Universidad de Costa Rica

Escuela de Matemática

Proyecto Individual

Por:

Dominick Rodríguez Trejos, B76600

Junio 2024

1. Pregunta de investigación

¿Los conocimientos de probabilidad y programación son útiles para la construcción de mazos competitivos para juegos de cartas (TCG)?

2. Probabilidad en juegos de cartas

Los conceptos de probabilidad estudiados están presentes en todos los juegos de cartas, ya sean los tradicionales como Poker, Blackjack o juegos más modernos y poco convencionales, mejor conocidos como TCG, como Yu-Gi-Oh!, Hearthstone, Magic: The Gathering y demás.

Es posible traducir todo evento que sucede en estos juegos de carta en términos de probabilidades, sin duda la probabilidad tiene un rol fundamental en las mecánicas y eventos presentes en todo juego de cartas, la pregunta de interés es ¿cómo los jugadores aprovechan estos conceptos para obtener una ventaja sobre sus oponentes?

En termino de juegos de carta convencionales, podemos considerar que los conceptos de probabilidad se aplican solamente al momento de tomar decisiones durante la partida, pues es imposible controlar o cambiar las cartas que conforman el mazo de juego, en comparación, los TCG agregan una capa extra de complejidad y competitividad pues los jugadores tienen control sobre las cartas que agregan al mazo de juego lo cuál, en mi opinión, abre posibilidades para que los jugadores expresen sus habilidades, conocimientos y preferencias.

Para este proyecto, se tomo un enfoque sobre la parte de construcción del mazo previa al juego, primeramente porque es uno de mis intereses personales y también se presta mucho más para la construcción de aplicaciones que funcionan como una herramienta extra para el jugador.

3. Yu-Gi-Oh!

Entre todas las opciones de TCG disponibles se eligió Yu-Gi-Oh! como el juego de estudio, principalmente por razones personales y también porque para lograr construir una herramienta del tipo que se desea es necesario tener un vasto conocimiento y entendimiento del juego a analizar, lo cuál es solamente posible con años de experiencia personal.

Yu-Gi-Oh!, originalmente basado en un juego ficticio presente en el manga del mismo nombre, escrito por Kazuki Takahashi, fue el primer TCG creado y desarrollado por Konami en 1998, tomando gran inspiración en el éxito Magic: The Gathering, Yu-Gi-Oh! siguió en los pasos del gigante en juegos de cartas y se convirtió en uno de los TCG más reconocidos y populares a nivel mundial.

La diferencia entre el juego que empezó en 1998 y el que continua siendo una sensación mundial en este 2024 es evidente, las formas de jugar Yu-Gi-Oh! y el estilo de juego adoptado por los jugadores ha cambiado exageradamente, y estos cambios también aplican a la forma en la que los jugadores construyen sus mazos.

Para construir una herramienta que ayude a los jugadores a construir sus mazos primero debemos entender cual es la mejor forma de construir un mazo, cuales cartas son mas importantes y lo más importante cuales son las probabilidades de que el mazo construido cumpla su propósito, este último será uno de los principales objetivos de nuestra herramienta.

4. ¿Cómo se gana en Yu-Gi-Oh!?

Por reglas del juego, el jugador que reduzca los puntos de vida (LP) de su oponente a cero primero es el ganador, pero como es el caso con muchos juegos de cartas competitivos esta condición de victoria se ha convertido en un objetivo secundario para los jugadores, hoy en día es muy sencillo disminuir los LP del oponente a cero, por lo que los jugadores han adoptado un filosofía diferente, el objetivo principal del jugador es, con su mano inicial en su primer turno, crear una situación en la que el oponente no logré ganar la partida durante su turno, y así en su próximo turno, disminuir los LP de su oponente a cero y ganar la partida, tanto así que la mayoría de partidas duran solamente 3 o 4 turnos en total.

Entonces surge la duda, como puede el jugador crear una situación de juego en la que su oponente no pueda ganar durante su turno, la respuesta es por medio de lo que la comunidad ha nombrado *combos*, combinaciones de jugadas y cartas en la mayoría de los casos bastante complicadas y largas, que resultan en situaciones que no permiten al oponente ganar la partida durante su turno y en consecuencia pasar el turno de vuelta al jugador, él cuál, termina la partida disminuyendo los LP de su oponente a cero.

