Cyril Roussillon

Serge Belinski

CS7630 – AUTONOMOUS ROBOTICS

Final Project Report

Georgia Institute of Technology

Spring 2008

1

Theoretical approach

1 Problem statement

Introduction

Many applications of autonomous robotics involve robot motion. For these applications (e.g. exploration, performing tasks in hard conditions, like in a very cold/hot environment, search for survivors in a cataclysm, objects delivery …) having good path-finding abilities is of primary importance. In particular, an emphasis should be put on obstacle avoidance. Obstacles may as well be expected or not. This is the reason why there are at least two valid approaches to solve those kinds of problems that come to one's mind. A natural approach may be to make robots will navigate and avoid obstacles using reactive behavior. This is fast, but not always sufficient to find its path. Deliberative planners are more efficient to find solutions, and can use a priori knowledge, but they are more sensitive to uncertain sensing.

So it seems more interesting to have a robot able to use both approaches and that will choose the most appropriate one depending on the situation. This is the reason why we wanted to be involved in a project with a hybrid architecture.

Our Goal

Robots begin to have pretty good abilities to simultaneously map and localize themselves in a building. However this is not always necessary of even desirable: it is not difficult to obtain a priori maps of buildings (e.g. from blueprints), it is more reliable to some extent, and it allows giving labels that represent shared knowledge between the robot and the users.

So our goal is to create a robot that, once provided an *a priori* map of a building, is able to navigate to reach any accessible room in the building, avoiding efficiently static and dynamic obstacles.

This can have practical applications, for instance a mail delivery robot in a given building, to deliver mail at people's office. Such a robot should obviously be able to localize itself with a more reliable method than only dead-reckoning if it has to run a long time or over long distances. The obstacle avoidance will be more efficient (faster and more reliable) using reactive behaviors with unanticipated and dynamic obstacles. However, a complete building is a true labyrinth for a purely reactive robot, and there are cases in which it will be unable to find its way without *a priori* knowledge of the world (deliberative approach). This is the reason why hybrid reactive-deliberative architecture is best suited to solve the problem of a mail-delivery robot in a huge building.

2 Proposed solution and justification

Intended solution

The idea to achieve our objective is to build a hybrid architecture combining the local path planner Vector Field Histogram (VFH) with the global path planner A*, using a ring of ultrasonic sensors and an *a priori* map of the building. This combination will allow us to have a robot that is robust to unexpected changes in the environment: once the planner found a way to the goal, provided a given map, the robot will start to navigate towards the goal. That alone could allow the robot to reach his goal. Nevertheless, there can be unexpected changes in the environment that will affect the map, so the map with which the robot was initially provided might become wrong. Moreover, it is difficult and inefficient to take into account uncertainties on the map using A* only. This is the reason why our robot is using VFH: that allows it to efficiently avoid unexpected obstacles, and find its way to the goal even with unexpected changes in the map. Our robot is also updating its map when it sees unexpected obstacles.

VFH is more powerful than simple reactive behaviours, and is able to avoid almost all traps inside a room (such as two couches positioned with a 90 degrees angle). However it is a local path planner (works with a limited active window), so it won't be able to always find its way in complex corridors.

On the contrary, A* with a building *a priori* map is able to find an optimal path between rooms, but cannot help the robot efficiently with all kind of static and dynamic obstacles (furniture, objects, people), because the a priori map doesn't contain this information.

Consequently the two methods are complementary. A* can give intermediary targets to VFH for getting out from rooms and navigating between rooms. In fact, Borenstein, the creator of VFH, points out in "The Vector Field Histogram – Fast Obstacle-Avoidance for Mobile Robots", that there are cases where VFH gets stuck in too complex environments. He claims that usual "cyclic behaviours" are too sub-optimal, and suggests a hybrid architecture that calls (with a postponing policy) a global path planner on the whole grid map, when trap situations are detected. Our approach is actually based on that suggestion, but we want to use an a priori map. This is justified by the facts that:

1. Even if the necessity of an a priori map can be seen as a limitation, it still has practical applications (navigating efficiently in a large known building).
2. It is coherent with the goal of efficiency and optimality (the grid map only contains limited information, and wouldn't prevent from exploring dead ends).
3. Aligning incrementally the grid map with a known a priori map is easier than dealing with the cycle closing problem if we want to create a coherent global grid map, and fits better to the time constraints of this project.
4. Stipulating the target on an a priori map makes more sense than stipulating a point on the grid map which is supposed to be unknown at first.

Repartition of work

During this project, we worked together to design the global architecture and the adaptations on A* and VFH that we were going to do. Then Serge implemented the A* part and Cyril implemented the VFH part.

