.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29 - 95.06) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

24 de noviembre de 2023

Alexis Herrera
104639

Edgardo Francisco Saez
104896

Franco Gazzola
105103

1. Introducción

El objetivo de este trabajo es ayudar a Scaloni a definir la cantidad mínima de jugadores que deberán jugar para satisfacer a los medios. Esto se puede modelar como un problema llamado Hitting Set Problem.

1.1. Análisis del problema

Hitting Set Problem:

Dado un conjunto A de n elementos m subconjuntos $B_1, B_2, ..., B_m$ de A ($B_i \subseteq A \forall i$) y un número k, ¿existe un subconjunto $C \subseteq A$ con $|C| \leqslant k$ tal que C tenga al menos un elemento de cada B_i?

2. Demostrar que el Hitting-Set problem se encuentra en NP

Para demostrar que un problema $X \in NP$ para cualquier solución de X la misma debe poder verificarse en tiempo polinomial.

En este caso X es una solución de Hitting-Set y la misma es de tamaño a lo sumo k. Para validarla debemos comparar el conjunto de tamaño a lo sumo k con todos los subconjuntos B_i y validar que haya al menos un elemento de k en todos los B_i. Esta validación puede hacerse en tiempo polinomial porqué a lo sumo tendremos que comparar cada elemento de nuestro conjunto solución con todos los elementos de los subcojuntos B_i, por lo tanto podemos decir que Hitting-Set se encuentra en NP.

3. Demostrar que el Hitting-Set es un problema NP-Completo

La demostración de que un problema X es NP-Completo requiere la realización de una reducción polinomial desde un problema NP-Completo conocido Y. Si logramos establecer que $Y <_p X$, entonces concluimos que X es NP-Completo.

Tomaremos el problema Vertex Cover ya conocido como NP-Completo. Nuestro objetivo es realizar una reducción polinomial desde Vertex Cover a Hitting-Set

Para lograr esto, tomaremos una instancia arbitraria de Vertex-Cover y la adaptaremos para que pueda ser resuelta por una caja negra que resuelva instancias del problema HittingSet. Si logramos resolver Vertex-Cover haciendo uso de esta caja podremos decir que Hitting Set es al menos tan difícil como Vertex Cover.

Vertex Cover:

El vertex cover de un grafo no dirigido es un subset de sus vértices tal que todas las aristas (u, v) del grafo, sea u o v están en el vertex cover. El set cubre todas las aristas del grafo dado. Dado un grafo no dirigido el problema de vertex cover es encontrar un vertex cover de tamaño mínimo.

Vertex Cover es un problema NP-Completo.

Reducción:

El problema de vertex cover es encontrar un conjunto de vértices que sea capaz de cubrir a otro conjunto en este caso de aristas. Debemos encontrar de que manera podemos adaptar esto para que pueda ser resuelto por Hitting-Set.

En Hitting-Set queremos encontrar un subconjunto de como mucho tamaño k cuyos elementos aparezcan por lo menos una vez en los subconjuntos de entrada. Además tenemos la condición de que los elementos de ambos subconjuntos pertenecen a un conjunto común A.

En Vertex-Cover sabemos que todas las aristas están relacionadas con al menos dos vértices u y v. Lo que buscamos es que al menos uno de estos vértices este en el conjunto solución de nuestro vertex cover H. De la misma manera en Hitting set queremos que dentro de cada uno de los subconjuntos B_i haya al menos un elemento que pertenezca a nuestro conjunto solución Z.

Entonces ahora podríamos establecer la siguiente relación: nuestras aristas del problema de vertex-cover (u_i, v_i) son los subconjuntos B_i y nuestro vertex cover H es el conjunto solución Z.

Ahora entonces lo que haremos es tomar todas las aristas de nuestro grafo G y construir los subconjuntos $S = \{(u_1, v_1), ..., (u_i, v_i)\}$ para entregarlos a nuestra caja negra junto con el número k para que la misma determine si existe una solución de Hitting-Set para esa instancia. Si la respuesta es positiva es porque existe un vertex-cover de al menos tamaño k, si es negativa es porque no lo hay.

Ahora podemos decir que Vertex-Cover $<_p$ Hitting-set lo que quiere decir que Hitting-set es al menos tan dificil como Vertex-Cover y por lo tanto es NP-Completo.

4. Algoritmo por BackTracking

En esta sección exploraremos la solución óptima al problema de **Hitting Set** utilizando **BackTracking**.

