UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Modelo de bases de datos e inteligencia artificial para el juego de cartas coleccionables 'Magic: The Gathering'

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Vicent Blanes Selva

Co-tutores: Samuel Morillas Gómez
Cristina Jordán Lluch

Curso 2016-2017

Resum

En el joc de cartes coleccionables *Magic: The gathering* se'ns planteja un problema d'elecció: un jugador disposa d'un conjunt d'unes 64 cartes y deu escollir les 23 millors per a augmentar les seues probabilitats d'èxit. El problema consistix en que no totes les posibilitats son factibles i a més les interaccions entre cartes solen jugar un paper molt important, tot aço influix en que la elecció del conjunt òptim no siga senzilla.

Per a realitzar aquesta feina, hem construit una aplicació modular en *Python 3* les parts principals de la qual han sigut: un mòdul de *web scrapping* per a recuperar informació d'internet, un conjunt de funcions per a guardarla i manipularla utilitzant bases de dades i el propi algoritme constructor de baralles que fa ús de tota la informació disponible per a realitzar la selecció.

El desenvolupament s'ha realitzat amb diversos mòduls de proves parametritzats que permeten ajustar el comportament de l'algoritme depenent de l'edició de cartes amb la que estem tractant, lo que ens ajuda a ajustar-nos a un comportament més proper al coneixement expert.

Paraules clau: web scrapping, python, bases de dades, grafs, heurística, Magic: The Gathering

Resumen

En el juego de cartas coleccionables *Magic: The Gathering* se nos plantea un problema de elección: un jugador dispone de un conjunto de alrededor de 64 cartas y debe elegir las 23 mejores para incrementar sus posibilidades de éxito. El problema estriba en que no todas las posibilidades son factibles y además las interacciones entre cartas suelen jugar un papel muy importante, esto influye en que la elección del conjunto óptimo no sea sencilla.

Para llevar a cabo esta tarea hemos construido una aplicación modular en *Python 3* cuyas partes principales han sido: un módulo de *web scrapping* para recuperar información desde Internet, un conjunto de funciones para guardarla y manipularla haciendo uso de bases de datos y el propio algoritmo constructor de mazos que hace uso de toda la información disponible para hacer la selección.

El desarrollo se ha completado con varios módulos de pruebas parametrizados que permiten ajustar el comportamiento del algoritmo dependiendo de la edición de cartas que este tratando, lo que nos ayuda a ajustarnos a un comportamiento más cercano al conocimiento experto.

Palabras clave: web scrapping, python, bases de datos, grafos, heurística, Magic: The Gathering

Abstract

There is a trading card game *Magic: The Gathering* that brings up an election issue: A game player has around a 64 cards and he has to choose the best 23rd cards, so he can improve his chances to win. The problem is, in one hand that not all the chances are feasible and in the other that an important role is played by the interaction between the cards so it makes the choice of the ideal combination quite difficult for the player.

With a view to accomplish this task, we have made a en *Python 3* modular application which main parts are: A *web scrapping* module to recover all the information; Some function as a whole to save it and keep it, plus using the database we can manipulate it; And finally, the deck cards builder algorithm which uses all the available information to make the selection.

Development of this application have been completed with some parameterized proof modules in order to adjust how the algorithm works when it does it with an edition of cards or another different. It helps us to adjust us to a closer performance of an expert knowledge.

Key words: web scrapping, python, database, graphs, heuristic, Magic: The Gathering

Índice general

Apéndices

Índice de figuras

Agradecimientos

A Cristina y Samuel por ayudarme, corregirme y motivarme a seguir trabajando en el proyecto. Me siento muy afortunado de haberos tenido como tutores.

A Marc y María por soportarme y estar conmigo en lo más duro.

A mi abuelo por hacerme llegar hasta aquí, espero que estés orgulloso.

«I wanted the whole world or nothing.» - Charles Bukowski.

CAPÍTULO 1
Introducción

1.1 Motivación

Magic: The Gathering es un juego de cartas coleccionables[1] (ver Figura 1.1) en el que dos o más participantes se enfrentan entre sí. Fue creado en 1993 por Richard Garfield, profesor de matemáticas y diseñador de juegos, actualmente entretiene a millones de personas en todo el mundo[2].

Se disputan, de manera oficial, multitud de torneos internacionalmente y existe la figura del jugador profesional. Toda esta industria ha propiciado su crecimiento y expansión, gracias a Internet podemos encontrar retransmisiones en directo de torneos, que son seguidos por miles de personas, análisis de barajas (mazos en argot), partidas de demostración y hasta lugares dónde se comercia con cartas (similares a portales de venta de segunda mano) donde se aplican las reglas de la oferta y la demanda.

Uno de los atributos más interesantes de Magic es que su elenco de cartas está compuesto por naipes de distinto tipo, cada uno con propiedades distintas y sujeto a unas reglas. Es muy importante que un mazo tenga un equilibrio con el número de cartas de cada tipo. Pasamos a enumerar los tipos más importantes:

- **Tierras**: Este tipo de cartas (*lands* en inglés, ver Figura 1.1) sirve para agregar maná. Cada jugador puede introducir una tierra por turno en la partida, la mecánica es muy sencilla, al girar un naipe del tipo tierra se agrega un maná del correspondiente color a un contador abstracto llamado **reserva de maná** [1]. A grandes rasgos, en Magic existen 5 colores: blanco **(W)**, negro **(N)**, verde **(G)**, rojo **(R)** y azul **(U)** que se corresponden con un tipo de tierra básica: **llanura, pantano, bosque, montaña e isla**. El contenido de la reserva de maná sirve para pagar el coste de las otras cartas (ver Figura 1.4). Existen más tipos de tierras en magic como *fecthlands*, *shocklands* o portales pero no tienen relevancia para nuestra aplicación.

- **Criaturas**: Este tipo de naipe se diferencia del resto por poseer atributos de ataque y defensa (Figura 1.1). Cada jugador puede introducir en el campo de batalla el número de criaturas que quiera mientras pueda pagar su coste de maná. Durante el turno de cada jugador existe una fase de combate en la que el jugador con turno activo puede declarar atacantes, girando las criaturas de su campo que considere oportunas (y no estén sujetas a ningún efecto que las impida atacar). Estás criaturas

1 Algunos jugadores utilizan una libreta para anotar su reserva de maná

giradas son nombradas como «atacantes», en ese momento el jugador contrario tiene la oportunidad de declarar «bloqueadores». Si alguna criatura atacante no es bloqueada, produce daño en las vidas del oponente igual al número de puntos de ataque. Si se produce bloqueo, atacante y bloqueador reciben los puntos de daño del rival en sus puntos de defensa, si alguna o ambas defensas llega a 0 o menos la criatura muere y va a la zona denominada como cementerio.

- **Instantáneos**: Este tipo de carta produce un efecto en la partida y puede ser jugado por cualquier jugador en todos los momentos de la partida. Dicho efecto depende del texto de la carta y puede variar desde *contrarrestar* un naipe del oponente hasta realizar puntos de daño a cualquier criatura o jugador pasando por ver la mano del oponente. Al igual que las criaturas, sólo pueden jugarse instantáneos si puede pagarse su coste de maná. Magic posee un sistema de **pila** para resolver casos en los que existen múltiples efectos simultáneos, como el lanzamiento de varios instantáneos.

- **Conjuros**: Idénticos a los instantáneos con la diferencia de que estos sólo pueden jugarse en las fases principales del turno activo, al igual que las criaturas.

- **Encantamientos**: Al igual que los conjuros, sólo pueden jugarse en turno activo y producen un efecto dependiendo de su texto, pero a diferencia de estos los encantamientos permanecen en el campo de batalla, prolongando su efecto hasta que son destruidos.

- **Auras y equipamientos**: Se trata de una categoría de cartas con las que se puede equipar a criaturas para mejorar sus propiedades, puede aumentar sus estadísticas o añadir algún efecto extra (ver Figura 1.2) Si la criatura equipada va al cementerio el aura le acompaña, mientras que en el caso de equipo este permanece en el campo de batalla.

Figura 1.1: Cartas de ejemplo

Figura 1.2: Ejemplo: aura y equipo

Existen varias formas de ganar una partida aunque la más común es dejar la vida del oponente, que empieza en 20 puntos, a 0. La manera más habitual de que esto suceda es atacar con nuestras criaturas e ir restando puntos de vida a nuestro rival. No obstante existen otras maneras de obtener la victoria:

- Puntos de veneno: Las criaturas con la cualidad *infect* realizan su daño como puntos de veneno en lugar de restar puntos de vida. Cuando un jugador acumula 10 puntos de veneno pierde la partida.
- No poder robar: Si el oponente está obligado a robar de su mazo[2], ya sea por ser su fase de robo o por algún efecto, y no tiene cartas suficientes, pierde automáticamente la partida.
- Otros efectos: Existen cartas que poseen condiciones de victoria (o derrota) (ver Figura 1.3).

Figura 1.3: Ejemplo: Cartas con condición de victoria

Además de las inmensas combinaciones de cartas, efectos y situaciones posibles, hay varias maneras de jugar a Magic. En algunos modos, como **Standar**, dos jugadores se enfrentan entre sí con un mazo que ha sido construido con antelación y que debe cumplir una serie de propiedades como, por ejemplo, que solo pueden cartas impresas en las ediciones del último año y medio. Además, el tamaño del mazo debe tener un mínimo de 60 cartas y cada carta, excepto tierras básicas, no puede repetirse más de 4 veces.

2

Se usan los términos mazo, biblioteca o *deck* indistintamente para referirse al conjunto de cartas disponibles para cada jugador durante la partida.

Otros formatos, como **limitado** y sus dos variantes: **draft** y **sellado**, se basan en la creación de un mazo a partir de cartas que cada jugador obtiene momentos antes de empezar la competición; en estos modos el tamaño mínimo del mazo es menor (40 cartas incluyendo cualquier cantidad de tierras básicas) y no existe límite de copias de cada carta.

Existen también otros modos como **commander** o **tiny leaders** donde cada mazo solo puede tener una copia de cada carta y existe la figura del comandante. No existe una sola forma de jugar a *Magic: the gathering*, si no que nuestra experiencia con el juego dependerá del formato escogido.

Figura 1.4: Atributos de una carta de Magic

1.2 Objetivos

El objetivo principal de este trabajo es desarrollar un software que ayude a los jugadores menos experimentados a construir sus mazos de forma automática en el formato **Sellado**.

La idea más importante reside en que el usuario, ya sea para entrenar o en el contexto de una partida real, pueda generar o introducir un *pool* de cartas a la aplicación y esta le devuelva el mazo óptimo. Este resultado puede servir tanto como guía a la hora de construir el mazo como para entrenamiento del usuario a la hora de construir bibliotecas, pues puede comparar su elección con la del algoritmo para un número indefinido de *pools* distintos y probar su evolución.

La aplicación debe ser ligera y capaz de proporcionar un subconjunto de cartas *óptimo* con el objetivo de maximizar la posibilidades de victoria. No se trata de una acción sencilla ya que existen muchos factores que afectan a la calidad de las bibliotecas; citamos a continuación los más importantes:

- **Número de criaturas/Número de hechizos**: Es primordial encontrar un equilibrio entre el número de criaturas que llevamos en nuestro *deck* y el número de hechizos. Lo ideal es tener más criaturas que hechizos.
- **Costes de maná**: Es muy importante que los costes estén bien distribuidos, lo que llamaremos la curva de maná. Ya que si todas nuestras cartas tienen el mismo coste tendremos problemas durante la partida.
- **Número de colores**: Cuantos más colores elijamos para nuestro mazo más opciones para elegir cartas tendremos, pero sufriremos una gran penalización, ya que deberemos tener tierras de más colores para poder pagar los costes de maná.

- **Valoración de una carta**: Aunque el propio juego no proporciona una valoración de las cartas, los jugadores experimentados saben valorar es cada una de ellas, lo que es un factor muy importante a la hora de elegir el subconjunto que formará nuestro *deck*.
- **Interacciones entre cartas**: También llamadas sinergias, es el valor añadido que se obtiene al elegir dos cartas cuya interacción es positiva.

Por tanto, el software que queremos desarrollar tendrá que elegir un subconjunto de entre todas las posibilidades, realizando una optimización multicriterio con los parámetros presentados. Así pues, primero tendremos que caracterizar la información más relevante de las cartas incluyendo aquellos campos que sean más subjetivos, como la puntuación individual de las cartas o la puntuación de sinergia.

El siguiente paso será combinar varias fuentes de información de forma persistente que nos permita tener almacenados precálculos. Por último, usaremos esta información para crear un algoritmo que trabaje sobre un grafo y sea capaz de elegir el subconjunto *óptimo*.

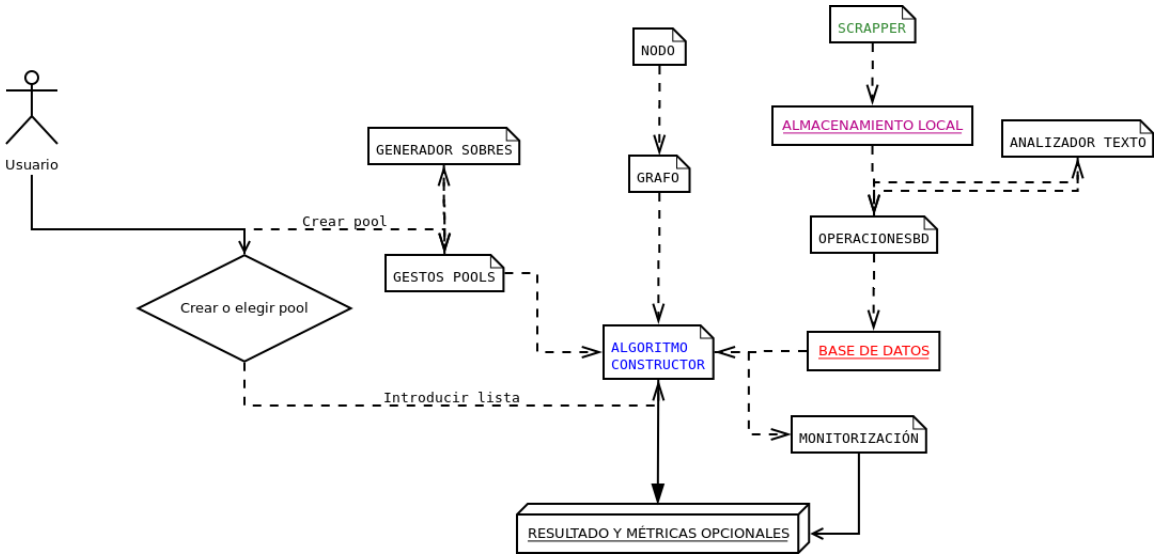
Para la implementación hemos decidido usar **Python 3** ya que es un lenguaje liviano y con mucha expresividad que nos va a permitir realizar de forma sencilla tanto el manejo de información como la interacción con la base de datos. Además al ser un lenguaje muy limpio y cercano al pseudocódigo es fácil de mantener y mejorar.

1.3 Estructura de la memoria

En los siguientes capítulos se presentará la memoria de desarrollo y las conclusiones. La memoria esta divida en cuatro partes: **obtención de información**, mediante *Web Scrapping*, el **tratamiento de la base de datos**, el **algoritmo constructor** de bibliotecas a partir de la información y **complementos** que ayuden a facilitar una solución. En cada una de las partes se explica detalladamente los pasos seguidos hasta obtener la implementación deseada.

En la Figura 1.5 podemos ver el diagrama de flujo de la aplicación para hacernos una idea previa de como van a ir encajando los distintos módulos implementados.

Figura 1.5: Diagrama de flujo de la aplicación



En las conclusiones analizaremos los resultados obtenidos que, pese a ser cualitativos, pueden ayudarnos a valorar nuestra elección de cartas. Como colofón, se sugerirán mejoras o ampliaciones que, por motivos de tiempo o ámbito, han quedado fuera del proyecto.

CAPÍTULO 2
Web Scrapping

Para desarrollar nuestro modelo vamos a necesitar, en primer lugar, caracterizar la información más relevante de las cartas. El problema estriba en que este tipo de datos no existe en ningún repositorio abierto, apareciendo sólo publicados, de forma no estructurada, en páginas web especializadas. Esto nos obliga a adquirirlos mediante la técnica del *Web Scrapping*.

Hemos de hacer hincapié en la cantidad de información necesaria para desarrollar un algoritmo de toma de decisiones, pues cuantos más datos tengamos sobre el problema, más informadas y certeras serán nuestras elecciones. Introducir este volumen de información de forma manual en una base de datos puede llegar a ser inabarcable y puede producir errores que el procesamiento automático de los datos mediante *web scrapping* puede evitar, motivo por el cual hemos considerado oportuno el desarrollo de este módulo.

Como ya hemos mencionado, se ha hecho uso de *Python*, en concreto de las librerías **urllib** y **beautifulsoup4**. La librería urllib es un conjunto de paquetes diseñados para hacer más fácil las interacciones a partir de peticiones web. Por su parte beautifulsoup4 permite tratar documentos html de cualquier tamaño para poder hacerlos más legibles o realizar consultas como, por ejemplo, obtener el primer texto que esté entre las etiquetas $< p >$.

A pesar de utilizar librerías estándar y código prácticamente extraído de la documentación [3] [4], la dificultad del *web scrapping* reside en la poca estandarización de la información. Este módulo se ha desarrollado para obtener los datos sobre la edición **Shadows over Innistrad**. Para utilizar el módulo en otra edición tendríamos que retocar los métodos que recopilan información de las páginas que nos interesan.

Cuando la información es introducida de forma manual el código que realiza el *web scrapping* suele requerir modificaciones. No se trata de un problema en nuestra implementación si no de algo inherente a la minería de información en la web.

Hemos obtenido los datos de tres páginas web especializadas: **TCGPlayer.com, Magiccards.info y ChannelFireball.com**. A continuación comentamos las páginas web y el método que hemos usado para la extracción de datos.

2.1 TCGPlayer

En nuestro código hemos creado una función que recupera el contenido de la página para una edición de cartas determinada a través de una url. La implementación de esta función es **tcg_player_scrap** (ver Listing 2.1), que se puede consultar en el anexo dentro del fichero **scrapper.py**. La función descarga el html de la página indicada y añade los datos a un diccionario cuya clave es el nombre de cada carta "limpio"(todo en minúsculas y sin caracteres especiales) y la clave es un vector de cuatro posiciones representando las características: **coste, rareza, coste medio monetario y el nombre original**.

```
def tcg_player_scrap(url, diccionario):
    req = Request(url,
                  headers={'User-Agent': 'Mozilla/5.0'})
    webpage = urlopen(req).read()
    html = BeautifulSoup(webpage, 'lxml')
    tdtags = html.find_all('td')
    # bandera tiene tres valores 0 = morralla principio, 1 = info, 2 = morralla final
    bandera = 0
    # este buccle sirve para eliminar la morralla del principio
    limpio = []
    estado = 0
    for celda in tdtags:
        if estado == 0 and 'SortOrder' in str(celda):
            estado = 1
        elif estado == 1 and '<b>Color</b>' in str(celda):
            estado = 2
        elif estado == 1 and 'SortOrder' not in str(celda):
            limpio.append(str(celda))

    # a partir de este punto tenemos solo las lineas que contienen la info
    puntero = 0
    while puntero < len(limpio):
        # puntero es la direccion base de la info de cada carta
        nombre = BeautifulSoup(limpio[puntero], 'lxml').find('a').getText()
        coste = BeautifulSoup(limpio[puntero + 1], 'lxml').find('td').getText()
        coste = re.sub('\s+', '', coste)
        rareza = BeautifulSoup(limpio[puntero + 3], 'lxml').find('td').getText()
        rareza = re.sub('\s+', '', rareza)
        costemedio = BeautifulSoup(limpio[puntero + 5], 'lxml').find('a').getText()
        diccionario[limpia_nombre(nombre)] = [coste, rareza, costemedio, nombre]
        puntero += 7
```

Listing 2.1: Código que descarga y trata la información de *TCGPlayer*

2.2 Magiccards

Magiccards.info es una página con mucha información útil para nuestra tarea, entre la que destaca sobre todo la existencia de una transcripción del texto para cada carta en varios idiomas. Hemos usado la técnica de *Web Scrapping* en esta web para obtener la descripción mencionada y el tipo de la carta. Para ello hemos utilizado un esquema idéntico al anterior, conectarnos a la web como si de un navegador se tratase, procesar el contenido y devolver un diccionario *Python* con la información extraída.

