Problemas en la implementación de algoritmos de *routing* de alta complejidad en dispositivos móviles: el caso Itiner@

Laia Descamps-Vila[(1)], Joan Casas[(2)], Jordi Conesa[(2)],A.Pérez-Navarro[(2)]

[(1)] I.C.A. Informática y Comunicaciones Avanzadas, S.L., C/ Almogàvers 107-119, 08018 Barcelona, ldescamps@grupoica.com.
[(2)] Estudis d'Informàtica, Multimèdia i Telecomunicació, Universitat Oberta de Catalunya, Rambla Poblenou 156, 08018 Barcelona, [jcasasrom|jconesac|aperezn]@uoc.edu.

RESUMEN

La aparición de terminales de telefonía móvil cada vez más potentes abre un nuevo abanico de posibilidades en cuanto a usos y aplicaciones. Sin embargo, y dadas las limitaciones tanto de memoria como de CPU que tienen estos dispositivos, algunas de las aplicaciones potenciales resultan muy difíciles o incluso imposibles de llevar a la práctica. Este es el caso, por ejemplo, de aplicaciones de cálculo de rutas.

En el contexto del proyecto Itiner@, un asistente para rutas turísticas completamente autónomo que debe funcionar incluso sin conexión a internet, todos los procesos deben ejecutarse íntegramente de forma local en el dispositivo móvil. Dado que es un proyecto orientado al ocio, es importante que la experiencia del usuario sea satisfactoria, por lo que además de poder ejecutar el algoritmo de cálculo de rutas, el sistema debe hacerlo de forma rápida. En este sentido, los algoritmos recursivos habituales son demasiado costosos o lentos para su uso en Itiner@ y ha sido necesario reinventar este tipo de algoritmos en función de las limitaciones que tienen estos dispositivos.

En el presente trabajo se presenta el proceso seguido y las dificultades encontradas para implementar un algoritmo recursivo de cálculo de rutas que se ejecute íntegramente en un dispositivo móvil Android de forma eficiente. Así, finalmente se llega a un algoritmo recursivo de cálculo de rutas para dispositivos móviles que se ejecuta de forma más eficiente frente a algoritmos directamente portados a dispositivos móviles.

La principal contribución del trabajo es doble: por un lado ofrece algunas guías útiles al desarrollo de algoritmos más eficientes para dispositivos móviles; y por el otro, muestra un algoritmo de cálculo de rutas que funciona con un tiempo de respuesta aceptable, en un entorno exigente, como es el de las aplicaciones de turismo en móviles.

Palabras clave: *Personalización, routing, Android.*

ABSTRACT

The emergence of smartphones increasingly powerful opens a new range of possibilities in terms of uses and applications. However, given their limitations of both memory and CPU, some of the potential applications are very difficult or even impossible to implement. This is the case, for example, of the route calculation applications.

In the context of Itiner@ project, a route assistant completely autonomous that should work even without internet connection, all processes must be fully implemented locally on the mobile device. Since it is a project focused on leisure, it is important to get a satisfactory user experience, so besides being able to execute the route calculation algorithm, the system must do it quickly. In this sense, the common recursive algorithms are too expensive regarding memory or to slow for use in Itiner@ and it has been necessary to develop new algorithms depending on the limitations of these devices.

This paper presents the process followed and difficulties encountered in implementing an algorithm to calculate routes that run entirely on a mobile device efficiently. So finally we get to a recursive algorithm for calculating routes within mobile devices that run more efficiently compared to algorithms directly ported to mobile devices.

The main contribution of this paper is double: on the one hand provides some useful guidance to the development of more efficient algorithms for mobile devices and, on the other hand, shows a routing algorithm that works with an acceptable response time in a demanding environment, as is that of tourism in mobile applications.

Key words: *Personalization, routing, Android*

INTRODUCCIÓN

Los dispositivos móviles se están convirtiendo día a día en dispositivos muy potentes, que están dando más oportunidades para desarrollar aplicaciones que eran inimaginables hace apenas unos años. Sin embargo, estos dispositivos tienen algunas limitaciones de hardware que pueden hacer que algunas ideas sean difíciles o incluso imposibles de implementar. Debido a la falta de velocidad de la CPU y a la memoria limitada de estos dispositivos, hay algunos proyectos que deben revisarse a fin de poderse utilizar en dispositivos móviles. Es el caso de cualquier proyecto que implique el uso de un algoritmo complejo que se deba ejecutar en el propio dispositivo, como ocurre en nuestro caso de estudio: Itiner@. [1] [2] [3]

Itiner@ es una aplicación móvil que ofrecer rutas turísticas adaptadas a cada visitante según sus preferencias, las condiciones de la visita y el lugar visitado. A través de esta información, la aplicación obtiene una base de conocimiento con la que trabaja para planificar rutas personalizadas de acuerdo a estas preferencias. Dado que la aplicación está pensada para utilizarse tanto en zonas urbanas como rurales, el usuario se puede encontrar en una zona donde no hay conexión a Internet disponible. Así, teniendo en cuenta este hecho, la aplicación está preparada para funcionar sin conexión a Internet.

