



Programación concurrente

TRABAJO PRÁCTICO FINAL

Sistema Doble de Procesamiento de imágenes

Los maestros de la multitarea

Integrantes:

- Capdevila Gastón Ezequiel
- Quaglia Soteras Mateo
- Seia Nicolas

Profesores:

- Ing. Ventre Luis
 - Ing. Ludemann Mauricio
-

Índice

Índice.....	1
Introducción.....	2
Presentación de la red.....	4
Propiedades de la red.....	5
Análisis de invariantes.....	7
T-invariantes.....	7
P-invariantes.....	8
P-Invariant equations.....	8
Tabla de estados y eventos del sistema.....	9
Tabla de estados de la red final.....	9
Tabla de eventos de la red final.....	10
Determinación de la cantidad de hilos.....	11
Algoritmo para la determinación de hilos máximos activos simultáneos.....	11
Red de Petri final.....	12
Política.....	13
Política 1.....	13
Política 2.....	14
Implementación en java.....	15
Imagen.....	15
Tiempo.....	16
Política.....	17
RdP.....	18
Monitor.....	19
Semántica temporal.....	20
Red de petri con transiciones temporales.....	20
Tiempo considerando una secuencialización del programa.....	21
Tiempo considerando una absoluta concurrencia.....	22
Expresiones regulares.....	24
Diagramas.....	25
Conclusión.....	26

Introducción

El presente informe detalla el trabajo realizado para abordar un problema de programación concurrente mediante el uso de redes de Petri y monitores. La programación concurrente se ha convertido en una disciplina esencial en el desarrollo de sistemas modernos, ya que permite aprovechar la capacidad de procesamiento de múltiples hilos de ejecución para mejorar la eficiencia y el rendimiento de las aplicaciones.

Sin embargo, la programación concurrente también presenta complejidades y desafíos únicos, como condiciones de carrera, bloqueos y problemas de sincronización. En este contexto, las redes de Petri emergen como una poderosa herramienta para modelar y analizar sistemas concurrentes, brindando una representación gráfica y formal de los estados y transiciones del sistema.

El objetivo principal de este informe es presentar el enfoque adoptado para resolver un problema específico de programación concurrente utilizando redes de Petri para modelar el sistema y monitores para gestionar la sincronización y el acceso seguro a los recursos compartidos.

A lo largo del informe se hacen referencia a algunas definiciones que pasaremos a detallar ahora:

¿Que es una red de petri?

Una Red de Petri (RdP) es un modelo matemático y gráfico utilizado para describir y analizar sistemas concurrentes y distribuidos.

¿Por qué utilizamos RdP?

Permiten modelar y visualizar casos de la vida real con paralelismo, concurrencia, sincronización e intercambio de recursos. Además tienen un formalismo matemático y gráfico que nos permite determinar el disparo de una RdP y el siguiente marcado. Las RdP nos permiten separar la lógica de la política. La lógica es lo que puedo hacer, y la política es lo que me conviene hacer.

Invariante de transición

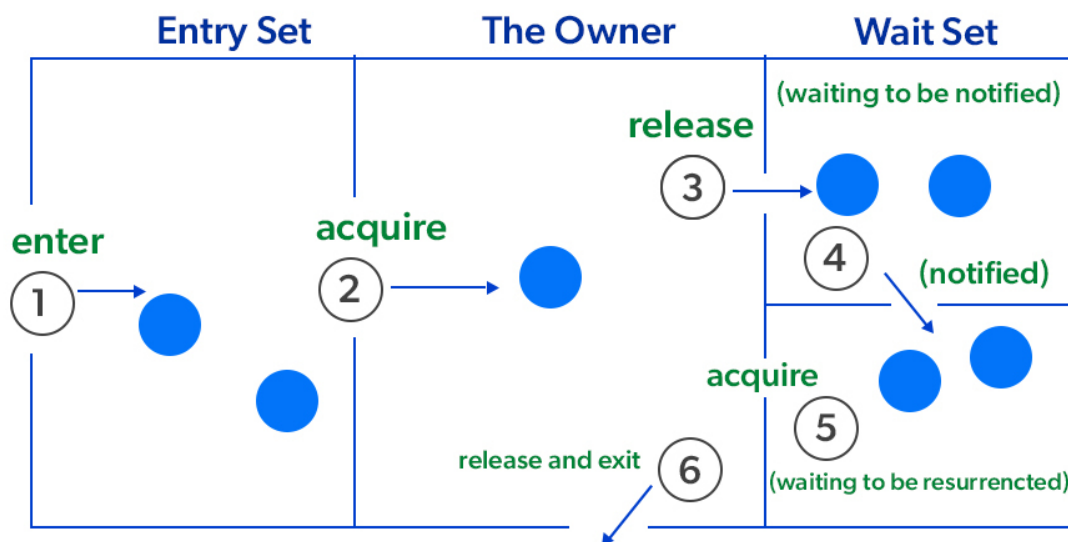
Es un vector conformado por números enteros asociados a una secuencia de disparos. Son útiles para determinar propiedades estructurales de una RdP en forma analítica. Un invariante de transición es el conjunto mínimo de transiciones que cuando las dispare vuelvo al estado inicial. Esto nos indica que algo se hizo. Si me fijo cuantos invariantes se completaron, voy a entender cuantos ciclos se completaron.

Invariante de plaza

Un invariante de plaza es el conjunto de plazas en donde la suma de sus tokens se mantiene constante a lo largo de todos los marcados de la red.

Monitor

Es un mecanismo de software (de alto nivel) para control de concurrencia que contiene los datos y los procedimientos necesarios para realizar la asignación de un determinado recurso o grupo de recursos compartidos reutilizables en serie.



