Un algoritmo GRASP para resolver el problema de la programación de tareas dependientes en máquinas diferentes (task scheduling)

Manuel Tupia
Pontificia Universidad Católica del Perú, Departamento de Ingeniería
Av. Universitaria cuadra 18 S/N
Lima, Perú, Lima 32
tupia.mf@pucp.edu.pe

David Mauricio
UPG-FISI Universidad Nacional Mayor de San Marcos
Av. Germán Amézaga S/N.
Lima, Perú, Lima 1
dms@terra.com

Abstract

The industrial planning has experienced notable advances from his origins around the middle of the century XX not only a meal in the efficiency in application importance inside all the industries where it is used, and sophistication of the algorithms that try to resolve the problems that generate all its existent variants. The interest for the heuristic-methods application in front of to give answers to the problems of the area of planning need has taken us to develop new algorithms to resolve one of the problem variants of the planning from the Artificial Intelligence's point of view: The programming of tasks or task scheduling: once an tasks set was given to be programmed in determined machines group, finding an order once was made suitable of execution that minimize the time once was accumulated total of processing of the machines or makespan. The present work GRASP to resolve programming the problem of dependent tasks in different machines shows a heuristic goal.

Keywords: GRASP algorithms, task scheduling, combinatorial optimization, meta heuristics, Artificial Intelligence

Resumen

La planificación industrial ha experimentado notables avances desde sus orígenes a mediados del siglo XX tanto en importancia de aplicación dentro de todas las industrias en donde es usada, como en la eficiencia y sofisticación de los algoritmos que buscan resolver los problemas que generan todas sus variantes existentes. El interés por la aplicación de métodos heurísticos ante la necesidad de dar respuestas a los problemas del área de planificación nos ha llevado a desarrollar nuevos algoritmos para resolver una de las variantes del problema de la planificación desde el punto de vista de la Inteligencia Artificial: la programación de tareas o task scheduling: dado un conjunto de tareas dependientes de una línea de producción a ser programadas en un determinado grupo de máquinas diferentes, encontrar un orden adecuado de ejecución que minimice el tiempo total de trabajo de las máquinas o makespan. El presente trabajo muestra una meta heurística GRASP para resolver dicha variante del problema del task scheduling.

Palabras claves: algoritmos GRASP, programación de tareas, optimización combinatoria, meta heurísticas, Inteligencia Artificial

1. INTRODUCCIÓN

El problema de la programación de tareas o task scheduling presenta sus antecedentes en la planificación industrial [1] y en la programación de trabajos de procesadores en los inicios de la micro electrónica [2]. Ese tipo de problema puede definirse, desde el punto de vista de la optimización combinatoria [3], como sigue:

- Se tienen M máquinas, también denominadas procesadores.
- Se tienen N tareas cada una de ellas con duración T_{ij} unidades de tiempo (el tiempo que demora en ser ejecutada la tarea j en la máquina i).
- El objetivo es programar las N tareas en las M máquinas procurando el orden de ejecución más apropiado, cumpliendo determinadas condiciones que satisfagan la optimalidad de la solución requerida para el problema [4].

El problema presenta una serie de variantes dependiendo de la naturaleza y el comportamiento tanto de las tareas y de las máquinas. Una de las variantes más difíciles de plantear, debido a su alta complejidad computacional es aquella en donde ***las tareas son dependientes y las máquinas diferentes***.

En esta variante cada tarea presenta una lista de tareas que la preceden y para ser ejecutada deben esperar el procesamiento de toda la lista completa de predecesoras. A esta situación hay que agregarle la característica de heterogeneidad de las máquinas: cada una de ellas se demora en ejecutar una tarea tiempos distintos entre sí. El objetivo será minimizar el tiempo acumulado de ejecución de las máquinas, conocido en la literatura como ***makespan***.

Al observar el estado del arte del problema vemos que tanto su aplicación práctica de forma directa en la industria y su importancia académica (al ser un problema NP-difícil) se justifica el diseño de un algoritmo heurístico que busque una solución óptima al problema, dado que no existen métodos exactos para resolver el problema. En muchas industrias como las del ensamblado, embotellado, manufactura, etc., vemos líneas de producción en donde los períodos de espera por trabajo de las máquinas involucradas y el ahorro del recurso tiempo son temas muy importantes y requieren de una conveniente planificación.

