UNIVERSIDAD MICHOACANA DE SAN NICOLAS DE HIDALGO

INSTITUTO DE FÍSICA Y MATEMÁTICAS

OPTIMIZACIÓN EN LA ELECCIÓN DE MAZOS EN JUEGOS DE CARTAS COLECCIONABLES

Tesina para optar al grado de maestro en Ciencias Matemáticas.

Autor:

Gerardo Lauro Maldonado Martínez

Supervisado por:

Dr. Edgardo Roldan Pensado
Centro de Ciencias Matemáticas, UNAM

Morelia, Michoacán
enero de 2019

Resumen

Muchos problemas que uno puede encontrar en la vida cotidiana se reducen a maximizar o minimizar una función; reducir el costo de transporte, aumentar las ganancias en un negocio, obtener el máximo rendimiento de un producto, etcétera. De hecho esta clase de problemas están tan arraigados a nosotros como seres humanos que se han desarrollado una gran cantidad de herramientas para afrontarlos, desde el análisis de clases de funciones especificas hasta el uso de métodos probabilistas, numéricos y heurísticos para aproximar resultados en el caso de no tener información suficiente sobre la función o alguna herramienta determinista para trabajar sobre ésta.
Un problema de esta clase es: Dado un juego en el que no conocemos un algoritmo eficiente para obtener una estrategia ganadora, ¿cómo mejoramos nuestras oportunidades de ganar?
Nosotros nos centramos en un juego muy popular actualmente, llamado *Hearthstone*. Sobre éste se propuso un sistema de calificación para los mazos y se busca, en efecto, maximizar ésta con el objetivo de encontrar la mejor combinación de cartas posible, para así mejorar nuestras oportunidades de ganar. Lo anterior basado en la suposición de que la calificación refleja si un mazo es mejor que otro. Para la búsqueda del máximo puntaje se hizo uso de una ligera variación de un algoritmo llamado *evolución diferencial*, el cual es uno de los clasificados como meta-heurísticas.

genético, algoritmo, heurística, estadística, evolución

Abstract

Most of the problems which we find in our daily life can be reduced to maximize o minimize a function; reduce transport cost, improve business, get the maximum profit of a product. In fact this class of problems are so attached to us as human beings at point we have developed many tools to confront those, examples given, specific function classes analysis and use of probabilities, numeric methods and heuristics to approximate results in the case of not have enough information about the function or a deterministic tool to work over that.

One problem of this type is: If we have a game in which we do not know an efficient algorithm to get a winner strategy, how we improve our chances to win?

We focused in a currently popular game called *Hearthstone*. On this game we proposed a qualification function for decks and we search with this maximize this function to find the best possible combination of cards and with this improve our chances to win. The stated before based in the assumption that the rate returned by the function show if one deck is better than other. For the search of maximum rate we used a variation of an algorithm called *differential evolution*, which is in the group known as meta-heuristics.

Índice general

1. Introducción **1**
1.1. El juego 1
1.2. ¿Qué hay que maximizar? 2
1.3. ¿Cómo maximizar? 3

2. Calificación de Mazos **5**
2.1. Modelado de mazos 5
2.2. Función de calificación 6
2.3. Dificultad de Maximizar Nuestra Función 8

3. Evolución Diferencial **10**
3.1. El Algoritmo 10
3.2. Nuestra Implementación 14

4. Resultados y Conclusiones **16**

Capítulo 1

Introducción

Para comenzar vamos a repasar los puntos fundamentales a tener en cuenta para conseguir nuestro objetivo y en los cuales se basa este trabajo. En la sección 1.1 revisamos el tipo de juego sobre los que se trabajará. No solo en sus reglas básicas sino en ciertas situaciones que nos dan alguna pista de qué podría hacerse en el modelado del problema y en la construcción de una función para calificar los mazos. En la sección 1.2 hablamos superficialmente sobre aspectos a tener en cuenta de la función de calificación, tales como su no unicidad, cambios forzosos a través del tiempo, la importancia de su elección y el dominio de ésta. Continuando en la sección 1.3 se descartan maneras de afrontar el problema de maximizar nuestra función además de sugerir una clase de algoritmos que encajan en nuestras necesidades.

1.1. El juego

Hoy en día los videojuegos en linea son más populares que nunca. Con un alza marcada en su popularidad es normal que muchos géneros ya existentes hicieran su transición a este enfoque en el que jugadores de todo el mundo pueden enfrentarse desde su lugar de residencia a través de Internet. Tal es el caso de los juegos de cartas coleccionables en el que dos jugadores, cada uno con su mazo de cartas, se enfrentan en partidas por turnos en las que buscan (al menos en los más populares) llevar a cero los puntos de vida del contrincante.

El juego en el que nos centramos para este trabajo, desarrollado y publicado por *Blizzard Entertainment*, es *Hearthstone*. Hasta la fecha que se escribe ésto es un juego donde cada jugador puede armar un mazo de exactamente treinta cartas haciendo elección de un "*héroe*" de una de las nueve clases disponibles, donde solo pueden seleccionar cartas (además de que tiene que tenerlas desbloqueadas) de la clase que se eligió y de una clase no arraigada a ningún héroe llamada "neutral". Para la construcción del mazo hay que tomar en cuenta que las cartas se pueden dividir en "normales" y "legendarias", esto es importante puesto que el jugador solo puede tener a lo más dos cartas iguales en el caso

de que sea “normal” y solo una copia de alguna carta en caso de ser “legendaria”. Naturalmente estas condiciones se tendrán en cuenta en la búsqueda de la solución. Para información detallada sobre las reglas del juego véase [9].

Algo que es normal en los juegos del género es que cada cierto tiempo (comúnmente de imprevisto) se liberan actualizaciones para corregir errores y balancear el juego. Esto es importante tomarlo en cuenta porque la eficiencia de algunas cartas sobre otras y sus interacciones cambian constantemente. Lo anterior agregado al hecho de que cada cierto tiempo (actualmente cada 3 meses) se libera un nuevo conjunto de cartas para su uso y cada año se eliminan las cartas que tienen cierta antigüedad. Los cambios constantes en el juego hacen importante que la forma de afrontar el problema los tome en cuenta. Además limitan el uso de ciertas opciones comúnes que dependen del conocimiento de ciertas características de la función en cuestión.

