A Performance Comparison of Rapidly-exploring Random Tree and Dijkstra's Algorithm for Holonomic Robot Path Planning

LUKAS KNISPEL, RADOMIL MATOUSEK
Institute of Automation and Computer Science
Brno University of Technology, Faculty of Mechanical Engineering
Technicka 2896/2, 616 69 Brno
CZECH REPUBLIC
lukas.knispel@gmail.com, matousek@fme.vutbr.cz http://sites.google.com/site/rrtexplorer

Abstract: The present article deals with a performance comparison of the Rapidly-exploring Random Tree (RRT) and Dijkstra's algorithm for the path planning of omnidirectional holonomic robots in known environments with static obstacles. The surveyed problem is applicable for a number of general 2D applications. It presents a unique implementation of the given problem by a novel grid approach to the Dijkstra's algorithm and custom optimized RRT solver. This article shows the analysis of performance comparison and graphic results of the given algorithms on open-space and maze-type maps of continuous. The conclusion of this article discusses benefits and disadvantages of both methods and suggests the choice for different configurations of the explored space.

Key-Words: Rapidly-Exploring Random Tree, RRT, Dijkstra's algorithm, holonomic robot, path planning, graph data structure, adjacency list, path optimization

1 Introduction

In these days, robots are able to solve complicated problems in many complex environments. Science of robotics is growing more important. One of the core problems in robotics is the *path planning*, (informally also known as the *"navigation problem"* or *"piano mover's problem"*). Generally, robot path planning is concerned with generating a suitable path for the robotic device, avoiding any possible collisions with known or unknown obstacles in the real space (Fig.1b).

There are two main groups of path planning algorithms based on the awareness of the robot about its surroundings – informed and uninformed methods. We speak about informed path planning when the robot has a complete knowledge of the environment around – both the information about the obstacles and the information about the position of the starting and finishing configuration.

This article approaches to the robot path planning by finding sub-optimal path for the robot, avoiding possible collisions with known obstacles in the given continuous space based on reality. It displays informed methods, which can be handled as a simple processing task before the start of the robot movement. In order to make the designed path more effective, the inclusion of corrective post-processing optimization algorithms is advisable.

Approaches to the path planning can be roughly divided based on the objective function that they optimize. The most common objective function is minimization of the path length, which leads to the shortest possible path between two points in space. The overall length of the path is equal to the sum of the lengths of the all sub-parts of the path (Fig.1c), thus the dynamic programming approaches are efficient for solving this kind of problems [1].

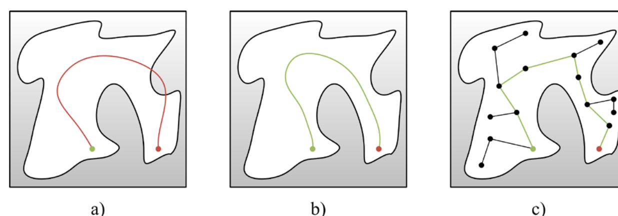


Fig.1: 2D Path planning a) invalid path example, b) valid path example, c) road map with valid path

Path planning is applicable in many sectors such as industrial robotics, autonomy, automation, robotic surgery, automated space exploration, computer graphics, video games artificial intelligence (AI), architectural design or animation. Modern aplications include not only gadgets like autonomous vacuum cleaners, loan mowers or pool cleaners, but also nobler examples of usage like land mine detectors or security robots.

ISBN: 978-960-474-351-3

1.1 Background

History of path planning algorithms is quite recent – first works appeared in this field in the late 1960s, started by Nils J. Nilsson – he introduced to the world the visibility graph method, co-invented the A* heuristic search and have described the mobile robotic system motion planning in its very basics. Path planning and motion planning became subject of research in 1970s, but rapid development of path planning methods started with progress of computers in 1980s. During these decades, number of techniques have been invented. The ultimate survey of algorithms and progress of path planning until the early 1990s brings Jean-Claude Latombe [2], followed by Steven M. LaValle [3] in 2000s.

In the current state of art, the research literature is adressing extensively path planning problem for one or more robots, most of the works are assuming that the environment is completely known before the robotic device begins to traverse. Algorithms presented so far are using the distance transform [4] or heuristic methods [5] in order to find the lowest cost path from the initial to the goal state of the robot (where the cost can be defined as a distance, energy or time spent). The trend of 1990s planners was using the randomization for best dealing with the high dimensionality of configuration space (they are probabilistically complete, if they are able to find solution in a given time). Nowadays, modern methods are aiming the disposal of randomness for the benefits of sampling-based approaches. [6]

The methods for 2D mobile robot path planning generally consist of two sub-tasks: first part, which describes the space with the graph or function, is called pre-processing. It is usually followed by a query phase, which is itself finding more or less optimal way between two points. Stationary robots usually require 3D planning (positioning of the arm). Next paragraph presents a brief classification and short description of the basic varieties of the informative path planning methods.

