Algorítmos para encontrar el camino más corto

David Albalate Cabedo

12 de mayo de 2017

1. Introducción

Actualmente existen varios métodos para conseguir encontrar una ruta entre dos puntos. Pero además de conseguir una ruta, son varios los métodos que se utilizan para encontrar la ruta más corta entre dos puntos predefinidos. Conseguir la ruta más corta es el objetivo de varias aplicaciones muy utilizadas actualmente, entre las que se encuentran los navegadores de los coches o aplicaciones móviles como *Google Maps* o *iOS Mapas*. En este artículo se van a explicar varios algoritmos que se utilizan para conseguir la ruta más corta entre dos puntos. Estos algoritmos se *testearán* en un el lenguaje de programación **Java**, en el entorno de desarrollo integrado (*IDE*) de **Eclipse**. Para la representación de estos caminos utilizaremos Grafos [9].

El grafo que utilizaremos será dirigido[10] y tendrán asociadas a sus aristas un valor equivalente.

Después de presentarlo los diferentes algoritmos que cumplen el objetivo que buscamos también se realizará una comparativa entre los diferentes tiempos de ejecución para grafos de diferentes características.

2. Algoritmos de camino más corto

En este apartado nos centraremos en la explicación del funcionamiento de cada uno de los algoritmos que cumplen con el objetivo de encontrar el camino más corto entre dos puntos. Los algoritmos de los que se hablarán a continuación están implementado en la librería **JGraphT** de **Java**.

2.1. Algoritmo Bellman-Ford

El algoritmo de Bellman-Ford permite encontrar el camino más corto, si existe. No es el más óptimo pero es eficiente en muchos casos. Este algoritmo es iterativo y se basa en la utilización de una matriz de longitudes. En un primer momento, se inicializan todas las distancias mínimas entre vértices a $+\infty$. Una vez inicializado se empiezan las iteraciones. En un primer momento cogemos la distancia del punto inicial y recorremos los arcos que tiene midiendo la distancia entre el punto inicial y los vértices con los que conecta. En la siguiente iteración recorreremos los vértices a los que hemos llegado en la iteración anterior. En cada iteración además de medir la distancia también se almacena el vértice del que proviene para así una vez llegamos al vértice objetivo podamos saber el camino que ha seguido[8]. Para utilizar este algoritmo utilizamos la clase **BellmanFordShortestPath**[2].

2.2. Algoritmo Dijkstra

Este algoritmo supone una mejora al **Algoritmo Bellman-Ford**. En este caso se inicializan dos vectores. Uno contendrá la distancia al nodo inicial que en todos los casos será $+\infty$, exceptuando el nodo inicial ya que la distiancia a sí mismo es de 0. En el otro vector guardaremos los precursores y lo inicializaremos con el contenido vacío. Una vez inicializados los vectores, buscamos en cada iteración el nodo no visitado con una menor distancia al nodo inicial. Una vez encontrado se visitan todos los nodos con los que conecta y se miden las distancias además de añadir los precursores. Como podeis observar es muy parecido al **Algoritmo Bellman-Ford**. La principal diferencia con respecto a ese algoritmo es que el **Algoritmo Dijkstra** pone la distancia al nodo una única vez en todo su recorrido y opta por el de distancia menor mientras que el **Algoritmo de Bellman-Ford** recorre todos los nodos con los que conecta[8]. Para utilizar este algoritmo utilizamos la clase **DijkstraShortestPath**[4].

2.3. Algoritmo Dijkstra Bidireccional

Este algoritmo utiliza la misma técnica de búsqueda **Algoritmo Dijkstra** pero realiza una búsqueda desde el nodo origen hacia el nodo destino y, al mismo tiempo, desde el nodo destino hacia el nodo origen, terminando donde estas búsquedas se reúnen[7]. Para utilizar este algoritmo utilizamos la clase **BidirectionalDijkstraShortestPath**[3].