A partir de la definición dada, podemos presentar el problema como un modelo matemático de optimización combinatoria en su forma más general, en la siguiente figura:

Minimizar X_0

s. a. $$X_0 \geq \sum_{i=1}^{n} T_{ij} * X_{ij} \quad \forall \ j \in 1..M$$

$$X_0 \geq \sum_{j=1}^{m} X_{ij} = 1 \quad \forall \ i \in 1..N$$

Figura 1: Modelo matemático del task scheduling

El presente escrito propone una heurística del tipo goloso-miope o voraz para resolver la variante antes mencionada del task scheduling.

2. ALGORITMOS META HEURÍSTICOS

Los procedimientos meta heurísticos son una clase de métodos aproximados que están diseñados para resolver problemas difíciles de optimización combinatoria, en los que las soluciones heurísticas clásicas no son efectivos. Proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la Inteligencia Artificial (en adelante IA), la evolución biológica y los mecanismos estadísticos.

Permiten generalizar los procedimientos de búsqueda heurística, generando esquemas que pueden ser adaptados a diversos dominios de problemas. Amplían los criterios y modos de optimización de funciones de las heurísticas reduciendo el riesgo de estancamiento en óptimos locales.

Se sitúan conceptualmente por encima de los heurísticos en el sentido que guían el diseño de éstos: así, al enfrentarnos a un problema de optimización, podemos escoger cualquiera de estos métodos para diseñar un algoritmo específico que lo resuelva aproximadamente. Pasaremos a describir una técnica meta heurística en particular: las llamadas técnicas GRASP

2.1 Algoritmos GRASP

Una técnica meta heurística es la de los algoritmos ***GRASP*** (del inglés ***Greedy Randomized Adaptative Search Procedure***). Esta técnica fue desarrollada por T. Feo y M. Resende a finales de los años 80 [10]. Mientras que el criterio goloso permitía seleccionar solamente el mejor valor de la función objetivo c a tratar, los algoritmos GRASP relajan o amplían este criterio de tal manera que, en vez de seleccionar un único elemento, forma un conjunto de elementos candidatos a ser parte del conjunto solución y que cumplen ciertas condiciones; es sobre este conjunto formado, que realizará una selección aleatoria de algún elemento. Tienen esta denominación por lo siguiente:

- Son procedimientos de búsqueda: dentro de un espacio de posibles soluciones se realizan búsquedas sin evaluar a todos los elementos del problema.
- Son voraces: porque en cada evaluación escoge a los mejores candidatos que cumplan ciertas condiciones.
- Son adaptativas: porque se adapta a la estructura de la instancia del problema que pretende resolver.

Son aleatorias: porque de entre las candidatas escoge aproximadamente al azar, aquellas que finalmente formarán parte de la solución

Procedimiento GRASP (Instancia del problema)
1. Leer (Instancia)
2. Mientras <no se cumpla condición de parada> hacer
2.1 Fase de Construcción (S_k)
2.2 Fase de mejoría (S_k)
Fin Mientras
3. Retornar (Mejor S_k)
Fin GRASP

Sobre este algoritmo podemos afirmar:

- Línea 1: se ingresan los datos que conforman la instancia del problema
- Línea 2: el proceso GRASP seguirá mientras no se cumpla una condición de parada. La condición de parada puede ser de varios tipos: optimalidad (el resultado obtenido presenta cierto grado de aproximación a la solución exacta o es lo suficientemente óptimo); número de ejecuciones realizadas (cantidad de iteraciones); tiempo de procesamiento (el algoritmo será ejecutado durante un determinado período de tiempo)
    - Líneas 2.1 y 2.2: se ejecutan las dos fases principales de un algoritmo GRASP, a seguir: ***etapa de construcción de la solución golosa adaptada y aleatoria; y la etapa de mejoría de la solución construida anteriormente.***
- Línea 3: finalmente se devuelve el resultado de la aplicación del algoritmo.

2.2 Fase de construcción GRASP

Esta etapa consiste en construir una solución golosa relajada o ampliada en donde exista más de una candidata a ser solución y a partir de la cual, se seleccionará aleatoriamente elementos para formar la solución final. Recordemos que el criterio goloso que buscaba optimizar la función objetivo podía ser planteado de la siguiente forma: $Mejor\{c(x) : x \in N\}$ en donde c es una función golosa.

