.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

25 de noviembre de 2023

Juan Pablo Carosi Warburg	Mateo Daniel Vroonland	Lucas Rafael Aldazabal
109386	109635	107705

Índice

1. Introducción

En este informe haremos un estudio del problema planteado, un caso particular del Hitting-Set Problem. El problema consiste en que dado un conjunto de n periodistas, donde cada periodista pide una cantidad k de jugadores, hay que encontrar el subconjunto mínimo de jugadores tal que este subconjunto tenga como mínimo un jugador pedido por cada periodista. Para resolver esto, primero demostraremos que este problema es NP-Completo. Luego, resolveremos el problema utilizando las técnicas de programación lineal entera y backtracking. Por último, realizaremos unos algoritmos para aproximar el resultado en tiempo polinomial, lo cual es de gran ayuda ya que este problema al ser NP-Completo, su versión de optimización se resuelve en tiempo exponencial. El inconveniente es que a costa de disminuir el tiempo de ejecución, perdemos precisión en el resultado.

Ejemplo de Resolución

2. Demostración de Hitting-Set Problem en NP

Un problema de decisión está contenido en NP si se demuestra que una solución candidata puede ser verificada para el algoritmo en tiempo polinómico.

En Hitting-Set una solución candidata S consiste en un set de elementos seleccionados donde se debe verificar tanto que su tamaño sea menor o igual a k, como que este cubra al menos un elemento de cada subconjunto.

Primero, validar que el tamaño de S sea menor o igual a k, esto es trivial, y obviamente, polinomial.

Luego, por cada set SC, siendo este un subconjunto del problema, se tiene que comprobar si al menos uno de los elementos de S está incluido en SC. Recorrer los subconjuntos tiene una complejidad O(n), y por cada subconjunto se verifica si hay al menos un elemento de S que esté incluido en el subconjunto, recorrer S es O(k) y ver si este esta en SC es complejidad O(1). Por lo que la verificación completa termina resultando en O(n) * O(k) * O(1) = O(n * k) siendo k una constante del problema de decisión por lo que la validación del subconjunto candidato también es polinomial.

```
def verificar(problema, solucion):
    for subconjunto in problema: # O(n)
        cubierto = False
        for seleccionado in solucion: # O(k)
            if seleccionado in subconjunto: # O(1)
                cubierto = true
        if not cubierto:
            return False
    return true
```

Una vez demostrado que el conjunto solución tiene tamaño igual o menor a k, y es verificable en tiempo polinomial, podemos afirmar que Hitting-Set es NP.

3. Demostración de Hitting-Set Problem en NP-Completo

Para demostrar que un algoritmo es, en efecto, NP-Completo es necesario haber verificado que este sea NP, cosa que ya está hecha en el punto anterior. La otra condición es poder reducir algún problema NP-Completo a este, y por consiguiente, cualquier problema NP-Completo.

En este caso elegimos reducir Vertex-Cover a Hitting-Set. Para esto hay que entender las condiciones que tienen ambas:

- Vertex-Cover busca un subconjunto de vértices que sea capaz de cubrir, con sus adyacencias, todas las aristas del grafo.
- Hitting-Set busca una selección de elementos que incluya al menos un elemento de cada subconjunto del problema.

A primera vista pareciera que realizan búsquedas totalmente distintas, pero se puede ver Vertex-Cover desde otra perspectiva que lo acerca más.

Podemos considerar que si su objetivo es cubrir todas las aristas, lo que termina buscando es seleccionar al menos un vértice por cada arista del grafo. Esto es mucho más cercano a los buscado por Hitting-Set y se puede hacer el paralelismo. Por cada arista se crea un subconjunto que la representa y los elementos del subconjunto son los dos vértices que conecta. De esta forma nos termina quedando.

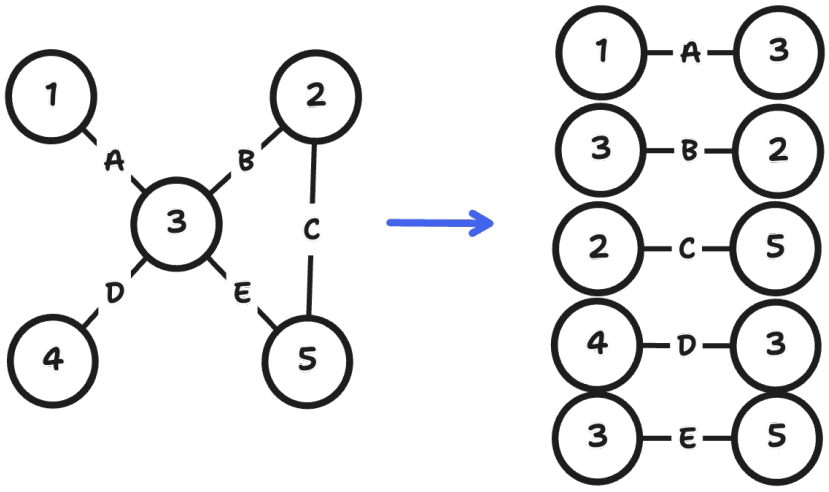


Figura 1: Otra Perspectiva para Vertex-Cover

- Subconjuntos = aristas
- Salida = selección de vértices que conectan todas las aristas

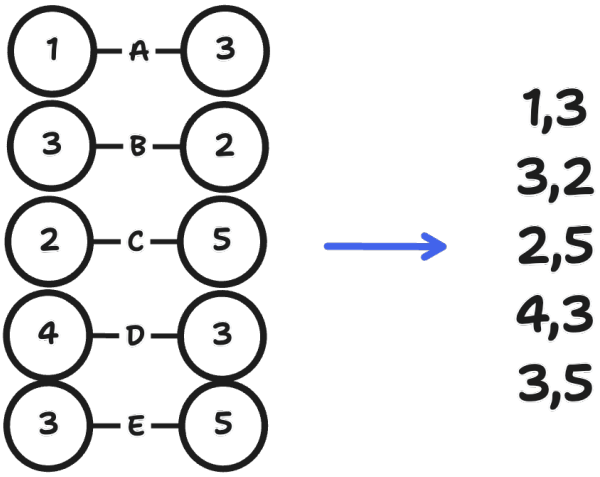


Figura 2: De Vertex-Cover a Hitting-Set

La transformación para llevar del grafo a los subconjuntos para utilizar Hitting-Set también es en tiempo polinomial ya que consiste en recorrer todas las aristas e insertar un subconjunto con dos elementos por cada una.

