Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

Algoritmos avanzados

Laboratorio 2 – Enfoque voraz

Christofer Rodríguez

Profesor: Cristián Sepúlveda S.

Ayudante: César Rivera M.

Santiago – Chile
1-2021

Tabla de contenidos

Índice de figuras

Índice de tablas

1. INTRODUCCIÓN

Las técnicas de diseño son un punto importante a la hora de construir un programa computacional que resuelva un problema, con esto en mente se construyeron 2 programas para solucionar 2 problemas planteados, el primero de estos consiste en que tenemos un conjunto de números enteros con un valor y ponderación para cada uno de ellos, con estos debemos obtener la suma total de valores del subconjunto que maximice la suma de estos valores sin superar un límite de ponderación establecido. El segundo problema consiste en obtener la cantidad mínima de semanas que se tardaría en completar una obra, la cual posee distintas tareas que demoran una semana en completarse, pero existen tareas que no pueden hacer a la misma vez que otras. Para solucionar estos problemas es necesario implementar un programa en el lenguaje de programación C, el cual sea diseñado siguiendo un enfoque voraz.

Los objetivos de este informe son explicar el método utilizado para resolver estos problemas, describir los experimentos realizados para la obtención de datos, para finalmente analizar los resultados obtenidos, identificando los principales hallazgos y limitaciones de las soluciones propuestas.

Esto se llevará a cabo, primeramente, describiendo la metodología utilizada, explicando en que consiste el tipo de algoritmo utilizado, el proceso de implementación y los resultados obtenidos en los distintos experimentos realizados; luego se analizarán los resultados obtenidos en la sección de discusión, en esta sección se expondrán los principales hallazgos, falencias de la implementación y se comparará con los resultados obtenidos en el anterior laboratorio, finalmente sugerirán mejoras para futuras investigaciones e implementaciones.

2. MÉTODO

Para comprender la implementación realizada, es necesario primero comprender en que consiste un algoritmo voraz; este tipo de enfoque consiste en a partir de un conjunto de candidatos (entradas del problema) en cada momento se elige al candidato más prometedor, se comprueba si al incluir dicho elemento el subconjunto solución que poseemos sigue siendo una posible solución, dicha condición está dada por la naturaleza del problema; una vez ya se ha comprobado todos los candidatos, se ha alcanzado el objetivo o cuando el candidato más prometedor no genera una posible solución si es ingresado al subconjunto solución, se entrega este subconjunto solución obtenido. Este tipo de enfoque disminuye el tiempo empleado para encontrar una solución, pero a su vez disminuye la calidad de la respuesta, esto significa que dependiendo del problema la solución obtenida puede o no ser la solución óptima del problema.[1]

Para solucionar los dos problemas propuestos con un enfoque voraz en el lenguaje de programación C, se desarrollaron 2 implementaciones, una para cada uno de estos problemas:

2.1. Solución problema 1

Para esta implementación se decidió hacer uso de estructuras para representar los elementos con sus respectivos valores, ponderaciones, además un valor que representa la relación entre su valor con respecto a su ponderación, este valor se obtuvo dividiendo ambos datos respectivamente.

Ya que los datos para este problema son ingresados mediante un archivo de texto plano con cierta estructura, se implementó un algoritmo que, tras recibir el nombre del archivo de entrada, abre dicho archivo en modo lectura, y a la vez que este era leído, la información contenida en él se almaceno en la estructura que representa cada elemento. Luego de generar el elemento con la información correspondiente, este se agrega a un arreglo que, tras haber leído la totalidad del archivo, es retornado con todos los elementos en él.

