Journal of Intelligent & Robotic Systems (2018) 91:425–444
https://doi.org/10.1007/s10846-017-0748-6

L* Algorithm—A Linear Computational Complexity Graph Searching Algorithm for Path Planning

Adam Niewola[1] · **Leszek Podsędkowski**[1]

Received: 13 May 2017 / Accepted: 22 November 2017 / Published online: 11 December 2017

Abstract
The state-of-the-art graph searching algorithm applied to the optimal global path planning problem for mobile robots is the A* algorithm with the heap structured open list. In this paper, we present a novel algorithm, called the L* algorithm, which can be applied to global path planning and is faster than the A* algorithm. The structure of the open list with the use of bidirectional sublists (buckets) ensures the linear computational complexity of the L* algorithm because the nodes in the current bucket can be processed in any sequence and it is not necessary to sort the bucket. Our approach can maintain the optimality and linear computational complexity with the use of the cost expressed by floating-point numbers. The paper presents the requirements of the L* algorithm use and the proof of the admissibility of this algorithm. The experiments confirmed that the L* algorithm is faster than the A* algorithm in various path planning scenarios. We also introduced a method of estimating the execution time of the A* and the L* algorithm. The method was compared with the experimental results.

Keywords Computational complexity · Graph searching · A* Algorithm · Shortest path planning · Bucket priority queue

1 Introduction

The graph searching algorithms are used for various applications. One of them is the mobile robot optimal path planning. The optimal[1] path planning problem has been widely investigated in the last 50 years. Many path planning methods for collision-free optimal path finding were discovered and developed. They can be divided into two main groups:

- roadmap methods (e.g., Voronoi diagrams [23], visibility graphs [22], probabilistic roadmaps [24]),
- potential methods (e.g., potential fields [10]).

The main focus of interest of this paper is the global path planning problem. It can be defined as a process of finding an ordered set of intermediate points connecting the start point and the goal point. Each point of this set must be located in the free configuration space of the mobile robot.

One of the most popular graph searching algorithms is the A* algorithm. It was presented by Nilsson, Hart, and Raphael in 1968 [8]. It was widely investigated and commonly used for developing new modified methods of mobile robot path planning. In this paper, our algorithm will be compared with the best version of the A* algorithm (with the heap structured open list).

For comparison of various algorithms, the notion of the computational complexity is used [1]. It says how fast the number of the basic operations of the algorithm increases with the growth of the input of the algorithm. The space complexity says how fast the memory resources increase with the increase of the input of the algorithm.

In this paper, we present a new method of graph searching, in particular, applied to the path planning problem. It is a modification of the A* algorithm. The L* algorithm uses a modified heuristic cost function and a modified open list based on the bucket structure. It is

[1] For the most of the path planning methods, the optimal path means the shortest path. In this paper, the experimental work presents the solutions of the shortest path planning problem on 2D grid type maps as well as the optimal path planning problem with respect to applied optimization criteria.

✉ Adam Niewola
adam.niewola@gmail.com

Leszek Podsędkowski
lpodsedk@p.lodz.pl

[1] Institute of Machine, Tools and Production Engineering, Łódź University of Technology, Łódź, Poland

not necessary to maintain each bucket sorted. The proposed algorithm has a linear computational complexity. For this reason, it is called the L* algorithm. We present this graph searching algorithm in the path planning on a 2D occupancy grid map as a leading example. This approach was widely used in other works concerning path planning for the holonomic mobile robots which have the 2D configuration space, e.g., [9, 13, 19]. It is usually assumed that a mobile robot can move to one of four or eight neighboring cells. However, the same algorithms of graph searching can be used for the nonholonomic mobile robots with 3D configuration space (position x, position y, orientation o) [16]. This paper also presents the admissibility of the L* algorithm and shows our method of estimation of the execution time of both L* and A* graph searching algorithms. The main purpose of the paper is a presentation of the properties of the L* algorithm.

The rest of this paper is organized as follows: in Section 2 the problem is formulated. The most popular graph searching methods are described in Section 3. Section 4 presents the modification of the A* algorithm—the L* algorithm with the restrictions for its use. Section 5 presents the properties of the L* algorithm with its most significant advantage—linear computational complexity. Section 6 contains the results of the simulation tests of the path planning with the use of the A* and the L* algorithm. We compared our L* algorithm to A* with heap based open list as well as the bucket based open list. In Section 7, conclusions are presented.

2 Problem Statement

The A* algorithm nowadays is used in the global path planning. However, in the case of detecting new obstacles on the map, the path replanning is executed with the use of modifications of the A* algorithm (e.g., dynamic A* algorithm—the D* algorithm [19] and similar [18, 20]). These algorithms can react better to the changes in the workspace and do not need to explore all of the nodes again after detecting the change. Most of these dynamic path planning algorithms also need to use the list of nodes for expansion that has to be sorted by the key value.

Most of the graph searching algorithms use the open list as a priority queue of the nodes expansion. In the A* algorithm, the nodes on this list are sorted by the f cost which is the total cost of the node:

$$f(N) = g(N) + h(N), \tag{1}$$

where: $g(N)$ is the cost of the path from the start node to the node N and $h(N)$ is the heuristic estimate of the cost of the path from the node N to the goal. The $g(N)$ is calculated as follows:

$$g(N) = g(pred(N)) + dg(pred(N), N), \tag{2}$$

where: $pred(N)$ is the predecessor node of the node N in the graph and dg is the cost of the graph edge from the node $pred(N)$ to N.

If the heuristic function h fulfills the admissibility restriction, the A* algorithm always returns the optimal path from the start node to the goal node if it exists. Such algorithm is said to be admissible. The admissibility restriction can be written as follows:

$$0 \leq h(N) \leq h*(N), \tag{3}$$

where $h*(N)$ is the true cost of the path from node N to the goal node N_g.

Moreover, if the heuristic function fulfills the monotone requirement, it ensures that for every node selected for the expansion from the open list, the algorithm has already found the optimal path. The monotone restriction can be formulated as follows:

$$0 \leq h(N) \leq dg(N, N') + h(N'), \tag{4}$$

where N' is the successor node of the node N in the current spanning tree

In the A* algorithm and its modifications, the most effective approach is the use of the heap structure of the open list. In the path planning problems, the common approach is that the number of edges connected to each node is the same (or it can be bounded above). Given so, we can assume that the complexity of the algorithm depends on the total number of nodes in the graph. The binary heap open list structure provides the $O(nlog(n))$ computational complexity of the algorithm, where n is the total number of nodes in the graph. This feature, for huge maps, may cause longer computation time which rises rapidly with the growth of the open list.

Considering the increasing requirements for the data processing systems of the mobile robots, the necessity of the path planning on larger and larger maps, the need of considering more parameters (not only the length) in the optimal path planning problem, the idea of lowering the computational complexity of the graph searching algorithm can be crucial. Therefore, the L* algorithm can be a good alternative to the A* algorithm.

3 Related Work

Although the A* algorithm was presented in 1968, it is still one of the most popular graph searching algorithms used in the path planning problem. This algorithm also has many modifications, e.g., the LPA* algorithm [12], D* algorithm

[19], D* Lite [13] or Focused D* [20], Theta* algorithm [3]. All of these graph searching algorithms require the priority queue in order to find the optimal path. All of the algorithms mentioned above are based on the expansion of the nodes collected on the open list in the non-descending order of the key value. The typical approach is to use a binary heap as a data structure for the open list. It strongly affects the computational complexity of the algorithm. However, there were several efforts for decreasing the computational complexity of the graph searching algorithm.

One of the first approaches for decreasing of the computational complexity of graph searching was Dial's algorithm [4] based on Dijkstra's algorithm. This method assumes that the cost of each edge is expressed by a positive integer number. Therefore, the cost of each path in the spanning tree is also expressed by the integer value. The open list is divided into a set of buckets. Each bucket contains only the nodes with a specified value of the f cost. Because all of the nodes in each bucket have the same value of the cost, it is not necessary to sort the buckets, hence, the computational complexity of the graph searching algorithm can be decreased.

The main disadvantage of this algorithm is the lack of the heuristic function. It causes that many more nodes have to be expanded than in the A* algorithm in order to find the path to a specific goal node. Moreover, the assumption that the cost is expressed with the integers can be ineffective, especially for the cost functions including components dependent on height differences between the nodes, surface quality or terrain roughness (which are common in non-urbanized terrain path planning).

Bucket based open lists can also be used for the A* algorithm [5]. The method is similar to Dial's algorithm. The cost is expressed as the integer numbers. The open list is divided into the buckets. Each bucket contains the nodes with the same f value. In order to speed up the computation, the second level of the buckets can be used [7]. In the second level of buckets, the nodes are sorted by the g cost. The strong limitation of this method is the requirement of expressing the cost of the path by the integers. The integer-domain cost functions can be only used in straightforward graph searching problems. In more complex problems, the cost function is usually expressed by floating-point numbers. Obviously, the transition from floating-point numbers to integers can be easily introduced by using the 10^P multiplier (where P is a natural number greater than 0) and omitting the fractional part of the cost after multiplying. However, there is always a problem with selection a priori the appropriate value of the P parameter. Selection of too small multiplier may cause non-optimality of the final path (costs are rounded). Selection of too big multiplier value can cause the huge amount of empty buckets. The number of empty buckets limits the performance of the algorithm. However, it was experimentally proved that, in some workspaces, this approach provides better results than heap structured open list A* algorithm; in particular, when an appropriate value of the multiplier is selected. The bucket based open list is valuable only if the number of nodes with the same total cost is big and the difference between the cost of the nodes is relatively small [11].

If the f cost is expressed by the floating-point numbers, the nodes can also be divided into the buckets. Each bucket contains the nodes with a specific range of cost. In this situation, while selecting a new node from the open list for expansion, the algorithm has to determine the node with the lowest f cost—the only optimal node in the current bucket. Each bucket can be simply sorted by searching node by node. This approach is effective if the number of nodes in each bucket is relatively small. This approach can also give good results in several domains, but the selection of the appropriate range of buckets cost can be crucial. It can be tough to determine the proper range of buckets f cost before starting the graph searching. The number of nodes in each bucket strongly affects the computation time of the algorithm.

Reducing the computational complexity of the A* algorithm can also be obtained with the use of the modified method of the open list processing. The binary heap is one of the most common approaches. However, there are several modifications of this kind of data structure. The modified approaches ensure lower computational complexities in terms of specific conditions. One of them is the Fibonacci heap [6]. It is a modification of the binomial queues. The Fibonacci heap consists of a set of heap-ordered trees. It improves the operation of inserting a new node to the heap and the operation of decreasing the cost of a node in the heap (complexity $O(1)$). The operation of deleting the lowest cost node has the complexity $O(log(n))$.

A relatively novel approach is to improve the time of heap operation with the use of a strict Fibonacci heap [2]. This method is a pointer-based implementation of a standard Fibonacci heap. It achieves the complexity of the Fibonacci heap in the worst case. In the A* algorithm, the heap structured open list is a more general solution than the bucket based open list up to now.

