

Fast Geometric Path Planning for the Smallsize Robocup League

Lina Ourima
lwolf@inf.fu-berlin.de
Freie Universität Berlin
Studienarbeit betreut von Prof. Rojas

4.6.04

Contents

1	Introduction	1
2	Related Work	3
2.1	Grid Based A* Path Planning	4
3	Fast Geometric Path Planning	4
3.1	The Visibility Graph	5
3.2	Avoiding Computation of the Whole Visibility Graph	6
3.3	Dealing with intersections	7
4	Geometrical details	7
4.1	Intersections obstacle - line	7
4.2	Position of a Point Relative to a Polygon	8
4.3	Tangents between a point and several intersecting Obstacles .	10
4.4	Tangents between two obstacles	10
4.5	Tangents from one group of obstacles to another group	11
5	Improving the Algorithm	11
5.1	Avoiding unnecessary additional part-paths	11
5.2	Avoiding multiple computation of the same part-paths	12
5.3	Not computing the whole path	12
6	Smoother Paths	13
6.1	Characteristics of Bezier-Curves	13
6.2	Use of Bezier Curves in the Program	15
7	Results	15
8	Future Work	16

Abstract

Presented is an algorithm for planning the motion of a robot moving on a rectangular field with obstacles. The positions of every obstacle and the robot are known. Searched is the first intermediate point on a path leading to a given endpoint. The algorithm takes a geometric approach where every point of the path is the corner of a polygonal obstacle or a tangent on a circular obstacle. The algorithm was implemented in the case of the FU-Fighter's soccer-robots.

1 Introduction

In a robot soccer game in the smallsize league each team has up to 5 autonomous mobile robots max. 18 cm in diameter and 15 cm high. The main sensor is one or several cameras above the field that catch an image of the whole field (see [8], chapter 2). The Fu-Fighters currently use two cameras each viewing half the field to get a better solution (see [9]).

The field is rectangular and after the official rules for 2004 4000 X 4900 mm large. The size of the field can change from year to year. On the field there are several things that should be viewed as obstacles in a robots path-planning: the goals, opponent robots, team-mates, the space between ball and opponent goal when a team-mate tries to shoot a goal, All obstacles are circles or convex polygons. Most obstacles are moving. The robots can be 2 m/s fast, The ball even faster. The cameras work with 30-60 Hertz. A circle in the robots behaviour has to be completed until the new picture comes from the camera. For this cycle the old path planer became to slow when the speed of the camera was increased from 30 to 60 Hertz and the size of the field almost doubled. The running time of the old path planer depends on the size of the field.

When a robot moves about the field in a robot-soccer game and there are obstacles in his path a decision has to be made in real-time which path around the obstacle is the best. Since most obstacles are moving robots (team-mates and opponents) the situation of the obstacles changes while the robot is on its way. Therefore it is not possible to compute the whole path once and then for the robot to go that way. It can't be desired that a robot waits for a few seconds while he computes his path either. Therefore the algorithm should return the next intermediate point for the robot to go to as fast as possible. When the robot reaches this point the algorithm has to be called again using the fresh data of the changed obstacle situation.

Usually the data about the positions of the obstacles and the robot himself is not very accurate. And since all robots move, a possible path might open or close until the robot gets there. Therefore an algorithm is desired which will return the next intermediate point as fast as possible accepting wrong or sub optimal solutions from time to time.

In general Geometric Path Planning all obstacles are represented by polygons. The nodes of the graph are the points of the polygons. Between two nodes there is an edge, iff the edge between the two corresponding points does not intersect with any obstacle. The weight of each edge is its Euclid-

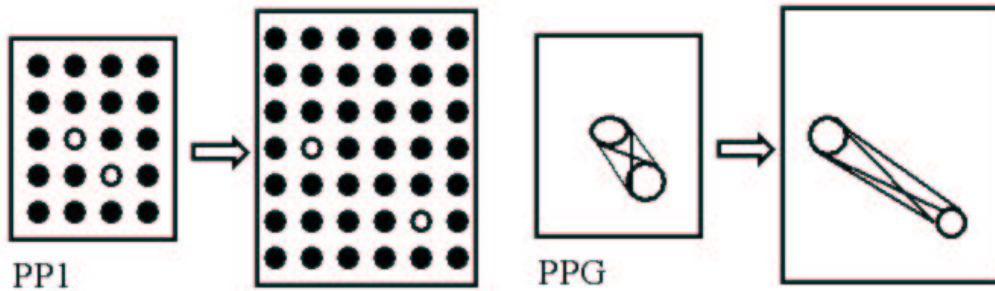


Figure 1: When the size of the field increases the running time of PP1 grows linear while the running time of PPG stays constant as long as the number of obstacles doesn't change.

ean length. This graph is called visibility graph. This graph is searched with an A* algorithm and finds the shortest path.

An advantage of the presented Fast Geometric Path Planning algorithm (PPG) is that the increasing of the size of the field has no effect on the calculation time or accuracy. In the old path planner (PP1) either the number of grid points has to be increased or there is a loss of accuracy as the grid points are farther away from each other when the size of the field is increased. In PPG the length of a path has no influence on the calculation time, but only the number of obstacles in the possible ways (in more advanced implementations then visibility graph + Dijkstra). With some adaptations it is possible to simulate a behaviour of the path that is much closer to nature than the Euclidean shortest path. The disadvantage of PPG is that weighted regions are complicated to implement and since they increase the number of points in the visibility graph, calculating it becomes much slower. The weight of an edge then is multiplied by the weight of the area it crosses. Weighted areas where not implemented.

A robot moving is not able to stop immediatly. In order to make a corner in his path he would have to stop and turn which costs much time. Therefore the shortest Euclidean path is in many cases not the fastest. Also the robot is not able to exactly stay on that path. An obstacle might be on the robots path in reality even though it was not in the straight line planning. For example see Figure 2.

