UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

PRÁCTICA SISTEMAS INTELIGENTES

Pilar García Martín de la Puente
Mª Dolores Sesmero Pozo
Laura Fernández Trujillo

Asignatura: Sistemas Inteligentes
Grupo: 3ºB
Titulación: Grado en Ingeniería Informática
Fecha: 6/12/201

ÍNDICE

1. INTRODUCCIÓN AL PROBLEMA Y DOMINIO

ENTRADA:

- Las coordenadas geográficas de las esquinas opuestas de un recuadro o un fichero osm.
- Un estado inicial (Loc,[LocV1,..,LocVm]), que crearemos a partir de un identificador del nodo inicial y los identificadores de los nodos objetivos.
- Un valor entero identificador de cada estrategia.
- Un valor entero máximo de profundidad.
- Un valor entero para la elección de realizar la poda o no encada estrategia.

SALIDA:

- Fichero GPX.
- Costo de la solución.
- Complejidad espacial (nº de nodos del árbol generados).
- Complejidad temporal (Tiempo tardado en generar la solución).

Se quiere realizar 3 casos de estudio con diferentes profundidades (pequeña, mediana y grande) con todas las estrategias implementadas (anchura, profundidad, costo uniforme, voraz y A*) y comparar los resultados de las complejidades espaciales y temporales y las complejidades obtenidas y teóricas.

2. ESTRUCTURAS Y CLASES DEFINIDAS

Class Enlace

Esta clase representa los enlaces entre un nodo y sus adyascentes, con los atributos:

- NodoDestino: almacena ese nodo adyascente. Será del tipo Nodo.
- distancia: nos da los metros que hay entre un nodo y su nodo adyascente.

En esta clase solo encontramos los métodos get() y set() de los correspondientes atributos.

Class Grafo

Esta clase representa un grafo que estará formado por los nodos obtenidos de nuestro mapa OSM. Se creará una TablaHash para ayudarnos a formar la estructura de ese grafo, contendrá el id del nodo y el objeto nodo.
Esta clase sólo tendrá ese atributo que acabamos de mencionar:

- tablaNodo: del tipo Hashtable.

Además de los métodos imprescindibles de consulta y modificación (get() y set()) tenemos los métodos:

- addEnlace(): pasándole el id del nodo origen y destino y la distancia entre ellos, se crea un enlace entre esos dos nodos. Este método llama internamente al método *addenlace()* de la clase *Nodo.*
- addNodo(): añade a la tabla hash el id del nodo y el objeto nodo leídos.
- deleteNodosTabla(): para borrar los nodos de la tabla hash
- getNodo(): coge el nodo que se le pasa a éste método de la tabla hash

Class Nodo

Es utilizada para almacenar los nodos leídos del mapa en la tabla hash de la clase grafo para así crearnos el grafo que utilizaremos a lo largo del problema.
Sus atributos son:

- node: tipo de dato leído del archivo OSM
- enlaces: cada nodo va a tener una lista de sus enlaces a los nodos adyascentes, donde cada enlace tiene el id del nodo destino y el peso del enlace, esta ArrayList de ocupa de almacenarlos.
- numEnlaces: numero de enlaces que tiene un nodo, se utiliza para segurarnos que un nodo tiene enlaces o no y decidir si es un nodo útil para nuestro problema.

Sus métodos destacables son:

- addEnlace(): este método es llamado desde *addEnlace()* de la clase *Grafo* y añade a la lista *enlaces* el nodo destino y su distancia desde el nodo que se indica desde ese método. Por cada enlace que se hace, aumenta el contador del atributo *numEnlace*.
- existeEnlace(): comprueba que un nodo tiene edyascentes, si es así, devuelve el número de enlaces que tiene, si no tuviera enlaces, notificará tal cosa.