Aquí el criterio GRASP modifica al goloso ampliándolo, de forma tal que se pueda construir una lista de varios elementos candidatos a ser solución. Primero determina los mejores y valores de c para luego formar la lista de candidatas como vemos en la siguiente figura:

$\beta = Mejor\{c(x) : x \in N\}$
$\tau = Peor\{c(x) : x \in N\}$

Figura 2: Valores extremos de la función c

El conjunto de candidatas RCL (del inglés: *Restricted Candidates List*): se formará con todos los elementos x de N que cumplan con la condición: $RCL = \{x \in N : \beta \leq c(x) \leq \beta + \alpha(\tau - \beta)\}$

Al parámetro α que aparece en la definición de RCL se le conoce como ***parámetro de relajación*** y es el responsable de tornar la solución que surgirá, más amplia que la solución voraz. Del conjunto RCL se toma un elemento al azar que pasa a formar parte del conjunto S, repitiéndose el proceso hasta que se hayan procesado todos los elementos de N.

Una estrategia aleatoria viene a ser particularmente útil cuando existen varias formas para realizar un paso dentro de un algoritmo en general, y donde resulta difícil garantizar totalmente que alguna de ellas es la mejor elección. Eso ocurre con el criterio goloso al no analizar más allá los efectos de la selección del mejor elemento de c en cada iteración.

El criterio GRASP indica que el candidato a ser parte del conjunto solución es así, un elemento seleccionado en forma aleatoria desde el conjunto RCL (no pertenece a S y su valor de la función objetivo no necesariamente es el mejor en ese instante).Como se puede apreciar, depende del valor de α empleado.

Según el valor que tome α, el comportamiento del criterio GRASP puede variar desde meramente aleatorio hasta volverse totalmente goloso. Veamos:

$\alpha = 0$	→	Criterio totalmente aleatorio
$\alpha = 1$	→	Criterio totalmente goloso
$0 < \alpha < 1$	→	Criterio GRASP convencional que requerirá de calibración adecuada a cada problema.

Figura 3: Posibles valores de la constante de relajación α

3. MÉTODOS EXISTENTES PARA RESOLVER EL PROBLEMA DE LA PROGRAMACIÓN DE TAREAS Y SUS VARIANTES

Resumiremos aquí partes importantes de trabajos que pretenden resolver el problema tanto de forma exacta como aproximada.

3.1 Métodos Existentes

Las soluciones existentes que pretenden resolver el problema de la planificación industrial en general pueden ser divididos en dos: métodos exactos y métodos aproximados. Los métodos exactos pretenden hallar un plan jerárquico único analizando todos los posibles ordenamientos de las tareas o procesos involucrados en la línea. Sin embargo, una estrategia de búsqueda y ordenamiento que analice todas las combinaciones posibles es computacionalmente cara y sólo funciona para algunos tipos de instancia.

Los métodos aproximados por su parte sí buscan resolver las variantes más complejas en las que interviene el comportamiento de tareas y máquinas como se mencionó en el apartado anterior. Ellos no analizan exhaustivamente todas las posibles combinaciones de patrones del problema, sino que más bien eligen los que cumplan determinados criterios. Obtienen finalmente, soluciones lo suficientemente óptimas para las instancias que resuelven, lo que justifica su uso.

Los sistemas existentes en el mercado que pueden resolver el problema, no aplican técnicas heurísticas o meta heurísticas para su desarrollo [6]. Este hecho se puede deber a la antigüedad de los mismos y los métodos empleados (exactos) [7][8]. Al no conseguirse soluciones exactas para muchos de los ámbitos industriales aplicados se obliga al uso de técnicas híbridas o de otro tipo que se ocupen de arrojar resultados aproximados y óptimos.

3.1.1. Job Scheduling

Dentro de la teoría de colas podemos encontrar la primera y más estudiada variante del scheduling: el problema de la planificación de trabajos en cola o atención en cola conocido en la literatura como job scheduling [9]. Este problema consiste en:

- Dado un lote finito de trabajos a ser procesados
- Dado un infinito número de procesadores (máquinas).
- Cada trabajo (job) esta caracterizado por un conjunto de operaciones convenientemente ordenadas; por su parte, las máquinas pueden procesar una única tarea a la vez, sin interrupciones.

El objetivo del JSP es encontrar una programación determinada que minimice el tiempo de procesamiento.

Existen numerosos algoritmos heurísticos planteados para resolver el problema del job scheduling, entre los que destacamos:

- Usando algoritmos GRASP: Binato, Hery y Resende [11]
- Usando algoritmos Genéticos: trabajos de Goncalves, De Magalahes y Resende [12] y Davis [13]
- Usando algoritmos de Ramificación y Acotación: trabajo de Brucker, Jurisch y Sievers [14]

3.1.2. Task Scheduling

Los algoritmos para esta variante pueden ser usados para resolver complicados problemas de planificación o programación de acciones y operaciones en células de trabajo. Se plantea un determinado número finito de máquinas y tareas y se busca, al igual que en el JSP, una programación adecuada donde se minimice el tiempo de procesamiento de lote o makespan.

