Instituto Politécnico Nacional

ESCUELA SUPERIOR DE CÓMPUTO

PROYECTO. 'PLANIFICACIÓN DE TAREAS'

Análisis de algoritmos
Dr. Miriam Pescador Rojas.

3CM1:
Cano Rodríguez Adriana
Elizarrarás Ortiz Carlos Alán

3CM4:
Cuevas Naranjo Luis Alejandro

Julio 2020

ÍNDICE

I. Marco teórico

Un problema es un determinado asunto o una cuestión que requiere de una solución.

La planificación es el proceso y efecto de organizar con método y estructura los objetivos trazados en un tiempo y espacio.

Una tarea es un determinado trabajo o actividad que se lleva a cabo.

Si juntamos los 3 conceptos obtenemos el problema de la planifación de tareas o job-shop scheduling problem como se le conoce en inglés.

I-A. Job Shop Scheduling Problem

La planificación de tareas es un problema de optimización combinatoria. La función del scheduling es la asignación de recursos limitados a tareas a lo largo del tiempo y tiene como finalidad la optimización de uno o más objetivos.

El Job Shop puede ser un problema difícil, que de hecho, cae en la categoría de los problemas NP-difícil.

La complejidad NP-difícil es la característica que define a una clase de problemas que son, de manera informal, "al menos tan duro como los problemas más difíciles en NP" ya que se trata del conjunto de los problemas de decisión que contiene los problemas H tales que todo problema L en NP puede ser transformado polinomialmente en H.

Aunque no está claro quién debería llevarse el crédito por haber propuesto por primera vez el Job Shop Problem, se acepta que el libro "Industrial Scheduling" editado en 1964 por Muth y Thompson, constituye la base para la mayoría de las investigaciones que siguieron.

Las técnicas de resolución del Job Shop Problem se pueden dividir en dos categorías; las técnicas de optimización, que producen una solución globalmente óptima, pero requiere un tiempo computacional muy alto, y las técnicas de aproximación, que proporcionan una buena solución en un tiempo aceptable.

I-B. Problema a resolver: 'Flexible Job-Shop Scheduling Problem'

Como sabemos, el problema de planificación de tareas es una rama de la programación de producción y los problemas de optimización combinatoria. El JSSP clásico consiste en programar un conjunto de trabajos en un conjunto de máquinas bajo la restricción de que cada trabajo tiene un orden de procesamiento determinado.

El problema flexible de planificación de tareas (FJSSP) es una extensión del problema clásico JSSP en el que cada operación debe procesarse en una máquina elegida entre un conjunto de las disponibles.

El problema de planificar el trabajo en FJSSP es que puede estar compuesto de subproblemas: asignar la operación a la máquina y secuenciar las operaciones en las máquinas para minimizar los indicadores de rendimiento. Por lo tanto, la combinación de dos decisiones presenta una complejidad adicional.Siendo así que, el problema de FJSSP también es NP-difícil ya que es una extensión del problema de planificación de tareas clásico.

En [2] se expone que el tiempo de procesamiento de cada operación depende de la máquina. Denotamos con d_{ijk} como el tiempo de procesamiento de la operación O_{ij} cuando se ejecuta en la máquina M_k. No se permite la preferencia, es decir, cada operación debe completarse sin interrupción una vez iniciada. Además, Las máquinas no pueden realizar más de una operación a la vez. Todos los trabajos y máquinas están disponibles en el momento 0.

El problema es asignar cada operación a una máquina apropiada (problema de enrutamiento) y secuenciar las operaciones en las máquinas (problema de secuencia) para minimizar el *makespan*, es decir, el tiempo necesario para completar todos los trabajos, que se define como $MK = C_{máx}(S)$, donde S, es la secuencia con la solución óptima al FJSSP, donde cada elemento de S representa a la operación 0_{ijk} es decir la operación j del trabajo i en la máquina k. Cuando es implementada la aplicación, S luce como una arreglo bidimensional, donde cada subconjunto en él, es un triada de la forma (i, j, k) que representan los elementos ya mencionados.

I-C. Algoritmo para resolver el FSSP

Para dar solución al FJSSP, los integrantes del equipo decidimos crear un algoritmo, basado en las referencias revisadas y en la información obtenida de la clase de la Dra. Miriam Pescador. Se trata del Algorithm 1, que como entrada recibe un arreglo TOperaciones, con los tiempos de cada operación en cada máquina. Se trata de un arreglo bidimensional, donde cada subarreglo representa una operación, y donde cada elemento de esos subarreglos dicta el tiempo de la operación en las máquinas. Todas los subarreglos (operaciones) son de igual de tamaño, pues todas tienen la disponibilidad de correr en todas las máquinas. Por otro lado, está como entrada el arreglo Jobs, también es bidimensional, y donde cada subarreglo es un trabajo, y cada elemento de los trabajos, representan las operaciones que debe realizar, en orden.