Después de años de desarrollo en lo que los jugadores denominan *Combo theory* se ha vuelto evidente que para lograr tener una posibilidad competitiva el mazo del jugador debe tener la habilidad de completar su combo la mayor cantidad de veces posibles. Si la mejor opción para ganar en Yu-Gi-Oh! es completar el combo respectivo de nuestro mazo, entonces nuestra herramienta debería ayudar al jugador a construir un mazo que tenga la mejor posibilidad posible de completar su combo.

5. Traduciendo conceptos reales a objetos de Python

Primeramente, esto parece una tarea sencilla, los mazos usados por los jugadores tienen claras características y atributos que pueden ser traducidos a variable y atributos de objetos en Python, e incluso las cartas son similares, cada una teniendo un grupo de particularidades que pueden ser divididas en categorías distintas, pero al analizar y observar todas estas nos podemos dar cuenta de que algunas son irrelevantes en la ejecución del combo y que son demasiadas para considerar, con esto en mente se debe hacer un análisis a profundidad del juego y de las cartas para determinar cuales aspectos son los más importantes y los menos importantes para nuestros propósitos.

Basándonos en nuestro criterio personal, enforzado por más de 5 años de experiencia con el juego y con la opinión de múltiples jugadores recolectadas durante los años en foros en línea, salas de chat en línea y reportes de torneos competitivos, se determino que lo más importante es la clara clasificación de las cartas del mazo según su utilidad, y con esto en mente se logró dividir las cartas en seis grandes categorías, a continuación las podemos ver ordenadas de la más importante a la menos importante:

- Starters: las cartas más importantes del mazo, son capaces de empezar y completar el combo por si solas, siempre y cuando el oponente no cuente con las cartas necesarias para interrumpirlas o contrarrestarlas.
- Defensives: se refiere a cartas que tienen funciones defensivas, y funcionan para proteger nuestras cartas de los intentos de nuestro oponente de contrarrestarlas y también permiten contrarrestar las cartas de nuestro oponente.
- Extenders: son cartas que nos permiten continuar nuestro combo en caso de que nuestros Starters sean contrarrestados, aunque usualmente resultan en un combo un poco menos fuerte y una situación que abre más posibilidades de que el oponente logré ganar durante su turno.
- Combo pieces o piezas del combo: son cartas que se deben jugar para completar el combo, pero no necesariamente se quieren tener en la mano inicial, aunque si llegan a tener no tienen ningún impacto negativo en la ejecución del combo.
- Non engine: se refiere a cartas que no pertenecen al *engine* o motor base del combo, estas usualmente funcionan como una opción que fortalece o complementa nuestro combo pero usualmente no de manera significativa.
- Garnets: son cartas que se deben poner en el mazo por obligación, ya sea porque son necesarias para activar o usar otras cartas, pero nunca se quieren ver en la mano inicial.

6. Construcción de la herramienta

El código utilizado para la construcción de la herramienta y la interfaz puede encontrarse en el siguiente repositorio de GitHub en este documento se exploraran los métodos más importantes:

Repositorio del Proyecto Individual

6.1. Métodos importante

Para la toma de manos se crearon dos métodos, los cuales se construyeron considerando el proposito de la toma de la mano, si la mano que se desea tomar es una cualquier y las cartas tomadas no se devuelven a la mano, entonces se usa el método hand_sample() que toma cartas del mazo y las quita de la lista de cartas disponibles a tomar, en el caso contrario se usa el método starting_hand_sample() el cuál toma siempre cinco cartas (tamaño de la mano inicial de Yu-Gi-Oh!) pero no las quita de la lista de cartas disponibles del mazo.