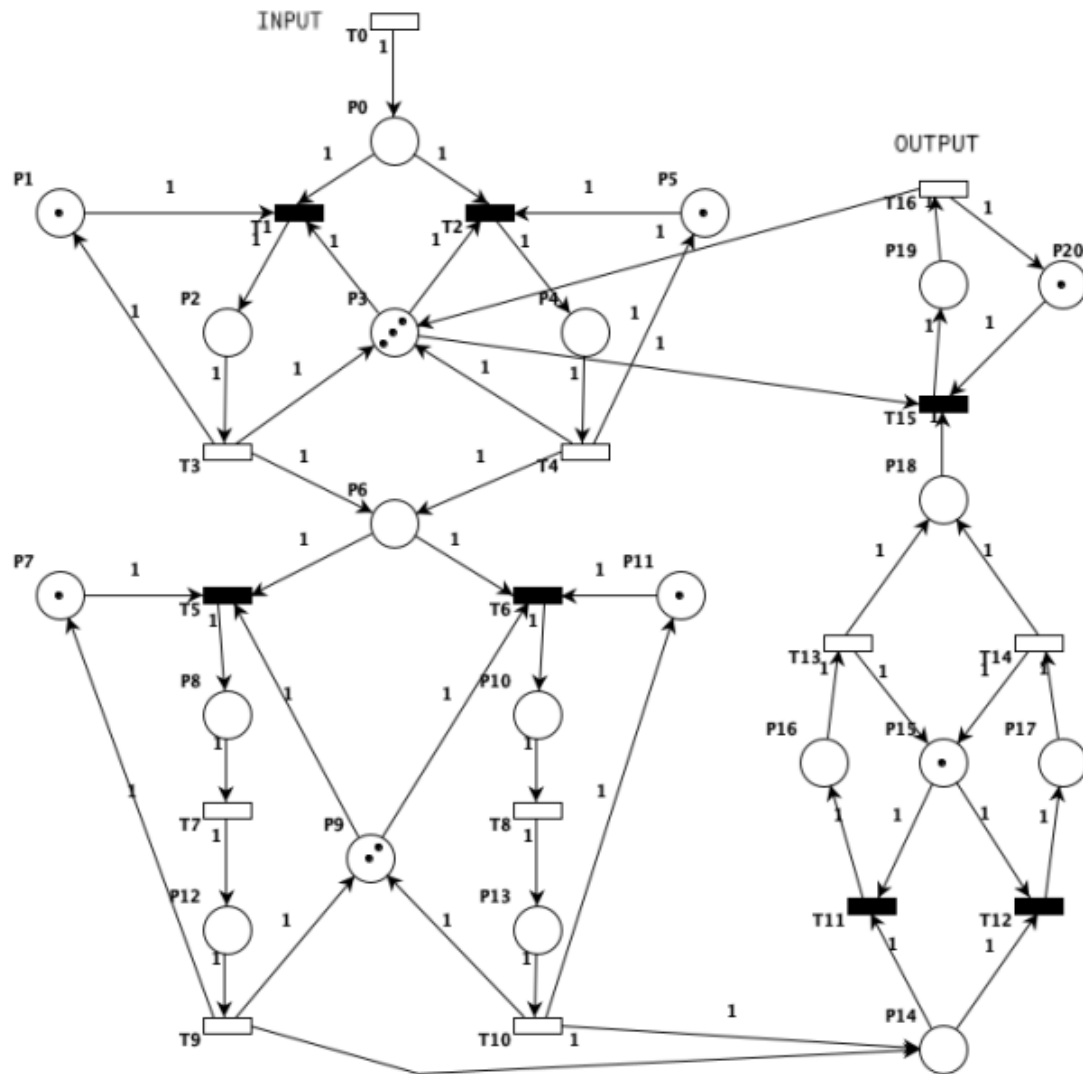
Política

Es un conjunto de reglas que define el comportamiento del monitor. En el cuál le otorgará más o menos prioridad a los distintos tipos de colas del monitor. Existen diversas políticas en la programación concurrente, y cada una tiene sus propias ventajas y desventajas dependiendo del escenario específico y de los objetivos del sistema.

- Signal and Continue (SC)
- Signal and Wait (SW)
- Signal and Urgent Wait (SU)
- Signal and Exit (SX)

Presentación de la red

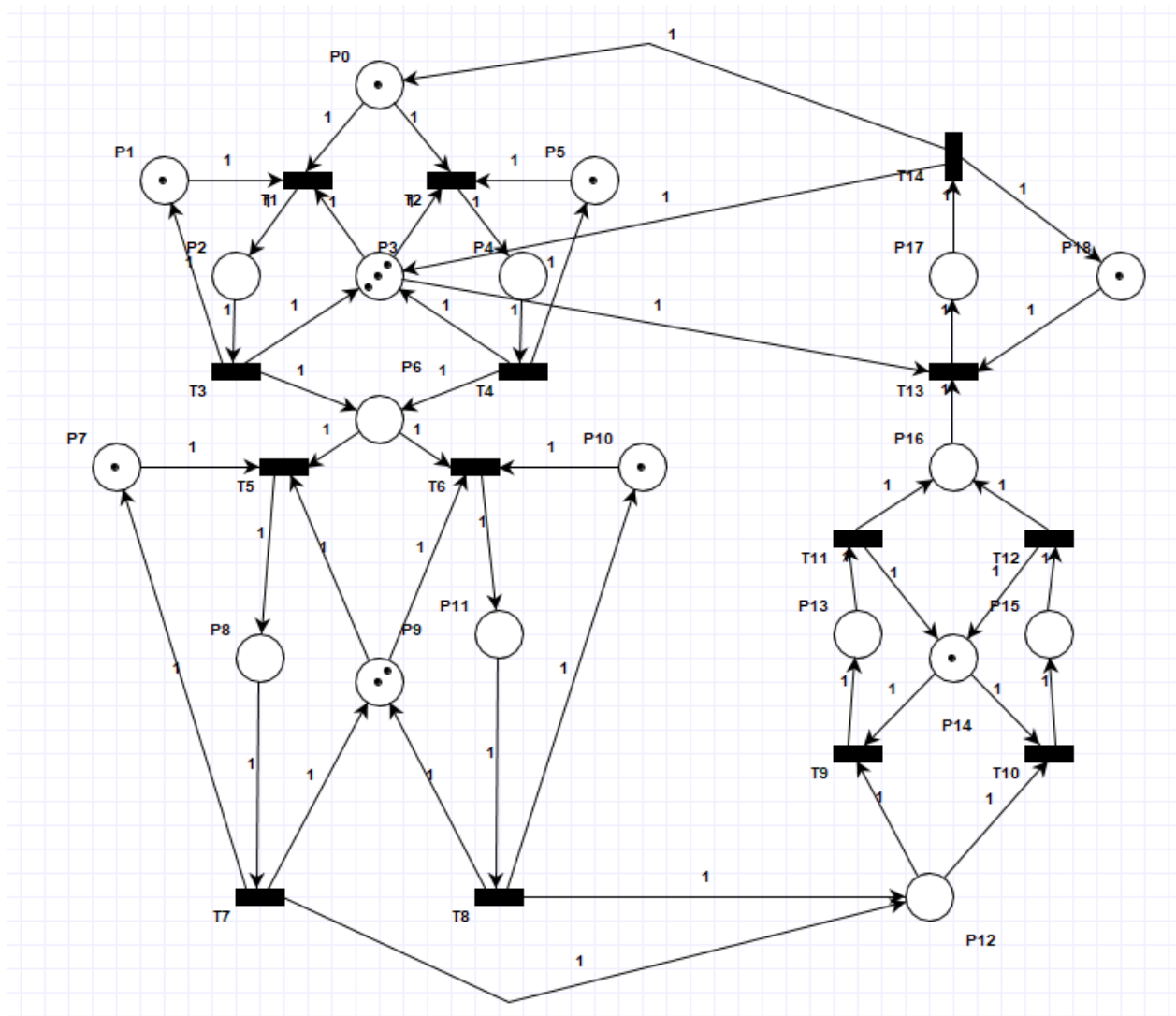
Red de Petri Sistema Doble de Procesamiento de imágenes



Propiedades de la red

Para la determinación de las propiedades de la red se le realizó un cambio para que pueda funcionar en la herramienta PIPE o PETRINATOR.

Red modificada:



Petri net classification results

State Machine	false
Marked Graph	false
Free Choice Net	false
Extended Free Choice Net	false
Simple Net	false
Extended Simple Net	true

Petri net state space analysis results

Bounded	true
Safe	false
Deadlock	false

¿Que significa que nuestra red sea una red limitada?

En nuestro caso la red original no sería limitada, pero con los cambios que le hicimos para poder analizar en pipe o petrinator si. Una RdP es considerada "limitada" cuando tiene una cantidad finita de marcas en cada plaza del sistema, es decir, existe un número máximo definido para la cantidad de fichas que pueden estar presentes en cada plaza.

¿Que significa que nuestra red no sea una red segura?

Se dice que una RdP es segura para una marca inicial si para cada marca alcanzable, cada lugar contiene un cero o un token. Nuestra red no es segura porque para cada marca existe más de un token en algunas plazas, eso quiere decir que puede haber más de una imagen en algunas plazas.

¿Qué significa que nuestra red no tenga deadlock?

