RISI **1**(1), 9-18 (2004)
Rev. investig. sist. inform.
Facultad de Ingeniería de Sistemas e Informática
Universidad Nacional Mayor de San Marcos
ISSN: 1815-0268 (impreso)

ARTÍCULOS

UN ALGORITMO VORAZ PARA RESOLVER EL PROBLEMA DE LA PROGRAMACIÓN DE TAREAS DEPENDIENTES EN MÁQUINAS DIFERENTES

Manuel Tupia*, David Mauricio**

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 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* definida como 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 un algoritmo voraz para resolver dicha variante del problema del *task scheduling*.

Palabras clave: algoritmos golosos, programación de tareas, optimización combinatoria, heurísticas, Inteligencia Artificial.

A GREEDY ALGORITHM FOR TASK SCHEDULING PROBLEM

ABSTRACT

The industrial planning has experimented great advances since its beginning for a middle of 20th century. It has been demonstrated its applications importance into the several industrial where its involved even though the difficult of design exact algorithms that resolved the variants. It has been applied heuristics methods for the planning problems due their high complexity; especially Artificial Intelligence when develop new strategies for resolving one of the most important variant, called *task scheduling*.

It is able to define the task scheduling problem like: a set of N production line 's tasks and M machines, which ones can execute those tasks. The goal is to find an order that minimize the accumulated execution time, known as *makespan*. This paper presents a meta heuristic strategy GRASP for the problem of programming of dependent tasks in different single machines.

Keywords: greedy myopic algorithms, task scheduling, combinatorial optimization, heuristics, Artificial Intelligence.

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 microelectrónica [2]. Ese tipo de problema puede definirse, desde el punto de vista de la optimización combinatoria [3], como sigue:

* Pontificia Universidad Católica del Perú, Departamento de Ingeniería, Sección Informática, Lima-Perú. E-mail: tupia.mf@pucp.edu.pe
** Universidad Nacional Mayor de San Marcos, Instituto de Investigación de la Facultad de Ingeniería de Sistemas e Informática, Lima-Perú. E-mail: dms@terra.com

Dadas M máquinas (consideradas procesadores) y N tareas con T_{ij} unidades de tiempo de duración de cada tarea i-ésima ejecutada en la máquina j-ésima. Se desea 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.

El problema del *scheduling* presenta una serie de variantes dependiendo de la naturaleza y el comportamiento; tanto de las tareas como 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 son diferentes*.

En esta variante cada tarea presenta una lista de tareas que la preceden y para ser ejecutada deben esperar el procesamiento de dicha lista en su totalidad. A esta situación hay que agregarle la característica de heterogeneidad de las máquinas: cada tarea demora tiempos distintos de ejecución en cada máquina. El objetivo será minimizar el tiempo acumulado de ejecución de las máquinas, conocido en la literatura como *makespan* [3].

Al observar el estado del arte del problema, vemos que tanto su aplicación práctica de forma directa en la industria como su importancia académica, al ser un problema NP-difícil, se justifica en 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 y en donde:

- X_0 representa al *makespan*
- X_{ij} será 0 si la máquina j-ésima no ejecuta la tarea i-ésima y 1 en caso contrario

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

Figura N.º 1. Modelo matemático del *task scheduling*

II. 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 del *scheduling* en general tanto de forma exacta como aproximada.

2.1 Métodos existentes

Las soluciones existentes que pretenden resolver el problema de la planificación industrial en general pueden ser divididas en dos grupos: *métodos exactos y métodos aproximados*.

Los *métodos exactos* [6,7,8,9,10] pretenden hallar un plan jerárquico único analizando todos los posibles ordenamientos de las tareas o procesos involucrados en la línea de producción (exploración exhaustiva). Sin embargo, una estrategia de búsqueda y ordenamiento que analice todas las combinaciones posibles es computacionalmente cara y sólo funciona para algunos tipos (tamaños) de instancia.

Los *métodos aproximados* [3,4,5], 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. Dichos métodos 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 buenas para las instancias que resuelven, lo que justifica su uso.

Seguidamente presentamos los métodos de solución existentes para las variantes más conocidas del *scheduling*.

