Universal Autonomous Robot Navigation Using Quasi Optimal Path Generation

Áron Lászka
aron@laszka.hu

Gábor Pék
kheelr@gmail.com

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

2008.
Budapest, Hungary
Abstract

Autonomous robot navigation is an important research field because these robots can solve problems where the human presence is impossible, dangerous, expensive, or uncomfortable. In this paper, a new hybrid autonomous navigation method is introduced. The algorithm is composed of visible/shortest path global navigation and simple potential field based local navigation parts. It applies a new automated graph generation method which may become necessary if, because of the observed new obstacles, a new path should be generated. The quasi optimal route is found by applying a modified D* algorithm on the graph. The presented technique offers a quasi optimal universal navigation technique which can successfully be used in all, known, unknown, and dynamically changing environments.
## Contents

1 INTRODUCTION .................................................. 4

2 KEYWORDS ...................................................... 6

3 NAVIGATION ...................................................... 8
   3.1 Global navigation ........................................... 8
      3.1.1 Roadmap algorithms .................................... 8
      3.1.2 Graph search algorithm ................................. 9
      3.1.3 Automated graph generation ............................ 10
      3.1.4 Visibility/shortest path graphs ....................... 11
      3.1.5 Generalized Voronoi diagrams ......................... 12
      3.1.6 Silhouette method ..................................... 13
      3.1.7 Rapidly Exploring Random trees - RRTs ............... 14
   3.2 Local navigation ............................................. 15
      3.2.1 Potential field based navigation ..................... 16
      3.2.2 Vector field based navigation ......................... 16
      3.2.3 Bug algorithm family .................................. 17

4 THE NEW IMPROVED NAVIGATION SYSTEM ................... 20
   4.1 Global navigation ........................................... 22
   4.2 Local navigation ............................................. 24
   4.3 Quasi optimal graph regeneration ......................... 24
   4.4 Path finding using modified D* ............................ 26

5 EXAMPLES ...................................................... 28

6 CONCLUSION .................................................... 31

List of Figures .................................................. 34
1 INTRODUCTION

As time goes on, the advancement of technology keeps thriving, consequently more and more sophisticated solutions are invented. One of the most significant fields of science which has to be taken into consideration is autonomous robot navigation. There are situations where human interference is not an option and there is no other solution but applying autonomous robots to tackle these problems.

Firstly, there are examples that deal with problem solving which comprise even the carrying of the radioactive by-products of nuclear power plants or lifesaving in case of disasters by taking medicines to the injured if they cannot be approached only by means of little autonomous machines. Secondly, examples can be enumerated, which concentrate on information gathering in order to support certain scientific activity. These involve e.g. the exploration of the surface of Mars which was performed by NASA’s Sojourner, the autonomous robot carried by Pathfinder [2], which provided valuable pieces of information that water had been existed on Mars. However, the aforementioned are extreme cases, but problems can be found in all walks of life. Autonomous robots could substitute people where the task is too monotonic, such as the collection of litter in a building or looking out for them at night.

The navigation system of an autonomous robot is one of its components which is responsible for the decision where to move in the next moment. The word “autonomous” stands for a concept, which considers that the robot does not communicate with the outside world under normal circumstances. This system requires pieces of information in order to reach its goal apart from what its current position is.

On one hand, the navigation system can have a priori, pre-supplied data about its environment, and on the other hand the sensory data is the one which carries information on the momentary state of the immediate surrounding of the robot. However there are solutions where the computation burden is not on the shoulders of a robot, but its environment direct is. For the sake of example, the work of Maxim A. Batalin et al. [3], who proposed an algorithm for robot navigation using sensor networks, neglecting the use of pre-supplied maps, GPS or localization on the part of robot, (so the environment directs the agent), can be successfully used in an environment, where the placement of sensors is feasible.

As far as the types of obstacles are concerned, there can be static and dynamic ones. The former refers to objects the positions of which are pre-supplied into the robot and the latter to those ones which are detected by means of sensors. The issue of obstacle is
avoidance is appealing field for many researchers, because real time problems can appear at any time in a dynamic environment. One such a work is the proposal of Marcelo Becker et al. [4] who developed a new obstacle avoidance procedure based on the Velocity Obstacle Approach (VOA). VOA defines the set of robot velocities that would result in a collision between the robot and an obstacle moving at a given velocity[5].

The alignment of navigation systems fall into three categories: global, local, and hybrid. In case of global navigation, only pre-supplied data are accessible, such as the map of the environment, while in case of local navigation, only the sensory measurements are taken into account. Both concept have advantages and disadvantages as well. The former can be successfully applied in an known environment, so the emplacement of obstacles are fix and there are not unexpected objects that influences the result of calculations (path generation). The latter includes algorithms that do not suggest anything about the agent’s surrounding, therefore pre-supplied map is not required, so can be successfully utilized in unknown environments, however, contrary the global systems these are mainly not completes. One such a paper of the latter field is the work of Koren and Borenstein [6] which presented a systematic overview and a critical discussion on the inherent problems of potential field based methods (PFB), based on mathematical analysis.