Class Estado

Esta clase se utiliza para representar el estado de un nodo árbol, es decir, es la representación del nodo en el que nos encontramos y los objetivos que nos quedan por visitar.
Los atributos de esta clase son:

- long idNodo: representa el id del nodo en el que nos encontramos.
- double latitud: latitud del nodo que representa el estado.
- double longitud: longitud del nodo que representa el estado.
- ArrayList <Long> objetivos: lista con los id de los nodos objetivo que nos quedan por visitar.

Esta clase contiene los métodos get, set y toString correspondientes.
También encontramos los métodos importantes:

- *sucesores()*: el cual se encarga de obtener los sucesores de un estado dado con la utilización del grafo. Para ello necesitamos crearnos un *nodoInicial* de tipo Nodo con el estado que le hemos pasado al método, del cual vamos a obtener sus adyacentes. De cada enlace adyacente obtenido vamos a sacar su destino, los objetivos nuevos y su distancia para crearnos un objeto sucesor. Para obtener los objetivos nuevos vamos a comprobar que el id del nodo obtenido de los enlaces no está en los objetivos del estado inicial. Si estuviera, se borra ese id y si no se deja. Cada sucesor obtenido se añadirá una *listaSucesores* que devolverá el método.

```
public ArrayList <Sucesor> sucesores (Estado e, Grafo g){
    Double distancia;
    Nodo nodoInicial= g.getNodo(e.getIdNodo());
    ArrayList <Enlace> adyacentes = nodoInicial.getEnlaces();
    ArrayList <Long> objetivos= new ArrayList<Long>();
    ArrayList <Sucesor> listaSucesores= new ArrayList <Sucesor>();
    objetivos=e.getObjetivos();

    for(int i=0; i<adyacentes.size(); i++){
        long [] accion = new long [2];
        accion[0]=e.getIdNodo();
        Nodo nodoSucesor=adyacentes.get(i).getDestino();
        accion[1]=nodoSucesor.getNode().getId();
        ArrayList <Long> objetivosNuevos= new ArrayList<Long>();

        for(int j=0; j<objetivos.size(); j++){
            objetivosNuevos.add(objetivos.get(j));
            if(accion[1]==objetivos.get(j)){
                objetivosNuevos.remove(j);
            }
        }
        distancia= adyacentes.get(i).getDistancia();

        Estado estadoNuevo =new Estado(accion[1], nodoSucesor.getNode().getLatitude(),
                nodoSucesor.getNode().getLongitude(), objetivosNuevos);
        Sucesor suc1= new Sucesor(accion, estadoNuevo , distancia);
        listaSucesores.add(i, suc1);
    }
    return listaSucesores;
}
```

- *EsValido():* este método va a devolver una variable boolean. Devolverá false si dado un estado, este no se encuentra en el grafo, y true en caso contrario.

- *EsObjetivo():* va a comprobar si un estado es ya objetivo. Para ello va a comprobar si la variable objetivos que es una lista está vacía(no le quedan más objetivos por recorrer).

Class Sucesor

Esta clase sirve para representar el sucesor de un estado, es decir, representa un el nodo adayacente al nodo del estado dado, con los objetivos actualizados y la distancia que hay entre ambos.
Los atributos de esta clase son:

- long [] accion= es un vector de dos posiciones el cual, la primera posición representa el id del nodo del estado actual, y la segunda posición indica el id del nodo adyacente.
- Estado estadoSucesor: representa el estado del sucesor (con el id del nodo adyacente y los objetovos actualizados)
- double costeAccion: indica el coste de ir de un nodo a otro, es decir, la distancia entre ambos.

Esta clase contiene los métodos get, set y toString correspondientes.

Class Frontera

Esta clase representa la frontera necesaria para resolver nuestro problema. En otras palabras, es la estructura que necesitamos para guardar los nodoArbol correspondientes y que después vamos a ir sacando para explorarlos en función de su prioridad.
El atributo de esta clase es:

- PriorityQueue<NodoArbol> colaprioridad: vamos a utilizar este objeto de tipo cola como estructura para representar nuestra frontera. Es una cola de prioridad, es decir, sus elementos van a estar ordenados según algún parámetro dado. Esta ordenación se hace en la clase NodoBusquedaComparator.