Ambos métodos funcionan con ayuda de la librería random, la cual nos ayuda a obtener un número aleatorio entre 0 y el número de cartas del mazo, y luego a la mano, la cual es una lista al inicio vacía, se agrega la carta en la posición del número aleatorio en la lista de cartas disponibles, y así sucesivamente hasta que se tome la cantidad de cartas deseadas por el usuario. Podemos ver el código en la siguiente imagen:

```python
def hand_sample(self, num_draw):
    # Creo una lista vacía, aquí pondré las cartas que toma aletoriamente del deck
    mano = []
    # Importo random para usar randint y así obtener un número aleatorio entre 0 y el tamaño
    # del deck - 1, uso el comando pop() porque cuando se toma una carta del deck se debe
    # eliminar de las posibles cartas a tomar la próxima vez que se tome una carta del deck
    # Importar los paquetes al inicio
    import random
    for i in range(0, num_draw):
        draw = self.__deck_list.pop(random.randint(0, len(self.__deck_list) - 1))
        mano.append(draw)
        # Reviso la utilidad de la carta tomada del mazo, esto lo hago en varios métodos.
        indice_draw = self.__card_stats.index[self.__card_stats["Carta"] == draw][0]
        if self.__card_stats.loc[indice_draw, "Utilidad"] == "Starter":
            self.__starters = self.__starters - 1
        elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Extender":
            self.__extenders = self.__extenders - 1
        elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Defensive":
            self.__defensives = self.__defensives - 1
        elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Combo piece":
            self.__combo_pieces = self.__combo_pieces - 1
        elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Garnet":
            self.__garnets = self.__garnets - 1
        elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Non engine":
            self.__non_engine = self.__non_engine - 1
    return mano
```

Figura 1: Método hand_sample() en Spyder

También tenemos un método que agrega una sola carta a la mano, este funciona muy similar al anterior, pero recibe como parámetro la mano que el usuario tiene en este momento y le agrega una carta aleatoria con el mismo proceso al anterior, en la siguiente imagen podemos ver el código correspondiente a este método:

```python
def draw_card(self, mano):
    draw = self.__deck_list.pop(random.randint(0, len(self.__deck_list) - 1))
    indice_draw = self.__card_stats.index[self.__card_stats["Carta"] == draw][0]
    mano.append(draw)
    if self.__card_stats.loc[indice_draw, "Utilidad"] == "Starter":
        self.__starters = self.__starters - 1
    elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Extender":
        self.__extenders = self.__extenders - 1
    elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Defensive":
        self.__defensives = self.__defensives - 1
    elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Combo piece":
        self.__combo_pieces = self.__combo_pieces - 1
    elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Garnet":
        self.__garnets = self.__garnets - 1
    elif self.__card_stats.loc[indice_draw, "Utilidad"] == "Non engine":
        self.__non_engine = self.__non_engine - 1
```

Figura 2: Método draw_card() en Spyder

Estas cartas que el usuario toma del deck pueden guardarse en una variable y con ayuda del método rank_hand() podemos darle una calificación, este método consiste de una secuencia lógica de if's bastante larga, lo que hace es revisar la mano y determinar la cantidad de cartas que contiene la mano para luego aplicar nuestro criterio y darle una calificación, el código correspondiente se puede ver en la siguiente imagen:

```python
if starters >= 1:
    if non_engine >= 1:
        if defensives >= 1:
            if extenders >= 1:
                return "S+"
            else:
                return "A"
        else:
            if extenders >= 1:
                return "B"
            else:
                return "C"
    else:
        if defensives >= 1:
            if extenders >= 1:
                return "S"
            else:
                return "A"
        else:
            if extenders >= 1:
                return "B"
            else:
                return "C"
else:
    if garnets >= 1:
        return "F"
    else:
        return "D"
```

Figura 3: Método rank_hand() en Spyder

Más importante e interesante es el criterio que uso para calificar las manos, para esto se uso nuestro criterio personal basado en que tan importante es cada carta, ya tenemos cada tipo de carta categorizada y ordenada según su importancia, ahora debemos definir cuales combinaciones de cartas son más poderosas o más importantes para ganar una partida, después de un análisis intensivo de las mecánicas del juego y de diversas pruebas se tomo la decisión de establecer seis categorías diferentes para las manos, a continuación podemos ver los detalles de cada categoría:

- S+: La mejor mano que se puede tomar, tiene 1 starter, 1 carta defensiva, 1 extender y 1 una carta non engine, con la última carta siendo de cualquier otro tipo.

- S: Manos que tienen 1 starter, 1 carta defensiva, 1 extender, 0 non engine y las otras dos cartas pueden ser de cualquier otro tipo.
- A: Manos que tienen 1 starter, 1 carta defensiva, 0 extender, y las otras tres cartas pueden ser de cualquier otro tipo.
- B: Manos que tienen 1 starter, 0 cartas defensiva, 1 extender, y las otras tres cartas pueden ser de cualquier otro tipo.
- C: Manos que tienen 1 starter, 0 cartas defensivas, 0 extenders y las otras cuatro pueden ser de cualquier otro tipo.
- D: Toda mano que no tenga 1 starter.
- F: Toda mano que no tenga 1 starter y que contiene 1 o más garnets.