De esta forma demostramos que Vertex-Cover se puede reducir con una transformación en tiempo polinomial y un uso de Hitting-Set, haciendo que este último sea NP-Completo

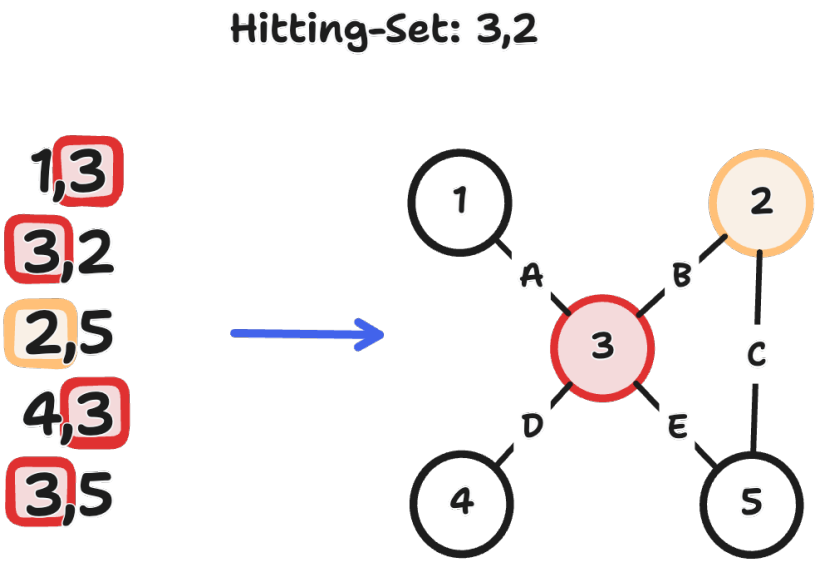


Figura 3: Solución de Hitting-Set en Vertex-Cover

4. Resolución óptima del problema

Como fue aclarado anteriormente, este problema fue resuelto en su manera óptima utilizando backtracking y programación lineal entera. A continuación, vamos a explicar el funcionamiento de cada algoritmo y cómo se comportan los mismos con sets con volumen de información elevado.

4.1. Backtracking

El problema de este algoritmo fue que en un inicio lo implementamos erróneamente ya que hicimos su versión de decisión, la cual era más simple. El tema es que esto nos generó una perspectiva errónea ya que, al ser de decisión con cierto k, nos permitía podar múltiples caminos fácilmente. Entonces, cuando quisimos trasladarlo a su versión de optimización, no lográbamos llegar al resultado esperado debido a la poda excesiva.

Allí fue que nos dimos cuenta que en la versión de optimización es siempre necesario recorrer por ambos lados del árbol, o sea, considerando el jugador y no considerándolo, ya que no tenemos forma de saber previamente por cual camino va a resultar mejor. Esto ralentizó fuertemente el algoritmo, pero nos permitió llegar a los resultados esperados sin inconvenientes. Cabe recalcar que previamente a llamar el algoritmo, leemos y guardamos todos los subconjuntos de cada periodista y, guardamos los jugadores totales sin repeticiones ya que uno puede ser pedido por múltiples periodistas.

Para armar este algoritmo, nos basamos fuertemente en la receta para crear algoritmos de backtracking. La cual está dada por:

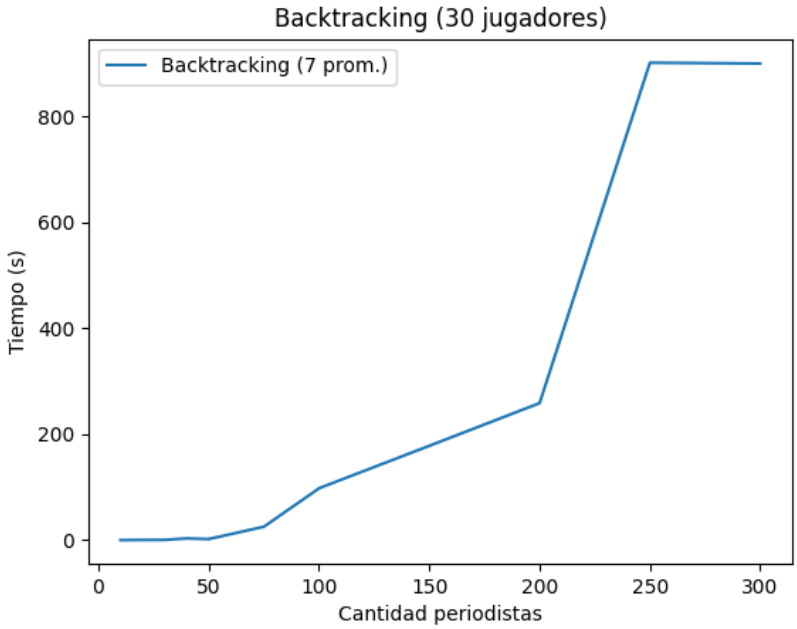
1. Verificamos si es una solución válida.
2. Si la solución actual es peor que mejor solución podamos esa rama del árbol. Caso contrario reemplazamos la mejor solución.
3. Expandimos el árbol llamando recursivamente teniendo en cuenta el jugador actual
4. Luego eliminamos ese jugador de los considerados y expandimos el árbol llamando recursivamente sin considerar a ese jugador.

Como hay que recorrer diversos caminos, esto nos dificultó notablemente al momento de la poda, por lo tanto, solo conseguimos podar caminos donde su solución actual es peor que la mejor solución guardada en ese momento.

