Algoritmos y Estructuras de Datos

Backtracking

Segundo trimestre de 2025

Maestría en Inteligencia Artificial

Universidad de San Andrés

Enumeraciones combinatorias

Backtracking y podas

Método minimax

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Dado $x \in X$, los métodos de **fuerza bruta** consisten en:

- Generar una lista $y_1, \dots, y_n \in Y$ de candidatos.
- Para cada candidato $y \in Y$:
  - Si y tiene las propiedades esperadas de $f(x)$, devolver y.

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Dado $x \in X$, los métodos de **fuerza bruta** consisten en:

- Generar una lista $y_1, \ldots, y_n \in Y$ de candidatos.
- Para cada candidato $y \in Y$:
  - Si y tiene las propiedades esperadas de $f(x)$, devolver y.

Calcular la solución y se reduce a **verificar** si y es solución.

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Dado $x \in X$, los métodos de **fuerza bruta** consisten en:

- Generar una lista $y_1, \ldots, y_n \in Y$ de candidatos.
- Para cada candidato $y \in Y$:
  - Si y tiene las propiedades esperadas de $f(x)$, devolver y.

Calcular la solución y se reduce a **verificar** si y es solución.

En general es extremadamente ineficiente.

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Dado $x \in X$, los métodos de **fuerza bruta** consisten en:

- Generar una lista $y_1, \ldots, y_n \in Y$ de candidatos.
- Para cada candidato $y \in Y$:
  - Si y tiene las propiedades esperadas de $f(x)$, devolver y.

Calcular la solución y se reduce a **verificar** si y es solución.

En general es extremadamente ineficiente.
Pero:

- Puede ser practicable cuando los datos de entrada son chicos.

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Dado $x \in X$, los métodos de **fuerza bruta** consisten en:

- Generar una lista $y_1, \ldots, y_n \in Y$ de candidatos.
- Para cada candidato $y \in Y$:
  - Si y tiene las propiedades esperadas de $f(x)$, devolver y.

Calcular la solución y se reduce a **verificar** si y es solución.

En general es extremadamente ineficiente.
Pero:

- Puede ser practicable cuando los datos de entrada son chicos.
- Hay problemas para los que no hay (o no se conoce) una solución esencialmente mejor.

Métodos de fuerza bruta

Queremos calcular una función $f : X \to Y$.

Dado $x \in X$, los métodos de **fuerza bruta** consisten en:

- Generar una lista $y_1, \dots, y_n \in Y$ de candidatos.
- Para cada candidato $y \in Y$:
  - Si y tiene las propiedades esperadas de $f(x)$, devolver y.

Calcular la solución y se reduce a **verificar** si y es solución.

En general es extremadamente ineficiente.
Pero:

- Puede ser practicable cuando los datos de entrada son chicos.
- Hay problemas para los que no hay (o no se conoce) una solución esencialmente mejor.
- Útil para hacer testing.

Métodos de fuerza bruta

Ejemplo

- La clave del usuario es un PIN de 10 dígitos.

Métodos de fuerza bruta

Ejemplo

- La clave del usuario es un PIN de 10 dígitos.
- El sistema no guarda la clave K sino $H(K)$.

Métodos de fuerza bruta

Ejemplo

- La clave del usuario es un PIN de 10 dígitos.
- El sistema no guarda la clave K sino $H(K)$.
- H es una función de *hash criptográfico*.
  (Calcular la inversa H^{-1} es impracticable).

Métodos de fuerza bruta

Ejemplo

- La clave del usuario es un PIN de 10 dígitos.
- El sistema no guarda la clave K sino $H(K)$.
- H es una función de *hash criptográfico*.
  (Calcular la inversa H^{-1} es impracticable).
- Si averiguo $H(K)$, ¿cómo puedo averiguar K?