Un Deadlock es una marca tal que no se puede disparar ninguna transición. Se dice que una RdP está libre de interbloqueo para una marca inicial m_0 si ninguna marca alcanzable es un Deadlock.

Entonces podemos decir que nuestra red es una **RdP viva** para la marca inicial m_0 , pero por las propiedades de vivacidad e interbloqueo podemos decir que: Si una RdP está libre de interbloqueo (deadlock) para m_0 , no es necesariamente así para $m' > m_0$.

Análisis de invariantes

T-invariantes

T7	T8	T6	T3	T2	T4	T9	T10	T11	T12	T13	T14	T1	T5
0	1	1	0	1	1	0	1	0	1	1	1	0	0
0	1	1	0	1	1	1	0	1	0	1	1	0	0
1	0	0	0	1	1	0	1	0	1	1	1	0	1
1	0	0	0	1	1	1	0	1	0	1	1	0	1
0	1	1	1	0	0	0	1	0	1	1	1	1	0
0	1	1	1	0	0	1	0	1	0	1	1	1	0
1	0	0	1	0	0	0	1	0	1	1	1	1	1
1	0	0	1	0	0	1	0	1	0	1	1	1	1

La red tiene T-Invariantes positivas, por lo que podría estar acotada y viva.

En nuestro modelo, los invariantes de transición desempeñan un papel fundamental al verificar y rastrear las secuencias o caminos que sigue una imagen, desde su creación hasta su exportación, es decir durante todo su proceso. Actúan como marcadores que registran el progreso de la imagen a medida que se procesa a través de los diferentes pasos, como mejora, recorte y exportación. En resumen, los invariantes de transición funcionan como una certificación que verifica que el programa sigue fielmente el camino predefinido para transformar una imagen antes de ser exportada.

P-invariantes

P6	P12	P16	P4	P1	P0	P5	P14	P7	P10	P9	P18	P3	P11	P8	P13	P15	P17	P2
1	1	1	1	0	1	0	0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	1	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	1	0	0	0	0	1	1

La red tiene P-invariantes positivas, por lo que está acotada.

P-Invariant equations

1) $M(P6) + M(P12) + M(P16) + M(P4) + M(P0) + M(P11) + M(P8) + M(P13) + M(P15) + M(P17) + M(P2) = 1$

2) $M(P1) + M(P2) = 1$

3) $M(P4) + M(P5) = 1$

4) $M(P14) + M(P13) + M(P15) = 1$

5) $M(P7) + M(P8) = 1$

6) $M(P10) + M(P11) = 1$

7) $M(P9) + M(P11) + M(P8) = 2$

8) $M(P18) + M(P17) = 1$

9) $M(P4) + M(P3) + M(P17) + M(P2) = 3$

Tabla de estados y eventos del sistema

Tabla de estados de la red final

Plaza	Estado
P0	Buffer de entrada de imágenes al sistema
P1	Recursos compartidos en el sistema
P2	Imágenes listas para ser enviadas al contenedor para procesamiento
P3	Recursos compartidos en el sistema
P4	Imágenes listas para ser enviadas al contenedor para procesamiento
P5	Recursos compartidos en el sistema
P6	Representa el contenedor de imágenes a procesar
P7	Recursos compartidos en el sistema
P8	Imágenes listas para mejorar
P9	Recursos compartidos en el sistema
P10	Recursos compartidos en el sistema
P11	Imágenes listas para mejorar
P12	Contenedor de imágenes mejoradas en calidad
P13	Imágenes en recorte
P14	Recursos compartidos en el sistema
P15	Imágenes en recorte
P16	Imágenes en estado final
P17	Imágenes para exportar
P18	Recursos compartidos en el sistema

Tabla de eventos de la red final

Transición	Evento
T0	Input de imágenes al buffer de entrada
T1	Tomar imágenes del buffer
T2	Tomar imágenes del buffer
T3	Carga de imágenes en el contenedor para procesamiento
T4	Carga de imágenes en el contenedor para procesamiento
T5	Mejora calidad de imágenes
T6	Mejora calidad de imágenes
T7	Imagenes llevadas a contenedor de recorte
T8	Imagenes llevadas a contenedor de recorte
T9	Se realiza el recorte
T10	Se realiza el recorte
T11	Depositar imágenes en estado final
T12	Depositar imágenes en estado final
T13	Imágenes son exportadas fuera del sistema
T14	Imágenes son exportadas fuera del sistema

Determinación de la cantidad de hilos

Algoritmo para la determinación de hilos máximos activos simultáneos

1) Obtener los IT de la RdP:

1. {T1, T3, T6, T8, T10, T12, T13, T14}
2. {T1, T3, T6, T8, T9, T11, T13, T14}
3. {T1, T3, T5, T7, T10, T12, T13, T14}
4. {T1, T3, T5, T7, T9, T11, T13, T14}
5. {T2, T4, T6, T8, T10, T12, T13, T14}
6. {T2, T4, T6, T8, T9, T11, T13, T14}
7. {T2, T4, T5, T7, T10, T12, T13, T14}
8. {T2, T4, T5, T7, T9, T11, T13, T14}

2) Obtener el conjunto de plazas asociadas al IT:

1. {P0, P1, P2, P3, P6, P9, P11, P10, P12, P14, P15, P16, P17, P18}
2. {P0, P1, P2, P3, P6, P9, P11, P10, P12, P13, P14, P16, P17, P18}
3. {P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P16, P17, P18}
4. {P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P13, P16, P17, P18}
5. {P0, P3, P4, P5, P6, P9, P11, P10, P12, P14, P15, P16, P17, P18}
6. {P0, P3, P4, P5, P6, P9, P11, P10, P12, P13, P14, P16, P17, P18}
7. {P0, P3, P4, P5, P6, P7, P8, P9, P12, P13, P14, P16, P17, P18}
8. {P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P16, P17, P18}

3) Determinar las plazas relacionadas a acciones de cada IT:

1. P2, P11, P15, P17
2. P2, P11, P13, P17
3. P2, P8, P15, P17
4. P2, P8, P13, P17
5. P4, P11, P15, P17
6. P4, P11, P13, P17
7. P4, P8, P13, P17
8. P4, P8, P15, P17