2.4. Algoritmo A*

El **Algoritmo A*** no asegura de encontrar la ruta más corta como hacen los anteriores pero generalmente siempre devuelve buenas opciones. Aunque no es recomendable en caso de tener laberintos representados en grafos. El **Algoritmo A*** utiliza información del punto de destino y intenta ir de la manera más recta posible. La primera necesidad que surge es encontrar una manera para encontrar la distancia de un nodo respecto a punto de destino para poder orientarnos. Cuanto mejor sea la manera de aproximar estos puntos mejores serán los resultados obtenidos por el algoritmo. Este algoritmo como los anteriores guarda la distancia al origen y el predecesor pero se guía por la suma entre la distancia al origen y la distancia estimada al destino. Una vez tenemos esa distancia podemos optar por el camino más corto[8]. Para utilizar este algoritmo utilizamos la clase **AStarShortestPath**[1]. El algoritmo que utilizamos de aproximación es muy vago y devuelve una distancia equivalente para todos los nodos ya que el grafo se genera aleatoriamente.

2.5. Algoritmo Floyd-Warshall

El **Algoritmo Floyd-Warshall** encuentra el camino más corto a partir de dos matrices: una matriz de pesos y otra de caminos.La matriz de pesos contendrá en cada celda la distancia de los dos nodos que representan sus índices, es decir, la celda [1][2] contiene el peso de la union entre el nodo 1 y el nodo 2 pero no del nodo 2 al nodo 1. En la matriz de caminos se representarán desde que nodo se llega destino. En el caso de que exista una conexión en la celda [1][2] el contenido será el 1 ya que desde el nodo 1 conseguimos llegar al nodo 2. En el caso de que dos nodos no conecten el contenido de la matriz de pesos será $+\infty$ mientras que el contenido de la matriz de caminos será -1. El funcionamiento de este algoritmo es simple. Recorreremos la matriz de pesos reservando la fila y columna de esa iteración, es decir, en la iteración 2 quedan reservadas la fila 2 y la columna 2. Una vez reservadas ahora recalcularemos los pesos como la suma del contenido de los sitios respectivos a nuestra reserva. En el caso de ser la iteración 2 y querer saber el contenido de la celda [1][1], sumaremos el contenido de la celda

[1][2] y de la [2][1]. Si esta suma es menor al contenido original de la celda entonces cambiaremos el contenido de la celda de la matriz de pesos a la suma y modificaremos el contenido de la celda en la matriz de caminos para sustituirlo por el valor de la iteración, en el caso actual la cambiaríamos por un 2. Esto hace que una vez acabemos todas las iteraciones tendremos una matriz con los pesos de los caminos mínimos y el camino que debemos seguir para llegar de un nodo a otro. Así podremos calcular el camino[12]. Para utilizar este algoritmo utilizamos la clase **FloydWarshallShortestPaths**[5]

2.6. Algoritmo K caminos más cortos

El **Algoritmo K caminos más cortos** permite conseguir una cantidad de K caminos más largos entre dos puntos. Su funcionamiento se basa en la creación de una estructura de datos ordenada, como un montículo, en el que se irá introduciendo los caminos auxiliares que se recorran ordenados por el coste del camino. Como es una estructura ordenada, se irán recorriendo camino y hasta que la cabeza de la estructura de datos no acabe en el destino, es decir que sea uno de los caminos, se irán creando caminos a partir de sus auxiliares y reintroduciendo los nuevos caminos en la estructura de datos. La creación de los caminos auxiliares se hace a través de los nodos adyacentes al consultado y así se van creando caminos con la suma del coste del camino original más la arista al nodo adyacente[11].Para utilizar este algoritmo utilizamos la clase **KShortestPaths**[6]

3. Comparativa de tiempos de ejecución

Para crear las siguientes comparativas hemos utilizado una generación de grafos aleatoria con diferentes números de nodos y de aristas. Esto puede afectar a la eficiencia de los diferentes algoritmos ya que estos algoritmos consiguen una máxima eficiencia en entornos diferentes por lo que esta generación puede beneficiar a unos y perjudicar a otros.

3.1. Comparativa de tiempos en un grafo de 10 vértices y una probabilidad del 60 % de crear un enlace

Nombre del algoritmo	Tiempo de ejecución(s)	C.óptimo encontrado
Algoritmo A*	0.006	Si
Algoritmo Bellman-Ford	0.003	Si
Algoritmo Dijkstra	0.003	Si
Algoritmo Dijkstra bidireccional	0.003	Si
Algoritmo Floyd-Warshall	0.002	Si
Algoritmo K caminos más cortos	0.06	Si

