PROGRAMACIÓ AVANÇADA I ESTROCTURA DE DADES.

PRÁCTICA 2

13 de gener de 2019

Daniel Casado Faulí
daniel.casado
Marc Aynés i Rulló
marc.aynesi

Domingo 13 de Enero 2019

Índex

1 RESÚMEN DE LOS PROBLEMAS

1.1 Disponibilidad

En este apartado, el usuario introduce un servidor desde el que desea conectarse y el servidor final al que desea conectarse. El usuario también decide que algoritmo usar para encontrar una solución. Estos algoritmos son: Backtracking, Branch and Bound, Greedy y una combinación de estos como son Greedy + Backtracking y Greedy + Branch and Bound. La finalidad de este apartado es dar el mejor resultado de nodos por los que tendrá que pasar el usuario para encontrar la ruta con menos coste, y la ruta más fiable. Para calcular el coste, por cada nodo por el que se avanza, se suma la distancia entre el nodo origen y el nuevo nodo, acumulando este coste en la solución total (si no ha habido un movimiento de nodo a nodo, no se acumula coste). Para calcular la fiabilidad, por cada nodo por el que la información ha estado, se multiplican las fiabilidades de todos estos entre sí (si solo se ha pasado por un único nodo, la fiabilidad de dicho camino es la fiabilidad de este)

1.2 Distribución de carga

En este segundo apartado debemos distribuir los usuarios, en los servidores de forma equitativa, tiene que haber los mismos usuarios en todos los servidores, entre estas soluciones, debemos escoger la solución en que los servidores y los usuarios están más cerca entre sí.

A la hora de calcular una tolerancia a la que poder basarnos para tener en cuenta la distancia entre usuarios y los servidores, escogimos un 5 por ciento de tolerancia con respecto al nivel de carga total que pueden ofrecer los usuarios en un único servidor juntos ya que nos pareció que ese porcentaje no produciría grandes problemas en la distribución de los servidores, al ser una tolerancia tan pequeña, y al mismo tiempo da la oportunidad de que otras soluciones sean aceptadas si de verdad demuestran ser más viables con la distancia que tienen que recorrer los datos para ser transmitidos a los servidores, ya que i, por ejemplo, pusieramos una tolerancia del 1 por ciento, en algunos casos donde el nivel de carga no sea muy alto puede que llgue a ser imposible estar en esos niveles de tolerancia

Para poder tener en cuenta estas soluciones dentro de la tolerancia, guardamos siempre estas soluciones en una lista aparte, de forma que siempre que aparezca una mejor solucion con una menor equitividad de carga estas soluciones se vuelven a revisar para que sigan manteniendo la tolerancia con la nueva equitividad. Tras haber mirado finalmente todos los

caminos, en el caso de que haya una solucion guardada en esta lista externa, se compara dentro de la misma que solución tiene meenor diferencia de proximidad de servidor-usuario en total entre todos los usuarios guardados por cada servidor (dicha diferencia de proximidad la hemos calculado a partir de aplicar la fórmula de equitividad pero para la diferencia de distancias entre servidor y la suma de las localizaciones de cada uno de los posts que ha realizado el usuario, distancia calculada a partir del método haversine).

2 REPRESENTACIÓN DEL ESPACIO DE BÚSQUEDA

2.1 Disponibilidad

2.1.1 Backtracking

En el método Backtracking, el programa comienza analizando una ruta en concreto de camino posible, que es la primera ruta que se encuentra, y va avanzando en profundidad por ese camino hasta que llega al límite de este o cuando ya ha llegado a su destino, A partir de este momento, el programa se guarda la solución obtenida en ese camino, y retrocede un nivel de profundidad para seguir una ruta alternativa desde ese punto, avanzando en amplitud un nivel. Si ese nivel de amplitud no cumple las expectativas de mejor solución que se esperan, es decir, que superen al primer camino o solución obtenido, este camino se descarta como solución. Este proceso se repite un nivel hacia atrás cada vez hasta que llega al origen del camino, y repite el proceso hacia abajo de nuevo, pero esta vez basando como escoge los caminos que sigue a partir de la mejor solución del primer nivell de amplitud, y así hasta llegar al último nivel de amplitud.

