Automatic Connections in IEC 61131-3 Function Block Diagrams

Marcin Jamro and Dariusz Rzonca
Rzeszow University of Technology
Department of Computer and Control Engineering
al. Powstancow Warszawy 12, 35-959 Rzeszow, Poland
Email: {mjamro,drzonca}@kia.prz.edu.pl

***Abstract*—IEC 61131-3 standard defines five languages for programming industrial controllers. They support both textual and graphical development approaches. In case of Function Block Diagram graphical language, diagrams consist of a set of elements connected with lines, which have various length and shape. Development of an editor supporting diagrams design involves implementation of an algorithm, which is able to automatically find a suitable connection between blocks. In the paper an appropriate application of A* algorithm is proposed. The authors have ensured that the proposed solution is efficient and work smoothly. Relations between implementation details and performance are discussed. Achieved results caused that the mechanism has been introduced into graphics editors available in CPDev engineering environment for programming controllers.**

***Index Terms*—A* algorithm, graphics editors, IEC 61131-3, searching path.**

I. Introduction

GRAPHICS editors for various diagrams allow the user to create connections between symbolic blocks. The simplest approach is to draw an exact line or polyline by the user. It can be cumbersome and lead to errors, especially when the block, which is a start or end point for the connection, is moved later, because the connection is not updatable. A better solution is to draw connections between blocks automatically and also update them when necessary, without user attention. This scenario makes creation of the diagram easier and limits a number of errors. It is consistent with a process of diagram creation that starts from designing the first working version of control algorithm, without focusing on legibility of the diagram, and then moving elements to get more readable design that is easier to understand and maintain.

To implement such an approach, a dedicated algorithm is required. It complies with some rules corresponding to the diagram type (e.g. restrictions on overlapping lines and crossing other blocks) and takes into account user preferences (directions changed rarely). Moreover, the algorithm should determine all paths on the fly, immediately after creating or moving an element.

Results obtained during research caused an implementation of the mechanism of automatic connections finding in graphics editors inside the CPDev engineering environment. They work smoothly also on devices with limited resources and allow the designer to create and modify connections in almost imperceptible way. A process of updating connections between multiple elements typically takes only a few milliseconds.

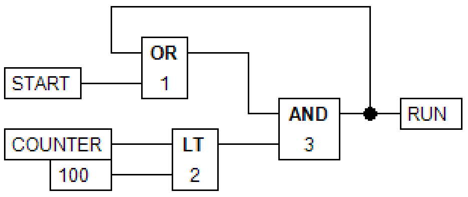


Fig. 1. FBD program that represents the following formula: $RUN = (RUN \vee START) \wedge (COUNTER < 100)$.

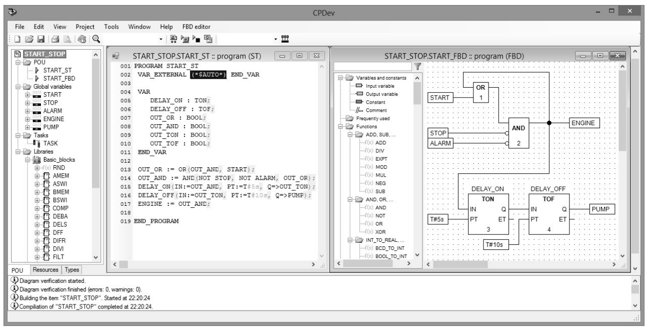
In this paper, the algorithm with an application for Function Block Diagram (FBD) graphical language from IEC 61131-3 standard is presented. The article is organized as follows. The second section reviews FBD language and CPDev engineering environment. A problem of path searching in a graph with analysis of A* algorithm modifications is described in the third section. The results of measurements and a short information about test software is presented in the fourth section.

II. FBD in CPDev Engineering Environment

Third part of the IEC 61131 standard [1] defines five programming languages for industrial controllers. Textual languages include ST (Structured Text) and IL (Instruction List). FBD (Function Block Diagram) and LD (Ladder Diagram) are graphical, while SFC (Sequential Function Chart) is mixed and requires parts implemented in other languages.

In the paper, the authors focus on FBD, that is a graphical language allowing users to create control programs in a visual way. POUs (Program Organization Units, i.e. programs, function blocks, and functions) defined in this language consist of rectangles that represent variables, constants, instances of function blocks, and functions. All of them are connected with lines as shown in Fig. 1.

