Navigation using general voronoi diagrams [HW2]

Special Topics in Robotics

Jakub Tomasek

November 12, 2013

Contents

1 Problem specification

Generally, the task was to implement general voronoi graph (GVG) navigation in an arbitrary environment.

For the assignment, I used the environment from the previous homework; see Fig. 1.1. I also opted for the sensor based approach for determination of distance to the obstacle.

2 Solution

For the solution, I used Matlab. I represented the map as a matrix/image 706×706 pixels with value 1 representing free space and value 2 representing obstacle. The task was in general divided into three parts:

1. generation of distance map and determination of equidistant points
2. generation of nodal points and edges,

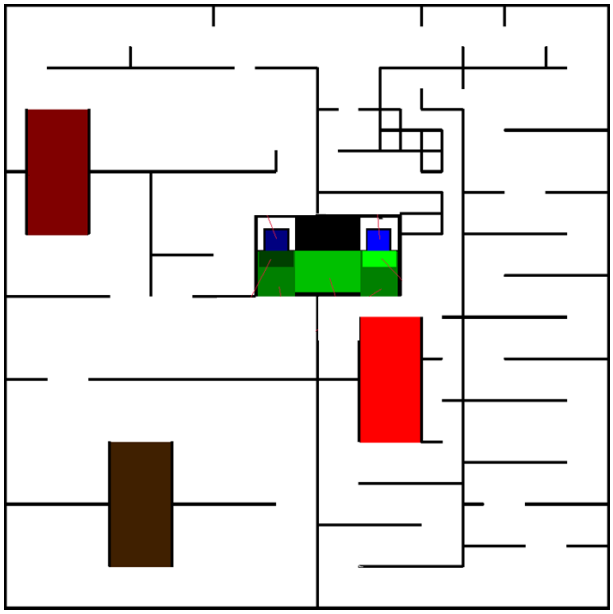


Figure 1.1: Map for path planning

3. robot exploration.

All three subtasks are explained separately in the following sections.

2.1 Equidistant points

In GVG navigation, the robot moves in between obstacles on equidistant curves. To determine and represent these curves, I had to find out which points in the given map belong to GVG, i.e. which points in the free space have equal distance from two obstacles.

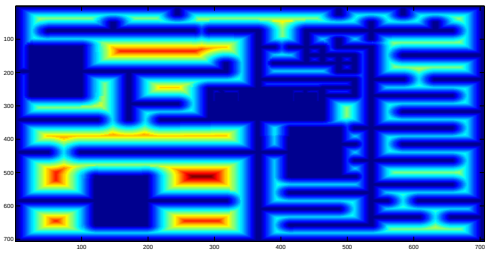
I determined the distance to the nearest obstacle for each pixel in the free space. Opting for the sensor based approach to determine the distance I could use the simple algorithm used and described in previous homework. Fig. 2.1a shows the result. Although the solution is rather non-optimal (it took almost half a hour to generate the distance map for all the map) there was no reason to seek better solution since in reality, these data would be straightly fetched from sensors.

Similarly to the real world the angles were discrete; number of angles was from 4 to 50 depending on the error. Hence the distance map wasn't perfectly smooth; this had negative effects in the following part.

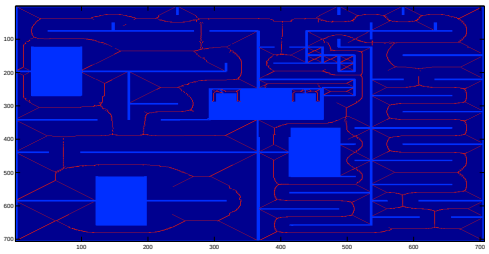
Next task was to determine the GVG points. Using internal Matlab functions, I used filter to "highlight" edges and then found local maxima and edges of plateaus at each row and column of the distance map using integrated *findpeaks* function. This yielded set of points organized in rows; see Fig. 2.1b.

2.2 Nodal points and edges

The most crucial task was to represent the generated points in a structure so that it might have been used for navigation and space exploration by the robot.



(a) Distance map



(b) Equidistant points

Figure 2.1: (a) shows the distance map where red is the farthest and dark blue is the obstacle, i.e. zero distance. (b) then displays the generated equidistant points, i.e. local minima.

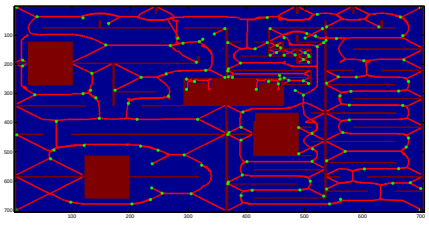


Figure 2.2: Partial success by reordering points into contours.

