

Documentación Práctica Sistemas Inteligentes

Juan Carlos Fernández Durán

December 6, 2014

Código fuente y Documentación realizada por Juan Carlos Fernández Durán

Contents

1	Entrega 1	2
1.1	Descripción del Problema	2
1.2	Lenguaje de Desarrollo	3
1.3	Implementación de Estado	3
1.4	Implementación de Espacio de Estados	4
1.5	Implementación de Problema	5
1.6	Implementación de Solución	6
1.7	Ejemplo de ejecución	6
2	Entrega 2	6
2.1	Implementación de Nodo de Arbol	6
2.2	Implementación del contenedor Nodos de Búsqueda	8
2.3	Comprobar la capacidad del sistema	9
2.3.1	Rendimiento de la aplicación	9
3	Entrega 3	10
3.1	Función de Búsqueda	10
3.2	Estrategia de búsqueda	13
3.2.1	Coste	13
3.2.2	Heurística	13
3.2.3	Selección de Valoración	14
4	Entrega 4	14
4.1	Optimización	14
5	Manual de Usuario	16
6	Conclusiones	17

1 Entrega 1

1.1 Descripción del Problema

Nuestro problema será el de crear un camino mínimo entre 2 puntos teniendo en cuenta el desnivel de altitud entre los puntos, con información de dicha altitud cada 25 metros, es decir, nuestro problema estará representado por una malla de casillas de 25 metros cuadrados que estarán definidos por una altitud.

También se tendrá en cuenta para el problema los cambios de dirección realizados. Para los ejemplos tomados en esta práctica tomaremos como referencia los Montes de Toledo, tomando las páginas correspondientes, siendo estas las 711, 712, 736 y 737, el programa podrá soportar sin embargo N mapas, sin ninguna restricción sobre si estos deben ser contiguos o no.

Se tomará la información de los archivos ZIP proporcionados para los mapas, una vez recuperada dicha información se estructurará y se guardará en archivos .h5, funcionalidad proporcionada por la librería HDF5, de modo que en los posteriores accesos al programa, se recuperará la información proporcionada por los mapas de archivos binarios, incrementando la eficiencia drásticamente

1.2 Lenguaje de Desarrollo

Dado que el lenguaje de programación es libre, he decidido optar por el lenguaje de programación C++ por 3 principales motivos.

El primero, es un lenguaje de programación eficiente, de más bajo nivel, esto nos permite gestionar la memoria, y nuestro código podrá estar mejor optimizado.

Segundo, el lenguaje utiliza el paradigma de Orientación a objetos, este paradigma nos permite resolver el problema con un nivel de abstracción fácil de entender.

Tercero, me quiero dedicar al desarrollo de videojuegos, y en la industria se utiliza principalmente este lenguaje, por lo tanto me interesa practicar el máximo posible con dicho lenguaje y habituarme a el, ya que, muy probablemente, será el lenguaje con el que tenga que tratar en mi día a día laboral.

1.3 Implementación de Estado

Estado está definido como una clase la cual toma como variables la página, la coordenada X [columna], coordenada Y [fila], la altura a la que se encuentra dicha ubicación, y la acción que hemos realizado para llegar a dicho estado.

```
1 class State{
```

```

2
3     int page, x, y;
4     float height;
5     std::string action;
6
7 public:
8
9     State(int n_page, int n_x, int n_y, int n_height, std::string &string);
10    State()
11    : page(0), x(0), y(0), height(0), action(*new std::string(""))
12    {
13    }
14
15
16    int getPage();
17    void setPage(int newPage);
18
19    int getX();
20    void setX(int newX);
21
22    int getY();
23    void setY(int newY);
24
25    float getHeight();
26    void setHeight(float newHeight);
27
28    std::string getString();
29    void setString(std::string &newString);
30
31
32 };

```

Listing 1: State.h

Esta clase tendrá sus respectivas funciones de consulta y constructor.

1.4 Implementación de Espacio de Estados

Espacio de Estados será devuelto como un vector de Sucesores, los cuales contienen los Estados, que será generado por la clase Problema, ya que dicha clase será la que obtenga los datos necesarios para generar los sucesores.