Hybrid method merges the advantages of both aforementioned systems. This concept was introduced by Kiss and Várkonyi-Kóczy [7], which based on indoor navigation, because the use of doors was a necessity to the path-finding.

In this paper, based on the results of Kiss and Várkonyi-Kóczy [7], a new improved hybrid navigation algorithm is presented which can be used both indoors and outdoors. The new method combines the advantages of the previous techniques and outperforms their behavior. It ensures optimality, handles the local minima problem systematically, and applies an automated graph generation for the path finding in maps.

The paper is organized as follows: Section 3 deals with certain kinds of navigation systems. The new navigation system is introduced in Section 4, while Section 6 concludes the paper.
2 KEYWORDS

- Global navigation
  A navigation algorithm is considered to be global, if it uses information of the robot’s whole area of operation. Since this information is usually not recent and can not be processed in real time, these algorithms are suitable for avoiding large sets of static obstacles.

- Local navigation
  A navigation algorithm is considered to be local, if it only uses information of the robot’s immediate surrounding. Since this information is usually recent and can be processed in real time, these algorithms are suitable for avoiding small sets of dynamic obstacles.

- Hybrid navigation
  A navigation algorithm is considered to be hybrid, if it comprises a local and a global navigation system, and accessory functionalities that help the previous two to work together. Accessory functionalities include for example local minima detection, or the use of obstacle memory.

- Optimality
  A navigation algorithm is considered to be optimal, if it produces such a collision free path for every situation, so that there is no other shorter collision free path.

- Handling local minima
  A situation is considered to be a local minima for a navigation algorithm, if every possible step of the algorithm results in a worse situation than the current one. For example, a point in space is a local minima for a potential field based algorithm (see 3.2.1), if the obstacle avoidance vector in that point has zero length.
As far as one might know, local minima detection and avoidance is one of the most significant tasks that an agent must handle. If this ability suffers from weaknesses, then any unexpected situation can end up in the instant failure of the robot. Local minima situations can be raised by a closed door which was previously open, when the path was designed by the global navigation system. On the other hand, an unanticipated event could produce a state from which the agent can not find a way out. Both cases have a common feature, which is the lack of progression of the robot towards the goal. In these situations, the robot can not follow the selected path, so it fails to work. This phenomena includes firstly the total inability to move, secondly the situation when the robot only revolves around a certain place, but there is no progression to the goal.
3 NAVIGATION

Navigation algorithms can be categorized in many ways. We have chosen to divide them by the information they use to global and local navigation algorithms. Global algorithms use information about the whole area of the robot’s operation. This is usually given in the form of a map containing static obstacles. Local algorithms use only information about the robot’s immediate surrounding. This can be given in many forms, e.g. values representing readings from distance sensors.

3.1 Global navigation

As these algorithms use information about the whole area, they have to handle more data and are more complex. Usually this means that we have to do as many calculations offline as possible. This suits the supplied information, since the robot’s sensors can only provide up-to-date information about the robot’s immediate surrounding.

3.1.1 Roadmap algorithms

Global navigation algorithms can be easily realized by first constructing a graph from the supplied map and then searching for the shortest path on this graph [8, 9]. The problem

\[ \{M, p_{\text{start}}, p_{\text{end}}\} \mapsto P \]  

of global navigation can be divided to

\[ M \mapsto G \]  

and

\[ \{G, p_{\text{start}}, p_{\text{end}}\} \mapsto P \]  

where \( M, G, \) and \( P \) are the sets of maps, graphs, and paths respectively while \( p_{\text{start}} \) and \( p_{\text{end}} \) are the start and end points. The nodes of the graph correspond to free points in real space and edges to paths between these points, which the robot can safely go along. The graph generation can be done as a precalculation step if the map only contains static obstacles. Searching for the shortest path can be done using the A* [10], D* [11] or other well-known algorithms. For the graph generation several different solutions exist, each having its own advantages and disadvantages.

One of the biggest drawbacks of roadmap algorithms is their low performance in dynamic environments, as current approaches simply replan from scratch when changes
are observed. This can be improved by continuously replanning in an anytime fashion as more and more information is available. The method can be further enhanced by extrapolating obstacle trajectories based on previous motion and adding a time dimension to the search space [12].

3.1.2 Graph search algorithm

Graph search algorithms try to find the least-cost path from a given initial node to one or more goal nodes.

A* algorithm

The A* [14] is one of the most popular and well-known algorithms. It uses a heuristic function, usually denoted \( f \), to decide in which order to visit the nodes. The above function is the sum of two functions: the path-cost function, which is usually denoted \( g \), and which is not heuristic, and a heuristic estimation of the distance to the goal, which is usually denoted \( h \).