2.1.1. Métodos heurísticos para resolver la variante del *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* [11]. 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 ramificación y acotación: Bard y Feo [12,13] presentan el conocido método de búsqueda en grafos dirigidos de ramificación y acotación, con el cual proceden a recorrer un árbol cuyos nodos son secuencias de operaciones en líneas de manufacturas. No consideran líneas con penalidades por retrasos ni por los pasos de las tareas de una máquina a otra; de allí la aplicabilidad directa de un método de barrido de árboles en el proceso de selección de instrumentos y trayectorias. Similares trabajos fueron presentados por Brucker, Jurisch y Sievers [14].
- Usando algoritmos GRASP: Binato, Hery y Resende [15] incorporan dos nuevos conceptos en el desarrollo de una GRASP convencional para el JSP: un procedimiento de intensificación estratégica (otra forma de distribución probabilística) para crear candidatas a solución y una técnica POP (del inglés *Proximate Optimality Principle*) también en la fase de construcción. Ambos conceptos ya habían sido aplicados para resolver el problema de la asignación cuadrática.
- Usando algoritmos genéticos: Trabajos de Goncalves, De Magalahes y Resende [16] y Davis [17]. La representación de los cromosomas se basa en tasas del elitismo aleatorias; los autores pretenden asemejar el comportamiento de un entorno de trabajo variable en una línea de producción flexible. Por otro lado, la investigación de Davis [18] consistió básicamente en la división de las tareas en cromosomas muy pequeños de tal forma que las poblaciones generadas sean lo más variadas posibles.
- Usando búsqueda Tabú: Taillard [19] consideraba como vecindades para la aplicación de las listas Tabú al movimiento de una candidata a operación i-ésima (tarea) a programar en una máquina hacia otra máquina, siendo la lista Tabú aquella matriz formada por las operaciones y las máquinas donde ejecutarlas; otros trabajos interesantes son los de Chambers y Barnes [20] que consideraban una ruta flexible; ruta flexible es aquella en la que existen máquinas que pueden operar más de un tipo de operación, extendiendo la definición del JSP; y finalmente los trabajos de Subramani [21] donde aplica los conceptos de JSP sobre las técnicas de Grid (Grid Computing) en la programación de tareas de súper computadoras. Las estrategias Grid consisten en la programación de varios trabajos recurrentes y simultáneos que aparecen en la atención de los procesadores.

2.1.2. Métodos heurísticos para resolver la variante del *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. El planteamiento que realizan los autores es a partir de la definición del problema como uno de programación discreta (posible al ser de la clase NP-difícil), como se vio anteriormente.

- Usando algoritmos voraces: Tupia [22] para máquinas diferentes y tareas independientes. El autor presenta el caso de las máquinas diferentes y tareas independientes. Se adaptó el modelo de Campello y Maculan considerando que habían tiempos distintos de ejecución para cada máquina: es decir aparece el concepto matricial de que es el tiempo que se demora la tarea i-ésima en ser ejecutada por la máquina j-ésima.
- Usando algoritmos GRASP: Tupia [22] presenta el caso de las máquinas diferentes y tareas independientes. En este trabajo, el autor amplió el criterio voraz del algoritmo anterior aplicando las fases convencionales de la técnica GRASP y mejorando en cerca del 10% los resultados del algoritmo voraz para instancias de hasta 12500 variables (250 tareas por 50 máquinas).

2.1.3 Métodos heurísticos para resolver el problema de la planificación de tareas en tiempo real (STR)

- Planificación basada en el reloj (basada en el tiempo) [23]: El plan de ejecución se calcula fuera de línea y se basa en el conocimiento de los tiempos de inicio y de cómputo de todos los trabajos. El plan de ejecución está en una tabla y no es concurrente.
- Planificación *round robin* [24] con y sin prioridades: Las tareas tienen prioridades asignadas por el usuario (fuera de línea). Dentro de cada prioridad, las tareas se planifican en *round robin* con un determinado tiempo asignado a cada una de ellas (quantum).
- Planificación basada en prioridades [25]: Las prioridades las asigna el algoritmo de planificación. La tarea con mayor prioridad se ejecuta en cualquier instante.
- Algoritmo *Dual Priority* y sus adaptaciones [26] para la asignación dinámica de tareas en entornos de multiprocesador con memoria compartida.