Esta clase "Sucesores" que contiene el estado encapsula toda la meta-información de los estados para la realización de nuestro problema, a continuación la especificación de la clase Sucesores

```

1 class Sucesor {
2     State* newState;
3     std::string action;
4     float absoluteSlope;
5     bool changeDirection;

```

```

6
7 public:
8
9     Sucessor(State &state, std::string &_action, float slope, bool _direction
10    );
11     Sucessor(State *state, std::string &_action, float slope, bool _direction
12    );
13
14     ~Sucessor();
15
16     State getState();
17     State* p_getState();
18     float getSlope();
19     std::string getAction();
20     bool getChangedDirection();
21 };

```

Listing 2: Sucessor.h

1.5 Implementación de Problema

Tiene sentido que Problema sea la clase que encapsule la mayor parte de la funcionalidad del programa, por tanto tendrá toda la información necesaria para la generación de sucesores [Espacio de estados], y resolución de nuestro problema [obtener solución].

```

1 class Problem{
2
3     GeographicalMap *currentMap;
4
5     State initialState;
6     State objective;
7     Frontera frontier;
8     std::vector<State> solution;
9     bool optimization;
10
11 public:
12
13
14     Problem(State &n_initialState, State &n_objective, GeographicalMap &
15     n_currentMap, bool n_optimization);
16
17     State getInitialState();
18     void setInitialState(State &n_initialState);
19
20     State getObjective();
21     void setObjective(State &n_objective);
22
23     std::vector<Sucessor>* followingStates(Sucessor &father);
24
25     std::vector<State*> getSolution(NodoArbolBusqueda* hijo);

```

```

25
26     void stressTest();
27     std::vector<State*> search(std::string estrategia, int max_prof, std::
vector<unsigned long*> metadata);
28     bool target(State *state);
29
30 };

```

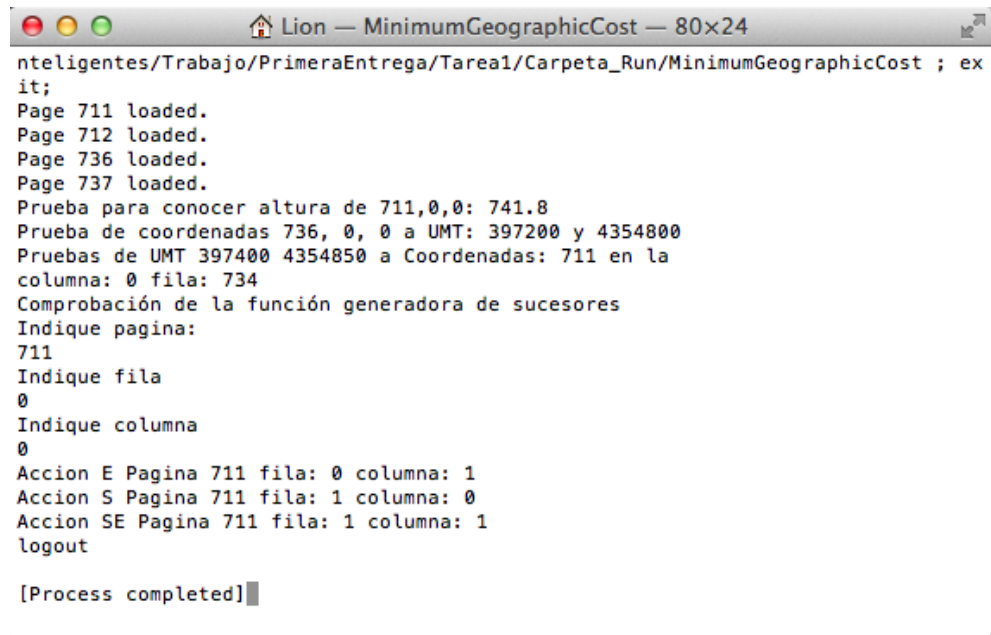
Listing 3: Sucessor.h

Nuestro problema estará compuesto de la información de nuestro mapa, con un puntero a clase GeographicalMap la cual nos proporcionará toda la información necesaria sin preocuparnos de la gestión de dichos datos. Nuestra clase problema también contendrá un Estado inicial en el que nos encontremos y un Estado objetivo, la comparación de si hemos llegado a nuestro objetivo se determinará mediante la función *taget(State)* sobre el que a partir de un estado determinaremos si hemos llegado a nuestro objetivo o no.