The path-cost function \( g \) is the minimum cost to reach to current node from the start node. The heuristic function \( h \) is an admissible heuristic, that is, it must not estimate greater distance from the current node to the goal, than the minimum cost. For navigation problems, straight-line distance to the goal is a commonly used heuristic, as it is physically the smallest distance possible.

The algorithm uses two lists containing the nodes of the graph. The OPEN list holds nodes that should be considered as the next node to be visited. The CLOSED list contains nodes that have already been visited. A node can only get into the CLOSED list, if all of it neighbours have been added to the OPEN list. At the beginning the OPEN list contains the start point and the CLOSED list is naturally empty. The algorithm is the following:

1. If there are no nodes in the OPEN list, then the goal is unreachable and the algorithm stops.
2. Select the node \( X \), which has the smallest sum of path-cost and heuristic estimate:
   \[
   f(X) = \min_{Y \in \text{OPEN}} f(Y) = \min_{Y \in \text{OPEN}} g(Y) + h(Y).
   \]
3. If the node \( X \) is the goal node, then we have found an optimal solution. The path can be traced back by looking up which node was reached from which node.
4. Else we add the neighbours of $X$ to the OPEN list, update their path-cost values, and record that they are reachable optimally from $X$.

5. Move $X$ to the CLOSED list, and continue from step 1.

It can be shown, that for a given heuristic, the steps of A* are optimal.

**D* algorithm**

The D* [15] is very effective in cases, where the environment is dynamically changing, thus it is very suitable for navigation problems. The strategy of the algorithm is to store a pointer for every node, which points toward the optimal goal path. During motion, the robot simply follows these pointers.

Whenever there is a change in the graph, i.e. edges are removed, the affected nodes are added to a priority queue, which contains nodes to be updated. The algorithm always chooses that node from the queue, which has the minimal goal distance estimate. After a node has been processed, any of its neighbours, that point to it, are changed and added to the queue.

The D* needs a lot of calculations for the first path to be found, but quick updates are possible afterwards.

### 3.1.3 Automated graph generation

Let $M$ be the set of possible maps and $G$ be the set of graphs. We are looking for a function $f: M \mapsto G$, so that for every $m \in M$ map the shortest path $p$ on the generated $g \in G$ graph is not significantly longer than the possible shortest path in real world space and $f$ can be fully automated.

In case of local minima the solution is the regeneration of the graph by means of the $f^*: M' \mapsto G$ function, where $M' = M \cup Mo$ and $Mo$ denotes the set of observed obstacles. Thus $M'$ contains either the static elements (all of them are part of set $M$) or observed obstacles that were supplied by the external sensors of the robot.
3.1.4 Visibility/shortest path graphs

Visibility graph is a graph generation method for polygonal obstacles [13, 9]. Each node corresponds to a vertex of an obstacle and two nodes are connected if the segment joining them does not intersect any obstacles (they are visible from each other, hence the name). If we add a start and end node to the graph corresponding to the start and end points in real space, then the shortest path between the end and start node on the graph is the shortest path in real space (hence the other name).

Consider the set of possible maps $M$ to comprise the sets of obstacles which are represented as polygons. Let $P$ be the set of polygons from which topological maps are built up. The procedure to generate a graph by these pieces of information is to extrude the vertices of each polygon one by one to a desired distance from the original positions. By this step we ensure that the resulted graph of the generation will decrease the risk of a possible bump of the robot with obstacle. In this way the agent will meet the extruded vertices while progressing to the goal. Let $V'$ be the set that includes these new vertices, by means of which the $h: V' \mapsto G$ can be defined. This is considered to be the graph generator function.

Figure 1 shows the visibility graph of an example map.

Figure 1: Visibility graph of an example map
3.1.5 Generalized Voronoi diagrams

A Voronoi roadmap is a set of paths in an environment that represents maximum clearance between obstacles. Due to this method the chance of collision, which can stem from the inaccuracy or failure of the agent’s sensors, with an obstacle is smaller. Voronoi roadmap is considered as a generalization of Voronoi diagrams for points. Let \( P \) be the set of \( n \) points. \( \forall p \in P \) points, the Voronoi cell of point \( p \) is the set of points that are closer to \( p \) than to any other points of \( P \). Voronoi diagram is the common name of the space partition induced by Voronoi cells. From this point it easy to imagine how this method can be used to path generation. Considering the edges of Voronoi cells to sections of the path and the points to obstacles, this method provides a safe way to keep to robot at maximum distance from the obstacles.

