UNIVERSIDAD DE LA LAGUNA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

DISEÑO Y ANÁLISIS DE ALGORITMOS

PRÁCTICA 7 – ALGORITMOS CONSTRUCTIVOS Y BÚSQUEDAS POR ENTORNO

Daniel Oria Martín

1. Introducción a la práctica

En esta práctica se estudiará un problema de secuenciación de tareas en máquinas paralelas con tiempos de setup dependientes. 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.

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 *pj*.
- Hay tiempos de setup de la máquina sij para procesar la tarea *j* justo después de la tarea *i*, con *sij*, *sji*, en general. Hay un tiempo de setup *s0*j 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

2. Tareas a realizar

A. Diseñar e implementar un algoritmo constructivo voraz descrito por el siguiente pseudocódigo.

```
1: Seleccionar la m tareas j1, j2, .., jm con menores valores de
t0j para ser introducidas en las primeras posiciones de los arrays
que forman la solución S;
2: S = {A1 = {j1}, A2 = {j2}, .., Am = {jm}};
3: repeat
4:    S* = S;
5:    Obtener la tarea-maquina-posición que minimiza el incremento
del TCT;
6:    Insertarla en la posición que corresponda y actualizar S * ;
```

```
7: until (todas las tareas han sido asignadas a alguna máquina)
8: Devolver S* ;
```

B. Diseñar e implementar un nuevo algoritmo voraz para el problema
C. Diseñar e implementar un GRASP para el problema. Implementar sólo la fase constructiva.
D. Diseñar e implementar un Método Multiarranque para el problema. Para ello se deberán definir las estructuras de entorno correspondientes a los siguientes 4 movimientos:
    a. Intercambio de tareas entre máquinas
    b. Intercambio de tareas en la misma máquina
    c. Re-inserción de una tarea en otra posición de otra máquina
    d. Re-inserción de una tareas en otra posición de la misma máquina.
E. Diseñar e implementar una Búsqueda por Entorno Variable General para el problema.

3. Algoritmos y resultados

1. GREEDY

Este algoritmo está basado en el pseudocódigo que se encuentra en el apartado A del punto 2 del informe. Consiste en que, mientras queden tareas que no están asignadas a una máquina, se obtiene la tarea-máquina-posición que minimiza el incremento del TCT.

Resultados

Enlace a las gráficas (Ir a la página de específica de GREEDY).
Como se puede apreciar en las gráficas, el resultado del GREEDY es siempre el mismo, pues solo existe un camino mínimo desde el principio siguiendo siempre el menor recorrido, lo que sí varían son los tiempos de CPU que, aun así, son ínfimos.

2. GREEDY 2

Para este segundo Greedy, en lugar de elegir la tarea que minimice el incremento del TCT, se ha implementado un algoritmo que minimiza la suma del tiempo de setup sumado con el tiempo de procesado. Este algoritmo actúa de forma voraz pues siempre cogerá el menor camino.

Resultados

Enlace a las gráficas (Ir a la página de específica de GREEDY).

Como se puede apreciar, los resultados son algo peores que los del GREEDY original y no se está ahorrando tanta operatoria como para que haya un cambio significativo en el tiempo de ejecución, por tanto estamos ante una opción peor que el primer algoritmo.

3. GRASP

El algoritmo GRASP (Greedy Randomized Adaptive Search Producers), en lugar de seleccionar la mejor tarea-máquina-posición como hace el GREEDY normal, lo que se hace es seleccionar una lista restringida de candidatos. En nuestro caso lo haremos por cardinalidad, seleccionando las k tareas que minimicen el TCT y más tarde eligiendo una de estas k tareas para ser metida en la máquina.

El GRASP tiene un método constructivo por el cual se crea la primera solución. Una vez acabado llamamos al método exploreLocal, el cual irá intercambiando tareas siguiendo cuatro posibles patrones que son, intercambio de tareas entre máquinas y en la misma máquina, y reinserción de tareas entre máquinas y en la misma máquina. Siguiendo uno de estos patrones el método exploreLocal irá mejorando progresivamente la solución durante *x* iteraciones o *x* iteraciones sin mejora.

El método **grasp()** y el método **exploreLocal()** pertenecen a la clase **Machines.h** y hacen uso del método **getBetterTime()**, también de esa clase, el cual se usa para encontrar las k tareas con menor incremento de TCT. El método **exploreLocal()** hace uso de un objeto de la clase **LocalSearch.h** la cual contiene los distintos algoritmos de intercambio de tareas.

Resultados

Enlace a las gráficas (Ir a la página de específica de GRASP).
Como podemos observar, en la mayor parte de los casos, el método de exploración Greedy es más lento que el Ansioso pero tiene mejores resultados. Otra cosa que se observa es que a medida que aumentan el número de máquinas disminuye considerablemente el TCT.

4. VND

El algoritmo VND (Variable Neighborhood Descent), consiste en elegir una solución inicial x generada mediante el GRASP. En la práctica se ha implementado la versión Greedy, esto quiere decir que, a nuestra solución inicial, se le irán encontrando todos los vecinos intercambiando tareas mediante los

cuatro métodos implementados en la clase LocalSearch, en busca de la mejor solución de todas.

El algoritmo está implementado en la clase **Machines.h**, un método llamado **vnd()**. Este método usa la clase **LocalSearch.h**, al igual que el GRASP, para ir buscando una mejor solución hasta que haya el mínimo local.

Resultados

Enlace a las gráficas (Ir a la página de específica de VND).
Como podemos apreciar en las tablas, en este algoritmo ya encontramos algo más de diferencia entre los tiempos de CPU para 100 y 1000 iteraciones. En cuanto al TCT se nota también una mejora frente al GRASP y el GREEDY y esto se debe a que estamos llegando a distintos mínimos locales.

5. GVNS

El último algoritmo implementado es el GVNS, el cual busca, mediante el uso del VND, ir generando mínimos locales y saltando a distintos puntos aleatorios desde estos para así terminar llegando al máximo global. Para el funcionamiento de este algoritmo tenemos una *k* y una *kmax*, siendo *k* el número de saltos efectuados y *kmax* el número de máximos saltos posibles, por defecto se ha puesto que *kmax* pueda valer un máximo de 5. Una vez hallada la *kmax* se generan saltos aleatorios y se llama seguidamente a **vnd**, llegando así a un posible mínimo local, en caso de ser mejor que el de antes del salto, empezamos de nuevo el proceso con k = 1 y este nuevo punto, si no es mejor seguimos aumentando la *k* mientras esta sea menor o igual que la *kmax*.

Este algoritmo se ha implementado dentro de la clase **Machines.h**, **gvns()**, junto con otros métodos como **generateKMax()** para hallar el máximo número de saltos posibles y **generateRandomKPoints(k)** al que se le pasa el número de saltos actuales y los efectúa. Al hacer uso del método **vnd()** también usaría los métodos y clases explicados en dicho algoritmo.

Resultados
Enlace a las gráficas (Ir a la página de específica de GVNS).
Como podemos ver en las tablas, al usar el método GVNS en búsqueda de máximos globales, generalmente el TCT disminuye, encontrando así una mejor solución, aunque elevando en cantidades considerables el tiempo de CPU.