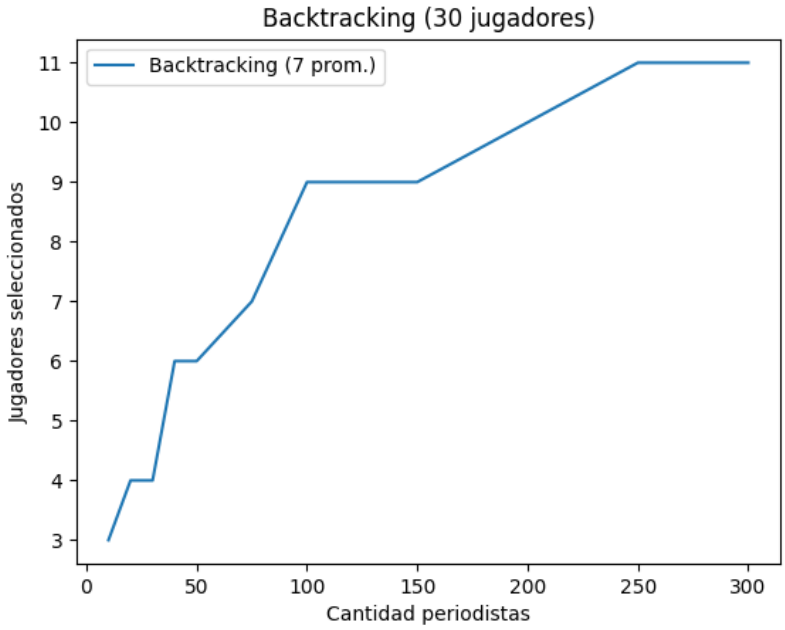
Para validar si la solución actual es válida, simplemente recorremos los subconjuntos de cada periodista guardados anteriormente y verificamos que al menos un jugador de nuestra solución haya sido pedido por ese periodista.

```
def es_valido(seleccionados, subconjuntos):
    for subset in subconjuntos:
        if not any(e in seleccionados for e in subset):
            return False
    return True
```

Cabe recalcar que los jugadores totales pedidos por los periodistas sin repeticiones los almacenamos en una lista ya que usando un set el algoritmo no daba siempre la misma solución.



En este primer gráfico se puede observar fácilmente como hasta con una cantidad pequeña de periodistas el algoritmo demoraba un tiempo notable. Además, se pude notar la exponencialidad del tiempo tardado en comparación con la cantidad total de periodistas en cada set. Para este gráfico se utilizó una base de 30 jugadores totales y cada periodista pide 7 jugadores en promedio.



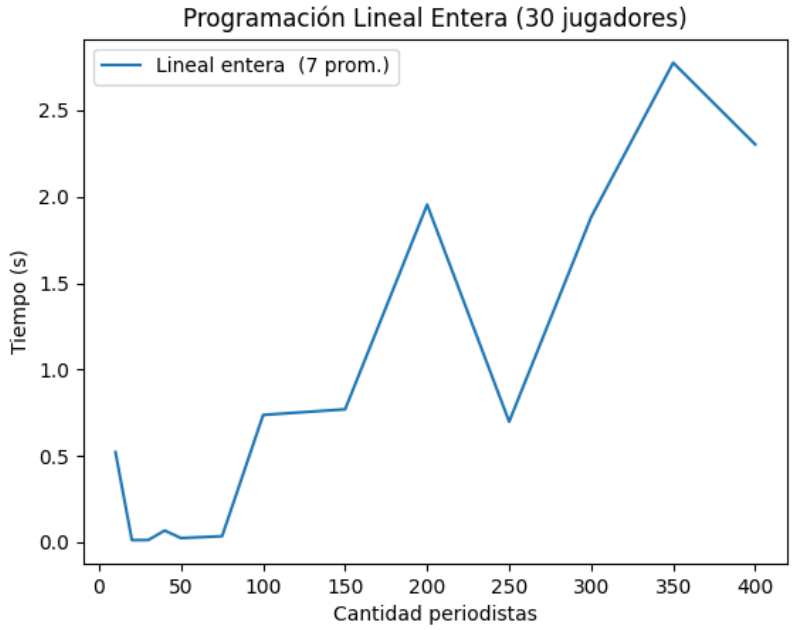
Por otro lado, con este segundo gráfico se puede observar la cantidad mínima de jugadores seleccionados en comparación contra la cantidad de periodistas en cada set.

4.2. Programación Lineal Entera

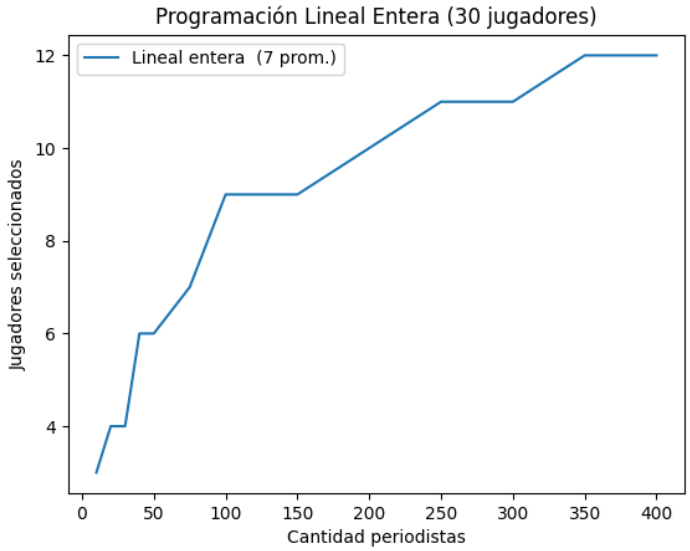
Este algoritmo fue más fácil de idear que el anterior, sin embargo, de todas maneras también trajo ciertas complicaciones.