Una vez leído el archivo de entrada, se comprueba que no haya existido un error al momento de leer dicho archivo. Si se comprueba que no existió un problema, se procede a ordenar los elementos de mayor a menor con respecto a su valor por ponderación, esto se realizó utilizando el ordenamiento de merge sort, ya que el objetivo de esta experiencia no es implementar un algoritmo de ordenamiento, se utilizó una implementación existente y se modificó para ser utilizada con el tipo de dato del problema. [2]

Ya habiendo ordenado los elementos de mayor a menor, sabemos que el primer elemento de la lista siempre es el candidato más prometedor, ahora se procede a generar la solución al problema. Mediante un bucle se comprueba si el candidato más prometedor de la lista (primer elemento inicialmente) al ser incluido a la solución, genera una solución prometedora, esto se comprueba sumando su ponderación con la ponderación de los elementos ya incluidos en

la solución prometedora y comparándola con el límite de ponderación, si se genera una solución prometedora, se suma el valor de este elemento con el valor de la solución prometedora y se aumenta en 1 el valor para indexar la lista de candidatos; en el caso de que no se cumpla con la ponderación límite, solamente se aumenta en 1 el índice para comprobar el siguiente elemento más prometedor en el siguiente ciclo.

El bucle que genera la solución termina cuando ya se han comprobado todos los candidatos o cuando no se pueden seguir agregando candidatos a la solución, en este caso, la ponderación total de la solución prometedora es igual al límite de ponderación.

Finalmente, una vez terminado el bucle, se muestra por consola el valor total de la solución obtenida.

Una de las principales consideraciones al momento de realizar esta implementación, fue el criterio de ordenamiento para los candidatos, ya que si los elementos fueran ordenados de mayor a menor en cuanto a su valor, probablemente estos elementos tuvieran una mayor ponderación, en el caso en que se ordenaran de menor a mayor de acuerdo a su ponderación, era posible que dichos elementos poseyeran una menor valoración; por lo anterior se decidió ordenar de mayor a menor con respecto a la relación de valor por ponderación, de esta manera se logró alcanzar un punto intermedio entre los anteriores 2 casos.

El código de esta solución fue analizado contando la cantidad de instrucciones a ejecutar para obtener el tiempo de ejecución teórico y el orden de complejidad respectivamente:

- $T(n) = nlog(n) + 16n + 15$
- $O(nlog(n))$.

La ecuación del orden de complejidad se utilizó para representar el aumento en el tiempo de ejecución del algoritmo en el siguiente gráfico:

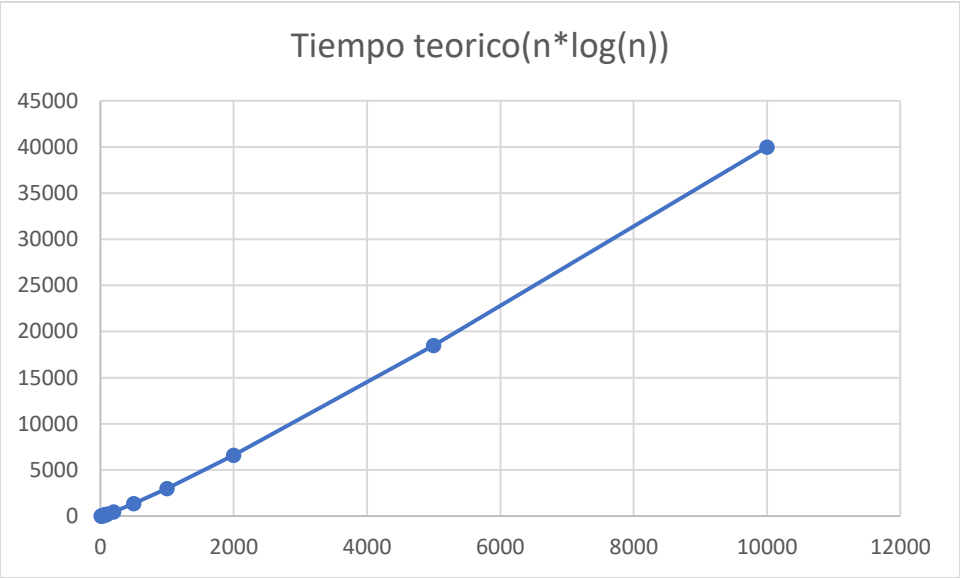

Ilustración 1: Gráfico complejidad algorítmica, solución problema 1

Además, para contrastar el tiempo teórico con el tiempo real de ejecución, se hizo uso de la librería **time.h** para la toma del tiempo de ejecución del programa con las entradas.

