el rango de búsqueda para analizar nuevas zonas del espacio que no estaban presentes en la primera generación inicial.

Ejemplo 24.

Supongamos que tenemos una cadena A de longitud 8 y dos esquemas H_1 y H_2 que contienen a dicha cadena,

$$\begin{aligned} A &= 1\,1\,1\,0\,1\,1\,0\,0, \\ H_1 &= 1 * * * 1 * * 0, \\ H_2 &= * * * * * 1\,0 * . \end{aligned}$$

Supongamos que vamos a hacer una reproducción entre la cadena A y otra cadena que hayamos elegido. Tenemos 7 espacios por donde podemos partir la cadena, escogemos un número al azar entre 1 y 7. Supongamos que escogemos el 4 y partimos la cadena y los esquemas por este espacio:

$$\begin{aligned} A &= 1\,1\,1\,0\,|\,1\,1\,0\,0, \\ H_1 &= 1 * * * |1 * * 0, \\ H_2 &= * * * * |* \ 1\,0 * . \end{aligned}$$

Vemos que el esquema H_1 se destruirá cuando hagamos la mutación, ya que dejamos posiciones fijas a ambos lados del corte de la cadena. En cambio, el esquema H_2 deja intactas las posiciones fijas tras este corte. Esto no es casualidad, tiene más posibilidades de sufrir una partición el esquema H_1 que el H_2. Esto podemos verlo rápidamente con las nociones explicadas anteriormente ya que,

$$\delta(H_1) = 8 - 1 = 7 > \delta(H_2) = 7 - 6 = 1.$$

Más precisamente, el esquema H_1 se destruirá con probabilidad $p_{dH_1} = \delta(H_1)/(8-1) = 7/7$ y sobrevivirá con probabilidad $1 - p_{dH_1} = 0$. En cambio, la probabilidad de ser destruido de H_2 es $p_{dH_2} = 1/8$.

[7] También podemos ver esa ecuación como un modelo exponencial discreto.

Realizando el mismo procedimiento del ejemplo para cualquier esquema H de logitud l, la probabilidad de sobrevivir en un cruce simple (como el anterior) será:

$$p_{sH} = 1 - p_{dH} = 1 - \frac{\delta(H)}{l-1}.$$

Si añadimos la probabilidad p_c de que sea escogido el esquema en el proceso de cruce, tenemos la siguiente desigualdad,

$$p_{sH} \geq 1 - p_c \frac{\delta(H)}{l-1},$$

si escogemos $p_c = 1$, obtenemos la expresión anterior.

La incorporación de esta nueva información nos brinda la oportunidad de realizar una estimación del número de cadenas que pertenecerán a cada esquema H en la siguiente generación.

$$m(H, t+1) \geq m(H, t) \frac{f_H}{\hat{f}} \left(1 - p_c \frac{\delta(H)}{l-1}\right).$$

Al comparar esta nueva expresión con la anterior que consideraba exclusivamente la reproducción, podemos apreciar el efecto combinado del cruce y la reproducción. El esquema H experimenta un crecimiento o deterioro en función de un factor multiplicativo. Con la presencia tanto del cruce como de la reproducción, dicho factor depende de dos aspectos fundamentales: si el esquema se encuentra por encima o por debajo del rendimiento promedio de la población y si el esquema tiene una longitud de definición relativamente corta o larga. Es claro que aquellos esquemas con un rendimiento superior al promedio y una definición más corta serán seleccionados con tasas de crecimiento exponencialmente mayores.

Únicamente nos falta agregar el proceso de mutación para completar nuestro estudio. Supongamos que la mutación a realizar será la alteración de una posición de la cadena escogida al azar con probabilidad p_m. Para que un esequema H sobreviva deben mantenerse todas las posiciones fijas, como cada posición de la cadena tiene una probabilidad $1 - p_m$ de sobrevivir y hay $o(H)$ posiciones fijas, la probabilidad de que un esquema sobreviva tras una mutación será,

$$(1 - p_m)^{o(H)}.$$

Para probabilidades p_m bajas ($p_m << 1$), el esquema sobrevivirá tras la mutación aproximadamente $1 - o(H)p_m$, por lo que concluimos,

$$m(H, t+1) \geq m(H, t) \frac{f_H}{\hat{f}} \left(1 - p_c \frac{\delta(H)}{l-1} - o(H)p_m\right).$$

Aún agregando el proceso de mutación, vemos que la tendencia de los esquemas sigue siendo la que anticipamos anteriormente: los esquemas genéticos prometedores, caracterizados por ser cortos, de bajo orden y tener un rendimiento superior al promedio, tienden a recibir un número creciente de copias en las generaciones sucesivas de un algoritmo genético.[8]

[8] Esta afirmación se conoce como el teorema del esquema de Holland o teorema fundamental de los algoritmos genéticos.

Ejemplo 25. Algoritmo genético.

Supongamos que eres un economista encargado de determinar la asignación óptima de recursos en una empresa manufacturera. La función $f(x, y) = \frac{xy}{(x^2+y^2)}$ representa la eficiencia de producción en diferentes combinaciones de los factores x e y, donde x e y representan la cantidad de dos recursos clave utilizados en el proceso de fabricación.

Sin embargo, debido a restricciones presupuestarias y limitaciones de disponibilidad de recursos, debes asegurarte de que las cantidades utilizadas para cada factor no excedan ciertos límites. Por ejemplo, supongamos que los límites para x e y son $0 \leq x \leq 10$ y $0 \leq y \leq 8$.

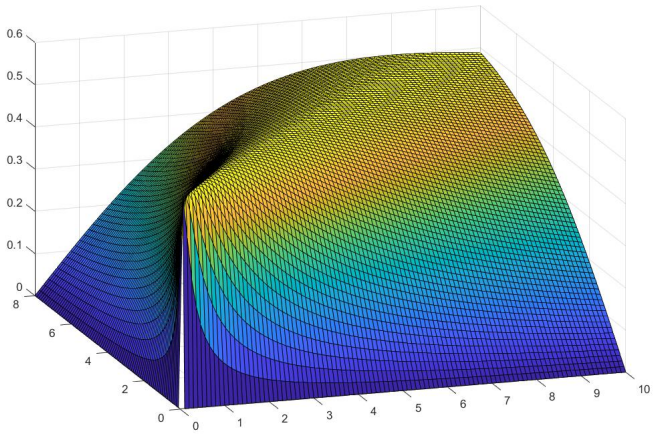


Figura 2.8: $f(x, y) = xy/(x^2 + y^2)$

El objetivo es encontrar la combinación óptima de recursos, es decir, determinar los valores de x e y que maximicen la eficiencia de producción.

Podemos intentar solucionar este problema analíticamente, pero para ello necesitamos calcular las soluciones de las siguientes ecuaciones para encontrar los puntos críticos de la función:

$$0 = \frac{\partial f(x,y)}{\partial x} = \frac{y(x^2+y^2) - xy(2x)}{(x^2+y^2)^2} = \frac{yx^2 + y^3 - 2x^2y}{(x^2+y^2)^2},$$

$$0 = \frac{\partial f(x,y)}{\partial y} = \frac{x(x^2+y^2) - xy(2y)}{(x^2+y^2)^2} = \frac{xy^2 + x^3 - 2xy^2}{(x^2+y^2)^2}.$$

Resolver ecuaciones de dos variables no lineales puede ser algo tedioso o no viable, por lo que utilizaremos un algoritmo genético para encontrar la mejor solución. Representaremos los individuos de nuestra población como la concatenación de dos cadenas de bits (una por variable). Cada variable será representada por ocho bits, que representará un número natural en binario.

Para trasladar este vector de números naturales $\mathbf{n} = (n_1, n_2)$ con n_1, n_2 menores que un natural N, a un intervalo $[a_1, b_1] \times [a_2, b_2]$ en el que estamos interesados utilizaremos

la siguiente escala:

$$x_1 = a_1 + \frac{(b_1 - a_1)n_1}{N},$$
$$x_2 = a_2 + \frac{(b_2 - a_2)n_2}{N}.$$

Para representar nuestras posibles soluciones, utilizaremos una cadena de 8 bits para cada variable, que representará un número natural en binario, que luego trasladaremos a nuestro intervalo. Siguiendo este procedimiento, tendremos que $N = 2^8 - 1$.

La aptitud de cada individuo (x_1, x_2) será el valor de $f(x_1, x_2)$, pues cuanto mayor sea este valor, mejor solución será. Escogeremos dos individuos (padres) de manera aleatoria según la aptitud de cada uno de ellos, los cruzaremos de manera simple (cortando aleatoriamente las cadenas y juntándolas) para crear dos individuos nuevos (hijos).

Para el proceso de mutación realizaremos el siguiente proceso en cada uno de los nuevos individuos:

1. Fijamos la probabilidad de mutación p_m (recordemos que $p_m << 1$).
2. Recorreremos toda la cadena de bits generando un número aleatorio entre 0 y 1, si este valor es menor que nuestro p_m cambiaremos el valor del bit.

Una vez tengamos calculados todos los nuevos individuos, sustituimos la población actual por la nueva. Realizamos este proceso tantas veces como queramos para acercarnos al máximo de la función.

Para observar mejor los resultados, consideraremos una población con pocos elementos e iremos ampliando el número de generaciones. En las siguientes gráficas podemos observar en una población de 7 y 15 elementos respectivamente, cuantas más generaciones realizamos mejor es la solución al problema.

Comencemos con 7 elementos. En la siguiente gráfica podemos observar como el algoritmo se acerca rápidamente a la solución, pero busca otras soluciones en el espacio, este es uno de los puntos fuertes de los algoritmos genéticos.

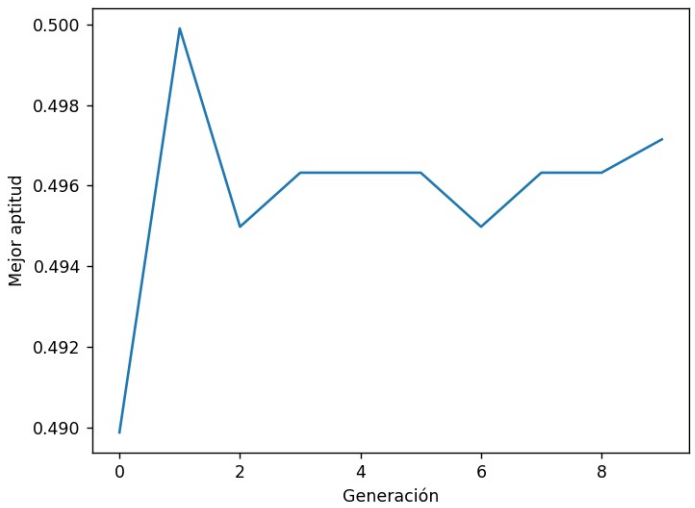


Figura 2.9: Algoritmo genético con 7 elementos.

Tenemos como salida del programa los siguientes valores:

```
Mejor individuo: [0 1 0 0 0 1 0 1 0 1 1 0 0 0 0 0]
Valores reales: (2.7058823529411766, 3.011764705882353)
Con valor f(x,y)= 0.49714613799858154
```

Con el aumento de elementos a 15, también incrementaremos el número de generaciones en el algoritmo genético. Observamos que, gracias al proceso de mutación, los individuos continúan explorando soluciones incluso cuando están cerca de una solución óptima.

El aumento en el número de elementos proporciona un espacio de búsqueda más amplio y desafiante para el algoritmo genético. Al introducir mayor variabilidad a través de la mutación, se evita que los individuos se estanquen en mínimos locales y se les permite seguir explorando el espacio de soluciones en busca de la mejor opción.

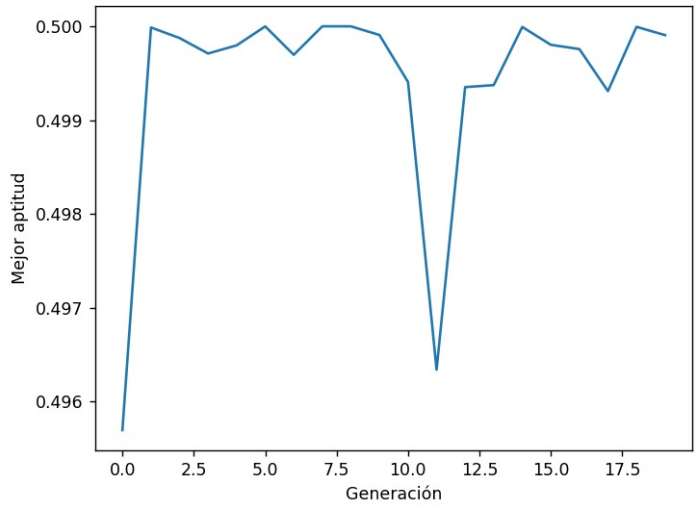


Figura 2.10: Algoritmo genético con 15 elementos.

Veamos las soluciones que presenta el algoritmo:

```
Generación 0: Mejor solución = (2.1176470588235294, 2.4156862745098038)
Generación 1: Mejor solución = (6.352941176470588, 6.305882352941176)
Generación 2: Mejor solución = (6.352941176470588, 6.211764705882353)
Generación 3: Mejor solución = (5.8431372549019605, 5.647058823529412)
Generación 4: Mejor solución = (5.8431372549019605, 5.67843137254902)
Generación 5: Mejor solución = (7.647058823529412, 7.623529411764705)
Generación 6: Mejor solución = (7.764705882352941, 7.4980392156862745)
Generación 7: Mejor solución = (7.764705882352941, 7.749019607843137)
Generación 8: Mejor solución = (5.7254901960784315, 5.741176470588235)
Generación 9: Mejor solución = (5.7254901960784315, 5.6156862745098035)
Generación 10: Mejor solución = (5.019607843137255, 5.270588235294118)
Generación 11: Mejor solución = (8.784313725490197, 7.780392156862745)
Generación 12: Mejor solución = (5.098039215686274, 5.364705882352941)
Generación 13: Mejor solución = (5.490196078431373, 5.772549019607843)
Generación 14: Mejor solución = (5.490196078431373, 5.458823529411765)
Generación 15: Mejor solución = (5.764705882352941, 5.929411764705883)
Generación 16: Mejor solución = (6.117647058823529, 5.929411764705883)
Generación 17: Mejor solución = (6.117647058823529, 5.803921568627451)
```

```
Generación 18: Mejor solución = (5.96078431372549, 5.929411764705883)
Generación 19: Mejor solución = (5.568627450980392, 5.67843137254902)
Mejor individuo: [1 0 0 0 1 1 1 0 1 0 1 1 0 1 0 1]
Valores reales: (5.568627450980392, 5.67843137254902)
Con valor f(x,y)= 0.4999046948484648
```

Al comparar las soluciones obtenidas con las aptitudes representadas en la gráfica, podemos pensar (acertadamente) que las soluciones del problema son los puntos de la forma $(x, y) = (y, y)$.

Este proceso de búsqueda continua y adaptación es una de las ventajas clave de los algoritmos genéticos en problemas complejos, ya que permiten la exploración de soluciones potenciales incluso en condiciones cercanas a la optimidad.

2.2.2 Aprendizaje por refuerzo

Tras explorar las bases principales de los algoritmos genéticos como una herramienta poderosa para abordar problemas de optimización y búsqueda en entornos complejos, nos adentramos ahora en un campo de estudio relacionado pero distinto: los algoritmos de aprendizaje por refuerzo. A diferencia de los algoritmos genéticos, que se centran en encontrar soluciones óptimas a través de la evolución y selección de poblaciones, los algoritmos de aprendizaje por refuerzo se enfocan en el aprendizaje de un agente interactivo a través de la retroalimentación proporcionada por un entorno dinámico.

En el contexto de los algoritmos de aprendizaje por refuerzo, un agente se encuentra inmerso en un entorno con el objetivo de aprender a tomar decisiones óptimas que maximicen la recompensa acumulada a lo largo del tiempo. Este tipo de enfoque de aprendizaje se basa en la idea de que el agente puede aprender de forma autónoma a través de la exploración y la interacción continua con su entorno, sin la necesidad de datos de entrenamiento, como otros tipos de Inteligencia Artificial.

Al igual que los algoritmos genéticos, los algoritmos de aprendizaje por refuerzo se inspiran en la naturaleza y utilizan técnicas de búsqueda y optimización. Sin embargo, su enfoque difiere en la forma en que se maneja la información. En lugar de depender de una representación codificada genéticamente, los algoritmos de aprendizaje por refuerzo emplean funciones de valor y políticas para guiar las acciones del agente en función de la recompensa esperada.

