Camino Mínimo de Hiper Grafos Dirigidos

Estructura de Datos y Algoritmos

Informe de Desarrollo

Primer Cuatrimestre de 2013

Javier Fraire

Federico Tedin

Introducción:

_____El objetivo de éste trabajo práctico es el siguiente: dado un hipergrafo dirigido con peso positivo, encontrar el camino mínimo desde un nodo origen hasta un nodo destino (predeterminado en cada hipergrafo). Para lograr esto, se debieron desarrollar dos algoritmos distintos: en primer lugar, un algoritmo que calculara el camino mínimo del hipergrafo exacto, y otro que aproximara dicho camino utilizando distintas heurísticas. El algoritmo exacto tiene una complejidad temporal y espacial mucho mayor que la del algoritmo aproximado, lo cual lo hace mucho mas lento y exigente en términos de memoria y poder de procesamiento.

Primero se comienza describiendo la estructura. Luego se explican los algoritmos y se realizan comparaciones.

Estructuras:

Para representar a un Hiper Grafo se utilizó la clase **HyperGraph**. Dentro de ella se definieron las clases **HyperEdge** y **Node**.

La clase **Node** cuenta con un **String** name que indica su nombre y un **boolean** que indica si fue visitado o no. Además posee dos **List**, una de hiper ejes cola, y otra de hiper ejes cabeza. También posee un **int** que representa la cantidad temporal de padres.[1] La clase **HyperEdge** también posee las variables **name** y **visited**, y posee su peso. A su vez cuenta con tres **List**, una de nodos cola, otra de nodos cabeza y otra que contiene a los hiper ejes "padre", es decir, los hiper ejes que se encuentran a un nodo de distancia hacia "arriba". Además, cuenta con un boolean **isTaboo** que indica si esta prohibido o no para el algoritmo aproximado. HyperEdge cuenta con los booleans **isTop** e **isBottom** que indican si el hiper eje es hijo de el nodo inicio o si incide en el nodo destino. También posee un **EdgePath** para marcar cual es el camino actual hasta llegar al hiper eje en el algoritmo aproximado. A su vez, cuenta con una variable **currentEntriesCount** de tipo **int** que indica la cantidad de nodos que inciden en él en un determinado momento. Para el hash se utilizó el hash por dirección de memoria, ya que cada hiper eje es único.

La clase **HyperGraph** posee un **String name** y dos nodos, **start** y **end**. A su vez cuenta con dos **List**, una de **Node** y otra de **HyperEdge**, que se utilizarán para generar los archivos .dot y .hg.

La clase **EdgePath** contiene un **HashSet** de hiper ejes que indica el camino necesario para llegar a él en determinado momento. También posee el peso total de ese camino.

EdgeSet cuenta con un **HashSet** de **HyperEdge,** que representa una combinación de hiper ejes, el peso total de esta combinación y posee un puntero a otro **EdgeSet**. De esta forma se puede formar un camino.

Los algoritmos se encuentran en las clases **MinimumPathExactAlgorithm** y **MinimumPathApproxAlgorithm**[2]. Se tomó esta decisión ya que es más claro que se encuentren en clases separadas que en la clase **HyperEdge**, y muchos de sus métodos era **static.**

Todas las variables se hicieron **public** debido a la separación del proyecto en paquetes. Si bien esto viola el paradigma de objetos, es mucho más claro para leer y más rápido que utilizar getters y setters.

[1] Su uso se encuentra explicado en el código.
[2] Se recomienda mirar el paquete main.algorithms

Algoritmos:

El proyecto cuenta con dos algoritmos distintos, que cumplen la misma función: buscar el camino mínimo de un hipergrafo dirigido con peso. El primer algoritmo, el algoritmo exacto, encuentra el camino mínimo absoluto del hipergrafo, mientras que el otro encuentra una estimación de este camino, en un cierto tiempo determinado.