4.1. Código

A continuación se muestra el código de la solución al problema.

```python
def _hitting_set(universo, subconjuntos, indice_elemento, asignacion_actual, mejor_asignacion):

    # Poda, si la asignacion actual es peor que la mejor asignacion, devuelvo mejor asignacion
    if mejor_asignacion and len(asignacion_actual) > len(mejor_asignacion):
        return mejor_asignacion

    # Devuelvo mejor solucion hasta el momento si es que la asignacion actual es solucion
    if es_solucion(subconjuntos, asignacion_actual):
        if not mejor_asignacion or len(asignacion_actual) < len(mejor_asignacion):
            return asignacion_actual
        else:
            return mejor_asignacion

    if indice_elemento >= len(universo):
        return mejor_asignacion

    # Llamada recursiva con asignacion_actual actualizada
    resultado = _hitting_set(universo, subconjuntos, indice_elemento + 1, asignacion_actual + [universo[indice_elemento]], mejor_asignacion)

    # Actualizacion de mejor_asignacion
    if not mejor_asignacion or len(resultado) < len(mejor_asignacion):
        mejor_asignacion = resultado

    # Llamada recursiva sin modificar asignacion_actual
    resultado = _hitting_set(universo, subconjuntos, indice_elemento + 1, asignacion_actual, mejor_asignacion)

    # Actualizacion de mejor_asignacion
    if resultado and (not mejor_asignacion or len(resultado) < len(mejor_asignacion)):
        mejor_asignacion = resultado

    return mejor_asignacion
```

```python
def es_solucion(subconjuntos, asignacion_actual):
    for subconjunto in subconjuntos:
        contiene_elemento = False
        for elemento in subconjunto:
            if elemento in asignacion_actual:
                contiene_elemento = True
                break
        if not contiene_elemento:
            return False
    return True
```

4.2. Tests

En esta sección mostraremos los diferentes pruebas que realizamos a lo largo del trabajo en la resolución del problema utilizando BackTracking.

4.2.1. Tests cátedra

En primer lugar se mostrara las pruebas con los textos proporcionados por la cátedra

```
def test_ejemplo_5():
    universo, subconjunto = parsear_archivo('TP3/5.txt')
    assert hitting_set(universo, subconjunto) == ["Barcon't", "Messi"]


fanusaez
def test_ejemplo_7():
    universo, subconjunto = parsear_archivo('TP3/7.txt')
    assert hitting_set(universo, subconjunto) == ["Mauro Zarate", "Pezzella"]


fanusaez
def test_ejemplo_10_pocos():
    universo, subconjunto = parsear_archivo('TP3/10_pocos.txt')
    assert hitting_set(universo, subconjunto) == ["Chiquito Romero", "Di Maria", "Casco"]


fanusaez *
def test_ejemplo_10_todos():
    universo, subconjunto = parsear_archivo('TP3/10_todos.txt')
    assert hitting_set(universo, subconjunto) == ['Dibu', 'Cuti', 'Molina', 'Guido Rodriguez', 'Paredes',
                                                  'Palacios', 'Messi', 'Garnacho', 'Lautaro', 'Perrone']


fanusaez *
def test_ejemplo_10_varios():
    universo, subconjunto = parsear_archivo('TP3/10_varios.txt')
    assert hitting_set(universo, subconjunto) == ['Palermo', 'Dibu', 'Beltran', 'Riquelme', 'Dybala',
                                                  'Di Maria']
```

```python
def test_ejemplo_15():
    universo, subconjunto = parsear_archivo('TP3/15.txt')
    assert hitting_set(universo, subconjunto) == ['Chiquito Romero', 'Palermo', 'Luka Romero', 'Dybala']


fanusaez
def test_ejemplo_20():
    universo, subconjunto = parsear_archivo('TP3/20.txt')
    assert hitting_set(universo, subconjunto) == ["Barcon't", 'Riquelme', 'El fantasma de la B',
                                                  'Mauro Zarate', 'Ogro Fabianni']


fanusaez *
def test_ejemplo_50():
    universo, subconjunto = parsear_archivo('TP3/50.txt')
    assert hitting_set(universo, subconjunto) == ["Casco", "Barcon't", "Tucu Pereyra",
                                                  "Dybala", "Armani", "Langoni"]


fanusaez *
def test_ejemplo_75():
    universo, subconjunto = parsear_archivo('TP3/75.txt')
    assert hitting_set(universo, subconjunto) == ['Simeone', 'Riquelme', 'Casco', 'Palermo', 'Chiquito Romero',
                                                  'Ogro Fabianni', 'Cuti Romero', 'Beltran']


fanusaez *
def test_ejemplo_100():
    universo, subconjunto = parsear_archivo('TP3/100.txt')
    assert hitting_set(universo, subconjunto) == ["Barcon't", 'Armani', 'Gallardo', 'Langoni', 'El fantasma de la B',
                                                  'Soule', 'Wachoffisde Abila', 'Messi', 'Changuito Zeballos']
```

```python
def test_ejemplo_200():
    universo, subconjunto = parsear_archivo('TP3/200.txt')
    assert hitting_set(universo, subconjunto) == ['Mauro Zarate', 'Tucu Pereyra', 'Beltran', 'Gallardo', 'Pity Martinez',
                                                  "Barcon't", 'Palermo', 'Soule', 'Chiquito Romero']
```