Esta clase contiene los métodos importantes:

- insertarLista(): este método va a insertar en la frontera los nodos que le vamos a pasar en una lista, llamando al método *insertar().*
- *insertar():* va a utilizar un objeto de tipo NodoBusquedaComparator para insertar en su posición correspondiente el nodoArbol que se le ha pasado.
- eliminar(): elimina un nodo de la frontera.
- esVacia(): comprueba si la cola que representa la frontera está o no vacía.

Class NodoArbol

Representa los nodos que van a ser elegidos del grafo para expandirlos y así ir formando la ruta solución hasta recorrer todos los objetivos introducidos.
Los atributos de esta clase son:

- *private NodoArbol padre*: representa el nodo padre del nodo actual al que nos referimos
- *private Estado estado*: representa el estado del nodo actual
- *private double costo*: represena el costo del nodo actual, el cual es la suma de la distancia del padre al nodo actual más la el costo del padre

- *private long accion[]:* es un vector de dos elementos los cuales representan el identificador del nodo padre (primera posición) y el identificador del nodo adyacente (segunda posición)
- *private int profundidad:* representa la profundidad del nodo actual, la cual es la profundidad del nodo padre más 1.
- *private double valor:* es un atributo que cambia en función de la estrategia elegida, y es utilizado para la preferencia en el orden de los elementos de la frontera

Esta clase contiene los métodos get, set y toString correspondientes.
También encontramos los métodos CreaListaNodosArbol y distanciaMaxima.

- *CreaListaNodosArbol:* en este método se crea una lista de tipo NodoArbol que contendrá todos los nodos que serán añadidos a la frontera.
  Recorremos todos los adyacentes/sucesores que tenga el nodo actual (el que estamos sacando de la frontera para explorarlo y comprobar si es solución o no, y si no es solución expandirlo). Por cada adyacente nos creamos un nodo del tipo NodoArbol con sus correspondientes atributos y modificando el atributo "valor" dependiendo de la estrategia que estemos realizando en cada momento, y lo metemos en la lista de nodos a incluir en la frontera teniendo en cuenta que la profundidad de ese nodo sea menor o igual que la máxima profundidad impuesta y que en el caso de podar realice la poda correspondiente.
  Por último, una vez recorrido todos los adyacentes y comprobar si se añaden o no a la frontera, devolvemos la lista creada de tipo NodoArbol con todos los nodos válidos.

```
public ArrayList<NodoArbol> CreaListaNodosArbol (Frontera frontera, Problema problema,
        ArrayList<Sucesor> ls,NodoArbol nodo_actual,int prof_max,int estrategia, Grafo g, int realizarPoda){

    ArrayList<NodoArbol> listanodos = new ArrayList<NodoArbol>();

    for(int i=0; i<ls.size(); i++){

        Sucesor sucesor=ls.get(i);
        long [] accion =  new long [2];
        accion=sucesor.getAccion();
        double valor=0;
        NodoArbol nodoSucesor= null;
        if(estrategia==1){

            valor=(nodo_actual.getProfundidad()+1);

        }else if(estrategia==2){


            valor= prof_max - (nodo_actual.getProfundidad()+1);

        }else if(estrategia==3){

            valor= sucesor.getCosteAccion()+nodo_actual.getCosto();

        }else if(estrategia==4){
```

```
            valor= distanciaMaxima(sucesor, g);

        }else if(estrategia==5){

            valor= distanciaMaxima(sucesor, g)+sucesor.getCosteAccion()+nodo_actual.getCosto();

        }
        nodoSucesor= new NodoArbol(nodo_actual, sucesor.getEstadoSucesor(),
                sucesor.getCosteAccion()+nodo_actual.getCosto(), accion, nodo_actual.getProfundidad()+1, valor);



        if(realizarPoda==1){//ralizamos la poda
            if(nodoSucesor.getProfundidad() <= prof_max  && !problema.poda(nodoSucesor, estrategia)){

                listanodos.add(nodoSucesor);
            }
        }else{//no realizamos la poda
            if(nodoSucesor.getProfundidad() <= prof_max){
                listanodos.add(nodoSucesor);
            }
        }

    }


    return listanodos;

}
```

