Universidad
Nacional
de Córdoba

Universidad Nacional de Córdoba
Facultad de Ciencias Exactas, Físicas y Naturales

Programación Concurrente
Trabajo Final Integrador

Preparado por:
Borgatello, Ignacio

Profesores:
Ventre, Luis
Ludemann, Mauricio

Índice

1. Propiedades

En la Figura 1 se observa una red de Petri que modela un sistema doble de procesamiento de imágenes. Las plazas {P1, P3, P5, P7, P9, P11, P15} representan recursos compartidos en el sistema. La plaza {P0} es una plaza idle que corresponde al buffer de entrada de imágenes al sistema. En las plazas {P2, P4} se realiza la carga de imágenes en el contenedor para procesamiento. La plaza {P6} representa el contenedor de imágenes a procesar. Las plazas {P8, P10, P11, P13} representan los estados en los cuales se realiza un ajuste de la calidad de las imágenes. Este ajuste debe hacerse en dos etapas secuenciales. Con la plaza {P14} se modela el contenedor de imágenes mejoradas en calidad, listas para ser recortadas a su tamaño definitivo. En las plazas {P16, P17} se realiza el recorte. En la plaza {P18} se depositan las imágenes en estado final. La plaza {P19} representa el proceso por el cual las imágenes son exportadas fuera del sistema.

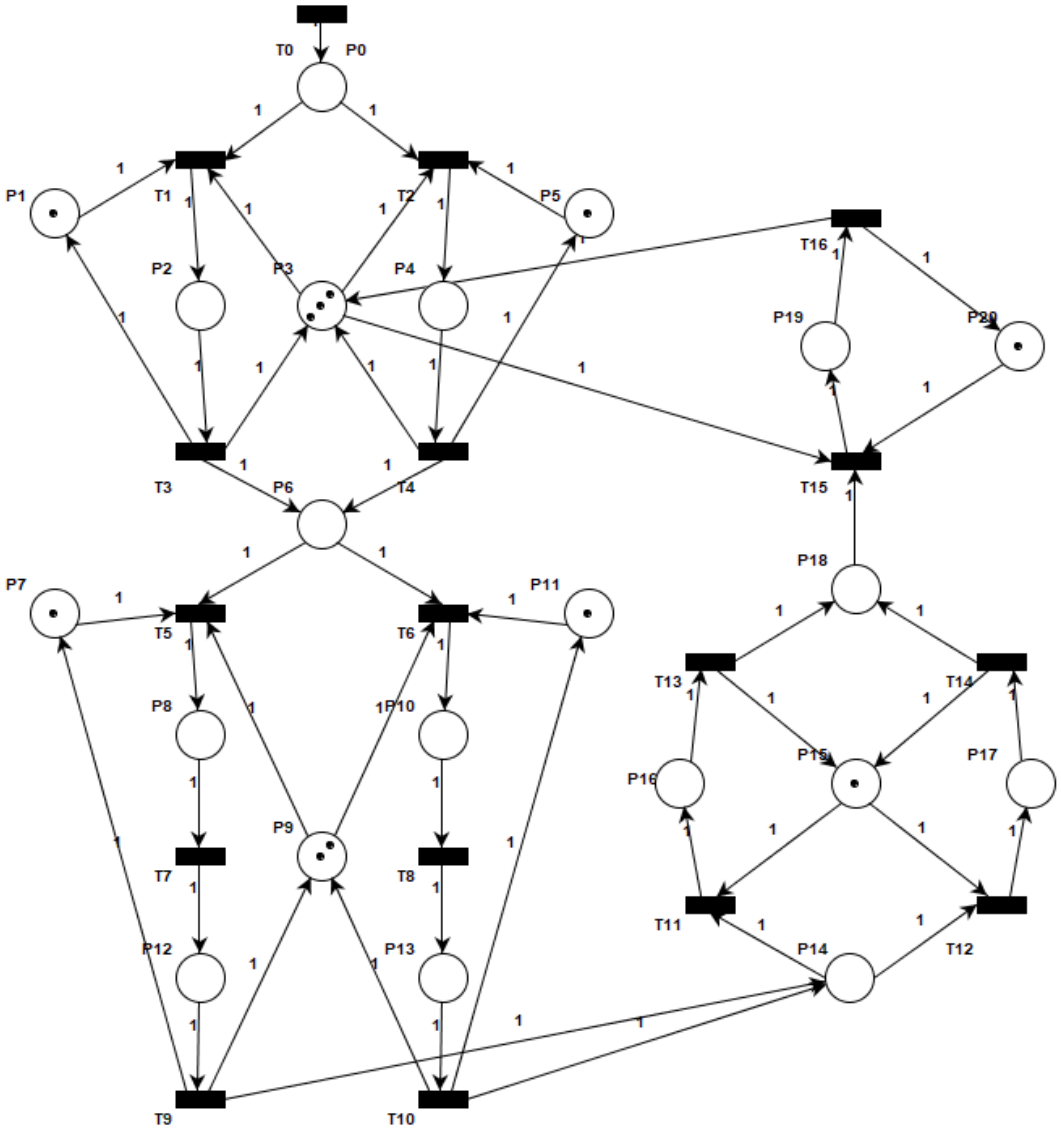


Figura 1. Red de Petri que modela el sistema de imágenes.

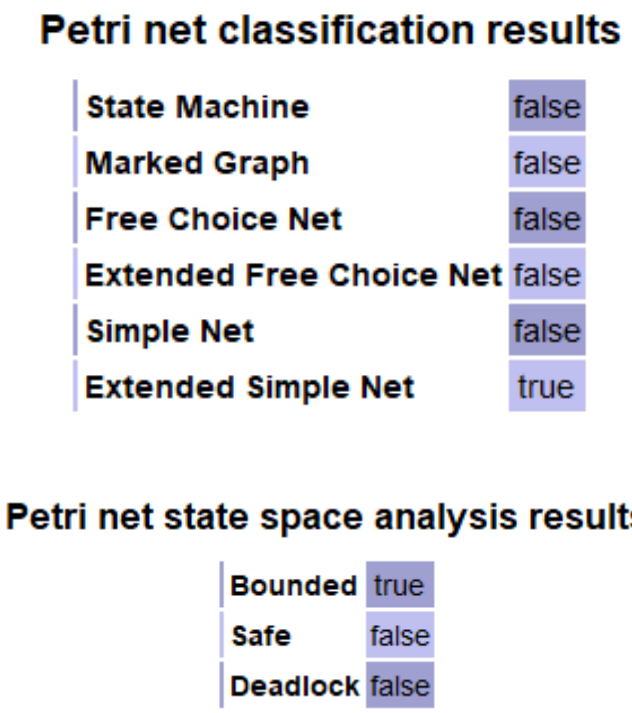


Figura 2. Clasificación de la Red de Petri con PIPE.

Nota 1: Para realizar el análisis de propiedades estructurales en la herramienta, fue necesario unir la transición {T16} con la plaza {P0} y eliminar la transición {T0}.

Nota 2: Para llevar a cabo el análisis de invariantes en la herramienta, se marcaron todas las transiciones como inmediatas (no temporizadas).

1.1 Deadlock, vivacidad, seguridad

Se observa que existen transiciones que tienen más de un lugar de entrada, lo que implica que **la red no puede representar una máquina de estados**. **La red no es un grafo de marcado** porque por ejemplo, la plaza P0 cuenta con dos transiciones de salida, la plaza P18 tiene dos transiciones de entrada, y las plazas P6, P9 y P15 presentan dos transiciones de entrada y dos de salida.