Comencemos esta sección con algunas definiciones.

Definición 26. *Llamamos **conjunto de estados** al conjunto $S = \{s_1, s_2, ..., s_N\}$, donde $N \in \mathbb{N}$. Cada s_i será el **estado** en el instante $i \leq N$.*

Definición 27. *El conjunto $A = \{a_1, a_2, ..., a_M\}$ con $M \in \mathbb{N}$, será el **conjunto de acciones**. Cada a_i será una acción. Llamaremos $A(s) \subset A$ al conjunto de acciones que se pueden realizar en un estado concreto $s \in S$.*

En el contexto del aprendizaje por refuerzo, el sistema toma una acción en un estado específico, lo que resulta en una transición hacia otro estado. Por lo que tenemos una función de transición $f : S \times A \to S$. La transición desde s a s' tomando la acción a será:

$$f(s, a) = s'.$$

Con estos conceptos, podemos interpretar el sistema como un proceso estocástico, el estado siguiente del sistema no es determinista, sino que se modela como una variable aleatoria. Por ejemplo, supongamos que el agente está jugando un juego en el que tiene que tomar decisiones en cada estado. La acción que el agente elige en un estado particular no garantiza un estado siguiente específico. En cambio, se define una distribución de probabilidad sobre los posibles estados siguientes y cada estado tiene una probabilidad asignada. El resultado exacto se determina al azar, de acuerdo con esta distribución de probabilidad.

En este escenario, la función transición es sustituida por una función de transición probabilística $\hat{f} : S \times A \times S \to [0, 1]$. La probabilidad de que el estado siguiente a s tomando la acción a sea s' será:

$$P(s'|s, a) = \hat{f}(s, a, s').$$

Con esta definición, se tiene que $\sum_{s'} \hat{f}(s, a, s') = 1$.

Al igual que en los procesos estocásticos, diremos que el sistema **posee la propiedad de Markov**[9] o es **markoviano** si:

$$P(s_{k+1}|s_k, a_k, s_{k-1}, a_{k-1}, ...) = P(s_{k+1}|s_k, a_k) = \hat{f}(s_k, a_k, s_{k+1}),$$

lo que quiere decir que el resultado de aplicar una acción únicamente depende del estado y acción actuales, no importan las tomadas en el pasado. Llamaremos MDP[10] a este modelo que definiremos formalmente en los siguientes párrafos.

Todavía tenemos que establecer una de las claves más importantes para los algoritmos de aprendizaje por refuerzo: ¿cómo sabe el agente si está tomando decisiones acertadas? Aquí es donde la función de recompensa entra en juego.

La **función de recompensa** es el elemento que asigna una medida numérica a cada par estado-acción en un MDP, proporciona información al agente sobre el desempeño de sus acciones en términos de éxito o fracaso. Denotaremos esta función por $\rho : S \times A \to \mathbb{R}$ tal que:

$$\rho(s_k, a_k) = r_{k+1}.$$

Cuando el sistema es estocástico, la recompensa estará asociada a transiciones, ya que éstas no dependen únicamente del estado y acción actuales. Entonces, podremos representar la función recompensa como $\hat{\rho} : S \times A \times S \to \mathbb{R}$ definida como:

$$\hat{\rho}(s_k, a_k, s_{k+1}) = r_{k+1}.$$

La función nos devuelve un valor en una transición, pero nuestro objetivo será maximizar la suma de las recompensas a lo largo del tiempo.

Definición 28. *Un **proceso de decisión de Markov** MDP es la tupla $\{S, A, f, \rho\}$, donde S es un conjunto de estados, A un conjunto de acciones, f una función de transición y ρ una función de recompensa.*

[9] Consultar *Machine learning: aprobabilistic perspective.* [5]
[10] Por sus siglas en inglés: Markov Decision Process.

Definición 29. *Llamaremos* ***estado terminal*** *al estado s cuando $\hat{f}(s,a,s) = 1$ y $\hat{\rho}(s,a,s') = 0$ para todos los estados y acciones posibles.*

Estos estados se refieren a los estados con los que terminamos el proceso, por ejemplo, en un robot que tiene que llegar a un punto dado, una vez alcanzado habríamos llegado a un estado terminal, o en una partida de tres en raya, cuando hemos terminado en empate.

Ejemplo 26. Representación de MDP.

Generalmente se representan estos procesos como un diagrama de grafos. En el siguiente ejemplo, debajo de cada arista tenemos dos valores, el primero es la imagen de la función de pagos, el segundo se refiere a la probabilidad de escoger cada acción.

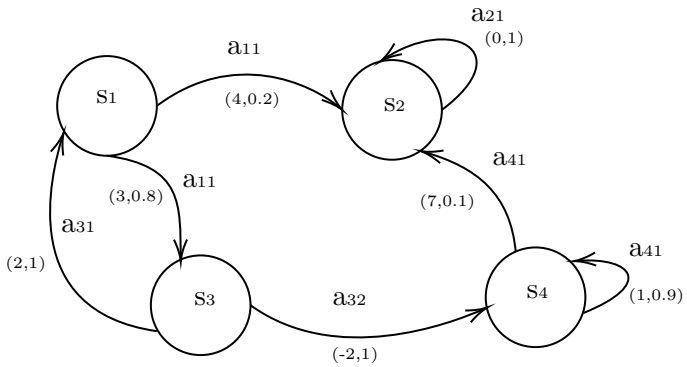


Figura 2.11: Ejemplo de MDP.

Por ejemplo, si se escoge la acción a_{41}, un 90 % de las veces volveremos al estado S_4 con un pago de 1, y un 10 % de las veces nos moveremos al estado S_2 con un pago de 7.

El estado S_2 es un estado terminal.

El objetivo principal de nuestro agente es maximizar las ganancias a largo plazo. Supongamos que en un estado dado, nuestro agente se enfrenta a dos opciones: elegir la acción A, que resulta en un pago inmediato de 4, o elegir la acción B, que tiene un pago de 3 pero con la posibilidad de acceder a otros estados donde se pueden obtener pagos mayores. Nuestro agente preferirá la acción B, ya que a largo plazo el pago total será mayor. Ahora bien, ¿cómo formalizamos esto? Aquí es donde entran en juego las políticas. Las políticas [11] son estrategias o reglas que determinan cómo un agente de aprendizaje debe actuar en un entorno con el fin de maximizar su recompensa.

Dado un MDP existen diferentes tipos de políticas, entre los cuales se encuentran los siguientes:

- Política determinista: Establece una correspondecia uno a uno entre los estados y las acciones, es decir, la política determina exactamente qué acción debe escoger el agente. La política será una función $\pi : S \to A$ tal que $\pi(s_k) = a_k$.
- Política estocástica: Especifica la probabilidad de seleccionar cada posible acción en un estado dado. En lugar de elegir una única acción, el agente puede tomar diferentes con distintas probabilidades. En este caso $\pi : S \times A \to [0,1]$ cumpliéndose para cada $s \in S$.

[11] Consultar *Deep Learning meets Game Theory.* [9]

$$\pi(s,a) \geq 0,$$
$$\sum_{a \in A} \pi(s,a) = 1.$$

- Política episódica: Define la distribución de probabilidad de las acciones a lo largo de un episodio completo. Nos referimos a un episodio como una secuencia completa de interacciones desde el estado inicial hasta la obtención de una recompensa final. Tiene en cuenta todas las acciones tomadas durante un episodio, en lugar de únicamente el estado actual.

- Política basada en valores: Se basa en la evaluación de la calidad o el valor esperado de cada acción en función de la función de valor.

Llamaremos retorno a la suma de todas los pagos recibidos bajo la política π. Viene dado por:

$$R^{\pi}(s_0) = \sum_{k=0}^{K} \gamma^k r_{k+1} = \sum_{k=0}^{K} \gamma^k \rho(s_k, a_k),$$

donde $\gamma \in [0,1)$ es una medida del peso que tienen las recompensas futuras para calcular el retorno. Podemos tener dos tipos de retornos, de horizonte infinito ($K = \infty$) o de horizonte finito ($K \in \mathbb{N}$). Generalmente, en los retornos de horizonte finito tomamos $\gamma = 1$ ya que la importancia de este factor reside en la necesidad de la convergencia de la serie cuando $K = \infty$.

En el aprendizaje por refuerzo, el objetivo primordial es maximizar el retorno, que representa la recompensa acumulada a largo plazo que un agente puede obtener al interactuar con un entorno. Sin embargo, para lograr esta maximización, resulta fundamental contar con una forma de evaluar y comparar la calidad de los estados y acciones. Es aquí donde entra en juego la función de valor, la cual permite estimar el valor esperado de estar en un estado particular o tomar una acción determinada. En esencia, la función de valor actúa como una guía para el agente al proporcionar una medida cuantitativa del valor o utilidad de cada estado o acción en función de las recompensas futuras que se esperan obtener. Al utilizar la función de valor, el agente puede tomar decisiones informadas y estratégicas para maximizar su retorno a largo plazo en el entorno de aprendizaje por refuerzo.

Definición 30. *La función retorno Q de una política π estima el valor esperado de tomar una acción específica en un estado dado bajo. Es la función $Q^{\pi} : S \times A \to \mathbb{R}$ tal que,*

$$Q^{\pi}(s,a) = \sum_{k=0}^{K} \gamma^k \rho(s_k, a_k).$$

En otras palabras, la función de retorno Q evalúa cuánto valor a largo plazo se puede esperar al tomar una acción a en un estado s considerando las recompensas futuras y las decisiones posteriores del agente, se utiliza para guiar la toma de decisiones del agente. Podemos hacer algunas modificaciones sobre esta ecuación para conseguir la ecuación de Bellman, que consigue expresarla de forma recursiva:

$$\begin{aligned} Q^{\pi}(s,a) &= \sum_{k=0}^{K} \gamma^{k} \rho(s_k, a_k) \\ &= \rho(s,a) + \gamma \sum_{k=1}^{K} \gamma^{k-1} \rho(s_k, \pi(s_k)) \\ &= \rho(s,a) + \gamma \left[\rho(f(s,a), \pi(f(s,a))) + \gamma \sum_{k=2}^{K} \gamma^{k-2} \rho(s_k, \pi(s_k)) \right] \\ &= \rho(s,a) + \gamma Q^{\pi}(f(s,a), \pi(f(s,a))). \end{aligned}$$

Queremos encontrar la mejor política posible que obtenga un retorno mayor. Luego, buscamos

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a).$$

Aplicando esta notación a la ecuación de Bellman, obtenemos que

$$Q^{*}(s,a) = \rho(s,a) + \gamma \max_{a'} Q^{*}(f(s,a), a').$$

La función de valor de acción Q asigna un valor a cada par estado-acción, indicando la utilidad esperada de tomar esa acción en ese estado. Por otro lado, podemos definir la función de valor de estado V que asigna un valor a cada estado, la bondad de estar en un estado específico.

Definición 31. *La función valor V de una política π asigna a cada estado un valor numérico que representa la utilidad de estar en ese estado. Es la función $V^{\pi} : S \to \mathbb{R}$ tal que,*

$$V^{\pi}(s) = \sum_{k=0}^{K} \gamma^{k} \rho(s_k, \pi(s_k)) = R^{\pi}(s).$$

La diferencia principal entre ambas radica en la información que proporcionan y cómo se utilizan. La función de valor de acción Q proporciona información sobre el valor de tomar acciones en un estado específico, mientras que la función de valor de estado V proporciona información sobre el valor de estar en un estado específico. Ambas funciones son utilizadas por el agente en el aprendizaje por refuerzo para tomar decisiones que maximicen las recompensas a largo plazo, pero cada una se enfoca en un aspecto diferente del proceso de toma de decisiones.

Es importante tener en cuenta, que definidas V y Q bajo una misma política π, tenemos que $V^{\pi}(s) = Q^{\pi}(s, \pi(s))$. Por lo tanto, podemos intentar conseguir la función V óptima siguiendo los mismos pasos que para Q^{*},

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) = \max_{a} Q^{\pi}(s,a).$$

Podemos caracterizar las funciones V^{π} y V^{*} por las ecuaciones de Bellman de igual forma que con Q y Q^{*}. Obtenemos entonces,

$$\begin{aligned} V^{\pi}(s) &= \rho(s, \pi(s)) + \gamma V^{\pi}(f(s, \pi(s))), \\ V^{*}(s) &= \max_{a} \left[\rho(s,a) + \gamma V^{*}(f(s,a)) \right]. \end{aligned}$$

Estas funciones Q y V se utilizan en el algoritmo de Iteración de Políticas de Howard, que es un enfoque iterativo para encontrar la política óptima en un entorno. A través de la evaluación y la mejora iterativa de la política, este algoritmo utiliza las funciones Q y V para actualizar gradualmente la política y maximizar las ganancias a largo plazo. De esta manera, las funciones Q y V juegan un papel central en el proceso de optimización de la política y el logro de un comportamiento óptimo del agente en el aprendizaje por refuerzo.

Algoritmo de Iteración de Políticas

El algoritmo de Iteración de Políticas de Howard[12] es un método utilizado en el aprendizaje por refuerzo para encontrar la política óptima en un entorno. Se compone de dos etapas que se repiten alternativamente hasta que se alcanza la convergencia en la política óptima. Estas etapas son: *Evaluación de la política actual* y *Mejora de la política*.

En la etapa de *Evaluación de la política actual*, se calculan los valores de los estados utilizando la función de valor V o la función de valor de acción Q de acuerdo con la formulación del problema. En esta etapa, se estiman los valores de los estados utilizando la ecuación de Bellman, que tiene en cuenta las recompensas inmediatas y los valores de los estados vecinos ponderados por las probabilidades de transición.

Una vez que se han evaluado los valores de los estados, se procede a la etapa de *Mejora de la política*. En esta etapa, se actualiza la política actual asignando una probabilidad más alta a la acción con el valor de estado o valor de acción más alto en cada estado. Esto se hace para maximizar las ganancias a largo plazo y mejorar gradualmente la política.

Estas dos etapas se repiten de forma iterada hasta que la política converja en la política óptima. Generalmente, en la primera iteración se fija una política π_0 arbitraria para iterar repetidamente ambos pasos. En cada iteración, la evaluación de la política actual proporciona una estimación más precisa de los valores de los estados, y la mejora de la política actualiza la política para tomar decisiones más óptimas en función de los valores estimados,

$$\pi_0 \to Q^{\pi_0} \to \pi_1 \to Q^{\pi_1} \to ... \to \pi_l \to Q^{\pi_l},$$
$$\pi_0 \to V^{\pi_0} \to \pi_1 \to V^{\pi_1} \to ... \to \pi_l \to V^{\pi_l}.$$

Proporciona una forma sistemática de mejorar gradualmente la política a través de la evaluación y la mejora iterativa. Su convergencia está garantizada siempre que se realice un número suficiente de iteraciones (cuando $l \to \infty$) y las probabilidades de transición sean bien definidas.

Q-Learning

Después de explicar el algoritmo de iteración de políticas, se puede aplicar el algoritmo Q-learning[13] para encontrar la política óptima en un entorno de aprendizaje por refuerzo. El

[12] Consultar *Reinforcement learning: An introduction.* [8]
[13] Ver nota anterior.[8]

algoritmo Q-learning utiliza una tabla de valores Q para almacenar los valores de recompensa esperados para cada par estado-acción. El agente utiliza esta tabla para seleccionar la acción óptima en cada estado.

El proceso de iteración de políticas se utiliza para actualizar los valores Q y seleccionar la acción óptima en cada estado. En la evaluación de la política, se calcula el valor de cada estado bajo una política dada. En la mejora de la política, se actualiza la política para seleccionar la acción óptima en cada estado.

El algoritmo Q-learning utiliza una política ϵ-greedy para seleccionar la siguiente acción. La política ϵ-greedy es una política que se utiliza en el aprendizaje por refuerzo para equilibrar la exploración y la explotación. La política selecciona la acción óptima con probabilidad $1-\epsilon$ y una acción aleatoria con probabilidad epsilon. La probabilidad ϵ se reduce a medida que el agente aprende más sobre el entorno. Por ejemplo, si $\epsilon = 0{,}1$, entonces el agente seleccionará la acción óptima con una probabilidad del 90 % y una acción aleatoria con una probabilidad del 10 %. A medida que el agente aprende más sobre el entorno, la probabilidad de seleccionar una acción aleatoria disminuye.

Ejemplo 27. Laberinto.

