

Obstacle Avoidance and Path Planning

Using Mixed Integer Programming

Dave Coleman

Department of Computer Science
University of Colorado Boulder
Boulder, CO, USA
david.t.coleman@colorado.edu

Abstract— In this paper it is demonstrated that obstacle avoidance and shortest-route path planning, traditionally highly non-linear problems, can be modeled as linear problem and solved as a mixed integer programming. In particular, any number of obstacles each reduced to convex polygons can be navigated in two dimensions using discretized time steps from an initial to target point. A benefit of this approach is the problem can be readily solved with off the shelf optimization software such as GLPK. A simple implementation is presented with visualizations and computational efficiency results are discussed.

Keywords – mixed integer programing, linear programing, obstacle avoidance, shortest path, path planning, robot

I. INTRODUCTION

Robot navigation is a well-developed field to which many existing solutions already exist. Grid-based search, geometric algorithms, potential fields and sampling-based algorithms are all common methods for reducing a set of known obstacles, a start configuration and a goal configuration into a set of feasible continuous motions [1][2].

This work is an extension of two previous papers work on obstacle avoidance and takes their methods a step further. Shukla et. al. proposes a linear programming-based approach for optimizing the path of a robot arm in an obstacle oriented work cell [3]. It is imagined that a 3-dof overhead gantry robot is navigating a workspace consisting of various obstacles. It is assumed that every obstacle can be approximated as being cloaked with rectangle and that the robot arm path is only has to avoid one obstacle at a time. This assumes that obstacles are never within close proximity or partially overlapping. The LP problem is then two solve the shortest of only 3 potential paths – going left, going right or going above the rectangle. While computationally very efficient, it is felt that the assumptions are to constraining to any practical real-world application.

More recent work on obstacle avoidance and path planning by Schouwenaars et. al. proposes a new approach to fuel-optimal path planning of multiple vehicles using a both linear and integer programming [4]. In this work it is demonstrated that stationary and moving obstacles can be planned along with multiple other moving vehicles at the same time. Both time and fuel is optimized using space equations of a dynamical system. Again, the assumption of rectangular obstacles are assumed and more complex convex polygons are ignored.

This paper presents the various stages of development of the presented ILP problem and describes the reasoning behind the various chosen constraints and values. A software implementation using the Modeling Language for Mathematical Programming (AMPL) and GNU Linear Programming Kit (GLPK) with a C++ and OpenGL interface to solve the mathematical program is presented. Finally a comparison of the computational time for various sized polygons is presented.

II. BASIC OBSTACLE AVOIDANCE OF RECTANGLE

Initially the work of Shukla was replicated in two dimensions where the LP problem was simply to choose between moving left or right around a rectangle to optimize the distance traveled to a goal point.

Figure 1. Two possible paths to travel: to the left of the obstacle or to the right.

To move around a triangle at most two points need to be found, \bar{x}_1 and \bar{x}_2 , as shown in Figure 1. The constraints on these two points change depending on whether the shortest path is to the left or to the right of the triangle. This decision is described in the LP program by adding two binary decision variables A and B. Both A and B must be the values of 0 or 1 and only one can be true at a time. As such, when the LP program solves the variable A to be 1, the optimal path is to the left, and when the program, solves B to be 1, the optimal path is to the right.

Using these two variables A and B, the proper constraints can be enabled by multiplying the binary variables by the obstacle's dimensions. For instance, the x component of the \bar{x}_1 and \bar{x}_2 coordinates must both be less then the left hand side of the obstacle when path A is chosen, but must be greater than

the right hand side of the rectangle when path B is chosen. This can be expressed in AMPL format as the following:

$$\begin{aligned}
 &\text{var } A, \text{ binary;} \\
 &\text{var } B, \text{ binary;} \\
 &c1: A + B = 1; \\
 &c2: 0 \quad * A + x_{\text{obstacle_right}} * B \leq x1; \\
 &c3: x_{\text{obstacle_left}} * A + x_{\text{world}} * B \geq x1;
 \end{aligned} \tag{1}$$