1.2. ¿Qué hay que maximizar?

Ya teniendo presente el juego y el hecho de los cambios constantes en las cartas disponibles, lo subsecuente es hacernos la siguiente pregunta. ¿De qué manera vamos a decidir si un mazo es mejor que otro? Rápidamente podemos decir que lo que necesitamos es una función $f : D \to \mathbb{R}$ que califique un mazo y así los podremos comparar con el orden usual en $\mathbb{R}$.

Un punto a considerarse es el hecho de que la cantidad de mazos posibles es bastante grande por lo cual no tenemos en cuenta hacer una comparación directa de éstos para ordenarlos. Por otro lado, dada la situación del cambio constante en las cartas del juego, no es fácil construir una función que se base en las características individuales de las cartas. Lo que si podemos hacer es uso de las estadísticas de los mazos, por ejemplo, la cantidad de victorias que consigue cada mazo desde la última actualización, el promedio de éstas entre el número de partidas jugadas, etcétera. Con lo último notemos que de hecho requerimos una función $f_t : D \to \mathbb{R}$ que de hecho varia en el tiempo, aunque por conveniencia y por la construcción propuesta que se expone en 2.2 la seguiremos denotando como una única función f.

Antes de decidir como construir la función cabe resaltar la importancia del armado de ésta. De esto depende casi en su totalidad el éxito de nuestros demás esfuerzos por lo que tenemos que evitar la mayor cantidad de fallos posibles. Por ejemplo si decidimos calcular $f(d)$ como el total de victorias de un mazo, es común que aunque sea el más ganador también sea muy utilizado y de hecho sume más derrotas en promedio que uno no tan requerido por los jugadores. Otro ejemplo puede ser el uso directo del promedio de victorias de los mazos, pues un mazo muy poco requerido puede tener un promedio muy alto el cual no reflejaría muy bien su eficiencia, al contar con muy pocas partidas para ser tomado en cuenta. Los anteriores solo son algunos ejemplos sencillos de porque se necesita una función menos obvia para que, en base a las estadísticas, califique nuestros mazos y así poderlos comparar.

Después de ver algunos motivos de porque no se puede depender de las

estadísticas globales de los mazos, veamos ahora el otro extremo. Si utilizamos las estadísticas de cada carta en el mazo para hacer la valoración (por ejemplo con un promedio de la puntuación de cada carta) nos toparíamos con otra clase de problemas. El primero es que ésto no refleja de ninguna manera las interacciones entre las cartas, puesto que mide el rendimiento de cada una, lo cual no consiguen necesariamente solo con las cartas en el mazo en turno. Lo anterior nos conduce a considerar la posibilidad de llevar la puntuación a un punto medio. En lugar de considerar las estadísticas de cada carta en el mazo, podemos tomar en cuenta las estadísticas de las tuplas de tamaño $1 \leq n \leq 30$ y así tener un mejor panorama de como se comportan las cartas entre si. Claro, lo primero que salta a la palestra es el hecho de que $\binom{30}{n}$ puede ser bastante grande para ser procesado, pero en una partida promedio el jugador no cuenta con muchas cartas en juego y en su mano a la vez, así que la consideración de *n-tuplas* con n pequeño (tal que nos permita realizar los cálculos con los recursos disponibles) se esperaría suficiente.

1.3. ¿Cómo maximizar?

Ahora toca hablar un poco sobre la elección del método para maximizar nuestra función de calificación, asumiendo que contamos con ella. En primera instancia hay que mencionar que en la actualidad existen una gran cantidad de algoritmos para optimizar funciones, desde los que hacen uso de la diferenciabilidad (como el descenso por gradiente) hasta los heurísticos como lo son los algoritmos genéticos, entre otros.

Con un amplio catálogo de opciones, ¿cuál parece más adecuada para afrontar nuestro problema? Esta pregunta como casi todas las anteriores y las que están por venir no tiene una respuesta sencilla. Lo primero a considerar es el dominio y codominio de nuestra función. El codominio como ya fue mencionado es $\mathbb{R}$ (aunque de hecho puede ser cualquier conjunto dotado con un orden total) y el dominio es en principio discreto puesto que los mazos posibles son finitos. Así que los métodos de optimización que hacen suposiciones sobre la continuidad no parecen ser de mucha utilidad para nuestro propósito. En el mismo renglón tenemos otro problema pues de hecho al ser la función cambiante a lo largo del tiempo no podemos hacer cualquier suposición sin la posibilidad de que la situación cambie a futuro.

Añadido a nuestro escaso conocimiento de la función hay que tomar en cuenta otra situación. Al querer considerar las estadísticas de *n-tuplas* de cartas en los mazos estamos asumiendo un consumo considerable de recursos (memoria y tiempo) por lo cual necesitamos un algoritmo que utilice muy pocos recursos en hacer los cálculos que no involucran evaluaciones en f, para así compensar el gasto y tener una herramienta funcional.

Por éste motivo y otro que se expone en la sección 2.3, se tomó la decisión de utilizar un algoritmo evolutivo, específicamente uno llamado ***Evolución Diferencial***. En el capítulo 3 se hace una descripción detallada del algoritmo de los considerados como meta-heurísticas, por ahora cabe mencionar que cumple con

nuestras condiciones deseadas: cuenta con buenos resultados, es simple, eficiente con los recursos, no necesita ninguna suposición sobre la función y aunque concebido originalmente para funciones de $\mathbb{R}^n$ en $\mathbb{R}$, es directamente adaptable a un dominio discreto encajado.

Capítulo 2

Calificación de Mazos

Como ya se mencionó en el capítulo 1 la construcción de la función con la que se hará la valoración de los mazos es de suma importancia. Cuando tratamos un problema fuera de la teoría es natural que el modelado juegue el papel más importante, puesto que de una buena interpretación depende que todos nuestros esfuerzos dentro de ésta den resultados. En la sección 2.1 se explica cómo llevamos los mazos a un conjunto en el que podemos aplicar nuestro algoritmo y además vemos las cotas que delimitan el conjunto de mazos dentro de éste. En la sección 2.2 se muestra de manera explicita la función de calificación que se usará haciendo especial énfasis en una de sus partes, la función que recolecta las estadísticas de las n-tuplas de cartas.