Supongamos que un agente debe moverse por un laberinto y encontrar una salida. En cada estado del juego, el agente tiene varias acciones posibles que puede tomar. El objetivo del agente es encontrar la salida del laberinto lo más rápido posible.

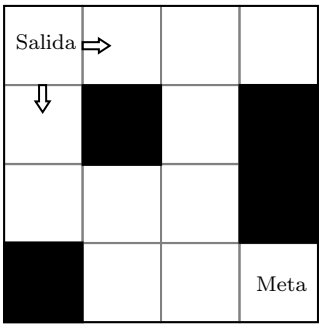


Figura 2.12: Ejemplo de posible laberinto.

Para resolver este problema, se puede aplicar el algoritmo Q-learning con una política epsilon-greedy para encontrar la política óptima en el entorno del laberinto. El algoritmo utiliza una política ϵ-greedy para seleccionar la siguiente acción y actualiza los valores Q en cada iteración. El proceso sería el siguiente:

1. Definir el entorno. En primer lugar, se debe definir el entorno en el que el agente va a aprender. En este ejemplo, el entorno es un laberinto de 4x4.

2. Definir la función de recompensa. En este ejemplo, la función de recompensa podría ser una función que otorga una recompensa positiva al agente por encontrar la salida del laberinto y una recompensa negativa por cada paso adicional que tome. Por ejemplo, se podría otorgar una recompensa de +10 al agente por encontrar la salida del laberinto y una recompensa de -1 por cada paso adicional que tome.

3. Inicializar la tabla de valores Q. La tabla de valores Q es una tabla que almacena los valores de recompensa esperados para cada par estado-acción. En este ejemplo,

la tabla de valores Q podría ser una matriz donde las filas representan los estados y las columnas representan las acciones posibles. Cada valor en la tabla representa el valor Q para ese par estado-acción.

4. Seleccionar una acción. Para seleccionar la siguiente acción, se utiliza una política ϵ-greedy. La política selecciona la acción óptima con probabilidad $1 - \epsilon$ y una acción aleatoria con probabilidad ϵ. Por ejemplo, si $\epsilon = 0{,}25$, entonces el agente seleccionará la acción óptima con una probabilidad del 75 % y una acción aleatoria con una probabilidad del 25 %.

5. Actualizar los valores Q. Después de seleccionar una acción y observar la recompensa recibida, se actualiza el valor Q correspondiente en la tabla de valores Q utilizando la ecuación de Bellman para Q.

6. Repetir los pasos 4 y 5 hasta que se alcance un estado final o se alcance un número máximo de iteraciones.

7. Después de completar el aprendizaje, se puede utilizar la política óptima para guiar al agente a través del laberinto y encontrar la salida lo más rápido posible.

2.3 Aplicaciones de la Inteligencia Artificial a la Teoría de Juegos

En los apartados anteriores, hemos explorado dos ramas principales de la Inteligencia Artificial, pero es importante destacar que este campo es extremadamente amplio y abarca una variedad de enfoques y técnicas[14]. Uno de esos enfoques es el uso de redes neuronales, que juegan un papel fundamental en muchas aplicaciones de Inteligencia Artificial.

Las redes neuronales son un modelo computacional inspirado en el funcionamiento del cerebro humano. Están compuestas por una colección interconectada de unidades de procesamiento llamadas neuronas artificiales o nodos, que trabajan en conjunto para resolver problemas complejos de manera eficiente. Cada neurona artificial procesa la información recibida, realiza cálculos y transmite el resultado a través de conexiones ponderadas a otras neuronas. Estas conexiones, a menudo representadas por pesos, determinan la importancia relativa de las señales de entrada en el proceso de toma de decisiones de la red neuronal.

Una de las características clave de las redes neuronales es su capacidad para aprender y adaptarse a partir de datos. A través de un proceso de entrenamiento, las redes neuronales pueden ajustar los pesos de las conexiones internas para optimizar su rendimiento en una tarea específica. El aprendizaje se realiza mediante la presentación repetida de ejemplos de entrada y la comparación de las salidas generadas por la red con las salidas deseadas. A medida que la red se expone a más ejemplos, los pesos se ajustan gradualmente para mejorar su capacidad de generalización y realizar predicciones precisas incluso en situaciones nuevas.

Las redes neuronales han demostrado ser muy efectivas en una amplia gama de aplicaciones de Inteligencia Artificial. Por ejemplo, en el campo del procesamiento del lenguaje natural, las redes neuronales se utilizan para tareas como el reconocimiento y generación

[14] Consultar *Artificial Intelligence and Games*. [10]

de texto, la traducción automática y la clasificación de sentimientos. En el campo de la visión por computadora, las redes neuronales han revolucionado el reconocimiento de objetos, el etiquetado de imágenes y el seguimiento de objetos en videos. También se utilizan en áreas como la medicina, la robótica, las finanzas y muchos otros campos donde se requiere el análisis y procesamiento de grandes cantidades de datos.

Además de la amplia gama de aplicaciones de las redes neuronales en la Inteligencia Artificial, es interesante explorar cómo se combinan con la Teoría de Juegos.

En particular, se ha observado que el aprendizaje profundo puede contribuir significativamente en juegos de suma cero, donde las ganancias de un jugador son exactamente iguales a las pérdidas del otro jugador. A través del análisis de grandes volúmenes de datos, las redes neuronales pueden descubrir estrategias óptimas directamente de los datos, evitando la necesidad de cálculos exhaustivos de estrategias equilibradas de Nash. Esto no solo puede ser más eficiente, sino también más preciso en muchos casos.

Otro aspecto interesante es el papel del aprendizaje profundo en juegos con información incompleta. Al modelar y analizar datos de juegos complejos con información parcial, las redes neuronales pueden capturar patrones sutiles y tendencias emergentes. Esto puede proporcionar a los jugadores una comprensión más profunda de las dinámicas estratégicas y ayudar en la toma de decisiones más informadas. Por ejemplo, en áreas como la economía, donde la información incompleta es común, el aprendizaje profundo puede ofrecer una nueva perspectiva en la toma de decisiones estratégicas.

La combinación del aprendizaje profundo con la Teoría de Juegos abre nuevas posibilidades y desafíos en el análisis y la resolución de problemas estratégicos. Al aprovechar las capacidades del aprendizaje profundo para descubrir patrones complejos, aprender de los datos y representar información de manera más precisa, se pueden obtener resultados más precisos en la predicción de estrategias óptimas y en la comprensión de las interacciones estratégicas.

Un enfoque interesante que combina la Teoría de Juegos con las redes neuronales es considerar cada neurona en la red como un jugador que lucha estratégicamente con otras neuronas. En esta configuración, se puede utilizar la Teoría de Juegos para analizar las interacciones y la dinámica competitiva entre las neuronas en el proceso de aprendizaje y toma de decisiones.

Cada neurona en la red puede representarse como un jugador que tiene sus propias acciones y estrategias. Estas acciones pueden corresponder a la activación o desactivación de la neurona, mientras que las estrategias pueden referirse a los patrones de activación que la neurona adopta en respuesta a las entradas recibidas. Al considerar cada neurona como un jugador estratégico, se puede emplear la Teoría de Juegos para analizar cómo estas decisiones individuales afectan el rendimiento global de la red y cómo se pueden lograr equilibrios estratégicos.

La Teoría de Juegos puede proporcionar herramientas y conceptos útiles para modelar las interacciones entre las neuronas y estudiar cómo las estrategias de una neurona pueden influir en las decisiones de otras neuronas. Por ejemplo, se pueden utilizar conceptos como la estrategia dominante, el equilibrio de Nash y la maximización de utilidad para analizar

cómo las neuronas pueden llegar a una configuración estable donde cada una juega su mejor estrategia dada la estrategia de las demás.

Este enfoque basado en la Teoría de Juegos y redes neuronales tiene el potencial de mejorar la capacidad de adaptación y aprendizaje de la red. Al considerar las neuronas como jugadores estratégicos, se pueden modelar y analizar escenarios competitivos en los cuales las neuronas ajustan sus estrategias y acciones para maximizar su rendimiento individual y, en última instancia, el rendimiento colectivo de la red.

2.3.1 Minimax

A lo largo de los capítulos anteriores, hemos explorado diversos conceptos y algoritmos relacionados con la Teoría de Juegos y los algoritmos de búsqueda en el contexto de la Inteligencia Artificial. Sin embargo, hay un algoritmo que destaca como uno de los más famosos y poderosos en este ámbito, y es crucial mencionarlo: el algoritmo Minimax[15].

El algoritmo Minimax es ampliamente reconocido y utilizado en la Teoría de Juegos y la toma de decisiones estratégicas. Su objetivo principal es encontrar la estrategia óptima en juegos de suma cero, donde las ganancias de un jugador se corresponden exactamente con las pérdidas del otro jugador. El algoritmo Minimax se aplica en la Inteligencia Artificial para crear sistemas que puedan jugar a juegos como el ajedrez, las damas, el tres en raya o el go, entre otros.

La idea central detrás del algoritmo Minimax es simular todas las posibles secuencias de movimientos en el juego, construyendo un árbol de búsqueda en el que cada nivel representa un turno de juego. En cada nivel, el algoritmo alterna entre los jugadores, asumiendo que cada uno de ellos elige la acción que maximiza su propia ganancia y minimiza la ganancia del oponente.

El objetivo del algoritmo Minimax es encontrar el mejor movimiento posible para el jugador actual, asumiendo que el otro jugador también actúa racionalmente y trata de maximizar su beneficio.

Veamos el funcionamiento de este método con un ejemplo.

Ejemplo 28. Tres en raya.

Jugaremos al 3 en raya, donde nosotros utilizaremos la ficha X. Queremos conseguir la mayor puntuación posible cuando el juego termine, que será 10 si hacemos una linea de tres X, -10 si el otro jugador hace una línea de tres O y cero si ninguno hace una línea, teniendo así un juego de suma cero.

Para resumir la explicación del algoritmo a algunos pasos, supongamos que nos encontramos en la siguiente jugada,

[15]Consultar *Artificial Intelligence and Games*. [10]

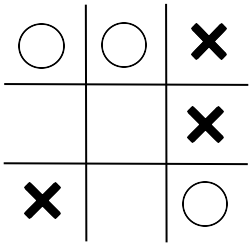


Figura 2.13: Estado de la partida, con el siguiente turno para el jugador con la ficha X.

Tenemos tres posibles huecos para poner nuestra X y queremos maximizar nuestra ganancia, por lo tanto, nuestro objetivo sería escoger la casilla que me de mejor pago (casilla del medio). En cambio, nuestro adversario querrá minimizar nuestro pago.

Los pasos del algoritmo Minimax serían:

1. Asignar una puntuación a cada estado final del juego (10, -10, 0).
2. Explorar todas las posibles jugadas desde el estado actual, alternando entre maximizar la puntuación (en nuestro caso) y minimizar la puntuación (en el turno del oponente).
3. Devolver las jugadas que tiene puntuación máxima.

Veamos los posibles caminos que puede seguir el juego según nuestra elección para realizar el algoritmo:

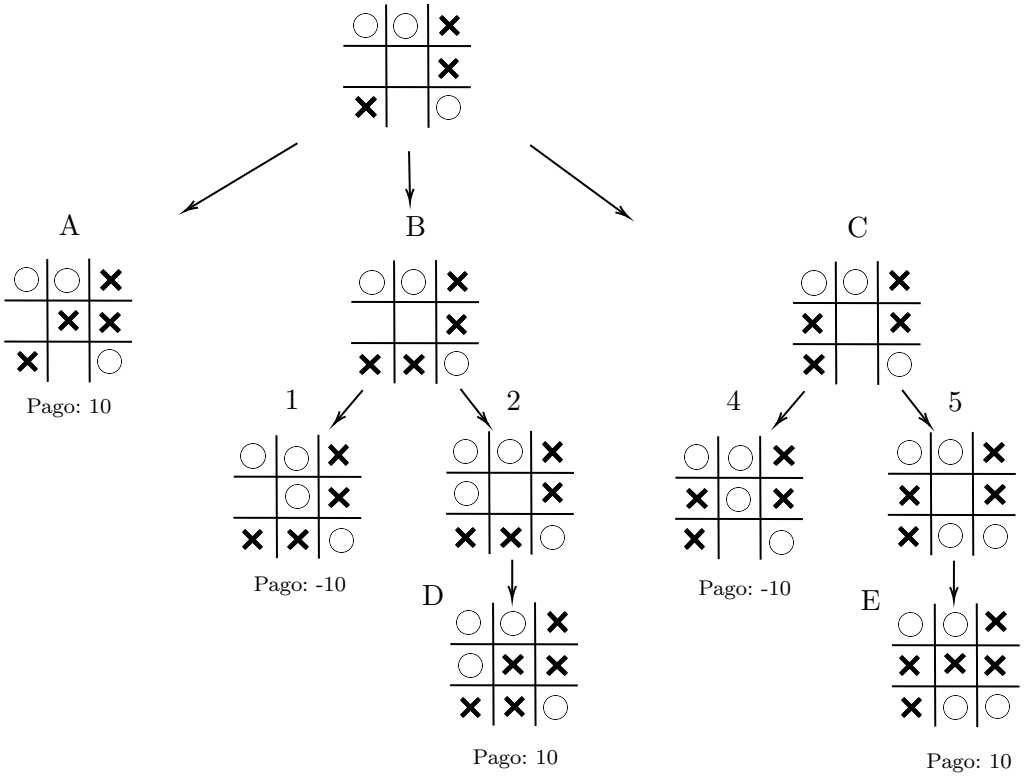


Figura 2.14: Diferentes estados del juego según las posibilidades de cada jugador.

Los pasos que sigue el algoritmo son:

1. En nuestro turno, el algoritmo genera las 3 posibilidades que tenemos, los estados A, B y C.

2. El estado A nos da directamente un pago de 10 y el fin del juego.

3. Los estados B y C generan dos estados nuevos cada uno. El estado B genera los estados 1 y 2 y llama al algoritmo Minimax. El mismo proceso se realiza con los estados 4 y 5.

4. En los estados 1 y 4 se devuelve un pago de -10.

5. Los estados 2 y 5 generan un estado nuevo respectivamente, ambos con un pago de 10.

6. En los estados A, B y C es el turno del adversario (ficha O) que intentará minimizar nuestro pago, por lo tanto, escogerá la opción 1 y 4.

7. Como finalmente tenemos como elección los estados: A (pago de 10), 1 (pago de -10) y 4 (pago de -10), la elección óptima que debemos realizar será el estado A.

Resumiendo, en nuestro turno querremos maximizar la ganancia, mientras que el adverario querrá minimizarla. Alternando estos estudios durante todos los posibles estados obtendremos uno o varios caminos óptimos a seguir. En la situación en la que se encontraba el juego, la opción correcta es poner la ficha en el cuadrado del centro del tablero.

El estado en el que hemos realizado este ejemplo, es uno de todos los posibles estados si hubieramos empezado con un tablero vacío, siendo nuestra solución una de las ramas del grafo.

El algoritmo Minimax es una forma de encontrar el mejor movimiento posible en un juego de suma cero, donde el beneficio de un jugador es la pérdida del otro. El algoritmo asume que ambos jugadores son racionales y buscan maximizar su utilidad. Sin embargo, el algoritmo tiene algunas limitaciones que se pueden mejorar con diferentes métodos.

Uno de los problemas del algoritmo Minimax es que puede ser muy lento e ineficiente para juegos con un gran espacio de estados, como el ajedrez o el go. Para resolver este problema, se puede usar la poda alfa-beta, que elimina las ramas del árbol de búsqueda que no son relevantes para el resultado final. De esta forma, se reduce el número de nodos que se exploran, lo que ahorra tiempo y memoria. Otra forma de mejorar la eficiencia del algoritmo es usar heurísticas o funciones de evaluación que estimen el valor de un estado sin explorar todo el árbol. Estas funciones simplifican el cálculo del valor de un estado, lo que facilita la toma de decisiones.

Otro problema del algoritmo Minimax es que puede ser muy vulnerable a la incertidumbre y al azar, que son comunes en muchos juegos reales. Para resolver este problema, se puede usar la Teoría de Juegos bayesiana, que incorpora las creencias y las probabilidades de los jugadores sobre los estados y las acciones. De esta forma, se puede actualizar el valor de un estado según la información disponible, lo que hace al algoritmo más realista y adaptable. Otra forma de mejorar la robustez del algoritmo es usar el algoritmo expectimax, que calcula el valor esperado de un estado teniendo en cuenta las posibles acciones aleatorias del oponente o del entorno. Estas acciones pueden afectar al resultado

del juego, por lo que es importante considerarlas.

