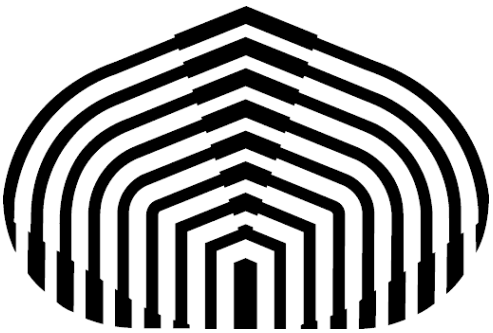

Universidad Simón Bolívar
Diseño de Algoritmos II (CI-5652)
Profesor Ricardo Monascal

Flow Shop Scheduling Problem

Elaborado por:
Ian Garcia 19-10087
Jose Revete 19-10040
Jesus Gutierrez 20-10332
Mauricio Fragachan 20-10265

Caracas, Febrero del 2026

Índice

Introducción

Un problema bastante reconocido en el ámbito de la computación es el de determinar la secuencia óptima para procesar n trabajos en m máquinas siguiendo un orden idéntico de visita, a este problema se le denomina el Problema de Programación de Taller de Flujo (FSSP). En este sentido, se puede presentar una restricción importante, la cual es que la secuencia de los trabajos debe ser la misma en todas las máquinas, dado paso a la variante de taller de flujo de permutación (PFSP), siendo esta la idea principal de este estudio. Esto define un espacio de búsqueda de $n!$ secuencias posibles, lo que hace necesaria una correcta selección del orden de entrada para la eficiencia de los Sistemas de Manufactura Flexible (FMS).

Viendo el problema desde el punto de vista de la complejidad computacional, la dificultad del mismo escala rápidamente según el número de máquinas involucradas en el planteamiento. En principio se presenta el caso más simple ($m = 1$), para minimizar el tiempo total cualquier secuencia es óptima; sin embargo, para minimizar el flowtime, el problema se resuelve trivialmente mediante la regla Shortest Processing Time (SPT). Luego se encuentra el escenario de dos máquinas ($m = 2$), el problema puede resolverse de manera exacta y eficiente mediante el Algoritmo de Johnson, definiendo así el caso polinomial. Finalmente el caso más difícil ($m \geq 3$), en este el PFSP se clasifica como NP-hard, dado que encontrar la secuencia óptima entre las $n!$ combinaciones posibles es computacionalmente "costoso" para instancias grandes, ya que no existe un algoritmo de tiempo polinomial que garantice la solución perfecta.

En la mayorías de recursos bibliográficos, se hace énfasis en la minimización del makespan para maximizar la administración de los recursos; sin embargo en la actualidad, el flowtime toma un papel importante con respecto a la optimización del mismo, dado que de esta manera es posible reducir el inventario en proceso (WIP) y mejorar la estabilidad del flujo de trabajo.

En el panorama actual las investigaciones se han orientado hacia el desarrollo de heurísticas y metaheurísticas eficientes, dado que los métodos de enumeración exacta resultan computacionalmente "costosos" para planteamientos de gran escala. Entre estas ramas de investigaciones aparece el algoritmo NEH, siendo este ampliamente reconocido como una de las mejores heurísticas, basándose en la inserción iterativa de trabajos, o el algoritmo PR-1 el cual es el mejor para la solución de este tipo de problemas (implementando Búsqueda Local Iterativa). De igual manera se encuentra el uso de técnicas de búsqueda local y metaheurísticas como la Búsqueda Local Iterada (ILS), permitiendo depurar estas soluciones para escapar de óptimos locales y alcanzar resultados cercanos al óptimo global en tiempos reducidos.

Finalmente, para la evaluación del desempeño de los algoritmos propuestos, se utiliza el benchmark de Taillard. Este conjunto de instancias es el estándar más claro en la mayoría de investigaciones computacionales, ya que abarca diversos tamaños de problemas y niveles

de dificultad, permitiendo una comparación cuidadosa y fácilmente adaptable de los resultados.