2.1.4 Métodos heurísticos para resolver la variante del *flow-shop scheduling*

- Usando algoritmos GRASP: Los trabajos de Feo, Sarathy, McGahan [27] para máquinas simples consistían en asignar un costo a la ejecución de cada trabajo. Representaba el costo de ejecutar inmediatamente la tarea j después de la tarea i; luego agregaba el tiempo de ejecución para cada trabajo y el concepto de penalidad por retrasos en la línea. Se pretendía aplicar las técnicas GRASP para minimizar la suma de las penalidades. En la misma línea investigativa, Ríos y Bard [28] plantean una GRASP para secuenciamiento de líneas de flujo de trabajo: encontrar una secuencia de N tareas en un ambiente de flow-shop. Presentaron dos heurísticas: la primera basada en el trabajo de Nawaz [29], quien busca construir una secuencia factible de ejecución, y otro algoritmo propuesto es un procedimiento GRASP cuya fase de construcción se basa en el algoritmo anterior; otro trabajo interesante de Feo y Venkatraman [30] para un único tipo de máquina con penalización de tiempo de flujo y costos por terminación adelantada.
- Usando heurísticas de búsqueda Tabú: Encontramos el trabajo de Acero y Torres [31] para líneas de producción flexible. Propusieron una representación natural de una solución factible por medio de un vector de dimensiones 2 x N x M componentes (siendo N el número de trabajos y M el número de etapas).
- Usando algoritmos voraces: Trabajos de Chand y Scheneeberger [32] para líneas con máquinas simples y trabajos sin demoras.

III. ALGORITMOS VORACES

Los algoritmos golosos-miopes o voraces (del inglés greedy-myopic) reciben esta denominación por las siguientes razones [34]:

- Es goloso o voraz porque siempre escoge el mejor candidato para formar parte de la solución: aquel que tenga mejor valor de la función objetivo, lo que constituye el cumplimiento de cierto criterio goloso de selección.
- Es miope porque esta elección es única e inmodificable dado que no analiza más allá los efectos de haber seleccionado un elemento como parte de la solución. No deshacen una selección ya realizada: una vez incorporado un elemento a la solución permanece hasta el final y cada vez que un candidato es rechazado, lo es permanentemente.

Se sabe sin embargo, que la calidad de los algoritmos golosos está en relación con las características de las instancias que pretenden resolver: puede arrojar muy buenos resultados para determinadas instancias del problema pero para otras no. Otro inconveniente que presentan es que

se estancan en óptimos locales de las funciones que pretenden optimizar y quizá no analizan vecindades más allá del criterio goloso por lo que pueden estar dejando de considerar al óptimo global. Analicemos la siguiente figura:

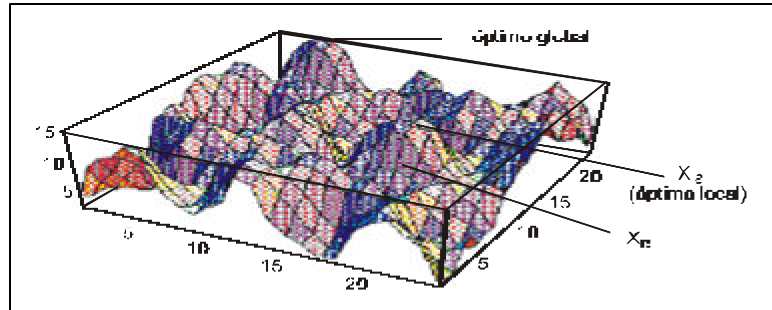

Figura 2: Miopía de los algoritmos voraces y estancamiento en óptimos locales.

Puede verse que en valores de entre 5 y 10, la función a optimizar arroja óptimos locales pero que no puede apreciar más allá, la presencia de un mejor valor global pico. Vamos a presentar un algoritmo voraz planteado en su forma genérica. Previamente consideremos las siguientes variables:

E = {e1, e2, e3, e4,.., eN} es una colección de N objetos o variables del problema.