4 L* Algorithm

4.1 General Idea of the L* Graph Searching Algorithm

In the typical A* algorithm used in the mobile robot path planning, the computational complexity is $O(log(n))$, where n is the total number of nodes. In the heap based A* as well as the bucket based A*, it is always necessary to determine the node with the lowest f cost (in the heap or the current

bucket). In the bucket based A* algorithm with the cost rounded to integers, it is always possible to receive the non-optimal final path due to the necessity of rounding the costs. For this reason, this approach is not in the scope of this paper.

Our purpose was to develop the algorithm (which will use the f cost expressed with the floating-point numbers) providing the $O(n)$ computational complexity. The main application of the algorithm should be the mobile robot path planning.

In our algorithm, the open list uses the buckets. The main difference between our L* algorithm and the bucket based A* algorithm is the optimality of the nodes in the current bucket. In the heap-based and bucket-based A* algorithm, we are sure that the node with the lowest f cost is optimal.

In the L* algorithm, we are confident that all of the nodes in the current bucket are optimal (when algorithm opens the bucket for the first time, we are sure that the shortest path to each node belonging to this bucket has already been found). This feature causes that the order of the nodes selected for the expansion within the current bucket can be arbitrary. Therefore, it is not necessary to choose the node with the lowest f cost for expansion from the current bucket. This effect was obtained by decreasing the power of the heuristic function and selecting the appropriate interval of the buckets. The heuristic cost is decreased by multiplying the cost by the w coefficient which has to be lower than 1 and greater than or equal to 0. The interval of the buckets depends on the w coefficient value. The interval has to be lower than or equal to $(1-w)dg_{min}$, where dg_{min} is the lowest possible cost of the edge in the graph (more details in Sections 4.2–4.5).

In the L* algorithm, the nodes in the current bucket are not sorted and can be selected in arbitrary order. Thus, we have $O(1)$ computational complexity of accessing any node in the bucket, and the complexity of the whole algorithm is $O(n)$, where n is the number of nodes.

The $O(n)$ complexity is a strong advantage of the L* algorithm. However, it is slightly decreased by the necessity of visiting more nodes than the A* algorithm (because the heuristic function for the L* algorithm is less informed than for the A* algorithm).

The assumptions for the use of the L* algorithm are listed in the Section 4.2. The only new assumptions which are not present in the A* algorithm are the value of the dg_{min} and dg_{max} (minimum and maximum cost of the edge in the graph). They have to be known a priori. The rest of Section 4 contains the guidelines for the open list building (bucket based open list) and the comparison of the L* algorithm with the A* algorithm. Section 5 contains the proof of optimality and computational complexity of the L* algorithm and our theoretical method for estimating the expected shortening of the computation time by the use of the L* instead of the A*. This method, as well as the empirical tests shown in paragraph 6, confirmed that the benefit from the use of the L* algorithm raises with the increase of the graph size. For this reason, we suggest using the L* algorithm for large graphs rather than for the small ones.

4.2 The Assumptions

The L* algorithm is an evolved A* algorithm with the non-standard structure of the open list. It is used to search a weighted graph $G = (V, E)$, where V is the set of vertices (nodes), and E is the set of edges connecting these nodes. The assumptions of the L* algorithm are as follows:

- weight dg of each graph edge e is positive but not infinite,

$$\forall(e \in E) \quad 0 < dg < \infty, \tag{5}$$

- the minimum value of graph edge weight dg_{min} is known a priori:

$$dg_{min} = min\{dg\}, \tag{6}$$

- the maximum value of graph edge weight dg_{max} is known a priori:

$$dg_{max} = max\{dg\}, \tag{7}$$

- heuristic cost function $h(N)$ fulfilling the requirement (3) and (4) is known,
- total f cost function for each node N can be written as follows:

$$f(N) = g(N) + h_L(N) = g(N) + wh(N), \tag{8}$$

where $0 \leq w < 1$ is the L* algorithm coefficient, which is constant during the graph searching process. The f cost function is allowed to be expressed by the floating-point values.

The knowledge of the dg_{min} and dg_{max} seems to be the most limiting, especially when dg cost depends on many arguments; not only the length of the path, and because dg cost has to fulfill restriction (5), the cost function has to be bounded below. Furthermore, because the equation of dg cost is known, the dg_{min} cost is known a priori. In the simplest path planning problem, the dg cost depends on the distance between the nodes on the map. Therefore, dg_{min} is the shortest distance between two adjacent nodes. In more complex problems, the dg_{min} has to be determined by the analysis of the cost function variability.

The knowledge of dg_{max} is not necessary for the L* algorithm optimality. It is only useful for the determination of the number of buckets for the open list priority queue—in the same way as in the bucket based A* algorithm. If we do

not know the dg_{max} a priori, the infinite list of buckets has to be used, but the L* algorithm maintains the optimality.

4.3 The Open List Structure

Because $h_L(N) < h(N)$, the L* algorithm is less informed than the A* algorithm [14], it means that each node expanded by the A* algorithm will also be expanded by the L* algorithm.

If the heuristic cost function $h_L(N) = wh(N)$ is used, the difference of the f cost between two adjacent nodes N and $N'(N'$ is the successor of $N)$ which are connected with the edge in the current spanning tree can be expressed according to the equation:

$$\begin{aligned} f(N') - f(N) &= dg(N, N') - w(h(N) - h(N')) \\ &\geq (1 - w)\, dg(N, N'). \end{aligned} \tag{9}$$

Marking df as a minimum difference between $f(N')$ and $f(N)$ in the whole graph, it can be written that:

$$df = min\{f(N') - f(N)\} = (1 - w)dg_{min} > 0. \tag{10}$$

Because for each pair of adjacent nodes the difference of the total f cost is positive and bigger than or equal to the value $(1 - w)dg_{min}$, the list of nodes that need to be expanded, called the open list O, can be organized in a table of non-sorted bidirectional sublists (buckets) S:

$$O = \{S_1, S_2, \ldots, Si, \ldots, S_n\}. \tag{11}$$

Each bucket S is the set of nodes with the f cost value from a specific f cost range. The bucket S_i can be written as follows:

$$S_i = \{N : f(N_s) + (i - 1)s \leq f(N) < f(N_s) + is\}, \tag{12}$$

where s is the interval (step) of the buckets. The interval of the buckets depends on the value of w coefficient (more details in Section 5). The example of the open list was shown in Table 1.

All of the nodes in Table 1 are shown in N_i^j configuration where i is the number of the node, j is the number of its predecessor in the current spanning tree. The open list is organized in the form of a table of unsorted bidirectional buckets. In order to determine which bucket S is appropriate for node N, the following equation is used:

$$S(N) = floor((f(N) - f(N_s))/s) + 1. \tag{13}$$

The idea of the L* algorithm is to keep each bucket unsorted and fulfill the admissibility restriction. In the subsequent part, the only considered version of the graph searching algorithm will be this one which also fulfills the monotone restriction. In the A* algorithm, fulfilling the monotone restriction means that after selecting any node for the expansion and deleting it from the open list, the optimal path to this node has already been found and this node will never be back on the open list [17]. In the L* algorithm, fulfilling the monotone restriction causes the same property as for the A* algorithm.

In the L* algorithm the buckets are non-sorted by the value of the f cost. Therefore, while selecting the nodes in any sequence from the considered bucket, the linear computational complexity can be maintained. Because we also want to keep the optimality of the path, it is necessary to maintain the nodes in the bucket optimal at the moment of accessing the bucket for the first time. The proposed method to keep the nodes (in the current bucket) optimal is to provide that when any node is taken for expansion from the considered bucket, its neighbors are added to one of the next buckets, never to the same bucket. If they were added to the same bucket, the priority queue would be damaged because of the possibility of existing the multiple paths to the same node. In the classic bucket based A*, the neighbor can be added to the same bucket as the parent. While, in the L* algorithm, the searching of the bucket is omitted. The problem is to find the appropriate value of the buckets interval s in order to fulfill the monotone and admissibility restriction. The interval s is strictly dependent on the minimum df cost increase between two adjacent nodes.

In the following part, it will be proven that for $s \leq df$, the L* algorithm fulfills both monotone and admissibility restriction, and provides finding the optimal path with the linear computational complexity.

4.4 The L* Algorithm—Step by Step

The L* algorithm is presented in a comparison to the A* algorithm. The comparison is presented in Table 2. The A* algorithm is shown in the case that both admissibility and monotone restrictions are fulfilled.

In the first step of the L* algorithm, the open list is initialized. Required parameters for the open list building

Table 1 The structure of the open list (example)

Bucket number	Bucket 1	Bucket 2	Bucket 3	Bucket 4	…
f cost range	< f(N$_s$), f(N$_s$)+s)	< f(N$_s$)+s, f(N$_s$)+2s)	< f(N$_s$)+2s, f(N$_s$)+3s)	< f(N$_s$)+3s, f(N$_s$)+4s)	…
Nodes	N_1	N_2^1	N_5^1	N_8^2	
		N_3^1	N_6^2	N_9^2	
		N_4^1	N_7^2		

Table 2 Comparison of the L* and the A* algorithm

```
(010) procedure Astar(G,Ns,Ng,dg,h)
(020)   initialize open list (OL)
(030)   put Ns to the OL with f = h(Ns),
        put Ns into tree T
(040)   pathExist = false
(050)   while OL is not empty
(060)       find N node with the lowest f cost
            on the OL, delete N from the OL
(070)       if N = Ng then
(080)           pathExist = true
(090)           goto label A
(100)       end if
(110)       for every N′ adjacent to N
(120)           if N′ is not marked 'visited' then
(130)              add N′ to T with a pointer toward N
(140)              f(N′)=g(N)+dg(N,N′)+h(N′)
(150)              insert N′ to open with f = f(N′)
                   and mark N′ 'visited'
(160)           else
(170)              if g(N)+dg(N,N′) < g(N′) then
(180)                  modify T by redirecting
                       pointer of N′ toward N
(190)                  f(N′)=g(N)+dg(N,N′)+h(N′)
(200)                  delete N′ from the open list
(210)                  insert N′ to open with f=f(N′)
(220)              end if
(230)           end if
(240)       end for
(250)   end while
(260)   label A
(270)   if pathExist = true then
(280)       return reconstructed path by
            tracing the pointers from Ng to Ns
(290)   else
(300)       return failure
(310)   end if
(320) end procedure
```

```
(010) procedure Lstar(G,Ns,Ng,dg,h,w,dgmin)
(020)   compute f0 = wh(Ns) and df,
        initialize open list—table of buckets (OL)
(030)   put Ns to the first bucket of the open list
        OL(1) with f = wh(Ns), put Ns into tree T
(040)   pathExist = false
(045)   currentBucketReadIdx = 1
(050)   while OL is not empty
(060)   for every node from the current bucket
(065)       get the first node from the
            current bucket N = OL(currentBucketReadIdx),
                delete N from the current bucket
(070)           if N = Ng then
(080)               pathExist = true
(090)               goto label A
(100)           end if
(110)           for every N′ adjacent to N
(120)               if N′ is not marked 'visited' then
(130)                   add N′ to T with a pointer toward N
(140)                   f(N′)=g(N)+dg(N,N′)+wh(N′)
(145)                   SL = floor(f(N′)-f0)/df) + 1
(150)                   insert N′ to the first position of
                        the bucket OL(SL) and mark N′
                        'visited'
(160)               else
(170)                   if g(N)+dg(N,N′) < g(N′) then
(180)                       modify T by redirecting
                            pointer of N′ toward N
(190)                       f(N′)=g(N)+dg(N,N′)+wh(N′)
(195)                       SL=floor(f(N′)-f0)/df)+1
(200)                       delete N′ from the previous
                            open list bucket
(210)                       insert N′ to the first position
                            of the bucket OL(SL)
(220)                   end if
(230)               end if
(235)           end for
(240)       end for
(245)       currentBucketReadIdx++
(250)   end while
(260)   label A
(270)   if pathExist = true then
(280)     return reconstructed path by
          tracing the pointers from Ng to Ns
(290)   else
(300)       return failure
(310)   end if
(320) end procedure
```

are computed—the f cost of the start node $f_0 = h(N_s)$ and the minimum difference of the f cost of two adjacent nodes df in the whole graph. With the use of these two parameters, the open list is divided into a set of empty buckets (line 020). Then the start node is placed in the first position of the first bucket (line 030) and is added to a spanning tree. The *currentBucketReadIdx* variable is set to 1 and the main while-loop begins reading the nodes from the first bucket.