Distance is calculated using a one-norm approximation, or Manhattan distance, to keep the problem linear. The objective function is thus to minimize the absolute value between each set of x and y points. Elimination of each absolute value requires that a new variable be added for representing absolute values in a linear function. The following AMPL expression is then created:

$$\begin{aligned}
 &\text{minimize distance: } \text{abs1} + \text{abs2} + \dots; \\
 &a1: x1 - xi - \text{abs1} \leq 0; \\
 &a2: -x1 + xi - \text{abs1} \leq 0; \\
 &a3: y1 - yi - \text{abs2} \leq 0; \\
 &a4: -y1 + yi - \text{abs2} \leq 0;
 \end{aligned} \tag{2}$$

A testing program is run to visualize the results of this basic planning optimizer and example results are shown in Figure 2. It should be noted that the solution path directly grazes the obstacle in both examples because the classic assumption has been made that the obstacles have already been enlarged to account for the size of the moving object, reducing the object itself to a moving point [1].

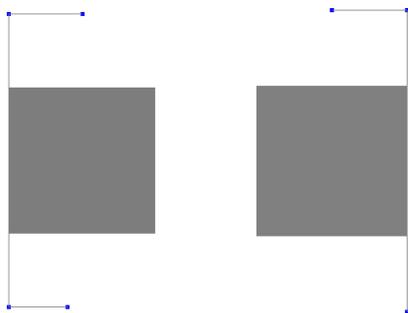


Figure 2. Two examples of a solved path from the initial bottom point to the goal top point around a rectangle obstacle.

As is readily evident in Figure 2, the results of Shukla’s LP algorithm are not the optimal path around the obstacle due to the sharp right angle taken at the two turns. The optimal path would follow a diagonal path directly from the initial position to the bottom corner of the obstacle, and from the top corner of the obstacle to the goal point. To improve this path planning model, this author modified the original LP problem’s objective function to increase the “importance” or cost of the distance between the midpoint points \bar{x}_1 and \bar{x}_2 . This was accomplished by simply multiplying the distance in the x and y directions for these points in the objective function by 2. The modified objective function then looked like the following:

$$\begin{aligned}
 &\text{minimize distance: } \text{abs1} + \text{abs2} + \text{abs3} + 2 * \text{abs4} + 2 * \text{abs5} \\
 &\quad + 2 * \text{abs6} + \text{abs7} + \text{abs8} + \text{abs9};
 \end{aligned} \tag{3}$$

The results were a set of points that followed the optimal path around the rectangular obstacle, as shown in Figure 3. The entire AMPL model of this implementation can be found in Appendix A of this paper.

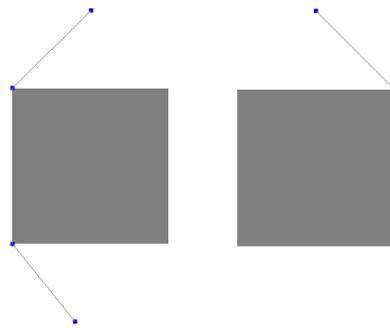


Figure 3. Two examples with same initial and goal locations as in Figure 4 but with optimal path objective function variation.

III. OBSTACLE AVOIDANCE OF CONVEX POLYGON

The previous implementation suffers from the need to have the start position below the rectangle, the goal position above the rectangle and no more than one obstacle of rectangular shape oriented normal to the axis. The following is a highly modified LP model of the previous problem that works for any number of arbitrary convex polygons in 2 dimensions. An example of its ability is demonstrated in Figure 4. The entire AMPL file for this implementation is presented in Appendix B.