1.6 Implementación de Solución

Será generada como un vector de Estados, que nos será devuelto por la clase Problema. Dicho vector debe de estar compuesto por un conjunto de estados formando un camino cuyo primer estado del vector sea nuestro estado inicial y la última posición del vector sea el estado final.

1.7 Ejemplo de ejecución



```
Lion - MinimumGeographicCost - 80x24
nteligentes/Trabajo/PrimeraEntrega/Tarea1/Carpeta_Run/MinimumGeographicCost ; ex
it;
Page 711 loaded.
Page 712 loaded.
Page 736 loaded.
Page 737 loaded.
Prueba para conocer altura de 711,0,0: 741.8
Prueba de coordenadas 736, 0, 0 a UMT: 397200 y 4354800
Pruebas de UMT 397400 4354850 a Coordenadas: 711 en la
columna: 0 fila: 734
Comprobación de la función generadora de sucesores
Indique pagina:
711
Indique fila
0
Indique columna
0
Accion E Pagina 711 fila: 0 columna: 1
Accion S Pagina 711 fila: 1 columna: 0
Accion SE Pagina 711 fila: 1 columna: 1
logout

[Process completed]
```

2 Entrega 2

2.1 Implementación de Nodo de Arbol

```
1 class NodoArbolBusqueda {
2
3     unsigned long id;
4     NodoArbolBusqueda* father;
5
6     /*Information*/
7     State* actualState;
8     std::string action;
9     int deep;
10    double valoration;
11    float coste;
12    float absoluteSlope;
13    float heuristic;
14    int f_uml_x;
15    int f_uml_y;
16    bool changedDirection;
17
18 public:
19
```

```

20     NodoArbolBusqueda(unsigned long name, Sucessor& n_sucessor ,
    NodoArbolBusqueda* n_father , std::string estrategia , GeographicalMap* map
    );
21     NodoArbolBusqueda(unsigned long name, Sucessor& n_sucessor ,
    NodoArbolBusqueda* n_father , std::string estrategia , GeographicalMap* map
    , State* objective);
22     int getDeep();
23     double getValoration() const;
24     State* getState();
25     std::string getAction();
26     float getAbsoluteSlope();
27     bool getChangedDirection();
28     NodoArbolBusqueda* getFather();
29     float getCoste();
30     unsigned long getId();
31     inline std::string getHashName(){
32         std::ostringstream stringstream;
33         stringstream << actualState->getPage() << "-" << actualState->getX()
    << "-" << actualState->getY();
34
35         return stringstream.str();
36     }
37 };

```

Listing 4: NodoArbolBusqueda.h

La clase `NodoArbolBusqueda` engloba toda la meta-información necesaria, su nombre único y un apuntador a su padre. Luego tenemos la información concerniente al problema generada normalmente por el espacio de estados (función generadora de sucesores) entre la que tenemos el estado, cómo hemos llegado, profundidad del árbol, valoración [implementada en el constructor para que se genere aleatoriamente], peldaño acumulado y si se ha cambiado de dirección.

Tendremos además una función `getHashName` que nos servirá para identificar dicho nodo en un diccionario cuando apliquemos técnicas de optimización para A

2.2 Implementación del contenedor Nodos de Búsqueda

En este caso, he optado por crear una clase frontera, para que en el caso de necesitar una estructura de datos más eficiente u se diera el caso necesario, poder hacer los cambios pertinentes beneficiándome del des-acoplamiento de esta clase respecto del problema.

```

1 class Frontera {
2     std::queue<NodoArbolBusqueda*> tree;
3     std::priority_queue<NodoArbolBusqueda*, std::vector<NodoArbolBusqueda*>,
    LessValoration>* frontier;
4     std::map<std::string , double>* optimizationDictionary;
5
6 public:
7
8     Frontera();