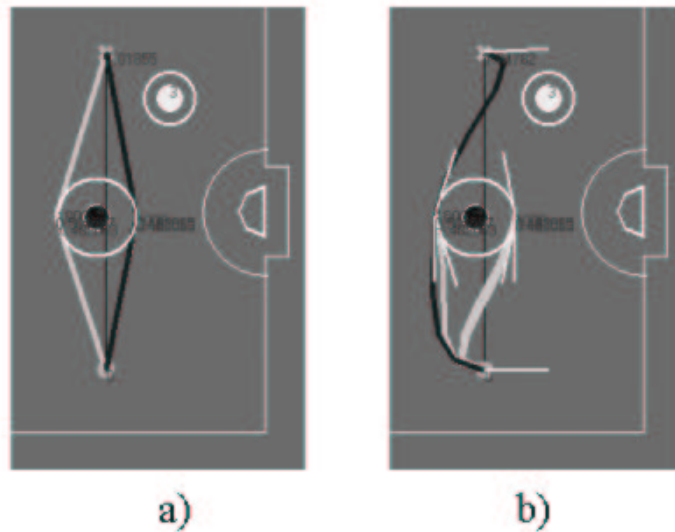


Figure 2: When the path is planed with straight lines robot 3 does not lie on the path. When the robots pace is taken into account(b) it does get in the way.

2 Related Work

In a soccer game we have 5 robots about which we could know the paths that they planed etc. Now the question occurs wouldn't it be helpful to make multidimensional path planning? That means we would plan the paths for all robots at once taking the plans of the other ones into account or even decide which robot should go where (since all robots are build the same way)? In the existing literature there are several papers on multidimensional path planning. None of the cases these papers talk about can be used for smallsize robot soccer.

C. Clark, S. Rock and J. Latombe present an algorithm that is for "large groups of robots" in an "unknown environment with moving obstacles". The robots have no "global knowledge" of their world. In opposite in Robocup there is a camera installed over the field which provides global knowledge of the world. Also the group of robots in Robocup is not "large" usually there are about 5 robots. [1]

D. Parson's and J.Canny's algorithm assumes a world of fixed obstacles in which paths for the robots can be computed multiple times, accepting a

longer pre-processing time for the environment. In Robocup there are hardly any fixed obstacles. When the next path has to be computed opposite robots, the ball etc. will very likely have moved. Also the algorithm is slow while it guarantees that each possible way is found. In Robocup wrong answers can be accepted from time to time as long as the program runs fast. [2]

B. Brummitt and A. Stenz solve the problem of Multiple Travelling Salesman where n similar robots have to visit m goals and return to their "recharging area" from time to time. The algorithm does "dynamic replanning as world knowledge increases". In Robocup the world is known all the time from the camera over the field. And the questions of Multiple Travelling Salesman never occurs. [3]

Bennewitz and Burgard apply a grid to the field, searching an optimal path for each robot by the A* algorithm. Then it resolves conflicts between the moving robots. It assumes, that there is not much space to move and that a possible solution without deadlocks is hard to find. But in Robocup almost all obstacles are moving. So if no correct path is found in one moment in the next moment the obstacles will be different and a path be found. And there is no path planning done for all robots at once usually. A robot gives his way to the planner only if there are obstacles in his way. To synchronize all robots would cost precious time and is not really necessary since a deadlock is very unlikely. [5]

In order to use the algorithm of Bennewitz and Burgard a path planning for each robot is to be done (often several times) and then combined. But even the existing path planner, using A* and a grid is way to slow.

The problem of multidimensional path-planning where several robots move from start to end positions avoiding obstacles and each other is "known to be PSPACE-hard" in the number of robots according to Parson and Canny ([2]). Even though the number of robots is quite low and fixed these algorithms are still much slower then one-dimensional path planning for each robot.

Therefore the goal of this work was set on a faster one-dimensional path planning giving better results.

2.1 Grid Based A* Path Planning

The existing path planner(PP1) in the FU-Fighters divides the field into a grid. Every point of the grid gets a certain weight. when a grid point lies inside an obstacle it will have the weight ∞ , a grid point in open space a low weight like 1. It is possible to define "you should only go here if you have to"

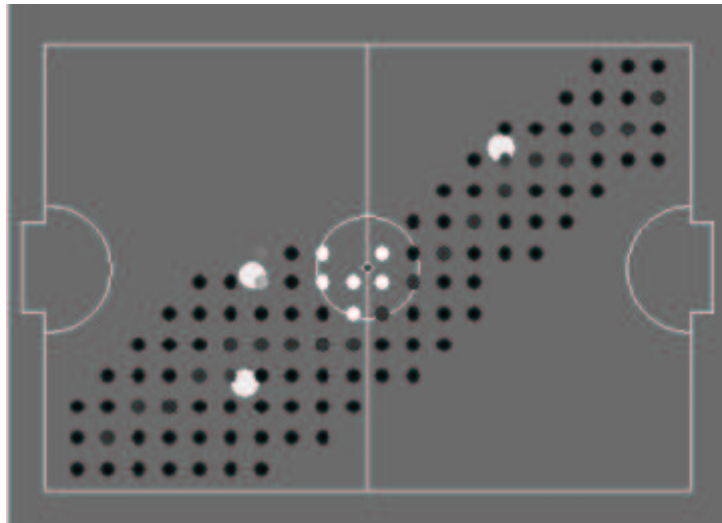


Figure 3: Sample output of PP1, dark grey is the planned path, light grey - white are obstacles, black are grid points with no additional weight. The larger white circles are robots.

areas by applying a different weight to the grid points in these areas. From every grid point one can travel north, east, west and south. The shortest way from point s to point t could be found by an algorithm like Dijkstra but the A^* algorithm is much faster.

3 Fast Geometric Path Planning

3.1 The Visibility Graph

A visibility graph is a graph with several polygons. The nodes of the graph are the points of the polygons. There is an edge between two nodes if one point is visible from the other point, which is equal to the line connecting the two points does not cut another polygon.

A complete visibility graph can be computed in $O(n^2 * k) = O(n^3)$ brute force, n being the number of points and k the number of polygons. For every pair of points P_i, P_j $i \neq j$ it has to be tested whether or not the edge $\overline{P_i P_j}$ cuts any of the obstacles. Since the obstacles have a maximum number of points k there are n/k obstacles. For each obstacle the test if a