3 Algorithm description

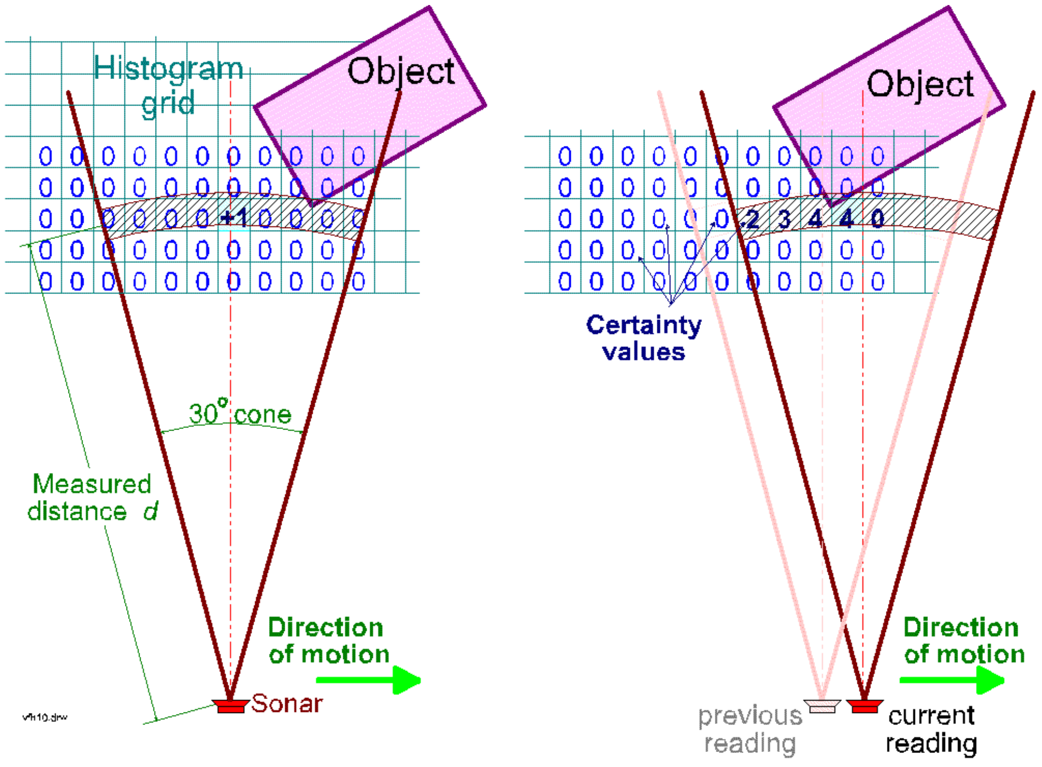
1 Local obstacle avoidance: Vector Field Histograms

References : Borenstein, J. and Koren, Y., 1991, "The Vector Field Histogram – Fast Obstacle-Avoidance for Mobile Robots." IEEE Journal of Robotics and Automation, Vol. 7, No. 3., June 1991, pp. 278-288.

1.3.1.1 Justification

Several techniques of obstacle avoidance have been developed, but all have shortcomings. For instance edge detection methods are too sensitive to sensor accuracy, certainty grids methods are too slow, potential field methods can be unstable.

Borenstein and Koren proposed the use of histogram grid instead of certainty grids, in order to have a faster processing. Instead of applying a probabilistic function, a sonar reading only votes for one cell, and the accumulation of fast readings creates a certainty map (next figure).



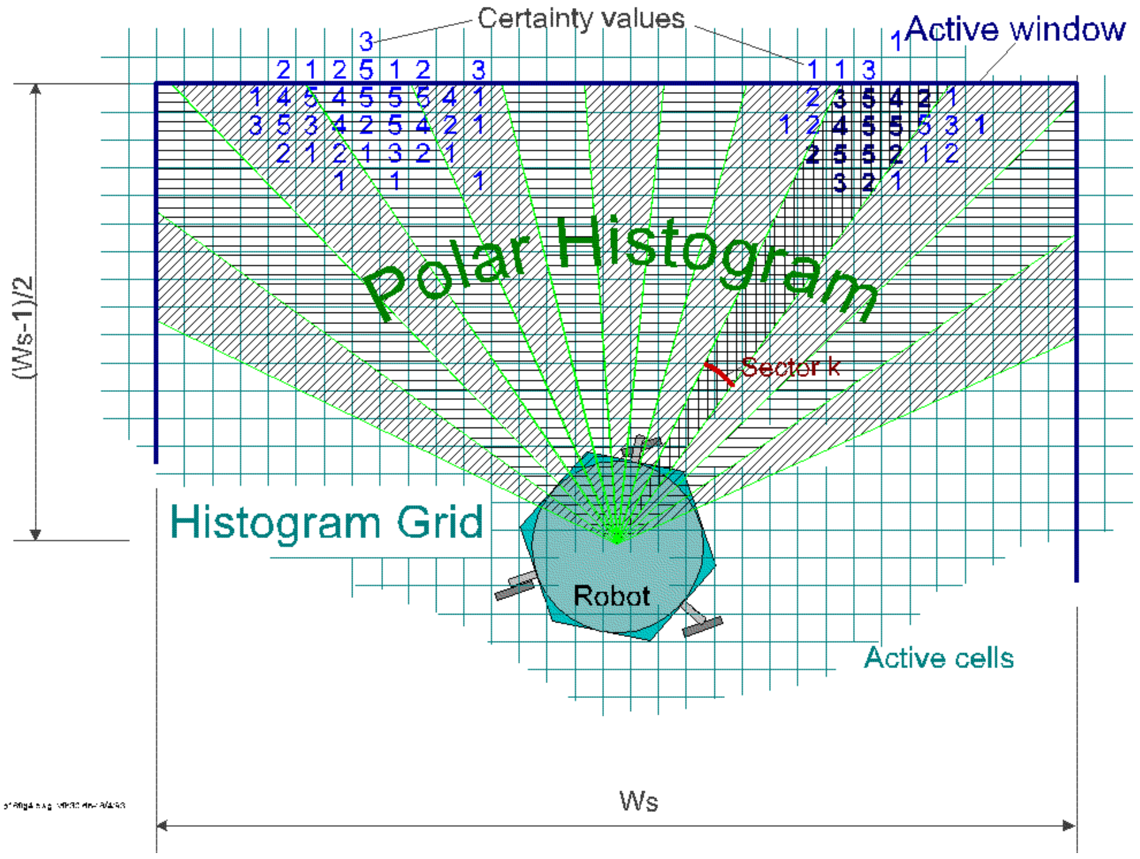
They tried to combine the histogram grid with potential fields in VFF (Virtual Force Field), but had to face unstability problems, in particular in corridors. VFH was designed to overcome this difficulty.