1.2 Informative Path Planning Algorithms

1.2.1 Grid-based and Decomposition Methods

Planning on the grid involves approximate methods, which are transforming space into the discrete form. The accuracy and computational complexity are depending on the size of the discrete cell of the space model. These methods often result into sub-optimal paths. There is a variety of algorithms that can be applied to the problems of grid-based planning like:

- Classic graph search methods (Breadth-first Search, Depth-first Search, Iterative Deepening Depth-fisrt Search)
- Potential Fields
- Dijkstra's algorithm
- Trapezoidal Decomposition

1.2.2 Roadmap Methods

These methods are creating a map of roads from continuous representation of the environment – a graph representig the free space. Graph edges are thus feasible paths, which robot can move freely along. For finding resulting paths searching graph algorithms are used. Among roadmap methods, we can arrange for example:

- Visibility Graphs
- Voronoi Diagram

1.2.3 Probabilistic Sampling-based planning

The position of the robot in the space is described by its configuration. Configuration space C_{space} is the set of all possible configuration. The sub-set of configurations, that avoids collisions with obstacles, is called the free space C_{free}, which makes the complement of C_{free} in C the obstacle region.

If the robot is a 2D shaped device capable of translation and rotation movements in 2D workspace, C is the Euclidean group:

$$SE(3) = \mathbb{R}^2 \cdot SO(2, \mathbb{R}) \tag{1}$$

$SO(2, \mathbb{R})$ represents the orthogonal group of 2D rotations. Configuration of the robot can be represented using three parameters (x, y, θ). For succesful planning there is a strong requirement that the initial configuration q_{init} and goal configuration q_{goal} must belong to the C_{free}.

In its simplest form, probabilistic planner operates in two stages: first, it generates a graph of path (road map) and then searches the path in this graph. Graph of roads is probabilistic, because it is created from randomly selected configurations from C_{free} (which can be connected to each other only by paths also belonging to C_{free}). This addition of edges is repeated until the goal configuration is reached. It is advisable to try to connect directly to the goal configuration after a certain number of iterations in order to accelerate the convergence to the goal.

Storing only the random samples significantly decreases the amount of primary storage (memory) needed for computation. The problem of probabilistic planners is the large number of short edges – hence, the result path is often complicated and contains lot of possibly redundant moves. It is appropriate to apply further optimization algorithms for smoothing the path. Description of C_{free} as one tree is convenient, but has a consequence that even relatively close configurations may appear like they are far from each other (if we use e.g. probabilistic map routes). This problem solves Rapidly-exploring Random Tree approach, which generates a new tree based on the positions of start and goal configurations q_{init} and q_{goal}. This algorithm will be more discussed further.

2 Used Algorithms

In this section, we first describe the original versions of Rapidly-exploring Random Tree and Dijkstra's algorithm. Then we discuss the need for customization and optimization for RRT planner. Finally, we present a novel way to the grid approach for Dijkstra's algorithm solver.

2.1 Rapidly-exploring Random Tree

"A Rapidly-exploring Random Tree (RRT) is a data structure and algorithm that is designed for efficiently searching non-convex high-dimensional spaces. RRTs are constructed incrementally in a way that quickly reduces the expected distance of a randomly-chosen point to the tree. RRTs are particularly suited for path planning problems that involve obstacles and differential constraints (nonholonomic or kinodynamic). RRTs can be considered as a technique for generating open-loop trajectories for nonlinear systems with state constraints. An RRT can be intuitively considered as a Monte-Carlo way of biasing search into largest Voronoi regions. Some variations can be considered as stochastic fractals. Usually, an RRT alone is insufficient to solve a planning problem. Thus, it can be considered as a component that can be incorporated into the development of a variety of different planning algorithms." [3] RRT was developed by Steven M. LaValle and James Kuffner.

