Universität zu Lübeck

Institute for Robotics and Cognitive Systems

Master Thesis

An approach to solving object displacement problems

written by

Maximilian Mühlfeld (580070)

Supervisor:

Prof. Dr.-Ing. Achim Schweikard

Lübeck, August 24, 2014

Assertion

I assure that the following work is done independently with the use of only stated resources.

Lübeck, August 24, 2014

Abstract

The scope of this thesis grasps the implementation and testing of algoritms to solve object displacement problems.
To achieve this, the search space is divided into multiple cells to reduce its size. Furthermore, instead of calculating the whole space at once, a local approach is used to only calculate the parts needed for the next search step. This needs a wide range of basic geometric and algebraic algorithms for object representation, collision detection and object translation/rotation.
On the resulting search space a simple and exchangable graph search algorithm is applied, which will give us a path in the object space from the start configuration to the desired target configuration.

Contents

1 Introduction

1.1 Motivation

1.1.1 Path planning in robotics

In robotics the path planning of a robot can be programmed as a fixed list of movements to be executed. This has multiple drawbacks as for every change in the enviroment the robot needs to manually be reprogrammed.
But what if we equip this robot with a camera that detects the shape of objects in the robots enviroment. This would give us a set of objects at certain position, a robot arm in a starting position and a target where the robots endeffector needs to work. If we would be able to solve this puzzle, we could direct the robot in a different way each time without the actual need to access its software.

1.1.2 Geometric riddles in gaming

Solving geometric riddles is a amazingly fun task for a human. This is the reason many games simply consists of such riddles ranking from easy to hard in difficulty. But even the easiest riddle for a human proposes a big challenge for a simple algorithm that searches trough the possible ways of solving it. Even more complicated is the generation of such riddles. Even for the human brain this task can be exceedingly stressful.
Now, if there would be a possibility to check if such a riddle has a solution, there would be the option to generate them randomly and check for feasability. This would release the developers of the need to manually create each riddle. Also the consumers, in this case the players, would have a never ending stream of new and different riddles to solve.

1.2 Idea

As the representation behind these two cases is the same, for simplification purposes we break the riddles down into two dimensional object displacement problems. The following riddles shall provide simple example of the problem:

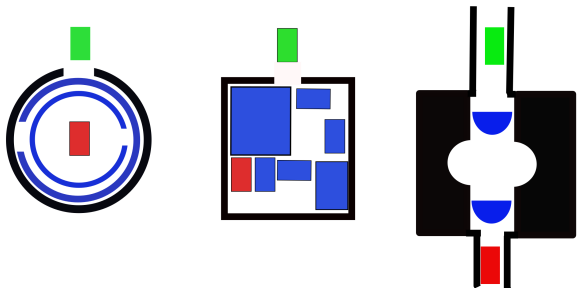

Figure 1.1: *riddle 1 with circles , riddle 2 with boxes , riddle 3 with 2 half-circles*

The riddle is solved, if the red box matches the green one. Blue objects are movable and black ones are stationary. The stationary ones will be referred to as rims hereafter.
As a human the way to solve those is quite obvious. The first riddle is solved by rotating the blue objects, the second through translation. The third needs both ways to be solved.
If we want to solve this with an algorithm, we would need to consider each objects collsision with the other objects and the rim. Also we would need to find a way to express the current configuration consisting of position (x,y) and rotation (ϕ) and the direction the main object needs to take. A simple idea would be to

1. generate all valid configurations per object in regard to the stationary obstacles as a configuration space
2. Create one collision space per object for collision with every other object.
3. Substract the collision space from the configuration space to get valid space in regard to all obstacles for one object.
4. Divide the space in cells and locate the valid ones.
5. Build a graph out of this starting position by adapting the space after each step.
6. Search in that graph to get path from start to target point.

The target point would then be a simple configuration vector holding each position of each

object. A user defined distance function beetween start and target point in that space can then be used as heuristic in the graph search (e.g. $h(start, target) = ||(x_{start} - x_{target}, y_{start} - y_{target})||$).

2 Enviroment and basics

2.1 Programming tools

As the target of this thesis is a mere proof of concept, the programming language of choice is matlab [2]. This is a reasobable decision because in the matlab language a huge part of the needed functionality concerning simple mathematical functions is already implemented and easy to use.
Pros:

- easy to use mathematical function
- fast and easy way to enter data
- good and simple ways of debugging and locating errors

Cons:

- slow computation speed
- needs translation in other language (e.g. c++) for further use

As versioning tool git [1] is used together with www.github.com as an open source storage plattform for the resulting code.

2.2 Basics

For better understanding of the following chapters the central mathematical formulas are repeated here.

2.2.1 basic vector math

If the objects are represented as a list of corner points, vector math comes in handy when describing the borders of the object. Lets say we have a simple object A defined by the following list:

$$A = (0, 0; 2, 0; 2, 2; 0, 2)$$

Each pair x,y defines one corner of A and the points are ordered to travel along the border of A counterclockwise. The lines connecting these points will be called the borders of A. They are calculated by substracting the points from each other such that the vectors point along the border clockwise. This leads to:

$$A_{vec} = (2, 0; 0, 2; -2, 0; 0, -2)$$

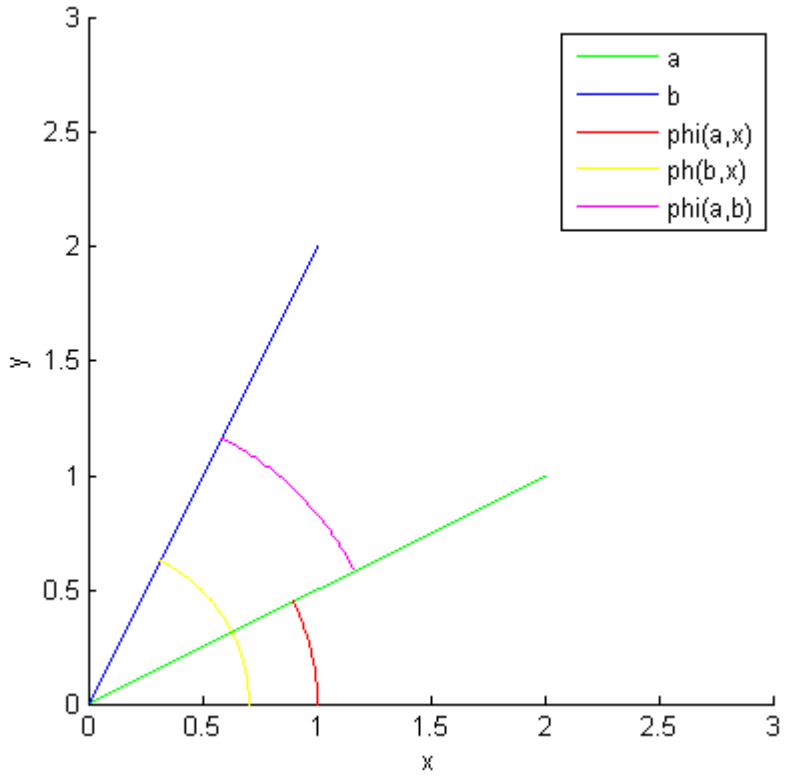

Figure 2.1: *Figure showing the calculation of the angle difference beetween vectors a and b via reference to x-axis*

Next will be the calculation of the angle beetween two lines a and b. This is done by taking a reference vector , for example the x-axis r=$(1, 0)$, calculating the angle to that and building the difference in angle. This way we can determine an angle beetween the vectors a and$b$ where the sign of the angle tells us which vector lies "below" the other.
The formula for this is:

$$\begin{aligned}\phi_{a,b} &= \phi_{a,r} - \phi_{b,r} \\ &= tan^{-1}(\frac{a(2)}{a(1)}) - tan^{-1}(\frac{b(2)}{b(1)});\end{aligned}$$

We need to use another vector as a reference to determine if the vectors angle difference is positiv or negativ. This information is needed to build the convex hull.

2.2.2 Minkowski sum

The minkowski sum is needed to calculate the collision beetween two poyleders. The formula for the minkowski sum is as follows:

$$A + B = \{\mathbf{a} + \mathbf{b} \,|\, \mathbf{a} \in A,\ \mathbf{b} \in B\}.$$

A and B are a set of vectors in 2 dimensional space.

2.2.3 convex hull

Furthermore we define the center of these objects to be $M_A = (1, 1)$ and $M_B = (0.5, 1)$ The convex hull for A to B is calculated with the minkowski sum of A' and B' with

$$\begin{aligned}A' &= \{-(\mathbf{a} - M_A) | \mathbf{a} \in A\} \\ B' &= \{\mathbf{b} - M_B | \mathbf{b} \in B\}\end{aligned}$$

Basically this means we add the points of A shifted with $-M_A$ and mirrored at M_A to each point of B shifted with $-M_B$ and then reshifted with M_B to get a list C_{temp} of $4 \cdot 4 = 16$ points.

$$C_{temp} = \begin{array}{rcccccccc} (& -1, & -1; & 1, & -1; & 1, & 1; & -1, & 1; \\ & 0, & -1; & 2, & -1; & 2, & 1; & 0, & 1; \\ & 0, & 1; & 2, & 1; & 2, & 3; & 0, & 3; \\ & -1, & 1; & 1, & 1; & 1, & 3; & -1, & 3) \end{array}$$

Each line represents one point of B with the points of A added. By choosing the outer points from C_{temp} and put them in C, such that no point from C_{temp} is still outside C we will get an object C which defines the space around B which the object A can not enter or they will collide.

$$C = (-1, -1; 2, -1; 2, 3; -1, 3)$$

2.2.4 Pathfinding algorithms on graphs

Pathfinding in general describes the way of finding a path beetween two points in a graph or net or nodes. The two major algorithms currently in use are A* and Dijkstra's algorithm. As A* is a variant of Djikstra's algorithm, Dijkstra's will be explained first and afterwards the differences will be highlighted.
Dijkstra's algorithm uses a weighted graph and begins on a starting node which is connected to a set of adjacent nodes. These adjacent nodes are the starting rim. Dijkstra works in these steps:

1. Take unvisited node with shortest distance to the start from rim and check if it is the target node.
2. If not target node, try to add all adjacent nodes to rim.
   - If adjacent node is not in rim, add it with predecessor and distance.
   - If adjacent node is in rim check distance:
     - If adjacent nodes distance is higher than node in rim, discard said node.
     - If adjacent nodes distance is smaller than node in rim, update predecessor and distance.
3. Mark node as visited and repeat.

As long as it is guaranteed that no negativ distance occures in the search graph, Dijkstra is bound to return the shortest possible path.

A* is a variant which adds a heuristic function measuring the distance beetween the current node and the target. The heuristic and the distance to the start build a weighted sum. This results in a faster exclusion of possibly wrong paths, but it no longer guarantees to find the optimal path.

3 Concept

3.1 Common defenitions and representation

3.1.1 Defining the configuration space

As the exakt representation of our objects should not matter, we will only define the common points needed for a clear communication of the stated problem.
The algorithms aim is to tell if there is a solution possible and, if so, present it. The object which needs to be moved from a starting configuration to a target configuration will be named main object M. Other movable objects are obstacles named Ob_i and the stationary objects are called rims R_i. This will be combined to the sets $O = \{M\} \cup \{Ob_i | i \in \mathbb{N}\}$ and $R = \{R_i | i \in \mathbb{N}\}$.

Each object O_A contains some data for representing its shape stored under O_A.data. Furthermore the configuration of O_A is given by the vector (x_A, y_A, ϕ_A) and stored under O_A.mid as the middle/reference point for said object where x_A and y_A gives the point around which the object will be rotated by ϕ_A. By substraction of the first two dimensions occupied by R from the possible space O_i per object in O, and, if we divide the space in two, selecting the one in which $O_i.mid$ lies at the start, we get a valid space C_{O_i-R} for the object O_i to be moved in (not taking into account other objects). This space is a simple 3 dimensional space with x_i, y_i and ϕ_i as base.
But as there is the need to check for collision with ALL other objects $O' = \{O_j, |j \neq i \wedge j \in \mathbb{N}\}$ we need to increase the dimensions of all spaces C_{O_i-R} by the number of objects in O'. Also for each set j of dimensions (x_j, y_j, ϕ_j) added, we will need to substract the current position of the corresponding object O_j from the space, such that all collision points are removed from C_{O_i-R}. This will give us the configuration space for object O_i, C_{O_i}.