Para la evaluación y comparación del desempeño de los algoritmos propuestos en la resolución de nuestro problema de estudio, se utilizará el benchmark de **Taillard**, el cual compone uno de los conjuntos de instancias más utilizados y aceptados en la literatura científica sobre problemas de programación de la producción.

Este benchmark fue propuesto por Éric Taillard en 1993 con el objetivo de proporcionar un conjunto estandarizado de instancias que permitiera evaluar de forma objetiva y reproducible distintos métodos de resolución para problemas de secuenciación, en particular para el *Flow Shop*. Desde su publicación, las instancias de Taillard han sido ampliamente utilizadas como referencia en una gran cantidad de trabajos de investigación, convirtiéndose en un estándar para la comparación de heurísticas, metaheurísticas y métodos exactos.

Las instancias de este benchmark se caracterizan por considerar diferentes tamaños de problema, variando el número de trabajos (n) y el número de máquinas (m). Comúnmente se incluyen combinaciones tales como 20, 50 y 100 trabajos, junto con 5, 10 y 20 máquinas, lo cual permite analizar el comportamiento de los algoritmos bajo distintos niveles de complejidad y escalabilidad. Por otro lado, los tiempos de procesamiento de cada trabajo en cada máquina son generados de manera pseudoaleatoria dentro de rangos definidos, garantizando así diversidad en las instancias sin perder control sobre sus características estadísticas.

Una de las principales ventajas del benchmark de Taillard es que proporciona instancias suficientemente grandes y complejas como para que los métodos exactos resulten impracticables en la mayoría de los casos, reflejando de forma realista los desafíos computacionales que presenta nuestro problema de estudio en aplicaciones de la vida real a gran escala. Al mismo tiempo, estas instancias permiten evaluar la robustez, eficiencia y calidad de las soluciones obtenidas por heurísticas y metaheurísticas, especialmente en términos de *makespan* y *flowtime*.

El *makespan* es el tiempo de finalización del último trabajo en la última máquina. Este se calcula de la siguiente manera:

$$C_{m,j} = \begin{cases} p_{m,j} & \text{si } m = 1 \text{ y } j = 1, \\ C_{m-1,j} + p_{m,j} & \text{si } j = 1, \\ C_{m,j-1} + p_{m,j} & \text{si } m = 1, \\ \max\left(C_{m-1,j},\, C_{m,j-1}\right) + p_{m,j} & \text{en otro caso.} \end{cases}$$

Donde $P_{m,j}$ representa el tiempo de procesamiento del trabajo j en la máquina m.

Una vez calculada la matriz C, el makespan se obtiene directamente como Makespan = $C_{m,j}$

Adicionalmente, el uso de este benchmark facilita la comparación directa de los resultados obtenidos en este estudio con los reportados en investigaciones previas, ya que muchas publicaciones presentan valores de referencia o mejores soluciones conocidas para estas instancias. Esto nos facilita validar el desempeño de los algoritmos planteados y a situar los resultados dentro del contexto de las demás investigaciones.

En consecuencia, la selección del benchmark de Taillard resulta adecuada y pertinente para los objetivos de nuestro trabajo, ya que permite una evaluación rigurosa, estandarizada y ampliamente comparable de las técnicas propuestas para la resolución de nuestro problema de estudio.

Solución exacta

Para tener un punto de referencia confiable y poder comparar con las soluciones aproximadas dadas por las heurísticas, se implementó una solución exacta del problema. Como sabemos, todos los trabajos deben pasar por las máquinas en el mismo orden, así que con definir un solo orden de los trabajos ya tenemos un horario completo. El problema es que el número de órdenes posibles crece muy rápido cuando hay muchos trabajos, lo que hace que intentar probar todas las combinaciones sea prácticamente imposible.

