Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Informe de la Práctica 7 de DAA
Curso 2023-2024

Parallel Machine Scheduling Problem with Dependent Setup Times

Elías Hernández Abreu

La Laguna, 14 de abril de 2024

Figura 1: Midjourney

La Figura 1 muestra una imagen creada por Midjourney recreando un ejemplo de aplicación real del problema abordado en esta práctica.

Licencia

Resumen

La idea es crear un programa que resuelva el "Scheduling Pallalel Machine Problemüsando el tiempo total de completación "TCTçomo heurística objetivo. Resulta intuitivo pensar que el costo de una máquina viene dado por su tiempo de preparación, único para cada tarea, y el tiempo de cambio de la tarea anterior a la nueva, resultando en una fórmula sencilla y lineal. Por ejemplo, una máquina con las tareas 1, 2 y 3, su tct podría calcularse como:
M1: p1+c1 + p2+c2 + p3+c3
Sin embargo, la qu vamos a usar es:
M1: p1+c1 + p1+c1+p2+c2 + p1+c1+p2+c2+p3+c3
Que puede reescribirse como:
*M1: (p1+c1)*3 + (p2+c2)*2 + (p3+c3)*1*
Que es fácilmente traducible a código, pero significa que las tareas con mayor peso deberían estar al principio de las máquinas, mientras que aquillas con menor peso deberían estar al final, importando el orden y posición de cada tarea.

Para resolver el problema se usan 4 algoritmos: Voraz, GRASP, GRASP con búsquedas locales y GVNS, que serán explicados más adelante.

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Contexto

El objetivo de la práctica es crear un programa capaz de resolver el PMSP usando diferentes algoritmos, de los cuales son destacables el algoritmo por búsquedas locales, el algoritmo GRASP, y el algoritmo GVNS.

1.1.1. Objetivos

Durante el desarrollo de esta práctica, se han cumplido los objetivos listados a continuación:

- Algoritmo voraz
- Algoritmo GRASP
- Algoritmo GRASP con búsquedas locales
- Obtención de incremento aislado en operaciones sobre la solución
- Algoritmo VND sobre soluciones existentes
- Algoritmo RVNS
- Algoritmo GVNS

1.2. Motivación

La motivación de la práctica es entender los algoritmos aproximados y su importancia a la hora de encontrar resultados, ya que no siempre se poseen los recursos para usar algoritmos exactos y no siempre basta con usar un algoritmo como el voraz.

Capítulo 2

Parallel Machine Scheduling Problem with Dependent Setup Times

2.1. Descripción

En esta práctica estudiaremos un problema de secuenciación de tareas en máquinas paralelas con tiempos de setup dependientes; Parallel Machine Scheduling Problem with Dependent Setup Times [1]. El objetivo del problema es asignar tareas a las máquinas y determinar el orden en el que deben ser procesadas en las máquinas de tal manera que la suma de los tiempos de finalización de todos los trabajos, es decir, el tiempo total de finalización (*TCT*), sea minimizado.

El tiempo de setup es el tiempo necesario para preparar los recursos necesarios (personas, máquinas) para realizar una tarea (operación, trabajo). En algunas situaciones, los tiempos de setup varían según la secuencia de trabajos realizados en una máquina; por ejemplo en las industrias química, farmacéutica y de procesamiento de metales, donde se deben realizar tareas de limpieza o reparación para preparar el equipo para realizar la siguiente tarea.

Existen varios criterios de desempeño para medir la calidad de una secuenciación de tareas dada. Los criterios más utilizados son la minimización del tiempo máximo de finalización (*makespan*) y la minimización del *TCT*. En particular, la minimización del *TCT* es un criterio que contribuye a la maximización del flujo de producción, la minimización de los inventarios en proceso y el uso equilibrado de los recursos.

El problema abordado en esta práctica tiene las siguientes características:

- Se dispone de *m* máquinas paralelas idénticas que están continuamente disponibles.
- Hay *n* tareas independientes que se programarán en las máquinas. Todas las tareas están disponibles en el momento cero.
- Cada máquina puede procesar una tarea a la vez sin preferencia y deben usarse todas las máquinas.
- Cualquier máquina puede procesar cualquiera de las tareas.
- Cada tarea tiene un tiempo de procesamiento asociado p_j.
- Hay tiempos de setup de la máquina s_{ij} para procesar la tarea j justo después de la tarea i, con $s_{ij} \neq s_{ji}$, en general. Hay un tiempo de setup s_{0j} para procesar la primera tarea en cada máquina.