Para la salida del algoritmo, tendremos un conjunto Solución S, que contendrá subarreglos, es una secuencia, la secuencia que debe seguirse para dar solución al problema. Cada subarreglo es una triada de la forma (i,j,k) donde i es el número de trabajo, j el número de la operación y k es la máquina que está ejecutando esa operación. Por otro lado, devolvemos el mayor tiempo de finalización, es decir el tiempo de finalización de la última operación de entre todas, esto para expresar que se trata del *makespan* mínimo para resolver todos los trabajos.

Antes de iniciar la asignación es necesario declarar un par de arreglos más y variables que serán útiles. El primero de los arreglos lo nombramos 'finalJobs', su tamaño es igual al número de trabajos que hay, y en cada posición irá almacenando el tiempo de finalización de la última operación completada de ese trabajo; es importante recalcar que la posición coincide con el número de trabajo, por ejemplo, en la posición 1 almacena el tiempo de finalización de las operaciones del trabajo 1. Seguido, creamos otro arreglo, que busca ir almacenando los tiempos de finalización pero ahora de cada máquina, y al igual que el anterior, las posiciones coinciden con el número de

Algorithm 1: Flexible Job-Shop Scheduling Problem.

Input: Conjunto de T operaciones, tiempos en cada máquina de cada operación. Conjunto Jobs, con las operaciones que hay en cada trabajo.

Output: Conjunto S, con la secuencia de operaciones que forman la solución. Conjunto M, con las secuencias de operaciones que se realizan por máquina, es un arreglo tridimensional.

1: Inicio:
2: finalJobs:[$O_{1]}$,O_2,...,O_j] *//Almacena el tiempo de finalización de la última operación en el trabajo j donde 1≤j≤ |Jobs|]*
3: finalMaq:[$O_{1]}$,O_2,...,O_m] *//Almacena el tiempo de finalización de la última operación en la máquina m donde 1≤m≤ |TOperaciones(i)|]*
4: *l*Jobs:[$O_{1]}$,O_2,...,O_j] *//Guarda la cantidad de operaciones que se van completando por trabajo.*
5: Continua=True *//Variable booleana para determinar el fin del algoritmo.*
6: **while** *Continua == True* **do**
7: **for** *i = 1 to |Jobs|* **do**
8: S = [] *//Subconjunto de S, en forma de triada.*
9: K = [] *//Subconjunto de máquina M, donde m⊆M.*
10: **if** *lJobslen(Jobs[i]* **then**
11: maqCandidatas = [O_1,...,$O_{|M|}$] *//|M|=Número de máquinas.*
12: numOperacion = Jobs[i][*l*Jobs[i] + 1]
13: numTrabajo = i
14: **for** *j = 1 to |M|* **do**
15: t = TOperaciones[numOperacion][j]
16: m = j *//Máquina con el tiempo t.*
17: **if** *finalMaq[m]≥finalJob[i]* **then**
18: maqCandidatas[m] = finalMaq[m] + t *//Máquina m es candidata.*
19: **else**
20: maqCandidatas[m] = Q *//Si m no es candidata, asignarle un valor Q muy grande para que sea descartada.*
21: tmin = min(maqCandidatas)
22: mE = maqCandidatas.index(t) *//índice de la máquina elegida con tmin.*
23: S = (numTrabajo, numOperación, mE)
24: S.append(s)
25: K = (numTrabajo, numOperacion)
26: M[m].append(K)
27: finalMaq[mE] = tmin
28: finalJob[i] = tmin
29: *l*Jobs[i]+ = 1
30: **if** *lJobs == TamJobs* **then**
31: Continua = False
32: **return** S and Max(finalJob)

máquina; habrá tantos elementos como máquinas, recibe el nombre de 'finalMaq'.

Un tercer arreglo, 'lJobs', es creado para ir almacenando la cantidad de operaciones que se van completando por trabajo, cada posición del arreglo coincide con el número de trabajo. Los 3 arreglos declarados anteriormente incializan cada una de sus posiciones con el valor 0 e irán creciendo conforme vaya avanzando. Un último arreglo se llama 'TamJobs', que guarda el tamaño (número de operaciones) de cada trabajo, la posición de cada valor corresponde al número de trabajo. Por último, se declara una variable 'Continua' con el valor booleano True, la cual permite que se sigan analizando las operaciones mientras se mantenga verdadera.