At first I spend a host of time by trying to order points using Matlab's internal geometrical function. This solution wasn't viable, although I had a partial success; see Fig. 2.2. Yet, I wasn't able to represent the edges in a way that I would be able to determine the nodal points robustly, that is also obvious from the image.

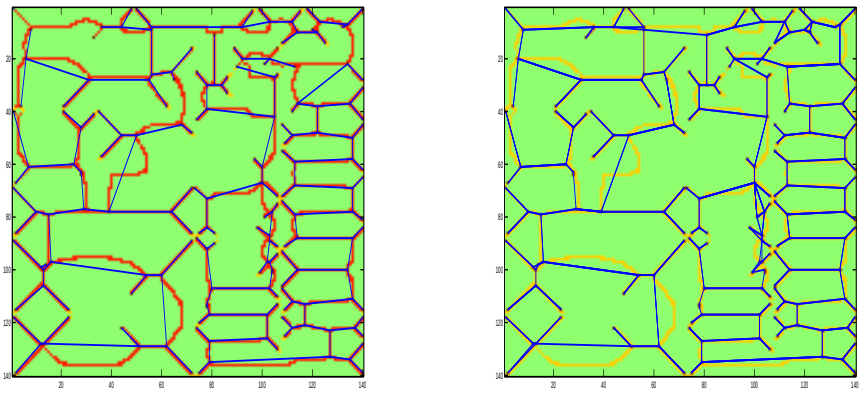
Thus I had to sort the points myself. To sort the points I used simple depth first search (DFS) algorithm. In fact, by slight alteration of DFS I could merge this task with the third subtask: exploration; I could also straightly simulate the space exploration by robot. All this is thus described in the next section.

2.3 Robot exploration

At first I had to create more coarse grid to lose the "curve error" caused by the discrete angles. Using the input the data from the first part I created grid with 5px width of each element. To simplify the task I also removed the GVG points from the "closed" and inaccessible rooms. The output was more coarse GVG; see for example Fig. 2.3a.

The robot starts to explore the area from the closest GVG point to the start location. The algorithm is classical DFS, i.e. we expand every point the robot reaches and add the expanded points at the beginning of the open list.

The difference of the exploration algorithm is that the robot cannot teleport to the next point from open list. Thus I integrated something I called travel list; a list which is very similar to the close list. Point is added to this list when it is expanded as well (i.e. when the robot reaches the point). But, the travel-list points can be used to travel in reverse direction when the robot reach a blind spot. Points from this list are popped out until the robot finds the point on top of the open list (= rollback).



(a) DFS (b) BFS

Figure 2.3: Figure show generated edges using (a) while exploring using DFS and (b) second iteration using separate BFS. Results of BFS is much better.

Other specific about the algorithm is the expand function. Although I use 8-way connectivity, I at first expand 4-way connectivity points and only then the rest. The reason is not to leave single points creating tilted line behind.

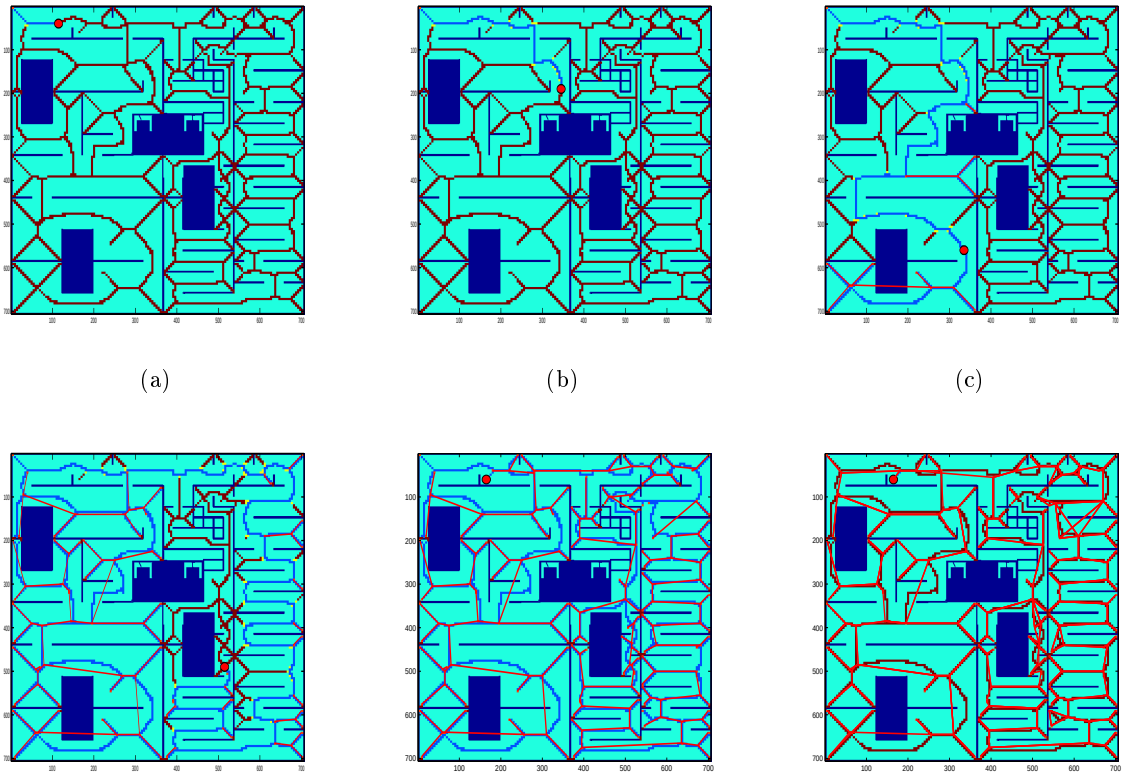
A task itself is determining whether point is a meet/blind point (= nodal points) or just an ordinary point. I came up with two rules carried out while exploring the map. Firstly, when the robot reaches a point with no neighbors it is marked nodal. Further, nodal points are also marked when the robot returns from a blind spot to the next item on the open list. Yet, we must distinguish lonely points from points where the GVG really branches. The algorithm makes the decision after expanding few more points; this confirms that GVG in the points really branches.