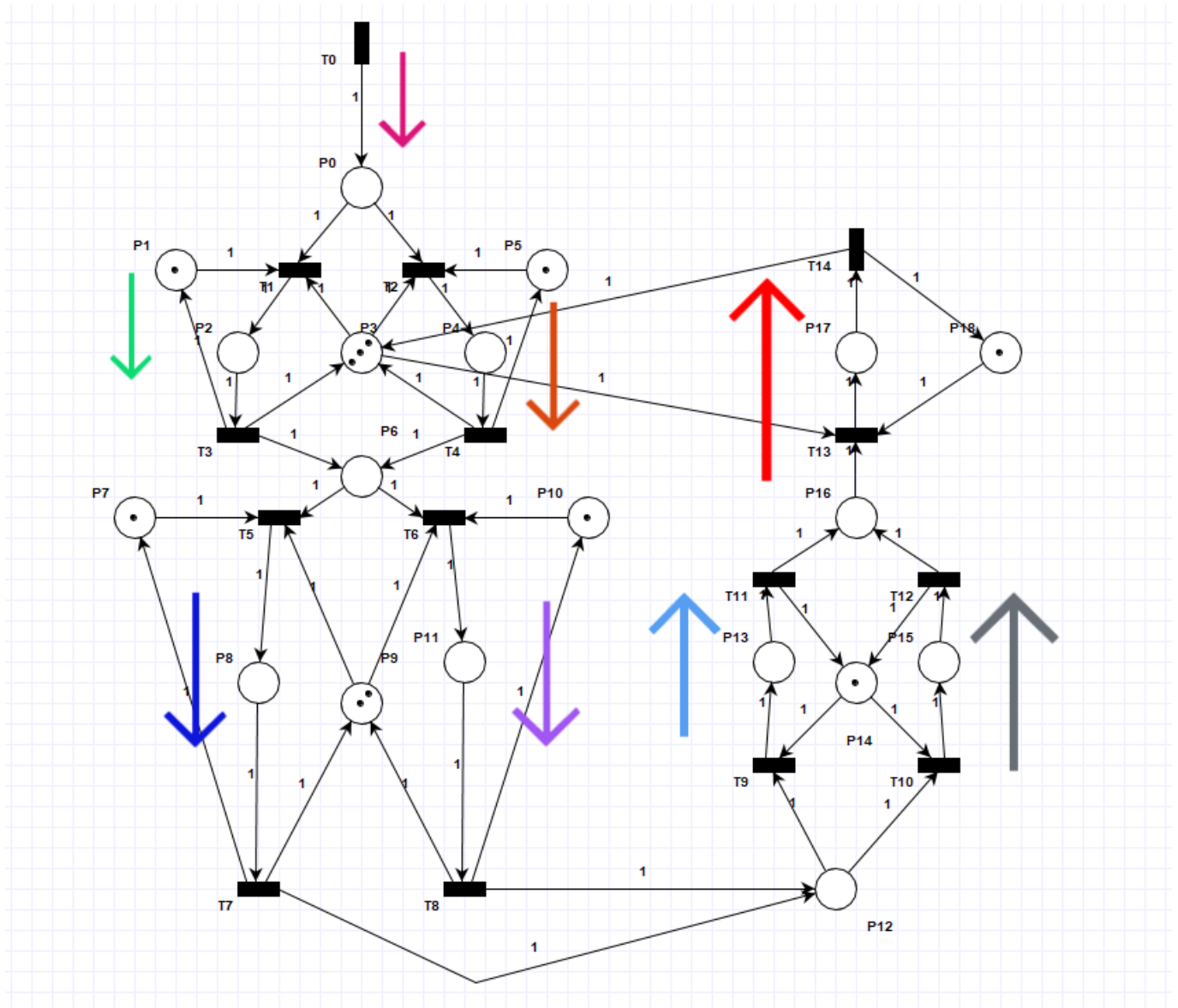
4) Obtenemos el conjunto de estados del conjunto de plazas PA

PA= {P2, P4, P8, P11, P13, P15, P17}

[TABLA CON POSIBLES MARCADOS](#)

Red de Petri final

Cantidad de hilos totales: 8



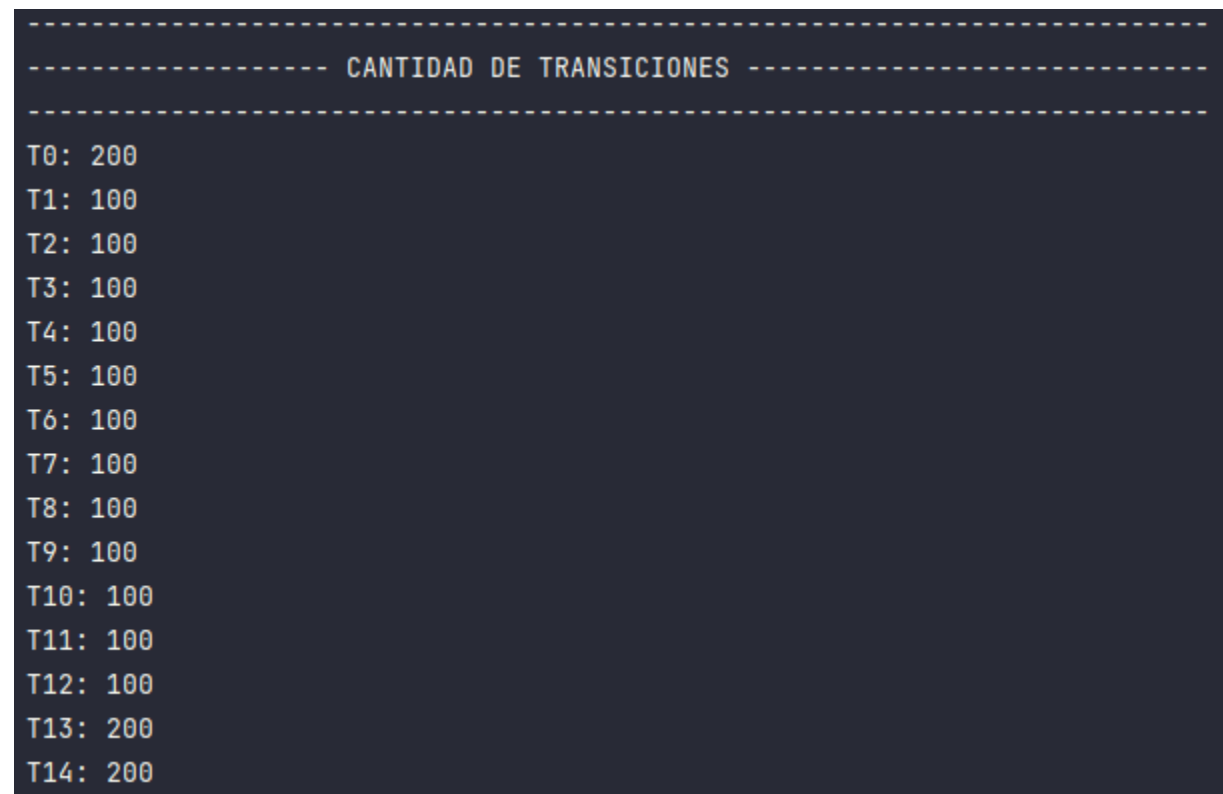
Aclaración: Si bien esta es nuestra red de petri final, con la que vamos a trabajar, cabe aclarar que en esta imagen no están contempladas las transiciones temporales, las cuales serían la T0, T3, T4, T7, T8, T11, T12, T14.

Política

Es necesario para el modelado del sistema implementar políticas que resuelvan los conflictos. Se requiere considerar dos casos (ejecutados y analizados por separado e independientes uno de otro):

1. Una política de procesamiento de imágenes balanceada. La cantidad de imágenes procesadas por cada segmento de la red al finalizar la ejecución, debe ser equitativa (izquierda vs derecha). Esto se debe corroborar al finalizar la ejecución mostrando la cantidad de imágenes procesadas por cada segmento. Para ello, se debe mostrar la cantidad de veces que se ejecutaron cada una de las transiciones pertenecientes a cada segmento.
2. Una política de procesamiento que priorice el segmento izquierdo en la etapa 3. Considere que el segmento izquierdo reciba el 80% de la carga {T9, T11}. Esto se debe corroborar de la misma manera que se indicó en el punto anterior.