Graphical languages benefit from visual programming. Their features include legibility of diagrams, easiness of program understanding or modification, and a possibility of attaching printouts directly to the documentation. Engineering tools (e.g. Beckhoff TwinCAT [2], CoDeSys [3], Control Builder F [4]) contain graphics editors that allow users to design programs graphically. They support drawing and



```
PROGRAM START_ST
 VAR_EXTERNAL (*$AUTO*) END_VAR

 VAR
    DELAY_ON : TON;
    DELAY_OFF : TOF;
    OUT_OR : BOOL;
    OUT_AND : BOOL;
    OUT_TON : BOOL;
    OUT_TOF : BOOL;
 END_VAR

 OUT_OR := OR(OUT_AND, START);
 OUT_AND := AND(NOT STOP, NOT ALARM, OUT_OR);
 DELAY_ON(IN:=OUT_AND, PT:=T#5s, Q=>OUT_TON);
 DELAY_OFF(IN:=OUT_TON, PT:=T#10s, Q=>PUMP);
 ENGINE := OUT_AND;

END_PROGRAM
```

```
Diagram verification started.
Diagram verification finished (errors: 0, warnings: 0).
Building the item "START_STOP". Started at 22:20:24
Compilation of "START_STOP" completed at 22:20:24.
```

Fig. 2. CPDev IDE with editors of ST and FBD languages.

updating connections between blocks automatically, using some proprietary solutions, without any clues of the used algorithms and implementation details. Development of the CPDev engineering environment required another solution, thus the authors proposed using the A* algorithm with some adjustments and tuning of parameters, as described later.

CPDev (Control Program Developer) is an engineering environment [5] developed in the Department of Computer and Control Engineering at Rzeszow University of Technology (Poland). It can be used for programming PLC, and PAC controllers, mini-DCSs [6], and secure NCS systems [7], according to IEC 61131-3 standard [1]. The CPDev environment is universal and generates code in the form that can be executed on various target platforms, including AVR, ARM, x86, and FPGA. It is achieved by using a virtual machine executing an intermediate code [8]. The environment is open for controller constructors and engineers that can implement low-level procedures and add them to the virtual machine. Developers using CPDev can create own libraries with POUs and reuse them in multiple projects. CPDev consists of several parts, including integrated development environment (Fig. 2), compilers, translators, configuration tools, testing application [9], and visualization mechanism [10]. CPDev has been applied for ship control and monitoring systems from Praxis Automation Technology B.V. (Leiderdorp, the Netherlands) [11] and for small PAC controller in measurement-and-control systems from LUMEL S.A. (Zielona Gora, Poland) [12].

All kinds of POUs, i.e. programs, function blocks, and functions, can be created using CPDev graphics editors [13]. They are equipped with typical functionalities such as basic edition operations (adding, moving, copying, pasting), translation to ST code, conversion to XML format, and printing accordingly to a template. The editors provide also an execution mode to run programs with support of tracing variable values and breakpoints.

Automatic connection finding is one of the most important features. It generates a connection between two elements on the diagram (variables, functions, or function blocks) automatically. Therefore, the user can focus on implementation of the control software, without paying special attention to connections. In CPDev graphics editors the lines are drawn automatically, just after selecting the beginning and the end of the connection. The problem can be interpreted as finding a path in a graph. However, some simplifications and modifications have to be done due to short time requirement.

III. Path searching in a graph

The problem of finding the shortest path in a graph is one of typical problems in discrete mathematics [14]. The classical approach, like the Dijkstra algorithm [15], focuses on examining all possible paths to find the shortest one. However, such a solution is not performance efficient. As an extension to the classical approach, a number of BFS (*Best-First Search*) algorithms have been proposed. For the problem specified above, the authors have chosen the A* algorithm [16], which combines traditional approach (similar to Dijkstra's) and heuristic one involving a metric. The A* algorithm requires estimation of a distance between the current node and the target for every node in a graph. Such a distance must be predicted in an optimistic way, i.e. real distance cannot be shorter than estimated. Numerous modifications of A* algorithm have been described [17], as well as applications including finding optimal routing in wireless sensor networks [18], [19], shortest road on a map [20], or even solving motion correspondence problem in computer vision [21].

Basically, the A* algorithm in every step tries to go the most promising way, i.e. chooses such a neighbor node that is close both to the current track and to the target node. At first, estimated distance from the target is calculated for every node, denoted as *H score*. While traversing the graph, real distances from the start node to neighbors of the current one are evaluated. The *G score* for every node reflects the distance from the start point via the shortest path already examined. The G score for every node can be updated later if a shorter path to this node is found. The *F score* is a sum of G and H scores. It represents estimated cost of using this node while directing to the target. In every step a node with the lowest F score from so-called *open set* (containing nodes to be traversed) is selected, and added to the *closed set* (nodes already traversed).