2.1. Modelado de mazos

Comenzamos esta sección fijando el hecho de que tenemos cada carta enumerada y contamos con $D \in \mathbb{N}$ cartas diferentes para elegir. Además como se dijo en la sección 1.1 hacemos la distinción entre cartas legendarias y no legendarias, por lo que suponemos que la enumeración tiene primero a las no legendarias y después a las legendarias, es decir, las cartas con índice $i < j$ son no legendarias y las que cuentan con índice $i \geq j$ los son, con $j \in \mathbb{N}$ el índice de la primera carta legendaria disponible.

El algoritmo que elegimos fue diseñado originalmente para funciones $f : A \subset \mathbb{R}^D \to \mathbb{R}$ por lo cual es natural llevar nuestros mazos a un subconjunto $A \subset \mathbb{R}^D$ para poder utilizarlo. Basados en esto consideramos cada mazo como un elemento en $x \in (\mathbb{Z}_{\geq 0})^D$ tal que $x_i = n$ si la *i-ésima* carta tiene n copias en el mazo. Para efectos de la notación denotamos el conjunto de mazos como $M \subset (\mathbb{Z}_{\geq 0})^D$.

La primera consecuencia de representar los mazos de esta manera es que en nuestro caso particular ninguna entrada será menor a cero ni mayor a dos (uno en caso de las legendarias), es decir, si j es el índice de la primera carta

legendaria entonces para cada $x \in M$ y $1 \leq i \leq D$ se cumple que

$$l_i \leq x_i \leq h_i, \tag{2.1}$$

con $l = (0, 0, \ldots, 0)$ y $h = (2, 2, \ldots, 2, 1, \ldots, 1)$ donde el primer índice donde aparece un uno es j.

Pero las cotas puestas por l y h no son las únicas, puesto que solo podemos llevar una cantidad de cartas determinada $L \in \mathbb{N}$. Entonces cada mazo x debe cumplir que

$$\sum_{i=1}^{D} x_i = L. \tag{2.2}$$

Éstas son las únicas dos condiciones que delimitan nuestro conjunto por lo que para mantener cierta generalidad podemos decir que un conjunto de mazos $M \subset (\mathbb{Z}_{\geq 0})^D$ es aquel en el que todos sus elementos cumplen con las condiciones (2.1) y (2.2).

Además de la facilidad con la que se pueden revisar las dos condiciones anteriores con este modo de ver las barajas, notemos que al ser M un subconjunto de $\mathbb{R}^D$, éste ha sido dotado de propiedades geométricas y algebraicas de las cuales hace uso el algoritmo explicado más adelante.

2.2. Función de calificación

Ya fijado el dominio de nuestra función solo queda realizar su definición. En primer lugar hay que recordar que queremos basar la calificación de los mazos no solo en el rendimiento de las cartas, sino que también queremos hacerlo en las interacciones de estas, por lo cual es conveniente fijar una notación que nos sea de utilidad para ello.

Consideremos un elemento $x \in (\mathbb{Z}_{\geq 0})^D$. Una n-tupla t de x, es un multiconjunto de tamaño n tal que contiene índices de x que son mayores a cero pero sin tener más de x_i copias de i. Siendo más específicos, si $s \in t$ entonces $x_s > 0$ y si tenemos i copias de s en t entonces $x_s \geq i$. También definimos como C_x^n al conjunto de todas las n-tuplas de x.

Para comenzar con la construcción de nuestra función calificadora necesitamos en primer lugar otra función la cual devuelva la "puntuación" de las tuplas, es decir, una función $g_n : C_x^n \to \mathbb{R}$. Puesto que queremos considerar las interacciones entre las cartas, en nuestro caso nos inclinamos por utilizar la cantidad de partidas ganadas entre la cantidad de partidas donde la tupla aparece.

Esta definición aunque conveniente y natural tiene un problema muy notorio. Lo que ocurre es que no todas las tuplas aparecen en la misma proporción en las partidas, así que podemos tener calificaciones altas de tuplas que aparecen muy poco y esto no reflejaría muy bien su efectividad. Para corregir esto lo que hacemos es básicamente tomar las n-tuplas que no tienen suficientes partidas y recalificarlas en base al rendimiento que tuvieron las $(n-1)$-tuplas que se pueden conformar con las cartas en la tupla en cuestión. Por ejemplo, si contamos con la

3-tupla $\{12, 3, 6\}$ y no cuenta con las suficientes partidas para su consideración, entonces la calificaremos en base al puntaje obtenido por $\{12, 3\}$, $\{12, 6\}$ y $\{3, 6\}$. En el caso especifico de que se tenga $n = 1$ eliminamos la carta en cuestión puesto que no tenemos información suficiente para evaluarla.

Antes de continuar cabe hacer una acotación que aplica a nuestro juego específicamente. En *Hearthstone* existen tarjetas que para realizar alguna acción necesitan que en el mazo se encuentren cartas con ciertas características y que por ejemplo pueden quedar inutilizables de haber alguna carta que no cumpla la condición. Este tipo de circunstancias hacen que sea más conveniente no realizar este llenado recursivo de los puntajes. Puede que parezca algo muy especifico del juego pero ésto ocurre en esencia porque calificar una pareja de cartas en base a sus estadisticas individuales no refleja de ninguna forma alguna interacción, a diferencia del caso $n > 2$ donde al menos tenemos interacciones de parejas. Aunque un problema, en la implementación estas opciones se encuentran disponibles.