Primero, antes de programar el algoritmo, intentamos pensar profundamente el sistema de ecuaciones que deberíamos pasarle a nuestro resolvedor de sistemas lineales. La primera de ellas era crear una variable binaria para cada jugador de los pedidos, sin repetidos. La segunda obviamente era minimizar la cantidad de jugadores. Y, por último, planteamos la restricción de que haya un solo jugador por cada subgrupo de periodista.

Sin embargo, esto fue los que nos generó inconvenientes ya que era incorrecto que tenga que haber exactamente un solo jugador. Al corregirlo para que haya al menos un jugador, el algoritmo lograba resolver exitosamente los casos provistos por la cátedra. Los resultados se pueden observar en el respectivo archivo markdown ubicado en el directorio de resultados.



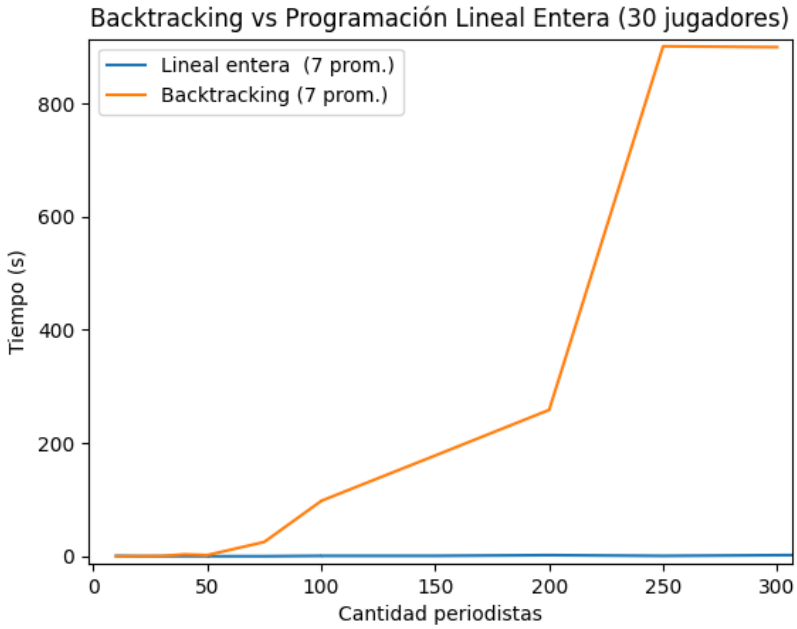
Podemos observar que el gráfico no sigue un patrón definido, tiene varios altibajos. Esto se nos ocurre que puede ser debido a que en ciertos sets, dado a la creación aleatoria de datos, puede ocurrir el caso donde un jugador o varios se repitan en múltiples pedidos de periodistas. Lo cual lleva a simplificar varias ecuaciones redundantes del sistema y que se ejecute en un tiempo menor.



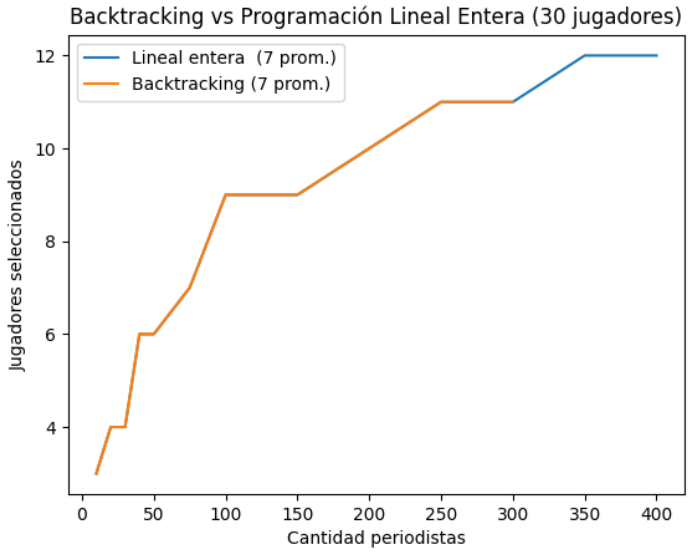
Por otro lado, con este gráfico podemos ver la cantidad de jugadores seleccionados por cada subconjunto.

4.3. Backtracking vs PLE

A continuación, si bien ya analizamos cada caso por separado, ahora vamos a observar ciertos gráficos donde se puede ver la información de cada algoritmo en conjunto para destacar ciertos aspectos importantes.



En primer lugar, por más que ambos tengan una complejidad exponencial, es muy notorio que el algoritmo de programación lineal entera es absurdamente más rápido. Esto seguramente sea debido a que como el resolvedor del sistema lineal es ajeno a nuestro código, debe estar sumamente optimizado para realizar su trabajo. Además, acá se destaca nuestro inconveniente de realizar una pequeña poda.



En segundo lugar, con este gráfico podemos determinar que ambos algoritmos creados consiguen el óptimo exitosamente con diversas cantidades de periodistas. Se puede observar que los resultados de ambos algoritmos son exactamente iguales.

5. Aproximaciones

Como vimos anteriormente, ambos algoritmos corren en tiempo exponencial y con un mínimo incremento de periodistas, el tiempo de ejecución crece rápidamente. Por ello es que vamos a realizar aproximaciones para lograr disminuir el tiempo de ejecución con cada set de periodistas.

5.1. Programación Lineal Relajada

El primer algoritmo de aproximación, tal como indica el enunciado, va a ser utilizando la programación lineal relajada.

Para ello, este algoritmo consiste en dejar que las variables de cada jugador tomen un valor real entre 0 y 1, y luego, tomar cierta condición para decidir con cuales jugadores nos quedamos. Esto fue provisto por la cátedra donde indica elegir aquellos jugadores que tengan un valor mayor o igual a 1/b, donde b es el tamaño del pedido con mayor cantidad de jugadores de ese set.