3.2. Comparativa de tiempos en un grafo de 100 vértices y una probabilidad del 10 % de crear un enlace

Nombre del algoritmo	Tiempo de ejecución(s)	C.óptimo encontrado
Algoritmo A*	0.007	Si
Algoritmo Bellman-Ford	0.006	Si
Algoritmo Dijkstra	0.004	Si
Algoritmo Dijkstra bidireccional	0.005	Si
Algoritmo Floyd-Warshall	0.017	Si
Algoritmo K caminos más cortos	0.262	Si

3.3. Comparativa de tiempos en un grafo de 500 vértices y una probabilidad del 2 % de crear un enlace

Nombre del algoritmo	Tiempo de ejecución(s)	C.óptimo encontrado
Algoritmo A*	0.021	Si
Algoritmo Bellman-Ford	0.013	Si
Algoritmo Dijkstra	0.01	Si
Algoritmo Dijkstra bidireccional	0.005	Si
Algoritmo Floyd-Warshall	3.5	Si
Algoritmo K caminos más cortos	28.437	Si

3.4. Comparativa de tiempos en un grafo de 1000 vértices y una probabilidad del 10 % de crear un enlace

Nombre del algoritmo	Tiempo de ejecución(s)	C.óptimo encontrado
Algoritmo A*	0.025	Si
Algoritmo Bellman-Ford	0.026	Si
Algoritmo Dijkstra	0.014	Si
Algoritmo Dijkstra bidireccional	0.004	Si
Algoritmo Floyd-Warshall	3.124	Si
Algoritmo K caminos más cortos	813.106	Si

4. Conclusión

Como hemos podido observar los tiempos entre los diferentes algoritmos difiere ampliamente. Esto problema se amplia cuando aumentamos el número de nodos y algoritmos como el **K Caminos más cortos** resultan muy perjudicados llegando a necesitar alrededor de 14 minutos para acabar su ejecución. Después de realizar este experimento podemos ver que las situaciones en las que estaban favorecian a algoritmos como **Dijkstra** o **Bellman-Ford**.

Referencias

[1] JGraphT community. Astarshortestpath. `http://jgrapht.org/javadoc/org/jgrapht/alg/shortestpath/AStarShortestPath.html`. [Consulta: 06 de Mayo de 2017].

[2] JGraphT community. Bellmanfordshortestpath. `http://jgrapht.org/javadoc/org/jgrapht/alg/shortestpath/BellmanFordShortestPath.html`. [Consulta: 06 de Mayo de 2017].

[3] JGraphT community. Bidirectionaldijkstrashortestpath. `http://jgrapht.org/javadoc/org/jgrapht/alg/shortestpath/BidirectionalDijkstraShortestPath.html`. [Consulta: 06 de Mayo de 2017].

[4] JGraphT community. Dijkstrashortestpath. `http://jgrapht.org/javadoc/org/jgrapht/alg/shortestpath/DijkstraShortestPath.html`. [Consulta: 06 de Mayo de 2017].

[5] JGraphT community. Floydwarshallshortestpaths. `http://jgrapht.org/javadoc/org/jgrapht/alg/shortestpath/FloydWarshallShortestPaths.html`. [Consulta: 06 de Mayo de 2017].

[6] JGraphT community. Kshortestpaths. http://jgrapht.org/javadoc/org/jgrapht/alg/shortestpath/KShortestPaths.html. [Consulta: 06 de Mayo de 2017].

[7] pgRouting Contributors. Camino más corto bidireccional de dijkstra. http://docs.pgrouting.org/2.0/es/src/bd_dijkstra/doc/index.html. [Consulta: 06 de Mayo de 2017].

[8] Alexander Samarin. *Inteligencia artificial para desarrolladores - Conceptos e implementación para C#*. Planeta, 1993.

[9] Wikipedia. Grafos. https://es.wikipedia.org/wiki/Grafo. [Consulta: 05 de Mayo de 2017].

[10] Wikipedia. Teoría de grafos. https://es.wikipedia.org/wiki/Teor%C3%ADa_de_grafos. [Consulta: 05 de Mayo de 2017].

[11] Wikipedia. Yen's algorithm. https://en.wikipedia.org/wiki/Yen%27s_algorithm. [Consulta: 06 de Mayo de 2017].

[12] YouTube. Floyd-warshal algorithm (all pair shortest path). https://www.youtube.com/watch?v=Dl-ne7Qd5PY. [Consulta: 06 de Mayo de 2017].