Por la naturaleza de las máquinas y las tareas, puede hacerse la siguiente subdivisión vista anteriormente:

- Máquinas idénticas y tareas independientes

- Máquinas idénticas y tareas dependientes
- Máquinas diferentes y tareas independientes
- Máquinas diferentes y tareas dependientes: el modelo más complejo que será motivo de estudio en este trabajo.

Algunos algoritmos planteados son:
- Usando algoritmos voraces: Campello, Maculan [3] para máquinas idénticas.
- Usando algoritmos voraces: Tupia [15] para máquinas diferentes y tareas independientes
- Usando algoritmos GRASP: Tupia [15] para máquinas diferentes y tareas independientes

4. ALGORITMO GRASP PROPUESTO

Debemos partir del supuesto que se tiene una instancia de trabajo completa que incluya lo siguiente: cantidad de tareas y máquinas (N y M respectivamente); matriz de tiempos de ejecución T y lista de predecesoras por tarea.

4.1 Estructuras de Datos Usadas por el Algoritmo

Consideremos que en el lote hay al menos una tarea sin predecesoras que se convertirá en la inicial, así como no existen referencias circulares entre las predecesoras de las tareas que impidan su correcta programación. En la siguiente figura vemos las estructuras de datos que vamos a necesitar para la presentación del algoritmo:

N: número de tareas $J_1, J_2, ..., J_N$

M: número de máquinas $M_1, M_2, ..., M_M$

Matriz T: $[T_{ij}]_{MxN}$ **de tiempos de procesamiento, donde cada entrada representa el tiempo que se demora la máquina j-ésima en ejecutar la tarea i-ésima.**

Vector A: $[A_i]$ **de tiempos de procesamiento acumulado, donde cada entrada** A_i **es el tiempo de trabajo acumulado de la máquina** M_i

P_k **: Conjunto de tareas predecesoras de la tarea** J_k

Vector U: $[U_k]$ **de tiempos de finalización de cada tarea** J_k

Vector V: $[V_k]$ **de tiempos de finalización de las tareas predecesoras de** J_k**, donde se cumple que** $V_k = \max\{U_r\}, J_r \in P_k$

S_i **: Conjunto de tareas asignadas a la máquina** M_i

E: Conjunto de tareas programadas

C: Conjunto de tareas candidatas a ser programadas

Figura 4: Estructuras de datos usadas por el algoritmo

4.2 Consideraciones Generales

Vamos a plantear dos criterios de selección durante el desarrollo del algoritmo GRASP lo que nos va a llevar a generar dos constantes de relajación en vez de una sola:
- Un criterio GRASP de selección aleatorio de la mejor tarea a programarse, empleando la constante de relajación α.
- Un criterio de selección aleatorio de la mejor máquina que ejecutará la tarea seleccionada antes, empleando un parámetro θ adicional.

4.2.1 Criterio de Selección de la Mejor Tarea

Este criterio se basa en los mismos principios que el algoritmo voraz presentado antes:
- Identificar las tareas aptas para ser programadas: es decir, las que no han sido programadas aún y sus predecesoras ya han sido ejecutadas (o no presentan predecesoras).
- Para cada una de las tareas aptas, generar la misma lista que en el algoritmo goloso: establecer los tiempos de ejecución acumulado a partir de la finalización de la última tarea predecesora que se ejecutó.

- Obtener el menor elemento de cada lista y almacenarlos en otra lista de mínimos locales. De esta nueva lista de mínimos locales seleccionaremos: el máximo y el mínimo valor las variables: *peor y mejor* respectivamente.
- Formaremos la lista de tareas candidatas RCL analizando cada entrada de la lista de mínimos locales: si la entrada correspondiente cumple con estar dentro del intervalo **[mejor, mejor + α* (peor-mejor)],** entonces pasa a formar parte de RCL.
- Se escoge una tarea al azar de las que constituyen RCL.