![Figure 2: Voronoi diagram and the selected path](image)

As figure 2 shows the path that was selected by the Voronoi method is the best one considering the maximum distance from obstacles. The creation of cells can be summarized with the following steps:

1. Select point \( p \in P \) for which its Voronoi cell is created
2. Select another point \( p' \in P \) and divide the plane equally into two half-planes between point \( p \) and \( p' \).
3. Iterate step 2, until \( \forall p \in P \) are examined.
4. Points that are situating at the half-planes, which contains point \( p \) are in its Voronoi cell.

However, in real life situations, obstacles are not point-like and can be forms polygons. In this case Generalized Voronoi Diagrams are used. Approximation with obstacle
discretization is preferred, because the boundaries of obstacles can be high-degree curves, the calculation of which is complex and takes long time. Obstacles are represented by set of points which are sampled along its boundaries. After that Voronoi cells have to be created with the method described previously. The resulting Voronoi diagram will have edges which are inside the obstacles, consequently these have to be neglected when traversable paths are required. Edges which are defined by two consecutive points of the same obstacle have to be ignored. The resultant edges can be traveled safely keeping maximal distance from the obstacles.

3.1.6 Silhouette method

Another approach is to decompose (or split up) the free space into polygonal regions, called cells. If the resulting regions are small and simple enough, then we can safely navigate inside them. Thus we can construct a graph, where the vertices correspond to cells, and the edges to connections between cells that share a common bounding segment. There are many ways to geometrically break up free space, each having its own advantages and disadvantages.

We have chosen to examine Canny’s silhouette method [16], as this algorithm has running time linearly proportional to the number of polygon vertices.

Canny’s algorithm is to decompose the environment into silhouette curves which represent boundaries of obstacles, so regions are no longer trapezoids or triangles. Consider a vertical line sweeping horizontally from the leftmost environment vertex to the rightmost.
As the line sweeps, the topmost and bottommost points form the silhouette boundary. When an obstacle is encountered, the sweepline is split into two vertical lines, one above the obstacle, the other below it. Two split lines merge, when their endpoints meet upon leaving the obstacle. As the line sweeps, the topmost and bottommost extreme points form the silhouette boundary.

A path can be found by adding extreme points of vertical lines passing through the start and goal points, then following silhouette curves. Vertical lines at sweepline splitting or merging, vertical lines at the start and goal points, and silhouette curves are the edges of the graph.

### 3.1.7 Rapidly Exploring Random trees - RRTs

RRT is an algorithm that can explore its environment in a rapid way. The nodes of the tree represent random agent configurations, such as the location of the agent in a given map. The current position of the robot is the root of the tree and it expands outward entirely randomly of biased somehow to the goal. The algorithm requires two input parameters: the number of nodes of the tree \( n \) and the length of the edges \( s \).

The functioning of the algorithm can be given with the following few lines. The output of the algorithm is graph \( G(V, E) \).

#### Pseudocode 1 The RRT algorithm

Let \( V \) contain the start_vertex and \( E \) be empty
Repeat {
    Let \( q \) be a random valid robot position // (i.e., random point)
    Let \( v \) be the node of \( V \) that is closest to \( q \).
    Let \( p \) be the point along the ray from \( v \) to \( q \) that is at distance \( s \) from \( v \).
    If (\( vp \) is a valid edge){ // i.e., does not intersect obstacles
        add new node \( p \) to \( V \) with parent \( v \) // i.e., add edge from \( v \) to \( p \) in \( E \)
    }
} Until \( V \) has \( n \) vertices.

Figure 4 shows the addition of a next point \( p \) into \( V \).

The problem with this solution is that the algorithm can stuck very easily, if narrow corridors are present. The other pitfall of this algorithm is that the tree can grow entirely randomly and it is not sure that there is a \( v \in V \) point that is near enough to the goal.
3.2 Local navigation

Since these algorithms use only information of the robot’s surrounding, they usually do not have to handle as much data as global algorithms. This allows them to do all the calculations online, which again suits the information as the robot’s immediate surrounding is usually known precisely at the very moment.
3.2.1 Potential field based navigation

One of the possible local navigation systems is the so called Potential Field Based (PFB) guiding [19, 6] which was our preferred one as well. The core idea of this concept is that the environment affects with repulsive forces to the agent according to the following expression:

\[ y = - \sum_{i} p(d_i) \cdot e_i \] (4)

where \( d_i \) is the distance between the \( i \)th obstacle and the robot, \( e_i \) is the unit vector pointing towards the \( i \)th obstacle from the robot, and \( p \) is a potential function. This potential function can be based on Coulomb’s law or some kind of variation of it or trained using neural networks. Vector \( y \) is the obstacle avoidance vector. In some cases \( p \) can have negative values which means attractive force. We are going to use this feature in case of obstacle avoidance. In that case the attractive point should be put to the goal.

