Graph-based Path Planning for Mobile Robots

A Thesis
Presented to
The Academic Faculty

by

David T. Wooden

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

School of Electrical and Computer Engineering
Georgia Institute of Technology
December 2006

Graph-based Path Planning for Mobile Robots

For mobile robots,

whose capricious perceptions

are such a bother.

ACKNOWLEDGEMENTS

When I entered the graduate ECE program at Georgia Tech in 2003, I had accepted a position working on correlating gene interaction based on time-series expression data. An unexpected set of events later led me to work on compressing digital signals for low-bandwidth communication, ostensibly for image transmission between mobile robots. Then, before I knew what hit me, I found myself neck deep in a multi-million dollar competitive robotics project funded by DARPA. I consider myself extremely lucky to have ended up in a position that allows me such extraordinary experience with field robotics, especially given the precious little programming and practical experience I had when I started.

The GRITS lab has been a stimulating place, and having the trio of iRobot Magellan “trashcans”, the NREC Lagr pair of robots, and the Porsche Cayenne “sting” robot has left me unusually spoiled. There aren’t many places where there a fresh young student can dive right into top-of-the-line robot hardware. My advisor, Professor Magnus Egerstedt, has been a motivating and ever animated mentor, and to whom I offer my deepest thanks. He gave me the freedom to explore whatever kinds of solutions intrigued me and was an irreplaceable support in guiding me along the way. I cannot say enough good things about these past few years in his lab.

In a similar way, Professor Tucker Balch has been a great pleasure to work with (and to work for). He has made a lasting influence on me, particularly in how I look at field robotics and tackle complex projects. The white board discussions and the field tests have certainly helped shape this thesis and contributed enormously to my understanding of robotics.

I would also like to thank all my cohorts in the GRITS and BORG labs, especially Matt Powers, without whose partnership on the LAGR project much of the practical application of this thesis might literally not have ever gotten rolling. Our discussions – which ranged from basic programming to esoteric control architectures to how to make cheesecake from scratch – have been very valuable to me, and his suggestions have greatly improved the quality of the implemented versions of the work presented below.

Much of my work is a direct result of problems that needed to be solved for the LAGR project, so I am thankful to have been invited to be part of it and to have been able to interact with so many talented researchers. Professors Frank Dellaert and Jim Rehg were a lot of fun to work for. I remember one working lunch with Dr. Rehg (he paid!) where we sat and hashed-out parameters for our multi-input costmap in a gratifyingly principled manner. Of the fellow students on the project, there are too many to mention here, but I am happy for having been able to work with such a capable group of guys. I am also thankful to the Defense Advanced Research Projects Agency, for providing funding support (through Grant number FA8650-04-C-7131).

My family and friends have my gratitude for having always been so encouraging and supportive. My brother Chris in particular deserves special thanks for being an indelibly positive sounding board for my ideas (as well as for providing Room and Board!).

Finally, I want to thank Alison, my loving wife for being so understanding of my ridiculous hours and for reminding me to eat. I love you, sweetheart!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In this thesis, questions of navigation, planning and control of real-world mobile robotic systems are addressed.

Chapter II contains the first contribution in this thesis, which is a modification of the canonical two-layer hybrid architecture: *deliberative* planning on top, with *reactive* behaviors underneath. *Deliberative* is used to describe higher-level reasoning that includes experiential memory and regional or global objectives. Alternatively, *reactive* describes low-level controllers that operate on information spatially and temporally immediate to the robot. In the traditional architecture, information is passed top down, with the deliberative layer dictating to the reactive layer. Chapter II presents our work on introducing feedback in the opposite direction, allowing the behaviors to provide information to the planning module(s).

The path planning problem, particularly as it as solved by the visibility graph, is addressed first in Chapter III. Our so-called *oriented visibility graph* is a combinatorial planner with emphasis on dynamic re-planning in unknown environments at the expensive of guaranteed optimality at all times. An example of single source planning – where the goal location is known and static – this approach is compared to related approaches (e.g. the reduced visibility graph).

The fourth chapter further develops the work presented in the Chapter III; the oriented visibility graph is extended to the *hierarchical oriented visibility graph.* This work directly addresses some of the limitations of the oriented visibility graph, particularly the loss of optimality in the case where obstacles are non-convex and where the convex hulls of obstacles overlap. This results in an approach that is a kind of middle-ground between the oriented visibility graph which was designed to handle dynamic updates very fast, and the reduced visibility graph, an old standard in path planning that guarantees optimality. Chapter V investigates path planning at a higher level of abstraction. Given is a weighted colored graph where vertices are assigned a color (or in other words *class*) that indicates a feature or quality of the environment associated with that vertex. The question is then asked, "what is the globally optimal path through this weighted colored graph?" We answer this question with a mapping from classes and edge weights to a real number, and use Dijkstra's Algorithm to compute the best path. Correctness is proven and an implementation is highlighted.

CHAPTER I

BACKGROUND

1.1 Introduction

The problem of controlling a mobile robot and planning its trajectory is dominated by the fact that the robot's knowledge of the world is constantly in flux. This is caused by the variation of the environment itself, by error in the robot's reckoning of its own position over time, and by its poor ability to make accurate and precise measurements of the environment. Complicating the problem further is the complexity inherent in a robotic system. The robot must integrate sensor information about itself and its environment from a large variety of sensor modalities which are fraught with error. Its control actions are limited and constrained. It must guarantee its own safety in real-time in the presence of sudden and unexpected changes in the environment, while completing its mission with as little cost or in as little time as possible. Moreover, the robot must learn its environment as it encounters it.

As the field of robotics has developed, increasingly more of these challenges have been addressed at a time. In the 1950s, the earliest robots were able only to achieve basic locomotion. In the 1970s and 80s, the optimal completion of global objectives dominated the active research. Later, real-time constraints gained attention at the cost of optimal planning. Field robotics has moved in recent years out of controlled laboratory and office environments into outdoor environments where less and less *a priori* knowledge of the structure of the world can be assumed. Sensing has shifted from SONAR and IR, to accurate and modellable LIDAR scanning, to color vision, both within and beyond stereo range. Moreover, robots have accelerated drastically, from 1969 Shakey's 2 meters/hour [53] to 2005 Stanley's 70 miles/hour.

The robotics community continues to be challenged by several issues:

- unknown unstructured environments,

- sensor errors,
- control constraints,
- real-time guarantees,
- global objectives,
- dynamic obstacles

This chapter describes background work in relating to these issues.

1.2 Control Architectures

In most cases, the overall robotic system is decomposed into separate interacting components, allowing for – as much as is possible – each of the issues listed above to be addressed independently. Figure 1.1 illustrates a very simple architecture for decomposing the overall robot control system.

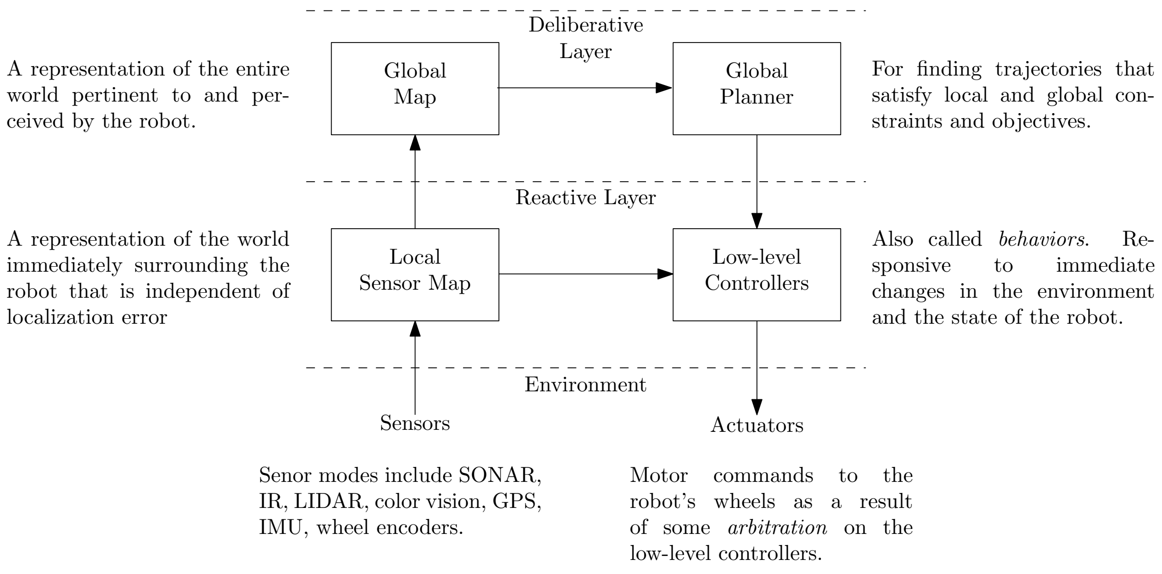

Deliberative Layer
Global Map
Global Planner
A representation of the entire world pertinent to and perceived by the robot.
For finding trajectories that satisfy local and global constraints and objectives.
Reactive Layer
Local Sensor Map
Low-level Controllers
A representation of the world immediately surrounding the robot that is independent of localization error
Also called *behaviors.* Responsive to immediate changes in the environment and the state of the robot.
Environment
Sensors
Actuators
Senor modes include SONAR, IR, LIDAR, color vision, GPS, IMU, wheel encoders.
Motor commands to the robot's wheels as a result of some *arbitration* on the low-level controllers.

Figure 1.1: Standard Hybrid Control System Block Diagram.

This canonical architecture provides two layers (using the notation of Arkin [3]): reactive and deliberative. The reactive low-level controllers are decoupled from the deliberative planner, allowing the controllers to address real-time constraints while the planner addresses global objectives. The idea behind the low-level controllers is that each controller, or *behavior*, uses a small set of sensor measurements with a simple reasoning process to control the robot for some narrowly-defined semantic purpose. For example, a behavior may be dedicated to preventing the robot from driving into suddenly appearing and unanticipated obstacles through laser data, or to forbid the robot to execute motion commands that would cause it to roll or crash. This *behavior-based* method of controlling a robot has received great attention in the robotics community [4, 7, 8, 13, 22, 28, 52, 61].

The global and local maps are used to handle sensor and localization error. The intent is that the local map is consistent and unaffected by localization error. The incremental map construction process is to be done in a such a way as to account for sensor error as well as to incorporate new or changing information about the environment. The global planning block is intended to dynamically re-plan given the global map's changing opinion about the world.

The two layers in Figure 1.1 are separated both in terms of time horizon and spatial extent. The reactive layer meets real-time constraints, while the deliberative layer is likely to require computation time dependent on the problem size. The reactive layer considers only "local" information, while the global layer considers the entire world. This two-layer approach is appropriate and sufficient for the work later described in this thesis. Note, however, that it is also common to divide the architecture into additional layers, allowing for finer control of how far and for how long a planning algorithm may operate. See for example [29, 30]. In [36], the deliberative layer is split into a regional waypoint planner which operates over some distance, and a subregional planner that provides precise paths for the underlying control layer.

Note the one-way flow of information: from the sensors, through the mapping processes, through the planning block, down into the controllers block, and finally out to the motor control. Our work described in Chapter 2 modifies this architecture to include a feedback mechanism from the controllers to the global map, inducing bidirectional information flow in the system.

1.3 Path Planning

1.3.1 Introduction

Over the historical development of robotics, the preferred manner in which planning has been incorporated into robotic architectures has changed, and can be divided into a few distinct time periods. The earliest forays into robotics were able only to attack a small part of the navigation problem; the technological challenges were too difficult to do anything more significant. Roboticists focused on building systems that could demonstrate some semblance of intelligence. Perhaps the first such example is from 1953, W. Grey Walter's *turtles*, Elmer and Elsie [74]. They were very simple rolling machines that sensed light and exhibited phototaxis (the behavior of driving towards or away from a light source). They stored no information about the environment, and operated purely on what their sensors detected instantaneously. Hence, planning was omitted entirely. This work focused on basic behavioral functionality without any notion of (complicated) global objectives or control constraints.

In the 1970s, the classic Artificial Intelligence approach became popular, where planning was the focus of extensive study. Known as the *deliberative* approach [3] (also as *sense-plan-act* [30]), the robot would make a sensor sweep of the environment, plan the optimal (in some sense) action to take next, and then execute that plan until the next sensor sweep finished. Deliberative methods often use a complex model of the environment and the robot, especially within the classic AI framework. Consequently, such planning has a fatal flaw: it often takes too much time and while the planning process runs, the world changes. As a result, the plan is often immediately invalidated, and the planning effort wasted. Moreover, this approach calls for an accurate and precise model of the environment and robot that is typically impossible or too expensive to come by. Under this paradigm, global optimality was emphasized to the exclusion of real-time effectiveness.

In the 1980s, the *reactive* (or behavior-based) approach attracted attention again, most famously with the work of Brooks [13, 14] who focused on constructing robots that entirely omitted planning. While attractive because each behavior is based on minimal modelling of the robot and the world, methods inspired by or related to this approach are subject to the presence of local minima and are unable to escape from simple traps in the environment [8, 63]. That is, a behavior-based robot, without the benefit of memory or spatial knowledge of sufficient extent would become stuck in a cul-de-sac or caught "like a fly at a window". The lesson: memory, modelling, and planning to some degree are generally needed.

In recent years, activity in robotics has swelled. Technological advances such as dramatically increasing computational power, memory size and communication bandwidth have lessened some of the old challenges of robotics. Furthermore, the community has more and more experience with the hardware challenges of building reliable robust platforms that can run for extended periods of time even in ever harsher environments. There exists also a push by funding agencies to produce practical robotics that run without human assistance in outdoor unknown environments, with sensing based on vision, and with the task of operating in a complex active environment with long-term global objectives. Such projects include PerceptOR, the Learning Applied to Ground Robots project, the DARPA Grand Challenge I and II, and most recently DARPA's Urban Challenge.

One of the lessons learned from the community is that the desire for optimal planning is outweighed by the need for a "good" plan now [41]. As a result, approximate solutions to the problem of path planning which meet the time-critical constraints of the robot may be (and often are) preferable to a slow optimal one. Chapters 3 and 4 follow this theme with our work on path planning.

1.3.2 Problem Formulation

We formulate the general path planning problem as follows. Given is the state of the robot, partial knowledge of the environment, and some model of how the robot interacts with environment, what is the minimum cost path between where the robot is and where the robot is told to go to?

This question is answered in this thesis with two different flavors. For Chapters 3 and 4, the problem is narrowed to finding a path in a Euclidean plane with a static goal location. Given is a dynamic set of polygons that represent obstacles the robot must avoid. For Chapter 5, path planning is done at a higher level of abstraction. Instead of having polygonal obstacles, distinctive features or places are given that belong to a small set of classes.

For both approaches, we will describe the robot's environment as a weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W_E)$ where

- $\mathcal{V}$ is the set of vertices (with which we will typically associate a position (physical location) in $\mathbb{R}^2$),
- $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is a set of directed edges (ordered pairs of vertices), and
- $W_E : \mathcal{E} \rightarrow \mathbb{R}^+$ is a cost associated with each edge.