In each iteration, one node from the bucket is collected and deleted from the currently set bucket (line 060). It does not matter which node from the bucket will be collected, but for simplification, it was assumed that only the first node is

always available. While adding a new node to the bucket, it will also be added to the first position.

The first step after acquiring the node from the open list is to check whether it is the goal node (lines 070–100). If the goal node is selected, then, the algorithm exits the while-loop (lines 090 and 260) and reconstructs the path (lines 270–280).

If a node selected for the expansion (line 060) is not a goal node, its successors are visited similarly to the A* algorithm. Newly visited nodes (line 120) are added to the buckets corresponding to their f cost values (lines 130–150). If the node was visited before and if its new g cost is lower than its current g cost (registered in the current spanning tree), it is deleted from the previous bucket and added to another one, always at the first position (lines 160–210).

After computing all of the nodes adjacent to the currently expanded node N, the algorithm selects for expansion the next node from the bucket indexed by the value of *currentBucketReadIdx* if the next node in the current bucket exists. Otherwise, it goes to the next non-empty bucket (line 246). The next while-loop iteration begins with collecting the first node from the bucket marked with the current value of index *currentBucketReadIdx*.

Leaving the while-loop is possible in one of two ways:

- that the algorithm finds the path from N_s to N_g and exits the loop by jumping to the label A,
- that the algorithm does not find the path from N_s to N_g because this path does not exist; then, the algorithm leaves the while-loop due to the lack of any more nodes on the open list (one node is deleted from the open list in each iteration and does not return to the open list, therefore, if the graph is finite, the open list becomes empty if the path from the start node to the goal node does not exist).

Depending on that which way the algorithm exited the while-loop, the value of variable *pathExist* is different. Therefore, the L* algorithm returns a reconstructed path if it exists or returns failure otherwise.

4.5 The Open List Operations

The main difference between the A* algorithm and the L* algorithm is the structure of the open list and the heuristic cost function. The heuristic cost function of the L* algorithm should be less informed than in the A* algorithm (the coefficient $w < 1$). It causes that after selecting an appropriate value of the interval of the buckets s, it is not necessary to determine the node with minimum f cost in the bucket. The open list is organized as a table of bidirectional sublists (called buckets). Each bucket stores only the pointer to the first node in the bucket and provides the direct access only to the first node in the bucket. The rest of the bucket is organized in the form of pointers attached to each node (typical for bidirectional lists). Each node stores the pointer to the parent node in the bucket and the pointer to the child node in the bucket. Each node also stores the bucket number which it is stored in.

The operations which the L* algorithm executes on the open list are the standard bidirectional list operations:

- adding a new node to the open list (the *insL()* procedure),
- deleting a node from the open list (the *delL()* procedure),
- getting the first node from the current bucket and deleting it from the open list (the *firstL()* procedure).

The procedure of adding a new node N to the open list can save this new node on any available position of the bucket corresponding to the f cost of the node (from the point of view of the L* algorithm admissibility). However, only for simplification, it was assumed that the algorithm has a direct access only to the first node in each bucket. For this reason, the new node can be stored only in the first position of the bucket which corresponds to its f cost. The complexity of adding a new node to the open list is *O(1)*.

The procedure of deleting a node N from a bucket S of the open list is executed when the L* algorithm detects a better path (with the lower g cost) than the path currently stored in the current spanning tree. It is necessary to know what is the parent node N_p and the child node N_c of the node N in the bucket S. These values are assigned to the node when it is added to the open list for the first time. It is also necessary to know which bucket the node N is stored in. The complexity of this procedure is *O(1)*.

The procedure of acquiring the first node in the bucket and deleting it from the open list *firstL()* requires the same operations as the procedure *delL()* described above. For *firstL()* procedure the node which has to be deleted is located in the first position of the list. The complexity of this procedure is *O(1)*.

5 Properties of the L* Algorithm

5.1 Proof of the L* Algorithm Admissibility and the Monotone Restriction

Fulfilling the monotone restriction for the A* algorithm means that when selecting for expansion the lowest f cost node from the open list, the optimal path to this node has already been found.

The L* algorithm is developed based on such kind of heuristic function h which fulfills the monotone restriction (4) for the A* algorithm. It can be modified to obtain a heuristic function which provides fulfilling the monotone restriction in the L* algorithm with the open list structured

according to the method described in Section 4. In the L* algorithm, we assume that the heuristic function fulfills the monotone restriction. For this reason, there it is not necessary to put the closed nodes to the open list again. There is also no need to come back to the closed buckets. In this case, first, the monotone restriction will be proven.

Theorem The L* algorithm which uses the heuristic function h_L less informed than the heuristic function h fulfilling the requirements (3) and (4):

$$h_L(N) = wh(N) \leq w(dg(N, N') + h(N')), \tag{14}$$

where w is the constant coefficient such that $0 \leq w < 1$, returns the optimal path from the start node N_s to any node N that it selects for expansion if the bucket interval s of the open list is lower than or equal to the lowest possible increase df of the f cost between two adjacent nodes N and N':

$$s < df = (1 - w)dg_{min} \leq f(N') - f(N), \tag{15}$$

where N' is the successor of N in the optimal spanning tree.

Proof We assume that the node N_0 is selected for expansion from the bucket S_k:

$$S_k = \{N : f(N_s) + (k-1)s \leq f(N) < f(N_s) + ks\}. \tag{16}$$

The f cost of the node N_0 is $f(N_0) = g(N_0) + wh(N_0)$. In order to prove that the g cost of this node $g(N_0)$ is optimal, we show that if in one of the following iterations a node N_1 (which has the edge $e(N_1, N_0) - N_0$ is a successor, N_1 is predecessor) is selected for expansion, the new cost $g_{new}(N_0)$ is greater than or equal to $g(N_0)$. One of the following options is possible:

- the node N_1 is in the same bucket as the node N_0, we can write:

$$-s < f(N_1) - f(N_0) < s, \tag{17}$$

- the node N_1 is in the further bucket than N_0, we can write:

$$0 \leq f(N_1) - f(N_0). \tag{18}$$

The node N_1 cannot be located in the previous bucket than N_0 because (according to the code of the algorithm presented in Table 2) when a current bucket becomes empty, then, the algorithm goes to the next non-empty bucket and never goes back.

If the node N_1 with the f cost $f(N_1) = g(N_1) + wh(N_1)$ is connected with an edge to the node N_0 (which is visited), it is necessary to check whether a new path to N_0 via N_1 is better than the path registered in the current spanning tree. A new g cost of the node N_0 is computed:

$$g_{new}(N_0) = g(N_1) + dg(N_1, N_0), \tag{19}$$

Based on that, we can write the difference between the new g cost and the previous g cost of the node N_0:

$$g_{new}(N_0) - g(N_0) = (f(N_1) - f(N_0)) + dg(N_1, N_0) - w(h(N_1) - h(N_0)). \tag{20}$$

We know that N_1 belongs to the same or the further bucket than N_0. Thus, according to the Eqs. 17 and 18, we can write:

$$f(N_1) - f(N_0) > -s. \tag{21}$$

Equation 21 corresponds to the worst option—when the nodes N_1 and N_0 belong to the same bucket, and the difference of their f costs is the difference between the lower and upper bound of the f cost of this bucket.

From the Eqs. 14 and 15 we know that:

$$dg(N_1, N_0) - w(h(N_1) - h(N_0)) \geq dg(N_1, N_0) + (-w)dg(N_1, N_0) \geq (1 - w)dg_{min} = df. \tag{22}$$

Including Eqs. 21 and 22 in Eq. 20, we have:

$$g_{new}(N_0) - g(N_0) \geq -s + df. \tag{23}$$

We know that $s \leq df$ (15). Given so, we can write:

$$g_{new}(N_0) - g(N_0) \geq 0. \tag{24}$$

Equation 24 is equivalent to a conclusion, that it is not possible to obtain a new path to the node N_0 (which has already been selected for expansion in one of the previous iterations), such that the new cost $g_{new}(N_0)$ of the path to N_0 via any node N_1 (selected for expansion later than N_0) is better than the cost of the node N_0 at the moment when it was chosen for expansion and deleted from the open list. It was proven that when the L* algorithm selects any node from the current bucket for expansion, the lowest g cost path to this node has already been found (every node in the current bucket is optimal). This node will never go back to the open list. Fulfilling the requirements (14) and (15) the L* algorithm provides that although the nodes from one bucket can be taken for the expansion in any sequence, the monotone restriction of any two adjacent nodes is fulfilled and the lowest g cost path to any node selected for expansion has already been found. □

Following this, it will be proven that the L* algorithm fulfilling the monotone restriction always returns the optimal path from the start node to the goal node if it exists.

Theorem (admissibility restriction) The L* algorithm fulfilling the monotone restriction always returns the optimal path from the start node to the goal node if it exists or returns failure otherwise.

Proof We note that the L* algorithm can terminate only if:

- it finds the path from the start node to the goal node,
- the open list is empty and there are no more nodes for expansion.

If the graph is finite, there is a finite number of nodes which have the f cost lower than cost $f(N_g)$. The L* algorithm expands all available nodes till the open list is empty. If it does not find the path, it terminates and returns failure. In the case of a finite graph, in each iteration, the L* algorithm adds to the open list the finite number of the successors and deletes from the open list one node which is currently expanded. It means that the L* algorithm must find the path to the goal node before the open list becomes empty if the finite path exists.

In the case of an infinite graph, it is not possible to return failure because an infinite branch of the spanning tree exists. If the path from the start to the goal exists, the algorithm must find it.

We proved that the L* algorithm must find a path from the start node to the goal node if it exists, instead of returning failure. Now we prove that the path which was found is optimal.

We know that the L* algorithm fulfilling the monotone requirement finds the optimal path to any node that it selects for expansion. Therefore, when it selected the N_g node for expansion, the optimal path to this node was found. Thus, when L* terminated by selecting N_g from the open list, the optimal path to this node was found.

