udp Escuela de Informática y Telecomunicaciones

Facultad de Ingeniería y Ciencias
Escuela de Informática y Telecomunicaciones

Algoritmos Exactos y Metaheurísticas
Tarea 01

Jazmín Almendra Jassive Cuitiño Mendoza
Felipe Nicolás Gutierrez Lazo
jazmin.cuitino@mail.udp.cl
felipe.gutierrez@mail.com

20 de abril de 2024

Índice general

1. Introducción

En el amplio mundo de la toma de decisiones, en ambientes como los empresariales, políticos, educacionales, legales, entre otros, es que se hace cada vez mas importante tomar las decisiones correctas, teniendo siempre como objetivo buscar los óptimos para los distintos problemas a los nos podemos enfrentar. Sin ir más lejos, por ejemplo en el ambiente empresarial a menudo se tienen situaciones en las que se debe asignar recursos limitados (como tiempo, dinero, personal) a distintos proyectos, cada uno con un determinado conjunto de tareas. En estos casos el objetivo es aumentar lo mas posible las ganancias sin perder de vista la principal restricción: no superar los recursos disponibles.

Es entonces en casos reales como estos donde cobran una vital importancia los **algoritmos de *backtracking***, que exploran exhaustivamente todas las posibles soluciones para encontrar un óptimo global. El problema de estos algoritmos, dado su complejidad y costo computacional, es que no siempre resultan ser los más eficientes, tardando en muchos casos más tiempo del que se puede destinar a este tipo de análisis. Es por esto que existen distintas técnicas y variaciones que hacen de este algoritmo uno mas robusto, rápido y certero. En particular, el ***Minimal Forward Checking (MFC)*** es una de las variante más utilizadas del backtraking.

Ahora bien, ya se sabe que backtraking y en particular el MFC van a iterar sobre todas las posibles soluciones, almacenando la mejor de ellas (la que tenga mayor beneficio y menor costo, aunque depende del contexto particular en que se aplique el algoritmo cual será la función objetivo del problema), y también se sabe que el vector de entrada (input) de este algoritmo no viene ordenado de una manera en particular, pero: ¿Que pasaría si logramos ordenar el vector de entrada de alguna forma conveniente, de tal modo que el MFC llegue a una solución óptima más rápidamente? Aquí es donde entra en juego el **K-Means**, un algoritmo de agrupamiento que busca patrones en grandes conjuntos de datos. Al aplicar K-Means al MFC se puede optimizar la búsqueda, reduciendo el espacio de búsqueda y acelerando la convergencia hacia la mejor solución.

El presente documento presenta al lector una detallada comparación entre el funcionamiento de un MFC convencional versus una versión de MFC combinada con K-Means, exponiendo y comentando los códigos diseñados en cada situación. Se realizan distintas pruebas aplicadas a un problema estándar, comparando los tiempos de ejecución, soluciones encontradas y valor de la función objetivo. Finalmente, se ofrece un análisis de estos resultados que permite evidenciar de manera clara el aporte de K-Means al algoritmo de backtraking diseñado.

2. Descripción Actividades

Esta actividad está orientada a la implementación de algoritmos de búsqueda completa para problemas de optimización. En este caso, se trata de un problema ***COP (Constraint Optimization Problems)***. Estos problemas cuentan con variables y restricciones que se deben cumplir para buscar el mínimo/máximo valor para la función objetivo. Esta actividad consta de maximizar la ganancia de una empresa en base a la realización de proyectos, donde cada proyecto cuenta con tareas. Estas tareas tienen un costo asociado. Los proyectos pueden tener tareas en común, restringiendo así la realización de estas a sólo una vez; es decir, si la tarea 1 es compartida por el proyecto 5 y el proyecto 18, solo debe ser realizada una vez por alguno de estos dos. Finalmente, existe un presupuesto que permite costear dichas tareas.

Para resolver esta problemática, se hace uso del algoritmo **Minimal Forward Checking**. Este algoritmo pertenece a una técnica llamada *Look-Ahead*, que explora las siguientes posibles soluciones del problema. Este algoritmo busca alguna solución que cumpla con todas sus restricciones, y la agrega a la posible mejor solución. Si en algún momento, alguna solución factible no cumple con las restricciones, el algoritmo realiza un *backtracking*, paso que permite retroceder y buscar nuevas posibles soluciones factibles.

Finalmente, luego de realizar este procedimiento, se implementa un algoritmo de *clusterización* (agrupación) para ver el impacto que tiene sobre el Minimal Forward Checking.

