Archives of Transport System Telematics

Volume 9

Issue 2

May 2016

Multithreaded enhancements of the Dijkstra algorithm for route optimization in urban networks

M. BAZAN[a], P. CISKOWSKI[a], R. DUDEK[a], K. HALAWA[a], T. JANICZEK[b], P. KOZACZEWSKI[a], A. RUSIECKI[a]
[a] WROCŁAW UNIVERSITY OF TECHNOLOGY, Faculty of Electronics, Department of Computer Engineering, Janiszewskiego 11/17, 50-372 Wrocław, Poland
[b] WROCŁAW UNIVERSITY OF TECHNOLOGY, Faculty of Electronics, Department of Control Systems and Mechatronics, ul. Janiczewskiego 11/17, 50-372 Wrocław, Poland
EMAIL: bazan@pwr.edu.pl

ABSTRACT
In this paper, we present a case study, showing step by step, how to speed up Dijkstra's method by parallelizing its computation and using different data structures. We compare basic algorithm with its bidirectional version and investigate two- and-multi-thread implementations based on Fibonacci heaps and regular priority queues. Experimental results obtained for artificially generated graphs as well as real-world road network data are presented and described.

KEYWORDS: Dijkstra algorithm for shortest paths, fastest routes in urban networks, route optimization

1. Introduction

Shortest path algorithms, though potentially applied to any static networks, are very often used in IP network routing protocols, route planning in road networks, and even Intelligent Transportation Systems [4]. In fact, such methods can be considered as the very basis of the graph theory with many practical applications [3]. In this article we focus on one of these methods, namely Dijkstra's algorithm, used to find the quickest connection in transportation network modelled as a graph with travel times on its edges, or shortest route in a road network.

Classical Dijkstra's algorithm is the most popular and simultaneously the most efficient sequential method on digraphs with nonnegative weights [2]. It is easy to notice, that a typical road network can be represented by a graph with vertices at the crossings and edges modelling streets. Each edge has its weight cost. Moving from one vertex to another is possible only if they are connected with an edge. Finding the path between two vertices is then finding all the edges connecting the source and destination vertex. The length of the path is defined as the sum of all weights on the route. Hence, the Dijkstra's algorithm can be easily applied to solve such problems.

The paper is organized as follows: we start with presenting a short description of the basic Dijkstra's algorithm and its modification. In particular, the ideas behind effective speeding up the algorithm are explained. Then, a description of the implementation of considered methods with the use of *LEDA* [8] and *Boost* [7] libraries is given. The following section describes the experiments performed for the implemented algorithm variants and makes an attempt to explain the results. In the last section we summarize and conclude the article.

2. Dijkstra's algorithm

In the Dijkstra's algorithm the vertices of the graph are organized into a priority queue [2, 4, 6]. At the beginning of the algorithm the distances for all vertices, save the source one, are set to infinity. The source node distance is zero.

In typical implementation each vertex is assigned a status: *not_visited* (the node hasn't been taken into account during the run of

the algorithm), *labeled* (the distance from the source to the labeled vertex has been already calculated), or *visited* (the node has been chosen as one of the vertices nearest to the source one). The tentative distances for each vertex are also maintained by the algorithm.

2.1. Basic Version

In each run of the loop, a vertex with the smallest distance is considered. Then all the nodes connected to this vertex are checked, whether their costs are larger than the summed distances. If their tentative distances are modified, the priority queue has to be reorganized.

The distance (or time) from the node i to the node j is denoted as $w(i,j)$, while the distance between the source vertex and the given node v is denoted as $d(v)$. The whole procedure of the Dijkstra's algorithm is then as follows [5]:

1. Start with initializing $d(v)$ for each v as "infinity" (maximum of *double* type). For the source node we set $d(start) = 0$.
2. Create a priority queue containing all the vertices, where the priority is given as $d(v)$.
3. Pick a vertex u with the smallest $d(v)$.
4. All outgoing edges of u are relaxed and the times $d(u) + w(u,v)$ are compared with $d(v)$. If the value in the vertex v is larger, then $d(v)$ is set to $w(u,v) + d(u)$. To help in calculating the shortest path, the predecessor u and the edge (u,v) are maintained in v. The vertex u is removed.
5. If the queue is not empty, go to step 3, otherwise go to step 6.
6. Calculate the shortest path from the destination to the source vertex, using pointers to the preceding nodes.