Las complejidades temporales y espaciales estos dos algoritmos son muy distintas: el algoritmo exacto cuenta con una complejidad mucho mayor a la del algoritmo aproximado. A continuación se explicara a qué se debe esta diferencia.

Algoritmo Exacto:

En el caso del algoritmo exacto, se busca el camino minimo exacto del hipergrafo. Para lograr esto, se debió desarrollar un algoritmo que analizara, de una forma, todas las posibles combinaciones de ejes posibles, formando caminos de pesos distintos.

La primer idea propuesta fue utilizar algo similar a el algoritmo de Dijkstra, pero con varias modificaciones: por ejemplo, se tendría en cuenta que para acceder a un eje, se debe llegar desde todos sus nodos cola. Ésta idea no funcionó ya que las modificaciones no fueron las suficientes como para que el algoritmo funcione correctamente en un hipergrafo.

A continuación, se probó otra solución, más compleja, pero que funcionó correctamente. Ésta solución consiste en un algoritmo recursivo, que por cada eje (comenzando desde la parte inferior del hipergrafo) analiza las combinaciones de ejes padre posibles (para llegar a dicho eje), y elige la menor, marcándola como parte del camino. Para comparar el peso de cada combinación, se llama recursivamente al algoritmo, con cada conjunto de ejes.

El pseudocódigo del algoritmo es, descontando las optimizaciones realizadas, el siguiente[3]:

```
minimumPathExact(edges) {
      IF (edges esta en el tope superior del grafo)
            RETURN;

      //Almacenar todas las posibles combinaciones de padres válidas
      //para llegar al conjunto de ejes actual

      combinations := parentCombinations(edges);
      min := combinations[0];

      FOREACH (sample: en combinations) {
            minimumPathExact(sample);
            IF (peso de la combinacion es menor a la anterior)
```

[3] La función descrita se haya en el paquete main.algorithms, bajo el nombre *minimumPathExact(EdgeSet)*

```
                min := sample;
      }

      edges.parent := min;
      edges.weight := ejes.weight + min.weight;
}
```

Como se puede ver en el pseudocódigo, la función recibe no un eje sino un conjunto de ejes (la clase **EdgeSet** mencionada anteriormente). La función se llama inicialmente con un conjunto de un solo eje, y termina cuando las llamadas recursivas llegan al caso base, es decir cuando el conjunto de ejes se encuentra en la parte superior del hipergrafo, directamente abajo del nodo inicio. El peso de un **EdgeSet** está determinado inicialmente por la suma de los pesos de los ejes que contiene, y más tarde por la suma de su propio peso más el peso de su conjunto padre (si lo tiene). La ventaja de utilizar ésta clase, con éste comportamiento es que se puede calcular fácilmente el peso de un camino que lleva a cualquier conjunto de ejes, ya que los pesos se van "acumulando" hacia abajo a medida que terminan las llamadas recursivas.

Para utilizar la función, simplemente se llama con cada eje individual pariente del nodo destino, y se comparando los resultados devueltos se elige el menor.

Luego de haber implementado el algoritmo que se terminó utilizando, se probó otra variación. El pseudocódigo es el siguiente:

```
minimumPathExact(edges) {
      min;
      combinations;

       IF (edges esta en el tope superior del grafo y el
      el peso es menor que el mínimo) {
            min = edges;
            RETURN;
      }
      IF (edges.weight > min.weight)
           RETURN;

       //Almacenar todas las posibles combinaciones de padres válidas
       //para llegar al conjunto de ejes actual

       combinations := parentCombinations(edges);

       FOREACH(sample: en combinations){
            sample.child = edges;
            sample.weight = sample.weight + edges.weight;
            minimumPathExact(sample);

       }
}
```