Hecha la acotación, vamos a aterrizar la definición de la función g_n que utilizamos. Sean $t \in C_x^n$ una tupla de cartas del mazo x, C_t^{n-1} el multiconjunto de $(n-1)$-tuplas formadas con elementos de t (cuando $n > 1$), $P_m \in [0, 1]$ la proporción mínima requerida para que la tupla conserve su calificación base, $P_t \in \mathbb{N}$ el total de partidas en nuestra base de datos, $t_p \in \mathbb{N}$ el total de partidas jugadas por t y $t_v \in \mathbb{N}$ el total de partidas ganadas por t. Entonces primero dejamos de considerar las cartas con menos de $P_m \times P_t$ partidas jugadas. Realizado ésto podemos definir g_1 simplemente como

$$g_1(t) = \frac{t_v}{t_p}.$$

En segundo lugar definimos g_n para $n > 1$ como

$$g_n(t) = \begin{cases} t_v/t_p, & \text{si } t_p \geq P_m \times P_t \\ r_n(t), & \text{en otro caso.} \end{cases}$$

Solo queda definir $r_n : C_x^n \to \mathbb{R}$ (para $n > 1$) que hace el paso recursivo mencionado anteriormente de la siguiente manera

$$r_n(t) = \left(\prod_{y \in C_t^{n-1}} g_{n-1}(y) \right)^{(1/|C_t^{n-1}|)}.$$

Aquí hacemos uso de la media geométrica para considerar las valuaciones de las $(n-1)$-tuplas en g. Esto se hace para evitar que si varias $(n-1)$-tuplas tienen valores en g muy bajos el valor total de $g(t)$ sea de todos modos alto debido a otra $(n-1)$-tupla que se califica muy bien, es decir, para mantener un mejor balance.

Ya definida nuestra función g hay que detenernos a revisar algunas cosas. La primera es que hacemos (y haremos) un abuso de notación al llamar g a la g_n requerida por la cardinalidad de la tupla a evaluar. También hay que notar

que la eliminación (de ser necesaria) de cartas que no cumplan con la cantidad mínima de partidas no cambia nada de lo anterior dicho en este capítulo y anteriores pues solo reduce el número D de cartas disponibles.

Dicho esto pasamos ahora a terminar la sección definiendo f. Para unir los valores que devuelve g hacemos uso de una norma p para contrastar, de un modo que podamos manipular, el valor de cada tupla. En fin, definimos $f_n : M \to \mathbb{R}$ como

$$f_n(x) := \sum_{t \in C} g(t)^p$$

donde $C = C_x^n$ y $p \in (0, \infty)$. Nótese que omitimos calcular al final la raíz p-ésima (para los p que lo admiten) para ahorrar tiempo de ejecución. Además de la definición anterior nos podemos permite definir

$$f_n(x) := max_{t \in C}\{g(t)\}$$

aludiendo a la norma para $p = \infty$.

Para terminar, es importante decir lo costoso que puede llegar a ser este procedimiento para valores altos de n, lo cual resalta la importancia de la simpleza, en su procedimiento, del algoritmo que abordamos en el capítulo 3.

2.3. Dificultad de Maximizar Nuestra Función

Ya definida nuestra función para calificar los mazos vale buscar como maximizarla. Llamemos D_p al número total de cartas, considerando dos copias de cada carta no "legendaria". Hacer la "fuerza bruta", precisamente revisar el valor en f_n de cada mazo posible, es impensable pues se tendrían que revisar los $\binom{D_p}{30}$ mazos.

Veamos que ocurre en el caso $n = 1$. De solo considerarse 1-tuplas, solamente los valores de las cartas individuales, basta con ordenar las cartas por su valor en g_1. Haciendo lo anterior solo hay que tomar las mejores treinta cartas (considerando dos copias de las que no sean "legendarias") para armar el mazo con valor máximo en f para cualquier $p \in [0, \infty]$. La idea no prospera al aumentar el valor de n. Para $n = 2$, al ordenar las todas las parejas de cartas, elegir las treinta que aparezcan en los pares más altos no nos asegura tener un mazo con valor en f_2 máximo. De hecho ésta búsqueda nos lleva a plantear el problema de otro modo, para que exponer la dificultad del problema.

Construyamos una gráfica completa pesada en la que cada vértice represente una carta disponible. En cada arista uv le asignamos como peso el valor $g(\{u, v\})^p$ para $p \in [0, \infty)$. Entonces si L es el número de cartas que debe tener un mazo (treinta en éste caso) el problema, en este contexto, es encontrar la subgráfica completa con L vértices de mayor peso.

El problema como se plantea anteriormente de buscar el K_L de mayor peso no es sencillo. Al ser una generalización del problema de buscar en una gráfica simple un clique de tamaño L, es un problema **NP-difícil** [8]. Para el caso $n > 2$ la generalización es en linea recta. Consideremos una hipergráfica pesada

$$G = (V, E, W, n)$$

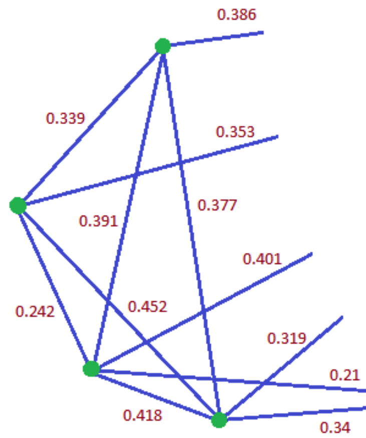

Figura 2.1: Ejemplo del modelado en una gráfica pesada.

donde el peso de una n-arista $u_1u_2...u_n$ es $g(\{u_1u_2...u_n\})^p$.

Dicho lo anterior, tenemos en paralelo nuestro problema a otro bastante complicado. Se conocen algoritmos para aproximar resultados bastante buenos para el caso $n = 2$, pero en nuestra búsqueda de una solución para casos $n > 2$ hemos decidido tomar otro enfoque el cual veremos en el capítulo siguiente.

Capítulo 3

Evolución Diferencial

La búsqueda de algoritmos para encontrar extremos globales de funciones se ha vuelto muy popular en los últimos años dados los grandes beneficios aplicados que ofrecen. Pero ésta búsqueda está lejos de ser resuelta puesto que existen una infinidad de funciones que aparecen no solo en la teoría sino también en la práctica que son muy dispares en propiedades como para tener un análisis generalizado, o que de plano ni siquiera se tiene un conocimiento suficiente de ellas como para hacer un análisis particular (como es en nuestro caso). Debido a ésto los algoritmos considerados meta-heurísticas se han convertido, a la par del problema, en herramientas populares para afrontar estos retos, debido a su falta de necesidad de requerir propiedades especificas sobre las funciones que buscan maximizar o minimizar. Esto último claro con la desventaja de que no aseguran siquiera un rendimiento en sus resultados.