Implementation of such an algorithm requires consideration of some data structures for these sets. Ordinary lists or hashtables can be used, but heaps or binary search trees (simple BST or self-balancing, e.g. red-black trees RBT) [22] usually turn out better. Selection of a node with the minimal F score from the open set (which is a priority queue) is one of the most time consuming parts in A* algorithm. The worst case computational complexity of such an operation is $O(n)$ for tables, $O(\log n)$ for balanced BST or RBT, and $O(1)$ for min heap. Choosing F score as a key for a complex data structure improves performance, but checking if the set already contains the node, by using structure with coordinates of the node as a key, is more convenient. Similarly, adding or removing a node from complex data structures, as well as updating their F score (and position in the structure) takes significantly more time. Thus, choosing the most efficient structure, as well as its key, in a given case requires some tests, as shown in the following section.

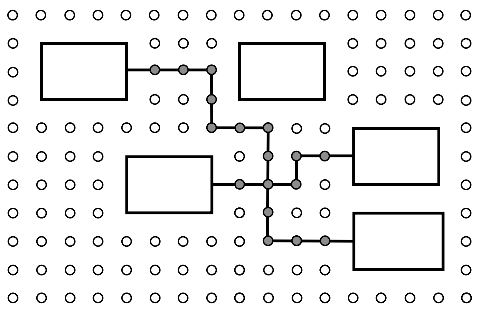


Fig. 3. Generated graph nodes.

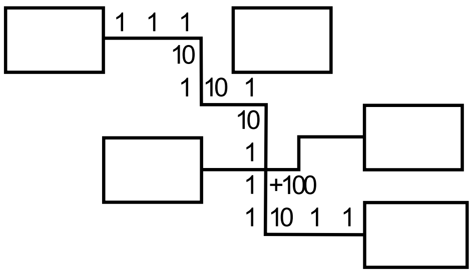


Fig. 4. Calculation of the path cost.

Another possibility for closed set implementation is addition of marks indicating whether the node has been already traversed. Such marks are included in internal class representation of the nodes. Thus, an additional data structure is unnecessary, which simplifies the implementation.

Finding an appropriate connection between elements in FBD diagrams can be considered as solving the shortest path problem in a graph. The connection should meet the following requirements:

- be found every time if elements are placed on the diagram correctly,
- pass round elements placed earlier,
- limit a number of intersections with other lines,
- change direction rarely,
- support additional margin around elements.

In our solution such a graph is created automatically. The nodes are simply diagram grid points, as shown in Fig. 3. Elements placed on the diagram (variables, functions, and function blocks) remove some nodes from the graph. Arcs connect the nodes vertically and horizontally, according to the neighborhood of the grid points. Initially weights of all arcs are equal. The start and end nodes (in the graph) are defined by positions of the elements, which should be connected.

A structure of the graph, which nodes represent grid points, is suitable for searching the shortest path using A* algorithm. The Manhattan (taxicab) distance is a natural choice for heuristic function to estimate a connection cost in such a case. The distance is calculated according to the formula $d(n_1, n_2) = |x_1 - x_2| + |y_1 - y_2|$, where $n_i = (x_i, y_i)$ denotes node i with coordinates x_i and y_i. Among other metrics, that could be also considered, the maximum (Chebyshev) one seems also promising, calculated as $d(n_1, n_2) = \max(|x_1 - x_2|, |y_1 - y_2|)$.

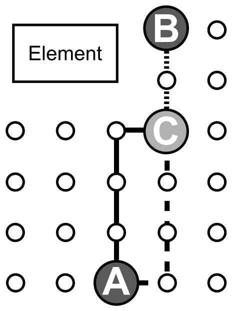


Fig. 5. Variants of the path between A and B nodes.

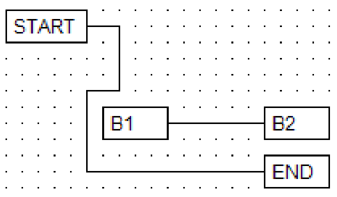


Fig. 6. Connections found with line cost intersection 50.

In order to find an optimal connection some additional conditions must also be considered while analyzing the diagram. Firstly, connection lines can not overlap, and can cross only when necessary. Overlapping can be easily solved by removing the arcs in the graph, which connect nodes representing fields that are already parts of any line. Crossing can be reduced by adding a high penalty for the G score. Secondly, lines in the diagram should go straight and change direction rarely. It can be achieved by setting an appropriate penalty on the G score, depending on a direction of actual path. An example of path cost calculation according to these rules is shown in Fig. 4.

It is worth mentioning, that G score rules sometimes lead to calculating different costs for a path between two nodes, depending on the previous path. Such a case is shown in Fig. 5. Considering the paths between A and B nodes, the algorithm can choose either one marked by a solid line or by dashed. In both cases the connection crosses C node. Two paths between A and C have the same cost, but the cost of the path between C and B is different, depending on the previous path. If the solid line between A and C is chosen, the connection changes direction in C towards B, thus some penalty for the G score is added.

