Proyecto Final: Lichess.
Bases de Datos No Estructuradas

Guillermo Gerardo Andrés Urbano
Fernando Avitúa Varela
Fernando Santa Rita Vizuet

30 de julio de 2022

Introducción

Lichess es una página gratuita que organiza información acerca de jugadores y partidas de ajedrez. Junto con Chess.com es una de las páginas más importantes para jugadores activos. Además de ser host de partidas y torneos, Lichess cuenta con secciones de estadísticas y quizzes que permiten a un jugador ir más allá de sólo jugar para mejorar. Dado que el ajedrez es un juego con reglas sencillas, este tipo de información es muy útil y fértil para responder preguntas acerca de la trayectoria del aprendizaje humano.

Actualmente, la página ya ocupa algoritmos de inteligencia artificial para analizar partidas y posiciones del juego, incluso a nivel de cada jugada. Así, este trabajo busca expandir el tipo de análisis que se puede realizar para tener un entendimiento cada vez mayor de qué es lo que hace a un buen jugador y cómo ayudar a las personas a mejorar.

Para esto, notamos que la estructura de enfrentamientos entre jugadores y su agrupación en equipos se presta a organizarla en un grafo donde los vértices son jugadores y las aristas entre ellos sus resultados.

La información de los jugadores se obtuvo de la API de Lichess. Contemplamos casos aislados dada la limitante del número de requests diarios, pero el análisis se puede extrapolar como si fuéramos dueños de la base de datos, añadiendo cada partida conforme vaya ocurriendo.

Diseño e Implementación

Para almacenar la información de los enfrentamientos de los jugadores usamos una base de datos tipo grafo alojada usando **Neo4j**. En dicha base sólo se almacenó por cada vértice el id del jugador.

Por otro lado, para no cargar la información de cada jugador en los nodos, se almacenó la información pertinente para cada análisis en una base de datos tipo documento (usamos **MonogoDB**). Así, el análisis

de los datos comienza haciendo consultas en el grafo de **Neo4j** y después, sabiendo los ids de los usuarios, se extraen las estadísticas de la base de datos en *MongoDB*. A continuación se muestra un ejemplo del diseño en dos partes:

Primero hacemos la siguiente consulta en **Neo4j**:

```
match (a:Usuario {id:"salvadoreps"}) return a
```

El resultado se puede mostrar en la interfaz grafo alojada en localhost

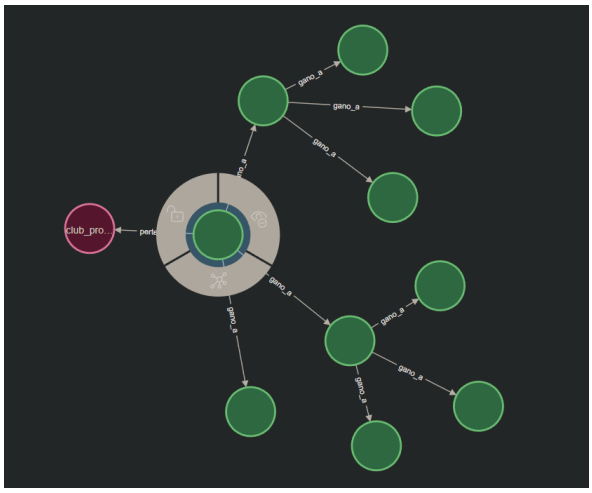

Figura 1: Resultado de la consulta que busca al usuario salvadoreps, podemos ver en rojo el equipo al que pertenece: *club_prometeo* así como aristas de aquellas personas que ha derrotado (y las que han derrotado aquellas personas), las cuales no pertenecen a ningún equipo.

Después vemos la entrada que almacenamos en mongoDB usando el software **MongoDB Compass**

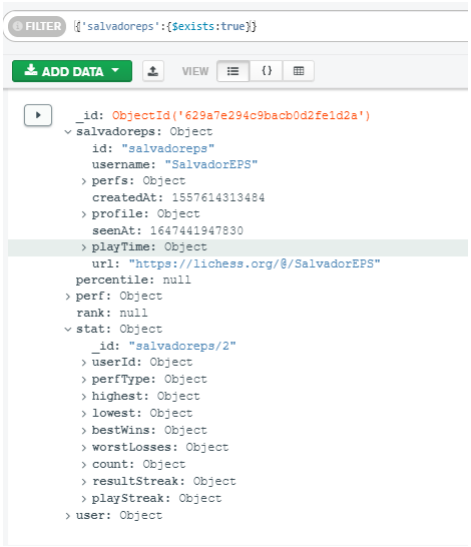

Figura 2: Entrada en **MongoDB** del usuario salvadoreps.

1. Consultas

1.1. Análisis de profundidad de la red de jugadores

En Lichess tenemos mucha información estadística de cada jugador, dentro de esta información se incluye:

- Rating: El rating actual de cada jugador en el sistema elo por cada modo de juego.
- Historial de rating: grafos de cómo ha aumentado o disminuido el rating a través del tiempo
- Mejores victorias: las victorias contra oponentes con mayor rating
- Peores derrotas: las derrotas contra oponentes con peor rating

Esta información nos ayuda a hacer un análisis de cada jugador, sin embargo, existen datos que podrían encontrarse en el grafo de enfrentamientos que podrían ayudarnos a obtener conclusiones novedosas para tener mejores sistemas de recomendación y detección de tendencias.

Las bases de datos de tipo grafo tienen la ventaja de poder realizar búsquedas a profundidad de manera mucho más rápida que cualquier otro tipo de base de datos. Si hiciéramos el almacenamiento de cada enfrentamiento en una base de datos relacional tendríamos una tabla como la que se observa a continuación

id1	id2
alanmezal	vladgert
alanmezal	fux58
alanmezal	medlar
medlar	sargan7
medlar	munjo123
...	...

Figura 3: Tabla relacional donde el *id1* es el jugador que venció al jugador con el *id2*

Para tener información de las personas que han sido derrotadas por aquellos que han derrotado otros jugadores tendríamos que hacer un join de la tabla 11 consigo misma, como se muestra a continuación

id1_x	id2_x	id1_y	id2_y
alanmezal	vladgert	vladgert	nassa2012
alanmezal	vladgert	vladgert	zen_knight
alanmezal	vladgert	vladgert	achitescu
alanmezal	fux58	fux58	nadir_sofiani
alanmezal	fux58	fux58	patrickmilen
...	...	...	...