Como se puede observar en el pseudocódigo no hay grandes diferencias en cuanto al código con el utilizado. Son similares . A diferencia del anterior, éste calcula el peso de la combinación actual antes de las llamadas recursivas. Por lo tanto se debe estabelcer la conexión entre ejes antes. Para ello se utilizó la variable **child** en lugar de usar **parent**, ya que se establece a la combinación anterior como hija de la combinación actual. Otra diferencia que se encuentra es que no se utiliza un mapa, por lo que el uso de memoria es más eficiente, pero naturalmente, el algoritmo es más lento. Ésta variación permite aplicar podas respecto al mínimo, es decir, si el camino actual pesa más que el mínimo, al tratarse de pesos no negativos, se descarta.[4]

Problemas y Decisiones: Algoritmo Exacto

La generación de combinaciones de ejes padres para cierto conjunto de ejes es muy costosa, y por ésta razón se decidió implementar algunas optimizaciones al algoritmo.

La primera optimización implementada fue en el caso donde uno de los nodos padre del conjunto actual tenga a su vez exactamente un eje padre. Para poder llegar hasta el conjunto de ejes actual, es necesario habilitar dicho nodo padre, y como éste nodo tiene tan sólo un eje que permite accederlo, no es necesario considerarlo en el momento de generar combinaciones, ya que necesariamente debe ser utilizado. A su vez, si otro nodo distinto también tiene a éste eje como padre, entonces tampoco se considerará en la generación de combinaciones, ya que se utilizará el eje mencionado para acceder a éste otro nodo (y utilizar otros ejes distintos a la vez solo sumaría al peso total del camino). De esta forma, se logra eliminar combinaciones, y por lo tanto, también llamadas recursivas.

Por el otro lado, también se agregó otra optimización a la cantidad de llamadas recursivas hechas. Al generar una combinación de padres, ésta se *hashea* y se busca en un **HashMap** donde se encuentran guardadas todas las combinaciones generadas y calculadas anteriormente. Si dicha combinación se hallaba en el mapa, entonces no es necesario re-calcularla, ya que simplemente se puede utilizar la referencia almacenada anteriormente. Ésta mejora es esencial al funcionamiento óptimo del algoritmo, ya que en muchos casos los pesos de las combinaciones de ejes generadas ya habían sido calculados anteriormente, y se evita una gran cantidad de llamadas recursivas repetidas.

La eficiencia de la solución descrita depende en gran forma de la estructura **HashMap.** Por esta razón, se necesitó que los elementos *hasheados* tengan una función **hashCode** adecuada. En el caso de la clase **HyperEdge**, se optó por utilizar el método **hashCode** de la clase **Object** de Java. Esto se debió a dos razones: primero, cada eje del hipergrafo es único, es decir, hay exactamente una instancia de **HyperEdge** (por grafo) que representa a dicho eje. Por esta razón, se puede utilizar la dirección de memoria de esta instancia única como identificador del eje. La segunda razón es que la función **hashCode** default provee una mejor distribución que, por ejemplo, **hashCode** aplicado a nombre del eje, o a un ID único por eje. El

[4] En el anexo se encuentran las modificaciones de código.

hashCode de **String** se descartó debido a su complejidad. Otra optimización realizada sobre el hash fue reducir la cantidad de veces que se hashea una combinación de 2 a 1.

Otras opciones consideras fueron, por ejemplo, utilizar un **TreeMap** en vez de un **HashMap** para almacenar combinaciones. El **HashMap** provee acceso a datos con una complejidad temporal promedio de O(1), mientras que el **TreeMap**, que utiliza un *red-black tree* como implementación, provee acceso a datos con complejidad O(log N). El **TreeMap** fue considerado ya que una de las desventajas del **HashMap** es que, cuando llena, la estructura necesita agrandar su tabla de *hasheo*, lo cual es un proceso que consume tiempo y memoria del heap. Una vez cambiada la estructura de datos, no se notó un cambio considerable en velocidad ni en consumo de memoria, por lo que se optó por dejar el **HashMap**.