```



```

9  ~Frontera();
10  double addNode(NodoArbolBusqueda *nodo, bool optimization);
11  NodoArbolBusqueda* getNode();
12
13  bool isEmpty();
14 };

```

Listing 5: Frontera.h

La clase Frontera encapsula una cola con prioridad, dicha clase permite añadir y obtener nodos. En la línea 2 se puede observar la creación de dicha cola con un algoritmo de ordenación propio llamado "LessValoration" que indica a la cola con prioridad cómo se deben ordenar, siendo el objeto Nodo Búsqueda con menor valoración el que mayor prioridad tendrá para ser obtenido.

Se ha elegido la estructura de datos de `priority_queue` de la biblioteca STL debido a que la inserción es de complejidad Logarítmica, ya que la estructura de datos está implementada internamente como un árbol binario, capaz de insertar elementos de forma ordenada muy eficientemente. La biblioteca STL es la biblioteca estándar de C++ cuya eficiencia ha sido probada y depurada numerosas veces, por lo tanto es adecuado utilizarla para nuestro problema. Otras posibles opciones pueden ser implementarnos nuestra propia cola con prioridad, o nuestro propio árbol binario ordenado, sin embargo, esta funcionalidad ya viene proporcionado por la propia librería Estándar de C++, cuyo propósito es de uso general, por tanto, por ahorro de tiempo y dada su excelente eficiencia, usaremos STL

Para nuestra optimización utilizaremos una estructura diccionario, indexada por cadenas String guardando un valor Double que será la valoración que tenemos para dicho nodo. Guardaremos la página y las coordenadas en formato String, a pesar de que el tratamiento de cadenas es algo ineficiente en tiempo de ejecución, nos permitirá un ahorro de tiempo muy grande para no explorar nodos que no nos conducirán a una buena solución

```

1  NodoArbolBusqueda* Frontera::getNode() {
2      NodoArbolBusqueda* value= frontier->top();
3      frontier->pop();
4      return value;
5  }

```

Listing 6: Frontera.cpp

Actualmente, la función `getNode()` elimina de la frontera el elemento, lo cual corresponde a sacar nuestro elemento de la frontera, y lo devolvemos para que pueda ser debidamente analizado por nuestro algoritmo de búsqueda.

2.3 Comprobar la capacidad del sistema

Se ha implementado una función de "estrés" cuyo único propósito es probar la capacidad del programa. A continuación se presenta el bucle principal de dicha función para probar

la capacidad del sistema.

```
1      for (std::vector<Sucesor>::iterator it = listSucesor->begin(); it !=
2          listSucesor->end(); ++it)
3      {
4          NodoArbolBusqueda* hoja = new NodoArbolBusqueda(++id, *it,
5          nodoPadre);
6          nanosec=frontier.addNode(hoja);
7          accumulated += nanosec;
8          if (nanosec < min)
9              min=nanosec;
10         if (nanosec > max)
11             max=nanosec;
12         if (id % 10000000 == 0){
13             std::cout << id << std::endl;
14             std::cout << (accumulated/(id+1)) << " de media" << std::endl
15         ;
16             std::cout << min << ": Min" << std::endl;
17             std::cout << max << ": Max" << std::endl;
18             std::cout << std::endl;
19         }
20     }
21     nodoPadre = frontier.getNode();
```