3.1.2 Building the configuration space

To build such a configuration space, every possible configuration of every movable object needs to be calculated. Even those who are NOT valid need to be computed at least once, to check if they are valid or not.
Each object A has three dimensions (x_A, y_A, ϕ_A), with x and y beeing a finite range from $r_x = [x_l, x_h]$ and $r_y = [y_l, y_h]$ defined by the stationary rims of the riddle. The rotation component ϕ is choosen from a set of angles $\Phi = \{\phi | \phi \in r_\phi = [1, 360]\}$. As this would lead to an infinite amount of possible x, y and ϕ, we could allow steps only, e.g. $x, y, \phi \in \mathbb{N}$.
If there are n movable objects we get a total number of possible combinations in the range of $(r_x \cdot r_y \cdot r_\phi)^n$. Under the premise of saving every calculated combination so that we only calculate each set once and assuming that the time t_s needed for collision check and calculating one set is constant, we get the following formula for the time to calculate the complete configuration space.

$$T_conf = (r_x \cdot r_y \cdot r_\phi)^n \cdot t_s$$

Now we define $t_s = 0.1ms$ set $r_x = r_y = 10$ and $r_\phi = 180$ with only two objects (one main object M one obstacle O) meaning $n = 2$ we get

$$\begin{aligned} T_conf &= (10 * 10 * 180)^2 \cdot 0.1ms \\ &= 324000000 \cdot 0.1ms \\ &= 3.24 \cdot 10^7 \cdot ms \qquad\qquad = 9h \end{aligned}$$

This means we would need to wait 9h to completely calculate all possible positions on a very raw grid (each step just 1 unit) without even having started to search on it.

So the idea is to interleave search and building of the configuration space in such a way, that only the needed nodes are calculated and checked. But still this would be very slow if we consider implementing it on such a grid. Therefore another approach is needed.
Instead of taking a grid, the configuration space is divided into cells depending on the current positition of the objects. This cell division will heavily decrease the number of search nodes in the space. The drawback on the other hand is, that this division is dependent on the object representation. So isntead of computing a independent configuration space for the search, the search needs to use some information from the objects. This will lead to an integration of search algorithm and object representation into one main algorithm.
But still this will give us a simple graph where the solution can then be found by searching for

a path for the main object M from start to target. In the following part a way of representing the objects with points will be used.

3.1.3 Objects as point list

One of the possible ways of describing an object in a two dimensional setting would be an ordered list of corner points. Together with an anchor point we can calculate all transformations needed. To see if this representation would work we take a short look at the algorith described in 1.2 and sketch a solution to each step.

1. Generate C_{O_i-R}:By identifing the main outer rim R_mo and computing its inner hull for O_i the space C_{O_i} is build. For all other rims R_j computing the convex hull with O_i and substracting them from C_{O_i} leads to C_{O_i-R}.
2. Generate $C_{O_i-O_j}$: Calculate the convex hull from O_i to each other object O_j.
3. Generate C_{O_i}: Substraction of $C_{O_i-R} - C_{O_i-O_j}$ for all $O_j \in O \wedge j \neq i$.
4. Divide search space in cells: By extending the vectors connecting the convex hull in $C_{O_i-O_j}$ we get a seperation of the space in C_{O_i} in multiple parts. Each step is a translation of the object O_i from one cell to another. The neighbour cells can be identified by iterationing over the objects O_j and calculating the nearest crossing along (x, y) with the extended vectors of its convex hull.
   Rotations are represented as a jump from one hyperplane to the next in search direction. There are multiple problems with rotation in this representation, that will be discussed later.
5. Construct the search graph: While moving along those cells, we adapt C_{O_i} for each $O_i \in O$ each step. These cells are then added to the graph.
6. Search for target: Again independently of the object representation a search can then be applied to the resulting graph.

So far this representation seems like a good choice in multiple ways with some drawbacks on the other hand.
Pros:

- Simple and intuitive representation of object itself
- Easy and fast to compute concerning the transformation of an object (translation, rotation

)

Cons:

- Rotation needs discrete steps along the config dimensions ϕ_i, therefore more exact calculations lead to higher need in computation power.
- The more corners an object has, the more points its convex hull with other object will have. One point more in object O_i can lead to n more points in $C_{O_i - O_j}$ with n beeing the number of points in O_j. As we use the vectors connecting the points in $C_{O_i - O_j}$, n more cells will arise in the search space due to those "ghost planes".

3.1.4 Objects as function list

Another option of describing an object is the representation with functions and definition ranges. Each function is then represented as a list of coefficients a, b, c describing the polynom ax^2+bx+c. Also an anchor point as a reference is needed for rotating the object.
The algorithm slightly changes for this representation, solving problems that existed with the point list representation, but introducing new ones. One step of moving an object in one direction would be described as follows:

1. The object O_i is moved function by function. So for each function f_{o_i} describing a border, this function will be moved in the searching direction. By eliminating all functions that are not in the way by looking at the definition/value ranges, a lot of computing time is saved.
   Iterating over all other objects $O_j \in O \wedge j \neq i$ and their functions f_{o_j} and solving $f_{o_i} = f_{o_j}$ then yields a solution with information about the distance in search direction.The new configuration for the object O_i can then be computed.
2. The function with the minimal distance to the object is saved as the closest function together with the new configuration. If no function could be found, the object is moved to the border in the searching direction.
3. The whole object will then be transformed according to the configuration saved, or if the object is already able to be moved to the target configuration in a direct line, the program ends. This is done by connection each corner of the main object with the target area and checking those functions for intersection.

As already mentioned, this representation also is a tradeoff.
Pros:

- The configuration space does not need to be computed, as it is defined directly by the anchor points of the objects itself. The outer rim R_{mo} is considered seperate from the objects. Each non-moving object is treated just like a normal object, only that its anchorpoints values in the configurationspace along x, y and ϕ are constant.
- With the ability to get rid of functions that are not in the way, the number of search cells goes down due to the absence of "ghost planes".

Cons:

- The object transformation need a little more time as for each function the parameters need to be recalculated. This
- Rotation still needs discrete steps along the config dimensions ϕ_i, therefore more exact calculations lead to higher need in computation power.