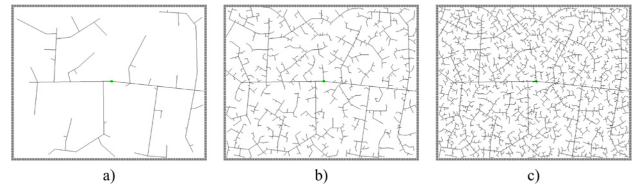


Fig.2: RRT algorithm after a) 100, b) 1000 and c) 3000 iterations

2.1.1 Single-tree Search

RRT with obstacle collision checking needs on its input the dense set of configuration space and the obstacle set. The goal is to get as close as possible to each configuration in free space, starting with initial configuration. Algortihm iteratively connects samples $\alpha(i)$ by new edges of the graph, which are the connection of random sample $\alpha(i)$ and the nearest point q_n on the tree graph G. RRT is a topological graph $G(V,E)$ of vertices V and edges E. The set of points reached by G indicates the set $S \subset C_{free}$.

In the cases where nearest configuration (closest point on tree) is located on the edge of the tree (it is not its node), the edge is split into two pieces and two new nodes are added: q_{near} and $\alpha(i)$, together with the edge between q_{near} and $\alpha(i)$ configuration nodes (Fig.4a).

The STOPPING_CONFIGURATION method returns point q_{stop}, which is constructed by trimming the connection line between random sample $\alpha(i)$ and nearest sample q_{near} accordingly to the obstacle boundary (Fig.4b).

```
RRT_SINGLE(q0)
1. G.init(q0);
2. for i = 1 to k do
3.    qn ← NEAREST(S, α(i));
4.    qs ← STOPPING_CONFIGURATION(qn, α(i));
5.    if qs ≠ qn then
6.       G.add_vertex(qs);
7.       G.add_edge(qn, qs);
```

Fig.3: RRT algorithm considering obstacles (pseudocode)

To be trully effective, the search algorithm is forced after certain number of iterations to use instead of random configuration $\alpha(i)$ directly the goal configuration q_{goal} for connecting new edge.

ISBN: 978-960-474-351-3

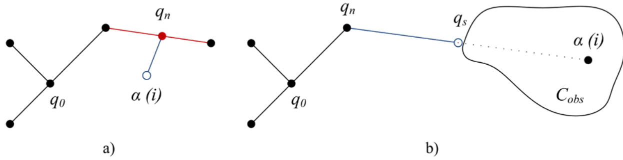


Fig.4: Connection of nearest sample a) splitting of edge in the case that nearest configuration is not a node, b) in the case of the obstacle the random sample is moved to the boundary point by collision detection algorithm

2.1.2 Balanced Bidirectional Search

Much better performance of the algorithm is achieved by using two trees – one exploring from the initial configuration q_{init}, and second that is going from the opposite direction, from the goal configuration q_{goal}. It must be ensured that both of the trees met each other while retaining their rapidly-exploring nature. This is acomplished best by *balancing* both searches.

RRT_BIDIRECTIONAL(q_i, q_g)
1. T_a.init(q_i); T_b.init(q_g);
2. **for** $i = 1$ **to** k **do**
3. $q_n \leftarrow$ NEAREST(S_a, $\alpha(i)$);
4. $q_s \leftarrow$ STOPPING_CONFIGURATION(q_n, $\alpha(i)$);
5. **if** $q_s \neq q_n$ **then**
6. T_a.add_vertex(q_s);
7. T_a.add_edge(q_n, q_s);
8. $q'_n \leftarrow$ NEAREST(S_b, q_s);
9. $q'_s \leftarrow$ STOPPING_CONFIGURATION(q'_n, q_s);
10. **if** $q'_s \neq q'_n$ **then**
11. T_b.add_vertex(q'_n);
12. T_b.add_edge(q'_n, q'_s);
13. **if** $q'_s = q_s$ **then return** SOLUTION;
14. if $|T_b| > |T_a|$ then SWAP(T_a, T_b);
15. **return** FAILURE;

Fig.5: Balanced bidirectional RRT search (pseudocode)

The graph G is decomposed into two trees T_a and T_b (one starting from q_{init}, another from q_{goal}. After given number of iterations, both trees are mutually exchanged. Therefore, the T_a may not always be the one that started from q_i configuration.

In every iteration, T_a grows as G in the example in Fig.3 pseudocode. If the new vertex q_s is added to T_a, the requirement to extend tree T_b is called. Rather than a new random configuration $\alpha(i)$ the same new vertex q_s which was already added to the T_a is used. This ensures that both trees are growing towards each other. Balancing is represented at the line 14 in Fig.5 – a new sample is always for a smaller tree (with fewer edges or the one with shortest length of all segments).

Usually, RRT algorithm is not powerful enough to find an optimal solution. It is very often included as a sub-component of path planning algorithms designed for specific tasks and environments.

It is very suitable for applying further algorithms for smoothing the result trajectory or other enhancements in order to improve the convergence of the solver to the goal configuration (e.g. generate random biased samples while taking into account position of the goal or another meaningful heuristics).