Figura 4: Tabla relacional donde el *id1_x* es el jugador que venció al jugador con el *id2_x* y *id2_y* son aquellos que venció *id2_x*

Usando la base de datos tipo grafo la información ya está organizada para no tener que hacer joins y la manera

de obtener la misma información se puede hacer sin crear una nueva tabla. A continuación se muestra la consulta que regresaría los segundos vecinos de cada usuario

```
match (u1:Usuario)
match (u2:Usuario)
match (u3:Usuario)
match (u1)-[:gano_a]->(u2)-[:gano_a]->(u3)
return u1,u3
```

Aprovechando esta funcionalidad, hicimos un análisis del equipo de la facultad de ciencias para tener una métrica distinta de qué jugador tiene más potencial dentro de la modalidad de juego *blitz*. Para esto, recopilamos los datos de los usuarios de las tres mejores victorias de cada usuario (*oponentes_1*, esto se realizó con la intención de obtener el potencial de un jugador con el supuesto de que la mejor victoria da una indicación del nivel máximo de cada jugador. Esta misma suposición se puede realizar con un nivel de profundidad mayor; es decir, recopilando la información sobre las mejores victorias de cada usuario en *oponentes_1*, así obteniendo una lista, *oponentes_2*.

Se obtuvo esta información usando la API de LIchess para los 3 vecinos más fuertes de cada jugador. Así, se crea el grafo que se muestra a continuación

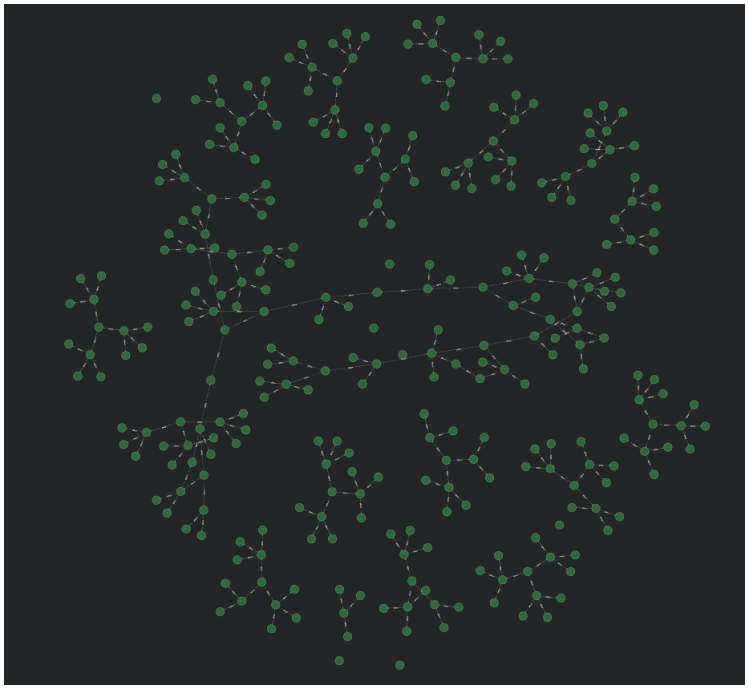

Figura 5: Representación de 300 vértices del grafo obtenido al buscar los 3 oponentes más fuertes de jugadores de *club_prometeo* a 2 niveles de profundidad

Primero observamos los rankings que haríamos usando sólo las métricas de un vértice. Para esto, usamos el rating máximo y el rating actual que ha alcanzado cada jugador en el equipo de *club_prometeo*. En la siguiente figura se muestran a los mejores 13 jugadores de acuerdo a cada métrica

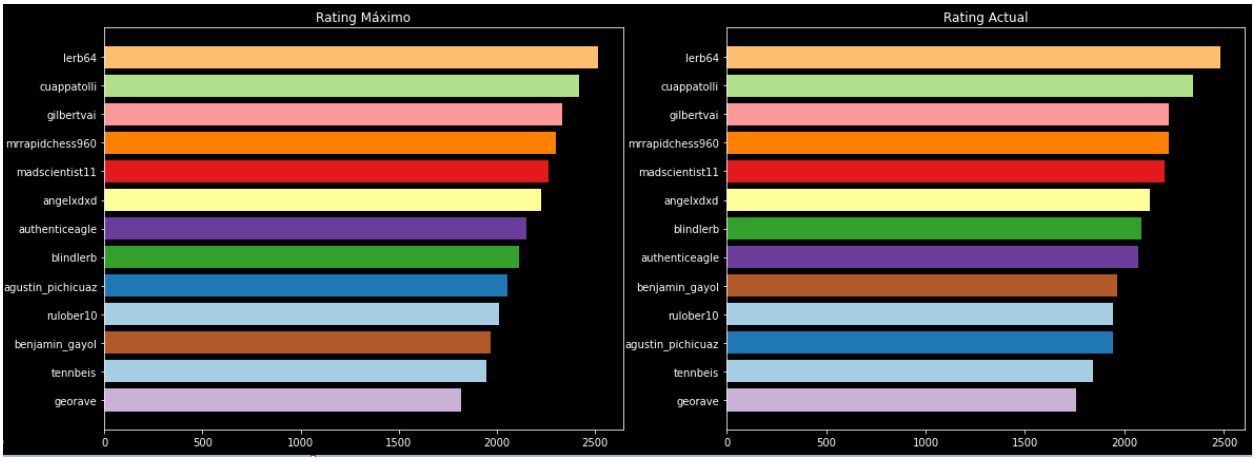

Figura 6: Mejores jugadores de acuerdo a las métricas de un vértice: rating máximo y rating actual

Vemos que las posiciones son casi idénticas.