Given the graph, a minimum cost path is found between the vertices representing the robot and the goal (if such a path exists). The path planning problem is, therefore, composed of two subproblems:

1. `generate-graph` - calculate or determine the vertices, edges and weights of the graph, and
2. `find-path` - find the minimum cost path over the edge weights between the start and goal vertices.

1.3.3 General Assumptions

In this section, we present some common terminology for describing robots and the world, as well as some assumptions generally accepted for global path planning.

The environment that the robot operates within is known as the *workspace*. We use the common assumption that it is populated by two kinds of objects:

obstacles - places where the robot cannot or certainly should not go, and

free space - places where the robot is free to move.

The workspace throughout this thesis will be $\mathbb{R}^2$.

For the purposes of planning, it is convenient to model the robot as a point in the plane that is free to drive in any direction. So, rather than planning over the workspace directly with a true physical model of the robot (e.g. with length, width and possibly dynamical constraints), the robot is condensed to a point and the workspace is transformed into what is known as the *configuration space*. This concept was introduced in the influential work by Lozano-Perez and Wesley [50, 51].

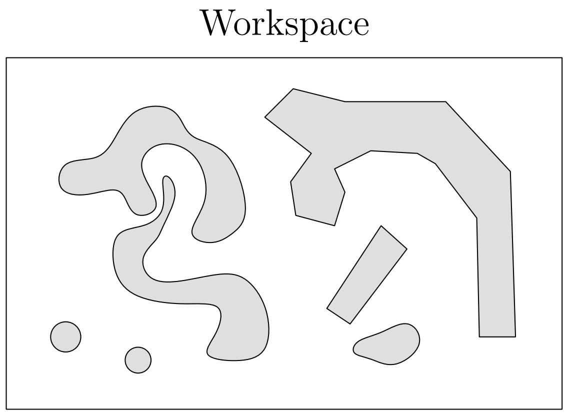


Figure 1.2: Workspace illustration. Shaded polygons represent obstacles.

The configuration space C is the same as the workspace, but the obstacles have been "bloated" by the footprint of the robot. That is, each obstacle in the workspace is transformed by an operator into a configuration space obstacle. This operator is the Minkowski sum, or mathematical morphology dilation, and is the convolution of the robot footprint with the obstacle footprint. The part of the configuration space occupied by obstacles is called C_{obs}. The remaining part, available for the robot to find a path, is known as C_{free}. (This terminology follows from [44]).

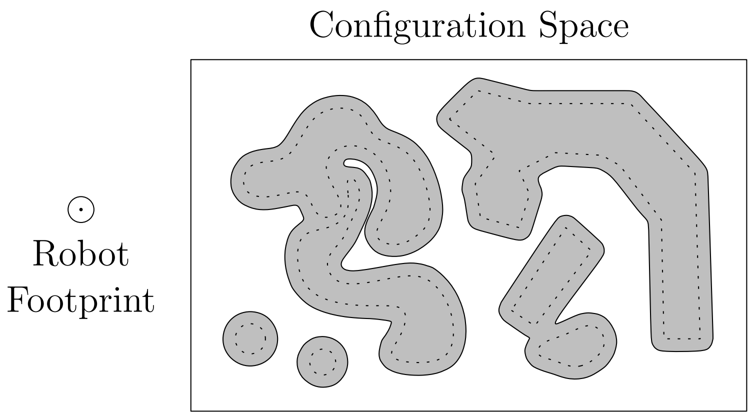


Figure 1.3: Illustration of morphology dilation of robot footprint.

So as to keep the planning problem tractable it is important that the space over which planning is computed be two dimensional; even in two dimensions, global path planning is computationally burdensome. Orientation, the third dimension of a planar robot, is then typically ignored, but this leads to complications in how to represent the robot's footprint and what C_{obs} is. To ignore orientation implies the footprint must be radially symmetric, which in general it is not. The right choice for approximating the footprint is not obvious. In Chapter 2, we address this conflict and present a solution.

Planning in the configuration space also ignores dynamic constraints to which the vehicle is subject, again for the sake of tractability of computing the solution. As previously stated, we make the assumption in this thesis that the low-level controllers are able to take the planning algorithm's output and transform it into motor commands that are safe for the robot both in terms of the presence of obstacles as well as the possibility of dynamic catastrophes (such as rolling the robot).

To summarize, the robot is a 2D point (without orientation) and impassable obstacles are dilated to form the configuration space, hence alleviating kinematic considerations. Dynamic constraints are left to be addressed by the reactive layer in the overall architecture. The path planning problem is described in two parts (which may be solved for simultaneously): `generate-graph` and `find-path`. The graph $\mathcal{G}$ is to be generated from sensor or map data, and a path is planned through this graph, from the robot to the goal.

The `find-path` problem is solved in one of three ways. The most general is the *all-pairs* problem, where the task of the planning algorithm is to find (optimal) paths between every pair of vertices in the graph. Much more common in robotics is the *single-source* problem where the goal location – the location which the robot is tasked with driving to – is known and in a fixed position. Paths are planned from every vertex in the graph to the goal. For the *point-to-point* problem (also known as *agent-centered search*), path planning is only needed between the robot and the goal point. Both points are known, and the goal point is assumed to be static.

1.3.3.1 Big-O Notation

Big-O notation is a well-known description of the time required to determine the solution to a problem given the size of the input or output (or both). Useful because it hides the effect of system-specific implementation details, it is a representation of the worst-case running time of an algorithm. While very important in providing a notion of the time upper-bound an algorithm may require, the constant terms that are hidden by this notation may be significant or even of primary concern in a real robotic system. Moreover, the worst-case time that this notation represents may occur in practice only very infrequently.

Two list sorting algorithms, heapsort and quicksort, provide an example of this observation. Heapsort has a running time of $O(n\ log(n))$. Quicksort has a running time of $O(n^2)$. In practice, however, quicksort is often faster, and certain design choices can be made to dramatically reduce the possibility of the worst-case quadratic running time.

In addition to running time, Big-O notation is used to describe the space required during the execution of an algorithm. Also, other important but less common notations are $\Theta()$, which represents a tight bound of the running time, and $o()$, which indicates that the term inside the parentheses is negligible.

1.4 Types of Planning

The centerpiece of this thesis is the *oriented visibility graph* (Chapter 3), which takes a combinatorial view of the robot and its environment. The path planning problem is typically approached using a method of one of three categories: search-based, sampling-based, or combinatorial (Figure 1.4).

By far, search-based methods currently dominate path planning in field robotics. The popularity of this method can be attributed first to the relative ease of its implementation and second to the early establishment of dynamic search-based algorithms. The sampling-based methods are relatively new to robotics and are the subject of a rush of recent research. These methods are particularly useful for planning of serial manipulators and in other situations where higher-dimensional planning is required. The combinatorial methods are the oldest and perhaps most studied branch of planning, having applicability in many areas,

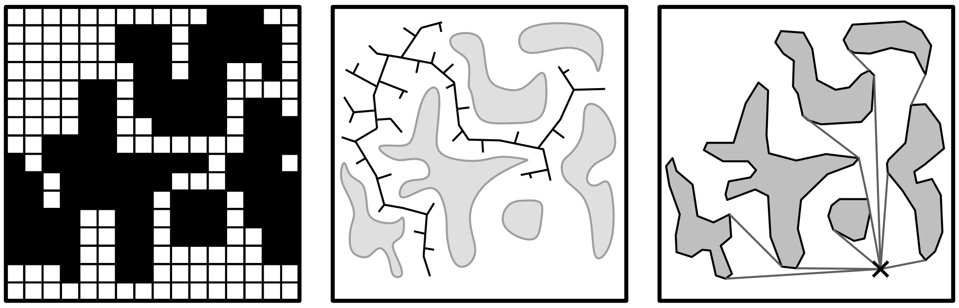

(a) Search-based (b) Sampling-based (c) Combinatorial

Figure 1.4: Illustration of Three Planning Methods.

ranging from computer graphics to VLSI design.

1.4.1 Search-based Planning

The basic idea behind search-based planning is this: a grid of regularly sized grid cells is used to represent the configuration space. The goal and robot locations are known within the grid and a search is run on the grid to solve the point-to-point `find-path` problem.

1.4.1.1 Algorithms

Search-based planning has benefited from a series of technological advances. The first algorithm was Dijkstra's Algorithm, which is simply a breath-first search [17]. In 1968, the A^* algorithm was introduced which uses an *admissible heuristic* to narrow the search. As long as a better-informed heuristic is not used, the A^* algorithm will find the solution as fast as or faster than any other method. A^* is a static algorithm, which means that when the configuration space changes (such as when an obstacle or the perception of an obstacle changes), the old path is invalidated and the A^* algorithm must be re-run from scratch [33].

The mid-1990s saw the introduction of the first dynamic search-based algorithm, D^* [69]. It is this algorithm (and its derivatives) that has been the basis of planning for the vast majority of field robots. *Focused* D^* and *quad-tree* D^* (among others) were introduced to address the time-complexity and space-complexity limitations of the D^*, respectively [16, 70, 82].

Within the past decade, a slightly different approach to the A^* algorithm was introduced by Likhachev and Koenig, called LPA^*. This has led to series of algorithms, including D^*-lite, anytime A^*, real-time A^*, and others [42, 48, 49]. (The name "lite" is perhaps an unfortunate choice, as the algorithm is indeed not a diluted or handicapped version of D^*.) D^*-lite is of particular interest because it is behaviorally very similar to the focused D^* algorithm, but D^*-lite is much simpler to code, understand, and prove properties of [41]. In fact, Stentz's lab (where D^* was introduced) uses D^*-lite in many of its current implementations in place of D^* [23].

1.4.1.2 Limitations

The success of search-based methods should not be underestimated. However, this approach is subject to certain limitations. The search is done on a grid which is useful because doing so removes the burden of having to maintain the graph structure. (That is, the `generate-graph` problem is solved trivially). The grid is a graph with a fixed topology. On the other hand, the resolution of the grid must be specified. Grid-based methods are subject to *resolution completeness*, meaning that the resulting optimal path is only optimal at the resolution of the grid employed. There exists a trade-off between increasingly finely resolved grids (and hence paths closer to optimal) and resources required for computation time and memory. A complete method, on the other hand, always returns a path through C_{free} (if such a path exists).

A regularly spaced grid can be decomposed into a hierarchical tree structure (as a quad-tree or framed quad-tree) by incrementally subdividing sectors of the search space (Figure 1.5). (Note however that this makes the `generate-graph` problem no longer trivial.) This approach has been successfully employed, but does increase the burden imposed by the algorithm in terms of coding cost and computation time [24].

Finally, we consider it a disadvantage of the search-based methods that the configuration space is represented with a grid. This representation seems not biologically inspired and we take the position that problem solving in robotics benefits from a natural correspondence with the manner in which path planning is solved by humans.

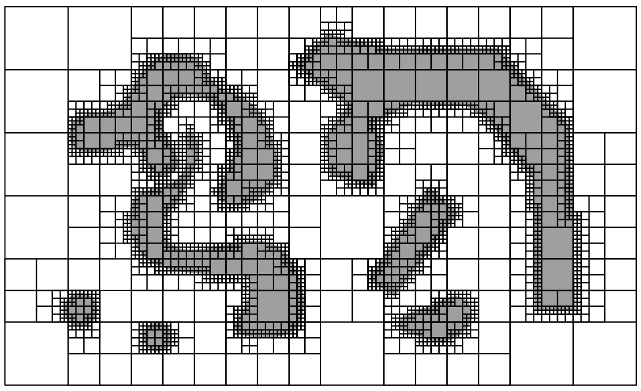


Figure 1.5: Illustration of a quad-tree decomposition.

1.4.2 Sampling-based Planning

The sampling-based methods solve the `generate-graph` problem with the following general form. (1) a new vertex v is generated. (2) A check is performed to determine if v is in the interior of an obstacle. If it is, either the cycle restarts or the vertex is somehow projected to the boundary of the obstacle. (3) When a good vertex is known, edges are added between it and its mutually visible immediate neighbors (usually its k-nearest neighbors or those neighbors in some neighborhood of v). Eventually, the graph contains a path between the start and goal vertices. When a good path is known, A^* search is run over the graph to solve the `find-path` problem.

Vertex generation was originally done on a probabilistic basis when this approach was first introduced as the *probabilistic roadmap* (PRM) [9,38,39,56], but the framework allows for it to be done deterministically as well. A similar method called the *rapidly-exploring random tree* (RRT) has also demonstrated success [45,46]. This type of approach has been proven to be highly effective in higher-dimensional spaces. In addition, recent work has been done to make this approach dynamic [25,84].

An attractive quality of the RRT is the overall simplicity of the method. There are two challenges in implementing this approach. First, a *kd-tree* must be implemented (or a library found). The kd-tree is a data structure that sorts k-dimensional vertices and allows for nearest-neighbor and range searching operations. Second, a method `visible(`v_1`,`v_2`)` must be written that returns whether an edge between vertices v_1 and v_2 is blocked by an obstacle.

Sampling-based methods rely on the notion of either resolution or *probabilistic completeness.* That is, these methods return an optimal solution with probability approaching 1 as the number of vertices increases.

As with search-based methods, the principle of randomly adding vertices to a graph and testing for connectivity is perhaps not the most natural or biologically-inspired approach to planning. This does not, however, take away from its apparent effectiveness, but may lead one to look for a method that is as fast yet more natural.

1.5 Combinatorial Planning

Combinatorial path planning essentially takes as input a polyhedral representation of the environment and augments it with additional edges or vertices. The result is a graph known as a *roadmap.* As with sampling-based planning, the shortest path is found between the goal and the robot. So, whereas the sampling-based method populated free space with new vertices, combinatorial methods use the boundary of the polygonal obstacles directly.

These roadmaps come in three flavors: shortest-path, maximum clearance, and cell decomposition. The shortest-path roadmap is the *visibility graph*, where edges are added between every mutually visible pair of vertices. This approach has been used for a wide variety of applications, including computer graphics, art gallery problems, VLSI design, and in pursuit-evasion problems for multiple robots. The visibility graph was the planning method used by the famous 1960s robot, *Shakey* [53].

1.5.1 Shortest-path roadmaps

1.5.1.1 Visibility Graph

The visibility graph is formed by augmenting a given weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W_E)$ with an additional set of edges $\mathcal{E}_v$:

$$\mathcal{E}_v = \{(v_i, v_j) \in \mathcal{V} \times \mathcal{V} \mid \mathit{visible}(v_i, v_j, \mathcal{E}_s)\}, \tag{1}$$

where *visible* returns true if the edges in $\mathcal{E}_s$ do not intersect the hypothetical edge (v_i, v_j). In other words, an edge is added to $\mathcal{E}_v$ for every pair of vertices in $\mathcal{V}$ that is mutually visible.

$\mathcal{E}$, the set of edges in $\mathcal{G}$, is partitioned into $\mathcal{E}_s$ – the edges that form the boundaries of the polygonal obstacles – and $\mathcal{E}_v$ – the augmented edges added by the visibility graph algorithm.