Política 1



```
----- CANTIDAD DE TRANSICIONES -----  
T0: 200  
T1: 100  
T2: 100  
T3: 100  
T4: 100  
T5: 100  
T6: 100  
T7: 100  
T8: 100  
T9: 100  
T10: 100  
T11: 100  
T12: 100  
T13: 200  
T14: 200
```

----- CANTIDAD DE TRANSICIONES -----	
T0:	200
T1:	100
T2:	100
T3:	100
T4:	100
T5:	100
T6:	100
T7:	100
T8:	100
T9:	100
T10:	100
T11:	100
T12:	100
T13:	200
T14:	200

```

----- INVARIANTES -----
IT(1): [1, 3, 5, 7, 9, 11, 13, 14] = 25
IT(2): [2, 4, 6, 8, 9, 11, 13, 14] = 25
IT(3): [1, 3, 6, 8, 9, 11, 13, 14] = 25
IT(4): [2, 4, 5, 7, 9, 11, 13, 14] = 25
IT(5): [2, 4, 5, 7, 10, 12, 13, 14] = 25
IT(6): [1, 3, 6, 8, 10, 12, 13, 14] = 25
IT(7): [1, 3, 5, 7, 10, 12, 13, 14] = 25
IT(8): [2, 4, 6, 8, 10, 12, 13, 14] = 25

TOTAL IT'S: 200

```

Política 2

```

----- CANTIDAD DE TRANSICIONES -----
T0: 200
T1: 100
T2: 100
T3: 100
T4: 100
T5: 100
T6: 100
T7: 100
T8: 100
T9: 158
T10: 42
T11: 158
T12: 42
T13: 200
T14: 200

```

----- INVARIANTES -----	
IT(1):	[1, 3, 5, 7, 9, 11, 13, 14] = 41
IT(2):	[2, 4, 6, 8, 9, 11, 13, 14] = 41
IT(3):	[1, 3, 6, 8, 9, 11, 13, 14] = 41
IT(4):	[2, 4, 5, 7, 9, 11, 13, 14] = 41
IT(5):	[2, 4, 5, 7, 10, 12, 13, 14] = 9
IT(6):	[1, 3, 6, 8, 10, 12, 13, 14] = 9
IT(7):	[1, 3, 5, 7, 10, 12, 13, 14] = 9
IT(8):	[2, 4, 6, 8, 10, 12, 13, 14] = 9
TOTAL IT'S: 200	

Implementación en java

Imagen

En esta clase se realiza el procesamiento de las “imágenes” ya que esta clase implementa la interfaz Runnable. Tiene varios métodos y un constructor. La clase tiene una variable estática monitor de tipo Monitor, que es una instancia singleton de la clase Monitor. Hay una variable llamada “contadorT0” de tipo int, y otra variable llamada “transiciones” de tipo Integer array.

El constructor de la clase toma como parámetro un array de Enteros y lo asigna a la variable de instancia transiciones. La clase sobrescribe el método run de la interfaz Runnable. Dentro del método run hay tres sentencias condicionales. La primera sentencia condicional comprueba si la longitud de la matriz transiciones es 1 y si el valor del índice 0 es 0. Si esta condición es cierta, llama al método “procesarT0” ya que acá queremos hacer el procesamiento sólo de la transición T0. Si ninguna de las condiciones anteriores es cierta, llama al método procesar.

El método “procesar” contiene un bucle while que se ejecuta hasta que el método monitor.finalizar() devuelve true. Dentro del bucle while, hay un bucle for que itera sobre cada elemento de la matriz transiciones. Si el método "monitor.finalizar()" devuelve false, se llama al método “monitor.dispararTransicion” con el elemento actual como argumento disparando la transición hasta que "monitor.finalizar()" devuelva true.

El método `procesarT0` contiene un bucle `while` que se ejecuta hasta que la variable `contadorT0` es menor que 200 ya que esto nos asegura de que se disparan 200 T0 lo que nos deja la rdp en su estado inicial. Dentro del bucle `while`, hay un bucle `for` que itera sobre cada elemento del array `transiciones`. Llama al método `monitor.dispararTransicion` con el elemento actual como argumento e incrementa la variable `contadorT0`.

Tiempo

Para comenzar a trabajar con el tiempo en el código, debemos plantear diversos recursos que utilizaremos para determinar el disparo de una transición. Utilizamos la variable `alfa`, para referirnos al “espacio de tiempo mínimo que se emplea para comenzar la tarea que se está haciendo” y a `beta` como “tiempo máximo en el que se va a dejar de sensibilizar la transición” debido a esto, al elegir un `beta` exageradamente grande, nos aseguramos de que la transición nunca se des-sensibilice. Ahora, en esta parte del código define una clase llamada `Tiempo` que representa la funcionalidad relacionada con el tiempo. Tiene varias variables y métodos privados y públicos. El constructor de la clase toma como parámetro un array de Enteros llamado `transiciones`. Inicializa los arreglos “`alfa`”, “`beta`” y “`Esperando`” con un tamaño igual al número de transiciones. Establece la matriz sensibilizada en el parámetro `transiciones`. También establece los valores de la matriz “`Esperando en`” `false`. La clase tiene varios métodos:

esTemporizada: un método privado que comprueba si un disparo Entero “`t`” está temporizado iterando sobre el arreglo `TTemp` que es donde tenemos el registro que transiciones están temporizadas y devolviendo `true` si se encuentra, en caso contrario `false`.

estaSensibilizado: método público que comprueba si un Entero disparo dado está sensibilizado. Comprueba si el valor de “sensibilizada” en el índice disparo es mayor o igual que 1 y si el elemento correspondiente en el array “`Esperando`” es falso. Si no está temporizado, devuelve `true`. Si está temporizado, llama al método `testVentanaTiempo` para comprobar además si está dentro de la ventana de tiempo.

testVentanaTiempo: método privado que comprueba si un entero disparo está dentro de la ventana de tiempo. Obtiene la marca de tiempo actual y comprueba si el tiempo transcurrido desde la última marca de tiempo es mayor o igual que `alfa[disparo]` y menor que `beta[disparo]`. Si está dentro de la ventana de tiempo, devuelve verdadero. En caso de que este antes de `alfa`, llama al método `antesDeLaVentana` para tratar el caso en que sea anterior a la ventana temporal. Además “`timeStamp`” se utiliza para realizar un seguimiento del tiempo en que se cambió el estado de cada transición sensibilizada.

setTiempos: método privado que establece los valores de los arreglos `alfa` y `beta` en función de si una transición está temporizada o no. Si una transición está temporizada, establece `alfa` en un valor que seleccionemos y `beta` en un valor grande.

antesDeLaVentana: método privado que gestiona el caso de que una transición se produzca antes de la ventana temporal. Pone el elemento correspondiente del array Esperando a true, imprime un mensaje, libera un mutex, calcula el tiempo a dormir, duerme durante ese tiempo, adquiere de nuevo el mutex, y pone el elemento correspondiente del array Esperando a false.

Política

Define una clase llamada Política. La clase tiene una variable final estática privada llamada rdp de tipo RdP. También tiene un constructor por defecto. La clase tiene dos métodos privados: Politica_1 y Politica_2.

El método **Politica_1** toma un array de Enteros como entrada y devuelve un Entero. Inicializa una variable minDisparos con el valor máximo posible de un Entero. También inicializa una variable “selecTransition” con 0. A continuación itera sobre la matriz de entrada y comprueba si el valor en cada índice es igual a 1 y si el valor correspondiente en la matriz Disparos del objeto rdp es menor que minDisparos. Si ambas condiciones son ciertas, actualiza los valores de minDisparos y selecTransition en consecuencia. Por último, devuelve el valor de selecTransition.