In the case of a finite graph the L* algorithm, in the pessimistic case, finds the path to the goal in the last iteration (if the path exists). In the case of an infinite graph, on the open list, there is always at least one node from the optimal path. In the beginning, it is the start node, when it is expanded—all its successors—including the second point of the optimal path, and so on. Because each node has a finite number of successors and each edge has the cost $dg(N, N') \geq dg_{min}$, there is a finite number of nodes which have the cost f lower than the cost $f(N_g)+s$. It means that when the L* algorithm selects the goal node for expansion, the optimal path to the goal node is found. □

To sum up, for the use of the L* algorithm, it is necessary to determine the w parameter which fulfills the requirement:

$$0 \leq w < 1. \tag{25}$$

Then, the interval of the buckets can be computed depending on the values of w and dg_{min} with the use of the equation:

$$s \leq df = (1 - w)dg_{min}. \tag{26}$$

Fulfilling the requirements (25) and (26) always maintains the correctness of the algorithm (the optimality of the final path) and the linear computational complexity with respect to the number of nodes in the graph.

The selection of the w coefficient affects the number of iterations of the L* algorithm (the strength of the heuristic function) and the number of empty buckets on the open list. The bigger the w coefficient is, the less iteration the L* algorithm has to perform. However, the bigger the w coefficient is, the lower the interval of the buckets s becomes (the number of empty buckets may grow up). To find a balance between this two features, the suggestion (based on the experimental results) is to select the w value between 0,99 and 0,9999. It causes a slight increase in the number of iterations (while comparing with the A* algorithm) but the number of empty buckets is relatively small and does not affect the performance of the L* algorithm. For lower values of the w coefficient, the L* is less efficient.

5.2 Computational Complexity

In the A* algorithm, while adding or deleting a node from the open list, it is necessary to perform various operations which increase the computational complexity. In the best case, it is $O(nlog(n))$ where n is the total number of nodes in the graph. Due to increased requirements for the time of computation and necessity of processing large amounts of data while operating on large maps, lowering the complexity of the graph searching algorithm can significantly improve the calculation time of the path planning process.

In the L* algorithm, the operations of adding or deleting a node from the open list require a constant number of basic operations and their complexity is *O(1)*. The remaining operations which are in fact the same as in the A* algorithm also do not depend on the open list size. Their computational complexity is also *O(1)*.

The only non-constant operations are:

- the for-loop (line 110 in Table 2) whose execution time depends on the number of available adjacent nodes of the node selected for the expansion (this is also a feature of A* algorithm) but in the most of the path planning problems the number of successors is constant or can be bounded above),
- the procedure of incrementing the variable *currentBucketReadIdx* which depends on the number of empty buckets, numerical tests showed that there is usually a small number of empty buckets. Therefore, this non-constant number of operations can be omitted.

None of these non-constant operations depend on the number of nodes on the open list. In the pessimistic case, the number of basic operations while generating the successors

in the for-loop is constant (each node has a constant number of successors in the graph), and the number of increments of the *currentBucketReadIdx* variable is constant in each iteration. Because in the pessimistic case the while-loop repeats n times (algorithm needs to explore all of the nodes to reach the goal node in the last iteration), the number of basic operations of the whole algorithm can be expressed as:

$$G_{CC}(n) = k_{preproc} + nk_{main} + k_{postproc}, \tag{27}$$

where: n—number of nodes in the graph, $k_{preproc}$—constant number of basic operations before entering the main while-loop, $k_{postproc}$—constant number of basic operations after exiting the main while-loop, k_{main}—constant number of basic operations in a single iteration of the main while-loop.

Because $G_{CC}(n)$ can be upper bounded by the linear function $F_{CC}(n) = Cn$ (where C is a constant value), it can be written that the computational complexity of the L* algorithm is expressed as:

$$O(n). \tag{28}$$

5.3 State Complexity

The required memory resources of the L* algorithm depend on the implementation of the open list. In Section 4.5 one of the best possible implementations of the open list is presented. We note that in each iteration, the open list contains only the nodes with the f cost between f and $f + 2dg_{max} + s$ where f is the lower boundary of the first non-empty bucket and dg_{max} is the largest weight of the edge in the graph:

$$\begin{aligned} &max\{f(N_1) - f(N_2)\} = max\{g(N_1) - g(N_2)\} \\ &+max\{h(N_1) - h(N_2)\} \leq 2dg_{max} + s, \end{aligned} \tag{29}$$

where N_1 is the maximum f cost node and N_2 is the lowest possible f cost node on the open list (lower bound of the current bucket) in a given iteration.

For this reason, the number of required buckets can be significantly reduced. The best implementation is the use of the one-dimensional table containing the pointers only to the first node of each bucket. The size of the table is determined by the difference between the lowest and the largest f cost on the open list in each iteration. It can be upper-bounded by $2dg_{max} + s$. Given so, the size of the table K_S representing the open list can be expressed as:

$$K_S = floor(2dg_{max}/s) + 2. \tag{30}$$

Then, if the appropriate absolute bucket number $S(N)$ calculated for the node N according to Eq. 13 is greater than K_S, assuming that the buckets are numbered from 1 to K_S, the appropriate index of the bucket $S_K(N)$ can be obtained from the following equation:

$$\begin{aligned} S_K(N) &= S(N) - K_S \cdot floor((S(N) - 1)/K_S) \\ &= ((S(N) - 1) mod K_S) + 1. \end{aligned} \tag{31}$$

The rest of each bucket is organized in the structure of the bidirectional list. Assuming that n is the number of nodes in the graph, the number of memory cells required for the L* algorithm implementation can be expressed as:

$$G_{SC}(n) = K_S + n. \tag{32}$$

The function $G_{SC}(n)$ can be upper-bounded by $F_{SC}(n) = Cn$ (where C is a constant value), therefore, the state complexity of the L* algorithm can be expressed as:

$$O(n). \tag{33}$$

5.4 Comparison of the Execution Time of the L* Algorithm and the A* Algorithm

Although the L* algorithm has a linear computational complexity, better than commonly used graph searching algorithms, it is necessary to note that the L* algorithm is always less informed than the A* algorithm. Therefore, the main disadvantage of the L* algorithm is the necessity of visiting slightly more nodes than the A* algorithm. In this subsection, the analysis of the L* and A* algorithms approximated execution time will be performed with the use of an example of the path planning problem for the mobile robot on a flat 2D map with randomly placed obstacles.

5.4.1 Estimation of the Number of Iterations in the A* Algorithm and the L* Algorithm

The time of execution of both algorithms was estimated with the use of the following assumptions:

- a graph edge weight is expressed as $dg(N_1, N_2) == L_{N1,N2}$, where $L_{N1,N2}$ is the Euclidean distance between two adjacent nodes N_1 and N_2,
- the heuristic function $h(N) = wL_{N,Ng}$, where $L_{N,Ng}$ is the distance between N node and goal node, the coefficient $w = 1$ for A* algorithm and $0 \leq w < 1$ for L* algorithm, fulfills the admissibility and monotone restriction,
- the location of the obstacles on the map is randomized but with the constant average density—it causes that for each node N whose distance to the start node N_s is sufficiently large, it can be written that the cost of the optimal path from start to N can be expressed as $g(N) \approx bL_{Ns,N}$, where b can be expressed as $b = g(N_g)/L_{Ns,Ng}$ (b is always greater than or equal to 1),
- the number of required iterations k is expressed by the number of nodes with the f cost lower than the g cost

of the goal node $g(N_g)$. Thus, it can be written that $k = |I|$, where: $I = \{N : f(N) < g(N_g)\}$.

Assuming these requirements, the f cost of the optimal path to the node N can be expressed as:

$$f(N) = g(N) + h(N) \approx bL_{Ns,N} + wL_{N,Ng}. \tag{34}$$

Because $g(N_g) = bL_{Ns,Ng}$, the I set can be written as:

$$I \approx \{N : bL_{Ns,N} + wL_{N,Ng} < bL_{Ns,Ng}\}. \tag{35}$$

Because $b \geq w$, the I set can be upper-bounded by I_B set expressed as:

$$I \leq I_B = \{N : f(N) = L_{Ns,N} + L_{N,Ng} < b/wL_{Ns,Ng}\}. \tag{36}$$

The I_B set corresponds to the ellipse with the focal points located in the nodes N_s and N_g and the longer axis length $b/wL_{Ns,Ng}$. Therefore, the number of visited nodes can be expressed as $k < |I_B|$. The $|I|/|I_B|$ ratio depends on the value of the coefficient w, the distance between the start node and the goal node and a method of graph nodes creation (4-directional propagation, 8-directional propagation, 16-directional propagation).

Assuming that c is the average density of the nodes on the unit of the workspace area, the total number of nodes within the area of the bounding ellipse can be expressed as:

$$k < 0.25\pi cbL^2_{Ns,Ng}((b/w)^2 - 1))^{0.5}/w. \tag{37}$$

Assuming that for the A* algorithm $w = 1$, the upper bound of the number of iterations k_A performed by the A* algorithm can be found from Eq. 36:

$$k_A = 0.25\pi cbL^2_{Ns,Ng}(b^2 - 1)^{0.5}|I|/|I_B|. \tag{38}$$

Assuming that $w < 1$, Eq. 36 provides the upper bound of the number of iterations performed by the L* algorithm:

$$k_L = 0.25\pi cbL^2_{Ns,Ng}((b/w)^2 - 1))^{0.5}/w|I|/|I_B|. \tag{39}$$

An example of the boundary ellipse (36) and the true boundary curve (35) is presented in Fig. 1.

5.4.2 Estimation of the Number of Nodes on the Open List for the A* Algorithm

It is necessary to determine the size of the open list for the A* algorithm because the number of operations in the main while-loop depends on the size of the open list. It is not necessary to estimate the open list size for L* algorithm