Al igual que en el método anterior, las claves son los nombres sin caracteres especiales y en minúsculas, los valores son listas de dos elementos que poseen el tipo de la carta y el texto completo de esta. El esquema mencionado corresponde a la función **magicinfo_scrap** (ver Listing 2.2) que puede consultarse en el anexo.

En la Figura 2.1 podemos ver un ejemplo de entrada en la página analizada que nos da una idea de como debemos tratar los datos.

```
def magicinfo_scrap(url, diccionario):
    globaltagsa = []
    globaltagsp = []
    req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})
    webpage = urlopen(req).read()
    # parsea con beautifulsoup
    html = BeautifulSoup(webpage, 'lxml')
    # nos interesan las a's para los nombres
    atags = html.find_all('a')
    # y las p's para el texto de la carta
    ptags = html.find_all('p')
    # en una primera pasada metemos los nombres
    inicio = False
    array_a_limpio = []
    for tag in atags:
        if '[' in tag.text:
            inicio = True
        elif '[' not in tag.text and inicio:
            array_a_limpio.append(tag)

    bandera = True
    for elem in array_a_limpio:
        if bandera:
            globaltagsa.append(elem.text)
            bandera = False
        elif str(elem.text) in 'all prints in all languages':
            bandera = True
    # en este punto tenemos introducidos los nombres de las cartas
    # ahora debemos coger los textos
    aux = []
    for i in range(len(ptags)):
        if i %5 == 0:
            aux.append(re.sub('\s+', ' ', ptags[i].text).split(',')[0])
        elif i %5 == 1:
            aux.append(re.sub('\s+', ' ', ptags[i].text))
        elif i %5 == 2:
            globaltagsp.append(aux)
            aux = []
    # ahora combinamos
    for i in range(len(globaltagsp)):
        diccionario[limpia_nombre(globaltagsa[i])] = globaltagsp[i]
```

Listing 2.2: Código que descarga y trata la información de *Magiccards*

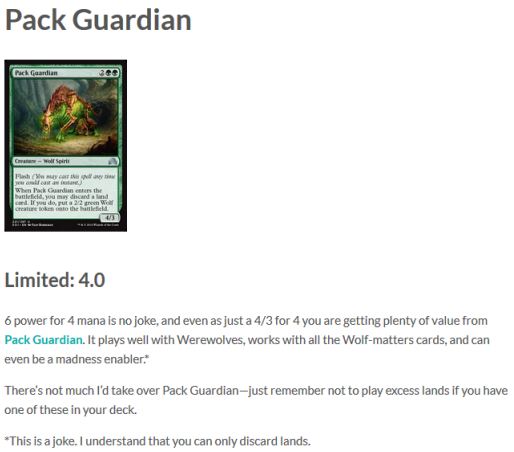
Figura 2.1: Tipo de información disponible en *Magiccards*

2.3 ChannelFireball

Una de las partes más importantes en el desarrollo de la aplicación consiste en valorar las cartas de manera individual, el problema reside en **como hacerlo**. *ChannelFireball* es una

página especializada que publica mucho contenido interesante y que cuenta con varios jugadores profesionales como publicadores, entre ellos destaca **Luis Scott-Vargas**, quien tiene un *blog* en el que valora de forma individual todas las cartas de una colección para el formato sellado. En la Figura 2.2 se puede ver un ejemplo de estas valoraciones.

Figura 2.2: Ejemplo de valoración de una carta por Scott-Vargas



Como se puede observar en la Figura 2.2, cada entrada cuenta con el nombre de la carta, su fotografía, una puntuación para limitado (dónde se incluyen *draft* y el modo que nos interesa, sellado) y una pequeña explicación sobre la utilidad que tiene.

Al igual que con las otras páginas, también disponemos de un método *Python* que realiza el *scrapping* (ver Listing 2.3), y en este caso devuelve un diccionario cuya clave es el nombre "limpio"de cada carta y cuyo valor es un número en el rango [0.0-5.0] correspondiente a la puntuación otorgada por Scott-Vargas.

```
def fireball_scrap(url, diccionario, color):
    # peticion web
    req = Request(url,
                  headers={'User-Agent': 'Mozilla/5.0'})
    webpage = urlopen(req).read()
    # parsea con beautifulsoup
    html = BeautifulSoup(webpage, 'lxml')
    # h1 son los nombres, h3 las notas
    nombres = html.find_all('h1')
    # tratamiento de los nombres
    nombres = nombres[3:-1]
    for i in range(len(nombres)):
        nombres[i] = limpia_nombre(nombres[i].text)
    #tratamiento de las notas
    splitted = html.find_all('h3')
    # array donde guardar las notas
    notas = []
    splitted = splitted[1:-3]
    for elem in splitted:
        if "Limited" not in elem.text:
            continue
        nota = elem.text.split(":")[1][:4]
        try:
            nota = float(nota)
        except ValueError:
            continue
        notas.append(nota)
    guarda_panda_csv(nombres, notas, color)
```

Listing 2.3: Código que descargar y trata la información de *ChannelFireball*

2.4 Fases web scrapping

Hemos dividido el módulo de *web scrapping* en dos fases con el objetivo de poder obtener datos intermedios que sean modificables y ayudar a mejorar la calidad de la información sintetizada. La primera fase se encargará de conseguir los datos en crudo y la segunda de sintetizar conocimiento sobre las cartas de una colección a partir de los distintos archivos temporales.

2.4.1. Primera fase: Recopilar

Aparte de las funciones para atacar las diferentes páginas web y obtener su información, el módulo de *web scrapping* diseñado cuenta con dos modos. El primero de ellos, mediante el uso de parámetros, utiliza las funciones explicadas anteriormente para obtener 3 diccionarios que posean toda la información. Los parámetros necesarios para poder realizar el *scrapeado* son:

- Un fichero con los **enlaces a ChannelFireball por colores**. Se encuentra en la carpeta **data/parametros** del proyecto.
- El enlace de *TCGPlayer* a la colección edición correspondiente, introducido como argumento a través de la línea de comandos (ver ejemplo de llamada).
- El enlace a la primera página (número 0) de una determinada colección en *Magiccards.info*, junto con el número de páginas totales. Ambos pasados como argumentos por línea de comandos.

Un ejemplo de llamada, en este caso para la colección **Shadows over Innistrad**, que iniciaría la recopilación de información por parte de nuestro módulo sería:

```
python3 scrapper.py recopilar
"http://magic.tcgplayer.com/db/search_result.asp?Set_Name=Shadows%20Over%20Innistrad"
"http://magiccards.info/query?q=e%3Asoi%2Fen&s=cname&v=card&p=" 16
```

El fichero de con los enlaces a *ChannelFireball* tendría la siguiente forma:

```
1 blanco|http://www.channelfireball.com/articles/shadows-over-innistrad-limited-set-review
      -white/
2 azul|http://www.channelfireball.com/articles/shadows-over-innistrad-limited-set-review-
      blue-cards/
3 negro|http://www.channelfireball.com/articles/shadows-over-innistrad-limited-set-review-
      black-cards/
4 rojo|http://www.channelfireball.com/articles/shadows-over-innistrad-limited-set-review-
      red/
5 verde|http://www.channelfireball.com/articles/shadows-over-innistrad-limited-set-review-
      green/
6 multicolor|http://www.channelfireball.com/articles/shadows-over-innistrad-limited-set-
      review-colorless-lands-and-gold/
```

Listing 2.4: Fichero con los enlaces a FireballChannel

Es muy importante indicar a qué color pertenece cada enlace, pues durante las pruebas se han detectado varios errores de formato en la propia página. *ChannelFireball* posee información de calidad pero la forma de mostrarla en muy «artesanal», lo cual genera muchos problemas a la hora del tratamiento automático. Añadiendo el nombre de cada color y el símbolo separador | podemos guardar cada listado de colores en un fichero usando un

formato revisable de modo manual, como csv.

Así pues, la salida obtenida al ejecutar el modo recopilar, cuyo código podemos ver en el Listing 2.5, es:

- El diccionario con la información obtenida de *tcgplayer* en **data/output/dicttcgplayer.p** (utilizando la librería *pickle* de *Python* para guardarlo en formato binario).

- El diccionario obtenido desde *Magiccards* en **data/output/dictmagicinfo.p**.

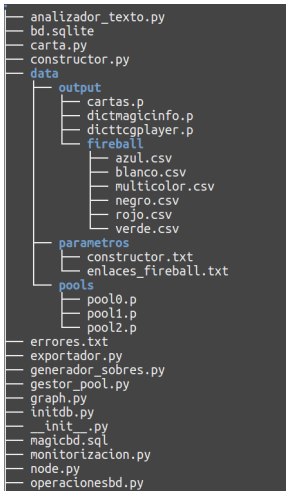
- El directorio **data/output/fireball** donde se encuentran 6 archivos csv, uno por cada color y el multicolor. Cada archivo posee dos columnas, el nombre "limpio"de la carta y su puntuación.

```
def recopilar_info():
    '''
    Este metodo hace el scrapping, se llamara si el usuario usa como primer
    parametro "recopilar"
    '''
    # crea los diccionarios que van a recibir la informacion
    dictfireball = {}
    dicttcgplayer = {}
    dictmagicinfo = {}
    # llama a los metodos concretos de cada pagina para rellenar los diccionarios
    tcg_player_scrap(diccionario=dicttcgplayer, url=sys.argv[2])

    # ahora la informacion de fireball
    enl_fireball = enlaces_fireball()
    for link in enl_fireball:
        # link = color:link
        parametros = link.split('|')
        fireball_scrap(diccionario=dictfireball, url=parametros[1], color=parametros[0])

    # por ultimo la informacion de magic info
    enl_magicinfo = enlaces_magicinfo(url=sys.argv[3], numeropags=int(sys.argv[4]))
    for link in enl_magicinfo:
        magicinfo_scrap(url=link, diccionario=dictmagicinfo)

    pickle.dump(dicttcgplayer, open("data/output/dicttcgplayer.p", "wb"))
    #pickle.dump(dictfireball, open("dictfireball.p", "wb"))
    pickle.dump(dictmagicinfo, open("data/output/dictmagicinfo.p", "wb"))
```

Listing 2.5: Método principal de recopilación

En la Figura 2.3 puede verse la estructura de directorios y ficheros necesarios para hacer funcionar nuestro software. Hemos utilizado el comando **tree** en la terminal de *Linux* para conseguir la instantánea.

Figura 2.3: Estructura de ficheros



2.4.2. Segunda fase: Combinar

Una vez descargada la información de las distintas fuentes y revisados los ficheros csv de *ChannelFireball*, el propio módulo de *scrapping* ofrece la posibilidad de combinar las distinta información mediante una validación cruzada de las distintas fuentes.

El método más importante en esta fase es **validacion_cruzada_scrapping** (ver Listing 2.6), que cruza los datos de los distintos diccionarios para cada nombre de carta distinto y mediante la orientación a objetos se construye un objeto de tipo carta[1]. Una vez se ha completado el proceso tenemos una lista *Python* compuesta de varios objetos tipo carta, usando *pickle* una vez más, la guardamos en **data/output/cartas.p**.

```
    '''
    Una vez obtenida toda la informacion este metodo la combina
    dicc_fireball: nombre : nota
    dicc_tcg_player: nombre : [coste, rareza, coste_medio, nombre_original]
    dicc_magic_info: nombre : [tipo, texto]
    '''
    array_cartas=[]
    cartas_error = open("errores.txt", "w")
    '''
    print(sorted(dicc_fireball.keys()))
    print('--------------\n')
    print(sorted(dicc_tcg_player.keys()))
    print('--------------\n')
    print(sorted(dicc_magic_info.keys()))
    '''
    for key in dicc_magic_info:
        try:
            elementostcg = dicc_tcg_player[limpia_nombre(key)]
            elementosinfo = dicc_magic_info[limpia_nombre(key)]
            if elementostcg[0] == '':
                continue
            #hay que eliminar algunos caracteres molestos
            nombre_original = elementostcg[3].replace("'", "''")
            costemedio = elementostcg[2].replace("$", '')
            rareza = elementostcg[1].replace("[", "")
            rareza = rareza.replace("]", "")
            tipo = elementosinfo[0]

            if key not in dicc_fireball:
```

[1] ver clase Carta en el anexo

```
                nf = 0
            else:
                nf = dicc_fireball[key]

            # constructor carta
            cartaux = Carta(nombre=key, nombre_original=nombre_original,coleccion='INS',
            color=calcular_color(elementostcg[0]), tipo=tipo,
            coste_mana=elementostcg[0], rareza=rareza, texto=texto,
            nota_fireball=nf, coste_medio=costemedio)
            array_cartas.append(cartaux)
        except:
            cartas_error.write(key+"\n")
    print(len(array_cartas))
    cartas_error.close()
    return array_cartas


def calcular_color(coste):
    conjunto = set()
    for character in coste:
```

Listing 2.6: Método que realiza la validación cruzada de datos

En caso de existir algún problema, el nombre de la carta que lo produce es copiado a un fichero de errores. Esto puede ocurrir, por ejemplo, con las cartas de criatura que pueden *darse la vuelta* y pasan a ser consideradas otra distinta. *Magiccards* cuenta la segunda cara de estas como una carta independiente ya que es una página centrada en el reglamento, mientras que no existen referencias en *TCGPlayer* y *ChannelFireball*.

Figura 2.4: «Avacyn, the Purifier» aparecería como carta independiente en *Magiccards.info*, pero no en las otras páginas ya que es el dorso de «Archangel Avacyn»


```
def calcular_info():
    '''
    Este metodo utiliza la informacion ya descargada, se llamara si es usuario
    utiliza el primer parametro "calcular"
    '''
    dictfireball = carga_panda_csv()
    dicttcgplayer = pickle.load(open("data/output/dicttcgplayer.p", "rb"))
    dictmagicinfo = pickle.load(open("data/output/dictmagicinfo.p", "rb"))
    return validacion_cruzada_scrapping(dicc_fireball=dictfireball,
    dicc_tcg_player=dicttcgplayer, dicc_magic_info=dictmagicinfo)
```

Listing 2.7: Método principal de combinar

2.5 Métodos auxiliares

Para hacer posible la realización de las principales funciones descritas en este apartado se han implementado una serie de funciones auxiliares que hacen mucho más sencillo el manejo de datos, las exponemos a continuación y pueden consultarse en el anexo.

- **combina_diccionarios**: Utiliza una lista de diccionarios como argumento y los fusiona para retornar uno solo en el que se encuentran todas las claves y valores de los demás (ver Listing 2.8).

```
def combina_diccionarios(diccionarios):
    '''
    Metedo auxiliar que fusiona varios diccionarios
    usado en carga_panda_csv
    '''
    resultado = dict()
    for elem in diccionarios:
        #print(elem)
        resultado.update(elem)
    return resultado
```

Listing 2.8: Método para combinar varios diccionarios

- **guarda_panda_csv**: Usando la librería **pandas** se crea un panel de datos con la información obtenida de *ChannelFireball*, marcando si existe un desfase entre el número de notas y nombres recuperados para que pueda ser revisado de manera manual (ver Listing 2.9).

```
def guarda_panda_csv(nombres, notas, color):
    '''
    Debido a que channel fireball tiene errores en los formatos este metodo
    crea una estructura pandas con los arrays de nombres y notas y lo guarda
    en csv para que el usuario pueda hacer cambios manualmente de
    manera sencilla
    '''
    while len(notas)<len(nombres):
        notas.append(-1)
    while len(nombres)<len(notas):
        nombres.append("null")
    d = {'Nombres': nombres, 'Notas':notas}
    data = pandas.DataFrame(d)
    data.to_csv("data/output/fireball/"+color+".csv")#, encoding='utf-8')
```

Listing 2.9: Método que guarda la información de *ChannelFireball* como panel de datos en csv

- **carga_panda_csv**: Realiza el proceso inverso, utilizando pandas recupera la información de los csv y la devuelve como un diccionario (ver Listing 2.10).

```
def carga_panda_csv():
    '''
    Carga y fusiona en un unico diccionario todos los csv de fireball
    '''
    path = "data/output/fireball/"
    archivos = listdir(path)
    diccionarios = []
    for elem in archivos:
        xx = pandas.read_csv(path+elem)
        di = dict(zip(map(limpia_nombre, xx['Nombres']), xx['Notas']))
        diccionarios.append(di)
    return combina_diccionarios(diccionarios)
```

Listing 2.10: Método que recupera los csv guardado por la función anterior

- **limpia_nombre**: Utiliza expresiones regulares, sustitución simple y paso a minúsculas para estandarizar el nombre de cada una de las cartas (ver Listing 2.11).

```
def limpia_nombre(nombre_orig):
    '''
    Limpia los nombres de las cartas de caracteres extranyos
    '''
    if '//' in nombre_orig:
        nombre_orig = nombre_orig.split('//')[0].strip()
    nom = nombre_orig.replace("'", "")
```

```
8     return re.sub('\s+', '', nom).lower().strip()
```

Listing 2.11: Método limpiador de caracteres

CAPÍTULO 3
Representación en forma de objetos

3.1 La clase carta

Con el objetivo de tratar la información producida por el scrapper de la manera más adecuada, hemos utilizado la orientación a objetos, presente en *Python 3*, para agrupar los distintos atributos de manera cohesionada. Para ello, hemos creado un fichero que contiene la clase carta y que puede revisarse en su totalidad en el anexo.

Para la representación hemos utilizado los siguientes atributos (ver Listing 3.1):

- **nombre**: El nombre de la carta sin caracteres extraños.
- **nombre_original**: Nombre de la carta tal y como aparece al realizar el *web scrapping*.
- **colección**: Colección a la que pertenece dicha carta.
- **color**: Carácter W, B, U, R, G, M, I correspondiente al color de la carta.
- **tipo**: Indica el tipo del naipe (criatura, instantáneo...).
- **coste_mana**: Cadena de texto que indica el coste de mana de la carta.
- **rareza**: Carácter indicando la rareza de la carta.
- **texto**: Descripción completa, incluyendo la parte de historia de la carta si la hubiera.
- **nota_fireball**: Nota otorgada por Scott-Vargas.
- **coste_medio**: Coste económico medio de la carta en el momento del *web scrapping*.

```
1     def __init__(self, nombre, nombre_original, coleccion, color, tipo, coste_mana,
      rareza, texto, nota_fireball, coste_medio):
2         #asignaciones
3         self.nombre = nombre
4         self.nombre_original = nombre_original
5         self.coleccion = coleccion
6         self.color = color
7         self.tipo = tipo
8         self.coste_mana = coste_mana
9         self.rareza = rareza
10        self.texto = texto
11        self.nota_fireball = nota_fireball
12        self.coste_medio = coste_medio
```

Listing 3.1: Método limpiador de caracteres

También se han desarrollado varios métodos para ayudar a la gestión, como uno que calcula el **coste convertido de mana** (ver Listing 3.2), el método para saber los colores cuando tenemos una carta Multicolor (Listing 3.3) o alguno que proporciona verbosidad para ayudar en la fase de pruebas o cuando se hace uso de interfaces de texto (Listing 3.4).

```
    def cmc(self):
        '''
        Devuelve el coste de mana convertido de una carta
        '''
        cmc = 0
        for el in list(self.coste_mana):
            if el.isdigit():
                cmc+=int(el)
            elif el != 'X':
                cmc+=1
        return cmc
```

Listing 3.2: Método que devuelve el cmc

```
    def colores(self):
        if self.color == 'M':
            colrs = set()
            for m in self.coste_mana:
                if not m.isdigit() and m != 'X':
                    colrs.add(m)
            return list(colrs)
        else:
            return  self.color
```

Listing 3.3: Método que devuelve la lista de colores de una carta

```
    def tostring(self):
        return str(self.nombre+' - '+self.nombre_original+' - '+self.coleccion+' - ' +
    self.color +
        ' - '+self.tipo+' - '+self.coste_mana+' - '+self.rareza+' - '+self.texto
        +' - '+str(self.nota_fireball)+' - '+str(self.coste_medio))
```

Listing 3.4: Método que da verbosidad sobre una carta

3.2 Modelado del pool como grafos

Hemos decidido utilizar un grafo como estructura de datos con la que representar todas las posibles opciones a la hora de construir nuestro mazo ideal. A nivel de diseño se ha decidido que cada vértice corresponda a una de las cartas disponibles y cada arista represente la relación entre los vértices que une, siendo su peso el valor de la sinergia existente. En el caso de que dos cartas no tengan ninguna relación consideraremos peso 0. Por ello, y dado que las relaciones entre cartas son simétricas, estamos ante un grafo no dirigido completo.