Comienza el algoritmo, en el primer trabajo y hasta el último de ellos, cada que se trate de un trabajo diferente al anterior creamos dos arreglos más, s y k, el primero de ellos será subarreglo de S y el segundo de M[m], donde m será la máquina elegida para la operación. En el 'if' de la línea 11, verifica que las operaciones completadas en el trabajo i sean menores a la cantidad de operaciones totales de ese mismo trabajo, de ser inválida, se entiende que el trabajo i ya completó todas sus operaciones, de lo contrario, creamos un arreglo con los tiempos de finalización de las máquinas candidatas para asumir la operación. En este subarreglo, 'maqCandidatas', habrá tantos elementos como máquinas haya, cada elemento se inicializará con 0. Para este punto, vamos en la línea 15 del Algorithm 1, donde con ayuda del arreglo TOperaciones, vamos a explorar los tiempos que ofrecen cada máquina para la operación en turno, declarando como t el tiempo en la máquina m. Ahora bien, aquí se viene la validación de unas de las condiciones en el FJSSP, que es evitar que haya un traslape, dos o más operaciones de un mismo trabajo no pueden llevarse a cabo al mismo tiempo. Para ello, en la línea 18 se valida que el tiempo de finalización de la máquina m debe ser mayor o igual al tiempo de finalización de la última operación completada en ese mismo trabajo, si es correcta, la máquina m resulta ser candidata y debe ser añadido su tiempo de finalización para la operación en turno al arreglo de maqCandidatas, eso sí, en la posición que le corresponda de acuerdo al número de la máquina m; pero, de no resultar candidata, el tiempo de finalización de la máquina m también se agrega al arreglo de maqCandidatas, pero con un tiempo de finalización arbitrario y excesivamente grande, esto para que sea descartada en el siguiente paso.

Una vez obtenidas las máquinas candidatas, se deberá cumplir con otras de las condiciones propuestas por el FJSSP, que es la de escoger la máquina que acabe antes, pero sin traslapes. Como la condición de los traslapes fue cumplida con el arreglo de maqCandidatas, solo resta escoger de entre ese arreglo el elemento menor (tmin), y la posición de ese elemento nos dará la máquina que debe ser elegida (mE). Con la máquina elegida, ya podemos agregar la triada s que es (número de trabajo, número de operación, número de máquina) al conjunto S, y también la dupla k que es (número de trabajo, número de operación) al subconjunto M[mE], es decir añadir una operación a la máquina elegida.

A continuación, se actualizan los tiempos de finalización para la máquina elegida, esto en el arreglo finalMaq, y también en el arreglo finalJob para el trabajo i (finalJob[i]) con el tiempo tmin en ambos arreglos. Antes del último paso, se añade el valor 1 al arreglo lJobs para el trabajo i, que simboliza que la operación en turno de ese trabajo ha sido completada.
Para finalizar, compara el arreglo que contiene el número de operaciones completadas de cada trabajo con el arreglo que contiene el número de operaciones totales de cada trabajo, si son iguales, la variable 'Continua' se torna Falsa, por lo tanto, todos los trabajos han concluido y se debe terminar el algoritmo, no sin antes devolver el conjunto S, que es la secuencia de cada operación de cada trabajo en alguna de las máquinas, y el valor máximo de entre los tiempos de finalización de cada trabajo (Max(finalJob)), que representa, nuevamente, el *makespan* mínimo.

II. Análisis 'Solving the job-shop scheduling problem optimally by dynamic programming'

El job-shop scheduling problem, o el problema de la programación de estaciones de trabajo, consiste en tener n trabajos por realizar en m máquinas. Cada trabajo deberá involucrarse en cada una de las máquinas de acuerdo a un orden, para ello, cada trabajo debe fraccionarse en partes, mismas que son llamadas operaciones. El tiempo que cada trabajo ocupará una máquina, es decir, el tiempo que una operación tomará en realizarse, depende de la operación misma y de la máquina. Los trabajos no se pueden traslapar y ningún trabajo puede procesarse en dos o más máquinas a la vez.
La meta de resolver este problema es minimizar el tiempo requerido para concretar los trabajos, a lo que los autores llaman *makespan*.
De manera formal, el problema job-shop scheduling define varios conjuntos, por ejemplo $J = \{j_1, j_2, ..., j_n\}$ nos indica la cantidad de trabajos que deben programarse, por otro lado $M = \{m_1, m_2, ..., m_m\}$ denota al conjunto de máquinas que están disponibles. Existe un tercer conjunto $O = \{o_1, o_2, ..., o_n, o_{n+1}, o_{n+2}, ..., o_{n*m}\}$ que definen a las operaciones que corresponden a los trabajos. Las primeras n operaciones corresponden a la primer operación de cada trabajo, esto, en orden de los trabajos del conjunto J, las operaciones $, o_{n+1}, o_{n+2}, ...$, representan la segunda operación de cada trabajo y así sucesivamente. Para una operación $o \epsilon O$, se denota $j(o)$ y $m(o)$ para el trabajo y máquina correspondiente a tal operación. Si se desea especificar la secuencia en que las operaciones de un trabajo ocuparán una máquina, se usa la notación $\pi_j(1), ..., \pi_j(m)$, tal que para una trabajo $j \epsilon J$, $\pi_j(i)$ es la i-ésima máquina que debe ocupar el trabajo j en una operación.
Además, los autores añaden la notación $p(o) = p_{m(o)j(o)}$ que indica el tiempo de procesamiento (makespan) de una operación $o \epsilon O$.
Por otra parte, definen 3 restricciones que especifican la manera en que una programación correcta de los trabajos funciona. Para ello utilizan la función ψ, que indica que para cada operación $o \epsilon O$, $\psi(o)$ denota el tiempo de inicio de la operación o, entonces declaran que:

1. $\psi(o) \geq 0$ para cada $o \epsilon O$
2. Para todo $o_k, o_l \epsilon O$ tal que $j(o_k) = j(o_l)$ y $k \leq l$ se tiene que $\psi(o_k) + p(o_k) \leq \psi(o_l)$
3. Para todo $o_k, o_l \epsilon O$ tal que $o_k \neq o_l$ y $m(o_k) = m(o_l)$ tenemos que $\psi(o_l) + p(o_l) \leq \psi(o_k)$ o $\psi(o_k) + p(o_k) \leq \psi(o_l)$

De modo que la restricción 1 indica que las operaciones deben iniciar en el tiempo 0 o en alguno posterior a este, no antes. La restricción 2, por su parte, indica que habrá dos operaciones k y l pertenecientes a un mismo trabajo j, y que k irá antes de l, por lo que l no podrá iniciar sin que k se haya completado antes; por lo tanto el tiempo de inicio l debe ser mayor o igual al tiempo en que k inició mas el tiempo en que le tomó concretarse. Por último, la tercer restricción indica que si dos operaciones distintas, k y l, deben ocupar la misma máquina m, k debe iniciar hasta que l se haya completado o viceversa, k no podrá iniciar hata que l haya terminado, en otras palabras, ninguna máquina m debe procesar dos operaciones simultáneamente.
De acuerdo al corolario 1, el cual está basado en la proposición 1, para cada problema job-shop scheduling, existe como solución óptima una única secuencia creciente de operaciones ordenadas para cada trabajo involucrado. A partir de ahí, tenemos que una secuencia parcial T estará formada por las primeras k operaciones para cualquier $k = 0, ..., |O|$; además, se tiene a $C_{max}(T)$ como el makespan correspondiente, es decir, el tiempo máximo para que las operaciones programadas finalicen en el orden de la secuencia T.

Para cada calendario factible del problema de planificación de tareas es posible asociar una secuencia de operaciones donde se conserva el orden de las operaciones procesadas en cada máquina, así como el orden definido para cada trabajo. Un ejemplo de este tipo de secuencia se obtiene ordenando todas las operaciones por su hora de inicio en el programa considerado.

Una formulación para problemas de secuencias desarrollada usando programación dinámica fue presentada por Held y Karp, la cual es muy famosa debido a su uso en el problema del agente viajero. En dicho problema se obtiene la ecuación de 'Bellman' como resultado.

Como se mencionó anteriormente, el problema de Planificación de tareas puede verse como un problema de secuencia. Por lo tanto, uno podría verse tentado a usar la ecuación de Bellman anterior usando la Cmax de secuencias parciales para calcular el mínimo en la ecuación de Bellman y considerando solo las expansiones ordenadas. Sin embargo, un pequeño ejemplo es suficiente para demostrar que la optimización del mínimo en el principio de la ecuación de Bellman no se cumple en el caso del problema de planificación de tareas.

Para utilizar la ecuación recursiva anterior, se proponen varios ajustes, que nos permiten restaurar.[el] principio de optimización. Para el ejemplo presentado tanto la solución parcial que puede ser completado a la solución óptima, así como a la

parcial. La solución que lo domina debe considerarse para la expansión.

*Restauración de la garantía de optimización: Para cada o ϵ ε(S) y para cada T ϵ ε(S) defina ψ(T,0) como el tiempo de inicio más temprano para la operación o si esta operación se agrega al final de la secuencia parcial ordenada T (denotada por T+0). Tenemos o ϵ η(T) si y solo si T+0 es una expansión ordenada de T.

En realidad, ζ(T,0) da un límite inferior de la finalización de o en cualquier secuencia ordenada que comience con T como una subsecuencia.

La prueba de esto está en que la proposición asegura que $\varepsilon(0) \neq 0$. Al no estar vacío, teniendo en cuenta la forma en que se construye el conjunto, concluimos que contiene un solo elemento, que es óptimo.

*Formulación de programación dinámica para el job-shop scheduling problem: La pregunta que surge ahora es cómo se puede obtener $\varepsilon(\Theta)$ progresivamente. Para dar una respuesta adecuada, definimos X(S) como el conjunto de todas las secuencias parciales ordenadas T ϵ ε(S) tal que T es una expansión de una secuencia parcial ordenada T', donde T ϵ $\varepsilon(S\{0\})$.

Finalmente, se han reunido todos los elementos necesarios para proponer un enfoque de Programación Dinámica para el problema de planificación de tareas.

El algoritmo que se propone puede describirse como: Se construyen las primeras n secuencias de una sola operación (la primera operación para cada trabajo). Hay que tener en cuenta que todas estas secuencias tienen un conjunto diferente de operaciones S. Después, para todos estos conjuntos S con |S|=1 estas secuencias se expanden con todas las operaciones posibles en o ϵ ε(S), las expansiones que conducen a secuencias parciales ordenadas se mantienen agrupadas para el siguiente paso (X(S')) por su nuevo conjunto S' con |S|=2. Cuando se agrega una secuencia parcial ordenada a un grupo no vacío, se compara con todos los elementos que ya están en ese grupo y se eliminan las secuencias dominadas. Después de que todas las secuencias de longitud l se expanden (y, por lo tanto, se crean secuencias de longitud l+1) procedemos a expandir todas las secuencias de longitud l+1, y así sucesivamente.