F: es una colección de subconjuntos de elementos de E que cumple con una determinada propiedad

c: E à R es la función a optimizar.

S: conjunto solución.

Figura N.º 3: Estructuras de datos para un algoritmo voraz.

El algoritmo voraz en su forma general sería como sigue:

```
Algoritmo Voraz General(N,c,S,E,F)
Inicio
  0. Leer N, c, E, F
  1. Ordenar bajo algún criterio, los elementos de
     E de acuerdo al valor de c
  2. S:= ∅
  3. Para i: 1 a N, hacer
  3.1. Si S ∪ {e_i} es una solución viable entonces S =
       S ∪ {e_i}
Fin Voraz General
```

Figura N.º 4: Algoritmo voraz en su formato genérico.

IV. ALGORITMO VORAZ 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 N.º 5. Estructuras de datos usadas por el algoritmo.

4.2 Consideraciones Generales

Los lineamientos generales de este algoritmo son los siguientes:

- Seleccionar las tareas aptas o candidatas a ser ejecutadas siguiendo los siguientes criterios:
  - √ El criterio goloso de selección de la mejor tarea a ejecutar: Se escoge aquella tarea cuyo tiempo de ejecución sea el más bajo para evitar perturbaciones.
  - √ El criterio goloso de selección de la mejor máquina ejecutora: La tarea se asigna a la máquina que la ejecuta en menor tiempo.
- Una tarea se considerará apta para ser ejecutada si *todas sus predecesoras ya han sido programadas o carece de ellas.*
- En el caso particular de las tareas cuyas predecesoras ya han sido ejecutadas en su totalidad seguiremos así:
  - √ Como la tarea J_i se debe ejecutar después de todas sus predecesoras, debemos ubicar la predecesora que finaliza más tarde (termina de ser ejecutada en el mayor tiempo).
  - √ La mejor máquina M_k se elegirá de entre los M valores posibles: será aquella que minimice el valor de la suma de la entrada de la matriz de tiempos de ejecución T_{ij} con el máximo de entre la entrada correspondiente al vector de tiempos acumulados respectiva A_k y al tiempo de finalización de la última de las predecesoras de: J_i: $V_k = \max\{U_r\}, J_r \in P_k$
  - √ A la *mejor máquina* le asignamos tarea J, actualizando al vector A en la entrada correspondiente a M_k.
- Se repite el proceso para el resto de tareas.
- Finalmente el mayor tiempo o entrada de la lista A será el *makespan*.

4.2.1 Criterio de selección de la mejor tarea

Sean por ejemplo, las tareas y tareas sin predecesoras; se tienen 3 máquinas M_1, M_2, M_3 y un extracto de la matriz de tiempos de ejecución T en la tabla N.° 1:

Tabla N.° 1. Extracto de una matriz de tiempos de ejecución T.

	M1	M2	M3
J1	10	12	11
J2	23	9	8

Sin perder generalidad y basándonos en el extracto de la tabla anterior formaremos dos listas[1] con los tiempos acumulados de trabajo:

- Lista para la tarea J_1: {10, 12, 11}. Determinamos el mínimo: 10
- Lista para la tarea J_2: {23, 9, 8}. Determinamos el mínimo: 8

A partir de los datos anteriores, nuestro *criterio de selección de la mejor tarea* escogería la tarea para ser ejecutada en primer lugar, debido a que su tiempo de acumulado goloso es más bajo (10>8) que el tiempo acumulado.

4.2.2 Criterio goloso de selección de la mejor máquina

Por su parte, el criterio goloso de selección de la mejor máquina será el de aquella que ejecuta la tarea apta para ser programada (elegida desde el criterio anterior) en el menor tiempo. Luego, siguiendo el ejemplo: para J_2 tenemos que el menor tiempo de ejecución es 8 unidades, correspondiente a la máquina M_3. A la máquina M_3 se le asignaría la ejecución de la tarea J_2.

