Universidad Zaragoza

Proyecto Fin de Carrera en Ingeniería en Informática

Optimización mediante Búsqueda Tabú para problemas no lineales en Redes de Petri

Luis Ignacio Frisón Alegre

Director: Ricardo J. Rodríguez Fernández

Codirector: Javier Campos Laclaustra

Departamento de Informática e Ingeniería de Sistemas, Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre de 2017
Curso 2016/2017

Agradecimientos

A mi familia,
A mis amigos,
A Gloria Aguilar,
A Javier Campos y Ricardo Rodríguez.

GRACIAS

Optimización mediante Búsqueda Tabú para problemas no lineales en Redes de Petri

RESUMEN

La metaheurística de la Búsqueda Tabú, que consiste en una mejora del método de Búsqueda Local mediante el uso de estructuras de memoria, trata de resolver el problema del estancamiento en un máximo local marcando la solución alcanzada como Tabú y evitándola un número de iteraciones dada (lo que se conoce como Tenencia Tabú), explorando así otras soluciones y permitiendo al algoritmo alcanzar soluciones mejores.

Las redes de Petri son una herramienta formal útil para el diseño, análisis e implementación de sistemas concurrentes y distribuidos. Existe una amplia gama de técnicas de análisis funcional y no funcional de este tipo de modelos; algunas de esas técnicas se basan en lo que se conoce como teoría estructural, es decir, en la estructura del modelo (matriz de incidencia, marcado inicial, temporización si la hay) y usan técnicas de programación matemática (como la programación lineal o no lineal) para obtener resultados sobre el comportamiento del modelo. En el ámbito del análisis no funcional, algunas de las propiedades o índices analizables tienen que ver con el rendimiento o prestaciones del sistema (*throughput*).

El problema min-max de Bernardi y Campos consiste en hallar una cota inferior del tiempo de ciclo de la transición que queramos para una red de Petri dada mediante un algoritmo de programación matemática no lineal. Este algoritmo se basa principalmente en elementos estructurales de la Red de Petri bajo estudio.
PeabraiN es una herramienta desarrollada en Java que ofrece al usuario una interfaz gráfica amigable para trabajar con Redes de Petri.

En este trabajo, se desea implementar la Búsqueda Tabú y adaptarla a la resolución del problema min-max de Bernardi y Campos, para después integrar dicha funcionalidad en PeabraiN y permitir al usuario aplicarla y obtener resultados. Después de la integración, se va a estudiar la aplicación de la búsqueda Tabú a diversos ejemplos con el fin de evaluar su funcionalidad.

Índice

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

Las redes de Petri son una herramienta formal útil para el diseño, análisis e implementación de sistemas concurrentes y distribuidos. Existe una amplia gama de técnicas de análisis funcional y no funcional de este tipo de modelos. Algunas de esas técnicas se basan en lo que se conoce como teoría estructural, es decir, en la estructura del modelo (matriz de incidencia, marcado inicial, temporización si la hay) y usan técnicas de programación matemática (como la programación lineal o no lineal) para obtener resultados sobre el comportamiento del modelo. En el ámbito del análisis no funcional, algunas de las propiedades o índices analizables tienen que ver con el rendimiento o prestaciones del sistema, como el *throughput* o la utilización. Para calcular ese tipo de índices se han propuesto algunos problemas de programación lineal que pueden resolverse eficientemente con algoritmos como *simplex*, aún siendo exponencial en el peor caso [Meg87]. Otros problemas de la literatura, sin embargo, son no lineales, como el problema de programación cuadrática que vamos a estudiar en esta memoria; y por tanto para este tipo de problemas es interesante explorar técnicas de resolución no exacta, como la metaheurística conocida como "Búsqueda Tabú" [Glo89].

La metaheurística de la Búsqueda Tabú, que consiste en una mejora del método de búsqueda local mediante el uso de estructuras de memoria, trata de resolver el problema del estancamiento en un máximo local marcando la solución alcanzada como Tabú y evitándola un número de iteraciones dada, lo que se conoce como Tenencia Tabú, explorando así otras soluciones y permitiendo al algoritmo alcanzar soluciones mejores.

PeabraiN es una herramienta desarrollada dentro del grupo de investigación GISED de la Universidad de Zaragoza como una herramienta para prototipado rápido y análisis de Redes de Petri [Rod17]. Esta herramienta permite realizar diferentes cálculos de propiedades y características de Redes de Petri, a través de problemas de optimización lineal. PeabraiN proporciona todas estas funcionalidades como plugins de una herramienta gráfica para modelado y verificación de redes de Petri llamada PIPE [BLP+07]. PeabraiN, como PIPE, está desarrollada con Java usando el entorno de desarrollo Eclipse.

En este trabajo, se pretende desarrollar un plugin de la herramienta PeabraiN para el cálculo de throughput mediante la heurística de optimización de Búsqueda Tabú en redes de Petri donde las técnicas tradicionales de cálculo de throughput no son aplicables. El uso de Búsqueda Tabú con este fin se comentó como sugerencia en [ZRS01].
Por último, la heurística de Búsqueda Tabú se comparará experimentalmente con otras técnicas existentes en la literatura para el cálculo de throughput.

1.1. Objetivos del proyecto

1. Implementar la metaheurística de la Búsqueda Tabú en la resolución del problema min-max de Bernardi y Campos [BC13] para Redes de Petri.
2. Integrar dicho algoritmo en una herramienta basada en PIPE, de nombre PeabraiN, de manera que permita una interacción directa, visual y sencilla con el usuario.

1.2. Organización de la memoria

En el capítulo 2 se explican los conceptos de metaheurística y de heurística, con ejemplos de los mismos. En el capítulo 3 se explica detalladamente en qué consiste la Búsqueda Tabú mediante un ejemplo sencillo (en concreto, su aplicación al problema de las n-reinas). En el capítulo 4 se explican los conceptos básicos de redes de Petri, el problema objetivo de estudio que queremos resolver con la Búsqueda Tabú y los detalles de la implementación de dicho algoritmo en dicho problema. En el capítulo 5 se explica brevemente en qué consiste y cómo es la arquitectura de PeabraiN, herramienta en la que queremos introducir nuestro algoritmo, seguido de los detalles de integración en dicha herramienta. En el capítulo 6 exponemos y analizamos los resultados obtenidos al aplicar nuestro algoritmo a distintos casos de estudio. Por último, en el capítulo 7 exponemos las conclusiones, incluyendo los objetivos cumplidos, las restricciones del algoritmo, el material sobre el que me he tenido que informar, una propuesta de trabajo futuro, lo que he aprendido y los problemas que me han surgido. En el anexo, comento cuántas horas y en qué proporción he dedicado al proyecto, desglosadas por meses y tareas en un diagrama de Gantt y, por otra parte, desglosadas por tareas en un diagrama de sectores.

Capítulo 2

Algoritmos heurísticos para la solución de problemas difíciles (NP)

Como se ha dicho anteriormente, la Búsqueda Tabú es una metaheurística, como también lo son otros algoritmos como Simulated Annealing, los Algoritmos Genéticos y Ramificación y Poda.
Este tipo de algoritmos se usan comúnmente para la resolución de los llamados problemas difíciles; es decir, aquellos problemas en los que a pesar de que somos capaces de verificar muy rápidamente si una solución es tal o no, no es posible obtener una solución en un tiempo rápido. En Ciencias de la Computación, es lo que se conoce como problemas de la clase NP.
A continuación, definiré lo que es una metaheurística y haré una breve reseña de dichas metaheurísticas, para que el lector comprenda mejor dicho término, y se ponga en contexto.

2.1. Heurística vs metaheurística

Heurística: una heurística es un algoritmo concreto que sacrifica el calcular el valor exacto de la solución de un problema concreto de optimización, conformándose con obtener una solución que muchas veces será subóptima, pero con la ventaja de ejecutarse en un tiempo menor que el que tendría el cálculo de la solución exacta. Normalmente se utilizan las heurísticas para resolver de forma aproximada problemas que están en la clase NP (es decir, aquellos para los que no se conoce una solución exacta en tiempo de ejecución polinómico).

Metaheurística: se habla de metaheurísticas cuando se tiene una técnica de resolución no de un problema concreto, sino de una familia de problemas amplia [GK03]. Son algo así como un "esquema general" para la resolución de problemas mediante algoritmos heurísticos. Los algoritmos voraces, los de escalada (*hill climbing*), los algoritmos genéticos, los de simulated annealing, búsqueda tabú, etc., son algunos ejemplos de metaheurísticas.

2.2. Simulated Annealing

En algunos procesos metalúrgicos se calienta un material y después se enfría gradualmente de manera controlada, para aumentar el tamaño de los cristales que lo componen y reducir sus defectos. Si la temperatura es suficientemente alta para asegurar un estado aleatorio y el enfriamiento es suficientemente lento para asegurar el equilibrio térmico, los átomos alcanzarán un estado siguiendo un patrón que corresponde al mínimo de energía global para obtener un cristal perfecto. El recocido consiste en conseguir un ablandamiento del material enfriándolo lentamente después de calentarlo.
La analogía del enfriamiento simulado con la resolución de problemas consiste en mediante la reducción de aceptar malas soluciones (peores) conforme exploramos el espacio de estados. Las malas soluciones contribuyen a hacer una búsqueda más extensiva.
El proceso del Simulated Annealing es el siguiente:

1. **Inicialización**. Solución aleatoria. "Temperatura" muy alta.
2. **Movimiento**. Perturbar la solución con algún tipo de movimiento.
3. **Evaluar**. Calcular la variación de la puntuación (energía).