En este capítulo se describe el algoritmo que se eligió para realizar la tarea de buscar el mazo con máxima valuación en f. Comenzando en la sección 3.1 presentamos la versión primigenia del algoritmo propuesto originalmente por Kenneth Price y Rainer Storn [1], además de agregar un parámetro aleatorio a la población para asegurar la convergencia en probabilidad del algoritmo. Continuamos con la sección 3.2 donde hacemos hincapié en las modificaciones necesarias para aplicar el algoritmo a nuestro problema así como otra modificación con la cual creemos que mejora su fidelidad.

3.1. El Algoritmo

Comenzamos la descripción del algoritmo resaltando, como se mencionó en la sección 2.1, que aunque fue diseñado para espacios reales, su implementación solo necesita que la función a maximizar (o minimizar) tenga por dominio un R-módulo libre [5] y por imagen un conjunto dotado con un orden total (para la comparación). Además aunque en su versión original y en nuestra implementación también se necesita que el subconjunto M del dominio a analizar esté acotado por dos elementos l y h en el dominio, tales que $\forall m \in M$ se tenga que

$l_i \leq m_i \leq h_i$ para $1 \leq i \leq D$, esto no es necesario.

En lo que resta supondremos sin perdida de generalidad que lo que se busca es maximizar una función. Para la descripción que se hace a continuación consideremos $f : M \subset \mathbb{R}^D \to \mathbb{R}$ la función que cumple que existe al menos un elemento máximo, es decir, existe $x \in M$ tal que para cada $y \in M$ tenemos que $f(x) \geq f(y)$. También requerimos elementos l y h que cumplen la condición antes descrita.

Dicho esto listamos los parámetros que utiliza el algoritmo.

- $C_d \in \mathbb{R}^+$ la constante de diferenciación.
- $C_m \in (0, 1)$ la constante de mutación.
- $N_g \in \mathbb{N}$ el número de generaciones.
- $N_p \in \mathbb{N} \setminus \{1, 2, 3\}$ el número de elementos en la población.

El algoritmo comienza eligiendo una población inicial $P \subset M$ de tamaño N_p de manera aleatoria. Hecho esto, recorre cada elemento $x \in P$ N_g-veces, donde en cada paso realiza el siguiente proceso:

1. (Elección). Se eligen de manera aleatoria tres elementos $w, y, z \in P$ tal que x, w, y y z sean distintos.

2. (Mutación). Se elige un índice $1 \leq j \leq D$ y después construimos un elemento $x' \in \mathbb{R}^D$ entrada por entrada como sigue:

$$x'_i = \begin{cases} z_i + C_d(y_i - w_i), & \text{Si } \alpha < C_m \text{ o } j = i \\ x_i, & \text{en otro caso.} \end{cases}$$

   Donde la asignación se realiza con probabilidad $\alpha \in [0, 1]$ cuando $j \neq i$.

3. (Normalización). Se verifica si x' se encuentra en M, de no ser el caso se revisa que cada entrada x'_i se encuentre entre l_i y h_i y de no ser así se remplaza por un elemento aleatorio en $[l_i, h_i]$.

4. (Mejora). Para concluir se comparan sus valores en f, más específicamente, si $f(x') > f(x)$ se remplaza x por x' en P.

Al terminar de repetir este proceso N_g veces tomamos el elemento de la población con mayor valuación en f, en efecto, tomamos $x \in P$ tal que $f(x) \geq f(y)$ para cada $y \in P$ y con esto terminamos.

Detengámonos a analizar algunas cosas. La creación de x' en el paso número 2 tiene un costo constante en tiempo, así que el algoritmo se ejecuta en orden de la cantidad de generaciones multiplicada por N_p y el costo C_f de evaluar un elemento en la función. En el mismo sentido cabe destacar también la simpleza del algoritmo en un proceso que no supone ninguna complicación en su ejecución. Para tener otra descripción del algoritmo presentamos un pseudo código en 3.1.1.

Algoritmo 3.1.1: Evolución Diferencial.

Entrada: Un subconjunto A de $\mathbb{R}^D$, una función $f : A \to R$ y un par de elementos $l, h \in \mathbb{R}^D$. Además de los parámetros N_g, N_p, C_m y C_d.

Salida: Un elemento $x \in A$ candidato a máximo.

1 $P = elementos_aleatorios(A, N_p)$
2 **para** $1 \leq i \leq N_g$ **hacer**
3   **para** $x \in P$ **hacer**
4   **repetir**
5             $w \leftarrow elemento_aleatorio(P)$
6   **hasta que** $w \neq x$;
7   **repetir**
8             $y \leftarrow elemento_aleatorio(P)$
9   **hasta que** $y \neq x$ y $y \neq w$;
10   **repetir**
11             $z \leftarrow elemento_aleatorio(P)$
12   **hasta que** $z \neq x$, $z \neq w$, $z \neq y$;
13         $j \leftarrow entero_aleatorio(1, D)$
14   **para** $1 \leq i \leq D$ **hacer**
15             $x'_i \leftarrow z_i + C_d(y_i - w_i)$
16             $\alpha \leftarrow aleatorio(0, 1)$ **si** $\alpha \geq C_m$ *o* $j = i$ **entonces**
17                 $x'_i \leftarrow x_i$
18   **fin**
19   **si** $l_i > x'_i$ *o* $h_i < x'_i$ **entonces**
20                 $x'_i \leftarrow aleatorio(l_i, h_i)$
21   **fin**
22   **fin**
23   **si** $f(x') > f(x)$ **entonces**
24             $P_x \leftarrow x'$
25   **fin**
26   **fin**
27 **fin**
28 $r \leftarrow elemento_aleatorio(P)$
29 **para** $x \in P$ **hacer**
30   **si** $f(x) > f(r)$ **entonces**
31         $r \leftarrow x$
32   **fin**
33 **fin**
34 **devolver** r