4.3 Presentación del algoritmo propuesto

Inicio Algoritmo_Voraz_HETDEP(M, N, T , A, S, U, V)

1. Leer e inicializar N, M, $J_1, J_2, ..., J_N$ **, T, A, S, U, V**

2. E $= \phi$

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

Inicio

3.1 C $= \phi$

3.2 Para l **: 1 a N, hacer**

Si $(P_l \subseteq E) \wedge (J_l \notin E) \Rightarrow C = C \cup \{J_l\}$

3.3 Para cada $J_l \in C$ **hacer**

Inicio

3.3.1 $V_l = \max_{J_l \in P_k}\{U_l\}$

3.3.2 k $= ArgMin_{\substack{i \in [1,M] \\ J_l \in C}}\{T_{il} + \max\{A_i, V_l\}\}$

3.3.3 i $= ArgMin_{p \in [1,M]}\{T_{pk} + \max\{A_p, V_k\}\}$

Fin para

3.4 $S_i = S_i \cup \{J_k\}$

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

3.6 $A_i = T_{ik} + \max\{A_i, V_k\}$

3.7 $U_k = A_i$

[1] La formación de las listas no es tan trivial como la simple copia directa de los datos de la matriz T. Más adelante se explica el proceso correspondiente.

Fin Mientras
4. makespan = $\max_{k \to 1..M} A_k$
5. Retornar makespan, $S_i\ \forall i \in [1, M]$
Fin Algoritmo_Voraz_HETDEP.

Figura N.º 6. Algoritmo voraz propuesto.

4.3.1 Comentarios

- Línea 1: Se inicializan las estructuras de datos.
- Línea 2: Se inicializa el conjunto E en vacío porque ninguna tarea ha sido programada aún.
- Línea 6: El proceso de programación terminará cuando en el conjunto E estén contenidas las N tareas (cardinalidad de E igual a N).
  - Línea 3.1: El conjunto C de tareas candidatas a ser programadas (aptas) se inicia en vacío para cada iteración.
  - Línea 3.2: Se procede a determinar las tareas candidatas. Sabemos que una tarea $\mathbf{J}_1$ formará parte de C si sus predecesoras ya han sido ejecutadas (incluidas en E) o no presenta predecesoras (vacío incluido en E) y si además dicha $\mathbf{J}_1$ no ha sido programada (no pertenece a E).
  - Línea 3.3: Para las tareas candidata de C seleccionaremos la mejor, así como la mejor máquina ejecutora siguiendo los criterios explicados en apartados anteriores.
    - √ Línea 3.3.1: Actualizamos al vector V. Escogemos la predecesora de la tarea $\mathbf{J}_1$ que finaliza más tarde (mayor entrada de U de las predecesoras de $\mathbf{J}_1$).
    - √ Línea 3.3.2: Criterio de selección de la mejor tarea: para cada tarea $\mathbf{J}_1$ en C, formamos una lista de M elementos conteniendo el tiempo de ejecución T_{il} para cada M_i máquina, más el máximo valor entre la entrada correspondiente de A_i y el tiempo de la última predecesora de $\mathbf{J}_{-l}$ en ser ejecutada (V_{-l}). Esto repetimos, debido a que $\mathbf{J}_{-l}$ debe programarse después de la finalización de *todas* sus predecesoras (en particular de la que acaba al final). De cada lista obtenemos el mínimo valor, y de entre todos los mínimos hallados (uno por cada tarea $\mathbf{J}_1$ candidata) escogemos el menor. Aquella tarea que posee ese tiempo mínimo (el mejor de los mínimos locales), será la mejor a programar (*argumento k*).
    - √ Línea 3.3.3: Criterio de selección de la mejor máquina: una vez conocida la tarea k-ésima (k calculado en la línea anterior) a programar, usando el criterio goloso de mejor a aquella máquina que ejecute a k en el menor tiempo, volvemos a generar la lista anterior para J_k: esta vez elegimos la menor entrada correspondiente a la mejor máquina (argumento i).
  - Línea 3.4: Actualizamos S para la entrada S_i, pues ahora la máquina M_i ejecuta a la tarea J_k.
  - Línea 3.5: Actualizamos E, con J_k porque ya está programada.
  - Línea 3.6: Actualizamos al vector A_i.
  - Línea 3.7: Actualizamos al vector U. El tiempo de finalización de J_k será igual a la entrada de A_i.
- Línea 4: Se calcula el *makespan* que es la mayor entrada del vector A.
- Línea 5: Se presentan el makespan y las tareas asignadas a cada máquina.