- *distanciaMaxima:* este método es utilizado sólo para las estrategias voraz y A* donde el atributo “valor” del NodoArbol es la distancia máxima a los objetivos y la distancia máxima a los objetivos más el costo del nodo respectivamente. En este método calculamos la distancia que hay entre el nodo que estamos comprobando en ese momento a cada uno de los objetivos que le queden por explorar y devolvemos la distancia máxima obtenida.

Class NodoBusquedaComparator

Esta clase es utilizada única y exclusivamente por la clase Frontera para poder ordenar los elementos de la cola con prioridad que representa la frontera en función del valor que tenga cada nodo introducido a la misma.

Class Problema

Esta clase es utilizada para almacenar el identificador del nodo inicial y el estado inicial de nuestro problema, además de una tabla hash donde almacenamos los nodos tipo NodoArbol para así poder hacer la poda y simplificar la complejidad del problema.

Los atributos de esta clase son:

- *private long idEstadoInicial*: representa el identificador del nodo inicial introducido por teclado.
- *private Estado estado:* representa el estado inicial que está compuesto por todos los objetivos introducidos por teclado y que hay que visitar.

- *static Hashtable<String, Double> tablaPoda = new Hashtable<String, Double>():* compuesta por un string que represena el estado de un nodo tipo NodoArbol y un double que representa el atributo valor de dicho nodo.

Esta clase contiene los métodos get, set y toString correspondientes.
También contiene el método poda, sucesores y estadoMeta.

- *Poda:* a este método le pasamos el nodo de tipo NodoArbol que estamos comprobando si podar o no y la estrategia que estamos utilizando en dicho momento. Este método lo que hace es transformar el estado del nodo que le pasamos en un string y lo comprobamos con los estados de todos los elementos introducidos en la tabla hash tablaPoda. En el caso de que encontremos un estado igual en la tabla comprobamos el valor de ese estado almacenado con el valor del nodo, si la estrategia es profundidad y el valor del nodo es mayor que el de la tabla debemos reemplazar dicho valor y por lo tanto no podamos y añadimos ese nodo a la frontera, pero en el resto de las estrategias haríamos lo mismo pero en caso de que el valor del nodo sea menor o igual que el de la tabla. Si no se cumple esta condición entonces no cambiamos el nodo de la tabla y podamos, por lo tanto no se introduce el nodo a la frontera. Por otro lado, si el estado del nodo que estamos comprobando no se encuentra en la tabla lo que hacemos es meter ese estado y el valor del nodo en la tabla y no podamos, es decir, introducimos ese nodo en la frontera. Este método devuelve un true o false en función de si podamos o no.

```
public boolean poda(NodoArbol nodo, int estrategia){
    String estado= nodo.getEstado().toString();
    Enumeration<String> e = tablaPoda.keys();
    boolean esta=false, poda=false;
    while(e.hasMoreElements()){
        String estadoTabla=e.nextElement();
        if(estado.equals(estadoTabla)){
            esta=true;
            if(estrategia==2){
                if(nodo.getValor()>tablaPoda.get(estadoTabla)){
                    poda=false;
                    tablaPoda.remove(estadoTabla);
                    tablaPoda.put(estado, nodo.getValor());
                }else poda=true;
            }else{
                if(nodo.getValor()<=tablaPoda.get(estadoTabla)){
                    poda=false;
                    tablaPoda.remove(estadoTabla);
                    tablaPoda.put(estado, nodo.getValor());
                }else poda=true;
            }
        }
    }
    if(esta==false){
        tablaPoda.put(estado, nodo.getValor());
        poda=false;
    }
    return poda;
}
```

- *Sucesores:* en este método se llama al método sucesores del objeto estado.