4. **Elegir**. Dependiendo del resultado de la evaluación, aceptar o rechazar. La probabilidad de aceptación depende de la "temperatura", esto es, si el vecino es mejor, lo elegimos, y si es peor, lo elegimos con una probabilidad que varía con la temperatura. A temperatura alta, aceptamos casi cualquier vecino, mientras que a temperatura baja, seguimos la dirección de mejora.
5. **Actualizar y repetir**. Reducir el valor de la temperatura. Volver al paso 2 hasta que alcancemos el "punto de enfriamiento".

2.3. Algoritmos Genéticos

Están inspirados en el proceso de evolución biológica. Utilizan el principio de selección natural (cruce y mutación) para resolver problemas difíciles. A continuación explicaremos dicha heurística con un ejemplo sencillo, consistente en maximizar $f(x) = x_2$ con x entero entre 0 y 31. El esquema básico es el siguiente:

1. **Codificación**: utilizar cadenas de bits para representar las soluciones (que pueden ser números enteros, reales, conjuntos,...). La ventaja es que los operadores de cruce y mutación son simples, y el inconveniente es que no siempre resulta "natural". Representación en binario: 01101.
2. **Población inicial generada aleatoriamente (tamaño 4)**: [01101, 11000, 01000, 10011].
3. **Función de calidad**: $f(x) = x_2$.

Cromosoma	x	f(x)
01101	13	169
11000	24	576
01000	8	64
10011	19	361

4. **Población intermedia:** cada individuo puede ser elegido con una probabilidad proporcional a su "calidad".

Cromosoma	x	f(x)	P(f(x))	copias
01101	13	169	0.14	1
11000	24	576	0.49	2
01000	8	64	0.06	0
10011	19	361	031	1

5. **Combinación**: se combinan parejas de la población intermedia de manera aleatoria.
6. **Cruce**: se elige un punto intermedio y se intercambian los genes de los "padres".
   110|00 11011
   100|11 => 10000
7. **Mutación**: cambio aleatorio de algún bit elegido al azar con probabilidad pequeña.

Cromosoma	x	f(x)
110 11	27	729
100 00	16	259
011 00	12	144
110 01	25	625

8. **Selección**: primero, se asigna una probabilidad de supervivencia proporcional a la calidad; segundo, se genera una población intermedia y tercero, se eligen parejas de forma aleatoria. Existe una restricción, y es que no se pueden cruzar elementos de dos generaciones distintas.

2.4. Ramificación y Poda

El esquema de Ramificación y Poda se aplica mayoritariamente para resolver cuestiones o problemas de optimización. La técnica de Ramificación y Poda se suele interpretar como un árbol de soluciones, donde cada rama nos lleva a una posible solución posterior a la actual. La característica de esta técnica con respecto a otras anteriores (y a la que debe su nombre) es que el algoritmo se encarga de detectar en qué ramificación las soluciones dadas ya no están siendo óptimas, para "podar" esa rama del árbol y no continuar malgastando recursos y procesos en casos que se alejan de la solución óptima.

A diferencia de otros esquemas, metaheurísticos, mencionados en esta memoria, un algoritmo de Ramificación y Poda siempre encuentra la solución óptima, si existe. Aunque su coste temporal de ejecución crece de forma exponencial, en general, suele dar buenos resultados en la práctica.

Capítulo 3

Búsqueda Tabú

3.1. Descripción detallada de la Búsqueda Tabú

La Búsqueda Tabú, como ya se ha comentado anteriormente, es una mejora del método de Búsqueda Local mediante el uso de estructuras de memoria [Glo89]. Dichas estructuras se pueden clasificar en dos tipos: memoria reciente, que es la básica que todo algoritmo de Búsqueda Tabú ha de tener, y memoria frecuente, que es opcional pero que en algunos casos supone una clara ventaja respecto a no tenerla. La memoria reciente consiste en la implementación del concepto propiamente dicho de Tabú del algoritmo. La memoria frecuente consiste en la implementación de los conceptos de diversificación e intensificación. La diversificación apoya la búsqueda en regiones no visitadas, mientras que la intensificación apoya los movimientos que históricamente se han encontrado buenos.

En términos generales, la búsqueda tabú es un algoritmo iterativo que progresa de iteración en iteración moviéndose en cada una desde una solución provisional a otra considerada mejor. Esta solución provisional es escogida de entre un vecindario de posibles soluciones. Este vecindario, del tamaño que queramos, es un conjunto de soluciones obtenidas a partir de la solución provisional. El método de obtención de estas soluciones del vecindario puede variar, ya sea a partir de una cota (como el algoritmo implementado), a partir del intercambio de dos elementos al azar (como en el problema de las n-reinas, explicado posteriormente), o a partir de cualquier otro método que permita obtener un "vecino" propiamente dicho. Un vecino se define como una solución próxima a la que sirve de base para construir el vecindario.
También está el concepto de criterio de aspiración. Este criterio determina que si una solución lo cumple y está la primera en la lista ordenada del vecindario, aunque la solución esté marcada como Tabú, es escogida como solución provisional. Existen múltiples criterios de aspiración:

1. **Aspiración por Defecto**: si todos los movimientos posibles son clasificados como tabú, entonces se selecciona el movimiento "menos tabú", es decir, si el movimiento m_1 está penalizado en la matriz tabú durante 2 iteraciones más y m_2 está penalizado durante 1, m_2 es menos tabú que m_1.
2. **Aspiración por Objetivo**: una aspiración de movimiento se satisface, permitiendo que un movimiento x sea un candidato para seleccionarse si, por ejemplo, F(x) < menor coste (en un problema de minimización). Este criterio es el que usaré en el algoritmo implementado.
3. **Aspiración por Dirección de Búsqueda**: un atributo de aspiración para la solución "s" se satisface si la dirección en "s" proporciona una mejora y el actual movimiento es un movimiento de mejora. Entonces, "s" pasa a ser considerado como candidato.

En concreto, el modo de desarrollarse de este algoritmo es el siguiente:

En cada iteración, se genera un vecindario del tamaño que quiera a partir de una solución provisional obtenida en la iteración anterior. Después, se ordena dicho vecindario en función de la función objetivo de cada solución (como es un algoritmo de minimización, de menor a mayor función objetivo), y se escoge la primera solución en dicho vecindario que no esté marcada como Tabú o que, a pesar de estar marcada como Tabú, cumpla con el criterio de aspiración, en el caso de que queramos que nuestro algoritmo de Búsqueda Tabú cuente con uno (es opcional, del mismo modo que la memoria frecuente). Dicha solución pasará a ser marcada como Tabú tantas iteraciones como la Tenencia Tabú, y a partir de ella generaremos el vecindario y repetiremos el proceso hasta llegar al número de iteraciones estipulado (salvo si hemos alcanzado una solución considerada como óptima, en cuyo caso el algoritmo acaba al instante), quedándonos con la solución a la que hemos llegado en ese momento y con su función objetivo.

3.2. Aplicación al problema de las n-reinas

A continuación explicaré más detalladamente el algoritmo con un ejemplo, una solución al problema de las n-reinas (en concreto, 8-reinas) aplicando la Búsqueda Tabú. Se explicará partiendo del caso de que no se usa la memoria frecuente. Esto es así porque no uso dicha memoria en mi algoritmo y por tanto lo considero innecesario y engorroso para el lector. La solución es la siguiente:

Paso 1: se escoge una solución inicial, que depende de la intuición del desarrollador del algoritmo; por ejemplo, podría ser un vector de tamaño el número de columnas del tablero (el número de reinas) en que cada posición se corresponde con una fila, sumando dos filas (con el módulo 8) a cada columna. La solución inicial quedaría así: [1,3,5,7,2,4,6,8]. Esta solución inicial pasa a ser la solución actual.
También se inicializan las variables correspondientes al número de iteraciones, a la tenencia tabú (el número de iteraciones que una solución escogida de entre el vecindario queda prohibida, salvo que dicha solución cumpla con un criterio de aspiración) y al tamaño del vecindario. En este caso, mostraremos la evolución del vecindario, solución escogida y matriz tabú para la resolución de dicho problema durante 3 iteraciones con un tamaño de vecindario de 4 y una tenencia tabú de 8.

Paso 2: se inicializa lo que en este caso sería la matriz tabú, de tamaño 8x8 (número de reinas x número de reinas), con todo ceros. La idea es almacenar en la matriz triangular superior menos la diagonal la memoria reciente, y dejar la matriz triangular inferior menos la diagonal, la frecuente (no utilizada en el ejemplo por simplicidad).