3. Desarrollo actividades

Uno de los grandes desafíos a la hora de enfrentar un problema de este tipo es modelarlo correctamente, pues un error en esta actividad puede ser catastrófico para el desarrollo del ejercicio y la correcta ubicación de las mejores soluciones. Teniendo en cuenta esto es que se plantea el problema de la optimización de la siguiente manera:

3.1. Ítem 01

Inicialmente, se genera el modelamiento matemático del problema. Se define de la siguiente manera:

- Índices
  - I = Proyectos, $i \in \{1, ..., M\}$.
  - J = Tareas, $j \in \{1, ..., N\}$.
- Parámetros
  - G_i = Ganancia del proyecto i, $\forall i \in I$.
  - C_j = Costo de la tarea j, $\forall j \in J$.
  - B = Presupuesto.
  - a_{ij} = Necesidad de la tarea j para el proyecto i, $\forall i, j \in I, J$.
- Variables
  - $X_i = \begin{cases} 1 & \text{Si el proyecto } i \text{ se realiza} \\ 0 & \text{e.o.c} \end{cases}$
  - $Y_j = \begin{cases} 1 & \text{Si la tarea } j \text{ se realiza} \\ 0 & \text{e.o.c} \end{cases}$
- Función objetivo
  - $\max \sum_{i=1}^{M} X_i \cdot G_i$
- Restricciones
  - No superar el presupuesto.
    - $\sum_{j=1}^{N} Y_j \cdot C_j \leqslant B$
  - Proyecto completado.
    - $\sum_{j=1}^{N} a_{ij} \cdot Y_j \geqslant \sum_{j=1}^{N} a_{ij} \cdot X_i, \forall i \in I$

3.2. Ítem 02

Una vez planteado el problema, el equipo se dispone a programar un MFC que se ajuste de buena manera al planteamiento diseñado. Para esto se elabora un código en Python, ya que tiene bibliotecas útiles que son importantes a lo largo del desarrollo.

El objetivo del algoritmo elaborado es recibir los distintos parametros contenidos en los archivos .txt, de tal forma que al iniciar el algoritmo se cuenta con los siguientes inputs:

- **Cantidad de proyectos (m):** Es la cantidad total de proyectos que contiene el archivo en cuestión. Esta cantidad varía dependiendo del archivo, y para efectos del algoritmo, la variable que contiene este valor es: ***projects.***
- **Cantidad de tareas (n):** Esta variable indica la cantidad total de tareas, cada proyecto para completarse debe realizar una cantidad de tareas especificas que no es igual para todos los proyectos. La variable que contiene esta información es: ***tasks.***
- **Budget (B):** Es el presupuesto total, la suma de los costos totales de las tareas pertenecientes a todos los proyectos elegidos no debe superar este valor. La variable en cuestión es: ***budget.***
- **Ganancia de cada uno de los proyectos:** Es un vector que contiene la ganancia (en unidades) de cada proyecto una vez que es seleccionado como parte de la solución final (sus tareas son todas realizadas). El vector que almacena esta información es: ***profits.***
- **Costo de cada tarea:** Es otro vector que va a almacenar el costo asociado a realizar cada una de las tareas. El nombre del vector es: ***costs.***
- **Tareas asociadas a cada proyecto:** Es una matriz de tamaño **mxn** que contiene que tarea esta asociada a cada proyecto, es decir, en la fila del proyecto **i**, cada **j** seteada en **1** indicara que para realizar ese proyecto se debe hacer esa tarea.

Profundizar en el código e ir a un aspecto mas técnico, se presentan a continuacion las dos principales funciones del algoritmo MFC:

Listing 3.1: Función que realiza el MFC

```
def check_constraints(i, proy_tasks, completed_tasks, quote, costs):
    required_tasks = {j for j, assigned in enumerate(proy_tasks[i]) if
       assigned == 1 and j not in completed_tasks}
    total_project_cost = sum(costs[j] for j in required_tasks)

    if quote >= total_project_cost:
        return True, total_project_cost, required_tasks
    return False, 0, set()

def mfc(projects, tasks, profits, costs, proy_tasks, actual_tasks,
   actual_projects, quote, times, profits_records, best_sol, best_profit,
   best_cost, start_time, output_file):
    while time.time() - start_time <= 30 * 60:  # Limit execution to 30
       minutes
        for i in range(projects):
            if i in actual_projects:
                continue
            is_viable, total_project_cost, p_tasks = check_constraints(i,
               proy_tasks, actual_tasks, quote, costs)
            if is_viable:
                new_quote = quote - total_project_cost
                latest_actual_projects = actual_projects | {i}
                latest_actual_tasks = actual_tasks | p_tasks
                profit = sum(profits[j] for j in latest_actual_projects)
                cost = sum(costs[j] for j in latest_actual_tasks)

                if profit > best_profit or (profit == best_profit and cost
                    < best_cost):
                    current_time = time.time() - start_time
                    best_profit = profit
                    best_cost = cost
                    best_sol = latest_actual_projects
                    with open(output_file, 'a') as file:
                        file.write(f"{current_time}, {best_profit}\n")

                # Recursive call
                best_sol, best_profit, best_cost = mfc(projects, tasks,
                   profits, costs, proy_tasks, latest_actual_tasks,
                   latest_actual_projects, new_quote, times,
                   profits_records, best_sol, best_profit, best_cost,
                   start_time, output_file)

        # If no new projects can be added, break the loop
        break

    return best_sol, best_profit, best_cost
```

En el código precedente se observan 2 funciones fundamentales para el correcto funcionamiento del algoritmo:

1. **mfc:** Esta función es la que contiene toda la lógica del algoritmo. El primer punto a destacar es que se ejecutara durante un periodo de tiempo especifico, en este ejemplo, esta configurado para correr durante 30 minutos. Una vez dentro del ciclo while, se itera sobre el largo de los proyectos, identificando que a los proyectos que no son parte de la solución actual. Cuando se encuentra un proyecto que no pertenece

al conjunto de soluciones, entonces se envía el índice de este proyecto junto con la información necesaria tal que la función ***check_constraints*** es capaz de verificar si la solución actual (incluyendo el proyecto sobre el que se esta iterando) cumple con la restricciones de budget; en caso de cumplir, retorna *True*, además del costo total de las tareas pertenecientes a ese proyecto y finalmente las tareas de ese proyecto; en caso contrario retornara un *False*. En caso de que este booleano sea true, se verifica si la ganancia de esta nueva solución es mejor que la mejor ganancia hasta ese momento, en caso de ser así, se hacen las asignaciones necesarias y se almacena en un txt el tiempo que ha transcurrido hasta encontrar la nueva solución y la ganancia total de la misma, para luego continuar con la recursividad.

2. **Check_constraints:** En resumidas cuentas, esta función se encarga de verificar que el costo de las tareas correspondientes al conjunto de proyectos seleccionados mas los costos de las tareas del proyecto que se esta analizando no sobrepase el budget (presupuesto).

Ahora que se entiende el funcionamiento del algoritmo, se ejecuta el mismo durante media hora, obteniendo los siguientes resultados:

```
0.0, 319
0.0, 359
0.0, 809
0.0, 1065
0.0, 1089
0.0, 1449
0.0, 1751
0.0, 2209
0.0, 2408
0.0, 2754
0.0, 2786
0.0, 2966
0.0, 3361
0.0, 3436
0.015618085861206055, 3503
0.015618085861206055, 3535
0.015618085861206055, 3570
0.015618085861206055, 3750
0.015618085861206055, 3826
0.03227639198303223, 3853
0.039173126220703125, 3894
0.10134148597717285, 4076
0.3267350196838379, 4098
1.1855967044830322, 4116
2.842454195022583, 4193
2.844003915786743, 4269
2.85089111328125, 4375
2.8527028560638428, 4451
12.613054275512695, 4475
40.99073576927185, 4527
123.70766639709473, 4652
170.21155095100403, 4694
170.22939276695251, 4726
170.23170685768127, 5121
```

Resultado MFC archivo 1

Se aprecia entonces que el itera en el ciclo, recorriendo todas las posibles soluciones (las que alcanza a recorrer dado el breve tiempo que se ejecuta), anotando en el archivo cada vez que se encuentra una solución mejor a la que ya había. Este mismo proceso se repite para los archivos 2 y 3, los resultados son expuestos en la sección de análisis.

3.3. Ítem 03

Luego de realizar lo solicitado en la actividad anterior, es decir, ejecutar el algoritmo MFC recursivo en los distintos archivos es que se aprecia que el algoritmo tarda demasiado en mejorar las soluciones y las ganancias no incrementan significativamente, esta situación encuentra su fundamento en que este algoritmo itera sobre las soluciones sin ningún tipo de ordenamiento previo. Esto hace que sea similar a un algoritmo de fuerza bruta, en que iterara sobre los proyectos sin reparar en las características de los mismo. Para mitigar esta premisa, es que se hace uso de **k-means** previo a la ejecucion del MFC, de tal manera de ordenar convenientemente el orden en que se iterara sobre los proyectos.