Para la implementación en *Python 3* hemos utilizado dos clases: **node.py** y **graph.py**. Ambas clases se han diseñado para tener una implementación general más allá de ser utilizadas únicamente en este proyecto. Aunque la implementación se corresponda con un grafo dirigido puede ser usada para nuestros fines si tenemos en mente que cada arista se tiene que añadir en ambos sentidos

La clase nodo cuenta con dos atributos, un nombre y una lista de vecinos (ver Listing 3.5). El atributo **name** puede contener objetos de distintos tipos, desde un entero, si solo deseamos que los vértices estén numerados, hasta objetos complejos. En el algoritmo constructor utilizaremos este atributo para almacenar los objetos de tipo carta. El atributo

edges es una lista de tuplas que contiene en su primera posición el nodo alcanzado por la arista y en la segunda el peso de dicha arista.

```
class Node:

    def __init__(self, name, edges = None):
        self.name = name
        self.edges = [] if edges is None else edges
```

Listing 3.5: Atributos de la clase nodo

Junto con el modelado de los atributos han sido desarrollado varios métodos que permiten realizar operaciones básicas (inserción, consulta y borrado) sobre las aristas de cada nodo (ver Listing 3.6).

```
    def add_edge(self, vertex, weight=0):
        '''
        vertex: an other node object
        weight: weight of the arc self -> otherVertex
        '''
        self.edges.append((vertex, weight))


    def add_multiple_edges(self, vertices, weights = None):
        '''
        vertices: list of node objects
        weights (optional): list of numbers
        '''
        if weights is None:
            weights = [0 for v in vertices]
        for i in range(len(vertices)):
            self.edges.append(tuple((vertices[i], weights[i])))

    def edge_list(self):
        return self.edges

    def delete_edge(self, node):
        for n in self.edges:
            if n == self:
                self.edges.remove(node)
                break
```

Listing 3.6: Metodos disponibles en la clase nodo

La clase grafo es bastante sencilla pues sólo se trata de una lista de nodos (ver Listing 3.7) y métodos para añadir, borrar y obtener un determinado nodo a partir de su nombre (ver Listing 3.8).

```
    def __init__(self, name, edges = None):
        self.name = name
        self.edges = [] if edges is None else edges
```

Listing 3.7: Lista de nodos que forman el grafo

```
        '''
        vertex: an other node object
        weight: weight of the arc self -> otherVertex
        '''
        self.edges.append((vertex, weight))


    def add_multiple_edges(self, vertices, weights = None):
        '''
        vertices: list of node objects
        weights (optional): list of numbers
        '''
        if weights is None:
            weights = [0 for v in vertices]
```

```
15          for i in range(len(vertices)):
```

Listing 3.8: Métodos disponibles en la clase grafo

CAPÍTULO 4
Interacción con la base de datos

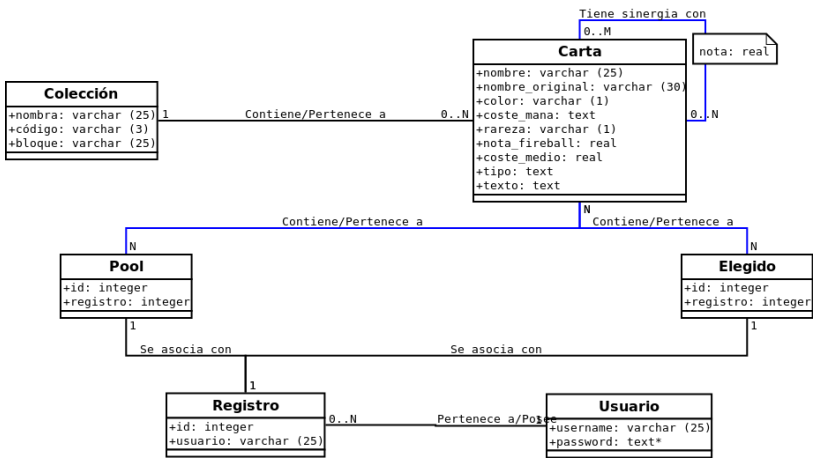
El objetivo de este fragmento del software es guardar y realizar precálculos en una base de datos a partir del fichero binario obtenido por el módulo de *web scrapping*. A continuación enunciaremos las fases seguidas para la obtención de información útil.

4.1 Estructura de la base de datos

A la hora de escoger la base de datos nos hemos decantado por **sqlite3**[5], cuyo logo podemos ver en la Figura 4.2, ya que es una base de datos potente, portable y multiformato. Su uso está recomendado para un tráfico de hasta **100 000 visitas diarias**, lo cual es más que suficiente para nuestros fines en este momento. Además, las consultas utilizadas son compatibles con otras bases de datos como **MySQL** o **MariaDB** por lo que es fácilmente escalable.

Se han diseñado varias tablas SQL para poder almacenar la información necesaria en el algoritmo constructor[1]. Las más importantes son **carta**, donde se guarda la información descargada de Internet y **sinergia**, dónde se representa la relación reflexiva con cardinalidad 1 a 1 de carta. A su vez existe la tabla **colección**, que nos permitirá manejarnos cuando tengamos cartas de varias ediciones (ver Figura 4.1). Y por último, aunque el objetivo principal del trabajo sea conseguir una selección de cartas *óptima*[2], se ha incluido un soporte para usuarios y sus bibliotecas en forma de varias tablas.

Figura 4.1: Diagrama UML de las tablas usadas por el algoritmo



[1] El algoritmo heurístico explicado en las secciones siguientes.
[2] En realidad subóptima.

El fichero **magicbd.sql** contiene el script para inicializar la base de datos de forma correcta. Puede consultarse en el anexo.

Figura 4.2: Logotipo de SQLite

4.2 Inserción de datos de una edición

Python cuenta con librerías por defecto en su distribución para manejar las interacciones con la base de datos SQLite3. Sobre estas se ha construido un módulo para el almacenamiento y recuperación de información relacionada con las cartas. En la clase **operacionesbd.py**, consultable en el anexo, se han implementado una serie de métodos que facilitan el acceso a la persistencia y que comentamos a continuación:

- Disponemos de un método, **crea_conexion** (ver Listing 4.1), que se conecta a nuestro fichero SQLite mediante su nombre y nos abre una conexión que usaremos para poder interactuar con él.

```
def crea_conexion():
    # crea la conexion a la bd
    #bd = pymysql.connect('localhost','magicuser','magictfg','magicbd')
    bd = sql.connect('bd.sqlite')
    return bd
```

Listing 4.1: Método que crea una conexión a una bd

- Los métodos **ejecuta_query** (ver Listing 4.2) y **ejecuta_multiples_queries** proporcionan una abstracción para ejecutar consultas SQL de forma rápida. Dichos métodos se encargan de todas las acciones a bajo nivel: crear la conexión, recuperar el cursor, ejecutar el *query* y realizar el *commit* a la base de datos. En caso de error la excepción es capturada, mostrando un mensaje de error por pantalla y realizando un *rollback* en la base de datos.

```
def ejecuta_query(query):
    # crea la conexion
    bd = crea_conexion()
    # obten el cursor
    cursor = bd.cursor()
    try:
        # query + commit
        cursor.execute(query)
        #bd.commit()
    except:
        return False
        # si error -> rollback
        print(sys.exc_info()[0])
        #bd.rollback()
    #devuelve el resultado del cursor para instrucciones select
    res = cursor.fetchall()
    bd.commit()
    bd.close()
    return res
```

Listing 4.2: Método para ejecutar un *query* de forma segura

- Por su parte, **crea_tablas_bd** (ver Listing 4.3), utiliza el *script* mencionado en el apartado anterior para inicializar las tablas de la base de datos. Su funcionamiento es sencillo, lee el fichero magicbd.sql como si de un texto plano se tratara e invoca a **ejecuta_query**.

```
def crea_tablas_bd():
    # crea la conexion
    bd = crea_conexion()
    # lee el script de creacion de la bd
    archivo = open('./magicbd.sql')
    script = archivo.read()
    archivo.close()
    # ahora lo ejecutamos como un query normal
    ejecuta_query(script)
    bd.close()
```

Listing 4.3: Método que crea las tablas de la BD

- La función **insertar_carta** (ver Listing 4.4) recibe como parámetro un objeto del tipo carta y construye una *query* en forma de cadena de texto teniendo en cuenta cada uno de sus atributos. El resultado se pasa a una invocación de **ejecuta_query**.

```
def insertar_carta(carta):
    # crea todo el query
    query = str("INSERT INTO Carta (nombre, nombre_original,coleccion, color,
    coste_mana, rareza,"
    +"nota_fireball, coste_medio, texto, tipo) VALUES('"+carta.nombre+"', '"+carta.
    nombre_original+"', '"+carta.coleccion+
    "', '"+carta.color+"', '"+carta.coste_mana+"', '"+carta.rareza+"', "+str(carta.
    nota_fireball)+
    ", "+str(carta.coste_medio)+", '"+carta.texto+"', '"+carta.tipo+"')")
    # ejecuto el query
    return ejecuta_query(query)
```

Listing 4.4: Método que inserta una carta en la bd

- Basándose en el método anterior, **introducir_coleccion_en_bd** (ver Listing 4.5) carga el fichero binario con todas las cartas de la colección y las va insertando una a una en la base de datos.

```
def introducir_coleccion_en_bd():
    '''
    Este metodo toma el path a el array con las cartas y las inserta en la bd
    '''
    cartas = pickle.load(open("data/output/cartas.p", "rb"))
    for elem in cartas:
        #print(elem)
        #print("------\n")
        if insertar_carta(elem) is False:
            print(elem.tostring())
```

Listing 4.5: Método que introduce una colección de cartas

- También disponemos de un método para insertar relaciones entre cartas (sinergias) en la base de datos: **insertar_relaciones_en_bd** (ver Listing 4.6), que toma como argumento una tripleta, (nombre de la primera carta, puntuación, nombre de la segunda carta).

```
def insertar_relaciones_en_bd(relaciones):
    '''
    dada una lista de tuplas que representa las relaciones entre cartas
    (com_carta1, puntuacion, nom_carta2)
```

```
    las introduce en la bd
    '''
    queries = []
    for rel in relaciones:
        #crea el query cart1, cart2, valor
        query = "INSERT INTO Sinergias values('" + str(rel[0] +"', '"+rel[2]+
        "', "+str(rel[1]))+")"
        queries.append(query)
    ejecuta_multiples_queries(queries)
```

Listing 4.6: Método que inserta sinergias entre cartas en una BD

- Por último existen dos métodos consultores, uno para obtener **todas** las cartas de una edición a partir de su nombre **listar_cartas** (ver Listing 4.7) y una función para consultar la sinergia entre dos cartas. Por motivos de ahorro de memoria, las relaciones puntuadas con un cero no se guardan, así que si ningún *query* es devuelto devuelve un 0.

```
def listar_cartas(coleccion):
    '''
    Este metodo devuelve un array con todas las cartas de la coleccion
    '''
    cartas = []
    query = "select * from Carta where coleccion = '"+coleccion+"'"
    #ejecutamos
    res = ejecuta_query(query)
    for linea in res:
        cartas.append(linea_cursor_a_carta(linea))
    return cartas
```

Listing 4.7: Método que lista las cartas de una edición

4.3 Cálculos previos

Para la realización de los precálculos se ha desarrollado un módulo, **analizador_texto.py**, cuyo objetivo principal es calcular la sinergias entre dos cartas. A pesar de que hay varios métodos desarrollados utilizando una librería avanzada para el tratamiento de textos como es **NLTK**[3], en la primera versión del software hemos optado por una implementación más sencilla, dejando a un lado el tratamiento complejo de texto.

En esta versión preliminar se recuperan todas las cartas de una colección y se evalúan por pares sin que se repitan las parejas, ya que las relaciones de sinergia son simétricas. En un primer momento se valora con un punto que las cartas pertenezcan al mismo color, a este puntuación se le añaden +0.5 por cada tipo y subtipo que compartan.

La principal virtud de este método es principalmente su sencillez, ya que agrupa cartas similares con un coste computacional muy bajo. La parte negativa reside en la posible omisión de factores positivos (o negativos) entre varias cartas, por ejemplo, cuando hablamos de *lords* ya que suelen ser criaturas individualmente no muy buenas pero que aportan bonificaciones notables a otras criaturas (ver Figura 4.3).

[3]Librería para el tratamiento de texto con soporte para múltiples idioma

Figura 4.3: Ejemplo de *lord*, otorga +1/+1 a criaturas aliadas del tipo zombie y esqueleto

A pesar de que se han conseguido buenos resultados con el método actual, podríamos refinar los cálculos utilizando tecnologías como *bag of words*, extracción de palabras clave y ponderando la similitud, entre otras.

CAPÍTULO 5
Algoritmo constructor

El algoritmo constructor es la pieza más importante de todo el proyecto y la que le da sentido. Su objetivo principal es recibir una lista de cartas, conseguir seleccionar el mejor subconjunto en relación a la probabilidad de ganar la partida y que este pueda ser mostrado al usuario. En Magic, para obtener un mazo competitivo, no es suficiente con combinar las cartas más poderosas de manera individual, es necesario que el conjunto elegido cumpla una serie de **restricciones**.

Estás restricciones se basan sobre todo en el maná, el coste que hay que pagar para poder realizar determinadas acciones. También existen condiciones menos duras como el número de criaturas/conjuros/tierras pero que son también muy importantes.

Para la creación de este algoritmo se propuso un método de programación por fases iterativas, intentando que en cada fase se obtuviera un resultado más refinado.

La versión actual es el resultado de **cuatro iteraciones**, en cada una de las cuales se refinaba y sofisticaba el algoritmo, en la búsqueda de la elección que realizaría un buen jugador.

5.1 Planteamientos previos

Hacemos notar que un algoritmo por fuerza bruta debería valorar cada una de las posibilidades en la elección de aproximadamente 23 cartas sobre 84 disponibles, lo que da lugar al número combinatorio:

$$\binom{84}{23} = 252571800009794690880$$

Dado el enorme número de posibilidades a estudiar nos decantamos por buscar otras alternativas de solución. Antes de decidir hacer uso de una función heurística se plantearon varias formas de resolución que proporcionaran una solución óptima real al problema. Una de las primeras fue la resolución mediante programación dinámica, la cual ayuda a resolver problemas muy complejos de forma óptima en tiempo eficiente, pero presentaba algunos problemas.

La principal dificultad encontrada fue el planteamiento de las ecuaciones recursivas dado que se trata de un problema con gran número de restricciones. Además, también es complicado conseguir un implementación eficiente dado el gran espacio de búsqueda de soluciones en el que nos movemos.

Otra de las metodologías planteada en un principio fue la de los algoritmos genéticos. Se realizó una implementación sencilla usando una lista como estructura de datos y la suma de los valores individuales y sinergias como valor. El problema detectado en este caso fue la lentitud en obtener soluciones, pues, incluso realizando pocas iteraciones, el algoritmo utilizaba un tiempo de respuesta inabarcable (*9 minutos* para 100.000 generaciones).

5.2 Primera iteración

En la primera versión del algoritmo el objetivo era conseguir un código que se ejecutara en un tiempo razonable y que produjera un resultado viable. Para ello se planteó el uso de grafos como estructura de datos para representar el *pool* y una función heurística para realizar la selección de las cartas de manera pseudoóptima y voraz.

Se diseñó una función de evaluación que puntuaba cada una de las cartas a partir de la última escogida.

```
1 por_nodo = 0.7
2 por_arista = 0.5
3 peso_cmc = 0.2
```

Listing 5.1: Factores incluidos en el código para realizar la puntación

```
1 def func_valor(arista):
2     '''
3     Funcion de evaluacion que da un valor subjetivo de una cartas
4     '''
5     #TODO --> afinado de la funcion
6     return (arista[0].name.nota_fireball*por_nodo + arista[1]*por_arista
7             - arista[0].name.cmc()*peso_cmc)
```

Listing 5.2: Primera función de evaluación

Tal y como se puede ver en el código(ver Listing 5.2) se usaron 3 factores para obtener las puntuaciones:

- **por_nodo**: Factor por el que se multiplica la nota individual, se suma a la nota final.
- **por_arista**: Factor por el que se multiplica la interacción entre la carta a evaluar y la última carta escogida, también se añade a la nota final.
- **peso_cmc**: Factor por el que multiplica el coste convertido de maná (cmc), se resta a la nota final para evitar que se acumulen costes de maná altos en la elección.

Como se ha explicado anteriormente, hemos usado un grafo completo para representar las cartas disponibles de cada jugador y tener una estructura de datos sobre la que operar. Para el grafo se ha utilizado una implementación muy similar a la de lista de adyacencia, dónde cada vértice es un objeto de tipo nodo que contiene toda la información de una carta y una lista de aristas que contienen peso y nodo destino. Pueden consultarse las clases **node.py** y **graph.py** en el anexo.

Hemos desarrollado un método para la construcción del grafo a partir de una lista de cartas dadas. El proceso tiene la siguiente estructura, ver Listing 5.3, que comentamos a continuación:

```
def construye_grafo(cartas):
    '''
    Dada una lista de cartas (las que tocan en el sobre)
    el metodo devuelve el grafo con los valores de la bd
    '''
    grafo = Graph()
    #anyado los vertices del grafo
    for carta in cartas:
        #si ya tengo un nodo con el mismo nombre hago tratamiento
        if grafo.get_node(carta) is None:
            grafo.add_node(carta)
        else:
            grafo.add_node(copy.copy(carta))
    #anyado las Sinergias
    nodos = grafo.nodes
    #todos contra todos
    for carta1 in nodos:
        pesos = []
        #dado un nodo calculamos los pesos para todo el resto
        for carta2 in nodos:
            pesos.append(bd.valor_sinergia(carta1.name, carta2.name))
        #anyadimos las aristas a dicho nodo
        carta1.add_multiple_edges(nodos, pesos)
    return grafo
```

Listing 5.3: Produce un grafo a partir de una lista de cartas

La idea principal es añadir los nodos uno a uno, de forma que, si es la primera vez que introducimos una carta se asigna a un nodo sin ninguna modificación, mientras que si esta misma carta ya existía la introducimos como una copia usando la librería **copy**, para que, aunque se trate de un objeto idéntico, tenga un *id Python* distinto. Las puntuaciones de las sinergias las extraemos usando el módulo analizado anteriormente **operacionesbd**.

Cuando hablamos en apartados anteriores de las restricciones relativas al coste de maná indicamos que la inestabilidad del mazo aumenta al aumentar el número de colores. En la práctica, la experiencia nos dice que en más del 90 % de los casos se usa una biblioteca compuesta por cartas de dos colores. En algunas ocasiones, y según la edición, se dan casos de *decks* tricolores pero no es la tónica habitual.