A method for adding nodes to the structure representing the open set is another issue. In every step one node with the lowest F score in the open set is selected, but sometimes there may be many nodes with the same minimal F score. Selection of the node in such a case depends on the order of adding nodes in previous steps. There are several possibilities to be considered. New nodes can be added before or after previous ones, and they can be unsorted or sorted by direction. Such modifications affect searching time and shape of resulting path.

The costs of line intersection and direction change have

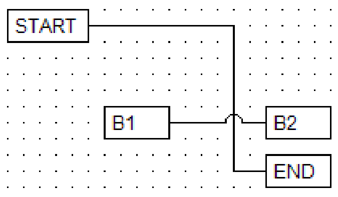


Fig. 7. Connections found with line cost intersection 5.

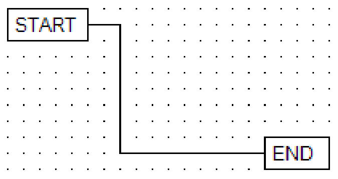


Fig. 8. Connection found with Manhattan metric.

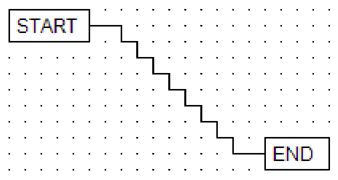


Fig. 9. Connection found with Maximum metric.

also an impact on the final path (Fig. 6, 7). If penalty for crossing a line is huge (as in Fig. 6), intersections will be avoided, but the path will go by roundabout way. The metric influences a shape of the path as well (Fig. 8, 9). For maximum metric the algorithm tries to minimize the distance in one of coordinates, the longest first. Such a metric can lead to generating ”stairs shaped” connection (Fig. 9) when a penalty for changing direction is low. Selection of appropriate cost values to find reasonable compromise in general case is not trivial.

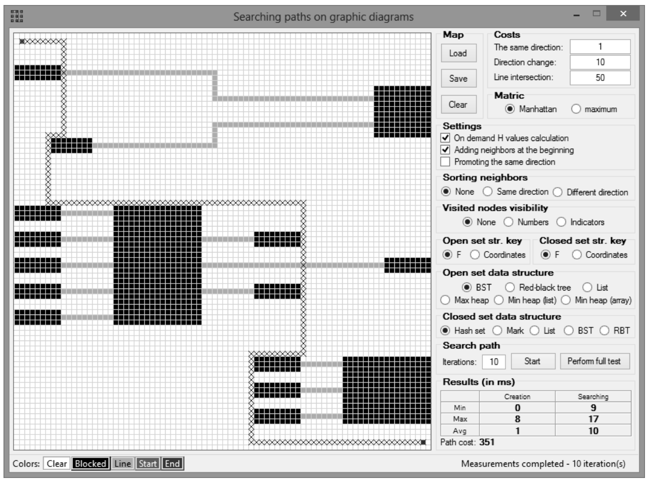
IV. Test software and performance results

A. Test software

Test results have been measured using a dedicated software with user interface shown in Fig. 10. That makes it possible to adjust settings and perform measurements for various combinations of parameter values. All tests are run on the map implemented as a matrix of integer values. They represent current states of fields: *start*, *end*, *blocked*, *line*, or *clear*. Available settings include:

- cost values (e.g. direction change or line crossing)
- open set data structure (BST, RBT, list, max heap, and min heap)
- closed set data structure (hash set, mark, list, BST, and RBT)
- keys for open and closed set structure (F value or coordinates)

The testing application consists of two parts, i.e. map and settings panel. In the example from Fig. 10, the map presents a


Searching paths on graphic diagrams
Map
Load
Save
Clear
Start
Perform full test
Colors: Clear Blocked Line Start End

Costs
The same direction: 1
Direction change: 10
Line intersection: 50

Matric
[✓ Manhattan] maximum

Settings
[✓ On demand H values calculation]
[✓ Adding neighbors at the beginning]
Promoting the same direction

Sorting neighbors
[✓ None] Same direction Different direction

Visited nodes visibility
[✓ None] Numbers Indicators

Open set str. key
[✓ F] Coordinates

Closed set str. key
[✓ F] Coordinates

Open set data structure
[✓ BST] Red-black tree List Max heap Min heap (list) Min heap (array)

Closed set data structure
[✓ Hash set] Mark List BST RBT

Search path
Iterations: 10

Results (in ms)

	Creation	Searching
Min	0	9
Max	8	17
Avg	1	10

Path cost: 351

Measurements completed - 10 iteration(s)

Fig. 10. The application for testing the mechanism of connection finding.