V. EXPERIENCIAS NUMÉRICAS

Las instancias con las que se probó el algoritmo están conformadas por la cantidad M de máquinas, N de tareas y por una matriz T de tiempos de ejecución. Los valores que se manejaron para M y N respectivamente fueron:

- Número de tareas N: En el intervalo 100 a 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.[2]

[2] Notemos que un tiempo de ejecución muy alto (+α) 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.

En total tenemos 16 combinaciones para las combinaciones de ***máquinas-tareas***. De la misma forma, para cada combinación se generarán 10 instancias diferentes, lo que arroja un total de ***160 problemas-test*** realizados. Se enfrentó el algoritmo voraz con una meta heurística GRASP [36]:

Tabla N.º 2. Eficiencia del algoritmo voraz frente a una meta heurística GRASP.

Máquinas\ Tareas	GOLOSO Makespan	GRASP C Makespan	EFICIENCIA %
100 \ 12	213.3	193.7	9.19%
100 \ 25	113.7	108.6	1.19%
100 \ 37	76.5	74.3	2.88%
100 \ 50	59.1	57.8	2.20%
150 \ 12	283.1	258.2	8.80%
150 \ 25	125.2	114	8.95%
150 \ 37	86.1	84.1	2.32%
150 \ 50	68.6	67.4	1.75%
200 \ 12	325.9	307	5.80%
200 \ 25	127	117	7.87%
200 \ 37	109.8	105.2	4.19%
200 \ 50	76.4	74.6	2.36%
250 \ 12	411.8	377.1	8.43%
250 \ 25	183.5	168.2	8.31%
250 \ 37	125.3	119.6	4.55%
250 \ 50	96.5	91.1	5.60%
Eficiencia algoritmo GRASP sobre el algoritmo Goloso			5.48%

De otro lado, para determinar la real calidad de las soluciones decidimos aplicar el modelo matemático del problema a paquetes solucionadores de problemas de la programación lineal como la herramienta **LINDO** [35] en su versión estudiantil[3], a fín de obtener soluciones exactas y enfrentarlas con las que arrojaba el algoritmo voraz propuesto. Para que los modelos sean manejables se disminuyó considerablemente el tamaño de las instancias de prueba. A continuación se presenta la configuración de dichas instancias (recuérdese que N es el número de tareas y M el número de máquinas):

Tabla N.º 3: Configuración de instancias N x M para los experimentos exactos.

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

Las Tablas N.os 10 y 11 resumen los resultados exactos obtenidos frente a los resultados heurísticos (solución voraz) y meta heurísticos (solución GRASP). Asimismo, presentaremos los valores de las constantes de relajación usadas para la fase de construcción GRASP en donde se realizaron cerca de 7000 iteraciones para todos los casos.

Los promedios de eficiencia están haciendo referencia a qué tan cerca está la solución del método respectivo de la solución considerada exacta.

Tabla N.º 4. Eficiencia de la solución voraz para instancias con M igual a 3.

Matriz	N	M	Resultado del LINDO	Algoritmo voraz	Algoritmo GRASP	% Exacto/Voraz	% Exacto/GRASP	T	θ
6x3_0	6	3	28	35	35	25.00%	25.00%	0.51	0.0[illegible]
6x3_1	6	3	67	67	67	0.00%	0.00%	0.31	0.0[illegible]
6x3_2	6	3	75	87	85	16.00%	13.33%	0.31	0.36
8x3_0	8	3	40	45	45	12.50%	10.00%	0.31	0.0[illegible]
8x3_1	8	3	88	102	102	15.91%	15.91%	0.31	0.0[illegible]
8x3_2	8	3	64	72	64	12.50%	0.00%	0.12	0.37
12x3_0	12	3	130	162	162	24.62%	24.62%	0.31	0.0[illegible]
12x3_1	12	3	121	129	121	6.61%	0.00%	0.3[illegible]	0.0[illegible]
12x3_2	12	3	80	90	89	12.50%	11.25%	0.3[illegible]	0.21
15x3_0	15	3	162	189	168	16.67%	3.70%	0.37	0.13
15x3_1	15	3	125	137	125	9.60%	0.00%	0.32	0.47
15x3_2	15	3	164	190	176	15.85%	7.32%	0.43	0.48
Promedios de eficiencia						**13.98%**	**9.26%**		

[3] Esta versión gratuita del LINDO tiene limitantes en el número de restricciones que puede contener el modelo, de allí que con las versiones profesionales del programa podríamos trabajar con instancias de mayor porte.