La estrategia que usamos es *Branch and Bound*. La idea es construir las secuencias de trabajo poco a poco: empezamos con una secuencia parcial y vamos agregando trabajos uno a uno. Para cada secuencia parcial calculamos los tiempos de finalización de todos los trabajos en todas las máquinas, respetando las reglas del flow shop: un trabajo no puede comenzar en una máquina hasta que haya terminado en la anterior y hasta que la máquina esté libre.

Para no explorar todas las combinaciones posibles, usamos una cota inferior que estima de manera optimista el flowtime total si completamos la secuencia con los trabajos que quedan. En nuestro código, esto se calcula sumando, para cada máquina, los tiempos de los trabajos que faltan y combinándolo con los tiempos de la secuencia parcial. Si esta estimación ya es peor que la mejor solución que hemos encontrado, simplemente descartamos esa rama y seguimos con otras opciones.

Cuando llegamos a una secuencia completa, calculamos su flowtime real y actualizamos la mejor solución si es necesario. Al final del proceso obtenemos la secuencia de trabajos óptima y su flowtime mínimo.

Aunque este método garantiza encontrar la solución exacta, tarda mucho si hay muchos trabajos o máquinas, incluso usando la cota. Por eso se utiliza principalmente para instancias pequeñas o medianas y sirve como referencia para evaluar la calidad de las soluciones aproximadas.

Heurística NEH

Para la resolución del PFSP, hace falta escoger una heurística que sea especializada en este tipo de problema, por lo que en este estudio se emplea el algoritmo NEH (Nawaz-Enscore-Ham), el cual es capaz de optimizar el flujo en talleres de producción. El objetivo principal de este algoritmo es determinar una secuencia de n trabajos que deben ser procesados en m máquinas para así minimizar los indicadores clave de rendimiento. Entre estos se consiguen indicadores bastante prácticos como:

- Makespan (C_{max}): tiempo total desde que inicia el primer trabajo hasta que finaliza el último en la última máquina, aunque también es altamente efectivo para reducir el
- Total Flowtime ($\sum C_j$): suma acumulada de los tiempos de finalización de cada trabajo individual. Esta heurística es altamente efectiva para reducir dicho indicador.

La lógica que se emplea en el NEH se basa en determinar el tiempo total de procesamiento por cada trabajo y darles una prioridad superior a aquellos (con mayor tiempo de procesamiento) en todas las máquinas, posicionándo estos al inicio de la programación. Entonces, como bien se comentó, el proceso comienza calculando el tiempo total de cada trabajo y ordenándolos desde el que tiene mayor tiempo de procesamiento hasta el menor. Luego, escogen los dos primeros trabajos de la secuencia ordenada y se evalúan las dos combinaciones posibles para elegir la mejor. En este punto, el algoritmo construye la solución final mediante un proceso iterativo de "ensayo y error". Cada trabajo es sometido a una prueba de posicionamiento en todos los espacios disponibles de la secuencia parcial (exactamente $k + 1$ opciones para el trabajo k). Solo la configuración que minimiza el

tiempo total se mantiene, sirviendo así de base para la inserción del siguiente elemento en la lista.

A pesar de que el NEH es un punto de referencia por su eficiencia y rapidez, también se encuentran alternativas más complejas en las investigaciones computacionales. Viendo heurísticas como:

- PR1(x): siendo esta una heurística compuesta, que intentan mejorar este resultado mediante un enfoque iterativo. La PR1 utiliza el concepto de LR-NEH(x) para generar diversas semillas iniciales basadas en el Índice de Prioridad ($\xi_{j,k}$) y el Tiempo de Inactividad ($IT_{j,k}$), buscando reducir la inactividad de las máquinas de una forma más agresiva u "obligada".
- Búsqueda Local (iRZ, Iterated Rajendran and Ziegler): tomando la solución terminada del NEH e intercambiando de posición los trabajos uno a uno para determinar si el tiempo total puede disminuir.
- Metaheurísticas (ILS, Algoritmos Genéticos): A diferencia del NEH, que es de "una sola pasada", estas técnicas simulan procesos naturales o de búsqueda repetitiva. Son mucho más potentes pero requieren mucho más tiempo de cómputo.