- *estadoMeta*: en este método comprobamos si un estado es objetivo o no, para lo cual tenemos que haber visitado todos los nodos objetivos que hayamos puesto al principio del problema. Aquí llamamos a un método que contiene el objeto estado que le pasamos y ahí es donde realmente realizamos la comprobación.

Class Principal

Esta clase es la que contiene el método main, donde se realiza la lectura del mapa .osm, se piden los datos necesarios para la ejecución incluyendo la estrategia deseada y se realiza el método de búsqueda necesario en función de cada estrategia, y se crea el fichero .gpx con la ruta resultante de la solución al problema.

Lo primero de todo es la lectura del fichero .osm para poder crear así el grafo con todos los nodos necesarios. Una vez construido el grafo se pedirá por teclado al usuario introducir la estrategia que quiere emplear, la profundidad máxima deseada y si quiere aplicar poda o no. Una vez elegidos correctamente todos estos datos se llama al método BusquedaArbol. Este método se encarga de buscar la solución a partir de un estado inicial recorriendo todos los nodos necesarios para visitar todos los objetivos establecidos. Aquí se explorarán y expandirán todos los nodos que haya almacenados en la frontera hasta que encontremos el estado objetivo o la frontera quede vacía, con lo cual se devolverá que no hay solución.
Una vez encontrada la solución se devuelve un ArrayList con los nodos que forman dicha solución, y utilizaremos los datos necesarios de éstos para crear el fichero .gpx para poder visualizar la ruta de la solución.

```
static ArrayList <NodoArbol> BusquedaArbol (Grafo grafo, Problema prob,int estrategia, int prof_max, int realizarPoda){
      //Proceso de inicialización
      Frontera frontera = new Frontera();

      long accion[]=null;
      NodoArbol padre=null;
      NodoArbol nodo_inicial= new NodoArbol(padre, prob.getEstado(),0,accion,0,0);

      frontera.insertar(nodo_inicial);

      complejidadEspacial++;

      boolean solucion=false;
      NodoArbol nodo_actual=null;


      //Bucle de búsqueda
      while( solucion==false && frontera.esVacia()==false ){
          ArrayList <Sucesor> ls= new ArrayList<Sucesor>();
          ArrayList <NodoArbol> ln= new ArrayList<NodoArbol>();

          nodo_actual=frontera.getPrimero();

        if (prob.estado_meta(nodo_actual.getEstado())){
           solucion=true;
        }else{
           ls=prob.Sucesores(grafo, nodo_actual.getEstado());

           ln=nodo_actual.CreaListaNodosArbol(frontera, prob, ls,nodo_actual,prof_max,estrategia, grafo, realizarPoda);

           int ramificacionAct=ln.size();

           if(ramificacionAct> ramificacion){
               ramificacion=ramificacionAct;
           }


            complejidadEspacial=complejidadEspacial + ln.size();

            frontera.insertarLista(ln);
        }
      }
      //Resultado
      if (solucion==true){

         return creaSolucion(nodo_actual);
      }else{
          return null;
      }


    }
```

- BusquedaArbol: primero de todo introducimos el nodo inicial a la frontera y aumentamos la complejidad espacial en uno, después vamos sacando nodos de la frontera mediante un “while” hasta q la frontera esté vacia o encontremos la solución. Con el nodo que sacamos de la frontera primero comprobamos si es un estado solución y si no lo es entonces lo expandimos obteniendo sus adyacentes, con estos adyacentes construiremos una lista de nodos de tipo NodoArbol con los nodos adyacentes que sean válidos teniendo en cuenta la estrategia, la profundidad del nodo y si realizamos poda o no, y esa lista la introduciremos en

la frontera. Cada vez que recorremos el bucle “while” se aumenta la complejidad con el número de nodos nuevos que introducimos a la frontera.
Una vez obtenido el nodo con el estado solución se llama al método creaSolución que devuelve un ArrayList con los nodos solución.