Métodos de fuerza bruta

Ejemplo

- La clave del usuario es un PIN de 10 dígitos.
- El sistema no guarda la clave K sino $H(K)$.
- H es una función de *hash criptográfico*.
  (Calcular la inversa H^{-1} es impracticable).
- Si averiguo $H(K)$, ¿cómo puedo averiguar K?
- Calcular:

  $$H(0000000000),\ H(0000000001),\ \ldots,\ H(9999999999)$$

  hasta dar con la entrada K que otorga el valor buscado.

Métodos de fuerza bruta

Ejemplo

- La clave del usuario es un PIN de 10 dígitos.
- El sistema no guarda la clave K sino $H(K)$.
- H es una función de *hash criptográfico*.
  (Calcular la inversa H^{-1} es impracticable).
- Si averiguo $H(K)$, ¿cómo puedo averiguar K?
- Calcular:

  $$H(0000000000),\ H(0000000001),\ \ldots,\ H(9999999999)$$

  hasta dar con la entrada K que otorga el valor buscado.
- 10^{10} cómputos se pueden hacer en cuestión de minutos.

Enumeraciones combinatorias: variaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (con repeticiones; el orden es relevante).

Enumeraciones combinatorias: variaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (con repeticiones; el orden es relevante).

Por ejemplo, si $A = [0, 1]$ y $k = 3$, el resultado es:

$$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]$$

Enumeraciones combinatorias: variaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (con repeticiones; el orden es relevante).

Por ejemplo, si $A = [0, 1]$ y $k = 3$, el resultado es:

$$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]$$

Si $|A| = n$, hay n^k variaciones con repetición.

Enumeraciones combinatorias: variaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (con repeticiones; el orden es relevante).

Por ejemplo, si $A = [0, 1]$ y $k = 3$, el resultado es:

$$[[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]]$$

Si $|A| = n$, hay n^k variaciones con repetición.

```
def variaciones_con_repeticion(a, k):
    if k == 0:
        return [[]]
    res = []
    for x in a:
        for v in variaciones(a, k - 1):
            res.append([x] + v)
    return res
```

Enumeraciones combinatorias: combinaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (sin repetir; el orden es irrelevante).

Enumeraciones combinatorias: combinaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (sin repetir; el orden es irrelevante).

Por ejemplo, si $A = [0, 1, 2, 3]$ y $k = 3$, el resultado es:

$$[[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]]$$

Enumeraciones combinatorias: combinaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (sin repetir; el orden es irrelevante).

Por ejemplo, si $A = [0, 1, 2, 3]$ y $k = 3$, el resultado es:

$$[[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]]$$

Si $|A| = n$, hay $\binom{n}{k}$ combinaciones sin repetición.

Enumeraciones combinatorias: combinaciones

Entrada: una lista A y un entero $k \geq 0$.
Salida: todas las listas de longitud k que se pueden formar tomando elementos de A (sin repetir; el orden es irrelevante).

Por ejemplo, si $A = [0, 1, 2, 3]$ y $k = 3$, el resultado es:

$$[[0, 1, 2], [0, 1, 3], [0, 2, 3], [1, 2, 3]]$$

Si $|A| = n$, hay $\binom{n}{k}$ combinaciones sin repetición.

```
def combinaciones_sin_repeticion(a, k):
    if k == 0:
        return [[]]
    elif len(a) == 0:
        return []
    res = []
    for c in combinaciones_sin_repeticion(a[1:], k - 1):
        res.append([a[0]] + c)
    for c in combinaciones_sin_repeticion(a[1:], k):
        res.append(c)
    return res
```

Enumeraciones combinatorias: permutaciones

Entrada: una lista A.
Salida: una lista con todas las permutaciones de A.

Enumeraciones combinatorias: permutaciones

Entrada: una lista A.
Salida: una lista con todas las permutaciones de A.

Por ejemplo, si $A = [0, 1, 2]$, el resultado es:

$$[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]$$

Enumeraciones combinatorias: permutaciones

Entrada: una lista A.
Salida: una lista con todas las permutaciones de A.

