Universidad de Santiago
Facultad de Ingeniería
Departamento de Ingeniería Informática

TAREA 2 LABORATORIO AAyED

Optimización de Procesamiento de Cargas

.

Integrantes: Aylin Castillo
Profesor: Cristián Sepúlveda
Fecha de realización: 28 de Junio del 2024
Fecha de entrega: 1 de julio de 2024
Santiago, Chile

Índice de Contenidos

Índice de Figuras

1. Introducción

En el presente informe se documentará el problema presentado para realizar la tarea 2 del laboratorio de la asignatura de Análisis de Algoritmos y Estructura de Datos, el cual se sitúa en la industria del transporte marítimo, específicamente en la empresa llamada Maersk Line, que está teniendo problemas con la optimización de los procesos de embarque de cargas. Estas deben pasar por una serie de procesos antes de ser enviadas a su destino final. Para esto, se solicitó hacer un programa en lenguaje C que minimice el tiempo total necesario para el procesamiento de las cargas dadas en un orden secuencial, cumpliendo con distintas restricciones especificadas.

Para realizar este programa, se hizo uso de tres estructuras de datos vistas en la cátedra del curso, las cuales son: lista enlazada, pila y cola. Estas se implementaron teniendo en cuenta los siguientes objetivos:

- Análisis de resultado obtenido.
- Obtener el tiempo mínimo posible para el procesamiento de cargas.
- Hacer uso del TDA lista enlazada, pila y cola en la implementación de la solución.
- Calcular correctamente la complejidad del algoritmo propuesto.

2. Método

Para la resolución del problema presentado anteriormente se hizo un computador modelo OMEN laptop 15 con procesador I5 10-3000 2.5 GHz, Windows 10.

Se usó como IDE y editor de código al programa Visual Studio Code para la implementación del código.

Para poner a prueba el programa realizado, se utilizaron los distintos ejemplos entregados junto al enunciado del problema, que contenían la cantidad de cargas, los procesos y el respectivo orden de los procesos por el que debe pasar cada carga con sus tiempos dados. Estos fueron probados en el siguiente algoritmo:

1. **TDA's utilizados**:
    - **TDA lista enlazada simple**: estructura de datos que consiste en la conexión de nodos que tienen dos partes: dato y puntero. En la implementación, se modificó para el problema específico, como se muestra a continuación:

```
typedef struct nodo{
    int carga;
    int proceso;
    struct nodo *siguiente;
    }nodo;


typedef struct{
    nodo *inicio;
    }lista;
```

Figura 1: Estructura TDA lista

- **TDA lista pila**: igual a la lista descrita anteriormente, pero almacena un puntero a una pila junto a la respectiva posición de la carga, la cual es equivalente a su número distintivo.

```
typedef struct nodoListaPila{
    int posicion;
    pila *pilaDatos;
    struct nodoListaPila *siguiente;
    }nodoListaPila;


typedef struct{
    nodoListaPila *inicio;
    }listaPila;
```

Figura 2: Estructura TDA lista pila

- **TDA lista cola**: igual a la lista descrita anteriormente, pero almacena un puntero a una cola, junto al número del proceso que se va a utilizar.

```
typedef struct nodoListaCola{
    int numeroProceso;
    cola *colaProceso;
    struct nodoListaCola *siguiente;
    }nodoListaCola;


typedef struct{
    nodoListaCola *inicio;
    }listaCola;
```

Figura 3: Estructura TDA lista cola

- **TDA pila**: estructura de datos que solo tiene operaciones que permiten ver el tope de la pila, apilar y desapilar. Fue modificada para tener el proceso y el tiempo en orden para cada carga.

```c
typedef struct nodoPila{
    int proceso;
    int tiempo;
    struct nodoPila *siguiente;
    }nodoPila;

typedef struct{
    int size;
    nodoPila *tope;
    }pila;
```

Figura 4: Estructura TDA pila

- **TDA cola**: estructura de datos que solo tiene operaciones para ver el inicio y el final de una lista simple enlazada. Tiene operaciones como encolar y descolar, que modifican el primer valor de la cola. En este caso, fue estrictamente utilizada para guardar el orden del procesamiento de todas las cargas. Cada cola representa un proceso, teniendo como dato su número distintivo.

```c
typedef struct nodoCola{
    int dato;
    struct nodoCola *siguiente;
    }nodoCola;


typedef struct{
    int size;
    nodoCola *n_frente;
    nodoCola *n_final;
    }cola;
```