2.1.2 Branch and Bound

En cada paso del Branch and Bound en este problema, vamos entrando en un nodo que conecte con el anterior, reducimos el problema, siempre que el nodo en el que estemos sea correcto (la mejor solución es peor que la que tenemos acumulada). En cada Opción del Branch and Bound vamos cogiendo aquellos que aún son prometedores y descartando los que no lo son, o por aquellos que ya hemos pasado anteriormente.

2.1.3 Greedy

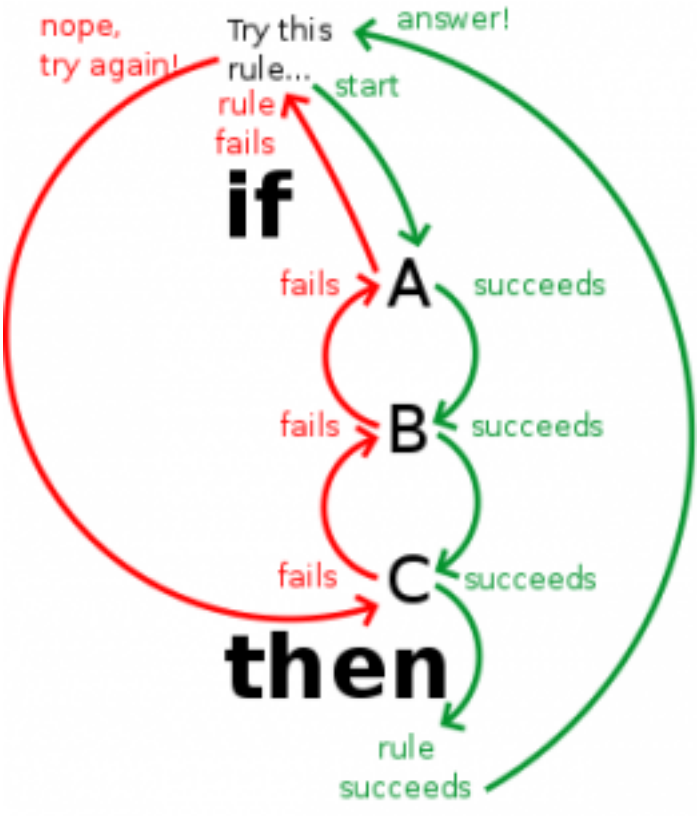
En cada paso de Greedy, cogemos un nodo y vamos acercándonos a la solución por cada paso que damos. En cada Opción seleccionamos únicamente aquella que es mejor, en nuestro caso o menor coste o más fiabilidad.