Para generar rutas turísticas personalizadas, se necesitan dos tipos de algoritmos:, un algoritmo de personalización que debe seleccionar los puntos de interés (POIs) más atractivos para el usuario según las condiciones de un momento determinado en

un sitio determinado para un usuario concreto; y , un algoritmo de *routing* que debe guiar al visitante a través de las calles del pueblo o ciudad a través del camino más corto, con el fin de visitar todos los puntos seleccionados por el algoritmo anterior.

Además, Itiner@ debe ser capaz de reaccionar a las acciones del usuario y, por ejemplo, planificar una nueva ruta a seguir si el usuario ha aparcado el coche lejos del punto de inicio de la ruta; o si el usuario, simplemente, no quiere visitar uno de los puntos planificados. En este caso Itiner@ vuelve a calcular la ruta para omitir el punto en cuestión. Esto significa que el móvil debe ser capaz de utilizar los algoritmos anteriormente citados junto con el almacenamiento de información en el propio dispositivo.

Para el caso del algoritmo de *routing*, versiones portadas al móvil de otros algoritmos recursivos no responden bien a las necesidades de los usuarios potenciales. Su ejecución necesita demasiado tiempo como para considerarse una opción aceptable en una aplicación real, donde el usuario debe obtener una experiencia satisfactoria.

Debido a las necesidades de CPU y a la limitación de memoria, este tipo de algoritmos recursivos de alta complejidad deben ser reinventados. Este artículo se basa en un caso real, la aplicación Itiner@, donde se ha desarrollado e implementado con éxito un algoritmo recursivo de *routing* junto con un algoritmo de personalización. En el artículo se describe el diseño y funcionamiento de ambos. Los dos se han integrado y testeado en un dispositivo Android, y se demuestra que tienen la eficiencia suficiente como para formar parte de una aplicación de usuario final que se puede ejecutar en el dispositivo sin el apoyo de Internet.

DISEÑO DE LOS ALGORITMOS

En este apartado se detalla cómo se han diseñado los algoritmos y sus características junto con las mejoras y limitaciones que hemos tenido que implementar para poder usarlos eficientemente en un dispositivo móvil. Es importante notar que el hecho de implementarlos en una aplicación final, hace que la parte de eficiencia tome especial relevancia.

- Algoritmo de personalización

Como se ha comentado anteriormente, el algoritmo de personalización interacciona con el usuario y selecciona los mejores puntos de interés para visitar según el momento, el lugar y el propio visitante. Este algoritmo se basa en dos puntos clave que marcan el diseño del mismo:

1) Sólo se ofrecen rutas viables, es decir, sólo se sugiere una visita si el usuario puede llevarla a cabo. Por ejemplo, no ofrece visitar un lugar que estará cerrado cuando el usuario llegue a visitarlo.

2) La ruta propuesta no propone necesariamente que el usuario visite el máximo número de puntos, ya que puede querer visitar menos puntos, pero que le resulten más interesantes. Así, dentro de las rutas viables siempre se sugiere la ruta más atractiva para el usuario.

Diseño del algoritmo

Para conseguir implementar las dos características anteriores, se ha diseñado un algoritmo iterativo. A continuación se detallan en profundidad los distintos pasos que se han seguido para conseguirlo.

Rutas viables

Esta primera característica se basa en obtener un conjunto de rutas factibles para un visitante en particular.

El primer paso es seleccionar un conjunto de POI teniendo en cuenta las preferencias del visitante: lamamos *n* la cantidad de puntos obtenidos. Con esta colección de puntos se genera un conjunto de rutas aleatorias: el algoritmo genera combinaciones sin repetición para ir creando distintos grupos de puntos, donde cada conjunto de puntos forma una ruta. Se crean combinaciones porque en este punto el orden de los puntos que forman la ruta no importa, ya que más adelante se van a ordenar los puntos según la distancia entre ellos. Además, no hay repetición porque se entiende que el usuario no desea visitar el mismo punto de interés dos veces.