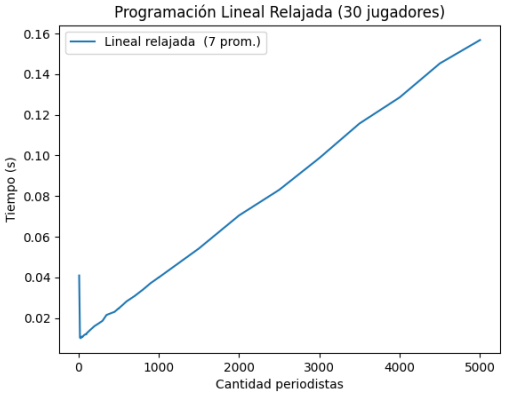


Figura 4: Tiempo de ejecución.

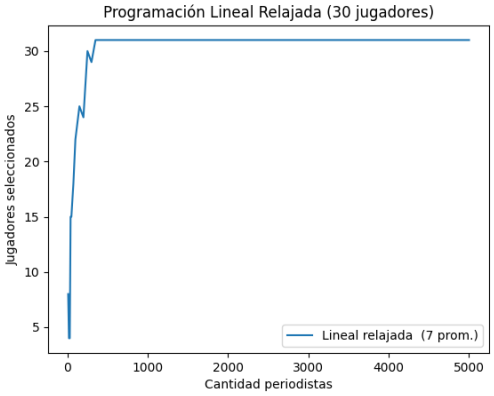


Figura 5: Longitud de la solución

Acá se observa la rapidez de nuestro algoritmo comparado al gráfico mostrado en la sección de programación lineal entera. La asíntota en la segunda imagen se debe a que la solución que devuelve es tan simple que termina utilizando todos los jugadores que teníamos registrados.

Esto genera que nuestro algoritmo resuelva en tiempo polinomial el problema, pero como trade off, devuelve un resultado bastante alejado del óptimo. Esto se puede observar en el siguiente gráfico comparando la performance tanto en tiempo como en precisión entre la programación lineal entera y la relajada.

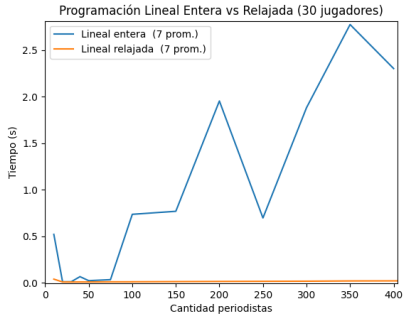


Figura 6: Tiempo de ejecución.

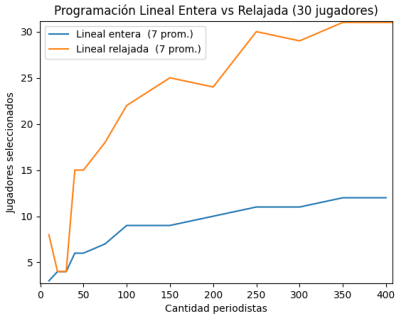
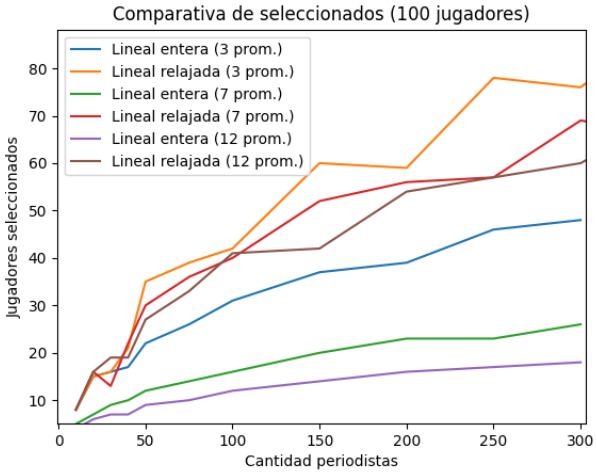


Figura 7: Longitud de la solución

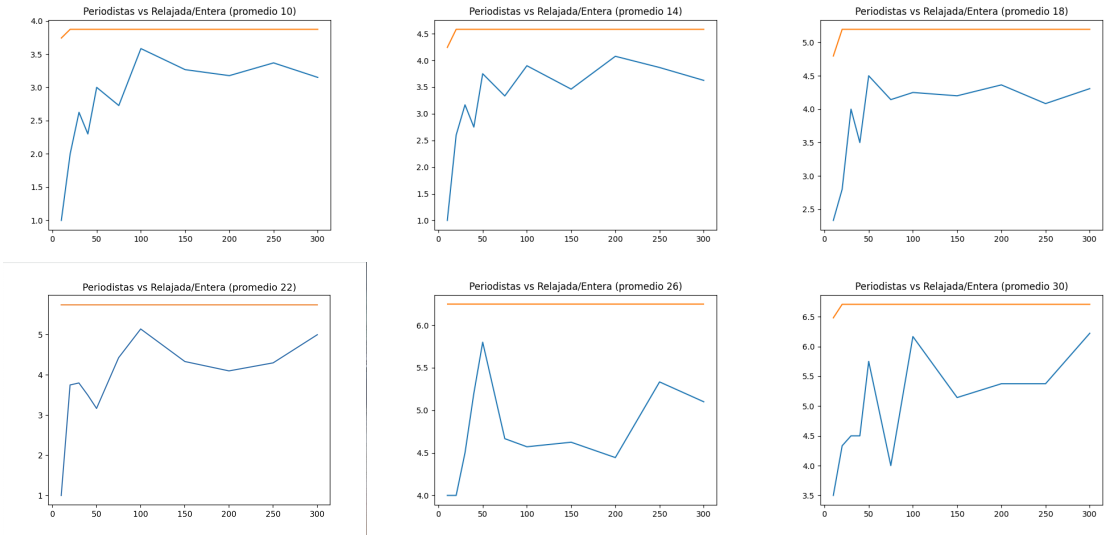
Pero esto nos lleva a otra pregunta, ¿Qué tan buena es nuestra aproximación? Porque podemos notar que no es muy favorable, pero deberíamos intentar acotar el resultado de cada aproximación.