También se muestran los ejemplos mencionados anteriormente, verificando que el programa devuelve lo esperado. En esta ocasión utilizamos la librería PyTest para ejecutar los casos de prueba. Para ejecutarlos ingresamos el siguiente comando: `pytest test_catedra_bt.py -v`.

```
(venv) fanu@fanu:~/Desktop/TDA-G04/TP3$ pytest test_catedra_bt.py -v
============================================================================ test session starts ============
platform linux -- Python 3.11.5, pytest-7.4.2, pluggy-1.3.0 -- /home/fanu/Desktop/TDA-G04/venv/bin/python
cachedir: .pytest_cache
rootdir: /home/fanu/Desktop/TDA-G04/TP3
plugins: anyio-4.0.0
collected 11 items

test_catedra_bt.py::test_ejemplo_5 PASSED
test_catedra_bt.py::test_ejemplo_7 PASSED
test_catedra_bt.py::test_ejemplo_10_pocos PASSED
test_catedra_bt.py::test_ejemplo_10_todos PASSED
test_catedra_bt.py::test_ejemplo_10_varios PASSED
test_catedra_bt.py::test_ejemplo_15 PASSED
test_catedra_bt.py::test_ejemplo_20 PASSED
test_catedra_bt.py::test_ejemplo_50 PASSED
test_catedra_bt.py::test_ejemplo_75 PASSED
test_catedra_bt.py::test_ejemplo_100 PASSED
test_catedra_bt.py::test_ejemplo_200 PASSED

============================================================================ 11 passed in 2982.06s (0:49:42) =====
```

4.2.2. Tests Propios

En esta siguiente sección presentaremos las pruebas con los textos generados a mano.

```python
def test_ejemplo_5_unico():
    universo, subconjunto = parsear_archivo('sets_generados/5_unico.txt')
    assert hitting_set(universo, subconjunto) == ["Messi"]


fanusaez
def test_ejemplo_5_todos():
    universo, subconjunto = parsear_archivo('sets_generados/5_todos.txt')
    assert hitting_set(universo, subconjunto) == ["Barcon't", "Colo Barco", "Wachoffisde Abila", "Messi", "Cuti Romero"]


fanusaez
def test_ejemplo_10_unico():
    universo, subconjunto = parsear_archivo('sets_generados/10_unico.txt')
    assert hitting_set(universo, subconjunto) == ["Dibu"]


fanusaez
def test_ejemplo_10_pocos():
    universo, subconjunto = parsear_archivo('sets_generados/10_pocos.txt')
    assert hitting_set(universo, subconjunto) == ["Messi", "Di Maria"]
```

```python
def test_ejemplo_10_varios():
    universo, subconjunto = parsear_archivo('sets_generados/10_varios.txt')
    assert hitting_set(universo, subconjunto) == ['Di Maria', 'Colo Barco', 'Tagliafico',
                                                  'Martinez', 'Alvarez', 'Perez']


fanusaez
def test_ejemplo_15():
    universo, subconjunto = parsear_archivo('sets_generados/15.txt')
    assert hitting_set(universo, subconjunto) == ["Alvarez", "Fernandez",  "Dibu"]


new *
def test_ejemplo_20():
    universo, subconjunto = parsear_archivo('sets_generados/20.txt')
    assert hitting_set(universo, subconjunto) == ["Alvarez", "Fernandez", "Dibu", "Palacios"]
```

También se muestran los ejemplos mencionados anteriormente, verificando que el programa devuelve lo esperado.