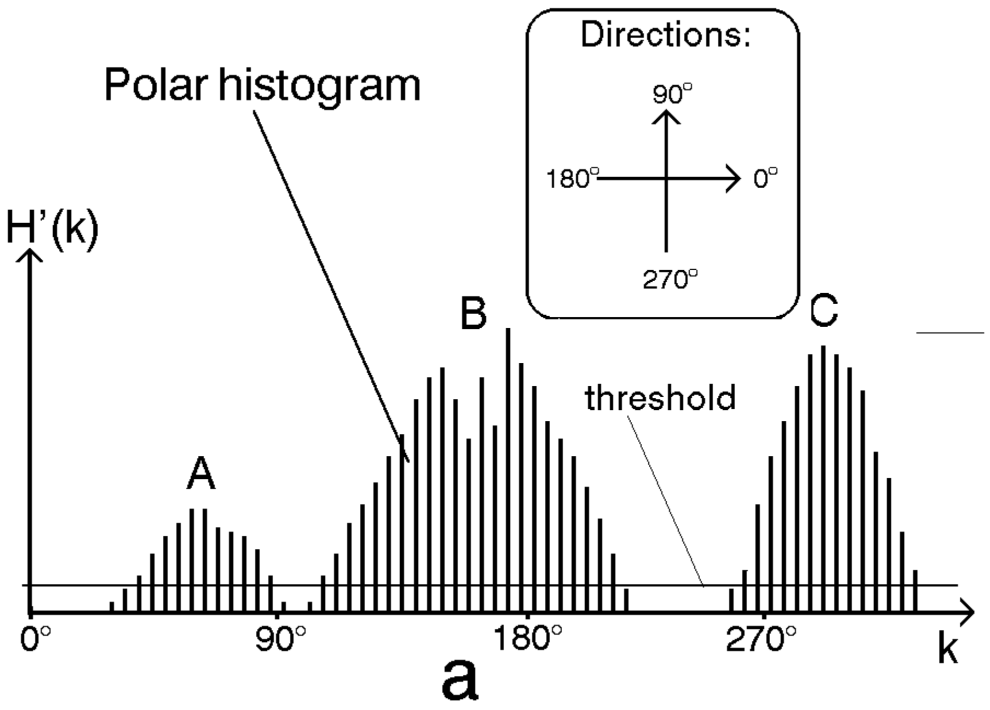
1.3.1.2 Algorithm description

VFH works with an active window centered around the robot. Then a one dimensional polar histogram is built with the values of the cells inside the active window. Each cell of the active window votes for the polar sector it belongs to :

$$m_{ij} = \left(c_{ij}^*\right)^2 \cdot (d_{\max} - d_{ij})$$

Where c_{ij}^* is the value of the cell, d_{ij} its distance to the robot, and d_{max} the maximum distance of a cell in the active window to the robot. The next figures illustrate polar histograms.





Then the polar histogram is smoothed, and compared to a threshold. Parts which are below the threshold are called valleys. The valley which is the closest from the direction of the goal is chosen. If the valley is large, an angular margin is kept from the border of the valley to give the control direction, and if this margin cannot be kept the robot drives in the middle of the valley.

1.3.1.3 Adaptations made to the algorithm

As presented by the authors, VFH cannot deal with dynamic objects, because once some histogram cells have increased, it never disappears. Nevertheless when a sonar outputs a range, it not only tell us something about the fact that there is probably an obstacle at that distance, but also about the fact that there is probably no obstacle between the robot and this distance. So we added a negative vote in this case. This allows to deal with dynamic objects as shown in demonstrations, because it can remove the obstacles that have disappeared. It is also a way to correct the map when the robot gets closer and has more reliable values.

If the robot is still, some cells will see their value increase linearly over time, and get very high values. However this is not fully justified by the high certainty that there is an obstacle at this position. When there are several reading the certainty is high, but then it only depends on how long the obstacle has been exposed to the sonar. As a result this obstacles will become very repulsive, even far away. That's why we have limited the maximum value in the histogram. Moreover it is then easier to remove the obstacle if it disappear.

We also reduced sensitivity to obstacles when the robot is getting close to its final goal, because sometimes it was preventing it from reaching the goal.

2 Global planning: A*

1.3.2.1 Justification

As we only need to find an optimal path in a known map, A* is sufficient, even though improvements such as D* (Dynamic A*) are still interesting in robotics to take into account inherent unpredictability that remains.

1.3.2.2 Description of the algorithm

A* is a graph best-first optimal path search, that uses a heuristic for estimating the distance between two points. A* is optimal only if the heuristic is a lower bound of the distance. In that case, the heuristic is said to be admissible.

The traditional A* algorithm works as described below.

Assume we have a graph modeling the environment of our robot, a start node and a goal node.

The algorithm is initialized on the node of the graph that corresponds to the initial position of the robot on the map. We place this node in a list that we will call "closed list", and it is the "current node". We initialize a list called "open list" as empty list. Further, this list will store all the nodes that can be reached from the starting node.

On the step N of the algorithm, we will iterate as follows:

We consider the “current node” and we look at his immediate neighbors on the map (graph). If a neighbor is an obstacle, we don’t consider it as a neighbor anymore, as well as if it is in the “closed list” (if the node is already in the closed list, it means that this node has already been considered and that no optimal path was found from this node to the goal). If the neighbor is already in the “open list”, if its evaluation is better than it was before (i.e. when it was accessed from another parent node), then we need to change its parent to the current node. Else, the neighbor is not is the “open list” and we add it to the “open list”, specifying that its parent is the current node.