Finalmente, es preferible el uso del NEH sobre métodos como PR1 o metaheurísticas debido a factores destacables como: excelente relación entre calidad de solución y esfuerzo computacional, amplia documentación y respaldo. Por otra parte, la PR1 requiere controlar mediante límites estrictos el tiempo de ejecución (como el criterio de $0.01mn$ segundos para no volverse ineficiente). Al no requerir iteraciones complejas, el NEH ofrece una solución "buena" de manera inmediata. Su compatibilidad con las instancias de Taillard permite que los resultados se midan directamente contra los estándares globales de la industria. Asimismo, destaca por su facilidad de implementación, al prescindir de la configuración de variables críticas que suelen complicar otros métodos heurísticos.

Búsqueda Local

En el caso de la solución mediante la búsqueda local, se definió un entorno de exploración a partir de una estructura de vecindad basada en el movimiento de inserción. Se define de la siguiente manera:

Dada una permutación

$$\pi = (\pi_1, \pi_2, \ldots, \pi_n)$$

un vecino se genera mediante:

1. Seleccionar un trabajo en la posición i.
2. Extraerlo de la secuencia.
3. Insertarlo en otra posición j, con $j \neq i$.

Para un problema con n trabajos, esta vecindad genera aproximadamente $n(n-1)$ soluciones vecinas posibles, lo que permite una exploración amplia del espacio de búsqueda alrededor de la solución actual. La decisión de utilizar esta vecindad, es que en diversos estudios, es la que mejor resultados provee.

La función de evaluación a utilizar es el propio *makespan*, que corresponde al tiempo de finalización del último trabajo en la última máquina. El cálculo del *makespan* se realiza utilizando la relación recursiva típica del flow shop, donde el tiempo de finalización de un trabajo en una máquina depende tanto de la finalización del trabajo anterior en esa máquina como de la finalización de ese mismo trabajo en la máquina previa. Esta función de evaluación es la más utilizada en problemas de flow shop, ya que mide directamente la duración total del plan de producción.

El procedimiento de búsqueda adopta una estrategia de *best improve*, es decir, en cada iteración se generan y evalúan todos los vecinos de la solución actual, seleccionando aquel que presenta el menor valor de makespan. Si dicho vecino mejora la solución actual, se reemplaza la solución y el proceso se repite. La búsqueda finaliza cuando ninguno de los vecinos produce una mejora, lo que indica que se ha alcanzado un óptimo local con respecto a la vecindad de inserción.

En términos de complejidad computacional, la exploración de la vecindad requiere evaluar del orden de $O(n^2)$ movimientos posibles. Dado que el cálculo del *makespan* para una secuencia de n trabajos en m máquinas tiene un costo del orden de $O(nm)$, una iteración completa de búsqueda local tiene un costo aproximado de $O(n^3 m)$. Este costo se repite en cada mejora realizada hasta alcanzar el óptimo local.

Búsqueda Local Iterada

Para la solución mediante búsqueda local iterada, se utiliza una perturbación la cual se basa en intercambios aleatorios de trabajos dentro de la secuencia. Concretamente, se seleccionan dos posiciones al azar y se intercambian los trabajos correspondientes. En este caso se emplea una fuerza de cinco intercambios, lo que genera una modificación moderada de la solución. Esta elección permite que la nueva solución se ubique fuera del óptimo local previo, favoreciendo la exploración de nuevas regiones del espacio de búsqueda, pero manteniendo suficiente estructura para que la búsqueda local posterior pueda explotarla eficientemente.

La perturbación cumple un rol de diversificación, mientras que la búsqueda local actúa como mecanismo de intensificación. La combinación de ambos procesos permite balancear exploración y explotación, reduciendo la probabilidad de quedar atrapado permanentemente en un único óptimo local y aumentando la calidad de las soluciones obtenidas.