- El objetivo es minimizar la suma de los tiempos de finalización de los trabajos, es decir, minimizar el TCT.

El problema consiste en asignar las *n* tareas a las *m* máquinas y determinar el orden en el que deben ser procesadas de tal manera que se minimice el TCT.

El problema se puede definir en un grafo completo $G = (V, A)$, donde $V = \{0, 1, 2, \cdots, n\}$ es el conjunto de nodos y A es el conjunto de arcos. El nodo 0 representa el estado inicial de las máquinas (trabajo ficticio) y los nodos del conjunto $I = \{1, 2, \cdots, n\}$ corresponden a las tareas. Para cada par de nodos $i, j \in V$, hay dos arcos $(i, j), (j, i) \in A$ que tienen asociados los tiempos de setup s_{ij}, s_{ji} según la dirección del arco. Cada nodo $j \in V$ tiene asociado un tiempo de procesamiento p_j con $p_0 = 0$. Usando los tiempos de setup s_{ij} y los tiempos de procesamiento p_j, asociamos a cada arco $(i, j) \in A$ un valor $t_{ij} = s_{ij} + p_j, (i \in V, j \in I)$.

Sea $P_r = \{0, [1_r], [2_r], \cdots, [k_r]\}$ una secuencia de $k_r + 1$ tareas en la máquina *r* con el trabajo ficticio 0 en el posición cero de P_r, donde $[i_r]$ significa el nodo (tarea) en la posición i_r en la secuencia *r*. Luego, el tiempo de finalización $C_{[i_r]}$ del trabajo en la posición i_r se calcula como $C_{[i_r]} = \sum_{j=1}^{i_r} t_{[j-1][j]}$. Tenga en cuenta que en el grafo *G* representa la longitud de la ruta desde el nodo 0 al nodo $[i_r]$.

Sumando los tiempos de finalización de los trabajos en P_r obtenemos la suma de las longitudes de las rutas desde el nodo 0 a cada nodo en P_r ($TCT(P_r)$) como:

$$TCT(P_r) = \sum_{i=1}^{k} C_{[i]} = kt_{[0][1]} + (k-1)t_{[1][2]} + \cdots + 2t_{[k-2][k-1]} + t_{[k-1][k]} \tag{2.1}$$

Usando lo anterior, el problema se puede formular como encontrar *m* rutas simples disjuntas en G que comienzan en el nodo raíz 0, que juntas cubren todos los nodos en *I* y minimizan la función objetivo.

$$z = \sum_{r=1}^{m} TCT(P_r) = \sum_{r=1}^{m} \sum_{i=1_r}^{k_r} (k_r - i + 1)t_{[i-1][i]} \tag{2.2}$$

Tenga en cuenta que el coeficiente $(k_r - i + 1)$ indica el número de nodos después del nodo en la posición $i_r - 1$.

2.1.1. Representación de las soluciones

Podemos generar un array por cada una de las máquinas, S={A_1, A_2,.., A_m}. En ellos se insertarán las tareas a ser procesadas en cada máquina en el orden establecido.

Capítulo 3

Algoritmos

3.1. Algoritmo consturictivo voraz

Un constructivo voraz

Un algoritmo constructivo voraz muy sencillo para este problema parte del subconjunto, S, formado por las m tareas con menores valores de t_{0j} asignadas a los m arrays que representan la secuenciación de tareas en las máquinas. A continuación, añade a este subconjunto, iterativamente, la tarea-máquina-posición que menor incremento produce en la función objetivo. El pseudocódigo de este algoritmo se muestra a continuación.