Con todo esto, la cantidad de rutas posibles puede ser muy grande si inicialmente se han obtenido muchos POI. Así, se aplican algunos criterios para restringir la cantidad de rutas válidas. Primero se limita el número máximo (n_{max}) y mínimo (n_{min}) de puntos que puede tener una ruta según la duración de la misma, ya que en tres o diez horas no se pueden visitar la misma cantidad de sitios.

Para obtener n_{max} y n_{min} se hace una estimación de cuantos puntos se pueden visitar dependiendo del tiempo disponible. Se ha considerado que con una hora se pueden visitar dos puntos y con cuatro horas unos seis. A partir de esta estimación, se ha obtenido la ecuación 1, que nos da una buena estimación de n_{max} para cualquier tiempo. n_{min} se crea sólo para acotar el número de rutas.

$$n_{max}(t) = 0.822 * t + 0.73 \quad (1)$$

$$n_{min}(t) = n_{max}(t) - 2 \quad (2)$$

Una vez se ha acotado el número máximo y mínimo de puntos según el tiempo de ruta, se sabe la cantidad de rutas, es decir, combinaciones, que obtendremos. Si el número de POI iniciales es *n* y elegimos *s* número de puntos por ruta, donde el valor de *s* está limitado por los valores n_{max} y n_{min}, obtenemos un número de rutas *C(t)*. Así, el número de rutas aleatorias que pueden existir según el tiempo total de ruta *t,* son las que nos muestran les ecuaciones 3 y 4:

$$C(s) = \frac{n!}{s!(n-s)!} = \binom{n}{s} \quad (3)$$

$$C(t) = \sum_{s=n_{min}(t)}^{s=n_{max}(t)} C(s) \quad (4)$$

El valor de *C(t)* se limita a un máximo de 1.000, para mejorar la eficiencia.

El siguiente paso es ordenar los puntos de cada ruta según la distancia euclídea entre cada uno de ellos, haciendo que la distancia total de la ruta sea la mínima. Para ello se ha desarrollado una función tipo TSP (*Travelling Salesman Problem*), dónde se busca siempre el punto más cercano del otro.

Una vez hecho esto, cómo se ha comentado, cada ruta tiene que ser viable, así que tiene que cumplir con una serie de restricciones, las cuales se detallan a continuación:

1. Se descarta una ruta si el tiempo total que el visitante necesita para hacer la ruta (t_{ruta}) es superior al tiempo de que dispone el usuario ($t_{visitante}$).

   Restricción 1:

$$r(t) = \left\{ \begin{array}{ll} 1 & if \quad t_{ruta} \leq t_{visitante} \\ 0 & if \quad t_{ruta} > t_{visitante} \end{array} \right\} \quad (5)$$

Conociendo el orden de los POI se puede saber el tiempo total que el visitante necesita para realizar la ruta. Se calcula el tiempo que empleará el usuario para ir de un punto a otro, t_{viaje}, que depende del modo de transporte y la distancia entre puntos. Se suma este valor junto con el tiempo de visita para cada punto y se sabe aproximadamente el tiempo necesario para realizar toda la ruta. En la ecuación 6 se detalla este cálculo.

$$t_{ruta} = t_o + \sum_{i=0}^{s} t_{visita}(i) + \sum_{i=0}^{s-1} t_{viaje}(i, i+1) \quad (6)$$

Donde *s* es la cantidad de puntos por ruta, e *i* es cada punto de la ruta.

2. Se descarta una ruta si se encuentra un punto de interés cerrado cuando el usuario llega a visitarlo. Aplicando la ecuación 7 sabemos a qué hora llega el visitante a un punto concreto *j*, dónde *j* es la posición del punto de interés dentro de la ruta.

$$t_{llega}(j) = t_o + \sum_{i=0}^{j-1} t_{visita}(i) + \sum_{i=0}^{j-2} t_{viaje}(i, i+1) \quad (7)$$

A continuación, según el horario de apertura de cada punto, dónde t_{abrir} es la hora de apertura y t_{cerrar} es la hora de cierre, podemos aplicar la segunda restricción.