Matriz tabú:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]

Iteración 1:

Paso 3: se copia la solución actual en la llamada solución transitoria para a partir de ella obtener el número de vecinos estipulado, ya que si cambiase la solución actual con cada vecino estipulado, dicho vecino lo sería del vecino anterior, y no de la solución obtenida como ganadora de la iteración anterior, que es lo que queremos).

Solución transitoria:
[1, 3, 5, 7, 2, 4, 6, 8]

Paso 4: se genera el vecindario cuyo tamaño ha sido previamente estipulado a partir de la solución transitoria (hallando para cada vecino la función objetivo, que varía según el problema a resolver; en este caso, se corresponde con el número de conflictos entre todas las reinas de dicho vecino). Acto seguido, se ordenan los vecinos en consonancia a su función objetivo, de menor a mayor función objetivo, y nos quedamos con el primero, salvo que esté marcado como tabú excepto si cumple con el criterio de aspiración. Si éste no satisface los requisitos, pasará a considerarse el siguiente, y así hasta recorrer todo el vecindario en caso necesario.

Cabe destacar que es una práctica recomendable escoger una tenencia tabú inferior al tamaño de la memoria reciente (salvo en el caso de que escojamos el criterio de aspiración por Defecto, en que nos es indiferente); si no, existe la posibilidad de que el algoritmo se atasque en dicha iteración (cuando ningún vecino cumpla el criterio de aspiración). En nuestro caso, debe ser inferior a $(8^2-8)/2$. En general, de ser la matriz tabú cuadrada, debe ser inferior a (tamaño_matriz_tabú^2 – (tamaño_matriz_tabú = número de elementos de la diagonal principal)) / 2. Dicha cota depende de la forma de la matriz tabú, ya que en otros casos esta fórmula no es aplicable.

Vecindario:
[[Reinas = [1, 3, 5, 7, 2, 8, 6, 4], Índice menor = 5, Índice mayor = 7, Función objetivo = 1],
[Reinas = [7, 3, 5, 1, 2, 4, 6, 8], Índice menor = 0, Índice mayor = 3, Función objetivo = 3],
[Reinas = [1, 3, 5, 4, 2, 7, 6, 8], Índice menor = 3, Índice mayor = 5, Función objetivo = 6],
[Reinas = [2, 3, 5, 7, 1, 4, 6, 8], Índice menor = 0, Índice mayor = 4, Función objetivo = 1]]

Vecindario ordenado:
[[Reinas = [1, 3, 5, 7, 2, 8, 6, 4], Índice menor = 5, Índice mayor = 7, Función objetivo = 1],
[Reinas = [2, 3, 5, 7, 1, 4, 6, 8], Índice menor = 0, Índice mayor = 4, Función objetivo = 1],
[Reinas = [7, 3, 5, 1, 2, 4, 6, 8], Índice menor = 0, Índice mayor = 3, Función objetivo = 3],
[Reinas = [1, 3, 5, 4, 2, 7, 6, 8], Índice menor = 3, Índice mayor = 5, Función objetivo = 6]]

La primera solución del vecindario ordenado satisface los requisitos, pues no está marcada como tabú.

Paso 5: se marca la posición correspondiente a dicha solución "ganadora" como tabú, con un entero no negativo igual a la tenencia tabú.

Matriz tabú antes de actualizarse:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 8]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]

Paso 6: se actualiza la matriz tabú, decrementando todos los elementos positivos de la memoria reciente en una unidad.

Matriz tabú actualizada:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 7]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]

Paso 7: se repite del paso 3 hasta este paso hasta que se hayan alcanzado o el número de iteraciones estipuladas o la solución óptima.

Estamos en la primera iteración y no hemos alcanzado la solución óptima, con lo que el algoritmo sigue iterando.

Iteración 2:

Paso 3:

Solución transitoria:
[1, 3, 5, 7, 2, 8, 6, 4]

Paso 4:

Vecindario:
[[Reinas = [1, 3, 5, 2, 7, 8, 6, 4], Índice menor = 3, Índice mayor = 4, Función objetivo = 4],
[Reinas = [1, 3, 5, 7, 8, 2, 6, 4], Índice menor = 4, Índice mayor = 5, Función objetivo = 4],
[Reinas = [1, 3, 5, 7, 2, 8, 4, 6], Índice menor = 6, Índice mayor = 7, Función objetivo = 4],
[Reinas = [1, 3, 4, 7, 2, 8, 6, 5], Índice menor = 2, Índice mayor = 7, Función objetivo = 4]]

Vecindario ordenado:
[[Reinas = [1, 3, 5, 2, 7, 8, 6, 4], Índice menor = 3, Índice mayor = 4, Función objetivo = 4],
[Reinas = [1, 3, 5, 7, 8, 2, 6, 4], Índice menor = 4, Índice mayor = 5, Función objetivo = 4],
[Reinas = [1, 3, 5, 7, 2, 8, 4, 6], Índice menor = 6, Índice mayor = 7, Función objetivo = 4],
[Reinas = [1, 3, 4, 7, 2, 8, 6, 5], Índice menor = 2, Índice mayor = 7, Función objetivo = 4]]

La primera solución del vecindario ordenado satisface los requisitos y es escogida, pues no está marcada como tabú.

Paso 5:
Matriz tabú antes de actualizarse:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 8, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 7]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]

Paso 6:

Matriz tabú actualizada:
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 7, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 6]
[0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0]

Paso 7:

Estamos en la segunda iteración y no hemos alcanzado la solución óptima, con lo que el algoritmo sigue iterando.

Iteración 3:

Paso 3:

Solución transitoria:
[1, 3, 5, 2, 7, 8, 6, 4]

Paso 4:

Vecindario:
[[Reinas = [1, 3, 6, 2, 7, 8, 5, 4], Índice menor = 2, Índice mayor = 6, Función objetivo = 5],
[Reinas = [1, 8, 5, 2, 7, 3, 6, 4], Índice menor = 1, Índice mayor = 5, Función objetivo = 2],
[Reinas = [1, 3, 5, 7, 2, 8, 6, 4], Índice menor = 3, Índice mayor = 4, Función objetivo = 1],
[Reinas = [1, 3, 2, 5, 7, 8, 6, 4], Índice menor = 2, Índice mayor = 3, Función objetivo = 5]]

Vecindario ordenado:
[[Reinas = [1, 3, 5, 7, 2, 8, 6, 4], Índice menor = 3, Índice mayor = 4, Función objetivo = 1],
[Reinas = [1, 8, 5, 2, 7, 3, 6, 4], Índice menor = 1, Índice mayor = 5, Función objetivo = 2],
[Reinas = [1, 3, 6, 2, 7, 8, 5, 4], Índice menor = 2, Índice mayor = 6, Función objetivo = 5],
[Reinas = [1, 3, 2, 5, 7, 8, 6, 4], Índice menor = 2, Índice mayor = 3, Función objetivo = 5]]

A pesar de estar marcada como tabú, como la primera solución del vecindario ordenado satisface el criterio de aspiración (mejora la función objetivo a minimizar respecto a la solución escogida en la iteración anterior), ésta es escogida, y es con la que nos quedamos.

Paso 7:
Estamos en la tercera y última iteración, y aunque no hemos alcanzado la solución óptima, el algoritmo acaba.

Paso 8: alcanzado el número de iteraciones estipulado, salvo que antes hayamos alcanzado la solución óptima (en cuyo caso el algoritmo terminará en dicho momento), nos quedamos con la función objetivo correspondiente a la solución actual, y esa será la solución.

Solución final:

La solución escogida como final para este algoritmo con los parámetros mencionados anteriormente sería: [1, 3, 5, 7, 2, 8, 6, 4]. Esto es así ya que es la que minimiza la función objetivo (número de colisiones entre reinas en el tablero) para el número de iteraciones y demás parámetros dados. Con más iteraciones y estos parámetros, la solución que hace que el número de colisiones entre reinas sea igual a cero se alcanza. Es recomendable por tanto ajustar la tenencia tabú y el tamaño del vecindario y el número de iteraciones (ambos parámetros se han considerado tan reducidos por simplicidad) al problema que estemos tratando, pues de ello dependerá obtener un resultado óptimo.

Cabe mencionar que este algoritmo es capaz de procesar este problema con un gran número de reinas (del orden de 60-70) en un tiempo ínfimo (segundos), pues el problema de las n-reinas es un problema difícil (pertenece a la clase NP).