Directamente relacionado al método para calificar manos, tenemos el método eval_deck() que se encarga de darle una calificación al mazo, considerando el énfasis que le hemos dado a la mano inicial, me parece que la calificación del mazo debe estar directamente relacionada a que tan buenas son las manos iniciales que puede producir el mazo y que tan frecuentemente las produce, para esto se tuvo que decidir en un número especifico de manos a evaluar. Para esto debemos hablar un poco del formato de torneos de Yu-Gi-Oh!

En un torneo Yu-Gi-Oh! Championship Series (YCS), considerando más de 512 participantes y un formato en que cada partida la gana el primer jugador en ganar dos duelos (mejor de tres), un jugador debería jugar al menos 15 partidas cada una con un máximo de 3 duelos, entonces se considero una muestra de 45 manos para calificar.

Con la cantidad de manos definida, ahora debemos definir el criterio que calificará el mazo, se decidió basarse puramente en la calidad de manos producidas, y entre mazos que producen manos de similar calidad se considero el formato de mejor de tres del torneo, considerando esto sería muy malo que el mazo produzca dos malas manos seguidas, pues si el jugador pierde dos duelos seguido entonces quedaría eliminado del torneo, entonces los mazos que producen manos de calidad similar se evaluaran dependiendo de si producen manos malas seguidas o no. Esto resulto en un sistema de calificación de 4 categorías, las cuales se van a detallar a continuación:

- Tier 0: los mazos más poderosos, mazos que nunca producen manos de menor calidad que A.
- Tier 1: mazos que producen tanto manos malas como buenas, osea de cualquier tipo, pero nunca producen dos manos peores o iguales a B dos veces seguidas.
- Tier 2: mazos similares a los anteriores, pero que si producen manos peores o iguales a B dos veces seguidas.
- Tier 3: mazos que no producen manos mejores a C.

Similar al método rank_hand() lo más interesante es nuestro criterio para calificar los mazos, en cuanto al código corresponde nuevamente a una secuencia lógica de if's que revisan la lista de manos totales y determina la clasificación dependiendo de la calidad de manos producidas y si produce manos malas seguidas o no. El código correspondiente se puede ver en la siguiente imagen:

```
# Ahora tengo contandas la cantidad de cada tipo de mano que obtuvo en las 45, procedo a revisar si
# el mazo solo produce manos malas osea peores que B, en ese caso solo puede ser Tier 3
if (("B" not in score_manos_total) and ("A" not in score_manos_total)
and ("S" not in score_manos_total) and ("S+" not in score_manos_total)):
    return "Tier 3"
# En caso que si produzca algunas manos buenas entonces se revisa si tiene manos malas, osea se
# se revisa si nunca produce una mano C, D, F o B
if (("B" not in score_manos_total) and ("C" not in score_manos_total)
and ("D" not in score_manos_total) and ("F" not in score_manos_total)):
    return "Tier 0"
# Si ninguno de los dos if anteriores devuelven un valor entonces se puede asumir que el mazo
# produce manos buenas y manos malas (caso usual), será Tier 1 o Tier 2, revisamos si produce manos
# malas seguidas para determinar cual de los dos
for i in range(0, len(score_manos_total)):
    # Solo nos importa si se repiten manos malas entonces reviso si la actual es una mala o buena
    if ((score_manos_total[i] == "C") or (score_manos_total[i] == "D")
    or (score_manos_total[i] == "C")):
        # Revisamos si la siguiente también es mala, como es posible que la mano mala sea la última
        # entonces reviso que i no sea de manera que cause un index out of range
        if (i < len(score_manos_total) - 1):
            if ((score_manos_total[i + 1] == "C") or (score_manos_total[i + 1] == "D")
            or (score_manos_total[i + 1] == "C")):
                return "Tier 2"
# Si se sale del for y no se devuelve el valor Tier 2 entonces el mazo debe ser Tier 1 entonces solo
# hago el return al final afuera del for
return "Tier 1"
```