Sin embargo, haciendo un promedio de los ratings de *oponentes_2* se obtiene el siguiente ranking

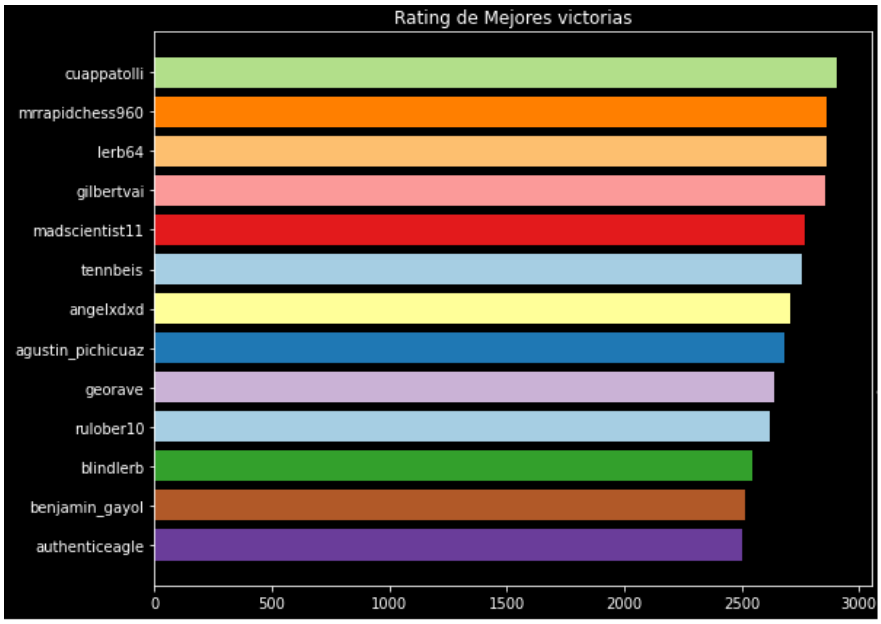

Figura 7: Mejores jugadores de acuerdo al promedio de rating de *oponentes_2* a dos niveles de profundidad en la grafo

Comparando con las posiciones en 6 vemos que hay muchos cambios en los rankings. Si escogemos este análisis para seguir la trayectoria de los jugadores podríamos observar qué tan correcto es para el desempeño de cada jugador en torneos. De probarse útil, se podría tomar esta métrica y tomar a cada uno de estos jugadores como prometedores. Así podríamos asignar mentores que pudieran aprovechar este potencial.

1.2. Análisis de los mejores jugadores de Blitz por cada equipo

Para esta consulta, haremos un muestro a través de la API con 10 jugadores por cada equipo, esto para no saturar la peticiones que nos ofrece API REST y no saaturar el tiempo de carga en la inserción y creación de lo nodos. Para este enfoque utilizaremos MongoDB para guardar cada JSON devuelto de los jugadores y equipos en diferentes colección para posteriormente extraer la información a través de una consulta.

```
equipo_top5 = {}

# Obtención de los equipos
query = f'''
MATCH (equipo:Equipo)
RETURN (equipo)
'''.replace('\n', '')
nodos_equipos = session.run(query).data()

# Obtencion de los usuarios de cada equipo
for equipo in nodos_equipos:
    nombre_equipo = equipo['equipo']['nombre']
    query=f'''
    MATCH (e:Equipo {{nombre: "{nombre_equipo}"}})
    MATCH (u:Usuario)-[pe:pertenece]-(e)
    RETURN u, e
    '''
    nodos_miembros = session.run(query).data()

    # Obteniendo el top 5 por equipo
    ratings_miembro = []
    for miembro in nodos_miembros:
        usuario_id = miembro['u']['id']
        usuario = usuarios_col.find_one({f'{usuario_id}.id': f'{usuario_id}'})
        rating_blitz = usuario[usuario_id]['perfs']['blitz']['rating']
        username = usuario[usuario_id]['username']
        ratings_miembro.append((username, rating_blitz))
    top5_blitz = sorted(ratings_miembro, key=lambda x: x[1], reverse=True)[:5]

    equipo_top5[nombre_equipo] = top5_blitz
```

Como se menciono anteriormente, el diseño estará compuesto de nodos que representaran jugadores que tendrán como atributos el id del jugador, con esto podremos unir las dos fortalezas de los diferentes bases, ya que usaremos mongo con el id del jugador para obtener rating en la cateogoria Blitz (juego de ajedrez rapido) y de esta manera obtener los mejores.

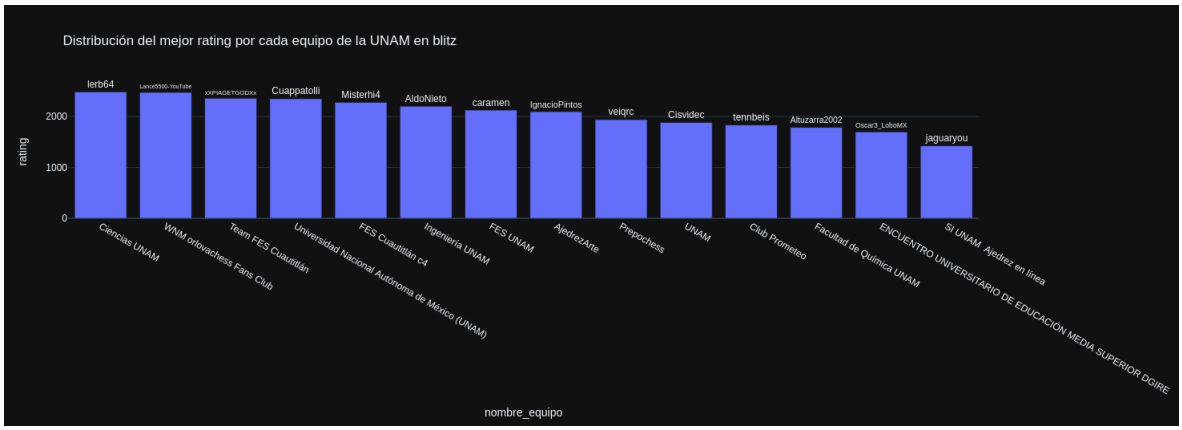

Figura 8: Distribución de los mejores rating por cada equipo de la UNAM

Cabe recalcar que nuestro análisis fue enfocado a los grupos de ajedrez de la UNAM, con esto podemos verificar a través de nuestra gráfica la distribuciones de los mejores jugadores a en base a su puntaje de Blitz. Podemos observar que existen grandes jugadores a cada uno de los equipos con puntajes mayores a 2000, como tambien otros equipos que estan bajos de puntaje con respecto al mejor de su miembros. Con esto vemos que la distribución no esta sesgado con el total de lo equipo analizados. Y como último comentario podemos descar el grupo de ajedrez de Ciencia UNAM el contiene al mejor de la UNAM en el momento que se hace este analizas a traves de nuestra muestra.