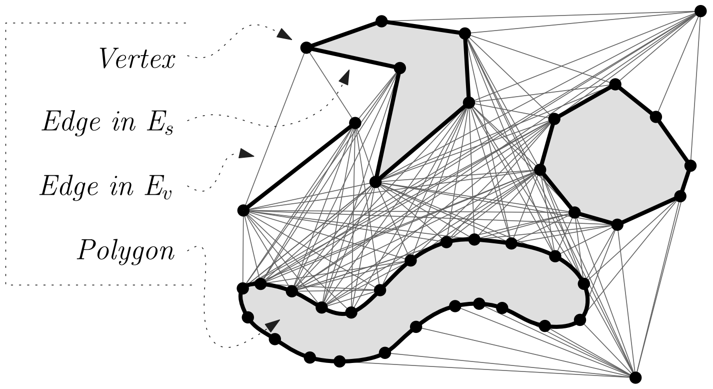


Figure 1.6: A sample visibility graph.

From the point of view of path planning, this results in a ridiculously dense graph. Recall that our purpose for this graph is that the shortest path through free space to the goal is contained in $\mathcal{E}$ (to the single-source planning problem). Even for the all-pairs problem (finding a path between every pair of vertices), most of the edges in the classic visibility graph are unnecessary.

First used in [53] is the *reduced visibility graph* (RVG). Here, $\mathcal{E}_v$ is redefined to be only those edges which are essential (for the all-pairs planning problem) based on local necessary conditions [44]. To distinguish between the classic visibility graph and the reduced visibility graph, we will refer to the former as the *full visibility graph* (FVG). A more detailed treatment of the RVG is given in Chapter 3, where we describe our related approach to path planning.

1.5.1.2 Similar Data Structures

A number of related structures have been presented in the literature that contain more or less the same information as the visibility graph. The visibility polygon is the subgraph of $\mathcal{G}$ that is visible from a given vertex [6,34,55]. The visibility tree is a subset of the visibility graph between the robot and the goal point. In [5], an algorithm is presented for a moving robot in a static environment using the visibility tree. The visibility complex (described

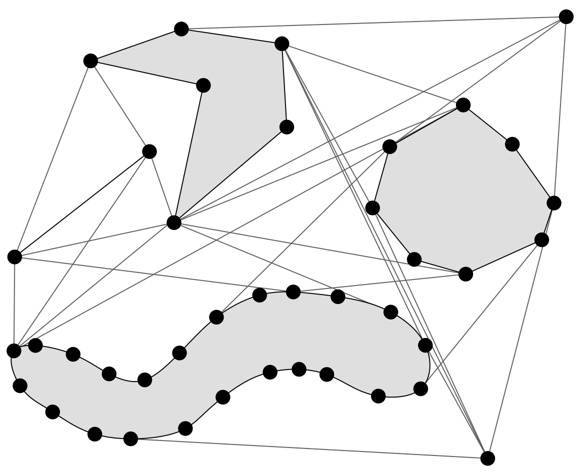


Figure 1.7: A sample reduced visibility graph.

in [59, 64]) is limited to convex polygons but contains more information than the visibility graph.

A related approach to path planning is the shortest-path map. The concept behind this approach is that of propagating a wave outwards from the goal point. This is done in C_{free} (rather than over the roadmap produced by the visibility graph). The algorithm presented in [35] is not dynamic, but does have an optimal $O(n\ log(n))$ complexity for static environments and also requires $O(n\ log(n))$ space complexity. The algorithm itself is complicated: "wavelets" are propagated through a quad-tree representation of the configuration space. Within each quad-tree cell, a Voronoi diagram (see Section 1.5.3) approximates connectivity in the graph, and path costs are computed simultaneously with the wave propagation.

1.5.2 Limitations of the Visibility Graph

Shortest-path roadmaps have traditionally been plagued by the undesirable effect that the optimal paths skirt the boundary of obstacles. This often causes a natural temptation to expand the robot footprint marginally to place the optimal path further from any obstacle. This, however, has the undesirable effect of potentially closing off goods paths for the robot through tight (but acceptable) corridors.

We take the approach in this thesis that the two-layered architecture described above ameliorates this issue. The global planner instructs the behaviors to drive at the boundary of an obstacle (which is on the shortest path to the goal). The behaviors adhere to this instruction in so much as the behaviors' notion of safety with respect to proximity to an obstacle or dangerous slope is not violated. Hence, the visibility graph can be used to produce a complete solution for path planning problems in two dimensions.

To extend the visibility graph as a roadmap for more than two dimensions in infeasible. Point-to-point planning in three dimensional configuration spaces with polyhedral obstacles is shown in [15] to be NP-hard. However, a number of approximate solutions have been proposed [50, 58, 62].

1.5.3 Other roadmaps

It would be impossible to provide an adequate introduction to the myriad other planning techniques used in robotics. Inasmuch as methods outside of the visibility graph are not a component of the work given below, we merely refer to other sources for such alternative approaches. Perhaps the most significant (especially in the Computational Geometry community) is the Generalized Voronoi Diagram (or Dirichlet Tessellation or retraction method), which provides *maximum clearance* roadmaps. Other important methods include the vertical cell decomposition, cylindrical algebraic decomposition, Canny's roadmap (or silhouette), and elastic strips, bands, and roadmaps. For general reference on these approaches (including the visibility graph), see [18, 68]. For an introduction geared more to robotics, see [44, 46]. Roos's PhD thesis [65] describes dynamic Voronoi diagrams. Elastic bands, strips, and roadmaps are described in [11, 60, 83].

1.6 Preview of Contributions

The work in this thesis is geared towards not just questions of optimality and worst-case complexity, but also practicality, and implementability, and real-world efficiency. Hence a theoretical look at the problems we address is given, as well as a serious effort at implementation in non-trivial application domains. In satisfying the multitude of objectives for a mobile robot – time-criticality, global objectives, quick response time, etc – a multi-layered control architecture has become a standard in system-level design. As a component of this architecture, there is typically a planning block. The work presented in this thesis is focused on this block, both in terms of its interaction with the underlying reactive layer, as well as with its internal workings.

The robotics community has devoted an enormous amount of energy at solving the path planning problem. There exist many algorithms for solving the `generate-graph` and `find-path` problems that solve for optimal or approximate paths in either optimal time, optimal space, or both. For the shortest-path planning problem, a gap remains for algorithms that accept dynamic revision of the graph and that are fast enough to meet the time-critical constraints in a real-world robotics application. Our work fills this gap and is described in this thesis in four chapters:

II - Simultaneous control and mapping

We extend the canonical two-layer hybrid architecture to include feedback from the reactive controllers to the deliberative planner. This provides a robust framework for handling configuration space ambivalence, sensor noise, and localization error. The feedback method is described and applications support the utility of the approach.

III - The oriented visibility graph

We develop a dynamical global combinatorial path planner for a mobile robot in the plane. This planner fits within the framework described in Chapter II. We present the algorithm in detail as well as its running time complexity. An extensive application supports the utility of the approach.

IV - The hierarchical oriented visibility graph

We further develop the oriented visibility graph, putting it into a hierarchical framework. We present the algorithm along with its running time complexity. Our implementation, an interactive simulator and graphical user interface, is described.

V - Path planning over colored graphs

We show how to plan globally optimal paths over weighted colored graphs. Here, the coloring is given as a representation of preferability of the corresponding terrain. Optimality of the paths is proven and a discussion of an implementation is given.

CHAPTER II

SIMULTANEOUS CONTROL AND MAPPING

Layered hybrid controllers typically include a planner at the top level with reactive control at the lower levels. The planner considers the state of the robot in a global context. The low-level controllers consider only the local environment of the robot and are able to operate at a high frequency to ensure the safety of the robot. Also, it is often the case that the low-level controllers consider more aspects of the robot's state (e.g. kinematic constraints) than the planner. The consideration of such constraints at the planning level would prohibitively increase the state space the planner must consider and, accordingly, its running time and complexity. In this chapter, we investigate how we can take advantage at the planning level of domain knowledge encapsulated in the lower level controllers, and we introduce SCAM (Simultaneous Control and Mapping), a feedback mechanism that enables low-level controllers to influence the high-level planner.

2.1 Introduction

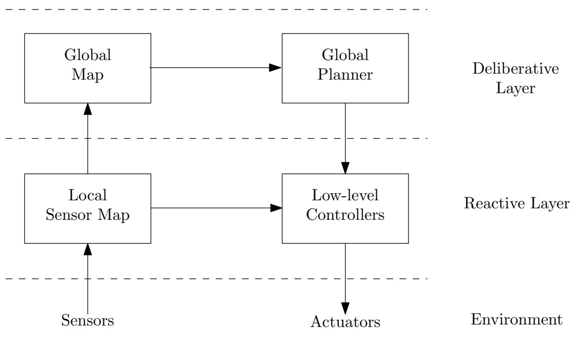


Figure 2.1: Standard Hybrid Control System Block Diagram.

A canonical two-layer control architecture that is common in mobile robotics [3,28,61,81] is illustrated in Figure 2.1. The controllers in the *reactive* layer concurrently compute control laws and each is defined with respect to its own tasks, sensory inputs, and operating points. Each controller could, for instance, produce an output of votes and vetoes for commanding the robots movement, based on the DAMN architecture [66] . In this chapter, we introduce a novel approach for allowing feedback information to flow "backwards", i.e. from the controllers to the map. In fact, we will let the veto mechanism (which blocks unsafe directions for the robot) trigger feedback signals to the *deliberative* layer.

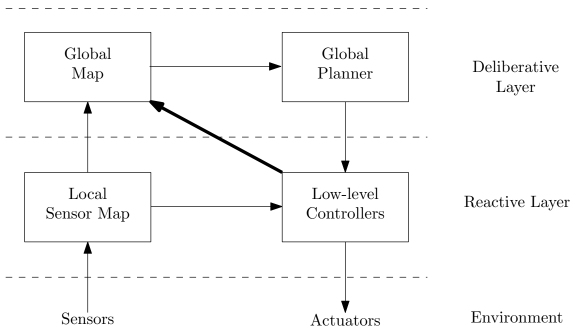


Figure 2.2: SCAM Block Diagram.

This feedback, depicted in Figure 2.2, represents a new pathway for information flow in the layered architecture. In the standard architecture, information passes from the repository of map-based data to the controllers. Our architecture, however, is bidirectional, adding the ability of the controllers to pass information back into the global map. Hence, this chapter describes a framework that simultaneously controls the robot and maps the environment.

The standard two-level architecture is employed in robotics applications for two reasons. First, the controllers must operate on a short time scale in order to guarantee that the robot is kept in a safe and allowable state. By decoupling the controllers from the mapping and planning processes, the deliberative layer is afforded more flexibility in terms of cycle regularity and frame rate. Second, from a complexity management point-of-view, the map is typically planar, i.e. contains a 2DOF description of the environment, while the controllers (which operate on a smaller spatial scale) can take the full kinematics and dynamics of the robot into consideration. For example, yaw, pitch and roll can be considered together with position, resulting in a higher dimensional configuration space. However, the planning process can barely keep up with real-time constraints in a planar world, and any attempt to plan paths through the full configuration space would be infeasible. As such we propose to project high-dimensional “obstacles” detected at the controller-level into the planar map.

Our overall approach is devised as follows. A global path planner continually re-plans based on updated sensory input incorporated into a global map and passes the path to the low-level controllers. We suggest the use of an “optimistic” planner, where the configuration space of the robot is reduced slightly, making the planner’s conception of the robot holonomic. The controllers, on the other hand, are able to operate on a precise and accurate model of the robot, and with the corresponding configuration space make better informed decisions about what local regions are better to travel over.

The outline of the chapter is as follows. In Section 2.2, we discuss previous work related to this chapter. Section 2.3 expounds on the configuration spaces of the two layers. Section 2.4 describes the general mapping, planning, and control components of our experimental system. Section 2.5 explains in more detail the operation of the SCAM framework. Section 2.6 describes experiments to support our approach. Finally, Section 2.7 discusses future work, followed by conclusions in Section 2.8.

2.2 Related Work

The dynamic window approach (DWA) is a method for accounting for the robot’s velocity and acceleration capability in a reactive-layer framework [26, 27]. This approach considers a short time window (e.g. 0.25 seconds) and, using knowledge of the robot’s current translational and rotational speed and maximum translational and rotational accelerations, a kind of velocity configuration space is computed. During this brief time period, the robot considers short arcs of constant curvature. Essentially, the DWA is a low-level controller that, as such, could be incorporated into the reactive layer described in this chapter.

In [10], the authors describe a method of using the DWA in combination with a local objective function and partial global path planning. This approach is attractive because it directly addresses the need for global path (re)planning in an unknown environment, but path planning is only done on a limited portion of the entire global space. The major difference described in this chapter is that we provide a mechanism for bidirectional communication between any global path planner (A^*, D^*, etc) and any suite of low-level controllers (possibly including DWA).

The standard two-level deliberative/reactive system architecture is well known. A general discussion of this type of architecture (and others) as well as robots that have used it is given in [3]. Our system architecture extends this approach by adding a new line of communication from the low-level controllers back up into the global map and plan.

2.3 Configuration Spaces

Central to this work is that different layers in a hybrid control architecture can use different configuration spaces. In fact, the configuration space is a geometric encoding of of all achievable poses, defined at a certain level of abstraction, with respect to the robot's kinematic constraints and, possibly, with respect to external constraints (i.e. obstacles). Changing the footprint of the robot necessarily changes the configuration space of the robot, as such a change forces the robot into a different set of configurations.

The particular implementation we will use to highlight our architectural ideas is one in which the robot uses a differential drive control mechanism, powered by two wheels toward the front of the robot, with casters in the back (Figure 2.3a). Translational drive is achieved by driving the two wheels together, while rotational drive is achieved by the difference between the velocities of the two drive wheels. By driving the two wheels at equal speeds but in opposite directions, the robot can turn in place.

To reduce the computational burden, the map-based planning layer uses a two-dimensional representation of the robot, which idealizes the robot's footprint as a circle with a diameter equal to the width of the robot at the drive wheels (Figure 2.3b). This simplification of the configuration space is valid as long as the robot is driving straight ahead, but underestimates the size of the robot as it turns, especially as it turns in place. In fact, because of the choice of a circle as the idealized footprint, rotational kinematics are not taken into account at all. The effect is that the planning layer is "optimistic" about the robot's capabilities

Because the low-level control layer is working over a smaller spatial and temporal window, this can afford to use the full configuration space (Figure 2.3c). (In other words, in evaluating the robot's kinematic constraints, one needs the robot's x and y Euclidean

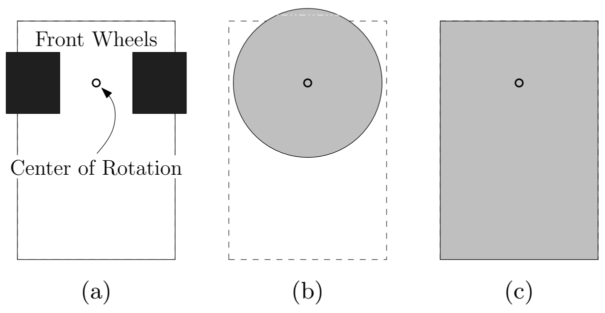


Figure 2.3: Footprints used for the LAGR robot. (a) A diagram of the robot (pointed up) and its center of rotation. (b) The "optimistic" footprint used by the planning process. (c) The "pessimistic"/accurate footprint used by the low-level controllers.

coordinates, as well as its orientation, θ.) This representation reflects the robot's actual kinematics and provides an accurate or even "pessimistic" (as a margin of safety is commonly added to the model of the robot) evaluation of the robot's capabilities.