Listing 7: Problema.cpp

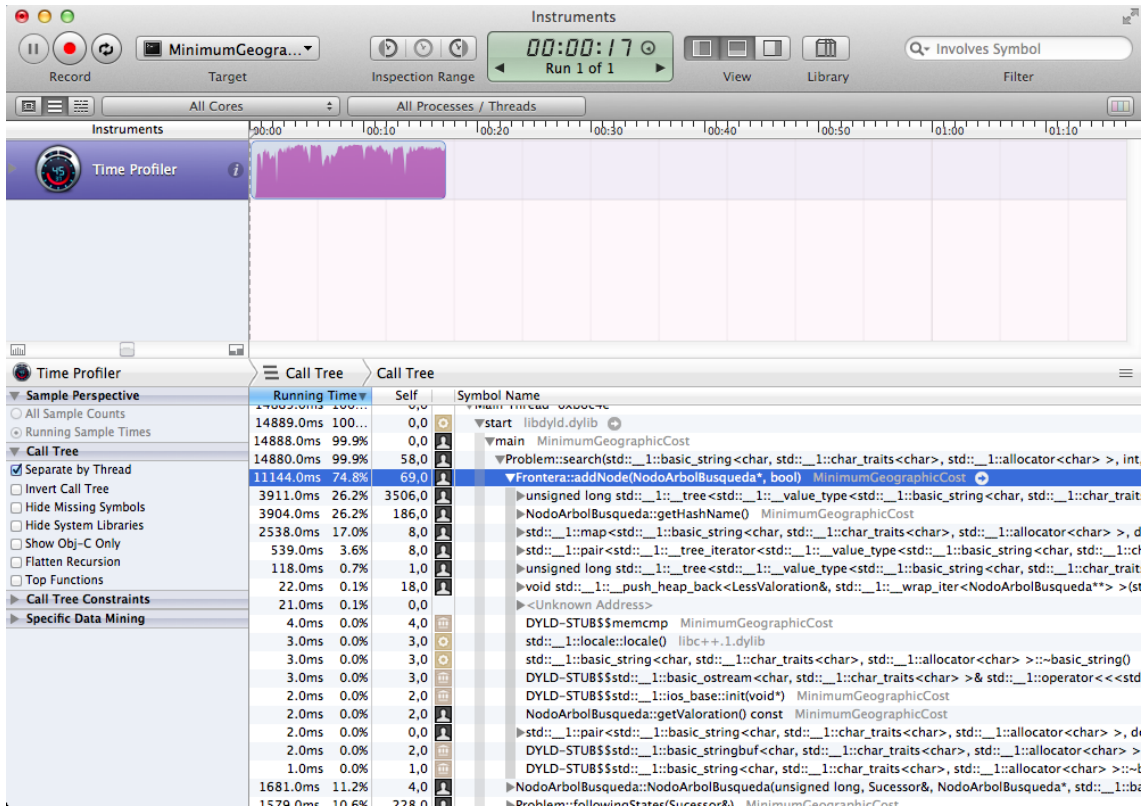
2.3.1 Rendimiento de la aplicación

A una cantidad de 150 millones de nodos, la herramienta que se encarga de medir el consumo de memoria del proceso, da un pico de bajada de 5-6GB a 2GB por tanto sospecho que, o bien de alguna forma conserva dicha información en disco o tan solo hace una limpieza de todos esos punteros que no se han llegado a utilizar.

Sin embargo, es a la cantidad de 550 millones de nodos cuando el propio sistema operativo me advierte de que el proceso está consumiendo demasiada memoria y de que debo removerlo para liberar al sistema, de forma que no he sido capaz de capturar la excepción como tal.

La condición de la línea 10 hará que las estadísticas se impriman cada 10 millones de nodos insertados, de manera que no suture nuestra consola de información.

Tras la ejecución he obtenido un tiempo medio de inserción de nodo de 589 nanosegundos, con 106 nanosegundos como mínimo y con 1.043e+10 nanosegundos como máximo.



Se puede observar en dicha herramienta de profiler que un 74,8% del tiempo de cómputo de nuestra aplicación corresponde únicamente a la función de añadir nodo a la frontera, de lo cual un 26% está destinado a la inserción del nodo de forma ordenada en la propia cola, y un 43% está destinado a la calcinación del nombre HASH y su inserción dentro del diccionario junto con su valor

3 Entrega 3

3.1 Función de Búsqueda

La implementación para la búsqueda en la clase Problema es la siguiente:

```

1 std::vector<State*> Problema::search(std::string estrategia, int max_prof, std
  ::vector<unsigned long>* metadata){
2
3     auto start = std::chrono::high_resolution_clock::now();
4     bool solution = false;
5     NodoArbolBusqueda* nodoActual = NULL;
6     std::vector<Sucesor> *listSucesor;
7
8     std::string cadnavacia("");