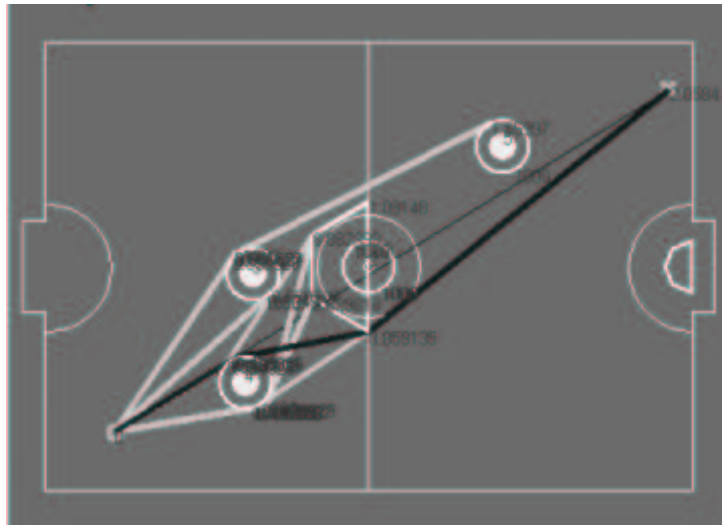


Figure 4: Sample output of PPG, thin white circles and polygons are obstacles. The bold black line is the path planned and the bold white lines show which path where tried while the planning. The thin black line is the direct path from S to T.

given edge cuts it or not would take brute force $O(k)$, testing for each edge of the obstacle whether it cuts the edge. Therefore the computation of the complete visibility graph takes $O(n * (n - 1) * n/k * k) = O(n^3)$. In [4] H. Alt and E. Welzl improve the time to $O(n^2 * \log n)$ with a bad constant and no existing implementation. Therefore it has to be avoided to compute a complete visibility graph. Furthermore the visibility graph is can't be computed like that when some of the obstacles are circles. A circle would have to be approximated with a polygon. Many computations are much slower using polygons instead of circles(see Section 4)

Prof. Alt explained to me, that no concave corners can be part of a shortest path.

When a path contains a point of a concave corner of a polygon there exists a shorter path that doesn't use this corner(see Fig. 5). This path can be found by moving the point where the two part paths meet away from the obstacle. Therefore concave corners can be ignored when searching the shortest path. All obstacles in our environment are concave, but when two obstacles intersect concave corners develop.

In the next explanations the visibility of an edge will be important. An edge

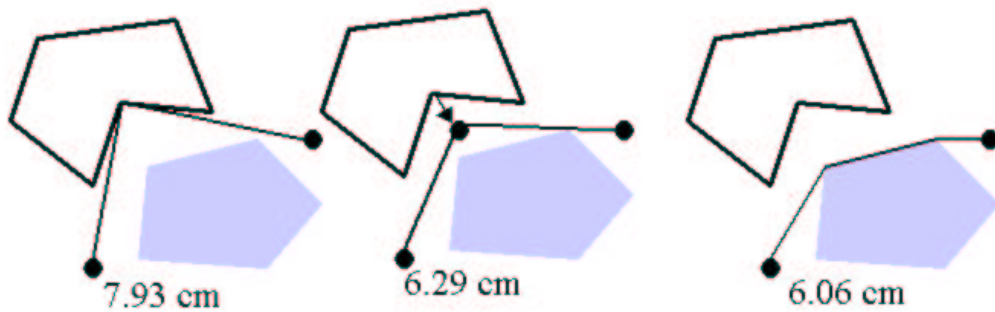


Figure 5: The first picture above, left uses a concave corner. the second picture above, right is already better. The third picture shows the optimal solution.

of a polygon is visible from a point P iff the triangle between this edge and P doesn't cut the polygon. This visibility should not be confused with the visibility in the visibility graph where one point is visible from a point if the line between these points cuts no obstacle.

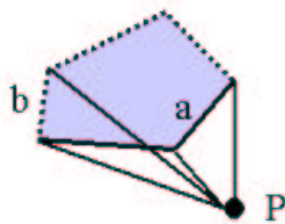


Figure 6: Edge a of the polygon is visible from point P , b is not visible.

Every corner can be part of a shortest path only when its one edge is visible while the other one is not from the point of which it is reached (see Fig. 7). Therefore the only edges need to be in the visibility graph that go from a point to an obstacle that are tangents after the definition of 4.2. With circles this means its always sufficient to compute paths on the tangential points.

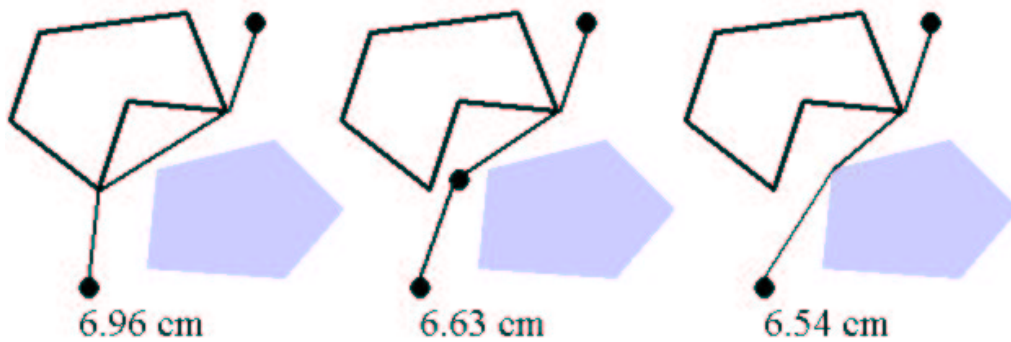


Figure 7: The first picture above, left, shows a path using a corner although both its edges are visible. The second is a better solution, the third picture is optimal.

3.2 Avoiding Computation of the Whole Visibility Graph

In the plane the direct way from one point to another point is always the direct edge between the two if it exists. This can be proved by the triangle inequality. Therefore if the edge between S and T exists it is not necessary to compute any more edges.

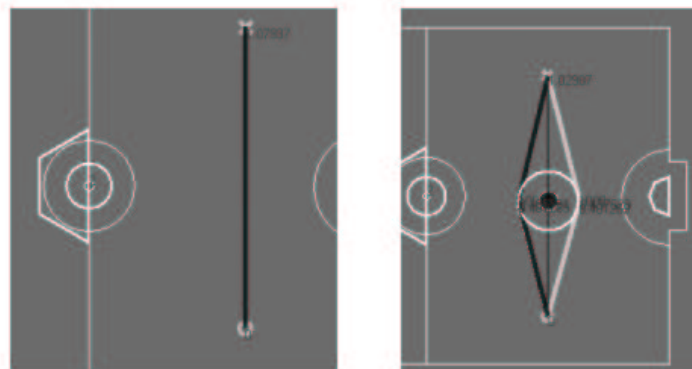


Figure 8: The shortest path between from S (point below) to T (cross above) without (left) and with obstacle, here an opponent robot

If \overline{ST} cuts obstacles the shortest path contains either one of the two tan-

gential points of the obstacle closest to t that can be reached without further intersection or one of the two series of Obstacles that are cut by the edge between s and the two tangential points on the obstacle closest to s . Once an obstacle O is reached a path has to be found from it to T . The point from which to start to T is one of the two tangential points between T and O .