An argument could be made that the planning layer could simply use a pessimistic over-estimate of the robot's footprint (perhaps a circle circumscribing the robot's actual footprint), always planning paths with wide safety margins. We find two problems with this strategy: First, it is possible that no solution exists for the pessimistic planner, when an accurate representation of the robot's kinematics would find a path. Second, especially when working with visual sensors, sensory data is less accurate far away from the robot. Using an optimistic planner effectively allows the sensory data a margin of error before closing off any path. While a pessimistic planner would plan around these borderline cases, an optimistic planner would bring the robot closer, allowing for better sensory input. It is this later problem – of reconciling an optimistic planner with the robot's true capabilities – that this work aims to address.

2.4 Mapping, Planning, and Control

In this section, we briefly describe our robot platform, system integration, and planning and control tools. A more complete description of this system is provided in [72] and [76].

2.4.1 The LAGR Setup

The test bed for this chapter's experiments is the LAGR robot. Learning Applied to Ground Robots (LAGR) is a DARPA-funded project with the goal "to develop a new generation of learned perception and control algorithms for autonomous ground vehicles, and to integrate these learned algorithms with a highly capable robotic ground vehicle" [2].

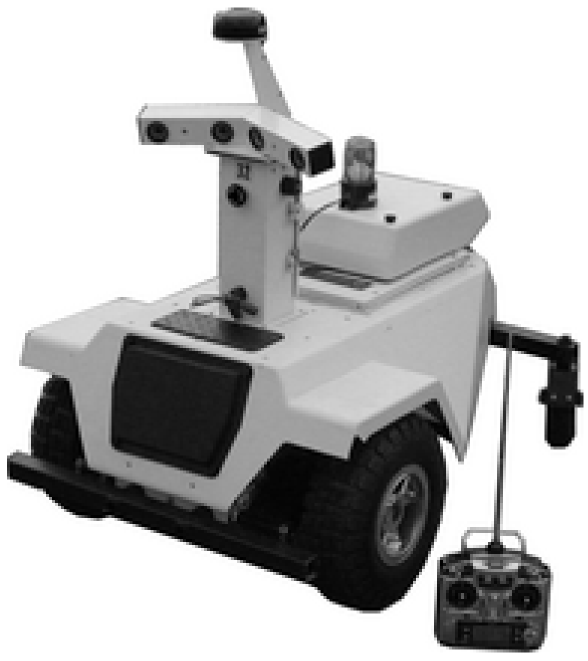


Figure 2.4: The LAGR Robot.

The LAGR robot, depicted in Figure 2.4, possesses four color cameras, a front bump switch, a Garmin GPS receiver, and an inertial measurement unit. The cameras are paired together so that each pair can provide stereo depth maps with a range of 6-10 meters. The robot's turning axis is centered just behind the front axle, with the rear unpowered wheels turning on casters. Its physical dimensions are 90 cm in length, 60 cm in width, and 60 cm in height, with a weight of about 90 kg.

Mapping, control and planning processes are run on a single Linux machine (1.4 GHz Pentium-M, 1 GB RAM, RedHat 9), the *planning computer.* There are three similar computers connected via Ethernet: 1 each for camera/stereo processing (called *eye* computers), and 1 for lower-level functions (e.g. radio-controller interfacing, GPS/IMU integration).

2.4.2 Map Processing

The two *eye* computers collect camera images, compute stereo depth maps, and using the robot's global position, transform a local terrain map onto a global coordinate frame. Then, this information is passed over Ethernet to the planning computer. On the same machines, the camera images are used to compute estimates of traversability of points in the global frame, as described in [40]. Both processes run steadily at 4Hz.

The planning computer receives the two streams of stereo and traversability information from both *eye* computers and incorporates it into global maps of terrain and traversability. This data is stored in grid cells of fixed size, with a resolution of 0.1 m. Under the assumption that the newest information is the most likely to be correct, previous information in a grid cell is overwritten with the new.

A separate map also stores the locations where the robot has encountered hits on its bumper, spikes in its motor amperes, and detections of wheel slippage.

2.4.3 Motion Planning

Incremental motion planning is executed over the global maps described above. The planning algorithm we use is either an A^*-type planner, or the combinatorial planner we described in [78]. However, for the purposes of this discussion, the particular planner to be used is of no real consequence. The mapper passes on points which have been modified by the stereo/traversability/etc processes to the planner, which incrementally updates its planned path.

2.4.4 Motion Control

Reactive control is implemented in a manner inspired by the DAMN [66] architecture. In this implementation, individual controllers, representing specific interests related to the robot's overall objective, are given an allotment of "votes" which they may cast for or against actions that will work to achieve their goals. An arbitrator sums the votes, choosing the action with highest tally. Similar implementations have been successfully deployed in several robotic navigation tasks [67, 75].

In our implementation, the actions evaluated are straight-line paths, at a resolution of 5 degrees around the robot. The controllers take the kinematic constraints of the robot into account by evaluating the effect of first turning to the desired direction (effecting the rotational component of the motor command) and then traveling in that direction (effecting the translational component of the motor command). This turns out to be a reasonable approximation of the robot's low-level controller, which implements a relatively aggressive rotational gain. The following voting controllers are used:

- **follow-plan** – casts positive votes in the direction of the next point on a list of waypoints provided by the planner. Votes are distributed according to a Gaussian function centered on the direction of next waypoint.

- **avoid-stereo-obstacles** – casts negative votes in the direction of any obstacles sensed by the stereo vision system. Votes are distributed according to a sum of Gaussian functions, each centered on a sensed obstacle.

- **avoid-color-obstacles** – casts negative votes in the direction of any obstacles sensed by the traversability (i.e. color-based) vision system. Votes are distributed according to a sum of Gaussian functions, each centered on a sensed obstacle.

One potential pitfall of arbitration over votes is misallocation of each controller's allotment of votes. If controllers' voting weights are not properly balanced, one controller may dominate the arbitration, either preventing the robot from making progress to higher-level goals or allowing the robot into undesirable states. Because this weighting is typically an empirical process and dependent on implementation and environment, we have added robustness in a manner similar to [71] by supplementing the voting scheme with "vetoes". Each controller, in addition to its allotment of votes is given the option to veto each of the available actions. The arbitrator respects the vetoes by ignoring actions that have been vetoed by at least one controller, regardless of how many votes those actions have received.

The strategy is that actions which are deemed to put the robot in imminent danger should be vetoed. What qualifies as "imminent danger" must be decided on a controller-by-controller basis. Because the burden is only to identify dangerous paths over a short distance, the full dynamics of the robot can be considered, including collision checking of rotations necessary to achieve the desired orientation and the feasibility of various maneuvers given the slope of the terrain.

Like the voting controllers, these vetoing controllers use a set of straight-line paths at 5-degree resolution as the action set:

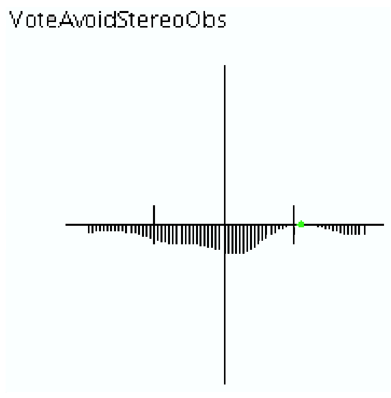
- **veto-stereo-obstacles** – casts vetoes against any direction which will bring the robot into a collision with a stereo vision-sensed obstacle within one meter of the robot's current position, taking into account the robot's configuration space.
- **veto-color-obstacles** – casts vetoes against any direction which will bring the robot into a collision with a color vision-sensed obstacle within one meter of the robot's current position, taking into account the robot's configuration space.

While the specific control mechanism is certainly an issue open for debate and research, it is not a central component to this work. In fact, the point should be made that any reactive controller (e.g. probabilistic voting, motor schemas) could be adapted to work in this architecture. The specific implementation presented here is given only as an example of how a system could be integrated into the larger architecture.

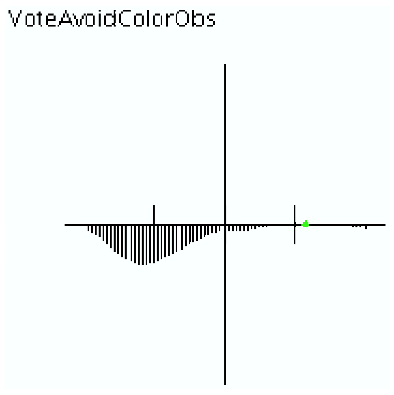
2.5 Feedback from Controller to Mapper

Simultaneous Control and Mapping, or SCAM, is an architecture where controllers drive the robot based on maps, and the maps are informed (in part) by the controllers. The resulting bidirectional information flow between operational layers thus consists of the standard map-to-controller flow as well as the novel controller-to-map flow (see Figure 2.2). Conceptually, this latter feedback mechanism from the controllers to the map, is executed by feeding back points to the map which correspond to conflict between the deliberative and reactive controllers.

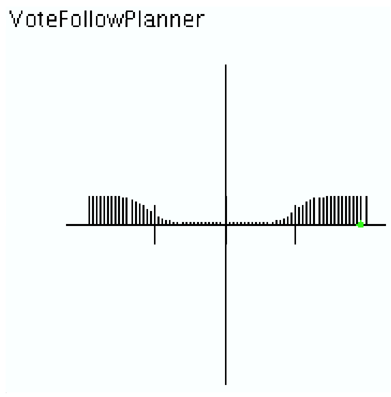
Refer to Figure 2.6; two obstacles leave a small opening, allowing a feasible path to pass between, given that the planner assumes some relatively optimistic configuration



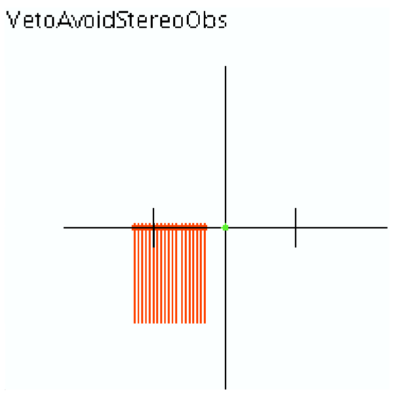
(a)



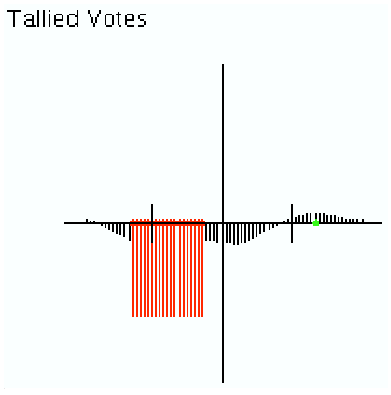
(b)



(c)



(d)



(e)

Figure 2.5: A graphical representation of the voting scheme employed to navigate the robot. The x axis of each plot represents an ego-centric angular distribution of possible paths around the robot in the range $(-\Pi, +\Pi\ ]$, with 0 being in front of the robot. The y-axis represents the relative preference of each path, according to the respective controller. Vetoes are drawn as large negative values. The last plot represents the sum of the votes provided by all the controllers. The largest non-vetoed value is chosen for action by the robot. In this example, the first behavior resists a stereo-perceived obstacles to the front and left of the robot. A color-based obstacle is perceived to the left. The plan tells the robot to go backwards, and the left is vetoed as a result of stereo obstacles. The tallied votes tell the robot to go to the right.

space for the robot. However, the controllers, which consider higher-dimensional kinematic/dynamical constraints do not allow this action. In the SCAM architecture, the controllers then detect this conflict and inform the map that the point (highlighted in the figure) should be marked as intraversable.

The points placed into the map can be set conservatively small, blocking only a small region in the map. This may then not block the passage through the offending region in a single step, but multiple planning/controller cycles through this region will place several points in the map, and eventually cover the kinematic obstruction.

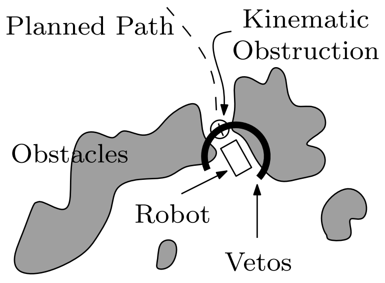


Figure 2.6: Illustration of Simultaneous Control and Mapping Implementation.

2.6 *Application*

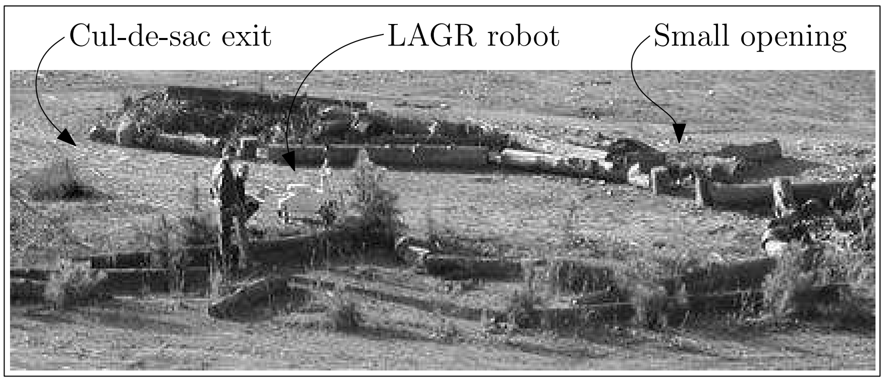


Figure 2.7: Overview of Experiment Site.

In order to highlight the benefit associated with the proposed method and to illustrate its practical usefulness, we ran a series of controlled experiments in an outdoor terrain populated by small pine trees, fallen logs and other vegetative obstacles, as seen in Figure 2.7.

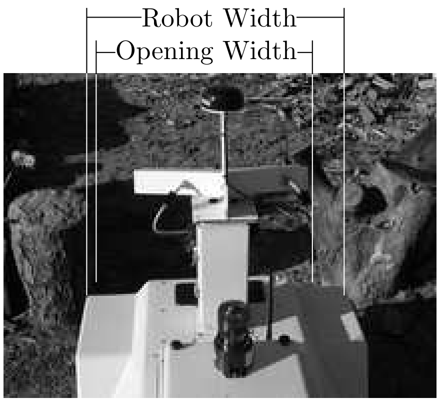


Figure 2.8: Opening in Cul-de-sac.

In the following subsections, we summarize the outcomes of these experiments and highlight the differences in performance.

The particular experimental setup that we considered was the following: The robot was started up in the interior of a cul-de-sac with one small opening along one of its walls. This opening was wide enough to tell the aggressive planner that a feasible path exists through the opening. However, the reactive controllers will find the opening to be too narrow for safe passage, and as a result, they will veto any attempt to drive through it. For each experiment, the robot was started with no *a priori* information about the environment except its relative position to the global goal.