Para atacar esta idea, primero intentamos buscar cierta correlación entre la cantidad de jugadores totales y la diferencia entre lacantidad mínima pedida y cantidad máxima pedida por un periodista del set. Sin embargo, esto no nos trajo buenos resultados.

Pero, por otro lado, gracias a este gráfico, notamos que a medida que el promedio de jugadores incrementaba, cada vez eran más disparejos los resultados entre PLE y su aproximación. Es por ello, que decidimos enfocarnos fuertemente en el promedio de jugadores, ya que, si a eso lo multiplicábamos por 1.5 (decisión tomada para generar los datos) obtenemos el máximo posible que puede pedir un periodista, o sea, nuestro b.



Además, en el gráfico se observa una curva cóncava que a medida que aumenta la cantidad de periodistas, el incremento de los jugadores seleccionados se iba atenuando. Por ello es que se nos ocurrió buscar una función matemática que tenga un comportamiento similar a este patrón. La función elegida seria la raíz cuadrada de b, y para comprobar que nuestro pensamiento sea correcto realizamos múltiples test con distintos b para comprobar que siempre se cumpla la cota esperada.



Cabe aclarar que la linea amarilla es la cota y la azul son los resultados del algoritmo. Gracias a estas pruebas, podemos ver que nuestro algoritmo a veces se aproxima a la cota pero nunca la supera, lo cual nos demuestra empíricamente que la cota está calculada correctamente.

5.2. Greedy

Para implementar la aproximación por greedy, vimos a primera vista la conveniencia de ordenar los jugadores según la cantidad de periodistas que los solicitan de manera descendiente con el objetivo de poder seleccionar los jugadores que más periodistas cubren, dando "zancadas"mas grandes hacia el objetivo de cubrir todos los subconjuntos.

Para esto hacemos un algoritmo que primero ordene los jugadores de la manera ya mencionada, y luego los vaya insertando uno por uno. Luego de cada inserción verificamos si se cubrió a todos los periodistas. De esta forma el algoritmo recorre todos los pedidos por cada periodista viendo si están incluidos en la nueva solución, y esto se repite hasta que se cubran todos, lo cual puede ser como máximo igual a la cantidad de periodistas.

Esta primer solución tiene como óptimo local el saber que cada jugador que insertas es el más solicitado no incluido hasta el momento por lo que es el que mas chances tiene de contentar a más periodistas. De esta forma con un solo ordenamiento, y por ende muy poco costo computacional, logramos un óptimo con bastante solidez para llegar a un resultado aproximado.

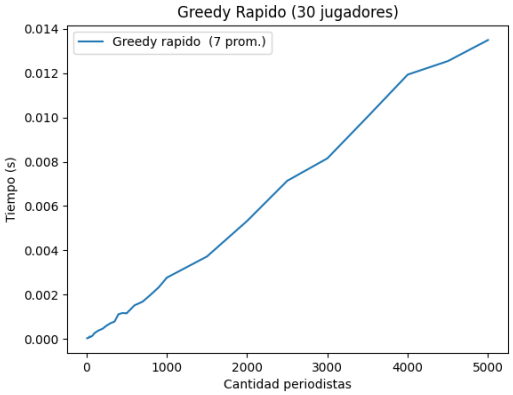


Figura 9: Tiempo de ejecución.

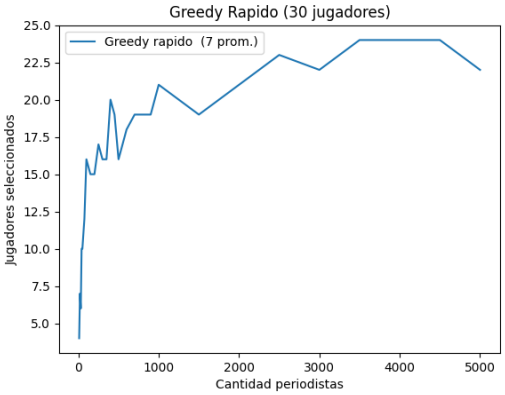


Figura 10: Longitud de la solución

Luego notamos que hay ciertas debilidades en el creer que agregar un jugador con más pedidos necesariamente puede contentar a mas nuevos periodistas sin tener en cuenta si sus periodistas ya estaban contentados por otros jugadores seleccionados previamente que también tenían en sus listas. Por ejemplo:

- Messi: lo piden 20 periodistas (lógico, es el mejor del mundo)
- De Paul: lo piden 8 periodistas
- Colo Barco: lo piden 3 periodistas (bosteros como toda persona de bien)

Primero se selecciona a Messi contentando esos primeros 20 periodistas. Luego iría De Paul para contentar a lo sumo 8 más, pero en este caso todos los que lo piden lo quieren siendo dupla de Messi por los que ningún periodista nuevo quedó satisfecho, mientras que eligiendo al Colo Barco, esos 3 periodistas bosteros (a los que Messi no está en su lista de prioridades) quedaban satisfechos.

Una vez vimos esto, tomamos dos decisiones:

1. Agregar una optimización sencilla al greedy actual para que no inserte jugadores nuevos que no contenten al menos un periodista nuevo.
2. Hacer un segundo algoritmo greedy que se comporte igual, pero actualizando el orden luego de cada inserción y teniendo en cuenta los periodistas ya satisfechos, haciéndolo mas costoso computacionalmente pero mas efectivo.

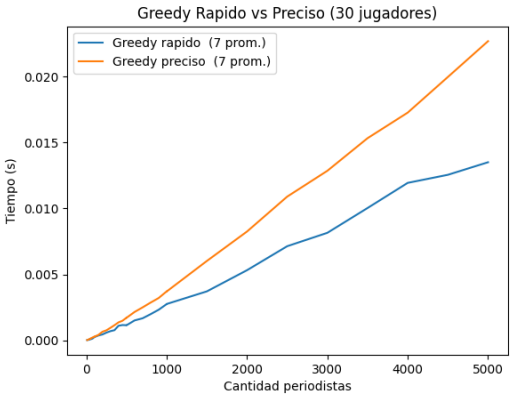


Figura 11: Tiempo de ejecución.