4.2.2 Criterio Goloso de Selección de la Mejor Máquina

Una vez seleccionada una tarea desde RCL, procedemos a buscar la mejor máquina que la pueda ejecutar. Los pasos a seguir son los siguientes:

- Se forma nuevamente el vector de tiempos acumulados en función de las predecesoras de la tarea j-ésima que está siendo objeto de análisis.
- Se encuentran los valores máximos y mínimos de en las variables: *peor y mejor* respectivamente.
- Procedemos a formar la lista de máquinas candidatas ***MCL***: toda máquina que ejecute a la tarea j-ésima en un tiempo que se encuentre en el intervalo **[mejor, mejor + θ* (peor-mejor)]** forma parte de MCL.
- Igualmente seleccionamos al azar una de las máquinas, que será la ejecutora de la tarea j-ésima.

4.3 Presentación del algoritmo

Inicio AlgoritmoGRASP_Construccion (M, N, T, A, α, θ)

1. Leer N, M, α, θ, $J_1, J_2, \ldots, J_N$

2. Para i:1 a N, hacer

Para j: 1 a M, hacer

Leer T_{ij}

3. Para i:1 a M, hacer

Inicio

A_i **= 0**

$S_i = \phi$

Fin Para

4. Para k: 1 a N, hacer

Inicio

U_k **= 0**

V_k **= 0**

Fin Para

5. E $= \phi$

6. Mientras $|E| \neq N$ **hacer**

Inicio

6.1 C = ϕ

6.2 mejor = $+\infty$

6.3 peor = 0

6.4 Para ℓ: 1 a N, hacer

Si ($P_\ell \subseteq E$) $\wedge$ ($J_\ell \notin E$) $\Rightarrow$ $C = C \cup \{J_\ell\}$

6.5 $B_{min} = \phi$

6.6 Para cada $J_\ell \in C$ **hacer**

Inicio

6.6.1 $V_l = \max_{J_l \in P_k} \{U_l\}$

6.6.2 $B_{min} = B_{min} \cup Min_{p \in [1,M]} \{T_{pl} + \max\{A_p, V_l\}\}$

Fin Para

{Selección de la mejor tarea. Formación de RCL}

6.7 mejor = $Min\{B_{\min}\}$

6.8 peor = $Max\{B_{\min}\}$

6.9 $RCL = \phi$

6.10 Para cada $J_\ell \in C$ **hacer**

Si $Min_{p \in [1,M]} \{T_{pl} + \max\{A_p, V_l\}\} \in$ **[mejor, mejor + α* (peor-mejor)]** $\Rightarrow$

$RCL = RCL \cup \{J_\ell\}$

6.11 k = $ArgAleatorio_{J_l \in RCL} \{RCL\}$

{Selección de la mejor máquina}

6.12 $MCL = \phi$

6.13 mejor = $Min_{p \in [1,M]} \{T_{pk} + \max\{A_p, V_k\}\}$

6.14 peor $= Max_{p \in [1,M]} \{T_{pk} + \max\{A_p, V_k\}\}$

6.15 Para i: 1 a M hacer

Si $T_{ik} + \max\{A_i, V_k\} \in$ **[mejor, mejor + θ* (peor-mejor)]** $\Rightarrow$

$MCL = MCL \cup \{M_i\}$

6.16 i = $ArgAleatorio_{M_i \in MCL} \{MCL\}$

6.17 $S_i = S_i \cup \{J_k\}$

6.18 $E = E \cup \{J_k\}$

6.19 $A_i = T_{ik} + \max\{A_i, V_k\}$

6.20 $U_k = A_i$

Fin Mientras

7. makespan = $\max_{k \to 1..M} A_k$

8. Retornar makespan, $S_i \ \forall i \in [1, M]$

Fin AlgoritmoGRASP_Construcción

Figura 5: Algoritmo GRASP Construcción.