2.6.1 Planner Only

In the first run, only the planner was affecting the motion of the robot, and the only active low-level controller was a path-following controller. As was to be expected, the planner found the opening and tried to push through (Figure 2.8), with the result that the robot crashed into one of the logs defining the boundary of the opening (Figure 2.9). It should be noted that this problem can be remedied by making the planner less aggressive and allowing for a larger, explicit safety-footprint. However, one of the basic ideas behind the system framework is to let the planner be aggressive and optimistic, and let the reactive low-level controllers ensure safety and robustness if the planned path is deemed unsafe.

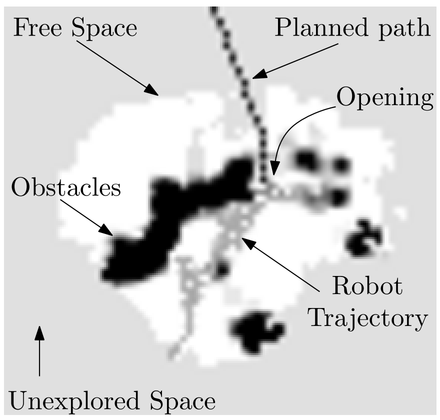


Figure 2.9: The map, trajectory and plan resulting from an experiment using only a planner. Because the planner is too optimistic for the configuration space of the physical robot, the robot collides with obstacles while trying to navigate through the narrow opening.

2.6.2 Reactive Controllers Only

In the second run, only the reactive, local controllers were active, and no global plan was provided from the planner. This control strategy exhibited the well-known and expected behavior of getting stuck in the cul-de-sac without any global information (aside from the heading to the goal) to guide the robot (Figure 2.10). It should be noted though that the safety controller did in fact veto the opening that the planner-only controller tried to push through.

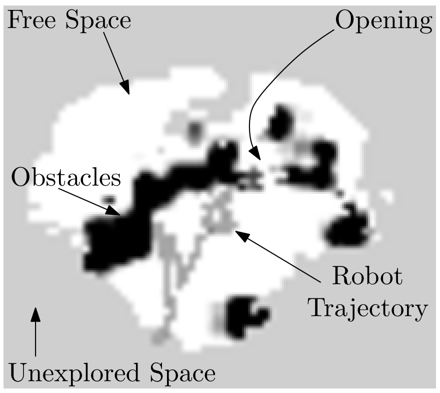


Figure 2.10: The map and trajectory resulting from an experiment using only reactive controllers. The safety-minded controllers kept the robot a safe distance from all obstacles, but did not allow progress to the goal location.

2.6.3 Planner Affecting the Reactive Controllers

In this scenario, there exists the potential for planning out of the cul-de-sac as well as proper maintenance of safe operation, but since the planner had no way of knowing that the opening was too narrow, it insisted on the robot driving through the opening. Meanwhile the safety-controller vetoed that action. As a result, the robot did not exhibit any improved behavior over the reactive-controller-only situation (see Figure 2.11). However, if the robot would have started outside the cul-de-sac, it is possible (but by no means guaranteed) that it would have eventually planned its way of the area based on the stored map information.

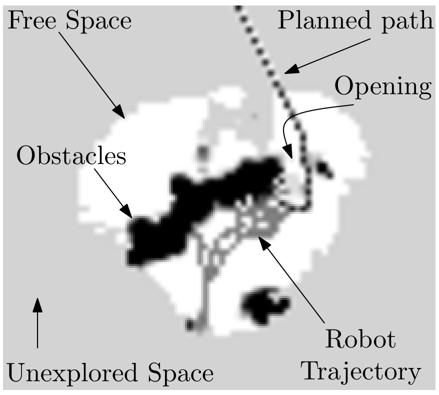


Figure 2.11: The map, trajectory and plan resulting from an experiment using a planner which influences reactive controllers. The optimistic planner guides the robot toward the narrow opening, while the safety-minded controllers prevent the robot from entering. The result is that the robot loiters around the mouth of the opening.

2.6.4 Feedback From the Controllers to the Planner

Here, the planner once again tried to force the robot through the opening in the cul-de-sac wall. However, the safety-controller vetoed this action as well as encoded this veto through the feedback mechanism as an obstacle in the map, and the planner then re-planned its course of action. As seen in Figure 2.12, after a bit of exploring of the cul-de-sac, the robot decided that there was no way forward through the cul-de-sac, and a path was planned out from the area, which enabled the robot to continue its mission.

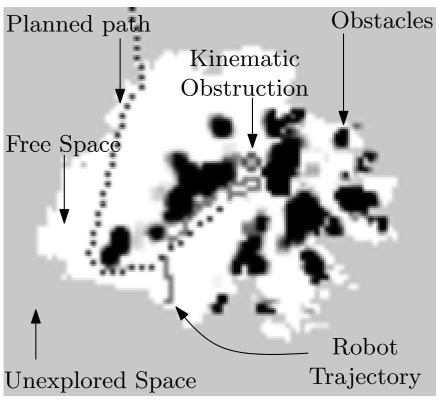


Figure 2.12: The map, trajectory and plan resulting from an experiment using a planner which influences reactive controllers with feedback back to the global map. The planner initially guides the robot toward the narrow opening, but the reactive controllers veto this action, noting that action in the global map. Using this information, the planner finds a path through the only safe opening in the cul-de-sac.

2.7 Future Work

The mechanism presented in this work provides several avenues for extensions and future work.

A major assumption of this work is the ability to project high-dimensional "obstacles" detected at the controller level into the planar map. Necessarily, information is lost in this projection, which translates the detected obstructions from a three-dimensional (x, y, θ) space into a two-dimensional (x, y) space. This lossy projection provides a practical heuristic for navigation, but precludes a proof of correctness. It may, however, be possible to prove correctness for certain spaces under certain conditions. Future work on this subject should address the possibility of a proof of correctness of this approach.

While this work addressed the problem of projecting high-dimensional *kinematic* constraints into a low-dimensional representation, it does not address *dynamic* constraints. A scenario could be constructed in which a robot's dynamic state must be known in order to evaluate whether the robot can safely traverse the region in question (e.g. the robot needs to carry momentum up a ramp to successfully move ahead). Currently the solution presented here has no explicit support for the dynamics of the robot. Extensions should be devised to address this need.

2.8 Summary

In this chapter, we argue that it is beneficial to introduce a feedback mechanism from the low-level controllers to the high-level mapping and planning processes. In particular, based on the performance of the low-level controllers (and their interaction with the environment), kinematic obstructions are encoded in the global map even though they may not be perceived as obstacles by the planner. We argue that this is beneficial for the following reasons:

- The low-level controllers typically operate at a shorter time scale than the planner. This implies that the obstructions can be detected sooner by the controllers and, as such, they can notify the planner directly by short-cutting the time-consuming perception processing step.

- Due to the computational burden of global path planning, maps are typically planar (or at least low-dimensional), which means that high DOF kinematics, dynamic constraints, or complex configuration spaces cannot be handled directly. Through the feedback from the controller to the mapper/planner they can, however, be incorporated in the lower-dimensional descriptions of the environment.

We illustrate this view through an experiment in which a robot is trying to negotiate a cul-de-sac. This experiment shows that the introduction of feedback between layers has a beneficial impact on the robot performance.

CHAPTER III

ORIENTED VISIBILITY GRAPHS

This chapter addresses the problem of computing a path from the free-space of the environment to a static goal point. The approach that we take is related to the class of visibility graph planning methods, especially the reduced visibility graph. Our contribution as related in this chapter is primarily the derivation of an algorithm for dynamic planning in an unknown planar environment. The algorithm is presented in pseudocode and analyzed with respect to running time, and a real-world application of this approach is described in detail.

For this chapter, we denote a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P})$ with

$\mathcal{V}$– the set of vertices. The cardinality of $\mathcal{V}$ is n.

$\mathcal{E}$– the set of edges $\subset \mathcal{V} \times \mathcal{V}$. This set is partition into

- $\mathcal{E}_s$ – the edges defining the boundary of the polygons
- $\mathcal{E}_v$ – the inter-polygonal edges generated by a visibility graph algorithm

$\mathcal{P}$– the set of polygons. Each polygon $p \in \mathcal{P}$ is described by a subset of vertices in $\mathcal{V}$, denoted by $vert(p)$, and a subset of edges in $\mathcal{E}$, denoted by $edge(p)$.

The sets $\mathcal{V}$, $\mathcal{E}_s$ and $\mathcal{P}$ will vary according to the external perception process that determines the location of polygons and are the input to the visibility graph algorithm. The inter-polygonal edges $\mathcal{E}_v$ are the output of the visibility graph algorithm. It will be convenient to refer to individual edges as $e_{ab} \in \mathcal{E}$ where e_{ab} passes from the vertex $a \in \mathcal{V}$ to the vertex $b \in \mathcal{V}$. Edges are undirected, meaning that e_{ab} is equivalent to e_{ba}.

It could hardly be overstated how much work the research community has contributed to what falls into the general field of "path planning". In the field of computational geometry, there exist *complete* solutions that have provably optimal shortest paths and which achieve the theoretic running time lower bound. Yet it is also true that these algorithms tend to be academic exercises that the use of such methods for field robotics is rare [46, 57].

Search-based methods and, increasingly, sampling-based methods have on the other hand been extremely popular. Implementations of this style of algorithms have been used on a large number of real-world robots [24, 43, 72]. What combinatorial methods have been missing compared to these methods is two fold: a fast algorithm for dynamic environments and low programming complexity. "Programming complexity" is the complexity of implementing an algorithm for a working system. It is our view (without diminishing the importance of other measures of an algorithm's quality: computational complexity, space complexity, optimality bounds, etc) that this measure of an algorithm must not be overlooked since our intent is only to design methods that can be run on robotic systems.

This chapter represents our work on a combinatorial method for path planning that meets both these objectives, that is a middle-ground between the algorithms about which we can prove much but with which we can do little and the algorithms which are (relatively) simple, powerful, and justified by practice.

3.1 Background

An ideal path planner for a mobile robot in the plane should satisfy certain requirements:

1. The plan should be optimal (in either cost or distance).
2. Given a natural representation of the world, the planner should produce a natural plan.
3. Revisions to the robot's perception of world should induce efficient incremental revisions of the plan.
4. The state of the planner should be human-readable.

The purpose of the first requirement is clear: the best planning algorithm returns an optimal path to the goal. The meaning of the second is that the robot ought not to follow jagged paths when a straight path is available. However, this is a common shortcoming of the planning approaches that are based on a quantization of the world into grid cells. The third requirement implies that when the robot's perception changes, the last plan does not necessarily need to be destroyed entirely. Instead, a subset of the plan is invalidated and must be reworked.

Requirement 4 is important because while every plan should quickly and accurately return a desired path or direction to the robot, the planner does not exist in a vacuum; it is integrated with dozens of preceding perceptual and subsequent reasoning modules. During the development of an implementation of any such module and analysis of this module's effect on other modules' interactions, the changing (internal) state of the planner can be supremely important. For example, when analyzing the robustness of a perception module to noise and evaluating the effect of noise reduction techniques, the varying state of the planner to such adjustments (not just the output path) could be essential in determining what approach is the best to use.

Recall from Chapter 1 the *reduced visibility graph* (RVG). This approach to path planning is complete, provides the shortest-path roadmap over C_{free}, but does not allow for dynamic updates of the polygonal domain.

3.1.1 Properties of a Reduced Visibility Graph

The reduced visibility graph exhibits two important properties.

1. The number edges in the graph is $O(p^2)$, where p is the number of polygons in the graph. In the case where all polygons are convex, four edges are added between each pair of polygons (if visibility is not violated) [44]. This property has two major ramifications. First, all reduced visibility graph algorithms have a lower bound on the running time of $O(p^2)$. In other words, we should expect that in general the amount of time required to plan a path between the robot and a goal will grow quadratically with the size of the graph.

2. An RVG is defined without respect to a goal point. In applications where planning between many locations in a static (or perhaps slowing-varying environment) is needed, this is a good property. However, in the case where planning is only to be done

to a static goal location, an RVG is populated with unnecessary and uninformative edges. In addition to the computation wasted on finding these edges, this extraneous polygonal interdependency adversely effects the efficiency of a dynamic algorithm.

It is these properties that this chapter's *oriented visibility graph* is designed to address.

3.1.2 Computing a Reduced Visibility Graph

Following the notation in [46], we let a *reflex vertex* be a vertex in $\mathcal{V}$where the angle interior to the polygon is less than π. A *bitangent edge* is formed between any pair of reflex vertices where the edge does not intersect the interior of any polygon in $\mathcal{P}$. For an example, see Figure 3.1.

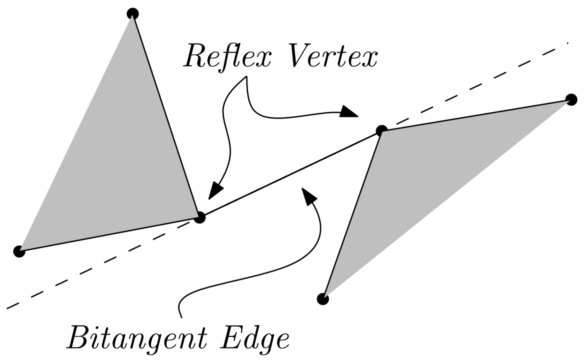


Figure 3.1: Illustration of a *reflex vertex* and a *bitangent edge*.

An RVG algorithm computes the inter-polygonal edges $\mathcal{E}_v$. Based on computational complexity, there have been four main advances in finding $\mathcal{E}_v$.

The Naive Algorithm - $O(n^3)$

$\mathcal{E}_v$ can be computed using a naive algorithm in cubic time, based on the number of nodes in the graph [51]. Assessing the mutual visibility of two given vertices takes $O(n)$ time, and there are $O(n^2)$ pairs of vertices to check. The overall running time then is $O(n^3)$.

Lee's Algorithm - $O(n^2\ log(n))$

The first non-trivial algorithm is due to Lee [47]. The complexity reduction is achieved by applying the *rotation sweep* method, vertices are sorted by their angle relative an arbitrary point. The sorting (e.g. heapsort, merge sort) can be done in time $O(n\ log(n))$. The visibility of all n vertices must be checked, yielding an overall time of $O(n^2\ log(n))$.

Quadratic Algorithms - $O(n^2)$

A number of quadratic algorithms are known [6,20,57]. In the sense that the cardinality of $\mathcal{E}_v$ is $O(n^2)$, these algorithms are running-time optimal. Most of these algorithms operate by first performing a triangulation over C_{free}or mapping $(\mathcal{V}, \mathcal{E}_s)$ to the dual space and using the rotation sweep method to soft checking of intersections.

Output-sensitive Algorithms - $O(n\ log(n)\ +\ card(\mathcal{E}_v))$

Introduced in 1991, Ghosh and Mount published the first output-sensitive algorithm for computing an RVG [31]. So, in the case that the number of edges in $\mathcal{E}_v$ is small (less than $O(n^2)$), this algorithm outperforms all previous algorithms, at least according to a worst-case asymptotic theoretical comparison. It is, however, well-cited in the literature (including by the authors themselves) that this algorithm is complicated to the point that implementing it is prohibitive, and constant terms that are lost in big-O notation are large.