2.2. Bidirectional Dijkstra (one thread)

In this case we consider two separate priority queues [4]. Basic Dijkstra's method is executed simultaneously from the source and from the target vertex. If some node is reached from both directions, the algorithm stops.

The first queue is used in the forward search from the source node and the second one in the backward search starting from the destination node. The algorithm picks and processes a vertex from the first priority queue, then picks and processes the vertex from the second queue. In such approach the objects of the type *Node* should also store the information about the search direction. The solution can be found, when one search picks a vertex that already has been processed by another direction.

2.3. Bidirectional Dijkstra (two-threaded)

Bidirectional search is implemented with two threads using separate priority queues. Each thread performs a search in one direction, so there is no need of mutex-based synchronization. Both threads work in parallel, so if one search picks a vertex already processed by another one, *bool* type variable is set to *true*. The variable is accessible by two threads, so once it is set, both searches stop.

2.4. Multithreaded Dijkstra

In the multithreaded approach the basic graph is divided into a set of subgraphs [2, 5]. For each subgraph, a separate thread, with its own priority queue, is called. Finding the smallest element in the whole graph is implemented by finding the smallest elements in each subgraph and comparing them. Then, the thread responsible for the chosen node, processes the vertices connected to it. All the calculations for given vertices are divided between threads.

The running time of the Dijkstra's algorithm without using self-organizing data containers is $O(V^2)$ [2, 4], because searching for the smallest element takes $O(V)$, as well as recalculating the vertices. If the algorithm is parallelized, the running time depends on the number P of subgraphs. Then, finding the smallest element takes $O(\log(P))$, and updating the values at vertices takes $O(V/P)$. For the procedure performed V times we obtain finally $O(V^2/P + V\log(P))$.

3. Implementation

The algorithms were implemented in C++ under MS Visual Studio 2012 with the use of *LEDA* [8] and *Boost* [7] libraries.

Searching for the shortest path is performed in a graph described by objects from LEDA library. We defined additional classes *Node* and *Edge*, based on the *node* and *edge* classes (elements of the *graph* class) of LEDA, to model the graphs being analyzed. Class *Node* consists of an object of the *node* class of LEDA library and additional elements used to implement Dijkstra's algorithm: node ID, latitude, longitude, node status in the Dijkstra's algorithm (possible values *not_visited, labeled, visited*), time of reaching a given node on the path. Additional elements used to extract the shortest path after the algorithm finished are the node predecessor on the path and the edge between the preceding and current node. In the bidirectional Dijkstra's search additional variable describing the search direction is used.

The *Edge* class consists of an object of the *edge* class from LEDA and additional elements used in the Dijkstra's method: ID of the node the edge points from, ID of the node the edge points to, maximal speed on the edge and the length of the edge.

Two arrays are used to store elements of *Node* and *Edge* types. In the class *Network* the basic graph and all the methods of calculating the shortest path in the graph are included. In particular, the class consists of the graph sizes, and dynamic arrays of pointers to vertices and edges.

3.1. Fibonacci heap

In the Dijkstra's algorithm, Fibonacci heaps can be used instead of regular priority queues. In such a case, the running time of the method with Fibonacci heaps is $O(n\log n + m)$, where n is the number of nodes, and m is the number of edges in the whole graph [1,2]. Fibonacci Heap data container was implemented with the use of *boost* library.

In the Fibonacci heap all the elements are sorted in a selected order, after a new element is inserted into the container. When the

© Copyright by PSTT , All rights reserved. 2016

elements are simple numbers, the only thing to be done is defining a heap as: `boost::heap::fibonacci_heap<int> fib_heap.` However, when one needs to store some more sophisticated objects, the elements of the class which are to be sorted, have to be specified. This is why a special structure `struct Komparator_fib` had to be created. Additional method takes two class elements as input arguments, and returns *bool* type value. In this case, the elements are sorted according to `m_dCurrentTravelTime,` the time of the path from the starting to the current vertex. If two elements of the heap have the same values, their ID numbers are compared. A structure with the given method has to be introduced as one of the Fibonacci heap parameters.