The main advantage of PFB guiding lies in its simplicity. The algorithm needs very little processing power and can be easily trained. Furthermore the inputs of a PFB system can be directly mapped to distance sensors of a robot. Nevertheless, a major disadvantage of this approach is that the robot can get trapped in local minimas other than the goal. This problem can be dealt with, but this usually makes the algorithm much more complex, losing its main advantage [20].

3.2.2 Vector field based navigation

The Vector Field Based (VFB) navigation [21] is an improved version of the PFB guiding. The main difference is that the direction of the repulsive force of an obstacle is not always parallel with the direction in which the obstacle is detected, but depends on it in a more complex way. This could be written in the following expression:

\[ y = \sum_{i} p(d_i) \cdot e_i \] (5)

where \( d_i \) is the distance between the \( i \)th obstacle and the robot, \( e_i \) is the unit vector pointing towards the \( i \)th obstacle from the robot, \( p \) is a potential vector function, and \( y \) is the obstacle avoidance vector.

Function \( p \) can be trained using neural networks. This allows us to teach the robot different kinds of navigation styles according to our needs [22]. For example we could
make it follow walls on its left side but avoid them as much as possible on the left side. Realization of these kind of complex navigation styles would not be possible with PFB.

The advantages of VFB are similar to the ones of PFB. The algorithm needs little processing power and can be easily realized. However, training the algorithm, which usually requires neural networks, is more complex. The disadvantages of VFB are also similar to the ones of PFB. The chances of getting trapped in a local minima are less likely, but still exist. To correct this, similar methods can be applied.

3.2.3 Bug algorithm family

There exists a family of algorithms which only use local knowledge, but are complete in the sense that they always find a safe path [23]. This family consists of Bug0, Bug 1, Bug 2, and a number of their derivatives [24]. Their name originates from the similarity between the robot model used by these algorithms and an insect. The robot has two simple behaviors: follow a wall (left or right) or move towards the goal in a straight line.

![Figure 6: Path of Bug 0 on an example map](image-url)
The simplest algorithm of the family is Bug 0, which cannot remember preceding
states, so it has no memory. Its functionality can be described with the following steps:

1. head toward the goal (this is the default movement)
2. if an obstacle is encountered, follow it, until it can head toward the goal again
3. continue

As Figure 6 shows, there are situations when Bug 0 fails to find the goal, so it is
neither complete nor optimal.

![Figure 7: Path of Bug 1 on an example map](image)

Bug 1 algorithm can be described with a set of simple rules:

1. head toward the goal by default
2. if an obstacle is reached, then circumnavigate it and remember the point that is
   closest to the goal
3. return to that point and continue

It is trivial that this algorithm is complete (if there are finite number of obstacles),
since the robot gets closer to the goal with each obstacle circumnavigated, and every
obstacle is circumnavigated at most once.
Bug 2 and their derivates work basically the same. At first glance these algorithms seem to be a perfect solution as they are complete and only use local knowledge which is always available. However, there are certain drawbacks, which make them inferior to hybrid navigation systems in many ways. First of all realizing the wall following behavior only using distance sensors is not an easy task. There are solutions which address these problems, but they can not eliminate it completely. But their biggest drawback lies in the fact that for typical real world situations the path given by Bug algorithms are far from optimal. The upper bound for the length of the path is proportional to the sum of every obstacles’ perimeter, which makes bug algorithms unsuitable for many applications.
The new method tries to utilize the advantages of both the hybrid system itself and to improve the effectiveness by applying well-known and quick local navigation (PFB) and optimal global part (shortest path, D*).

It is essential to emphasize what a hybrid system means exactly. There can be various interpretations, but in our case the idea is the next.

A hybrid navigation system comprises a local and a global navigation system and accessory functionalities that help the previous two to work together. The global and local part can be any kind of global and local algorithm. The accessory functionalities have to include an algorithm for detecting if the local system has encountered a situation it can not solve. Other functionalities can be added to improve the system, but this depends on the global and local parts chosen.

Figure 9: Block diagram of the new hybrid navigation system

Figure 9 shows how the new hybrid navigation system is built up. As it was previously mentioned a hybrid system comprised of its global and local part. The first one is responsible for designing a quasi optimal (optimal if only pre-known, static obstacles are present) path consisting of temporary goal-points and leading to the final goal and for the avoidance of static obstacles, while the second one makes the robot move to the next (temporary) goal. However, it is not enough to purely merge them, because their functionality has to be enhanced. This improvement includes the local minima detection and the obstacle memory itself. The new concept of obstacle memory covers dealing with the temporary storage of dynamic obstacles that have just been recognized. By merging the content of this memory and the set of static obstacles we get a wider look of the environment of the robot. After that the shortest path method must be applied on this union and a new path is searched on it to reach the goal. Of course the capacity of this temporary storage is limited, so only recent obstacles are taken into account. The recent expression refers to a specific surrounding of the agent, as if a square shaped net were
moving simultaneously with the robot which is positioned at its center. If an obstacle is nearby then it supposedly intersects with one or more segments of this grid and if so it is considered to be a dynamic obstacle and stored in this temporary memory. Note that this grid is solved by use of sensors that are constructed to the robot.