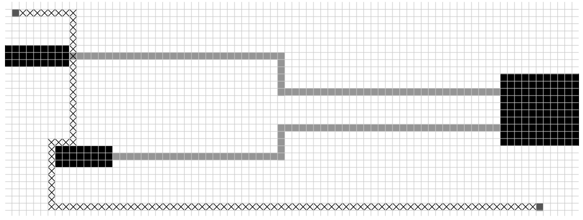


Fig. 11. The simple map for testing the mechanism of connection finding.

connection found by the mechanism between the start element (top-left) and the end (bottom-right). Some other elements, like blocks and variables from diagrams, are shown as black rectangles. They are connected by gray lines.

The measurements have been performed for a number of data sets, including simple and complex (Fig. 11, 12–15, respectively). The results have been calculated for various metrics, costs, structures, and keys for open and closed sets.

B. Simple map

The simple example from Fig. 11 consists of three blocks. Two of them represent input variables and the third one is an instance of function block. There are two lines from inputs to the block. The algorithm has to find a path between the start element (top-left) and the end (bottom-right) one. It is clear that for specified values of parameters the algorithm tries to avoid crossing lines, even if it requires to change direction and apply a longer path.

Improving performance is one of the most important reasons for testing the mechanism of finding connections. The results are presented in Table I. Following abbreviations have been assumed: *F* indicates F value, *C* – coordinates, *HS* – hash set structure, *Heap* – min heap structure, *Man* – Manhattan metric, and *Max* – Maximum metric. Cost is equal to 191 in all cases. Tests have been performed on PC with 2,5 GHz processor. Even for the simple example (Fig. 11) required times are different and depend on parameters, mainly on open and closed

TABLE I
RESULTS OF TESTING THE MECHANISM OF FINDING CONNECTIONS FOR THE SIMPLE MAP.

#	Open set		Closed set		Metr.	Required time [ms]
	Str.	Key	Str.	Key		
1	RBT	F	Mark	-	Man	0
2	BST	F	Mark	-	Man	1
3	BST	F	Mark	-	Max	1
4	RBT	F	Mark	-	Max	1
5	BST	F	HS	any	Man	1
6	RBT	F	HS	any	Man	1
7	Heap	F	Mark	-	Man	1
8	BST	F	BST	C	Man	3
9	List	F	Mark	-	Max	4
10	BST	F	RBT	F	Man	20
11	Heap	F	HS	F	Max	45
12	RBT	F	List	F	Max	64
13	List	F	List	C	Max	74
14	BST	F	BST	F	Man	118
15	List	C	BST	F	Man	125
16	Heap	C	BST	F	Max	174

set structures, as well as their keys. The metrics do not have such an important impact. The mechanism allows to calculate the path in the fastest way by using RBT or BST structures of the open set, and mark or hash set as a structure of the closed set (Table I, rows 1-6). The performance is significantly decreased by using a list or min heap as a structure of the open set, and BST, RBT, or list as a structure of the closed set (Table I, rows 15-16).

As mentioned in Section III, the results can be explained by internal concepts of various data structures and their computational complexity. In case of the open set it is important to select data structure that performs well while adding or removing an item, checking whether the structure contains specified item, or selecting an element with minimum value. For the closed set, only two operations should be well supported, i.e. adding an item and checking whether the structure contains specified item. Choosing a suitable combinations of structures for the open and closed sets significantly increases overall performance of the mechanism.

C. Complex map

The complex map (Fig. 12–15) consists of sixteen elements from the FBD diagram, i.e. ten input variables, three output variables, and three instances of function blocks. All of them are connected in more complicated way than before. The performance results for finding connections are presented in Table II. Cost is equal 351 in all cases.

The results confirm conclusions from the previous example. Again, the mechanism performs well with RBT or BST as a structure of the open set (Fig. 16), and with mark or hash set as a structure of the closed set (Fig. 17). However, differences between required times are higher, because significantly more operations must be performed while searching. In this case, the best combination of parameters finds the connection in 1 ms, but the worst in 781 ms.

TABLE II
RESULTS OF TESTING THE MECHANISM OF FINDING CONNECTIONS FOR THE COMPLEX MAP.

#	Open set		Closed set		Metric	Required time [ms]
	Str.	Key	Str.	Key		
1	RBT	F	Mark	-	Man	1
2	BST	F	Mark	-	Man	2
3	BST	F	HS	F	Man	3
4	RBT	F	HS	F	Man	3
5	Heap	F	Mark	-	Man	3
6	BST	F	RBT	C	Man	5
7	Heap	F	RBT	F	Max	103
8	Heap	F	BST	F	Max	781