Restricción 2:

$$r(t,j) = \left\{ \begin{array}{lll} 1 & if & t_{abrir}(j) < t_{llega}(j) < t_{cerrar}(j) \\ 0 & if & t_{abrir}(j) > t_{llega}(j) \\ 0 & if & t_{llega}(j) > t_{cerrar}(j) \end{array} \right\} \quad (8)$$

3. Se descarta una ruta si no hay un sitio para comer a la hora que el visitante quiere comer. Esta restricción sólo se aplica si el usuario va con niños o si lo pide explícitamente. Aparte de la comida, también se ofrece la posibilidad de escoger el desayuno y la cena como obligatorios. Si el usuario termina la ruta antes del horario de comida, esta restricción no se tiene en cuenta.

Restricción 3:

$$r(t,j) = \left\{ \begin{array}{ll} 1 \ if & t_{inicio\_comida} < t_{llega}(j = poi_{tipo\_comida}) < t_{final\_comida} \\ 0 \ if & t_{inicio\_comida} < t_{llega}(j \neq poi_{tipo\_comida}) < t_{final\_comida} \\ 0 \ if & t_{abrir}(j) > t_{llega}(j = poi_{tipo\_comida}) \\ 0 \ if & t_{llega}(j) > t_{cerrar}(j = poi_{tipo\_comida}) \end{array} \right\} \quad (9)$$

Dónde poi_{tipo_comida} es un punto como un restaurante o un bar, identificado como sitio donde ofrecen de comer, t_{llega} es la hora que el visitante llega al punto y t_{inicio_comida} y t_{final_comida} son los horarios en los que la persona quiere comer.

Rutas atractivas

Una vez tenemos un conjunto de rutas factibles, se selecciona la más atractiva para el visitante. Se ordenan las rutas obtenidas anteriormente en función de distintos parámetros, cómo puntos de interés preferidos, tipo de comida y diversas puntuaciones de cada punto. Se calcula una función de coste con la suma de estos parámetros y cuanto mayor sea el valor, más atractiva se considera que es la ruta para el visitante. A continuación se detallan todos estos parámetros:

1. *Importancia media*. Cada POI tiene un peso según la media de valoración que le dan todos los visitantes. Para calcular el valor de la importancia media de toda la ruta, se suman los valores de todos los puntos de la ruta, según la ecuación 10.

$$c_{medio} = \sum_{i=0}^{z} poi_{puntuación\_media} \quad (10)$$

2. *Importancia intrínseca*. Dependiendo con quien viaja el usuario, se recomiendan unos POI u otros. La importancia intrínseca de la ruta se obtiene sumando el valor del peso de cada punto según quién lo acompaña, cómo se muestra en las ecuaciones 11 y 12.

$$poi_{puntuación\_intrínseca} = \left\{ \begin{array}{cc} poi_{coste\_familia} & if \quad usuario_{viaja} = familia \\ poi_{coste\_pareja} & if \quad usuario_{viaja} = pareja \\ poi_{coste\_amigos} & if \quad usuario_{viaja} = amigos \\ poi_{coste\_solo} & if \quad usuario_{viaja} = sólo \end{array} \right\} \quad (11)$$

$$c_{intrínseco} = \sum_{i=0}^{z} poi_{puntuación\_intrínseca} \quad (12)$$

3. *Puntos de interés preferidos*. El usuario decide qué tipo de puntos le gusta visitar, por ejemplo: histórico, arte, arqueológico, etc., que se definen como *tipos*. Luego, si la ruta contiene algún punto de este tipo, se le da una puntuación de 5 por cada punto que haya, cómo se representa en la ecuación 13.

$$c_{tipo\_punto}(i) = \left\{ \begin{array}{cc} 5 & if \quad poi_{tipo\_punto}(i) \in ruta \\ 0 & if \quad poi_{tipo\_punto}(i) \notin ruta \end{array} \right\} \quad (13)$$

$$c_{tipo} = \sum_{i=0}^{z} c_{tipo\_punto}(i) \quad (14)$$

4. *Sitios de comida preferida*. Se procede de la misma forma que con los puntos de interés preferidos.

$$c_{tipo\_comida}(i) = \left\{ \begin{array}{cc} 5 & if \quad poi_{tipo\_comida}(i) \in ruta \\ 0 & if \quad poi_{tipo\_comida}(i) \notin ruta \end{array} \right\} \quad (15)$$

$$c_{comida} = \sum_{i=0}^{s} c_{tipo\_comida}(i) \quad (16)$$

5. *Importancia general*. Hay puntos de interés que son de interés general aunque no sean de un tipo que es de interés particular para el usuario. Por ejemplo, un monumento como la Sagrada Familia se considera que es imprescindible para todos los usuarios. Así, a este tipo de POIs, se les da un peso extra según si son imprescindibles o interesantes, lo que está representado en la ecuación 17.