After updating the “open list”, we look for the best node in it: if the “open list” is empty, then there is no solution to our path-planning problem (it means that we tried all the possible neighbors of all the possible nodes and no node led to the goal), else, we remove the best node of the “open list” and we put it in the “closed list”.

Eventually, we iterate with the last node of the “closed list” as current node (step N+1).

If we never reach the case where there is no solution, then the algorithm will find a path that will be given by considering all the parents of the goal. In each step, taking the best node of the open list guarantees that we have the shortest path to the goal at the end.

1.3.2.3 Adaptations made to the algorithm

In order to use A*, we have to determine which heuristic to use, and how to transform our obstacle map into a graph. We have chosen to use a grid map for reasons of homogeneity with VFH, and because it is easy to build and use.

In path finding problems there is an obvious admissible heuristic, which is the Euclidian distance. As the straight line is the shortest path from one point to another, the Euclidian distance is necessarily a lower bound of the actual path between two points.

Regarding the translation of the obstacle grid map into a graph, the usual way is to consider cells as vertices, and the 8 neighbors of a cell as neighbor vertices of the corresponding vertex.

However there are three problems with this approach for our problem, which can be formally described as the problem of finding a *continuous* path in a *discrete* map (e.g. grid map):

1. The path obtained at pixel level has no guarantee to be optimal for the continuous problem, because the distances are different and not equivalent. Depending on the case it can give the same result (e.g. a translation of vector (0; 2) has length 2 for both problems), of different result (e.g. a translation of vector (1; 2) has a length 1+sqrt(2)=2.14, whereas the continuous length is sqrt(5)=2.23). So it is impossible to choose at pixel level between two paths that have the same pixel level length, but different continuous lengths.
2. A* can become pretty slow, and above all very memory-consuming, with large grids (large map or small cells), though this problem can be solved with IDA* (Iterative Depth A*).
3. There is some extra-work to do to transform the path at pixel level, into a few intermediary points that are connectable by a straight line without obstacles (it is the only way to ensure that the reactive obstacle avoidance behavior won't fail).

We preferred, instead of developing a representation more adapted to our problem, to adopt the approach described below.

The idea lies into the fact that not all pixels are interesting to find intermediary points for a line-segments path. Indeed, let's consider a subset S of the cells of the map M. If for every cell of M, there is a cell of S that is *connectable by a straight line* without obstacle, then it is possible to find a path between every pair (a,b) of cells of M only by joining a to a cell of S, b to a cell of S, and cells of S with one another. Let's call such a subset a *complete* subset of cells.

Moreover an optimal path will necessary be *tangent* to obstacles (if not, the path could be shortened by getting closer to obstacles), and more precisely tangent to obstacles in points where the obstacle is *strictly convex* (curved toward the path). Actually the set of points that are tangent in convex points of obstacles is complete.

So if the vertices of the graph are the points of the grid that are tangent in convex points of obstacles, and if the edges only connects cells that are connectable by a straight line without obstacles, we will find the optimal path and get only the intermediary points of the line-segments path, that is what we are interested in. The next figure shows an example of what is a set of such points.

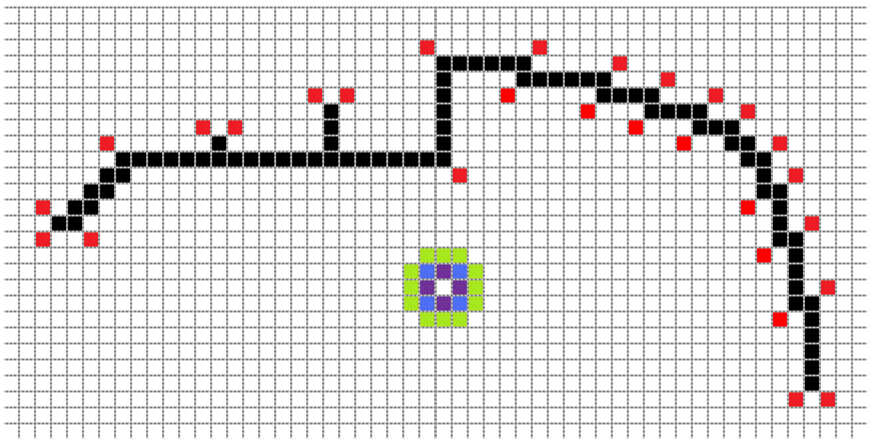


Fig.: Black cells are obstacles and red points are the candidate intermediary cells.
The green/blue/purple mask is used to characterize red points.

Some points seem to be in globally concave parts of obstacles, but due to sampling, there are actually convex parts in those regions. Not only they would demand a difficult global convexity determination to be rejected, but actually they should not be rejected because they are the only way to get access to some cells in the sampled map: this is the reason why these points are necessary for the planner.

The mask at the bottom of the figure characterizes the candidate intermediary points with the following policy:

- No purple cell is an obstacle (purple cells are 4 straight neighbors)
- Exactly one blue cell is an obstacle (blue cells are 4 diagonal neighbors)
- At most one “green side” (3 cells long line) contains more than one obstacle cell. This last constraint allows removing all points near 45° sampled lines.

Clearly, the cell of the goal and the cell of the current position of the robot must be added to the graph too. As a result, there are very few points considered by A*, so it demands a lot **less memory** and is

faster. Moreover, it gives us exactly the result we want: intermediary points connectable by a straight line.