Tabla 1: Tamaño entrada vs tiempo de ejecución, solución 1

Tamaño de entrada	Tiempo de ejecución (seg)
10	0
20	0
30	0
40	0
1000	0
10000	0

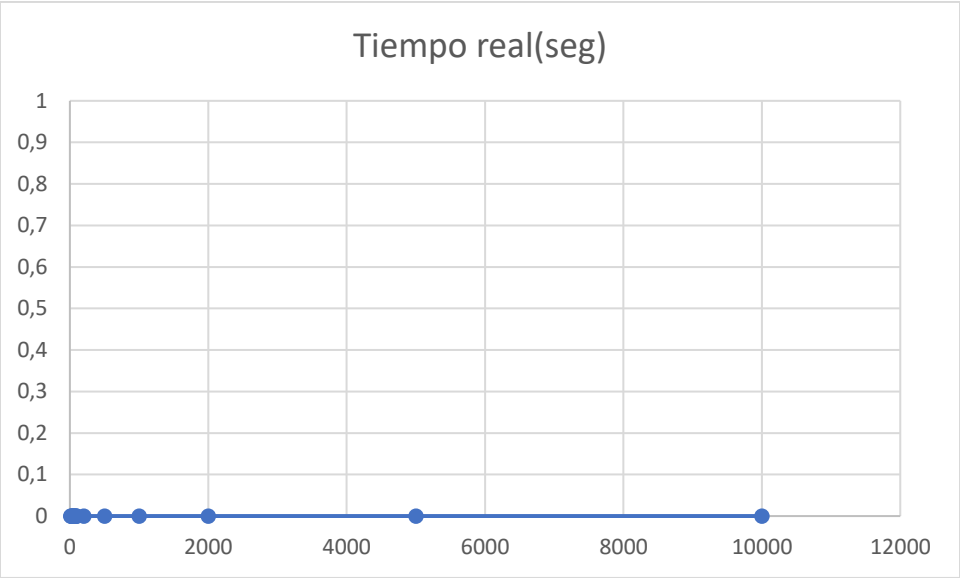

Ilustración 2: Tiempo de ejecución real, solución problema 1

2.2. Solución problema 2

Para la resolución del segundo problema planteado, el cual consiste en obtener el mínimo de semanas que se podría terminar una obra, distribuyendo cada tarea en una semana en la cual no se debe encontrar otra tarea con la cual presente conflicto (no se pueden realizar a la vez). Para modelar este problema se decidió utilizar un grafo como estructura de dato para la implementación, esta estructura posee 3 componentes, un entero que almacena la cantidad de nodos (tareas) presentes en el grafo, la cantidad de aristas (conflictos) y un arreglo de tipo entero de 2 dimensiones, este será utilizado para representar el grafo mediante una matriz de adyacencia, en la cual las aristas entre cada nodo, representará la existencias de un conflicto entre las tareas conectas por esta arista.

Al igual que el problema anterior, se comienza leyendo el archivo de entrada que contiene los datos del problema, primero se leen los 3 datos de la primera línea los cuales corresponden a un identificador, la cantidad de nodos y la cantidad de aristas respectivamente, el primero de estos no es utilizado mientras que los 2 restantes son utilizados para crear un grafo con una matriz de adyacencia de tamaño nxn, siendo n la cantidad de nodos(tareas), que inicialmente solo contendrá 0's. Una vez creado el grafo, se procede a leer el resto del archivo de entrada, cada línea siguiente representa el conflicto entre dos tareas, en este caso, una arista entre ambos nodos, una vez leído los valores se accede a la matriz de adyacencia y se escribe un 1 en la posición (i, j) y (j, i), siendo estos la primera tarea y la segunda respectivamente; de esta manera se representa la existencia de una arista entre ambos nodos. Dicho proceso se repite para cada línea del archivo, para una vez terminado, retornar dicho grafo.