4 Implementation

4.1 Algorithms and Functions

4.1.1 Main search loop

As proposed in 3.1.2, we create the graph while searching. This requires a certain level of interleaving beetween the search algorithm and the way we work with our objects. In this example, the functions working on the objects are *oneStep*(...) and *isValid*(...). All of the rest is needed for the search algorithm, in this case a simple A*.

```
%while target is not in rim
while(~ismember(target,R))
    %select current node from rim
    next = getNodeFromRim();
    %calculate rimnodes of current node
    for i=1:length(directions) % find next node for each search direction
        [possible_next,next_collision_set] = oneStep(next,....);
        if isValid(possible_next,riddle.b,next_collision_set) % check if node
            is valid and...
            if ~isInRim(possible_next,R) % ...not in the rim
                R=[R;possible_next]; %if so, add it
                collision_set{length(D)+1} = next_collision_set; %set his
                    collision information
                D=[D;D(next_position)+0.1]; %enter distance to predecessor
                P=[P;next]; %enter next as predecessor
                H=[H;heuristic(possible_next,target)]; %calculate heuristic
                    value
                V=[V;0]; %mark as not visited
            else                    %... already in the rim
                pn_position=find(possible_next,R); %search the node in the rim
                if(D(pn_position)>D(next_position)+0.1) % if node in rim is
                    further away as new node...
                    D(pn_position)=D(next_position)+0.1; %... update distance
```

```
20                      P(pn_position,:)=next; % and chance predecessor
21                  end %else do nothing
22              end
23      end
24  end
```

$isValid(...)$ is a simple function testing the given configuration from $oneStep(...)$ against configuration space. This is done by simply checking if the configuration point is inside any convex hulls stored in the configuration space.
The functions oneStep and getRim are now explained in detail for the representation of objects as points list and function list.

4.1.2 Implementation with point list

As described in 3.1.3 an object can be described as a list of ordered corner points. Starting from there the configuration space for each object is the set of convex hulls of its collision points with each other object. A point in the configuration space can then be described as the set of said subsets.
These points form the search nodes for the pathfinding algorithm.

The function oneStep

$[\textbf{nextNode}, \textbf{newCollSet}] = oneStep($**node, direction, collSet,riddle, jump_over**$)$ is the function that provides us with the next node in a specific direction and an updated collision set for that node. This new collision set fits to the new point in the configuration space at the position of the next node.
Parameters:

- **node**: a point in the configurationspace C used as the starting point.
- **direction**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **collSet**: i sets of $n-1$ sized sets giving the collisionsets $C_{O_i-O_j}$ per object O_i for all n objects.
- **riddle**: the original set of information for the starting riddle. Needed for recalculation of collSets.

- **jump_over**: flag for signaling if next node should be on the rim of current cell (jump_over = -1), or in the next cell (jump_over = 1).

Returns:

- **nextNode**: the next point in the configuration space from **node** along the dimension **direction**.
- **newCollSet**: the new collisionset for the objects at the configuration **nextNode**.

The fucntion can be divided in two different step types: rotation and translation.
All rotations are executed by changing the dimension **direction** depending on the sign of **direction** by one step. The size of the step is determined by the needed accuracy. The function rotateObject rotates all cornerPoints of the object around its anchor point.

```
1  ...
2      tempAdd = zeros(1,length(node)); %build mask for step in rotation
           direction
3      tempAdd(abs(direction))=sign(direction)*rotationStep; %roationstep is set
           depending on the needed accuracy
4      nextNode = node + tempAdd; % add mask to old node
5
6      for object=1:length(riddle.o)
7          riddle.o{object} = changeOneObject(nextNode((object-1)*3+1:object*3),
               riddle.o{object});    %change object according to its new rotation
8      end
9
10     for object=1:length(riddle.o) %for each object
11         temp = riddle.o;
12         temp(object) = [];
13         newCollSet{object}= getRims(riddle.o{object}.data,temp,... %generate
               new collision sets
14             length(riddle.o{object}.data),riddle.o{object}.mid);
15     end
16     return; % and return
```

On the other side stands the translation step. As our goal is to identify the next cell in the dimension **direction**, we iterate over all convex hulls stored in **collSet** for the object that needs changing and extend their borders. A jump over/to the nearest border in a given direction equals a jump into the next / to the border of the cell.
First two points are taken from the convex hull (named points) to recalculate the vector. Due to

the fact that each object only resideds in a 2 dimensional space, by solving a linear equation the crossing points on the dimension that is NOT our search dimension *direction* can be obtained, so that the point in this direction can be calculated.

```
...
        %calculate line from points
        offset = points(i,:);
        vector = points(mod(i,length(points))+1,:)-points(i,:);

        %solve for x and y points
        x =  vector\(node((object_pos-1)*3+1:(object_pos-1)*3+2) - offset) ;
...

...
        %get point on same x,y coordinate
        %check if line is parallel to searching direction
        if(vector(mod(mod(abs(direction),3),2)+1)<0.001)
            continue;
        end

        %get x to move in y direction and otherwise
        if(mod(abs(direction),3)==1)
            if(x(2,2)<0.001)
                continue;
            end
            p = offset + x(2,2)*vector;%get point on same y
        else
            if(x(1,1)<0.001)
                continue;
            end
            p = offset + x(1,1)*vector;%get point on same x
        end
```

Before we calculate this point, we check if we already reached our target cell by trying to get a non-intercepted connection from the current **node** to the target node taken from **riddle**. The flag needs to be tchecked for ALL vectors.

```
    %check if point is in same cell as target
        if(inTargetCell)
            %find out if direct way to target is possible
            temp=(node(1:2) - offset)';
            A=[vector', -node_to_target];
            sol = A\temp;
```

```
7
8               point_on_line = offset + sol(1)*vector;
9               point_on_line = node(1:2)' + sol(2)*node_to_target;
10
11              %check if lines intersect (aka way to target is free )
12              if(sol(1)>=0 && sol(1)<=1 && sol(2)>=0 && sol(2)<=1)
13                  inTargetCell = inTargetCell && false;
14              end
15          end
```