El método **Politica_2** toma una matriz de números enteros como entrada y devuelve un número entero. Crea un nuevo objeto Random. Genera un número aleatorio entre 0 y 99 (ambos inclusive) y comprueba si es menor que 80. Si lo es, comprueba si el número es mayor que 80. En caso afirmativo, comprueba si el valor del índice 9 de la matriz de entrada es mayor o igual que 1. En caso afirmativo, devuelve 9. En caso contrario, comprueba si el valor del índice 11 de la matriz de entrada es mayor o igual que 1. En caso afirmativo, devuelve 11. Si no se cumple ninguna de estas condiciones, llama al método Politica_1 con la matriz de entrada y devuelve su resultado. **Para el correcto funcionamiento de esta política, se tuvo que agregar un hilo al segmento de las transiciones T9 y T11, esto es por que cuando se hace el “AND” de que si la transición está sensibilizada y si tiene hilos esperando, al agregarle un hilo, la mayoría de la veces va a haber 1 hilo esperando esa transición, pero en caso contrario cuando no se le agregó un hilo las transiciones se mantenian equilibradas.**

La clase también tiene un método público llamado aplicarPolitica que toma un array de Enteros y un entero política como entrada y devuelve un Entero. Comprueba el valor de política y si es 1, llama al método Politica_1 con el array de entrada y devuelve su resultado. Si es 2, llama al método Politica_2 con el array de entrada y devuelve su resultado. Si el valor de política no es ni 1 ni 2, lanza una IllegalArgumentException con el mensaje "POLÍTICA NO VÁLIDA".

RdP

Representa un modelo de Red de Petri y proporciona métodos para interactuar con él. La clase tiene varias variables y métodos privados y públicos.

1. Variables: **MtzIncidencia**: Una matriz de Enteros que representa la matriz de incidencia de la Red de Petri. **Marcado**: Un array de Enteros representando el marcado (número de tokens) en cada lugar de la Red de Petri. **TsensA**: Una instancia de la clase Tiempo, que representa la sensibilidad actual de cada transición en la Red de Petri. **Disparos**: Un array de enteros que representa el número de veces que se ha disparado cada transición. **timeStamp**: Un array de Longs que representa la marca de tiempo de cada transición en la Red de Petri.

2. Constructor: Inicializa las variables MtzIncidencia y Marcado con valores predefinidos. Crea un nuevo array Disparos con el tamaño de CANTIDAD_TRANSICIONES e inicializa todos los elementos a 0. Crea un nuevo array timeStamp con el tamaño de CANTIDAD_TRANSICIONES y establece cada elemento a la hora actual del sistema.

3. Métodos: **TransicionSens()**: Calcula la sensibilidad temporal de cada transición en función del marcado actual y la matriz de incidencia. Devuelve un array de Enteros representando la sensibilidad temporal de cada transición. **Disparar(Integer disparo)**: Dispara la transición especificada si está sensibilizada. Actualiza el marcado, la sensibilidad temporal, el número de disparos y registra la transición. Devuelve true si la transición se disparó con éxito, en caso contrario false. **ActualizarMarcado(Integer disparo)**: Actualiza el marcado de la Red de Petri después de disparar una transición. Calcula el nuevo marcado sumando el producto de la matriz de incidencia y el vector de disparo al marcado actual, osea utilizando la ecuación fundamental de la rdp. También comprueba si el marcado actualizado satisface los invariantes de plaza y lanza una excepción en caso contrario. **ActualizarT()**: Actualiza la sensibilidad temporal de cada transición basándose en el marcado actual. **cumpleIP()**: Comprueba si el marcado actual satisface los invariantes de lugar. Devuelve verdadero si se satisfacen todas las invariantes, falso en caso contrario. **Fin()**: Comprueba si el disparo de la transición 14 ha alcanzado un determinado umbral (200). Devuelve verdadero si se alcanza el umbral, falso en caso contrario. **ActualizarDisparos(Integer disparo)**: Actualiza el número de veces que se ha disparado una transición. **getTimeStamp()**: Devuelve la matriz de marcas de tiempo de cada transición. **setTimeStamp(Integer[] nuevaT)**: Actualiza la marca de tiempo de cada transición si su sensibilidad temporal ha cambiado. Este código representa un modelo de red de Petri y proporciona métodos para disparar transiciones, actualizar el marcado, comprobar invariantes, y realizar un seguimiento del número de disparos y marcas de tiempo de las transiciones.

Monitor

Esta clase define un Monitor con múltiples mecanismos de sincronización para controlar el acceso a un recurso compartido. Forma parte de un sistema más grande que involucra una Red de Petri. Vamos a analizar los principales componentes y funcionalidades de la clase Monitor:

Patrón Singleton: La clase utiliza el patrón de diseño Singleton para asegurar que solo se cree una instancia de la clase Monitor. Esto se logra a través del método `InstanceMonitor`, que crea una nueva instancia o devuelve la existente.

Semáforos: La clase utiliza semáforos para la sincronización, en particular `Mutex` y `ColaCondition`. Los semáforos se utilizan para coordinar y controlar el acceso al recurso compartido. `Mutex` es un semáforo binario utilizado para la exclusión mutua (garantizando que solo un hilo acceda a la sección crítica a la vez) es decir al monitor, y `ColaCondition` es un array de semáforos que representan colas de condición asociadas con diferentes transiciones, donde duermen los hilos que no pudieron disparar alguna transición tanto por falta de tokens o por no cumplir las condiciones necesarias del tiempo requerido de alfa y beta, estos hilos se quedarán en `ColaCondition` hasta ser despertados para intentar nuevamente disparar la transición.

`tomarMutex()`: Este método es utilizado por los hilos para adquirir el semáforo `mutex`. Asegura que solo un hilo pueda entrar en la sección crítica a la vez.

`liberarMutex()`: Libera el semáforo `mutex` y potencialmente despierta a un hilo en espera en `ColaCondition`.

`dispararTransicion(Integer t)`: Este método se utiliza para disparar una transición en la Red de Petri. Primero adquiere el `mutex`, luego llama al método `disparar` para activar la transición y potencialmente cambia el estado de la Red de Petri.

`disparar(Integer transición)`: Este método intenta disparar una transición. Si la transición no puede dispararse y la Red de Petri no está en un estado final, libera el `mutex` y coloca al hilo actual en la cola de condición de la transición. Luego intenta nuevamente disparar la transición de manera recursiva.

`transiciones()`: Este método verifica el estado de sensibilización de cada transición. Devuelve un array que indica qué transiciones están sensibilizadas y cuáles no, basado en el estado de la Red de Petri y las colas de condición. Particularmente devuelve las transiciones que están sensibilizadas y que tienen un hilo esperando.

`LiberarCola()`: Verifica las transiciones sensibilizadas, aplica una política para elegir una transición y, si hay hilos en espera en la cola de condición para esa transición, libera uno de ellos.

