Proyecto Final de DAA

Alfredo Nuño Oquendo, David Lezcano Becerra, Marlon Díaz Pérez

February 7, 2025

1 Definición del problema

Una empresa encargada de hacer desarrollo de software quiere ver como repartir tareas a sus trabajadores. Cada tarea tiene un tiempo estimado de realización, una ganancia y una fecha límite para entregarse. Los trabajadores tienen un salario por tarea que puede ser distinto para cada trabajador y una experiencia que influye en el tiempo que demoran en resolver la tarea. Además solo pueden resolver una tarea a la vez.

El objetivo es buscar la mejor forma de asignar las tareas de modo que se obtenga la ganancia máxima y se realicen todas la tareas. Si no es posible realizar todas las tareas, entonces la mayor cantidad.

2 Importancia del problema

Resolver este problema permite maximizar las ganancias totales de la empresa. Asignar las tareas de una mejor manera, garantizando que se cumpla la mayor cantidad posible y dentro del tiempo establecido, lo que permite ganar y mantener la reputación de cualquier empresa. Además brinda un mejor uso de los recursos humanos, evitando la sobrecarga de algunos trabajadores y el subuso de otros.

3 Complejidad

Este problema es un caso de optimización combinatoria y, en general, es NP-hard. Los factores que contribuyen a la complejidad son:

• Número de variables: El número de posibles asignaciones de tareas a trabajadores crece exponencialmente con el número de tareas y trabajadores. Cada tarea puede asignarse a varios trabajadores, añadiendo más complejidad.

• Restricciones: Las restricciones (fechas límite, salarios variables) hacen que el espacio de búsqueda de soluciones sea más complejo de explorar.

4 Demostración de NP-Hard

Para demostrar que nuestro problema es NP-Hard, intentaremos reducir un problema que ya sabemos es NP-Hard al nuestro, en este caso nos apoyaremos en el problema de la mochila.

Definición del problema de la mochila: Dado un conjunto de n elementos, cada uno con un peso w_i y un valor v_i , y una capacidad máxima W , el objetivo es seleccionar un subconjunto de elementos tal que el valor total sea maximizado y la suma de los pesos no exceda W.
Llamemosle a nuestro problema: Problema de Asignación de Tareas
Definición: dado un conjunto de tareas T = t_1, t_2, ..., t_n donde cada tarea t_i tiene:
- Un tiempo estimado de realización p_i - Una ganancia g_i - Una fecha límite d_i - Un conjunto de tareas precedentes t_i - Un conjunto de trabajadores o_i - Una matriz A que representa el salario que cobra el trabajador i por realizar la tarea j - La experiencia de cada trabajador e_i que influye en el tiempo en que demora en realizar la tarea

El objetivo es asignar las tareas a los trabajadores de manera que se maximice la ganancia total, respetando las restricciones de precedencia y las capacidades de los trabajadores.

Dado una instancia del problema de la mochila construiremos una instancia del problema de asigar tareas:
Primero diremos que cada elemento en el problema de la mochila corresponde a una tarea El peso de cada elemento corresponde al tiempo estimado de realización para cada tarea. El peso que puede soportar la mochila corresponde a la fecha límite para las tareas, es decir que todas las tareas tendrán la misma fecha límite. El valor de cada objeto corresponde a la ganancia obtenida por la tarea correspondiente a ese objeto. Las tareas no dependerán de ninguna tarea y solo habrá un trabajador el cual tendrá el valor 1 en su salario por tarea y experiencia.

Haciendo esta construcción podemos ver que si encontramos una solución óptima al problema de asiganar tareas habremos encontrado una solución al problema de la mochila, dado que bastaría tomar el objeto correspondiente a cada tarea realizada.
Notemos que la transformación del problema de la mochila al problema de asignar tareas puede hacerse en tiempo lineal, ya que solo sería crear valores equivalentes de un problema a otro, luego como sabemos que el problema de la mochila es NP-Hard entonces el problema de asignar tareas también.