Para definir entonces en que orden ingresaran los proyectos al MFC es que se definen dos métricas distintas que posteriormente son comparadas entre si:

1. La primera iteración con K-Means *(mfc+kmeans1)* tiene las siguientes condiciones:
    - Agrupa por la cantidad de tareas de cada proyecto. Es decir, si los proyectos tienen una cantidad similar de tareas por realizar, se agrupan en el mismo clúster.
    - Dentro del clúster, se ordenan los proyectos por la razón costo/ganancia que tengan. Si un proyecto entrega una gran ganancia, a poco costo, se posiciona de los primeros para ser considerado en la solución factible.
    - Finalmente, para decidir sobre que clúster iterar primero, se calcula el promedio de la cantidad de tareas de cada uno. A mayor cantidad de tareas promedio, más factible es para ser iterado primero que los demás.

    A continuación, se exponen las dos funciones que hacen esto posible:

Listing 3.2: Funciones k-means similitud cantidad de tareas

```
        def exec_kmeans_task_similarity(proy_tasks):
    # proy_tasks ya es una lista de listas con vectores binarios de
       tareas requeridas por cada proyecto
    kmeans = KMeans(n_clusters=4, random_state=0).fit(proy_tasks)
    clusters = kmeans.labels_
    return clusters

def sort_projects_by_cluster(profits, clusters):
    cluster_dict = {}
    for i, cluster in enumerate(clusters):
        if cluster not in cluster_dict:
            cluster_dict[cluster] = []
        cluster_dict[cluster].append((profits[i], i))

    # Ordenar cada lista de proyectos en cluster_dict por ganancia
       decreciente
    for cluster in cluster_dict:
        cluster_dict[cluster].sort(reverse=True, key=lambda x: x[0])

    # Ordenar los clusters por la ganancia promedio de sus proyectos
    sorted_clusters = sorted(cluster_dict.items(), key=lambda x: sum(
       proj[0] for proj in x[1])/len(x[1]), reverse=True)

    # Aplanar la lista de proyectos ordenados
    sorted_project_indices = [proj[1] for cluster in sorted_clusters
       for proj in cluster[1]]
    return sorted_project_indices
```

2. La segunda iteración con K-Means *(mfc+kmeans2)* tiene las siguientes condiciones:

   - Agrupa por la similitud entre las tareas de cada proyecto. Es decir, si las tareas de cada proyecto son parecidas (pertenecen al vecindario), se agrupan.
   - Para ordenar el clúster se utiliza el criterio de mayor ganancia, de manera descendente. Es decir, si un proyecto aporta una ganancia muy alta, se posiciona de los primeros para ser iterado.
   - Finalmente, el criterio para poder iterar sobre los clústers es seleccionar el grupo que en promedio tenga una mayor ganancia.

   Las funciones asociadas a este k-means son las siguientes:

Listing 3.3: Funciones k-means similitud de tareas

```
        def exec_kmeans_task_similarity(proy_tasks):
    # proy_tasks ya es una lista de listas con vectores binarios de
       tareas requeridas por cada proyecto
    kmeans = KMeans(n_clusters=4, random_state=0).fit(proy_tasks)
    clusters = kmeans.labels_
    return clusters

def sort_projects_by_cluster(profits, clusters):
    cluster_dict = {}
    for i, cluster in enumerate(clusters):
        if cluster not in cluster_dict:
            cluster_dict[cluster] = []
        cluster_dict[cluster].append((profits[i], i))

    # Ordenar cada lista de proyectos en cluster_dict por ganancia
       decreciente
    for cluster in cluster_dict:
        cluster_dict[cluster].sort(reverse=True, key=lambda x: x[0])

    # Ordenar los clusters por la ganancia promedio de sus proyectos
    sorted_clusters = sorted(cluster_dict.items(), key=lambda x: sum(
       proj[0] for proj in x[1])/len(x[1]), reverse=True)

    # Aplanar la lista de proyectos ordenados
    sorted_project_indices = [proj[1] for cluster in sorted_clusters
       for proj in cluster[1]]
    return sorted_project_indices
```

4. Análisis y Resultados

Para poder realizar un análisis de los resultados, se adjunta un gráfico que contiene el comportamiento de cada archivo. En el gráfico, se refleja la variación del valor de la función objetivo con respecto al tiempo de CPU. A continuación, se señala la figura:

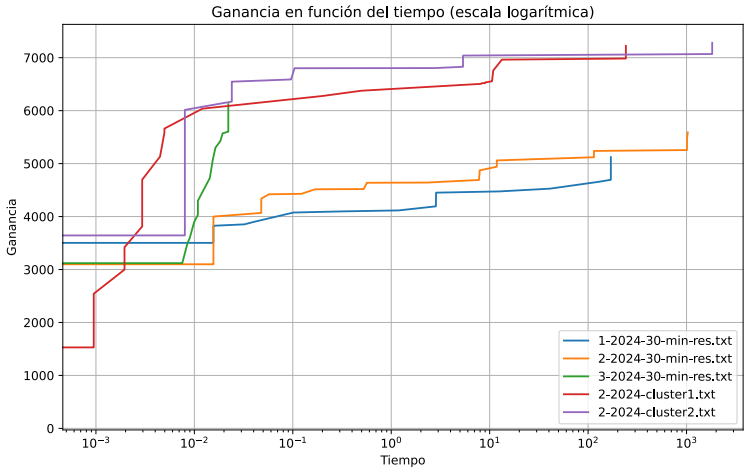

Figura 4.1: Gráfico de comportamiento MFC sobre archivos.

Como se observa en la figura 4.1, el MFC permite que, a medida que pasa el tiempo, se encuentren mejores soluciones. Por ejemplo, en el archivo *1-2024-30-min-res.txt*, la ganancia aumenta de una manera considerablemente alta en muy poco tiempo, para luego tener un estancamiento. Este estancamiento se produce en todas las ejecuciones que se realizan, y se debe al funcionamiento del algoritmo. MFC es un algoritmo de búsqueda completa, por lo que itera las veces que sea necesario para encontrar todas las posibles soluciones.

Por ejemplo, en el archivo *1-2024-30-min-res.txt*, la cantidad de proyectos corresponde a $m = 200$, mientras que las tareas corresponden a $n = 185$. Al verificar las restricciones que cumplen, el algoritmo recorre una matriz de $200 \cdot 300$ por cada nodo que genera. Además, el algoritmo es recursivo, por lo que el orden es, aproximadamente, de $O(2^n)$, siendo n el tamaño de cada entrada.

De esta manera, al iterar durante un tiempo, genera un árbol de recursión tan grande que para abarcarlo completamente se debe dejar el equipo funcionando por días, inclusive semanas.

	1-2024	2-2024	3-2024	K-Means1	K-Means2
Ganancia máxima	5121	5588	6123	7223	7281
Tiempo (s)	170.2317	1029.8092	0.0221	242.1194	1822.6440

Tabla 4.1: Comparación de ganancia máxima v/s tiempo de CPU en cada ejecución.

Como se puede observar en la tabla anterior, el archivo *3-2024* encuentra una solución bastante buena en un tiempo realmente pequeño. Esto se debe a que el archivo cuenta con un total de proyectos y tareas más bajo, generando un espacio de búsqueda mucho más acotado. De esta manera, genera la explotación mucho más cerca de la solución óptima en el espacio de búsqueda definido.

Con algoritmos de este tipo (búsqueda completa) se puede asegurar llegar a un óptimo global, por la característica que presentan para explorar todas las soluciones factibles dentro del espacio de búsqueda. Sin embargo, no son la mejor opción cuando se trabajan con grandes volúmenes de datos, ya que su orden de ejecución crece drásticamente. Con esto, se tienen dos puntos de vista:

- Si se busca obtener una solución en poco tiempo, es necesario utilizar un algoritmo de búsqueda incompleta, como una metaheurística. Sin embargo, al utilizar técnicas de este tipo se corre el riesgo de no llegar al óptimo global, ya que todo depende de las heurísticas empleadas, y de como se elijan las regiones prometedoras para explotar.

- Si se busca obtener la solución óptima a toda costa, se puede implementar este algoritmo, pero requiere de mucho más tiempo para encontrarlo. También se pueden agregar técnicas para hacer saltos más eficientes, como por ejemplo, técnicas de poda del árbol, como podas por factibilidad, o poda por optimalidad.

Además, se puede observar que aplicar técnicas de agrupamiento, como el *K-Means*, permite encontrar mejores soluciones en la misma cantidad de tiempo de ejecución. Esto se debe a que al agrupar los proyectos por ciertas condiciones/criterios, el espacio de búsqueda se ve segmentado, por ende, cada búsqueda abarca un dominio considerablemente más acotado que el espacio completo de soluciones factibles.