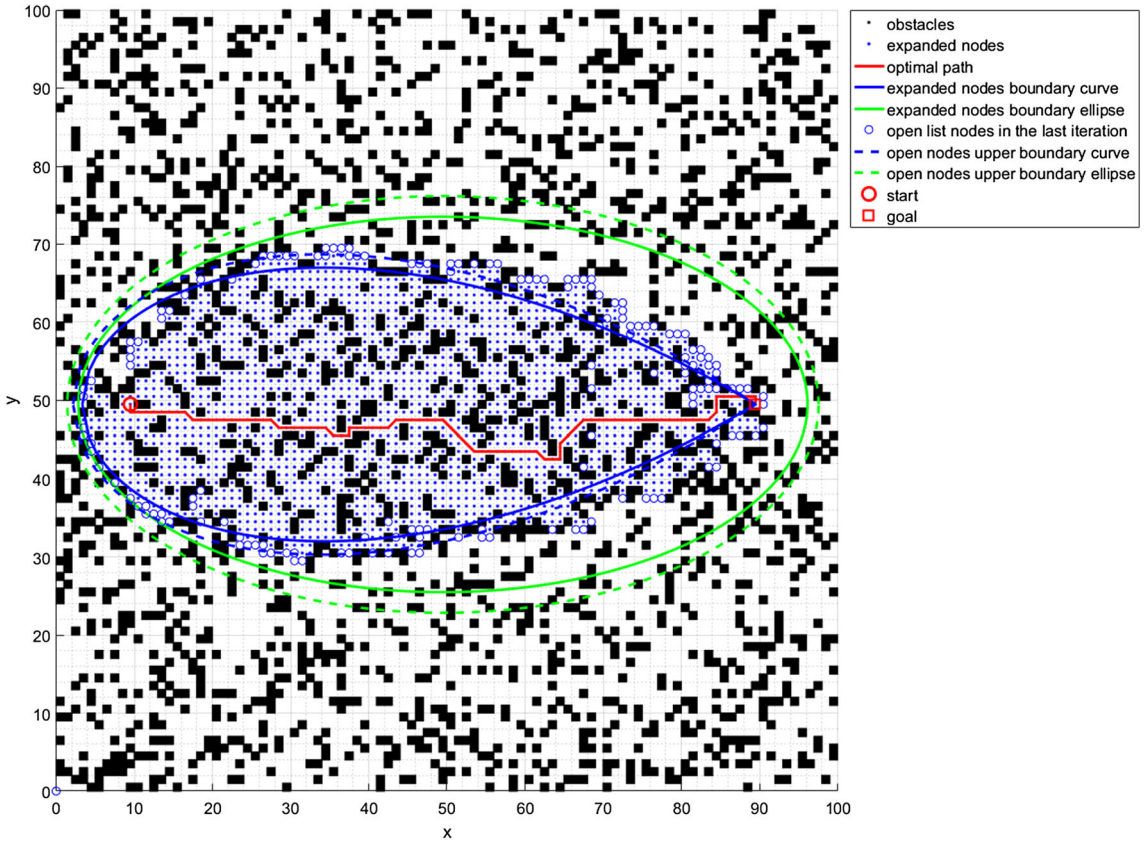

Fig. 1 An example of the path found by the L* algorithm with the expanded nodes and open list nodes

because the number of basic operations in the main while-loop of this algorithm does not depend on the open list size.

Assuming that for the A* algorithm the range of the f cost on the open list is r_f (fulfills the restriction (1) for $s = 0$), the number of nodes on the open list for the A* algorithm in the last iteration can be expressed as:

$$q_A = 0.25\pi c L_{start,goal}(3(b^2+b-1)-2b(b^2-1)^{0.5})r_f L/L_B. \quad (40)$$

Analogously, the number nodes in the last iteration of the L* algorithm can be expressed as:

$$q_L = 0.25\pi c L_{start,goal}(3((b/w)^2 + b/w - 1) - 2b/w((b/w)^2-1)^{0.5})r_f L/L_B, \quad (41)$$

where L/L_B coefficient represents the circuit ratio of the boundary ellipse and the true boundary curve of expanded nodes.

5.4.3 Execution Time Comparison

In order to estimate the profit of the use of the L* algorithm compared to the A* algorithm, it is necessary to assume the single operation time. It was assumed that t_L is the time required for a single while-loop in the L* algorithm. The time of execution of a single while-loop in the A* algorithm consists of a constant part t_A (which is nearly the same as for the L* algorithm), and the part depended on the size of the open list. The single f cost comparison (performed during heap operations) is indicated as t_h. The total time needed by the A* algorithm for a single while-loop execution can be expressed as $t_A + t_h log_2 q_A$. It is the time of the longest while-loop so assuming that the open list size is constant during the execution of the A* algorithm, the total time of execution of the algorithm can be estimated by $k_A(t_A + t_h log_2(q_A))$. In consequence, the relation between the time of execution of the L* and A* can be written as follows:

$$T_A/T_L = k_A(t_A + t_h log_2((q_A))/(k_L t_L). \quad (42)$$

Equation 42 can be used to determine the conditions $(L_{Ns,Ng}, w, c, b)$ when the L* algorithm is more efficient than the A* algorithm.

The presented method of estimation of the execution time and the numerical tests confirmed that the L* algorithm with w coefficient close to 1 always needs less time than the A* algorithm to determine the optimal path (except very short paths).

6 Results of the Simulation Tests

Both types of algorithms (A* and L*) were tested on various kinds of maps:

- 2D occupancy grid maps—test no. 1, map size from 1100 × 1100 to 5500 × 5500, maps with randomly located obstacles, obstacle occurrence probability was from 5 to 15%,
- a 2D map of 4000 × 2500 cells with randomly located obstacles with increased dg cost while exploring the nodes close to the obstacles—test no. 2,
- a 2.5D map of the surface of the Mars with the height values attached to each cell—test no. 3,
- 2D occupancy grid maps—test no. 4, all of the maps were taken from ([Online] http://www.movingai.com/benchmarks/) [21]:
  - with randomly located obstacles (obstacle occurrence probability was 10%—'random 512-10-0' map, and 30%—'random 512-30-9' map)— base map size 512 × 512,
  - 'maze 512-8-9' labyrinth-type map—base map size 512 × 512,
  - 'Room16_000' room-type map—base map size 512 × 512,
  - Starcraft 'Inferno' map—base map size 768 × 768,
  - Starcraft 'Frozen sea' map—base map size 1024 × 1024,
  - Warcraft 'The crucible' map—base map size 512 × 512.

Each map was tested with different $L_{Ns,Ng}$ distance values. In all tests, we used standard, 8-directional propagation. In the A* algorithm the open list was built with the use of binary heap structure (version 1—A*(H)). In test no. 4 we compared the L* algorithm with the A* algorithm with the heap based open list as well as with the A* algorithm with the bucket based open list. The interval of the buckets was the same as for the L* algorithm. Each bucket was organized with a bidirectional list searched node-by-node (version 2—A*(B)). The L* algorithm was implemented as a traditional serial algorithm. The experiments were run on the computer with configuration: Intel(R) Core (TM) i7-4720HQ CPU @2,60 GHz, 16 GB of RAM, 64-bit operating system. Both algorithms were implemented in C++. The programming framework was the same.[2]

[2]The source code will be published at: https://github.com/adamniewola/Lstar

Table 3 Test no. 1—experimental results

(1) $L_{Ns,Ng}$	(2) p_{obs}	(3) $T_{Aexp.}$ [ms]	(4) $k_{Aexp.}$	(5) $q_{Aexp.}$	(6) $T_{Lexp.}$ [ms]	(7) $k_{Lexp.}$	(8) $g(N_g)_{exp.}$	(9) $T_{Aexp.}$ / $T_{Lexp.}$	(10) $T_{Aest.}$/ $T_{Lest.}$
1000	0,05	11,45	51 643	2 605	5,80	51 799	1014,08	1,97	1,79
2000	0,05	41,98	175 144	5 304	19,68	176 207	2024,02	2,13	1,86
3000	0,05	102,88	402 499	7 996	47,99	406 612	3036,45	2,14	1,90
4000	0,05	173,63	661 229	10 221	84,10	664 984	4045,56	2,06	1,93
5000	0,05	283,88	1 036 119	13 217	137,52	1 041 588	5057,16	2,06	1,95
1000	0,10	20,48	86 222	2 394	11,35	86 443	1027,34	1,81	1,80
2000	0,10	76,76	303 841	4 764	41,22	304 555	2048,05	1,86	1,87
3000	0,10	189,12	713 886	7 203	99,96	715 693	3074,56	1,89	1,91
4000	0,10	320,47	1 145 662	9 708	172,50	1148 821	4091,13	1,86	1,94
5000	0,10	508,08	1 742 759	11 944	270,13	1 747 772	5111,01	1,88	1,96
160	0,15	0,66	2199	393	0,59	2204	165,56	1,59	1,82
1000	0,15	25,91	105 679	2 340	15,14	105 841	1038,52	1,71	1,80
2000	0,15	107,02	406 701	4 477	61,42	407 366	2072,07	1,74	1,87
3000	0,15	268,76	970 373	6 737	153,57	971 790	3114,32	1,75	1,91
4000	0,15	443,31	1 514 307	9 062	248,07	1 517 210	4137,52	1,79	1,94
5000	0,15	728,92	2 405 987	11 264	410,52	2 410 501	5173,14	1,78	1,97

(1) start-to-goal Euclidean distance, (2) obstacle occurrence probability (3) A* algorithm experimental execution time, (4) experimental number of iterations of the A* algorithm, (5) experimental maximum open size list in the A* algorithm, (6) L* algorithm experimental execution time, (7) experimental number of iterations of the L* algorithm, (8) experimental g cost of the goal node, (9) experimental execution time ratio, (10) estimated execution time ratio

6.1 Test 1

In this test, a grid type map with randomly placed obstacles was used. Map sizes were 1100 × 1100, 2200 × 2200, 3300 × 3300, 4400 × 4400 and 5500 × 5500. Obstacle occurrence probability p_{obs} was 5, 10 and 15%. The w coefficient for this test was set to 0,9999. The heuristic cost function was the Euclidean distance from a given node to the goal node:

$$h(N) = L_{N,Ng}. \tag{43}$$

The cost of each edge was dependent only on the distance between the nodes connected by this edge:

$$dg(N, N') = L_{N,N'}. \tag{44}$$

For comparison of the empirical experiments with the estimated values obtained from the equations presented in Section 5.4, the average computation time was estimated:

- the constant time of performing an expansion of the node taken from the open list by the L* algorithm, $t_L = 0,000160$ ms,

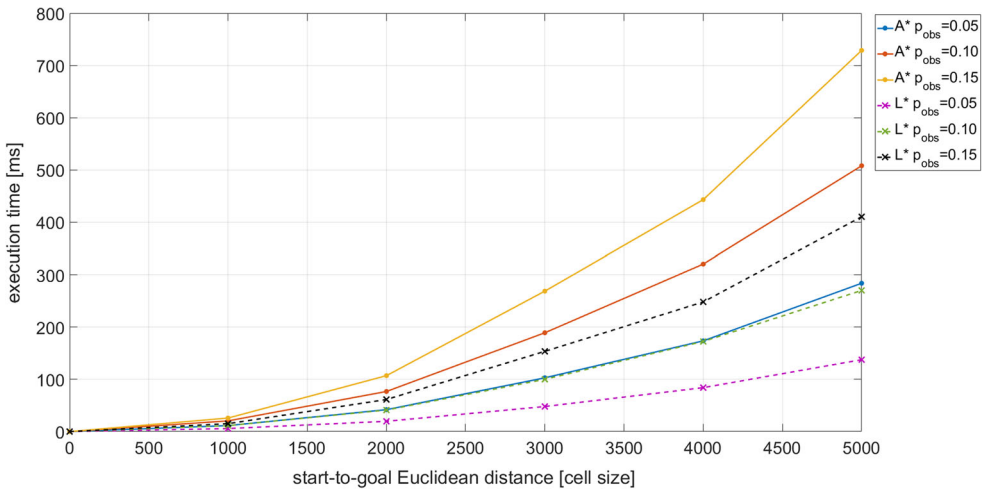

Fig. 2 The execution time of the L* and the A* algorithm in test no. 1

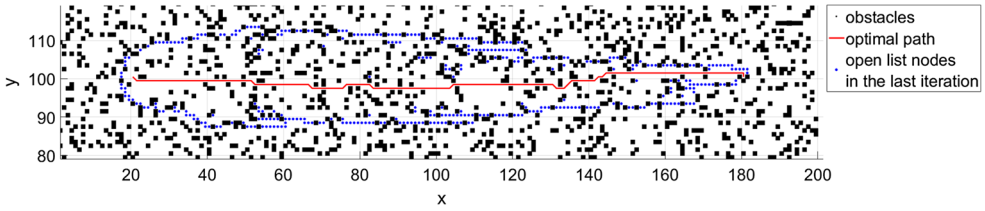

Fig. 3 The path found by the L* algorithm on the random occupancy grid map in test no. 1 (example for $L_{Ns,Ng} = 160$, $p_{obs} = 0, 15$)