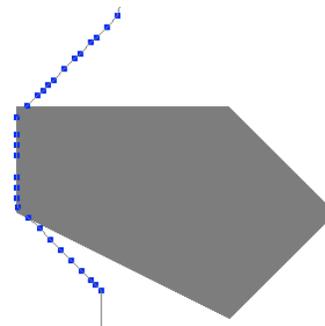


Figure 4. Example of obstacle avoidance of convex polygon

A. Convex Polygon

It is assumed that each obstacle is presented as an ordered list of points that already form a convex polygon. If a non-convex obstacle is desired to be included, use of a preexisting method for computing the convex hull such as the Graham scan must be first run [5].

B. Time Steps

In the previous implementation of a rectangular obstacle avoidance scheme most of the problem parameters were already known and hard-coded into the solution. In contrast, this second implementation only assumes the location of the start and goal points are in a non-conflicting state, i.e. not within an obstacle. Their location within the solution space is

allowed to be anywhere. Additionally, it is important that this second implementation work for any number of obstacles each with any number of edges. It is therefore impossible to hard code a pre-determined number of paths and solution points.

As such, in this LP implementation, N number of time step coordinates are added. Each coordinate consists of an x and y point that are each treated as problem variables in the LP problem. Constraints are added for each time step point such that there must be at least 0.5 units and no more than 1 units of absolute one-norm distance from the previous time step point.

The purpose of the 0.5 units of distance minimum between points is to ensure forward progress of the solution path and prevent all of the points from clustering at one location. The purpose of the 1 unit of distance maximum between points is to ensure that obstacles are not bypassed or “jumped” over by having two consecutive points locate themselves on opposite sides of an obstacle, forcing the path into the impossible condition of cutting through an obstacle. Still, the maximum of 1 unit allows the corners of obstacles to be slightly “cut” as shown in Figure 5. This condition is addressed by remembering that obstacles are enlarged to account for the size of moving object and can be further increased as needed. Decreasing the maximum spacing between time step points would reduce the amount of corner cutting but at the cost of computation.

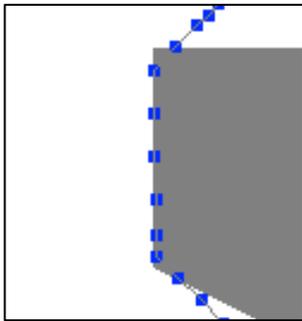


Figure 5. Corner-cutting issue demonstrated

The calculation of how many time steps N are necessary to find an optimum path is still an area of further research. Currently it is found by calculating the one-norm distance between the initial and goal points and increasing that number by 20%. Setting the number of time steps N too low can result in the LP problem becoming infeasible because of insufficient “stretch” of the path to go around the obstacle. Setting N too high can result in the path “over crowding” or overlapping such that the path take unnecessary diversions, such as shown in Figure 6.

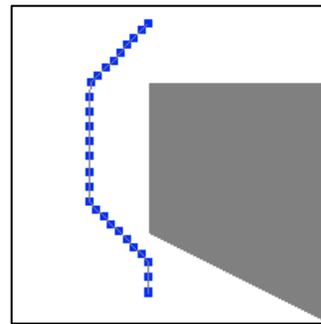


Figure 6. Too many time-step points results in unnecessary diversion and sub-optimal path.

C. Adding Polygon Constraints

For a solution space of T time steps, P polygons each with E edges, $T \times P \times E$ number of constraints are added to the LP program. For each polygon an *or* statement is created using an binary variable that requires at least one of the E number of edge constraints to be true for every point in the path. An edge constraint is easily represented by using the two-point form of a line and solving for x and y .

$$m = \frac{y_2 - y_1}{x_2 - x_1} \tag{4}$$

$$-mx + y \leq y_1 - mx_1 \tag{5}$$

Where x and y are the problem variables. Value m and the right hand side of (5) are pre-computed in the C++ interface to reduce redundant computation in the AMPL file.