3 EXPLICACIÓN DE LOS ALGORITMOS

3.1 Bactracking

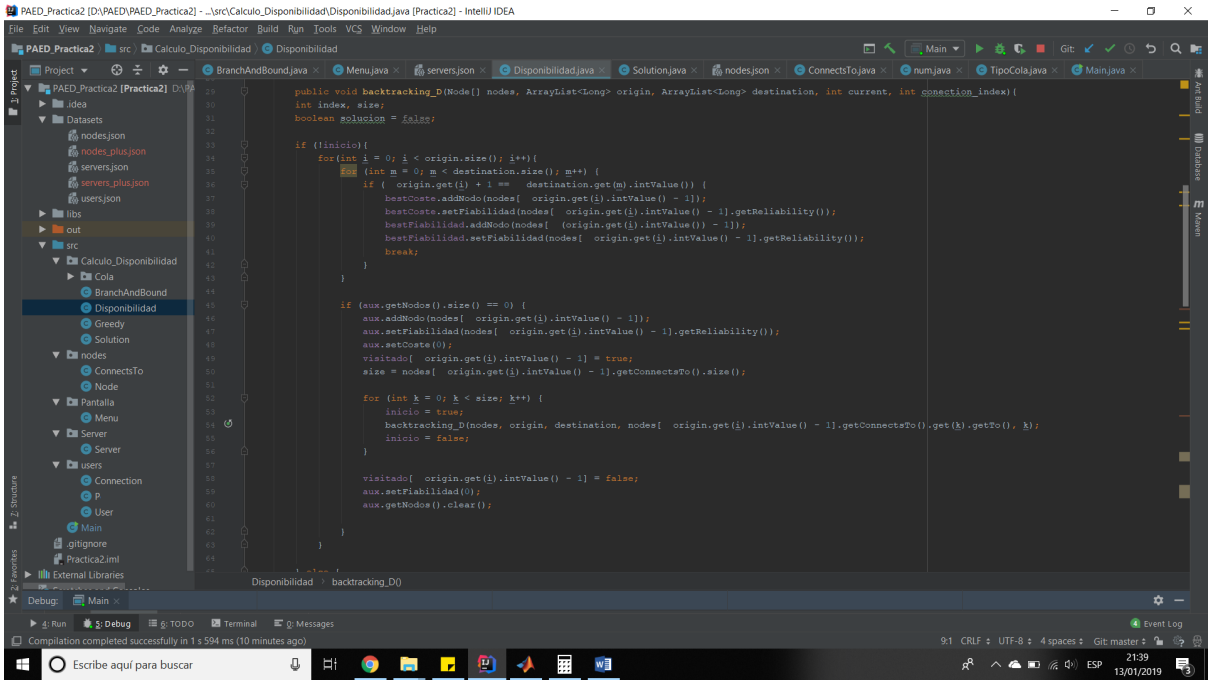
Backtracking es un algoritmo recursivo que consiste en ir rama a rama del problema, incrementando el nivel de profundidad hasta llegar al límite (solución) de esta, descartando de esta forma aquellas soluciones a las que se alcancen que no cumplan con los requisítos o que sean inferiores en nivel de importancia con otras soluciones ya encontradas.

Lo más importante de este método es su uso de la comprobación de amplitud en profundidad dependiendo de si es prometedor el resultado o no. Esto implica que si un resultado, a la mitad del recorrido de su rama, no puede mejorar su resultado lo suficiente hasta llegar a su límite como para competir con el resto de soluciones, esta solución es inmediatamente descartada del resto, reduciendo así el coste temporal de busqueda.