Para ejecutarlos ingresamos el siguiente comando: `pytest test_propios_bt.py -v`.

```
============================================================================ test session starts ============
platform linux -- Python 3.11.5, pytest-7.4.2, pluggy-1.3.0 -- /home/fanu/Desktop/TDA-G04/venv/bin/python
cachedir: .pytest_cache
rootdir: /home/fanu/Desktop/TDA-G04/TP3
plugins: anyio-4.0.0
collected 7 items

test_propios_bt.py::test_ejemplo_5_unico PASSED
test_propios_bt.py::test_ejemplo_5_todos PASSED
test_propios_bt.py::test_ejemplo_10_unico PASSED
test_propios_bt.py::test_ejemplo_10_pocos PASSED
test_propios_bt.py::test_ejemplo_10_varios PASSED
test_propios_bt.py::test_ejemplo_15 PASSED
test_propios_bt.py::test_ejemplo_20 PASSED

============================================================================ 7 passed in 8.56s ============
```

4.3. Mediciones de Tiempo

Para evaluar la eficiencia y rendimiento de nuestro algoritmo, implementamos un sistema de medición del tiempo utilizando el módulo *time* de Python. Esta herramienta nos permite cuantificar el tiempo que tarda nuestro código en ejecutarse.

```
universo, subconjunto = parsear_archivo('TP3/5.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
```
Executed at 2023.11.24 15:57:12 in 42ms

```
Ejecutado en: 0.0008301734924316406 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/7.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
```
Executed at 2023.11.24 15:57:12 in 41ms

```
Ejecutado en: 0.001489400863647461 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/10_pocos.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
```
Executed at 2023.11.24 15:57:12 in 41ms

```
Ejecutado en: 0.018796920776367188 segundos
```

Figura 1: Mediciones de tiempo 1

En esta imagen podemos observar una ejecución de los algoritmos en tiempos menores a un segundo. Esto se debe al poco volumen de los textos y una solución óptima de pocos elementos.

```
universo, subconjunto = parsear_archivo('TP3/10_todos.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 12:58:57 in 19m 19s 968ms
```

```
Ejecutado en: 1159.9635400772095 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/10_varios.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 12:59:01 in 4s 797ms
```

```
Ejecutado en: 4.848559379577637 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/15.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 12:59:02 in 505ms
```

```
Ejecutado en: 0.49121546745300293 segundos
```

Figura 2: Mediciones de tiempo 2

En esta figura podemos ver tiempos de ejecución de pocos segundos en los dos últimos casos, excepto por el primero cuyo tiempo de ejecución es de aproximadamente 1200 segundos.

Este conjunto presenta un escenario donde la estrategia de poda, implementada en nuestro algoritmo de `hitting set` por backtracking, no logra generar mejoras sustanciales en el tiempo de ejecución.

La técnica de poda se introduce para optimizar el proceso de búsqueda, permitiendo la interrupción temprana de ramas del árbol de decisiones que no conducirán a una solución óptima. En situaciones típicas, donde no existe la posibilidad de encontrar una solución más eficiente con menos elementos, la poda juega un papel crucial en reducir el tiempo de ejecución al evitar exploraciones innecesarias.

Sin embargo, en el caso concreto de '10_todos', la estrategia de poda se revela como ineficaz. En este contexto particular, no podemos descartar ramas de la solución durante la exploración, ya que la solución óptima requiere seleccionar una cantidad de jugadores equivalente a la cantidad de subconjuntos. Es decir, no podremos descartar soluciones en donde la longitud de la asignación que estamos explorando sea mayor a la mejor asignación encontrada hasta el momento.

```
universo, subconjunto = parsear_archivo('TP3/20.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 12:59:04 in 1s 785ms

  Ejecutado en: 1.8211183547973633 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/50.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 12:59:17 in 13s 200ms

  Ejecutado en: 13.143906116485596 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/75.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 13:01:12 in 1m 54s 722ms

  Ejecutado en: 114.71563863754272 segundos
```

Figura 3: Mediciones de tiempo 3

En estos 3 casos podemos observar un salto exponencial entre el tiempo tomado para el texto con 50 subconjuntos y el de 75 subconjuntos.

```
universo, subconjunto = parsear_archivo('TP3/100.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 13:07:05 in 5m 53s 481ms

Ejecutado en: 353.5146062374115 segundos
```

```
universo, subconjunto = parsear_archivo('TP3/200.txt')
inicio_tiempo = time.time()
hitting_set(universo, subconjunto)
fin_tiempo = time.time()
print(f"Ejecutado en: {fin_tiempo - inicio_tiempo} segundos")
Executed at 2023.11.15 13:29:54 in 22m 48s 347ms

Ejecutado en: 1368.4363543987274 segundos
```

Figura 4: Mediciones de tiempo 4

Por ultimo podemos apreciar con los dos archivos mas extensos el aumento de tiempo en base a la cantidad de elementos. Llegando a una medición de 1368 segundos con el texto de 200 subconjuntos.