Ahora, ya que para el algoritmo voraz debemos ordenar nuestros candidatos de manera decreciente en cuanto a qué tan favorable es, para esto se crea un arreglo de una estructura llamada **Elemento**, esta estructura representada un nodo (tarea), en ella se almacena el identificador del nodo y la cantidad de aristas que posee (conflictos). Esto se realiza mediante la función **contarConflictos** que se encarga de sumar cada columna de la fila correspondiente a cada nodo. Una vez realizado esto, se ordena esta lista de manera creciente con respecto a la cantidad de aristas que poseen; al igual que en el primer problema, se utilizó el mismo algoritmo de ordenamiento merge sort, solamente que modificado para ordenar este tipo de datos.

Antes de comenzar a asignar las tareas, se crea un arreglo de tipo entero inicialmente lleno de ceros, este arreglo será utilizado para almacenar la semana en la que se asignó cada tarea, de manera en la que la semana en la que se asignó la tarea n, se encontrara en la posición $n - 1$ dentro del arreglo.

Luego para asignar cada tarea en una semana, se realizará usando un bucle en donde el índice de este bucle será utilizado para acceder al elemento en esa posición dentro del arreglo que contiene los nodos ordenados, de esta manera se asignará primero los candidatos más prometedores, a su vez también tenemos las semanas, estas también son candidatos y el que tan prometedores son, corresponde a que tan pequeño es el número, con este número siempre mayor o igual a 1, así se intentará asignar la tarea primero en las semanas que ya está en uso y no agregar semanas extras innecesariamente.

Dentro del bucle, el flujo se divide principalmente en dos bloques, primero se comprueba si la tarea actual se puede asignar en la primera semana, esto a través de la función **puedoPonerTarea**, esta revisa dentro del arreglo que contiene la semana en la que se asignó cada tarea si hay alguna tarea en la semana que se desea asignar la nueva tarea, si ya hay una tarea es esta semana se comprueba dentro del grafo si existe una arista entre la nueva tarea y la tarea ya asignada a esta semana, esto se repite hasta comprobar con todo el arreglo. En el caso de que no exista conflicto con ninguna tarea ya asignada en esta semana, la tarea es asignada en esta y se continua con el siguiente ciclo del bucle para asignar la tarea a continuación.

Si es que en la comprobación se encuentra que la tarea no pude ser asigna en la primera semana, se realiza en segundo bloque, en este inicialmente se comprueba si la tarea puede ser asignada en alguna de las semanas que ya tienen al menos una tarea asignada, si se puede asignar a una de estas semanas ya ocupadas, se sigue la misma lógica que en el primer bloque, se guarda la información de la semana en la que se asignó y se continua con el ciclo principal. En la situación en la que se comprueban todas las semanas ya ocupadas y la tarea no se puede asignar en ninguna de estas, se asigna en una nueva semana, se aumenta en 1 la cantidad de semanas totales (semanas con al menos una tarea asignada) y se continua con el ciclo principal.

Finalmente, terminada la asignación de todas las tareas del problema, se muestra por consola el resultado obtenido mediante este programa, el cual representa el mínimo de semanas requeridas para realizar todas las tareas presentes.

El código de esta solución fue analizado contando la cantidad de instrucciones a ejecutar para obtener el tiempo de ejecución teórico y el orden de complejidad respectivamente:

$$T(n) = 1 + 7n + 9n^2 + 3n^2$$

$$O(n^3)$$

La ecuación del orden de complejidad se utilizó para representar el aumento en el tiempo de ejecución del algoritmo en el siguiente gráfico:

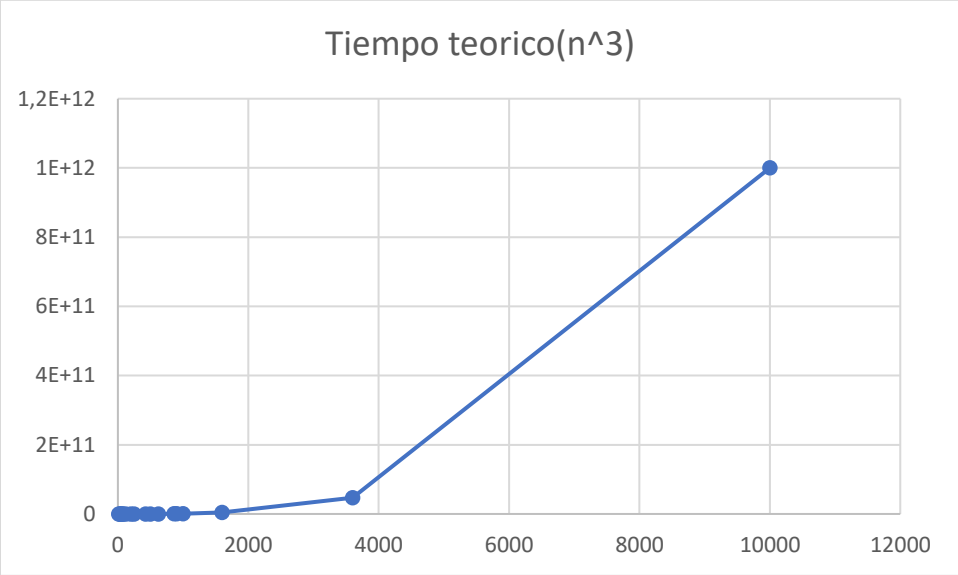

Ilustración 3: Gráfico complejidad algorítmica, solución problema 2

Además, para contrastar el tiempo teórico con el tiempo real de ejecución, se hizo uso de la librería **time.h** para la toma del tiempo de ejecución del programa con las entradas.

Tabla 2: Tamaño entrada vs tiempo de ejecución, solución 1

Tamaño de la entrada	Tiempo (seg)
11	0
37	0
81	0
197	0
500	0
864	0
1000	0
10000	6

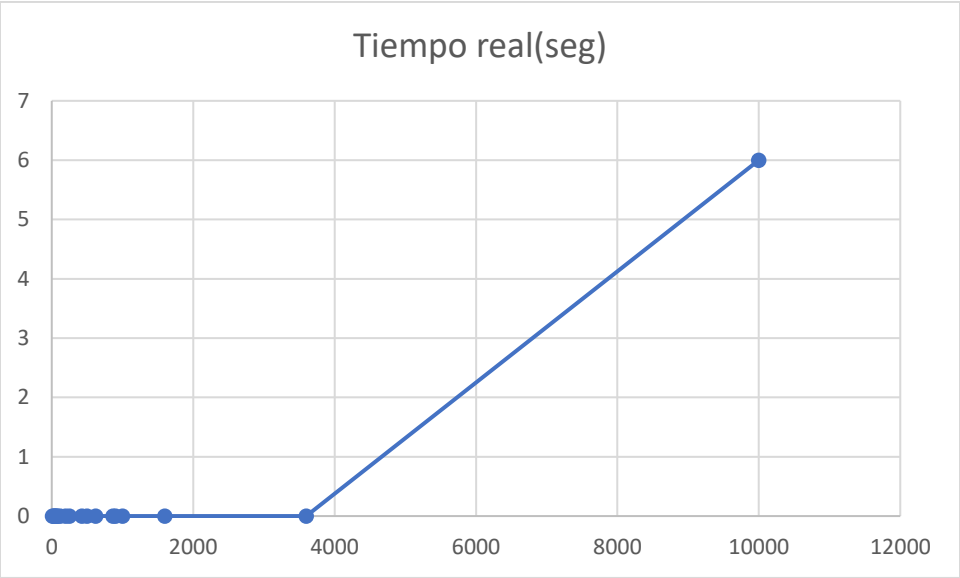

Ilustración 4: Tiempo de ejecución real, solución problema 2

(*) Es importante señalar que los experimentos realizados con ambos programas se realizaron en un computador común con sistema operativo Windows 10 , procesador Intel Core i3-8100 de 3.60 GHz y 16 GB de memoria RAM. Los resultados pueden variar en un computador, entorno o situación diferente.