Otro de los problemas encontrados en la implementación del algoritmo fue el consumo de memoria, de variables locales del método principal del mismo. Una vez calculadas las combinaciones de padres posibles para el conjunto de ejes actual, no era necesario conservar las variables auxiliares utilizadas para realizar dicho cálculo. El problema que surgió fue el siguiente: éstas variables auxiliares no se liberaban, porque la función se llamaba recursivamente. Las variables mencionadas ocupaban memoria durante todas las llamadas recursivas de la función, y considerando la gran cantidad de llamadas, la cantidad de memoria ocupada innecesariamente crecia rapidamente. Para evitar este problema, se delegó la generación de combinaciones a distintas funciones auxiliares, lo cual permitió que el **Garbage Collector** libere los recursos utilizados.

Comparaciones: Algoritmos Exactos

- Algoritmo final: el que se utilizó finalmente
- Variación con poda: algoritmo que es similar al utilizado pero que calcula el peso antes de la llamda y poda cuando se superó el peso del camino mínimo.
- Variación hash con **String**: versión en la que se utilizaba el hash de **String**
- Versión sin mapa: versión la que no se guardaban las combinaciones ya realizadas en un mapa. Esta versión se cortó a los 30 minutos para optimizarla, debido a su lentitud.

HiperGrafo	Algoritmo final (Tiempo en segundos)	Variación con poda (Tiempo en segundos)	Variación hash con **String** (Tiempo en segundos)	Versión sin mapa (Tiempo en minutos)
A.hg	16.734	58.385	228.0	Más de 30 minutos
B.hg	0.233	17.308	0.359	Más de 30 minutos

C.hg	0.041	3.405	0.047	N/A

Cálculo de la complejidad

Para calcular la complejidad se puede dividir el hiper grafo en “niveles”. Un nivel estaría representado por todos los hiper ejes padres de un conjunto de hiper ejes determinado. Siendo **m** la cantidad de hiper ejes en un determinado nivel y siendo **n** la cantidad de nodos cola de dicho conjunto de hiper ejes padre. En el peor caso en un nivel al invocar a **generateParents** se genera una lista de listas que contiene todas las colas de lo nodos del nivel. En el peor caso todos los hiper ejes padres contienen en su cola a todos los nodos del nivel. Por lo que todas las listas de la lista de listas mencionada anteriormente son iguales y contiene a todos los hiper ejes padre. Entonces al invocar **parentCombinations** se manda como parámetro dicha lista. Por cada nodo se consideran **m** hiper ejes para considerar como padres, lo cual implica que si la cantidad de nodos es **n**, la cantidad de combinaciones total es:

$$O(m,n) = m^n$$

Siendo H la cantidad de “niveles” en el grafo. En el peor caso en todos los niveles todos los hiper ejes padres contienen en sus colas a todos los nodos del nivel. El orden de la complejidad total es:

$$O(m,n,H) = m^n \times H$$

Algoritmo Aproximado:

En el caso del algoritmo aproximado, se busca una estimación del camino mínimo. Para lograr esto se decidió utilizar la heurística *Taboo Search*.

La idea consiste en utilizar un algoritmo *Best First Search* para obtener un primer resultado. Luego aplicar repetidamente este mismo algoritmo *Best First Search* pero con la diferencia de ir prohibiendo ciertos ejes de manera que se fuerza al algoritmo a elegir otros caminos que pueden ser mejores.