While rolling back, data about nodal points are collected. From these data edges between nodal points are determined. This worked well, only I didn't figure out a way to robustly finish cycles in the created graph when using DFS; see Fig. 2.3a. Thus I additionally implemented a BFS algorithm which is executed after the initial exploration is finished and nodal points are determined. For each nodal point it finds all other reachable nodal points. The output is graph induced by the nodal points; see 2.3b. This can be used for navigation in our environment.

I captured video from the robot exploration, please see http://youtu.be/dt-euA-x5J8 . Fig. 2.4 captures several frames from the exploration. Finally, commented and explained source codes of the main functions are included in the appendix.

3 Conclusion

In this homework I implemented navigation for robot using general voronoi diagrams method. My implementation should be usable for any given map in image format where the free space is white. Output is GVG map, ordered GVG points, nodal points, and weighted edges between nodal points. This can be easily used as a map to find the shortest distance between any start and any goal.



(a) (b) (c)

(d) (e) (f)

Figure 2.4: Frames from the robot exploration. I added determined edges at the stage. Dark blue are obstacles, light blue are points in close list, yellow are points in open list, magneta are the nodal points, and dark red are unexplored points. Robot is the red circle.

I also simulated the exploration using map from previous homework. The simulation is captured in video.

The weak spot of my approach is in the generation of the equidistant points which firstly takes lots of time and secondly redundant points are considered to belong to GVG. To improve that I would rather implement brushfire algorithm next time.

4 Appendix: source codes

4.1 Obstacle distance

Function getDistance is used for each point to determine the distance to the nearest obstacle.

```
%find the distance for a point
% inputs:
%     -surroundings:surroundings visible to robot,
%     -N:number of angles
%     -v:visibility
function dist=getDistance(surroundings,N,v)
    distances=zeros(N,1);
    minDistance=Inf;
    %at every step we rotate image and get line at zero degree
    for k=0:N-1
        rotated=imrotate(surroundings,360/N*k);
        c=round(size(rotated)/2);
        line=rotated(c(2),c(1)+1:c(1)+v);
        l=1;
        while line(l)==1
            l=l+1;
            if(l==v)
                %distances(k)=Inf;
                break;
            end
        end
        if(l<minDistance)
            minDistance=l;
        end
     dist=minDistance;
end
```

