APPLICATION OF SELF-ORGANIZING MIGRATING ALGORITHM ON THE SHORTEST PATH PROBLEM

Tomáš HORÁK¹, Pavel VAŇACHA²

¹ Czech Technical University in Prague, Faculty of Transportation Sciences, Department of Logistics and Transportation Processes, Horská 3, 128 03 Praha 2, Czech Republic, xhorak@fd.cvut.cz
² Tomas Bata University in Zlín, Faculty of Applied Informatics, Department of Informatics and Artificial Intelligence, Nad Stráněmi 4511, 760 05 Zlín, Czech Republic, varacha@fai.utb.cz

Abstract

Traditionally, the shortest path in a weighted graph problem is solved by well-known algorithms such as Dijkstra’s, Floyd–Warshall, Bellman–Ford algorithms or the A-star algorithm. This paper employs an evolutionary algorithm called Self-Organizing Migrating Algorithm (SOMA) for the task and shows benefits as well as drawbacks of such approach. For this purpose, a sample weighted graph was defined and the shortest path problem was solved by asynchronously parallel distribution of SOMA. Obtained results of the experiment were compared with the traditional algorithms for reference.

Keywords: weighted graph, shortest path problem, Self-Organizing Migrating Algorithm

1. INTRODUCTION

This paper compares evolutionary algorithm called Self-Organizing Migrating Algorithm (SOMA) with traditional algorithms commonly used for solving shortest path problem in a weighted graph such as Dijkstra’s, Floyd-Warshall, Bellman-Ford or A-star. Aim of the paper is to find out how SOMA performs compared to these traditional algorithms measured by runtime. This paper should be understood as an introductory paper into the matter.

2. METHODOLOGY

For purpose of comparison we defined simple two-dimensional grid-like weighted graph as shown in Fig. 1. The upper-left node was defined as the starting point, the node in the lower-right was defined as the destination node. The shortest path is then quite easily verifiable, being, obviously, the diagonal line. Twenty-five nodes shown in Fig. 1 depict starting level or “Level 1” we used for comparison of the algorithms runtime. Level 2 consequently had 100 nodes, while number of nodes in Level 3 increased to 400, in Level 4 to 900 and, finally, in Level 5 to 1 600 (exponential increase).

Length of each edge between nodes is 1.

We are not going to describe the programming procedure for traditional algorithms separately, since it is generally known, however, we include short description of each used algorithm in the next chapter. Since SOMA is not that thoroughly known, next sub-chapter has description of methodology used with SOMA. More general description of SOMA follows later in chapter 3.

Fig. 1 – Weighted graph used to compare SOMA with traditional algorithms

2.1 Methodology used with SOMA

At first, a table containing all nodes is created and stored in computer’s memory. Every node gets unique ID. Every node is assigned a list of all neighboring nodes together with length (weight) of all connecting edges.
Every connection resulting from such list gets a unique number from 1 to m, where m is the number of edges originating in the node. Specimen for SOMA is set as Real (0, n), where n is the maximum number of edges originating from any chosen node (in Fig. 1 n equals to 8). This applies to all dimensions. Number of dimensions of the solved problem is equal to the total number of nodes in the graph. Subsequently, IDs of start and destination nodes are set.

Cost Function calculates fitness of every individual in this way: Value of the first individual’s position is rounded up. This number indicates which edge will be chosen from the pool of edges originating in the nodes. If the number is bigger than the actual number of edges, this number is subtracted and the selection process starts over. Algorithm then moves to the next node over the edge that was chosen in the previous step and the whole process starts over again, this time with the second individual’s position. The edge that was already processed by the algorithm is removed from the list of eligible edges and its value (weight) is added to the Cost Function. This edge is also added to the list of edges creating the path. These steps are repeated until the algorithm arrives at the node marked as the destination node. Value of the individual’s Cost Function is then returned and SOMA continues its search for the most optimal (shortest) path. When algorithm finishes, list of edges included in the shortest path is returned.

Correction procedures include: (a) If the algorithm reaches a node from which there is not any edge available for selection, such node is removed from the list of nodes as well as from the path. The algorithm then moves one step back to the previous node. (b) If the algorithm reaches node marked as the starting node from which there is no edge available for selection, the search is finished and the Cost Function returns infinite number.

The SOMA algorithm was programmed in JAVA and run on multi-thread SuperMicro server. Further details can be found in [1].

3. TRADITIONAL ALGORITHMS
This chapter gives a brief introduction on traditional shortest path problem solving algorithms we used for comparison with SOMA.