Estos métodos no eliminan el algoritmo Minimax, sino que lo mejoran. Permiten al algoritmo Minimax encontrar el mejor movimiento posible de forma más rápida, eficiente y robusta. Así, se puede aplicar el algoritmo Minimax a juegos más complejos y realistas.

Capítulo 3

Algoritmos genéticos aplicados a la Teoría de Juegos

En este capítulo abordaremos el problema de diseñar agentes inteligentes que puedan jugar a juegos estratégicos mediantes algoritmos genéticos. En particular, estudiaremos dos juegos: el dilema del prisionero repetido (Ejemplo 15) y la guerra de precios repetida. Estos juegos modelan situaciones en las que los agentes deben decidir entre competir o cooperar con otros agentes (incluído él mismo) teniendo en cuenta las consecuencias a largo plazo de sus acciones.

Estos juegos son relevantes para entender el comportamiento humano y social en diversos contextos, como la economía, la política, la biología o la psicología. Para representar las acciones de los agentes, utilizaremos el concepto de autómata finito, que es un modelo computacional que realiza cómputos en forma automática sobre una entrada para producir una salida.

Todos los algoritmos utilizados en este apartado pueden encontrarse en el Apéndice A.

3.1 Autómatas finitos

Como vimos en el apartado de algorítmos genéticos, un problema a resolver es como podemos representar nuestras soluciones. En este caso utilizaremos autómatas finitos[1]. Los autómatas finitos son modelos computacionales que realizan cómputos de forma automática para producir una salida.

La teoría de autómatas es el estudio de las máquinas abstractas, en particular las que poseen un número finito de estados. Es una rama de la informática y las matemáticas, con aplicaciones en la automatización de tareas como el análisis del lenguaje y el reconocimiento de patrones.

Estos modelos se pueden utilizar para reconocer lenguajes formales, que son conjuntos de cadenas de símbolos que siguen unas reglas de formación. Los lenguajes formales se clasifican según su complejidad en la jerarquía de Chomsky, que es una división jerárquica de cuatro niveles:

[1] Consultar *The coevolution of automata in the repeated prisioner's dilemma.* [4]

- Gramáticas regulares: generan los lenguajes que pueden ser reconocidos por un autómata finito. Son los lenguajes más simples y tienen una estructura lineal.
- Gramáticas libres de contexto: son los que pueden ser reconocidos por un autómata con pila. Son lenguajes que tienen una estructura jerárquica y que pueden tener recursividad.
- Gramáticas sensibles al contexto: los lenguajes que pueden ser reconocidos por un autómata linealmente acotado. Son aquellos lenguajes que tienen una estructura más compleja y que pueden depender del contexto en el que se encuentran las cadenas.
- Gramáticas sin restricciones: son los lenguajes que pueden ser reconocidos por una máquina de Turing, son los más generales y complejos, sin ninguna limitación en su escritura.

Tenemos una relación de inclusión entre ellos:

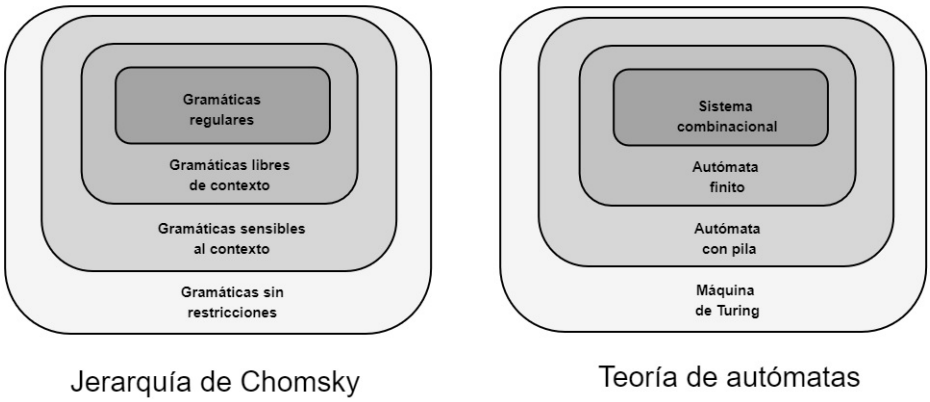


Figura 3.1: Esquemas de Jerarquía de Chomsky y de la Teoría de autómatas.

Nos centraremos en los autómatas finitos. Formalmente, un autómata finito puede definirse por 4-upla $(Q, S_0, \lambda, \delta)$ donde Q es un conjunto finito de estados internos, $S_0 \in Q$ es el estado inicial, $\lambda : Q \to S_l$ donde S_l es el movimiento del jugador en el siguiente periodo, y δ es la función de transición que envía el estado interno actual de la máquina y el movimiento reportado del oponente en un nuevo estado interno.

Para mayor simplicidad, veamos algún ejemplo típico de autómata finito.

Ejemplo 29. Autómatas finitos.

Generalmente los autómatas finitos se representan por grafos. Sin pérdida de generalidad, supongamos que nuestros autómatas pueden tomar dos decisiones cooperar (C) ó competir (D), algunos de los autómatas más simples son los siguientes:

- **Siempre C**: el autómata siempre coopera, independientemente de lo que haga su adversario. La figura muestra el autómata con un círculo para cada estado, donde se indica la acción que elige el autómata. Las flechas representan las acciones del adversario (que pueden ser C ó D) que provocan el cambio de estado.

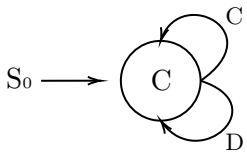


Figura 3.2: Siempre C.

Observamos que nuestro autómata elige cooperar sin importar la acción que haga el adversario.

- **Ojo por ojo**: es el autómata que coopera en la primera ronda y luego repite lo que hizo el otro jugador en la ronda anterior. Es una estrategia recíproca y equilibrada, que puede fomentar la cooperación o el castigo según el comportamiento del otro jugador.

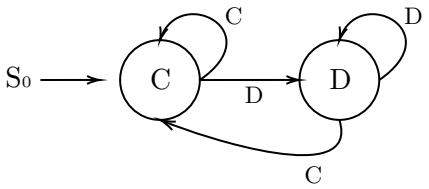


Figura 3.3: Ojo por ojo.

- **Castigar dos veces**: es el autómata que coopera en la primera ronda y luego compite dos veces seguidas si el otro jugador compitió en la ronda anterior. Es una estrategia recíproca y vengativa, que busca disuadir al otro jugador de competir mediante un castigo mayor.

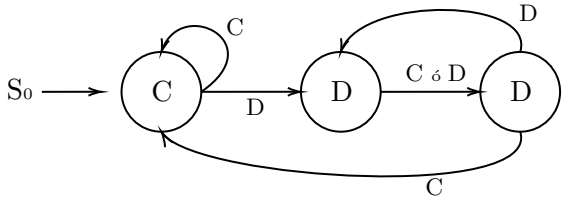


Figura 3.4: Castigar dos veces.

Con los autómatas finitos podemos modelar estrategias de diversa complejidad y variedad. Estos modelos son sencillos pero eficaces para representar a los agentes que usamos en Inteligencia Artificial en muchos problemas. A continuación, veremos cómo diseñar e implementar estos autómatas para que nuestros agentes aprendan a jugar al dilema del prisionero repetido y a una guerra de precios repetida.

3.2 Dilema del prisionero repetido con coevolución de autómatas

Recordamos un posible enunciado para el dilema del prisionero, que ya introdujimos en el Ejemplo 5:

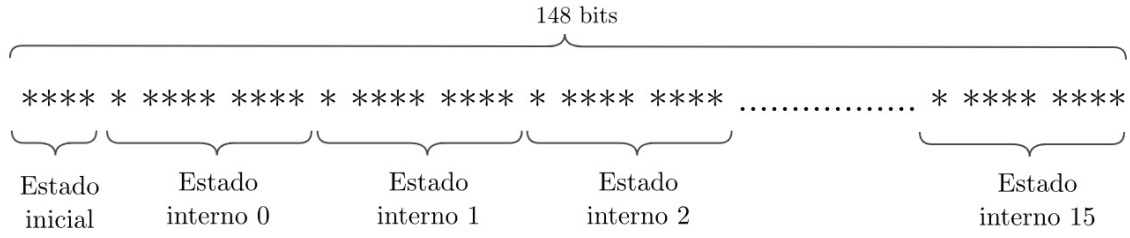
Dos jóvenes sospechosos de un crimen son arrestados y llevados a una comisaría. El fiscal les ofrece un trato a cada uno por separado: si confiesa y delata a su cómplice, quedará libre y el otro recibirá un condena de 5 años; si no confiesa y el otro lo hace, rescibirá una condena de 5 años y el otro quedará libre; si ambos confiesan, recibirán una condena de 4 años cada uno por buena conducta; si ninguno confiesa, recibirán una condena de 1 año cada uno por un delito menor del que se tienen pruebas suficientes. Los sospechosos no pueden comunicarse entre ellos y deben tomar su decisión sin saber lo que realizará el otro. ¿Qué deben hacer los sospechosos?

La matriz de pagos para este juego (tomando números positivos) será:

		Joven 2	
		Callar	Confesar
Joven 1	Callar	(4,4)	(0,5)
	Confesar	(5,0)	(1,1)

Cuadro 3.1: Dilema del prisionero en forma normal con valores positivos.

Nuestro propósito es analizar la evolución de las estrategias de nuestros agentes al participar de forma recurrente en este juego mediante un algoritmo genético. Los individuos de nuestra población se modelarán mediante autómatas finitos expresados como cadenas de bits (0 y 1). Los autómatas tendrán 148 bits separados en 1 estado inicial y 16 estados internos, que almacenarán la información de las decisiones de nuestros autómatas en su turno de juego. Tendrán la siguiente forma:



Seguiremos la notación introducida en el apartado sobre algoritmos genéticos, es decir, $*$ corresponde a un 0 ó a un 1. Veamos como funciona cada estado.

- El estado inicial es una subcadena de 4 caracteres, esta subcadena corresponde con un número natural en su forma binaria, tendremos desde el número 0 (la cadena 0000) hasta el número 15 (cadena 1111). Este estado inicial indica hacia qué estado debemos movernos en el primer turno.
- Cada estado interno consta de 3 subcadenas. El bit inicial determina la acción de nuestro autómata, siendo 0 para Callar y 1 para Confesar. Los 8 bits restantes se dividen en dos partes: los primeros cuatro bits indican (en binario) el estado al que transita nuestro autómata si el adversario elige Callar; los últimos cuatro bits señalan el estado siguiente si el adversario opta por Confesar.

Por ejemplo, consideremos el siguiente autómata:

0010 0 1011 0011 0 0001 1101 1 0001 0010 0 1101 1 1111 0000 ..... 1 0011 0101

El estado inicial es 0010, que equivale al número 2 en binario, por lo que pasamos al estado interno 2. Este estado es 1 0001 0010. Nuestro autómata optará por Confesar y transitará al estado correspondiente según la elección del adversario. Observemos que si el adversario elige Callar, nuestro autómata se desplaza al estado 1 (cuya representación en binario es 0001), donde también seleccionará Callar; sin embargo, si el adversario prefiere Confesar, será sancionado por nuestro autómata, que regresará al estado 2 en el que estaba, para volver a escoger Confesar.

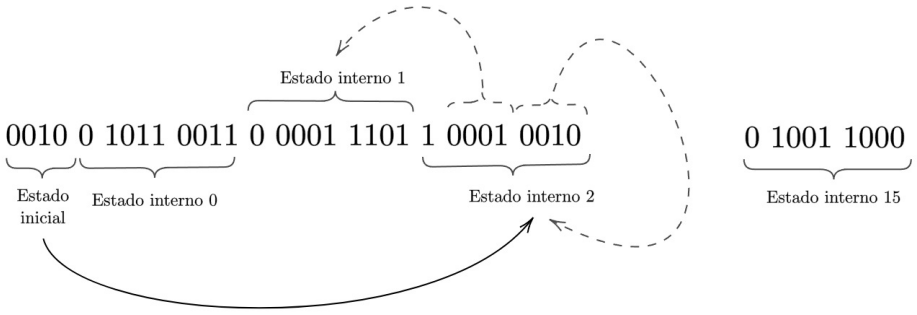


Figura 3.5: Primer movimiendo de un autómata finito. El estado inicial nos lleva al estado interno 2, dependiendo de la opción escogida por el adversario nos moveremos al estado interno 1 ó al estado interno 2 de nuevo.

De esta manera, dispondríamos de una forma de representar a nuestros agentes mediante autómatas finitos que pueden adaptarse a las decisiones del adversario y participar de forma recurrente en el juego del dilema del prisionero.

3.2.1 Metodología

Nuestro objetivo es analizar la evolución de las estrategias de nuestros agentes a través de varias generaciones para hallar una aproximación a la solución óptima del juego. Para ello, simularemos una serie de enfrentamientos entre cada individuo de una población y el resto de los individuos, incluyéndose a sí mismo. En cada enfrentamiento, asignaremos una puntuación al individuo según la matriz de pagos anterior, que refleja los beneficios y costes de cada acción posible.

El algoritmo que utilizaremos para simular la evolución de las estrategias de nuestros agentes será desarrollado en el lenguaje de programación Python. En las siguientes páginas se comentarán algunos extractos del mismo que ilustran las partes más relevantes del algoritmo y emplearemos el mismo nombre de las variables que se usan en el código para referirnos a los conceptos que representan.

Trabajaremos con los siguientes datos base:

- **AUTOMATOM_LENGTH**. Estos serán la cantidad de bits de cada autómata, en este problema el valor siempre será de 148.
- **NUM_AUTOMATA**. Utilizaremos esta variable para almacenar el número de autómatas que crearemos. Iremos variando esta cantidad para analizar el algoritmo con distintos datos de entrada.
- **NUM_MATCHES**. Esta cantidad representa el número de repeticiones del juego entre dos autómatas.
- **NUM_CHILDREN**. Será el número de hijos que crearemos en cada generación a partir de los mejores padres de cada una de ellas una vez finalizados todos los emparejamientos. Siempre será un número menor o igual a NUM_AUTOMATA y será un número par.
- **GENERATIONS**. En esta variable almacenaremos cuántas generaciones queremos realizar.
- **Pm**. Este será la probabilidad de mutación p_m.

En cada generación, cada autómata se enfrentará al resto de autómatas y a él mismo, sumando a cada uno la puntuación obtenida en cada caso. Con una elección ponderada de los autómatas, los cruzaremos y mutaremos para crear el número de hijos deseados. Estos hijos sustituirán a los peores autómatas, para volver a realizar el proceso en la siguiente generación.

Cada autómata será una instancia de la clase Automata, en la que almacenaremos información y métodos útiles para cada uno de ellos. En esta clase se guardará la estructura de cada autómata (las cadenas de bits) y la puntuación que vaya acumulando. Además, dispondremos de algunos procesos internos, que serán los siguientes:

- *search_position*. En el que calcularemos la nueva posición a la que deberá moverse nuestro autómata dependiendo de la decisión tomada por el adversario.
- *binary_to_decimal*. Utilizaremos esta función para pasar de binario a decimal, lo utilizaremos para calcular la posición a la que debemos movernos.
- *search_state*. Con esta función nos movemos hasta la cadena de 9 caracteres del estado interno que deseamos.
- *reset*. Esta función la utilizaremos para ir directamente al primer estado marcado por el estado inicial en cada nueva generación.

Para evaluar los resultados, consideraremos la frecuencia con la que nuestros estados eligen cada acción. Observaremos tanto las decisiones que adopta el autómata que estamos analizando frente al resto de los individuos, como las decisiones que toman los demás adversarios en cada enfrentamiento.

Para crear nuestros primeros autómatas haremos uso de la función *create_automata*, que generará todos los autómatas con puntuación nula y estructura aleatoria.

Para el cruce de nuestros autómatas aplicaremos el proceso descrito anteriormente. Seleccionaremos dos individuos con una probabilidad proporcional a su puntuación obtenida tras haberse enfrentado con toda la población. Los individuos con mayor puntuación

tendrán más posibilidades de ser elegidos para ser cruzados. Se escogerán 2 individuos y un número aleatorio para el punto de corte de las cadenas, para combinar la primera parte del primer padre con la segunda del segundo y viceversa, generando así dos hijos.