4.2 Robot exploration

The main procedure:

```
start=[1;1];
openList=nearestPoint(start,points2); %DFS open list
closeList=[]; %DFS close list
travelList=[]; %similar to close list but here robot can return
points2(:,1)=[];

nPoints=openList; %nodal points
lastnPoint=0;  % counter for nodal point
nPointCand=[0;0]; %candidate for nodal point

previousPoint=[0;0];
```

```
edges=[];
while size(openList,2)>0
    %until we haven't explored every point of the path

   %we pop first element from openList;
   point=openList(:,1);
   openList(:,1)=[];

   %we expand point and add neighborhood
   neighbors=expand(point,toExplore,closeList,map);
   closeList=[point closeList];
   travelList=[point travelList];


   for l=1:size(neighbors,2)
        p=neighbors(:,l);
        if(ismember(p',openList','rows'))
            %in case we have it in openlist we remove it to put it at the
            %beggining
            for i=1:size(openList,2)
               if(openList(1,i)==p(1) & openList(2,i)==p(2))
                   openList(:,i)=[];
                   break;
               end
            end
        end
        openList=[p openList]; %DFS add to the beggining
    end

   dist=norm(point-previousPoint);
   if((size(neighbors,2)==0 & dist<2))
        %in case we reached blind spot ... we know this must be nodal
          point
      nPoints=[nPoints point];
        %and we roll back to the next element from the open list
        [travelList,nPointCand,critPoint,e]=returnToExplore(openList,
          travelList,closeList,point,nPoints,map,toExplore,point);
      %while rolling back we look for nodal points so we can create edges
       edges=[edges e];
      lastnPoint=1;
   else
       if(lastnPoint>0)
          lastnPoint=lastnPoint+1;
       end
       %in case we returned where the point is alone
       if((lastnPoint==2 | lastnPoint==3) & size(neighbors,2)==0)
           lastnPoint=0;
       %we continue to travel
            [travelList,nPointCand,critPoint,e]=returnToExplore(openList,
              travelList,closeList,point,nPoints,map,toExplore,critPoint)
              ;
           edges=[edges e];
           lastnPoint=1;
       end
       if(lastnPoint>3)
        %if the branch is longer then three .. we can add this point to
          the nodal points
          nPoints=[nPoints nPointCand];
          lastnPoint=0;
          edges=[edges [critPoint; nPointCand]];
          critPoint=nPointCand;
```

```
            %we also add it to the travel list so it was noticed when
              rolling back to create an edge
            travelList=[travelList(:,1:3) nPointCand travelList(:,4:end)];
        end
    end

    %plot results, record video etc.
    displayProcess(openList,closeList,point,nPoints,map,toExplore);
    previousPoint=point;
end
```

Expand function:

```
function sur=expand(point,toExplore,closeList,map)
    n=size(toExplore,2);

    %determine the neighboring points
    y=point(2)-1:point(2)+1;
    dy=[];
    x=point(1)-1:point(1)+1;
    dx=[];
    %we don't search points out of range
    for k=1:3
        y(k);
        if((y(k)<=0)|(y(k)>n))
           dy=[dy k];
        end
    end
    y(dy)=[];
    for k=1:3
        if((x(k)<=0)|(x(k)>n))
            dx=[dx k];
        end
    end
    x(dx)=[];

    neighborhood=toExplore(y,x);
    neighbors=[];


    %we find all neighbors

    for k=1:size(neighborhood,2)
        for l=1:size(neighborhood,1)
            if(~(x(k)==point(1) & y(l)==point(2)) & neighborhood(l,k)==1)
                if(norm(point-[x(k);y(l)])>1)
                    neighbors=[neighbors [x(k);y(l)]];
                else
                    neighbors=[[x(k);y(l)] neighbors ];
                end
            end
        end
    end
    %we determine whether the member is in the closed list
    % if yes we skip
    delete=[];
    for k=1:size(neighbors,2)
        p=neighbors(:,k);
        if(size(closeList,2)>0)
            if(ismember(p',closeList','rows'))
            delete=[k,delete];
```

```
            end
        end
        %i must check that there is no obstacle between the point and
        %the neighbors
        if(~obstacle(point*5+[2;2],p*5+[2;2],map))
            delete=[k,delete];
        end

    end
    neighbors(:,delete)=[];
    sur=fliplr(neighbors);
end
```

4.3 BFS to determine edges

```
%% find edges bfs
edges=[];
%for each nodal point
for k=1:size(nPoints,2)
    point=nPoints(:,k);
    openList=[point];
    closeList=[];
    while(size(openList,2)>0)
        p=openList(:,1);
        openList(:,1)=[];
        closeList=[p closeList];
        neighbors=expand(p,toExplore,closeList,map);

        flag=1;
        %determine if one of the expanded is nodal point
        for l=1:size(neighbors,2)
            o=neighbors(:,l);
           if(ismember(o',nPoints','rows'))
               flag=0;
               break;
           end
        end

        if(flag)

            for l=1:size(neighbors,2)
                o=neighbors(:,l);

                if(~ismember(o',openList','rows'))
                    %BFS ... add to the end of openlist
                    %only in case it is not on the open list already
                    openList=[openList o];
                end
            end
        else

            edges=[edges [point;neighbors(:,l)]];
        end


    end


end
```