# 3. Pregunta 3

## 3.1. Actividad 1

La poda alfa beta es una técnica de búsqueda que reduce el número de nodos evaluados en un árbol de juego por el algoritmo Minimax. (Wikipedia, 2025).

Sabemos que el problema en particular de minimax es que la cantidad de nodos a explorar es exponencial al número de movimientos, por lo que debemos buscar formas de hacer más eficiente este proceso. La poda alfa beta nos sirve para esto.

Para hacer esta implementación, nos basaremos en el código publicado aquí pero evaluado en nuestro caso.

Ahora modificamos `alphabeta = TRUE` y evaluamos.

| Algoritmo | Profundidad | J1 (Tiempo Promedio) | J2 (Tiempo Promedio) |
|---|---|---|---|
| Minimax sin poda | 1 | 0.003 s | 0.002 s |
| Minimax con poda | 1 | 0.003 s | 0.003 s |
| Minimax sin poda | 2 | 0.05 s | 0.049 s |
| Minimax con poda | 2 | 0.051 s | 0.052 s |
| Minimax sin poda | 3 | 0.951 s | 0.859 s |
| Minimax con poda | 3 | 0.282 s | 0.281 s |
| Minimax sin poda | 4 | 17.525 s | 16.122 s |
| Minimax con poda | 4 | 1.822 | 1.73 |

Como podemos ver en cuanto al tiempo promedio de ejecución de archivos, a medida que aumenta la profundidad, el tiempo promedio de respuesta y el tiempo total aumentan igualmente.

Sin embargo, vemos que el uso de la poda alpha beta ayuda a disminuir el tiempo de ejecución, esto debido a que como el algoritmo ayuda a podar ciertos nodos que no son necesarios de abrir, reduciendo así la cantidad de nodos expandidos.

## 3.2. Actividad 2

Ahora veremos la evaluación entre los distintos valores de profundidad evaluando en `eval_func = score_circle_values`

| Algoritmo | Profundidad . | Rojo | Profundidad | Verde | Ganador |
|---|---|---|---|---|---|
| Minimax con poda | 1 | 0.003s | 1 | 0.003s | Verde |
| Minimax con poda | 1 | 0.003s | 2 | 0.057s | Verde |
| Minimax con poda | 2 | 0.053s | 1 | 0.004s | Verde |
| Minimax con poda | 1 | 0.003s | 3 | 0.238s | Verde |
| Minimax con poda | 3 | 0.24s | 1 | 0.003s | Rojo |

La profundidad del algoritmo refiere a que tantas expansiones hace este sobre sus hijos antes de decidir su ejecución, a tener una mayor cantidad de profundidad ralentizaremos el código ya que debemos expandir una mayor cantidad de opciones, sin embargo es una forma más certera de definir cuales son las mejores ejecuciones.

## 3.3. Actividad 3

Ahora evaluaremos el rendimiento de `score_circle_amount` y `score_circle_values` para distintos valores de profundidad, manteniendo siempre el algoritmos de Minimax con poda.

| `score_circle_` | **Profundidad** . | **Rojo** | `score_circle_` | **Profundidad** | **Verde** | **Ganador** |
|---|---|---|---|---|---|---|
| amount | 1 | 0.003s | values | 1 | 0.003s | Rojo |
| values | 1 | 0.003s | amount | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | values | 3 | 0.189s | Verde |
| amount | 3 | 0.629s | values | 1 | 0.004s | Rojo |

La función que es mejor ante estos resultados es `score_circle_amount`, ya que para condiciones iguales de profundidad e intercambiando los jugadores, esta resultó ser la predominante en ambas ocasiones.

Esto se puede deber a la forma en la que se compone la función, donde amount prioriza la cantidad de circulos antes que el número en particular de cada circulo, lo que en términos prácticos termina siendo más eficiente ya que se acerca más al objetivo inicial del juego.

## 3.4. Actividad 4

Para esta actividad usaremos la misma función `score_circle_amount` solo que modificaremos los contadores para sumar `+= 5`, a esta función la llamaremos `score_circle_amount_2`.

```
## AGREGAR AQUÍ TUS FUNCIONES DE EVALUACIÓN
def score_circle_amount_2(circles, JUGADOR):
    contador_j1 = 0
    contador_j2 = 0
    resta = 0
    for list_circle in circles:
        for circle in list_circle:
            if circle.id == 0:
                contador_j1 += 0.5

            elif circle.id == 1:
                contador_j2 += 0.5
    if JUGADOR.id == 0:
        resta = contador_j1 - contador_j2
    else:
        resta = contador_j2 - contador_j1

    return resta
```

Al ejecutar el código con profundidad 1 para ambos jugadores, de 10 ejecuciones, 7 fueron ganadas por el modelo asociado a esta nueva función. Por lo que podemos decir que tiene una efectividad del 70 %.

| `score_circle_` | **Profundidad .** | **Rojo** | `score_circle_` | **Profundidad** | **Verde** | **Ganador** |
|---|---|---|---|---|---|---|
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Rojo |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.002s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Rojo |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | amount2 | 1 | 0.003s | Verde |
| amount | 1 | 0.003s | amount2 | 1 | 0.002s | Rojo |

Esta función resulta ser más eficienta ya que al usar un número menor a 1, se reduce el peso de la cantidad de círculos controlados, lo que da más margen para que otros factores estratégicos puedan ser utilizados para ganar.

## 3.5. Actividad Bonus

Una función que puede ser más estrategica y potencialmente mejor que `score_circle_amount` puede ser una función que priorice más la potencial explosividad de las posiciones antes que buscar una mayor cantidad de círculos.