However, one could point out that checking if two vertices are neighbors is a heavy task as it requires tracing the whole line-segment between the two cells and checking that it doesn't contain an obstacle cell. Actually the whole graph can be *precomputed* (identifying vertices and their vicinity), so that it takes no time during running to find neighbors of a vertex. It will consume little memory too (less than the grid map anyway). Still, as the goal changes at every mission, its neighbors have to be recomputed at the beginning of every mission, and as the current position of the robot keeps changing, its neighbors have to be recomputed every time planning is executed. The neighbors of the current position seem to be a real problem for large maps, because it could have to check all graph vertices. It is possible to reduce the number of vicinity checks by checking it only *after* the possible neighbor has been selected to be processed because it has a promising heuristic, and by planning the path from the goal to the current position (because heuristic is more reliable at the end of the path than at the beginning). However, in large maps, when there are obstacles lying on the path the first one is quickly found, so this is not that much time consuming. Actually, our tests show that the overhead introduced by the processing in the priority queue of a lot of additional false neighbors (O(log n) for insertion and deletion) seems to be larger than the overhead introduced by checking unnecessary neighbors. For a practical use it would be preferable to compare both solutions (we implemented both).

3 Embodiment issue

Having trajectories tangent to obstacles gives the optimal path. However, one should not forget that the robot has a size and that having it too close to obstacles makes it very likely that it will bump into it. To deal with that issue, we introduce obstacle dilatation: it makes the obstacles on the map appear larger than they really are. Doing so allows to execute A* with larger obstacles and to find a path that is farther from the obstacles than the theoretical one. The path is thus a little less optimal but at least, we are certain that the robot is not going to collide with obstacles.

2

Implementation and results

4 Statement

As stated in the first part of this report, the goal of the method we proposed is to enable a robot to navigate in a building. The robot has to avoid obstacles in a dynamic environment that will include both known and unknown obstacles, and both static and dynamic obstacles. The robot will be provided with a map of the environment in some cases, but the map will be inaccurate because we will add unexpected obstacles.

We restrained the domain to a smaller made-up environment for practical reasons, that involves obstacle avoidance with both planning and VFH. The aim of the testing is to ensure that each component works and that both algorithms complement each other well. Because of time constraints, our testing remains quite basic.

5 High-level description of the program

Our program is divided into 3 parts:

- VFH, that implements the VFH obstacle avoidance algorithm described in the first part
- Modified A* planner that uses the optimizations explained in detail in the first part
- Common data structures (maps, robot state, and communication with the robot's hardware).

The main loop of the program keeps updating the map with new sensors readings and the control commands, and calls the global planner at a given frequency.

6 Implementation details

Complete source code can be downloaded here:

http://crteknologies.free.fr/robot/gatech/hvfh.zip

It is written in C++ and a makefile is provided to compile under Linux with GCC. Four programs were created:

- hvfh (make): the main program that implements Hybrid VFH and controls the robot
- test_astar (make astar): independent test of A*
- test_vfh (make vfh): independent test of VFH
- test_robot: (make test): independent test of the robot control

The dependences are libxml2 and MissionLab (its install path can be modified in the makefile).

The global maps are provided to the program as bitmap images (24 bits BMP format), with obstacles in black and free areas in white. All configuration parameters can be given either in an XML configuration file, or in parameters of the command line interpreter. "hvfh --help" lists all the available configuration parameters with their description.

Moreover image files that represent information about the state of the robot can be generated periodically. This information includes:

- the a priori map (obstacles are in green),
- the histogram grid (red cells),
- the active window (yellow area around the robot),
- the polar histogram (as a frame around the map, red is obstacle, white or green is free),
- the position of the robot (blue cell),
- the position of the goal and intermediary planned goals (black cells),
- the current direction of the robot and the direction output of VFH control (black and blue dots in the polar histogram).