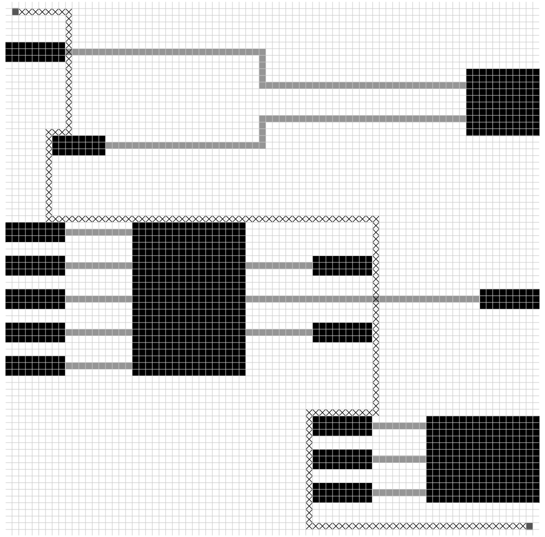


Fig. 12. Connections found for the Manhattan metric and 50 as a cost of line intersection.

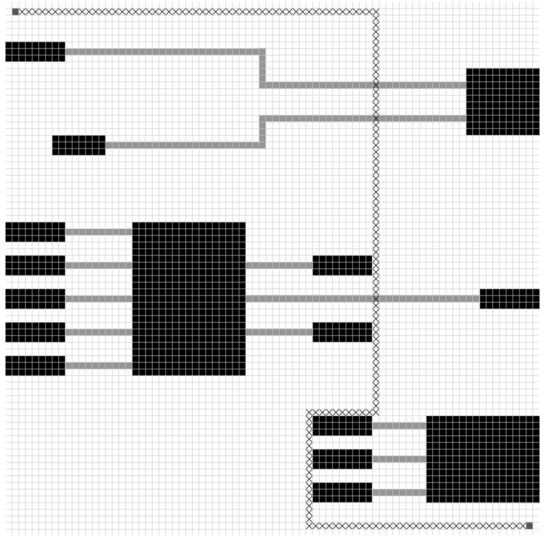


Fig. 13. Connections found for the Manhattan metric and 20 as a cost of line intersection.

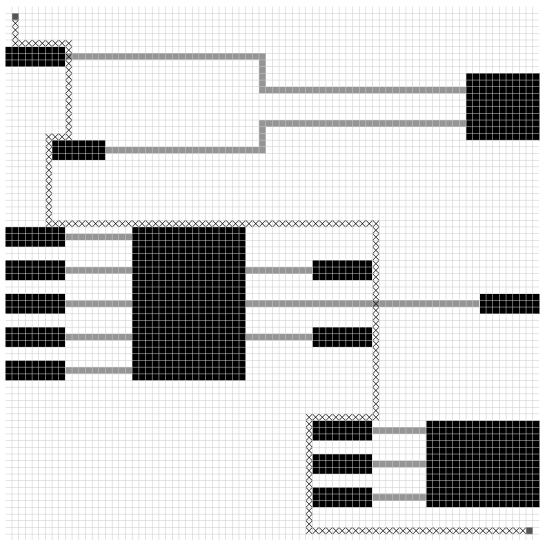


Fig. 14. Connections found for the Maximum metric and 50 as a cost of line intersection.

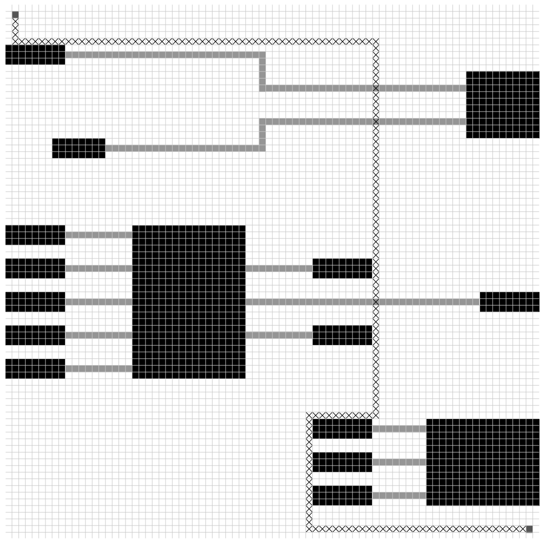


Fig. 15. Connections found for the Maximum metric and 20 as a cost of line intersection.

Using a proper open set structure (for fixed closed set structure) improves performance even a few times (Fig. 16). The difference is also seen in case of mark and hash set as structures of the closed set (Fig. 17). The mark can lead up to 50% increase of performance. Choosing a key for the closed set structure also affects performance, but only in case of structures other than mark.