*Reducción del espacio de estados:Se propone una reducción en el espacio de estado del enfoque de Programación Dinámica para el problema de planificación de tareas.

Esta reducción del espacio de estado puede incorporarse fácilmente en el algoritmo propuesto en la opción anterior al verificar las condiciones para cada T1. Cuando se cumplan estas condiciones, no se expande T1 más. hay que tener en cuenta que T1 debe agregarse a X(S) para permitirle descartar otras secuencias (cuando T1 es menor que T2).

Hablando de la complejidad del algoritmo mencionado en el artículo previamente vemos que en el cuerpo principal del algoritmo, los dos primeros ciclos FOR juntos se repiten en todos los conjuntos SDO, siendo como máximo un 2 elevado a la m*n, pero este valor puede disminuir.

El valor final obtenido para este algoritmo es: $\Theta((Pmax)^{2n}(m+1)^2)$

Si se comparara este resultado con la utilización de fuerza bruta, determinamos que para un Pmax fijo, el enfoque de Programación dinámica que se propone para el problema de planificación de tareas tiene una complejidad que es exponencialmente menor que la fuerza bruta en n y en m.

Lo que este análisis muestra es que el nuevo algoritmo resuelve el problema de planificación de tareas con una complejidad exponencialmente menor que la "fuerza bruta".

Para evaluar el análisis de complejidad del nuevo algoritmo, se realizaron algunas pruebas computacionales considerando instancias de referencia para el problema de planificación de tareas

Se implemento el nuevo algoritmo, incluida la reducción del espacio de estado que se platicó anteriormente con C++ y las pruebas se realizaron en una máquina Windows de 64 bits con una CPU de 2,66 GHz y 8 GB de memoria.

Para comparar estos resultados con la fuerza bruta, se implementó un algoritmo de fuerza bruta (también en C ++) y se probaron las mismas instancias con este algoritmo en la misma máquina. Después de 10^i soluciones, se terminaron las ejecuciones para poder extrapolar el tiempo total de ejecución del algoritmo de fuerza bruta.

Se debe enfatizar que, aunque utilizando una cantidad exponencial de memoria, el algoritmo puede implementarse de tal manera que solo se encuentre el valor óptimo ahorrando un factor constante en la memoria. Sin embargo, para los casos en que se pudo ejecutar no solo el valor óptimo, sino también se encontró una solución óptima.

III. Descripción lenguajes de programación, bibliotecas o APIs

El lenguaje de programación que se utilizó para realizar la aplicación es Python, en su versión 3.7.7.
Python es un lenguaje de scripting independiente de plataforma y orientado a objetos, preparado para realizar cualquier tipo de programa, desde aplicaciones Windows a servidores de red o incluso, páginas web. Es un lenguaje interpretado, lo que significa que no se necesita compilar el código fuente para poder ejecutarlo, lo que ofrece ventajas como la rapidez de desarrollo e inconvenientes como una menor velocidad.

Para construir la interfaz gráfica, que es la que lleva la interacción con el usuario final, se hizo uso del módulo Tkinter, este, entre otras cosas, ofrece un paquete multiplataforma estándar de python para crear interfaces gráficas de usuario (GUI). Proporciona acceso a un intérprete de Tcl subyacente con el kit de herramientas Tk, que en sí mismo es una biblioteca de interfaz de usuario gráfica multiplataforma y multiplataforma. La mayor fortaleza de Tkinter es su ubicuidad y simplicidad. Funciona de forma inmediata en la mayoría de las plataformas (Linux, OSX, Windows) y se completa con una amplia gama de widgets necesarios para las tareas más comunes (botones, etiquetas, lienzos de dibujo, texto de varias líneas, etc.). A juzgar por la experiencia de nosotros, observamos que hay similitud con Javascript, pues el manejo de los widgets es muy parecido, siendo este el elemento que nos llevará a la elección del módulo.

Para crear la tabla que contiene los tiempos de cada operación,

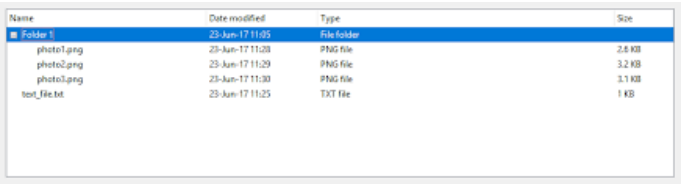


Figura 1. 'treeview' del módulo tkinter en python

ocupamos una instancia de la clase ttk, perteneciente al mismo módulo tkinter. Dentro de esta clase, existe el treeview (vista de árbol) que nos permite de manera sencilla y cómoda organizar los datos en una tabla, mostrando un resultado elegante y claro, pues presenta la información de modo jerárquico (ver Figura 1).

