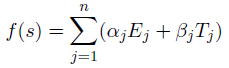
**2. Declaración de problemas y modelo matemático**

El problema UPMS (Programación de máquinas paralelas no relacionadas) examinado en este documento se establece de la siguiente manera. Hay un conjunto J = {1, ..., n} de n que debe procesarse en exactamente una máquina de un conjunto M = {1, ..., m} de m máquinas paralelas. Cada máquina está continuamente disponible y puede procesar como máximo un trabajo a la vez. No se permiten privilegios de trabajo. Cada trabajo j se vuelve disponible en el tiempo cero, tiene un tiempo de procesamiento Pj,k en la máquina k, una fecha de vencimiento dj y precocidad (αj) y tardanza (βj) penalizaciones. Entre el procesamiento de dos trabajos consecutivos i y j en la máquina k se considera un tiempo de configuración dependiente de la secuencia Si, j, k.

No consideramos los tiempos de preparación antes de procesar el primer trabajo en una máquina. En este problema, se permite la aparición del tiempo de inactividad de la máquina. El tiempo de inactividad en una máquina puede ser requerido para completar un trabajo en su fecha de vencimiento, evitando la precocidad. El objetivo del problema es determinar un cronograma factible para que se reduzcan al mínimo las penalizaciones por antigüedad y tardanza totales de los trabajos. Para un horario s, el criterio minimizado se calcula como:



donde, Ej = max {0, dj - Cj} es la precocidad del trabajo j y Tj = max {0, Cj - dj}

es la tardanza del trabajo j, siendo Cj el tiempo de finalización del trabajo j. Tenga en cuenta que, un trabajo j es temprano cuando se completa antes de su fecha de vencimiento (Cj < dj). De manera similar, cuando un trabajo j termina después de su fecha de vencimiento (Cj> dj), tiene tardanzas. Las penalizaciones incurridas en la precocidad y la tardanza están determinadas por αjEj y βjTj, respectivamente.

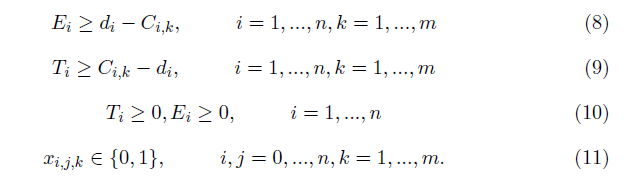
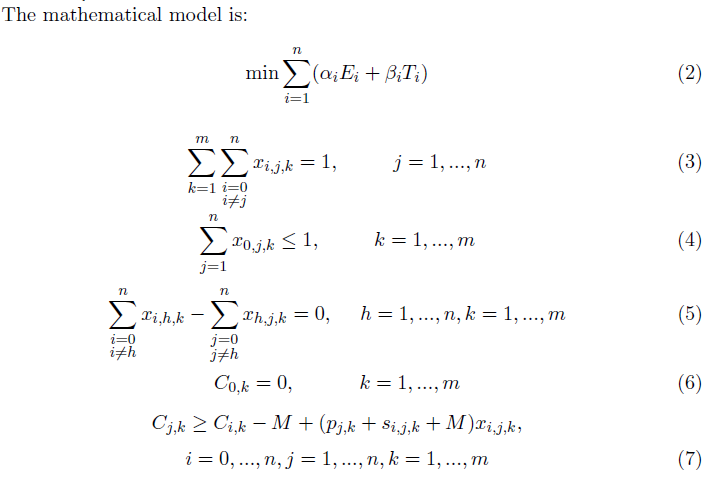
Siguiendo la notación de tres campos de Graham et al. [20], el problema que estudiamos se puede denotar como R / sijk / Σ (αiEi + βiTi), donde R es para el entorno de máquina paralelo no relacionado, sijk es máquina y tiempos de configuración dependientes de secuencia de trabajo y Σ (αiEi + βiTi) es la función objetivo.