Para el algoritmo de *Best First Search* la idea principal fue adaptar, en cierta forma, Dijkstra a hipergrafos teniendo en cuenta ciertas consideraciones. El algoritmo elige el hiper eje de menor peso acumulado (peso que tiene acumulado el camino hasta el hiper eje). Para ello se utiliza una cola de prioridades. Luego trata de ir a todos los hiper ejes “hijos”, es decir, todos aquellos en los que inciden sus nodos cabeza. No se va a aquellos que estén prohibidos (taboo). Esto se hace a través de los nodos de la cabeza. Si puede llegar, es decir, si el camino ya cuenta con el resto de los nodos necesarios para alcanzar al hiper eje (**currentEntriesCount** es igual la cantidad de nodos cola) , se elige la combinación de hiper ejes de menor peso que lo satisfaga (ésta queda guardada en **parents**), se calcula su peso total y se encola. De no ser así, si el nodo no está visitado, se visita el nodo ,se le “avisa” al hiper eje hijo mediante la variable **currentEntriesCount** que se ha llegado a él por otro nodo (sumándole uno) y se agrega al hiper eje a la lista **parents.** Esto se repite hasta que se haya llegado al destino. El pseudocódigo del algoritmo sería el siguiente (sin verificar que quede tiempo) :

```
bestFirstSearch() {
       priorityQueue;

       edge;


       FOREACH(edge:  en la cabeza de start)
           IF (no es taboo)
              encolar(hyperEdge);

       WHILE (cola no vacía){
           edge := poll();

            //Verificar si se llego al destino
            IF (edge isBottom) {
               RETURN edge;

         //Aquí se marcan o desmarcan los nodos y si es necesario
         //se calcula el peso acumulado y se encola.
         processEdge();
```

```
      }

      RETURN null;
```

Se decidió utilizar este algoritmo y no se cambio, ya que devuelve un primer camino con un error bajo, de aproximadamente 10%. Luego a través de la heurística *taboo Search* se reduce mucho ese error.

Primero se probó hacer estrictamente *taboo Search.* Se tomó el camino generado por el algoritmo anterior, se elegía un hiper eje al azar, se marcaba como prohibido y luego se apliacaba el algoritmo nuevamente. Después se desmarcaba como tabú al hiper eje. Si el camino generado era menor al actual se guardaba ,y luego se utilizaba como camino actual al obtenido, independientemente de si era mejor que el camino anterior o no. Este proceso se repetía hasta que se termine el tiempo.

La segunda variación que se consideró fue que solo se mueve a un vecino si se encontro un mejor camino o si ya se han marcado todos los hiper ejes del camino. La diferencia con lo propuesto anteriormente es que este no se mueve siempre a un vecino, pero todavía los hiper ejes se van marcando y desmarcando de a uno. Estos se van obteniendo de un **HashSet** de hiper ejes que representan el camino que se está analizando actualmente. Como el hash de **HyperEdge** es por dirección de memoria, los ejes se marcan y desmarcan en forma “pseudoaleatoria”. Además los ejes no se desmarcan como prohibidos si se encontró un mejor camino, para evitar que se vuelva rápidamente al mismo camino. También se agregó un **HashSet** de tabúes recientes, es decir, elementos que se prohibieron recientemente, que cada n iteraciones se “reinicia” (todos los hiper ejes contenidos en el son marcados como no prohibidos y se vacía el set).

Finalmente se mejoró la segunda variación para contar con un buen algoritmo aproximado. El pseudocódigo del algoritmo es el siguiente:

```
minWeight;
minPath;
getBetterMinPath() {
     n;
     edge;
     current;
     numberOfTaboos;
     result;
     recentTaboos;

      WHILE (haya tiempo) {
           IF (no hay siguiente en current) {
              IF (numberOfTaboos >= tamaño del camino current)
                 current := pickNeigbour();
                 cantidadtaboos := 1;
           }
           else {
                numberOfTaboos := numberOfTaboos + 1;
                reiniciar Iterador;
```

```
            }

            resetear marcas y variables;

            marcar taboosy agregar a recenTaboos;

            result := bestFirstSearch();

            IF (resultado es mejor que minimo) {
                current := camino de resultado;
                minWeight := peso de resulado;
                minPath := camino de resulado;


                numberOfTaboos := 1;
            }
            ELSE
                 desmarcar taboos;

            IF (pasaron n iteraciones)
               reiniciar recentTaboos;

        n := n +1;

       }
}
```