2.1 Dijkstra's Algorithm

Dijkstra's algorithm was published by Dutch computer scientist Edgser Wybe Dijkstra in 1959 [7]. It is a graph search algorithm that solves the single-source shortest path problem for a graph with positive edge path costs – it produces a shortest path tree. For a given initial node in the graph, the algorithm finds the path with lowest cost (shortest path) from that node to every other node. It can be used for finding shortest path from one node to another by stopping the algorithm once the shortest path to the destination node has been determined.
The original variant of Dijkstra's algorithm does not use a priority queue and runs in $O(|N|^2)$ time [8], usually implemented as the adjacency matrix. The implementation based on adjacency lists or Fibonacci heap with priority queue runs in $O(|E| + |N|\log|N|)$ time, which is often denominated by $O(|E|\log|N|)$, assuming the graph is fully connected [9]. It is asymptotically the fastest single-source shortest-path algorithm for graphs with unbounded non-negative costs.

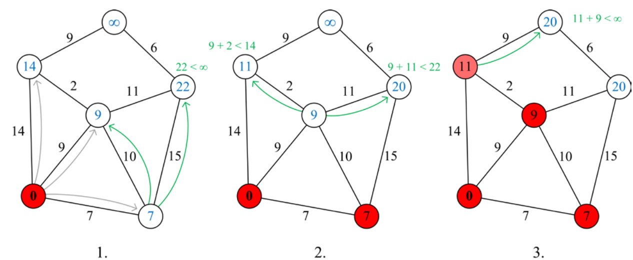


Fig.6: Dijkstra's algorithm – example of the graph evaluation

In the beginning, a distance value is set for each node: zero for the initial node q_{init} and infinity for all the others. All nodes are marked as unvisited and initial node set as a current. Then, a set of the unvisited nodes called the unvisited set is created,

ISBN: 978-960-474-351-3

containing all of the nodes except the initial one. For the current node, tentative distances to all its unvisited neigbors are calculated. If this distance is less than previously recorded tentative distance in node, it is overwritten. When considering all of the neighbors of the current node is done, current node is marked as visited and removed from the unvisited set – it will not be checked any more. If the destination node has been marked visited (path planning between two specific nodes), then stop. Generally, stop if the smallest tentative distance among the nodes in the unvisited set is infinity (path planning for the complete traversal). In the case that solution is not finished yet, the current node becames the node marked with smallest tentative distance among the nodes in unvisited set and algorithm goes to the next iteration. Procedure is shown in Fig.6.

```
DIJKSTRA(G, n_start, n_goal)
1.  foreach node n in G
2.      n.dist = infinity;
3.      n.previous = undefined;
4.  n_start.dist = 0;
5.  Q = set of all nodes in G;
6.  while Q is not empty
7.      u = node from Q with smallest distance;
8.      remove u from Q;
9.      if u = n_goal then
10.         break;
11.     foreach neigbor v of u
12.         alt = u.dist + distanceBetween(u, v);
13.         if alt < v.dist then
14.             v.dist = alt;
15.             v.previous = u;
```

Fig.7: Dijkstra's algorithm for path planning (pseudocode)

When the smallest tentative distance in node-to is overwritten, also the pointer to the node-from is stored in the node-to. Thus we can read the shortest path from goal to initial node by reverse iteration (while *u*.previous is defined *u* = *u*.previous).

3 Problem Approach

3.1 Informative Path Planning

In this article, informative path planning of the nonholonomic robot is formulated as a discrete problem. First, there is a graph G representing the mobility of the robot. Each node $v \in V$ is represents a waypoint, which can be visited by the robot. Each edge $e \in E$ stands for a movement of a robot between two waypoints – vertices. There is a cost $C(e)$ associated to every edge representing the difficulty of the robot traversal along the edge. Edge lengths are representing the distance, which in our case equals to the edge cost (in general cases, depending on the graph construction, lengths/costs can represent traversal times, amount of energy needed to travel between nodes etc.).

A robot path $P = [e_0, e_1, \dots, e_k]$ consist of a sequence of edges needed for traversal, total length of the path is a sum costs of all the edges:

$$\|P\| = \sum_{i \in P} C(e) \qquad (2)$$

It is clearer to represent a path as a sequence of edges to traverse rather than a sequence of nodes to visit, because there might be multiple edges suitable for the traversal between two nodes. For solving the general informative path planning problem, we need to find a path P^*such that:

$$P^* = \arg\min_P f(P) \qquad (3)$$

It is obvious, that $\|P\| < B$ where B is the maximum allowed path length. Objective function $f(P)$ captures the cost of the samples of the robot that will be taken as the robot traverses the path.