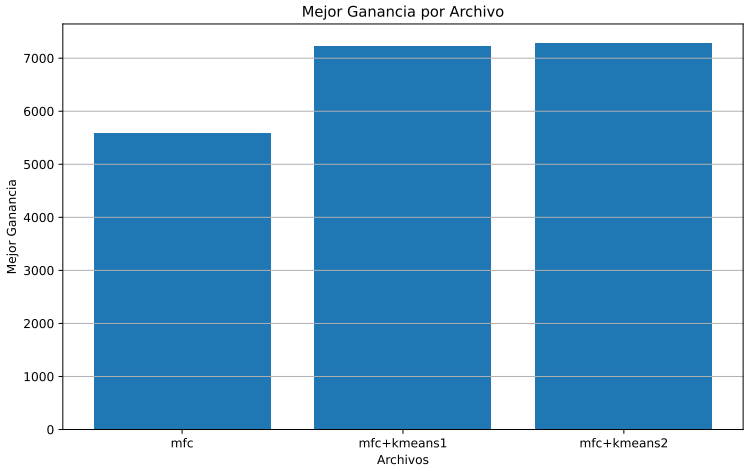

Figura 4.2: Gráfico de comportamiento MFC sobre archivos.

Como se observa en la figura 4.2, las dos iteraciones con *K-Means* logran encontrar soluciones que maximizan más la ganancia que solo aplicando el MFC. Esto se debe a que, como se menciona anteriormente, el espacio de búsqueda se ve considerablemente disminuido cuando se segmenta. Ahora, las elección de la heurística de selección de variable también tiene una alta influencia. En este caso, las heurísticas elegidas (detalladas en la subsección 3.3) generan mejores resultados sobre el archivo *2-2024*, el cual contiene el espacio de búsqueda mayor de los 3. Además, complementando el gráfico con la tabla, se observa que las soluciones obtenidas con las heurísticas elegidas mejoran en un 17 % y un 18 % respectivamente la solución.

Si bien es cierto que las soluciones encontradas aplicando K-Means mejoran con respecto a solamente el uso de *minimal forward checking*, existen factores que podrían generar una peor respuesta del algoritmo:

1. La semilla de generación para los centroides. Esta semilla es un factor que afecta en el resultado entregado. En este caso, la semilla utilizada es equivalente a 0, y todas las ejecuciones son realizadas con esta misma. El cambiar el valor de la semilla influye directamente en la generación de los centroides iniciales, cambiando así el resultado final de la ejecución.

2. La heurística de selección de variable. Si bien es cierto que las heurísticas seleccionadas funcionan de manera correcta, pueden haber heurísticas que empeoren la solución. Esto también tiene una implicancia grande en la solución encontrada.

5. Conclusiones

Luego de comprender el funcionamiento del algoritmo ***Minimal Forward Checking*** de manera teórica, se implementa de manera práctica en la resolución del problema propuesto. Inicialmente, se generaron muchas incertidumbres con respecto al código, ya que no realizaba pasos fundamentales del algoritmo, como el **backtracking**, o el cumplimiento de las restricciones propuestas. Luego de consultas e investigación sobre como implementarlo, se logra llegar al funcionamiento correcto, obteniendo soluciones estimativas del óptimo global.

Algo importante de este trabajo es que se comprenden de una manera empírica las ventajas y desventajas que presentan las técnicas de búsqueda completa, y a su vez, la mejora que producen en ellas las heurísticas de selección de variables. Una ventaja de estas técnicas de búsqueda completa es asegurar una solución óptima. Sin embargo, al tener un espacio de búsqueda tan grande, el tiempo de ejecución aumenta exponencialmente, incluso agotando los recursos de la máquina donde se realiza la ejecución. No obstante, el agregar **K-Means** al algoritmo principal permite reducir el tamaño del espacio de soluciones, por lo que la búsqueda se aproxima más a la solución óptima en un tiempo menor.

Sin embargo, por los problemas presentados en un comienzo, y el tiempo presentado para realizar pruebas, es que queda como tarea pendiente el poder realizar distintas ejecuciones modificando los parámetros de K-Means, como su semilla de generación, e implementando distintas heurísticas, viendo la influencia que tienen en los resultados obtenidos.

6. Anexos

Repositorio de la actividad: `https://github.com/iJass21/Algoritmos_exactos_y_metaheuristicas/`