Capítulo 4

Aplicación al análisis de prestaciones de Redes de Petri temporizadas y decisiones de implementación

4.1. Breve reseña de Redes de Petri

Una Red de Petri [Mur89] es una 4-tupla N = {P,T,Pre,Post}, donde P y T son conjuntos disjuntos no vacíos de lugares y transiciones, y Pre (Post) son las matrices de enteros no negativos de pre-(post-)incidencia de tamaño |P|x|T|. Las redes Ordinarias son aquellas cuyos arcos tienen peso 1. La matriz de incidencia de una Red de Petri se define como C = Post – Pre.
Un vector de marcado o marcado es un vector de tamaño el número de lugares que asigna un entero no negativo a cada lugar correspondiente al número de marcas de dicho lugar. Un sistema de Redes de Petri, o Red de Petri Marcada S = {N,m0}, es una red de Petri N con un marcado inicial m0.
La ecuación de estado lineal (o fundamental) de una Red, se define como $m = m0 + C * \sigma$. Dos transiciones t y t´ están en conflicto estructural si comparten, como mínimo, un lugar de entrada. Dos transiciones t y t´ están en conflicto efectivo para un marcado m si están en conflicto estructural y están ambas sensibilizadas en m. Dos transiciones t y t´ están en conflicto equitativo si las columnas de la matriz PRE asociadas a dichas transiciones son iguales. La relación de conflicto libre es una relación de equivalencia que parte el conjunto de particiones en clases de equivalencia ECS, llamadas conjuntos de conflicto equitativo. Las transiciones pertenecientes a un conflicto libre dado están en conflicto de libre elección extendido.
Una Red de Petri se dice que está fuertemente conexa si hay un camino dirigido uniendo cualquier par de nodos del grafo. Una máquina de estados es un tipo particular de red de Petri ordinaria donde cada transición tiene exactamente un arco de entrada y exactamente un arco de salida.
Un p-semiflujo es un vector de enteros no negativos que es el anulador izquierdo de la matriz de incidencia de la Red, $y^T * C = 0$. Un p-semiflujo implica una ley de conservación de marcas independiente de cualquier disparo de transiciones. Un t-semiflujo es un vector de enteros no negativos que es anulador derecho de la matriz de incidencia de la Red, $C * x = 0$. Un p-(o -t)semiflujo es mínimo cuando no es combinación lineal de ningún otro p-(-t)semiflujo.
Las redes de Petri tienen propiedades funcionales (alcanzabilidad, reversivilidad, ausencia de bloqueos, vivacidad, acotación y exclusión mutua) y no funcionales, que son las que nos interesan en nuestro caso, particularmente el "throughput" o su inverso, el tiempo de ciclo.

4.2. El problema min-max de Bernardi y Campos

En el artículo [ZRS01] se menciona que para resolver el problema presentado en dicho artículo, de características similares al que nos concierne, se utiliza la heurística de Simulated Annealing, aunque sugiere que se podría haber utilizado la de Búsqueda Tabú, lo que nos da pie a pensar que en nuestro problema también puede encajar bien la Búsqueda Tabú, y es por eso que nos decantamos por dicha heurística.

El problema a resolver es el problema min-max definido por Bernardi y Campos [BC13], consistente en hallar una cota inferior del tiempo de ciclo para una transición de referencia (que llamaremos transición t_i). El problema se define como sigue:

$\Gamma(t_i) >= \Gamma^{lb}(t_i) = \min \quad \max y * Pre * D^{ti}.$
$v_{ti} \in dom_v \quad y \in dom_y$

Sujeto a $dom_y : \{y^T * C = 0, y * m0 = 1, y>=0\}$
$dom_v : \{R * v_{ti} = 0, C * v_{ti} = 0, v_{ti} >=0, v_{ti}(t_i) = 1\}$

t_i es la transición sobre la que queremos hallar el tiempo de ciclo.
y es un p-semiflujo del sistema.
v_{ti} es el vector de ratios de visita normalizado para la transición t_i.
$D_{ti.} = \delta \Theta v_{ti}$ (producto de vectores por componentes) es el vector de demandas de servicio de transiciones medio.
Pre es la matriz de Pre-incidencia del sistema.
C es la matriz de incidencia del sistema.
R es la matriz de enrutado del sistema.
m0 es el marcado inicial del sistema.

La función objetivo de este problema es cuadrática, porque en ella aparecen multiplicadas las variables del vector y por las del vector v (que está implícito dentro de D). Por ello, no pueden utilizarse algoritmos de resolución de problemas de programación lineal.

A continuación se explican las decisiones de implementación. Es importante decir que se ha optado por construir una solución para problemas de optimización no exponenciales por los problemas de tiempo durante la ejecución. En general, el número de p-semiflujos (o t-semiflujos) puede crecer exponencialmente con el tamaño de una red (expresado el tamaño con el número de lugares y transiciones). Esta es una limitación de índole teórico, y podría suponer también un problema práctico en algunos casos. No obstante, en la práctica, es frecuente manejar modelos en los que el número de p-semiflujos (o t-semiflujos) no crece de forma tan rápida, con lo que la utilidad práctica se mantiene.
Además, el problema resuelto se puede considerar un problema min-max, pero no es el min-max descrito anteriormente, pues el simple hecho de entenderlo requiere un tiempo considerable del que no he dispuesto en la elaboración del proyecto.

4.3. Decisiones de implementación

En la figura 4.1 se muestra el diagrama de clases del algoritmo implementado

La clase "TabuSearch" la he intentado programar de la forma más genérica posible, para poder abarcar problemas de diversa índole, manteniendo así el esquema genérico de casi cualquier Búsqueda Tabú, tanto si el problema a resolver es de minimización como si es de maximización. Además de ello es una clase abstracta, de la que se heredan algunos métodos y atributos, obligando así al arquitecto del software a programar en su clase hija el programa que va a resolver el problema en cuestión valiéndose de los atributos y métodos heredados (algunos de estos métodos son abstractos, y por tanto es obligatorio implementarlos) y de su esquema ya mencionado.

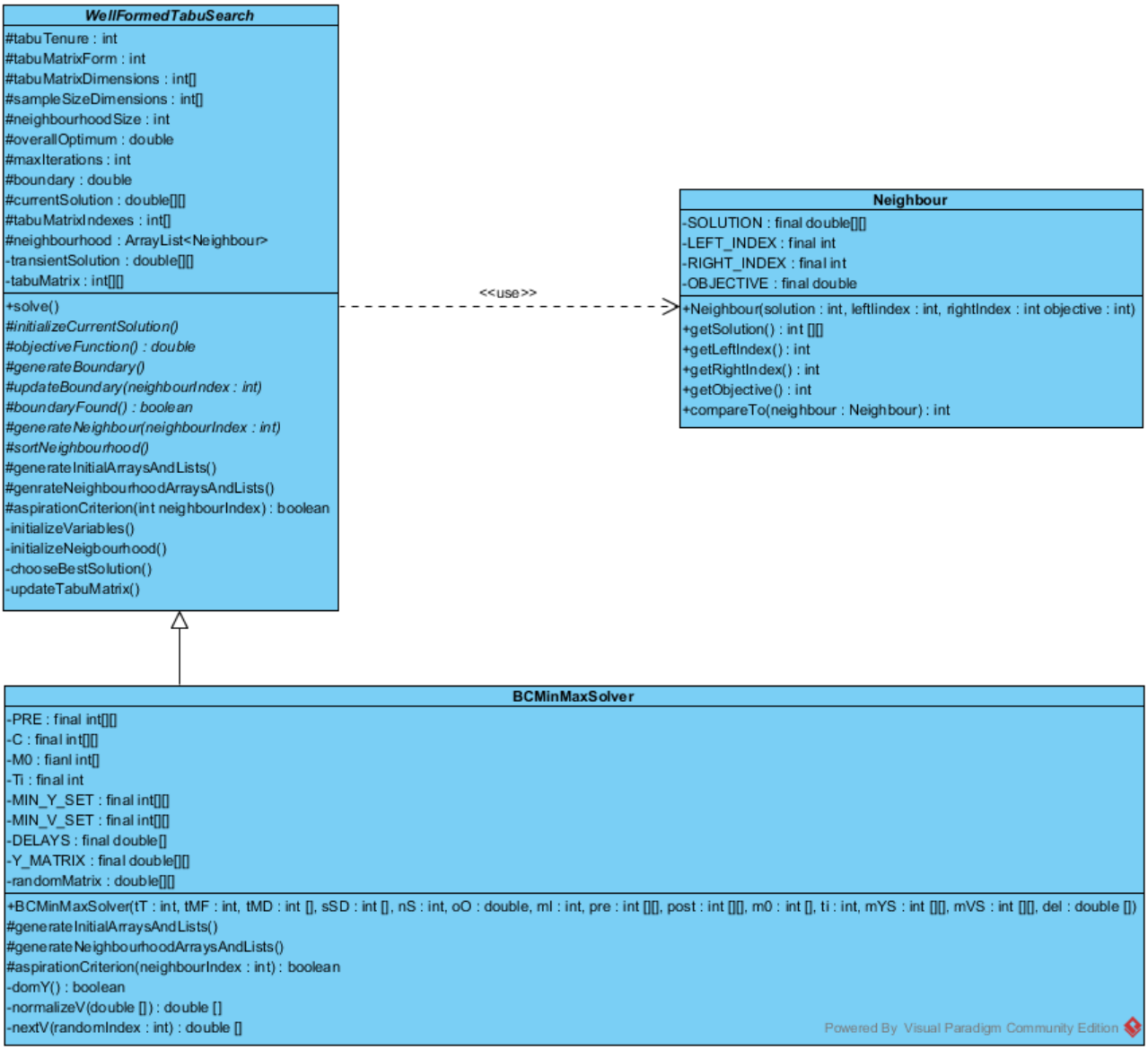

Figura 4.1: Diagrama de Clases del algoritmo implementado

El método principal, abstracto, de la clase "Tabu Search", que es el método público "solve()", y que devuelve un dato real de tipo "double", se encarga de implementar el esquema genérico de la Búsqueda Tabú. Dicho método procede como sigue:

1) Se inicializan las variables: el contador de iteraciones, la solución actual, la solución transitoria y los índices de la matriz tabú. Además, se crea la matriz tabú con todas sus componentes a cero a partir de las dimensiones definidas previamente y se genera la cota de la que partiremos (0 en nuestro caso, pues el criterio es de minimización y una cota no puede ser negativa). También se genera la matriz Y_MATRIX, compuesta de un subconjunto del conjunto de ys mínimos, seleccionando aquéllos que cumplen con su respectivo dominio. De esta matriz se sacarán todas las partes de solución correspondientes a ys útiles.