1.3. Análisis de coincidencias de las mejores jugadas en Blitz.

Para este próximo análisis queremos conocer si existen coincidencias con judagadores que han vencidos con jugadores del mismo equipo. Es decir, determinar si jugadores del mismo equipo han vencido a mismo jugadores. Con este análisis logramos determinar si los jugadores que vencen al mismo jugador tienen el mismo rating o que caracteristica deberia de tener.

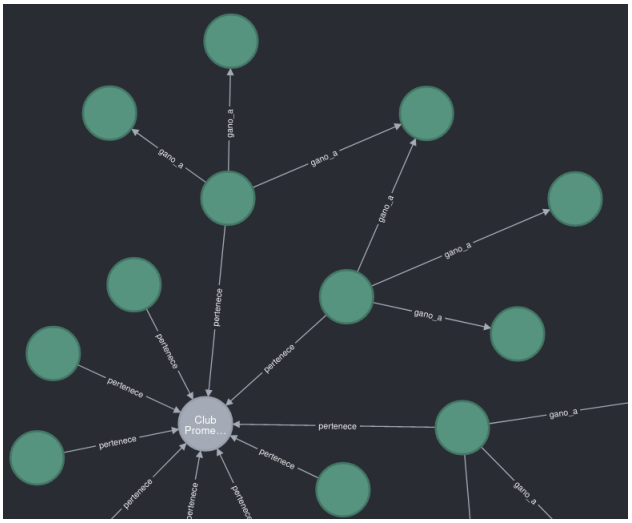

Figura 9: Ilustración de una coincidencia

Con esta gráfico podemos ver claramente un ejemplo de lo queremos lograr encontrar, buscar esos nodos en común dato un equipo. Para lograrlo generamos una query para encontrar los nodos que tienen el mismo equipo y regresar los nodos y aristas que coincida con la relación "gano_a"para despues a traves de python determinar las coincidencias. Como podemos en la salidos los jugadores .ªlejandror0210z .ªlanrv"le han ganado los dos al jugador 'maxmledesma' teniendo el mismo rating.

```
nombre_equipo = "Club Prometeo"

query = f'''
MATCH (e:Equipo {{nombre: "{nombre_equipo}"}})
MATCH (u:Usuario)-[ga:gano_a]-(u2:Usuario)
MATCH (e)-[pe:pertenece]-(u)
RETURN u, ga, u2
'''.replace('\n', '')
nodos_judadores = session.run(query).data()

relaciones_ganoa = [nodo_jugador['ga'] for nodo_jugador in nodos_judadores]
relaciones_ganoa = [(relacion[0]['id'] ,
                  relacion[2]['id'])for relacion in relaciones_ganoa]
usuarios2 = list(map(lambda x: x[1], relaciones_ganoa))
oponentes_coincidentes = list(filter(lambda x: usuarios2.count(x[1]) > 1,
                          relaciones_ganoa))
```

```
Output:
[('alejandror0210', 'maxmledesma'), ('alanrv', 'maxmledesma')]
Nombre Ranking
alejandror0210-1500
alanrv-1500
```

1.4. Análisis de jugadores que han crecido a lo largo del tiempo

Analizaremos el historial jugadores de un equipo para ver el crecimiento de cada uno de ellos y determinar quienes han ido mejorando en una ventana del tiempo.

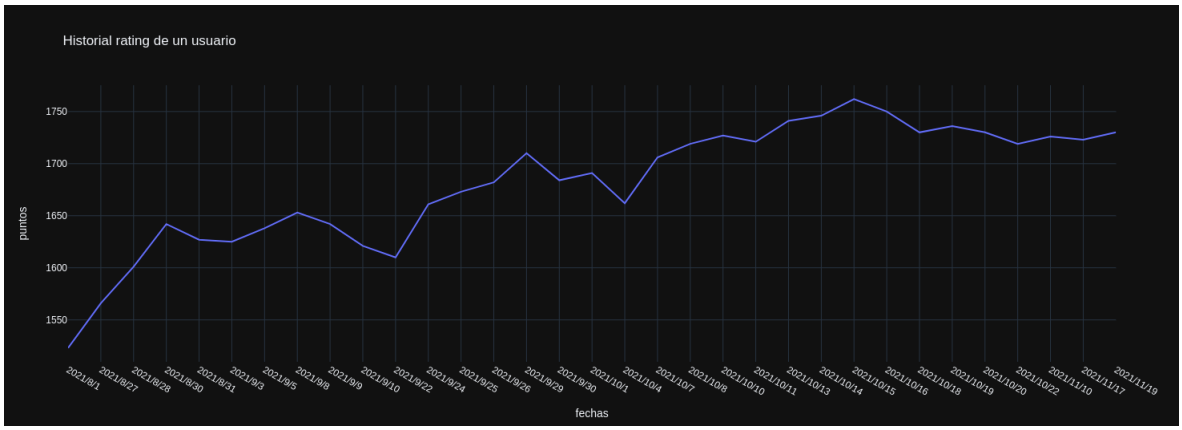

Figura 10: Historial del rating de un usuario

Por ejemplo, podemos observar el historial del rating de un jugador a traves de las fechas que nos proporciona

la API. Con esto podemos determinar si en las ultimas semanas el usuario a subiendo de poco en poco o a ido en picada. Para eso generamos una consulta de los usuarios de un equipo dado y a traves del id obtenemos el historial de rating, con tendremos la fechas y rating en un momento dado. Para determinar si un jugador sigue creciendo tomares las ultimas 5 fechas de la serie de tiempo y calcularemos sus diferencias entre ellas. Con esto podemos determinar si un jugador sigue creciendo o disminuyendo, ya que si disminuye su pendiente será negativa, en caso contrario su pendiente será positiva.