3. DISCUSIÓN

3.1. Análisis solución problema 1

Luego de analizar los resultados obtenidos de los experimentos realizados con el programa construido, se puede observar que la aplicación del enfoque voraz, mejora enormemente el tiempo empleado en entregar una solución para el problema. Si bien mediante la Ilustración 1 y la Ilustración 2, podemos notar que existe una diferencia entre el tiempo teórico de ejecución y el tiempo real, esto se debe a que el tiempo de respuesta del algoritmo voraz para este caso es increíblemente bajo, incluso para entradas grandes, quizás si se hubiera utilizado un método que entregará un tiempo de ejecución más preciso, la forma de las curvas sería semejante. Si comparamos los resultados obtenidos con este enfoque y los comparamos con los resultados obtenidos en la experiencia anterior, en donde se resolvió el mismo problema, pero mediante la aplicación de enumeración exhaustiva.

Tabla 3: Tamaño de entrada vs tiempo de ejecución, enumeración exhaustiva

Tamaño de entrada	Tiempo de ejecución (seg)
10	0
20	0,225
30	138,716
40	3600+

***Estos resultados de la experiencia anterior fueron medidos utilizando la misma máquina utilizada para probar los programas de este trabajo.**

Si comparamos los resultados obtenidos mediante el enfoque voraz (Tabla 1) con los obtenidos utilizando enumeración exhaustiva (Tabla 3), se puede ver de mejor que manera este mejoramiento de rendimiento. Si bien se logró una mejora en el tiempo de ejecución, la aplicación de este enfoque provocó que se perdiera calidad en la respuesta entregada, si bien la enumeración exhaustiva requiere un tiempo sumamente mayor para obtener una respuesta, a diferencia del enfoque voraz, tendremos la certeza de que obtendremos la respuesta correcta u óptima para nuestra problemática.