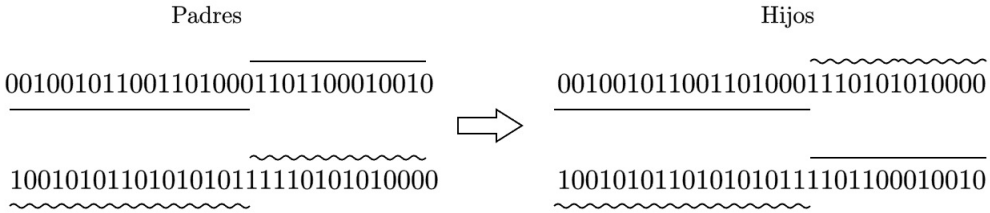


Figura 3.6: Proceso de cruce con dos autómatas de 31 bits.

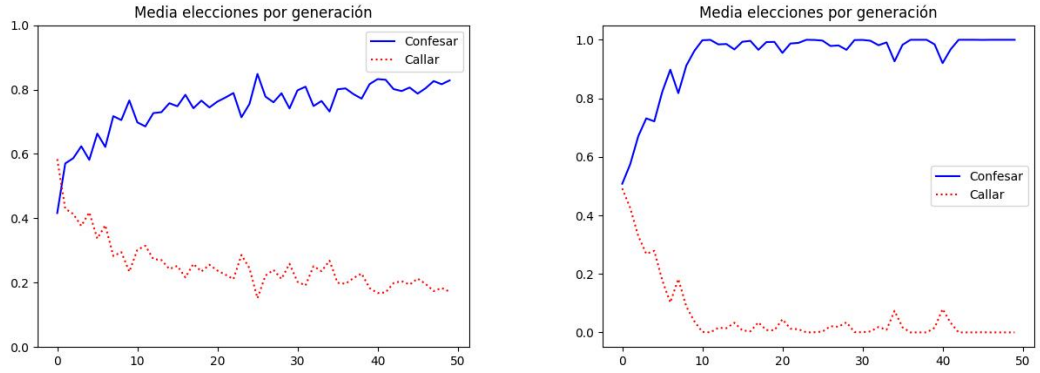
Como indicamos en la sección de algoritmos genéticos, la mutación puede implementarse de distintas formas, nosotros realizaremos pruebas con algunos tipos de mutación para comparar los resultados dados de cada una de ellas. Estas mutaciones se corresponden con las funciones *mutation_1* y *mutation_2*. En la primera mutación, recorremos toda la cadena de nuestro hijo alterando el valor de cada bit según la probabilidad de mutación que establezcamos. En cambio, la segunda mutación no tiene en cuenta este valor de probabilidad y es una mutación mucho más rara y lenta, en ella modificamos aleatoriamente 1 bit de toda la cadena. Esperamos que la primera mutación sea mejor que la segunda.

3.2.2 Evaluación de los resultados

Para comprender mejor el funcionamiento de nuestro algoritmo, realizaremos diferentes pruebas con distintos valores de los parámetros y métodos escogidos. Así podremos observar cómo cambia el comportamiento y el rendimiento del algoritmo según las condiciones que le imponemos. De esta forma, podremos extraer conclusiones sobre las ventajas y desventajas de cada opción.

Mutación

Consideraremos una población de 30 individuos que se enfrentarán entre sí 150 veces en 50 generaciones. Tomaremos un número de hijos de 20 y una probabilidad de mutación de 0.1. Veamos la salida del algoritmo utilizando ambos procesos de mutación:

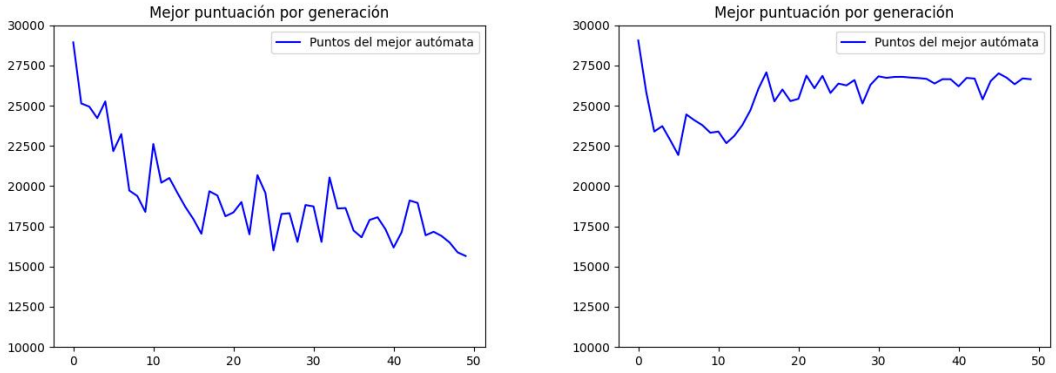


(a) *mutation_1* (b) *mutation_2*

Figura 3.7: Simulación de decisiones utilizando las dos posibles mutaciones con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50, Pm = 0.1.

Observamos que en ambos casos, nuestros autómatas se inclinan notablemente por la opción Confesar, que coincide con el equilibrio de Nash de la matriz de pagos, esto es bueno porque indica que nuestros agentes han aprendido a jugar de forma racional y óptima, maximizando su beneficio esperado y minimizando su riesgo. Esto implica que nuestros agentes han desarrollado una estrategia adaptativa que les permite reaccionar ante las decisiones del adversario y anticiparse a sus movimientos, buscando siempre la mejor respuesta posible. Así, nuestros agentes han logrado aproximarse a la solución del juego del dilema del prisionero repetido mediante el algoritmo genético.

Pero, ¿qué tipo de mutación es más conveniente? Podríamos realizar el experimento cambiando los datos, pero no hace falta. Si analizamos algunos de los autómatas con mayor puntuación de cada generación podremos ver qué opción es mejor. Obtenemos el siguiente resultado según los puntos acumulados:



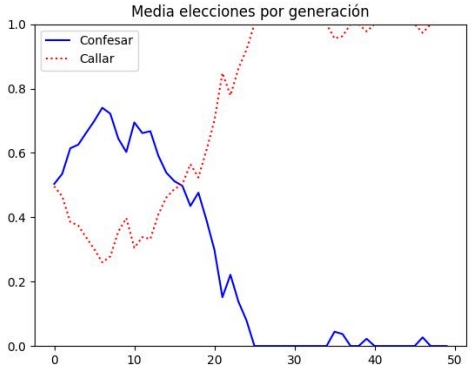
(a) *mutation_1* (b) *mutation_2*

Figura 3.8: Simulación de puntuación utilizando las dos posibles mutaciones con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50, Pm = 0.1.

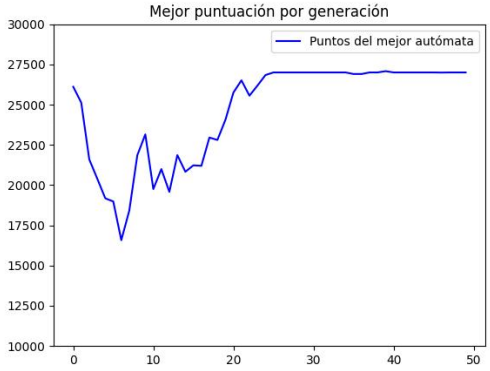
Podemos apreciar cómo cambia mucho más la primera mutación, en la que tenemos más probabilidad de alterar los bits de la cadena, explorando más opciones en el espacio de búsqueda. Esto nos permite encontrar soluciones más diversas y adaptativas, que pueden mejorar el rendimiento de nuestros agentes. En cambio, la segunda mutación es más conservadora y lenta, y puede quedarse atrapada en óptimos locales.

Comprobamos que la primera mutación es mucho mejor tanto para explorar el espacio de búsqueda, como para conseguir puntuaciones más altas. A partir de ahora, excepto que se especifique lo contrario, emplearemos únicamente la primera mutación, que recorre toda la cadena de bits alterando su valor según la probabilidad de mutación fijada. De todas formas, ambas mutaciones son correctas según el problema a resolver.

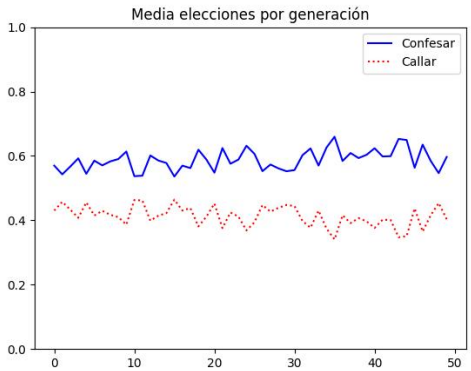
La mutación puede introducir diversidad en la población y explorar nuevas soluciones, pero también puede destruir buenas soluciones y alejarse del óptimo. Por lo tanto, es interesante analizar cómo la probabilidad de mutación afecta a los resultados obtenidos. Veamos qué sucede con una probabilidad demasiado alta y demasiado baja.



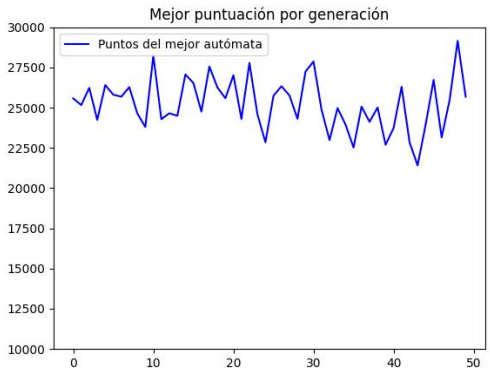
(a) Pm=0.001



(b) Pm=0.001



(c) Pm=0.65



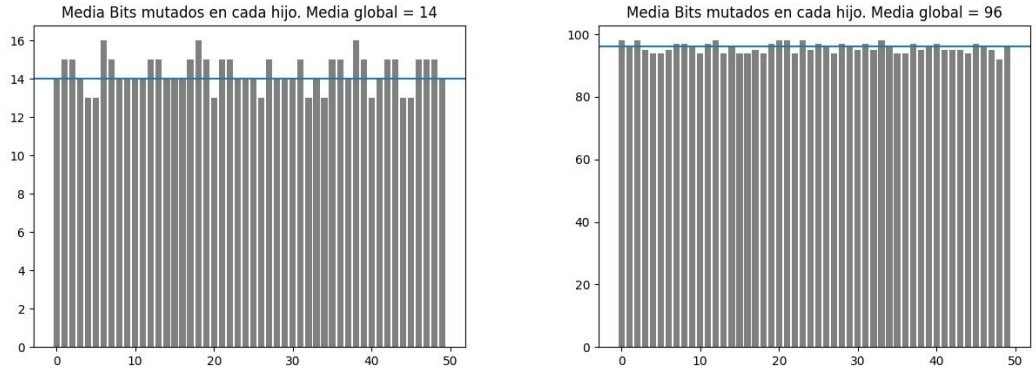
(d) Pm=0.65

Figura 3.9: Simulación utilizando dos posibles valores de la probabilidad de mutación con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50.

Con una probabilidad de mutación baja, el algoritmo se estanca en soluciones subóptimas, que no maximizan el beneficio de los individuos. Es más, el algoritmo no converge

hacia la solución esperada y la calidad de la solución no mejora. Por el contrario, con una probabilidad de mutación alta, el algoritmo puede encontrar soluciones mejores, que aumentan el beneficio de los individuos. Sin embargo, el algoritmo no alcanza el equilibrio de Nash, que es la estrategia óptima para ambos jugadores. En su lugar, el algoritmo se aproxima a una solución mixta, donde los individuos eligen entre Callar y Confesar con prácticamente igual probabilidad.

A primera vista, una probabilidad alta puede parecer más ventajosa, ya que la solución obtenida tiene un mayor beneficio y los individuos siguen eligiendo mayoritariamente la estrategia del equilibrio de Nash. Sin embargo, hay que tener en cuenta el coste de la mutación, que es el número de cambios que se producen en los genes de los individuos. Una mutación excesiva puede provocar una pérdida de información y una inestabilidad en la solución. Por eso, es conveniente comparar el nivel de mutación entre una probabilidad adecuada y una alta, para ver cómo afecta a los resultados del algoritmo.



(a) Pm=0.1 (b) Pm=0.65

Figura 3.10: Cantidad de mutaciones medias por generación. Simulación utilizando dos posibles valores de la probabilidad de mutación con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50.

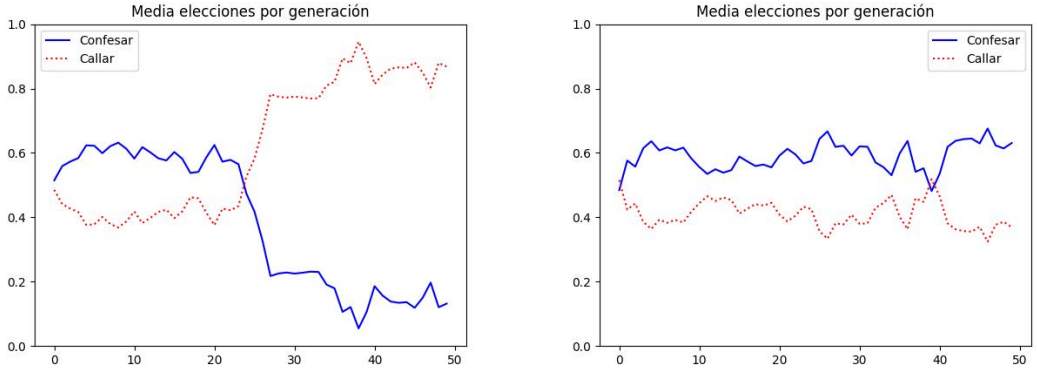
La diferencia entre el nivel de mutación de los dos casos es muy significativa. En el primer caso, con una probabilidad adecuada, el número medio de bits que cambian por hijo es de 14, lo que representa un 10 % de la cadena. En el segundo caso, con una probabilidad alta, el número medio de bits que cambian por hijo es de 96, lo que equivale a un 65 % de la cadena. Mutar tantos bits por hijo implica añadir mucha variabilidad y ruido a la solución, lo que dificulta la convergencia y la estabilidad del algoritmo. Además, se pierde la información útil que se ha acumulado en las generaciones anteriores, lo que reduce la calidad de la solución. Por lo tanto, una probabilidad de mutación alta no es recomendable para los algoritmos genéticos. De aquí en adelante fijaremos el valor 0.1 para la probabilidad de mutación excepto en los casos en los que se especifique lo contrario.

Generaciones y población

Otro aspecto importante que debemos considerar es el tamaño de la población, el número de hijos y el número de generaciones que usamos en el algoritmo genético. Estos parámetros determinan el equilibrio entre la exploración y la explotación de las soluciones del proble-

ma. Si tenemos una población grande pero pocos hijos, no podremos generar suficiente diversidad para buscar nuevas soluciones. Por otro lado, si tenemos pocas generaciones, no tendremos tiempo suficiente para que los individuos se adapten a la solución óptima.

Comparemos el efecto de tener una población con una gran diferencia entre el número de individuos e hijos y otra con una pequeña diferencia. Por ejemplo, una población de 40 individuos y 8 hijos frente a una población de 40 individuos y 38 hijos. En la primera población, habrá poca competencia entre los hijos para reemplazar a los padres, lo que provocará una alta presión selectiva y una rápida convergencia (posiblemente a valores erróneos). En la segunda población, habrá mucha competencia entre los hijos para reemplazar a los padres, lo que generará una baja presión selectiva y una mayor diversidad.



(a) NUM_CHILDREN=8 (b) NUM_CHILDREN=38

Figura 3.11: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=40. NUM_MATCHES = 150, GENERATIONS = 50, Pm=0.1.

Observamos que en una población con pocos hijos, los individuos se estancan en una solución errónea porque no exploran suficientemente el espacio de búsqueda y se quedan atrapados en un óptimo local. Por el contrario, en una población con muchos hijos, vemos que se conserva la tendencia a la solución correcta, pero con mucha variabilidad y no se alcanza claramente la solución óptima porque hay demasiado ruido y pérdida de información en cada generación. Esto se debe a que al generar muchos hijos, se produce una alta tasa de mutación, que introduce cambios aleatorios en los genes de los individuos. Estos cambios pueden destruir las buenas soluciones que se han encontrado en las generaciones anteriores y alejar al algoritmo del óptimo global.

El inconveniente de tener pocos hijos no se puede resolver simplemente incrementando el número de generaciones, ya que el algoritmo se queda atrapado en una solución local y no puede escapar de ella. Al generar pocos hijos, se reduce la diversidad en la población y se limita la capacidad de explorar nuevas soluciones. Por lo tanto, el algoritmo se vuelve muy dependiente de la solución inicial y no puede mejorarla significativamente.