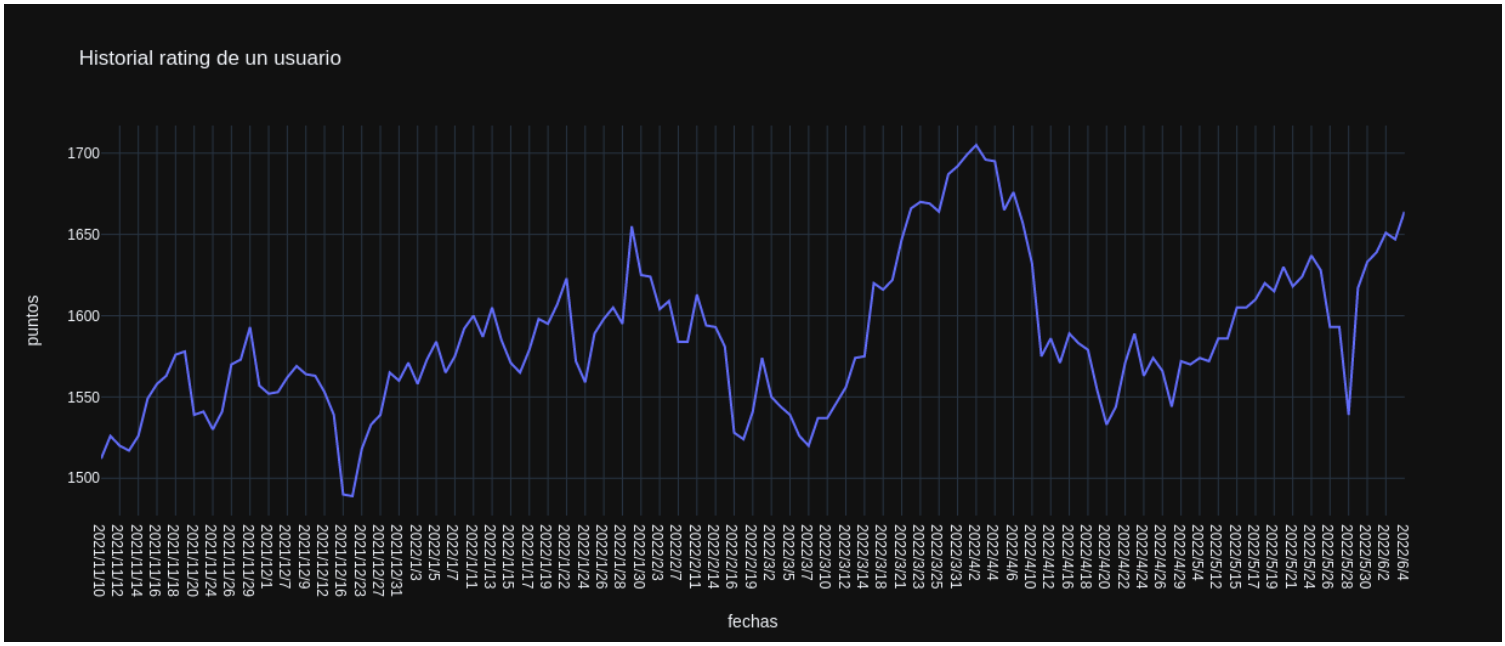

Figura 11: Ejemplo de un jugador que va creciendo

1.5. Análisis de los Juegos Universitarios 2022

Los Juegos Universitarios tienen por objetivo promover y fomentar estilos de vida saludable y trabajo en equipo, propiciando la integración entre las diversas entidades académicas de la UNAM y su Sistema Incorporado, fortaleciendo los procesos de detección temprana de talentos deportivos.

En el grupo oficial de ajedrez de la UNAM en Lichess podemos encontrar el evento deportivo referente a los Juegos Universitarios 2022 de ajedrez. Pueden formarse equipos en representación de cada una de las instituciones educativas (preparatorias, CCH's, Facultades, etc.).

Se tiene un torneo de tipo arena, lo cuál consiste en tener enfrentamientos con un control de tiempo de 3 minutos sin incremento por jugador. Acabando una partida, los jugadores se enfrentan contra otros jugadores, siempre ajenos a su propio equipo. El equipo (institución) con una mayor recolección de puntos será el ganador.

Se busca implementar una base de datos en **Neo4j** que contenga los resultados del torneo. Los jugadores constituyen los nodos, mientras que las relaciones entre nodos son aristas dirigidas con el resultado de las partidas jugadas. También existen relaciones del jugador a la institución para la cuál juega y, por tanto, la institución consta de otro nodo.

Las relaciones entre jugadores contendrán una métrica para medir que tan significativo puede ser el resultado en cada una de las partidas. Esto se hace mediante la asignación en cada una de las aristas el cociente:

$$\text{Valor de resultado} = \frac{\text{elo}_{loser}}{\text{elo}_{winner}}$$

Si el valor de resultado es mayor a 1, entonces tiene mucho mérito por parte del jugador más novato. Si el valor de resultado es menor a 1 y cercano a cero, entonces es un resultado esperado.

Si el promedio de ingrados es cercano a cero, te han ganado jugadores muy buenos y, por tanto, tuviste un torneo con el resultado esperado. Si el promedio de ingrados es mayor que 1, entonces te han ganado jugadores más débiles y, por tanto, tuviste mala suerte.

Si el promedio de exgrados es mayor a 1, tuviste un torneo muy bueno pues derrotaste a gente muy buena. Si el promedio de los exgrados es menor a 1, entonces ganaste sobre jugadores más débiles.