Figura 5: Estructura TDA cola

2. **Algoritmo principal**: El siguiente bloque de código está encargado de realizar el procesamiento de las cargas extraídas del archivo ingresado como entrada. Inicialmente, crea una lista enlazada auxiliar para ir almacenando los procesos actuales que se estuvieron utilizando, cumpliendo así con la restricción de que un proceso solo puede procesar una carga a la vez. Después, ejecuta la función **"sePuedeAsignar"**, la cual tiene como objetivo revisar si hay procesos disponibles. En el caso positivo, ejecuta la función **"asignarCargas"**, la cual tiene como objetivo actualizar la lista de procesos actuales con las nuevas cargas que se van a procesar. Después de esto, ejecuta la función **"avanzarProceso"**, la cual tiene como objetivo restar uno al tiempo de las cargas que estén en la lista de procesos actuales y modificarlas en el caso de que el proceso que se estaba realizando quede con tiempo igual a cero. Estas funciones serán explicadas a mayor detalle en las siguientes secciones. Esta función termina cuando la lista de cargas queda vacía y retorna el valor final del tiempo en que se procesaron todas las cargas con éxito.

```
Función principal:

procesamientoCargas(colasProcesos, pilasCargas, cantidadProcesos): num
procesosActuales ← crearListaVacia()
Para i ← cantidadProcesos  hasta 1:
   insertaNodoInicio(procesosActuales, 0, i)
tiempo ← 0
Mientras no (esListaPilaVacia(pilasCargas)) hacer:
   Si (sePuedeAsignar(procesosActuales, pilasCargas)):
      asignarCargas(procesosActuales, pilasCargas, colasProcesos)
      avanzarProcesamiento(procesosActuales, pilasCargas)
      tiempo ← tiempo + 1
   Sino:
      avanzarProcesamiento(procesosActuales, pilasCargas)
      tiempo ← tiempo + 1
liberarLista(procesosActuales)
devolver tiempo
```

Su complejidad es de $O(n^2 * m + n * m^2)$ por la utilización de las funciones **sePuedeAsignar** ($O(n * m)$), **asignarCargas** ($O(n * m^2)$) y **avanzarProcesamiento** ($O(n^2 * m)$) dentro del ciclo while que depende de la cantidad de cargas almacenadas en la listaPila. Cabe mencionar que aquí domina el orden de complejidad con el exponente más alto para así ponernos en la peor situación del problema presentado.

3. **Función sePuedeAsignar**: como se mencionó anteriormente, esta función es la encargada de verificar si existe algún proceso disponible para procesar alguna de las cargas. Revisa la lista de procesos actuales y al mismo tiempo los topes de las cargas representadas como pilas, para ver si alguna de ellas tiene ese proceso disponible y poder asignarlas más adelante. Una vez completado este proceso, retorna un 1 en caso de que haya al menos un proceso disponible y un 0 en caso contrario, simulando un booleano.

```
sePuedeAsignar(lista procesosActuales, listaPila pilasCargas): num
auxProcesos ← procesosActuales → inicio
Mientras auxProcesos <> nulo hacer:
   Si (auxProcesos → carga = 0):
      auxListaCargas ← pilasCargas → inicio
      Mientras auxListaCargas <> nulo hacer:
         Si ((tope(auxListaCargas → pilaDatos)→ proceso) = (auxProcesos→ proceso)):
            devolver 1
         auxListaCargas ← auxListaCargas→siguiente
   auxProcesos ← auxProcesos→siguiente
devolver 0
```

Su complejidad es de $O(n * m)$ debido al ciclo while (mientras) anidado que se ejecutará n veces según la cantidad de procesos (m) y cargas dadas (n).

4. **Función asignarCargas**: como se mencionó anteriormente, esta función es la encargada de actualizar la lista de procesos actuales con las cargas que tienen en el tope algún proceso disponible. A la vez, actualiza la listaCola de procesos realizados, la cual lleva el historial de todas las cargas procesadas hasta el momento según los procesos asignados.

```
asignarCargas(lista procesosActuales, listaPila pilasCargas, listaCola procesosRealizados)
auxProcesos ← procesosActuales.inicio
Mientras auxProcesos <> nulo hacer:
   auxListaCargas ← pilasCargas → inicio
   Mientras auxListaCargas <> nulo hacer:
      Si ((tope(auxListaCargas→pilaDatos)→proceso) = (auxProcesos→proceso)
       y (auxProcesos → carga = 0)):
        auxRealizados ← procesosRealizados→inicio
        auxProcesos→carga ← auxListaCargas→posicion
        Mientras auxRealizados <> nulo hacer:
           Si (auxRealizados→numeroProceso = auxProcesos→proceso):
              encolar(auxRealizados→colaProceso, auxListaCargas→posicion)
           auxRealizados ← auxRealizados→siguiente
      auxListaCargas ← auxListaCargas→siguiente
   auxProcesos ← auxProcesos→siguiente
```