Figure 10: Data and control flow diagram of the new hybrid navigation system

It is needless to say that if there is no outer interference the agent keeps trying to follow the selected path as much as it can.

Figure 10 demonstrates the data and control flow diagram of the new system. As it is visible, the graph generation requires information from the sensors, whether a new obstacle and/or local minima is encountered, and the map, which contains the known obstacles, that is supplied previously to the agent. If all the required pieces of information is ensured the graph generation is performed by the concept of visibility/shortest path. After that only a path finding algorithm, such as $A^*$ or $D^*$ has to be applied on it in order to get the optimal route to the goal. Last but not least the local navigation is going to use this path and the data from the sensors to span the distance between to nodes of the generated graph.

Consider the previously given graph generator function $h: V' \mapsto G$. The output of this function is a graph that is generated by the shortest path procedure. After we have a graph on which an optimal path finding is easy to be implemented (e.g. $A^*$) the navigation system is going to use it, if there is no unexpected obstacle en route. In this case we can simply rely on our global plan and touching the corresponding vertices as progressing to the goal on the edges. The navigation itself is realized by the local part between two vertices. It has simple reasons: we have to take the strict fact into consideration that the environment is diverse, where the surrounding of the agent can change in every moment. In details it means that people are going to roam all around in a building, or we cannot recognize a new obstacle it is evident that local navigation can only handle these situations.

The following sections will discuss the local navigation and the global one in details, respectively.
Pseudocode 2 The new hybrid navigation method

Navigate(goalPoint : Point, map : Set of Polygons) {
    graph : Graph<Points, Lines>;
    path : Array of Points;
    nextPointOnPath : Integer;
    previousPositions : Set of Points;

    graph = GenerateGraph(map);
    graph += position + goalPoint;
    path = FindShortestPath(graph, position, goalPoint);
    nextPointOnPath = 1;

    while (position is not near goalPoint) {
        if (position in previousPositions) {
            observedGraph = RegenerateGraph(map, observedObstacles);
            path = ModifyShortestPath(observedGraph - graph, path);
            nextPointOnPath = 1;
        }

        previousPositions += position;
        if (position is near path[nextPointOnPath])
            nextPointOnPath++;

        direction = PFB(sensor data, path[nextPointOnPath] - position);
        Move(direction);
    }
}

4.1 Global navigation

The global navigation system has to give a path for any start and end point pair on a map it has previously learned. We define a path as a sequence of points. The start and end points can be any points on the map that are not occupied by an obstacle. The fact that this system can learn the map before any task is given to it carries high importance to its performance, since this allows it to make calculations which are not feasible during the actual navigation. For most roadmap algorithms this means that the graph can be generated without time constraints, allowing us to use more complex algorithms.
As previously mentioned $M$ and $G$ are the sets of maps and graphs respectively while $p_{start}$ and $p_{end}$ are the start and end points, $p_i$ is the $i$th point of the path, $i = 1...n$, and $n$ is the number of points on the path. Using offline calculations the

$$g: \{M, p_{start}, p_{end}\} \mapsto (p_1, p_2, \cdots, p_n)$$

(6)

global navigation system can be divided into

$$g_{offline}: M \mapsto G$$

(7)
and

$$g_{online}: \{G, p_{start}, p_{end}\} \mapsto (p_1, p_2, \cdots, p_n)$$

(8)

where $g_{offline}$ is a more complex function which can not be executed real-time, while $g_{online}$ is a simpler one and can be executed real time.

For the global navigation system we made the following requirements:

- It has to be complete. That is, for every possible map and start and end point pair, it has to give a path that the robot can safely go along.

- It has to give such a sequence of points, so that between any point and the next one a local navigation system would be able to navigate through. For this requirement we defined that for any $p_i$ and $p_{i+1}$ the segment $p_ip_{i+1}$ should not intersect any obstacles.

- It has to give a path that is near optimal.

Many roadmap algorithms fulfil the above requirements, however, we were looking for the best solution. We have made several experiments with different graph generating algorithms and compared them based on

- their performance during modifying a graph with newly observed obstacles

- the optimality of the given path.

We have found that the visibility/shortest path method has adequate performance while (naturally) it gives an optimal path.
4.2 Local navigation

Between the points of the path given by the global algorithm, a local navigation system has to guide the robot. Since we defined that the segment $p_ip_{i+1}$ can not intersect any obstacles, in a static environment the simplest algorithm, which guides the robot to the next point on a straight line, could be used. Unfortunately, we can not expect the environment to be static, so we need more complex solutions.

Since our focus was on developing a hybrid systems, for the local navigation we have chosen one from the previous works.