Table 4 Test no. 2—experimental results

(1) $L_{Ns,Ng}$	(2) p_{obs}	(3) $T_{Aexp.}$ [ms]	(4) $k_{Aexp.}$	(5) $q_{Aexp.}$	(6) $T_{Lexp.}$[ms]	(7) $k_{Lexp.}$	(8) $g(N_g)_{exp.}$	(9) $T_{Aexp.}$ / $T_{Lexp.}$	(10) $T_{Aest.}$/ $T_{Lest.}$
1000	0,01	91,6	268 295	13 218	54,8	282 056	1169,3	1,67	1,85
2000	0,01	494,1	1 100 499	28 755	272,5	1 155 781	2260,5	1,81	1,89
3000	0,01	1182,7	2 225 743	44 379	616,3	2 358 170	3366,9	1,92	1,93
160	0,02	4,55	15 009	1 559	3,48	15 019	272,4	1,31	1,75
1000	0,02	212,2	590 682	12 963	121,4	600 835	1660,5	1,75	1,93
2000	0,02	1436,9	2 808 558	28 850	741,8	2 853 846	3312,2	1,94	2,00
3000	0,02	3478,1	5 844 759	42 925	1717,4	5 945 874	4854,8	2,03	2,04

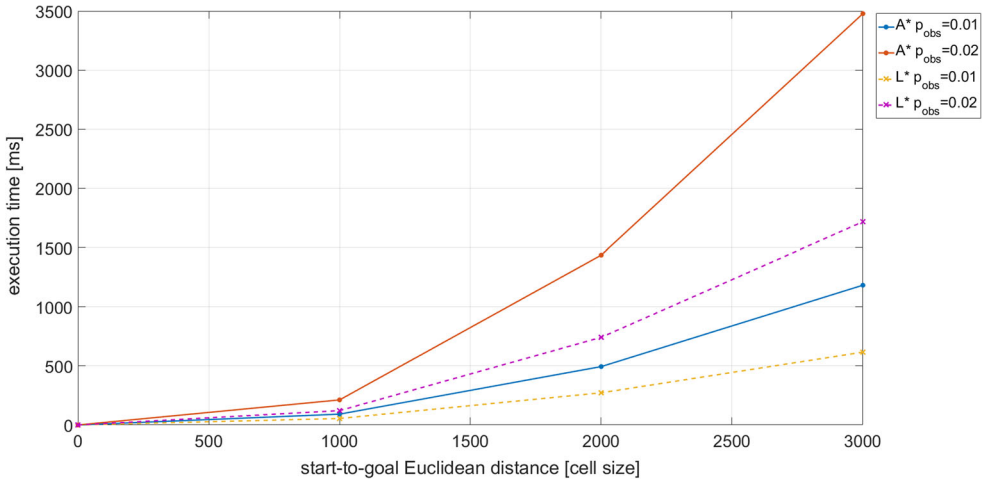

Fig. 4 Execution time of the L* and the A* algorithm in test no. 2

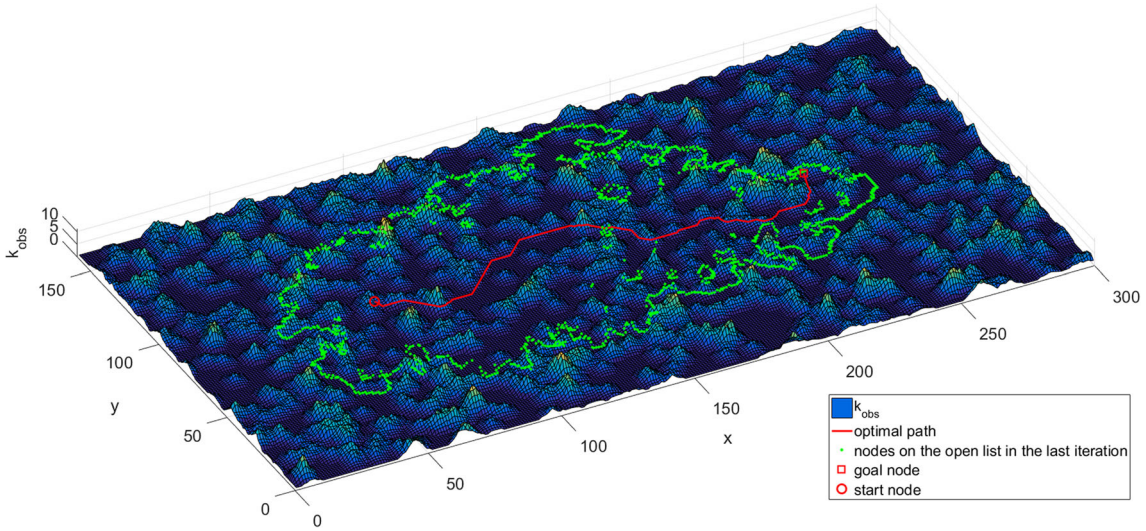

Fig. 5 The path found by the L* algorithm on the random occupancy grid map with k_{obs} coefficient used for dg cost computation in test no. 2 (example for $L_{Ns,Ng} = 160$, $p_{obs} = 0, 02$)

- the constant part of the time of performing an expansion of a node taken from the open list by the A* algorithm (without heap operations), $t_A = 0,000157$ ms,
- time of a single f cost comparison and swapping of the nodes in the heap, $t_h = 0,000011$ ms.

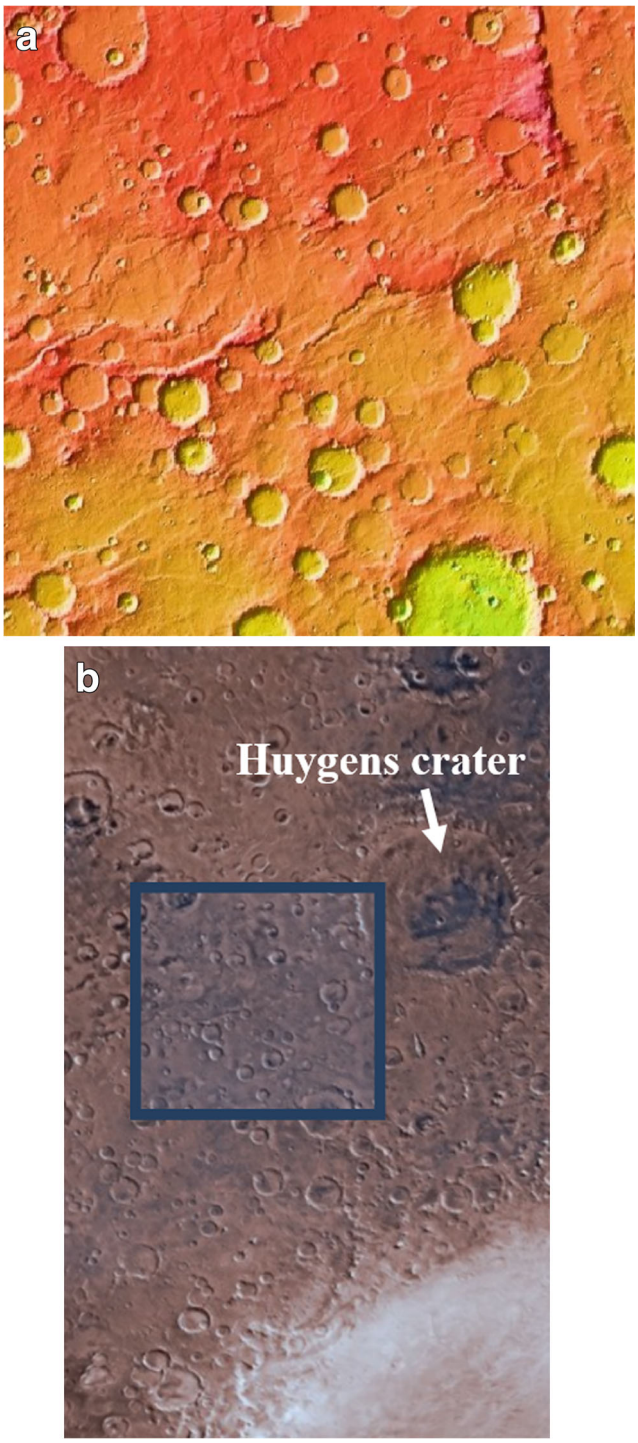

Fig. 6 The Mars map used in test no. 3 (**a** tested area elevation data, **b** tested area with respect to Huygens crater)

For computation of values t_A, t_L and t_h a micro-benchmark test was performed. We used average time values for different types of maps (maze maps, room maps, random maps, Starcraft and Warcraft maps ([Online] http://www.movingai.com/benchmarks/), 2.5D maps including terrain height). According to these values, it was possible to estimate the relationship between the time of the A* algorithm execution and the time of the L* algorithm execution. In the process of computing the estimated execution time ratios, the actual value (from the experiment) of b coefficient was used. The coefficient $|I|/|I_B|$ was set to 0,46. It is the average value of the occupancy grid map with randomly placed obstacles with p_{obs} value from 5 to 15%. The coefficient L/L_B was set to 0,92.

The results are shown in Table 3 and Fig. 2. The example of the path is presented in Fig. 3. In test no. 1 the A* algorithm needed 71–114% more time than the L* algorithm for finding the optimal path. The L* algorithm was faster, although it had to perform more iterations than the A* algorithm. The number of iterations grows nonlinearly with the distance between the start node and the goal node. In the worst case, the L* algorithm had to perform about 1% more iterations than the A* algorithm. This value depends on the value of w coefficient. The bigger the w coefficient is, the fewer iterations the L* algorithm has to perform. However, the interval of the buckets s becomes lower, and the number of empty buckets may grow up.

The size of the open list grows linearly in the function of the distance between the start and the goal node. In test the open list raised to 11264 nodes (the heap had 14 levels) for $L_{Ns,Ng} = 5000$ and $p_{obs} = 0,05$. It means that the *getFirstHeap()* procedure had to perform up to 13 f cost comparisons to rebuild the structure of the heap. In the case of adding a new node or changing the position of the visited node, the A* algorithm had to perform fewer operations than in the procedure of deleting the top node. Because all of these heap operations are not present in the L* algorithm, it is faster than the A* algorithm.

6.2 Test 2

In the 2nd test, we used the occupancy grid map with randomly placed obstacles, similar to the test no. 1. The difference was in the method of dg cost calculation. In test 2, the cost of each edge was increased in the area close to the obstacles:

$$dg(N, N') = k_{obs}(N')L_{N,N'}. \tag{45}$$

The coefficient k_{obs} was dependent on the distance to the obstacles. It was set to 1 if no obstacles were present in the neighborhood of the node N' and it was greater than 1 if the node N' was close to the obstacles. The closer to the obstacles the node was, the greater the value of k_{obs} was.