The direction of the equality sign in (5) must be chosen based on whether the edge of the polygon it represents is facing up or down with respect to one of the axis as shown in Figure 7. The y-axis is chosen as the determining axis based on intuition and the C++ interface is once again utilized to pre-process the facing direction of each edge. This is accomplished by first finding the minimum and maximum x values in the set of points in the polygon. The points are then looped through starting at the point with the minimum x value in a counter-clockwise direction until the point with the maximum x value is reached. All of the points in this half of the polygon are set to face down. The loop then continues until it reaches the minimum x value again. The points in the second half of the polygon are set to face up. This face up/face down setting is represented by giving each point a directionality parameter of either -1 for down or 1 for up. This variable can then be used in the AMPL format to change the direction of each edge constraint equation.

Figure 7. A polygon’s edges pointing either up or down

A special case exception of the above method for adding edge constraints is when a line is exactly vertical and parallel with the y-axis. In this condition the slope m is undefined because the denominator is 0. To avoid this issue a pre-processing check in C++ is performed that checks for any two equal consecutive x values. If one is found, one of the points is artificially perturbed by adding a small number, in this case 0.1. The amount 0.1 was chosen after the value 0.01 was demonstrated to occasionally present unsolvable problems, most likely due to rounding errors.

The afore described edge constraints are described in two lines in the AMPL file.

$$\begin{aligned} \text{s.t. obstacles}\{t \text{ in TT}, e \text{ in Edges}\}: & \quad (6) \\ \text{obst}[e,3]*\text{obst}[e,1]*\text{points}[t,1]-\text{obst}[e,3]*\text{points}[t,2] & \leq \\ -\text{obst}[e,3]*\text{obst}[e,2] + M*\text{orer}[t,e]; & \end{aligned}$$

$$\text{s.t. obstOR}\{t \text{ in TT}\}: \text{sum}\{e \text{ in Edges}\} \text{orer}[t,e] \leq E-1; \quad (7)$$

Where $obst$ is an array of edges e describing at index 1 the slope m , at index 2 the right hand value of the constraint, and at index 3 the direction of the constraint. The variable $orer$ represents the binary variable for the *or* condition. M is an arbitrarily large number that in this problem is set to 1,000. If set to 10,000 it was found to significantly increase the computation time of the LP problem.

D. Simulating Quadratic Distance

Similarly to Shukla's implementation of rectangular path planning, this algorithm suffers from the non-optimal solution of one norm distance measurements. This implementation's objective function is the sum of the absolute value of each point's one norm distance to the next point.

$$\min \sum_{i=1}^{T-1} |x_t - x_{t+1}| + |y_t - y_{t+1}| \quad (8)$$

This results in the LP problem having no notion of diagonal lines for shorter paths. This notion is introduced into the LP problem by adding an additional objective component – minimizing the difference between the change in the x and y component for each point.

$$\begin{aligned} \min \sum_{i=1}^{T-1} |x_t - x_{t+1}| + |y_t - y_{t+1}| + & \quad (9) \\ 0.5 * \sum_{i=1}^{T-2} ||x_t - x_{t+1}| - |y_t - y_{t+1}|| & \end{aligned}$$

This second half of the objective function encourages more horizontal movements – equal changes in the x and y direction – to occur. The 0.5 weight reduction factor ensures that the diagonal traverse movement is considered less important than overall shortest-path optimization and is necessary to prevent path wandering.

This addition to the objective function is effective in increasing the number of 45° path angles but is unable to encourage translations of any other angle such as 30° or 60°. Still, it is more optimal than using only a taxi-cab grid.

IV. RESULTS

The previously described AMPL model is solved using GLPK that is run from within a custom written C++ program. All adjustable parameters of the model including start and end location, polygon shapes and the number of time steps are passed to the AMPL model via a data.dat file that is written by the C++ program. Once GLPK finished solving the LP problem, the results are written to a results.dat file and re-read into the C++ program. The program then draws the obstacle and solved path using OpenGL.