Algoritmo constructivo voraz

1: Seleccionar la m tareas $j_1, j_2, .., j_m$ con menores valores de t_{0j} para ser introducidas en las primeras posiciones de los arrays que forman la solucion S;
2: S = {A$_1 = \{j_1\}, A_2 = \{j_2\}, .., A_m = \{j_m\}$};
3: **repeat**
4: $S^* = S$;
5: Obtener la tarea-maquina-posicion que minimiza el incremento del TCT;
6: Insertarla en la posicion que corresponda y actualizar S^*;
7: **until** (todas las tareas han sido asignadas a alguna maquina)
9: Devolver S^*;

Figura 3.1: Algoritmo constructivo voraz

3.2. Búsquedas Locales

El algoritmo de búsquedas locales consta en, a partir de una solución inicial, buscar un óptimo local aplicando un mismo tipo de movimiento repetidamente mientras mejore el TCT. Como se hace con un solo tipo de movimiento, no es una solución demasiado fiable aunque sí mejor que la ya generada, pues permite cambiar deciciones realizadas con anterioridad en favor de otras que hubieran sido mejores.

Búsquedas locales

1: A partir de una solución inicial S, mejorar la solución
mediante movimientos simiples hasta que no sea posible. Los movimientos
pueden ser reinserción en la misma máquina o global, y cambio de tareas
en la misma máquina o global.
2: **repeat**
3: $S^* = S$;
4: Obtener la lista de movimientos desde la solución actual;
5: Escoger el movimiento con menor TCT;
6 **if** (Es mejor que la anterior) $S = S^*$;
7: **until** (La solución no mejora)
8: Devolver S;

Figura 3.2: Algoritmo búsqudas locales

3.3. GRASP

El algoritmo GRASP es muy parecido al algoritmo voraz en el sentido en el que busca la mejor solución posible sin mirar atrás, la diferencia es que GRASP no escoge la mejor solución, sino una al azar de entre las mejores.

Algoritmo GRASP

1: Seleccionar la m tareas $j_1, j_2, .., j_m$ con menores valores de t_{0j} para ser introducidas en las primeras posiciones de los arrays que forman la solucion S;
2: S = {$A_1 = \{j_1\}, A_2 = \{j_2\}, .., A_m = \{j_m\}$};
3: **repeat**
4: $S^* = S$;
5: Obtener las n tareas-maquina-posicion que minimizan el incremento del TCT;
6: Escoger una de ellas al azar e insertarla en la posicion que corresponda y actualizar S^*;
7: **until** (todas las tareas han sido asignadas a alguna maquina)
9: Devolver S^*;

Figura 3.3: Algoritmo GRASP

3.4. GVNS

El algoritmo GVNS esta compuesto en dos subalgoritmos, el VND y el RVNS. El VND aplica movimientos sobre una solución hasta llegar a un óptimo local. El RVNS aplica un movimiento aleatorio a una solución para alejarse del óptimo local actual y poder encontrar el óptimo global. El GVNS consta de alejarse con RVNS y saturar la solución con VND, quedándose siempre con la mejor encontrada. El proceso se repite partiendo de varias soluciones creadas con GRASP para obtener resultados diversos, del cual se escoge el mejor.

$ **Algoritmo GVNS**

1: repeat (creciendo la distancia de la solución vecina;)
2: Generar una solución vecina aleatoria a la distancia dada;
3: VND para búsquedas locales;
4: Saturar la estructura de búsqueda principal.;
5: Buscar en las otras estructuras de búsqueda soluciones mejores hasta que no se pueda obtener;
6: la solución encontrada es mejor que la que se tenía, moverse;
Si

Figura 3.4: Algoritmo GRASP

Capítulo 4

Experimentos y resultados computacionales

4.1. Constructivo voraz

Algoritmo voraz					
Problema	n	m	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	2	1	7898	0.165 ms
$I40j_4m_S1_1$	40	4	2	8140	0.137 ms
$I40j_8m_S1_1$	40	8	3	4537	0.061 ms
$I70j_2m_S1_1$	70	2	4	45172	0.596 ms
$I70j_4m_S1_1$	70	4	5	24323	0.400 ms
$I70j_8m_S1_1$	70	8	6	13054	0.327 ms

Cuadro 4.1: Algoritmo voraz. Tabla de resultados

4.2. Búsquedas locales

Cambio local				
Problema	n	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	1	14876	3.855 ms
$I40j_4m_S1_1$	40	2	8132	2.676 ms
$I40j_8m_S1_1$	40	3	4687	2.349 ms
$I70j_2m_S1_1$	70	4	46553	13.331 ms
$I70j_4m_S1_1$	70	5	24286	8.872 ms
$I70j_8m_S1_1$	70	6	13660	6.966 ms