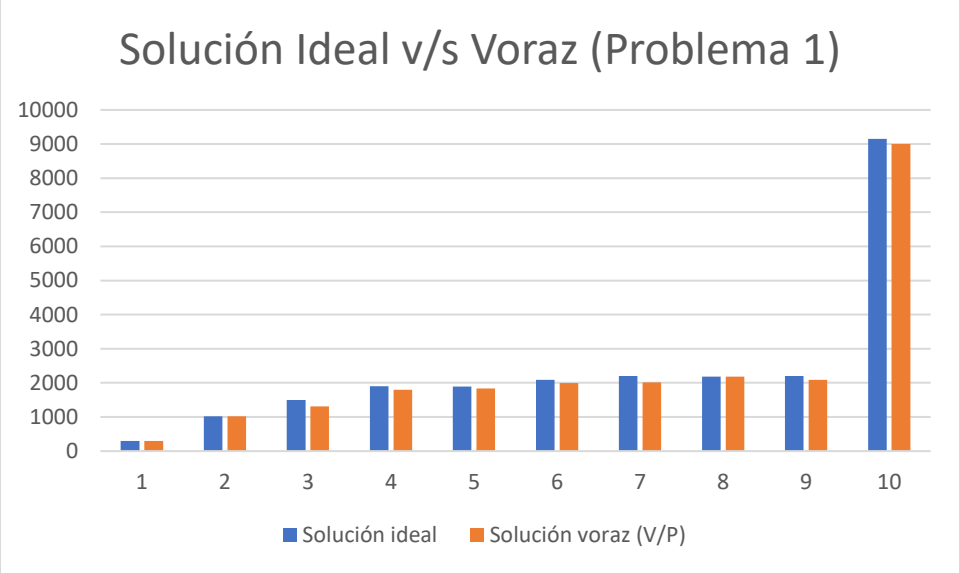

Ilustración 5: Solución ideal vs solución voraz (entradas pequeñas), problema 1

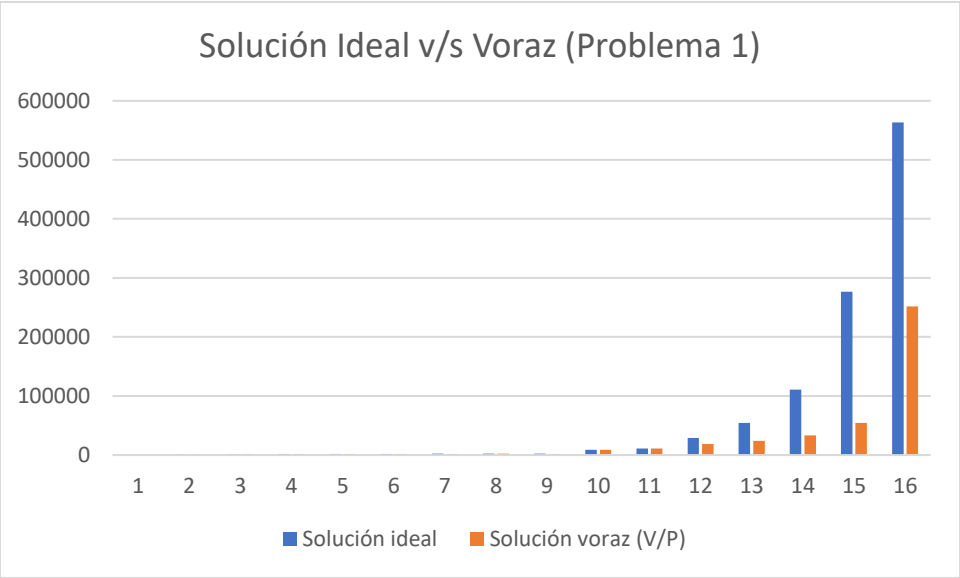

Ilustración 6: Solución ideal vs solución voraz (vista completa), problema 1

En el primero de los anteriores gráficos (Ilustración 5) podemos notar que con entradas pequeñas la solución obtenida mediante el enfoque voraz no difiere mucho de la solución ideal, incluso en el punto 8 de este grafico podemos notar que la solución voraz y la ideal son iguales. A su vez en el segundo grafico(Ilustración 6) podemos notar como a medida que el tamaño de la entrada crece, la diferencia entre la solución obtenida y la solución ideal es cada vez más grande, esto permite percibir que la calidad de la solución obtenida mediante un algoritmo voraz depende del problema, los datos y los criterios utilizados para considerar los candidatos más prometedores.

Teniendo en cuenta esta incerteza sobre la calidad de la respuesta entregada por el algoritmo voraz, la gran velocidad de respuesta y la simpleza del programa, se puede considerar ordenar de más de una manera los candidatos y entregar la mejor respuesta obtenida, por ejemplo: primero ordenar los candidatos de manera decreciente con respecto a su relación valor-ponderación y buscar la solución, luego ordenar los candidatos de manera creciente según su ponderación, obtener una solución, compararla la previamente obtenida y conservar la mejor. En este caso lo anterior podría ser una buena solución, ya que no existen muchas maneras diferentes de ordenar los candidatos y la respuesta es entregada de manera inmediata, pero quizás no sería una buena mejora para un algoritmo que demore mucho tiempo, debido a que aumentaría el tiempo de ejecución.

3.2. Análisis solución problema 2

Al igual que en el problema 1, en la Ilustración 4 se puede apreciar que el bajo tiempo de ejecución es el punto destacable de esta implementación utilizando un algoritmo voraz, siendo la obtención de una solución en casi la totalidad de los experimentos realizados de manera inmediata, a excepción de la entrada de 10000 elementos que se demoró 6 segundos, pero teniendo en consideración la gran cantidad de información que implica una entrada de se ese tamaño, 6 segundos sigue siendo un muy buen tiempo de respuesta. Además, fue posible comprobar la semejanza entre la curva del tiempo de ejecución teórico (Ilustración 3) y la curva del tiempo de ejecución real (Ilustración 4), por lo que el orden de complejidad represente de buena manera el programa construido.