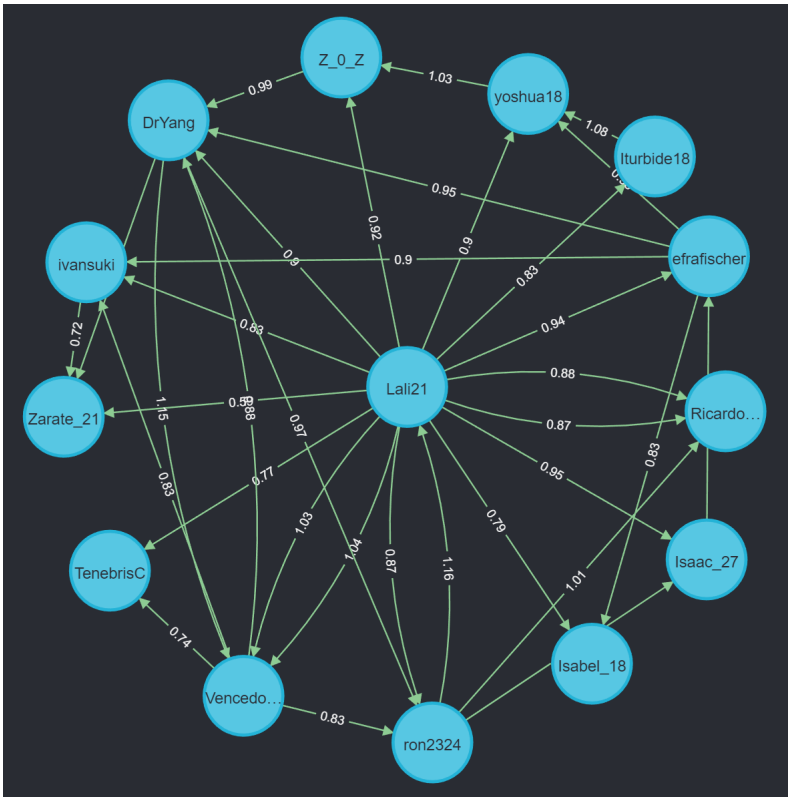

Figura 12: Partidas ganadas del jugador *Lali21* sobre otros jugadores en los Juegos Universitarios. Cada una de las aristas contiene el *valor de resultado* definido anteriormente.

1.5.1. Carga de la base de datos

Las instrucciones para la carga de cada una de las partidas emplea la siguiente función:

```
def agregar_neo4j(white, white_rating, white_team, black, black_rating, black_team,
                  result, valor_resultado):
    instruccion = f'''
    MERGE (u1:Usuario {{id:"{white}"}})
    MERGE (u2:Usuario {{id:"{black}"}})
    MERGE (t1:Equipo {{nombre_equipo:"{white_team}"}})
    MERGE (t2:Equipo {{nombre_equipo:"{black_team}"}})
    MERGE (u1)-[:juega_para]->(t1)
    MERGE (u2)-[:juega_para]->(t2)
    '''

    if result == 1:
        instruccion += f'''MERGE (u1)-[g:gano_a {{valor_resultado:"{valor_resultado}
        ", white_rating:"{white_rating}",black_rating:"{black_rating}"}}]->(u2)'''
    elif result == 0:
        instruccion += f'''MERGE (u2)-[g:gano_a {{valor_resultado:"{valor_resultado}
        ", white_rating:"{white_rating}", black_rating:"{black_rating}"}}]->(u1)'''
    elif result == 0.5:
        if white_rating >= black_rating:
            instruccion += f'''MERGE (u2)-[g:gano_a {{
            valor_resultado:"{valor_resultado}", white_rating:"{white_rating}",
            black_rating:"{black_rating}"}}]->(u1)'''
        else:
            instruccion += f'''MERGE (u1)-[g:gano_a
            {{valor_resultado:"{valor_resultado}", white_rating:"{white_rating}",
            black_rating:"{black_rating}"}}]->(u2)'''

    return instruccion
```

Obsérvese que las relaciones contienen la puntuación de los jugadores al momento de la partida, así como el *valor de resultado*.

1.5.2. Ranking de equipos

Aprovechando la estructura de la base de datos, podemos contar el número de exgrados de cada uno de los jugadores de cada equipo a fin da hallar el número de victorias de estos, lo cual permite la creación de un ranking por equipos.

Las funciones auxiliares para la extracción de exgrados y equipo de un jugador dado son:

```
def exgrados_jugador(jugador):
    """
    Obtiene los puntos por jugador via los exgrados de su nodo.
    """

    instruccion = f'''
    MATCH (u1:Usuario)
    MATCH (u2:Usuario)
    WHERE u1.id = '{jugador}'
```

```
    MATCH (u1)-[:gano_a]->(u2)
    RETURN count(*) as exgrados
    '''
    return session.run(instruccion).data()[0]['exgrados']

def equipo_jugador(jugador):
    """
    Obtiene el equipo de un jugador dado desde la base de Neo4j.
    """

    instruccion = f'''
    MATCH (u:Usuario)
    MATCH (t:Equipo)
    WHERE u.id = '{jugador}'
    MATCH (u)-[:juega_para]->(t)
    return t
    '''

    return session.run(instruccion).data()[0]['t']['nombre_equipo']
```