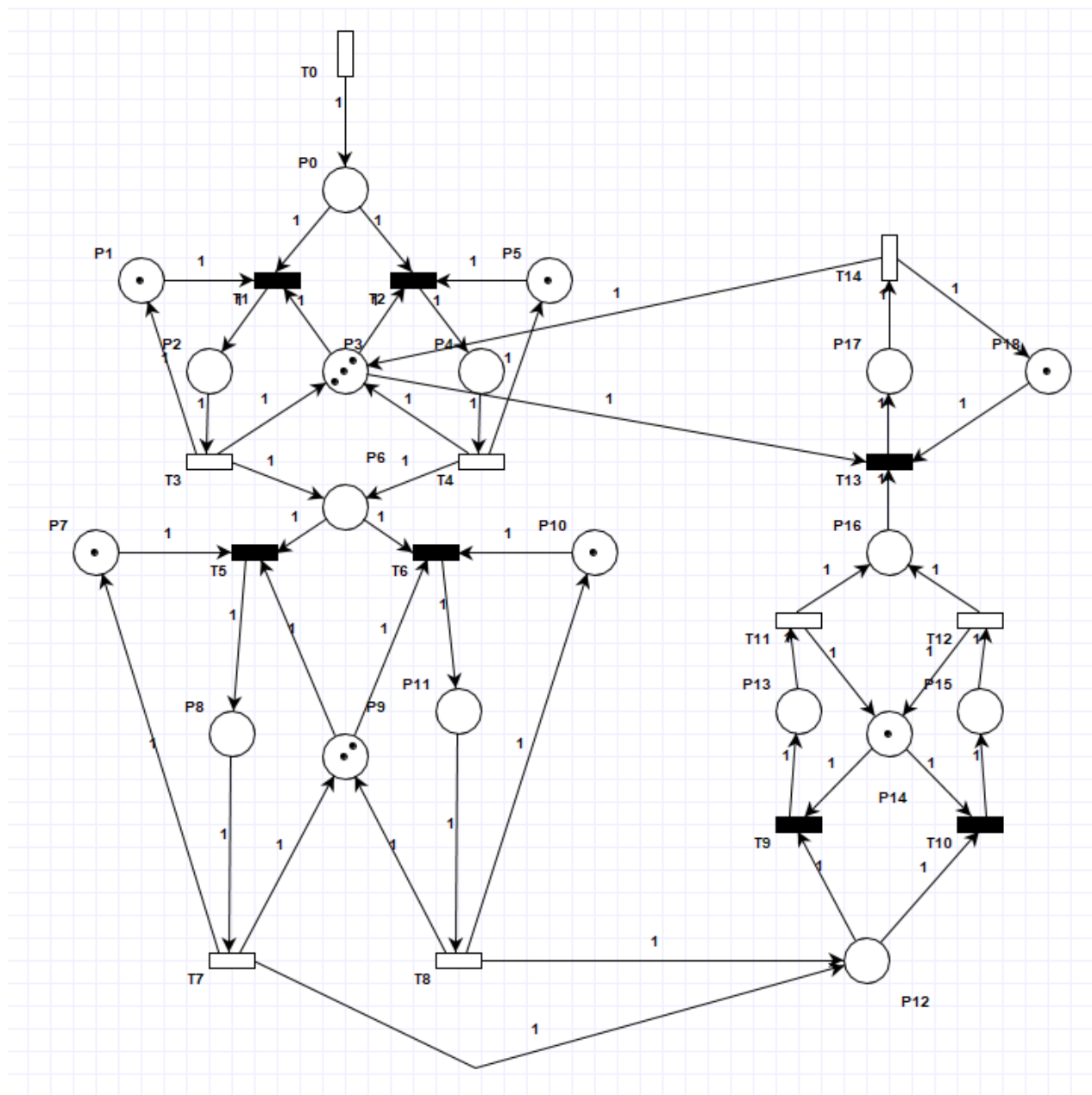
`finalizar()`: Verifica si la Red de Petri ha alcanzado un estado final. Si es así, cierra cualquier recurso abierto y libera cualquier hilo en espera de las colas de condición.

`selecPolitica()`: Acepta la entrada del usuario para seleccionar una política y la configura para el monitor.

DIAGRAMA DE CLASES

Semántica temporal

Red de petri con transiciones temporales



Transiciones Temporales: T0 - T3 - T4 - T7 - T8 - T11 - T12 - T14

ANÁLISIS TEMPORAL TESTEADO CON PROCESADOR HEXA-CORE : [Análisis Temporal](#)

ANÁLISIS TEMPORAL TESTEADO CON PROCESADOR DUAL-CORE: [Análisis Temporal 2](#)

Tiempo considerando una secuencialización del programa

Si el sistema fuera secuencializado solo un hilo ejecutaría todo, entonces el token o sea la “imagen” solo podría escoger un camino de todos los posibles.

Para la política 1

El tiempo que nos daría esto considerando que alfa es de 10ms sería:

$$T = (T_0 + T_1 + T_3 + T_5 + T_7 + T_9 + T_{11} + T_{13} + T_{14}) * (n^{\circ} \text{ Imagenes})$$

$$T = (10 + 0 + 10 + 0 + 10 + 0 + 10 + 0 + 10) * 200 = 10s$$

El tiempo que nos daría esto considerando que alfa es de 50ms sería:

$$T = (T_0 + T_1 + T_3 + T_5 + T_7 + T_9 + T_{11} + T_{13} + T_{14}) * (n^{\circ} \text{ Imagenes})$$

$$T = (50 + 0 + 50 + 0 + 50 + 0 + 50 + 0 + 50) * 200 = 50s$$

El tiempo que nos daría esto considerando que alfa es de 100ms sería:

$$T = (T_0 + T_1 + T_3 + T_5 + T_7 + T_9 + T_{11} + T_{13} + T_{14}) * (n^{\circ} \text{ Imagenes})$$

$$T = (100 + 0 + 100 + 0 + 100 + 0 + 100 + 0 + 100) * 200 = 100s$$

Para la política 2

Para que la política 2 funcione correctamente se tuvo que implementar un tiempo de sleep igual o mayor que alfa en la T11.

El tiempo que nos daría esto considerando que alfa es de 10ms sería:

$$T = (T_0 + T_1 + T_3 + T_5 + T_7 + T_9 + T_{11} + T_{13} + T_{14}) * (n^{\circ} \text{ Imagenes})$$

$$T = (10 + 0 + 10 + 0 + 10 + (10 + 20) + 0 + 10) * 200 = 14s$$

El tiempo que nos daría esto considerando que alfa es de 50ms sería:

$$T = (T_0 + T_1 + T_3 + T_5 + T_7 + T_9 + T_{11} + T_{13} + T_{14}) * (n^{\circ} \text{ Imagenes})$$

$$T = (50 + 0 + 50 + 0 + 50 + (50 + 100) + 0 + 50) * 200 = 70s$$

El tiempo que nos daría esto considerando que alfa es de 100ms sería:

$$T = (T0 + T1 + T3 + T5 + T7 + T9 + T11 + T13 + T14) * (n^{\circ} \text{ Imagenes})$$

$$T = (100 + 0 + 100 + 0 + 100 + (100 + 200) + 0 + 100) * 200 = 140s$$

Tiempo considerando una absoluta concurrencia

Para este caso tuvimos en cuenta varios métodos para calcular este tiempo teórico

$$T = T14 * (n^{\circ} \text{ Imagenes})$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas})$$

Para política 1

El cálculo del tiempo considerando que alfa es de 10ms lo obtenemos de la siguiente manera:

$$T = T14 * (n^{\circ} \text{ Imagenes}) = 10 * 200 = 2s$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas}) = (200 * 50) / 7 = 1,42s$$

El cálculo del tiempo considerando que alfa es de 50ms lo obtenemos de la siguiente manera:

$$T = T14 * (n^{\circ} \text{ Imágenes}) = 50 * 200 = 10s$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas}) = (200 * 250) / 7 = 7,42s$$

El cálculo del tiempo considerando que alfa es de 100ms lo obtenemos de la siguiente manera:

$$T = T14 * (n^{\circ} \text{ Imágenes}) = 100 * 200 = 20s$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas}) = (200 * 500) / 7 = 14,42s$$