$$c_{interés\_general}(i) = \left\{ \begin{array}{cc} 5 & if \quad poi(i) = imprescindible \\ 3 & if \quad poi(i) = interesante \\ 0 & if \quad poi(i) = prescindible \end{array} \right\} \quad (17)$$

$$c_{general} = \sum_{i=0}^{s} poi_{interés\_general} \quad (18)$$

Con todo esto, el atractivo final de la ruta se obtiene según la ecuación 19 y se selecciona la que tiene el valor máximo.

$$c_{ruta} = c_{medio} + c_{intrínseco} + c_{tipo} + c_{comida} + c_{general} \quad (19)$$

- **Algoritmo de *routing***

En este punto, tenemos un conjunto de POI que forman una ruta y es necesario guiar al visitante para ir de un lugar a otro por el camino más corto. El trayecto de ir de un POI a otro está formado por un conjunto de nodos, que unidos forman pequeñas rectas que representan las calles del pueblo, como se puede ver en la Fig.1.

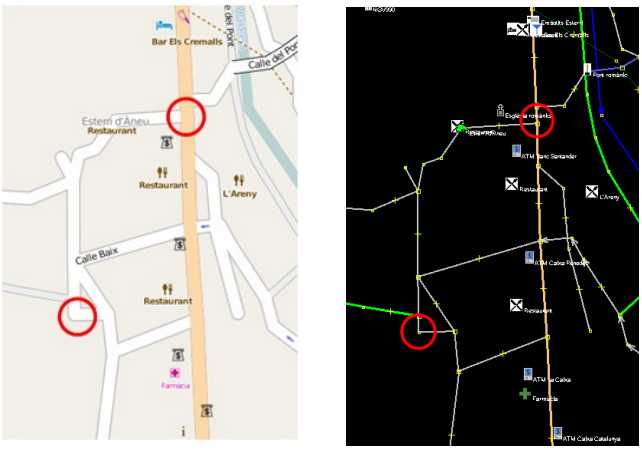

Figura 1: *A la izquierda, mapa raster de una zona de Esterri d'Àneu. A la derecha, mapa vectorial de la misma zona de Esterri d'Àneu, donde se pueden ver los nodos que forman las calles.*

Para encontrar este camino, se ha diseñado un algoritmo específico que se ejecuta en un dispositivo móvil y que se pueda implementar en una aplicación Android. Seguidamente se detallan las consideraciones que se han tenido en cuenta para optimizar el algoritmo.

Características del algoritmo

Camino óptimo. No se busca el camino óptimo, sino uno que sea aceptable. Como el algoritmo se va a usar en una aplicación de ocio, no es imprescindible que sea el camino óptimo, sino uno que lleve al visitante por un camino correcto sin dar vueltas.

Nodos visitados. No se repiten nodos. El algoritmo tiene en cuenta que sólo se pasa una vez por un nodo, ya que si no estaríamos en un camino cíclico o dando una vuelta para volver al mismo sitio.

Minimizar caminos. Se minimiza el número de caminos que no mejoran la solución actual. Cuando se trata de encontrar el mejor camino entre dos puntos, se busca reducir al mínimo la distancia entre ellos, así que una vez se ha encontrado un camino válido, se guarda la distancia de este camino. Luego, cuando el algoritmo está buscando un nuevo camino posible y éste se pasa de la distancia ya guardada, se puede asegurar que no es mejor que la solución guardada, por lo que se corta y se va a por otro intento. De esta manera, cada camino encontrado será mejor e incluso se reducirá el crecimiento innecesario del algoritmo, por lo que el camino será más pequeño para cada nueva solución.

Con todas estas restricciones, vemos que la mejor solución es diseñar un *backtracking* [4], ya que es un tipo de algoritmo que construye incrementalmente candidatos para las soluciones, y abandona cada posible candidato, es decir, retrocede, tan pronto como se determina que el candidato no puede ser completado con una solución válida.