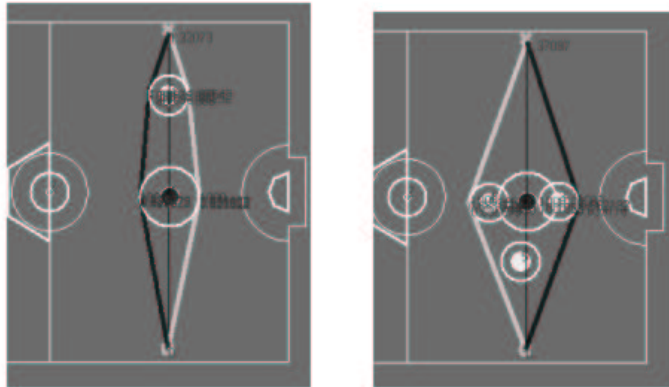


Figure 9: The shortest path between from S (point below) to T (cross above) a way $S \rightarrow$ obstacle $\rightarrow T$ (left) and $S \rightarrow$ obstacle1 \rightarrow obstacle2 $\rightarrow T$

The way from one point on the obstacle to another on it is made on the edges of the polygon. If the edge from the tangential point on O to T intersects another obstacle O_2 then a path has to be found from O to O_2 and from O_2 to T . In order to find the path from O to O_2 we need the Tangents of two Obstacles.

3.3 Dealing with intersections

Intersecting obstacles proved to be a big problem and they happen quite often: Every Obstacle has to have an additional radius of $1/2$ of the planning robots radius as safety distance at least, because the paths are always planed from the robots middle assuming he is a point. Opposite robots get a even bigger safety radius. That results in many intersecting obstacles. When two robots are closer then one robots diameter their corresponding obstacles already intersect.

To solve this problem a group of intersecting obstacles is viewed as a single

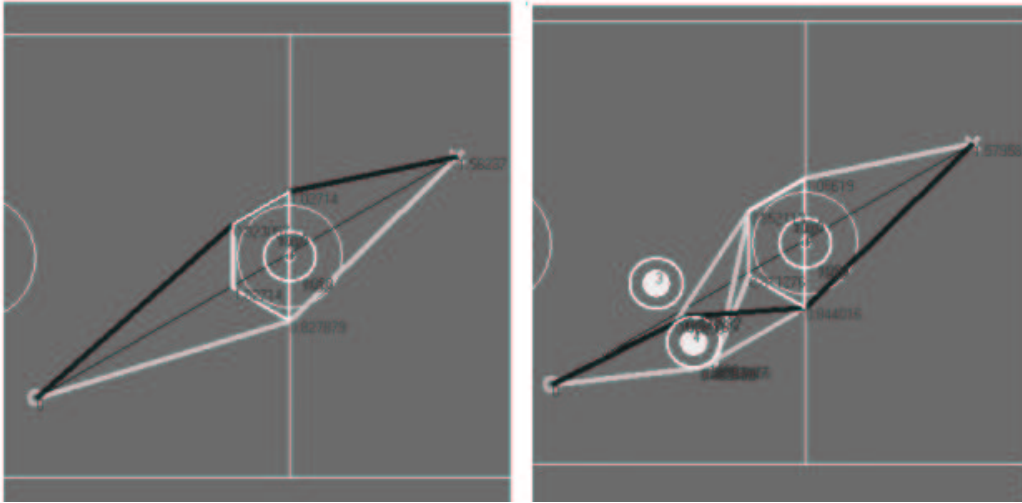


Figure 10: In the left picture a direct path to the obstacle is found. Because no direct path to the tangents of the polygonal obstacle exists in the right picture the path has to go to one of the circular obstacles first

obstacle in the process of planning. In most cases the group of intersecting obstacles is viewed as if only a bigger obstacle, the convex hull of the points of the intersecting obstacles would exist. Some problems arise: The convex hull would have infinite points when some of the obstacles are circles. The bigger obstacle is not convex but concave, the goal could lie inside the convex hull even though it does not lie inside an obstacle.

Therefore the obstacles are treated as groups. When a path to an obstacle group is computed the tangents from the starting point to the group of obstacles are computed (see 4.3).

4 Geometrical details

4.1 Intersections obstacle - line

A line intersects a circle iff the distance between the middle point of the circle and the line is less than or equal to the radius. The intersection of a polygonal obstacle and a line is not so easy however since it's necessary to compute this very fast. For each polygonal obstacle a bounding box is defined. If both the lines start and endpoint lie on one side of the box then the line can't cut

the obstacle. Otherwise additional computation would be needed. In the implementation these computations used up so much time and the results were quite seldom different from the given results that only the bounding box is used.

When the line is infinitive the polygon is cut when a pair of neighbour points

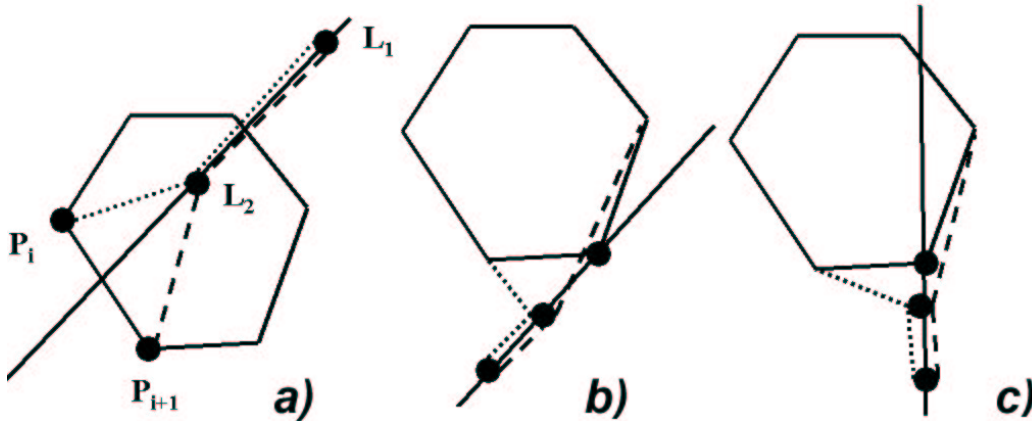


Figure 11: a) P_i lies "left" of the line while P_{i+1} lies "right" therefore the angle $L_1L_2P_i$ turns left while $L_1L_2P_{i+1}$ turns right. b) The line doesn't cut the polygon in the Point but only touches it. c) The line cuts the obstacle in the point.