Este planteamiento nos lleva a una función que evalúa la calidad de color en base a las cartas que lo representan, con el objetivo de ver que colores son mejores a priori y acotar la decisión.

```
def suma_valores(cartas):
    sumatorio = 0
    for c in cartas:
        sumatorio+=c.nota_fireball
    return sumatorio
```

Listing 5.4: Función que evalua la calidad de un color

A pesar de haber limitado la elección de colores a los dos mejores de nuestro *pool*, en magic existen cartas que poseen más de un color. Estas cartas necesitan de varios tipos de tierras para poder pagar su coste de maná. Parece importante pues, determinar si una carta dada es compatible con los colores elegidos para el mazo, ya sea por su condición de multicolor o por ser una carta incolora, es decir, cuyo coste puede ser pagado con cualquier combinación de tierras(ver Listing 5.5, ver Figura 5.1).

```
def colores_compatibles(colores, nodoposible):
    '''
    Funcion para saber si los colores de un vecino son compatibles con
```

```
    los colores sobre el que estoy construyendo el mazo
    '''
    if nodoposible.name.color in colores or nodoposible.name.color == 'I':
        return True
    elif nodoposible.name.color == 'M':
        for c in list(nodoposible.name.coste_mana):
            if not(c == 'X' or c.isdigit() or c in colores):
                return False
        return True
    else:
        return False
```

Listing 5.5: Método que indica la compatibilidad de un nodo dados unos colores

Figura 5.1: Ejemplo de criatura multicolor y criatura incolora

En el Listing 5.6, podemos observar cual es el proceso principal que se encarga de llamar a todos los demás para construir nuestra librería *óptima*. El **algoritmo_constructor** recibe la lista de cartas disponibles y un número elecciones, que por defecto es 23, y devuelve una lista de ese tamaño.

```
def algoritmo_constructor(pool, ncartas=23):
    #monta el grafo
    grafo = construye_grafo(pool)
    #buscamos los dos colores con mejor puntuacion individual
    nodos = grafo.nodes
    colores = ['B', 'W', 'G', 'U', 'R']
    valores =  [0, 0, 0, 0, 0]
    for c in range(len(colores)):
        for n in nodos:
            if n.name.color == colores[c]:
                valores[c]+=n.name.nota_fireball
    #me quedo con los 2 mejores colores -> TODO
    colores = [x for (y,x) in sorted(zip(valores,colores), reverse=True)][0:2]
    #busco primer nodo
    actual = None
    visitados = set()
    for nodo in nodos:
        if nodo.name.color == colores[0]:
            actual = nodo
            break
    visitados.add(actual)
    while(len(visitados)<ncartas):
        posibilidades = sorted(actual.edge_list(), key=lambda k:func_valor(k), reverse=
    True)
        for pos in posibilidades:
            if colores_compatibles(colores, pos[0]) and pos[0] not in visitados:
                actual = pos[0]
                visitados.add(actual)
                break
    pinta(list(visitados))
```

Listing 5.6: Función principal del módulo constructor

Como podemos comprobar en el Listing 5.6, la primera acción es construir el grafo a partir del *pool*, usando la función descrita anteriormente. Después, calcular el color con

mejores notas individuales a partir del cual seleccionamos un nodo arbitrario del primer color como primera carta. A continuación, se utiliza un bucle while (mientras no se complete el número de cartas) ordenamos las opciones según la función de valor descrita. Finalmente, se recorre la lista buscando la primera carta que no haya sido ya introducida y se añade a la lista de visitados, pasando a formar parte del conjunto *óptimo*.

5.3 Segunda iteración

Los objetivos en la segunda iteración de desarrollo del algoritmo consistieron, principalmente, en cambiar partes del código para que este fuera más configurable por el usuario, ya que utilizar parámetros *hardcoded*[1] no es la mejor de las prácticas en desarrollo software. Además, se implementaron algunos refinamientos para la elección de las cartas.

En primer lugar, se ideó un sistema de parámetros para no tener que modificar el código durante el ajuste del algoritmo y las pruebas. La implementación es muy sencilla, pues se trata de un fichero de texto: **data/parametros/constructor.txt** (ver Listing 5.7).

```
#fichero de parametros del constructor
#las lineas que empiecen por # son comentarios

#poderacion del valor de los nodos
peso_carta : 0.7

#ponderacion del valor de las aristas
peso_sinergia : 0.3

#queremos controlar la curva de mana
contrapeso : 0

#segmentos de la curva de mana
segmana : 6

#maximo de cartas de cada coste
c1 : 3
c2 : 5
c3 : 8
c4 : 8
c5 : 5
cmax : 3
#maximo numero de cartas por tipo
max_no_criaturas : 9
max_criaturas : 17

#numero de cartas no tierras en el deck
ncartas : 23

#numero de ciraturas minimas en combinacion colores
criaturas_min : 15

#numero de cartas maximas para el promedio
max_num_cartas : 32
```

Listing 5.7: Ejemplo de fichero de parámetros

De esta manera, utilizando una línea por parámetro y los dos puntos para separar, podemos declarar y modificar de forma sencilla cualquier parámetro del programa. Recordemos que *Python* ofrece diccionarios sin necesidad de ninguna librería, una estructura de datos muy potente y fácil de utilizar, de modo que al arrancar el algoritmo podemos crear un *dict* dónde el nombre del parámetro referenciado sea la clave y su valor lo especificado en el fichero.

[1] Valores que podrían ser variables pero son tratadas como constantes en el código

Además, se ha habilitado el carácter especial #[2] para realizar comentarios, del mismo modo que *Python*, funcionalidad que puede aprovecharse para anotar resultados durante las pruebas o documentar el porqué de alguna configuración de cara a la mantenibilidad en el conjunto del software (ver Listing 5.8).

```
#global
global parametros
#carga de parametros
def carga_parametros():
    '''
    Carga la lista del parametros en el fichero
    'data/parametros/constructor.txt'
    '''
    global parametros
    parametros = {}
    with open('data/parametros/constructor.txt') as f_in:
        #lineas que no son ni comnetarios ni blancos
        lineas = [lin.rstrip() for lin in f_in if lin.rstrip() and not lin.strip().
    startswith('#')]
        for lin in lineas:
            aux = lin.split(':')
            parametros[aux[0].strip()] = float(aux[1].strip())
```

Listing 5.8: Carga de parámetro en la versión 2

La principal diferencia con la primera iteración es que en esta simplemente sumábamos los valores individuales de las cartas para conocer que color era mejor. Esto podía llevar a una mala elección de colores, pues la puntuación del color dependerá de la cantidad de cartas de dicho color que tengamos en el *pool*, es decir, cuando elegíamos un color no sabíamos si era debido a que poseía los mejores naipes o a que nuestro conjunto disponibles tenía más cartas de este color que de otros.

Así pues, una de las mejoras introducidas en esta segunda versión es la realización de un promedio, con el objetivo de conocer que colores tienen mejores medias. Para ello hemos desarrollado la función **promedio_color** (ver Listing 5.9)

```
def promedio_color(cart_col, color):
    '''
    Con el objetivo de calcular los colores que vamos a elegir
    esta funcion devuelve un promedio de lo buenas que son las cartas
    el numero de cartas queda truncado a las 16 primeras y debe tener
    minimo. Estas comprovaciones las hace la propia funcion
    '''
    if len (cart_col) < 10:
        return 0
    elif len(cart_col)> 16:
        cart_col = sorted(cart_col, key =lambda k:k.nota_fireball)[:16]

    if len([car for car in cart_col if 'creature' in car.tipo.lower()])<parametros['
    criaturas_min']:
        return 0
    sumador = 0
    for c in cart_col:
        if c.color == color:
            sumador += c.nota_fireball
        elif c.color == 'M' and color in c.colores():
```

Listing 5.9: Método que promedia las cartas de un color

En esta versión, si un color disponía de menos de 10 cartas lo desestimábamos, asignándole un valor de 0. También valorábamos el que hubiera más de 16 cartas, en cuyo caso

[2] Los comentarios de una línea en *Python* usan el mismo símbolo, no se permiten comentarios multilínea en el fichero

las ordenábamos por nota individual y nos quedábamos con las mejores. Ambas constantes están basadas en un conocimiento experto del juego y la construcción de mazos, pues conociendo que la mayoría de *decks* cuentan con dos colores y alrededor de 23 cartas (que no son tierras básicas), 10 y 16 parecían una buena cota inferior y superior para un sólo color.

Una vez pasado el filtro establecido para el número de naipes, iterábamos sobre la lista y sumábamos sus valores individuales, para al final devolver el valor total partido por el número de cartas evaluadas, es decir, la media de las mejores cartas del color. Esto nos permitía ser un poco más precisos a la hora de calificar los colores, lo que nos ayudaba a mejorar las elecciones en el algoritmo principal.

Destacar que la mejora más importante en esta versión, que comentamos a continuación, reside en la inclusión de restricciones en el algoritmo, las cuales nos ayudan a dirigir nuestras restricciones en pro de conseguir un conjunto mejor escogido.

Uno de los problemas de la primera versión consistía en premiar las cartas con menor coste de maná, es decir, casi siempre se seleccionaban cartas con bajo coste lo cual resultaba contraproducente en partidas largas, pues al no disponer de cartas más poderosas (y con más coste) la desventaja iba aumentando respecto a un mazo con una curva de maná más equilibrada.

Para solventar este problema implementamos un techo para la curva de maná, es decir, el número máximo de cartas de un determinado coste que el mazo puede contener. Esto nos permite dejar a 0 la penalización de coste y nos asegura que las elecciones de nuestro algoritmo quedan dirigidas hacia un conjunto con costes equilibrados (ver Listing 5.10).

```
            if (costes_activos[ind] < parametros['c'+str(int(coste_vec)) if coste_vec < 6 else 'cmax']):
                #si el vecino que exploro es una criatura y cabe la inserto
                if 'creature' in pos[0].name.tipo.lower() and criaturas < parametros['max_criaturas']:
                    actual = pos[0]
                    visitados.append(actual)
                    costes_activos[ind] += 1
                    criaturas += 1
                    break
                #lo mismo para las nos criaturas
                elif 'creature' not in pos[0].name.tipo.lower() and  no_criaturas < parametros['max_no_criaturas']:
                        actual = pos[0]
                        visitados.append(actual)
                        costes_activos[ind] += 1
                        no_criaturas += 1
                        break
```

Listing 5.10: Fragmento modificado en el algoritmo constructor

Junto a los límites establecidos, se incluyeron restricciones para el número máximo de cartas criatura y no criaturas con la intención de dirigir el contenido de las elecciones. La experimentación posterior demostró que con algunos *pools* el algoritmo no terminaba porque no podía satisfacer todas las reglas. De modo que incluimos un fragmento más de código: si evaluamos la última carta y no la escogemos, reiniciamos las restricciones para poder terminar el proceso (ver Listing 5.11).

```
            # si he llegado al ultimo y no anyado algo no va bien
```

```
                # posiblemente demasiadas restricciones
                # reinicio todas las restricciones
                if pos == posibilidades[-1]:
                    print('Reiniciando restricciones')
                    costes_activos = [0]*6
                    criaturas = 0
                    no_criaturas = 0
```

Listing 5.11: Porción de código que reinicia las restricciones

5.4 Tercera iteración

En la tercera iteración se propusieron dos objetivos. Por un lado incorporar una mejora en la elección de los colores, pues supone una reducción brusca del espacio de búsqueda, y por otro crear una distinción entre restricciones duras (inquebrantables) y restricciones blandas (que se cumplieran mientras fuera posible).

Para la primera tarea se implementó una función que probaba todas las combinaciones posibles con dos colores, es decir, todas las posibles parejas de colores existentes (ver Listing 5.12). El objetivo ya no era saber que colores eran los mejores en solitario, sino conocer cuál era la mejor combinación. Al contrario que en la versión anterior, donde solo se sumaba la fracción de puntuación de una carta correspondiente a ese color (una carta roja y verde de 4 puntos sumaba 2 puntos al rojo y 2 al verde), al evaluar los colores por parejas únicamente tenemos en cuenta las cartas multicolor que encajen perfectamente con la combinación evaluada, mejorando la precisión a la hora de elegir.

```
def eleccion_colores(pool):
    '''
    input: Todas las cartas de un pool
    output: Lista de cartas validas segun los colores elegidos
    '''
    #calculo de los dos mejores colores usando el promedio
    colores = ['B', 'W', 'G', 'U', 'R']
    prodcolores = list()
    #calculo el producto cartesiano de los colores
    for x in range(len(colores)):
        for y in range(x+1, len(colores)):
            prodcolores.append(tuple((colores[x],colores[y])))

    #fabrico una lista de cartas del pool compatibles con el primer par de colores
    cc = [c for c in pool if colores_compatibles(prodcolores[0], c)]
    for i in range(1, len(prodcolores)):
        #cartas de esa combinacion de colores
        aux = [c for c in pool if colores_compatibles(prodcolores[i], c)]
        #si encuentro una combinacion mejor a mi actual, la adopto
        if promedio_color(aux) > promedio_color(cc):
            cc = aux
    return cc
```

Listing 5.12: Método que elige la mejor combinación de colores

La elección de colores como un conjunto podría haberse ajustado un poco más utilizando el cálculo de las sinergias, el problema estriba en que para este proceso se debería haber realizado un relevante número de operaciones, ya que se trata de elegir una pareja de entre 5 colores, $\binom{5}{2} = 10$ y, posteriormente calcular todas las sinergias de una carta del subgrupo con todas las demás. De manera que al final realizaríamos $10 * n * (n-1)$ (10 combinaciones, por n cartas y por la sinergia de una carta con el resto) operaciones donde n es el tamaño del subgrupo formado por cada combinación de colores. La condición de obtener un algoritmo rápido y teniendo en cuenta que bajo conocimiento experto este cálculo no supondría mucha mejora, nos llevó a descartarlo y utilizar únicamente el valor

individual de las cartas, que en nuestro formato suele ser mucho más importante que las sinergias.

Para el segundo objetivo, se decidió establecer como restricciones duras el número de cartas en un mazo, el número de criaturas mínimo necesario para evaluar una combinación de colores y el número máximo de estas sobre el que obtener el promedio. Es decir, estos valores se establecen en el fichero de parámetros y el algoritmo y sus métodos auxiliares no pueden ignorarlos bajo ninguna situación. Este cambio da lugar a una modificación en el método **promedio_color** (ver Listing 5.13).

```
def promedio_color(cart_col):
    '''
    Con el objetivo de calcular los colores que vamos a elegir
    esta funcion devuelve un promedio de lo buenas que son las cartas
    el numero de cartas queda truncado a las 16 primeras y debe tener
    minimo. Estas comprovaciones las hace la propia funcion
    '''
    max_num_cartas = parametros['max_num_cartas']
    #estas reglas van a cambiar
    if len (cart_col) < 10:
        return 0
    elif len(cart_col)> max_num_cartas:
        cart_col = sorted(cart_col, key =lambda k:k.nota_fireball)[:max_num_cartas]
    #la combinacion de colores debe tener un numero minimo de criaturas
    if len([car for car in cart_col if 'creature' in car.tipo.lower()])<parametros['
    criaturas_min']:
        return 0
    #si supero las restricciones fuertes calculo el valor...
    sumador = 0
    #LAS CARTAS SIEMRE TENDRAN UN COLOR VALIDO
    for c in cart_col:
        if c.color == 'M':
            sumador += c.nota_fireball / len(c.colores())
        else:
            sumador += c.nota_fireball
    return float(sumador)/len(cart_col)
```

Listing 5.13: Método promedio_colores en la versión 3 del algoritmo

Podemos ver en el Listing 5.13 como hemos utilizado el diccionario que contiene los parámetros para elegir los criterios a la hora de recortar el conjunto de cartas a evaluar. Se ha mantenido el 10 como un parámetro fijo ya que es muy improbable que en un conjunto de aproximadamente 64 cartas una combinación de 2 colores posea menos de 10 naipes.

Dejamos como restricciones blandas aquellas que el algoritmo puede reiniciar si durante la exploración del espacio de búsqueda es incapaz de obtener una solución que se ajuste a los parámetros establecidos. En nuestro caso, se tratan de la curva tope de maná y el número de cartas criatura y no-criatura que debe contener el mazo. El fragmento de código que reinicia las restricciones se puede consultar en el Listing 5.11.

5.5 Cuarta iteración

En esta cuarta y última iteración se han realizado cambios con el fin de permitir que el algoritmo tenga una mejor adaptación al *pool* recibido. En primer lugar se han creado las variables globales **parametros, media y visitados** con el objetivo de agilizar el proceso tratando de evitar el *overhead* (exceso de tiempo de computación y memoria) que se produce al pasar parámetros a funciones de forma reiterada.

Otro de los cambios llevados a cabo en esta iteración es la modificación de la función

de puntuación, que pasa a llamarse **heurística**. Hemos variado la forma de utilizar la sinergia con las otras cartas; hasta esta versión sólo se valoraba la sinergia con la última carta escogida, en esta versión hemos añadido un fragmento de código que revisa todas las cartas escogidas y promedia su sinergia (ver Listing 5.14).

Por último, se ha añadido una función que calcula la base da maná que requiere el mazo elaborado, es decir, el tipo y el número de tierra que el jugador debería incluir en su mazo para completarlo.

```
    #sumatorio de la Sinergias
    sinergia = 0
    for elem in visitados:
        for vecino in elem.edge_list():
            if vecino[0] == arista[0]:
                sinergia += vecino[1]
    sinergia /= ncartas_seleccionadas
```

Listing 5.14: Fragmento de código de la función heurística para calcular el valor de la sinergia

No obstante el cambio más importante reside en la manera de controlar la curva de maná, pues se ha incluido una media que las cartas escogidas van a intentar seguir, tratando de compensar así que las cartas se concentren en costes muy bajos o muy altos. Para esta tarea se han añadido dos parámetros más, **media_mana y factor_mana**, que sustituyen al anterior **peso_cmc**[3]. La nueva forma de calcular el valor de una carta en un momento dado se muestra en el Listing 5.15.

```
    # calculo la evolucion de la curva si elegimos la carta evaluada
    evol_curva = (media*ncartas_seleccionadas+arista[0].name.cmc())/(float(
    ncartas_seleccionadas+1))
    # peso individual
    peso_individual = arista[0].name.nota_fireball*parametros['peso_carta']
    peso_sinergia = sinergia*parametros['peso_sinergia']
    penalizacion_cm = abs(evol_curva-parametros['media_mana'])*parametros['factor_mana']

    return peso_individual + peso_sinergia - penalizacion_cm
```

Listing 5.15: Fragmento de código en la función heurística para calcular la utilidad de una carta

Durante la realización de pruebas en esta versión se trataron de suprimir las restricciones débiles referentes al número tope de cartas en la curva de maná. El resultado fue peor al que se obtenía en la tercera versión dado que la media concentraba todos los costes a su alrededor (que suele estar cerca de 3 en un mazo equilibrado) dejando muy pocos naipes de coste de maná convertido igual a 1 e ínfimos costes iguales o mayores a 6. En una prueba posterior se decidió aplicar ambos criterios, es decir, limitar el número de cartas que puede tener cada coste y tratar de concentrar los valores alrededor de la media. A juicio experto los resultados mejoraron y por tanto se optó por mantener esta restricción débil.

Respecto a la base de maná, hemos creado una función a la que se le pasan las cartas elegidas por nuestro algoritmo y nos devuelve el número y tipo de tierras básicas que el jugador debe utilizar para maximizar sus posibilidades de victoria. Para llevar a cabo este cálculo es necesario obtener el número de maná de un color concreto para poder jugar cada carta (por ejemplo, una carta con coste RR3 sumaría 2 al maná rojo necesario) y dividirlo entre el número de costes totales; después multiplicaríamos la proporción obtenida por el número de tierras a jugar y redondearíamos (ver Listing 5.16).