- FicheroGPX: En el fichero gpx se introducirá la latitud, longitud e identificador del nodo inicial y los objetivos. Además recorremos la solución obtenida y pondremos la latitud, longitud, identificador, altura (que siempre será 0), costo y profundidad de cada uno de los nodos que formen la solución al problema.

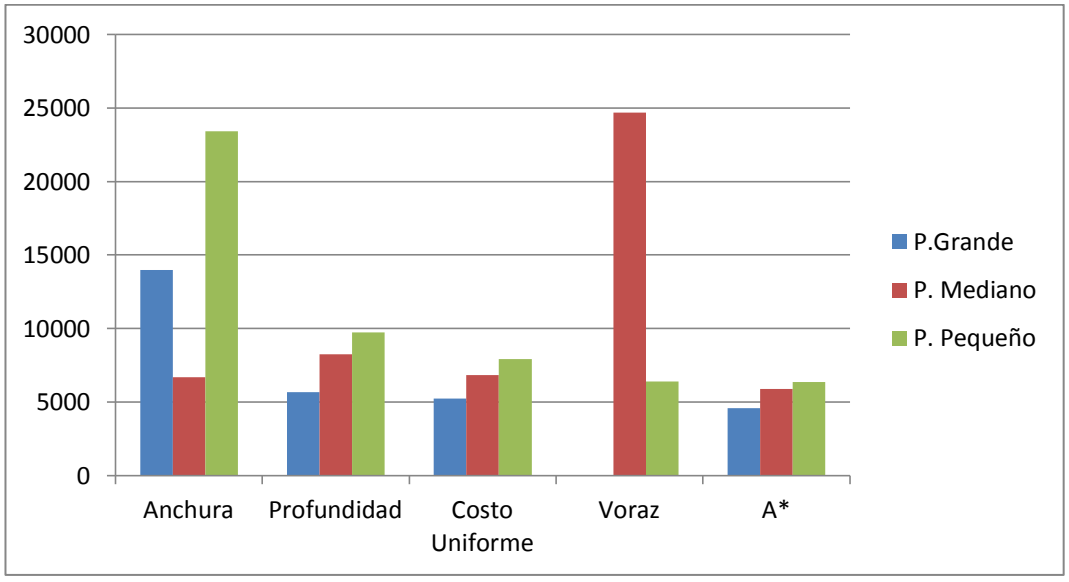
```
public static void ficherogpx(Estado estadoInicial, ArrayList <NodoArbol> solucion, RandomAccessFile ficheroGPX,long idPadre, ArrayList <Long> objetiv
    NodoArbol aux=null;

    ficheroGPX.writeBytes("<?xml version=\""+1.0+"\" encoding=\"UTF-8\"?>");
    ficheroGPX.writeBytes("\n<gpx>");
    ficheroGPX.writeBytes("\n<wpt lat=\""+estadoInicial.getLatitud()+"\" lon=\""+estadoInicial.getLongitud()+"\">");
    ficheroGPX.writeBytes("\n\t<name>"+estadoInicial.getIdNodo()+"</name>");
    ficheroGPX.writeBytes("\n</wpt>");
    for(int i=0; i<objetivos.size(); i++){
        ficheroGPX.writeBytes("\n<wpt lat=\""+ grafo.getNodo(objetivos.get(i)).getNode().getLatitude()+"\" lon=\""+grafo.getNodo(objetivos.get(i)).get
        ficheroGPX.writeBytes("\n\t<name>"+objetivos.get(i)+"</name>");
        ficheroGPX.writeBytes("\n</wpt>");
    }

    ficheroGPX.writeBytes("\n<trk>");
    ficheroGPX.writeBytes("\n\t<name>Route</name>");
    ficheroGPX.writeBytes("\n\t<trkseg>");


    for(int i=0; i<solucion.size(); i++){
        aux=solucion.get(i);


        ficheroGPX.writeBytes("\n\t\t<trkpt lat=\""+aux.getEstado().getLatitud()+"\" lon=\""+aux.getEstado().getLongitud()+"\">");
        ficheroGPX.writeBytes("\n\t\t\t<ele>0</ele>");
        ficheroGPX.writeBytes("\n\t\t\t<name>"+aux.getEstado().getIdNodo()+"</name>");
        ficheroGPX.writeBytes("\n\t\t\t<cost>"+aux.getCosto()+"</cost>");
        ficheroGPX.writeBytes("\n\t\t\t<deep>"+aux.getProfundidad()+"</deep>");
        ficheroGPX.writeBytes("\n\t\t</trkpt>");

    }

    ficheroGPX.writeBytes("\n\t</trkseg>");
    ficheroGPX.writeBytes("\n</trk>");
    ficheroGPX.writeBytes("\n</gpx>");
    ficheroGPX.close();
}
```