Para poder mostrar los resultados en una gráfica, se utilizó el módulo 'matplotlib', librería de Python que permite crear vistas estáticas, animadas o interactivas. Permitiendo mostrar a la salida un resultado como el de la figura 2.

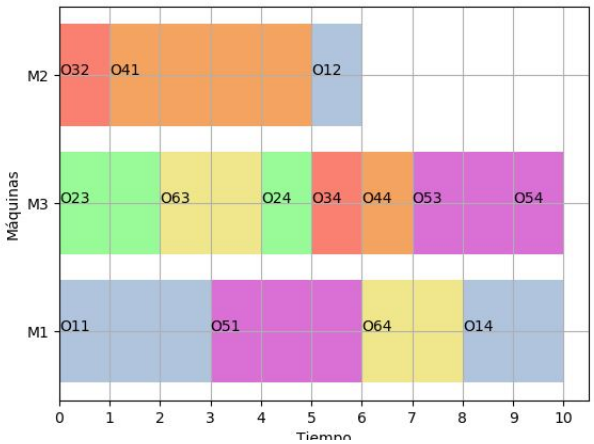


Figura 2. Gráfica hecha con matplotlib

IV. Desarrollo y descripción uso de la aplicación

En esta sección se describe la interacción que el usario final debe tener con al interfaz para poder ingresar datos, y visualizar los resultados tanto en la gráfica como en la tabla.

Una vez ejecutada la aplicación, aparece la interfaz gráfica GUI1 (ver figura 3). En ella, vemos 3 botones distribuidos sobre una sola columna pero en diferentes filas; al lado derecho de cada botón tenemos un campo de entrada, que permite al usuario establecer el número de operaciones, para el primer botón, el número de máquinas, en el segundo botón, y para el tercer botón, el número de operaciones. Una vez que el usuario ingresé el valor en algún campo, podrá pulsar el botón a la izquierda para establecer el valor de la variable correspondiente.

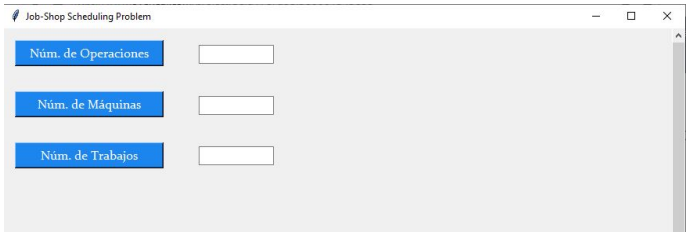


Figura 3. GUI1. Inicio de la aplicación

Cuando el usuario haya establecido (llenar los campos y pulsar el respectivo botón) los valores para el número de operaciones de máquinas, se creará dentro de GUI1, la sección 'Agregar Operaciones' mostrada en la Figura 4. De acuerdo al

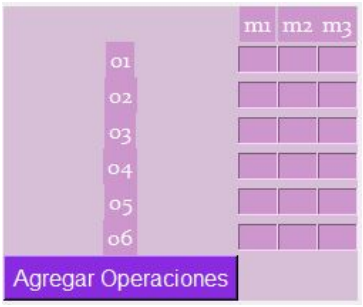


Figura 4. Sección para añadir los tiempos en cada máquina, de cada operación

número de operaciones y número de máquinas, se creará una tabla de tamaño nO*nM, donde nO es el total de operaciones representadas en filas y nM el total de máquinas representadas en columnas. Dentro de la tabla, habrá campos de entrada que van a permitir establecer el tiempo de cada una de las operaciones para cada máquina. Por ejemplo, si se quiere colocar el tiempo de la operación 2 en la máquina 3, se debe ubicar el valor en la fila 2, columna 3. Cuando el usuario haya ingresado todos los valores correspondientes, deberá pulsar el botón 'Agregar Operaciones' ubicado en la parte inferior de la sección. Este botón permitirá recoger los datos ingresados en la tabla para crear el conjunto TOperaciones, que es una de las entradas para el Algorithm 1. Una vez ingresada la cantidad de Operaciones y la cantidad de trabajos, se agrega a la GUI1, la sección 'Agregar trabajos' (ver Figura 5).

Figura 5. Sección para establecer las operaciones que forman parte de cada trabajo

Se crea una tabla, donde cada fila representan a un trabajo,

y cada columna representa una operación. A diferencia de la tabla para agregar los tiempos de las operaciones, en esta crea por cada posición un checkbutton, permitiendo de manera precisa, establecer las operaciones que forman parte de un trabajo. Por ejemplo si se desea establecer las operaciones 2 y 3 del trabajo 4, basta con ubicarse en la fila 4, y seleccionar los checkbutton de la columna 2 y 3 de esa fila, que representan respectivamente a las operaciones mencionadas. Cuando el usuario haya seleccionado las operaciones de cada trabajo, basta con oprimir el botón 'Agregar Trabajos' ubicado en la parte inferior de la sección, de esa forma la aplicación podrá construir, con las operaciones seleccionadas de cada trabajo, el conjunto Jobs, que es uno de los conjuntos que recibe como entrada el Algorithm 1.