Su complejidad es de $O(n * m^2)$ debido a que uno de los ciclos while (mientras) anidados depende de la cantidad de cargas (n) y el otro depende de la cantidad de procesos (m) al igual que el primer ciclo. También se hace uso de la función **encolar** (TDA cola) que tiene orden $O(1)$.

5. **Función avanzarProcesamiento**: como se mencionó anteriormente, esta función está encargada de restar el tiempo de las cargas que están en la lista de procesos actuales en 1 hasta llegar a 0. Cuando estas lleguen a 0, se desapilará la pila de la carga respectiva y en el caso de que esta pila quede vacía, se eliminará de la lista de pilas con la función **eliminarNodoListaPila**, la cual buscará la pila por su respectiva posición (número de carga).

```
avanzarProcesamiento(lista procesosActuales, listaPila pilasCargas):
auxProcesos ← procesosActuales→inicio
Mientras auxProcesos <> nulo hacer:
    auxCargas ← pilasCargas→inicio
    Mientras auxCargas <> nulo hacer:
        Si (auxProcesos→carga = auxCargas→posicion y tope(auxCargas→pilaDatos)→tiempo <> 0):
            tiempoCargaActual ← tope(auxCargas→pilaDatos)→tiempo
            tiempoCargaActual ← tiempoCargaActual - 1
            tope(auxCargas→pilaDatos)→tiempo ← tiempoCargaActual
        Si (tope(auxCargas→pilaDatos)→tiempo = 0):
            desapilar(auxCargas→pilaDatos)
            auxProcesos→carga ← 0
            Si es_pila_vacia(auxCargas→pilaDatos):
                eliminarNodoListaPila(pilasCargas, auxCargas→posicion)
        auxCargas ← auxCargas→siguiente
    auxProcesos ← auxProcesos→siguiente
```

Su complejidad es de $O(n^2 * m)$ debido a que el ciclo while (mientras) anidado depende de la cantidad de cargas (n) que tenga la listaPila y por el uso de la función **eliminarNodoListaPila** la cual tiene orden $O(n)$. Junto a esto se hace uso de la función **desapilar** (TDA pila) que tiene orden $O(1)$. Todas estas funciones anidadas dependen de la cantidad de procesos, como se ve en la condición del primer ciclo que tiene la función.

3. Resultados y Análisis

Después de realizadas las pruebas con los 2 archivos de ejemplo, se obtuvieron los siguientes resultados:

- **Cálculo del tiempo mínimo**: archivos "procesamiento_3_3.in" con 3 cargas y 3 procesos y "archivo procesamiento_4_4.in" con 4 cargas y 4 procesos.

Proceso	Orden de cargas
1	1 2 3
2	3 1 2
3	2 1 3

Proceso	Orden de cargas
1	1 2 4 3
2	1 3 4 2
3	3 1 2 4
4	3 1 2 4

El primer archivo da un tiempo total de 12 y el segundo archivo da un tiempo total de 419.

- **Tablas según la cantidad de cargas y procesos**:

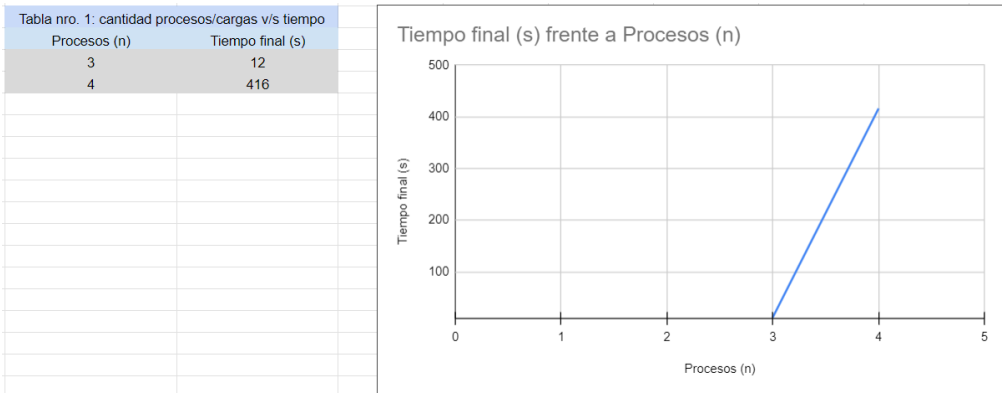
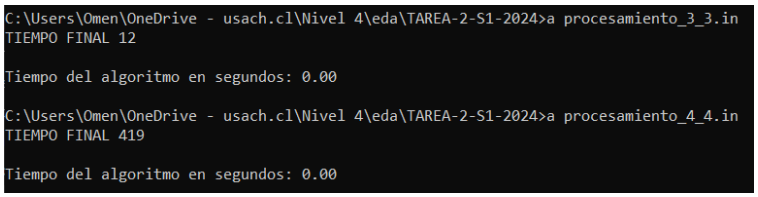