In the *Boost* library one may found many utilities to use the heap. Function *push(…)* after adding an element returns its handler. To enable a free access to such a type, a handler being an element of a heap needs to be defined.

Each element of the class *Node* needs a handler to itself from the Fibonacci heap. The handler type should be declared earlier but during such declaration one needs a call to the elements of a Fibonacci heap (objects of *Node* class), as parameters. To avoid such mutual calls and declarations a new class `Fibonacci_Heap_Handler` was implemented.

The problem was then solved as follows: at the beginning, the *Node* class with a pointer to the new class was declared. Then *Komparator* structure, handler type *fib_handle*, and finally *Fibonacci_Heap_Handler* class were declared.

3.2. Search method implementation

Searching for the shortest path in a graph is implemented in the methods of the *Network* class. Each function takes as its parameters IDs of the source and destination nodes. Basic implementation of the Dijkstra's algorithm uses *set* container from standard *std* library as a priority queue. The first step is initialization of nodes and corresponding weights (travel times) with the starting node initialized to 0. Then all the vertices are formed in a priority queue and the main body of the algorithm starts, taking the vertices one by one from the queue, until it is empty. The elements in the queue are sorted by the *m_dCurrentTravelTime*, so the nodes that haven't been considered, with the value set to *double limit*, are at the end of the queue. Then we need to find the nodes connected to the element currently considered. In the next step the chosen node is checked and if the value in this vertex is larger than the summed path length, it is replaced by the smaller value. After the value is changed, the pointer to the preceding vertex and the pointer to the edge leading to the considered node, are also stored. At last, the chosen node is removed from the top of the queue. The method finishes when the source vertex is chosen from the data container.

3.2.1 Bidirectional Dijkstra's algorithm, one thread

In this method two search directions are implemented: from the source to destination node and backwards, from the destination to the source vertex. After the initialization similar to the one of basic Dijkstra, two priority queues are created. Both queues contain all the vertices, however, one queue sorts its elements by *m_dCurrentTravelTime* while another is sorted by *m_dCurrentTravelTime_backward*. The rest of the code is very similar to the basic Dijkstra's implementation, but the method ends when it finds connection between forward and backward search. If a node has been already considered by the opposite direction, it stays unchanged. This method needs also additional phase, because during the backward search, pointers to the following, not preceding, vertices and edges are stored. Starting from the connection of both directions, the vertices are shifted to the source node.

3.2.2 Bidirectional Dijkstra's algorithm with two threads

In this method two threads, working in parallel, are applied. The threads are called with the use of *Boost* library. In one thread the forward search, and in another the backward search is performed. After the threads end, the pointers need to be shifted in the similar way as in the one-thread version of bidirectional search.

3.2.3 Multithread Dijkstra's implementation

Many threads are called using a reference to an object of class *Parallel_Info* containing: a data container (priority queue), an array of vertices' ids, and array of edges' ids, and an array of vertices values. In the multithread approach the graph is divided into clusters, taking advantage of the fact, that all the vertices are stored in a two-dimensional array. The division of the graph is performed as follows:

```
max_y = sqrt(maxID);
for(int i = max_y; i <= max_y + 3; i++)
if(max_y * i <= (maxID))
max_x = i;
```

Each cluster is then defined by its maximal row and column indices *max_x* and *max_y*. The segment size can be calculated as:

```
seg_x = N/max_x;
seg_y = M/max_y;
```

and the indices inside a cluster used by a thread *ID* as:

```
_x = (ID) / max_y;
_y = (ID) % max_y;
```

If a graph cannot be divided into equal parts, the remaining rows and columns are added to the nearest cluster. Each thread has its separate priority queue storing the vertices from one cluster. During the execution, each thread is responsible for finding the smallest element in its region. To synchronize the threads, barriers from the *Boost* library are applied. The barriers are implemented in the constructor of *BarrierGuard* structure. The synchronization is performed four times in the code. The first time is needed after the minima in each subgraph are found. Based on the local minima, the globally smallest node is determined. After the node with the smallest value is found, the second barrier is called, and the vertices connected to this node are considered. When the vertices have been stored, the third synchronization follows. The last stage in the function is recalculating the values in the vertices from the previous step. Each thread recalculates one vertex. If

there are more threads than vertices to process, some of them can be inactive. Then the last synchronization is called.