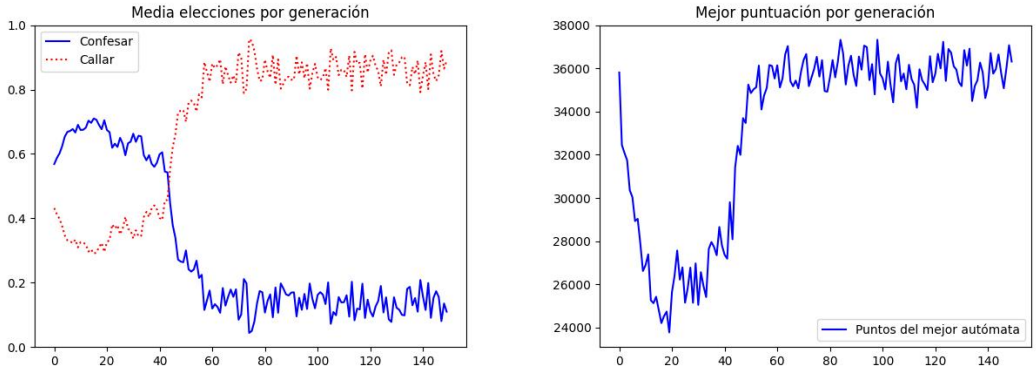


Figura 3.12: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=40. NUM_MATCHES = 150, NUM_CHILDREN=8, GENERATIONS = 150, Pm=0.1.

Esquemas

Para analizar el papel de los esquemas en nuestro ejemplo, queremos observar cómo se van transmitiendo estos patrones de bits en los individuos más aptos. Para ello, seguiremos la evolución del mejor autómata (con mayor beneficio) de cada generación. Mostraremos por pantalla el mejor autómata de cada generación junto con su beneficio obtenido, pero en lugar de mostrar los 148 bits, por cuestión de espacio omitiremos algunos estados de la cadena.

Realizando una simulación sobre 30 generaciones e imprimiendo los datos por pantalla obtenemos los siguientes resultados:

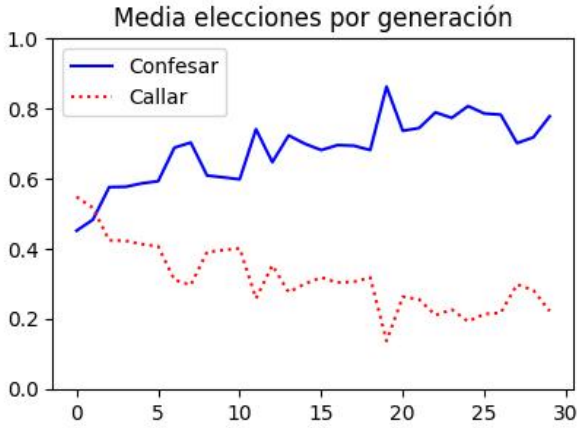


Figura 3.13: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=20. NUM_MATCHES = 150, NUM_CHILDREN=16, GENERATIONS = 30, Pm=0.1.

```
00- 0100 0 0011 0110 1 1010 1001 1 1101 0010 0 1011 0010 ... 1 1000 1000 0 1010 0110 1 0101 1011 1 1111 1011
01- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
02- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
03- 1101 1 1001 0101 1 1110 0110 0 1100 1110 1 0110 0101 ... 1 0001 1001 0 1001 0010 0 1111 1000 0 0011 1011
04- 1101 1 1001 0101 1 1110 0110 0 1100 1110 1 0110 0101 ... 1 0001 1001 0 1001 0010 0 1111 1000 0 0011 1011
05- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
06- 1101 1 1001 0101 1 1110 0110 0 1100 1110 1 0110 0101 ... 1 0001 1001 0 1001 0010 0 1111 1000 0 0011 1011
07- 1101 1 1001 0101 1 1110 0110 0 1100 1110 1 0110 0101 ... 1 0001 1001 0 1001 0010 0 1111 1000 0 0011 1011
08- 1101 1 1001 0101 1 1110 0110 0 1101 1110 1 0110 0101 ... 0 0111 0001 1 1101 0101 1 0010 0110 0 0100 1010
09- 1100 1 1001 0101 0 1110 0110 0 1100 1110 1 0110 0101 ... 1 0000 0101 0 0011 1010 0 1101 0000 0 0011 1011
10- 1101 1 1001 0101 1 1110 0110 0 1100 1110 1 0110 0101 ... 1 0001 1001 0 1001 0010 0 1111 1000 0 0011 1011
11- 1100 1 1001 0101 0 1110 0110 0 1100 1110 1 0110 0101 ... 1 0000 0101 0 0011 1010 0 1101 0000 0 0011 1011
```

```
12- 1000 1 0111 1111 0 1110 0010 0 1000 1110 1 0111 0110 ... 1 1111 0101 1 1101 1100 1 0110 1100 1 0000 0100
13- 1101 1 0001 1001 1 1110 0100 1 1000 1110 1 0111 1101 ... 1 0000 0001 0 0011 1010 1 1001 1110 0 0001 1011
14- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
15- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
16- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
17- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
18- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
19- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
20- 0001 1 1011 1101 1 0100 0011 0 1010 0110 1 0110 0101 ... 1 0000 1001 0 0011 1110 1 1101 0000 0 0011 1011
21- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
22- 0101 1 1001 1101 1 1100 0011 0 1110 0110 1 0110 0101 ... 1 0000 0001 0 0011 1010 1 1101 0000 0 0001 1011
23- 1100 1 1001 0101 0 1110 0110 0 1100 1110 1 0110 0101 ... 1 0000 0101 0 0011 1010 0 1101 0000 0 0011 1011
24- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
25- 1100 1 1001 0101 0 1110 0110 0 1100 1110 1 0110 0101 ... 1 0000 0101 0 0011 1010 0 1101 0000 0 0011 1011
26- 1100 1 1001 0101 0 1110 0110 0 1100 1110 1 0110 0101 ... 1 0000 0101 0 0011 1010 0 1101 0000 0 0011 1011
27- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
28- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
29- 1111 1 0001 1101 1 1110 0110 0 1100 1110 1 0111 0111 ... 0 0110 0111 1 1101 0001 1 0110 1110 1 0000 0110
```

En el proceso podemos observar cómo algunos patrones o esquemas se mantienen o se modifican a lo largo de las generaciones. Un ejemplo de esto lo encontramos en la tercera y cuarta columna de la tabla, que representa el estado al que se trasladará nuestro autómata si estamos en el estado interno 0. En estas columnas, podemos ver cómo surge el esquema $**11\ \ *1**$, este esquema desaparece en la siguiente generación para reaparecer modificado seguidamente, mostrando su utilidad temporal. Este esquema $***1\ \ ***1$ parece ser estable, pues sigue vigente durante todas las genraciones restantes.

Si volvemos a realizar el experimento con NUM_AUTOMATA = 30 y NUM_CHILDREN = 20, observamos que también existen estos patrones en los mejores autómatas tras finalizar todas las generaciones.

```
Últimos 10 mejores automatas tras 150 partidas entre ellos:
Automatom: 000001000011101 ... 101010110000100111   -   Points: 32742
Automatom: 010001010100001 ... 111111000000100100   -   Points: 32204
Automatom: 010101010100101 ... 101010110000100111   -   Points: 32204
Automatom: 010001101101001 ... 010000110101011011   -   Points: 32204
Automatom: 010111011101101 ... 000101010011000001   -   Points: 32204
Automatom: 010101011101110 ... 101100110011010011   -   Points: 32204
Automatom: 010001111101101 ... 111111000100100101   -   Points: 32204
Automatom: 010101011000110 ... 111000110111011111   -   Points: 32204
Automatom: 010011001101001 ... 111111001100011101   -   Points: 32204
Automatom: 010101010100101 ... 110100110010000011   -   Points: 32204
```

Equilibrio de Nash

Hemos comprobado que el algoritmo genético produce resultados satisfactorios con los parámetros que hemos elegido, pero ¿qué ocurriría si cambiamos las condiciones del juego? Para explorar esta cuestión, vamos a modificar los valores de las recompensas y las penalizaciones que reciben los agentes según sus acciones. Así, podremos observar cómo se altera el equilibrio de Nash y cómo se adaptan los agentes a las nuevas circunstancias.

Para empezar, vamos a ver qué pasa si el equilibrio de Nash se sitúa en otra posición. Para lograr esto, solo tenemos que intercambiar los números en la matriz de pagos, sin prestar atención al contexto del problema, ya que no se ajusta a la siguiente matriz de pagos. Supongamos que la matriz de pagos correspondiente es la siguiente:

Nuestros agentes aprenden cuál es la mejor opción y tienden a la solución esperada.

Pero, ¿qué sucede si aumentamos la puntuación del equilibrio de Nash? Cambiemos el pago (1,1) de la opción Callar-Callar de este nuevo problema por el pago (3,3). Una simulación nos da el siguiente resultado del Cuadro 3.2. :

		Joven 2	
		Callar	Confesar
Joven 1	Callar	(1,1)	(5,0)
	Confesar	(0,5)	(4,4)

Cuadro 3.2: Matriz de pagos del dilema del prisionero modificada.

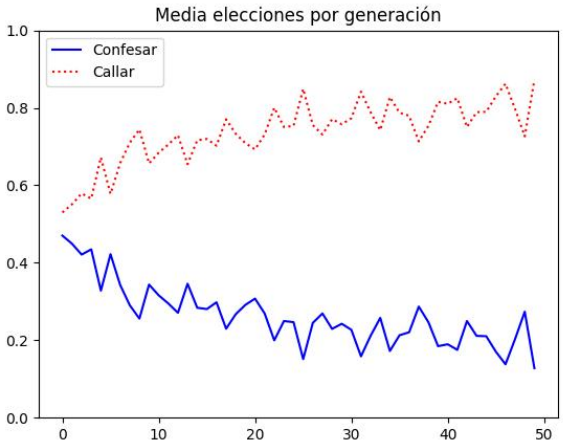


Figura 3.14: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_MATCHES = 150, NUM_CHILDREN=20, GENERATIONS = 50, Pm=0.1.

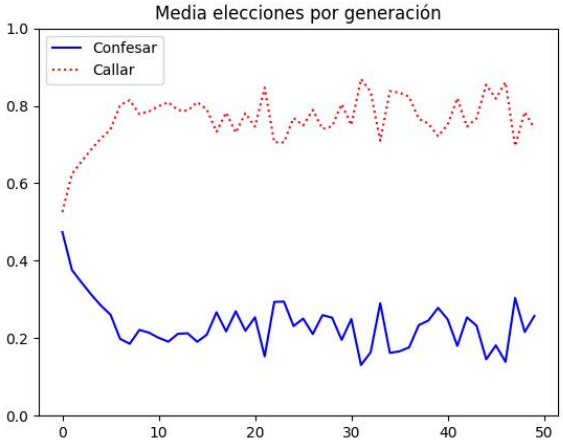


Figura 3.15: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_MATCHES = 150, NUM_CHILDREN=20, GENERATIONS = 50, Pm=0.1.

Los autómatas llegan más rápido a la solución del problema, lo cual tiene sentido, ya que cuanto mayor sea el beneficio por alcanzar este equilibrio, más se esforzarán por adaptarse a esta respuesta. Esto se hace más evidente si incrementamos el número de ge-

neraciones, porque así los autómatas tienen más oportunidades de mejorar sus estrategias y eliminar las menos eficientes.

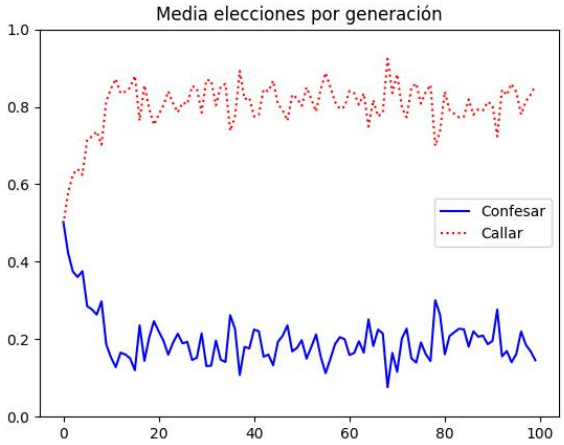
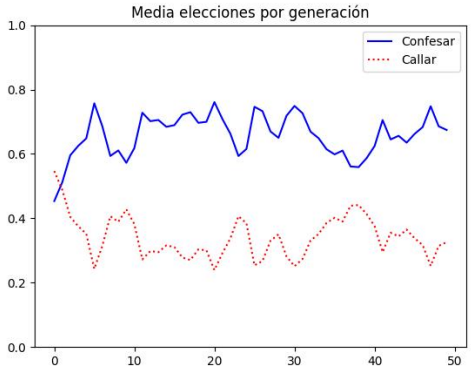


Figura 3.16: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_MATCHES = 150, NUM_CHILDREN=20, GENERATIONS = 100, Pm=0.1.

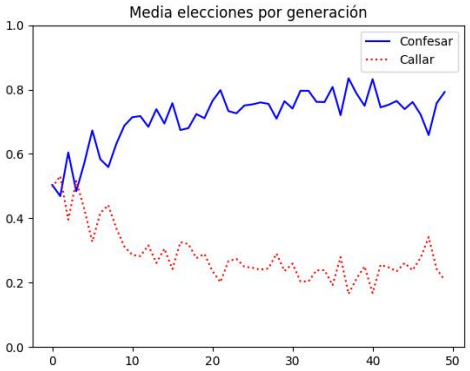
Número de repeticiones

Hemos experimentado con todos los datos iniciales, incluso con los del propio problema, para analizar el comportamiento de nuestros autómatas. El único factor que nos queda por explorar es el número de repeticiones. A diferencia de los demás datos, el número de veces que los autómatas se enfrentan entre ellos no influye tanto en el resultado porque no afecta a la estructura del juego ni a las preferencias de los agentes. Lo que sí puede influir es la variabilidad de los resultados, ya que a mayor número de repeticiones, menor será la probabilidad de que ocurran desviaciones aleatorias del equilibrio de Nash, pero peor será el tiempo de ejecución. Comparemos los resultados y el tiempo necesario[2] en dos casos con distinto número de repetición.

[2]El programa es ejecutado en una máquina procesador AMD Ryzen 5 5500U de 2.10 GHz, 6 núcleos y 12 hilos, RAM instalada de16.0 GB y 3200MHz. El sistema oprativo es Windows 10 Pro 64 bits, ejecutando el programa con Visual Studio 2022.



(a) NUM_MATCHES = 150



(b) NUM_MATCHES = 250

Figura 3.17: Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_CHILDREN=20, GENERATIONS = 150, Pm=0.1.

Ejecutando el programa 5 veces con cada número de repeticiones e imprimiendo por pantalla el mejor autómata de cada generación, se obtiene una media de 30,51889594395955 segundos para 150 repeticiones y 51,55614197254181 segundos para 250 repeticiones.

3.3 Programa sobre el oligopolio de Bertrand

En este apartado vamos a desarrollar un algoritmo similar al del caso anterior, pero aplicado a un problema distinto. Se trata de un ejemplo de un oligopolio de Bertrand discreto, un modelo de competencia en el que las empresas producen productos homogéneos y compiten a través de establecer precios simultáneamente, sin cooperar entre ellas. Los consumidores compran todo de la empresa con el precio más bajo. Este modelo fue tratado matemáticamente en el primer capítulo. Nuestro objetivo es utilizar los algoritmos genéticos para encontrar el equilibrio de forma eficiente. En este nuevo caso, el juego no es de suma cero y la matriz de pagos no será cuadrada.

3.3.1 Descripción del juego y algoritmo

Un posible enunciado del problema puede ser:

Burguer Donald SL y McKing SA son dos restaurantes rivales que se ubican en la misma calle. Ambos ofrecen una hamburguesa básica con distintos precios. Sin embargo, han contratado al mismo contable, un recién graduado en ADE, que les ha propuesto casi la misma carta de precios por pereza. Las dos empresas pueden elegir entre 3 precios: bajo, medio y alto. Pero como McKing SA le ofrece teletrabajar, el contable también le ha ofrecido la opción de poner un precio que no tiene la competencia, un ofertón. Los costes de producción son iguales para las dos empresas y los consumidores eligen el producto más económico. Las dos empresas quieren obtener el máximo beneficio a lo largo del año, lo que resulta en la siguiente matriz de pagos diaria:

		McKing SA			
		Ofertón	Bajo	Medio	Alto
Burguer Donald SL	Bajo	(5,<u>25</u>)	(10,10)	(15,5)	(20,0)
	Medio	(10<u>20</u>)	(15,15)	(20,10)	(25,5)
	Alto	(<u>15</u>,<u>15</u>)	(<u>20</u>,10)	(<u>25</u>,5)	(<u>30</u>,0)

Cuadro 3.3: Matriz de pagos del ejemplo de Bertrand.