As a consequence of the fact that the environment is to be changing as time goes on, we have to take certain things into consideration. Firstly, people can pass the territory where the robot moves on, secondly the surrounding of the robot can vary on a wide scale e.g. a door can be opened in one moment, and closed in the other. Given these facts and the desire that the algorithm be fast enough were an inspiration to implement the PFB local navigation algorithm. Due to its properties, it can handle easily the quick changes of the environment, however, it is our task to enhance it.

As we know, the potential field based guiding uses the resultant of the vectors of repulsive powers and the one directing toward the goal. In our case these conditions are given, but we enhanced it in order to satisfy our expectations.

The robot is about to follow the path that was designed by the global navigation system, while it is feasible and there is no unexpected obstacles en route to the goal. Given the segment $p_ip_{i+1}$, which intersects one or more $o \in O$ (member of the set of obstacles). In this case our agent can not pass through these obstacles, so there is no option but switching to the local navigation system, while it reaches the end vertex of the segment. Nevertheless, this subsystem knows nothing about the global goal and if so, the agent could not avoid the obstacle. That is the reason why a temporary target must be chosen to the local subsystem which attracts the agent so the PFB navigation will work fine. The best solution is to choose the end vertex of the segment $p_ip_{i+1}$.

4.3 Quasi optimal graph regeneration

As the local navigation of the new hybrid system is not a complete one, the robot can get trapped in a local minima. In this case the global navigation system has to merge the contents of the obstacle memory and the set of static obstacles, regenerate the graph,
and find a new path on it. The components of the hybrid system we have discussed so far can all be run in real time, even on devices of low performance (e.g. microcontrollers), however, the regeneration of the map can take much time for the visibility roadmap algorithm.

The most trivial improvement is to only modify the graph and not start from scratch. This can be done as the following:

1. Find all original edges that intersect any new obstacle and remove these edges.
2. For all vertices of new obstacles, connect them with all new or old vertices that are visible from them.

Step one guarantees that every possible path on the graph is also possible path in real space. Step two guarantees that every optimal path in real space is also a possible optimal path on the graph. Unfortunately this method is still too complex to be used in real-time, because the running time is proportional to the square of the number of obstacles.

Our solution is to generate a graph that is simpler than the one defined by the visibility algorithm. This can be done by connecting new vertices only to vertices that are nearer to them than a certain distance. Of course the generated graph is not necessarily connected. In this case the distance has to be increased and the connecting of new vertices has to be redone. The whole algorithm is the following:

1. Find all original edges that intersect any new obstacle and remove these edges.
2. For all vertices of new obstacles, connect them with all new or old vertices that are visible from them and closer than a certain distance.
3. If the graph is not connected, then increase the distance and continue with step 2.

So far we have not dealt with the complexity of finding intersecting obstacles for edges. This can be implemented using binary space partitioning techniques or hash maps. Our algorithm uses binary space partitioning.

Binary space partitioning techniques [25] recursively subdivide a space into convex sets using hyperplanes. The resulting tree representation of space, called a BSP tree, was first used to accelerate polygon rendering, but can also be applied to collision detection. If the tree is balanced, then the time needed for finding a number of sets in the tree is
proportional to the logarithm of the number of elements in the tree. In our case, the tree is generated for the whole map and we store in each leaf the obstacles, that the convex set contains. Then, we use this tree to find areas that a certain edge intersects. As the tree can be built so, that each leaf contains at maximum a constant number of obstacles, we can find obstacles that an edge intersects in time proportional to the logarithm of the number of obstacles.

The BSP tree can be built using the following steps:

1. Sort all obstacles on the map using one of their coordinates.

2. Divide the obstacles into two groups, based on whether their coordinate is greater or less than a constant. The constant should be chosen to make the two groups equal in size.

3. If the resulting groups contain more obstacles than a constant parameter, then continue recursively on them, using an other coordinate.

The above steps build a balanced BSP tree, where each leaf holds at maximum a constant number of obstacles. This can be done offline for the map. New obstacles can be easily added online, by finding the leaf that should contain it, in time proportional to the logarithm of the number of obstacles, and adding the obstacle to this leaf. Then, finding intersection obstacles for an edge can also be done in running time proportional to the logarithm of the number of obstacles.

Testing the connectivity of the graph is trivial.

The resulting graph is not an optimal one, but is very close to optimal, thus the path found on it can be accepted as quasi optimal. The running time of the graph regeneration is roughly proportional to the number of new obstacles and the logarithm of the size of the map.

4.4 Path finding using modified D*

Whenever a local minima is encountered, a path has to be found on the regenerated map. As the number of these situations grow, so does the running time taken by path finding. To lessen the effect of this, a modified D* algorithm should be used instead of A*.