Finalmente, cuando estén formados los conjuntos entrada, TOperaciones y Jobs, se podrá visualizar el botón 'Iniciar Simulación' dentro de GUI1 (ver Figura 6). Si el usuario presiona el botón, se invocará al Algorithm 1 dentro de la aplicación, resolviendo la instancia del problema para los datos ingresados. Cuando la función principal (JSSP) finalice, a la salida podremos observar 2 nuevas ventanas, la primera de ellas contendrá la gráfica (Figura 2) y la otra contendrá la tabla de tiempos (Figura 1).

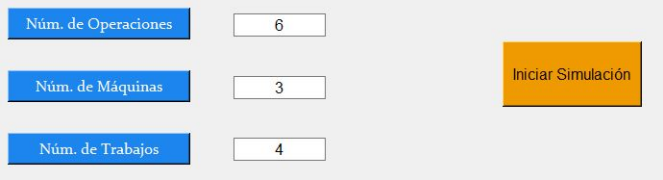


Figura 6. El botón que permite ver la gráfica y la tabla de tiempos del FJSSP

V. Pruebas

En este apartado, incluimos una ejecución de ejemplo, desde que es iniciada la aplicación, hasta llegar a ver la gráfica y la tabla de tiempos.
En la figura 7, declaramos que habrá 6 operaciones, 4 trabajos y 3 máquinas disponibles. Una vez establecidos los valores, agregamos los tiempos de cada operación en cada máquina como se muestra en la Figura 8.

En la Figura 9, seleccionamos las operaciones de cada trabajo.

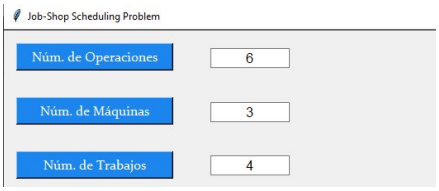


Figura 7. Valores para el ejercicio

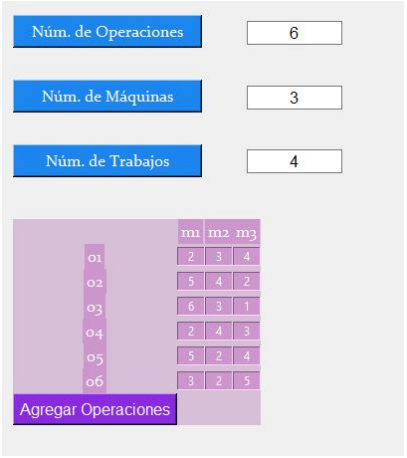


Figura 8. Tiempos de las operaciones para el ejercicio

Figura 9. Seleccionamos las operaciones en cada trabajo

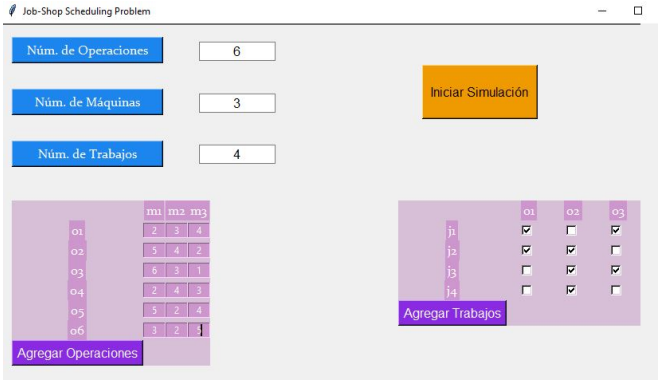
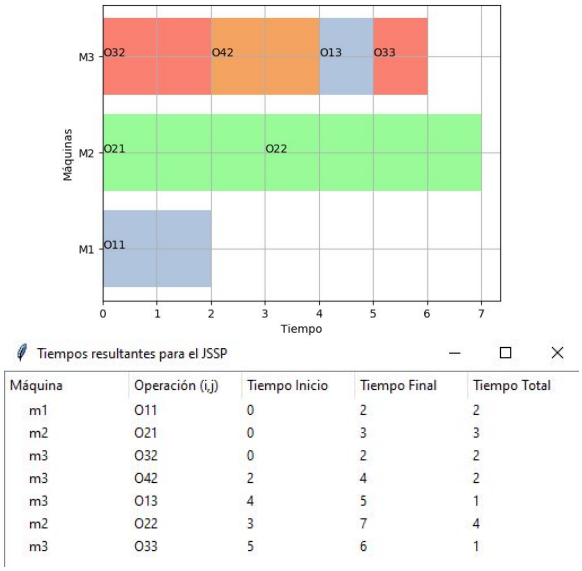


Figura 10. GUI1 completa y listo para la simulación

Posteriormente, en la Figura 10. vemos como aparece el botón 'Iniciar Simulación', dando opción de ejecutar el Algoritmo.
Por último, vemos en la Figura 11, los resultados arrojados después de la simulación. Arriba la gráfica y abajo, la tabla de tiempos.