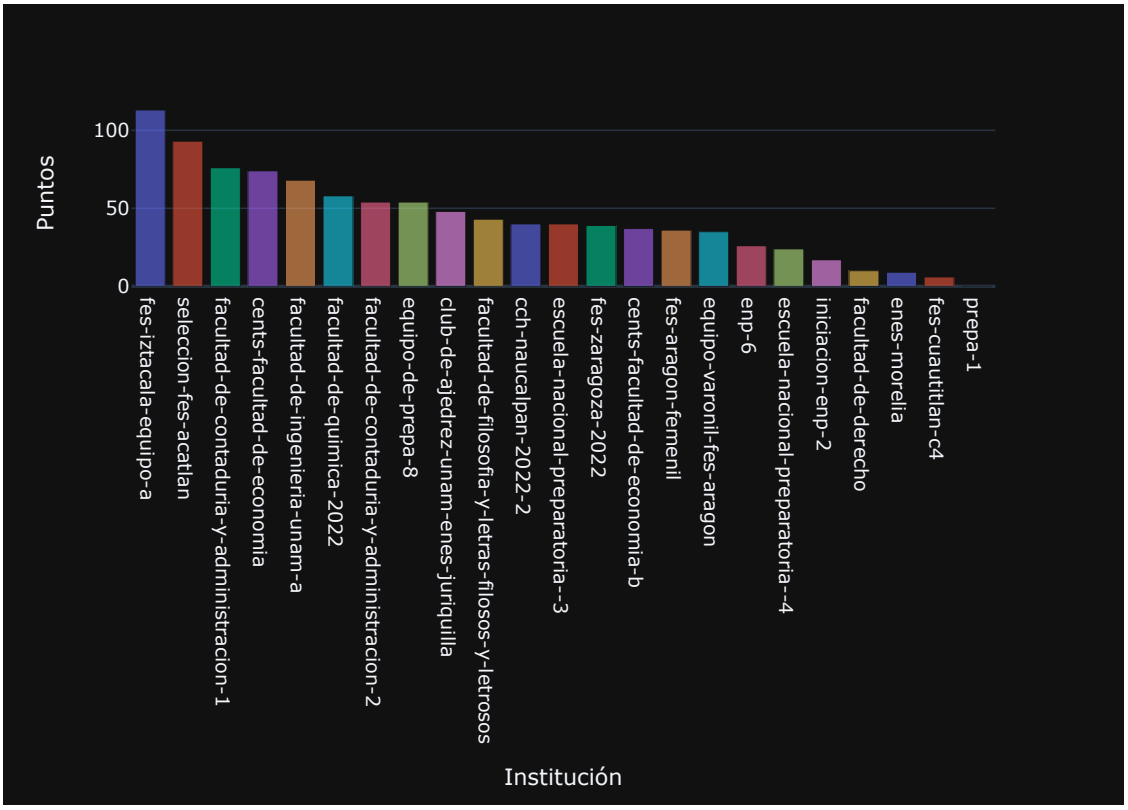

Figura 13: Ranking de equipos con más victorias.

1.5.3. Jugadores que superaron expectativas

Se hace la cuenta sobre las aristas salientes que tienen un *valor de resultado* mayor a 1. Esto es, vencer usuarios más fuertes.

La función auxiliar para la extracción es:

```python
def victorias_sobresalientes(jugador):
    """
    Cuenta el número de partidas ganadas contra jugadores con
    un rating mayor.
    """

    instruccion = f'''
    MATCH (u1:Usuario)
    MATCH (u2:Usuario)
    WHERE u1.id = '{jugador}'
    MATCH (u1)-[g:gano_a]->(u2)
    RETURN g.valor_resultado
    '''

    count = 0

    for resultado in session.run(instruccion).data():
        if float(resultado['g.valor_resultado']) > 1:
            count += 1

    return count
```

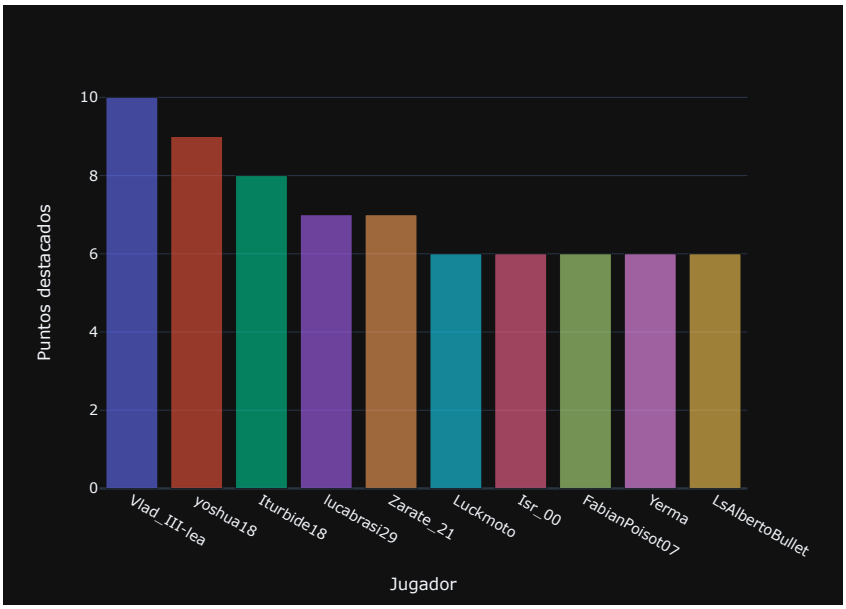

Figura 14: Número de partidas en las que el jugador venció jugadores más fuertes.

Es interesante pensar que estos jugadores tengan un gran campo de mejora pues tuvieron un performance superior al esperado. Si la UNAM estuviera en busca de jugadores promesa que pasan desapercibidos por su puntuación, estos serían serios candidatos pues tuvieron la oportunidad de enfrentarse a jugadores más fuertes y probar que tienen la capacidad de representar en torneos más formales.

Un dilema que puede surgir de partidas en línea es cuando un jugador muy débil es capaz de derrotar a jugadores mucho más superiores a él. Podría pensarse que dichos jugadores están haciendo alguna clase de trampa pues, estadísticamente, salen del comportamiento habitual. Una forma de corroborar esto es analizando si el jugador tiene un número de ingrados nulo (es decir, jamás fueron vencidos durante el torneo), lo cuál es algo improbable ya que hasta los mejores jugadores tienen alta probabilidad de perder.

1.5.4. Jugadores con más derrotas

El número de ingrados del nodo de un jugador permite identificar de una forma sencilla las veces que fue derrotado este. Los jugadores más derrotados puede hallarse mediante la siguiente función auxiliar.

```
def derrotas_jugador(jugador):
    """
    Número de derrotas por jugador.
    """

    instruccion = f'''
    MATCH (u1:Usuario)
    MATCH (u2:Usuario)
    WHERE u1.id = '{jugador}'
    MATCH (u2)-[g:gano_a]->(u1)
    RETURN count(*) as ingrados
    '''

    return session.run(instruccion).data()[0]['ingrados']
```