P_i, P_{i+1} of the corners of the polygon can be found that lie on opposite sides of the line. Be L_1 and L_2 any two points on the line and $L_1 \neq L_2$. When the angle $L_1L_2P_i$ turns left while the angle $L_1L_2P_{i+1}$ turns right P_i and P_{i+1} lie on different sides of the infinite line. Therefore the line cuts the edge connecting P_i and P_{i+1} and when the line cuts any edge of the polygon the line cuts the polygon.

Whether an angle makes a left or right turn can be computed in several ways. The easiest way to find out whether $P_1P_2P_3$ is to compute the cross product $(P_2 - P_0) \times (P_1 - P_0)$ When the result is positive the angle makes a left turn anticlockwise, when its negative a right turn and when 0 the points lie on a straight line.

When the line cuts the obstacle in the point P_i then $L_1L_2P_i$ is straight (result of the cross product = 0) but $L_1L_2P_{i-1}$ has to turn right and $L_1L_2P_{i+1}$ left or the other way round. Therefore when a Point P_i is found to lie on the line then its neighbours P_{i-1} and P_{i+1} have to be checked if they lie on different

sides of the line. where point 1 on the line l_1 , point 2 on the line l_2p_i turns left while $l_1l_2p_{i+1}$ turns right or the other way round. In this case at least one point of the polygon lies on either side of the line. Of course in case of infinite lines many cases can be ruled out by the bounding box.

4.2 Position of a Point Relative to a Polygon

Again for a circular obstacle this question is very trivial to answer: iff the distance between the point and the obstacle is less then or equal the radius the point lies inside. In polygonal obstacles the point lies outside when it lies outside the bounding box. If the point q lies inside the bounding box we test for each point p_i of the obstacle if $p_i p_{i+1} q$ turns left (this can be done by the cross product see [6]). If $p_i p_{i+1} q$ turns right at least once the point lies outside otherwise inside.

Tangents Between a Point and a Single Obstacle

How to compute the tangents from a point to a circle should be clear. The tangents from a point P to a polygon Y (as I am talking about tangents here I'm always interested in their points on the Obstacle) are the points whose one edge is visible while the other one is not and the line carried by P and this point may not intersect the polygon. These points can be found by sorting the points of the polygon by polar coordinates around the P . as long as P doesn't lie in the convex hull of Y the solution is correct (otherwise the tangents would not be defined).

Let the points of the polygon be $P_0 \dots P_{n-1}$ and the index computed modulo n if its to large or small. The indices are incremented counter clockwise. The smallest polar angle is the point P_i such that for all points $P_j, j \neq i$ the path $S P_i P_j$ turns left. The polar maximum angles is defined vice versa.

1) Every point found is a tangential point.

Without loss of generality the point P_i is the minimal polar coordinate. Now let us assume P_i is not a tangential point. If the point P_i was not tangential either both $\overline{P_i P_{i+1}}$ and $\overline{P_i P_{i-1}}$ are visible or none of them. Now we will see that both cases lead to a contradiction.

a) Both are invisible. Specially $\overline{P_i P_{i-1}}$ is invisible. This means the point P_{i-1} has to be right of the line $\overline{S P_i}$ (see Fig. 12, b). Now $S P_i P_{i-1}$ turns right $\Rightarrow P_i$ can't be the minimal polar angle. This is a contradiction to P_i is the

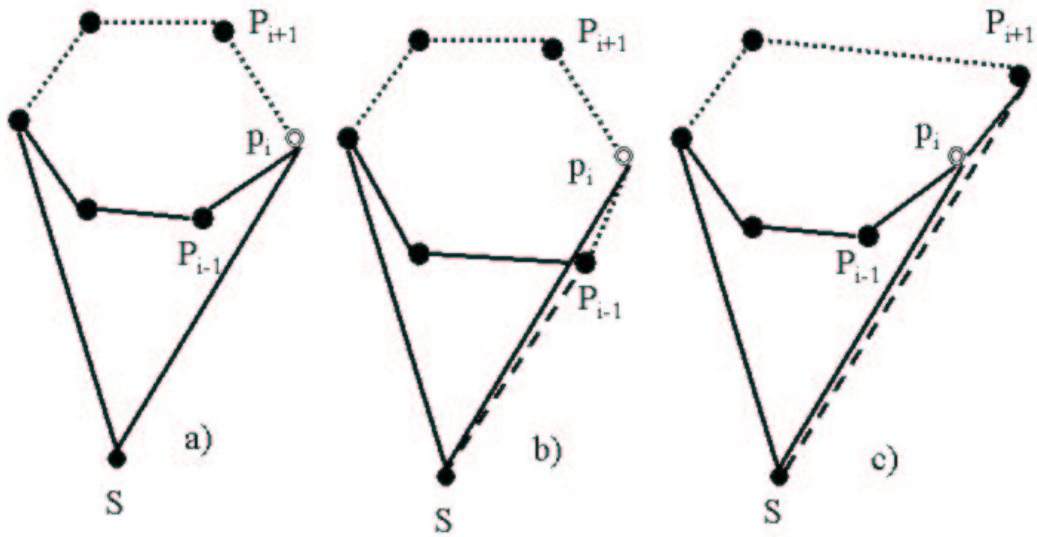


Figure 12: The white point P_i was the output as minimum of polar coordinates. a) is correct while in b) and c) there is a contradiction. The whole lines of the polygons are visible from S the dotted are not.

minimal polar coordinate. b) When both edges are visible $\overline{P_i P_{i+1}}$ has to be visible. Therefore P_{i+1} has to lie right of the infinite line on which S and P_i lie. Now again S $P_i P_{i+1}$ turns right.

2) Every tangential point is found.