3.2 Implementation of the Algorithms

The following chapter is devoted to the description of the implementation and functionality of algorithm solvers measured and compared in this article.

3.2.1 RRT Single and Bidirectional

RRT Single: is a single-searching RRT with collision detection routines (Fig.8), which is described in chapter 2.1.1 Single-tree Search. The data structure is a hierarchical tree with a set of linked edges. It is defined recursively as a collection of edges (starting at a root edge), where each edge is a data structure with a pointer to the parent edge, with the constraints that no edge is duplicated. The tree structure is a connected graph without cycles in which any two vertices are connected by exactly one simple path, but it is not full equivalent to the trees in graph theory – it is more a *rooted tree* with additional ordering of branches. The goal path is read from the goal edge to the source edge by periodically asking current edge for the parent edge (while *current*.parent is defined *current* = *current*.parent) – basically a reverse iteration process.

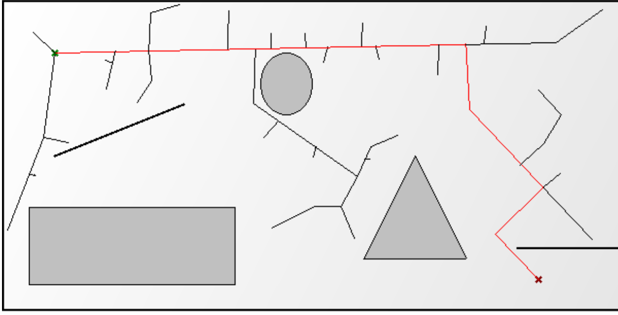


Fig.8: RRT Single on an open-space type map

RRT Bidirectional: represents balanced bidirectional search as it is described in chapter 2.1.2 Balanced Bidirectional Search and shown in Fig.9. Two single-search RRTs are periodically switched when one of them has more edges – new connection is always made on a smaller tree. Each time when one of the trees is extended, the random configuration tries to connect also to the other tree, which forces RRT Bidirectional to connect and find the solution faster than the previous single searching algorithm RRT Single. The goal path is obtained in the same way as in the single-searching tree, but final path edges from one tree need to be re-parented.

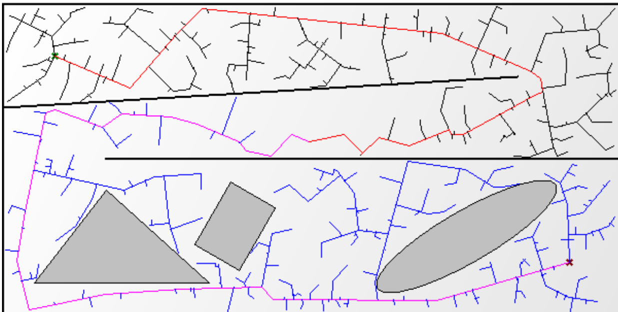


Fig.9: RRT Bidirectional on a maze-type map

3.2.2 Dijkstra Grid

As the representative of the Dijkstra's algorithm we present an unique novel implementation of the path planning solver: *Dijkstra's Grid* (Fig.10 and Fig.11). The algorithm consist of two steps: first, the grid is constructed in the explored space, removing all the edges and nodes colliding with obstacles. In the second step, the grid is evaluated by the Dijkstra's algorithm going from start node. When the goal node is reached, shortest path is obtained going from goal node to initial node in reverse loop. Grid size and inclusion of diagonals is optional.

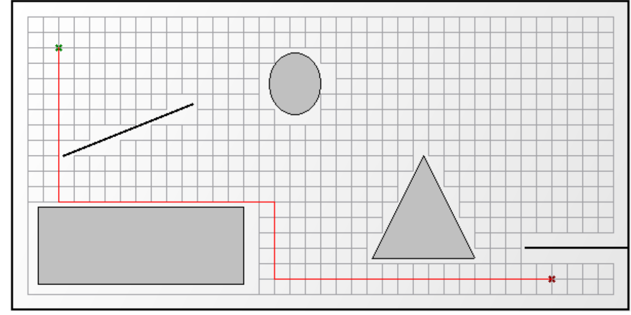


Fig.10: Dijkstra Grid (10 px size) on a open-space type map

The graph structure of the grid is a simple undirected graph weighted by length of the edges. It is defined as a adjacency list data structure – collection of unordered lists, one for each node in the graph. Each list desribes the set of neigbors of its node. The overall complexity of the prompting for edges adjacent to the current vertex is reduced to $O(|N| + |E|)$ in comparison with adjacency matrix, where drawing out such a information from the data structure results in overall complexity $O(N^2)$.