Tabla N.º 5. Eficiencia de la solución voraz para instancias con M igual a 5.

Matriz	N	M	Resultado del LINDO	Algoritmo voraz	Algoritmo GRASP	% Exacto/Voraz	% Exacto/GRASP	α	θ
10x5_0	10	5	45	51	49	13.33%	8.89%	0.01	0.17
10x5_1	10	5	40	52	44	30.00%	10.00%	0.01	0.04
10x5_2	10	5	61	70	66	14.75%	8.20%	0.01	0.41
15x5_0	15	5	59	60	59	1.69%	0.00%	0.09	0.27
15x5_1	15	5	64	71	68	10.94%	6.25%	0.01	0.13
15x5_2	15	5	41	58	43	41.46%	4.88%	0.07	0.34
20x5_0	20	5	88	109	106	23.86%	20.45%	0.01	0.18
20x5_1	20	5	66	70	66	6.06%	0.00%	0.05	0.14
20x5_2	20	5	82	92	87	12.20%	6.10%	0.28	0.15
25x5_0	25	5	96	115	113	19.79%	17.71%	0.36	0.01
25x5_1	25	5	109	137	129	25.69%	18.35%	0.27	0.05
25x5_2	25	5	86	96	90	11.63%	4.65%	0.04	0.08
Promedios de eficiencia						**17.62%**	**8.79%**		

VI. CONCLUSIONES

El algoritmo voraz arroja soluciones que están muy cerca de las soluciones exactas. El tiempo que le ocupa al computador ejecutar este algoritmo es insignificante para casi cualquier instancia de prueba (por debajo de los 2 segundos). Siendo tan solo una heurística, apenas es superada por los algoritmos meta heurísticos a los que se enfrentó, siendo muy buenos los resultados para instancias de gran porte.

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 (penalidades más que orden). Tampoco existen sistemas que planifiquen líneas de producción completas 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.

El tiempo de ejecución es bastante bajo para la configuración del hardware donde se realizaron las pruebas Esto nos lleva a inferir que, con una configuración más potente sí se podría pasar esta barrera. El tiempo de CPU empleado no pasaba de los 2 segundos para las instancias más grandes (250x50).

VII. BIBLIOGRAFÍA

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: 215–230 (1996).
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. Feo T., Resende M., Greedy Randomized Adaptive Search Procedure Journal of Global Optimization, 6:.109-133 (1995).
6. Rauch W. *Aplicaciones de la inteligencia Artificial en la actividad empresarial, la ciencia y la industria*. Tomo II, Editorial Díaz de Santos, España (1989).
7. Kumara P. *Artificial Intelligence: Manufacturing theory and practice*. Editorial NorthCross Institute of industrial Engineers, USA (1988).
8. Blum A., Furst M. *Fast Planning through Plangraph Analysis*. Article of memories from 14th International Joint Conference on Artificial Intelligence, pp. 1636-1642. Ediciones Morgan-Kaufmann - USA (1995).
9. Conway R. *Theory of Scheduling*. Addison–Wesley Publishing Company, Massachussets–USA, (1967).
10. Miller G; Galanter E. *Plans and the Structure of Behavior*. Editorial Holt, New York-USA, (1960).
11. 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).