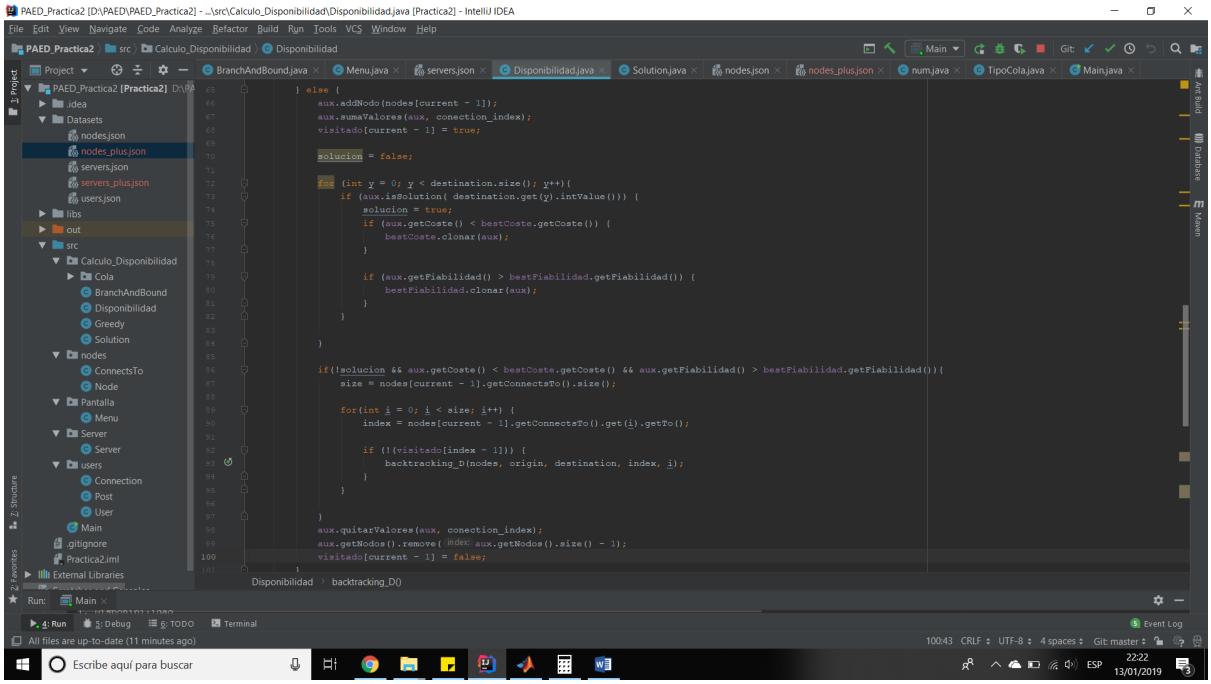


Para realizar este algoritmo, a la hora de codificar, dividimos el problema en dos partes:

1. El inicio del programa, al haber varios nodos por los que un servidor podia comenzar su camino, era necesario primero hacer una comprobación del nodo donde se origina el comienzo del camino era igual al destino, ya que eso implicaría que el mismo servidor puede conectar a través de ese nodo origen al siguiente servidor, y luego, si no eran iguales pero si es el comienzo del camino, comenzar haciendo llamadas a todos los nodos de connectTo (aumentando la anchura del problema) e inicializando los valores de la solución.



2. Después de iniciar el camino, añadimos el nodo que correspondería a avanzar en el camino y comprobamos si es solucion a partir de actalizar los valores de coste y fiabilidad a haber integrado el nuevo nodo. Si no es solución, se avanza un nivel más en la profundidad del camino escogido, siempre teniendo en cuenta que ese camino no haya sido visitado antes, y si no ha sido, se vuelven a hacer llamadas recursivas llamadas a todos los nodos de connectTo (aumentando la anchura del problema de nuevo por cada paso en el camino que hacemos)



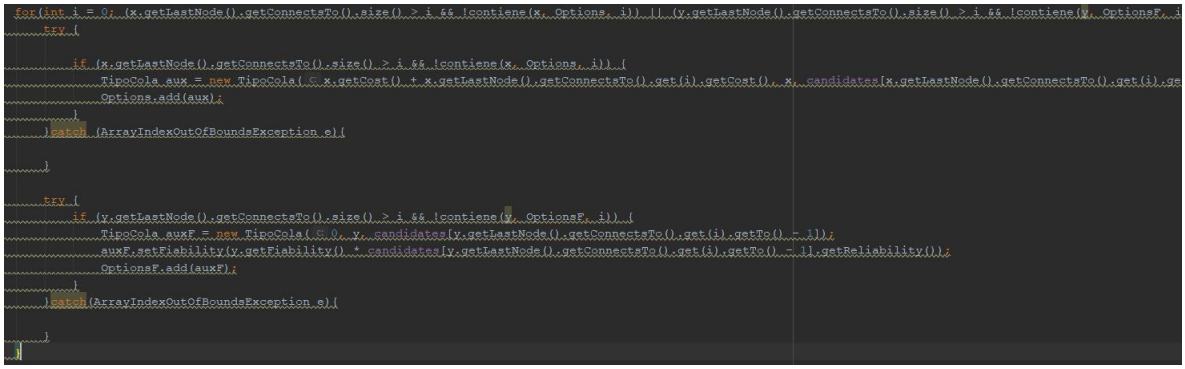
A la hora de saber si un camino es prometedor y hacer poda de caminos que no van a ser capaces de dar una mejor solución, aplicamos un condicional antes de realizar una llamada sucesora en el que si el coste de la solución actual es ya mayor que la de la mejor solución ó su fiabilidad es menor que la de la mejor (ya que al multiplicar siempre un numero por un valor menor que uno solo puede seguir disminuyendo su valor), pues ese camino queda automáticamente descartado y no se sigue haciendo recursividad, avanzando en su nivel de profundidad, en ese lado de la amplitud del problema.