Esta solución, adopta una estrategia de *first improve,* es decir, en el momento que encuentre una solución con un mejor *makespan,* termina la ejecución. Por otra parte, se estableció un máximo de 30 iteraciones.

En términos de complejidad, cada iteración de esta solución incluye una búsqueda local completa, cuyo costo es del orden de $O(n^3m)$. La perturbación tiene un costo muy bajo en comparación, ya que implica un número constante de intercambios. Si el algoritmo se ejecuta durante k iteraciones, la complejidad global es aproximadamente $O(kn^3m)$, lo que explica que ILS requiera mayor tiempo computacional que la búsqueda local simple, a cambio de una mayor capacidad de exploración del espacio de soluciones. Nuevamente se utiliza la heurística NEH, ya que se utiliza la búsqueda local implementada anteriormente.

Resultados

A continuación, se presentan los resultados obtenidos al ejecutar los 4 algoritmos explicados anteriormente, con 5 casos de prueba basados en el benchmark de **Taillard.** Cabe destacar,que el algoritmo de **Branch&Bound**, terminó su ejecución para el primer caso de

prueba el cual es el más pequeño. Para los demás casos de prueba, luego de una hora de ejecución no devolvió resultado.

	Makespan	Tiempo de Ejecución (s)
NEH	420	0.001
LS	420	0.002
ILS	420	0.33
Branch&Bound	406	82

Tabla 1. Resultados para caso de prueba 20 trabajos x 5 máquinas

	Makespan	Tiempo de Ejecución (s)
NEH	493	0.002
LS	493	0.006
ILS	488	0.026

Tabla 2. Resultados para caso de prueba 20 trabajos x 10 máquinas

	Makespan	Tiempo de Ejecución (s)
NEH	1006	0.01
LS	994	0.22
ILS	994	11.24

Tabla 3. Resultados para caso de prueba 50 trabajos x 10 máquinas

	Makespan	Tiempo de Ejecución (s)
NEH	1753	0.15
LS	1745	0.86
ILS	1745	69.64

Tabla 4. Resultados para caso de prueba 100 trabajos x 10 máquinas

	Makespan	Tiempo de Ejecución (s)

NEH	1817	0.15
LS	1796	3.4
ILS	1795	9.19

Tabla 5. Resultados para caso de prueba 100 trabajos x 20 máquinas

Conclusión

El análisis experimental realizado permite concluir que el problema de taller de flujo de permutación representa un desafío donde el equilibrio entre el tiempo de cómputo y la calidad de la solución es el factor determinante. A través de la implementación del algoritmo Branch and Bound, se constató la inviabilidad de los métodos exactos para entornos industriales dinámicos; aunque este método garantiza el óptimo global, como se observó en la instancia de 20x5, su crecimiento exponencial en el tiempo de ejecución lo hace prohibitivo para problemas de mayor escala. Esta limitación técnica justifica la transición hacia métodos aproximados, los cuales logran soluciones competitivas en una fracción del tiempo original.

En este contexto, la heurística NEH se consolidó como la herramienta más eficiente de este estudio. Su capacidad para generar resultados robustos de manera instantánea la posiciona como la base ideal sobre la cual construir estrategias de optimización más complejas. Los resultados muestran que, si bien el NEH ofrece una solución inicial sólida, su combinación con técnicas de Búsqueda Local (LS) y Búsqueda Local Iterada (ILS) permite alcanzar niveles de optimización superiores. La búsqueda local demostró que el entorno de inserción es altamente efectivo para refinar el cronograma, mientras que el ILS, mediante su mecanismo de perturbación, logró escapar de óptimos locales para hallar mejores valores de *makespan* en las instancias más críticas de Taillard, a pesar de requerir un esfuerzo computacional mayor.