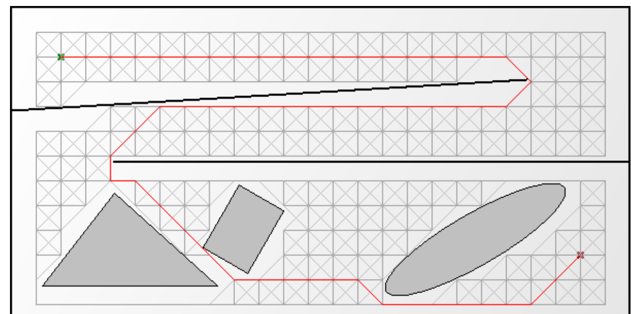


Fig.11: Dijkstra Grid (25 px with diagonals) on a maze-type map

4 Comparisons and conclusion

In this section, we present a performance comparison of Rapidly-exploring Random tree and Dijkstra's algorithm on two types of the map of the environment:

- *open-space type map* (Fig.8 and Fig.10)
- *maze-type map* (Fig.9 and Fig.11)

Based on the measurements of the performance comparison, following results arised:

- Dijkstra Grid finds solution on maze-type maps in significantly better total times, where both RRT Single and RRT Bidirectional are more or less busy with exploring the space while the grid is constructed in exact time.

- RRTs perform better on open-space type maps, where the construction of the grid over the whole map can be redundant – RRT connects to the goal node in few iterations.
- RRT Bidirectional is always better than a single-search RRT in the time spent to finding goal, while the length of the result path does not vary much.
- RRTs need to be optimized – there are substantially different results between the instances of the algorithm (which is obvious from the huge amount of flier points in boxplots). In a nutshell, 100 different instances of RRT results in a 100 different result paths.
- Dijkstra grid is exact, for the 100 instances of the algorithm there is always the same result path (only the total times vary because of the PC computational overhead. But, there is a need to find the optimal size of the grid cell.
- Dijkstra Grid usually ends up with a shorter path with fewer edges than RRTs (even without post-optimization of the path).
- It is necessary to polish the path given by RRT and decrease the number of result edges.

4.1 Result Measurements

Following figures displays statistical results from measurements on given maps – graphs with various parameters and box plots with whisker plots extending from lower to upper quartile values of the data, with a line at the median. The whiskers extend from each box to show the range of the data. Flier points are those past the end of the whiskers (outliers).