2) Se copia la solución actual en la solución transitoria. Ambas soluciones son matrices formadas por dos vectores, el primero se corresponde con el y mínimo que cumple su dominio respectivo, y el segundo con el v normalizado que es combinación lineal de otros vs mínimos. El tamaño de cada solución se corresponde con el del mayor de los tamaños de cualquier vector y o de cualquier vector v (todos los vectores y e v tienen el mismo tamaño, respectivamente). La solución transitoria es la solución a partir de la cuál generamos todo el vecindario, y la solución actual es la solución que estamos considerando en el momento actual. Así pues, se copia la solución actual inicial o la solución actual ganadora de entre cada vecindario en la solución transitoria para que todos los vecinos sean vecinos de uno inicial, y no el 2º del 1º, el 3º del 2º, etc.

3) Se genera el vecindario. Para ello, se llama a la función "generateNeighbourhood()".

4) Se ordena el vecindario. Dependiendo de si se quiere maximizar o minimizar, se ordenará de mayor a menor función objetivo, o de menor a mayor función objetivo, respectivamente. En el caso del algoritmo implementado, se ordena de mayor a menor función objetivo, pues se quiere minimizar.

5) Se escoge la mejor solución, siguiendo los criterios de la Búsqueda Tabú, es decir, la primera de entre el vecindario ordenado que cumpla el criterio de aspiración (por objetivo, en este caso) o que no lo cumpla y no esté marcada como tabú. Marcamos como tabú dicha solución y actualizamos la mejor cota hallada hasta el momento, en caso de que nuestra cota actual sea mayor que la mejor cota hallada hasta el momento. En caso contrario, no actualizamos la cota.

6) Se actualiza la matriz tabú, decrementando en una unidad cada posición de la memoria reciente mayor que 0 (en caso de que utilizásemos memoria frecuente, tendríamos que incrementar en una unidad la posición de dicha memoria correspondiente a la solución elegida). En la actual implementación, la forma de la matriz tabú admite dos tipos, seleccionados en función de una variable booleana ("tabuMatrixForm"): una matriz cuadrada pensada para que la matriz triangular superior menos la diagonal corresponda con la memoria reciente y la matriz triangular inferior menos la diagonal corresponda con la memoria frecuente; y una matriz rectangular de dos filas, pensada para que la memoria reciente se corresponda con la primera fila y así dejar espacio para la memoria frecuente en la segunda fila. La lista tabú puede tener muchas dimensiones, pero yo considero que la matriz bidimensional es la que más sentido cobra en la mayoría de problemas, y es por esto por lo que me he decantado por ella en la implementación. Al principio, en el algoritmo, la matriz tabú tenía una única forma, la cuadrada, pero por razones de adaptación al problema a resolver, decidí añadir la forma rectangular de dos filas.

7) Se incrementa el contador de iteraciones en una unidad. Ir a 2) hasta que el contador de iteraciones alcance el número máximo de iteraciones fijado o se obtenga la cota deseada, de existir una (en el caso de este algoritmo, que es de minimización, ponemos una cota igual a 0, pues ninguna cota real alcanzará este valor, ya que todas las cotas mínimas posibles son mayores que 0).

8) Una vez salgamos del bucle, devolvemos como resultado la mínima cota alcanzada hasta el momento.

El método privado "initializeVariables()", que no devuelve ningún valor, procede como se indica a continuación:

1) Se inicializan la solución actual y la solución transitoria con las dimensiones definidas de antemano, con todas sus componentes a 0. En este caso, dichas dimensiones son 2 * Máx(tamaño de cualquier vector y mínimo, tamaño de cualquier vector v mínimo). He decidido que las soluciones sean un par de valores y,v debido a que son los únicos valores variables necesarios para la obtención de la función objetivo.

2) Se inicializa el vector que contendrá los índices correspondientes a cada vecino con tamaño 2 y sus componentes a 0.

3) Se generan los arrays y listas auxiliares iniciales necesarios. En este caso, se crea la matriz de ys mínimos que satisfacen su dominio y se redefinen las dimensiones de la matriz tabú, en el caso de este algoritmo, a 2 * el tamaño de la matriz de ys mínimos creada previamente.

4) Se genera la cota de partida. Como nuestro algoritmo es minimizador, ponemos una cota muy alta, ("Double.MAX_VALUE") en este caso. Esta cota representa el menor valor de entre todas las funciones objetivo correspondientes a la solución ganadora de cada iteración. En cada iteración, esta cota se actualizará con el valor correspondiente.

5) Se inicializa la matriz tabú con las dimensiones definidas previamente y todas sus componentes a 0.

El método privado "generateNeighbourhood()", que no devuelve ningún valor, procede como se indica a continuación. Primero de todo, se generan los arrays y listas auxiliares que necesitemos. En el caso de este algoritmo, esta función genera una matriz de números aleatorios cuyo valor es un real entre 0 y 1. Las dimensiones de esta matriz son: número de vecinos * número de vs mínimos. Se crea un arraylist vacío de objetos de la clase "Neighbour" (explicada más adelante). Posteriormente, comienza el bucle de generación del vecindario. Este bucle itera un número de veces igual al número de vecinos previamente definido. En cada iteración, realiza las siguientes operaciones:

3.1) Se copia la solución transitoria en la solución actual, para así generar todos los vecinos de una única solución global para todo el vecindario y no generar cada vez un vecino del anterior elegido.
3.2) Se generan las componentes del vecino considerado, pasándole como parámetro el número de vecino correspondiente (el índice dentro del bucle de generación de vecinos).
3.3) Se obtiene la función objetivo a partir del método "objectiveFunction()" y se guarda en una variable de tipo "double".
3.4) Se genera un objeto de la clase "Neighbour" a partir de la solución actual, los índices correspondientes a la solución actual dentro de la matriz tabú y la variable que contiene la función objetivo.
3.5) Se añade el vecino de la clase "Neighbour" al arraylist que representa el vecindario.

El método privado "chooseBestSolution()", que no devuelve ningún valor, recorre el vecindario ya ordenado de mayor a menor cota en este caso, escogiendo el primer vecino que cumpla el criterio de aspiración (que mejore la cota ofreciendo una menor) o que no lo cumpla pero no esté marcado como tabú. Es esa solución escogida la que se considera la nueva solución actual, pasando a ser marcada además como tabú. También se actualiza la cota menor obtenida hasta el momento en caso necesario.

El método privado "updateTabuMatrix()", que no devuelve ningún valor, decide primero si la forma de la matriz tabú es de "tipo 1" o de "tipo 2", y la actualiza en consecuencia, decrementando en una unidad, en ambos casos, todas sus componentes positivas correspondientes a la memoria reciente.

Por otra parte, están los métodos protegidos abstractos "initializeCurrentSolution()", "objectiveFunction()", "generateBoundary()", "updateBoundary(int)", "boundaryFound()", "generateNeighbour(int)", "sortNeighbourhood()", y los protegidos "generateIninitalArraysAndLists()", "generateNeighbourhoodArraysAndLists()" y "aspirationCriterion(int)", que en este algoritmo, son redefinidos en la clase hija "BCMinMaxSolver".

Capítulo 5

Integración en PeabraiN

PeabraiN, acrónimo de "Performance Estimation bAsed (on) Bounds (and) Resource optimisAtIon (for Petri) Nets", está hecha a partir de un conjunto de módulos conforme a los módulos de la herramienta PIPE [BLP+07]. En esta sección, primeramente introducimos su arquitectura con detalle y después las características que provee la herramienta.

5.1. Arquitectura

PeabraiN está programado con Java e integrado en PIPE. Aparte de las dependencias de la librería PIPE, depende de otras librerías para llevar a cabo su funcionalidad. Estas librerías están relacionadas con operaciones computacionales en matrices (JAMA), funciones de distribución de probabilidad (SSJ), y la interfaz del solucionador de problemas lineales para Java (Java ILP).

PeabraiN ha sido diseñado siguiendo el patrón de diseño Model-View-Controller (MVC), y su arquitectura está compuesta de tres capas. Cada capa corresponde con una componente en MVC. Las clases de la capa de datos representan la información necesaria para ejecutar las funcionalidades proporcionadas, en términos de problemas de Programación Lineal (LP). Por ejemplo, las restricciones relacionadas con el problema LP, la definición de función de optimización, tipos de variables, etc. El patrón estructural Facade ha sido seguido en esta capa para manejar y minimizar la complejidad de las clases. La capa intermedia contiene las clases que implementan los algoritmos y las características que provee PeabraiN. Finalmente, la capa GUI envuelve las clases relacionadas con interfaces gráficos para acumular/mostrar de/al usuario datos de entrada y resultados de las funcionalidades.