Two important observations should be made. First, these algorithms are static and do not address shortest-path planning for unknown, dynamic, or slowly varying environments. Second, since the introduction of D^*, the use of visibility graph based planning in field robotics is rare.

3.1.3 Dynamic Reduced Visibility Graph

While there are numerous static visibility graph algorithms, making an efficient dynamic version is much harder and largely unknown. The reason for this is that when a polygon is retracted (that is, when it vacates space it previously occupied), any other pair of polygons in $\mathcal{P}$may consequently have a new bitangent edge that must be generated. Hence, when a polygon is modified, the connectivity of every other polygon must be re-evaluated, and essentially the graph is re-created from scratch.

In the case where polygons can only be added and expanded, an extension of Lee's implementation of the RVG algorithm can be given. We present it here for comparative purposes. The input to the algorithm is a set of polygons where each polygon is either (1) a new *addition* to $\mathcal{V}$, $\mathcal{E}$, and $\mathcal{P}$, or (2) a *modification* that replaces elements of $\mathcal{V}$, $\mathcal{E}$, and $\mathcal{P}$.

Addition The addition of a polygon p would require $O(k\ n\ log(n))$ time (where k is the cardinality of $vert(p)$) to generate the bitangent edges between p and the other polygons in $\mathcal{P}$. That is, for each $\{v \in vert(p)\}$, the (n-1) other vertices in $\mathcal{V}$are sorted around v and checked for visibility.

In addition, detecting the possible intersection of p with the existing edges in $\mathcal{E}_v$ requires $O(card(E_v))$=$O(n^2)$ time.

Modification The modification of a polygon $p \in P$ can be done in three steps: (1) Delete the old p (cropping $vert(p)$ from $\mathcal{V}$and $edge(p)$ from $\mathcal{E}$) (there may be more than one old p), (2) Checking each edge $e \in \mathcal{E}_v$ if it intersects the new p, and (3) adding the new p as described above.

If we say that the number of vertices in a polygon is $O(1)$, then the addition step can be said to use $O(n\ log(n))$ which is then the overall running time.

3.2 Approach

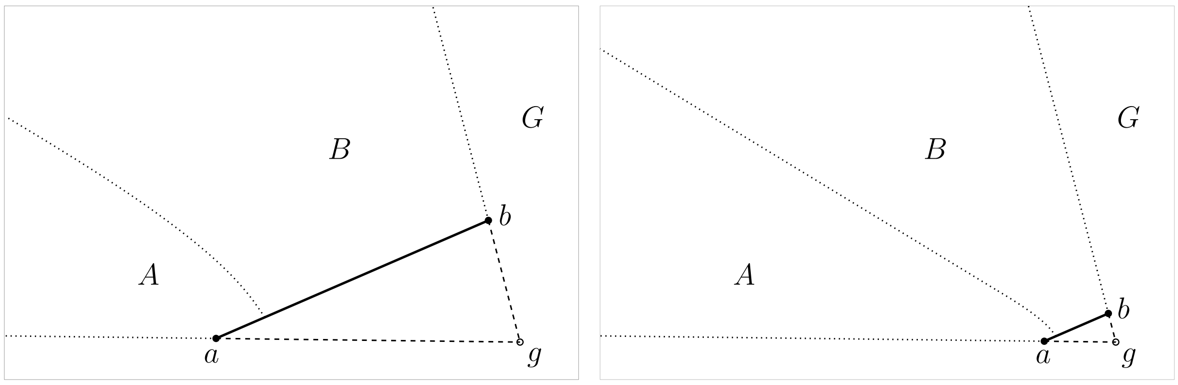
In this section, we formalize the assumptions that are made for the remainder of this chapter.

First, we assume the environment is described as a polygonal domain. This means that impassable obstacles are described as (possibly non-convex) polygons. We also assume that the goal location is not in the interior of a polygon. Next, we assume the goal location is static, implying that we are solving the *single-source* path-planning problem. This assumption is the first major difference between the oriented visibility graph and the reduced visibility graph. Third, we assume the environment is planar and that path planning can be done as if the robot were holonomic, as is common in search-based and combinatorial path planning approaches. Hence the configuration space is two dimensional and the robot is modelled as a "free-flying" point.

Our last assumption is that in order to connect a polygon p to $\mathcal{P}$, only two edges are needed to provide an optimal path. This assumption is in general false, but in many scenarios is a close approximation. It is this assumption that allows for efficient dynamic updates and leads to the oriented visibility graph algorithm.

3.2.1 The Two-Edge Approximation

In general, the number of edges needed to connect a polygon to the rest of a roadmap grows quadratically with the size of the graph. However, in some cases, exactly two edges are all that is required. Moreover, in many situations where more than two edges are strictly necessary, apply the right two edges is a close approximation.

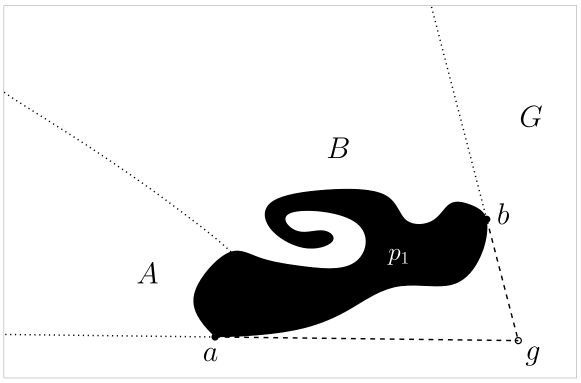


(a) Close view (b) Distant view

Figure 3.2: Illustration of the *two-edge* approximation. C_{free} is partitioned into three regions, A, B, and G. The optimal path for any point starting in Region A includes vertex a. No point starting in Region G will have in its optimal vertices a or b.

To illustrate the basis for this approach, we present Figure 3.2. In Figure 3.2, a line segment $\overline{ab}$ is an obstacle. This induces a partitioning of C_{free} into three regions. When the robot is located in Region G, the optimal plan is to drive straight to the goal. When in Region A, the optimal plan is to drive to vertex a and then drive straight to the goal. Similarly, in Region B, the optimal plan is to drive to vertex b and then straight to the goal. The edge $\overline{ab}$ blocks a region Region AB, the union of Region A and Region B. All optimal paths that start in (or enter) this region will necessarily cross either vertex a or vertex b. The inter-polygonal set of edges $\mathcal{E}_v$ in an RVG will be composed of exactly the edges e_{ag} and e_{bg}.

In Figure 3.3, the previous example is generalized slightly. The polygon is no longer of zero measure and is non-convex. C_{free}, however, is still partitioned into three regions and while $\mathcal{E}_v$ is composed of more edges than from Figure 3.2, it is true that every path starting in Region AB will include the edges e_{ag} and e_{bg}.



(a)

Figure 3.3: Illustration of the *two-edge* approximation applied to a more complex obstacle. Note how with this single obstacle, C_{free} is partitioned still into three Regions: A, B, and G. All points in Region A pass through vertex a on the path to g (and similarly for Region B). All other points connect directly to the goal.

Now, consider Figure 3.4 where the dashed lines show the edges in $\mathcal{E}_v$ generated by the RVG algorithm. With the goal located at vertex g, the heavy dashed edges are necessary to provide the optimal roadmap. The lightly dashed lines are unnecessary. So, again, this figure illustrates a case when two edges per polygon provide the optimal path to the goal.

The local optimality condition used in [44] to prune to full visibility graph to the reduced visibility graph is a sufficient but not necessary condition for providing a roadmap with optimal paths to a static goal. On the other hand, two edges per polygon is not a sufficient condition.

Figure 3.5 shows a reduced visibility graph for a simple set of obstacles. Our approach is to add edges such as is shown in Figure 3.6, where indeed this is an optimal roadmap. The key but simple idea is that for any obstacle we must circumnavigate, the path we will follow will take us by one side of a polygon of the other.

Even in the case of a domain restricted to convex polygons, two edges per polygon is not sufficient to guarantee optimal paths to a static goal in all obstacle configurations

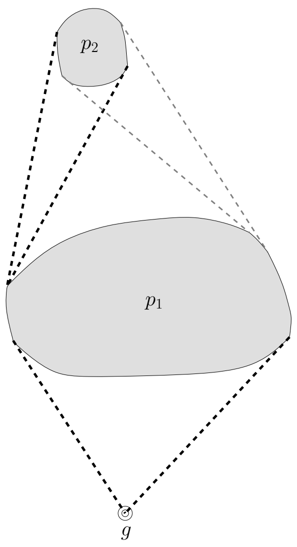


Figure 3.4: Illustration of unnecessary edges in a reduced visibility graph.

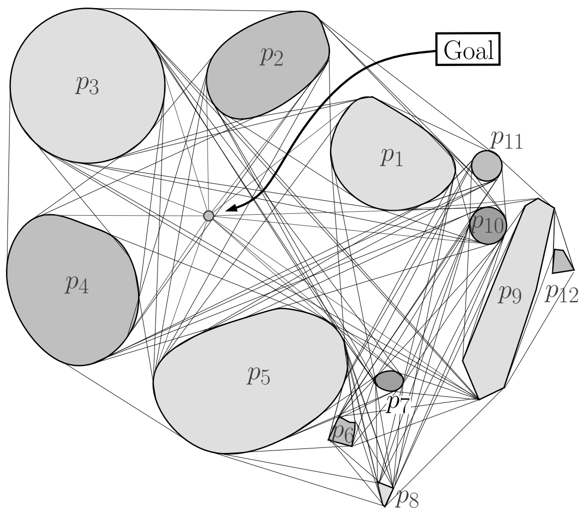


Figure 3.5: Reduced Visibility Graph.

(as is discussed further in Section 3.4.2.1). However, Section 3.3 describes algorithms to generate our oriented visibility graph on this assumption. This discussion includes a set of restrictions that lead to an optimal roadmap from the OVG, and a general algorithm where these restrictions are lifted.

3.2.2 Shadow Regions

Before introducing the algorithms used to generate an oriented visibility graph, it is instructive to examine the shadow region from the simple graph in Figure 3.2. Region AB is an

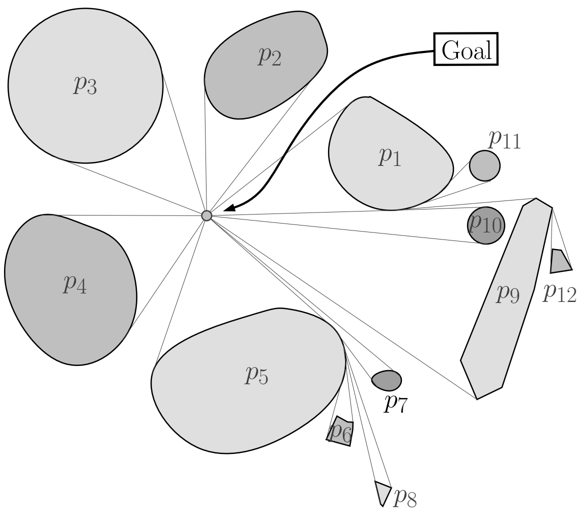


Figure 3.6: Oriented Visibility Graph.

open bounded set described as the intersection of three halfspaces:

Halfspace 1:

$$\{(x,y) \mid c_0 + c_1 x + c_2 y < 0\} \tag{2}$$

where

$$c_0 = g_y + m g_x, \tag{3}$$

$$c_1 = m, \tag{4}$$

$$c_2 = 1, \tag{5}$$

$$m = \frac{a_y - g_y}{a_x - g_x}, \tag{6}$$

and for some vertex v, v_x denotes the x-coordinate associated with v.

Halfspace 2:

$$\{(x,y) \mid c_0 + c_1 x + c_2 y < 0\} \tag{7}$$

where

$$c_0 = g_y + m g_x, \tag{8}$$

$$c_1 = m, \tag{9}$$

$$c_2 = 1, \tag{10}$$

$$m = \frac{b_y - g_y}{b_x - g_x}. \tag{11}$$

Halfspace 3: $\{(x,y) \in \mathbb{R}^2 \mid c_0 + c_1 x + c_2 y < 0\}$ where $c_0 = a_y + m a_x$, $c_1 = m$, $c_2 = 1$, $m = \frac{b_y - a_y}{b_x - a_x}$.

Region AB is bifurcated by the curve

$$\{(x,y) \mid c(a) + d(x - a_x, y - a_y) = c(b) + d(x - b_x, y - b_y)\} \tag{12}$$

which in general is a hyperbola. Here, $d(x,y)$ denotes the Euclidean distance $\sqrt{x^2+y^2}$ and $c(v)$ is the path cost to the goal from vertex v. This bifurcating curve asymptotically approaches a line.

That the bifurcation is determined by a hyperbola makes directly using its partition of C_{free}prohibitive. Instead, we will use the notion of what we will call "shadow-casting" vertices to motivate the two-edge principle generating the graph. In Figure 3.7, the required to provide the optimal roadmap have been added to the polygons.

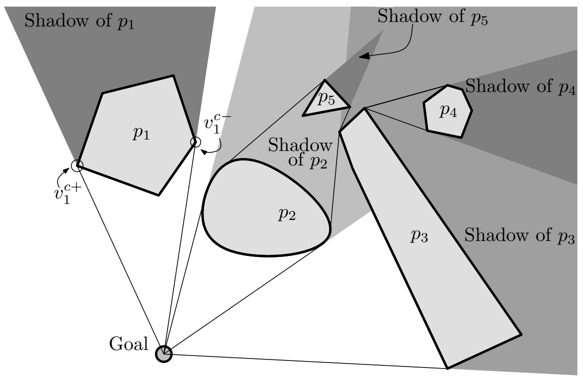


Figure 3.7: Polygon Shadows.

The vertices to which these edges connect at each polygon can be seen to cast a "shadow" to the rear the polygon. This shadow identifies the portion of C_{free} that may have to pass by the shadow-casting vertices on the optimal path to the goal. (Note that these vertices are not necessarily on the optimal path. However, an optimal path that does not start in the shadow of a polygon will never enter it.) These shadow are defined by the inter-polygonal edges added to each polygon.

The shadow is an instructive metaphor for what the edges generated by the oriented visibility graph algorithm are utilized for. It is the shadow region of a polygon that these edges are meant to provide connectivity to the rest of the graph.

3.3 Algorithms

3.3.1 A Restricted Approach

We first construct a planning algorithm for static environments. This planner will both have low complexity and be optimal under certain restrictive assumptions (denoted *R.A.* for "Restrictive Assumptions"). We will then show how to generalize this algorithm to situations where the assumptions are violated while maintaining the low-complexity nature of the planner. In this case, however, optimality can no longer be guaranteed. In other words, we are well-aware of the fact that the assumptions under which this planner is in fact optimal are not feasible in most realistic scenarios. However, they are to be thought of as enabling a structured design methodology that stresses feasibility and low complexity.