Como se puede ver en el pseudocódigo se agregó la variable **numberOfTaboos** que representa la cantidad de hiper ejes que se marcan y desmarcan en cada iteración. Esto implica que se pueden marcar varios hiper ejes simultáneamente (sin contar a los hiper ejes que se encuentran en recentTaboos). Si la cantidad de hiper ejes que marcó simultáneamente es mayor que la cantidad de ejes del camino se genera un nuevo vecino, pues ya no quedan hiper ejes para marcar en mi camino viejo. Luego se itera por el camino actual prohibiendo ejes hasta llegar a **numberOfTaboos.** También se puede observar que si el camino encontrado es menor que el que tenía se avanza a ese camino. Sino, se desmarcan los taboos. Luego si pasaron n iteraciones se reinicia **recentTaboos**, es decir, se desmarcan todos sus ejes y se vacía la colección.

Esta variación se ajusta en gran medida a *Taboo Search* pero no la es estrictamente, pues utilizar *Taboo Search* implica moverse a un vecino en cada paso independientemente de si es mejor que la solución actual. Se ajusta en gran medida porque se cuenta con una memoria temporal de hiper ejes prohibidos.

El pseudocódigo del algoritmo completo sería:

```
minWeight;
```

```
minPath;

minmumPathApprox{
        firstResult;

        firstResult := bestFirstSearch();

        minPath := camino de primerResultado;
        minWeight := peso de primerResultado;

        resetear marcas y variables;

        getBetterMinPath();

        markPath();

        RETURN minWeight;

}
```

Problemas y Decisiones: Algoritmo Aproximado

Se dicidió utilizar *Taboo Search* porque ya se contaba con el algoritmo **BestFirstSearch()** y funcionaba muy bien. El problema con **BestFirstSearch()** es que prioriza la combinación de menor peso en cada paso, y existe la posibilidad de que haya una combinación con mayor peso acumulado en ese paso pero que permita elegir combinaciones de menor peso más adelante que lleven a un camino resultante de menor peso.En tal caso, habría convenido elegir dicha combinación aunque sea más pesada. Es decir, la elección de hiper ejes de menor peso en un paso puede condicionar el peso del camino encontrado, pues se puede perder la posibilidad de acceder a los hiper ejes que llevan a un camino con un peso mejor. Con *Taboo Search* se fuerza al camino a evitar ciertos hiper ejes. Esto le permite a **BestFirstSearch()** evitar muchos de esos hiper ejes que condicionan al camino.

La primera versión no fue buena, por un lodo, porque sólo prohibía de a un eje entonces en ciertas ocasiones no lograba que Best First Search encuentre un camino mejor. En ciertas ocasiones, se necesitaban bloquear varios ejes para que llegar a una mejor solución. Por el otro, el hiper eje se elegía de forma aleatoria y no se guardaba temporalmente que hiper ejes habían sido prohibidos, por lo que era muy probable volver al mismo camino muchas veces. Esto hacía que se requiriera muchísimo tiempo para encontrar una mejor solución (si es que la encontraba). También causaba que se “estanque” muy rápido en un mínimo.

La segunda versión propuesta seguía teniendo el mismo problema que la anterior, seguía marcando y desmarcando taboos de a 1.

En la versión definitiva se decidió utilizar como n (valor que indica cada cuanto se reinician los taboos recientes) a la suma de las cabezas de todos los hiper ejes más la suma de todas las cabezas de los nodos. Este número se determinó experimentalmente. Se decidió usar

éste número ya que está relacionado con la cantidad de caminos que hay. No se utilizó la cantidad de caminos total real del hiper grafo, ya que su cálculo de esta era muy costoso, y experimentalmente el número elegido resulto adecuado. En cuanto a cuantos vecinos se marcan y desmarcan como taboo se decidió no hacer todas las combinaciones ya que en un grafo grande esto es costoso y puede tardar mucho tiempo

Comparaciones: Algoritmos Aproximados

- 1er Variación: selecciona ejes de a 1 y no tiene **recentTaboos**
- 2da Variación: selecciona ejes de a 1 y tiene **recentTaboos**
- Versión final: selecciona varios ejes simultáneos y tiene **recentTaboos**