The following table shows the results of 5 scenarios and their required computation time on a 2 Ghz Quad Core i7 MacBook Pro running with an Ubuntu virtual machine. The screenshots of each result is then shown after.

TABLE I. OBSTACLE AVOIDANCE COMPUTATION TIME

	<i>Edges</i>	<i>GLPK Time</i>	<i>Figure #</i>
1	4	0.6 s	Figure 8
2	4	2.9 s	Figure 9
3	5	1.5 s	Figure 10
4	5	2.6 s	Figure 11
5	6	3.1 s	Figure 12
6	8	0.2 s	Figure 13

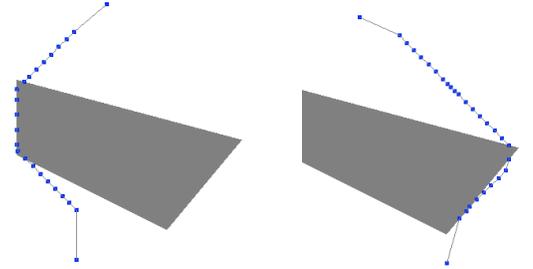


Figure 8

Figure 9

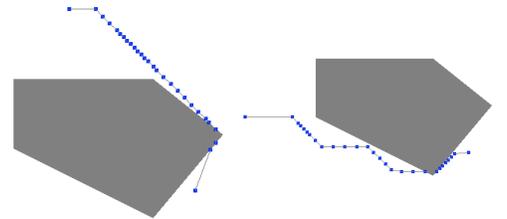


Figure 10

Figure 11

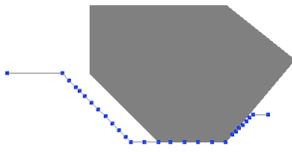


Figure 12

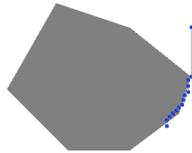


Figure 13

V. FUTURE WORK

Because the number of time steps N directly correlates to the computational time needed to solve a problem, finding the best value for N is an area of future work. Better handling of the corner-cutting issue is another area of future work. Finally, applying this method to a third dimension of path planning is a large area of exploration.

VI. CONCLUSION

The proposed method of path planning around polygon-shaped obstacles still has many issues to overcome before being a practical algorithm. The method had very poor computational efficiency, requiring $T \times P \times E$ number of

constraints and a huge number of variables. In the 8-edge example shown in Figure 13 the generated LP problem contained 186 rows, 138 columns and 77 binary integer variables. Overall, there are many more-efficient methods for planning around obstacles than the presented LP problem in this paper.

ACKNOWLEDGMENT

Thanks to Dr. Sriram Sankaranarayanan for his teaching the Linear Programming class in Fall 2011 at CU Boulder.

REFERENCES

- [1] J. Latombe, Robot Motion Planning, Kluwer Academic Publishers, 1991
- [2] J. Barraquand, L.E. Kavraki, J.C. Latombe, T.Y. Li, R.Motwani, and P.Raghavan. A random sampling scheme for path planning. International Journal of Robotics Research , 16(6):759.774, 1997.
- [3] A. Shukla, D. Sule, D. Furtado, A Linear Programming Approach For Optimizing the Path of Robot Arm In An Obstacle Oriented Work Cell, Computers and Industrial Engineering Vol. 23, 1992
- [4] T. Schouwenarrs, B. Moor, E. Feron, J. How, Mixed Integer Programming for Multi-Vehicle Path Planning, ECC2001 Conference, 2001.
- [5] R. Graham, An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set, Information Processing Letters, 1972