If all checks out, we calculate the vector from **node** to the crossing point on the border. If the sign of the vector along the dimension **direction** equals that of **direction** a distance is calculated and, if lower than the current minimum, stored together with a possible new node **nextNode**. This next node is choosen depending on the flag **jump_over** to either be directly infront or behind the point on the border.

```
1           %vector from node to temp point on line
2           node_to_point = p-node((object_pos-1)*3+1:(object_pos-1)*3+2);
3           %get distance to those points if direction is ok
4           if sign(node_to_point(mod(abs(direction),3)))~= sign(direction)
5               d = inf;
6           else
7               d = norm(p - node((object_pos-1)*3+1:(object_pos-1)*3+2));
8           end
9
10          %save new minimum and new point on line
11          if d < min_dist
12              min_dist = min(min_dist,d);
13              if (jump_over==1)
14              nextNode((object_pos-1)*3+1:(object_pos-1)*3+2) = p + (
                    node_to_point~=0)*0.001;
15              else
16              nextNode((object_pos-1)*3+1:(object_pos-1)*3+2) = p - (
                    node_to_point~=0)*0.001;
17              end
18          end
19
20  ...
```

Now after this loop has finished, we have found either a new minimum in the searching direction or we are inside the same cell as the target.
If we are in the same cell as the target, we just set the main object M to the targets configuration

and return.

```
if inTargetCell && object_pos==1
    nextNode(1:3) = riddle.t.mid;
    return
end
```

If we found a new minimum, the collisionsets need to be adapted. For that we recalculate the new configuration state **nextNode** of our objects with the help of *changeOneObject*(...).

```
   for object=1:length(riddle.o)
        riddle.o{object} = changeOneObject(nextNode((object-1)*3+1:object*3),
            riddle.o{object});
    end
```

Afterwards we iterate over the newly generated objects and recalculate the collision sets per object with *getRims*(...).

```
for object=1:length(riddle.o)
        temp = riddle.o;
        temp(object) = [];
        newCollSet{object}= getRims(riddle.o{object}.data,temp,...
            length(riddle.o{object}.data),riddle.o{object}.mid);
end
```

This **newCollSet** is then returned together with **nextNode**.

getRims

[**realRims**] = *getRims*(**objectPoints, rims, objectCount, mid**) calculates the convex hull for one object to all rims/objects for one configuration. This function is used to update the collision set of an object.

- **objectPoints**: the points of the object.
- **rims**: all rims/objects to which the convex hull needs to be calculated.
- **objectCount**: The number of points the object has (length of objectPoints).
- **mid**: the configuration of the object

Returns:

- **realRims**: the set of convex hulls to all rims/objects for the object with the configuration mid

The function starts by iterating over all entities in rims. As a rim is merely a fixed object, there is no need for differenciating beetween object and rim in rims.

```
realRims=cell(length(rims),1);
rimcount=1;
%for each rim in rims
for rim=rims
    rimData = rim{1}.data;
    %split up the rim in convex parts
    convexRim = [];
    start=rimData(1,:); %pick start point
    vecOld = rimData(2,:) - rimData(1,:); %set start vec
    subsetCount = 1;
    subsets{subsetCount}=rimData(1,:);
    ...
```

For each entity the change in angle beetween the vectors connecting the points is calculated, and each object is divided into its convex parts. This is done by searching along the points of the choosen entity and for each point calculating vector from the previous and the pre-previous point to the current point and comparing their angle. If the angle difference is greater zero, the point is starting a new convex subset.
These parts are stored in subset. If the entity is convex itself, subset is of the length one, holding all points of the entity in the first position.

```
...


    for  i=2:length(rimData)
        stop=rimData(mod(i,length(rimData))+1,:); %pick end point
        vec=stop-start; %get vector from start to end
        %get angle diffrence to x-axis for each two vectors
        dirStop = sign((vecOld(2)*vec(1) - vecOld(1)*vec(2)))...
            *acos( (vec(1)*vecOld(1) + vec(2)*vecOld(2))/(norm(vec)*norm(
                vecOld)));

        if dirStop<=0 %smaller means part will be concave
            subsets{subsetCount} = [subsets{subsetCount};rimData(i,:)];
        else %greater zero means new point is starting new part
            subsets{subsetCount} = [subsets{subsetCount};rimData(i,:)];
            subsetCount = subsetCount+1;
            subsets{subsetCount}=[];
            %subsets{subsetCount} = rim{1}(i,:);
```

```
18          end
19          start=rimData(mod(i-1,length(rimData))+1,:);
20          vecOld = stop - start;
21      end
22
23  ...
```

Afterwards the convex hull is calculated for each subset using the minkowski sum.

```
1   ...
2
3       subsetCount=1;
4       convexSubsets = cell(length(subsets),1);
5       masks = cell(length(subsets),1);
6       poly = cell(length(subsets),1);
7       for set = subsets %for each set in subsets
8           mSet = cell2mat(set);
9           for i=1:size(mSet,1) %we take each value
10              v=mSet(i,:);
11              %and add the points of the object as seen from the origin (
                    minkowski sum )
12                  poly{subsetCount} =...
13                      [poly{subsetCount};...
14                      (ones(objectCount,1)*(v - mid(1:2))...
15                      +objectPoints(1:objectCount,:))];
16          end
17
18          %extract the points for a convex polygon
19          if size(mSet,1)>2
20              [convexSubsets{subsetCount},masks{subsetCount}] = getConvexPolygon
                    (mSet,poly{subsetCount},objectCount);
21          else
22              convexSubsets{subsetCount} = poly{subsetCount};
23              masks{subsetCount} = ones(length(poly{subsetCount}),1);
24          end
25          subsetCount = subsetCount + 1;
26      end
27
28  ...
```