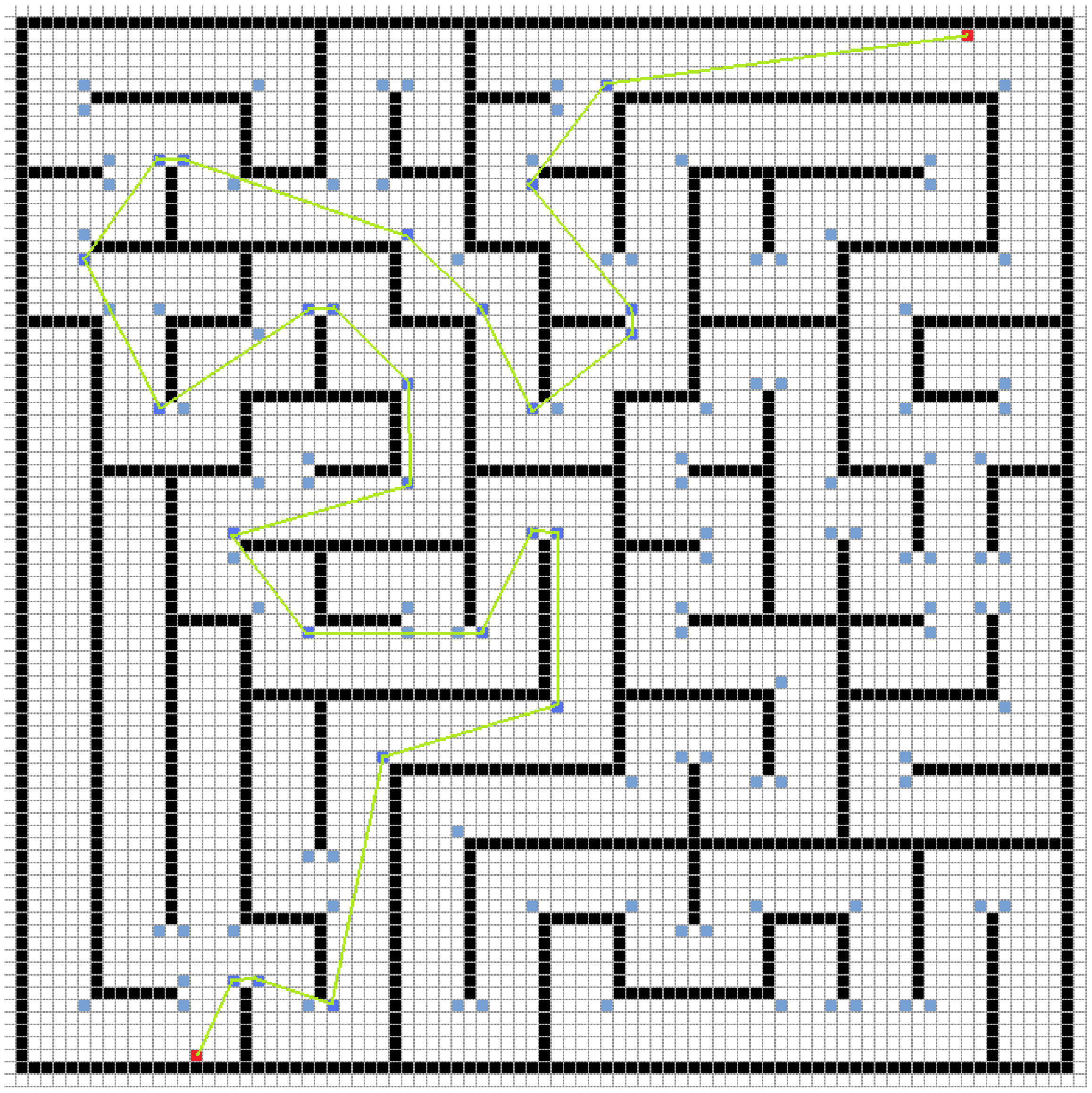
This allows understanding easily the behavior of the robot.

7 Test cases, evaluation and demonstrations

1 A* planner execution test (on a map, without robot)

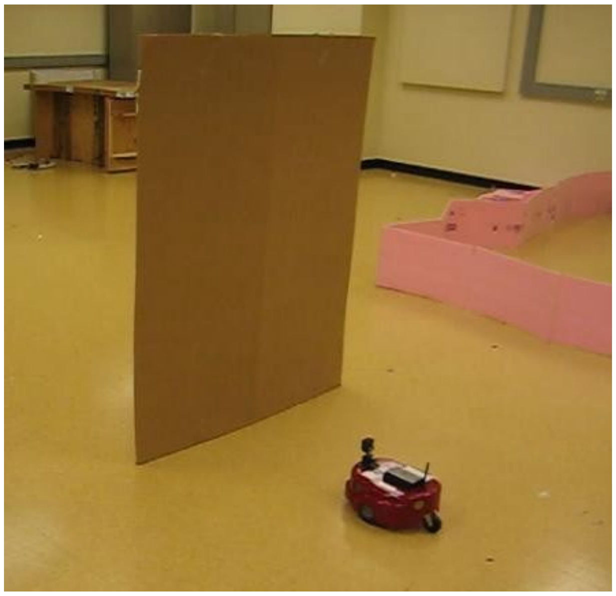
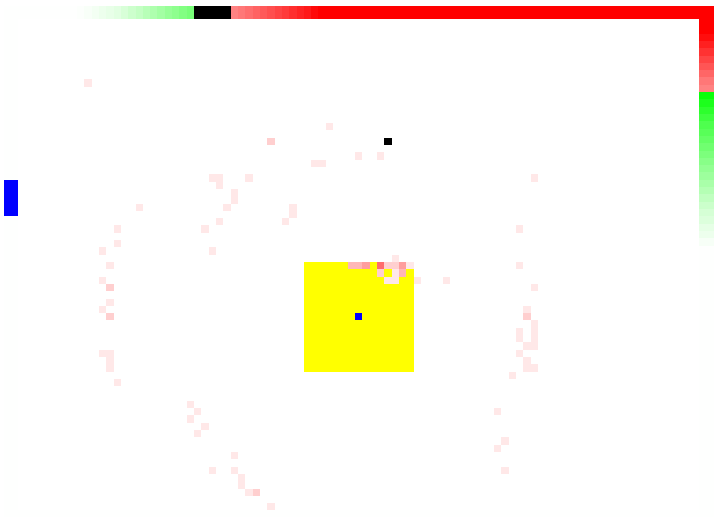
We executed our enhanced A* planning algorithm on several maps. The algorithm accurately gives the shortest path from the initial point to the goal.

The following picture shows an example of execution on a labyrinth kind map. Walls are in black, the start point and goal are in red, the blue cells are the intermediary candidate cells, and the green line shows the path found.



2 Avoidance of one obstacle (VFH, with robot)

The robot is given a goal that is 3 meters in front of it. We place an obstacle on the path between the robot and the goal. The robot is not provided with any map of the environment, it is only given a starting point and a goal.

The robot successfully avoids the obstacle and reaches its goal. However when the robot is driving too fast, it sees the obstacles too late and doesn't have a smooth trajectory.