Proporcionamos un modelo de Programación de enteros mixtos (MIP) para el problema de UPMS con tiempos de configuración dependientes de la secuencia. El modelo se basa en el modelo MIP presentado por Morabito et al. [4] para la precocidad total y la minimización de tardanzas. El modelo usa un trabajo ficticio 0 para marcar el comienzo y el final de una secuencia de trabajos en cada máquina. El modelo involucra las siguientes variables de decisión:

Ci,k = Tiempo de finalización del trabajo i en la máquina k

Ei = Precocidad del trabajo i

Ti = Tardanza del trabajo i



La objetivo de la función (2) es minimizar las penalizaciones de precocidad/tardanza ponderadas totales. El conjunto de restricciones (3) garantiza que cada trabajo j se asigna exactamente a una máquina y tiene exactamente un predecesor. Las restricciones (4) limitan el número de sucesores del trabajo ficticio 0 a un máximo de uno en cada máquina k. Las restricciones (5) garantizan que cada trabajo h tenga exactamente un sucesor, a excepción del trabajo ficticio que establece el comienzo y el final de una secuencia de trabajo en una máquina k. Para el trabajo 0, las restricciones (6) establecen que el tiempo de finalización de este trabajo en cada máquina es igual a cero. El conjunto de restricciones (7) es para controlar los tiempos de finalización de los trabajos en las máquinas. Si se asigna un trabajo j a la máquina k después del trabajo i (es decir, xi, j, k = 1), su tiempo de finalización Cj, k debe ser mayor que el tiempo de finalización de i, Ci, k, más el tiempo de configuración entre i y j y el tiempo de procesamiento de j. Si xi,j,k = 0, entonces la gran constante M hace que la restricción sea redundante. Las Restricciones (8) y (9) definen la precocidad y la tardanza de cada trabajo, respectivamente. Finalmente, el conjunto de restricciones (10) identifica las condiciones de no negatividad y el conjunto (11) define las variables binarias.

**3. Algoritmos heurísticos propuestos**

En este trabajo, para obtener soluciones casi óptimas del problema R / sijk / Σ (αiEi + βiTi), desarrollamos algoritmos heurísticos de árbol basados ​​en la metaheurística Procedimiento de búsqueda adaptativa aleatoria codificada (GRASP). GRASP, propuesto originalmente por Feo y Resende [14], es un método bifásico de múltiples etapas (iterativo) que consiste básicamente en una fase de construcción de la solución y una fase de mejora. La fase de construcción construye aleatoriamente una solución codiciosa paso a paso, agregando elementos a una solución parcial. Cuando se ha construido una solución factible, se explora su vecindario en una fase de búsqueda local hasta que se encuentre un óptimo local. La mejor solución producida después de un determinado número de iteraciones previamente especificado (o criterio de finalización) se devuelve como salida.

El algoritmo GRASP es fácilmente adaptable y se ha aplicado con éxito para varios problemas de optimización combinatoria NP-hard [15]. En este trabajo, primero desarrollamos una adaptación del algoritmo GRASP básico al problema sijk / Σ (αiEi + βiTi). Luego usamos la técnica Path Relinking (PR) para mejorar el rendimiento de GRASP. Esta técnica se utiliza como una estrategia de intensificación para combinar las mejores soluciones obtenidas en el proceso iterativo. También proponemos una heurística híbrida que combina el algoritmo GRASP con la metaheurística Iterated Local Search (ILS) [32]. ILS se usa como un sustituto de la búsqueda local estándar en el algoritmo GRASP. ILS es un algoritmo iterativo que en cada iteración aplica perturbaciones a las soluciones óptimas locales y las soluciones perturbadas resultantes se envían luego a una búsqueda local.

Las tres heurísticas desarrolladas se denominan GRASP básico, GRASP + PR y GRASP + ILS + PR, respectivamente. En las siguientes subsecciones, primero, describimos la representación y evaluación de una solución factible. Luego describimos cada fase del algoritmo GRASP (construcción y mejora), la técnica PR y el algoritmo de búsqueda local ILS.

**3.1 Representación y evaluación de soluciones**

Una solución del problema R / sijk / Σ (αiEi + βiTi) está representada por m listas de trabajos vinculadas (una por máquina). Cada lista representa el orden de procesamiento de los trabajos asignados a una máquina. Por ejemplo, una solución para una instancia con n = 6 trabajos y m = 2 máquinas, se representa por s = [s1, s2] = [[1, 4, 6], [2, 3, 5]], donde s1 = [1, 4, 6] y s2 = [2, 3, 5] representan el orden de procesamiento de los trabajos asignados a las máquinas 1 y 2, respectivamente.