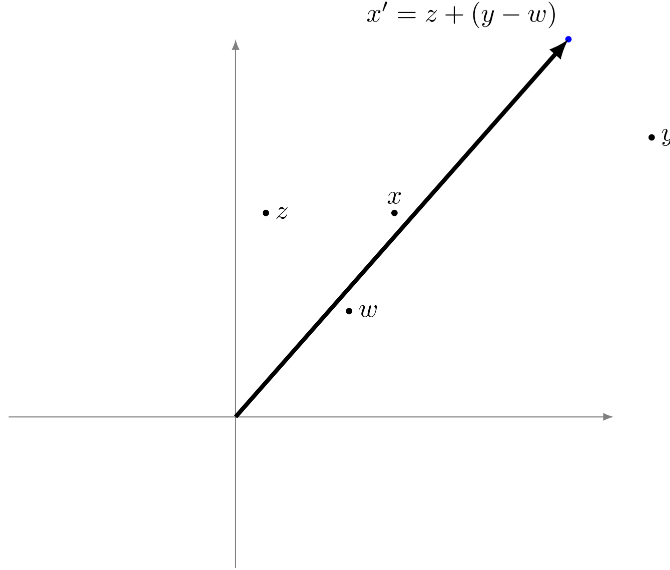

Figura 3.1: Creación de x' antes de la recombinación

Expuesto el algoritmo original queda claro que hay que realizar algunas modificaciones en el para adaptarlo a nuestro problema en concreto. Estos cambios se muestran en la siguiente sección. Para terminar la presente se muestra una pequeña modificación al algoritmo que busca mantener la diversidad de los elementos en la población. Esta modificación básicamente consiste en mantener un porcentaje de la población aleatoria en cada generación, pero sin olvidar el objetivo inicial, es decir, remplazando los elementos con menor valor en f.

El cambio sobre el algoritmo consiste en dada una razón $P_a \in [0, 1]$, al terminar cada generación se remplacen los $\lfloor (P_a) \times N_p) \rfloor$ elementos con menor valuación en f por elementos aleatorios en M. Para ésto es conveniente que al terminar las mutaciones se mantenga ordenado P en base a f. Con ésto se asegura la convergencia en probabilidad a una solución porque eventualmente se elegirá de manera aleatoria al máximo global. Más importante aún, se tiene una mayor diversidad en la población, lo cual es útil para evitar que la población se estanque en su búsqueda de mejorar sus elementos.

Para hablar del estancamiento hay que analizar el proceso de mutación, precisamente el armado de x' antes de la recombinación coordenada a coordenada con x (figura 3.1). Supongamos que nuestra función cuenta con dos máximos locales muy separados llamados m_l y m_g, y que aunque nuestra población inicial está bien distribuida se ha comenzado a inclinar en dirección al máximo no global m_l como en la figura 3.2. Notemos que aunque se cuente con una población relativamente grande, x' tiene limitados resultados posibles (a lo más $\binom{|P|-1}{3} \times 6$) y con una constante de diferenciación $C_d \leq 1$ (la cuál es recomen-

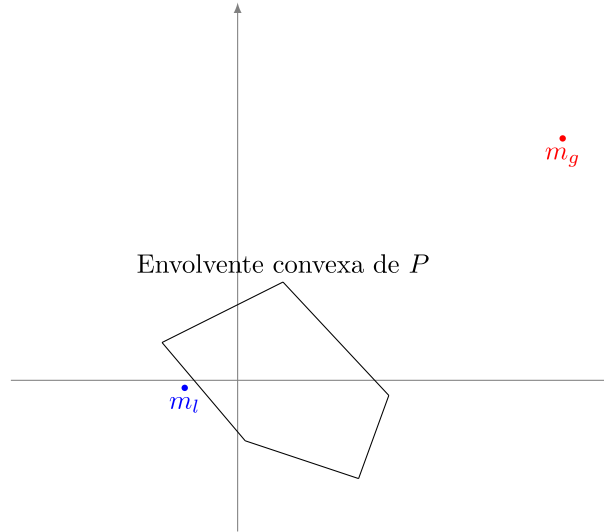

Figura 3.2: Ejemplo de estancamiento.

dable para tener una convergencia en ciertos casos) no puede ir muy lejos de la envolvente convexa de P. Por tanto nuestra población estaría estancada pues aunque podría seguir mejorando en dirección al máximo local m_l cercano, no puede ir en dirección al máximo global m_g pues se encuentra fuera de su alcance. Al tener un cierto porcentaje de la población elegida de manera aleatoria, se busca eludir esta situación, es decir, que en efecto tengamos la posibilidad de salir de estos “baches”. Con elementos aleatorios en la población, la creación de x' puede efectuarse con cierta probabilidad en alguna vecindad cercana al máximo global en cualquier momento, pero sin dejar de lado el proceso principal del algoritmo. Lo último se intenta conseguir haciendo el remplazo justo por los elementos con menor valor $f(x)$.

3.2. Nuestra Implementación

Ya descrito el algoritmo elegido para buscar seleccionar los mejores mazos cabe incluir la descripción de los cambios que se incluyeron para la implementación especifica en nuestro problema y uno que se agrega con la intención de mejorar la fidelidad al problema y por tanto los resultados.

Comenzamos con la adaptación recordando que los mazos son representados como elementos $x \in \mathbb{N}^D$ acotados por $l = (0, \ldots, 0)$ y $h = (2, \ldots, 2, 1, \ldots, 1)$ donde hay algún $1 \leq j \leq D$ tal que h_j es el primer 1 de h. Ahora dada la restricción de entradas en $\{0, 1, 2\}$ la constante de diferenciación es conveniente no solo restringirla a valores en $\mathbb{N}$ sino que dejarla como 1. Esto debido a que de ser mayor que 1 no habría distinción entre cuanto se alejan las entradas de x' de las

entradas de h, además que ésto ocurriría en todos los casos. El segundo cambio es que los mazos no solo tienen que cumplir tener sus parámetros acotados sino que la cantidad de cartas es constante, digamos $L \in \mathbb{N}$. En términos de nuestra representación debe ocurrir que

$$\sum_{i=1}^{D} x_i = L.$$

Para mantener nuestros mazos cumpliendo la condición anterior basta con agregar ó quitar las cartas restantes, proceso que podemos realizar como parte de la normalización de x'. De hecho la normalización está directamente relacionada con la última modificación que proponemos para la adaptación.