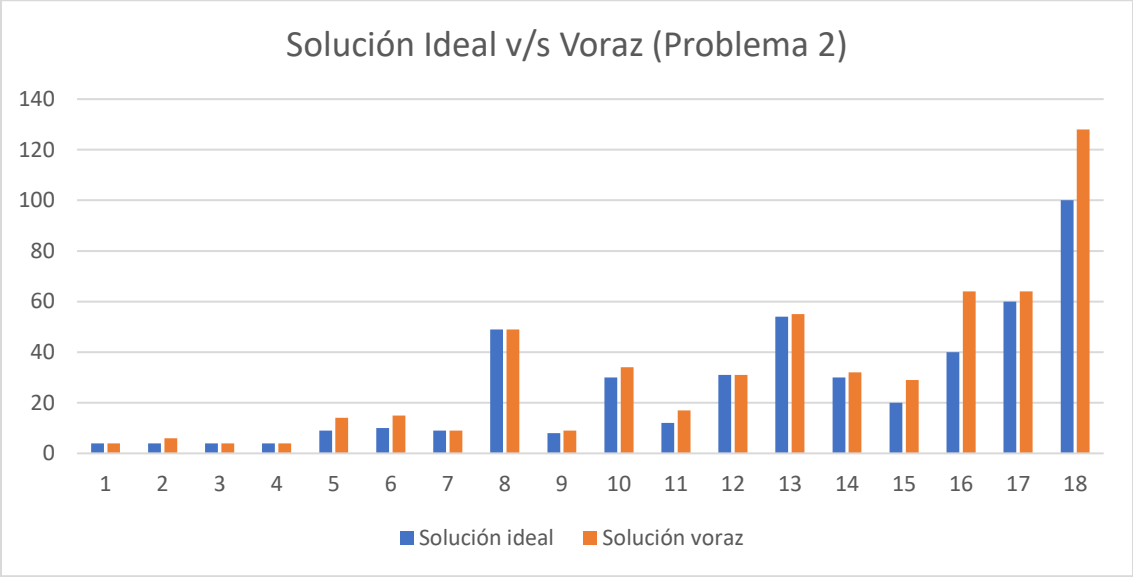

Ilustración 7: Solución ideal vs solución voraz (problema 2)

A su vez, en la Ilustración 7 podemos ver que en este caso la diferencia entre la solución ideal y la obtenida crece a medida que aumenta el tamaño de la entrada. La diferencia que existe entre la solución ideal y la obtenida se puede deber a que, a diferencia del primer problema donde al ordenar candidatos de manera decreciente según su valor nos puede asegurar que intentaremos incluir los elementos que posean mayor valor u ordenando de manera creciente según su ponderación nos permite ingresar una mayor cantidad de elementos, en el caso del problema 2, a pesar de que se decidió ordenar las tareas de manera creciente con respecto a la cantidad de conflictos que poseían, esto no nos asegura que el candidato más prometedor no posea conflicto con el segundo candidato más prometedor, haciendo esto que se tenga que incrementar aumentar la cantidad de semanas mínimas en el segundo ciclo.

Una manera de mejorar la calidad de las respuestas, de igual manera que para el primer problema, sería intentar ordenar los candidatos según diferentes criterios y conservar la mejor solución

4. CONCLUSIÓN

Si consideramos la situación real en la que se encontraba cada uno de los problemas analizados en este trabajo, se pudo llegar a la conclusión de que un algoritmo voraz por si sola, quizás no sería la mejor opción al momento de analizar una gran problemática en la cual es importante tener una solución precisa, a pesar de esto se logró comprobar que el comportamiento de la curva que representa el tiempo de ejecución real frente a diferentes tamaños de entrada se asemejaba bastante a la curva de la complejidad algorítmica obtenida del tiempo de ejecución teórico, además que se consiguió conocer las ventajas y limitaciones de algoritmos con enfoque voraz de manera aislada y las que posee frente a algoritmos de enumeración exhaustiva. Como principal aprendizaje se puede destacar la vital importancia de considerar la precisión necesaria de la respuesta, el tiempo empleado en obtener esta y la complejidad de implementar el programa al momento de elegir el enfoque más adecuado para enfrentar la problemática. Todo lo aprendido y obtenido en la realización de este laboratorio, será beneficioso para la realización de futuros laboratorios o implementaciones a desarrollar.

5. REFERENCIAS

1. Abad, M. (2008). Algoritmos voraces. (Recuperado el 13/06/2021) https://www.cs.upc.edu/~mabad/ADA/curso0708/GREEDY.pdf

2. GeeksforGeeks(2021). Merge Sort (Recuperado el 06/06/2021) https://www.geeksforgeeks.org/merge-sort/