PIPE permite ser extendida a través de módulos que deben implementar la interfaz "Module". Cabe notar que la integración sigue una arquitectura abierta, de modo que las clases de la capa superior se comunican con las clases de la capa inferior. La clase "PetriNetModel", que es una representación matricial de la red de Petri actual en formato PNML [HKP+10], es creado por cada uno de los módulos de PeabraiN.

A continuación, se presenta un diagrama que muestra la integración de PeabraiN en PIPE (Figura 5.1). Este diagrama muestra los módulos y clases existentes en PIPE y PeabraiN, desglosados por capas (Modelo, Vista y Controlador). Además, muestra gráficamente cómo están interrelacionados.

Seguidamente, en la Figura 5.2, se presenta el diagrama de clases UML de PeabraiN.

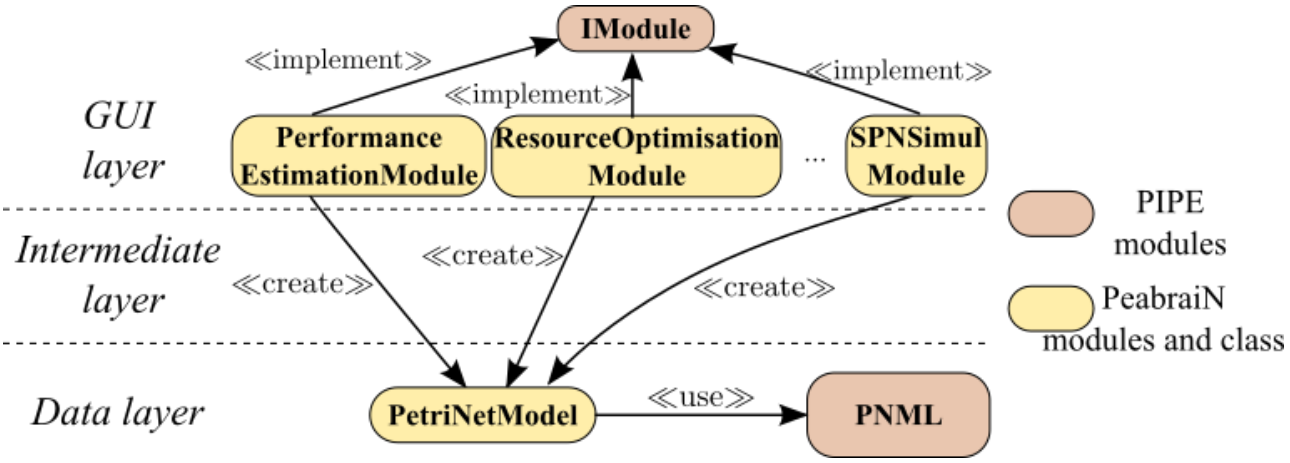

Figura 5.1: Integración de PeabraiN en PIPE

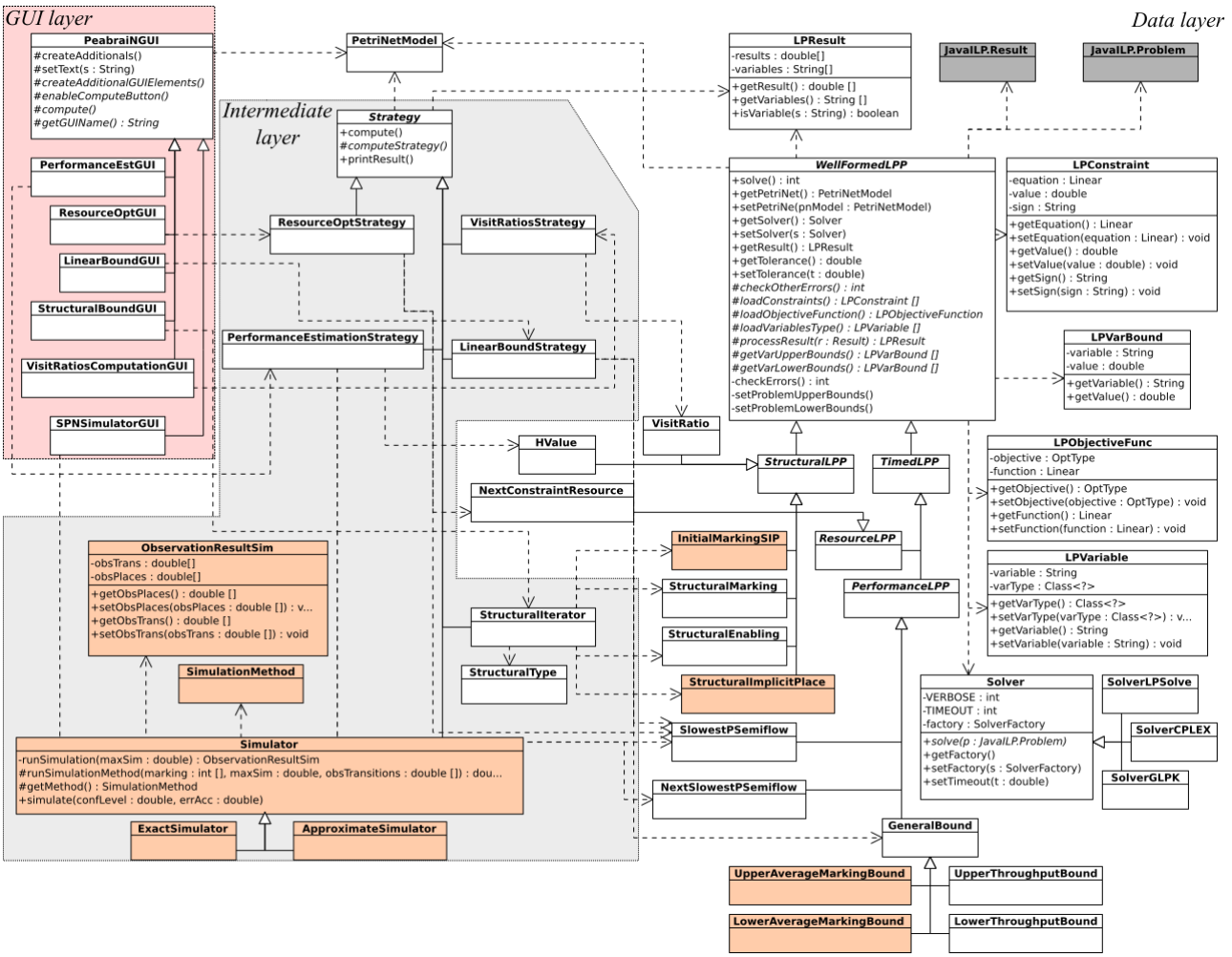

Figura 5.2: Diagrama de Clases UML de Peabrain

5.2. Integración de la Búsqueda Tabú en PeabraiN

Una vez explicada la herramienta en que se introduce el algoritmo de Búsqueda Tabú, es importante mencionar que dicha herramienta carece de algoritmos heurísticos para problemas de complejidad no lineal (cuadráticos, en nuestro caso de estudio), razón por la cual introducimos esta técnica en la herramienta.

PeabraiN fue diseñada siguiendo también una arquitectura modular para facilitar su expansión. Esta arquitectura hace más fácil añadir nuevas funcionalidades. De hecho, este es el mayor beneficio de esta arquitectura: permite una implementación rápida de un prototipo de cualquier algoritmo que trata con Redes de Petri y problemas de programación matemática.

A continuación explicaremos cómo PeabraiN puede ser fácilmente expandido con el ejemplo de TabuSearch.

Lo primero que hay que hacer es crear una clase "WellFormedTabuSearch", y rellenar los métodos apropiadamente. Recordemos que la Red de Petri está representada en forma matricial mediante la clase "PetriNetModel", y los métodos de TabuSearch representan las variables, restricciones, función objetivo, etc de "WellFormedTabuSearch". Tras esto, una capa intermedia debe ser creada "TabuSearchStrategy", expandiendo la clase abstracta "Strategy". Esta clase debe crear y comunicarse apropiadamente con la nueva clase creada ("TabuSearch"). Después de crearla, es necesario rellenar los argumentos necesitados por "TabuSearch" y entonces invocar al método "solve()". Finalmente, una clase debería ser creada en la capa GUI como una extensión y por la clase abstracta "PeabraiNGUI" (supongamos que esta clase se llama "TabuSearchGUI"). Como último paso, la integración con PIPE debe ser llevada a cabo. Así, es creado un nuevo módulo llamado "TabuSearchModule" implementando "Module" y usando "PetriNetModel".
Una vez que el "TabuSearchModule" es instanciado, crea un objeto "PetriNetModel", tomando como parámetro el PNML actual fijado por PIPE. Después, se crea una "TabuSearchStrategy" y una "TabuSearchGUI", que toma como parámetro de entrada el objeto recientemente creado "PetriNetModel". La GUI también recoge los datos de entrada provistos por el usuario ("ti", "tabuTenure", "neighbourhoodSize" y "maxIterations"), y entonces el método de Búsqueda Tabú es utilizado para dar una solución al problema min-max [BC13] sobre la Red de Petri considerada, obteniendo como resultado del cómputo la cota superior de throughput para dicho problema.