A comparison between four combinations of structures of open and closed sets (RBT or BST, mark or hash set), depending on the metric, is shown in Fig. 18. The difference in case of the complex map is not high between BST and RBT used as the open set. Performance for mark and hash set as structures of the closed set is also similar. There is a difference in average time required to find connection when

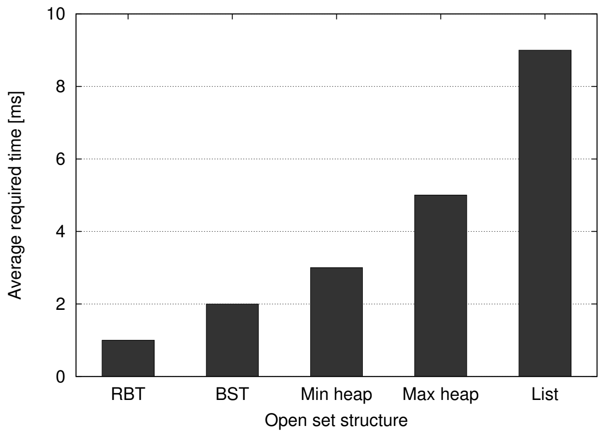


Fig. 16. Average required time for finding connection by open set structure (mark, F value as closed set structure and key, Manhattan metric).

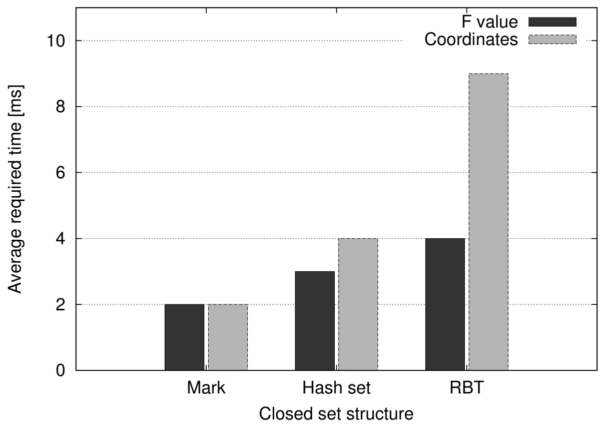


Fig. 17. Average required time for finding connection by closed set structure, open and closed set keys (BST as open set structure, Manhattan metric).

the mechanism uses Manhattan and maximum metrics. It is caused mainly by a different number of fields analyzed during searching. The difference is specific to the map.

Parameters have an impact not only on performance, but also on the path found. Depending on the metric and costs, it can change a direction more or less frequently, promote variants without crossing other lines, or even promote a specific direction. Differences are presented in Figs. 12–15 on a set of maps for the complex example.

The first connection (Fig. 12) is found for the Manhattan metric and 50 as a cost of crossing other lines. In this case the algorithm avoids crossings even by changing directions more frequently (see the top left part). In the second connection (Fig. 13) the cost of crossing is smaller and equals to only 20. It promotes crossing other lines instead of changing direction. Because of it, the first half of the connection differs significantly from the previous case. Similar conclusions can be made for the third and fourth connections (Figs. 14, 15), however, shape of the connection is different. The change

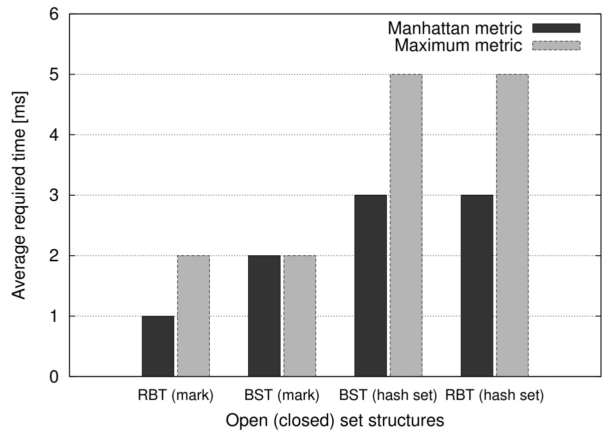


Fig. 18. Average required time for finding connection, depending on metric, open and closed set structures, with F values as keys.

is caused by the metric, either Manhattan (Figs. 12, 13) or maximum (Figs. 14, 15). The authors empirically found that values about 10-20 for direction change and 20-50 for line crossing give satisfactory results in case of FBD and LD diagrams.

V. Conclusion

The problem of finding a suitable path in a graph is typical in discrete mathematics. However, its application in FBD graphic diagrams to automatically find connections between elements requires some additional research and discussion. The mechanism meets a set of requirements which contain an execution in short time allowing to use on devices with limited resources. It supports a process of almost imperceptible updating and redrawing connections after moving any block on the diagram. Thus, the developer can easily adjust design of the diagram after creation of the first working version of POU. It can significantly increase legibility of the diagram and makes it easier to understand, modify, and maintain. Taken assumptions caused some modifications and tuning of the A* algorithm. As measured, values of parameters have significant impact on performance, length, and shape of the connection.