5. Programación Entera

5.1. Reducción

A continuación mostraremos como el problema de Hitting-Set se puede reducir polinomialmente al problema de la programación lineal entera.

Tenemos un set $A = \{a_1, ..., a_n\}$ y una colección $B_1, .., B_m$ de subsets de A. Además cada elemento de $a_i \in A$ tiene un peso $w_i \geq 0$.

Cada variable de nuestro problema representará un elemento de un subconjunto B_i. Las ecuaciones representan los subconjuntos B_i. Se necesita que al menos uno de los elementos de cada B_i se utilice en la solución. Las desigualdades se conformaran como $x_1 + x_2 + ... + x_n \geq 1$ con los $x_i \in B_i$.

Entonces nuestro objetivo será minimizar las siguiente función:

$$\sum_{i \in B} w_i x_i, \ x \in \{0, 1\}$$

Donde w_i es el costo del elemento a utilizar y x_i es el factor que indica si el elemento se utiliza o no tomando los valores 1 o 0.

Ahora podemos decir que Hitting set $<_p$ Programación entera. Entonces si además podemos demostrar que programación entera se encuentra en NP, se podría decir que pertenece a NP-Completo.

Archivos	BackTracking	PLE	Cantidad de elementos
5.txt	0s	0.024s	2
7.txt	0.001s	0.019s	2
10_pocos.txt	0.012s	0.014s	3
10_todos.txt	1374s	0.026s	10
10_varios.txt	5.793s	0.016s	6
15.txt	0.593s	0.029s	4
20.txt	2.273s	0.025s	5
50.txt	16.23s	0.032s	6
75.txt	138.3s	0.246s	8
100.txt	406.0s	0.453s	9
200.txt	1491s	0.874s	9

5.2. Código

```python
from pulp import LpProblem, LpMinimize, LpVariable, LpStatus, lpSum

def hitting_set(universo, subconjuntos):
    # Define el problema de optimizaci n
    prob = LpProblem("HittingSet", LpMinimize)

    # Crea variables binarias para cada elemento en el conjunto universo
    x_vars = LpVariable.dicts("x", universo, cat='Binary')
    print(x_vars)

    # Funcion objetivo: suma de todas las x_i
    prob += lpSum(x_vars[i] for i in universo)

    # Restricciones: cada subconjunto debe tener al menos un elemento seleccionado
    for subset in subconjuntos:
        prob += lpSum(x_vars[i] for i in subset if i in universo) >= 1

    # Resolver el problema
    prob.solve()

    # Revisar si la soluci n es  ptima  y devolver el resultado
    if LpStatus[prob.status] == 'Optimal':
        # Devuelve los elementos seleccionados para el hitting set
        return [i for i in universo if x_vars[i].value() == 1]
    return None
```

5.3. Mediciones de tiempo y comparación con backtracking

Ambos algoritmos han demostrado ser capaces de encontrar la solución óptima en todos los casos evaluados. Sin embargo, se observa una diferencia significativa en los tiempos de ejecución a medida que el tamaño de los conjuntos de datos aumenta.

Para conjuntos de datos pequeños (por ejemplo, 5 y 7 elementos), la diferencia en el tiempo de ejecución entre ambos algoritmos es mínima y apenas perceptible.

A medida que el tamaño de los conjuntos de datos aumenta (por ejemplo, 10, 15, 20 y mas elementos), la diferencia en el tiempo de ejecución se vuelve más evidente, mostrando una clara tendencia de aumento en la disparidad de rendimiento.