Table 5 Test no. 3—experimental results

(1) $L_{Ns,Ng}$	(3) $T_{Aexp.}$ [ms]	(4) $k_{Aexp.}$	(5) $q_{Aexp.}$	(6) $T_{Lexp.}$[ms]	(7) $k_{Lexp.}$	(8) $g(N_g)_{exp.}$	(9) $T_{Aexp.}$ / $T_{Lexp.}$
100	6,57	20670	1327	4,78	20751	693,77	1,37
300	57,48	197308	4905	43,34	198189	1872,5	1,33
500	110,51	365104	5506	69,1	366455	2688,04	1,60
700	186,21	624624	6353	113,32	626237	3316,39	1,64
900	277,63	770308	6397	180,44	771209	4274,59	1,54

Test 2 was performed on 4000 × 2500 map filled with the obstacles with the probability $p_{obs} = 0,01$ or 0,02. The w coefficient of the L* algorithm was set to 0,99. The heuristic cost function was the same as in test no. 1 (43). The start-to-goal Euclidean distance was 1000, 2000 and 3000. The coefficient $|I|/|I_B|$ was set to 0,37. The coefficient L/L_B was set to 1,55. The average r_f value was computed as the weighted average value of dg for the cells with obstacles and without obstacles. The results of this experiment are presented in Table 4 and Fig. 4. The example of the path found by both algorithms with respect to the map of the k_{obs} parameter is shown in Fig. 5.

The experimental results showed that the L* algorithm is faster than the A* algorithm for such kind of map. The A* algorithm needed approximately from 1.8 up to 2 times more time than the L* algorithm for executing the procedure of pathfinding. The open list of the A* algorithm reached the number of 44000 nodes (16 heap levels). The number of iterations of the L* algorithm in the worst case was 5% bigger than in the A* algorithm. This value could be decreased by increasing the w coefficient, but it could lead to the growth of the number of empty buckets (due to big differences in the weights of graph edges).

6.3 Test 3

In test no. 3 the 2.5D map was used. The map was obtained from the Mars surface digital elevation data. Based on the data from Google Mars ([Online] https://www.google.com/mars/) and NASA ([Online] http://marstrek.jpl.nasa.gov/) the map of 1000 × 1000 cells was prepared. The selected area was located near to the Huygens crater (Fig. 6). In test no. 3, it was assumed that the cost of each graph edge dg depends on the distance between the nodes (map cells) and the height difference between the cells according to the equation:

$$dg(N, N') = L_{N,N'} + w_{Hg}|(H_{N'} - H_N)|, \tag{46}$$

where w_{Hg} is the coefficient which represents the robot sensitivity to the height differences. The absolute value of the height difference was added to the edge cost in order to provide that the algorithm would avoid the height differences. The heuristic cost function was represented only by the Euclidean distance to the goal node (42).

The graph searching was repeated for different path lengths. The coefficient w_{Hg} was set to 10. The results of the execution time are presented in Table 5 and Fig. 7.

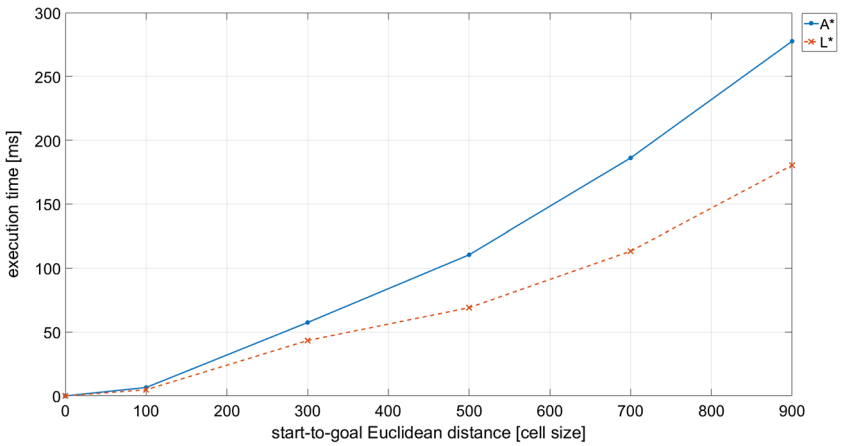

Fig. 7 The execution time of the L* and the A* algorithm in test no. 3

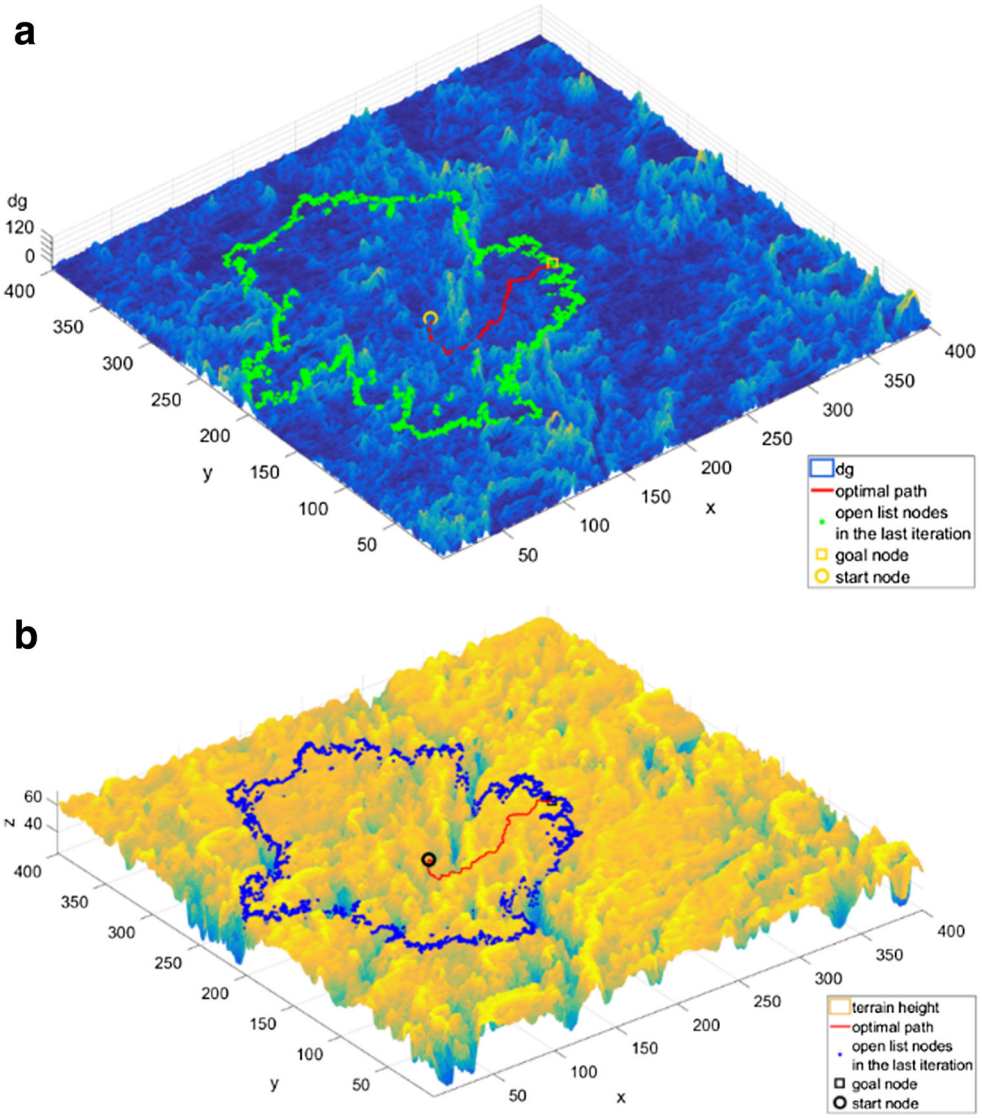

Fig. 8 The path found by the L* algorithm on the Mars map in test no. 3 (example for $L_{Ns,Ng} = 100$, **a** *dg* cost map, **b** terrain height map)

Figure 8 shows the path obtained by both algorithms for the $L_{Ns,Ng} = 100$ with respect to the map of average *dg* cost calculated for each cell.

The L* algorithm was faster than the A* algorithm in test 3. Due to the bigger differences in *dg* costs, the open list size increased significantly (to ~6400 when the start-to-goal distance was 900). However, the time execution ratio was lower than in the 1st test because the searched area was lower than in test no. 1 and the iterations ratio k_A/k_L was lower than in test no. 1.

6.4 Test 4

In this test, standard grid-type maps were used. Each map was tested in four versions—scaled 1× , 2×, 4× and 8×. The maps and the paths (start and goal coordinates) were taken from the research of Sturtevant ([Online] http://www.movingai.com/benchmarks/). For each map, 30 paths were taken into account. The results show average values for these 30 paths. The *w* coefficient for the L* algorithm was 0,99.

The results are presented in Table 6. The example of the paths found by both algorithms is presented in Fig. 9.

The results confirmed that the L* is faster than the A* algorithm for different path planning scenarios. We also compared the results with the A* algorithm with bucket based open list with the bucket interval the same as the bucket interval in the L* algorithm. In some cases, this implementation of the bucket based A* was faster than the heap based A* version (the maps are quite small, the number

Table 6 Test no. 4—experimental results

(1) Map name	(2) Scaled map size	(3) $T_{A(H)exp.}$ [ms]	(4) $k_{Aexp.}$	(5) $T_{A(B)exp.}$ [ms]	(6) $T_{Lexp.}$[ms]	(7) $k_{Lexp.}$	(9) $T_{A(H)exp.}$ / $T_{Lexp.}$	(10) $T_{A(B)exp.}$/ $T_{Lexp.}$
Random	512 × 512	57,4	63 837	49,3	45,7	69 650	1,256	1,079
512-10-0	1024 × 1024	181,6	184 142	156,3	142,2	210 505	1,277	1,099
	2048 × 2048	765,6	659 197	732,8	564,5	766 898	1,356	1,298
	4096 × 4096	3 850,8	2 453 643	4 558,5	2 876,9	2 891 018	1,339	1,585
Random	512 × 512	83,5	98 350	68,5	63,2	100 627	1,321	1,084
512-30-9	1024 × 1024	292,5	309 123	235,4	211,2	321 216	1,385	1,115
	2048 × 2048	1 249,0	1 140 376	1 026,2	852,2	1 192 984	1,466	1,204
	4096 × 4096	6 431,8	4 324 500	5 894,9	4 202,2	4 553 927	1,531	1,403
Maze 512-8-9	512 × 512	137,5	193 477	119,7	113,0	193 653	1,217	1,059
	1024 × 1024	552,1	723 850	468,4	441,8	724 974	1,250	1,060
	2048 × 2048	2 218,8	2 691 799	1 834,6	1 704,6	2 697 481	1,302	1,076
	4096 × 4096	110 081	10 842 246	7 947,9	7 204,5	10 860 352	1,399	1,103
Room16_000	512 × 512	98,7	111 350	80,62	74,4	114 568	1,327	1,084
	1024 × 1024	413,8	417 609	333,8	296,8	432 150	1,394	1,125
	2048 × 2048	1 903,5	1 626 457	1 570,7	1 271,3	1 685 967	1,497	1,236
	4096 × 4096	9 386,3	6 486 403	8 757,0	6 022,8	6 721 657	1,558	1,454
Starcraft	768 × 768	223,5	280 949	191,7	179,5	281 538	1,245	1,068
'Inferno'	1536 × 1536	933,7	1 122 541	759,8	713,5	1 125 007	1,309	1,065
	3072 × 3072	4 055,7	4 487 646	3 452,6	3037	4 497 654	1,335	1,137
	6144 × 6144	18 096	17 945 480	14 095,7	12 333	17 985 749	1,467	1,143
Starcraft	1024 × 1024	307,5	321 613	250,4	232,1	332 076	1,325	1,079
'Frozen sea'	2048 × 2048	1 396,0	1 350 497	1 134,4	969,8	1 390 130	1,439	1,170
	4096 × 4096	6 496,1	5 320 030	5 639,8	4 241,1	5 480 273	1,532	1,330
	8192 × 8192	33 806	21 254 077	30 935,9	21 459,9	21 893 183	1,575	1,442
Warcraft	512 × 512	18,9	23 280	16,3	15,7	24 298	1,204	1,038
'The crucible'	1024 × 1024	79,9	92 343	67,3	63	96 540	1,268	1,068
	2048 × 2048	352,5	368 251	294,3	262,3	385 211	1,344	1,122
	4096 × 4096	1 614,7	1 470 717	1 389,6	1 117,1	1 538 910	1,445	1,244