Por ejemplo, si $A = [0, 1, 2]$, el resultado es:

$$[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]$$

Si $|A| = n$, hay $n!$ permutaciones.

Enumeraciones combinatorias: permutaciones

Entrada: una lista A.
Salida: una lista con todas las permutaciones de A.

Por ejemplo, si $A = [0, 1, 2]$, el resultado es:

$$[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]$$

Si $|A| = n$, hay $n!$ permutaciones.

```
def permutaciones(a):
    if len(a) == 0:
        return [[]]
    res = []
    for i in range(len(a)):
        for p in permutaciones(a[:i] + a[i + 1:]):
            res.append([a[i]] + p)
    return res
```

Enumeraciones combinatorias

Backtracking y podas

Método minimax

Backtracking

El *backtracking* es una técnica algorítmica para buscar soluciones.

Backtracking

El *backtracking* es una técnica algorítmica para buscar soluciones.

Consiste en las siguientes ideas:

1. Cada vez que la búsqueda presenta varias opciones, elegir alguna de ellas.

Backtracking

El *backtracking* es una técnica algorítmica para buscar soluciones.

Consiste en las siguientes ideas:

1. Cada vez que la búsqueda presenta varias opciones, elegir alguna de ellas.
2. Cuando se llega a un estado que no conduce a una solución, retroceder hasta el último punto en el que se eligió una opción, y probar con otra opción.

Backtracking

El *backtracking* es una técnica algorítmica para buscar soluciones.

Consiste en las siguientes ideas:

1. Cada vez que la búsqueda presenta varias opciones, elegir alguna de ellas.
2. Cuando se llega a un estado que no conduce a una solución, retroceder hasta el último punto en el que se eligió una opción, y probar con otra opción.
3. Cuando se agotan las opciones, seguir retrocediendo.

Backtracking

Ejemplo — Laberinto

1. Un laberinto es una matriz de n filas por m columnas.

Backtracking

Ejemplo — Laberinto

1. Un laberinto es una matriz de n filas por m columnas.
2. El inicio está en la posición $(0, 0)$.

   El final está en la posición $(n - 1, m - 1)$.

Ejemplo — Laberinto

1. Un laberinto es una matriz de n filas por m columnas.
2. El inicio está en la posición $(0, 0)$.

   El final está en la posición $(n - 1, m - 1)$.
3. Cada celda del laberinto contiene uno de tres posibles valores:
   - 3.1 `VACIO`: camino por el que es posible transitar.
   - 3.2 `PARED`: pared infranqueable.
   - 3.3 `HILO`: parte del camino que ya fue recorrida.

Ejemplo — Laberinto

1. Un laberinto es una matriz de n filas por m columnas.
2. El inicio está en la posición $(0, 0)$.

   El final está en la posición $(n-1, m-1)$.
3. Cada celda del laberinto contiene uno de tres posibles valores:
   3.1 `VACIO`: camino por el que es posible transitar.
   3.2 `PARED`: pared infranqueable.
   3.3 `HILO`: parte del camino que ya fue recorrida.

Programemos un algoritmo con el siguiente contrato:

- Entrada: un laberinto y una posición (i, j) dentro del laberinto.
- Salida: un booleano que indica si es posible llegar a la salida desde la posición (i, j). Además, se modifica el laberinto dejando un rastro de "hilo" desde la posición (i, j) hasta la salida.

Backtracking

Ejemplo — Sudoku

1. Un sudoku es una matriz de 9 filas por 9 columnas.

Backtracking

Ejemplo — Sudoku

1. Un sudoku es una matriz de 9 filas por 9 columnas.
2. Cada celda puede estar vacía o contener un dígito entre 1 y 9.

Backtracking

Ejemplo — Sudoku

1. Un sudoku es una matriz de 9 filas por 9 columnas.
2. Cada celda puede estar vacía o contener un dígito entre 1 y 9.
3. El problema consiste en completar todas las celdas:
    3.1 No debe haber dígitos repetidos en ninguna fila.