3.2 Branch and Bound

Branch and Bound es un algoritmo iterativo que utiliza una cola que contiene los nodos, y ordenada, en nuestro caso, hay dos colas, una que ordena por fiabilidad y la segunda que ordena por coste. Para ordenar las colas utilizamos el algoritmo QuickSort enseñado este curso en esta asignatura.

```
public void BranchAndBound(Server[] server){
    TipoCola Best = new TipoCola();
    Best.setCost(999999999);
    TipoCola BestF = new TipoCola();

    while(!cola.Vacio() || !colaF.Vacio()){
        TipoCola x;
        ArrayList<TipoCola> Options = new ArrayList<>();
        if(!cola.Vacio()) {
            x = cola.dequeue();
        }else{
            x = new TipoCola();

        }

        TipoCola y;
        ArrayList<TipoCola> OptionsF = new ArrayList<>();
        if(!colaF.Vacio()){
            y = colaF.dequeue();

        }else{
            y = new TipoCola();
        }
```

Al empezar el algoritmo creamos un nodo y lo inicializamos en este caso al coste máximo que puede tener. El algoritmo funcionara mientras la cola ordenada no esté vacía, en ese caso quiere decir que ya no quedan nodos por los que ir, o por que el coste es más elevado que el que tenemos en la variable Best (el mejor caso hasta el momento), o porque ya hemos pasado por allí.



En esta imagen podemos ver el "expand". Cada nodo se conecta a otros, guardamos cada uno de estos nodos en una array llamada Opciones

En esta última imagen podemos ver las comparaciones para saber si es solución. Si es solución y es mejor que el "Best", se guarda en la variable Best si no es solucion pero si es mejor, de momento, que el best se añade a la cola, y se ordena con QuickSort. Cuando la cola esta vacía, calculamos el coste total y la fiabilidad total, y enseñamos por pantalla los nodos por los que pasamos y el total calculado.

3.3 Greedy

Greedy es un algoritmo iterativo que pretende buscar una solución buena rapidamente. Greedy escoge siempre el siguiente mejor nodo al que ir, el nodo con mas fiabilidad o con menos coste. Es por ello que este algoritmo es capaz de reducir problemas de optimización combinatoria, de coste temporal asintótico exponencial) a un coste temporal asintótico polinómico, pero sacrificando la veracidad total de los resultados obtenidos.

```
public TipoCola calculateGreedy(Server[] a){

    Node c = candidates[(a[(int) (servidor)].getReachableFrom().get(0)).intValue() - 1];
    int distancia = candidates.length;

    Solution.add(c);
    distancia--;
    while(!found && distancia!= 0){
        c = selecciona(c);      //selecciono el siguiente node que tiene menos coste y que no h
        if(c.getId() == -1){
            break;
        }
        Solution.add(c);
        for (int i = 0; a[(int) destino].getReachableFrom().size() > i; i++) {
            if (c.getId() == a[(int) destino].getReachableFrom().get(i)) {
                found = Boolean.TRUE;
                break;
            }
        }

    }
```

Greedy empieza cogiendo el primer nodo de todos, a partir de este selecciona el siguiente nodo conectado que tenga menor coste y que no haya pasado por el, añadiendolo a solución. Sigue así hasta que encuentra un nodo que conecta el servidor final o no tenga mas camino que seguir, si este ultimo es el caso con Greedy no hay solución.

4 COMPARATIVA DE ALGORITMOS

	Tiempo (ms)					
Método	**Opcion 1**			**Opcion 2**		
Intento	**Intento 1**	**Intento 2**	**Intento 3**	**Intento 1**	**Intento 2**	**Intento 3**
Backtracking	0,43ms	0,612ms	0.4ms	-	-	-
Branch & Bound	4,09ms	5,016ms	5,325ms	-	-	-
Greedy	0,387ms	0,27ms	0,271ms	-	-	-
Backtracking + Greedy	0,308ms	0,175ms	0,239ms	-	-	-
Branch & Bound + Greedy	2,56ms	0,882ms	1,274ms	-	-	-

En la imagen superior podemos ver la comparación de los algoritmos en tiempo de ejecución. Podemos ver como el algoritmo más rápido en nuestro caso es el Greedy + BackTracking esto se puede deber, entre muchos casos, a que el tiempo de ejecución depende del sistema operativo ya que también ejecuta otros procesos en segundo plano.