También se observa que las plazas P0, P3, P6, P9, P14 y P18 pueden contener un marcado mayor a 1, lo que significa que **la red no es segura**.

La red de Petri **no presenta deadlock**, lo que es una propiedad crucial en sistemas concurrentes. Un deadlock es una situación en la que ninguna transición del sistema puede ser disparada, provocando que el sistema se quede atascado en un estado inactivo. Las propiedades de vivacidad y libre de deadlock no son independientes. Como la red de petri está libre de interbloqueo, entonces está viva.

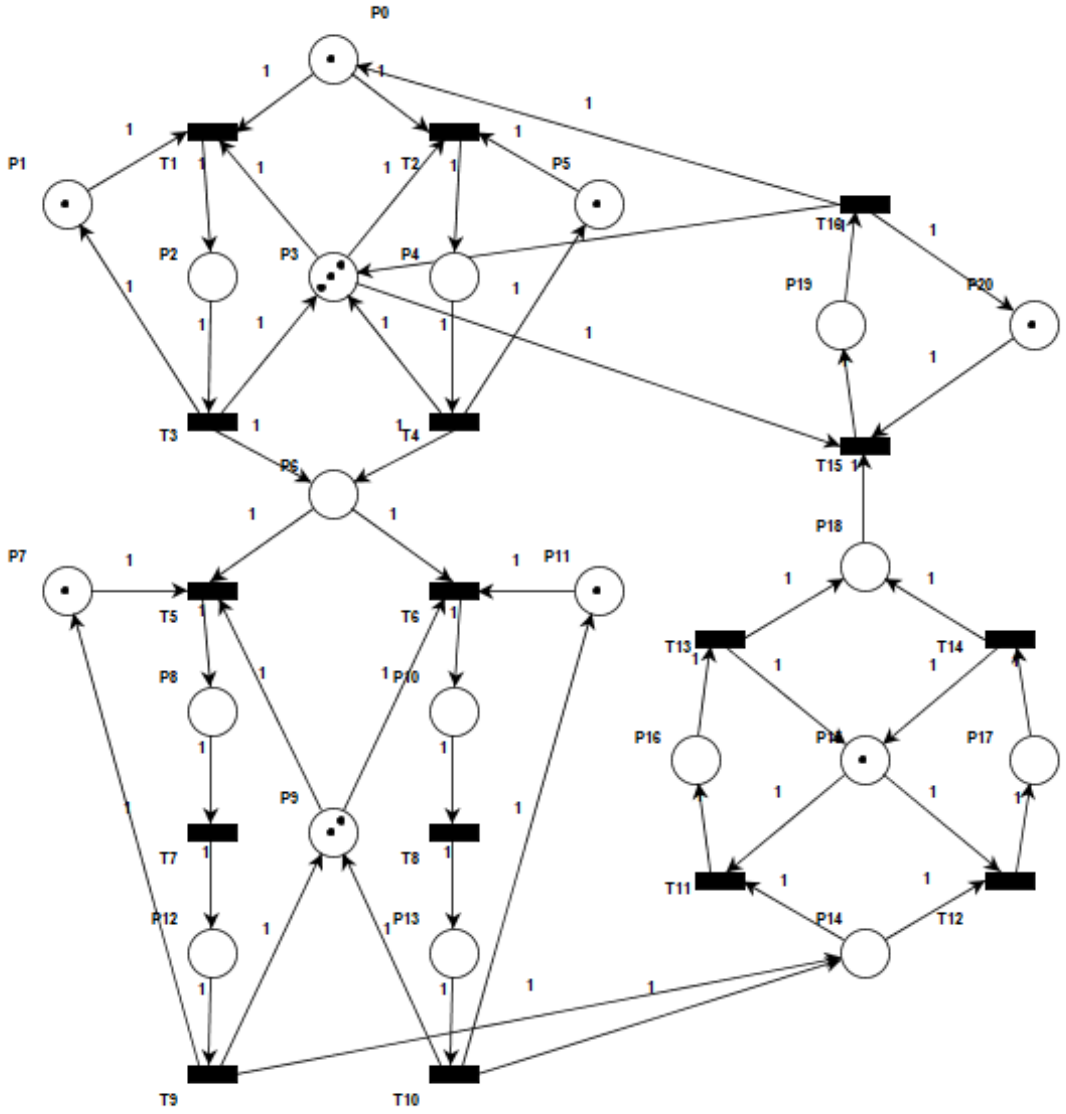


Figura 3. Red de Petri modificada para cálculo de invariantes.

2. Invariantes

Para definir los invariantes de plaza y de transición de la red, se utilizó el software TINA. A partir de la red de Petri modificada, se determinaron los siguientes invariantes de plaza:

P-Invariante	Invariante	Comentarios
1	P1 + P2 = 1	Tienen que ver con el proceso de carga. En este caso, se considera que el cargador está o no está cargando una imagen al buffer.
2	P4 + P5 = 1	

3	P2 + P3 + P4 + P19 = 3	Este invariante lo define la plaza P3. Puede considerarse como que dos imágenes pueden cargarse mientras otra está siendo exportada al mismo tiempo.
4 5	P7 + P8 + P12 = 1 P10 + P11 + P13 = 1	Invariante de la etapa de Mejorado. Indica que tan solo una imagen por segmento puede estar siendo mejorada. El mejorador está o no está mejorando la imagen.
6	P8 + P9 + P10 + P12 + P13 = 2	Implica que dos imágenes pueden estar siendo mejoradas a la vez.
7	P15 + P16 + P17 = 1	Indica la exclusión mutua que existe durante la etapa de cortado. En este caso debe hacerse de una imagen.
8	P19 + P20 = 1	Al igual que sucede con el cargador, el exportador puede exportar una imagen y puede o no estar haciéndolo.
9	P0 + P10 + P12 + P13 + P14 + P16 + P17 + P18 + P19 + P2 + P4 + P6 + P8 = n donde n es m0(P0)	Cantidad de imágenes en el sistema. Depende exclusivamente del marcado inicial de P0.

Los invariantes de transición determinados a partir de la red son:

T-Invariante	Invariante
1	T1, T3, T5, T7, T9, T11, T13, T15, T16
2	T1, T3, T5, T7, T9, T12, T14, T15, T16
3	T1, T3, T6, T8, T10, T11, T13, T15, T16
4	T1, T3, T6, T8, T10, T12, T14, T15, T16
5	T2, T4, T5, T7, T9, T11, T13, T15, T16
6	T2, T4, T5, T7, T9, T12, T14, T15, T16
7	T2, T4, T6, T8, T10, T11, T13, T15, T16
8	T2, T4, T6, T8, T10, T12, T14, T15, T16

Para modelar la evolución de la red y desarrollar el modelo del sistema en el lenguaje de programación JAVA, se obtuvo la matriz de incidencia de la red y, a partir de ella, la ecuación de estado. La matriz de incidencia fue determinada utilizando la herramienta PIPE.