Finalmente, el uso del Benchmark de Taillard fue fundamental para validar la consistencia de los algoritmos propuestos frente a estándares internacionales. La experimentación deja claro que para la gestión de sistemas de manufactura, la elección del algoritmo debe depender de la urgencia de la decisión: mientras que el NEH es óptimo para una planificación inmediata, el ILS representa la mejor opción cuando se dispone de un margen de tiempo mayor para maximizar la utilización de los recursos. En definitiva, este estudio reafirma que la restricción de permutación, lejos de ser una limitante, permite aplicar lógicas de inserción y búsqueda que son fundamentales para la competitividad en la logística y producción moderna.

Anexos

Se anexan las implementaciones de las soluciones en el lenguaje de programación C++:

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <limits>
#include <chrono>

using namespace std;

// Función para calcular la matriz de tiempos de finalización para una secuencia parcial
vector<vector<int>> calculate_partial_completion(const vector<int>& partial_seq,
                                                 const vector<vector<int>>& processing_times) {
    /*
    Calcula la matriz de tiempos de finalización para una secuencia parcial de trabajos
    en un Permutation Flow Shop.

    Args:
        partial_seq: Índices de trabajos ya secuenciados
        processing_times: Matriz MxN de tiempos de procesamiento (M máquinas x N trabajos)

    Returns:
        Matriz de tiempos de finalización de tamaño M x len(partial_seq)
    */

    if (partial_seq.empty()) {
        return vector<vector<int>>(processing_times.size(), vector<int>());
    }

    int num_machines = processing_times.size();
    int num_jobs = partial_seq.size();
    vector<vector<int>> C(num_machines, vector<int>(num_jobs, 0));
```

```
    for (int j = 0; j < num_jobs; j++) {
        int job = partial_seq[j];
        for (int m = 0; m < num_machines; m++) {
            if (j == 0 && m == 0) {
                C[m][j] = processing_times[m][job];
            } else if (j == 0) {
                C[m][j] = C[m-1][j] + processing_times[m][job];
            } else if (m == 0) {
                C[m][j] = C[m][j-1] + processing_times[m][job];
            } else {
                C[m][j] = max(C[m-1][j], C[m][j-1]) +
processing_times[m][job];
            }
        }
    }

    return C;
}

// Función para calcular la cota inferior en Branch and Bound
int lower_bound_bb(const vector<vector<int>>& C_partial,
                   const vector<int>& remaining_jobs,
                   const vector<vector<int>>& processing_times) {
    /*
    Calcula una cota inferior realista para una secuencia parcial en
Branch and Bound.

    La cota inferior se calcula como el máximo entre los tiempos de
finalización actuales
    de la secuencia parcial y una estimación optimista de los trabajos
restantes.
    */

    if (remaining_jobs.empty()) {
        if (C_partial.empty() || C_partial[0].empty()) {
            return 0;
        }
        return C_partial.back().back(); // Último elemento de la última
máquina
    }

    int num_machines = processing_times.size();
```

```
    // Últimos tiempos de finalización de la secuencia parcial
    vector<int> last_times(num_machines, 0);
    if (!C_partial.empty() && !C_partial[0].empty()) {
        for (int m = 0; m < num_machines; m++) {
            last_times[m] = C_partial[m].back();
        }
    }

    // Estimación optimista: suma de tiempos de procesamiento de
trabajos restantes por máquina
    vector<int> est_remaining(num_machines, 0);
    for (int m = 0; m < num_machines; m++) {
        for (int job : remaining_jobs) {
            est_remaining[m] += processing_times[m][job];
        }
    }

    // Calcular el máximo considerando las restricciones de las máquinas
    int bound = 0;
    for (int m = 0; m < num_machines; m++) {
        int machine_bound = last_times[m] + est_remaining[m];
        if (machine_bound > bound) {
            bound = machine_bound;
        }
    }

    return bound;
}

// Variables globales para la recursión
int best_flowtime;
vector<int> best_sequence;