(1) map name, (2) map size after scaling, (3) heap based A* algorithm experimental execution time, (4) experimental number of iterations of the heap based A* algorithm, (5) bucket based based A* algorithm experimental execution time, (6) L* algorithm experimental execution time, (7) experimental number of iterations of the L* algorithm, (8) experimental g cost of the goal node, (9) experimental execution time ratio for heap based A* and L* algorithm, (10) experimental execution time ratio for bucket based A* and L* algorithm

of nodes in the buckets is not significant). Nevertheless, in all examples the L* was the fastest algorithm. It was confirmed that the larger the map is, the more the advantage of the bucket based A* algorithm is decreased (many nodes in each bucket has to be checked). The L* algorithm is not sensitive to the number of nodes in each bucket. Moreover, for the larger map sizes, its advantages are more significant.

7 Conclusion

The performed tests confirmed the advantages of the L* algorithm. The experimental research results showed that the difference between A* and L* execution time depends on the distance between the start and goal node and the shape and positions of the obstacles. The proposed method of dividing the whole open list into buckets (without necessarily determining the lowest f cost node within the bucket) can be implemented in more types of graph searching algorithms instead of selecting the minimum f cost node from the open list.

Our method of estimation of the execution time provides results close to experimental values. The difference becomes lower for the longer paths. For shorter paths, our assumption that the size of the open list is constant for the A* algorithm seems to be too strong. For this reason, the estimated values may not be accurate.

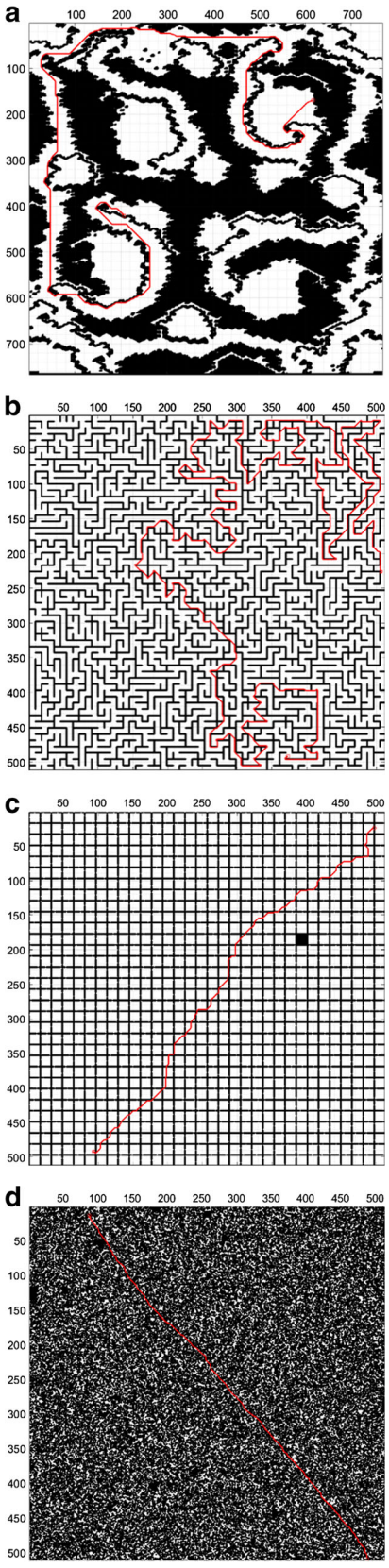

Fig. 9 The path found by the L* algorithm on the benchmark test maps in test no. 4: **a** Starcraft 'Inferno' 768 × 768 **b** Maze512_8_9 512 × 512 **c** Room16_000 512 × 512 **d** Random512_30_9 512 × 512

Most of the tests were performed on a flat terrain with the use of the cost functions depending only on the distance between the nodes. It is planned to expand the cost function and to include the elements depending on the terrain height, roughness and confidence factor, according to the workspace description proposed in [15]. Moreover, it is planned to verify the method for the nonholonomic robot path planning problem using the method of creating the graph nodes presented in [16] and for the dynamic path planning problem.

Although we tested the L* algorithm only for simple graphs for the path planning problem, the method is general—it works for any type of graph (finite or infinite) and ensures that the optimal path is found. In order to achieve the optimality of the L* algorithm, the strength of the heuristic function has to be appropriately decreased (with the use of the w coefficient). Our tests confirm that simply assuming w close to 1 (from 0,99 to 0,9999), the serial L* can be faster than the A* up to 2 times. The values of w bigger than 0,9999 can cause that the number of iterations of the L* algorithm decreases, but the number of empty buckets raises and the performance of the L* algorithm is reduced.

All of the tests were performed with the serial version of the L* algorithm. The future works will also consider the parallel L* algorithm (the open list processing and the nodes generation can be parallelized) with the use of GPU computation. This feature could improve the efficiency of the L* algorithm significantly.

References

1. Arrora, S., Barak, B.: Computational Complexity: A Modern Approach, 594 p. Cambridge University Press, New York (2009)
2. Brodal, G.S., Lagogiannis, G., Trajan, R.: Strict fibonacci heaps. In: Proceedings of the 44th Symposium on Theory of Computing, p. 1177 (2012)
3. Daniel, K., Nash, A., Koenig, S., Fernel, A.: Theta*, any-angle path planning on grids. J. Artif. Intell. Res. **39**, 533–579 (2010)
4. Dial, R.B.: Algorithm 360: shortest-path forest with topological ordering. Commun. ACM **12**(11), 632–633 (1969)
5. Edelkamp, S., Schroedl, S.: Heuristic Search—Theory and Applications. Morgan Kaufmann, San Mateo (2012)
6. Fredman, M.L., Trajan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. Assoc. Comput. Mach. **34**(3), 596–615 (1987)
7. Hatem, M., Burns, E., Ruml, W.: Faster Problem Solving in Java with Heuristic Search. IBM Developer Works (2013)

8. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination f minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)
9. Hernandez, C., Baier, J.A., Asin, R.: Making A* run faster than D*-lite for path-planning in partially known terrain. In: Twenty-Fourth International Conference on Automated Planning and Scheduling (2014)
10. Khatib, O.: Real-time obstacle avoidance for manipulators and mobile robots. Int. J. Robot. Res. **5**(1), 90–98 (1986)
11. Kliemann, L., Sanders, P.: Algorithm Engineering: Selected Results and Surveys. Springer International Publishing, Basel (2016)
12. Koenig, S., Likhachev, M., Furcy, D.: Lifelong planning A*. Artif. Intell. **155**(1–2), 93–146 (2004)
13. Koenig, S., Likhachev, M.: D* Lite. In: Proceedings of the AAAI Conference on Artificial Intelligence, pp. 476–483 (2002)
14. Latombe, J.-C.: Robot Motion Planning, p. 651. Kluwer Academic Publisher, Dordrecht (1991)
15. Niewola, A.: Rough surface description system in 2,5d map for mobile robot navigation. J. Autom. Mob. Robot. Intell. Syst. **7**(3), 57–63 (2013)
16. Niewola, A., Podsędkowski, L.: Nonholonomic mobile robot path planning with linear computational complexity graph searching algorithm. In: 2015 10th International Workshop on Robot Motion and Control (RoMoCo), pp. 217–222 (2015)
17. Nilsson, N.: Principles of Artificial Intelligence, p. 476. Tioga Publishing Company, Pato Alto (1980)
18. Podsędkowski, L.: Path planner for nonholonomic mobile robot with fast replanning procedure. In: IEEE International Conference on Robotics and Automation, pp. 3588–3593 (1998)
19. Stentz, A.: Optimal and efficient path planning for partially-known environment. In: Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94), pp. 3310–3317 (1994)
20. Stentz, A.: The focussed D* algorithm for real-time replanning. The focussed D* algorithm for real-time replanning. In: Proceedings of the International Joint Conference on Artificial Intelligence, pp. 1652–1659 (1995)
21. Sturtevant, N.R.: Benchmarks for grid-based pathfinding. IEEE Trans. Comput. Intell. AI Games **4**(2), 144–148 (2012)
22. Nguyet, T.T.N., Hoai, T.V., Thi, N.A.: Some advanced techniques in reducing time for path planning based on visibility graph. In: 2011 Third International Conference on Knowledge and Systems Engineering (KSE), pp. 190–194 (2011)
23. Ho, Y.-J., Li, J.-S.: Collision-free curvature-bounded smooth path planning using composite bezier curve based on voronoi diagram. In: 2009 IEEE International Symposium on Computational Intelligence in Robotics and Automation, pp. 463–469 (2009)
24. Zhiye, L., Xiong, C.: Path planning approach based on probabilistic roadmap for sensor based car-like robot in unknown environments. In: 2004 IEEE International Conference on Systems, Man and Cybernetics, pp. 2907–2912 (2004)

Adam Niewola received M.Sc. degree in Automatics and Robotics from the Faculty of Mechanical Engineering, Łódź University of Technology, Poland in 2011. His researches include mobile robots path planning and localization problem, sensor data fusion for orientation estimation, signal filtering and telemanipulation. He was involved in research project of a device for measuring femur displacement in damaged hip joint operations.

Leszek Podsędkowski received M.Sc. degree in 1989, PhD degree in 1993 from the Faculty of Mechanical Engineering, Technical University of Łódź, Poland. Since 2013 he is a full professor. He is a manager of the Automation and Robotics Division of Institute of Machine Tools and Production Engineering and supervisor of the specialization of Automatics and Robotics at the Faculty of Mechanical Engineering, Łódź University of Technology. His research areas include mobile robots path planning, localization and control, telemanipulators, in particular for cardiac surgery, robots construction and diagnostics and sensor systems for robotics. He was involved in many research projects, e.g. creation of RobIn Heart, manipulator for cardiac surgery.