[3]Factor por el que se multiplicaba el peso de maná convertido

```
def base_mana(eleccion, ncartas=17):
    '''
    Calcula la base mana necesaria para
    las cartas elegidas por el algoritmo
    '''
    opts = {'R', 'G', 'W', 'B', 'U'}
    costes = list()
    for carta in eleccion:
        for elem in carta.coste_mana:
            if elem in opts:
                costes.append(elem)
    col1 = [c for c in costes if c == costes[0]]
    col2 = [c for c in costes if c != costes[0]]
    num = round((len(col1)/len(costes))*ncartas)
    return ('#####\nBASE DE MANA\n#####\n'+str(col1[0]) + ' -> '
    + str(num) + '\n' + str(col2[0]) + ' -> ' + str(ncartas - num))
```

Listing 5.16: Método que calcula la base de maná para un mazo

5.6 Ejemplo de ejecución

Vamos a utilizar el generador de *pools* (explicado en profundidad en la sección 6.2) para simular el contenido que podríamos encontrar al abrir 6 sobres justo antes de empezar una partida de sellado; para ello ejecutaríamos la siguiente instrucción:

```
python3 gestor_pool.py
```

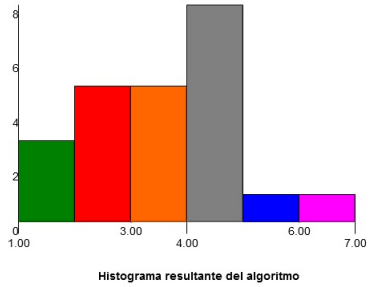
Este comando producirá un archivo binario en **data/pools/poolX.py** siendo X = 0 si es la primera vez que lo ejecutamos, 1 si el segunda... De modo que para ejecutar nuestro algoritmo con el *pool* que acabamos de crear (suponiendo que es el primero que creamos) tendríamos que lanzar un comando como este:

```
python3 constructor.py 0
```

El programa generará un fichero en formato cod (explicado detalladamente en la sección 6.3) donde se podrá visualizar el pool disponible y la elección realizada por el algoritmo. Además hemos incluido un pequeño resumen por pantalla que nos permitirá visualizar la curva de maná, es decir, la cantidad de cartas de cada coste que nuestro algoritmo ha elegido (ver Figura 5.3 y Figura 5.2).

Figura 5.2: Llamada al algoritmo con un pool arbitrario

Figura 5.3: Curva de maná correspondiente a la Figura 5.2



Aparte del resultado en formato cod y el histograma por consola, también hemos habilitado la opción de mostrar la elección de las cartas en un fichero de texto plano una vez finaliza el algoritmo constructor. Para ello hemos creado un pequeño *script* con dos funciones, una que calcula el histograma de las cartas elegidas y otra que vuelca a un fichero de texto tanto el pool disponible como la elección realizada (ver Listing 5.17 y 5.18).

```
def histograma(cartas, texto=True):
    '''
    Dado un conjunto de cartas seleccionadas devuelve
    un histograma. Si se espeficica texto = false devuelve un diccionario con
    las frecuencias. En caso contrario un string con la informacion
    '''
    frec = dict()
    for c in cartas:
        if c.cmc() in frec:
            frec[c.cmc()] += 1
        else:
            frec[c.cmc()] = 1
    if texto:
        cadena = '#####\nHISTOGRAMA\n#####\n'
        llaves = sorted(frec.keys())
        for k in llaves:
            cadena+= "Coste "+str(k)+" -> "+str(frec[k])+"\n"
        return cadena
    else:
        return frec
```

Listing 5.17: Función que obtiene el histograma a la salida del algoritmo

```
def imprime_resultados(pool, elegidos):
    f = open('resultado_algoritmo.txt', 'w')
    f.write('################\nPOOL\n################\n')
    for p in pool:
        f.write(str(p)+' -> '+str(p.nota_fireball)+'\n')
    f.write('################\nELECCION\n################\n')
    for e in elegidos:
        f.write(str(e)+' -> '+str(p.nota_fireball)+'\n')
    f.close()
```

Listing 5.18: Función que crea un fichero de texto con los resultados

La salida en texto plano para el ejemplo mostrado en la Figura 5.2 sería la siguiente:

```
################
POOL
################
Grotesque Mutation -> 1B -> 2.0
Macabre Waltz -> 1B -> 2.5
Macabre Waltz -> 1B -> 2.5
Murderous Compulsion -> 1B -> 3.0
Pale Rider of Trostad -> 1B -> 2.0
Groundskeeper -> G -> 0.0
Moonlight Hunt -> 1G -> 1.0
Howlpack Resurgence -> 2G -> 1.0
Weirding Wood -> 2G -> 0.5
```

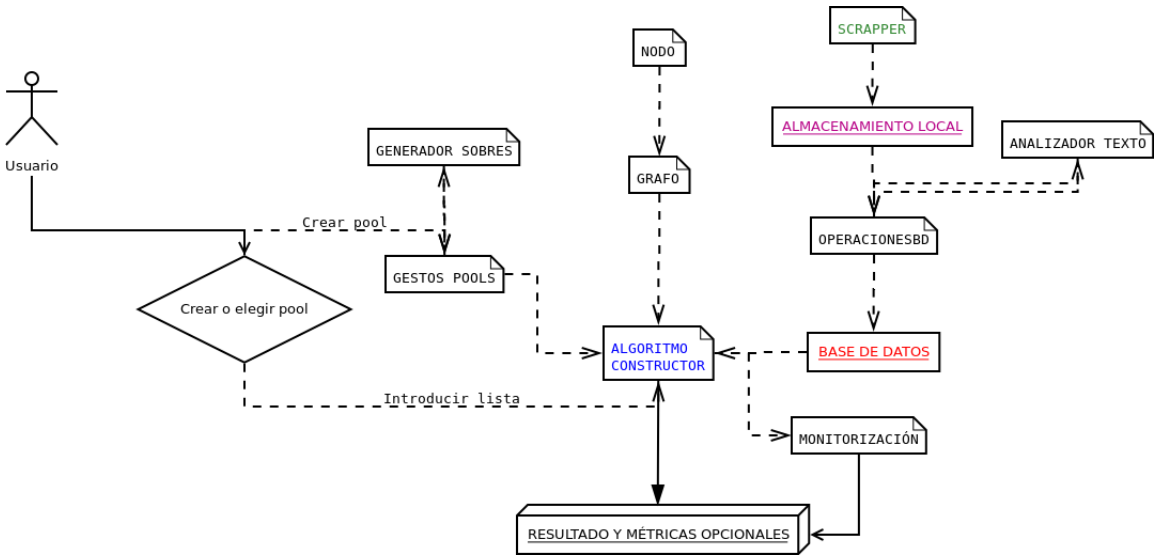
```
Equestrian Skill -> 3G -> 1.0
Explosive Apparatus -> 1 -> 1.0
True-Faith Censer -> 2 -> 1.0
Epiphany at the Drownyard -> XU -> 2.5
Dual Shot -> R -> 1.5
Insolent Neonate -> R -> 1.5
Skin Invasion -> R -> 2.0
Ember-Eye Wolf -> 1R -> 1.5
Magmatic Chasm -> 1R -> 1.0
Senseless Rage -> 1R -> 1.0
Tormenting Voice -> 1R -> 3.0
Bloodmad Vampire -> 2R -> 2.0
Bloodmad Vampire -> 2R -> 2.0
Convicted Killer -> 2R -> 1.5
Fiery Temper -> 1RR -> 3.5
Stensia Masquerade -> 2R -> 1.0
Ulrich's Kindred -> 2R -> 3.0
Hulking Devil -> 3R -> 1.0
Structural Distortion -> 3R -> 0.5
Structural Distortion -> 3R -> 0.5
Structural Distortion -> 3R -> 0.5
Voldaren Duelist -> 3R -> 3.0
Gatstaf Arsonists -> 4R -> 3.0
Devils' Playground -> 4RR -> 4.0
Invasive Surgery -> U -> 0.5
Furtive Homunculus -> 1U -> 2.5
Nagging Thoughts -> 1U -> 1.5
Press for Answers -> 1U -> 1.5
Press for Answers -> 1U -> 1.5
Seagraf Skaab -> 1U -> 1.0
Catalog -> 2U -> 2.0
Stitched Mangler -> 2U -> 2.0
Silent Observer -> 3U -> 3.0
Silent Observer -> 3U -> 3.0
Pore Over the Pages -> 3UU -> 3.5
Silburlind Snapper -> 5U -> 1.5
Chaplain's Blessing -> W -> 0.5
Chaplain's Blessing -> W -> 0.5
Stern Constable -> W -> 1.0
Thraben Inspector -> W -> 2.5
Topplegeist -> W -> 1.0
Topplegeist -> W -> 1.0
Topplegeist -> W -> 1.0
Hanweir Militia Captain -> 1W -> 3.5
Unruly Mob -> 1W -> 2.5
Vessel of Ephemera -> 1W -> 2.5
Vessel of Ephemera -> 1W -> 2.5
Angelic Purge -> 2W -> 3.0
Dauntless Cathar -> 2W -> 3.0
Ethereal Guidance -> 2W -> 0.5
Paranoid Parish-Blade -> 2W -> 2.0
################
ELECCION
################
Dead Weight -> B -> 2.0
Dead Weight -> B -> 2.0
Asylum Visitor -> 1B -> 2.0
Rancid Rats -> 1B -> 2.0
Merciless Resolve -> 2B -> 2.0
Accursed Witch -> 3B -> 2.0
Stallion of Ashmouth -> 3B -> 2.0
Stromkirk Mentor -> 3B -> 2.0
Deathcap Cultivator -> 1G -> 2.0
Quilled Wolf -> 1G -> 2.0
Quilled Wolf -> 1G -> 2.0
Byway Courier -> 2G -> 2.0
Stoic Builder -> 2G -> 2.0
Tireless Tracker -> 2G -> 2.0
Briarbridge Patrol -> 3G -> 2.0
Intrepid Provisioner -> 3G -> 2.0
Intrepid Provisioner -> 3G -> 2.0
Solitary Hunter -> 3G -> 2.0
Thornhide Wolves -> 4G -> 2.0
Watcher in the Web -> 4G -> 2.0
```

```
87 Watcher in the Web -> 4G -> 2.0
88 Harvest Hand -> 3 -> 2.0
89 Murderer's Axe -> 4 -> 2.0
```

Listing 5.19: Resultado del algoritmo en texto plano

5.7 Visión general

Figura 5.4: Diagrama de flujo de la aplicación



En la Figura 5.4 presentamos un diagrama de uso de nuestro software para ayudar a obtener una visión global de como se posicionan y encajan cada uno de los módulos y cómo es su interacción con el usuario.

CAPÍTULO 6
Complementos al desarrollo

Con el objetivo de enriquecer nuestro *software* y poder realizar pruebas de manera realista se han desarrollado tres ficheros *Python* con funcionalidades auxiliares. El primero de estos complementos al conjunto total de software es un simulador de sobres. El objetivo de este módulo es simular de manera realista el contenido de un sobre, lo que nos ayudará a realizar las pruebas. La segunda aportación, aunque relacionada con la primera, es un gestor de *pools*[1], que nos ayudará a crear y cargar *pools*. Por último, se ha desarrollado un módulo que será el encargado de exportar los resultados del algoritmo constructor a formatos útiles.

6.1 Generador de sobres

El generador de sobres tiene como objetivo simular el contenido de un sobre a partir de las cartas de una edición. Cada sobre tiene un total de 14 cartas de la colección y una tierra básica o una ficha[2](ver Figura 6.1). La problemática es que el contenido de los sobres no es completamente aleatorio y está sometido a una serie de reglas con el objetivo de acortar la diferencia de nivel entre los sobres.

Figura 6.1: Carta que introduce tokens y ejemplo de token

El método principal del generador es **simular_sobre** (consultar en Listing 6.1). En Magic la rareza de las cartas determina su probabilidad de aparición en un sobre, de modo que las cartas de rareza común tienen una posibilidad de aparición mucho más alta que una carta de rareza mítica. En general las 14 cartas se distribuyen de las siguiente manera:

- **10 cartas comunes**: En nuestro código utilizamos una lista intensional para cargar todas las cartas comunes y después con la ayuda de un bucle realizamos 10 elecciones aleatorias utilizando el método **choice** de la clase **random**.

[1]Recordemos que un *pool* es el conjunto de cartas disponible para un jugador
[2]Una carta no jugable que puede aparecer a partir de otra carta

- **3 cartas infrecuentes**: Elegidas de manera mímetica a las comunes se añaden a estas en el array que nos devolverá el resultado.
- **Una carta rara o mítica**: La última elección corresponde a una de las cartas de las mayores rarezas[3]. La posibilidad de que la carta sea mítica es de $\frac{1}{8}$, de modo que haciendo uso de la función **randint** (también de la clase random) podemos simular tirar un dado, si el resultado es menor o igual que 7 añadiremos una carta rara, si es 8 una mítica (ver Listing 6.1).

```
    def simular_sobre(self):
        selec = []
        comunes = [carta for carta in self.cartas if carta.rareza == 'C']
        infrecuentes = [carta for carta in self.cartas if carta.rareza == 'U']
        for i in range(10):
            selec.append(choice(comunes))
        for i in range(3):
            selec.append(choice(infrecuentes))
        raras = []
        decision = randint(1,8)
        if decision <= 7:
            raras = [carta for carta in self.cartas if carta.rareza == 'R']
        else:
            raras = [carta for carta in self.cartas if carta.rareza == 'M']
        selec.append(choice(raras))
        return selec
```

Listing 6.1: Función simular_sobre

6.2 Gestor de pools

El gestor de *pools* es una clase que se encarga de recuperar *pools* guardados como un ficheros binarios o generar uno aleatoriamente utilizando el generador de sobres. Se ha implementado con la finalidad de poder realizar pruebas rápidas. Por un lado, no es necesario introducir las cartas una a una, lo que sería una gran perdida de tiempo, y por otro, al mantener en memoria los *pools* generados podemos ver como evoluciona la decisión del algoritmo para el mismo conjunto de cartas.

Cada *pool* está identificado por un número, que se asigna automáticamente según el orden de creación, de modo que para cargar un *pool* sólo necesitamos conocer el identificador que nos interesa y cargarlo haciendo uso del método **load** contenido en la librería **pickle** (Listing 6.2).

```
def cargar_pool(numpool):
    '''
    Carga y devuelve un pool de cartas
    '''
    global path
    return pickle.load(open(path+'/pool'+str(numpool)+'.p', 'rb'))
```

Listing 6.2: Función cargar_pool

La creación del conjunto de cartas también es sencilla tras disponer de la implementación del generador de sobres. Se simulan 6 sobres utilizando el generador y se acumulan en una lista, después se ordenan por el criterio (color, coste de maná convertido y orden alfabético) y se evalúa el parámetro «retorno», donde indicamos si devolvemos el *pool* como un objeto *Python* o lo guardamos utilizando **pickle** (ver Listing 6.3).

[3] Recordemos que las rarezas en magic, de menor a mayor, son: comunes, infrecuentes raras y míticas

```
def crear_pool(retorno = False):
    '''
    Crea un pool aleatorio. Si retorno es falso, lo escribe en la
    carpeta /data/pool. Si es true lo devuelve con un return
    '''
    pool = []
    g = Generador()
    for i in range(6):
        pool = pool+g.simular_sobre()
    pool = sorted(pool, key=lambda k:(k.color, k.cmc(), k.nombre))
    if retorno:
        return pool
    else:
        global path
        numeracion = len(listdir(path))
        pickle.dump(pool, open(path+'/pool'+str(numeracion)+'.p', 'wb'))
```

Listing 6.3: Función crear_pool

También se ofrece en este módulo la posibilidad de cargar un *pool* utilizando un fichero de texto con los nombres normalizados de las cartas; la función cargará dicho fichero y recuperará los datos completos de la carta desde la base de datos con el objetivo de ir añadiéndolas a una lista que posteriormente devolverá (ver Listing 6.4).

```
def cargar_lista(path):
    lineas = open(path, 'r').read().split('\n')
    #print(lineas)
    cartas = []
    for lin in lineas:
        if not lin.strip():
            continue
        query = "select * from Carta where nombre='"+lin.strip()+"'"
        c = obd.ejecuta_query(query)
        if c:
            cartas.append(obd.linea_cursor_a_carta(c[0]))
    return cartas
```

Listing 6.4: Función que carga un pool desde un fichero de texto

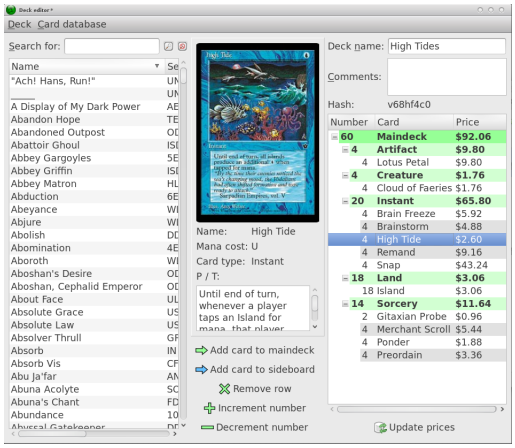
6.3 Exportador

Una de las funcionalidades más importantes a nivel de usuario es la posibilidad de exportar el resultado del algoritmo en un formato que permita una visualización adecuada. También puede resultar muy interesante de cara a la realización de pruebas, pues cuanto más fácil sea reconocer las cartas escogidas y las cartas totales más rápido podremos (y con menos margen de error) evaluar la actuación de nuestra heurística.

6.3.1. Cockatrice

Cockatrice es un programa multiplataforma que permite jugar a Magic en línea con jugadores de todo el mundo de manera gratuita; puede descargarse en https://cockatrice.github.io/. Además de permitir partidas el programa posee un editor de mazos que nos ayuda a configurar y visualizar de forma rápida nuestros mazos (ver Figura 6.2).

Figura 6.2: Editor de mazos en Cockatrice



Una de las principales virtudes del editor es que es capaz de, solo con el nombre de la carta, descargar su imagen y todos los datos complementarios. Además posee la funcionalidad de importar y exportar ficheros en formato **.cod**.