Restrictive Assumption 1 *All polygons are convex.*

We will need to establish some initial notation. Given a polygon p, we will let $vert(p)$ denote the set of vertexes of p, and by $int(p)$ we will understand the interior of p. Moreover, we will, with a slight abuse of notation, let $v \in vert(p)$ denote both a given vertex as well as its position, while we notate the face between vertexes v and u by the line segment

$$\ell(v,u) = \bigcup_{\alpha=0}^{1} (\alpha v + (1-\alpha)u).$$

It will also prove convenient to define a distance measure from a polygon p to the goal located at v_g, and we do this through the following two operations:

$$\begin{aligned} \overline{d}_g(p) &= \max_{v \in vert(p)} \|v - v_g\| \\ \underline{d}_g(p) &= \min_{v \in vert(p)} \|v - v_g\|. \end{aligned}$$

In order to be able to produce a compact description of the initial algorithm, we will moreover assume that we can introduce an ordering over the obstacles as follows:

Restrictive Assumption 2 *The obstacles can be ordered as $p_1, p_2, \ldots, p_k$, in such a way that $\underline{d}_g(p_j) > \overline{d}_g(p_{j-1}),\ j = 2, \ldots, k$.*

The final piece of the puzzle is the notion of a tangent segment between polygons. Following the development in [44], we say that a line segment $\ell(v, v')$ passing through $v \in vert(p)$ is *tangent* to p at v if the interior of p lies entirely on a single side of $\ell(v, v')$. Moreover, if $v' \in vert(p')$, $p' \neq p$ then $\ell(v, v')$ is a *tangent segment* if $\ell(v, v')$ is tangent to p at v as well as to p' at v'. Now, as shown in [44], between any convex obstacles there are exactly four tangent segments and between a convex obstacle and a point, there are two. Hence, let $\mathcal{V}_i$ be the vertex pair $\{v_i^{c+}, v_i^{c-}\} \subset vert(p_i)$, such that $\ell(v_i^{c+}, v_g)$ and $\ell(v_i^{c-}, v_g)$ are tangent segments.

Restrictive Assumption 3 *If $\ell(v_j, v_g) \cap int(p_i) \neq \emptyset$ for some $v_j \in \mathcal{V}_j$ then $\ell(v_j, v_i^{c+})$ and $\ell(v_j, v_i^{c-})$ (where $\mathcal{V}_i = \{v_i^{c+}, v_i^{c-}\}$) are both tangent segments between polygons p_i and p_j.*

Restrictive Assumption 4 *The initial robot position v_r is such that if $\ell(v_r, v_g) \cap int(p_j) \neq \emptyset$ for some j, then $\ell(v_r, v_j^{c+})$ and $\ell(v_r, v_j^{c-})$ (where $\mathcal{V}_j = \{v_j^{c+}, v_j^{c-}\}$) are both tangent segments between v_r and p_j.*

Now, the proposed planner will be based on the directed graph $\mathcal{G} = \mathcal{V} \times \mathcal{E}$, where $\mathcal{V}$ is a set of vertexes and $\mathcal{E}$ is a subset of ordered pairs over $\mathcal{V}$. (With a slight abuse of notation we will let $v \in \mathcal{V}$ and $\ell \in \mathcal{E}$ denote both combinatorial graph objects as well as their planar geometry. This should, however, be completely clear from the context.) Moreover, we will associate an edge cost $\mathcal{W} : \mathcal{E} \to \mathbb{R}^+$ and a vertex cost $\mathcal{C} : \mathcal{V} \to \mathbb{R}^+$, with each edge and vertex in $\mathcal{G}$ respectively. In particular, the graph will be constructed in such a way that the out-degree of each vertex (except the goal point) is exactly equal to one, and hence the final plan is directly given by the only path connecting v_r with v_g.

Theorem 3.3.1 *Under the four restrictive assumptions, the shortest path from v_r to v_g is given by the unique path from v_r to v_g obtained through Algorithm 1.*

Proof: The main idea behind the proof is that edges between vertexes are only kept if they are in fact optimal. From [44] we know that only tangent segments satisfy the necessary optimality conditions and hence our graph should only contain such segments as vertexes, which is in fact the case. Moreover, by construction, only one edge in the graph originates from each vertex. If this edge does not connect to a goal, it is the tangent segment that connects with the polygon blocking the goal that has the lowest vertex cost. This vertex cost should in fact be interpreted as the total cost to go and through Assumptions RA1 and RA3 we know that this construction is well-defined. However, it is still possible that cycles may appear in the graph, and hence that the algorithm would not terminate. But, if polygon p_j is in the path connecting polygon p_i to the goal, then, through Assumption RA2, polygon p_i cannot be in polygon p_j's path. And hence the graph is in fact simple. The

Algorithm 3.1: Restricted Algorithm.

```
1  Let the k polygons be arranged according to R.A. 2, and let
       $$\mathcal{V} = \bigcup_{i=1}^{k} \mathcal{V}_i \bigcup \{v_r\} \bigcup \{v_g\}.$$
   $\mathcal{E}, \mathcal{W}$, and $\mathcal{C}$ are given as follows:
2  $i \leftarrow 1$
3  for $c \in \mathcal{V}_i$ do
4      if $\ell(c, v_g) \cap int(p_l) = \emptyset,\ l = 1, \dots, i$ then
5          $\mathcal{E}.add((c, v_g))$
6          $\mathcal{W}(c, v_g) \leftarrow \|c - v_g\|$
7          $\mathcal{C}(c) \leftarrow \mathcal{W}(c, v_g)$
8      else
9          $v' \leftarrow \text{argmin}_{v \in \mathcal{V}_l \mid \ell(v,v_g) \cap int(p_l) \neq \emptyset,\ l \leq i} \{\mathcal{C}(v) + \mathcal{W}(c, v)\}$
10         $\mathcal{E}.add((c, v'))$
11         $\mathcal{W}(c, v') \leftarrow \|c - v'\|$
12         $\mathcal{C}(c) \leftarrow \mathcal{W}(c, v') + \mathcal{C}(v')$
13     end
14 end
15 $i \leftarrow i + 1$
16 while $i \leq k$ do
17     if $\ell(v_r, v_g) \cap int(p_l) = \emptyset,\ l = 1, \dots, k$ then
18         $\mathcal{E}.add((v_r, v_g))$
19         $\mathcal{W}(v_r, v_g) \leftarrow \|v_r - v_g\|$
20         $\mathcal{C}(v_r) \leftarrow \mathcal{W}(v_r, v_g)$
21     else
22         $v' \leftarrow \text{argmin}_{v \in \mathcal{V}_l \mid \ell(v_r,v_g) \cap int(p_l) \neq \emptyset,\ l \leq k} \{\mathcal{C}(v)\}$
23         $\mathcal{E}.add((v_r, v'))$
24         $\mathcal{W}(v_r, v') \leftarrow \|v_r - v'\|$
25         $\mathcal{C}(v_r) \leftarrow \mathcal{W}(v_r, v') + \mathcal{C}(v')$
26     end
27 end
```

only remaining part of the proof is to connect the initial position, along a tangent segment, to the vertex that has the lowest cost, and the proof follows.

■

As an example, consider the scenario in Figure 3.8 where the restrictive assumptions do in fact hold.

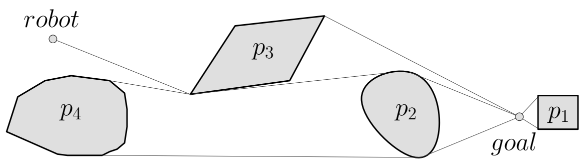


Figure 3.8: Sample Restrictive Geometry.

3.3.2 General Polygonal Environments

In this section we present an algorithm for maintaining the oriented visibility graph.

We will make use of the following notation. As before, we have the graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P})$. There exists a known goal polygon $p_g \in \mathcal{P}$ that is composed of a single vertex $v_g \in \mathcal{V}$, the goal vertex. Let $poly(v)$ denote the polygon $p \in \mathcal{P}$ to which v belongs. The source vertex and destination vertex of an edge e are denoted $src(e)$ and $dest(e)$ respectively. The robot vertex is denoted v_r.

3.3.2.1 Discovery

Recall from Chapter 1 that the path planning problem is composed of two parts, `generate-graph` and `find-path`.

One of the key ideas behind this algorithm can informally be called as "discovery". That is, we will solve the `generate-graph` problem be trying to connect obstacle polygons directly to the goal and discover what other polygons block this connection. When a new polygon is added to the graph, the "shadow-casting" vertices will be connected (hypothetically) to the goal vertex v_g. If this edge intersects the interior of any polygons in $\mathcal{P}$, a hypothetical edge is formed between one the vertices of these polygons (the shadow caster) and the new polygon. This discovery process continues, checking for a polygon to connect to, checking for intersections, and the looking for a new polygon to connect to, until the hypothetical edge does not intersect any polygons. This edge is added to $\mathcal{E}_v$. This is simplistic view of how the graph is generated.

Hence, the running time of the algorithm depends, in a sense, on the complexity of the environment. That is, our algorithm will use computation time that depends on the number of iterations of discovery that occur. Parasitically complicated geometric arrangements of polygons will incur the worst-case running time of the algorithm, but our approach is motivated by the need for fast re-planning in typical scenarios. In such situations, this discovery process can be very fast.

3.3.2.2 General Algorithm

The algorithm we present accepts as input an unordered list of polygon updates $\mathcal{P}_u$. That is, an element of the input list is either

1. a new polygon to be added to $\mathcal{P}$,
2. an existing polygon to be removed from $\mathcal{P}$, or
3. an existing $p \in \mathcal{P}$ with modified structure (i.e. its vertex list has changed).

The operation $blocks(p, e)$ returns true if the boundary of p intersects e. During the last step, only those polygons whose vertexes fall within the circle centered on the goal with radius $\|v_r - v_g\|$ or within a user-defined locus of the robot are actually considered. The reason for this is that polygons outside this set are unlikely to be encountered by the robot on its quest to the goal, and are probably not worth our effort.

3.3.2.3 Edge Assignment

By far, the final step carries the most computational burden, but before jumping into the details, we must introduce even more notation. The function $angle(v, g, p)$ returns the angle between the vector g to v and the vector g to the center of mass of p. The visibility between points s from x is returned by $visible(x, s, \mathcal{P}_k)$, considering only polygons in $\mathcal{P}_k$. Polygons which block the visibility of two points is returned by $blockers(s, g) = \{p \in \mathcal{P} \mid p \text{ blocks } line(s, g)\}$.

3.3.3 Example of the General Algorithm

Figure 3.9 illustrates the addition of polygon p_e to an existing oriented visibility graph. Each step (computing the shadow casting vertex, discovering blocking polygons, finding potential vertices to connect to) is presented individually. In (a), the graph is shown. The new polygon is added, disconnected, in (b). The clockwise shadow-casting vertex is highlighted on p_e in (c). This vertex v_1 is connected (hypothetically) to the goal vertex, and two polygons are found that block this connection, p_a and p_b (d). The vertex v_2 with minimum cost and not blocked by polygons $\mathcal{P}_b = \{p_a, p_b\}$ is found for p_b. No such vertex exists on p_a as they are allowed blocked by p_b. A new shadow-casting vertex is found on p_e, relative to v_2 (thought it turns out it is still v_1).

In (e), an edge is considered between these vertices. (f) Polygon p_c blocks this edge and is added to $\mathcal{P}_b$. (g) Now, the new vertex in $\mathcal{P}_b$ that is visible is computed: v_3. Relative to it, a new shadow-casting vertex is found on p_3: v_4. This edge is not blocked by any polygon in $\mathcal{P}$, and it is added to $\mathcal{E}_v$. (h) A similar process is run on the counter-clockwise

Algorithm 3.2: Oriented Visibility Graph Algorithm.

```
1. Find polygons "upstream" of $\mathcal{P}_u$:
   · Initialize a list of polygons $\mathcal{P}_{upstream}$ to $\mathcal{P}_u$.
   · for each $p \in \mathcal{P}_{upstream}$, for each $e \in \mathcal{E}$, if the $dest(e) \in vert(p)$, add $poly(src(e))$
   to $\mathcal{P}_{upstream}$.

2. Prune edges that will be modified:
   · for each $e \in \mathcal{E}$, if $poly(src(e)) \in \mathcal{P}_{upstream}$, remove $e$ from $\mathcal{E}$.

3. Remove all updated polygons:
   · for $p \in \mathcal{P}_u$, remove $p$ from $\mathcal{P}$.

4. Recreate new/modified polygons:
   · for nonempty $p \in \mathcal{P}_u$, create vertexes $vert(p)$, add to $\mathcal{V}$, and add $p$ to $\mathcal{P}$.

5. Remove blocked edges of unmodified polygons:
   · for $p \in \mathcal{P}_u$, for each $e \in \mathcal{E}$, remove $e$ if $blocks(p, e)$.

6. Sort the modified polygons and reset each the path cost of each vertex in these
   polygons:
   · sort $\mathcal{P}_{upstream}$ by $\underline{d}_g(p)$, $\forall p \in \mathcal{P}_{upstream}$.

7. Add edges and calculate path costs for all upstream polygons.
```

shadow-casting vertex on p_e and the two new edges are added to the graph.

3.3.4 Extension

We describe two additional extensions of the OVG algorithm. The term “extension” is used because in the implementation and application of our method discussed in Section 3.5, these methods are not employed.