The full video of the experiment can be downloaded at:

http://crteknologies.free.fr/robot/gatech/one-obstacle.avi (video camera)

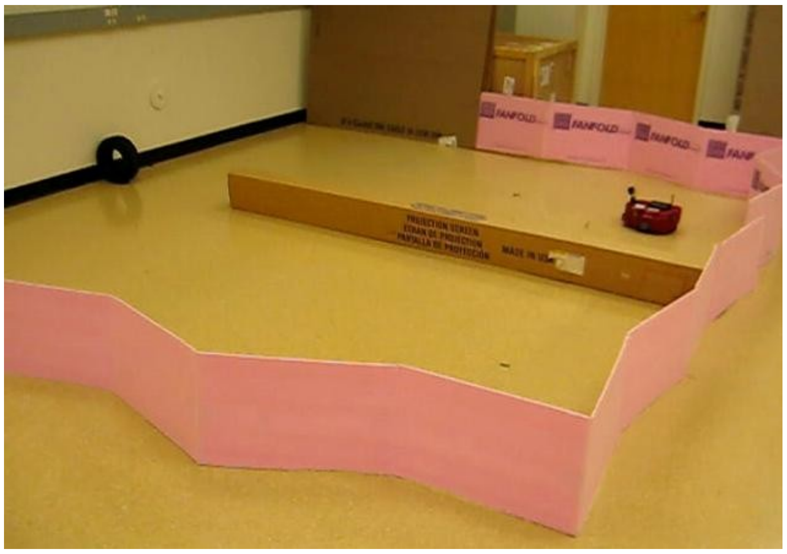
http://crteknologies.free.fr/robot/gatech/one-obstacle_data.avi (internal data)

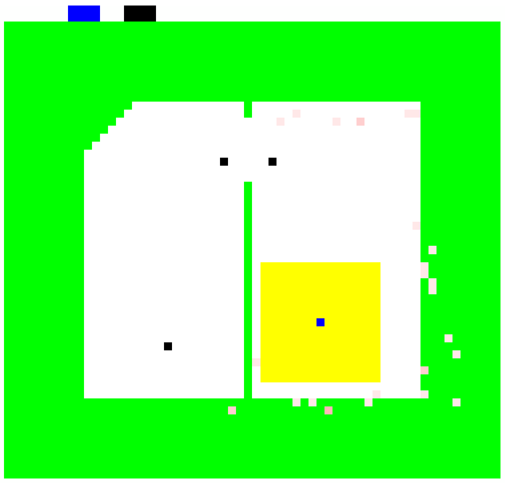
N.B.: if you encounter any problem while trying to download the file, please just go to the parent directory http://crteknologies.free.fr/robot/gatech/ and download the one-obstacle.avi file as well as the one-obstacle_data.avi file.

3 Navigation towards a goal in a two room environment

2.4.3.1 A*, with a robot

We put the robot in the following two room environment:





The robot is provided with a full map, the A* algorithm successfully find the shortest path to the goal, and the robot successfully follows the path returned by A* and reaches its goal (VFH was activated and prevented the robot to go too close to the walls, but didn't play a major role).

The full video of the experiment can be downloaded at:

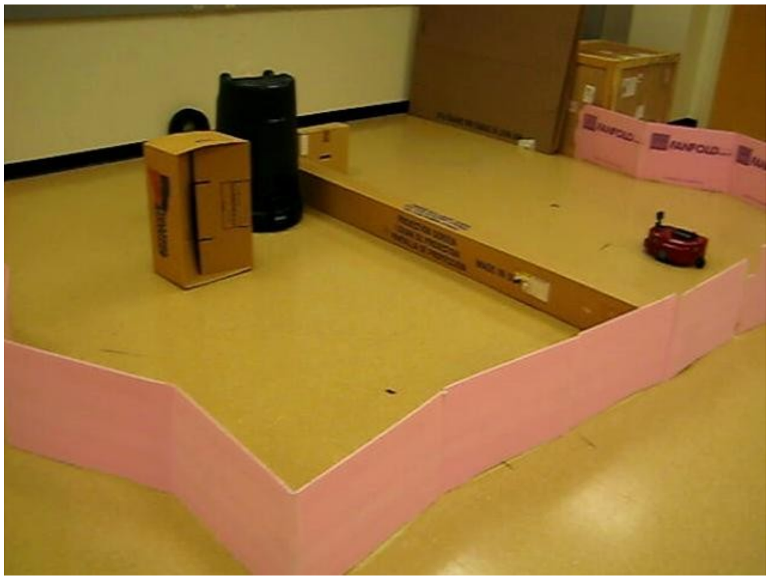
http://crteknologies.free.fr/robot/gatech/only-global.avi (video camera)

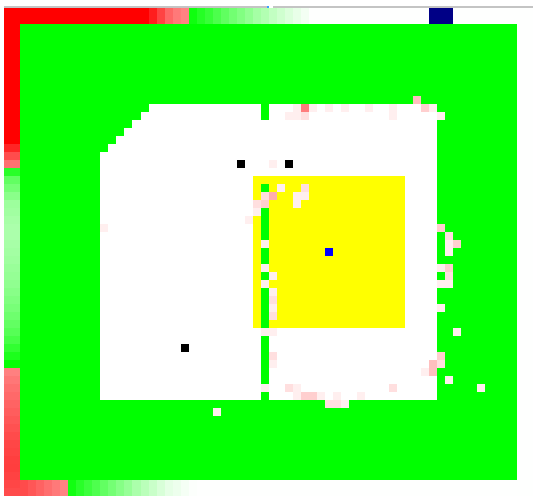
http://crteknologies.free.fr/robot/gatech/only-global_data.avi (internal data)