Notemos que en el algoritmo 3.1.1, a la par de la mutación, si el nuevo elemento x' tiene parámetros fuera de los limites se elige un nuevo elemento al azar para reemplazarlo. Ésto lo remplazamos regresando el parámetro al elemento más cercano en el rango permitido justo al terminar la mutación, con el fin de que el elemento x' sea lo más parecido posible al resultado de su mutación. Las últimas dos modificaciones quedan descritas a continuación.

- Revisamos cada coordenada x'_i de x' y redefinimos este último como sigue

$$x'_i = \begin{cases} l_i, & \text{Si } x'_i \leq l_i \\ h_i, & \text{Si } x'_i \geq h_i \\ x'_i, & \text{En otro caso.} \end{cases}$$

- Vemos cuantas cartas tiene el mazo, es decir, $\sum_{i=1}^{D} x'_i$. Si $\sum_{i=1}^{D} x'_i > L$ retiramos de manera aleatoria cartas hasta tener la igualdad, respetando que $x'_i \geq l_i$. En el caso de que $\sum_{i=1}^{D} x'_i < L$ las añadimos respetando $x'_i \leq h_i$.

Una de las grandes ventajas del algoritmo original es una de la cual en este punto el lector ya se ha percatado. Ésta es que las diferencias entre éste y el algoritmo resultante es muy sutil, así que en realidad la adaptación es bastante sencilla.

Capítulo 4

Resultados y Conclusiones

Ya expuestas las tres partes de las que consta el trabajo (modelado de los mazos, construcción de la función y el algoritmo) y discutidas las decisiones que se tomaron sobre éstas, lo que queda es presentar los resultados de la ejecución del algoritmo y como nos ha resultado en el campo de juego.

La manera en la que presentamos los resultados es sencilla. Las tablas 4.2, 4.3 y 4.1 muestran el número del mazo (cuya asignación puede verse en la tabla 4.4), el valor de n que representa la cantidad de elementos en las tuplas, el valor de p para la norma, el valor que devuelve f para el mazo y para las que tenemos disponibles, resultados. En ésta última columna presentamos el porcentaje de victorias que se obtuvo al jugarlos o su porcentaje de victorias en `HSReplay.net`. Los resultados corresponden a la ejecución realizada el 28 de Septiembre del 2018. La cantidad de partidas que jugamos es pequeña debido a que no contamos con la capacidad para jugar una cantidad significativa de veces, además de que no pudimos hacerlo con todos los mazos debido a que se necesitan armar y no contamos con las cartas para hacerlo.

Cabe mencionar que el código que se encuentra en la tabla 4.4 después de la dirección `https://deck.codes/`, es el código del mazo que asigna Hearthstone en base 64. Se incluyó el enlace a dicha web para su mejor visualización.

Las ejecuciones del algoritmo fueron realizadas con los siguientes parámetros

- $C_m = 0.2$.
- $Np = 40$. Con un solo elemento aleatorio en la población.

y se corrían tres veces por cada clase. Aunque el número de generaciones con el que ejecutábamos era alto, consideramos que la población se estabilizaba alrededor de las 500 iteraciones. Lo anterior es sorprendente porque el rendimiento fue mejor al esperado originalmente y fue bastante conveniente dado el costo de evaluar la función para cuartetas y quintetas.

Expuestos los resultados obtenidos queda como conclusión que aunque los resultados son bastante favorables, lo que sigue es la publicación del programa para el uso público. Esto con el propósito de ver como funciona con una cantidad

mayor de pruebas, además del largo del tiempo. El proyecto se encuentra actualmente alojado en `https://github.com/XGEu2X/Hearthstone-DeckFinder` con éste fin.

n	**p**	**Mazo**	**f(x)**	**Resultados**
Warrior				
WR1	3	0.3	3436.19	-
	3	1	2328.32	
	3	3	766.179	
WR2	4	0.3	23219.8	58.1 %[1]
	4	1	15773.9	
	4	3	5227.17	
	5	0.3	120743	
	5	1	82186	
	5	3	27388.7	
Hunter				
H1	3	0.3	3426.86	58.4 %[1]
	3	1	2308.64	
	4	0.3	23154.6	
	4	1	15629.4	
	5	0.3	120433	
	5	1	81445.1	
	5	3	26645.6	
H2	3	3	747.127	-
H3	4	3	5094.76	-
Mage				
M1	3	0.3	3392.36	56.5 % [1]
	3	1	2230.74	
	4	0.3	22910.2	
	5	0.3	119176	
	5	1	78527.1	
M2	3	3	688.527	-
M3	4	1	15221.7	70 %
	4	3	4773.81	
	5	3	25017.6	

Tabla 4.1: Mazos arrojados para la cada clase.

[1] En `HSReplay.net`.

Mazo	**n**	**p**	**f(x)**	**Resultados**
Druid				
D1	3	0.3	3422.02	65 %
	3	3	734.982	
	4	1	15547.8	
	4	3	5005.33	
	5	0.3	120287	
D2	3	1	2296.48	-
	4	0.3	23119.5	
	5	1	80995.1	
	5	3	26167.3	
Paladin				
PA1	3	0.3	3452.17	-
	3	1	2364.6	
	4	0.3	23321.3	
	4	1	16004.7	
PA2	3	3	805.96	50 %
	4	3	5493.02	
	5	1	83391.3	
	5	3	28689.4	
PA3	5	0.3	121321	-
Rogue				
R1	3	0.3	3425.23	58.1 % [1]
	3	1	2303.62	
	3	3	741.687	
	4	0.3	23134.1	
	4	1	15580.4	
	4	3	5036.47	
	5	0.3	120354	
	5	1	81145.8	
	5	3	26314.4	

Tabla 4.2: Mazos arrojados para la cada clase.