4.3.1. Comentarios sobre el algoritmo GRASP propuesto

- Líneas 1-5: ingreso de las variables e inicialización de las estructuras de datos necesarias para la instancia de trabajo: N, M, T, A, U, V, E, S_i para cada máquina, α, θ, etc.
- Línea 6: al igual que en el algoritmo voraz, el proceso acaba cuando en el conjunto E se encuentren todas las tareas (E conjunto de las tareas programadas)
    - Línea 6.1: se inicializa en vacío la lista de tareas aptas C
    - Línea 6.2: se inicializan las variables mejor y peor con las que se trabajarán los intervalos de los criterios de relajación.
    - Línea 6.4: se forma la lista de tareas aptas C
    - Línea 6.5: se inicializa en vacío la lista de mínimos locales B_{min}
    - Líneas 6.6 – 6.6.2: se actualiza las entradas correspondientes de la lista V; se forma la lista B_{min} agregándole cada menor elemento de la lista de tiempos acumulados de cada tarea.
    - Líneas 6.7-6.8: se le asignan los valores máximos y mínimos de B_{min} a las variables peor y mejor respectivamente.
    - Línea 6.9: se inicializa en vacío la lista RCL
    - Líneas 6.10-6.11: se forma la lista RCL cuando se cumpla la condición $Min_{p\in[1,M]}\{T_{pl} + \max\{A_p, V_l\}\} \in$ [mejor, mejor + α* (peor-mejor)]; luego se escoge un elemento al azar de ésta lista (k)
    - Líneas 6.12-6.16: se escoge el mínimo y el máximo tiempo de ejecución para la tarea escogida en 6.11. Se formará MCL a partir de las máquinas que ejecuten dicha tarea cumpliendo la condición: $T_{ik} + \max\{A_i, V_k\} \in$ [mejor, mejor + θ* (peor-mejor)]. Finalmente también se escoge una máquina de forma aleatoria (i)
    - Líneas 6.17-6.20: se actualizan las estructuras E, A, U y S_i donde corresponda.
- Línea 7: se determina el makespan como la mayor entrada de A
- Línea 8: se devuelven los resultados de asignación hallados.

5. EXPERIENCIAS NUMÉRICAS

Las instancias de prueba con las que se probó el algoritmo están conformados por la cantidad de máquinas, tareas y por una matriz de tiempos de ejecución. Los valores que se manejarán serán los siguientes:

- Número de tareas N: en el intervalo 100…250 tomando como puntos de referencia los valores de 100, 150, 200, 250.
- Número de máquinas M: un máximo de 50 máquinas tomando como puntos de referencia los valores de 12, 25, 37, 50
- Matriz de tiempos de proceso: se generará de forma aleatoria con valores entre 1 y 100 unidades de tiempo.[1]

En total tenemos 16 combinaciones para las combinaciones ***máquinas-tareas***. De la misma forma, para cada combinación se generarán 10 instancias distintas, lo que arroja un total de ***160 problemas test*** realizados**.** Ante la no existencia en la literatura de instancias de prueba predeterminados para el problema en cuestión, decidimos enfrentar los resultados con los que ofrecía un ***algoritmo goloso o voraz*** [5], diseñado por los mismos autores. ***Igualmente*** se procedió a crear instancias lo suficientemente pequeñas (manejables) para probarlas y obtener su solución exacta planteándolas como problemas de la programación lineal, de la mano del modelo matemático presentado en anteriormente.

Así pues, creamos instancias que iban a ser resueltas por medio del programa **LINDO** [16], para obtener sus soluciones exactas. Luego, compararíamos esas soluciones exactas con los resultados arrojados por el algoritmo propuesto. Las instancias usadas para ser resueltas de forma exacta han tenido los siguientes tamaños (recuérdese que N es el número de tareas y M el número de máquinas):

[1] : Notemos que un tiempo de ejecución muy alto (+∞) puede interpretarse como que la máquina no ejecuta una tarea determinada. Usar en este caso un tiempo de ejecución igual a 0 podría confundirse como que la máquina ejecuta *tan rápido la tarea* que se puede asumir que lo hace de forma instantánea, sin ocupar tiempo.

N	M
6	3
8	3
12	3
15	3
10	5
15	5
20	5
25	5

Tabla 1: Tamaños de las instancias de prueba exacta

Máquinas\ Tareas	GOLOSO	GRASP C			EFICIENCIA
	Makespan	Makespan	α	θ	%
100 \ 12	213.3	193.7	0.26	0.04	9.19%
100 \ 25	113.7	108.6	0.24	0.01	4.49%
100 \ 37	76.5	74.3	0.155	0.01	2.88%
100 \ 50	59.1	57.8	0.145	0.01	2.20%
150 \ 12	283.1	258.2	0.25	0.01	8.80%
150 \ 25	125.2	114	0.2055	0.01	8.95%
150 \ 37	86.1	84.1	0.155	0.01	2.32%
150 \ 50	68.6	67.4	0.115	0.01	1.75%
200 \ 12	325.9	307	0.07	0.01	5.80%
200 \ 25	127	117	0.247	0.01	7.87%
200 \ 37	109.8	105.2	0.112	0.01	4.19%
200 \ 50	76.4	74.6	0.155	0.01	2.36%
250 \ 12	411.8	377.1	0.15	0.01	8.43%
250 \ 25	183.5	168.2	0.28	0.01	8.34%
250 \ 37	125.3	119.6	0.26	0.01	4.55%
250 \ 50	96.5	91.1	0.123	0.01	5.60%
Eficiencia algoritmo GRASP sobre el algoritmo Goloso					**5.48%**