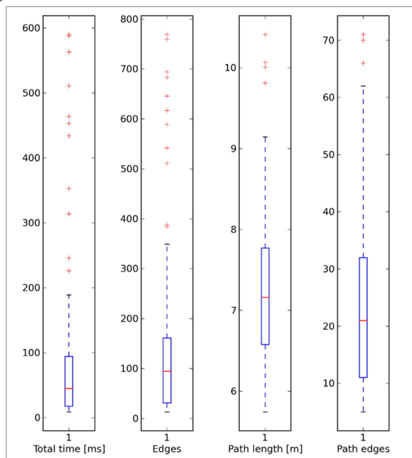


Fig.12: RRT Single – Open-space Type Map

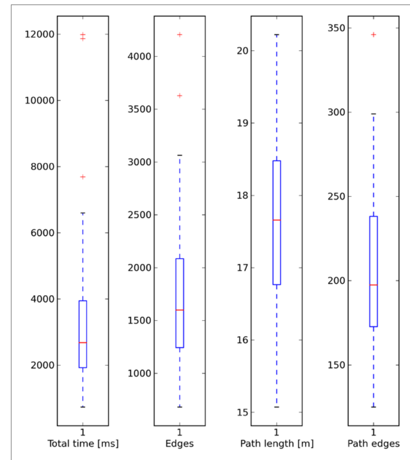


Fig.13: RRT Single – Maze-type Map

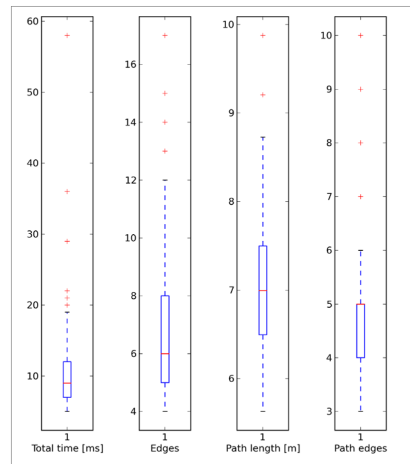


Fig.14: RRT Bidirectional – Open-space Type Map

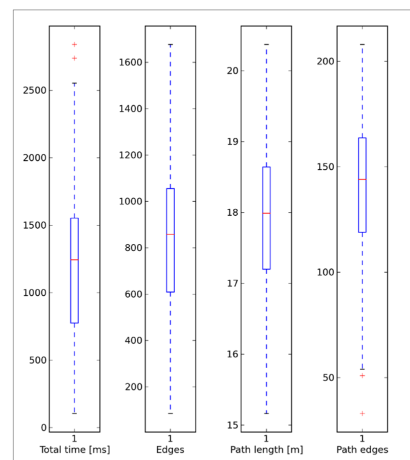
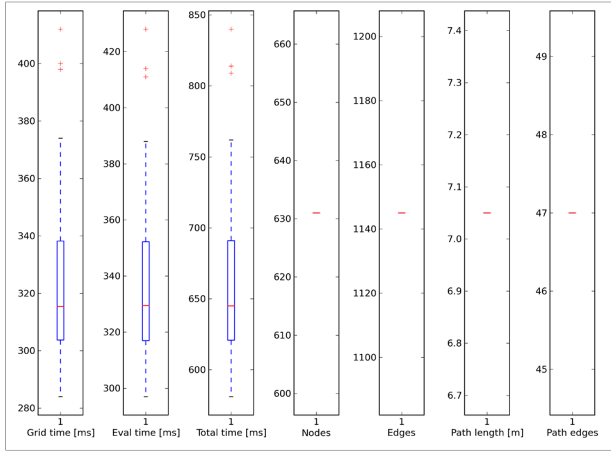


Fig.15: RRT Bidirectional – Maze-type Map

10 px No Diagonals:



10 px Including Diagonals:

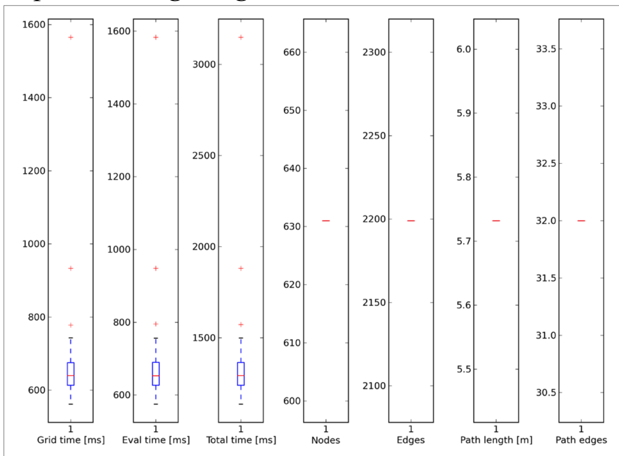
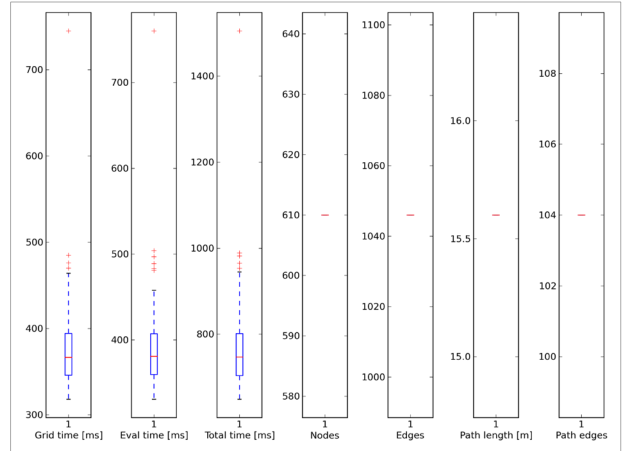


Fig.16: Dijkstra Grid – Open-space Type Map

10 px No Diagonals:



10 px Including Diagonals:

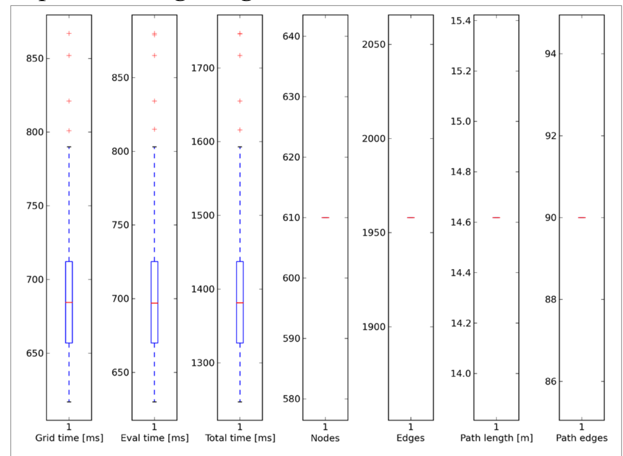


Fig.17: Dijkstra Grid – Maze-type Map

4.2 Comparison

In this section, open-space type map and maze-type measurements are compared for all algorithms. There is one more comparison of the Dijkstra Grid – different grid sizes on a maze-type map.