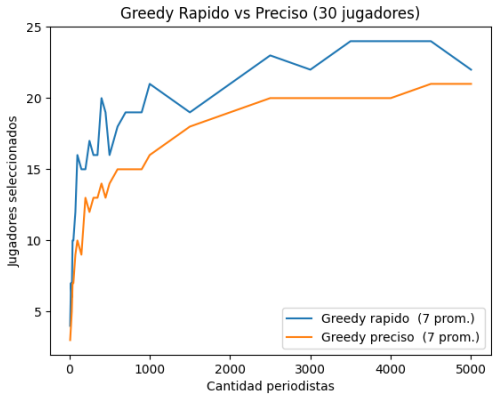
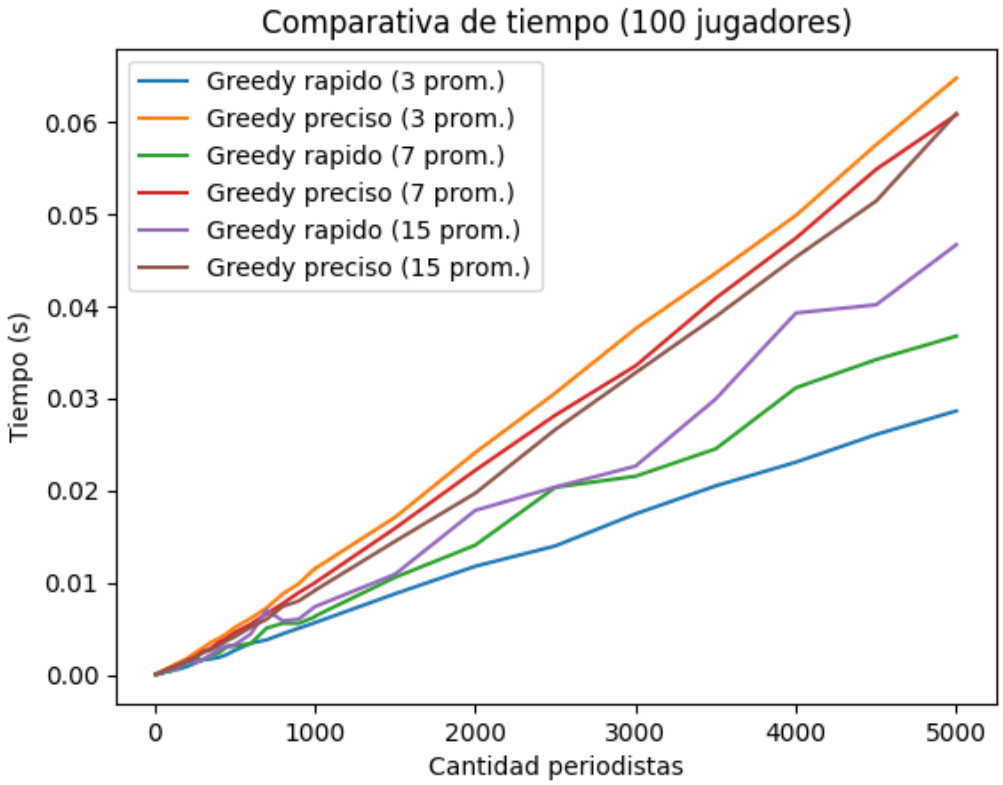


Figura 12: Longitud de la solución

A partir de ahora llamaremos "Greedy Rápidoz "Greedy Preciso".

Como podíamos intuir, el "Greedy Preciso.es consistentemente mas cercano al óptimo que el "Greedy Rápidoz, aunque es mas costoso, no tarda excesivamente más incluso con grandes volúmenes de periodistas (hasta 5000). Esto se debe en parte a que cada iteración sea mucho mas efectiva.

Otra cosa que vimos con las pruebas es que cuantos más jugadores pida cada periodista, más peticiones repetidas va a haber, perjudicando fuertemente al "Greedy Rápido"mientras que el "Greedy Preciso"no se ve alterado.



5.3. Greedy vs Programación Lineal Relajada

Como ultima observación en esta sección de aproximaciones, vamos a comparar las 3 aproximaciones que creamos, y que ya fueron explicadas anteriormente.

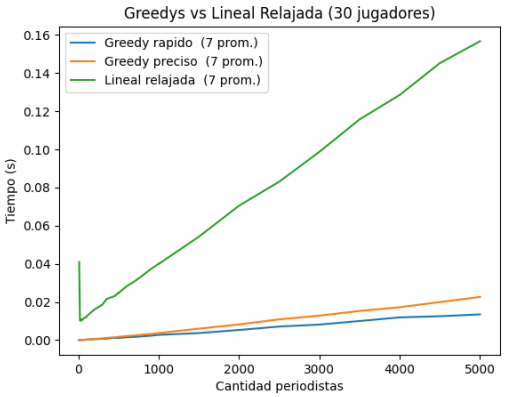


Figura 13: Tiempo de ejecución.