12. Bard J; Feo T. *Operations sequencing in discrete parts manufacturing Journal of Management Science*. 35: 249-255 (1989)
13. Bard J; Feo T. *An algorithm for manufacturing equipment selection problem*. IEEE Transactions, 23: 83-91, (1991).
14. Brucker P; Jurisch B; Sievers B. *A branch and bound algorithm for the job-shop scheduling problem*. Journal of Discrete Applied Mathematics, 49:105-127 (1994)
15. 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).
16. Gonçalves J., Magalhäes J., Resende M. *A Hibrid Genetic algorith for the Jos Shop Scheduling*. AT&T Labs Research Technical Report TD-5EAL6J (2002).
17. Davis L. *Job shop scheduling with genetic algorithms*. First International Conference on Genetic Algorithms and their Applications, 136-140. Morgan – Kaufmann USA, (1985).
18. International Conference on Genetic Algorithms and their Applications (1, 1985, USA). *Job shop scheduling with genetic algorithms*. (reporte) Ed. Davis L., Morgan – Kaufmann USA, 136-140.
19. Taillard E. *Parallel Taboo Search Technique for the Job shop Scheduling Problem*. Journal on Computing Science, 6: 108-117, (1994).
20. Chambers J; Barnes W. *Taboo Search for the Flexible-Routing Job Shop Problem*. Department of Computer Sciences, University of Texas-USA, Reporte técnico TAY 2.124, (1997).
21. Subramani V; Kettimuthu R; Srinivasan S; Sadayappan P. *Distributed Job Scheduling on Computational Grids using Multiple Simultaneous Requests*. Department of Computer and Information Science of Ohio State University, USA. Disponible http://www.gridforum.org (2003).
22. Tupia, M. *Un algoritmo Voraz Adaptativo y Randómico para resolver el problema de la Programación de Tareas independientes en máquinas homogéneas*. Pontificia Universidad Católica del Perú (2001).
23. Kleinrock L. *Quering Systems*. John Wiley Editions, USA, (1974).
24. Yongpei Guan Y, Wen-Qiang Xiao, Cheung R.; Chung-Lu Lin. *A multiprocessor task scheduling model for berth allocation heuristic and worst-case analysis*. Operations Research Letters, 30: 343 – 350, Elsevier Science, (2002).
25. AFIPS Fall Joint Computer Conference (1962, USA). *An experimental time-sharing system*. memoria Eds. Corbato F, Merwin M, Daley R., USA, 335-344.
26. Banús J; Moncusí A; Labarta J. *The Last Call Scheduling Algorithm for Periodic and Soft Aperiodic Tasks in Real-Time Systems*. Departament d'Enginyeria Informàtica Universitat Rovira i Virgili, Tarragona – España (2000).
27. Feo T; Sarathy K; McGahan J. *A GRASP for single machine scheduling with sequence dependt setup cost and linear delay penalities*. University of Texas – USA, reporte técnico TX 78712-1064 (1994).
28. Ríos R; Bard J. *Heurísticas para el secuenciamiento de tareas en líneas de flujo (en línea)*. Universidad Autónoma de Nuevo León (México) Disponible http://osos.fime.uanl.mx/~roger/ papers (2000).
29. Nawaz M; Enscore E; Ham I. *A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem*. Omega, 11(1): 91-95 (1983).
30. Feo T; Venkatraman K. *A Grasp for a difficult single machine scheduling problem*. Journal of Computer and Operation Research, No.18 (1991).
31. Acero R; Torres J. *Aplicación de una heurística de búsqueda tabú en un problema de programación de tareas en línea flexible de manufactura*. Revista de la Sociedad de Estadística e Investigación Operativa, Colombia, 5(2): 283–297, (1997).
32. Chand S; Schneeberger H. *Single machine scheduling to minimize earliness subject to notardy jobs*. European Journal of Operational Research, 34: 221-230 (1988).
33. Laguna M; Gonzalez J. *A search heuristic for just-in-time scheduling in parallel machines*. Journal of Intelligent Manufacturing, No. 2:253-260, (1991).
34. Cormen T.; Leiserson Ch.; Rivest R. *Introduction to Algorithms*, MIT Press, Editorial McGraw Hill (2001)
35. Scharage L. *Optimization modeling with LINDO*. Duxbury Press. USA (1997).
36. Conferencia Latinoamericana de Informática (30,2004, Perú) (a imprimirse) Un algoritmo GRASP para resolver el problema de la programación de tareas dependientes en maquinas diferentes, memoria Eds. M. Solar, D. Fernández-Baca, E. Cuadros-Vargas, Arequipa – Perú, p. 129—139, (2004).