5.4. Tests

También comprobamos que devuelve la solución esperada con los ejemplos de la cátedra y con ejemplos creados por nosotros, se pueden probar de la siguiente forma

```
pytest -v test_programacion.py
```

```
test_programacion.py::test_ejemplo_5 PASSED
test_programacion.py::test_ejemplo_7 PASSED
test_programacion.py::test_ejemplo_10_pocos PASSED
test_programacion.py::test_ejemplo_10_todos PASSED
test_programacion.py::test_ejemplo_10_varios PASSED
test_programacion.py::test_ejemplo_15 PASSED
test_programacion.py::test_ejemplo_20 PASSED
test_programacion.py::test_ejemplo_50 PASSED
test_programacion.py::test_ejemplo_75 PASSED
test_programacion.py::test_ejemplo_100 PASSED
test_programacion.py::test_ejemplo_200 PASSED
test_programacion.py::test_ejemplo_5_unico PASSED
test_programacion.py::test_ejemplo_5_todos PASSED
test_programacion.py::test_ejemplo_10_unico PASSED
```

Figura 5: Test de programación lineal entera con test de cátedra y propio

5.5. Aproximación

Ahora analicemos qué ocurre al relajar las restricciones sobre x, permitiéndole tomar valores dentro de un intervalo continuo. La complejidad de estos problemas con variable continua son resueltos por el método simplex con complejidad $O(n^3)$ pero en general se resuelve en $O(n)$.

Entonces el modelo, con las variables relajadas serían de la siguiente forma:

$$\text{mín} \sum_{i=1}^{n} w x_i$$

$$0 \leq x_i \leq 1, \quad i = 1, \ldots, n$$

$$\sum_{i:a_i \in B_j} x_i \geq 1, \quad j = 1, \ldots, m$$

Sea S el conjunto de variables tales que $x_i >= 1/b$. Entonces vemos que S es un hitting set. Esto porque como el modelo esta restringido a que la sumatoria de x_i para cada subconjunto sea mayor o igual a 1, y que cada subconjunto sea a lo mucho de largo b entonces esto implica algún x_i de cada subconjunto pertenece a S.

Por otra parte, vamos a acotar los pesos de la solución S, es decir: $w(S) = \sum_{a_i \in S} w_i$

$$w(S) = \sum_{a_i \in S} w_i \leq \sum_{a_i \in S} w_i b x_i \leq b \sum_{i=1}^{n} w_i x_i = b w_{LP}$$

Ver que primera desigualdad, resulta de que $x_i \geq 1/b$ que es lo mismo que $b \cdot x_i \geq 1$ y también que definimos $w_{LP} = \sum_{i=1}^{n} w_i x_i$.

Ahora si definimos a S^* como el hitting set óptimo del problema, y sea $x_i^* = 1$ si a_i pertenece a S^*, 0 si no. Entonces se cumple que:

$$w_{LP} \leq \sum_{i=1}^{n} w_i x_i^* = \sum_{a_i \in S^*} w_i = w(S^*)$$

Dado que el peso total de la solución óptima del problema relajado (el cual puede incluir fracciones de elementos) siempre será menor o igual al peso total de la solución óptima del problema original (que sólo incluye elementos enteros)

Entonces como $w(S) \leq bw_{LP} \leq bw(S^*)$:

$$w(S) \leq bw(S^*)$$

Ajustando a las definiciones del enunciado, $A(I) = w(S)$ y $z(I) = w(S^*)$ y por lo tanto:

$$\frac{A(I)}{z(I)} \leq r(A) = b$$

Realizamos las siguientes mediciones, verificando que se cumple la cota teórica dada con el siguiente comando: `pytest -rP test_cota_pl_aproximado.py`

```
test_cota_pl_aproximado.py .........                                    [1

==================================== PASSES ====================================
_____________________________ test_5_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 2, z(i): 2, ratio: 1.0, b:6
_____________________________ test_7_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 2, z(i): 2, ratio: 1.0, b:6
_____________________________ test_15_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 11, z(i): 4, ratio: 2.75, b:7
_____________________________ test_20_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 12, z(i): 5, ratio: 2.4, b:7
_____________________________ test_50_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 13, z(i): 6, ratio: 2.1666666666666665, b:8
_____________________________ test_75_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 20, z(i): 8, ratio: 2.5, b:8
_____________________________ test_100_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 23, z(i): 9, ratio: 2.5555555555555554, b:8
_____________________________ test_200_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 27, z(i): 9, ratio: 3.0, b:10
_____________________________ test_300_cota_pl_aproximado _____________________________
----------------------------- Captured stdout call -----------------------------
A(I): 62, z(i): 20, ratio: 3.1, b:10
============================== 9 passed in 16.83s ==============================
```