En la tabla se ha identificado el equilibrio de Nash, siguiendo el criterio y notación que se expuso en el primer capítulo. Recordemos que el equilibrio de Nash es una situación en la que ningún jugador tiene incentivos para desviarse de su estrategia, dado que los demás jugadores no cambian la suya.

3.3.2 Metodología

En este caso, tenemos que modificar la longitud de los autómatas para adaptarlos al nuevo problema. Ahora las cadenas tendrán 292 bits en lugar de 148 bits. La estructura de los autómatas será similar a la anterior, con un estado inicial y 16 estados independientes. Sin embargo, cada estado interno tendrá 2 bits iniciales (en vez de 1) que indicarán la acción tomada por nuestro autómata. Además, cada estado interno tendrá 4 subcadenas de 4 bits cada una, que representarán el estado al que nos trasladaremos según la acción de nuestro adversario. De esta forma, podremos modelar las diferentes situaciones que se pueden dar en el juego.

$$\underbrace{1011}_{\text{Estado inicial}}\ \underbrace{01\ 1100\ 0011\ 1011\ 1010}_{\text{Estado interno 0}}$$

Según el enunciado del problema, el número de repeticiones que se realizarán en el juego será de 365, pero podríamos usar otro número similar para entrenar a nuestros autómatas, ya que hemos comprobado que este factor no influye tanto en el resultado como el cambiar el pago o los valores iniciales. Lo importante es que el número de repeticiones sea suficientemente grande como para que los autómatas puedan aprender y adaptarse a las estrategias de sus adversarios.

Como la matriz no es cuadrada, podría ocurrir que al enfrentar nuestros autómatas, el que se enfrenta con el resto (que usa los pagos de Burguer Donald SL) podría tener la opción disponible de Ofertón, ya que cada estado tiene 4 subcadenas. Este problema se puede resolver de varias formas, algunas de las que vamos a emplear son:

1. Penalizamos negativamente a nuestro autómata con un valor negativo en la puntuación y premiamos al adversario con una puntuación positiva, con el objetivo de que nuestro autómata evite escoger esta opción no disponible. De esta forma, pretendemos que nuestro autómata aprenda a adaptarse a las restricciones del problema.

2. Ignoramos este caso. No otorgamos ninguna puntuación a ningún autómata si alguno de los dos escoge una opción no disponible.

3. Asignamos aleatoriamente alguna de las otras tres puntuaciones disponibles al autómata que ha escogido la opción no válida. Esto equivale a hacerle cambiar forzosamente y de forma aleatoria a otra opción. De esta manera, evitamos que el autómata se quede estancado en una opción que no le corresponde.

Nuestro análisis consistirá en contrastar estas tres alternativas manteniendo constantes los valores iniciales del problema.

3.3.3 Evaluación de los resultados

Primera opción

Iniciemos con el algoritmo implementando la primera opción, que consiste en sancionar negativamente a los autómatas que optan por el ofertón cuando no está dentro de sus posibilidades. Los resultados que logramos con esta opción son los siguientes:

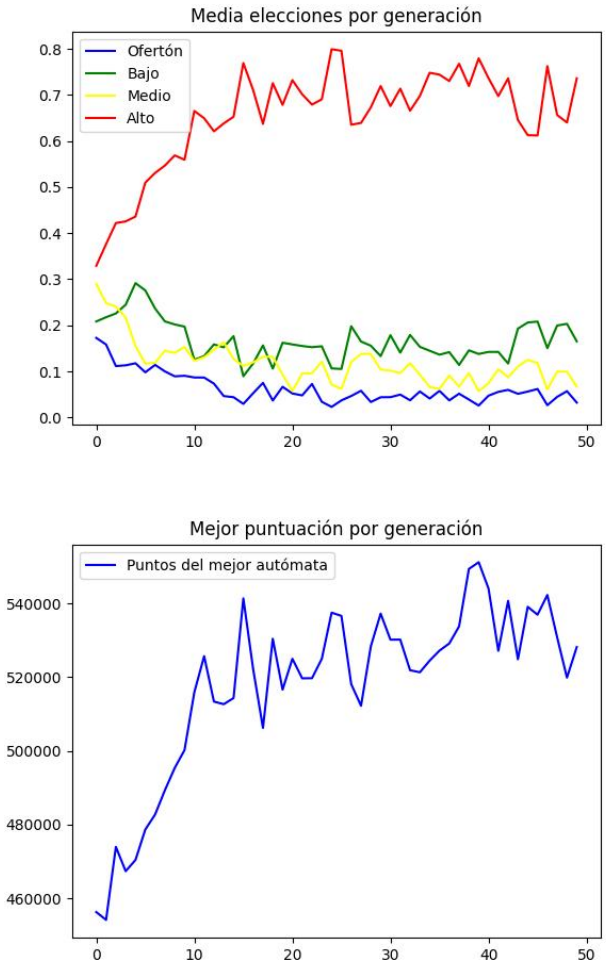


Figura 3.18: Simulación de la primera opción con: NUM_AUTOMATA=50. NUM_MATCHES = 365, NUM_CHILDREN=30, GENERATIONS = 150, Pm=0.1.

Podemos apreciar que la opción de Ofertón es la que menos usan todos los autómatas, ya que se les castiga cuando les corresponde ser enfrentados por el resto. Esto hace que

se modifique el equilibrio de Nash a otro punto, ya que se podría corresponder con un subjuego donde hemos eliminado la columna de Ofertón. El nuevo equilibrio de Nash en la submatriz sería la elección Alto-Bajo, que es la que maximiza el beneficio conjunto.

		McKing SA		
		Bajo	Medio	Alto
Burguer Donald SL	Bajo	(10,<u>10</u>)	(15,5)	(20,0)
	Medio	(15,<u>15</u>)	(20,10)	(25,5)
	Alto	(<u>20</u>,<u>10</u>)	(<u>25</u>,5)	(<u>30</u>,0)

Cuadro 3.4: Equilibrio de Nash en el subjuego.

Podemos notar que los individuos muestran una inclinación por la opción Alto, y como segunda opción prefieren el precio Bajo antes que el Medio, ya que les reporta una mayor ganancia.

Segunda opción

En la segunda opción no asignamos ninguna puntuación a ninguno de los dos autómatas si alguno de ellos elige una opción no disponible. De esta forma, ignoramos el resultado de esa partida y no afectamos al aprendizaje de los autómatas.

Podemos ver en la Figura 3.19 que los autómatas siguen mostrando una preferencia por elegir el valor del precio alto. Esto se debe a que estamos omitiendo muchos pagos del Ofertón, ignorando así la posibilidad de que sea elegido una gran cantidad de veces. Esta puede ser una de las peores opciones para encontrar resultados válidos y reales, ya que estamos dejando sin puntuar todas las veces que el autómata principal que se enfrenta con el resto elige la opción no válida. El autómata principal no aprende sobre esta decisión, ni tenemos en cuenta el resto de opciones que eligen los demás autómatas para enfrentarse a él en ese momento.

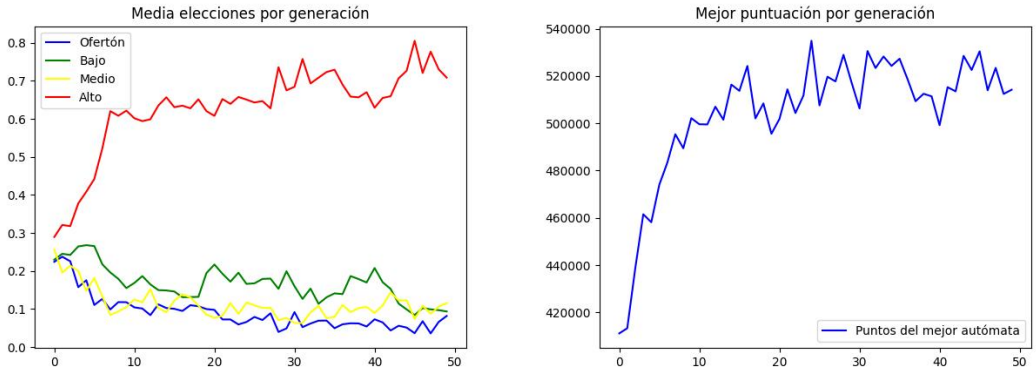


Figura 3.19: Simulación de la segunda opción con: NUM_AUTOMATA=50. NUM_MATCHES = 365, NUM_CHILDREN=30, GENERATIONS = 150, Pm=0.1.

Tercer caso

Finalizaremos con la tercera opción, si el autómata principal selecciona una opción que no es válida según las reglas del sistema, entonces se reemplazará automáticamente y de forma aleatoria por otra opción que sí sea válida entre las que tenga disponibles.

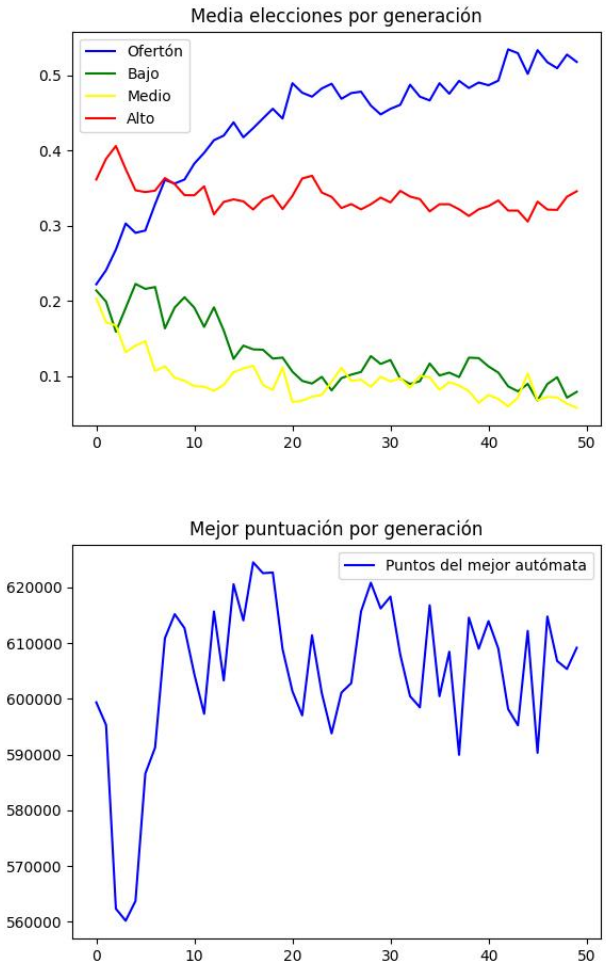


Figura 3.20: Simulación de la tercera opción con: NUM_AUTOMATA=50. NUM_MATCHES = 365, NUM_CHILDREN=30, GENERATIONS = 150, Pm=0.1.

De las tres opciones que hemos analizado, la tercera es la que ofrece mejores resultados tanto en términos de ganancia como de coherencia. Al aplicar esta opción, se incrementa la ganancia en un 13 % aproximadamente (pasando de tener una ganancia de aproximadamente 540.000 a entorno 610.000) y se obtiene una solución que se acerca al equilibrio de Nash del juego. Esto se debe a que los autómatas eligen preferentemente la opción de Ofertón, que es la más rentable, y en caso de no tenerla disponible, optan por la opción de Alto, que es la más defensiva. Así, se genera una dinámica de competencia entre las empresas que refleja la lógica del juego.

Bajo el supuesto de que ambos restaurantes logran vender todas sus hamburguesas, podemos concluir que la nueva oferta diseñada por el contable es una estrategia ventajosa para McKing SA (aunque tampoco perjudica demasiado a Burguer Donald SL).

Capítulo 4

Conclusiones

4.1 Resumen del trabajo

En este trabajo, se exploraron diferentes conceptos relacionados con la Teoría de Juegos y su aplicación en el campo de la Inteligencia Artificial. Se abordaron los fundamentos de la Teoría de Juegos, incluyendo juegos estáticos y dinámicos, así como la importancia de la información completa e incompleta en estos juegos. Además, se examinaron algoritmos de búsqueda y algoritmos de aprendizaje automático, como los algoritmos genéticos y el aprendizaje por refuerzo, destacando su relevancia en la resolución de juegos estratégicos.

Luego, se procedió a explorar la aplicación de la Inteligencia Artificial en la Teoría de Juegos, en particular, con los algorítmos de búsqueda. Se discutió cómo este enfoque puede utilizarse para maximizar la utilidad esperada en juegos estratégicos de suma cero.

Posteriormente, se presentó una aplicación específica de algoritmos genéticos a la Teoría de Juegos, utilizando autómatas finitos. Se exploró el dilema del prisionero repetido y se propuso una metodología basada en la coevolución de autómatas para encontrar estrategias óptimas. Se llevaron a cabo evaluaciones de resultados y se analizó cómo los parámetros de entrada afectan los resultados obtenidos.

Además, se desarrolló un programa de simulación relacionado con el oligopolio de Bertrand, abordando el desafío de la matriz de pagos no cuadrada. Se describieron tres variantes del programa que permiten superar este problema utilizando autómatas. Se evaluaron los resultados obtenidos y se compararon las distintas variantes para determinar cuál ofrece un mejor desempeño.

En resumen, este trabajo investigó y aplicó diferentes enfoques de la Inteligencia Artificial a la Teoría de Juegos, utilizando algoritmos genéticos y autómatas finitos. Se llevaron a cabo evaluaciones y comparaciones de resultados para analizar la eficacia de las estrategias propuestas. A continuación, en las conclusiones derivadas de los resultados, se presentarán las principales observaciones y hallazgos obtenidos en este estudio.

4.2 Conclusiones derivadas de los resultados

Tras analizar detenidamente los resultados obtenidos en este trabajo y considerando la sólida base teórica previamente comentada sobre la Teoría de Juegos, se pueden extraer

conclusiones significativas que arrojan luz sobre la aplicación de algoritmos genéticos y autómatas finitos en este campo.

En primer lugar, la aplicación de algoritmos genéticos en la coevolución de autómatas finitos ha demostrado ser una estrategia efectiva para encontrar estrategias óptimas en juegos estratégicos. Esta metodología permite que los autómatas finitos evolucionen y se adapten a lo largo del tiempo, explorando diferentes posibilidades y descubriendo estrategias más efectivas para maximizar las recompensas en el dilema del prisionero repetido. Estos resultados respaldan la viabilidad y la utilidad de los algoritmos genéticos en la resolución de problemas complejos en situaciones estratégicas.

En segundo lugar, el programa de simulación desarrollado para abordar el oligopolio de Bertrand y superar la matriz de pagos no cuadrada ha sido exitoso en ofrecer soluciones viables. Las tres variantes propuestas han demostrado ser eficaces para evitar esta limitación y proporcionan opciones alternativas para analizar el comportamiento estratégico en este escenario. Cada variante presenta ventajas y desventajas específicas, y la elección de la mejor opción dependerá de los objetivos y las características del juego en cuestión. Estos resultados respaldan la relevancia y la utilidad práctica de aplicar autómatas finitos en la resolución de juegos complejos y reales.

En resumen, los resultados obtenidos en este trabajo demuestran que la combinación de algoritmos genéticos y autómatas finitos es un enfoque efectivo y prometedor para abordar problemas estratégicos en la Teoría de Juegos. Al comentar la Teoría de Juegos previamente, se estableció un contexto sólido y se identificaron desafíos y oportunidades específicos. Esta base teórica fortalece la validez y el impacto de las conclusiones, resaltando la importancia de la teoría en la aplicación de la Inteligencia Artificial en la Teoría de Juegos. Estos hallazgos contribuyen al avance de la comprensión y aplicación de la Inteligencia Artificial en la Teoría de Juegos y pueden servir de base para futuras investigaciones y aplicaciones prácticas en diferentes campos.

4.3 Recomendaciones para mejoras en los métodos

Basado en los resultados y las conclusiones obtenidas en este trabajo, se pueden sugerir algunas recomendaciones para mejorar y expandir los métodos aplicados en la investigación sobre algoritmos genéticos, autómatas finitos e Inteligencia Artificial en la Teoría de Juegos.