El formato cod no es más que una versión de xml que utiliza etiquetas preestablecidas para que Cockatrice sea capaz de reconocer la información (ver Listing 6.5). Por tanto, si conseguimos exportar el resultado de nuestro algoritmo a un fichero compatible con Cockatrice tendremos al instante una representación visual de nuestra elección que hará nuestra evaluación mucho más sencilla.

```
<?xml version="1.0" encoding="UTF-8"?>
<cockatrice_deck version="1">
<deckname></deckname>
<comments></comments>
<zone name="pool">
  <card number="2" price="0" name="Epiphany at the Drownyard"/>
  <card number="3" price="0" name="Skin Invasion"/>
  <card number="1" price="0" name="Ember-Eye Wolf"/>
```

Listing 6.5: Fragmento de ejemplo e formato cod

6.3.2. Código exportador

Con todo esto en mente, hemos incluido en nuestro módulo una función que permite crear un fichero .cod con dos secciones, la zona de las cartas elegidas y la zona del *pool*. El método es relativamente sencillo ya que, como las cabeceras y la mayoría del contenido son iguales, podemos tratar los ficheros cod como texto plano y escribir en ellos utilizando el método **write** presente en el manejo de ficheros que posee *Python* (ver Listing 6.6).

```
def exportDoc(pool, eleccion, nombrefich = 'res_algoritmo.cod'):

    # abro el fichero
    fich = open(nombrefich, 'w')
    # copio el esquema basico
    fich.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    fich.write('<cockatrice_deck version="1">\n')
    fich.write('<deckname></deckname>\n')
    fich.write('<comments></comments>\n')
    fich.write('<zone name="pool">\n')

    pool = convertir_diccionario(pool)
    eleccion = convertir_diccionario(eleccion)
    #pinto el pool
    for k in pool:
```

```
16          fich.write('\t<card number="'+str(pool[k])+'" price="0" name="'+k+'"/>\n')
17      #cierre pool
18      fich.write('</zone>\n')
19      fich.write('<zone name="eleccion">\n')
20      #pinto la eleccion
21      for e in eleccion:
22          fich.write('\t<card number="'+str(eleccion[e])+'" price="0" name="'+e+'"/>\n')
23      fich.write('</zone>\n')
24      #cierro el fichero
25      fich.write('</cockatrice_deck>')
26      fich.close()
```

Listing 6.6: Función simular_sobre

El método auxiliar **convertir_diccionario** se encarga de devolver un diccionario de frecuencias en el que se recoge el número de veces que aparece una carta con el mismo nombre. Es una función básica y puede verse en el Listing 6.7.

```
1  def convertir_diccionario(cartas):
2      '''
3      Esta funcion convierte una lista de cartas en un diccionario
4      para facilitar la expotacion a .cod y derivados
5      '''
6      dicc = {}
7      for c in cartas:
8          if c.nombre_original in dicc:
9              dicc[c.nombre_original] += 1
10         else:
11             dicc[c.nombre_original] = 1
12     return dicc
```

Listing 6.7: Función simular_sobre

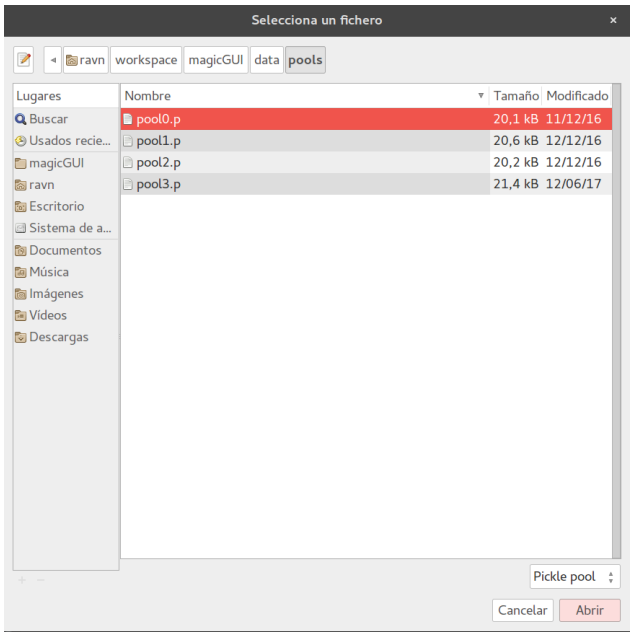
6.4 Interfaz en JavaFX

Además de lo presentado en el apartado 5.6 hemos decidido desarrollar una pequeña interfaz gráfica que facilite la interacción del usuario con el motor de construcción de mazos. La implementación se ha realizado en *JavaFX* ya que de forma rápida y sencilla se consiguen buenos resultados. En la Figura 6.3 se puede observar la pantalla principal de la interfaz con las dos opciones disponibles: elegir un fichero de *pool* o generar un *pool* aleatorio.

Figura 6.3: Pantalla principal de la interfaz

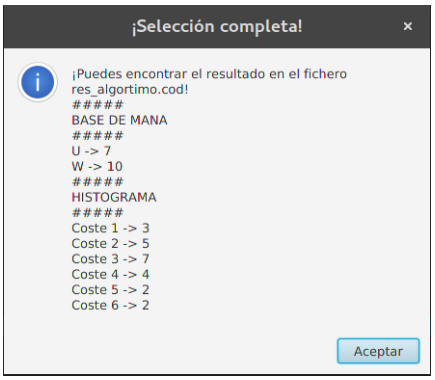
Si el usuario hace *click* sobre el botón seleccionar la aplicación abre una ventana de selección de ficheros(*filechooser*, ver Figura 6.4) donde el usuario puede elegir un fichero de texto o bien un fichero de *pickle* que contiene la lista de cartas que forman el *pool*.

Figura 6.4: Selector de archivos o filechooser



Una vez se ha seleccionado el fichero, el programa realiza una llamada al sistema para ejecutar el código *python* de la misma manera que se haría por consola. Una vez se completa el proceso, la aplicación nos muestra un diálogo con lo que obtendríamos como salida por consola (ver Figura 6.5).

Figura 6.5: Diálogo que nos indica que el proceso ha finalizado



Si el usuario selecciona la opción de generar, la interfaz invoca de forma idéntica al módulo constructor pero indicando que genere un *pool* aleatorio. Cuando se complete el proceso se obtendrá una ventana idéntica a la mostrada en 6.5. Todas las clases java y ficheros que componen la interfaz pueden consultarse en el anexo.

CAPÍTULO 7
Conclusiones

7.1 Resumen general

La última versión del algoritmo es un motor eficiente que realiza la selección de cartas de un *pool* de manera rápida y ofreciendo buenos resultados. El proceso de elección de un mazo conveniente a partir del *pool* se ha dividido en tres etapas: la obtención de información a través de páginas web especializadas (*web scrapping*), el desarrollo de métodos eficientes para añadir esa información a un base de datos y la implementación de un algoritmo heurístico que trabaja con grafos y que va eligiendo la mejor carta del espacio de búsqueda en cada momento.

Aunque en el mercado, sobre todo en lo referente a páginas web, existe algún simulador de sellado (uno de los más destacables es `http://draftsim.com/`) ninguno consigue un nivel de detalle en las elecciones comparable al motor desarrollado en este trabajo de final de grado.

Por tanto, si lo incrustáramos en un servicio web, o transformáramos el código en un lenguaje compatible para una implementación en teléfono móvil, tendríamos un producto genuino y que podría presentar un beneficio para todos los jugadores que se inician en el juego o en este modo de juego y no tienen muy claro como construir sus mazos. También podría ser de utilidad a jugadores más experimentados como plataforma para contrastar sus decisiones o como segunda opinión en caso de duda.

7.2 Posibles mejoras

El desarrollo del trabajo se ha centrado en lo que sería la parte más importante del software, formas de conseguir y tratar la información y un núcleo en el que hemos usado técnicas de inteligencia artificial clásica para realizar las elecciones. No obstante, existen varios aspectos mejorables o posibles líneas de trabajo para complementar y expandir lo presentado en este trabajo.

Tal y como comentamos en la sección anterior, una idea de expansión podría ser colocar nuestra base de datos y algoritmo en un servicio web, de modo que fuera capaz de atender peticiones desde distintas plataformas. De este modo, solo necesitaríamos interfaces donde introducir la información y con capacidad de atacar un servicio web, no sería necesario ningún tipo de lógica de negocio adicional.

Otra posibilidad de mejora, en un aspecto crítico a nivel de experiencia de usuario, sería

la opción de desarrollar una interfaz de comunicación con el usuario que fuera capaz de reconocer las cartas disponibles para este a través de la cámara de un dispositivo móvil. Se podría desarrollar una pieza de software que reconociera los bordes de la carta para conocer la posición relativa de sus elementos y un **OCR**[1] para tomar el nombre de esta.

También podríamos trabajar sobre la base de datos para que recopilara información sobre los usuarios y poder ofrecerles a estos estadísticas e históricos de sus elecciones. Un buen planteamiento que incluyera *feedback* podría ayudar a los usuarios a ver su evolución en la habilidad de construir mazos y también ayudar a ajustar los parámetros de nuestro modelo para que se adapte a las distintas ediciones.

Durante el desarrollo nos hemos centrado en la variante **sellado** dentro del modo **limitado**, una posible ampliación podría consistir en modificar el algoritmo para adaptarlo a la variante de **draft**, dónde las cartas se escogen individualmente de pools más pequeños. Es decir, el algoritmo debería conocer cada uno de estos pequeños pools (siempre menores de 14 cartas) para elegir cual es la mejor carta a añadir al conjunto.

Por último, una posible mejora, tal y como se comentó en la sección 4.3, sería introducir técnicas de reconocimiento de texto para hacer cálculo de sinergias. Podríamos desarrollar un módulo más complejo que permitiera una extracción de características del texto para, a posteriori, experimentar con distintos modelos y evaluar los resultados.

Con el objetivo de poder ampliar y mejorar el trabajo, todo el software se ha diseñado con la intención de conseguir la mayor modularidad posible a fin de poder sustituir piezas de software en el proceso sin que ello afecte a los demás componentes.

[1] Optical character recognition

Bibliografía

[1] Página oficial de Magic con información sobre el juego
Consultado el 25 de Abril de 2017.
http://magic.wizards.com/en/content/history.

[2] Página oficial de Magic con información sobre el juego.
Consultado el 3 de Mayo de 2017.
https://www.echomtg.com/blog/post/45/the-number-of-magic-players-worldwide-by-year/.

[3] Documentación oficial sobre la librería urllib.
Consultado el 11 de Mayo de 2017.
https://docs.python.org/3.6/library/urllib.html.

[4] Documentación oficial sobre la librería beautifulsoup4.
Consultado el 11 de Mayo de 2017.
https://www.crummy.com/software/BeautifulSoup/bs4/doc/.

[5] Recomendaciones de uso en SQLite.
Consultado el 13 de Mayo de 2017.
https://sqlite.org/whentouse.html

APÉNDICE A
Código completo

A.1 Algoritmo constructor

```
from carta import Carta
from graph import Graph
import operacionesbd as bd
import copy
import sys
import gestor_pool as gp
import exportador as xprt
import monitorizacion as mnt

'''
Clase principal para construir la baraja, necesita que la
base de datos este inicializada. Tiene parametros dinamicos para
que se pueden modificar sin tocar el codigo
'''

#global
global parametros
global media
global visitados

#carga de parametros
def carga_parametros():
    '''
    Carga la lista del parametros en el fichero
    'data/parametros/constructor.txt'
    '''
    global media
    media = 0
    global parametros
    parametros = {}
    with open('data/parametros/constructor.txt') as f_in:
        #lineas que no son ni comnetarios ni blancos
        lineas = [lin.rstrip() for lin in f_in if lin.rstrip() and not lin.strip().
    startswith('#')]
        for lin in lineas:
            aux = lin.split(':')
            parametros[aux[0].strip()] = float(aux[1].strip())


def construye_grafo(cartas):
    '''
    Dada una lista de cartas (las que tocan en el sobre)
    el metodo devuelve el grafo con los valores de la bd
    '''
    grafo = Graph()
    #anyado los vertices del grafo
    for carta in cartas:
        #si ya tengo un nodo con el mismo nombre hago tratamiento
        if grafo.get_node(carta) is None:
            grafo.add_node(carta)
        else:
```

```
            grafo.add_node(copy.copy(carta))
    #anyado las Sinergias
    nodos = grafo.nodes
    #todos contra todos
    for carta1 in nodos:
        pesos = []
        #dado un nodo calculamos los pesos para todo el resto
        for carta2 in nodos:
            pesos.append(bd.valor_sinergia(carta1.name, carta2.name))
        #anyadimos las aristas a dicho nodo
        carta1.add_multiple_edges(nodos, pesos)
    return grafo


def heuristica(arista):
    '''
    Funcion de evaluacion que da un valor subjetivo de una arista y un vertice
    '''
    global parametros
    global visitados
    ncartas_seleccionadas = len(visitados)

    #sumatorio de la Sinergias
    sinergia = 0
    for elem in visitados:
        for vecino in elem.edge_list():
            if vecino[0] == arista[0]:
                sinergia += vecino[1]
    sinergia /= ncartas_seleccionadas

    # el numero determina a partir de que punto empiezo a tener en cuenta la curva de
    mana
    '''
    if len(visitados)>12:
        #calcula como evolucionaria la curva de mana
        evol_curva = (media*ncartas_seleccionadas+arista[0].name.cmc())/(float(
    ncartas_seleccionadas+1))

        return (arista[0].name.nota_fireball*parametros['peso_carta'] +
                sum_sin*parametros['peso_sinergia'] -
                abs(evol_curva-parametros['media_mana'])*parametros['factor_mana'])
    else:
        return (arista[0].name.nota_fireball*parametros['peso_carta'] +
                sum_sin*parametros['peso_sinergia'])

    '''
    # calculo la evolucion de la curva si elegimos la carta evaluada
    evol_curva = (media*ncartas_seleccionadas+arista[0].name.cmc())/(float(
    ncartas_seleccionadas+1))
    # peso individual
    peso_individual = arista[0].name.nota_fireball*parametros['peso_carta']
    peso_sinergia = sinergia*parametros['peso_sinergia']
    penalizacion_cm = abs(evol_curva-parametros['media_mana'])*parametros['factor_mana']

    return peso_individual + peso_sinergia - penalizacion_cm


def colores_compatibles(colores, carta):
    '''
    Funcion para saber si los colores de un vecino son compatibles con
    los colores sobre el que estoy construyendo el mazo
    '''
    #si la carta es de uno de los colores o es incolora es compatible
    if carta.color in colores or carta.color == 'I':
        return True
    #si es M, miro que todos sus colores esten entre mis elegidos
    elif carta.color == 'M':
        bandera = True
        for col in carta.colores():
            if col not in colores:
                bandera = False
        return bandera
    #si no es monocolor aceptada, ni M, es de otro color, descarto
    else:
```

```
122         return False
123
124 def promedio_color(cart_col):
125     '''
126     Con el objetivo de calcular los colores que vamos a elegir
127     esta funcion devuelve un promedio de lo buenas que son las cartas
128     el numero de cartas queda truncado a las 16 primeras y debe tener
129     minimo. Estas comprovaciones las hace la propia funcion
130     '''
131     max_num_cartas = int(parametros['max_num_cartas'])
132     #estas reglas van a cambiar
133     if len (cart_col) < 10:
134         return 0
135     elif len(cart_col) > max_num_cartas:
136         cart_col = sorted(cart_col, key =lambda k:k.nota_fireball)[:max_num_cartas]
137     #la combinacion de colores debe tener un numero minimo de criaturas
138     if len([car for car in cart_col if 'creature' in car.tipo.lower()])<parametros['
    criaturas_min']:
139         return 0
140     #si supero las restricciones fuertes calculo el valor...
141     sumador = 0
142     #LAS CARTAS SIEMRE TENDRAN UN COLOR VALIDO
143     for c in cart_col:
144         if c.color == 'M':
145             sumador += c.nota_fireball / len(c.colores())
146         else:
147             sumador += c.nota_fireball
148     return float(sumador)/len(cart_col)
149
150
151 def eleccion_colores(pool):
152     '''
153     input: Todas las cartas de un pool
154     output: Lista de cartas validas segun los colores elegidos
155     '''
156     #calculo de los dos mejores colores usando el promedio
157     colores = ['B', 'W', 'G', 'U', 'R']
158     prodcolores = list()
159     #calculo el producto cartesiano de los colores
160     for x in range(len(colores)):
161         for y in range(x+1, len(colores)):
162             prodcolores.append(tuple((colores[x],colores[y])))
163
164     #fabrico una lista de cartas del pool compatibles con el primer par de colores
165     cc = [c for c in pool if colores_compatibles(prodcolores[0], c)]
166     maxim = promedio_color(cc)
167
168     for i in range(1, len(prodcolores)):
169         #cartas de esa combinacion de colores
170         aux = [c for c in pool if colores_compatibles(prodcolores[i], c)]
171         #si encuentro una combinacion mejor a mi actual, la adopto
172         if promedio_color(aux) > maxim:
173             cc = aux
174             maxim = promedio_color(aux)
175     return cc
176
177
178 def algoritmo_constructor(pool, ncartas=23):
179     '''
180     Devuelve una lista de cartas escogidas por el algoritmo a
181     partir de un pool
182     '''
183     #variables globales
184     global parametros
185     global visitados
186
187     #variables que controlan las restricciones
188     global media
189     costes_activos = [0]*6
190     criaturas = 0
191     no_criaturas = 0
192
193     #cartas del pool original compatibles con los colores elegidos
194     pool = eleccion_colores(pool)
```

```
195     #monta el grafo
196     grafo = construye_grafo(pool)
197     #buscamos los dos colores con mejor puntuacion individual
198     nodos = grafo.nodes
199 
200     #busco primer nodo (mejor del mejor color)
201     visitados = []
202     actual = sorted([n for n in nodos],
203                     key=lambda k:k.name.nota_fireball, reverse=True)[0]
204     visitados.append(actual)
205 
206     #busco los demas
207     while(len(visitados)<ncartas):
208         posibilidades = sorted(actual.edge_list(), key=lambda k:heuristica(k), reverse=
    True)
209         #para todos los veciones ordenados por puntos...
210         for pos in posibilidades:
211             #Si no lo he visitado...
212             #pos[0] es la carta, pos[1] la sinergia
213             if pos[0] not in visitados:
214                 coste_vec = pos[0].name.cmc()
215                 #y si ademas cumple mis exigencias sobre la curva de mana
216                 #variable auxiliar para que no casque por arrayindex
217                 ind = int(min(coste_vec-1, parametros['segmana']-1))
218                 #restriccion debil: curva de mana
219                 if (costes_activos[ind] < parametros['c'+str(int(coste_vec)) if
    coste_vec < 6 else 'cmax']):
220                     #restriccion debil: numero maximo de criaturas
221                     if 'creature' in pos[0].name.tipo.lower() and criaturas < parametros
    ['max_criaturas']:
222                         actual = pos[0]
223                         visitados.append(actual)
224                         costes_activos[ind] += 1
225                         criaturas += 1
226                         break
227                     #restriccion debil: numero maximo de NO criaturas
228                     elif 'creature' not in pos[0].name.tipo.lower() and  no_criaturas <
    parametros['max_no_criaturas']:
229                         actual = pos[0]
230                         visitados.append(actual)
231                         costes_activos[ind] += 1
232                         no_criaturas += 1
233                         break
234 
235 
236                 # si he llegado al ultimo y no anyado algo no va bien
237                 #posiblemente demasiadas restricciones
238                 #me cargo todas las restricciones
239                 if pos == posibilidades[-1]:
240                     print('Imposible cumplir las resticciones debiles')
241                     costes_activos = [0]*6
242                     criaturas = 0
243                     no_criaturas = 0
244 
245     return visitados
246 
247 
248 def nodos_a_cartas(nodos):
249     cartas = list()
250     for n in nodos:
251         cartas.append(n.name)
252     return cartas
253 
254 def base_mana(eleccion, ncartas=17):
255     '''
256     Calcula la base mana necesaria para
257     las cartas elegidas por el algoritmo
258     '''
259     opts = {'R', 'G', 'W', 'B', 'U'}
260     costes = list()
261     for carta in eleccion:
262         for elem in carta.coste_mana:
263             if elem in opts:
264                 costes.append(elem)
```

```
265     col1 = [c for c in costes if c == costes[0]]
266     col2 = [c for c in costes if c != costes[0]]
267     num = round((len(col1)/len(costes))*ncartas)
268     return ('#####\nBASE DE MANA\n#####\n'+str(col1[0]) + ' -> '
269     + str(num) + '\n' + str(col2[0]) + ' -> ' + str(ncartas - num))
270
271
272 def Main():
273     carga_parametros()
274     eleccion = sys.argv[1]
275     pool = None
276     if eleccion == '-1':
277         #-1 significa generame el pool
278         pool = gp.crear_pool(retorno = True)
279     else:
280         # si el archivo cargado es un txt
281         if eleccion.split('.')[-1] == 'txt':
282             pool = gp.cargar_lista(eleccion)
283         else:
284             pool = gp.cargar(eleccion)
285
286
287     res = sorted(nodos_a_cartas(algoritmo_constructor(pool, parametros['ncartas'])), key
    =lambda k:(k.color,k.cmc(),k.nombre))
288
289     pool = [p for p in pool if p not in res]
290     pool = sorted(pool, key=lambda k:(k.color,k.cmc(), k.nombre))
291     xprt.exportDoc(pool, res)
292     mnt.imprime_resultados(pool, res)
293     print(base_mana(res))
294     print(mnt.histograma(res))
295
296 if __name__ == '__main__':
297     Main()
```