Backtracking

Ejemplo — Sudoku

1. Un sudoku es una matriz de 9 filas por 9 columnas.
2. Cada celda puede estar vacía o contener un dígito entre 1 y 9.
3. El problema consiste en completar todas las celdas:
   3.1 No debe haber dígitos repetidos en ninguna fila.
   3.2 No debe haber dígitos repetidos en ninguna columna.

Ejemplo — Sudoku

1. Un sudoku es una matriz de 9 filas por 9 columnas.
2. Cada celda puede estar vacía o contener un dígito entre 1 y 9.
3. El problema consiste en completar todas las celdas:
   3.1 No debe haber dígitos repetidos en ninguna fila.
   3.2 No debe haber dígitos repetidos en ninguna columna.
   3.3 No debe haber dígitos repetidos en ningún cuadrante.

Backtracking

Ejemplo — Sudoku

1. Un sudoku es una matriz de 9 filas por 9 columnas.
2. Cada celda puede estar vacía o contener un dígito entre 1 y 9.
3. El problema consiste en completar todas las celdas:
   3.1 No debe haber dígitos repetidos en ninguna fila.
   3.2 No debe haber dígitos repetidos en ninguna columna.
   3.3 No debe haber dígitos repetidos en ningún cuadrante.

Programemos un algoritmo con el siguiente contrato.

- Entrada: un sudoku y una posición (i, j).
  Todas las filas anteriores a la fila i deben estar ya completas.
  Las celdas de la fila i anteriores a la columna j deben estar ya completas.
- Salida: en caso de que exista solución, devuelve el sudoku completo. Si no existe, devuelve `None`.

Enumeraciones combinatorias

Backtracking y podas

Método minimax

Minimax

El método MINIMAX sirve para tomar decisiones que maximicen el beneficio obtenido por un jugador en el contexto de un *juego*.

Minimax

El método MINIMAX sirve para tomar decisiones que maximicen el beneficio obtenido por un jugador en el contexto de un *juego*.

Vamos a enfocarnos en juegos:

- De dos jugadores.
- De información perfecta.
- De suma cero.

Minimax

El método MINIMAX sirve para tomar decisiones que maximicen el beneficio obtenido por un jugador en el contexto de un *juego*.

Vamos a enfocarnos en juegos:

- De dos jugadores.
- De información perfecta.
- De suma cero.

Ejemplos: ta-te-ti, ajedrez, damas, go, reversi, ...

Minimax

El método Minimax sirve para tomar decisiones que maximicen el beneficio obtenido por un jugador en el contexto de un *juego*.

Vamos a enfocarnos en juegos:

- De dos jugadores.
- De información perfecta.
- De suma cero.

Ejemplos: ta-te-ti, ajedrez, damas, go, reversi, ...

Para empezar, supondremos además que:

- El juego termina al cabo de un número finito de movidas.
- No hay empates.

Hay versiones más generales.

Minimax

Dada una posición P de un juego:

- $M(P)$ es el conjunto de posiciones después de un movimiento.

Minimax

Dada una posición P de un juego:

- $M(P)$ es el conjunto de posiciones después de un movimiento.
- Si $M(P) = \{P_1, \ldots, P_n\}$ el jugador tiene n opciones.

Minimax

Dada una posición P de un juego:

- $M(P)$ es el conjunto de posiciones después de un movimiento.
- Si $M(P) = \{P_1, \ldots, P_n\}$ el jugador tiene n opciones.
- En cada posición P_i le toca jugar al oponente.

Minimax

Dada una posición P de un juego:

- $M(P)$ es el conjunto de posiciones después de un movimiento.
- Si $M(P) = \{P_1, \ldots, P_n\}$ el jugador tiene n opciones.
- En cada posición P_i le toca jugar al oponente.
- A su vez, $M(P_i) = \{Q_1, \ldots, Q_m\}$ son opciones del oponente.