As we then hold convex hulls of parts of the entitiy, we need to put them back together. To do that a connection beetween each start and end point of a subset is made. By looking at the angles beetween all possible connections of the points from the minkowski sum, the most outer

connection will be choosen.

```
...

    if(length(subsets)==1)
        convexRim = convexSubsets{1};
    else
    for i = 1:length(subsets)-1
        %add first point of first subset
        if i==1
            convexRim = convexSubsets{i}(1,:);
        end

        stop = subsets{i};
        stop = stop(size(stop,1),:); % take last point of a set

        start = subsets{mod(i,length(subsets))+1};
        start = start(1,:); %take first point of next set
        if(stop==start)
            start = subsets{mod(i,length(subsets))+1};
            start = start(2,:); %take second point
        end

        connect = start - stop; % create vector beetween those two

        %find point to the right of connect from stop which is in a convex
        %subset.

        %create vectors for stop
        pointsStopSet = poly{i};
        pointsStop = pointsStopSet(size(pointsStopSet,1)...
            -objectCount+1:size(pointsStopSet,1),:);
        vectorStop = pointsStop - ones(size(pointsStop,1),1)*stop;
        dirStop = acos(vectorStop(:,1)./sqrt(sum(vectorStop.^2,2)))...
            - ones(size(pointsStop,1),1)*acos(connect(1)/norm(connect));
        [v,pStop] = min(dirStop); %select the one farest to the right

       ...

        pointsStartSet = poly{mod(i,length(subsets))+1};
        pointsStart = pointsStartSet(1:objectCount,:);
        vectorStart = pointsStart - ones(size(pointsStart,1),1)*start;
        dirStart = acos(vectorStart(:,1)./sqrt(sum(vectorStart.^2,2)))...
```

```
42              - ones(size(pointsStart,1),1)*acos(connect(1)/norm(connect));
43          [v,pStart] = min(dirStart); %select the one farest to the right
44
45  ...
```

Afterwards some minor adjustments take place to remove unnecessary lines and points from the hull to reduce "ghost planes". Then the finished convex hull of the object with the choosen entity is saved. After each entity in rims is checked, the new set of convex hulls is returned.

4.1.3 Implementation with function list

The implementation as function lists 3.1.4 describes the configuration space simply by the objects at their positions. After eacj step, the object that was moved will be updated and a copy of that set is stored as the new ciollision set for the configuration.

The function oneStep

[**nextNode**, **collision_set**] = *oneStep*(**node, direction, cur_collision_set,riddle**) is the function that provides us with the next node in a specific direction and an updated collision set for that node. This new collision set fits to the new point in the configuration space at the position of the next node.
Parameters:

- **node**: a point in the configurationspace C used as the starting point.
- **direction**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **curr_collision_set**: i sets of $n-1$ sized sets giving the collisionsets $C_{O_i-O_j}$ per object O_i for all n objects.
- **riddle**: the original set of information for the starting riddle. Needed for recalculation of collSets.

Returns:

- **nextNode**: the next point in the configuration space from **node** along the dimension **direction**.
- **collision_set**: the new collisionset for the objects at the configuration **nextNode**.

In comparison to the header of the other *oneStep*(...) implementation, only the parameter **jump_over** is no longer needed.
The function can again be divided into two different steps: rotation and translation.
The fucntion can be divided in two different step types: rotation and translation.
All rotations are executed by changing the dimension **direction** depending on the sign of **direction** by one step. The size of the step is determined by the needed accuracy. The function rotateFunc recalculates the function parameters after the rotation.

```
...
    tempAdd = zeros(1,length(node)); %generate mask
    tempAdd(abs(direction))=sign(direction)*rotationStep; %rotationStep
        depends on the needed accuracy
    nextNode = node + tempAdd; %adapt old node

    %rotate object to fit new configuration
    curr_collision_set{object_pos} = rotateFunc(nextNode((object_pos-1)*3+1:
        object_pos*3),curr_collision_set{object_pos});
    collision_set = curr_collision_set; % save new collision set
   return; %and return
```

On the other side stands the translation step. Due to the fact, that there are no cells, the goal is to move the object in the **direction** until it collides with either the border or another object. This is done by checking all elements in the **curr_collision_set** for collision depending on the **direction**. If none could be found the border of the **riddle** is used.
As the object to be moved is defined by **direction**, all other objects need to be checked for a possible collision. Therefore a loop iterates over all functions in all objects in **curr_collision_set**, checking if the moving object can be moved to the target without collision and calculating the configuration node if a collision exists in **direction** by calling *moveToFunc*(...)

```
...
  if(object_number==object_pos) %skip if object is moving object
        continue;
    end
    %pick object
    object = curr_collision_set{object_number};

    %pick function from object
    for function_number=1:length(object.coeff)
        func=object.coeff{function_number};
        def=object.def{function_number};
```

```
12 ...
13
14 ...
15         %% check if point is in same cell as target
16         if(inTargetCell)
17
18             %check if x is ok
19             inTargetCell = inTargetCell &&...
20                 sign(node(1)-def(1)) == sign(riddle.t.mid(1) - def(1)) &&... %
                       left border
21                 sign(node(1)-def(2)) == sign(riddle.t.mid(1) - def(2)) &&... %
                       right border
22                 (func(1)==0||(sign(node(1)-ext_x) == sign(riddle.t.mid(1) -
                       ext_x))); %quadratic function only
23
24             %check if y is ok
25             inTargetCell = inTargetCell &&...
26                 sign(node(2)-max_y) == sign(riddle.t.mid(2) - max_y) &&... %
                       upper border
27                 sign(node(2)-min_y) == sign(riddle.t.mid(2) - min_y); %lower
                       border
28         end
29
30         %% get nextNode in search direction. if function not in the way, dist
               = inf.
31         [tempNode,dist] = moveToFunction(node,direction,curr_collision_set{
               object_pos},riddle.b,func,def,object.above{function_number});
32
33         %save new minimum ( closest function in the way ) and nextNode
34         if abs(dist) < abs(min_dist)
35             min_dist = min(abs(min_dist),abs(dist))*sign(dist);
36             nextNode=tempNode;
37         end
38 ...
```

After that if a node with the closest distance was choosen the collision set is recalculated. If not $moveFunc(...)$ moves the object to the border of **riddle**. Also if the object can be moved to the target without collision, the target will be set as the next node and the function returns.

```
1 ...
2 %% build new collision set from chosen node
3 curr_collision_set{object_pos}=moveFunc(min_dist,direction,curr_collision_set{
      object_pos});
```

```
collision_set = curr_collision_set;

if inTargetCell && object_pos==1
    nextNode(1:3) = riddle.t.mid;
    return
end
```

moveFunc

[**object**] = *moveFunc*(**diff, dir, object**) recalculates the function parameters stored in **object**.coeff depending on the direction **dir** and distance **diff** the **object** needs to be moved.
Parameters:

- **diff**:the signed distance the object is moved.
- **dir**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **object**: the object which needs adaption.

Returns:

- **object**: the adapted object.