Grafo	1er Variación (Peso promedio)	2da variación (Peso promedio)	Versión final (Peso promedio)
H.hg (100 veces, 20 segundos)	1266	1235.8	1227.6
aproximado1.hg (20 veces, 20 segundos)	1127	1109.1	1104.9
aproximado2.hg(30 veces, 20 segundos)	609	608	595.06
A.hg (50 veces, 15 segundos)	211	211	196.6
B.hg (30 veces, 20 segundos)	505	501.9	497.26

Clases Adicionales:

Para el funcionamiento adecuado del trabajo práctico, se crearon clases adicionales que permiten manipular hipergrafos con más facilidad:

- **GraphSaver:** contiene los métodos necesarios para guardar hipergrafos a archivos con extensión **.hg** y **.dot**. Permite guardar grafos con el camino hallado resaltado en color.
- **GraphLoader:** permite cargar un archivo de hipergrafo con extensión **.hg**. Esta clase devuelve una instancia de tipo **HyperGraph**.
- **GraphSolver:** la clase que compone el front-end del proyecto. Se encarga de recibir los parámetros especificados por el usuario, y realizar las operaciones necesarias (cargar un hipergrafo, aplicar el algoritmo necesario). Finalmente imprime el peso del camino calculado.
- **InvalidTimeException:** clase heredada de **Exception** que habilita a **MinimumPathApproxAlgorithm** a comunicar que el tiempo especificado por el usuario no es válido.

Conclusiones:

A pesar de las dificultades encontradas durante el desarrollo de los dos algoritmos, se lograron desarrollar dos soluciones funcionales al problema propuesto por el trabajo práctico. Se encontraron muchas dificultades a la hora de encontrar el algoritmo exacto, pero finalmente se pudo encontrar. También se encontraron dificultades a la hora establecer cuales son los vecinos.

Por un lado, se creó un algoritmo exacto, que encuentra el camino mínimo el en hipergrafo. Éste algoritmo terminó teniendo una complejidad mucho mayor a la del algoritmo aproximado. Dónde el algoritmo exacto tarda minutos en un hipergrafo, el algoritmo aproximado logra calcular una solución muy cercana en menos tiempo (normalmente en segundos).

Consideramos que nuestro diseño de algoritmo aproximado muy preciso, ya que con restricciones de tiempo cortas logra calcular una respuesta cercana a la respuesta devuelta por el algoritmo exacto.

Anexo:

A continuación se presenta el código de la variación del exacto:

```
public class EdgeSet implements Iterable<HyperEdge>{

//Resto del código
//Se remueve la variable parent

public EdgeSet child;

public void setParent(EdgeSet child)
      {
            if(parent != null){
               this.totalWeight += child.totalWeight;
            this.child = child;
            }
      }

//resto del código
}
public class MinimumPathExactAlgorithm{

//resto del código

private static EdgeSet minPath;


public static double execute(HyperGraph hyperGraph)
      {
            graph = hyperGraph;

            for (HyperEdge edge : graph.end.tail)
            {
                  EdgeSet aux = new EdgeSet(edge);

                  minimumPathExact(aux);

            }

            markPath(minPath);
            graph.end.visited = true;

            return minPath.totalWeight;

      }
```

```
public static void minimuPathExact(EdgeSet current){
   double currentWeight = current.getTotalWeight();

           if (current.iterator().next().isTop)
           {
           if (minPath == null || (currentWeight <
minPath.getTotalWeight()))
                 {
                       minPath = current;
                 }
                 return;
           }

           if (minPath != null && (currentWeight >
minPath.getTotalWeight()))
           {
                 return;
           }

   HashSet<Node> nodes = getParentNodes(current);
   HashSet<EdgeSet> combinations = generateCombinations(nodes);

   for (EdgeSet sample : combinations)
   {
       sample.setChildW(current);
      minimumPathExact(sample);
  }
}

//resto del código
```