Listing A.1: Fichero constructor.py

A.2 Módulo de Web Scrapping

```
1  from urllib.request import Request, urlopen
2  from bs4 import BeautifulSoup
3  import sys
4  import re
5  from carta import Carta
6  import pickle
7  import pandas
8  from os import listdir
9
10 def combina_diccionarios(diccionarios):
11     '''
12     Metedo auxiliar que fusiona varios diccionarios
13     usado en carga_panda_csv
14     '''
15     resultado = dict()
16     for elem in diccionarios:
17         #print(elem)
18         resultado.update(elem)
19     return resultado
20
21 def guarda_panda_csv(nombres, notas, color):
22     '''
23     Debido a que channel fireball tiene errores en los formatos este metodo
24     crea una estructura pandas con los arrays de nombres y notas y lo guarda
25     en csv para que el usuario pueda hacer cambios manualmente de
26     manera sencilla
27     '''
28     while len(notas)<len(nombres):
29         notas.append(-1)
30     while len(nombres)<len(notas):
31         nombres.append("null")
```

```
32      d = {'Nombres': nombres, 'Notas':notas}
33      data = pandas.DataFrame(d)
34      data.to_csv("data/output/fireball/"+color+".csv")#, encoding='utf-8')
35
36  def carga_panda_csv():
37      '''
38      Carga y fusiona en un unico diccionario todos los csv de fireball
39      '''
40      path = "data/output/fireball/"
41      archivos = listdir(path)
42      diccionarios = []
43      for elem in archivos:
44          xx = pandas.read_csv(path+elem)
45          di = dict(zip(map(limpia_nombre, xx['Nombres']), xx['Notas']))
46          diccionarios.append(di)
47      return combina_diccionarios(diccionarios)
48
49
50  def limpia_nombre(nombre_orig):
51      '''
52      Limpia los nombres de las cartas de caracteres extranyos
53      '''
54      if '//' in nombre_orig:
55          nombre_orig = nombre_orig.split('//')[0].strip()
56      nom = nombre_orig.replace("'", "")
57      return re.sub('\s+', '', nom).lower().strip()
58
59
60  def fireball_scrap(url, diccionario, color):
61      # peticion web
62      req = Request(url,
63                    headers={'User-Agent': 'Mozilla/5.0'})
64      webpage = urlopen(req).read()
65      # parsea con beautifulsoup
66      html = BeautifulSoup(webpage, 'lxml')
67      # h1 son los nombres, h3 las notas
68      nombres = html.find_all('h1')
69      # tratamiento de los nombres
70      nombres = nombres[3:-1]
71      for i in range(len(nombres)):
72          nombres[i] = limpia_nombre(nombres[i].text)
73      #tratamiento de las notas
74      splitted = html.find_all('h3')
75      # array donde guardar las notas
76      notas = []
77      splitted = splitted[1:-3]
78      for elem in splitted:
79          if "Limited" not in elem.text:
80              continue
81          nota = elem.text.split(":")[1][:4]
82          try:
83              nota = float(nota)
84          except ValueError:
85              continue
86          notas.append(nota)
87      guarda_panda_csv(nombres, notas, color)
88
89
90  def tcg_player_scrap(url, diccionario):
91      req = Request(url,
92                    headers={'User-Agent': 'Mozilla/5.0'})
93      webpage = urlopen(req).read()
94      html = BeautifulSoup(webpage, 'lxml')
95      tdtags = html.find_all('td')
96      # bandera tiene tres valores 0 = morralla principio, 1 = info, 2 = morralla final
97      bandera = 0
98      # este buccle sirve para eliminar la morralla del principio
99      limpio = []
100     estado = 0
101     for celda in tdtags:
102         if estado == 0 and 'SortOrder' in str(celda):
103             estado = 1
104         elif estado == 1 and '<b>Color</b>' in str(celda):
105             estado = 2
```

```
106         elif estado == 1 and 'SortOrder' not in str(celda):
107             limpio.append(str(celda))
108 
109     # a partir de este punto tenemos solo las lineas que contienen la info
110     puntero = 0
111     while puntero < len(limpio):
112         # puntero es la direccion base de la info de cada carta
113         nombre = BeautifulSoup(limpio[puntero], 'lxml').find('a').getText()
114         coste = BeautifulSoup(limpio[puntero + 1], 'lxml').find('td').getText()
115         coste = re.sub('\s+', '', coste)
116         rareza = BeautifulSoup(limpio[puntero + 3], 'lxml').find('td').getText()
117         rareza = re.sub('\s+', '', rareza)
118         costemedio = BeautifulSoup(limpio[puntero + 5], 'lxml').find('a').getText()
119         diccionario[limpia_nombre(nombre)] = [coste, rareza, costemedio, nombre]
120         puntero += 7
121 
122 
123 def magicinfo_scrap(url, diccionario):
124     globaltagsa = []
125     globaltagsp = []
126     req = Request(url, headers={'User-Agent': 'Mozilla/5.0'})
127     webpage = urlopen(req).read()
128     # parsea con beautifulsoup
129     html = BeautifulSoup(webpage, 'lxml')
130     # nos interesan las a's para los nombres
131     atags = html.find_all('a')
132     # y las p's para el texto de la carta
133     ptags = html.find_all('p')
134     # en una primera pasada metemos los nombres
135     inicio = False
136     array_a_limpio = []
137     for tag in atags:
138         if '[' in tag.text:
139             inicio = True
140         elif '[' not in tag.text and inicio:
141             array_a_limpio.append(tag)
142 
143     bandera = True
144     for elem in array_a_limpio:
145         if bandera:
146             globaltagsa.append(elem.text)
147             bandera = False
148         elif str(elem.text) in 'all prints in all languages':
149             bandera = True
150     # en este punto tenemos introducidos los nombres de las cartas
151     # ahora debemos coger los textos
152     aux = []
153     for i in range(len(ptags)):
154         if i %5 == 0:
155             aux.append(re.sub('\s+', ' ', ptags[i].text).split(',')[0])
156         elif i %5 == 1:
157             aux.append(re.sub('\s+', ' ', ptags[i].text))
158         elif i %5 == 2:
159             globaltagsp.append(aux)
160             aux = []
161     # ahora combinamos
162     for i in range(len(globaltagsp)):
163         diccionario[limpia_nombre(globaltagsa[i])] = globaltagsp[i]
164 
165 
166 def substring(s, first, last):
167     try:
168         start = s.index(first) + len(first)
169         end = s.index(last, start)
170         return s[start:end]
171     except ValueError:
172         return ""
173 
174 
175 def validacion_cruzada_scrapping(dicc_fireball, dicc_tcg_player, dicc_magic_info):
176     '''
177     Una vez obtenida toda la informacion este metodo la combina
178     dicc_fireball: nombre : nota
179     dicc_tcg_player: nombre : [coste, rareza, coste_medio, nombre_original]
```

```
    dicc_magic_info: nombre : [tipo, texto]
    '''
    array_cartas=[]
    cartas_error = open("errores.txt", "w")
    '''
    print(sorted(dicc_fireball.keys()))
    print('-------------\n')
    print(sorted(dicc_tcg_player.keys()))
    print('-------------\n')
    print(sorted(dicc_magic_info.keys()))
    '''
    for key in dicc_magic_info:
        try:
            elementostcg = dicc_tcg_player[limpia_nombre(key)]
            elementosinfo = dicc_magic_info[limpia_nombre(key)]
            if elementostcg[0] == '':
                continue
            #hay que eliminar algunos caracteres molestos
            nombre_original = elementostcg[3].replace("'", "''")
            costemedio = elementostcg[2].replace("$", '')
            rareza = elementostcg[1].replace("[", "")
            rareza = rareza.replace("]", "")
            texto = elementosinfo[1].replace("'", "")

            if key not in dicc_fireball:
                nf = 0
            else:
                nf = dicc_fireball[key]

            # constructor carta
            cartaux = Carta(nombre=key, nombre_original=nombre_original,coleccion='INS',
            color=calcular_color(elementostcg[0]), tipo=tipo,
            coste_mana=elementostcg[0], rareza=rareza, texto=texto,
            nota_fireball=nf, coste_medio=costemedio)
            array_cartas.append(cartaux)
        except:
                cartas_error.write(key+"\n")
    print(len(array_cartas))
    cartas_error.close()
    return array_cartas


def calcular_color(coste):
    conjunto = set()
    for character in coste:
        if not character.isdigit():
            conjunto.add(character)
    if len(conjunto) == 0:
        return 'I'
    elif len(conjunto) >= 2:
        return 'M'
    else:
        return str(list(conjunto)[0])


def enlaces_fireball():
    return open('data/parametros/enlaces_fireball.txt').read().split('\n')[:-1]


def enlaces_magicinfo(numeropags, url):
    enlaces = []
    for i in range(1, numeropags + 1):
        enlaces.append(url + str(i))
    return enlaces

def calcular_info():
    '''
    Este metodo utiliza la informacion ya descargada, se llamara si es usuario
    utiliza el primer parametro "calcular"
    '''
    dictfireball = carga_panda_csv()
    dicttcgplayer = pickle.load(open("data/output/dicttcgplayer.p", "rb"))
    dictmagicinfo = pickle.load(open("data/output/dictmagicinfo.p", "rb"))
    return validacion_cruzada_scrapping(dicc_fireball=dictfireball,
```

```
254     dicc_tcg_player=dicttcgplayer, dicc_magic_info=dictmagicinfo)
255
256 def recopilar_info():
257     '''
258     Este metodo hace el scrapping, se llamara si el usuario usa como primer
259     parametro "recopilar"
260     '''
261     # crea los diccionarios que van a recibir la informacion
262     dictfireball = {}
263     dicttcgplayer = {}
264     dictmagicinfo = {}
265     # llama a los metodos concretos de cada pagina para rellenar los diccionarios
266     tcg_player_scrap(diccionario=dicttcgplayer, url=sys.argv[2])
267
268     # ahora la informacion de fireball
269     enl_fireball = enlaces_fireball()
270     for link in enl_fireball:
271         # link = color:link
272         parametros = link.split('|')
273         fireball_scrap(diccionario=dictfireball, url=parametros[1], color=parametros[0])
274
275     # por ultimo la informacion de magic info
276     enl_magicinfo = enlaces_magicinfo(url=sys.argv[3], numeropags=int(sys.argv[4]))
277     for link in enl_magicinfo:
278         magicinfo_scrap(url=link, diccionario=dictmagicinfo)
279
280     pickle.dump(dicttcgplayer, open("data/output/dicttcgplayer.p", "wb"))
281     #pickle.dump(dictfireball, open("dictfireball.p", "wb"))
282     pickle.dump(dictmagicinfo, open("data/output/dictmagicinfo.p", "wb"))
283
284
285 # main
286 if __name__ == '__main__':
287     if sys.argv[1] == 'recopilar' and len(sys.argv)==5:
288         recopilar_info()
289         print("Archivos descargados correctamente!")
290     elif sys.argv[1] == 'calcular' and len(sys.argv) == 2:
291         res = calcular_info()
292         print("Se ha podido consegir informacion sobre", len(res), 'cartas')
293         pickle.dump(res, open("data/output/cartas.p", "wb"))
294         print("Archivo disponible en data/output/cartas.p")
295     else:
296         print("Este programa tiene dos posibles usos:")
297         print("python3 scrapper recopilar url_tcg url_magicinfo_sin_pagina
    pags_magicinfo")
298         print("python3 scrapper calcular")
```

Listing A.2: Fichero scrapper.py

A.3 Clase Carta

```
1 class Carta:
2
3     # esto es un variable estatica
4     numeroCartas = 0
5     # constructor de la clase
6     def __init__(self, nombre, nombre_original, coleccion, color, tipo, coste_mana,
      rareza, texto, nota_fireball, coste_medio):
7         #asignaciones
8         self.nombre = nombre
9         self.nombre_original = nombre_original
10        self.coleccion = coleccion
11        self.color = color
12        self.tipo = tipo
13        self.coste_mana = coste_mana
14        self.rareza = rareza
15        self.texto = texto
16        self.nota_fireball = nota_fireball
17        self.coste_medio = coste_medio
18
```

```
19
20      def __str__(self):
21          return self.nombre
22
23      def cmc(self):
24          '''
25          Devuelve el coste de mana convertido de una carta
26          '''
27          cmc = 0
28          for el in list(self.coste_mana):
29              if el.isdigit():
30                  cmc+=int(el)
31              elif el != 'X':
32                  cmc+=1
33          return cmc
34
35      def colores(self):
36          if self.color == 'M':
37              colrs = set()
38              for m in self.coste_mana:
39                  if not m.isdigit() and m != 'X':
40                      colrs.add(m)
41              return list(colrs)
42          else:
43              return  self.color
44
45
46      # aqui tocara escribir algun metodo auxiliar
47      def tostring(self):
48          return str(self.nombre+' - '+self.nombre_original+' - '+self.coleccion+' - ' +
        self.color +
49          ' - '+self.tipo+' - '+self.coste_mana+' - '+self.rareza+' - '+self.texto
50          +' - '+str(self.nota_fireball)+' - '+str(self.coste_medio))
```

Listing A.3: Fichero carta.py

A.4 Clase Nodo

```
1  '''
2  Node for Graph class
3  David Picornell <dapicar(at)inf(dot)upv(dot)es>
4  Vicent Blanes <viblasel(at)inf(dot)upv(dot)es>
5  '''
6  class Node:
7
8      def __init__(self, name, edges = None):
9          self.name = name
10         self.edges = [] if edges is None else edges
11
12
13     def __str__(self):
14         s = str(self.name) +' -> '
15         for e in self.edges:
16             s+='('+str(e[0].name)+", "+str(e[1])+')\n\t'
17         return s
18
19     def add_edge(self, vertex, weight=0):
20         '''
21         vertex: an other node object
22         weight: weight of the arc self -> otherVertex
23         '''
24         self.edges.append((vertex, weight))
25
26
27     def add_multiple_edges(self, vertices, weights = None):
28         '''
29         vertices: list of node objects
30         weights (optional): list of numbers
31         '''
32         if weights is None:
```

```
33                 weights = [0 for v in vertices]
34             for i in range(len(vertices)):
35                 self.edges.append(tuple((vertices[i], weights[i])))
36 
37         def edge_list(self):
38             return self.edges
39 
40         def delete_edge(self, node):
41             for n in self.edges:
42                 if n == self:
43                     self.edges.remove(node)
44                     break
```

Listing A.4: Fichero node.py

A.5 Clase Grafo

```
1  from node import Node
2  '''Graph class for general purpouses
3  David Picornell <dapicar(at)inf(dot)upv(dot)es>
4  Vicent Blanes <viblasel(at)inf(dot)upv(dot)es>
5  '''
6  class Graph:
7  
8      def __init__(self):
9          '''
10         Standar constructor, make a void graph
11         '''
12         self.nodes = list()
13 
14     def __str__(self):
15         s = ''
16         for i in self.nodes:
17             s+= str(i)+'\n'
18         return s
19 
20     def add_node(self, node):
21         self.nodes.append(Node(node))
22 
23     def delete_node(self, node):
24         self.nodes.remove(node)
25         for v in self.nodes:
26             v.delete_edge(node)
27 
28     def get_node(self, name):
29         node = None
30         for i in self.nodes:
31             if i.name == name:
32                 node = i
33                 break
34         return node
35 
36 def test():
37     nodes = []
38     g = Graph()
39     for i in range (5):
40         nodes.append(Node(str(i)))
41     for n in nodes:
42         if n.name == '3':
43             g.add_node(n)
44             continue
45         n.add_multiple_edges(nodes, [10*i for i in range(len(nodes))])
46         g.add_node(n)
47     print(g)
48 
49 
50 if __name__ == '__main__':
51     test()
```

Listing A.5: Fichero graph.py

A.6 Analizador texto

```
from carta import Carta
import operacionesbd as obd
from nltk.corpus import stopwords


def extraer_tipo(carta):
    '''
    Dada una carta, si se trata de una criatura extrae su tipo
    '''
    tipo = carta.tipo
    if "Creature" in tipo:
        #recortamos la palabra Creature
        tipo = tipo.split('—')[1].strip()
        #buscamos donde aparece el ataque
        for i in range(len(tipo)):
            if tipo[i].isdigit():
                return tipo[:i].strip()


def obtener_tipos_criaturas(coleccion):
    '''
    Este metodo devuelve una lista de cadenas con los distintos
    tipos de criaturas existentes en la coleccion.
    Esta informacion es necesaria para evaluar las interacciones
    entre cartas
    '''
    cartas = obd.listar_cartas(coleccion)
    col_tipos = set()
    #por cada carta, si criatura queremos saber su tipo
    for c in cartas:
        tipo = extraer_tipo(c)
        #si no era una critura, sera nonetype
        if tipo is not None:
            col_tipos.add(tipo)
    return sorted(list(col_tipos))

#TODO
def calcular_sinergia(carta1, carta2):
    '''
    calcula la puntuacion entre dos cartas
    '''
    puntuacion = 0
    for color in carta1.color:
        if color in carta2.color:
            puntuacion+=1
    for subtipo in carta1.tipo.split():
        if subtipo in carta2.tipo:
            puntuacion = puntuacion + 0.5
    return puntuacion



def crear_relaciones(cartas):
    '''
    este metodo devuelve una lista de tuplas para poder ser insertada en la bd
    (nombre_carta_1, puntuacion, nombre_carta_2) : puntuacion > 0
    la manera de explorar asegura que no exista la misma relacion
    con orden distinto en las cartas, ej:
    (tragic slip, 2, gravecrawler) y (gravecrawler, 2, tragic slip)
    '''
    relaciones = []
    #ATENCION A LOS INDICES
    for i in range(len(cartas)):
        for j in range(i, len(cartas)):
            puntuacion = calcular_sinergia(cartas[i], cartas[j])
            if puntuacion == 0:
                continue
            relaciones.append((cartas[i].nombre, puntuacion, cartas[j].nombre))
    return relaciones

def extraer_palabras_clave(texto):
```

```
    palabras_limp = set()
    #quizas necesitemos tratamiento previo
    #descompon en palabras
    palabras = texto.split()
    #stopwords del ingles
    #usando un set porque tiene coste 1
    swords = set(stopwords.words('english'))
    for pal in palabras:
        if pal not in swords:
            palabras_limp.add(pal)
    #ahora tengo un palabras_limp (set) todas las palabras que no sean sw


if __name__ == '__main__':
    cartas = obd.listar_cartas("INS")
    relaciones = crear_relaciones(cartas)
    obd.insertar_relaciones_en_bd(relaciones)
```

Listing A.6: Fichero analizador_texto.py

A.7 InitBD

```
import operacionesbd as bd
import analizador_texto as an


#introduco todas las castas en la bd
bd.introducir_coleccion_en_bd()
#introducir las aristas
cartas = bd.listar_cartas("INS")
rel = an.crear_relaciones(cartas)
bd.insertar_relaciones_en_bd(rel)
```

Listing A.7: Fichero initbd.py

A.8 Exportador

```

'''
Esta clase dispone de funciones para exportar a diferentes formatos
(por ahora solo .cod para visualizacion en cockatrice)
'''
def exportDoc(pool, eleccion, nombrefich = 'res_algoritmo.cod'):

    # abro el fichero
    fich = open(nombrefich, 'w')
    # copio el esquema basico
    fich.write('<?xml version="1.0" encoding="UTF-8"?>\n')
    fich.write('<cockatrice_deck version="1">\n')
    fich.write('<deckname></deckname>\n')
    fich.write('<comments></comments>\n')
    fich.write('<zone name="pool">\n')

    pool = convertir_diccionario(pool)
    eleccion = convertir_diccionario(eleccion)
    #pinto el pool
    for k in pool:
        fich.write('\t<card number="'+str(pool[k])+'" price="0" name="'+k+'"/>\n')
    #cierre pool
    fich.write('</zone>\n')
    fich.write('<zone name="eleccion">\n')
    #pinto la eleccion
    for e in eleccion:
        fich.write('\t<card number="'+str(eleccion[e])+'" price="0" name="'+e+'"/>\n')
    fich.write('</zone>\n')
    #cierro el fichero
```

```
    fich.write('</cockatrice_deck>')
    fich.close()


def convertir_diccionario(cartas):
    '''
    Esta funcion convierte una lista de cartas en un diccionario
    para facilitar la expotacion a .cod y derivados
    '''
    dicc = {}
    for c in cartas:
        if c.nombre_original in dicc:
            dicc[c.nombre_original] += 1
        else:
            dicc[c.nombre_original] = 1
    return dicc
```