Tabla nro. 1: cantidad procesos/cargas v/s tiempo	
Procesos (n)	Tiempo final (s)
3	12
4	416

Figura 6: Tabla y gráfico cantidad procesos v/s tiempo

- **Tiempo de ejecución**: el algoritmo entregó con ambos ejemplos el valor de 0.00, por lo que no se puede hacer algún análisis sobre esto comparando ambos archivos, más allá de confirmar que es eficaz.



```
C:\Users\Omen\OneDrive - usach.cl\Nivel 4\eda\TAREA-2-S1-2024>a procesamiento_3_3.in
TIEMPO FINAL 12

Tiempo del algoritmo en segundos: 0.00

C:\Users\Omen\OneDrive - usach.cl\Nivel 4\eda\TAREA-2-S1-2024>a procesamiento_4_4.in
TIEMPO FINAL 419

Tiempo del algoritmo en segundos: 0.00
```

Figura 7: Tiempo de ejecución por ejemplo

De lo anterior, se puede observar que es evidente que el valor final del tiempo depende de la cantidad de cargas y procesos que se evalúan, ya que a medida que estas aumentan, aumenta también el tiempo final.

Por otro lado, fue entregado un excel con los valores óptimos que debería entregar el algoritmo para cada ejemplo, los cuáles fueron los siguientes:

Nombre ejemplo	Solución óptima
procesamiento_3_3.in	12
procesamiento_4_4.in	361

Figura 8: Tiempo final esperado

De estos fue posible cumplir con el tiempo del archivo “procesamiento_3_3.in”, pero con el archivo “procesamiento_4_4.in” hubo una diferencia de 58 segundos, lo cual es muy cercano a lo esperado, por lo que se podría decir que el algoritmo cumplió con el objetivo principal de la optimización del procesamiento de cargas.

4. Discusión

Finalizada la etapa de pruebas del algoritmo se puede comprobar que se hizo uso del TDA lista enlazada, pila y cola en la implementación de la solución presentada para el problema de optimización de cargas, ya que se aplicó el TDA pila para manejar las cargas, el TDA cola para los procesos y el TDA lista para mantener un conjunto de cargas y procesos y así controlar de mejor manera la información obtenida. Junto a esto se calculó el orden de complejidad de cada función que estuviera compuesta de alguno de los TDA mencionados, los cuales se pueden ver a detalle en la sección de "Resultados y Análisis", donde también se concluyó que el orden de complejidad final es de $O(n^2 * m + n * m^2)$, siendo "n" la cantidad de cargas y "m" la cantidad de procesos.

Como fue mencionado al final de la sección de "Resultados y Análisis", al comparar los resultados obtenidos con los resultados esperados, se tuvo una leve diferencia de 58 segundos en el archivo "procesamiento_4_4.in", pero en el otro archivo dio el mismo tiempo, lo que indica que el algoritmo se aproximó al mínimo tiempo esperado para el procesamiento de las cargas con las restricciones dadas e implementadas en las distintas funciones presentadas.

Por otro lado, las limitaciones que se tuvieron al implementar la solución fueron principalmente que no dio el tiempo final esperado. En primer lugar, se consideró otra propuesta de solución que incluía una çarga en espera", la cual priorizaba el procesamiento de una carga que hubiera tenido un proceso repetido. Esto se basaba en çiclos"por cada lista auxiliar que almacenaba los topes de las cargas con su respectiva información. Sin embargo, este algoritmo arrojaba resultados aún más distantes, ya que con el archivo "procesamiento_3_3.in" tomaba 14 segundos y con "procesamiento_4_4.in" tomaba 512 segundos. Debido a esto, se decidió abordar el problema con otro enfoque, ya que el enfoque anterior dependía principalmente de las cargas. En cambio, la solución presentada en este informe se centra en los procesos y en el tiempo necesario para terminar el procesamiento de las cargas en paralelo, sin priorizar alguna con proceso repetido, revisando permanentemente si se vacía la lista de procesos actuales para asignarle cargas.