VI. Conclusiones individuales

- Cano Rodríguez Adriana
  Como sabemos, el Job Shop Scheduling entra en la categoría de problemas NP, esto debido a que no es fácil resolverlos, pudimos darnos cuenta al intentar resolver el algoritmo, las restricciones que éste tiene no son pocas ni son sencillas de implementar. Hay que



Máquina	Operación (i,j)	Tiempo Inicio	Tiempo Final	Tiempo Total
m1	O11	0	2	2
m2	O21	0	3	3
m3	O32	0	2	2
m3	O42	2	4	2
m3	O13	4	5	1
m2	O22	3	7	4
m3	O33	5	6	1

Figura 11. Resultados de la simulación del ejercicio

tener cuidado en que no se traslapen las operaciones, de lo contrario, si lo vemos en un ambiente real nos causaría mucho más complicaciones. Además hay que tener presente que no sólo busca asignar las tareas y los recursos así porque sí, sino se busca una optimización, de lo contrario no tendría mucha razón de ser. El problema de asignación de tareas no es algo a lo que no estemos acostumbrados, al contrario, lo vemos cada día en nuestras computadoras por ejemplo, la máquina debe procesar diversas tareas buscando el mejor rendimiento, sin afectar y saturar los recursos de la computadora y el mejor servicio a nosotros que somos los ” clientes”. También hay que tener especial atención a la diferencia que existe entre Job Scheduling y Job Shop Scheduling, en el primero no existe la precedencia de operaciones, dos tareas que exigen el mismo número de recursos pueden ejecutarse en paralelo y una tarea puede requerir varios recursos simultáneamente para ejecutarse, todo lo mencionado no sucede en el Job Shop.

- Cuevas Naranjo Luis
  El Job Shop Scheduling Problem en su versión normal resulta interesante de analizar, tomando en cuenta que el punto de partida es el problema de la candelarización de actividades, temá que forma parte del temario de la materia. Es un problema que lleva a, de alguna manera, organizar el recurso para tomar la mayor cantidad de actividades posibles; sin embargo en el JSSP, el problema rebasa ese límite de tener 1 sólo recurso, en este caso, hay mayor complejidad a la hora de organizar y sincronizar todos los recursos para que puedan operar en conjunto, respetando las condiciones establecidas, un recurso no puede ejecutar 2 operaciones a la vez o que un trabajo no puede ser operado por más de 1 un recurso a la vez, inlcuso la condición más sencilla: todos los recursos deben estar disponibles a partir del tiempo 0, no antes. El ejercicio toma un giro aún más interesante cuando se busca implementar el Flexible JSSP, pues ahora no sólo hay que superar la barrera impuesta por las condiciones y escoger cualquier máquina disponible, sino que hay que encontrar a la mejor máquina disponible, es decir, aquella que además de cumplir con las condiciones, ofrezca el menor tiempo de finalización. A partir del nuevo estado del problema, se implementó un mecanismo para guardar los tiempos de finalización de cada trabajo, pero también de cada máquina, y esa fue la primicia que permitió dar una solución al ejercicio.

- Elizarrarás Ortiz Carlos
  Con la realización de este proyecto pude entender más a fondo de que trata el Job.shop Scheduling Problem y la manera en la que opera un algoritmo que resulve dicho problema. Este problema me resultó de gran interés porque omplicaba un problema de optimización combinatoria, temas que se abarcaron en el semestre, esto, junto con el hecho de que esta problemática tiene como objetivo organizar un sólo recurso, lo cual lo hace tener una dificultad aún mayor, hicieron que este fuera nuestra elección de proyecto. Mientras indagabamos sobre como resolver el problema, surgoó la idea de implementar el Flexible Job-Shop Scheduling Problem, el cual nos presentó un mayor reto en cuanto a como encarar ek problema pero que nos resultó de gran satisfacción al implementarlo y así poder darle una solución a esta la situación presentada en los requerimientos del proyecto. Sin duda alguna, el tener la oportunidad de realizar este ejercicio nos permitió reforzar nuestros conocimientos tanto teóricos como prácticos sobre esta clase de algoritmos tan intereresantes, además de que nos abripo los ojos sobre las dificultades que presentan las restricciones de los mismos, por lo que se debe trabajar arduamente para no toparse con ellas.

REFERENCIAS

[1] Gromicho, J., van Hoorn, J., Saldanha, F., Timmer, G. (2015). *Solving the job-shop scheduling problem optimally by dynamic programming*. Recuperado de: https://linkinghub.elsevier.com/retrieve/pii/S0305054812000500

[2] Pezzella, F., Morganti, G., Ciaschetti, G. (2007). *A genetic algorithm for the Flexible Job-Shop Scheduling Problem*. Recuperado de: https://www.sciencedirect.com/science/article/abs/pii/S0305054807000524

[3] Driss, I., Mouss, K., Laggoun, A. (2015). *An effective Genetic Algorithm for the Flexible Job Shop Scheduling Problems*. Recuperado de: https://www.researchgate.net/publication/276859974