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16
P0	1	-1	-1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
P1	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
P2	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0	0
P3	0	-1	-1	1	1	0	0	0	0	0	0	0	0	0	0	-1	1
P4	0	0	1	0	-1	0	0	0	0	0	0	0	0	0	0	0	0
P5	0	0	-1	0	1	0	0	0	0	0	0	0	0	0	0	0	0
P6	0	0	0	1	1	-1	-1	0	0	0	0	0	0	0	0	0	0
P7	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0	0
P8	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	0	0
P9	0	0	0	0	0	-1	-1	0	0	1	1	0	0	0	0	0	0
P10	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0	0
P11	0	0	0	0	0	0	-1	0	0	0	1	0	0	0	0	0	0
P12	0	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0	0
P13	0	0	0	0	0	0	0	0	1	0	-1	0	0	0	0	0	0
P14	0	0	0	0	0	0	0	0	0	1	1	-1	-1	0	0	0	0
P15	0	0	0	0	0	0	0	0	0	0	0	-1	-1	1	1	0	0
P16	0	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	0	0
P17	0	0	0	0	0	0	0	0	0	0	0	0	1	0	-1	0	0
P18	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	-1	0
P19	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-1
P20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-1	1

3. Cálculo de hilos y segmentos

Utilizando el algoritmo de Ref /1/, se calculó la cantidad máxima de hilos activos simultáneos para el sistema modelado por la red.

Se calcularon los invariantes de transición y se identificaron los conjuntos de plazas asociadas a cada invariante de transición.

T-Invariante	Plazas asociadas
1	P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20
2	P0, P1, P2, P3, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20
3	P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20
4	P0, P1, P2, P3, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20
5	P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P16, P18, P19, P20
6	P0, P3, P4, P5, P6, P7, P8, P9, P12, P14, P15, P17, P18, P19, P20
7	P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P16, P18, P19, P20
8	P0, P3, P4, P5, P6, P9, P10, P11, P13, P14, P15, P17, P18, P19, P20

Luego, en base a la descripción de las plazas presentada anteriormente, se determinaron los siguientes conjuntos de plazas que no son plazas de acción:

Plazas Idle	P0, P6, P14, P18
Plazas de restricción	P15
Plazas de recursos	P1, P3, P5, P7, P9, P11, P20

Al sustraer los conjuntos de plazas idle, de recursos y de restricción de las plazas asociadas a cada invariante de transición, se obtuvieron los conjuntos de plazas de acción correspondientes a cada invariante:

T-Invariante	Plazas
1	P2, P8, P12, P16, P19
2	P2, P8, P12, P17, P19
3	P2, P10, P13, P16, P19
4	P2, P10, P13, P17, P19
5	P4, P8, P12, P16, P19}
6	P4, P8, P12, P17, P19
7	P4, P10, P13, P16, P19

8 P4, P10, P13, P17, P19

Finalmente, el conjunto de plazas de acción está compuesto las plazas:

P2, P4, P8, P10, P12, P13, P16, P17, P19

La cantidad máxima de hilos activos se determina tomando el valor máximo de la suma de tokens en las plazas de acción PA, considerando todos los marcados posibles MA. Dado que los MA dependen del marcado inicial y que el número de tokens en la plaza P0 es variable ya que representa las imágenes que ingresan a la red, se generaron los árboles de alcanzabilidad para diferentes cantidades de tokens iniciales en la plaza P0.

Los árboles de alcanzabilidad fueron analizados utilizando la herramienta TINA. A continuación, se muestra la salida correspondiente, con el marcado inicial establecido en P0 igual a 1:

```
      des(0,29,13)
      (0,"S.`p0` S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7`
S.`p9`*2",0)
      (0,"E.`t1`",1)
      (0,"E.`t2`",2)
      (1,"S.`p11` S.`p15` S.`p2` S.`p20` S.`p3`*2 S.`p5` S.`p7` S.`p9`*2",1)
      (1,"E.`t3`",3)
      (2,"S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*2 S.`p4` S.`p7` S.`p9`*2",2)
      (2,"E.`t4`",3)
      (3,"S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p6` S.`p7`
S.`p9`*2",3)
      (3,"E.`t5`",4)
      (3,"E.`t6`",5)
      (4,"S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p8` S.`p9`",4)
      (4,"E.`t7`",6)
      (5,"S.`p1` S.`p10` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`",5)
      (5,"E.`t8`",7)
      (6,"S.`p1` S.`p11` S.`p12` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p9`",6)
      (6,"E.`t9`",8)
      (7,"S.`p1` S.`p13` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`",7)
      (7,"E.`t10`",8)
      (8,"S.`p1` S.`p11` S.`p14` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7`
S.`p9`*2",8)
      (8,"E.`t11`",9)
      (8,"E.`t12`",10)
      (9,"S.`p1` S.`p11` S.`p16` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`*2",9)
      (9,"E.`t13`",11)
      (10,"S.`p1` S.`p11` S.`p17` S.`p20` S.`p3`*3 S.`p5` S.`p7`
S.`p9`*2",10)
      (10,"E.`t14`",11)
      (11,"S.`p1` S.`p11` S.`p15` S.`p18` S.`p20` S.`p3`*3 S.`p5` S.`p7`
S.`p9`*2",11)
      (11,"E.`t15`",12)
```

```
      (12,"S.`p1`   S.`p11`   S.`p15`   S.`p19`   S.`p3`*2   S.`p5`   S.`p7`
S.`p9`*2",12)
      (12,"E.`t16`",0)
```

Por ejemplo, esto se lee de la siguiente manera:

(index,marcado,transiciones)

```
(0,"S.`p0` S.`p1` S.`p11` S.`p15` S.`p20` S.`p3`*3 S.`p5` S.`p7` S.`p9`*2",0)
```

(index,transición habilitada,destino)

```
(0,"E.`t1`",1)
```

Para procesar este archivo, se analiza su contenido utilizando el código `parcer.py`.

A continuación, se muestra la salida correspondiente, con el marcado inicial establecido en P0 igual a 1:

```
Línea 1: [1, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 0
Línea 2: [0, 0, 1, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 3: [0, 1, 0, 2, 1, 0, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 4: [0, 1, 0, 3, 0, 1, 1, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 0
Línea 5: [0, 1, 0, 3, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 6: [0, 1, 0, 3, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 7: [0, 1, 0, 3, 0, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 8: [0, 1, 0, 3, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
Línea 9: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 1] | Suma: 0
Línea 10: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1] | Suma: 1
Línea 11: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1] | Suma: 1
Línea 12: [0, 1, 0, 3, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1] | Suma: 0
Línea 13: [0, 1, 0, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0] | Suma: 1

Línea con la mayor suma:
Marcado: [0, 0, 1, 2, 0, 1, 0, 1, 0, 2, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1] | Suma: 1
```

Realizando esto para distintos marcados iniciales en P0, los resultados obtenidos fueron los siguientes:

marcado(P0)	Max(SumaTokens(PA))
1	1
2	2
3	3
4	4
5	5
6	6

7	6
8	6
9	6
10	6

Se puede observar que la suma máxima de marcas en las plazas de acción aumenta conforme incrementa el marcado inicial en P0, alcanzando un máximo de 6. Por lo tanto, el número máximo de hilos activos simultáneos para el sistema es 6.

Cabe señalar que el análisis se realizó con la red modificada, lo que excluye la transición T0. Se debe considerar dicha acción como un hilo adicional. En este caso, al incluir T0, la cantidad máxima de hilos activos simultáneos sería de **7**.