Appendix A: Rectangular Obstacle Avoidance

```
# Parameters -----
# Initial Point:
param xi; param yi; param zi;

# Final Point:
param xf; param yf; param zf;

# Obstacle Coordinates
param xol; param yol; param zol;
param xo2; param yo2; param zo2;
param xo3; param yo3; param zo3;
param xo4; param yo4; param zo4;

# Workspace Constraints
param xm; param ym; param zm;

# Variables to Solve for -----

var x1; var y1; var z1; # first point
var x2; var y2; var z2; # second point
var abs1; var abs2; var abs3; var abs4; var abs5; var abs6; # absolute values
var abs7; var abs8; var abs9; var abs10; # absolute values
var A, binary; var B, binary; # path choices

# Problem -----
minimize distance: abs1 + abs2 + abs3 + 2*abs4 + 2*abs5 + 2*abs6 + abs7 + abs8 + abs9;

# path choices
path: A + B = 1;

# x1
l1: 0 *A + xo2*B <= x1; # x1 greater than
h1: xol*A + xm *B >= x1; # x1 less than

l2: 0 <= y1;
h2: yol*A + yo2*B >= y1;

l3: 0 <= z1; h3: z1 <= zm;

# x2
l4: 0 *A + xo3*B <= x2;
h4: xo4*A + xm *B >= x2;

l5: yo4*A + xo3*B <= y2;
h5: ym >= y2;

l6: 0 <= z2; h6: z2 <= zm;

# Distance 1
a1: x1 - xi - abs1 <= 0; # | x1 - xi |
a2: -x1 + xi - abs1 <= 0;
a3: y1 - yi - abs2 <= 0; # | y1 - yi |
a4: -y1 + yi - abs2 <= 0;
a5: z1 - zi - abs3 <= 0; # | z1 - zi |
a6: -z1 + zi - abs3 <= 0;

# Distance 2
a7: x2 - x1 - abs4 <= 0; # | x2 - x1 |
a8: -x2 + x1 - abs4 <= 0;
a9: y2 - y1 - abs5 <= 0; # | y2 - y1 |
a10: -y2 + y1 - abs5 <= 0;
a11: z2 - z1 - abs6 <= 0; # | z2 - z1 |
a12: -z2 + z1 - abs6 <= 0;

# Distance 3
a13: xf - x2 - abs7 <= 0; # | xf - x2 |
a14: -xf + x2 - abs7 <= 0;
a15: yf - y2 - abs8 <= 0; # | yf - y2 |
a16: -yf + y2 - abs8 <= 0;
a17: zf - z2 - abs9 <= 0; # | zf - z2 |
a18: -zf + z2 - abs9 <= 0;

solve; # directive to solve
display x1, y1, z1, x2, y2, z2, A, B; # print result

printf: "%.3f %.3f %.3f\n", x1, y1, z1 > "result.dat";
printf: "%.3f %.3f %.3f\n", x2, y2, z2 >> "result.dat";
printf: "%d %d\n", A, B >> "result.dat";

end;
```