Tabla 2: Eficiencia entre algoritmo GRASP y goloso para instancias grandes

Matriz	N	M	Resultado del LINDO	Algoritmo voraz	Algoritmo GRASP	% Exacto/Voraz	% Exacto/GRASP	a	θ
6x3_0	6	3	41	41	41	0.00%	0.00%	0.01	0.01
6x3_1	6	3	109	109	109	0.00%	0.00%	0.01	0.01
6x3_2	6	3	98	108	98	10.20%	9.26%	0.01	0.01
8x3_0	8	3	84	83	83	1.19%	0.00%	0.01	0.01
8x3_1	8	3	153	180	153	17.65%	15.00%	0.01	0.01
8x3_2	8	3	94	94	94	0.00%	0.00%	0.01	0.01
12x3_0	12	3	252	252	252	0.00%	0.00%	0.01	0.01
12x3_1	12	3	238	253	238	6.30%	5.93%	0.01	0.02
12x3_2	12	3	168	168	165	0.00%	1.79%	0.01	0.45
15x3_0	15	3	300	301	262	0.33%	12.96%	0.01	0.41
15x3_1	15	3	193	193	181	0.00%	6.22%	0.47	0.37
15x3_2	15	3	290	292	241	0.69%	17.47%	0.43	0.02
Diferencia						**3.03%**	**5.72%**		

Tabla 3: Eficiencia de la solución golosa para instancias con N igual a 3

Matriz	N	M	Resultado del LINDO	Algoritmo voraz	Algoritmo GRASP	% Exacto/Voraz	% Exacto/GRASP	a	θ
10x5_0	10	5	95	95	95	0.00%	0.00%	0.01	0.01
10x5_1	10	5	106	109	109	2.83%	0.00%	0.01	0.01
10x5_2	10	5	122	123	122	0.82%	0.81%	0.001	0.02
15x5_0	15	5	139	139	119	0.00%	14.39%	0.01	0.09
15x5_1	15	5	157	157	149	0.00%	5.10%	0.06	0.17
15x5_2	15	5	97	103	96	6.19%	6.80%	0.01	0.04
20x5_0	20	5	210	210	189	0.00%	10.00%	0.21	0.01
20x5_1	20	5	132	132	119	0.00%	9.85%	0.01	0.08
20x5_2	20	5	155	156	155	0.65%	0.64%	0.09	0.01
25x5_0	25	5	255	264	255	3.53%	3.41%	0.34	0.07
25x5_1	25	5	249	283	249	13.65%	12.01%	0.37	0.19
25x5_2	25	5	219	219	211	0.00%	3.65%	0.19	0.29
Diferencia						**2.31%**	**5.55%**		

Tabla 4: Eficiencia de la solución golosa para instancias con N igual a 5

6. CONCLUSIONES

Los sistemas existentes en el mercado que pueden ser considerados como planificadores de tareas, no han aplicado técnicas meta heurísticas sino más bien métodos exactos. No buscan la optimización del ordenamiento de las tareas a ejecutar, sino más bien dan énfasis a encontrar planes factibles de ser ejecutados.

Tampoco existen sistemas que planifiquen una línea de producción completa considerando entre otras cosas: el desplazamiento de los productos, la ejecución de las operaciones, la disposición de la maquinaria (layout) y de las facilidades (instalaciones), potenciales reducciones de los tiempos muertos, etc.

Este panorama le imprime un carácter novedoso a la investigación realizada ya que, si bien en la actualidad las técnicas GRASP son frecuentemente empleadas para la resolución de distintas clases de problemas, hasta ahora no se había planteado una GRASP para resolver el problema del task scheduling para tareas dependientes ejecutadas en máquinas diferentes.

Como parte de los nuevos planteamientos presentados en los algoritmos, debemos centrarnos en:

- Un criterio de selección de la mejor tarea a programar
- Un criterio de selección de la mejor máquina a ejecutar la tarea seleccionada con el criterio anterior.

Ambos criterios se amoldaron al tipo de algoritmo presentado: los criterios eran voraces en el caso del algoritmo voraz (una selección inmodificable); o eran adaptativos y aleatorios como en el caso del algoritmo GRASP. En la literatura no existen algoritmos GRASP que consideren un doble criterio de relajamiento al momento de hacer las selecciones correspondientes: las GRASP convencionales, solo formaban una lista RCL de candidatas para las tareas a ser ejecutadas.

