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, July 12, 2014

Assertion

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

Lübeck, July 12, 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:

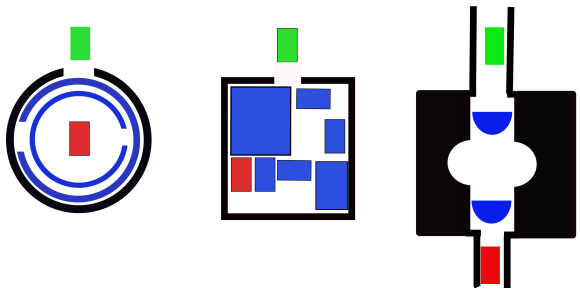

Figure 1.1: *riddle 1 with circles , riddle 2 with boxes , riddle 3 with 2 half-circles*

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:

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

Next will be the calculation of the angle beetween two vectors 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:

$$\phi = anglebeetweenvectors formula$$

2.2.2 convex hull

Next is the calculation of the convex hull of an object for another object. Lets take object A from before and another object B defined by:

$$B = (0, 0; 1, 0; 1, 2; 0, 2)$$

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 after the formula

$$hull = formula$$

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.

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

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.

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

2.2.3 linear programming

Following the convex hull, as a way to actually compute if a collision took place or not, we use linear programming. For that we represent the convex hull of object A to object B $Hull_{A.B}$ with its vector representation.
Now we take the center of A and check if its inside the convex hull.

$$A(1,:) + A_{vec}(1,:)$$

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.

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 djikstra.

```
%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
```

4.1.2 The function oneStep

[**nextNode**, **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 equals a new hyperplane 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.
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 we take two points from the convex hull (named points) and recalculate the vector. Due to the fact that each object only resideds in a 2 dimensional space, by solving a linear equation we get the crossing points on on the dimension that is NOT our search dimension *direction*, so that we can calculate the point in this direction.

```
...
        %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)';
```

```
5               A=[vector', -node_to_target];
6               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**.

4.1.3 isValid

4.1.4 Helper functions

Providing a better insight into the code, some of the most important helper functions are explained in detail. For all other functions, the code manual will be used as a reference.

getRims

changeOneObject

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

6.1 Problems and Solutions

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.

6.1.1 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???

6.1.2 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.

List of Figures

List of Tables

bibliography

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

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