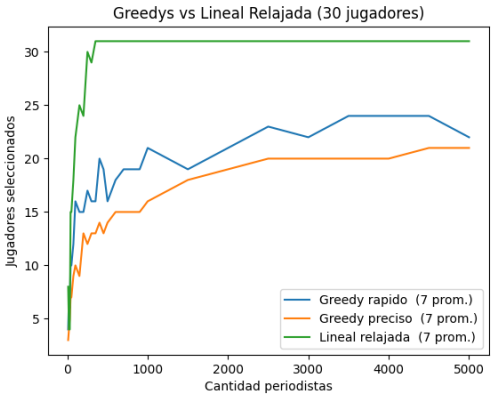
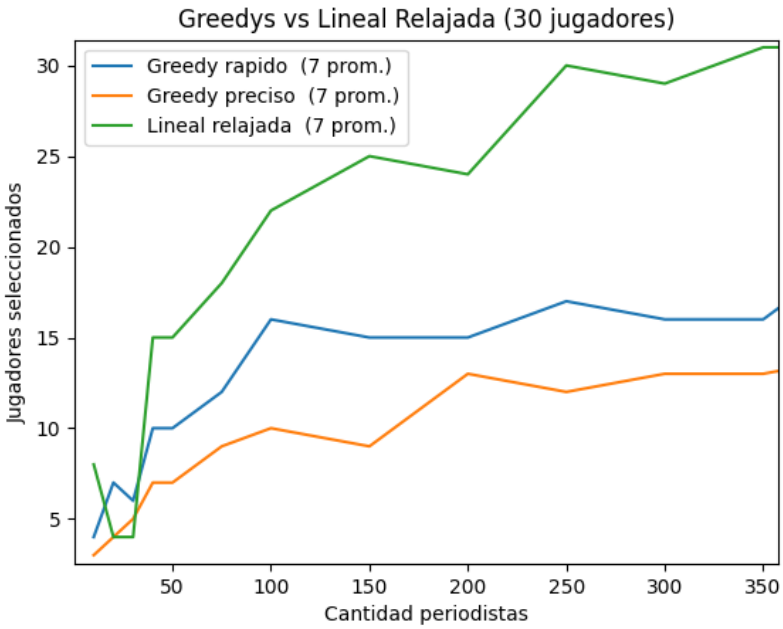


Figura 14: Longitud de la solución

Se puede observar que no solo nuestros algoritmos greedys resultaron mas performantes, sino que también resuelven de una manera más óptima el problema. A continuación veremos otro gráfico pero con más zoom y detalle para dejar bien evidenciadas estas ideas.



6. Anexo diagramas

Por último, queríamos mostrar unos diagramas interesantes que muestran una comparativa del comportamiento de todos los algoritmos en un mismo gráfico. Esto sirve para realmente dimensionar las diferencias entre los mismos.

.

.

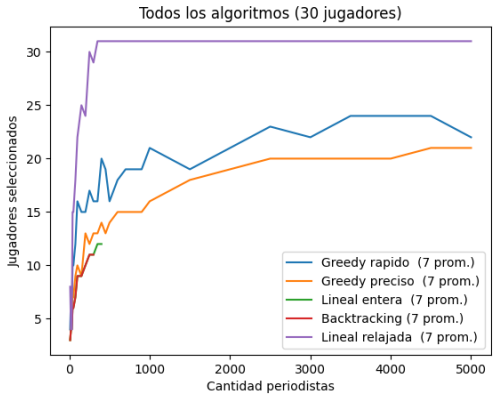


Figura 15: Longitud de la solución.

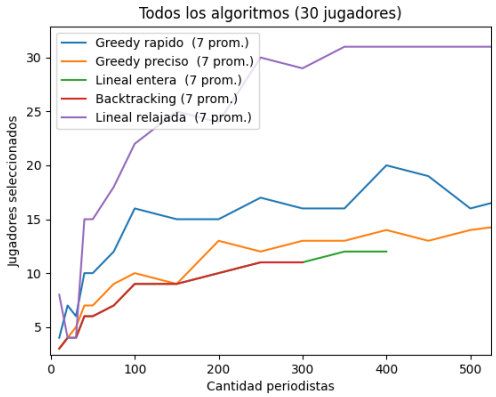


Figura 16: Longitud de la solución con zoom

En estos dos gráficos se puede visualizar la diferencia de la calidad de la solución dada.

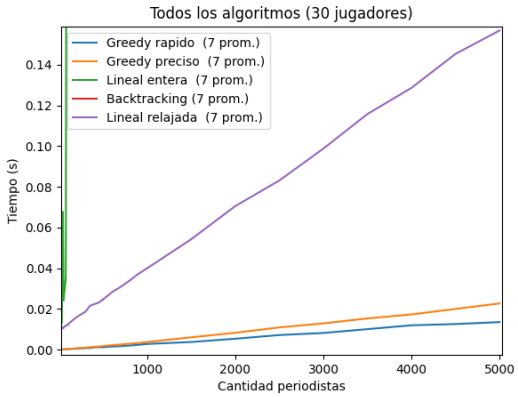


Figura 17: Tiempo de ejecución enfocando la PL relajada

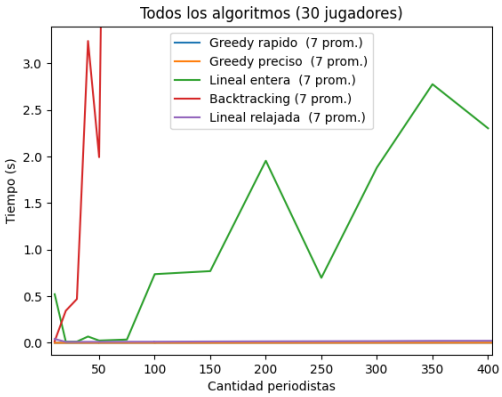


Figura 18: Tiempo de ejecución enfocando PL entera

En esto dos últimos gráficos se puede apreciar la diferencia entre los tiempos de ejecución de cada algoritmo.

7. Conclusiones

Como conclusión, queríamos destacar el gran comportamiento de nuestro algoritmo greedy preciso, el cual es increíblemente rápido y logra dar un solución muy acorde comparándola a la óptima.

Además, otro punto a destacar que si bien lo esperábamos, pero no es semejante magnitud, es la diferencia de tiempo en ejecución entre el algoritmo de backtracking y el de programación lineal entera.

Como regla general que nos llevamos del análisis es que cuando se cuando se tiene un problema de optimización complejo y con muchos datos, generalmente conviene hacer algunos algoritmos de aproximación como el greedy para poder obtener resultados rápidamente con una cercanía al óptimo suficiente para analizar como conviene proseguir. Recién luego de tener el comportamiento aproximado y las decisiones tomadas conviene hacer algoritmo exponencial que de el resultado exacto pero demorando muchas ordenes de magnitud mas que el de aproximación.

PD: este TP muestra por que Scaloni se quiere ir de la selección.