Nuestro trabajo amplía esa visión en el caso del algoritmo GRASP, al proporcionarle un componente adaptativo y aleatorio a la selección de la mejor máquina ejecutora; lo que al final afectó a la eficiencia del algoritmo.

- El algoritmo GRASP mejora el 100% de los casos al resultado del algoritmo voraz para las instancias de prueba suficientemente elevadas (proporción 4 a 1, 5 a 1). En instancias pequeñas como mínimo lo iguala al voraz o es superado en un bajísimo porcentaje.
- El porcentaje de mejora del algoritmo GRASP frente al algoritmo goloso es de 5% en promedio.
- El tiempo de ejecución de la fase de construcción GRASP se basa en la cantidad de iteraciones a realizarse: para las instancias mayores (250 x 50) se trataron de realizar más de 1000 iteraciones, pero aparentemente la configuración del hardware de prueba no lo permitió. Esto nos lleva a inferir que, con una configuración de equipo más potente sí se podría pasar esta barrera. El tiempo de CPU empleado no pasaba de los 3 minutos lo que da un promedio también bajo en la duración de la fase de construcción.

La etapa de mejoría GRASP, se limitó a no más que un intercambio de los valores encontrados en la etapa anterior de construcción. ***No brindó mejoras en ninguno de los casos***, lo que nos conduce a pensar que la etapa de construcción es lo suficientemente robusta, para requerir las permutaciones que busquen mejorar su resultado. Es por eso que no se utilizó la mejora.

Referencias

[1]: Miller G., Galanter E. Plans and the Structure of Behavior, Editorial Holt, New York-USA (1960)

[2]: Drozdowski M. Scheduling multiprocessor tasks—an overview, European Journal . Operation Research 94 (1996) 215–230.

[3]: Campello R., Maculan N. Algoritmos e Heurísticas Desenvolvimiento e avaliaçao de performance, Apolo Nacional Editores, Brasil (1992).

[4]: Pinedo M. Scheduling, Theory, Algorithms and Systems, Prentice Hall (1995 y 2002)

[5]: Cormen T.; Leiserson Ch.; Rivest R. Introduction to Algorithms (2da edición), MIT Press, Editorial McGraw Hill, Inglaterra, 2001

[6]: Rauch W. Aplicaciones de la inteligencia Artificial en la actividad empresarial, la ciencia y la industria - tomo II. Editorial Díaz de Santos, Madrid España pp. 685 (1989).

[7]: Kumara P. Artificial Intelligence: Manufacturing theory and practice. Editorial NorthCross Institute of industrial Engineers, USA pp. 686 (1988).

[8]: Blum A., Furst M. Fast Planning through Plan-graph Analysis. Article of memories from 14th International Joint Conference on Artificial Intelligence, pp. 1636-1642. Ediciones Morgan- Kaufmann - USA (1995).

[9]: Suárez R., Bautista J., Mateo M. Secuenciación de tareas de ensamblado con recursos limitados mediante algoritmos de exploración de entornos, www.upc.es/ sol.upc.es/~suarez/pub.html Instituto de Organización y Control de Sistemas Industriales Universidad Politécnica de Cataluña (1999).

[10]: Feo T., Resende M., Greedy Randomized Adaptive Search Procedure Journal of Global Optimization, número 6, pp. 109-133 (1995).

[11]: Binato S., Hery W., Loewenstern D. y Resende M. A GRASP for Job Scheduling. Technical Report N° 00.6.1 AT&T Labs Research (1999-200)

[12]: Gonçalves J., Magalhäes J., Resende M. A Hibrid Genetic algorith for the Jos Shop Scheduling AT&T Labs Research Technical Report TD-5EAL6J (2002).

[13]: Davis L. Job shop scheduling with genetic algorithms. First International Conference on Genetic Algorithms and their Applications, pp. 136-140. Morgan – Kaufmann USA, (1985).

[14]: Brucker P., Jurisch B. and Sievers B. A branch and bound algorithm for the job-shop scheduling problem. Journal of Discrete Applied Mathematics, número 49, pp. 105-127 (1994).

[15]: Tupia, M. Un algoritmo Goloso Adaptativo y Randómico para resolver el problema de la Programación de Tareas independientes en máquinas homogéneas, Tesis presentada para optar por el título de Ingeniero Informático, Pontificia Universidad Católica del Perú (2001).

[16]: Scharage L. 1997. Optimization modeling with LINDO. Duxbury Press. USA.