```

```

9      Sucesor root(initialState , cadenavacia , 0, false);
10     unsigned long id = 0;
11     unsigned long maxDeep = 0;
12     NodoArbolBusqueda *nodoPadre = new NodoArbolBusqueda(id , root , NULL,
13     estrategia , currentMap);
14     frontier.addNode(nodoPadre , optimization);
15
16
17
18     while (!solution && !frontier.isEmpty()){
19         nodoActual = frontier.getNode();
20         if (target(nodoActual->getState()))
21             solution = true;
22         else{
23             if (nodoActual->getDeep()<max_prof){
24                 std::string cadena=nodoActual->getAction();
25                 Sucesor sucesorNodoActual(nodoActual->getState() , cadena ,
26                 nodoActual->getAbsoluteSlope() , nodoActual->getChangedDirection());
27
28                 listSucesor = followingStates(sucesorNodoActual);
29
30                 if(nodoActual->getDeep()+1 > maxDeep){
31                     maxDeep= nodoActual->getDeep()+1;
32                 }
33
34                 for(std::vector<Sucesor>::iterator it = listSucesor->begin
35                 (); it != listSucesor->end(); ++it){
36                     NodoArbolBusqueda* hijo = new NodoArbolBusqueda(++id , *it
37                     , nodoActual , estrategia , currentMap , &objective);
38                     frontier.addNode(hijo , optimization);
39                 }
40             }
41
42
43
44     }
45     auto finish = std::chrono::high_resolution_clock::now();
46
47
48     if(solution){
49         metadata->push_back(nodoActual->getCoste());
50         metadata->push_back(nodoActual->getDeep());
51         metadata->push_back(id);
52         metadata->push_back(std::chrono::duration_cast<std::chrono::
53         nanoseconds>(finish-start).count());
54         metadata->push_back(maxDeep);

```

```

54     }
55
56     if (solution) {
57         return getSolution(nodoActual);
58     } else {
59         return std::vector<State*>();
60     }

```

Listing 8: Problema.cpp

El primer nodo insertado será el Raiz, creado sin padre, se añadirá a la frontera y se procederá a ejecutar el algoritmo de búsqueda

Para esta función se ha añadido un nuevo constructor a la clase `NodoArbolBusqueda` admitiendo la estrategia, el mapa actual [de tipo `GeographicalMap`] y el estado objetivo para poder calcular la función heurística dentro de dicha función.

El estado inicial es el primer nodo insertado en la frontera, después se da paso al algoritmo general, en el que comprobamos si existe una solución o si la frontera está vacía, extraemos un nodo de la frontera y comprobamos si es solución, si no lo es, comprobamos si nuestra profundidad está acotada y pasamos a generar los Sucesores a partir de la clase `Sucesor` [clase estado enriquecida de meta-información para nuestro problema], generamos los sucesores de nuestro nodo y a continuación recorreremos todos esos sucesores para crear nodos del árbol, donde se calculará el coste de acceder a dicho nodo, su función heurística y su valoración dependiendo de la estrategia escogida.

La insertaremos en la frontera, y la frontera será la encargada de analizar si dicho nodo debe ser o no insertado en la frontera dependiendo de si se debe aplicar poda o no.

La llamada `getSolution` construirá la solución con la ayuda de un vector y una pila de manera que quedará ordenado un camino desde el estado inicial hasta el estado final.

Si no existe solución se devolverá una lista de sucesores vacía que indicará que no existe solución.

```

1  std::vector<State*> Problem::getSolution(NodoArbolBusqueda* hijo){
2      std::vector<State*> path = std::vector<State*>();
3      NodoArbolBusqueda* actualNode = hijo;
4      std::stack<NodoArbolBusqueda*> stack;
5
6      while (actualNode->getFather() != NULL){
7          stack.push(actualNode);
8          actualNode = actualNode->getFather();
9      }
10     stack.push(actualNode);
11
12     while (!stack.empty()){
13         actualNode= stack.top();
14         stack.pop();
15         path.push_back(actualNode->getState());
16     }

```

```

17
18
19     return path;
20 }

```

Listing 9: Problema.cpp

La inserción de la línea 10 se realizará para que el nodo raíz también aparezca en la solución

3.2 Estrategia de búsqueda

A continuación, se mostrará distintas secciones del constructor NodoArbolBusqueda para mostrar cómo se asigna el coste, la heurística y finalmente una valoración en función de la estrategia elegida.