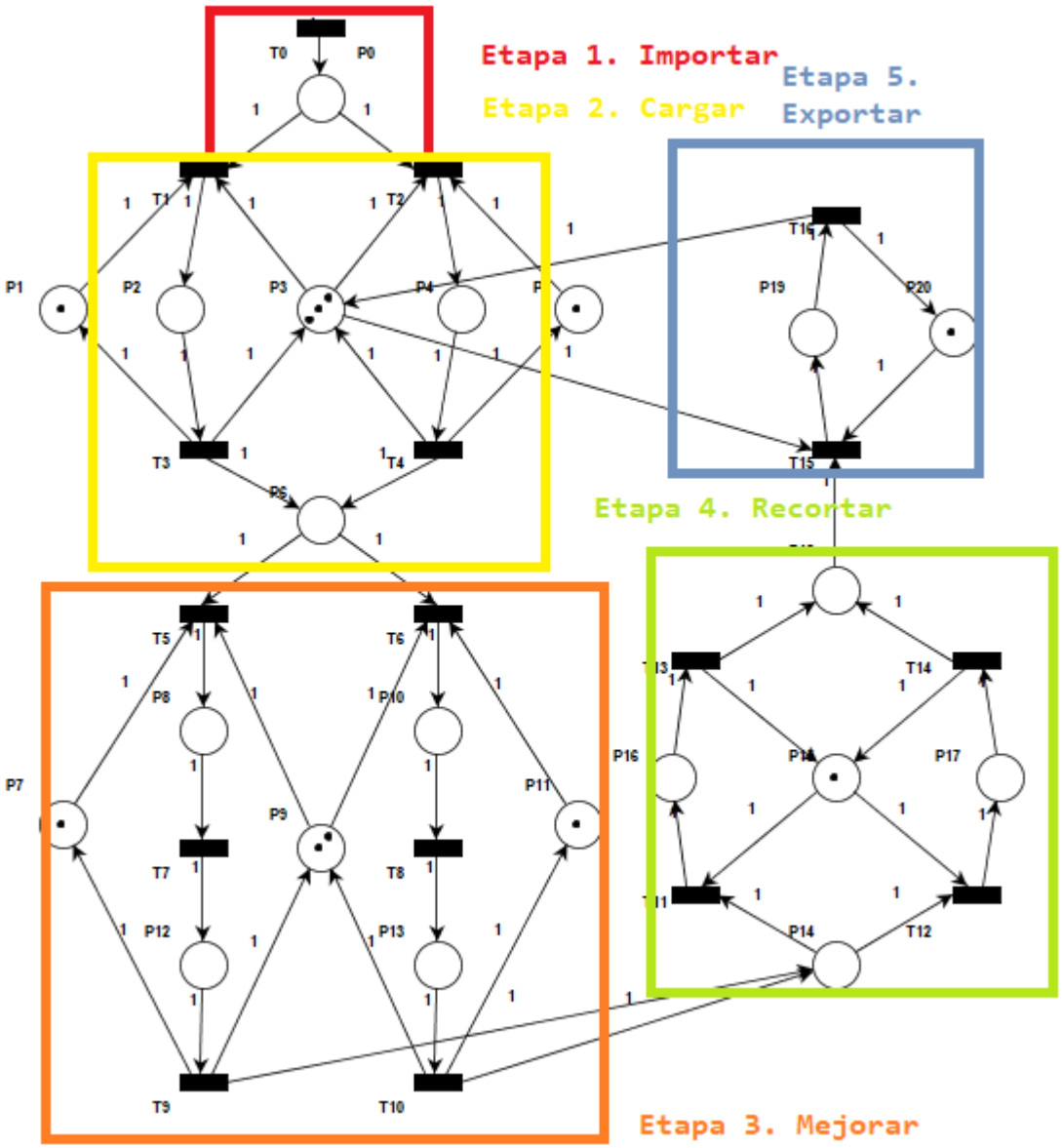
La red presenta puntos de conflictos y uniones. El primer fork ocurre en la plaza P0, de la cual surgen dos segmentos. Estos segmentos se unen en la plaza P6, que a su vez funciona como otro punto de conflicto. Nuevamente, se generan dos segmentos que se unen en la plaza P14. La plaza P14 también se divide en dos segmentos, pero estos deben ejecutarse en exclusión mutua debido a la presencia de la plaza P15. Finalmente, estos segmentos se unen en una última union en la plaza P18, que da lugar al último segmento de la red. Por lo tanto, los segmentos y las plazas asociadas a cada segmento son:

Etapa	Segmento	Plazas asociadas
Importar	S1	P0
Cargar	S2	P2
	S3	P4
Mejorar	S4	P8, P12
	S5	P10, P13
Cortar	S6	P16
	S7	P17
Exportar	S8	P19

Y las transiciones asociadas a cada segmento son:

Etapa	Segmento	Transiciones
Importar	S1	T0
Cargar	S2	T1, T3

	S3	T2, T4
Mejorar	S4	T5, T7, T9
	S5	T6, T8, T10
Cortar	S6	T11, T13
	S7	T12, T14
Exportar	S8	T15, T16



La cantidad máxima de hilos por segmento se presentan en la tabla siguiente:

Etapa	Segmento	Max(Suma(PS))
Importar	S1	1
Cargar	S2	1
	S3	1
Mejorar	S4	1
	S5	1
Cortar	S6	1
	S7	1
Exportar	S8	1

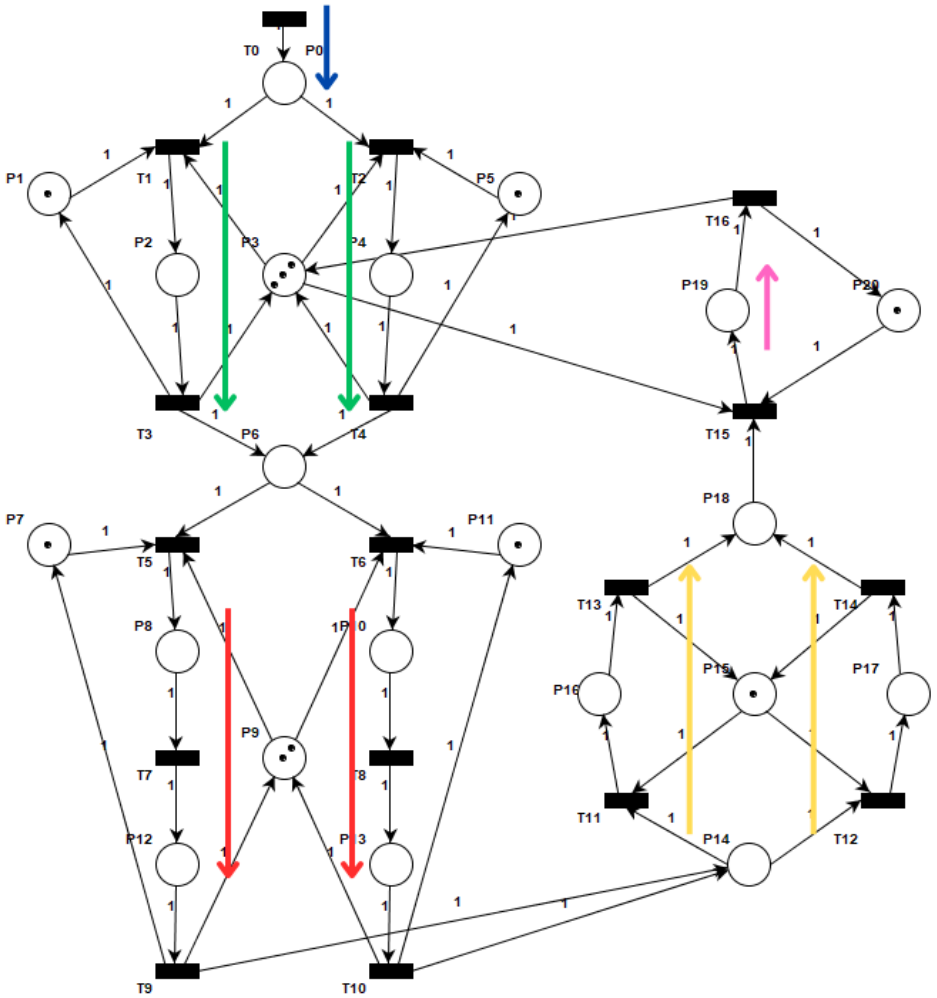


Figura 4. Red de petri en hilos

4. Implementación

4.1 Proceso

La clase Proceso es una clase abstracta que representa un concepto genérico de "proceso" que puede ejecutarse en un hilo (Runnable). Su propósito principal es servir como base para crear clases concretas que implementan la lógica de ejecución de un proceso con transiciones específicas, sincronizado mediante un monitor.

```
public Proceso(String nombre, int[] transiciones, long tiempo, Monitor
monitor) {
    this.nombre = nombre;
    this.transiciones = transiciones;
    index = 0;
    this.tiempo = tiempo;
    cuenta = new int[transiciones.length];
    this.monitor = monitor;
    timeStamp = new ArrayList<Integer>();
}
```