3.1 Dijkstra’s algorithm
Dijkstra’s algorithm is used to find all shortest paths from a given node to the other nodes in a graph that contains no edges of negative length. The algorithm was invented by a Dutch computer scientist Edsger Dijkstra in 1959. Dijkstra’s algorithm can be understood as a generalized breadth-first search through a graph, where the wave does not propagate based on the number of edges from the source but rather on the distance from the source (meaning edge weights). Therefore, the wave processes only those nodes to which we have found the shortest path already. Dijkstra’s algorithm keeps all nodes in a priority queue; that is, the top of the queue is always an element with the highest priority. The nodes are ordered according to their distance from the source—in the first iteration, only the source has a 0 distance, all other nodes have an infinite distance. In each step, the algorithm selects a node with the highest priority from the queue (the shortest distance from the already processed part) and puts it among the already processed nodes. Subsequently, it passes through all its yet unprocessed descendants, adds them to the queue, unless they are already contained there, and verifies whether they are not closer to the source now than before the currently selected node was included among the processed nodes. That means that it verifies the following inequality for all its descendants:

\[ \text{distance}_{\text{processed}} + \text{length of the Edge}_{\text{processed, descendant}} < \text{distance}_{\text{descendant}} \]  \hspace{1cm} (1)

If the inequality holds then the algorithm will give the descendant a new distance and mark the currently processed node as its ancestor. After passing through all the descendants, the algorithm selects a node with the highest priority from the queue and repeats the entire step. The algorithm is terminated when all the nodes are processed (the priority queue is empty). The algorithm’s complexity depends on how we realize
the priority queue. The priority queue can be implemented in a field—then we will get a total time $O(|V|^2)$, or by means of a binary heap—then we will get a total time $O(|E| \log |V|)$ [2], [3].

### 3.2 Bellman–Ford algorithm

The Bellman–Ford algorithm finds the shortest path in an oriented graph with arbitrary edge weights. The algorithm was invented by Richard Bellman and Lester Ford. The Bellman–Ford algorithm is based on an operation called relaxation. This operation comprises two nodes and an edge that passes between them. If the distance of the source node added to the length of the edge is shorter than the actual distance of the target node, then the source node is marked as a predecessor of the target node on the shortest path. If the inequality does not hold then the edge does not make the path shorter and thus no changes are performed. The length of a path from the source to any of the target nodes may be $|V| - 1$ edges at most (because otherwise it would have to contain a cycle). If we perform the relaxation operation with all graph edges ($|V| - 1$) times we will find all the shortest paths. We can verify this by running the relaxation of all edges one more time. If a relaxation occurs, then the graph contains a cycle of negative length; if no relaxation occurs, the algorithm can return a result. The asymptotic complexity of the algorithm is $O(|V||E|)$, because the outer loop runs exactly ($|V| - 1$) times and the inner loop runs through all graph edges [2], [3].

### 3.3 Floyd–Warshall algorithm

The Floyd–Warshall algorithm works on an oriented graph, which contains no negative cycles and finds the shortest oriented paths between each pair of vertices. In addition, it will find a path with the lowest number of edges among all paths of the same length. Input to the Floyd-Warshall algorithm is a matrix of lengths $D^0$. If an edge of length $l$ runs between two nodes $(i, j)$, then the matrix contains this very value at the index $(i, j)$. The matrix has only zeros on the diagonal, while at the other indices that do not correspond to an edge, it has infinity. In other words, the matrix contains distances between nodes that do not run through any intermediary. In each iteration of the Floyd–Warshall algorithm, the matrix is recalculated to express the distance of all node pairs through a gradually increasing set of admissible intermediaries. In short, the $D^k$ matrix will express the distance of all nodes admitting the use of one (given) intermediary, $D^2$ is the distance admitting the use of two (given) intermediaries, $D^k$ admitting the use of $k$ intermediaries. For $k \geq 1$, this transformation can be expressed by the following recurrent relation:

$$d_{ij}^{k} = \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}) \quad (2)$$

The output from the Floyd–Warshall algorithm is a matrix of predecessors. The path between nodes $(i, j)$ is expressed as a field with indices row $i$, column $j$. If the field is zero, there is no such path; otherwise it specifies a predecessor of the terminal node $j$ on this path. The time complexity of the Floyd–Warshall algorithm is $O(n^3)$ [2], [3].

### 3.4 A-star algorithm

Algorithm for finding optimum paths in positively weighted graphs. Like Dijkstra's algorithm, it utilizes a priority queue; the nodes, however, are weighted according to a special function $f(x) = g(x) + h(x)$, where the function $g(x)$ represents a distance between the initial and a particular node and the function $h(x)$ represents the so-called heuristic function. This function represents an estimate of distance from a particular node to the target. The distance estimate is obtained based on at least partial knowledge of the problem's structure; the estimate may not be greater than the real distance to the target. The straight-line distance can be used as the heuristic function, for example. A priority queue of the yet unvisited nodes is created during the algorithm. The lower the function $f(x)$ of the node $x$, the higher its priority. The node is removed from the queue in each step of the algorithm and the values of functions $f(x)$ are calculated for its adjacent nodes; these are then added to the priority queue. The algorithm is terminated if the value $f(x)$ of the terminal node is lower than the value of any of the nodes in the queue or if the queue is empty. The value of the function $f(x)$ of the terminal node then represents the shortest path through the graph. The asymptotic complexity depends on the
applied heuristic function, but generally it is not worse than the complexity of the breadth-first search algorithm [4].