3.2.1 Coste

```

1     if (n_father != NULL){
2         if (action.compare("N") == 0 || action.compare("S") == 0 ||
3             action.compare("E") == 0 || action.compare("O") == 0)
4             ld = 25;
5         else
6             ld = sqrtf(2*25*25);
7
8         coste = sqrtf(ld*ld+absoluteSlope*absoluteSlope);
9         if (changedDirection)
10            coste = coste * 1.1;
11
12            coste = coste + n_father->getCoste();
13    } else
14        coste=0;

```

Listing 10: NodoArbolBusqueda.cpp

Si la acción realizada es Norte, Sur, Este o Oeste el coste será 25, de lo contrario será un movimiento a NW, NE, SW, SE, de modo que será la raíz cuadrada de 2 veces 25

Más tarde, utilizamos esa distancia junto con la diferencia de altura para calcular la longitud real de un paso a otro, con la correspondiente penalización del 10% más si cambiamos de dirección

3.2.2 Heurística

```

1     f_uml_x = umdState->at(0);
2     f_uml_y = umdState->at(1);
3
4     int xdxn =abs(( umdObjective->at(0)-umdState->at(0)));
5     int ydyn =abs(( umdObjective->at(1)-umdState->at(1)));
6

```

```

7   int cuadX = xdxn * xdxn;
8   int cuadY = ydyn * ydyn;
9
10  heuristic = sqrt(cuadX+cuadY);

```

Listing 11: NodoArbolBusqueda.cpp

A partir del estado actual y el estado objetivo, obtenemos las coordenadas UMT gracias a la función coordToUMT de nuestro GeographicalMap, con esta distancia UMT obtenemos la distancia euclídea

3.2.3 Selección de Valoración

```

1   if(estrategia.compare("profundidad") == 0){
2       valuation= 1/deep;
3   }else if(estrategia.compare("anchura") == 0){
4       valuation= deep;
5   }else if(estrategia.compare("uniforme") == 0){
6       valuation= coste;
7   }else if(estrategia.compare("voraz") == 0){
8       valuation= heuristic;
9   }else if(estrategia.compare("A") == 0){
10      valuation= coste+heuristic;
11  }

```

Listing 12: NodoArbolBusqueda.cpp

A partir de una determinada estrategia, establecemos una valoración determinada que será utilizada como criterio de ordenación en la frontera.

4 Entrega 4

4.1 Optimización

Se ha incorporado una estructura de tipo diccionario para no incluir nodos en la frontera de los que tengamos una valoración para los mismos menor del que queremos insertar, de esta forma eliminamos la posibilidad de explorar caminos que no ayudan proporcionar una mejor solución. Aplicando esta poda optimizaremos mucho más el algoritmo ya que acotamos mucho más la bifurcación del árbol

```

1   double Frontera::addNode(NodoArbolBusqueda *node, bool optimization){
2       double valuation = node->getValoration();
3
4       if (!optimization)
5           frontier->push(node);
6       else if (optimizationDictionary->count(node->getHashName())){
7
8
9           if(valoration < 0) //Profundidad
10              valuation = -valuation;
11

```

```

12         if (optimizationDictionary->at(node->getHashName()) > valoration){
13             optimizationDictionary->erase(node->getHashName());
14             optimizationDictionary->insert ( std::pair<std::string, double>(
node->getHashName(),
15
16             valoration) );
17             frontier->push(node);
18         }
19
20     }else{
21         frontier->push(node);
22         optimizationDictionary->insert ( std::pair<std::string, double>(node->
getHashName(),
23
24         valoration) );
25     }
26     return 0;
27 }