4.2 Importador, Cargador, Mejorador, Cortador, Exportador

Todas estas clases se heredan de la clase abstracta Proceso. Su función principal es ejecutar de manera continua su conjunto de transiciones a través de un objeto Monitor, respetando un intervalo de tiempo entre cada transición. Cabe destacar que en el caso del Importador, recibe un parámetro adicional que es la cantidad de imágenes a importar dentro del sistema y es el encargado de comenzar la detención del programa al finalizar la ejecución. En general, el método run() para las distintas tareas es similar al codigo a continuacion:

```
@Override
public void run() {
    while (!monitor.debeDetener()) {
        try {

            long antes = System.currentTimeMillis();
            monitor.dispararTransicion(transiciones[index]);
            long despues = System.currentTimeMillis();
            if (!monitor.debeDetener()) {
                setCuenta(index);
                setTimeStamp(antes,despues);
            }

            index = (index + 1) % transiciones.length;
            TimeUnit.MILLISECONDS.sleep(tiempo);
        } catch (InterruptedException e) {
                        System.err.println(getNombre()  +  ":  Proceso
interrumpido.");
            break;
        } catch (PInvariantesException e){
```

```
                    System.err.println(getNombre() + ": Error durante el
disparo de transición: " + e.getMessage());
                break;
            }
        }
        printStats();
    }
```

4.3 Colas

La clase Colas implementa semáforos para gestionar la espera y liberación de hilos en función de que estén habilitadas transiciones específicas. Sus métodos más importantes son el esperar() y liberar(). Ambos funcionan de manera semejante a los métodos acquire() y release() de la clase semáforos, sin embargo, estas funciones reciben como parámetro la transición específica para tomar y liberar el recurso.

```
    public void esperar(int transicion) {
        validarIndice(transicion);
        try {
            colas.get(transicion).acquire();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
    }

    public void liberar(int transicion) {
        validarIndice(transicion);
        colas.get(transicion).release();
    }
```

4.4 Política

La clase Políticas implementa la lógica para determinar cómo seleccionar transiciones luego de que estas hayan hecho el acquire() en las colas de condición del sistema. Su objetivo principal es definir diferentes políticas para elegir qué transición debe dispararse, considerando factores como balanceo de disparos, priorización de segmentos, y las transiciones habilitadas. Su método más importante es seleccionarTransición(). Este decide qué transición disparar basándose en la política elegida por el usuario.

4.4.1 Aleatoria

La política aleatoria devuelve cualquiera de las transiciones habilitadas que cuentan con hilos esperando en las colas de condición.

```
    case ALEATORIA:
        // Resto del codigo
        // Elegimos aleatoriamente uno de los índices guardados
        Random rand = new Random();
        transicionSeleccionada=indices.get(rand.nextInt(indices.size()));
        break;
```

4.4.2 Balanceada

La política balanceada mantiene el balance de disparos de los segmentos de las transiciones habilitadas que cuentan con hilos esperando en las colas de condición.

En primer lugar distribuye entre T1 y T2, luego lo hace para T5 y T6 y por último elige de manera equitativa entre la que menos se ha disparado hasta el momento.

```
        /*
           Balance en la etapa de Cargar
        */

        /*
           Balance en la etapa de Mejorar
        */

        // Recorre todas las transiciones para encontrar la habilitada con
menor cantidad de disparos.
        int i = 1;
        while (i < cantidadTransiciones) {
            if (transicionesHabilitadas[i] == 1 && disparos[i] < minValue &&
i!=2 && i!=6) {
                minValue = disparos[i];
                transicionSeleccionada = i;
            }
            i++;
        }
        break;
```

4.4.3 Prioritaria

La política prioritaria mantiene el balance de disparos de los segmentos de las transiciones habilitadas que cuentan con hilos esperando en las colas de condición con la condición de que se prioriza un segmento (IZQUIERDO O DERECHO) en la Etapa 3.

```
/*
      Política de procesamiento que prioriza un segmento en la etapa
seleccionada.
*/
if(Objects.equals(segmento, Segmento.IZQUIERDA)){
   if(transicionesHabilitadas[transicion]==1 && disparos[transicion-1]!=0){
       double relacion = disparos[transicion]/disparos[transicion-1];
       if (relacion<(1-prioridad)){
           return transicion;
       }
   }
} else if (Objects.equals(segmento, Segmento.DERECHA)){
   if(transicionesHabilitadas[transicion]==1 && disparos[transicion+1]!=0){
       double relacion = disparos[transicion]/disparos[transicion+1];
       if (relacion<(1-prioridad)){
           return transicion;
       }
   }
}
```

4.5 Monitor

La clase Monitor implementa una estructura para sincronizar la red de Petri, manejando los disparos de transiciones y controlando el acceso concurrente mediante semáforos. Su método más importante es dispararTransición(). Este recibe la transición a disparar y lo hace dependiendo del estado de la red de Petri. De no ser posible, pregunta cuáles fueron las causas por las que no pudo disparar y actúa dependiendo de eso. A continuación se muestra de manera resumida el código implementado en este metodo:

```
public void dispararTransicion(int transicion) {

   // Intenta adquirir el semáforo para entrar al monitor.
   try {
       mutex.acquire();
   } catch (InterruptedException e) {
       throw new RuntimeException(e);
   }
       /*
           Resto del codigo
        */
      k = petriNet.disparar(transicion);
       if (k) {

           // Registra el disparos de la transición actual.
           try {
               log.addTransicion(transicion);
           } catch (IOException e) {
               throw new RuntimeException(e);
           }

           // Incrementa el contador de disparos para la transición actual.
           contarDisparo(transicion);

       } else {
           /*
Si la transición no fue disparada, puede deberse a las siguientes razones:
1. La transición no cuenta con los tokens necesarios.
2. El tiempo alfa aún no se ha cumplido, por lo que el hilo debe esperar un
tiempo igual a alfa - ahora antes de volver a intentar disparar.
3. El tiempo beta ha sido superado, lo que impide volver a disparar la
transición.
            */
           if(!petriNet.estaHabilitada(transicion)){
                 // 1. La transición no cuenta con los tokens necesarios. El
hilo se bloquea en la cola correspondiente.
               mutex.release();
                 colas.esperar(transicion); // El hilo entra en la cola de la
transición.
               k = true;
            } else if (petriNet.getVentanaTiempos()[transicion]==1) {
               // 2. El tiempo alfa aún no se ha cumplido, por lo que el hilo
debe esperar un tiempo igual a alfa - ahora antes de volver a intentar
disparar.
```

```
            } else if (petriNet.getVentanaTiempos()[transicion]==2) {
                // 3. El tiempo beta ha sido superado, lo que impide volver a
disparar la transición.
            }
        }
    }
    mutex.release(); // Libera el semáforo al salir del monitor.
}
```