En primer lugar, se recomienda explorar y evaluar diferentes estrategias de selección y operadores genéticos en el contexto de la coevolución de autómatas finitos. Dado que la elección de estos elementos puede influir significativamente en el rendimiento y la convergencia de los autómatas, es importante investigar nuevas técnicas y comparar su efectividad en diferentes escenarios de juegos estratégicos. Esto podría incluir el análisis de estrategias de selección elitistas, operadores de mutación adaptativa o el uso de técnicas de recombinación más avanzadas.

Además, es recomendable investigar la aplicación de otros algoritmos de aprendizaje automático en la Teoría de Juegos, más allá de los algoritmos genéticos, por ejemplo con algoritmos de aprendizaje por refuerzo. La Inteligencia Artificial cuenta con una amplia

gama de técnicas y algoritmos, como redes neuronales, aprendizaje profundo y algoritmos basados en agentes, que podrían ofrecer nuevas perspectivas y enfoques para resolver problemas estratégicos en la Teoría de Juegos. Explorar estas alternativas puede brindar oportunidades para descubrir estrategias más sofisticadas y adaptativas en juegos dinámicos y complejos.

Asimismo, se sugiere considerar la aplicación de los métodos propuestos en contextos más amplios y en juegos de mayor escala. Ampliar el alcance de la investigación permitiría analizar cómo se comportan los algoritmos genéticos y los autómatas finitos en situaciones más realistas, donde la complejidad estratégica y la interacción entre múltiples jugadores son más pronunciadas. Esto ayudaría a evaluar la escalabilidad y la robustez de los métodos propuestos, así como a identificar posibles limitaciones o desafíos adicionales que puedan surgir en contextos más complejos.

La Teoría de Juegos, con su capacidad para modelizar y analizar situaciones estratégicas, se revela como una herramienta poderosa y versátil para abordar una amplia gama de problemas matemáticos complejos. Un ejemplo destacado es el problema de la asignación óptima de recursos limitados en proyectos de ingeniería. Supongamos que tenemos un conjunto de proyectos con requerimientos de recursos diferentes y queremos asignarlos a un grupo de equipos con habilidades específicas. Este problema se asemeja a un juego estratégico en el que cada equipo puede decidir qué proyectos asumir para maximizar su eficiencia y minimizar los recursos utilizados.

Para resolver este desafío, se podría aplicar la Teoría de Juegos y algoritmos genéticos, como los utilizados en este trabajo. Se modelaría la interacción estratégica entre los equipos y los proyectos como un juego estático con información completa, donde cada equipo representa un jugador y las asignaciones de recursos a los proyectos son sus estrategias. Mediante la aplicación de algoritmos genéticos, se puede evolucionar la asignación de recursos a lo largo de varias generaciones para encontrar una solución óptima o cercana a ella. Los equipos que adopten estrategias más eficientes y efectivas para abordar proyectos serán favorecidos, y así se obtendrá una asignación óptima de recursos en un tiempo razonable.

Por último, es importante fomentar la colaboración interdisciplinaria entre la Teoría de Juegos, la Inteligencia Artificial y otras áreas relacionadas. La sinergia entre estos campos puede generar nuevas ideas, enfoques y perspectivas para abordar los desafíos en la Teoría de Juegos y en la aplicación de la Inteligencia Artificial. Esto podría incluir colaboraciones con expertos en economía, ciencias sociales y teoría de sistemas, entre otros, para enriquecer aún más la comprensión y las aplicaciones prácticas de los métodos propuestos.

Apéndice A

Algoritmos

Algoritmo genético del Ejemplo 25

Para mejor visualización y ejecución, puede encontrar el algoritmo en los siguientes enlaces en el archivo *aproximacion.py*:

- Google Drive: `https://bit.ly/drive-aproximacion`
- Dropbox: `https://bit.ly/3NCiu7K`

Si tiene algún problema póngase en contacto en la siguiente página web: `https://ismaelalba.web.app`. oTambién puede enviar un correo electrónico a la siguiente dirección: `ismaelalbagallego@gmail.com`

```
# Importamos las librerías necesarias
import numpy as np
import matplotlib.pyplot as plt

# Definimos la función objetivo que queremos maximizar
def f(x1, x2):
    return (x1*x2)/(x1**2+x2**2)

# Definimos los intervalos donde buscamos el máximo
a1 = 0 # Límite inferior de la primera variable
b1 = 10 # Límite superior de la primera variable
a2 = 0 # Límite inferior de la segunda variable
b2 = 8 # Límite superior de la segunda variable

# Definimos el tamaño de la población
N = 15

# Definimos la longitud del cromosoma (número de bits)
L1 = 8 # Longitud de la primera parte del cromosoma
L2 = 8 # Longitud de la segunda parte del cromosoma
L = L1 + L2 # Longitud total del cromosoma

# Definimos la probabilidad de mutación
pm = 0.1

# Definimos el número máximo de generaciones
G = 20
```

```
# Creamos una población inicial aleatoria
poblacion = np.random.randint(0, 2, (N, L))

# Función para decodificar un cromosoma en dos valores reales
def decodificar(cromosoma):
    # Separamos el cromosoma en dos partes
    parte1 = cromosoma[:L1]
    parte2 = cromosoma[L1:]
    # Convertimos cada parte en un número entero
    n1 = int("".join(map(str, parte1)), 2)
    n2 = int("".join(map(str, parte2)), 2)
    # Escalamos los números a los intervalos deseados
    x1 = a1 + (b1 - a1) * n1 / (2**L1 - 1)
    x2 = a2 + (b2 - a2) * n2 / (2**L2 - 1)
    return x1, x2

# Función para calcular el aptitud de un individuo
def aptitud(individuo):
    # Decodificamos el individuo en dos valores reales
    x1, x2 = decodificar(individuo)
    # Evaluamos la función objetivo en esos valores
    y = f(x1, x2)
    # Devolvemos el resultado como aptitud
    return y

# Función para seleccionar un individuo por torneo binario
def seleccionar(poblacion):
    # Elegimos dos individuos al azar
    i, j = np.random.choice(len(poblacion), 2)
    # Comparamos su aptitud
    if aptitud(poblacion[i]) > aptitud(poblacion[j]):
        # Devolvemos el mejor
        return poblacion[i]
    else:
        return poblacion[j]

# Función para cruzar dos individuos con un punto de corte
def cruzar(individuo1, individuo2):
    # Elegimos un punto de corte al azar
    c = np.random.randint(L)
    # Creamos dos nuevos individuos combinando las partes los padres
    nuevo1 = np.concatenate((individuo1[:c], individuo2[c:]))
    nuevo2 = np.concatenate((individuo2[:c], individuo1[c:]))
    # Devolvemos los nuevos individuos
    return nuevo1, nuevo2

# Función para mutar un individuo con una probabilidad dada
def mutar(individuo, probabilidad):
    # Recorremos cada gen del individuo
    for i in range(L):
        # Generamos un número aleatorio entre 0 y 1
        r = np.random.rand()
        # Si el número es menor que la probabilidad, cambiamos el gen
        if r < probabilidad:
            individuo[i] = 1 - individuo[i]
    # Devolvemos el individuo mutado
```

```
    return individuo

# Creamos una lista para guardar la evolución del mejor aptitud
mejor_aptitud = []

# Repetimos el proceso por un número de generaciones
for g in range(G):
    # Evaluamos el aptitud de toda la población
    valores_aptitud = [aptitud(individuo) for individuo in poblacion]
    # Encontramos el mejor individuo
    mejor_individuo = poblacion[np.argmax(valores_aptitud)]
    # Guardamos su aptitud en la lista
    mejor_aptitud.append(aptitud(mejor_individuo))
    # Mostramos el resultado por pantalla
    print(f"Generación {g}: Mejor solución = \
        {decodificar(mejor_individuo)}")
    # Creamos una nueva población vacía
    nueva_poblacion = []
    # Repetimos hasta tener N nuevos individuos
    while len(nueva_poblacion) < N:
        # Seleccionamos dos individuos de la población actual
        padre1 = seleccionar(poblacion)
        padre2 = seleccionar(poblacion)
        # Los cruzamos para obtener dos nuevos individuos
        hijo1, hijo2 = cruzar(padre1, padre2)
        # Los mutamos con una probabilidad dada
        hijo1 = mutar(hijo1, pm)
        hijo2 = mutar(hijo2, pm)
        # Los añadimos a la nueva población
        nueva_poblacion.append(hijo1)
        nueva_poblacion.append(hijo2)
    # Reemplazamos la población actual por la nueva
    poblacion = np.array(nueva_poblacion)

# Mostramos el mejor individuo encontrado y sus valores reales
print(f"Mejor individuo: {mejor_individuo}")
print(f"Valores reales: {decodificar(mejor_individuo)}")
print(f"Con valor f(x,y)= {aptitud(mejor_individuo)}")

# Graficamos la evolución del mejor aptitud
plt.plot(mejor_aptitud)
plt.xlabel("Generación")
plt.ylabel("Mejor aptitud")
plt.show()
```

Algoritmos del Capítulo 3

Para que los códigos sean más fáciles de leer y ejecutar, se pueden ver y descargar desde diferentes sitios web. El algoritmo que se usa en el dilema del prisionero repetido se llama *prisionero.py*, y el algoritmo del oligopolio de Bertrand se llama *oligopolio_Bertrand.py*.

Es imprescindible descargar también el archivo *functions.py*, donde están las funciones que utilizan ambos programas y que son esenciales para que funcionen correctamente.

A continuación se dejan algunos enlaces donde pueden encontrarse:

- Google Drive: `https://bit.ly/drive-prisionero-bertrand-functions`
- Dropbox: `https://bit.ly/dbox-prisionero-bertrand-functions`

En caso de no funcionar correctamente alguno de los sitios web anteriores, los archivos pueden encontrarse en la siguiente página:

`https://ismaelalba.web.app`

Si existiera algún otro problema, o se desea el envío directo de los mismos, póngase en contacto en el siguiente correo:

`ismaelalbagallego@gmail.com`

Índice de figuras

1.1 Juego en forma extensiva. 7
1.2 Juego dinámico. 9
1.3 Dilema del prisionero modificado en forma extensiva. 21
1.4 Dilema del prisionero modificado en forma extensiva con pagos calculados. . 22
1.5 Otra modificación del dilema del prisionero. 22
1.6 Ejemplo del dilema del prisionero repetido. Representación extensiva del juego dinámico. 30
1.7 Subjuegos en el dilema del prisionero repetido. 32
1.8 Subjuegos del juego del trespiés. 36
1.9 Pasos del algoritmo de inducción hacia atrás en el juego del trespiés. 36

2.1 Ejemplo de un estado objetivo. Es importante no confundir con la solución. 45
2.2 Posibles viajes representados en un grafo. 46
2.3 Ejemplo del recorrido del algoritmo de búsqueda en amplitud en un grafo. . 48
2.4 Grafo para ejemplificar el algoritmo de búsqueda en profundidad. 49
2.5 Representación de una posible solución. 54
2.6 Función de Weierstrass con $a = 1/2$ y $b = 3$. 57
2.7 Grafo del polinomio $p(x) = x^6 - x^5 - x^3 - 2x^2 + x$ en el intervalo $[0, 2]$. . . 59
2.8 $f(x, y) = xy/(x^2 + y^2)$ 65
2.9 Algoritmo genético con 7 elementos. 66
2.10 Algoritmo genético con 15 elementos. 67
2.11 Ejemplo de MDP. 70
2.12 Ejemplo de posible laberinto. 74
2.13 Estado de la partida, con el siguiente turno para el jugador con la ficha X. . 78
2.14 Diferentes estados del juego según las posibilidades de cada jugador. 78

3.1 Esquemas de Jerarquía de Chomsky y de la Teoría de autómatas. 82
3.2 Siempre C. 83
3.3 Ojo por ojo. 83
3.4 Castigar dos veces. 83
3.5 Primer movimiendo de un autómata finito. El estado inicial nos lleva al estado interno 2, dependiendo de la opción escogida por el adversario nos moveremos al estado interno 1 ó al estado interno 2 de nuevo. 85
3.6 Proceso de cruce con dos autómatas de 31 bits. 87
3.7 Simulación de decisiones utilizando las dos posibles mutaciones con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50, Pm = 0.1. 88

3.8 Simulación de puntuación utilizando las dos posibles mutaciones con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50, Pm = 0.1. 88
3.9 Simulación utilizando dos posibles valores de la probabilidad de mutación con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50. 89
3.10 Cantidad de mutaciones medias por generación. Simulación utilizando dos posibles valores de la probabilidad de mutación con: NUM_AUTOMATA = 30, NUM_MATCHES = 150, NUM_CHILDREN = 20, GENERATIONS = 50. 90
3.11 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=40. NUM_MATCHES = 150, GENERATIONS = 50, Pm=0.1. 91
3.12 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=40. NUM_MATCHES = 150, NUM_CHILDREN=8, GENERATIONS = 150, Pm=0.1. 92
3.13 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=20. NUM_MATCHES = 150, NUM_CHILDREN=16, GENERATIONS = 30, Pm=0.1. 92
3.14 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_MATCHES = 150, NUM_CHILDREN=20, GENERATIONS = 50, Pm=0.1. 94
3.15 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_MATCHES = 150, NUM_CHILDREN=20, GENERATIONS = 50, Pm=0.1. 94
3.16 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_MATCHES = 150, NUM_CHILDREN=20, GENERATIONS = 100, Pm=0.1. 95
3.17 Simulación utilizando dos poblaciones distintas con: NUM_AUTOMATA=30. NUM_CHILDREN=20, GENERATIONS = 150, Pm=0.1. 96
3.18 Simulación de la primera opción con: NUM_AUTOMATA=50. NUM_MATCHES = 365, NUM_CHILDREN=30, GENERATIONS = 150, Pm=0.1. 98
3.19 Simulación de la segunda opción con: NUM_AUTOMATA=50. NUM_MATCHES = 365, NUM_CHILDREN=30, GENERATIONS = 150, Pm=0.1. 99
3.20 Simulación de la tercera opción con: NUM_AUTOMATA=50. NUM_MATCHES = 365, NUM_CHILDREN=30, GENERATIONS = 150, Pm=0.1. 100

Índice de cuadros

1.1 Juego en forma normal. 6
1.2 Dilema del prisionero en forma normal. 10
1.3 Dilema del prisionero en forma normal. 12
1.4 Juego piedra, papel o tijera en forma normal. 13
1.5 Equilibrio de Nash en el juego piedra, papel o tijera. 14
1.6 Juego de las monedas en forma normal. 14
1.7 Matriz de pagos en el juego de las monedas. 17
1.8 Dilema del prisionero ahora en forma matricial. 23
1.9 Matrices de pago en el juego de la caza de la liebre. 27
1.10 Matrices de pago en el juego de cara o cruz mixto. 27
1.11 Subjuegos del dilema del prisionero repetido en forma normal. 38
1.12 Tabla de resultados del algoritmo hacia atrás en el dilema del prisionero repetido. 38

3.1 Dilema del prisionero en forma normal con valores positivos. 84
3.2 Matriz de pagos del dilema del prisionero modificada. 94
3.3 Matriz de pagos del ejemplo de Bertrand. 97
3.4 Equilibrio de Nash en el subjuego. 99

Bibliografía

[1] Alpaydin, E. Introduction to machine learning. *MIT press* (2010).

[2] Goldberg, D. E. *Genetic algorithms in search, optimization, and machine learning.* Addison-Wesley Professional, 1989.

[3] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning.* MIT press, 2016.

[4] Miller, J. H. The coevolution of automata in the repeated prisoner's dilemma. *Journal of Economic Theory 59*, 1 (1993), 21–51.

[5] Murphy, K. P. *Machine learning: a probabilistic perspective.* MIT press, 2012.

[6] Osborne, M. J., and Rubinstein, A. *An introduction to game theory.* Oxford University Press, 2004.

[7] Russell, S. J., and Norvig, P. *Artificial Intelligence: A Modern Approach.* Pearson, 2016.

[8] Sutton, R. S., and Barto, A. G. *Reinforcement learning: An introduction.* MIT press, 2018.

[9] Tembine, H. Deep learning meets game theory. *arXiv preprint arXiv:1502.03442* (2015).

[10] Yannakakis, G. N., and Togelius, J. *Artificial Intelligence and Games.* Springer, 2018.