Tratamiento y almacenamiento de datos

Aparte del tipo de algoritmo utilizado, los datos también son una parte muy importante para que el algoritmo sea eficiente. Los datos vectoriales, es decir, la información de los nodos que forman las calles, se obtienen de OpenStreetMap (OSM) en formato .xml. A continuación, a través de la librería Travelling Salesman [5] se transforman a una BD SQLite, ya que así se puede acceder a esta información con facilidad a través de una aplicación Android. El problema que hemos encontrado es que el algoritmo necesita consultar información de la BD constantemente y acceder a los datos que están en una BD SQLite es lento. Para corregir este problema se ha considerado un método de almacenamiento alternativo, con el fin de tener un acceso más rápido a los datos vectoriales del mapa. Los métodos más habituales para hacerlo, y los que se han considerados y probado, son los siguientes:

Matriz de adyacencia

Se ha diseñado una matriz cuadrada NxN, donde N es el número de nodos de la gráfica. Cada posición de la matriz *(i, j)* puede contener información sobre la conexión entre los nodos *i* y *j* (es decir, si pertenecen a la misma calle), o el valor de la distancia entre ambos. Este método ofrece una forma rápida de acceder a la información ya que es fácil de iterar. Por el contrario, el principal inconveniente es que consume una gran cantidad de memoria. La matriz no sólo almacena las conexiones entre cada par de nodos, sino también las no-conexiones entre los nodos, almacenando un 0 o *null*. Aunque esto no es un problema importante cuando se trabaja con gran cantidad de memoria disponible, sí que lo es cuando se trabaja con los dispositivos móviles. Un dispositivo Android sólo dispone de una pequeña cantidad de memoria disponible cuando se ejecuta una aplicación, 16MB.

Para poner un ejemplo, se ha creado una matriz a partir de los datos vectoriales de de la ciudad de Barcelona, que tiene alrededor de 49.000 nodos. Se han hecho pruebas en el ordenador analizando el espacio de memoria necesario para cargar la información en una matriz de adyacencia con diferentes tipos de datos. En la Tabla 1 podemos ver los resultados.

Tabla 1: Memoria consumida al cargar una matriz de adyacencia

	Bit por datatype	MB cargados
int array	32	> 9,000 MB
short array	16	> 4500 MB
byte array	8	> 2250 MB
boolean array	1	> 250 MB

Analizando los resultados, vemos que no hay ninguna opción viable para ponerlo en marcha en un dispositivo móvil, en la mayoría de los casos ni siquiera es viable para ejecutarse en un ordenador. La solución, entonces, es limitar el tamaño de los datos cargados desde la base de datos. Pero antes de eso, se ha comprobado si con listas de adyacencia es viable, ya que sería otra opción válida.

Lista de adyacencia

Con esta estructura de datos se puede evitar desperdiciar espacio de memoria con datos innecesarios, como son los 0 o *nulls* de la matriz de adyacencia. En este caso, sólo tendremos una lista de tamaño N, donde cada posición *i* hace referencia a los nodos con los que está conectado un nodo concreto. Sin embargo, aunque el acceso a la matriz de adyacencia es muy rápido, el acceso a la lista de adyacencia no es tan rápido. Para ahorrar espacio de memoria, una lista de adyacencia debe ser del tamaño exacto a las conexiones que tiene cada nodo. Así, se trata con listas de tamaños variables, que son más complicados de manejar que una matriz cuadrada simple.

De esta forma, las listas de adyacencia son más difíciles y más lentas de construir y acceder. El acceso a una matriz se realiza al instante, sólo es necesario especificar los índices de la posición a la que se accede. Por el contrario, acceder a una lista de tamaño variable requiere iterar secuencialmente sobre el conjunto de listas para saber qué nodos están conectados y cuáles no, lo que requiere más tiempo. Debido a esto, y debido a que la aplicación debe estar preparada para dar una experiencia satisfactoria para el usuario, elegimos usar una matriz y ganar velocidad de ejecución.

Así, volviendo a la matriz de adyacencia, ¿cómo podemos conseguir usarla y evitar los problemas de memoria?

Simplificaciones para trabajar en el móvil con matrices de adyacencia

Podemos encontrar dos formas de usar menos espacio de memoria: 1) reducir la cantidad de nodos, creando matrices más pequeñas, 2) usar datos que ocupen menos memoria. Para recudir la cantidad de nodos, se pueden aplicar dos métodos:

- *Parejas de nodos*. El algoritmo debe buscar la mejor ruta para pasar por todos los puntos de interés. Pero cómo puede haber muchos puntos por los que pasar

y eso puede depender mucho del tipo de ruta, puede llegar a ser un proceso muy costoso para el móvil. Para simplificarlo, se divide la ruta en parejas de POI, así si tenemos cinco puntos para visitar, {A, B, C, D, E} los agruparemos en parejas {A-B}, {B-C}, {C-D}, {D-E} y se buscara el camino para cada una de ellas en operaciones independientes. Esto reduce considerablemente la longitud del camino y consecuentemente, la cantidad de nodos por los que pasar en una misma operación.