4.6 PetriNet

La clase modela el comportamiento dinámico de una Red de Petri, permitiendo representar su estructura, evaluar su estado (marcado) y gestionar disparos de transiciones bajo restricciones de habilitación por tokens y ventanas temporales. Su método más importante es disparar(). Este recibe una transición y evalúa si está habilitada por token y, en caso de estar temporizada, si el disparo llegó dentro de la ventana de tiempos para disparar.

```
    public boolean disparar(int transicion) {

        // Verifica si la transición es válida.
        if (transicion < 0 || transicion >= cantidadTransiciones) {
                throw new IllegalArgumentException("Índice de transición
inválido");
        }

    // Verifica si la transición está habilitada por cantidad de tokens en
las plazas.
        if (estaHabilitada(transicion)) {

    // Verifica si la transición está dentro de la ventana de tiempos.
            long tiempo = System.currentTimeMillis();
            if (validarTiempos(transicion,tiempo)) {

                int[] nuevoMarcado = operar(transicion);

                  if (Arrays.stream(nuevoMarcado).anyMatch(valor -> valor <
0)) {
                      System.out.println("Los valores del marcado no pueden
ser negativos.");
                    return false;
                }

    // Verifica que no se hayan corrompido los invariantes de plaza.
                if (!verificarInvariantesPlaza()) {
    // Actualiza el marcado y las transiciones habilitadas según los
tokens disponibles.
                    actualizarHabilitadas(nuevoMarcado);
                    return true;
                } else {
                    throw new PInvariantesException();
                }
```

```
            }
        }
        return false;
    }
```

4.7 Plazas y Transiciones

Las clases Plazas y Transiciones modelan los elementos que componen la red de petri. En el caso de las plazas, tiene en cuenta los tokens que esta posee y lo utiliza luego para corroborar los Invariantes de Plaza, propiedad que no se puede incumplir a lo largo de la evolución de la red. En el caso de las transiciones, tienen en cuenta el tiempo alfa y beta en caso de estar temporizadas y si está habilitada o no por tokens.

```
    public Plaza(int numero, int tokens) {
            // P-numero
            this.nombre = "P" + numero;
            this.tokens = tokens;
        }

    public Transición(int numero, long alfa, long beta) {
        // T-numero
        this.nombre = "T" + numero;
        this.alfa = alfa;
        this.beta = beta;
        this.esInmediata = alfa == 0;
        this.habilitada = 0;
    }
```

4.8 Main

La clase Main coordina la ejecución de procesos concurrentes que representan distintas etapas de un sistema de producción (como importar, cargar, mejorar, cortar y exportar). Su objetivo es asegurar que se cumpla un número máximo de disparos de transiciones en una red de Petri, deteniendo los procesos de forma ordenada al finalizar.

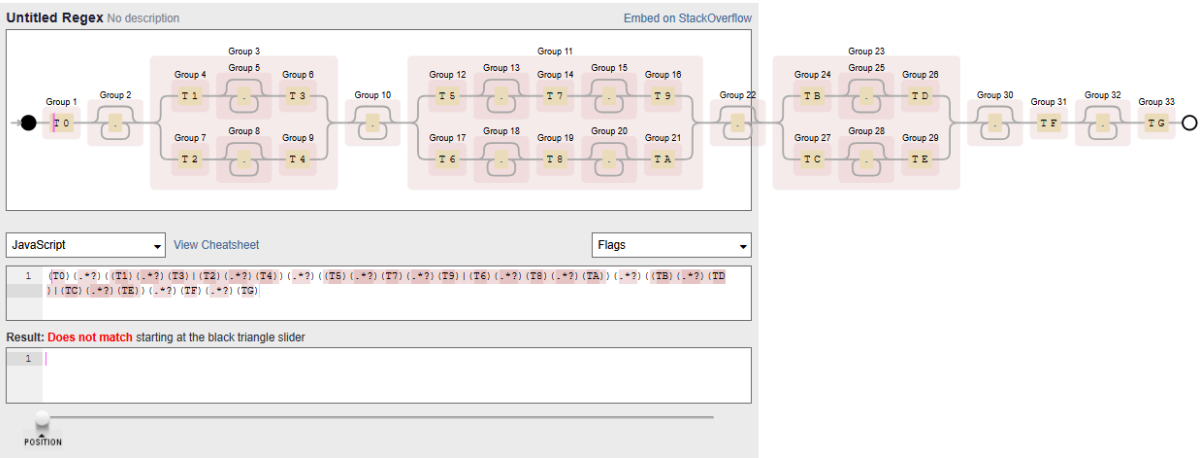
```
public static void main(String[] args) {
    SimpleDateFormat formatter = new SimpleDateFormat("EEE MMM dd HH:mm:ss z
yyyy");

    logMessage("Se inició la ejecución de la red: " + formatter.format(new
Date()));

   try {
        // Ruta del archivo de log
        String LOG_PATH = "log/log.txt";
        Log log = new Log(LOG_PATH);
        Monitor monitor = new Monitor(log,politica,segmento,prioridad,red);
        Proceso[] procesos = inicializarProcesos(monitor);

        Thread[] threads = startProcesos(procesos);
        esperarProcesos(monitor);
```

```
            pararProcesos(monitor,threads);
            printStats(monitor);

            log.closeFile();
        } catch (IOException e) {
            System.err.println("Error al inicializar el log: " + e.getMessage());
        }

        logMessage("Se finalizó la ejecución de la red: " + formatter.format(new
Date()));
    }
```

5. Análisis de Invariantes de Transición

Debemos asegurar que luego de una ejecución de n transiciones, una red de petri siempre cumpla con sus invariantes de transición. Para ello, fue necesario realizar un script en el lenguaje Python, que lea de un archivo .txt las líneas de caracteres y las procese con una expresión regular. A continuación, se detalla la expresión regular utilizada y el script generado para el análisis.

```
(T0)(.*?)((T1)(.*?)(T3)|(T2)(.*?)(T4))(.*?)((T5)(.*?)(T7)(.*?)(T9)|(T6)(.*?)(
T8)(.*?)(TA))(.*?)((TB)(.*?)(TD)|(TC)(.*?)(TE))(.*?)(TF)(.*?)(TG)
```



A continuación, se muestra el resultado para el análisis de un log donde la red ejecuto 5 invariantes de transición:

Salida del log:

T0T1T0T2T3T5T7T4T6T0T1T9T11T8T3T0T2T5T10T13T12T15T4T0T1T16T7T6T14T15T3T9T11T8T16T5T10T13T12T15T16T7T14T15T9T11T16T13T15T16

El resultado obtenido es el siguiente:

```
Resultado:
Número de reemplazos realizados: 5
Éxito
```

Si incluimos una transición al azar dentro del log:

T0T16T1T0T2T3T5T7T4T6T0T1T9T11T8T3T0T2T5T10T13T12T15T4T0T1T16T7T6T14T15T3T9T11T8T16T5T10T13T12T15T16T7T14T15T9T11T16T13T15T16

El resultado obtenido es el siguiente:

```
Resultado: TG
Número de reemplazos realizados: 5
Falló
```

6. Tests

Se evaluaron las funcionalidades principales de las clases críticas del sistema, asegurando que cada componente se comporte de acuerdo con los requisitos esperados. Se realizaron tests unitarios para verificar la correcta inicialización de objetos, la lógica detrás de la sincronización y el disparo de transiciones en la clase Monitor, hasta la implementación de políticas de selección de transiciones en Políticas.