Cuadro 4.2: Cambio en misma máquina

Cambio global

Problema	n	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	1	14883	3.869 ms
$I40j_4m_S1_1$	40	2	7619	3.566 ms
$I40j_8m_S1_1$	40	3	4499	3.232 ms
$I70j_2m_S1_1$	70	4	46682	14.400 ms
$I70j_4m_S1_1$	70	5	24229	10.011 ms
$I70j_8m_S1_1$	70	6	24229	10.414 ms

Cuadro 4.3: Cambio entre diferentes máquinas

Reinserción local

Problema	n	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	1	14251	3.689 ms
$I40j_4m_S1_1$	40	2	7995	2.719 ms
$I40j_8m_S1_1$	40	3	4679	2.480 ms
$I70j_2m_S1_1$	70	4	46166	13.619 ms
$I70j_4m_S1_1$	70	5	24755	9.042 ms
$I70j_8m_S1_1$	70	6	13653	7.515 ms

Cuadro 4.4: Reinserción en la misma máquina

Reinserción global

Problema	n	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	1	14140	4.243 ms
$I40j_4m_S1_1$	40	2	7385	3.477 ms
$I40j_8m_S1_1$	40	3	4475	2.755 ms
$I70j_2m_S1_1$	70	4	44364	17.398 ms
$I70j_4m_S1_1$	70	5	23244	11.392 ms
$I70j_8m_S1_1$	70	6	12691	9.435 ms

Cuadro 4.5: Reincersión en diferentes máquinas

4.3. GRASP

GRASP					
Problema	n	$\|LRC\|$	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	3	1	15356	3.309 ms
$I40j_4m_S1_1$	40	3	2	8217	2.519 ms
$I70j_2m_S1_1$	40	3	4	4668	2.660 ms
$I40j_8m_S1_1$	70	3	3	47430	11.911 ms
$I70j_4m_S1_1$	70	3	5	25233	8.666 ms
$I70j_8m_S1_1$	70	3	6	13847	7.328 ms

Cuadro 4.6: GRASP. Tabla de resultados

4.4. GVNS

Debe incluir el diseño de experimentos para el ajsute de parámetros y el análisis de los resultados obtenidos

GVNS					
Problema	m	k_{max}	Ejecución	TCT	CPU
$I40j_2m_S1_1$	40	5	1	13173	1.789 s
$I40j_4m_S1_1$	40	5	2	7077	1.281 s
$I70j_2m_S1_1$	40	5	3	4169	0.745 s
$I40j_8m_S1_1$	40	5	4	42360	15.473 s
$I70j_4m_S1_1$	40	5	5	22145	5.848 s
$I70j_8m_S1_1$	40	5	6	12327	2.738 s

Cuadro 4.7: GVNS. Tabla de resultados

4.5. Análisis compartativo entre algoritmos

Como era esperable, el algoritmo GVNS da los mejores resultados para los TCTs pero es el más lento por una gran diferencia, llegando incluso a 15 segundos en algunas instancias. Por otro lado, las búsquedas locales muestran resultados decentes a pesar de ser basados en una solución aleatoria. El algoritmo voraz es mejor que el GRASP, ya que éste posee un componente aleatorio que lo aleja de la mejor solución local, aunque podría encontrarla debido al mismo, por lo que por lo general el voraz tiene mejores resultados en la mayoría de los casos.

Capítulo 5

Conclusiones y trabajo futuro

Los algoritmos vistos en esta práctica sirven de ayuda para resolver problemas complejos como el pmsp, aunque no de forma perfecta, sí de forma muy aproximada. Obviamente en la práctica han surgido complicaciones y retrasos debido a la complejidad de la misma.

Como parte negativa del trabajo, la parte más complicada ha sido trabajar con LaTeX, pues no conozco la gramática y no tengo tiempo de aprenderla para poder entregar un informe decente, además de comportamiento inesperado como texto insertándose antes de tablas aunque aparezca después en el código y falta de documentación online, y aquella disponible usa reglas que no existen en el entorno de overleaf.

Bibliografía

[1] S. Báez, F. Angel-Bello, A. Alvarez, and B. Melián-Batista. A hybrid metaheuristic algorithm for a parallel machine scheduling problem with dependent setup times. *Computers and Industrial Engineering*, 131:295–305, 2019.