The original D* algorithm was created for modifying paths on graphs, from which some edges or vertices have been removed. However, in our case there can be new edges
and vertices added to the original graph. First of all, we should simplify the problem. Let $G$ be the original and $G'$ be the modified graph, then

$$G' = G \setminus \ E_{\text{deleted}} \cup \ V_{\text{added}} \cup \ E_{\text{added}},$$

where $E_{\text{deleted}}$ are edges that have been removed, $V_{\text{added}}$ are vertices that have been added, and $E_{\text{added}}$ are edges that have been added.

Adding new unconnected vertices is trivial, because the path does not have to be modified, as it remains a valid path and no shorter path is created. Thus we only have to deal with adding and removing edges.

The modified D* algorithm's update rule is the following: Whenever an edge is added or removed, the two affected nodes are added to a priority queue, which contains nodes to be updated. After all the modifications have been done, the algorithm processes the queue. As long as the queue is not empty, it always chooses that node, which has the minimal goal distance estimate. After a node has been processed, any of its neighbours, that point to it, are changed and added to the queue.

The initial path finding is done exactly as in the original D*.

Using this method, path finding in local minima situations can be done quickly.
5 EXAMPLES

The new algorithm has already been introduced, however, it is significant to compare it with the aforementioned solutions. For illustrating the performance of the introduced navigation technique, in the following, we show certain situations which can easily be solved by the new method.

![Figure 11: Path of the previous hybrid method](image1)

Figure 11 shows the path generated by the method described in [7]. The route is far from optimal and not all static obstacles are considered.

![Figure 12: Path of the new hybrid method](image2)

Figure 12 shows the near optimal path of the new hybrid method.
Figure 13 demonstrates the case when the agent does not know whether the door is closed or not. In this case the shortest path graph is generated by not taking the door into consideration (it can be opened) and calculating the path through it. However, when the robot concludes that it is closed it cannot move ahead. This is one case of local minima, because there is no progression to the goal. By generating a new graph, which includes the door as an obstacle, the problem is solved and another route can be taken, however, it is more expensive compared to the original one. This situation is represented by Figure 14. So even in case of local minima the new hybrid method can find a feasible way to reach the goal. This draw shows also the path of the agent to the goal, including
the avoidance of the unexpected obstacle: Dynamic Obstacle 1

A simple interactive application demonstrating some of the features of the new hybrid navigation system is available at http://www.laszka.hu/aron/tdk09navigation.
6 CONCLUSION

We have shown that there is a need for a universal navigation algorithm, presented a new hybrid navigation method, and demonstrated through examples, that it can be successfully applied in various situations.

In Section 1 we have introduced the problem of autonomous navigation. We have shown, that nowadays there is an ever increasing need for autonomous agents, and that these agents can not operate without suitable navigation algorithms.

In Section 2 we have clarified the definitions of the most important keywords used in this paper.

In Section 3 we have presented a comparison of navigation algorithms, both global and local. We have shown, that although global algorithms are complete, they can be used only in static environments, as they are not suitable for continuous replanning. We have also shown, that although local algorithms are fast, they are not complete. Thus, there are no universal solutions, and there is a need for a hybrid navigation method.

In Section 4 we have presented a new hybrid navigation method. The method combines the so called potential field based guiding (PFB), which is its local part, and a roadmap algorithm which is its global part, and a new concept, the graph regeneration based on the obstacle memory. As global navigation algorithms were designed for static environments, their performance is inadequate for multiply replannings during one navigation task. Thus we have developed a quasi optimal graph regeneration method and a path finding algorithm based on D*. The graph regeneration method creates a graph that is similar to the one defined by the visibility method, so the path found on it can be accepted as quasi optimal. We have shown that this regeneration method can be implemented in such a way, that its running time is proportional to the logarithm of the number of obstacles.

In Section 5 we have presented a number of examples to demonstrate, that the new hybrid method can cope with various situations. Simulations proved that it can be successfully applied in those situations too where obstacle and local minimal avoidance is the case. As a consequence we offer a new approach by means of which many unexpected conditions can be tackled.

The new method is universal and can successfully be applied in real situations in all walks of life.
References


List of Figures

1 Visibility graph of an example map ........................................ 11
2 Voronoi diagram and the selected path ................................. 12
3 Voronoi diagram of an example map ....................................... 13
4 Addition of a next point by the RRT algorithm ..................... 15
5 Example showing that the RRT algorithm is not complete ....... 15
6 Path of Bug 0 on an example map ........................................ 17
7 Path of Bug 1 on an example map ........................................ 18
8 Path of Bug 2 on an example map ........................................ 19
9 Block diagram of the new hybrid navigation system ............... 20
10 Data and control flow diagram of the new hybrid navigation system .... 21
11 Path of the previous hybrid method .................................... 28
12 Path of the new hybrid method ........................................... 28
13 Graph regeneration, part 1 ................................................ 29
14 Graph regeneration, part 2 ................................................ 29