En esta demostración se redujo el problema de la mochila al de asignación de tareas con un solo trabajador, pero se puede ver que para más trabajadores el problema sigue siendo NP-Hard, ya que podemos reducir el problema de asiganrle tareas de forma óptima a un trabajador a asignárselas a muchos. Pues si asiganrle a más de un trabajador se pudiera resolver en tiempo polinomial podemos tomar una instancia del problema de asignarle tareas a un trabajador y crear nuevas tareas y otro trabajador por cada tarea

creada, la tarea tendrá ganancia cero para todos lo trabajadores excepto para el que fue creado para ella, podemos ver que la forma óptima de resolver esta instancia es asignar cada tarea creada a su trabajador correspondiente y al primer trabajador asiganarle de forma óptima las otras tareas, por lo que si este problema se puede hacer en tiempo polinomial, asiganrle a un solo trabajador también y ya vimos que no.

5 Soluciones

Como ya vimos que el problema es NP-Hard sabemos que no existe un algoritmo eficiente para resolverlo. Una posible solución que cumpla con todas las restricciones del problema sería:
Modelar el problema como un grafo dirigido y acíclico (DAG)
Cada tarea es un nodo en el grafo.
Las dependencias entre tareas se representan como aristas dirigidas. Por ejemplo, si la tarea A debe realizarse antes que la tarea B, hay una arista de A a B.

Usar un ordenamiento topológico para procesar las tareas en un orden que respete las dependencias. Lo cual garantiza que una tarea solo se asigne después de que todas sus predecesoras hayan sido completadas.

Luego por cada tarea probamos asignársela a un trabajador y marcamos hasta que tiempo el mismo estará ocupado realizandola y pasamos a la siguiente tarea, luego probamos para otro trbajador y así hasta probar todas las posibles asignaciónes válidas, siendo n la cantidad de tareas y m la cantidad de trabajadores podemos ver que el costo de este algoritmo es $O(m^n)$, lo cual es extremadamente ineficiente.

Vamos a resolver una versión con menos restricciones de nuestro problema pero aún así interesante y con una dificultad alta.
Dado n tareas y n trabajadores, queremos ver la mejor forma repartir estas tareas entre los trabajadores, cada trabajador puede realizar a lo sumo una tarea y no hay dependencia entre las tareas.
Notemos que si modelamos el problema con un gafo bipartito donde los nodos de un subconjunto son los trabajadores y los del otro son las tareas, la idea sería obtener un matching perfecto de costo máximo. Además podemos observar que en este ejercicio estamos intentando maximizar la suma de las ganancias, que es los mismo que multiplicar las ganancias por -1 y minimizar esta suma.
Teniendo estas anotaciones podemos aplicar el método de optimización conocido como método húngaro para resolver nuestro problema.

El método consta de los siguientes pasos:
1- Dada la matriz de costos(ganancias multiplicadas por -1 en este caso) C, se construye la matriz de costo C' hallando el menor elemento por fila y restándoselo a cada elemento de esa fila, notemos que restarle este elemento a una fila es una tansformación equiva-

lente dado que afecta el valor del matching, pero no al matching, en este paso logramos que haya al menos un 0 por fila.

2- Se encuentra el menor elmento por columna y se resta a cada uno de los elementos de esa columna, al igual que el paso anterior es una transformación equivalente y garantiza que haya al menos un 0 por columna.

3- En este paso consideramos el grafo en el cual un subconunto son los trabajadores i y el otro las tareas j, y tendremos aristas (i, j) si en la casilla i,j hay un 0. Lo que queremos es verificar si para cada fila exite una columna con costo 0 que no ha sido asignada a otra fila, lo cual signidica que se podría asignar ese trabajador a esa tarea. Determinamos en este grafo un matching máximo, si la cardinalidad del matching es igual a la cantidad de vértices de un subconjunto, entonces encontramos una asignacion correcta ya que las filas tienen a lo menos una intersección con costo cero que no ha sido ocupada por otra fila, estamos en el óptimo. Termina el algoritmo. Además por el teorema de Koning sabemos que la cardinalidad del matching máximo en un grafo bipartito es igual al mínimo vertex cover, por lo que si el matching es menor es porque todavía quedan tareas o trabajadores sin asignar.