Assume there is a point P_i that is tangential after our definition above and has neither the minimal nor maximal polar angle. One of $\overline{P_i P_{i-1}}$ and $\overline{P_i P_{i+1}}$ now has to be visible and the other one not. Let us assume without loss of generality $\overline{P_i P_{i+1}}$ to be visible and $\overline{P_i P_{i-1}}$ not. Since P_i does not have the minimal polar angle there has to be a point $P_j, i \neq j$ that so that S $P_i P_j$ turns right. But S P_i, P_{i-1} has to turn left or $\overline{P_i P_{i-1}}$ would be visible as well. Between $i-1$ inclusive and j exclusive there has to be a point k for which S $P_k P_{k-1}$ turns right and S $P_k P_{k+1}$ turns right as well. \Rightarrow at P_k there is an angle greater 180° . \Rightarrow The polygon was not convex. Which is a contradiction because all given polygons are convex.

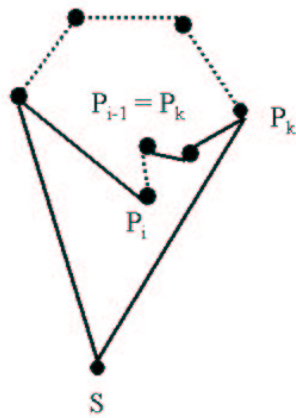


Figure 13: The whole lines of the polygons are visible from S the dotted are not.

4.3 Tangents between a point and several intersecting Obstacles

The tangents to every single obstacle are computed and just like with polygons the points with the maximum or minimum polar coordinates are the tangential points.

4.4 Tangents between two obstacles

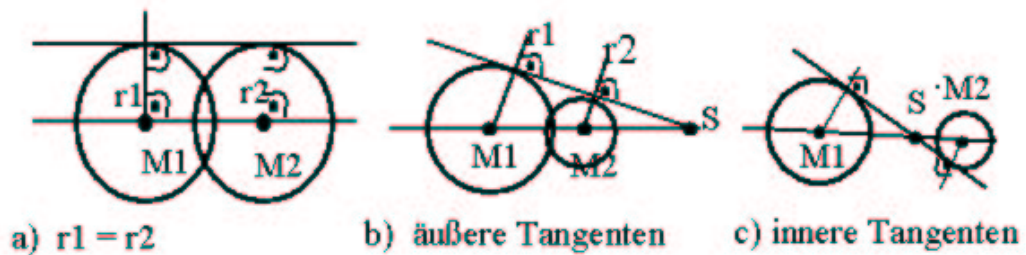


Figure 14: Tangents of a) two circles with the same radius b) inner tangents c) outer tangents

When the radices of the two circles are the same length then the outer

tangents are the two lines parallel to the line that lies on both middle points g_m . The tangents have the radius as distance from the point. Therefore the points where the tangents touch the circle can be computed by $(M_1 - M_2)$ normalized ($= d$) as the direction g_m . The vector which stands orthogonal on this vector can be computed with help of the cross product \bar{d} . Now the tangential points are $m_1 + r * \bar{d}$ and $m_2 + r * \bar{d}$ that belong together and $m_1 - r * \bar{d}$ and $m_2 - r * \bar{d}$.

When the radices are of different length then a point S exists, where the tangent cuts g_m . It is for both tangents the same point. When one computes from this point the left tangents on both circle or the right tangents on both circles these are the points that belong together. Since both vectors of the radices r_1, r_2 stand orthogonal on the tangent they have to be parallel. Therefore by the theorem on intersecting lines $\frac{|r_1|}{|SM_1|} = \frac{|r_2|}{|SM_2|}$ By simple calculations now S can be found.

If both circles do not intersect the inner tangents can be found in a similar way.

When one of the obstacles is a polygon from each of its points the tangents

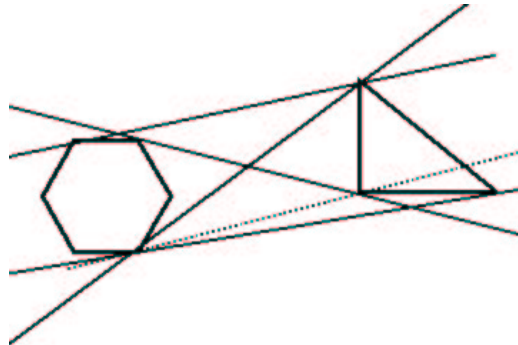


Figure 15: The dashed line is no tangent since it would cut the triangle

on the other one are computed and all tangents that cut one or both obstacles are thrown away.

4.5 Tangents from one group of obstacles to another group

Are computed brute force by computed by finding tangents between every obstacle from one group to the other group and testing for intersections. If a better way could be found to compute this case many cases of path planning could be made faster.

5 Improving the Algorithm

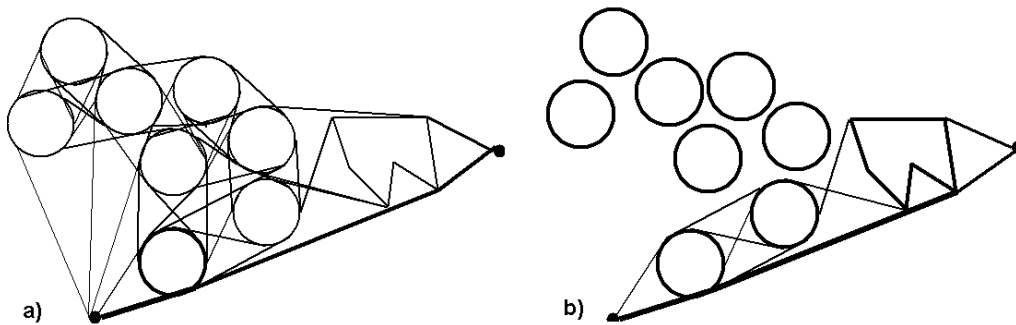


Figure 16: In a) many paths that are too long anyway are searched. b) is improved.

5.1 Avoiding unnecessary additional part-paths

Often while computing paths the way described above paths are computed that lead very far away from the goal, even if a much shorter path is already found. In order to prevent this kind of situation calls to compute part-paths are not made recursively but saved in a priority queue. The priority queue is ordered after the approximate cost of the complete weight of a path. A part-path is either from a point (in most cases the start point) to an obstacle or from obstacle to obstacle or from an obstacle to a point (usually the endpoint). When the start is an obstacle then it saves a list of points on the obstacle already reached. The weight is either the weight of the cheapest

start point(the point or one of the points on the obstacle) and the distance from this point to the endpoint, or the cheapest start point + distance of this point and the middle point of the next obstacle to be crossed + the distance of this obstacles middle point to the endpoint. When a path to the endpoint is found one tests weather there is still a part-path in the queue that promises to be better. If not the algorithm returns the first intermediate point that the robot should go to.