In the paper the authors considered some implementation details of A* algorithm, created a dedicated software for performance testing, and presented conclusions. The measurements indicate data structures that are efficient for the A* algorithm, and group of structures with performance problems. An impact on the shape of connection depending on path costs and metric is also described. All of these aspects are combined to tune properly the mechanism of finding connections in IEC 61131-3 Function Block Diagram editor implemented in the CPDev engineering environment.

References

[1] "IEC 61131-3 - Programmable controllers - Part 3: Programming languages," 2003.

[2] "Beckhoff TwinCAT website," 2013, http://www.beckhoff.com/english/twincat.

[3] "CoDeSys website," 2013, http://www.codesys.com.

[4] "Control Builder F website," 2013, http://www.abb.com/product/seitp334/ee37d357581192 adc12571ca00431c6e.aspx.

[5] "CPDev website," 2013, http://cpdev.kia.prz.edu.pl.

[6] D. Rzonca, A. Stec, and B. Trybus, "Data Acquisition Server for Mini Distributed Control System," in *Computer Networks*, ser. Communications in Computer and Information Science, A. Kwiecien, P. Gaj, and P. Stera, Eds. Springer Berlin Heidelberg, 2011, vol. 160, pp. 398–406.

[7] W. Rzasa, D. Rzonca, A. Stec, and B. Trybus, "Analysis of Challenge-Response Authentication in a Networked Control System," in *Computer Networks*, ser. Communications in Computer and Information Science, A. Kwiecien, P. Gaj, and P. Stera, Eds. Springer Berlin Heidelberg, 2012, vol. 291, pp. 271–279.

[8] D. Rzonca and B. Trybus, "Hierarchical Petri Net for the CPDev Virtual Machine with Communications," in *Computer Networks*, ser. Communications in Computer and Information Science, A. Kwiecien, P. Gaj, and P. Stera, Eds. Springer Berlin Heidelberg, 2009, vol. 39, pp. 264–271.

[9] M. Jamro, D. Rzonca, and B. Trybus, "Communication Performance Tests in Distributed Control Systems," in *Computer Networks*, ser. Communications in Computer and Information Science, A. Kwiecien, P. Gaj, and P. Stera, Eds. Springer Berlin Heidelberg, 2013, vol. 370, pp. 200–209.

[10] M. Jamro and B. Trybus, "IEC 61131-3 Programmable Human Machine Interfaces for Control Devices," in *Human System Interactions (HSI), 2013 6th International Conference on*, 2013, pp. 48–55.

[11] "Praxis Automation Technology B.V. website," 2013, http://www.praxis-automation.nl.

[12] "Lumel S.A. website," 2013, http://www.lumel.com.pl/en/.

[13] M. Jamro, "Graphics editors in CPDev environment," *Journal of Theoretical and Applied Computer Science*, vol. 6, no. 1, pp. 13–24, 2012.

[14] P. Festa, "Shortest Path Algorithms," in *Handbook of Optimization in Telecommunications*, M. G. Resende and P. M. Pardalos, Eds. Springer US, 2006, pp. 185–210.

[15] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[16] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100 –107, July 1968.

[17] A. V. Goldberg, "Point-to-Point Shortest Path Algorithms with Preprocessing," in *SOFSEM 2007: Theory and Practice of Computer Science*, ser. Lecture Notes in Computer Science, J. Leeuwen, G. Italiano, W. Hoek, C. Meinel, H. Sack, and F. Plasil, Eds. Springer Berlin Heidelberg, 2007, vol. 4362, pp. 88–102.

[18] I. S. AlShawi, L. Yan, W. Pan, and B. Luo, "Lifetime Enhancement in Wireless Sensor Networks Using Fuzzy Approach and A-Star Algorithm," *Sensors Journal, IEEE*, vol. 12, no. 10, pp. 3010–3018, Oct. 2012.

[19] K. Rana and M. Zaveri, "A-Star Algorithm for Energy Efficient Routing in Wireless Sensor Network," in *Trends in Network and Communications*, ser. Communications in Computer and Information Science, D. Wyld, M. Wozniak, N. Chaki, N. Meghanathan, and D. Nagamalai, Eds. Springer Berlin Heidelberg, 2011, vol. 197, pp. 232–241.

[20] F. Hahne, C. Nowak, and K. Ambrosi, "Acceleration of the A*-Algorithm for the Shortest Path Problem in Digital Road Maps," in *Operations Research Proceedings 2007*, ser. Operations Research Proceedings, J. Kalcsics and S. Nickel, Eds. Springer Berlin Heidelberg, 2008, vol. 2007, pp. 455–460.

[21] K.-Y. Eom, J.-Y. Jung, and M.-H. Kim, "A heuristic search-based motion correspondence algorithm using fuzzy clustering," *International Journal of Control, Automation and Systems*, vol. 10, pp. 594–602, 2012.

[22] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.