- *Bounding Box*. Se selecciona un área geográfica reducida, que sólo incluya un rectángulo que contenga el POI inicial y final (ver fig. 2). Esto hace que el número de nodos se reduzca mucho y que la matriz sea mucho más pequeña. El tamaño del rectángulo está en función de la latitud y longitud entre el POI inicial y final. En el caso de que no se encuentre un camino posible porque este camino pase por fuera de la área seleccionada, se va ampliando el rectángulo un % determinado hasta que se encuentre un camino válido.

Por otro lado, para trabajar con datos que ocupen menos memoria, se puede renunciar a usar la latitud y longitud de los nodos que forman el camino, y evitar almacenar la distancia entre ellos. Así, si sólo se almacena la información de si están conectados entre ellos, se trabaja sólo con booleanos, y se evita usar números enteros, y como se ve en la Tabla 1, se reduce mucho la cantidad de memoria usada. Aplicando esto, se pierde la distancia entre nodos, por lo que se supone que ésta siempre es igual a 1. Esta aproximación no nos dará el camino óptimo, pero sí uno aceptable, ya que por regla general, el camino más corto será el que contenga menos nodos.

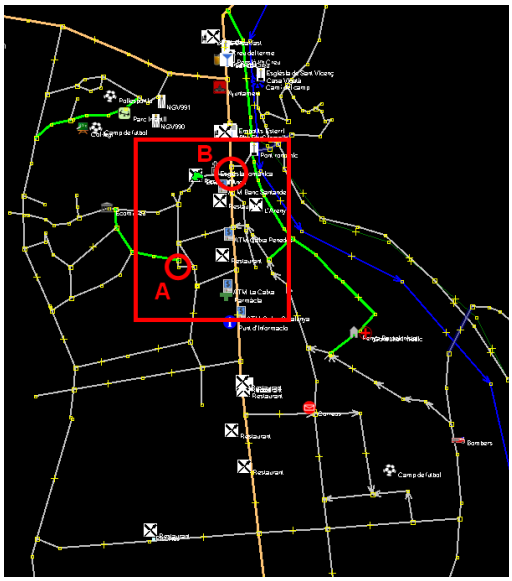

Figura 2: *Ejemplo de cómo se selecciona una pareja de puntos {A,B} y una área geográfica delimitada que incluye el punto inicial {A} y el punto final {B}.*

ANALISIS RESULTADOS

Con todo lo detallado anteriormente, se ha implementado el algoritmo en el móvil y se ha comprobado que estas medidas han funcionado. Aún así, la rapidez de ejecución del *backtracking* todavía no era suficiente rápida, por lo que aún se han aplicado las siguientes restricciones/mejoras:

1. Limitar el número de caminos que se buscan, para acotar el tiempo de búsqueda. Inicialmente se establece un número máximo de caminos a buscar, que después de un pequeño estudio se establece en 24, ya que en la mayoría de casos nos da un camino aceptable. Sin embargo, en otros casos el camino encontrado está muy lejos de ser una buena ruta, porque si el algoritmo inicia la búsqueda en la dirección opuesta a la que está el punto final, tal vez no será capaz de encontrar un camino mínimamente bueno con 24 rutas. Así, dividimos equitativamente la cantidad de rutas a buscar según la cantidad de conexiones del primer nodo, cómo se detalla en la ecuación 20. De esta forma se busca el mismo número de caminos en todas las direcciones.

$$número\ caminos/nodo = \frac{24}{conexiones\ nodo} \qquad (20)$$

Por ejemplo, si el POI inicial está conectado con 3 nodos, se buscan 8 caminos en cada dirección, asegurando que busque caminos por todos los lados.

2. Como se ha explicado anteriormente, es muy importante que el algoritmo empiece a buscar por la dirección correcta, lo que puede ahorrar muchas nuevas ramas que se desvían del punto final. Una forma de saber la dirección correcta, es encontrar el nodo que conecta con el POI inicial que a la vez está más cerca del POI final.

Para conseguir esto, se ordenan los nodos que conectan con el POI inicial según su distancia al POI final. El orden de estos nodos se usa para crear la matriz de adyacencia aplicando el mismo orden. Como la matriz se recorre secuencialmente, si está ordenada, primero se consultan los nodos más cercanos al POI final. Así el algoritmo siempre empezará a buscar caminos por los nodos que están más cerca del POI de destino. Después de algunas pruebas, se ha visto que esta fórmula de ordenar los nodos ha significado un gran aumento en la eficiencia del algoritmo.