Table 1: Open-space Type Map

	RRT Single	RRT Bidirectional	Dijkstra Grid	Dijkstra Grid Diag.
Time [ms]	92.84 ± 131.99	10.84 ± 6.99	657.22 ± 51.15	1322.97 ± 211.42
Edges	147.78 ± 179.16	7.22 ± 2.83	1145.00 ± 0.00	2199.0 ± 0.00
Path length [m]	7.28 ± 0.96	7.03 ± 0.79	7.05 ± 0.00	5.73 ± 0.00
Path edges	23.75 ± 15.73	4.86 ± 1.41	47.00 ± 0.00	32.00 ± 0.00

Table 2: Maze-type Map

	RRT Single	RRT Bidirectional	Dijkstra Grid	Dijkstra Grid Diag.
Time [ms]	3084,88 ± 1866.35	1229.71 ± 588.01	771.04 ± 106.66	1399.13 ± 103.66
Edges	1703.12 ± 620.95	841.84 ± 329.19	1046.00 ± 0.00	1958.00 ± 0.00
Path length [m]	17.62 ± 1.06	17.87 ± 1.06	15.60 ± 0.00	14.62 ± 0.00
Path edges	205.11 ± 43.13	140.33 ± 34.38	104.00 ± 0.00	90.00 ± 0.00

Table 3: Various Sizes of Dijkstra Grid on Maze-type Map

	DG Diagonal 8 px	DG Diagonal 16 px	DG Diagonal 24 px
Time [ms]	7129.80 ± 513.07	1182.04 ± 57.25	495.52 ± 30.33
Edges	8791.00 ± 0.00	1899.00 ± 0.00	713.00 ± 0.00
Path length [m]	14.31 ± 0.00	14.66 ± 0.00	14.43 ± 0.00
Path edges	166.00 ± 0.00	85.00 ± 0.00	56.00 ± 0.00

Acknowledgement

This paper was supported by the IGA VUT Brno, FSI No.: FSI-J-13-2136 and FSI-S-11-31 project.

Appendix: Note on Implementation

Measurements and algorithms presented by this article are based on the unique custom application RRTExplorer 2.0 [10] written in *C++* programming language in essentially multiplatform framework *Qt*. Measurements and post-processing routines are written in *Python* scripting language, where the graph plotting is provided by the matplotlib library with statistical computations in *SciPy* and *NumPy* libraries.

References:

[1] J. Klapka, P. Popela and J. Dvořák, *Metody operačního výzkumu* (in Czech), Brno: VUTIUM, 2001.

[2] J.-C. Latombe, *Robot motion planning*, Boston: Kluwer Academic Publishers, 1991.

[3] S. M. LaValle, *Planning algorithms*, New York: Cambridge University Press, 2006.

[4] A. Zelinsky, R. A. Jarvis, J. C. Byrne and S. Yuta, "Planning paths of complete coverage of an unstructured environment by a mobile robot," in *Proceedings of International Conference on Advanced Robotics*, 1993.

[5] N. J. Nillson, *Principles of artificial intelligence*, Berlin, New York: Springer-Verlag, 1982.

[6] S. M. LaValle and M. Branicky, "On the Relationship Between Classical Grid Search and Probabilistic Roadmaps," *The International Journal of Robotics Research,* vol. 23, 2003.

[7] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik,* vol. 1, no. 1, pp. 269-271, 1959.

[8] M. Leyzorek, R. S. Gray, A. A. Johnson, W. C. Ladew, S. R. Meaker Jr., R. M. Petry and R. N. Seitz, "Investigation of Model Techniques — First Annual Report — 6 June 1956 — 1 July 1957 — *A Study of Model Techniques for Communication Systems*," Case Institute of Technology, Cleveland, Ohio, 1957.

[9] M. L. Fredman and R. E. Tarjan, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM,* vol. 34, no. 3, pp. 596-615, 1987

[10] L. Knispel, "RRT Explorer," 11 October 2011. [Online], [Accessed October 2013], Available: http://sites.google.com/site/rrtexplorer/.

ISBN: 978-960-474-351-3