Minimax

Dada una posición P de un juego:

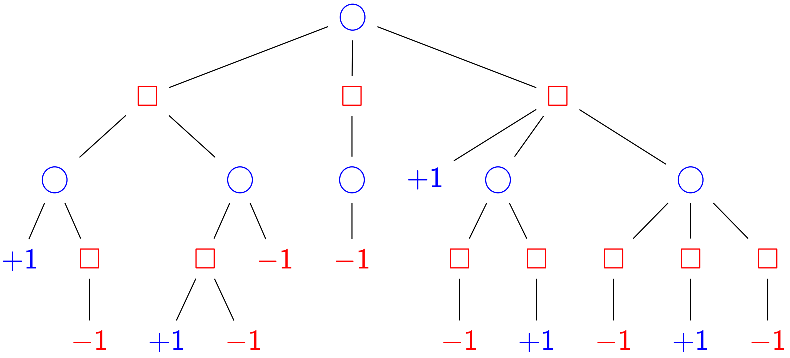
- $M(P)$ es el conjunto de posiciones después de un movimiento.
- Si $M(P) = \{P_1, \ldots, P_n\}$ el jugador tiene n opciones.
- En cada posición P_i le toca jugar al oponente.
- A su vez, $M(P_i) = \{Q_1, \ldots, Q_m\}$ son opciones del oponente.

Nota. Por convención, suponemos que $M(P) \neq \emptyset$. Cuando el juego termina, el jugador tiene sólo una opción.

Minimax

○ = turno del jugador
$+1$ = gana el jugador

□ = turno del oponente
-1 = gana el oponente



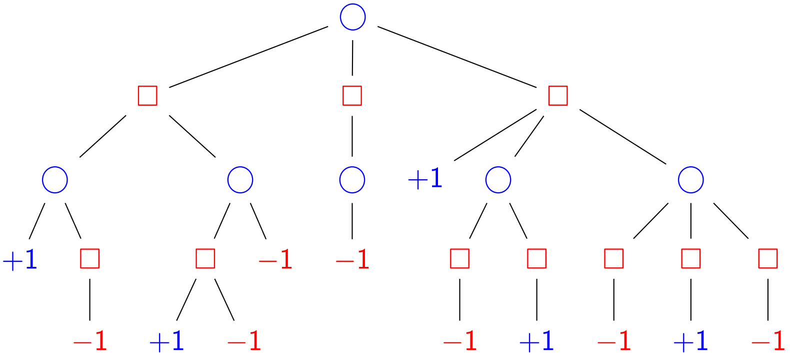
Minimax

○ = turno del jugador

$+1$ = gana el jugador

□ = turno del oponente

-1 = gana el oponente

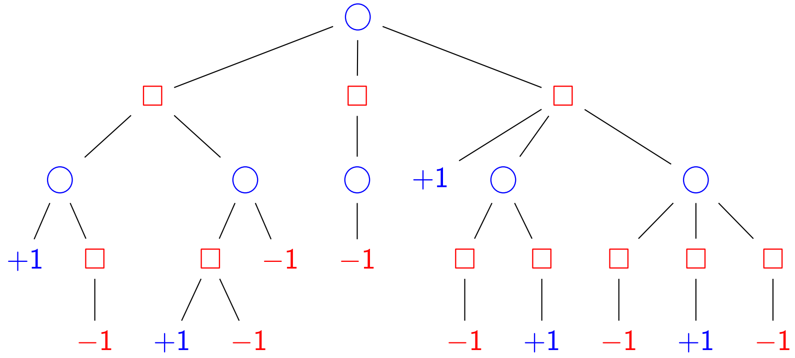


Al jugador le conviene elegir la opción que maximiza el resultado,

Minimax

○ = turno del jugador
$+1$ = gana el jugador

□ = turno del oponente
-1 = gana el oponente



Al jugador le conviene elegir la opción que maximiza el resultado, asumiendo que el oponente elige la opción que lo minimiza.