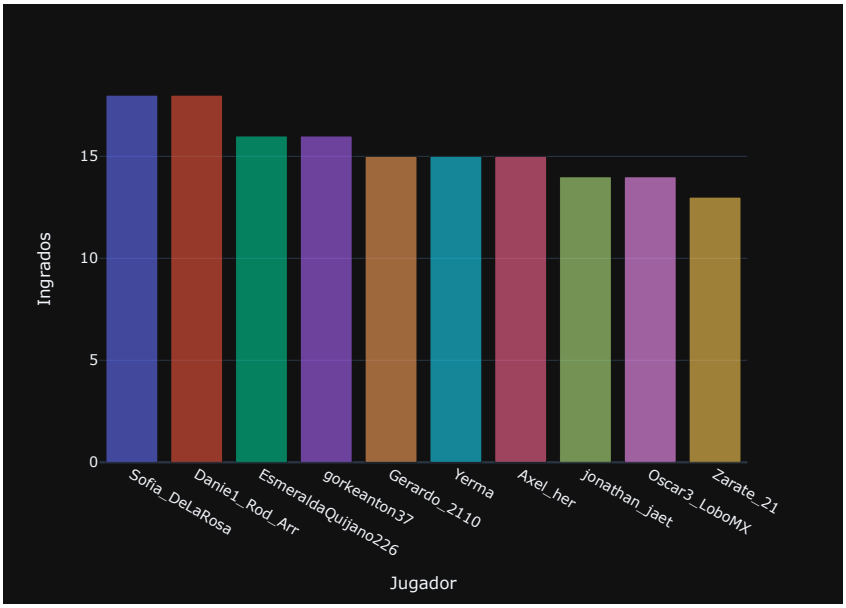

Figura 15: Jugadores con más derrotas del torneo, es decir, jugadores que tienen el mayor número de ingrados.

En general, los usuarios anteriores representaron una mayor pérdida de puntos que un beneficio real a sus equipos pues, la mayoría, no fueron capaces de ganar partidas. Se podría sugerir a los equipos de dichos jugadores que tomen otra estrategia de selección de jugadores para su representación.

1.5.5. Pérdidas contra jugadores inferiores

Tomando ventaja de los atributos de relación, podemos identificar qué tipos de derrotas experimentaron cada uno de los jugadores. La siguiente función auxiliar se encarga de realizar la cuenta de las derrotas hechas por jugadores inferiores a ellos.

```python
def peores_derrotas(jugador):
    """
    Cuenta el número de derrotas contra jugadores inferiores
    en el momento de la partida.
    """

    instruccion = f'''
    MATCH (u1:Usuario)
    MATCH (u2:Usuario)
    WHERE u1.id = '{jugador}'
    MATCH (u2)-[g:gano_a]->(u1)
    RETURN g.valor_resultado
    '''

    count = 0

    for resultado in session.run(instruccion).data():
        if float(resultado['g.valor_resultado']) > 1:
            count += 1

    return count
```

De acuerdo a la figura 16, los mejores jugadores (de mayor *rating*) fueron los que experimentaron más este efecto pues casi cualquier jugador del torneo es inferior a ellos. Si bien la probabilidad de que un jugador fuerte pierda contra uno más débil es baja, es importante destacar el comportamiento de juego que existe pues factores psicológicos están involucrados en esta clase de deporte. Si los jugadores más fuertes experimentan esta clase de errores, la probabilidad de perder en torneos de mayor importancia es superior.

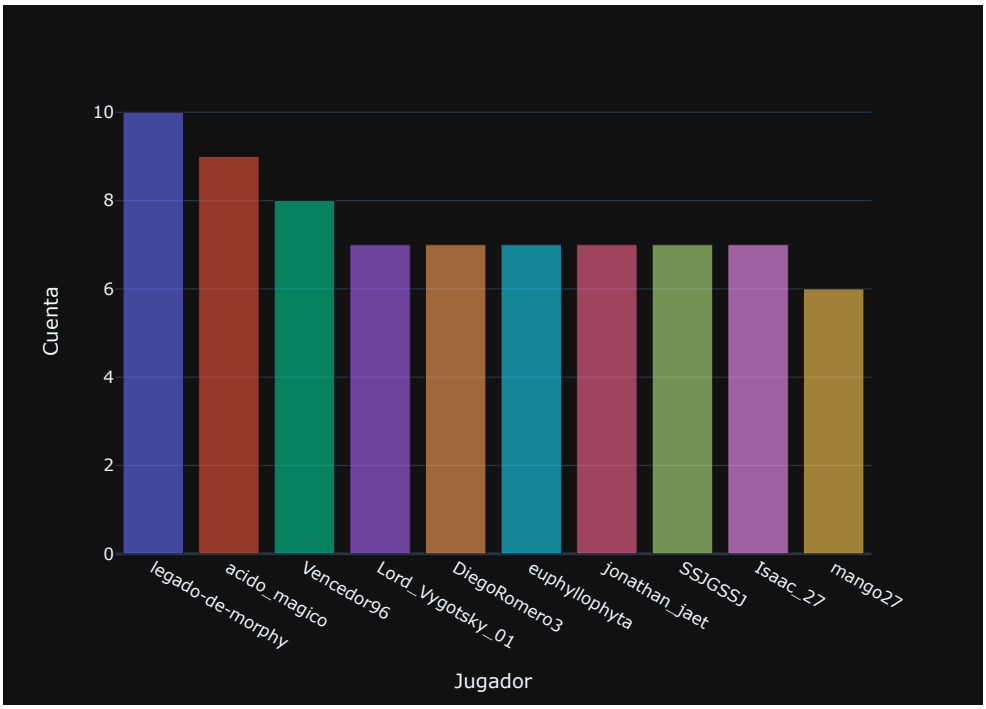

Figura 16: Jugadores que tuvieron mayores derrotas de jugadores inferiores a ellos.

2. Conclusiones

La sinergia que se puede conseguir al trabajar con distintos tipos de bases de datos se convierte en una herramienta muy poderosa mientras se implemente de la forma correcta y para los objetivos específicos que estas fueron creadas. La versatilidad de ofrece Mongo para el manejo de bases de datos es el compañero perfecto de Neo4j pues, mientras Neo4j se encarga de conseguir a los nodos de interés particular, Mongo nos devuelve la información relacionada a los nodos encontrados.

La posibilidad de evitar hacer operaciones costosas de las bases de datos relaciones constituye una ventaja sustancial considerando esquemas de comunicación entre usuarios. Los sistemas en los cuales se implique el manejo de usuarios requiere esquemas escalables que se pueden conseguir mediante MongoDB y Neo4j.