```

Listing 13: Frontera.cpp

Si la valoración es negativa quiere decir que la estrategia es en profundidad, de manera que pasaremos dicho valor a positivo para que nuestra poda siga siendo efectiva en esta estrategia

Primero comprobamos mediante nuestra función Hash si el nodo está previamente insertado, si no es la primera vez comprobaremos que valoración es mejor para insertar el nuevo o no insertarlo. La función `node->getHashName()` está implementada como una función "inline" para ayudar a la eficiencia del código

5 Manual de Usuario

```

Page 711 loaded.
Page 712 loaded.
Page 736 loaded.
Page 737 loaded.
Para el estado Inicial
Indique UMT_X
397388
Indique UMT_Y
4353788
Para el estado Final
Indique UMT_X
399690
Indique UMT_Y
4360861
Indique estrategia a aplicar para resolver

```


Primero, se le indicarán las páginas UTM que han sido cargadas satisfactoriamente, a continuación se le pedirá las coordenadas para resolver el problema, es decir, el estado inicial (Donde se encuentra usted) y el estado final (Donde quiere llegar)

Deberán ser especificadas en el formato UTM, primero para X y luego para Y, más tarde se le solicitarán los mismos datos para el estado final.

Luego se le solicitará que indique una estrategia, debe escribir alguna de las siguientes opciones

- **profundidad**
- **anchura**
- **uniforme**
- **A**
- **voraz**

Se recomienda la utilización de las estrategias A y uniforme, ya que proporcionan las mejores soluciones en un tiempo aceptable, profundidad puede tardar mucho más tiempo y sin embargo no encontrar una buena solución, voraz, dado el funcionamiento de la heurística, tan solo le indicará un camino en línea recta, ignorando el desnivel y el cambio de sentido.

```
Indique UTM_X
397388
Indique UTM_Y
4353788
Para el estado Final
Indique UTM_X
399690
Indique UTM_Y
4360861
Indique estrategia a aplicar para resolver
A
Indique profundidad maxima
1000000
Indique opción:
1.-Optimización activa
0.-Optimización inactiva
```

Más tarde se le preguntará por la profundidad máxima que puede alcanzar el árbol, se recomienda un número elevado salvo para la estrategia de profundidad, en la cual conviene proporcionar una profundidad acotada y suficiente para encontrar la solución. Dicha profundidad se puede saber calculando la distancia euclídea del estado inicial y final, y dividir dicha distancia entre 25.

Se le preguntará si desea que se optimice el algoritmo, se le recomienda optimizarlo ya que reduce drásticamente el tiempo de búsqueda de la solución.

Una vez seleccione la opción de optimización el programa comenzará a ejecutar, la duración de esta operación puede variar en función de la estrategia, tardando una solución 12 segundos para uniforme, puede tardar 0,1 segundos para voraz.

Una vez calculada la solución se imprimirá por consola y escrito en un fichero aparte junto al ejecutable la información referente a la solución proporcionando los puntos que se deben seguir paso por paso para alcanzar la solución. En caso de no haberse encontrado se indicará también por consola y en el fichero correspondiente.

6 Conclusiones

Es importante conocer de primera mano lo costoso que pueden llegar a ser estas estrategias y el camino que puede llegar a ahorrar tener el suficiente conocimiento del problema como para proporcionar una heurística efectiva. Ha sido muy instructivo contar con la oportunidad de luchar de primera mano con estos algoritmos cuya complejidad puede ser excesivamente grande si no se aplica una buena poda.

El uso efectivo de los recursos es esencial, en nuestro caso, el tiempo y la memoria.

La programación de estos algoritmos tampoco es una tarea trivial ya que en este proyecto de importante embergadura me ha resultado especialmente complicado encontrar varios errores de programación que hacían que la lógica del problema cambiara drásticamente, por ejemplo, la decisión de qué clave hash utilizar podía generar bucles en búsquedas de A, o el uso de entero para guardar la información de valoración repercutía en una pérdida de información que se veía reflejado en la lógica de todo el algoritmo de búsqueda.