5.2 Avoiding multiple computation of the same part-paths

Often it happens, that part-path's to be computed are the same. For example obstacle o1 was reached on two different paths. But in both cases to reach the endpoint a way from o1 to o2 has to be found. To compute such paths twice (or even more often) will kill the running time very fast. One part-path to compute can lead to several new part-paths that have to be computed. That is why when adding a part-path to the queue it has to be tested weather the same path is already in the queue. If there is one, both part path have to be merged. If they have a list of points these lists are merged. They get the lower weight of both part paths.

5.3 Not computing the whole path

When a robot wants to walk a complicated path he sends the data to the path planer which will return him an intermediate point (and direction) which is the next point to go to. A short time before he reaches that point he will send his data to the path planer again to receive the next intermediate point. Therefore no information about the path as whole is needed but only the first point to go to after the start point.

Every time a part-path is taken from the queue a counter is incremented. Now it is possible to set a maximum of part-paths to be computed. On the one hand this will guarantee a maximum search time no matter how complicated a situation on the other hand computation of long paths can be made much smaller even though some accuracy might be lost. Most of the time the whole part can be computed with around 10 steps. A step costs in debugging mode approximately 0.0005 to 0.001 s. 10 steps times 0.001 s would be already 0.01 s a computing time that cant be accepted. Reducing

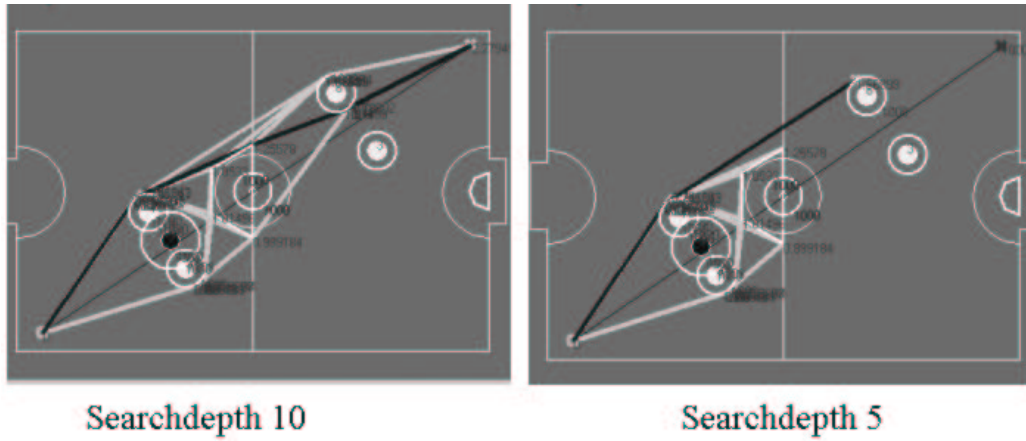


Figure 17: Searching 10 part-paths the endpoint is reached. Searching 5 part-paths the first intermediate point is still the same.

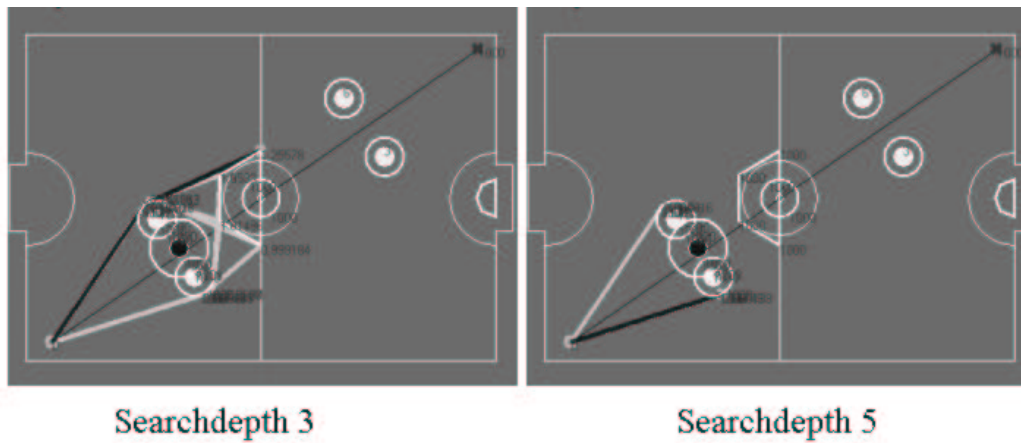


Figure 18: Only when the number of part-paths searched is decreased to 2 does the first intermediate point change.

the number of steps to 5 will almost cut the running time to half while in most of the situations the results will be similar.

6 Smoother Paths

6.1 Characteristics of Bezier-Curves

In the Algorithm only Bezier curves are used with 4 control points. Important characteristics of these curves for this program are:

The Bézier curve always passes through the first and last control points and lies within the convex hull of the control points. The curve is tangent to P_0, P_1 and $[P_2, P_3]$ at the endpoints.

after [10]

The curve is not only tangent at the endpoints, but the longer the distance between the endpoint and the next point the smaller is the second derivation and therefore the curvature is. The second derivation equals the curvature. This approximates the reality of computers, where the faster the robot is the more space he needs to turn around.

The second derivation of the curve is easy to compute, another characteristic of the bezier curves.

After a homepage about the bezier-curves, [11], an equation for the bezier curves is

$$x(t) = a_x t^3 + b_x t^2 + c_x t + x_0 \quad (1)$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + y_0 \quad (2)$$

$$(3)$$

where t ranges from 0 to 1 and

$$c_x = 3(x_1 - x_0) \quad (4)$$

$$b_x = 3(x_2 - x_1) - c_x \quad (5)$$

$$a_x = x_3 - x_0 - c_x - b_x \quad (6)$$

$$(7)$$

$$c_y = 3(y_1 - y_0) \quad (8)$$

$$b_y = 3(y_2 - y_1) - c_y \quad (9)$$

$$a_y = y_3 - y_0 - c_y - b_y \quad (10)$$

$$(11)$$

Now the first derivation S at a point t_0 is:

$$S(t_0) = \lim_{h \rightarrow 0} \left(\frac{a_x(t_0^3 - (t_0 + h)^3) + b_x(t_0^2 - (t_0 + h)^2) + c_x(t_0 - (t_0 + h)) + x_0}{a_y(t_0^3 - (t_0 + h)^3) + b_y(t_0^2 - (t_0 + h)^2) + c_y(t_0 - (t_0 + h)) + y_0} \right) \quad (12)$$