Si el matching fue menor al número de vértices entonces se busca el menor número de líneas necesarias a trazar en la matriz para cubrir todos los ceros(el minimo vertex cover), de los elementos que quedaron sin cubrir por una linea se busca el menor de ellos, sea m este elemento, y se le resta todos los elementos sin cubrir y se le suma a todos los elementos que fueron cubiertos por dos líneas. Notemos que esto es una transformación equivalente pues sea S, el costo de una asignación mínima en la matriz antes de este paso, en la nueva matriz el costo de esa asignación es $S - mk$ donde k es la difernecia entre la cardinalida del matching y la cantidad de nodos de un subcojunto(n en este caso). Veamos por qué: Al no poder encontar un matching perfecto se debe seleccionar un elemento de una fila no cubierta, que en la matriz nueva tiene su valor reducido en m, entonces si la diferencia es de k elementos el costo en la nueva matriz es $S - mk$, ahora si en el matching se seleccionó un elemento que es cubierto por dos líneas entoces ya no se puede seleccionar otro elemento de esa fila o columna, por lo que se elimina la posibilidad de seleccionar otro elemento en alguna de esas dos líneas, lo que obliga a seleccionar otro elemento no cubierto pero como al elemento cubierto por las dos líneas se le había aumentado su valor en m y el nuevo elemento seleccionadp tiene su valor disminuido en m, el valor resultante sigue siendo el mismo, no se afecta. Por tanto si el costo de una asignación en la primera matriz es S enla segunda el costo es $S - mk$, pero como mk es una constante podemos ver que minimizar la suma en la primera matriz es equivalente a minimizar la suma en la segunda matriz.
El objetivo de este paso es crear nuevos ceros, pero no cualquier cero, podemos ver que al menos se crea un nuevo cero que es independiente en fila y columna a los que ya habían anteriormente.
Después de hacer esto volvemos al paso 3 y calculamso nuevamente un matching máximo y así hasta encontrar uno que sea perfecto.

Pseudocodigo:

```
def algoritmo(matriz):\\
    matriz1 = encontar min por fila(matriz)----------------O(n^2)
    matriz1 = encontar min por columna(matriz1)------------O(n^2)
    while True:
        G = construir_grago(matriz1)-----------------------O(n^2)
        vetex_cover, matching_max  = Edmond-Karp(G)--------O(n^3)
        if vetex_cover == n:
            break
        matriz1=ajustar_matriz_sum_y_rest_m(matriz1, vertex_cover-O(n^2)
    return Calcular_ganancia(matriz, matching_max)
```

Notemos que el ciclo while a lo sumo se ejecuta n veces, dado que en cada paso se crea un nuevo cero independiente en fila y columna y esto a lo sumo puede ser n veces, por tanto la complejidad del algoritmo es $O(n^4)$.

Ejemplo de aplicación del algoritmo: supongamos que despues de restar el menor elemento por fila y columna en el paso 1 y 2 tenemos la siguiente matriz C'

```
        1 2 7 3 0
C' =    3 4 1 0 2
        2 3 6 0 0
        0 6 1 1 0
        2 0 0 4 5
```

Al construir el grafo obtendríamos el siguiente matching máximo: M = (1,5); (2,4); (4,1); (5,3)
Este no es perfecto, para cubrir todos los ceros basta con trazar una linea por la fila 4 y 5 y por la columnas 4 y 5
Analizamos la submatriz
1 2 7
3 4 1
2 3 6

Al buscar el mínimo el cual es 1, obtenemos la nueva matriz entoces:
0 1 6 3 0
2 3 0 0 2
1 2 5 0 0
0 6 1 2 1
2 0 0 5 6

Y en esta si obtenemos el matching M = (4,1); (5, 2); (2,3); (3, 4); (1, 5) el cual es perfecto y fue resultado de minimizar los costos.