4. SELF-ORGANIZING MIGRATING ALGORITHM

Self-Organizing Migration Algorithm is based on self-organizing behavior of groups of individuals in “social environment”. It can also be classified as evolutionary algorithm despite the fact that no new generations of individuals are created during search process (philosophy of this algorithm). Only positions of the individuals in the searched space are changed during generation called “migration loop”. The algorithm, developed by prof. Zelinka in 1999. Several different versions of SOMA exist. This chapter describes the most common All-to-One version. All basic All-to-One SOMA principles important for correct understanding of the algorithm are described below:

Parameter definition

Before starting the algorithm, SOMA’s parameters Step, PathLength, PopSize, PRT and the Cost Function need to be defined. The Cost Function is simply a function which returns a scalar that can directly serve as a measure of fitness.

Creation of Population

Population of individuals is generated randomly. Each parameter for each individual has to be chosen randomly from the given range <Low, High>.

Migration loop

Each individual from population (PopSize) is evaluated by the Cost Function and the Leader (individual with the highest fitness) is chosen for the current migration loop. Then all other individuals begin to jump, (according to Step parameter definition) towards the Leader. Each individual is evaluated after each jump using the Cost Function. Jumping continues until a new position defined by PathLength has been reached. The new position xi,j after each jump is calculated by (1). This is shown graphically in Fig. 2. The individual then returns to the position where it found the best fitness on its trajectory

\[ x_{i,j}^{ML_{new}} = x_{i,j,start}^{ML} + (x_{i,j}^{ML} - x_{i,j,start}^{ML})t \text{PRTVector}_j \]  

where \( t \in <0, \text{by Step to, PathLength}> \) and \( ML \) is actual migration loop

Before an individual begins jumping towards the Leader, a random number rnd is generated (for each individual’s component) and then compared with PRT. If the generated random number is larger than PRT, then the associated component of the individual is set to 0 by means of PRTVector.

\[ \text{if } \text{rnd}_j < \text{PRT} \text{ then } \text{PRTVector}_j = 0 \text{ else } 1, \text{ where } \text{rnd} \in <0, 1> \text{ and } j = 1, \ldots \text{nparam} \]

Hence, the individual moves in the N-k dimensional subspace, which is perpendicular to the original space. This fact establishes higher robustness of the algorithm. Earlier experiments have demonstrated that without the use of PRT SOMA tends to find only local optimum rather than the global one. [5].

Fig. 2 – PRTVector and its effect on individual’s movement
Test for stopping condition

If the maximum number of migration loops has been reached, algorithm stops and recalls the best solution(s) found during search process.

**Tab. 1 – An example of PRTVector for 4 parameters individual with PRT = 0,3**

<table>
<thead>
<tr>
<th>j</th>
<th>rndj</th>
<th>PRTVector</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0,234</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0,545</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0,865</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0,012</td>
<td>1</td>
</tr>
</tbody>
</table>

5. **COMPARISON OF ALGORITHMS**

Results obtained from testing of all four traditional algorithms and SOMA are shown in Tab. 2 and – to get a more graspable idea – also in Fig. 3.

**Tab. 2 – Algorithms runtime depending on the number of nodes**

<table>
<thead>
<tr>
<th>Number of nodes</th>
<th>Dijkstra</th>
<th>Bellman-Ford</th>
<th>Floyd-Warshall</th>
<th>A-star</th>
<th>SOMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0,04</td>
<td>0,06</td>
<td>0,29</td>
<td>0,03</td>
<td>4 122,26</td>
</tr>
<tr>
<td>100</td>
<td>0,15</td>
<td>3,34</td>
<td>23,20</td>
<td>0,07</td>
<td>7 174,82</td>
</tr>
<tr>
<td>400</td>
<td>1,21</td>
<td>176,00</td>
<td>7 176,00</td>
<td>0,56</td>
<td>114 057,10</td>
</tr>
<tr>
<td>900</td>
<td>5,19</td>
<td>2 497,00</td>
<td>40 353,00</td>
<td>2,26</td>
<td>459 528,60</td>
</tr>
<tr>
<td>1600</td>
<td>15,50</td>
<td>13 985,00</td>
<td>307 473,00</td>
<td>6,30</td>
<td>817 473,00</td>
</tr>
</tbody>
</table>

Tab. 2 in graphical representation while using logarithmic scale on y-axis and connecting trend curves:

![Graphical representation of Tab. 2](image)

**Fig. 3 – Algorithms runtime depending on the number of nodes (graphical interpretation)**
6. CONCLUSION

Apparently, SOMA has proven itself as being able to solve the shortest path problem in a weighted two-dimensional graph. It is evident that SOMA is slower than all tested algorithms, being Dijkstra’s, Floyd-Warshall, Bellman-Ford or A-star. SOMA cannot guarantee that the found solution is always the best one possible, however, SOMA has always managed to find a suitable solution. SOMA has always found the initial solution very fast and then focused on making it more precise. Unlike the above mentioned algorithms, computation time required by SOMA does not grow exponentially.

LITERATURE