n	**p**	**Mazo**	**f(x)**	**Resultados**
Priest				
PR1	3	0.3	3406.87	58.1 % [1]
	3	1	2263.14	
	3	3	704.384	
	4	0.3	23049.8	
	4	1	15393	
	4	3	4859.94	
	5	0.3	119996	
	5	1	80347.2	
	5	3	25552.2	
Shaman				
S1	3	0.3	3433.18	57.7 % [1]
	3	1	2321.5	
	3	3	759.147	
S2	4	0.3	23194.3	59.1 % [1]
	4	1	15716.2	
	4	3	5170.14	
	5	0.3	120713	
	5	1	81957.2	
	5	3	27115	
Warlock				
WL1	3	0.3	3419.95	58.7 % [1]
	3	1	2292.03	
	3	3	731.202	
	4	0.3	23121.7	
	4	1	15554.4	
	5	1	81267.6	
WL2	4	3	5020.69	57.8 % [1]
	5	0.3	120387	
	5	3	26480.4	

Tabla 4.3: Mazos arrojados para la cada clase.

D1	https://deck.codes/AAECAZICAl+Z0wIOQPOC9wPmBYUI5AigzQKHzgKY0gKe0gLb0wKE5gLX7wLi+AIA
D2	https://deck.codes/AAECAZICAuQImdMCDkBf/QL3A+YFhQigzQKHzgKY0gKe0gLb0wKE5gLX7wLi+AIA
H1	https://deck.codes/AAECAR8Gxw0uBp3MAobTApjwAqeCAwyeAagCtQOHBMkE6wf+DI7DAt3SAt/SAuPSAovlAgA=
H2	https://deck.codes/AAECAR8GxwPJBK4GhtMCmPACp4IDDJ4BqAK1A4cE6wf+DI7DAp3MAt3SAt/SAuPSAovlAgA=
H3	https://deck.codes/AAECAR8EncwChtMCmPACp4IDDZ4BqAK1A8cDhwSuBusH/gy0wwLd0gLf0gLj0gKL5QIA
M1	https://deck.codes/AAECAf0EBHHtBaLTAu72Ag27ApUDqwS0BOYElgXsBcHBApjEAo/TAvvsApX/Arn/AgA=
M2	https://deck.codes/AAECAf0EApvTAqPrAg5NyQPeBewH+wzKwwLTxQKWxwKbywKgzgLb0wLX4QKW5ALi+AIA
M3	https://deck.codes/AAECAf0EBKDOAsLOApvTAqPrAg1NyQPeBewH+wzKwwLTxQKWxwKbywLb0wLX4QKW5ALi+AIA
PA1	https://deck.codes/AAECAZ8FAqICnvgCDkanBfEF9QX5CpvCAuvCArjHAuPLApXOAvvTAtHhAtblArXmAgA=
PA2	https://deck.codes/AAECAZ8FAqICnvgCDkanBfEF9QWbwgKDxwK4xwLjywKVzgL70wLR4QKL5QLW5QK15gIA
PA3	https://deck.codes/AAECAZ8FAoPHAp74Ag5GpwXxBfUF+QqbwgLrwgK4xwLjywKVzgL70wLR4QLW5QK15gIA
PR1	https://deck.codes/AAECAa0GAsUEkNMCDpACoQTlBPYHjQjyDPsM0cECyccC6NACy+YC/0oCvfMC4vgCAA==
R1	https://deck.codes/AAECAaIHBKICsgKvBJ74Ag2MAssD1AX1Bd0IgcICn8IC68IC0eECi+UCpu8Cx/gC4vgCAA==
S1	https://deck.codes/AAECAafuAgggmQL+BfPCAsLOAvbsAqfuAs30Agu9AdMB2QfwB7EIkcECrMICm8sClugClO8CsPACAA==
S2	https://deck.codes/AAECAafuAgggmQL+BfPCAsLOAvbsAqfuAs30Agu9AdMB2QfwB7EIkcECm8sClugClO8CsPACioADAA==
WL1	https://deck.codes/AAECAf0GAq8En0ICDjCEAfIFzgfCCPcMysMCm8sC980Cn84C8tAC0eECh+gC7/ECAA==
WL2	https://deck.codes/AAECAf0GArYHn0ICDjCEAfIFzgfCCPcMysMCm8sC980Cn84C8tAC0eECh+gC7/ECAA==
WR1	https://deck.codes/AAECAZL4AgZL08MC08UCz8cCnvgCoIADDKICogTeBf8Hm8ICysMCoscCyucCquwC4vgCjvsCnvsCAA==
WR2	https://deck.codes/AAECAZL4AgaiAtPDAtPFAs/HAp74AqCAAwxLogTeBf8Hm8ICysMCoscCyucCquwC4vgCjvsCnvsCAA==

Tabla 4.4: Códigos de los mazos resultantes.

Bibliografía

[1] R. Storn, K. V. Price, *Differential evolution - a simple and efficient adaptive scheme for global optimization over continuous spaces*, Journal of global optimization, 1997, vol. 11, no 4, pp. 341–359.

[2] R. Storn, K. V. Price, *Minimizing the real functions of the ICEC 1996 contest by differential evolution*, Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 842–844.

[3] V. Feoktistov, *Differential Evolution in Search of Solutions*, Springer, First Edition, 2006.

[4] D. H. Wolpert, W. G. Macready, *No Free Lucnh Theorem for Optimization*, IEEE transactions on evolutionary computation, 1997, vol. 1, no 1, pp. 67–82.

[5] N. Jacobson, *Basic Algebra I*, Dover, Second Edition, 2009.

[6] U. K. Chakraborty, *Advances in Differential Evolution*, Springer, First edition, 2008.

[7] G. C. Onwubolu, D. Davendra, *Differential Evolution: A Handbook for Global Permutation-Based Combinatorial Optimization*, Springer, First edition, 2009.

[8] Sanjev Arora, Boaz Barak, *COMPUTATIONAL COMPLEXITY A Modern Approach*, Cambridge University Press, First edition, 2009.

[9] How To Play, Hearthstone Official Game Site, `https://playhearthstone.com/en-us/game-guide/`.