The function *moveFunc*(...) distinguishes beetween movement in x- and y-direction.
For movement along the$x$-axis the objects functions coefficients are recalculated by substraction of **diff** from x for every function in the object. After finishing iterating over the functions, the definition range is adapted and the new anchor point is set.

```
...
for function_number=1:length(object.coeff)%iterate over all functions
       func = object.coeff{function_number};
       if func(1)==0 %function is linear
           %b(x-d)+c = bx + ( -bd + c)
           func(3)=-func(2)*diff+func(3);
       else %function is quadratic
           %a(x-d)^2 + b(x-d) + c =
           %ax^2 - 2adx + ad^2 + bx - bd + c =
           %ax^2 + (-2ad + b)x + (ad^2 - bd + c)
           b = -2*func(1)*diff + func(2);
           c = func(1)*diff^2 - diff*func(2) + func(3);
           func(2)=b;
           func(3)=c;
```

```
15          end
16          object.coeff{function_number}=func; %save new coefficients
17          object.def{function_number}=object.def{function_number}+diff;% save new
                definition range
18      end
19      object.mid(1) = object.mid(1) + diff;%set new anchor point
20  ...
```

Movements alogn the y-axis are easier, as they require to only chance the constant coefficient of a function.

```
1   ...
2   for function_number=1:length(object.coeff)%iterate over all functions
3           func = object.coeff{function_number};
4           func(3) = func(3) - diff; %change constant coefficient
5           object.coeff{function_number}=func;
6       end
7       object.mid(2)=object.mid(2) - diff; %set net anchorpoint
8   ...
```

moveToFunction

[**nextNode**, **dist**] = $moveFunc$(**node, direction, object, border, func, def, above**) checks if the function **func** defined in the range **def** lies in the way of the **object** trying to move in the **direction** or if the **border** needs to be used. The function calculates the configuration point **nextNode** together with the travelled distance **dist**. Parameters:

- **node**: the current node the movement should be executed from.
- **direction**: the dimension of C to search a new node on. Signed means search backward, unsigned forward.
- **object**: the object which wants to be moved.
- **border**: the border of the riddle.
- **func**: the function which could be laying in the way.
- **def**: the definition range of the function.
- **above**:wether the object the function **func** borders lies above or below **func**

Returns:

- **nextNode**: the next configuration point after moving the **object**.
- **dist**: the distance the **object** has been moved.

The function *moveFunc*(...) iterates over all functions in **object** and checks if there exists a crossing point beetween them and the function **func**. This point is then saved together with the distance and the point with the minimal distance is returned.
First the values of the function **func** are calculated and if the movement is along the x-coordinate, compared to those of the **object** functions. If no common range exists, the function can not collide.

```
...
%calculate funcValue at ends of defenition range
funcValue_min = (def(1)^2)*func(1) + def(1)*func(2) + func(3);
funcValue_max = (def(2)^2)*func(1) + def(2)*func(2) + func(3);

for obj_function_number = 1:length(object.coeff)
    %get current function and range
    obj_func = object.coeff{obj_function_number};
    obj_def = object.def{obj_function_number};


    if(mod(abs(direction),3)==1)
        %  move along x coordinate (right/left)
        %% calculate function values
        objValue_min = (obj_def(1)^2)*obj_func(1)  + obj_def(1)*obj_func(2) +
            obj_func(3);
        objValue_max = (obj_def(2)^2)*obj_func(1)  + obj_def(2)*obj_func(2) +
            obj_func(3);

        %% get the y-range used by both function ( inner borders )
        min_y = (objValue_min>funcValue_min)*objValue_min + (objValue_min<=
            funcValue_min)*funcValue_min;
        max_y = (objValue_max<funcValue_max)*objValue_max + (objValue_max>=
            funcValue_max)*funcValue_max;

        if(min_y > max_y) % if no common range exists, jump to next function
            if(sign(direction)==-1)
                diff_min = border(1,1) - obj_def(1);
                diff_max = border(1,1) - obj_def(2);
            else
                diff_min = border(3,1) - obj_def(2);
                diff_max = border(3,1) - obj_def(1);
```

```
29              end
```

If a common range exists, the x values at the edges of the overlapping y values are calculated.

```
1  ...
2
3               %% calculate x values to min_y and max_y
4               %linear functions
5               if(func(1)==0) %distinguish beetween linear and quadratic
6                   if(func(2)==0)
7                       funcX_min = def(1);
8                       funcX_max = def(2);
9                   else
10                      funcX_min = (min_y - func(3))/func(2);
11                      funcX_max = (max_y - func(3))/func(2);
12                  end
13              else%quadratic functions
14                  [x1_min,x2_min]=solve(poly2sym([func(1),func(2),func(3)-min_y
                        ]));
15                  [x1_max,x2_max]=solve(poly2sym([func(1),func(2),func(3)-max_y
                        ]));
16
17                  %choose x nearest to the object using search direction sign
18                  if(sign(direction)==-1)
19                      funcX_min = (x1_min>x2_min)*x1_min +(x1_min<=x2_min)*
                            x2_min;
20                      funcX_max = (x1_max>x2_max)*x1_max +(x1_max<=x2_max)*
                            x2_max;
21                  else
22                      funcX_min = (x1_min<x2_min)*x1_min +(x1_min>=x2_min)*
                            x2_min;
23                      funcX_max = (x1_max<x2_max)*x1_max +(x1_max>=x2_max)*
                            x2_max;
24                  end
25              end
26 ...
```

The same calcuations are also made for the current **object** function to calculate $objX_min$ and $objX_max$. The difference beetween the **object** function and the **func** values are used to determine the distance the **object** needs to be moved towards the function **func**. The minimum distance is then saved together with the new configuration point.

```
1  ...
```

```
2
3             diff_min = funcX_min - objX_min; %calculate distances
4             diff_max = funcX_max - objX_max;
5         end
6
7         if(diff_min==0 || diff_max==0) %check if object lies besides function
8             if(sign(direction)==sign(above)) %if object is moving away from
                  function
9                 continue;
10            end
11        end
12
13        %% choose smaller distance...
14        if(abs(diff_min)<abs(diff_max)&&abs(diff_min)<abs(dist))
15            nextNode(abs(direction)) = node(abs(direction)) + diff_min; %...
                  and nextNode;
16            dist = diff_min;
17        elseif(abs(diff_max)<abs(dist))
18            nextNode(abs(direction)) = node(abs(direction)) + diff_max; %...
                  and nextNode;
19            dist = diff_max;
20        end
21    end
22 ...
```