4. Experimental results

To compare all the considered approaches, many experiments were performed. We tested basic Dijkstra's algorithm, its bidirectional version, bidirectional method using two threads, multi-thread implementation with 4 threads for forward and 4 for backward search, and one-directional multi-thread implementation with 4 threads. The analyzed performances were obtained on the Intel® i7 CPU @ 3.20GHz (4 physical cores). Presented results are averaged over 15 runs of each algorithm.

4.1. Artificial data

Table 1 contains processing times of four methods for the data generated as square grids with random edges. For each grid size several levels of density, defined as the ratio of edges to nodes, were generated. This preliminary study performed for basic variations of the considered algorithms can be used as referential to the real-world data presented in the following section. As one may notice looking at the Table 1, the basic version of Dijkstra method is always outperformed by the bidirectional approach. Multi-threaded algorithm reveals its superiority over the original one, for graphs of higher density. The best performances were obtained for bidirectional version implemented on two threads.

Table 1. Processing times for the artificially generated data (averaged over 15 runs of the algorithms)

Grid size	Edges/ Nodes	Dijkstra	Bidirectional, one thread	Bidirectional, two threads	Multithread
50x50	3	0.091	0.089	0.067	0.183
	4	0.129	0.077	0.065	0.194
	6	0.119	0.120	0.080	0.176
	8	0.255	0.111	0.094	0.168
100x100	3	0.489	0.263	0.190	0.548
	4	0.482	0.255	0.157	0.614
	6	0.711	0.350	0.266	0.591
	8	0.882	0.347	0.319	0.744
250x250	3	2.142	1.116	0.815	2.239
	4	1.562	1.116	0.888	2.253
	6	3.709	1.773	1.384	2.637
	8	2.378	2.410	1.543	2.247
500x500	3	4.403	2.169	1.073	6.666
	4	4.592	2.423	1.511	5.566
	6	9.723	4.796	2.475	8.791
	8	10.500	6.101	4.021	8.682
1000x1000	3	10.964	5.720	4.185	10.451
	4	17.258	9.348	5.451	16.554
	6	27.327	14.056	10.401	22.461
	8	45.239	25.631	15.485	37.152

In the Table 2 results of comparing the algorithms using different data structures were presented. The artificial data were generated as square grids with each node connected only to its four nearest neighbours (or less, in the case of grid borders). For such graphs, implementing the algorithms with Fibonacci heaps clearly improved their processing times.

Table 2. Processing times for the artificially generated data (square grids, averaged over 15 runs of the algorithms)

Grid size	Dijkstra	Bidirectional Dijkstra	Bidirectional Dijkstra (two threads)	Dijkstra with Fibonacci heap	Bidirectional Dijkstra with Fibonacci heap	Bidirectional Dijkstra with Fibonacci heap (two threads)
50x50	0.217	0.054	0.038	0.032	0.024	0.028
100x100	1.509	0.192	0.119	0.123	0.094	0.106
250x250	3.168	1.033	0.655	0.437	0.378	0.791
500x500	18.343	3.739	2.482	1.402	1.203	1.166
1000x1000	44.674	18.636	12.130	5.422	5.008	4.443

4.2. Real-world dataset

The real-world data were obtained from urban network of the city of Wrocław, Poland. Based on this urban road network, using OpenStreetMap® [9] we generated two graphs: one without taking into account the street directions, and another with the street directions included in the graph description. On such graphs we tested the algorithms on two tasks. The first one was to find the route between the node 0 and the node with the highest ID (Task 1), the second one was to find the quickest route between randomly chose nodes (Task 2).