Capítulo 6

Evaluación del algoritmo

En este capítulo se muestran los resultados (cotas inferiores de tiempo de ciclo) de unas cuantas pruebas realizadas para la red mostrada a continuación, escogiendo como transición de referencia T0, 25 vecinos como tamaño de vecindario y haciendo la media de 100000 ejecuciones del algoritmo para el cálculo de resultados. Dichos resultados se aproximan bastante al resultado óptimo, obtenido mediante análisis analítico usando la herramienta GreatSPN [CFG+95]. Nótese que este resultado es el valor exacto calculado suponiendo tiempos exponenciales resolviendo la cadena de Markov en tiempo continuo subyacente.

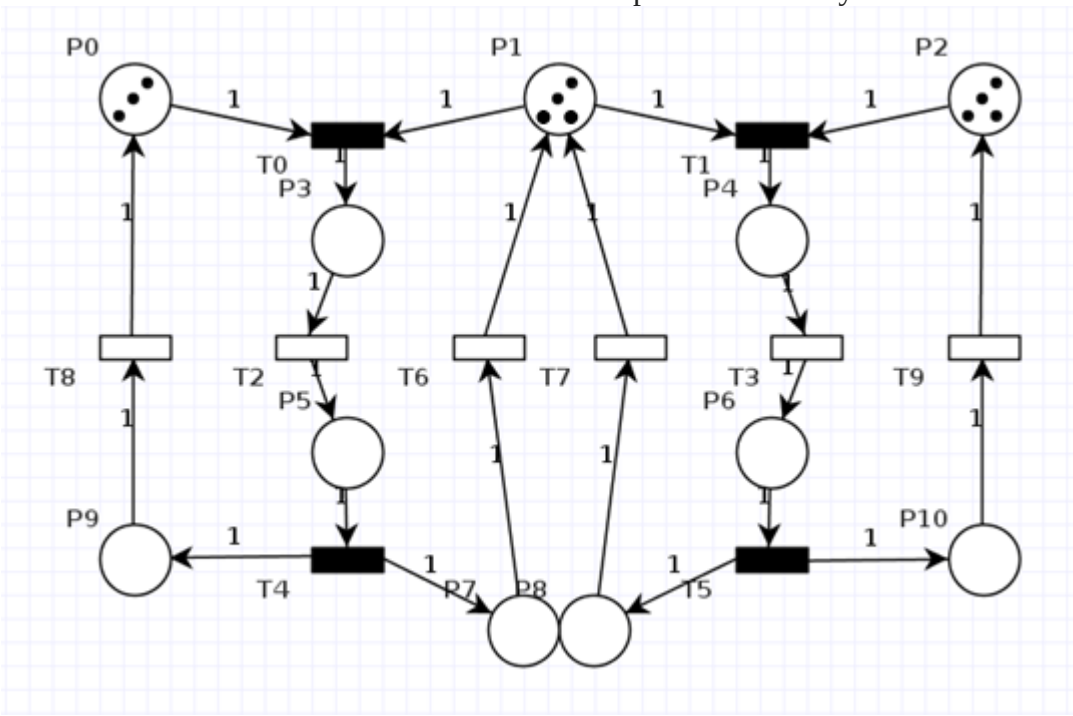

Figura 6.1: Red de prueba

Las transiciones dibujadas con un rectángulo negro son inmediatas, y las dibujadas con un rectángulo blanco, temporizadas. Todas las transiciones temporizadas tienen una ratio de disparo de 1.

Tenencia Tabú / Máx. Iteraciones	0	1	2
50	1,388515	1,387779	0,929408
100	1,248233	1,248122	0,783353
250	1,118675	1,118800	0,677581
500	1,045956	1,045793	0,666841
1000	0,986608	0,986761	0,666662

Tabla 6.1: Cotas inferiores de Tiempo de Ciclo correspondientes a la aplicación del algoritmo a la Figura 6.1

La Tabla 6.1 recoge los valores de la cota para el tiempo de ciclo de la transición T0 en el ejemplo de la Figura 6.1. El valor real de la transición T0 es **1,12**. Observamos que para 250 iteraciones y una Tenencia Tabú de 0 ó 1, se obtiene un resultado muy próximo a la cota exacta.

El algoritmo, tal y como se ha explicado anteriormente, escoge la menor de entre todas las cotas superiores halladas hasta el momento.

Al aumentar el tamaño del vecindario, la cota aumenta. Esto es así porque hay más vecinos de entre los cuales escoger la cota deseada y además se escoge el primero no tabú de entre una lista ordenada de mayor a menor cota.

Se han escogido 25 vecinos para la realización de las pruebas porque era el tamaño de vecindario que ofrecía resultados más aproximados al resultado exacto.

Como el algoritmo es principalmente minimizador, a más iteraciones, la cota mínima es menor.

Por último, la cota disminuye, salvo en casos anómalos, al aumentar la Tenencia Tabú. Esto se debe a que al aumentar dicho parámetro, cobra más importancia el criterio de aspiración (recordemos que nuestro criterio de aspiración consiste en escoger la primera solución cuya cota asociada sea menor a la cota mínima hallada hasta el momento). Por tanto, cuantas más veces se cumpla el criterio de aspiración, menor será la cota mínima.

Se ha escogido una Tenencia Tabú entre 0 y 2 ya que debido a como está implementado el algoritmo, es la forma de obtener resultados mejores. Esto es debido a las dimensiones de la Matriz Tabú.
Dichas dimensiones para este algoritmo con este caso de prueba son de 2x3 (número de memorias * número de ys mínimos satisfactorios). Si la Tenencia Tabú fuese superior a 2, igual a 3 por ejemplo, a la tercera iteración, no habría solución no Tabú posible y entraría en juego únicamente el criterio de Aspiración, reduciendo drásticamente la cota obtenida como resultado. Esto significaría que obtendríamos un resultado bastante más alejado del exacto.

Capítulo 7

Conclusiones

Respecto al cumplimiento de los objetivos de este proyecto, se ha logrado proponer una técnica de Búsqueda Tabú, aplicarla al problema en cuestión y, por último, integrar dicha técnica en la herramienta PeabraiN.

Por motivos de tiempo, el algoritmo tiene varias restricciones:

1. No resuelve el problema min-max, sino que únicamente se ha implementado un método heurístico para aproximar el valor del min-max. Cabe decir que esta restricción es debida a una complejidad importante del problema a resolver y no es fácilmente subsanable.
2. No logra resolver el asunto de crecimiento exponencial en los p-(t-) semiflujos, con lo que su aplicación está restringida a ciertas redes de Petri. Aún así, se presenta un algoritmo que puede servir de base para un trabajo futuro que logre resolver dicho problema.
3. No se ha implementado el concepto de Búsqueda Tabú de memoria frecuente, que podría mejorar las soluciones obtenidas, ni se ha probado a utilizar otro criterio de aspiración distinto al de aspiración por objetivo, o a no utilizar ninguno.

Para la elaboración del proyecto, he tenido que informarme sobre redes de Petri, que ya había dado en la carrera pero sin excesiva profundidad. Además, he tenido que informarme sobre tres conceptos que no había visto, como el de Búsqueda Tabú, el problema min-max [BC13], y también sobre la herramienta PeabraiN. También he tenido que aprender mucho sobre el lenguaje Java, utilizado en mi algoritmo, que a pesar de haber visto en la carrera, no había conseguido entender suficientemente para la realización de este proyecto.

Como trabajo futuro, plantearía resolver primeramente el problema de crecimiento exponencial, pues es la principal limitación que le veo al algoritmo. Después, intentaría utilizar las herramientas de la Búsqueda Tabú de memoria frecuente y distintos criterios de aspiración (incluso probar a no utilizar ninguno, o a utilizar el que sea a partir de cierta iteración del algoritmo). Por último, trataría de afinar correctamente el cálculo del min-max siguiendo estrictamente la definición de la función objetivo del problema, que se ha simplificado aquí para facilitar la implementación del algoritmo de Búsqueda Tabú. Supongo que podría desarrollarse alguna técnica en "dos pasos", todavía basada en Búsqueda Tabú, que calculase primero soluciones del problema de cálculo de P-semiflujo más lento de una red de Petri [CS92], tomado ese máximo en todos los valores "y" de su dominio para distintos valores del vector "v", y luego, aplicar de nuevo una Búsqueda Tabú para calcular el mínimo de las soluciones de todos los problemas anteriores, haciendo variar el "v" en todos los valores de su dominio. Pero eso queda fuera de este proyecto por su dificultad teórica.

Respecto a lo que he aprendido, aparte de sobre lo que he tenido que informarme, he aprendido a mantener una comunicación fluida con mis directores vía correo electrónico o entrevistas, además de conseguir un hábito de trabajo superior al conseguido hasta la fecha. He aprendido también que el lenguaje Java, como todos los lenguajes, supongo, tiene unos cuantos entresijos que es conveniente tener claros antes de afrontar un proyecto de tal magnitud, pues tal desconocimiento que tenía me ha supuesto bastante retraso en el proyecto.