7. Resultados

Los tiempos elegidos inicialmente para las tareas son los siguientes:

Tareas	Tiempo [ms]
Importador	100
Cargador	100
Mejorador	100
Cortador	100
Exportador	100

La figura a continuación muestra la ejecución del programa con 1000 invariantes como objetivo. La política seleccionada es BALANCEADA y las transiciones son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 500 veces.
La transición 12 se disparó 500 veces.
La transición 13 se disparó 500 veces.
La transición 14 se disparó 500 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 10:38:20 GMT-03:00 2025
```

Se observa que el balance entre las distintas ramas de la red se mantiene constante a lo largo de toda la ejecución.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 80% para el segmento izquierdo de la etapa 3, es decir, las transiciones T11 y T13 reciben el 80% de las imágenes, y las transiciones son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 833 veces.
La transición 12 se disparó 167 veces.
La transición 13 se disparó 833 veces.
La transición 14 se disparó 167 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 10:45:22 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{833}{1000}100 \ = \ 83,3\%$$

Se observa que el segmento izquierdo de la etapa 3 recibe 83,3% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente. Por último, cabe destacar que el resto de las transiciones mantienen el balance en los disparos.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 70% para el segmento derecho de la etapa 3, es decir, las transiciones T12 y T14 reciben el 70% de las imágenes, y las transiciones son inmediatas.

Con esto se busca demostrar que la política PRIORITARIA funciona tanto para el segmento izquierdo como el derecho, y el peso a considerar en el porcentaje de imágenes que procesa un lado y el otro es a libre elección del usuario.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 501 veces.
La transición 6 se disparó 499 veces.
La transición 7 se disparó 501 veces.
La transición 8 se disparó 499 veces.
La transición 9 se disparó 501 veces.
La transición 10 se disparó 499 veces.
La transición 11 se disparó 232 veces.
La transición 12 se disparó 768 veces.
La transición 13 se disparó 232 veces.
La transición 14 se disparó 768 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 10:53:36 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{768}{1000}100 \ = 76,8\%$$

Se observa que el segmento derecho de la etapa 3 recibe 76,8% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente para ambos segmentos de la etapa 3 de la red.

Ahora desbalanceamos las líneas de trabajo, es decir, asignamos tiempos más rápidos en un lado que en el otro, política BALANCEADA sin asignar prioridad a hilos y manteniendo sin tiempos las transiciones.

Tareas	Tiempo [ms]
Importador	100
Cargador 1	100
Cargador 2	50
Mejorador 1	100
Mejorador 2	85

Cortador	100
Exportador	100

```
Resultados:
Cantidad de invariantes: 200
La transición 0 se disparó 200 veces.
La transición 1 se disparó 99 veces.
La transición 2 se disparó 101 veces.
La transición 3 se disparó 99 veces.
La transición 4 se disparó 101 veces.
La transición 5 se disparó 92 veces.
La transición 6 se disparó 108 veces.
La transición 7 se disparó 92 veces.
La transición 8 se disparó 108 veces.
La transición 9 se disparó 92 veces.
La transición 10 se disparó 108 veces.
La transición 11 se disparó 100 veces.
La transición 12 se disparó 100 veces.
La transición 13 se disparó 100 veces.
La transición 14 se disparó 100 veces.
La transición 15 se disparó 200 veces.
La transición 16 se disparó 200 veces.
Se finalizó la ejecución de la red: Fri Jan 31 12:08:10 GMT-03:00 2025
```

Se observa un desbalance mínimo entre las ramas. Lo interesante de esto es que la diferencia entre ramas es considerablemente mayor al porcentaje de desvío de imágenes entre ramas. Podemos considerar que se mantiene un balance óptimo para condiciones extremas.

8. Tiempos

Los tiempos elegidos inicialmente para las transiciones son los siguientes:

Transición	(**α**, β) [ms]
T0	(100, 300)
T3	(100, 300)
T4	(100, 300)
T7	(100, 300)
T8	(100, 300)

T9	(100, 300)
T10	(100, 300)
T13	(100, 300)
T14	(100, 300)
T16	(100, 300)

La figura a continuación muestra la ejecución del programa con 1000 invariantes como objetivo. La política seleccionada es BALANCEADA y las transiciones son temporizadas según la tabla anterior.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 500 veces.
La transición 12 se disparó 500 veces.
La transición 13 se disparó 500 veces.
La transición 14 se disparó 500 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 11:08:40 GMT-03:00 2025
```

Nuevamente se observa que el balance entre las distintas ramas de la red se mantiene constante a lo largo de toda la ejecución, independientemente de las transiciones temporales.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 80% para el segmento izquierdo de la etapa 3, es decir, las transiciones T11 y T13 reciben el 80% de las imágenes, y las transiciones no son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 833 veces.
La transición 12 se disparó 167 veces.
La transición 13 se disparó 833 veces.
La transición 14 se disparó 167 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 11:15:45 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{833}{1000}100 \ = \ 83,3\%$$

Se observa que el segmento izquierdo de la etapa 3 recibe 83,3% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente a pesar de incluir los tiempos en las transiciones. Por último, cabe destacar que el resto de las transiciones mantienen el balance en los disparos.

A continuación, se muestra la ejecución del programa con 1000 invariantes como objetivo, la política seleccionada en PRIORITARIA, con un peso del 80% para el segmento derecho de la etapa 3, es decir, las transiciones T12 y T14 reciben el 80% de las imágenes, y las transiciones no son inmediatas.

```
Resultados:
Cantidad de invariantes: 1000
La transición 0 se disparó 1000 veces.
La transición 1 se disparó 500 veces.
La transición 2 se disparó 500 veces.
La transición 3 se disparó 500 veces.
La transición 4 se disparó 500 veces.
La transición 5 se disparó 500 veces.
La transición 6 se disparó 500 veces.
La transición 7 se disparó 500 veces.
La transición 8 se disparó 500 veces.
La transición 9 se disparó 500 veces.
La transición 10 se disparó 500 veces.
La transición 11 se disparó 167 veces.
La transición 12 se disparó 833 veces.
La transición 13 se disparó 167 veces.
La transición 14 se disparó 833 veces.
La transición 15 se disparó 1000 veces.
La transición 16 se disparó 1000 veces.
Se finalizó la ejecución de la red: Thu Jan 23 11:21:43 GMT-03:00 2025
```

Teniendo en cuenta la figura anterior,

$$\frac{833}{1000}100 \ = 83,3\%$$

Se observa que el segmento derecho de la etapa 3 recibe 83,3% de las imágenes que la red procesa a lo largo de toda la ejecución. Por lo tanto, concluimos que la política PRIORITARIA funciona correctamente para ambos segmentos de la etapa 3 de la red independientemente de las transiciones temporales.

8.1 Tiempos propuestos para las tareas

Es necesario hacer un análisis considerando paralelismo, y además mostrar la variación del tiempo total de ejecución con base en los tiempos elegidos para cada proceso. Para ello, se realizaron pruebas, donde en cada una de ellas tomamos como pivote cada proceso, probando con distintos tiempos y manteniendo el tiempo de los demás, y así observar cuánto afecta en el tiempo total de ejecución.

Este análisis se realizó bajo la política BALANCEADA y considerando transiciones inmediatas. Se realizaron 200 invariantes de transición en cada ejecución.