// Función recursiva de exploración en Branch and Bound
void explore(const vector<int>& partial_seq,
             const vector<int>& remaining_jobs,
             const vector<vector<int>>& C_partial,
             const vector<vector<int>>& processing_times) {

    // Calcular cota inferior
    int lb = lower_bound_bb(C_partial, remaining_jobs,
processing_times);
```

```
    // Podar si la cota es mayor o igual al mejor flujo encontrado
    if (lb >= best_flowtime) {
        return;
    }

    // Si no quedan trabajos, hemos encontrado una secuencia completa
    if (remaining_jobs.empty()) {
        if (!C_partial.empty() && !C_partial[0].empty()) {
            int current_flowtime = C_partial.back().back();
            if (current_flowtime < best_flowtime) {
                best_flowtime = current_flowtime;
                best_sequence = partial_seq;
            }
        }
        return;
    }

    // Explorar todos los trabajos restantes
    for (size_t i = 0; i < remaining_jobs.size(); i++) {
        // Crear nueva secuencia parcial
        vector<int> new_seq = partial_seq;
        new_seq.push_back(remaining_jobs[i]);

        // Crear nueva lista de trabajos restantes
        vector<int> new_remaining;
        for (size_t j = 0; j < remaining_jobs.size(); j++) {
            if (j != i) {
                new_remaining.push_back(remaining_jobs[j]);
            }
        }

        // Calcular nueva matriz de finalización
        vector<vector<int>> C_new =
calculate_partial_completion(new_seq, processing_times);

        // Llamada recursiva
        explore(new_seq, new_remaining, C_new, processing_times);
    }
}

// Función principal de Branch and Bound
pair<vector<int>, int> branch_and_bound(const vector<vector<int>>&
processing_times) {
```

```
    /*
    Resuelve el problema PFSP usando Branch and Bound exacto.

    Args:
        processing_times: Matriz MxN de tiempos de procesamiento

    Returns:
        pair<mejor_secuencia, mejor_flowtime>
    */

    int num_jobs = processing_times[0].size();

    // Inicializar variables globales
    best_flowtime = numeric_limits<int>::max();
    best_sequence.clear();

    // Crear lista de todos los trabajos
    vector<int> all_jobs;
    for (int i = 0; i < num_jobs; i++) {
        all_jobs.push_back(i);
    }

    // Iniciar exploración
    vector<int> empty_seq;
    vector<vector<int>> empty_C(processing_times.size(), vector<int>());
    explore(empty_seq, all_jobs, empty_C, processing_times);

    return make_pair(best_sequence, best_flowtime);
}

// Función para transponer una matriz
vector<vector<int>> transpose_matrix(const vector<vector<int>>& matrix)
{
    if (matrix.empty()) return {};

    int num_machines = matrix[0].size();
    int num_jobs = matrix.size();

    vector<vector<int>> transposed(num_machines, vector<int>(num_jobs,
0));

    for (int i = 0; i < num_jobs; i++) {
        for (int j = 0; j < num_machines; j++) {
```

```
            transposed[j][i] = matrix[i][j];
        }
    }

    return transposed;
}
```

Anexo 1. Implementación Algoritmo Branch & Bound

```
#include <iostream>
#include <vector>
#include <numeric>
#include <algorithm>
#include "NEH.h"

using namespace std;

// Funcion para calcular el makespan (Cmax)
int calcular_makespan(const vector<int>& secuencia, const
vector<vector<int>>& tiempos, int n_maquinas) {
    int n_tareas = secuencia.size();
    if (n_tareas == 0) return 0;

    vector<vector<int>> C(n_tareas + 1, vector<int>(n_maquinas + 1, 0));
    for (int i = 1; i <= n_tareas; ++i) {
        for (int j = 1; j <= n_maquinas; ++j) {
            C[i][j] = max(C[i-1][j], C[i][j-1]) +
tiempos[secuencia[i-1]][j-1];
        }
    }
    return C[n_tareas][n_maquinas];
}

vector<int> neh(int n, int m, const vector<vector<int>>& tiempos) {
    // Ordenar tareas por suma total de tiempos
    vector<pair<int, int>> sum_tiempos;
    for (int i = 0; i < n; ++i) {
        int sum = accumulate(tiempos[i].begin(), tiempos[i].end(), 0);
        sum_tiempos.push_back({sum, i});
    }
    sort(sum_tiempos.begin(), sum_tiempos.end(), greater<pair<int,
int>>());

    vector<int> secuencia_actual;
```

```
    int mejor_makespan_final = 0;