El principal problema que he tenido es que este es un problema abierto, no estudiado hasta la fecha, lo que junto a restricciones temporales ha supuesto la parcialidad en sue resolución.

Bibliografía

[BC13] Bernardi, S. & Campos, J. "A min-max problem for the computation of the cycle time lower bound in interval-based Time Petri Nets", IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 43, iss. 5, pp. 1167-1181, 2013.

[BLP+07] Bonet, P.; Lladó, C.; Puigjaner, R. & Knottenbelt, W. PIPE v2.5: A Petri Net Tool for Performance Modelling Proceedings of the 23rd Latin American Conference on Informatics (CLEI), 2007.

[CFG+95] Chiola, G.; Franceschinis, G.; Gaeta, R. & Ribaudo, M. GreatSPN 1.7: Graphical editor and analyzer for timed and stochastic Petri nets Perform. Eval., Elsevier Science Publishers B. V., 1995, vol. 24, pp. 47-68.G.

[CS92] Campos, J. & Silva, M. Structural Techniques and Performance Bounds of Stochastic Petri Net Models, 1992.

[Glo89] Glover, F. "Tabu Search -- Part 1", ORSA Journal on Computing, vol. 1, iss. 3, pp.190-206, 1989.

[GK03] Glover, F. & Kochenberger, G. Handbook of Metaheuristics. Kluwer Academic Publishers, 2003.

[HKP+10] Hillah, L.; Kordon, F.; Petrucci, L. & Trèves, N. PNML Framework: An Extendable Reference Implementation of the Petri Net Markup Language Applications and Theory of Petri Nets, Springer, 2010, 6128, 318-327.

[Meg87] Megiddo, N. On the complexity of linear programming Advances in Economic Theory: Fifth world congress, Cambridge University Press, 1987, pp. 225-268.

[Mur89] Murata, T. Petri Nets: Properties, Analysis and Applications Proceedings of the IEEE, 1989, 77, 541-580.

[Rod17] Rodríguez, R. J. A Petri Net Tool for Software Performance Estimation Based on Upper Throughput Bounds Automated Software Engineering, 24, iss. 1, pp. 73-99, 2017.

[ZRS01] Zimmermann, A.; Rodríguez, D. & Silva, M. A two phase optimization method for Petri net models and manufacturing systems. Journal of Intelligent Manufacturing, 2001, 12, 409-420.

ANEXO

Horas de trabajo

El trabajo se ha prolongado durante un año, controlando en este período de tiempo el tiempo empleado en cada tarea. En la figura Anexo.1 se muestra un Diagrama de Gantt que representa el esfuerzo dedicado desglosado por semanas y tareas.

El trabajo ha consistido en diferentes tareas: (i) obtener y organizar el material teórico, (ii) estudiar las Redes de Petri, (iii) estudiar la Búsqueda Tabú, (iv) estudiar la herramienta PeabraiN, (v) estudiar sobre el problema min-max [BC13], (vi) aprender a manejar la máquina virtual,
(vii) elaborar el pseudocódigo y el diagrama de clases, (viii) implementar la Búsqueda Tabú con ejemplos sencillos, (ix) profundizar en el lenguaje Java, (x) aplicar la Búsqueda Tabú al problema en cuestión, (xi) realizar la integración en PeabraiN, (xii) elaborar y analizar el resultado de las pruebas y (xiii) elaborar la memoria.

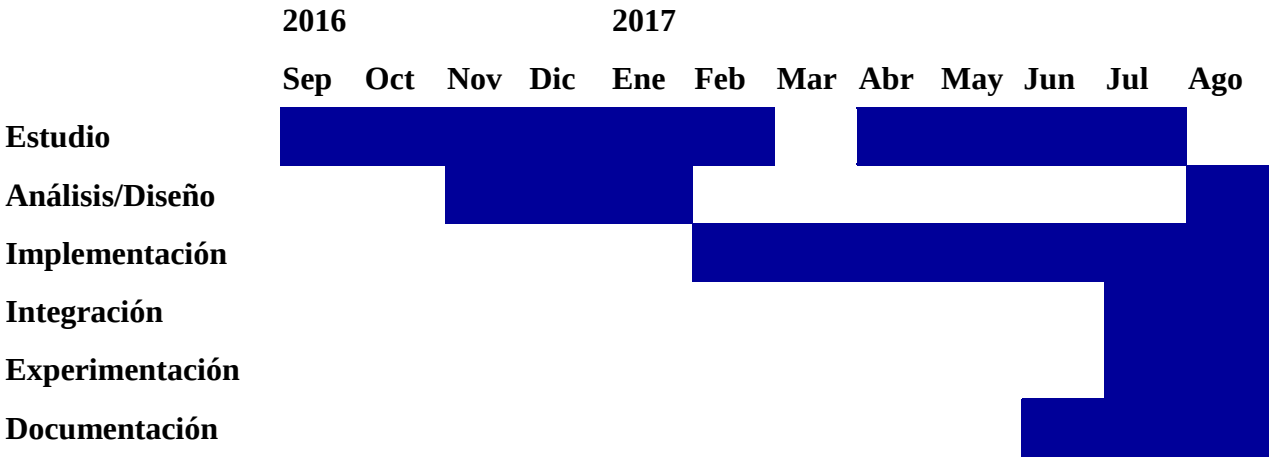

Figura ANEXO.1: Diagrama de Gantt con el esfuerzo invertido por meses

En total, se ha invertido en el proyecto un total de **472,5** horas. Para complementar la planificación del proyecto, se muestra en la figura Anexo.2 una visión de la cantidad de trabajo dedicada a cada una de las tareas.

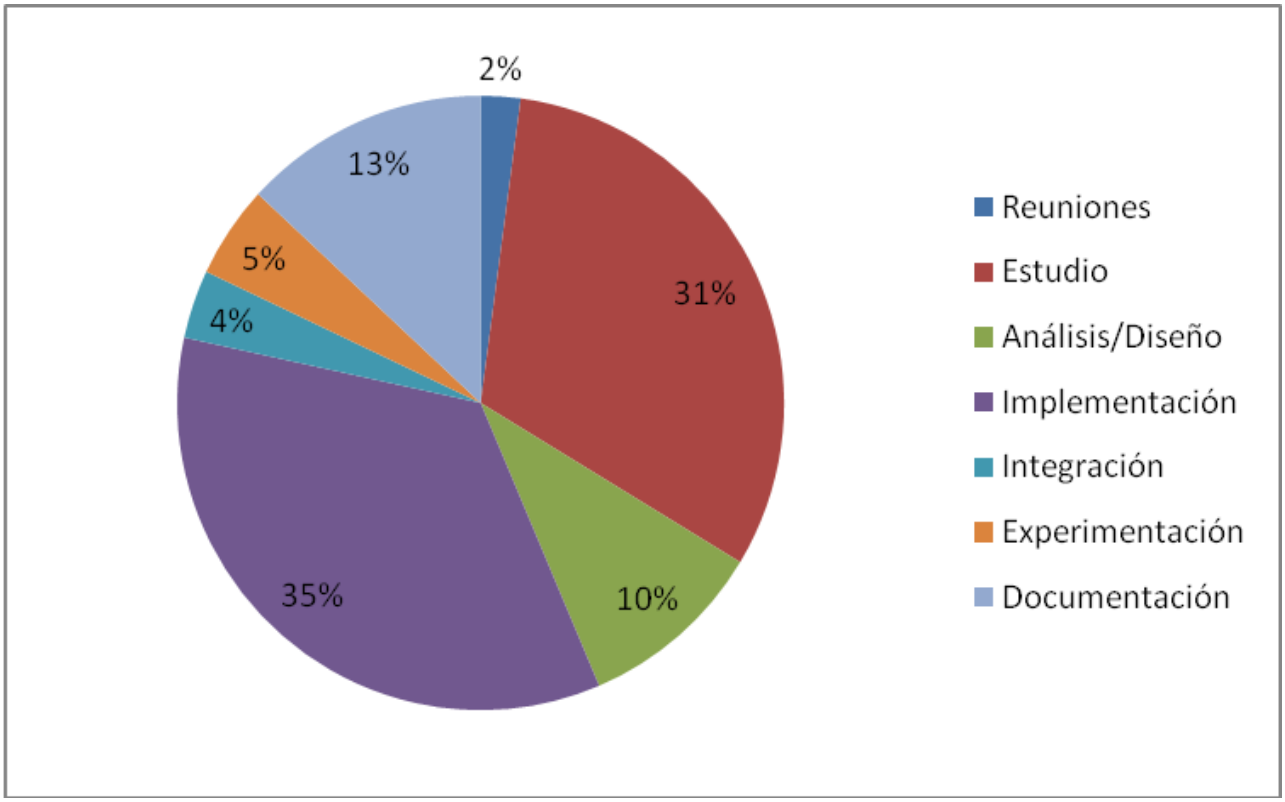

Figura ANEXO.2: Esfuerzo dedicado por tareas