Analysing figures 1 and 2, one may observe how the considered variants of the algorithm performed for the real-world data. It is clearly evident, that the best performances can be observed for bidirectional implementations. What may be surprising is that more sophisticated versions using Fibonacci heaps and multi-thread approach seem to be much slower. Only the one-directional version of the multi-thread implementation with Fibonacci heap achieved similar processing time as basic Dijkstra method. Moreover, using theoretically better data structure (a heap instead of regular priority queue), always slowed down the calculations. These results appear to contradict the conclusions that might have been formulated based on the analysis of processing times for the artificial datasets (Tables 1 and 2). The explanation of this phenomenon is that the real-world road network's structure is different than those traditionally used as testing benchmarks for shortest paths algorithms.

© Copyright by PSTT , All rights reserved. 2016

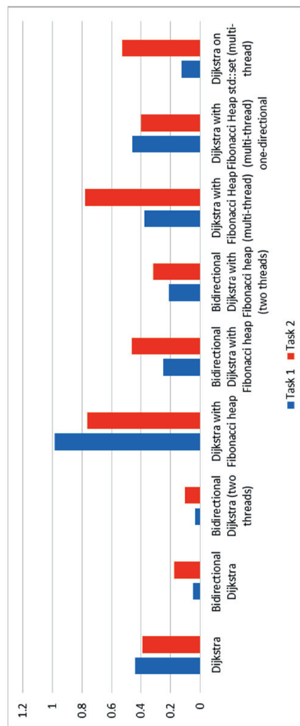

Fig. 1. Processing times of the algorithms tested on the real-world graph without street directions (averaged over 15 runs of the algorithms) [own study]

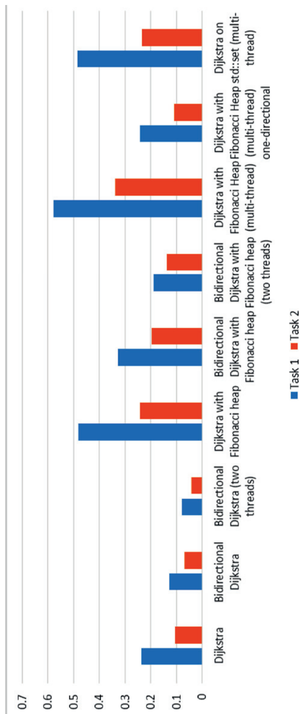

Fig. 2. Processing times of the algorithms tested on the real-world graph with street directions (averaged over 15 runs of the algorithms) [own study]

5. Conclusion

In this paper, we presented possible ways of speeding up traditional Dijkstra's algorithm by parallel multi-thread implementation and using more efficient data structures. The experimental study revealed that some modifications, save reasonable from theoretical point of view, are rather impractical when applied to the real road network data. Presented simulation results indicate, that the simplest and simultaneously most effective way to improve the considered method is to implement its bidirectional version on two threads. However, there may exist another solution that the authors plan to exploit in future. Our efforts may be directed towards formulating parallel multi-thread algorithm that makes use of the information about characteristic of real urban networks.

Acknowledgements

The authors would like to acknowledge a partial support of the WUT grant no. S50242.

Bibliography

[1] BAST, H., et al.: Route Planning in Transportation Networks, Technical Report of Microsoft Research, January 2014.

[2] CRAUSER, A., et al.: A Parallelization of Dijkstra's Shortest Path Algorithm, in MFCS'98, LNCS 1450, pp. 722-731, Springer-Verlag 1998.

[3] DELLING, D., WAGNER, D.: Time-dependent route planning, in Robust and Online Large-Scale Optimization, LNCS 5868, pp. 207–230, Springer 2009.

[4] DELLING D., et al.: Engineering Route Planning Algorithms in: Algorithmics, LNCS 5515, pp. 117–139, Springer-Verlag 2009.

[5] DUDEK, R.: Analysis and evaluation of a set of shortest paths of passing on traffic control system, Thesis, Wroclaw University of Technology 2014.

[6] JASIKA, N., et al.: Dijkstra's shortest path algorithm serial and parallel execution performance analysis, MIPRO 2012, May 21-25, Croatia 2012.

[7] KARLSSON, B.: Beyond the C++ Standard Library: An Introduction to Boost, Addison-Wesley 2006.

[8] LEDA Homepage: http://www.algorithmic-solutions.com/leda/index.htm, [date of access: 05.01.2016].

[9] OpenStreetMap Homepage: https://www.openstreetmap.org, [date of access: 05.01.2016].