Para calcular la función objetivo f (s) de una solución dada s, primero, se calculan los tiempos de inicio óptimos de los trabajos. En el cálculo de estos tiempos, se puede permitir la ocurrencia del tiempo de inactividad de la máquina. Es decir, una máquina puede estar inactiva hasta que se inicie el procesamiento del siguiente trabajo. Szwarc y Mukhopadhyay [45] J. P. de C. M. Nogueira 58 y col. / Notas electrónicas en Theoretical Computer Science 302 (2014) 53-72 y Wang y Yen [49] propusieron algoritmos de temporización óptima para calcular los tiempos de inicio óptimos de los trabajos en la programación de una sola máquina. En este trabajo, adaptamos estos algoritmos para el caso de la máquina paralela. El algoritmo implementado decide el tiempo de inicio óptimo y el tiempo de finalización de acuerdo con la fecha de vencimiento correspondiente para cada trabajo. El objetivo del algoritmo óptimo es minimizar las penalizaciones de precocidad y tardanza totales para una solución previamente determinada.

Para comprender mejor la inserción de tiempo de inactividad de la máquina, utilizamos un problema de ejemplo con seis trabajos y dos máquinas (n = 6, m = 2). Supongamos que las fechas de vencimiento de los trabajos 1, ..., 6 son d1 = 7, d2 = 10, d3 = 18, d4 = 20, d5 = 27 y d6 = 30, respectivamente. Deje s = [[1, 4, 6], [2, 3, 5]] una solución para esta instancia. En la Figura 1 se muestra un cronograma óptimo. En este cronograma, la secuencia de trabajos en la primera máquina (M1) es [1, 4, 6] y la secuencia de trabajos en la segunda máquina (M2) es [2, 3, 5]. No es así, todos los trabajos terminan exactamente en su fecha de vencimiento. Es sencillo ver que todas las máquinas tienen tiempos de inactividad. Por ejemplo, hay un tiempo de inactividad de 3 unidades de tiempo en la máquina M2 entre el tiempo de finalización del trabajo 2 (después del tiempo de configuración s2,3,2) y el comienzo del trabajo 3. En este ejemplo, los tiempos de inicio óptimos de los trabajos 1, ..., 6 son 3, 4, 15, 14, 22 y 25, respectivamente. Podemos notar que los tiempos de inactividad de la máquina son necesarios para evitar la precocidad laboral. Para la misma instancia, la Figura 2 muestra una programación que no permite tiempos de inactividad. En este cronograma, se generan penalizaciones por precocidad porque todos los trabajos se completan antes de las fechas de vencimiento.

**3.2 Fase de construcción de solución**

En la fase de construcción del algoritmo GRASP, se genera iterativamente una solución s = [s1, ..., sm], donde sk es la secuencia de trabajos en la máquina k. Comenzando con una solución vacía (sk = ∅, ∀k = 1, ..., m), en cada iteración, ¿un trabajo j? se selecciona uniformemente al azar de una lista restringida de candidatos RC y se agrega a exactamente una máquina k? (o secuencia s? k). La lista RC se forma de la siguiente manera. Primero, cada trabajo no programado j se asigna temporalmente a la máquina k que produce el valor más bajo de la función objetivo (las penalizaciones por antigüedad y tardanza totales de los trabajos de

los trabajos ya programados). Luego, los trabajos se organizan de acuerdo con el valor producido, formando la lista de candidatos C de trabajos no programados.