El algoritmo que debería ser más rápido, por lo que podemos comprobar, es el Greedy ya que es iterativo y solo escoge un camino a diferencia de los otros 2 algortmos que van entre distintas ramas, haciendo del problema de coste asintótico N. Sin embargo, podemos ver que el método de Backtracking + Greedy llega a ser incluso más optimo que el Greedy, ya que tienen casi el mismo coste temporal y el seguno método si que realmente da el mejor resultado a diferencia del Greedy, y esta competitividad en el tiempo de ejecución se puede deber a que el Greedy realmente llegue a obtener el mejor resultado o el segundo mejor, por lo que, a la hora de de realizar el Backtracking, este puede hacer poda rápidamente de muchos caminos o incluso directamente descartarlos todos, dando así el resultado en un coste mínimo.

En el caso del método de Branch & Bound contiene una lista ordenada con un QuickSort, hay que recordar que el QuickSort con pocos datos es mas lento que otros metodos de ordenación como es el caso, pero si tenemos muchos datos, es muy rápido. Esta cola hace que el tiempo del algoritmo augmente significtaivamente, sin contar el propio tiempo del algoritomo, y los programas que pueden haber en segundo plano, y que el sistema operativo ejecuta.

La Opción 2 no la hemos podido realizar debido a un error en el backtracking que no sabemos resolver a día de hoy y que nos hizo retrasarnos en la codificación del resto de métodos.

5 MÉTODO DE PRUBAS USADO

Para hacer pruebas con nuestro algoritmo hemos usado los datasets dados en el eStudy, como que el dataset "servers"y el dataset "servers_plus"tienen campos distintos (el campo "reachable_from"en el primero es un tipo simple y en el segundo es una array) hemos modificado el primer dataset para que los dos sean arrays y así poder trabajar con arrays de nodos.
Para hacer pruebas tambien usamos un debugger para poder ver que hace exactamente nuestro programa a cada paso, así hemos hecho pruebas para ver como responde nuestro programa con ciertos datos de entrada.

6 PROBLEMAS OBSERVADOS

En esta práctica hemos tenido muchos problemas de compilación y errores con los algoritmos ya que pensábamos que los entendíamos perfectamente, pero se nos escapaban algunos puntos internos, que hemos ido aprendiendo durante la realización de este trabajo.
Un problema importante que hemos tenido cuando habíamos terminado el primer problema fue que en el "dataset++"en el JSON "server"había una variable distinta que nosotros tratábamos como un l·long"pero que era un array eso provocó que tuvieramos que cambiar parte del código de los algoritmos ya que no teníamos pensado que esta variable fuese distinta y pensamos el algoritmo como si no fuera un "array". También modificamos manualmente el dataset más básico para hacer que fuera un array y así no nos diera problemas a la hora de compilar con él.
En el momento de juntar el algoritmo Greedy, con los otros dos, nos surgieron problemas al ver los resultados, ya que nos daban resultados erroneos, este error fue solucionado contando de forma correcta el coste total, y la fiabilidad total de un camino.
Por culpa de todos estos problemas, no tuvimos tiempo de terminar el trabajo, el apartado 2 no está funcional ya que no conseguimos que compile correctamente. El apartado 2 no será accesible ya que el programa deja de funcionar o da la información incorrecta si se accede a él.

7 CONCLUSIONES

A nivel personal ha estado interesante el uso de algoritmos como el Backtracking, Brach and bound y el Greedy, con sus respectivas combinaciones, ya que en clase podemos ver sobre papel la teoría, pero implementarla es mucho más difícil que sobre papel.
Un punto que nos ha gustado ha sido la implementación de la cola ordenada del Branch and Bound ya que usado algoritmos aprendidos y utilizados en la práctica anterior como es el Backtracking.
También hemos podido observar las principales diferencias entre los algoritmos a nivel interno siendo estos un poco distintos de lo que pensábamos.

8 BIBLIOGRAFIA

Stack Overflow - Where Developers Learn, Share, & Build Careers. *Stack Overflow* [online]. Disponible a: <https://stackoverflow.com>. [Consultado el 4 de Enero de 2019].

Paraschiv, E. 2018. Java, Spring and Web Development tutorials. *Baeldung* [online]. Disponible a:<https://www.baeldung.com>. [Consultado el 9 de Enero de 2019].