Figura 6: Mediciones exacto vs aproximación

Finalmente buscamos un volumen que resulte inmanejable para el algoritmo de programación lineal entera y verificamos la cota calculándola nuevamente con programación lineal continua.

```
/home/fanu/Desktop/TDA-G04/venv/bin/python /home/fanu/Desktop/TDA-G04/TP3/programacion_lineal_volumen.py
Archivo: 300.txt
20.61753s
20

Archivo: 400.txt
256.01144s
23

Archivo: 500.txt
3359.18604s
25
```

Figura 7: Tiempos y soluciones Programación Lineal Entera

```
/home/fanu/Desktop/TDA-G04/venv/bin/python /home/fanu/Desktop/TDA-G04/TP3/benchmark_pl_volumen.py
Archivo: 300.txt
0.02552s
62

Archivo: 400.txt
0.03217s
71

Archivo: 500.txt
0.03869s
75
```

Figura 8: Tiempos y soluciones Programación Lineal Continua

Observamos que a pesar de no haber encontrado la solución óptima, la cota antes calculada sigue verificando y el tiempo tardado es mucho menor.

6. Aproximación Greedy

El algoritmo "hitting_set_greedy"se ha implementado con el objetivo de encontrar un conjunto de jugadores denominado "hitting set"de manera voraz (greedy) a partir de un conjunto dado de subconjuntos que representan diferentes combinaciones de jugadores.

6.1. Código

```
def hitting_set_greedy(subconjuntos):
    # Lista para la asignacion final
    asignacion = []

    while not es_solucion(subconjuntos, asignacion):
        apariciones = contar_apariciones(subconjuntos, {})

        # Ordena el diccionario de apariciones de forma descendente
        apariciones_ordenado = dict(sorted(apariciones.items(), key=lambda item:
    item[1], reverse=True))

        # Agrega el jugador m s frecuente a la asignaci n
        asignacion.append(next(iter(apariciones_ordenado)))
```

```
13
14          if es_solucion(subconjuntos, asignacion):
15              return asignacion
16
17          # Elimina los subconjuntos que contienen al  ltimo  jugador agregado
18          subconjuntos = eliminar_subconjuntos_con_jugador(subconjuntos, asignacion
    [-1])
19
20      return asignacion
```

```
1  def es_solucion(subconjuntos, asignacion_actual):
2      for subconjunto in subconjuntos:
3          contiene_elemento = False
4          for elemento in subconjunto:
5              if elemento in asignacion_actual:
6                  contiene_elemento = True
7                  break
8          if not contiene_elemento:
9              return False
10     return True
```

```
1  def contar_apariciones(subconjuntos, apariciones):
2      for subconjunto in subconjuntos:
3          for jugador in subconjunto:
4              if jugador not in apariciones:
5                  apariciones[jugador] = 1
6              else:
7                  apariciones[jugador] += 1
8      return apariciones
```

```
1  def eliminar_subconjuntos_con_jugador(subconjuntos, jugador):
2      nuevo_subconjunto = []
3
4      for subconjunto in subconjuntos:
5          if jugador not in subconjunto:
6              nuevo_subconjunto.append(subconjunto)
7
8      return nuevo_subconjunto
```

6.2. Resultados y Comparación con PLC

	Tiempo		Elementos	
Archivos	PLC	Greedy	PLC	Greedy
5.txt	0.012s	0s	2	2
7.txt	0.013s	0s	2	2
10_pocos.txt	0.013s	0s	6	3
10_todos.txt	0.022s	0s	10	10
10_varios.txt	0.017s	0s	6	7
15.txt	0.014s	0s	11	5
20.txt	0.018s	0s	12	5
50.txt	0.021s	0s	13	7
75.txt	0.019s	0.001s	20	9
100.txt	0.022s	0s	23	11
200.txt	0.050s	0.002s	27	10

Figura 9: Tiempos y soluciones Programación Lineal Continua

Podemos notar que, aunque los tiempos de la Programación Lineal continua (PLC) se mantienen en niveles bajos, la ejecución del algoritmo propuesto exhibe tiempos significativamente menores. Además, a lo largo de múltiples ejecuciones, la implementación mediante el enfoque greedy logra conjuntos con menos elementos en comparación con PLC.

Considerando estos resultados, podemos concluir que la aproximación mediante el enfoque greedy es superior en términos de eficiencia y rendimiento en comparación con la Programación Lineal Continua.

6.3. Complejidad Temporal

En esta sección explicaremos la complejidad temporal del algoritmo propuesto (Greedy)

6.3.1. Es Solución

La función **es_solución** verifica si la asignación actual constituye una solución válida revisando cada subconjunto. La complejidad de esta función es $\mathcal{O}(m * n)$, donde m es la cantidad de subconjuntos y n es el tamaño promedio de cada subconjunto. En el peor caso, se deben revisar todos los subconjuntos para determinar si la asignación actual es una solución.