3. CASOS DE ESTUDIO

		ANCHURA	PROFUNDIDAD	COSTO UNIFORME	VORAZ	A*
		T/O	T/O	T/O	T/O	T/O
Problema A: prof 10	CT	23423	9726	7942	6412	6375
	CE	$O(3^{3+1})/19$	O(3*10)/75	$O(3^{(166.56/E)})/15$	$O(4^{10})/11$	$O(3^{10})/10$
Problema B: prof 18	CT	6679.0	8240.0	6851.0	24700.0	5887.0
	CE	$O(4^{15+1})/1273$	O(4*18)/978	$O(3^{1125.53/E})/558$	$O(4^{10})/3339646$	$O(3^{18})/87$
Problema C: prof 50	CT	14001.0	5688.0	5232.0	No encuentra solución	4574.0
	CE	$O(4^{27+1})/28722$	O(4*50)/13686	$O(4^{(2145.50/E)})/2341$	No encuentra solución	$O(4^{50})/658$

T: Teórica
O: Obtenida
CT: Complejidad Temporal
CE: Complejidad Espacial

Diagrama de la Complejidad Temporal Obtenida:



En el caso del problema con complejidad pequeña (en el cual se utiliza un solo objetivo y la profundidad máxima es 10) se puede observar que la búsqueda con complejidad espacial mayor es la búsqueda en anchura, y el resto tienen una complejidad espacial parecida que ronda entre los 6000 y 9000 nodos aunque se puede observar que la complejidad espacial más pequeña es la de A*. En el problema con

complejidad media (en el cual se utilizan dos objetivos y la profundidad máxima es 18) se puede observar que el que tiene mayor complejidad espacial es el voraz y la que menos es la A*. Por último, en el problema con complejidad grande (en el cual se utilizan 3 objetivos y la profundidad máxima es de 50) el algoritmo en voraz no encuentra solución ya que es una búsqueda que no es completa y se puede dar el caso que según la complejidad del problema se quede atascada en un bucle.
En los tres tipos de problemas se puede observar que la búsqueda A* es la que tiene menor complejidad espacial ya es un algoritmo que es completo y a la vez óptimo, por lo tanto debe de sacar siempre la mejor solución posible.

Realizamos todas las búsquedas con poda:

- **ANCHURA**

PROBLEMA GRANDE (C):	PROBLEMA MEDIANO (B):	PROBLEMA PEQUEÑO(A):
Ramificación = 4	Ramificación = 4	Ramificación = 3
Prof = 27	Prof =15	Prof = 3
Costo = 2555.97	Costo = 1361.53	Costo =166.5381

- **PROFUNDIDAD**

PROBLEMA GRANDE (C):	PROBLEMA MEDIANO (B):	PROBLEMA PEQUEÑO(A):
Ramificación = 4	Ramificación = 4	Ramificación = 3
Prof = 50	Prof =18	Prof = 3
Costo = 4141.21	Costo = 1462.24	Costo = 166.5381

- **COSTO UNIFORME**

PROBLEMA GRANDE (C):	PROBLEMA MEDIANO (B):	PROBLEMA PEQUEÑO(A):
Ramificación = 4	Ramificación = 3	Ramificación = 3
Prof = 35	Prof = 17	Prof = 3
Costo = 2145.50	Costo = 1125.53	Costo = 166.538