Listing A.8: Fichero exportador.py

A.9 Generador de sobres

```
import operacionesbd as bd
from carta import Carta
from random import choice
from random import randint

class Generador:
    def __init__(self):
        self.cartas = bd.listar_cartas('INS')


    #TODO
    #Hay que mejorar la simulacion del sobre con reglas extras
    def simular_sobre(self):
        selec = []
        comunes = [carta for carta in self.cartas if carta.rareza == 'C']
        infrecuentes = [carta for carta in self.cartas if carta.rareza == 'U']
        for i in range(10):
            selec.append(choice(comunes))
        for i in range(3):
            selec.append(choice(infrecuentes))
        raras = []
        decision = randint(1,8)
        if decision <= 7:
            raras = [carta for carta in self.cartas if carta.rareza == 'R']
        else:
            raras = [carta for carta in self.cartas if carta.rareza == 'M']
        selec.append(choice(raras))
        return selec


if __name__ == '__main__':
    g = Generador()
    print(g.simular_sobre())
```

Listing A.9: Fichero generador_sobres.py

A.10 Gestor de pools

```
import pickle
from os import listdir
from generador_sobres import Generador
import sys
import operacionesbd as obd
'''
Este fichero proporciona los recursos para gestionar
```

```
pools de cartas predefinidos con el objetivo de hcer pruebas
de forma "derterminista"
'''
global path
path = 'data/pools'
def crear_pool(retorno = False):
    '''
    Crea un pool aleatorio. Si retorno es falso, lo escribe en la
    carpeta /data/pool. Si es true lo devuelve con un return
    '''
    pool = []
    g = Generador()
    for i in range(6):
        pool = pool+g.simular_sobre()
    pool = sorted(pool, key=lambda k:(k.color, k.cmc(), k.nombre))
    if retorno:
        return pool
    else:
        global path
        numeracion = len(listdir(path))
        pickle.dump(pool, open(path+'/pool'+str(numeracion)+'.p', 'wb'))

def cargar_pool(numpool):
    '''
    Carga y devuelve un pool de cartas
    '''
    global path
    return pickle.load(open(path+'/pool'+str(numpool)+'.p', 'rb'))

def print_pool(pool):
    '''
    Pinta por pantalla un pool dado
    en sistema unix se puede redireccion con '>'
    '''
    for p in pool:
        print(p)

def cargar_lista(path):
    lineas = open(path, 'r').read().split('\n')
    #print(lineas)
    cartas = []
    for lin in lineas:
        if not lin.strip():
            continue
        query = "select * from Carta where nombre='"+lin.strip()+"'"
        c = obd.ejecuta_query(query)
        if c:
            cartas.append(obd.linea_cursor_a_carta(c[0]))
    return cartas

def cargar(abspath):
    return pickle.load(open(abspath, 'rb'))

if __name__ == '__main__':
    print_pool(cargar_lista('test.txt'))
    #print_pool(cargar_pool(sys.argv[1]))
    '''
    if len(sys.argv)==2:
        print_pool(cargar_pool(sys.argv[1]))
    else:
        crear_pool()
    '''
```

Listing A.10: Fichero gestor_pool.py

A.11 Monitorización

```
def histograma(cartas, texto=True):
    '''
    Dado un conjunto de cartas seleccionadas devuelve
```

```
    un histograma. Si se espeficica texto = false devuelve un diccionario con
    las frecuencias. En caso contrario un string con la informacion
    '''
    frec = dict()
    for c in cartas:
        if c.cmc() in frec:
            frec[c.cmc()] += 1
        else:
            frec[c.cmc()] = 1
    if texto:
        cadena = '#####\nHISTOGRAMA\n#####\n'
        llaves = sorted(frec.keys())
        for k in llaves:
            cadena+= "Coste "+str(k)+" -> "+str(frec[k])+"\n"
        return cadena
    else:
        return frec


def imprime_resultados(pool, elegidos):
    f = open('resultado_algoritmo.txt', 'w')
    f.write('################\nPOOL\n################\n')
    for p in pool:
        f.write(str(p)+' -> '+str(p.nota_fireball)+'\n')
    f.write('################\nELECCION\n################\n')
    for e in elegidos:
        f.write(str(e)+' -> '+str(p.nota_fireball)+'\n')
    f.close()
```

Listing A.11: Fichero monitorizacion.py

A.12 OperacionesBD

```
#import pymysql
import sqlite3 as sql
from carta import Carta
import sys
import pickle
import operacionesbd as obd

def crea_conexion():
    # crea la conexion a la bd
    #bd = pymysql.connect('localhost','magicuser','magictfg','magicbd')
    bd = sql.connect('bd.sqlite')
    return bd


def ejecuta_query(query):
    # crea la conexion
    bd = crea_conexion()
    # obten el cursor
    cursor = bd.cursor()
    try:
        # query + commit
        cursor.execute(query)
        #bd.commit()
    except:
        return False
        # si error -> rollback
        print(sys.exc_info()[0])
        #bd.rollback()
    #devuelve el resultado del cursor para instrucciones select
    res = cursor.fetchall()
    bd.commit()
    bd.close()
    return res

def ejecuta_multiples_queries(queries):
    #crea la conexion
    bd = crea_conexion()
```

```
38      # obten el cursor
39      cursor = bd.cursor()
40      try:
41          for query in queries:
42              # query + commit
43              cursor.execute(query)
44          #bd.commit()
45      except:
46          # si error -> rollback
47          print(sys.exc_info()[0])
48          #bd.rollback()
49      #devuelve el resultado del cursor para instrucciones select
50      res = cursor.fetchall()
51      bd.commit()
52      bd.close()
53      return res
54
55  def crea_tablas_bd():
56      # crea la conexion
57      bd = crea_conexion()
58      # lee el script de creacion de la bd
59      archivo = open('./magicbd.sql')
60      script = archivo.read()
61      archivo.close()
62      # ahora lo ejecutamos como un query normal
63      ejecuta_query(script)
64      bd.close()
65
66  def insertar_carta(carta):
67      # crea todo el query
68      query = str("INSERT INTO Carta (nombre, nombre_original,coleccion, color, coste_mana
        , rareza,"
69      +"nota_fireball, coste_medio, texto, tipo) VALUES('"+carta.nombre+"', '"+carta.
        nombre_original+"', '"+carta.coleccion+
70      "', '"+carta.color+"', '"+carta.coste_mana+"', '"+carta.rareza+"', "+str(carta.
        nota_fireball)+
71      ", "+str(carta.coste_medio)+", '"+carta.texto+"', '"+carta.tipo+"')")
72      # ejecuto el query
73      return ejecuta_query(query)
74
75  def introducir_coleccion_en_bd():
76      '''
77      Este metodo toma el path a el array con las cartas y las inserta en la bd
78      '''
79      cartas = pickle.load(open("data/output/cartas.p", "rb"))
80      for elem in cartas:
81          #print(elem)
82          #print("------\n")
83          if insertar_carta(elem) is False:
84              print(elem.tostring())
85
86
87  def listar_cartas(coleccion):
88      '''
89      Este metodo devuelve un array con todas las cartas de la coleccion
90      '''
91      cartas = []
92      query = "select * from Carta where coleccion = '"+coleccion+"'"
93      #ejecutamos
94      res = ejecuta_query(query)
95      for linea in res:
96          cartas.append(linea_cursor_a_carta(linea))
97      return cartas
98
99  def linea_cursor_a_carta(lineacursor):
100     '''
101     Esta funcion convierte una linea recuperada del cursor en
102     un objeto del tipo carta
103     '''
104     return Carta(
105                 nombre = lineacursor[0],
106                 nombre_original = lineacursor[1],
107                 coleccion = lineacursor[2],
108                 color = lineacursor[3],
```

```
                coste_mana = lineacursor[4],
                rareza = lineacursor[5],
                nota_fireball = lineacursor[6],
                coste_medio = lineacursor[7],
                tipo = lineacursor[8],
                texto = lineacursor[9]
    )

def insertar_relaciones_en_bd(relaciones):
    '''
    dada una lista de tuplas que representa las relaciones entre cartas
    (com_carta1, puntuacion, nom_carta2)
    las introduce en la bd
    '''
    queries = []
    for rel in relaciones:
        #crea el query cart1, cart2, valor
        query = "INSERT INTO Sinergias values('" + str(rel[0] +"', '"+rel[2]+
        "', "+str(rel[1]))+")"
        queries.append(query)
    ejecuta_multiples_queries(queries)

def valor_sinergia(carta1, carta2):
    query = str("select * from Sinergias where (carta1 ='" + carta1.nombre
    +"' and carta2='"+carta2.nombre+"') or (carta2 ='"+carta1.nombre+"'and carta1='"+
    carta2.nombre+"')")
    out = ejecuta_query(query)
    #si no hay lineas en la respuesta la sinergia entre 2 cartas es 0
    if len(out)==0:
        return 0
    #si existe una linea devuelvo es valor de su sinergia (linea 0 pos 2)
    else:
        return out[0][2]

# main
if __name__ == '__main__':
    print(valor_sinergia('aberrantresearcher', 'fleetingmemories'))
```

Listing A.12: Fichero operacionesbd.py

A.13 SQL: Fichero para crear las tablas en la BD

```
DROP TABLE IF EXISTS Carta_Pool;
DROP TABLE IF EXISTS Carta_Elegido;
DROP TABLE IF EXISTS Pool;
DROP TABLE IF EXISTS Elegido;
DROP TABLE IF EXISTS Registro;
DROP TABLE IF EXISTS Usuario;
DROP TABLE IF EXISTS Sinergias;
DROP TABLE IF EXISTS Carta;
DROP TABLE IF EXISTS Coleccion;

ALTER DATABASE magicbd CHARACTER SET utf8 COLLATE utf8_unicode_ci;

CREATE TABLE Coleccion(
  nombre VARCHAR (25) UNIQUE NOT NULL,
  codigo VARCHAR(3) PRIMARY KEY,
  bloque VARCHAR (25)
);

CREATE TABLE Carta(
  nombre VARCHAR(25) NOT NULL,
  nombre_original VARCHAR(30) NOT NULL,
  coleccion VARCHAR(3) NOT NULL,
  color VARCHAR (1) NOT NULL,
  coste_mana TEXT NOT NULL,
  rareza VARCHAR(1) NOT NULL,
  nota_fireball REAL NOT NULL,
  coste_medio REAL,
  tipo text NOT NULL,
```

```
29   texto text NOT NULL,
30   FOREIGN KEY (coleccion) references Coleccion(codigo),
31   PRIMARY KEY (nombre, coleccion)
32 );
33
34 CREATE TABLE Sinergias(
35   carta1 VARCHAR(25) NOT NULL,
36   carta2 VARCHAR(25) NOT NULL,
37   nota REAL NOT NULL,
38   FOREIGN KEY (carta1) references Carta(nombre),
39   FOREIGN KEY (carta2) references Carta(nombre),
40   PRIMARY KEY (carta1, carta2)
41 );
42
43   CREATE TABLE Usuario(
44     username VARCHAR(25) NOT NULL PRIMARY KEY,
45     password TEXT NOT NULL
46   );
47 CREATE TABLE Registro(
48   id INTEGER AUTO_INCREMENT PRIMARY KEY,
49   usuario VARCHAR(25) NOT NULL,
50   FOREIGN KEY (usuario) references Usuario(username)
51 );
52
53 CREATE TABLE Pool(
54   id INTEGER AUTO_INCREMENT PRIMARY KEY,
55   registro INTEGER NOT NULL,
56   FOREIGN KEY (registro) references Registro(id)
57 );
58
59 CREATE TABLE Elegido(
60   id INTEGER AUTO_INCREMENT PRIMARY KEY,
61   registro INTEGER NOT NULL,
62   FOREIGN KEY (registro) references Registro(id)
63 );
64
65 CREATE TABLE Carta_Pool(
66   carta VARCHAR(25) NOT NULL,
67   id_pool INTEGER NOT NULL,
68   FOREIGN KEY (carta) references Carta(nombre),
69   FOREIGN KEY (id_pool) references Pool(id),
70   PRIMARY KEY (carta, id_pool)
71 );
72
73 CREATE TABLE Carta_Elegido(
74   carta VARCHAR(25) NOT NULL,
75   id_elegido INTEGER NOT NULL,
76   FOREIGN KEY (carta) references Carta(nombre),
77   FOREIGN KEY (id_elegido) references Elegido(id),
78   PRIMARY KEY (carta, id_elegido)
79 );
80
81 ALTER TABLE Carta CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
82 ALTER TABLE Sinergias CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
83 ALTER TABLE Usuario CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
84 ALTER TABLE Registro CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
85 ALTER TABLE Pool CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
86 ALTER TABLE Elegido CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
87 ALTER TABLE Carta_Pool CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
88 ALTER TABLE Carta_Elegido CONVERT TO CHARACTER SET utf8 COLLATE utf8_unicode_ci;
```

Listing A.13: Fichero magicbd.sql

APÉNDICE B
Código interfaz javafx

B.1 Clase principal

```
1  package vista;
2
3  import java.io.IOException;
4
5  import javafx.application.Application;
6  import javafx.fxml.FXMLLoader;
7  import javafx.stage.Stage;
8  import javafx.scene.Scene;
9  import javafx.scene.layout.AnchorPane;
10
11
12 public class Principal extends Application {
13
14   private Stage primaryStage;
15     private AnchorPane rootLayout;
16
17
18   @Override
19   public void start(Stage primaryStage) {
20     this.primaryStage = primaryStage;
21         this.primaryStage.setTitle("Interfaz constructor de mazos");
22         initRootLayout();
23   }
24
25   public void initRootLayout() {
26         try {
27             // Load root layout from fxml file.
28             FXMLLoader loader = new FXMLLoader();
29             loader.setLocation(Principal.class.getResource("Principal.fxml"));
30
31             rootLayout = (AnchorPane) loader.load();
32
33             // Show the scene containing the root layout.
34             Scene scene = new Scene(rootLayout);
35             scene.getStylesheets().addAll(this.getClass().getResource("style.css").
     toExternalForm());
36             rootLayout.setId("princ");
37             primaryStage.setScene(scene);
38             primaryStage.setResizable(false);
39             //primaryStage.setResizable(false);
40             primaryStage.show();
41
42         } catch (IOException e) {
43             e.printStackTrace();
44         }
45     }
46
47   public static void main(String[] args) {
48     launch(args);
49   }
50 }
```

Listing B.1: Fichero Principal.java

B.2 Controlador de la interfaz

```
package vista;

import java.io.BufferedReader;
import java.io.File;
import java.io.InputStreamReader;
import java.net.URL;
import java.util.ResourceBundle;

import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.control.Alert;
import javafx.scene.control.Alert.AlertType;
import javafx.scene.control.Button;
import javafx.stage.FileChooser;
import javafx.stage.FileChooser.ExtensionFilter;
import javafx.stage.Stage;

public class PrincipalController implements Initializable {

  @FXML
  private Button btSeleccionar;
  @FXML
  private Button btGenerar;

  public void initialize(URL location, ResourceBundle resources) {
    // TODO Auto-generated method stub
    btSeleccionar.setId("record-sales");
    btGenerar.setId("record-sales");

  }

  public void seleccionar() {
    String path = fchooser();
    String res = ejecutarconsola("python3 constructor.py "+ path);
    muestraAlert(res);

  }

  public void generar() {
    String res = ejecutarconsola("python3 constructor.py -1");
    muestraAlert(res);
  }

  private String ejecutarconsola(String command) {
    StringBuffer output = new StringBuffer();

    Process p;
    try {
      p = Runtime.getRuntime().exec(command);
      p.waitFor();
      BufferedReader reader = new BufferedReader(new InputStreamReader(p.getInputStream
    ()));

      String line = "";
      while ((line = reader.readLine()) != null) {
        output.append(line + "\n");
      }

    } catch (Exception e) {
      e.printStackTrace();
      return null;
    }

    return output.toString();

```

```
65    }
66
67    /*
68     * ESTE METODO ABRE UNA VENTANA FILECHOOSER Y RETORNA EL PATH DEL FICHERO
69     * ELEGIDO O NULL
70     */
71    private String fchooser() {
72      FileChooser fileChooser = new FileChooser();
73      fileChooser.setTitle("Selecciona un fichero");
74      fileChooser.getExtensionFilters().addAll(new ExtensionFilter("Text Files", "*.txt"),
75          new ExtensionFilter("Pickle pool", "*p"));
76      File selectedFile = fileChooser.showOpenDialog(new Stage());
77      if (selectedFile == null)
78        return null;
79      return selectedFile.getPath();
80    }
81
82    /*MUESTRA UN ALERT CUANDO EL ALGORITMO TERMINA*/
83    private void muestraAlert(String texto){
84      Alert alert = new Alert(AlertType.INFORMATION);
85      alert.setTitle("Seleccion completa!");
86      alert.setResizable(true);
87      alert.setHeaderText(null);
88      alert.setContentText("Puedes encontrar el resultado en el fichero res_algortimo.cod
        !\n"+texto);
89      alert.showAndWait();
90    }
91
92  }
```

Listing B.2: Fichero PrincipalController.java

B.3 Fichero FXML de la interfaz

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <?import java.lang.*?>
4  <?import javafx.scene.control.*?>
5  <?import javafx.scene.layout.*?>
6  <?import javafx.scene.layout.AnchorPane?>
7  <?import javafx.scene.paint.*?>
8  <?import javafx.scene.text.*?>
9
10 <AnchorPane prefHeight="446.0" prefWidth="563.9999000000025" xmlns:fx="http://javafx.com
       /fxml/1" xmlns="http://javafx.com/javafx/2.2" fx:controller="vista.
       PrincipalController">
11   <!-- TODO Add Nodes -->
12   <children>
13     <Button fx:id="btSeleccionar" layoutX="84.0" layoutY="363.0" mnemonicParsing="false"
        onAction="#seleccionar" prefWidth="135.0" text="Seleccionar" />
14     <Button fx:id="btGenerar" layoutX="361.0" layoutY="363.0" mnemonicParsing="false"
       onAction="#generar" prefWidth="135.0" text="Generar" />
15     <Separator layoutX="282.0" layoutY="12.0" minWidth="6.103515625E-5" prefHeight="
       434.0" prefWidth="6.103515625E-5" />
16     <Label alignment="CENTER" contentDisplay="CENTER" layoutX="41.0" layoutY="85.0"
       prefHeight="122.0" prefWidth="202.0" text="Selecionar pool" textAlignment="LEFT"
       wrapText="true">
17       <font>
18         <Font name="System Bold" size="18.0" fx:id="x1" />
19       </font>
20       <textFill>
21         <Color blue="0.400" green="0.688" red="1.000" fx:id="x2" />
22       </textFill>
23     </Label>
24     <Label alignment="CENTER" font="$x1" layoutX="301.0" layoutY="81.0" prefHeight="
       131.0" prefWidth="236.0" text="Generar pool aleatorio" textAlignment="RIGHT"
       textFill="$x2" textOverrun="CLIP" />
25   </children>
26 </AnchorPane>
```

Listing B.3: Fichero Principal.fxml

B.4 CSS de la interfaz

```
#princ{
        -fx-background-image: url("./imagefx.jpg");
        -fx-background-repeat: stretch;
        -fx-background-size: 900.0 506.0;
        -fx-background-position: center center;
        -fx-effect: dropshadow(three-pass-box, black, 30.0, 0.5, 0.0, 0.0);
    }

#record-sales {
    -fx-padding: 8 15 15 15;
    -fx-background-insets: 0,0 0 5 0, 0 0 6 0, 0 0 7 0;
    -fx-background-radius: 8;
    -fx-background-color:
        linear-gradient(from 0% 93% to 0% 100%, #a34313 0%, #903b12 100%),
        #9d4024,
        #d86e3a,
        radial-gradient(center 50% 50%, radius 100%, #d86e3a, #c54e2c);
    -fx-effect: dropshadow( gaussian , rgba(0,0,0,0.75) , 4,0,0,1 );
    -fx-font-weight: bold;
    -fx-font-size: 1.0em;
}
```

Listing B.4: Fichero style.css