$$\quad (13)$$

$$S(t_0) = \lim_{h \rightarrow 0} \left(\frac{a_x t_0^3 + b_x t_0^2 + c_x t_0 + x_0}{a_y t_0^3 + b_y t_0^2 + c_y t_0 + y_0} \right) \quad (14)$$

$$\quad (15)$$

$$\quad (16)$$

$$S(t_0) = \lim_{h \rightarrow 0} \left(\frac{a_x t_0^3 + b_x t_0^2 + c_x t_0 + x_0}{a_y t_0^3 + b_y t_0^2 + c_y t_0 + y_0} \right) \quad (17)$$

$$\quad (18)$$

$$\quad (19)$$

$$S(t_0) = \lim_{h \rightarrow 0} \left(\frac{-h(3a_x t_0^2 + 2b_x t_0 + c_x)}{3a_y t_0^2 + 2b_y t_0 + c_y} \right) \quad (20)$$

$$\quad (21)$$

$$S(t_0) = \frac{-h(3a_x t_0^2 + 2b_x t_0 + c_x)}{3a_y t_0^2 + 2b_y t_0 + c_y} = \frac{x'(t_0)}{y'(t_0)} \quad (22)$$

[...] no line can have more intersections with a Bézier curve than with the curve obtained by joining consecutive points with straight line segments.

after [10]. Therefore it is quite easy to test whether the line can intersect the curve.

6.2 Use of Bezier Curves in the Program

In the program the curves are approximated by smaller straight lines. It is possible to adjust the number of lines to make the program faster if necessary. The start end endpoints are the points on the path. The other two points are computed of the speed and direction the robot moves at that moment and is supposed in the end. The maximum speed is set to a certain length (it has to be tested with real robots which numbers work best). When the robot is not moving the start and first reference point are identical. The same is with the endpoint. The path is computed as written above. but the test weather or not a part-path cuts an obstacle is not made with a line but with the curve. It is not necessary to test weather the curve cuts the obstacle in many cases. Often its enough to find out that the obstacle and the convex hull don't cut. The speed for the robot in between is always set to maximum (= 1) the direction is parallel to the tangent on the obstacle it uses. This causes that the robot walks as far away from the obstacle as possible.

7 Results

In a test 5+5 robots played against each other. The time using the old and new path planer where logged. From 2 times 500 times path planning PP3 needed 0.022 s / 0.017 s in average and 0.17 s / 0.12 s as a maximum. PPG needed with a search depth of 10 0.0025 s in average and 0.08 s as a maximum. Reducing the search depth to 2 did bring no advantage (which i cant explain): 0.0026 s in average and 0.096 s in maximum. Using beziers only made a slight difference in time: the average raised to 0.0028 s. PPG is about 10 times as fast in average case then the older path planer PP3.

8 Future Work

References

- [1] C. Clark, S. Rock and J. Latombe "Motion Planning for Multiple Mobile Robot Systems Using Dynamic Networks"
- [2] D. Parson and J. Canny "A Motion Planner for Multiple Mobile Robots"

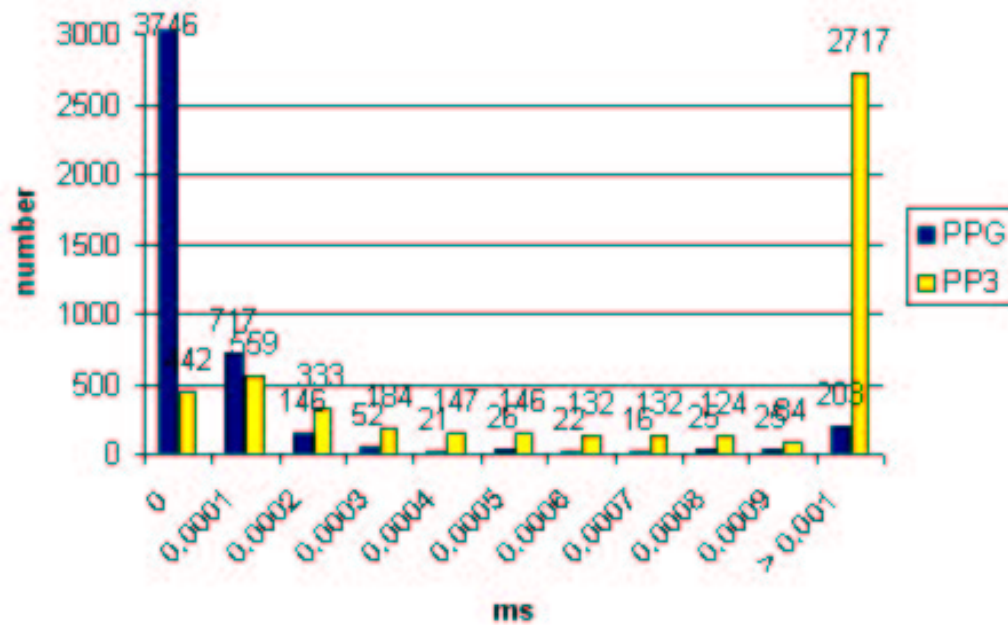


Figure 19:

- [3] B. Brummitt, A. Stenz "Dynamic Mission Planning for Multiple Mobile Robots"
- [4] H. Alt and E. Welzl "Visibility Graphs and Obstacle avoiding Shortest Paths"
- [5] M. Bennewitz, W. Burgard "Finding Solvable Priority Schemes for Decoupled Path Planning Techniques for Teams of Mobile Robots"
- [6] Cormen, Leiserson, Rivest, Stein "Introduction to Algorithms", second edition, The MIT Press, 2001
- [7] E. J. Borowsky and J. M. Borwein Dictionary Mathematics
- [8] Anna Egorova "MAAT - Multi Agent Authoring Tool Programming Autonomous Mobile Robots", Diplomarbeit bei Prof. Rojas Freie Universität Berlin, Institut für Informatik, 2004

- [9] A. Glove, M. Simon, A. Egorova, F. Wiesel, O. Tenchio, M. Schreiber and R. Rojas, "Hardware and Software of the FU-Fighters 2003" Freie Universität Berlin, Institut für Informatik, 2003

Online References

- [10] <http://mathworld.wolfram.com/BezierCurve.html>
- [11] <http://www.moshplant.com/direct-or/bezier/math.html>