If the movement is along the y-coordinate a simple look at the defenition ranges of the **object** functions rules out all functions that are not in the way. If that is the case, the border is taken as the nearest function.

```
1 ...
2
3 % move along y coordinate ( up/down)
4 else
5         %% get the x-range used by both function ( inner borders )
6         min_y = (def(1)>obj_def(1))*def(1) + (def(1)<=obj_def(1))*obj_def(1);
7         max_y = (def(2)<=obj_def(2))*def(2) + (def(2)>obj_def(2))*obj_def(2);
8
9         %% check if function lies in the way
10        if(min_y > max_y) % if no common range exists, take borders
11            %% calculate function values
12            objValue_min = (obj_def(1)^2)*obj_func(1) + obj_def(1)*obj_func(2)
                   + obj_func(3);
```

```
13              objValue_max = (obj_def(2)^2)*obj_func(1) + obj_def(2)*obj_func(2)
                     + obj_func(3);
14
15      ...
```

If there is a common range, the $objValue$s are calculatet with the common function range and the distance is stored together with the new configuration point.

```
1           elseif(min_y<=max_y) %start to find collision
2
3               %% calculate function values
4               objValue_min = (min_y^2)*obj_func(1) + min_y*obj_func(2) +
                    obj_func(3);
5               objValue_max = (max_y^2)*obj_func(1) + max_y*obj_func(2) +
                    obj_func(3);
6       ...
7
8       ...
9               diff_min = funcValue_min - objValue_min; %calculate distances
10              diff_max = funcValue_max - objValue_max;
11
12              if(diff_min==0 || diff_max==0) %check if object lies besides
                    function
13                  if(sign(direction)==sign(above)) %if object is moving away
                        from function
14                      continue;
15                  end
16              end
17          end
18
19          %% choose smaller distance...
20          if(abs(diff_min)<abs(diff_max) && abs(diff_min)<abs(dist))
21              nextNode(abs(direction)) = node(abs(direction)) + diff_min; %...
                    and nextNode;
22              dist = diff_min;
23          elseif(abs(diff_max)<abs(dist))
24              nextNode(abs(direction)) = node(abs(direction)) + diff_max; %...
                    and nextNode;
25              dist = diff_max;
26          end
27
28  end
```

5 Results

If we apply the finished Algorithm to various test data sets, we can get an impression of what the program is capable of.

5.1 Small riddles

Small riddles containing at most two objects beside the main object.

5.2 Medium riddles

Medium riddles containing at most 10 objects beside the main object.

5.3 Big riddles

Big riddles with more than 10 objects beside the main object.

5.4 Interpretation of testdata

Compare the different timeframes to the amount of object, placement and, if solved, the way the algorithm took.

6 Discussion

Through applying a local approach to builing the configuration space, the riddles are starting to become solvable in an acceptable timeframe. Still, the time it takes to solve a riddle can vary strong depending on the number of objects, their start configuration and their shape. Furthermore there are still some unsolved problems in general and depending on the used implementation

6.1 General Problems

6.1.1 Infinite rotation

GRAPHIC WITH A HUGE SPIRAL REQUIERING THE OBJECT TO ROTATE OVER 360° TO SOLVE

As earch object can be rotated a total 360° to get back to its starting position, the first thing that comes into mind is that rotations are infinite. The problem with this viewpoint is, that the configuration space would also be infinite in each rotation direction per object. This makes it impossible to calculate the configuration space beforehand, because to do so, each possible combination of objects needs to be checked. But with multiple infinite axis, this would be an infinite number and calculations would never come to an end.

The solution to this problem lies in the local search method applied in the algorithm. It views the rotation axis as a stack of configuration hyperplanes per object. If you rotate an arbitrary object o, you simply move up/down in the stack of hyperplanes for said object.

The downside to that is, that in order to build a stack, there is the need to define a stepsize in which to move up and down the stack and change the rotation. This leads to another scale factor which needs to be set so that adequate accuracy can be achieved. One has to choose either less accuracy for less calculations or more calculations for higher accuracy.

6.2 Problems with point list implementation

6.2.1 Invisible plains

An invisible plain is a plain extending from the vector of one object beetween two of its points. This plain splits the search space in two parts. Due to the fact, that we use this effect to split the space in cells, which we use for nodes in our search graph, changes to these plains affect our graph.
GRAPHIC WITH TWO OBJECTS PLAINS INTERFEARING
If we take a look at this example, we can see how the object o1 dictates the cell border for object o2 on the left. If we move o2 over the border, it now takes over the cell border for object o1. Without a given heuristic which object is better to be moved to the right, this will end in alternating movements for both objects.
But if we choose o1 to be the main object, which has a target on the right, the heuristic would need to tell our search algorithm that the point with o1 beeing right of o2 is the better point, and search from there on further, meaning o1 beeing moved to the right cell border again.
GRAPHIC WITH O1 AS MAIN OBJECT???

6.3 Problems with function list implementation

6.3.1 Vertical lines

Due to the nature of a function, vertical lines can not be represented. This leads to the rule that all objects in the riddle can not have a vertical line even after rotating. To still be able to draw an object with a vertical line, a function is defined with a very high gradient and setting the starting point at the x-coordinate of the vertical line. This leads to a near vertical line, that is still a function, and as such can be used in the calculations.
The drawback is clearly the loss of preciseness. How strong this loss is depends on the initial gradiant one gives the function. A seemingly good idea would be to use the max value of the variables used in calculations (e.g. double $= 1.7977 \cdot 10^3 08$). But if we do this without beeing sure that this value will never be used in a way, such that the solution of a calculation could be greater than the gradiant, we would get a overflow.
Thus, a value needs to be found that leaves enough room for calculations and still keeps the precisionloss as small as possible. The exakt number depends on the system the algorithm is

used in. Also one needs to cap the gradient after each rotation, so that it does not exceed the given max value.

List of Figures

List of Tables

bibliography

[1] Chacon, S. *Pro Git*, July 2009.

[2] MathWorks, Inc. *Matlab Language Reference Manual*.