Appendix B: Polygon Obstacle Avoidance

```

# Other Parameters -----
param N default 4; # number of time steps / points
param NN := N - 1;
param M := 1000; # arbitrary large positive number
param m := 0.001; # arbitrary small number to prevent division by zero
param E default 4; # number of edges in polygon

set T := 1..N; # number of time steps / points
set TT := 1..NN; # one less than number of time steps
set Dims := 1..2; # 2 dimensions: x and y
set C := 1..4; # number of manually created abs constraints
set Edges := 1..E; # number of total edges in polygons
set ConstData := 1..3; # slope, constraint, direction

# Input Parameters -----

# Initial Point:
param xi; param yi; param zi;

# Final Point:
param xf; param yf; param zf;

# Obstacle Description - edges x 3 data
param obst{e in Edges, c in ConstData};

# Variables to Solve for -----

var points{t in T, d in Dims} >= 0; # set of x,y points for problem solution
var abs1{t in TT, d in Dims}; # middle steps
var abs2{c in C}; # initial and final steps
var abs3{t in TT}; # optimize for hypotenus
#var abs4{c in C}; # limit distance in initial and final steps
var orer{t in T, e in Edges} binary; # vars used for doing ORs

# Objective Function -----
minimize distance: sum{t in TT, d in Dims} abs1[t,d] + # all constraints between initial and final
                    sum{c in C} abs2[c] + # initial and final point constraints
                    .5*sum{t in TT} abs3[t]; # optimize hypotenus

# Constraints -----

# ABS Distance between midpoints
s.t. abs_min{t in TT, d in Dims}: points[t+1,d] - points[t,d] - abs1[t,d] <= 0;
s.t. abs_max{t in TT, d in Dims}: -points[t+1,d] + points[t,d] - abs1[t,d] <= 0;

# Point to point distance limiter - more than .5 less than 2
s.t. abs_diff1{t in TT}: abs1[t,1] + abs1[t,2] >= .5;
s.t. abs_diff2{t in TT}: abs1[t,1] + abs1[t,2] <= 1;

# Point to point distance limiter for init and end
s.t. abs_diff3: abs2[1] + abs2[2] <= 8;
s.t. abs_diff4: abs2[3] + abs2[4] <= 7;

# ABS Distance Between change in x and y per point - optimize to hypotenus
s.t. abs_diff5{t in TT}: abs1[t,1] - abs1[t,2] - abs3[t] <= 0;
s.t. abs_diff6{t in TT}: -abs1[t,1] + abs1[t,2] - abs3[t] <= 0;

# Obstacle Constraints - Square
# direction* slope * x -direction* y <= -direction*constraint+ OR
s.t. obstAll{t in TT, e in Edges}: obst[e,3]*obst[e,1]*points[t,1]-obst[e,3]*points[t,2] <= -obst[e,3]*obst[e,2] +
M*orer[t,e];

#s.t. obst1{t in TT}: -m4*points[t,1] + points[t,2] <= (y01-m4*x01) + M*orer[t,3]; # x_min
#s.t. obst3{t in TT}: -m1*points[t,1] + points[t,2] <= (y01-m1*x01) + M*orer[t,3]; # y_min
#s.t. obst4{t in TT}: m3*points[t,1] - points[t,2] <= -(y03-m3*x03) + M*orer[t,4]; # y_max

#s.t. obst1{t in TT}: points[t,1] <= x01 + M*orer[t,1]; # x_min
#s.t. obst2{t in TT}: -points[t,1] <= -x02 + M*orer[t,2]; # x_max
#s.t. obst3{t in TT}: points[t,2] <= y01 + M*orer[t,3]; # y_min
#s.t. obst4{t in TT}: -points[t,2] <= -y03 + M*orer[t,4]; # y_max
s.t. obstOR{t in TT}: sum{e in Edges} orer[t,e] <= E-1; # at least one must be true

# Initial X Axis
s.t. a1: points[1,1] - xi - abs2[1] <= 0;
s.t. a2: -points[1,1] + xi - abs2[1] <= 0;

# Initial Y Axis
s.t. a3: points[1,2] - yi - abs2[2] <= 0;
s.t. a4: -points[1,2] + yi - abs2[2] <= 0;

# Final X Axis
s.t. a5: points[N,1] - xf - abs2[3] <= 0;

```

```
s.t. a6: -points[N,1] + xf - abs2[3] <= 0;

# Final Y Axis
s.t. a7: points[N,2] - yf - abs2[4] <= 0;
s.t. a8: -points[N,2] + yf - abs2[4] <= 0;

# Solve -----

solve;
#display m1, m2, m3, m4;

printf: "%d\n", 999 > "result.dat";
#printf{t in T, d in D} "%d,%d %.3f \n", t,d,points[t,d] >> "result.dat";
printf{t in T, d in Dims} "%.3f ", points[t,d] >> "result.dat";

#printf: "%.3f %.3f %.3f\n", x1, y1, z1 > "result.dat";
#printf: "%.3f %.3f %.3f\n", x2, y2, z2 >> "result.dat";
#printf: "%d %d\n", A, B >> "result.dat";

end;
```