Figura 4: Método eval_deck() en Spyder

Por último, el método que me parece más importante entre todos, y es el constituye la calculadora hipergeométrica de nuestra herramienta, el método calc_hypergeom(), como hemos aprendido en los cursos de la carrera, la probabilidad

de tomar una carta especifica de un mazo cuando se toma una cantidad especifica de cartas sin devolución se comporta como una distribución hipergeométrica, la cuál se puede calcular con la siguiente formula:

$$P(X = k) = \frac{\binom{K}{k}\binom{N-K}{n-k}}{\binom{N}{n}}$$

Donde:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

N = número de cartas en el mazo.
K = número de cartas que cumplen con la categoría deseada en el mazo.
n = número de cartas que se tomaran del mazo
k = número de cartas de la categoría deseada que se desean tomar del mazo

El método se encarga de tomar los valores necesarios para aplicar esta formula y calcular la probabilidad de éxito solicitada por el usuario, en la siguiente imagen podemos ver el código correspondiente a este método:

```python
def calc_hypergeom(self, util_ideal, cant_deseo, cant_draw):
    util_lower = util_ideal.lower()
    J = getattr(self, util_lower)
    j = cant_deseo
    N = len(self.__deck_list)
    n = cant_draw
    bino_1 = (math.factorial(J)) / ((math.factorial(j)) * (math.factorial(J - j)))
    bino_2 = (math.factorial(N - J)) / ((math.factorial(n - j)) * (math.factorial(N - J - n + j)))
    bino_3 = (math.factorial(N)) / ((math.factorial(n)) * (math.factorial(N - n)))
    prob_exito = (bino_1 * bino_2) / bino_3
    return prob_exito * 100
```

Figura 5: Método calc_hypegeom() en Spyder

Estos me parece que son los métodos más importantes de la herramienta, en cuanto a la construcción de la interfaz, se hizo uso de la libreria Tkinter, la cuál es ideal para la creación de aplicaciones robustas y sencillas. A continuación podemos ver los método utilizados y el formato que se le dio a la herramienta.

Para la herramienta se creo una ventana principal, con cuatro pestañas: Crear mazo, Calculadora hipergeométrica, Modo juego y Modo torneo, cada una de estas pestañas tiene botones y casillas que le permiten al usuario acceder a los métodos anteriores.

Para crear la ventana principal se hace uso del comando Tk(), este se asocia a una variable llamada root, a la cuál se le agregarán todos los demás *widgets*, para las pestañas se crearon cuatro botones con el comando Button() los cuales corresponde a cada una de las pestañas, y la manera en la que funcionan es que al presionar cualquiera de los botones se esconde el contenido que se podía ver en la ventana y se muestra el contenido que se tiene en la pestaña que se selecciono, esto se hace por medio de una función que ejecuta lo establecido anteriormente cuando se presiona alguno de los botones. A continuación podemos ver el código correspondiente a estos elementos:

```python
# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Ventana principal~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
root = tk.Tk()
root.title("Deck Building Tool")
root.geometry("900x650")
pestanas = tk.Frame(root)
pestanas.pack()

# ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~Pestañas~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# Uso Frame() para crear el espacio donde voy a poner los botones y casillas de texto de cada
# pestaña
tab_1_content = tk.Frame(root)
tab_2_content = tk.Frame(root)
tab_3_content = tk.Frame(root)
tab_4_content = tk.Frame(root)

# Creo la función que activa o desactiva las pestañas, en esencia lo que hace es cambiar lo que
# aparece en la ventana cuando se presiona un botón

def show_tab(num_tab):
    tab_1_content.pack_forget()
    tab_2_content.pack_forget()
    tab_3_content.pack_forget()
    tab_4_content.pack_forget()

    if num_tab == 1:
        tab_1_content.pack(fill = tk.BOTH, expand = True)
    elif num_tab == 2:
        tab_2_content.pack(fill = tk.BOTH, expand = True)
    elif num_tab == 3:
        tab_3_content.pack(fill = tk.BOTH, expand = True)
    elif num_tab == 4:
        tab_4_content.pack(fill = tk.BOTH, expand = True)
```

Figura 6: Código que corresponde a la creación de la ventana principal y de las pestañas en Spyder

Además, me parece importante explorar lo que hacen algunos de los comandos más comunes que utilizamos:

- Frame(): crea como un tipo de rectángulo dentro de la ventana principal en donde se podrán insertar widgets como botones y casillas, recibe varios parámetros pero el más importante sería la posición donde se crea el frame y se pone de primero, usualmente se pondría root para que el frame se creé dentro de la ventana principal.