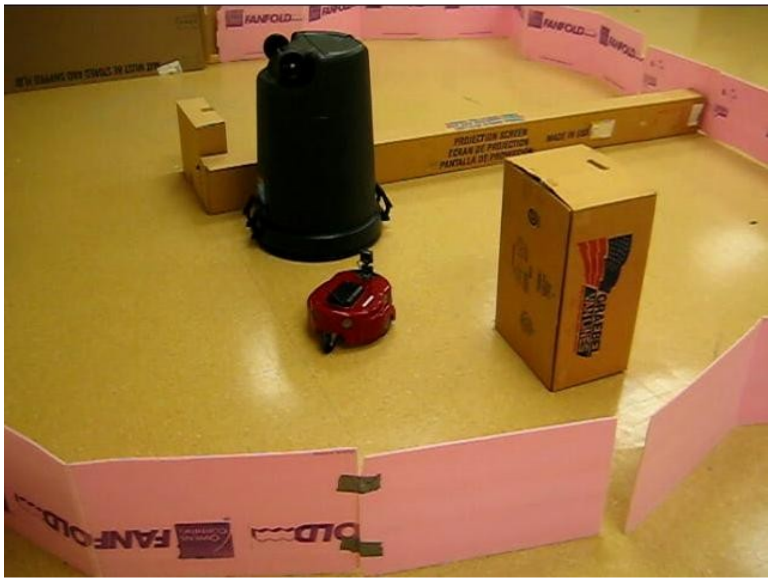
N.B.: if you encounter any problem while trying to download the file, please just go to the parent directory http://crteknologies.free.fr/robot/gatech/ and download the only-global.avi file as well as the only-global_data.avi file.

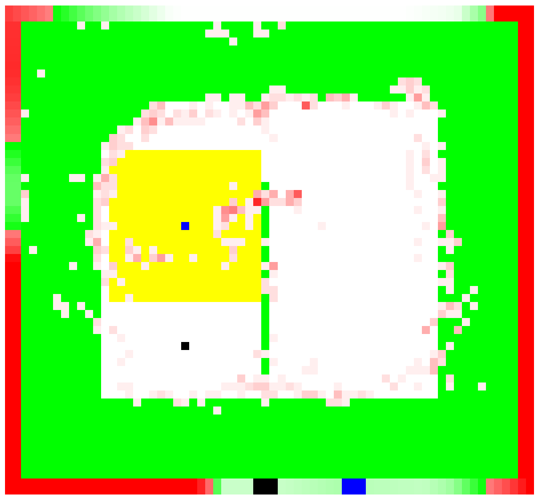
2.4.3.1 A* and VFH, with a robot

We put the robot in the same two room environment, providing it with the same map as before. But we also put unexpected obstacles in the environment to force the robot to use VFH and A* at the same time.









The A* algorithm successfully returns the shortest path between the starting point and the goal. The VFH algorithm allows the robot to avoid unexpected obstacles successfully, even dynamic ones (we forced it to pass where it had seen an obstacle previously by moving it).

With Endo:

http://crteknologies.free.fr/robot/gatech/hybrid.avi (video camera)

http://crteknologies.free.fr/robot/gatech/hybrid_data.avi (internal data)

With some remaining bugs corrected and dynamic obstacle (if there is only one video to see, this is that one):

http://crteknologies.free.fr/robot/gatech/hybrid-dynamic.avi (video camera)

http://crteknologies.free.fr/robot/gatech/hybrid-dynamic_data.avi (internal data)

N.B: if you encounter any problem while trying to download the files, please just go to the parent directory http://crteknologies.free.fr/robot/gatech/ and download the video you wish to see.

8 Problems, limitations and improvements

1 The platform (Amigobot, MissionLab and HClient)

The HClient interface of MissionLab has appealing features: high-level API, independent from the robot, client and server can be distant.

However we had several problems with the actual implementations.

The most annoying was probably the fact that some functions don't seem to do what they are supposed to. For example the function "hclient_steer" is described as follow:

```
    // Set robot angular speed
    int hclient_steer(float avel); // degrees per second
```

But it actually interprets the argument as an angle in degrees, and makes the robot turn this angle.

The function:

```
    // Steer toward theta and when within drive_wait_angle start to
```

```
drive at speed
int hclient_steer_toward_and_drive(float speed,float theta,float
drive_wait_angle = 15.0,int use_reverse = true);
```

actually seems to make the robot advance from the distance given in the first parameter, and turn the angle given in the second parameter.

Sonar readings are position-stamped (the position of the robot when the sonar measure was taken), that is very useful because the robot can have moved quite a lot between the time the sonar measure is taken and the time it is retrieved. However the dead-reckoning used is different than the one that returns the position of the robot, and it is not affected by position initialization commands. So it always assumes that the robot start position is (x=0,y=0,theta=0), and we had to change the coordinate system to use the position-stamps. Moreover, it prevents from running the program twice without disconnecting the robot.

A missing feature is that HClient doesn't explicitly tell which sonar values are updated and which are not. However it is necessary in our case because we don't want have the same reading vote several times. So we tried to infer if the readings were updated by testing if position-stamp or value changed since last time, but there is still a problem "out of range" values when the robot is still (it is taken into account only once).

The sonar on AmigoBot is pretty slow (25Hz, but they are updated sequentially so each sonar is updated at 3Hz), so we had to slow down the robot because it didn't have enough time to see some obstacles, particularly on the side in front of it. Moreover one sensor was not functioning on our Amigobot Huey (-44°), and another one was returning values less often than the others (+12°). In addition, as we can see on the internal data videos, there are a lot of spurious readings.

2 The method

If the method is to be used in a practical application to navigate in a building, dead-reckoning would not be sufficient, and it will be necessary to do some more elaborate localization, for instance by comparing the histogram grid map to the *a priori* map.

Another problem that can appear is when there are two valleys at similar distance from the goal, the robot can fluctuate between. This could be improved with an adequate heuristic to try to stick a little bit more to its choice.