9.1.1 Prueba 1

N°	Importador	Resto de tareas	Tiempo total
1	1 ms	100 ms	45 s
2	100 ms	100 ms	44 s
3	200 ms	100 ms	45 s

9.1.2 Prueba 2

N°	Cargador	Resto de tareas	Tiempo total
4	1 ms	100 ms	45 s
5	100 ms	100 ms	44 s
6	200 ms	100 ms	45 s

9.1.3 Prueba 3

N°	Mejorador	Resto de tareas	Tiempo total
7	1 ms	100 ms	44 s
8	100 ms	100 ms	46 s
9	200 ms	100 ms	63 s

9.1.4 Prueba 4

N°	Cortador	Resto de tareas	Tiempo total
10	1 ms	100 ms	45 s
11	100 ms	100 ms	45 s
12	200 ms	100 ms	45 s

9.1.5 Prueba 5

N°	Exportador	Resto de tareas	Tiempo total
13	1 ms	100 ms	35 s
14	100 ms	100 ms	45 s
15	200 ms	100 ms	85 s

Con base en los resultados obtenidos se presenta una propuesta de tiempos para cada tarea. Los tiempos asignados a cada una deben ser seleccionados considerando su impacto en el tiempo total de ejecución y priorizando la eficiencia de la red.

Tareas	Tiempo [ms]
Importador	100 ms
Cargador	100 ms
Mejorador	80 ms
Cortador	100 ms
Exportador	50 ms

Con estos tiempos propuestos, volvemos a ejecutar el programa para 200 invariantes de transición a realizar y obtenemos el siguiente resultado:

```
Se inició la ejecución de la red: Thu Jan 23 12:31:50 GMT-03:00 2025
Resultados:
Cantidad de invariantes: 200
La transición 0 se disparó 200 veces.
La transición 1 se disparó 101 veces.
La transición 2 se disparó 99 veces.
La transición 3 se disparó 101 veces.
La transición 4 se disparó 99 veces.
La transición 5 se disparó 100 veces.
La transición 6 se disparó 100 veces.
La transición 7 se disparó 100 veces.
La transición 8 se disparó 100 veces.
La transición 9 se disparó 100 veces.
La transición 10 se disparó 100 veces.
La transición 11 se disparó 100 veces.
La transición 12 se disparó 100 veces.
La transición 13 se disparó 100 veces.
La transición 14 se disparó 100 veces.
La transición 15 se disparó 200 veces.
La transición 16 se disparó 200 veces.
Se finalizó la ejecución de la red: Thu Jan 23 12:32:18 GMT-03:00 2025
```

8.2 Tiempos propuestos para las transiciones

Este análisis se realizó bajo la política BALANCEADA y considerando transiciones temporales. Se realizaron 200 invariantes de transición en cada ejecución. Los tiempos de las tareas son los obtenidos en el apartado anterior.

Transición	(α1, β1) [ms]	Tiempo total	(α2, β2) [ms]	Tiempo total	(α3, β3) [ms]	Tiempo total
T0	(100, 300)	83 s	(100, 300)	79 s	(100, 300)	72 s
T3	(100, 300)		(100, 300)		(100, 300)	
T4	(100, 300)		(100, 300)		(100, 300)	
T7	(1, 200)		(100, 300)		(200, 400)	
T8	(1, 200)		(100, 300)		(200, 400)	
T9	(1, 200)		(100, 300)		(200, 400)	
T10	(1, 200)		(100, 300)		(200, 400)	
T13	(100, 300)		(100, 300)		(100, 300)	
T14	(100, 300)		(100, 300)		(100, 300)	
T16	(100, 300)		(100, 300)		(100, 300)	

Transición	(α1, β1) [ms]	Tiempo total	(α2, β2) [ms]	Tiempo total	(α3, β3) [ms]	Tiempo total
T0	(100, 300)	73 s	(100, 300)	76 s	(100, 300)	75 s
T3	(100, 300)		(100, 300)		(100, 300)	
T4	(100, 300)		(100, 300)		(100, 300)	
T7	(100, 300)		(100, 300)		(100, 300)	
T8	(100, 300)		(100, 300)		(100, 300)	
T9	(100, 300)		(100, 300)		(100, 300)	
T10	(100, 300)		(100, 300)		(100, 300)	
T13	(100, 300)		(100, 300)		(100, 300)	
T14	(100, 300)		(100, 300)		(100, 300)	
T16	(1, 200)		(100, 300)		(200, 400)	

Con base en los resultados obtenidos se presenta una propuesta de tiempos para cada transición.

Transición	(**α**, β) [ms]
T0	(100, 300)
T3	(100, 300)
T4	(100, 300)
T7	(200, 400)
T8	(200, 400)
T9	(200, 400)
T10	(200, 400)
T13	(100, 300)
T14	(100, 300)
T16	(1, 200)

Con estos tiempos propuestos, volvemos a ejecutar el programa para 200 invariantes de transición a realizar y obtenemos el siguiente resultado:

```
Se inició la ejecución de la red: Thu Jan 23 13:15:31 GMT-03:00 2025
Resultados:
Cantidad de invariantes: 200
La transición 0 se disparó 200 veces.
La transición 1 se disparó 100 veces.
La transición 2 se disparó 100 veces.
La transición 3 se disparó 100 veces.
La transición 4 se disparó 100 veces.
La transición 5 se disparó 100 veces.
La transición 6 se disparó 100 veces.
La transición 7 se disparó 100 veces.
La transición 8 se disparó 100 veces.
La transición 9 se disparó 100 veces.
La transición 10 se disparó 100 veces.
La transición 11 se disparó 100 veces.
La transición 12 se disparó 100 veces.
La transición 13 se disparó 100 veces.
La transición 14 se disparó 100 veces.
La transición 15 se disparó 200 veces.
La transición 16 se disparó 200 veces.
Se finalizó la ejecución de la red: Thu Jan 23 13:16:39 GMT-03:00 2025
```

9. Conclusiones

Se demostró la viabilidad de utilizar Redes de Petri para modelar y gestionar sistemas concurrentes. El diseño de la red permitió representar de manera eficiente la interacción entre hilos, asegurando la sincronización y el cumplimiento de invariantes estructurales y de transición. Estas propiedades destacan la robustez del sistema frente a posibles condiciones de carrera y conflictos de recursos compartidos.

Además, se realizaron análisis de las políticas de ejecución aplicadas. La política BALANCEADA logró mantener un equilibrio constante en la distribución de las tareas entre las ramas del sistema, mientras que la política PRIORITARIA permitió priorizar segmentos específicos de procesamiento, como lo demuestran los resultados obtenidos tanto en las transiciones inmediatas como en las temporizadas. Estas pruebas validaron la correcta implementación y versatilidad del sistema para adaptarse a distintas prioridades operativas.

Finalmente, el análisis de los tiempos asignados para cada tarea y transición subraya la importancia de un ajuste óptimo de parámetros para maximizar la eficiencia del sistema. Se determinó que ciertas configuraciones de tiempo influyen directamente en la duración total del procesamiento, lo que permitió proponer ajustes que optimizan la utilización de los recursos y reducen los tiempos sin comprometer la funcionalidad del sistema. Esto confirma la utilidad de este modelo en escenarios reales donde la concurrencia y la gestión eficiente de recursos son cruciales.

10. Referencias

/1/: Artículo cantidad de hilos.
https://www.researchgate.net/publication/358104149_Algoritmos_para_determinar_cantidad_y_responsabilidad_de_hilos_en_sistemas_embebidos_modelados_con_Redes_de_Petri_S_3_PR

11. Repositorio de GitHub

nachoborgatello/Trabajo-Final-Programacion-Concurrente