    //Insercion iterativa de tareas
    for (int i = 0; i < n; ++i) {
        int tarea_actual = sum_tiempos[i].second;
        int mejor_makespan_iteracion = 2e9;
        vector<int> mejor_secuencia_iteracion;

        for (int pos = 0; pos <= (int)secuencia_actual.size(); ++pos) {
            vector<int> temp_secuencia = secuencia_actual;
            temp_secuencia.insert(temp_secuencia.begin() + pos,
tarea_actual);

            int ms = calcular_makespan(temp_secuencia, tiempos, m);
            if (ms < mejor_makespan_iteracion) {
                mejor_makespan_iteracion = ms;
                mejor_secuencia_iteracion = temp_secuencia;
            }
        }
        secuencia_actual = mejor_secuencia_iteracion;
        mejor_makespan_final = mejor_makespan_iteracion;
    }
    return secuencia_actual;
}
```

Anexo 2. Implementación Algoritmo Heurística NEH

```
#include "LS.h"
#include "NEH.h"
#include <iostream>
using namespace std;

vector<int> local_search_insertion(vector<int> secuencia,
                                   const vector<vector<int>>& tiempos,
                                   int m) {
    bool mejora = true;
    int n = secuencia.size();

    while (mejora) {
        mejora = false;
        int mejor_makespan = calcular_makespan(secuencia, tiempos, m);
        vector<int> mejor_vecino = secuencia;

        for (int i = 0; i < n; ++i) {
```

```
            for (int j = 0; j < n; ++j) {
                if (i == j) continue;

                vector<int> vecino = secuencia;
                int tarea = vecino[i];
                vecino.erase(vecino.begin() + i);
                vecino.insert(vecino.begin() + j, tarea);
                int ms = calcular_makespan(vecino, tiempos, m);

                if (ms < mejor_makespan) {
                    mejor_makespan = ms;
                    mejor_vecino = vecino;
                    mejora = true;
                }
            }
        }
        secuencia = mejor_vecino;
    }
    return secuencia;
}
```

Anexo 3. Implementación Búsqueda Local

```
#include "ILS.h"
#include "LS.h"
#include "NEH.h"
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

vector<int> perturbacion_swap(vector<int> secuencia, int fuerza) {
    int n = secuencia.size();

    for (int k = 0; k < fuerza; ++k) {
        int i = rand() % n;
        int j = rand() % n;
        swap(secuencia[i], secuencia[j]);
    }
    return secuencia;
}
```

```
vector<int> ILS(const vector<int>& solucion_inicial,
                const vector<vector<int>>& tiempos,
                int m,
                int max_iters) {

   vector<int> actual = local_search_insertion(solucion_inicial,
tiempos, m);
   vector<int> mejor = actual;

   int mejor_coste = calcular_makespan(mejor, tiempos, m);

   for (int iter = 0; iter < max_iters; ++iter) {

       // Perturbacion
       vector<int> perturbada = perturbacion_swap(actual, 5);

       // Busqueda local
       vector<int> refinada = local_search_insertion(perturbada,
tiempos, m);
       int coste = calcular_makespan(refinada, tiempos, m);

       // Aceptacion
       if (coste < mejor_coste) {
           mejor = refinada;
           mejor_coste = coste;
           break;
       }
       actual = refinada;
   }
   return mejor;
}
```

Anexo 4. Implementación Búsqueda Local Iterativa