Además, como se tenía que hacer uso del TDA lista, pila y cola, se tuvo que pensar en la solución haciendo el mejor uso posible de estos y sus respectivas operaciones, dejando atrás la costumbre de la utilización de arreglos e índices.

5. Conclusión

Para concluir, en lo que respecta al problema de búsqueda del mínimo tiempo de procesamiento de cargas para la empresa marítima Maersk Line, que maneja n cargas y m procesos, se logró presentar un algoritmo que pudo acercarse a los valores esperados del tiempo final de los ejemplos entregados, alcanzando en uno de ellos el tiempo exacto buscado.

Los algoritmos presentados cumplen en su totalidad con la restricción de solo utilizar los TDA lista, pila y cola vistos en la cátedra del curso, los cuales fueron modificados específicamente para este problema, respetando la estructura y operaciones de cada TDA.

Por otro lado, se puede confirmar que el algoritmo es eficaz y eficiente, ya que logró ejecutar ambos ejemplos en un tiempo total de 0.00 segundos para dar el resultado final, y además logra aproximarse a los tiempos mínimos esperados.

6. Apéndice

Para poder hacer uso del programa entregado, se debe seguir con los siguientes pasos:

1. Comprobar que se tiene el archivo principal “T2_A1_Aylin_Castillo.c”, los TDA .h y sus implementaciones junto a ambos archivos .in en la misma carpeta.

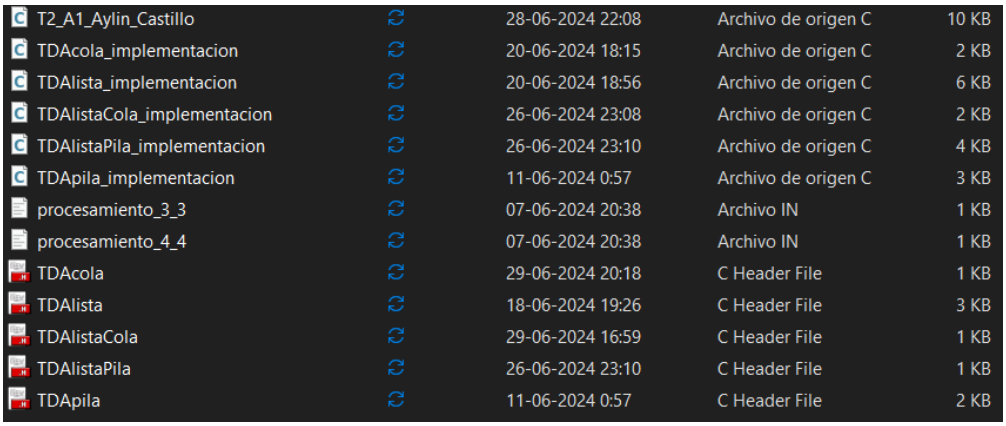


Figura 9: Archivos necesarios

2. Compilar el archivo “T2_A1_Aylin_Castillo.c” junto a los .c de la implementación en la consola cmd en el caso de uso de Windows.

```
C:\Users\Omen>cd C:\Users\Omen\OneDrive - usach.cl\Nivel 4\eda\TAREA-2-S1-2024\T2_A1_Aylin_Castillo

C:\Users\Omen\OneDrive - usach.cl\Nivel 4\eda\TAREA-2-S1-2024\T2_A1_Aylin_Castillo>gcc T2_A1_Aylin_Castillo.c TDAcola_implementacion.c
TDAlista_implementacion.c TDAlistaCola_implementacion.c TDAlistaPila_implementacion.c TDApila_implementacion.c
```

Figura 10: Archivos a compilar

3. Ejecutar el ejecutable junto al ejemplo que se quiere probar, de este solo va cambiando el número.

```
C:\Users\Omen\OneDrive - usach.cl\Nivel 4\eda\TAREA-2-S1-2024\T2_A1_Aylin_Castillo>a procesamiento_3_3.in
TIEMPO FINAL 12

Tiempo del algoritmo en segundos: 0.00
```

Figura 11: Ejecución del programa