Para política 2

El cálculo del tiempo considerando que alfa es de 10ms lo obtenemos de la siguiente manera:

$$T = T_{14} * (n^{\circ} \text{ Imágenes}) = 10 * 200 = 2s$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas}) = (200 * 70) / 7 = 2s$$

El cálculo del tiempo considerando que alfa es de 50ms lo obtenemos de la siguiente manera:

$$T = T_{14} * (n^{\circ} \text{ Imágenes}) = 50 * 200 = 10s$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas}) = (200 * 350) / 7 = 12s$$

El cálculo del tiempo considerando que alfa es de 100ms lo obtenemos de la siguiente manera:

$$T = T_{14} * (n^{\circ} \text{ Imágenes}) = 100 * 200 = 20s$$

$$T = ((n^{\circ} \text{ Imágenes}) * (\text{Tiempo que tarda 1 imagen})) / (n^{\circ} \text{ Imágenes simultáneas}) = (200 * 700) / 7 = 20s$$

Con los resultados obtenidos del análisis de los tiempos y contemplando que el proyecto fue ejecutado con procesadores diversos (tanto hexa-core como dual-core), verificamos que para los diversos alfa que planteamos, las ejecuciones se encuentran dentro del tiempo límite planteado, tanto con respecto al peor y al mejor caso, nos encontramos dentro de los márgenes óptimos de funcionamiento, comparando los tiempos obtenidos de forma práctica con los tiempos teóricos propuestos, tanto para en el caso de concurrencia absoluta (mejor de los casos) como para secuencialidad absoluta (peor de los casos).

Esto nos deja en claro que nuestro programa está funcionando correctamente y de una manera óptima.

POLÍTICA 1			
ALFA [ms]	TIEMPO T [s] PROMEDIO Nuestros datos	TIEMPO T [s] Mejor caso	TIEMPO T [s] Peor caso
10	3,194	1,42	10
50	12,689	7,42	50
100	22,255	14,42	200
POLÍTICA 2			
ALFA [ms]	TIEMPO T [s] PROMEDIO Nuestros datos	TIEMPO T [s] Mejor caso	TIEMPO T [s] Peor caso
10	3,218	2	14
50	12,823	12	70
100	22,439	20	140

Como podemos observar en los cuadros, los tiempos obtenidos de la ejecución utilizando un procesador hexa-core, se corresponden con un “correcto y óptimo” funcionamiento, viéndose claramente la tendencia hacia el tiempo que se obtendría en el mejor de los casos, teniendo en cuenta la variación correspondiente de los alfa.

Expresiones regulares

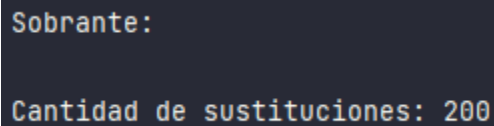
[REGEX](#)

Lo que se buscó con la regex implementada es que se puedan encontrar todos los invariantes de transición en una misma expresión regular, es decir debíamos poder verificar que todas las secuencias por las que pasaban las imágenes en su proceso eran un invariante de transición. Con esto lo que hacemos es que verificamos una vez más el correcto funcionamiento de nuestro programa.

En primeras instancias tuvimos problemas por cómo separamos las transiciones en el logger, ya sea con un espacio, o sin tener algún tipo de carácter. Ya que esto cambiaba nuestra expresión regular, pasamos por varias expresiones y después de varias pruebas llegamos a la conclusión de que la mejor manera o la que nos funcionaba de una manera acorde, era separar las transiciones por espacios.

Con esto surgió un último problema, que era que cuando agregamos una transición que no pertenecía a un IT, el script nos daba un sobrante de esa transición que agregamos más un invariante de transición completo, esto se solucionó contemplando los espacios en la última transición, es decir T14.

Salida de la ejecución del script en python:



```
Sobrante:  
Cantidad de sustituciones: 200
```

Como se puede verificar en esta salida, nuestro script realiza 200 sustituciones con el método **re.subn**, lo que nos muestra que encontró 200 invariantes de transición como estaba pactado durante la ejecución del programa. **re.subn** lo utilizamos para especificar cómo vamos a formar el resultado después de hacer la sustitución en la regex original, para esto utilizamos las expresiones `g<x>` para representar los grupos de captura de la regex, por ejemplo con el grupo de captura `g<3>` nos referimos a la captura de T1 y T3 o T2 y T4.

Se puede ver en el script los siguientes métodos:

Contador de transiciones: este método está para corroborar que la cantidad de transiciones que se ve en la ejecución del main, sea la misma cantidad leída del txt. Asegurándonos de que no se pierda nada en la escritura del logger.

Después podemos ver la estructura principal del código, en esta sección está la corroboración de los sobrantes. La ejecución no frenará hasta que lo que quede en “resultado” sea igual a lo que se encuentra en “nuevo_resultado”. Para que esto suceda, a través de los métodos de expresión regular, utilizamos la función **re.subn**, a la cual le pasamos la expresión regular, las transiciones, el grupo por el cual se reemplazarán, y un contador.

Por último se ve en pantalla lo obtenido, llegando a los resultados correctos, con ningún sobrante y la cantidad de sustituciones.

Diagramas

[Diagrama de secuencia](#)

[Diagrama de clases](#)

Conclusión

Como conclusión podemos destacar la eficiencia de utilizar una red de petri en problemáticas complejas referidas a la concurrencia, más allá de la complejidad de la misma, el gran aporte que realiza el poder trabajar con transiciones y plazas para controlar el desarrollo y correcto funcionamiento del problema, nos facilitó estructurar algunas aplicaciones complejas de la concurrencia, nos permitió trabajar con una gran cantidad de threads al mismo tiempo si presentar ninguna falla típica vista en problemas de sincronización, como errores no determinísticos, starvation y en el peor de los casos deadlock. Además de contar con la red de petri, la implementación de un monitor de concurrencia encargado de secuencializar, gestionar recursos compartidos, gestionar la asignación óptima de recursos y asegurar que los hilos no mueran por diversas causas, fue una gran ayuda a la hora de manejar la concurrencia del programa y con la ayuda de los semáforos binarios. Esto nos permitió llevar adelante el problema planteado y asegurar su funcionamiento óptimo. Si bien no fue sencillo aplicar todos los conceptos vistos en clases a este trabajo final, fue un gran avance en nuestra experiencia con el desarrollo de problemáticas de concurrencia.

Algunos puntos a tener en cuenta a la hora de hablar de la importancia de trabajar con una red de petri y un monitor de concurrencia frente a problemáticas complejas de programación concurrente podemos destacar los siguientes puntos que nos fueron de utilidad:

- **Modelado**: La Red de Petri nos permitió modelar de manera precisa el comportamiento concurrente de un sistema, pudiendo representar claramente los estados y las transiciones del sistema, lo que nos facilitó la comprensión de su funcionamiento y la identificación de problemas potenciales, esto lo hicimos con la diversas herramientas de modelado de redes de petri como pipe o petrinator.
- **Visualización** : La Red de Petri nos proporcionó una representación visual clara de la lógica concurrente del problema abordado en el trabajo práctico.
- **Rendimiento y escalabilidad**: Al poder diseñar un sistema concurrente con precisión y control, pudimos optimizar su rendimiento y escalabilidad, como bien pudimos observar en el desarrollo de la semántica temporal y nuestra aproximación a los resultados de tiempo ideales calculados. Además, la escalabilidad y precisión, puede ser útil en aplicaciones modernas que deben aprovechar al máximo los recursos de hardware multi core, como pudimos observar con el desarrollo de los tiempos con los distintos procesadores del equipo (hexa-core/ quad-core).