6.3.2. Contar Apariciones

La función **contar_apariciones** itera sobre cada subconjunto y jugador, actualizando un diccionario de apariciones. La complejidad de esta función es $\mathcal{O}(m * n)$.

6.3.3. Eliminar Subconjuntos con Jugador

La función **eliminar_subconjuntos_con_jugador** recorre cada subconjunto y elimina los subconjuntos que contengan el jugador recibido. La complejidad es $\mathcal{O}(m * n)$, similar a la función

anterior. Dado que debemos iterar sobre todos los subconjuntos hasta encontrar el jugador.

6.3.4. Hitting Set Greedy

La función principal **hitting_set_greedy** realiza un bucle hasta que se alcanza una solución. En cada iteración también se calcula la frecuencia de cada jugador y se elimina los subconjuntos que poseen el jugador agregado. La cantidad de iteraciones hasta encontrar la solución depende del número total de jugadores pertenecientes a la solución k que en el peor de los casos podría llegar a ser la totalidad, que podemos definirlo como j. Por lo tanto la complejidad es:

$$\mathcal{O}(j) * (\mathcal{O}(m * n) + \mathcal{O}(m * n) + \mathcal{O}(m * n)) = \mathcal{O}(j) * 3(\mathcal{O}(m * n))$$

Si acotamos, podemos concluir que la complejidad temporal del algoritmo es:

$$\mathcal{O}(j) * \mathcal{O}(m * n)$$

6.4. Complejidad Espacial

La complejidad espacial del algoritmo se centra en la cantidad de memoria utilizada por las estructuras de datos. En este caso, el uso principal de la memoria proviene del diccionario utilizado para contar las apariciones y el arreglo para guardar la solución. Siendo j la cantidad total de jugadores, la complejidad espacial en el peor de los casos donde todos los jugadores son parte de la solución, sera:

$$\mathcal{O}(j) + \mathcal{O}(j) = 2\mathcal{O}(j)$$

Si acotamos, podemos concluir que la complejidad espacial del algoritmo es:

$$\mathcal{O}(j)$$

7. Conclusiones

7.1. Backtracking y PLE

Ambos algoritmos encuentran la solución óptima al problema con la diferencia de que por lo menos hasta un volumen de 200 elementos la programación lineal entera tiene una performance ampliamente mayor que la de backtracking debido a que optimiza de una mejor forma.

7.2. Programación lineal continua

La programación continua resulta eficiente en cuanto a tiempo sin embargo los resultados que consigue están considerablemente lejos del óptimo a pesar de que se encuentren acotados por la cota calculada en la aproximación.

7.3. Programación Greedy

La aproximación obtenida por el algoritmo greedy resulto muy intuitiva de desarrollar, obteniendo resultados muy cercanos al óptimo con tiempos de ejecución muy bajos comparados con el resto de los algoritmos implementados.

7.4. Mediciones

Mediciones de Tiempo por Algoritmo				
Archivos	BackTracking	PLE	PLC	Greedy
5.txt	0s	0.024s	0.012s	0s
7.txt	0.001s	0.019s	0.013s	0s
10_pocos.txt	0.012s	0.014s	0.013s	0s
10_todos.txt	1374s	0.026s	0.022s	0s
10_varios.txt	5.793s	0.016s	0.017s	0s
15.txt	0.593s	0.029s	0.014s	0s
20.txt	2.273s	0.025s	0.018s	0s
50.txt	16.23s	0.032s	0.021s	0s
75.txt	138.3s	0.246s	0.019s	0.001s
100.txt	406.0s	0.453s	0.022s	0s
200.txt	1491s	0.874s	0.050s	0.002s

Figura 10: Mediciones de tiempo

Cantidad de Elementos Seleccionados por Algoritmo				
Archivos	BackTracking	PLE	PLC	Greedy
5.txt	2	2	2	2
7.txt	2	2	2	2
10_pocos.txt	3	3	6	3
10_todos.txt	10	10	10	10
10_varios.txt	6	6	6	7
15.txt	4	4	11	5
20.txt	5	5	12	5
50.txt	6	6	13	7
75.txt	8	8	20	9
100.txt	9	9	23	11
200.txt	9	9	27	10

Figura 11: Soluciones