- Label(): corresponde a texto que se inserta en el espacio, admite varios parámetros que están más que todo relacionados al formato del texto escrito.
- Entry(): corresponde a una casilla en blanco donde el usuario puede escribir texto e insertar valores que la herramienta puede guardar, admite varios parámetros relacionados al formato y apariencia de la casilla y del texto que escribe el usuario dentro de ella.
- Button(): corresponde a botones que el usuario puede presionar, estos botones le permiten al usuario ejecutar funciones y métodos con solo presionarlos, admiten varios parametros relacionados al formato y apariencia del botón, pero el más importante sería el parámetro *command* que me permite asociar el botón a funciones y métodos que ya están incluidos en la libreria o de creación propia.
- pack(): este comando se usa para colocar el widget dentro del espacio, igual admite varios parametros, principalmente relacionados a la posición y el espacio entre cada widget que se inserta.

Por último, me parece importante destacar que el texto que el usuario escribe en la casilla no se guarda en una variable inmediatamente después de escribirlo, por lo tanto si necesito esa entrada debo crear una función que tome el texto que está en la casilla y lo guarde en un variable y esta función se debe asociar a un botón, esto resulta en que cada casilla y botón tenga un formato similar al que se ve en la siguiente imagen:

```
ruta_deck = None
# Se crea el label que sirve como instrucción de lo que se debe insertar en la casilla
ins_ruta = tk.Label(tab_1_content, text = "Ingrese la ruta del mazo: ", font = ("Arial", 12))
ins_ruta.pack(side = tk.TOP, pady = 8)

# Se crea la casilla donde el usuario escribe la ruta del deck
cas_ruta = tk.Entry(tab_1_content, width = 80, font = ("Arial", 12), justify = tk.CENTER)
cas_ruta.pack(side = tk.TOP, pady = 8)

# Para guardar el texto se tiene que hacer con una función similar a como se hizo con el mazo.
def guardar_ruta_deck():
    '''
    Método que guarda la ruta del archivo de Excel que contiene la información del deck, la ruta
    el usuario la escribe en una casilla

    '''
    global ruta_deck
    ruta_deck = cas_ruta.get()

# Ahora creo el botón que guarda la ruta
guardar_ruta_btn = tk.Button(tab_1_content, text = "Guardar", bg = "gray", fg = "black",
                             font = ("Arial", 12, "bold"), command = lambda: guardar_ruta_deck())
guardar_ruta_btn.pack(pady = 20)

# Por último creo el botón que se encarga de crear y guardar el deck
crear_deck_btn = tk.Button(tab_1_content, text = "Crear mazo", bg = "gray", fg = "black",
                           font = ("Arial", 12, "bold"), command = lambda: guardar_deck(ruta_deck))
crear_deck_btn.pack(padx = 100, pady = 100)
```

Figura 7: Formato de casillas y botones de Tkinter en Spyder

7. Conclusiones

- Los conocimientos de probabilidad y programación que tenemos son muy útiles para la construcción de mazos de TCG en ambientes competitivos.
- La herramienta construido es bastante útil y la parte más importante es la Calculadora Hipergeométrica.
- Al momento de tomar decisiones, en Yu-Gi-Oh! específicamente, lo principal a tener en cuenta es que la mano inicial es la más importante, por lo que se debe maximizar las probabilidades de tener las cartas más importantes entre esas primeras cinco.
- Los proyectos de programación relativamente sencilla son increíblemente útiles para practicar el razonamiento y la lógica, así podremos aumentar nuestra habilidad de "escribir código en papel"

8. Limites y recomendaciones

- La manera de cargar o leer el mazo es bastante robusta y nada elegante, pero las otras opciones fueron imposibles de implementar debido a la complejidad del formato .YDK que se utiliza para codificar los mazos en las plataformas digitales.
- Establecer conexión con las plataformas digitales de Yu-Gi-Oh! es difícil pero no imposible, lograr esta conexión permitiría al usuario conectar la herramienta con sus cuentas que ya contienen sus mazos.
- Considerando la naturaleza del juego, las opiniones acerca de que tipo de cartas son más importantes siempre será diferente dependiendo de la persona, se recomienda tomar en cuenta diferentes opiniones y hacer pruebas con estas en mente.
- La programación no solamente se trata de escribir código en un compilador, es buena idea mantener las prácticas de codificar en papel y tener a mano hoja y lápiz para apoyarse.