El Candidato Restringido J.P. de C. M. Nogueira et al. / Notas electrónicas en Informática teórica 302 (2014) 53-72 59 lista RC está formada por los trabajos de C que tienen los mejores valores. El tamaño de RC es r = max (1, α × | C |), donde α ∈ [0, 1] es el parámetro que controla las cantidades de codicia y aleatoriedad en el algoritmo de construcción. En cada iteración de construcción, el trabajo seleccionado j? se elimina de C y se reordena la lista de trabajos no programados. El algoritmo para construir una solución aleatoria codiciosa finaliza cuando todos los trabajos se asignan a las máquinas, es decir, cuando C = ∅. En este punto, se devuelve una solución completa s. El pseudocódigo completo del procedimiento constructivo se puede encontrar en el algoritmo 1.

Tenga en cuenta que, α = 0 (| RC | = 1) corresponde a una construcción codiciosa en la que siempre se selecciona el primer trabajo de C. α = 1 (| RC | = | C |) produce una construcción aleatoria en la que un trabajo se elige aleatoriamente de C. La forma de ordenar los trabajos no programados también se usa en la heurística constructiva DJASA (Asignación dinámica de tareas con Asignación de recursos de configuraciones) propuesto por Ruiz y Andrés [42]. DJASA es una heurística codiciosa determinista en la que siempre se selecciona el primer trabajo de la lista ordenada, es decir, el trabajo que proporciona el menor incremento en el valor de la función objetivo.

-----------------------------------------------------------------------------------------------------------------------------------------

Algoritmo GreedyRandomizedConstruction(*α*)

-----------------------------------------------------------------------------------------------------------------------------------------

Sea sk = ∅, ∀k = 1, ..., m;

Agregue todos los trabajos a una lista de trabajos no programados C (lista de candidatos);

**mientras** C = ∅ **hacer**

**para**, para cada trabajo pendiente j ∈ C, **hacer**

Asigne temporalmente el trabajo j a la máquina k que produce el valor más bajo de la función objetivo

(la máquina correspondiente de trabajo j es k);

Sea g (j) el valor de la función objetivo para la solución parcial obtenida;

**fin para**

Organice los trabajos en C en orden creciente de la función *g*;

Deje RC la lista de candidatos restringidos formada por los primeros trabajos max (1, α × | C |) de C;

seleccione el trabajo j´ al azar de RC;

Asigne el trabajo j´ a la máquina correspondiente k´ (sk´ = sk´ ∪ {j´});

Elimine el trabajo j´ de C;

**fin mientras**

retornar la solución obtenida s;

-----------------------------------------------------------------------------------------------------------------------------------------

**3.3 Fase de mejora**

Cada solución construida en la fase constructiva es el punto de partida para un procedimiento de búsqueda local en el que tratamos de mejorar la solución. El método Local Search implementado en este trabajo se basa en la búsqueda de vecinos. Este método genera nuevas soluciones (soluciones vecinas) a través de inserciones de trabajo realizadas en la solución actual s. Un movimiento de inserción genera una nueva solución eliminando un trabajo de su posición original uy insertándolo en la posición v de la misma máquina o diferente.

El vecindario contiene todas las soluciones alcanzadas a través de movimientos individuales realizados en la solución actual. En este barrio, se recoge una solución que es mejor que la solución actual. La solución elegida se convierte en una nueva solución (o actual) y el proceso continúa hasta que se alcanza un mínimo local. El pseudocódigo del procedimiento de búsqueda local se puede ver en el algoritmo 2.

La vecindad de inserción de una solución s, con sk = ∅, ∀k = 1, ..., m, tiene tamaño (n2 - n + m) si m es par, o tamaño (n2-m) si m es impar. Considere el siguiente ejemplo con cuatro trabajos y dos máquinas. El entorno de la solución s = [[2, 1], [3, 4]] está formado por 14 soluciones: s1 = [[1, 2], [3, 4]], s2 = [[1], [2 , 3, 4]], s3 = [[1], [3, 2, 4]], s4 = [[1], [3, 4, 2]], s5 = [[2], [1, 3 , 4]], s6 = [[2], [3, 1, 4]], s7 = [[2], [3, 4, 1]], s8 = [[3, 2, 1], [4 ]], s9 = [[2, 3, 1], [4]], s10 = [[2, 1, 3], [4]], s11 = [[2, 1], [4, 3]] , s12 = [[4, 2, 1], [3]], s13 = [[2, 4, 1], [3]] y s14 = [[2, 1, 4], [3]].