Minimax

Algoritmo Minimax

Entrada: una posición P del juego.
Salida:

- $+1$ si P es una posición ganadora para el jugador.
- -1 si P es una posición ganadora para el oponente.

Minimax

Algoritmo MINIMAX

Entrada: una posición P del juego.
Salida:

- $+1$ si P es una posición ganadora para el jugador.
- -1 si P es una posición ganadora para el oponente.

Procedimiento:

- Si P es una posición terminal: (Ej. "jaque mate").
  - Devolver $+1$ si gana el jugador.
  - Devolver -1 si gana el oponente.

Minimax

Algoritmo Minimax

Entrada: una posición P del juego.
Salida:

- $+1$ si P es una posición ganadora para el jugador.
- -1 si P es una posición ganadora para el oponente.

Procedimiento:

- Si P es una posición terminal: (Ej. "jaque mate").
  - Devolver $+1$ si gana el jugador.
  - Devolver -1 si gana el oponente.
- En caso contrario:
  - Devolver $\max_{P' \in M(P)} \min_{P'' \in M(P')} \textsc{Minimax}(P'')$.

Minimax

Algoritmo $\textsc{Minimax}$

Entrada: una posición P del juego.
Salida:

- $+1$ si P es una posición ganadora para el jugador.
- -1 si P es una posición ganadora para el oponente.

Procedimiento:

- Si P es una posición terminal: (Ej. "jaque mate").
  - Devolver $+1$ si gana el jugador.
  - Devolver -1 si gana el oponente.
- En caso contrario:
  - Devolver $\max_{P' \in M(P)} \min_{P'' \in M(P')} \textsc{Minimax}(P'')$.

Idea: el jugador puede **forzar** al oponente a perder.

Minimax

- El método anterior es “perfecto”.

Minimax

- El método anterior es “perfecto”.
- Se vuelve inviable cuando el árbol de juego es grande.
  (P. ej. es viable para el ta-te-ti pero no para el ajedrez).

Minimax

- El método anterior es “perfecto”.
- Se vuelve inviable cuando el árbol de juego es grande.
  (P. ej. es viable para el ta-te-ti pero no para el ajedrez).
- El método se adapta para usar una **función de evaluación**.

Algoritmo MINIMAX (con función de evaluación)

Entrada: una posición P del juego.

Salida: valor heurístico para la posición P
(positivo si favorece al jugador / negativo si favorece al oponente).

Minimax

Algoritmo MINIMAX (con función de evaluación)

Entrada: una posición P del juego.
Salida: valor heurístico para la posición P
(positivo si favorece al jugador / negativo si favorece al oponente).

Procedimiento:

- Si P es una posición terminal o se alcanzó la profundidad máxima:
  - Devolver $+1$ si gana el jugador.
  - Devolver -1 si gana el oponente.

Minimax

Algoritmo MINIMAX (con función de evaluación)

Entrada: una posición P del juego.
Salida: valor heurístico para la posición P
(positivo si favorece al jugador / negativo si favorece al oponente).

Procedimiento:

- Si P es una posición terminal o se alcanzó la profundidad máxima:
  - Devolver $+1$ si gana el jugador.
  - Devolver -1 si gana el oponente.
- Si se alcanzó la profundidad máxima de exploración:
  - Aplicar la función de evaluación heurística sobre P.

Minimax

Algoritmo $\textsc{Minimax}$ (con función de evaluación)

Entrada: una posición P del juego.
Salida: valor heurístico para la posición P
(positivo si favorece al jugador / negativo si favorece al oponente).

Procedimiento:

- Si P es una posición terminal o se alcanzó la profundidad máxima:
  - Devolver $+1$ si gana el jugador.
  - Devolver -1 si gana el oponente.
- Si se alcanzó la profundidad máxima de exploración:
  - Aplicar la función de evaluación heurística sobre P.
- En caso contrario:
  - Devolver $\max_{P' \in M(P)} \min_{P'' \in M(P')} \textsc{Minimax}(P'')$.