Aunque parezca contradictorio, el punto 1 continua siendo necesario, porque puede pasar que el nodo más cercano no sea la dirección más rápida para llegar al final, así que es necesario comprobar todas las direcciones, aunque la mayoría de veces, el nodo más cercano al POI final es el camino correcto. En la fig. 3 se puede ver un ejemplo de esto. En la parte izquierda vemos con un recuadro verde el nodo más cercano al POI de destino que está conectado con el POI inicial. Si se sigue esta dirección se encontrará el camino más rápido. En cambio, en la parte derecha vemos un ejemplo donde esto no se cumple. El nodo del recuadro azul está más cercano al punto de destino que el verde, pero no es el mejor camino para llegar al destino.

Figura 3: *Comparativa sobre cuál es la mejor dirección para llegar al punto final.*

Diseño final del algoritmo

Con todo esto, el diseño final del algoritmo de *routing* queda de la siguiente forma. Se crean parejas de puntos de interés para las cuales hay que buscar un camino para guiar al visitante. Por cada pareja se selecciona una área geográfica y se obtienen los nodos que forman parte de esta zona seleccionada. Se ordenan los nodos según la distancia al POI de destino y se crea una matriz de adyacencia según este orden. A continuación, se usa *backtracking* para buscar caminos recorriendo la matriz hasta que encuentra el camino más óptimo según un número límite de caminos.

CONCLUSIONES

Los algoritmos de *routing* y de personalización se han implementado con éxito en la aplicación para móviles Itiner@, dónde se generan rutas turísticas adaptadas al visitante y el móvil es capaz de guiarlo a través de los distintos POIs sin necesidad de conexión a Internet en ninguna de las operaciones. En la fig. 4 podemos ver una captura de pantalla de la aplicación, donde se ha generado una ruta de demostración por el pueblo de Esterri d'Áneu. Se puede ver cómo se dibuja el camino a seguir (línea azul) para llegar al siguiente punto de interés así como todos los POI de la ruta (chinchetas de colores). Se han hecho varias pruebas y los algoritmos funcionan de forma suficientemente rápida para ser implementados en una aplicación de ocio, dónde el usuario está interaccionando constantemente con el móvil.

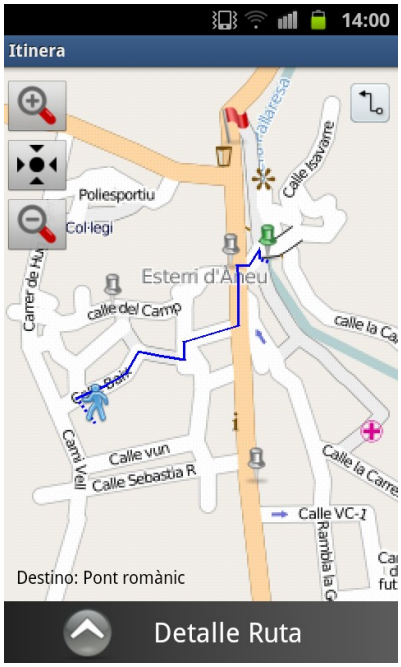

Figura 4: Captura de pantalla de la Aplicación Itiner@ dónde se usan los dos algoritmos implementados

AGRADECIMIENTOS

Este artículo ha estado desarrollado gracias al soporte del proyecto TSI-020110-2009-442 y al Instituto *Internet Interdisciplinary Institute* de la Universitat Oberta de Catalunya.

REFERENCIAS

[1] L. DESCAMPS-VILA, A. PEREZ-NAVARRO, J. CONESA, J. CASAS: “Cómo introducir semántica en las aplicaciones SIG móviles: expectativas, teoría y realidad.” *V Jornadas SIG Libre de Girona*, 2011.

[2] L. DESCAMPS, A. PEREZ-NAVARRO, J. CONESA, J. CASAS: “Personalización de servicios basados en localización: un caso práctico.” *V Jornadas SIG Libre de Girona*, 2011.

[3] L. DESCAMPS-VILA, A. PEREZ-NAVARRO, J. CONESA, J. CASAS: “Hacia la mejora de la creación de rutas turísticas a partir de información semántica” *V Jornadas SIG Libre de Girona*, 2011.

[4] BACKTRACKING [web page]. [Última consulta 12 marzo 2012]. http://en.wikipedia.org/wiki/Backtracking

[5] TRAVELLING SALESMAN LIBRARY [web page]. [Última consulta 12 marzo 2012]. http://wiki.openstreetmap.org/wiki/Traveling_salesman.