- **VORAZ**

PROBLEMA GRANDE (C):	PROBLEMA MEDIANO (B):	PROBLEMA PEQUEÑO(A):
(Se queda pillado)	Ramificación = 4 Prof = 15 Costo = 1392.08	Ramificación = 4 Prof = 3 Costo = 166.5381

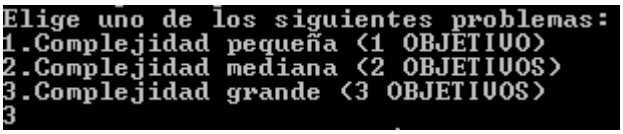
- **A***

PROBLEMA GRANDE (C):	PROBLEMA MEDIANO (B):	PROBLEMA PEQUEÑO(A):
Ramificación = 4 Prof = 35 Costo = 2145.50	Ramificación = 3 Prof = 17 Costo = 1125.53	Ramificación = 3 Prof = 3 Costo = 166.5381

4. MANUAL DE USUARIO

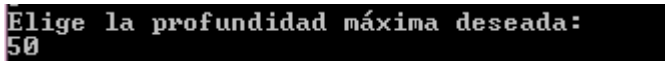
1. Para ejecutarlo abrimos la consola y nos metemos a la carpeta donde se encuentra el ejecutable y ahí ponemos java -jar NombreEjecutable.jar

2. A continuación, te pide que elijas uno de los tres problemas posibles (Se debe de elegir un dato numérico entre el 1 y el 3 según la opción deseada, en el caso de poner cualquier otra cosa te pedirá que introduzcas de nuevo su elección).

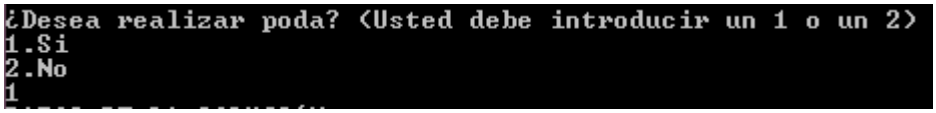


3. Una vez elegido el problema hay que elegir la estrategia que se desea aplicar (se debe elegir un número entre el 1 y el 5)

4. Después te pide que introduzcas la profundidad máxima del problema deseada, donde puedes introducir cualquier número (de forma razonable para que salga algo coherente)



5. A continuación, se elige si se quiere realizar poda o no (Si ponemos un 1 es que si, si pulsamos un 2 es que no).



6. Después de esto te mostrará la solución donde aparecerá la profundidad, el valor y el costo de la solución, además de la complejidad espacial, temporal y la ramificación del árbol de búsqueda.

7. Por último, si se quiere visualizar la ruta generada por la solución se hará mediante el fichero gpx creado y guardado en el proyecto mediante un visualizador web.

Para este ejemplo de uso de nuestro programa y los tres tipos de problemas con distintas complejidades hemos usado un mapa propio no el otorgado como ejemplo por el profesor.

5. OPINIÓN PERSONAL

Esta práctica nos ha servido para profundizar y entender mejor la asignatura de sistemas inteligentes así como llevarla a la práctica, pero aparte también nos ha hecho mejorar en muchos otros aspectos no directamente relacionados con la asignatura.

Es muy interesante para saber manejar algo tan cotidiano como buscar rutas en un mapa. Esta práctica también nos ha ayudado mejorar nuestras habilidades de programación sobre todo en la eficiencia del código, un ejemplo de ello es la incorporación de tablas hash para la implementación de grafos para mejorar la eficiencia.

Gracias a Open Steet Map hemos podido comprenderlas estructuras de datos que utilizan los mapas y hemos aprendido a utilizar la API de OSM para Java.

6. APÉNDICE

File: EntregaFinal.zip
File size: 8,68 MB (9.098.238 bytes)
MD5 checksum: C688105F6A61A815A1EA2F228F7E853A