3.3.4.1 Intersection Caching

When a polygon is modified (i.e. its vertex list is revised), we have described the process for handling this update as first a removal followed by an addition. As is common, a modification to a polygon may have little effect on its connectivity to the rest of the graph. The series of iterations in the edge-addition algorithm that check for blocking polygons can

Algorithm 3.3: OVG Edge Addition Algorithm.

```
foreach $p \in \mathcal{P}_{upstream}$ do
    $g \leftarrow v_g$
    find the shadow casting vertexes:
      $v_p^{c+} \leftarrow \arg\max_{v \in p}(angle(v, g, p))$
      $v_p^{c-} \leftarrow \arg\min_{v \in p}(angle(v, g, p))$
    foreach $s \in \{v_p^{c+}, v_p^{c-}\}$ do
        $O \leftarrow \mathcal{P}$
        $Q \leftarrow \emptyset$
        $P_b \leftarrow blockers(s, g, \mathcal{O})$
        $\mathcal{O} \leftarrow \mathcal{O} \setminus \mathcal{P}_b$
        repeat
            $\mathcal{V}_q \leftarrow \emptyset$
            for $q \in \mathcal{P}_b$ do
                $w \leftarrow \{x \in q \mid visible(x, s, \mathcal{P}_b)\}$
                $v_q = \arg\min_{x \in w}(\mathcal{C}(s) + \mathcal{W}(x, s))$
                append $v_q$ to $\mathcal{V}_q$
            end
            $g = \arg\min_{v \in \mathcal{V}_q}(\mathcal{C}(v))$
            recompute the shadow caster $s$
             recompute $\mathcal{P}_b = \mathcal{P}_b \cup blockers(s, g, \mathcal{O})$
            $\mathcal{O} \leftarrow \mathcal{O} \setminus \mathcal{P}_b$
        until $s$ and $\mathcal{P}_b$ stabilize
        add the edge between $s$ and $g$
    end
    assign path costs for $v \in \mathcal{V}_p$
end
```

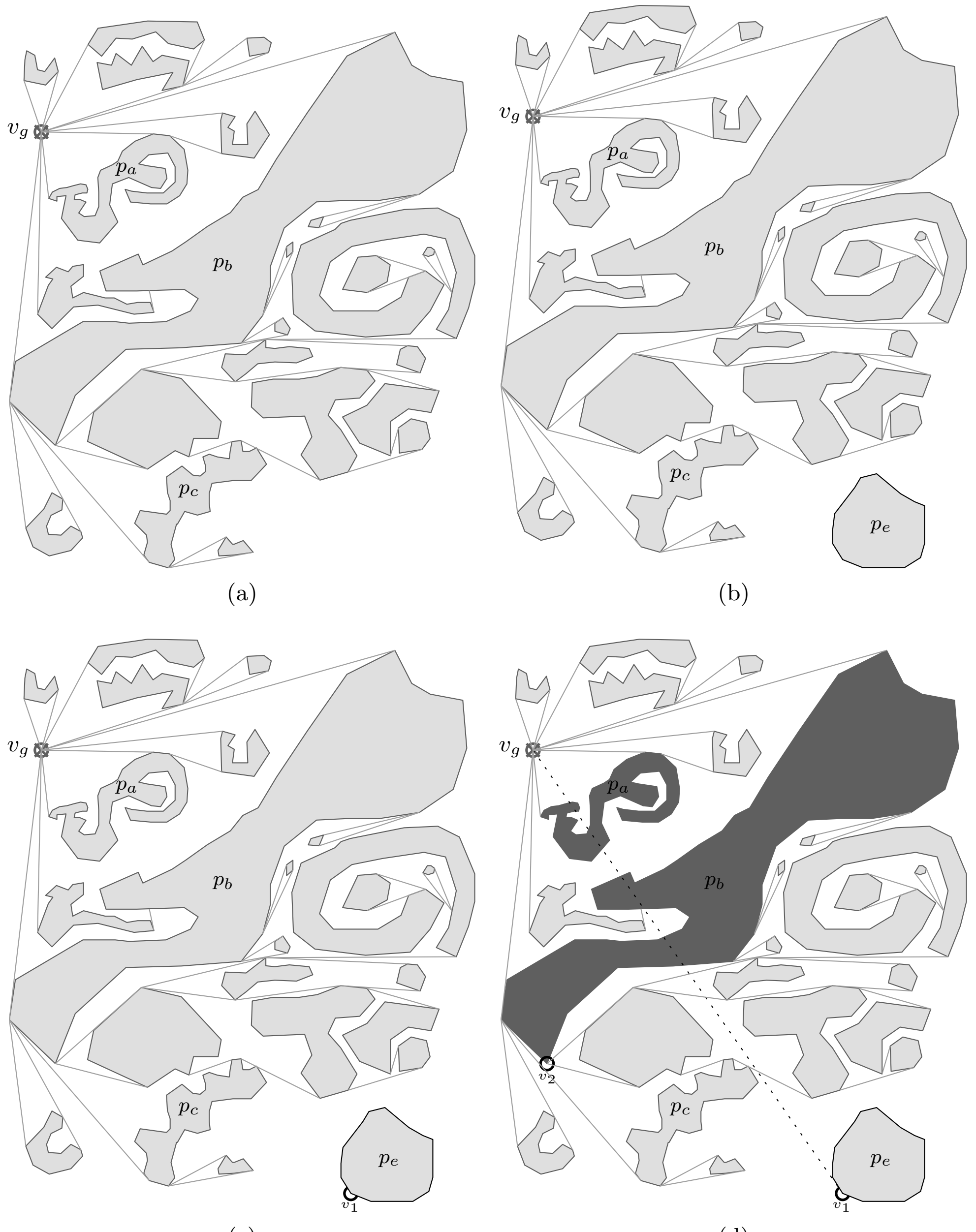


Figure 3.9: An example of the general OVG edge addition algorithm. (a) The graph is in this state when (b) A new polygon p_e is added. (c) The clockwise shadow-casting vertex v_1 is computed for p_e relative to v_g, the goal. (d) With the hypothetical edge v_1 and v_g, two blocking obstacles are discovered, p_a and p_b. (Continued in next figure...)

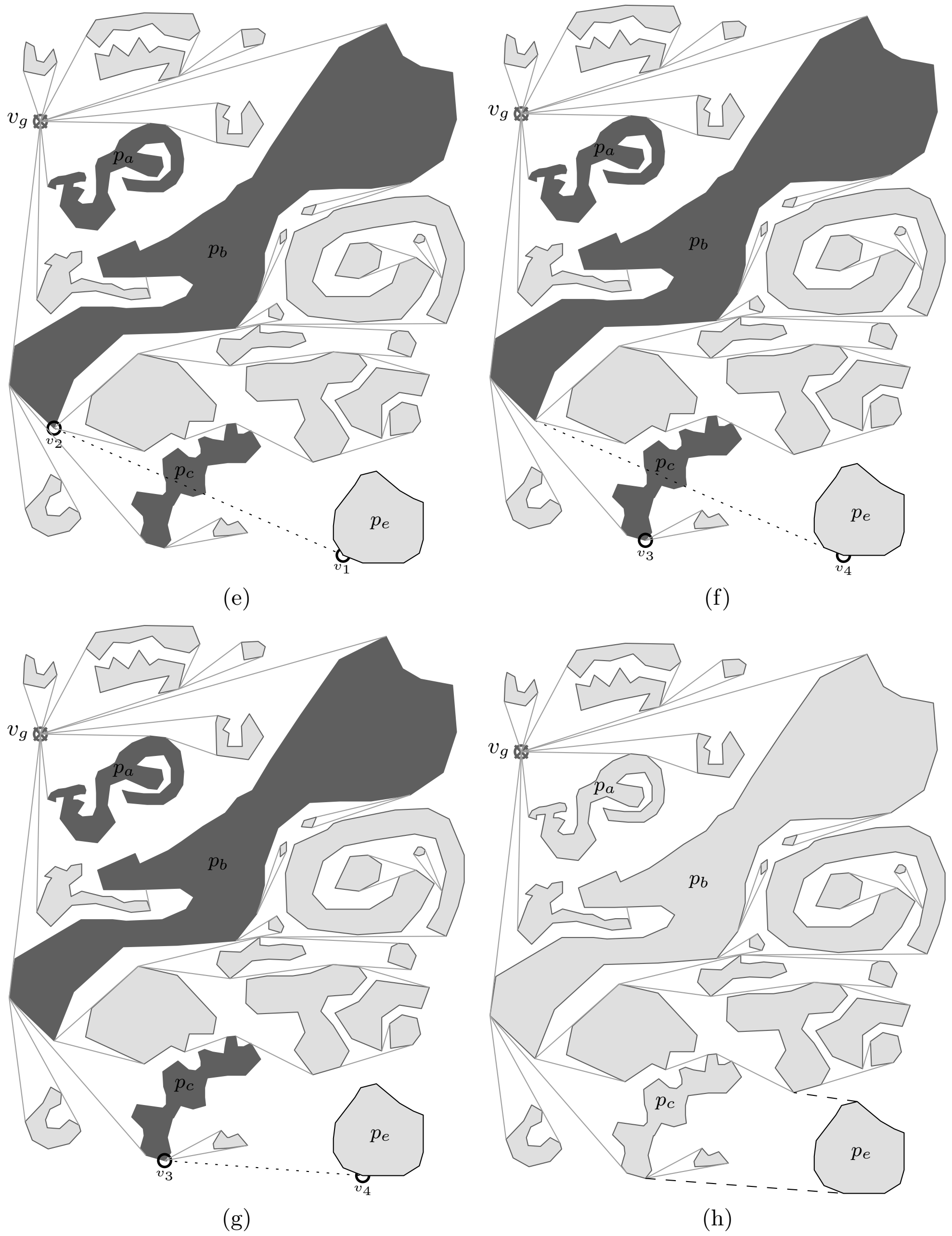


Figure 3.9: (Continued from previous figure. (e) The shadow-casting vertex is recomputed (and is still v_1) and the vertex from $\bigcup_{\mathcal{P}_b} vert(p)$ with minimum path cost to v_1 is found: v_2. (f) The hypothetical edge between v_1 and v_2 is blocked by p_c. Now $\mathcal{P}_b$ is composed of $\{p_a, p_b, p_c\}$. The vertex in $\mathcal{P}_b$ with minimum cost to v_1 is v_3. The shadow-casting vertex is recomputed with respect to v_3, resulting in vertex v_4. (g) The hypothetical edge between v_3 and v_4 is not blocked. (h) Finally inter-polygonal edges for polygon p_e are added to $\mathcal{G}$.

be short-cut by caching the polygons that were discovered from the previous addition of the polygon to the graph. Hence, the discovery process is shortened, which is where much of the computational burden of this approach is incurred. The worst-case running time is not reduced by using this extension to the algorithm, but a substantial reduction to the running time in an implemented version well likely be observed.

3.3.4.2 Polygon Dependency Graph

Recall that in the edge addition algorithm, repeated calls are made check for visibility of an edge against the set of polygons $\mathcal{P}$. This represents the most computationally burdensome "bottleneck" in our approach. The set of polygons queried by `blockers` can be reduced using the dependency inherent in the oriented visibility graph.

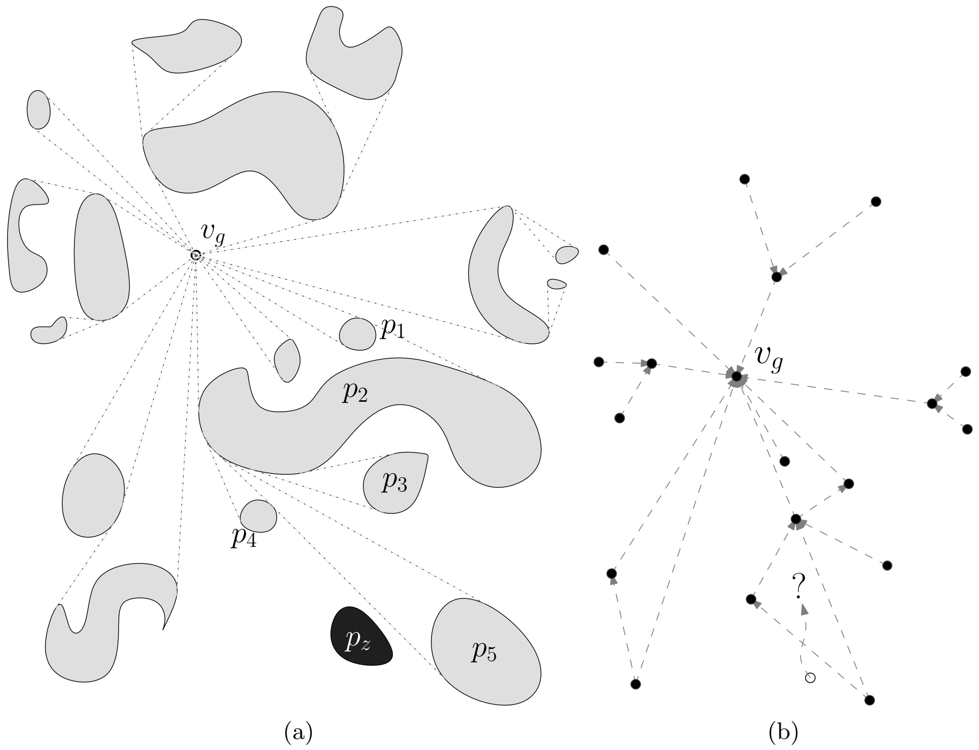


Figure 3.10: Example dependency graph. The OVG is shown in (a) with the corresponding obstacle dependency graph shown in (b).

That is, consider Figure 3.10. An OVG is shown in (a). In (b), the obstacles have been collapsed to vertices and each obstacles' out-going edges in $\mathcal{E}_v$ is used to define the edges in this graph. When a new polygon p is added to $\mathcal{G}$ the initial set of blockers $\mathcal{P}_b$ is computed by checking the edge between the goal an initial estimate of a shadow-casting vertex of p. This list of blocking polygons can be used to extract a subset of polygons from $\mathcal{P}$. The polygons not extracted are necessarily not able to be involved in the edge assignment of p. This necessary subset of blocking polygons can be found by traversing the dependency graph from each initial blocking polygon to the goal.

This approach does not reduce the worst-case complexity of the edge addition algorithm, however. This method would be effective in scenarios where extensive knowledge of the environment is known, such that the fraction of dependent polygons is small.

3.3.4.3 From single-source to point-to-point

The algorithm we have described so far maintains the entire visibility graph, regardless of where the robot is. That is, the OVG is defined to solve the single-source planning problem. When all that is required is the solution to the point-to-point problem, the edge-addition algorithm should ignore those polygons that are out of the pertinent portion of the graph relative to the robot's current position. This has the effect of disconnecting polygons that are out of the scope of the point-to-point problem. Should the robot venture back to where these polygons become pertinent again, the discovery process will induce the edge addition algorithm to re-add the edges to these obstacles.

This does not reduce the worst-case running time analysis of the algorithm, but the computation used in practice would however be reduced substantially in many scenarios.

3.3.4.4 Polygons that are not piece-wise linear

The polygons that are used in our application of this approach to path planning, as well as the example used in the motivation of this chapter have all been piecewise linear approximations of the real-world obstacle. What the OVG method requires is the ability to compute the intersection of an edge in the graph with the polygonal obstacles. As long as this determination can be made, the polygons may be described differently, e.g. as cubic splines.

3.4 Analysis

3.4.1 Complexity

For this complexity analysis, we assume the number of vertices in any polygon is $O(1)$, meaning that as the number of vertices (or polygons) in the graph grows, the number of vertices per polygon is roughly constant.

First, let us establish the complexity of three functions used in the edge addition algorithm.

Finding the shadow-caster vertices takes $O(1)$ for a polygon. The vertices of a polygon are sorted relative to their angle with respect to a given point.

The function $blocker(a, b, \mathcal{O})$ takes $O(card(\mathcal{O}))$, since it takes constant time per polygon p to check if p blocks line $\overline{ab}$.)

The function $visible(a, b, \mathcal{Q})$ is almost the same function as $blockers$, and similarly requires $O(card(\mathcal{Q}))$.

The result of the `for`-loop on line 12 is that $\mathcal{V}_q$ contains the node from each polygon in $\mathcal{P}_b$ that is visible to the shadow-casting vertex c and is not blocked by any polygon in $\mathcal{P}_b$. This is $O(card(\mathcal{P}_b)^2)$.

In the worst case – when each iteration of the `do`-loop on line 10 appends one more polygon to the set $\mathcal{P}_b$ – the edge addition algorithm will take $O(P^3)$ time per updated polygon, where P is the number of polygons in the graph. While this worst-case time is not encouraging, it is a very conservative bound; the running-time in practice is much lower.

We expect that in most applications, a small number of polygons will be involved in the blockers-visible interaction of the edge addition algorithm. By taking the assumption that the cardinality of $\mathcal{P}_b$ is always less than some constant $k = O(1)$, then we find the `for`-loop takes $O(k^2)$ time and the `do`-loop takes $O(k^2\ card(\mathcal{O}))$ time. This implies that the overall algorithm takes $O(card(\mathcal{O}) + k^2\ card(\mathcal{O})) = O(card(\mathcal{O}))$, under this assumption.

In other words, given k, *the time to dynamically update the graph grows linearly with the size of the graph.* The value k can be thought of as a measure of the clutter of the

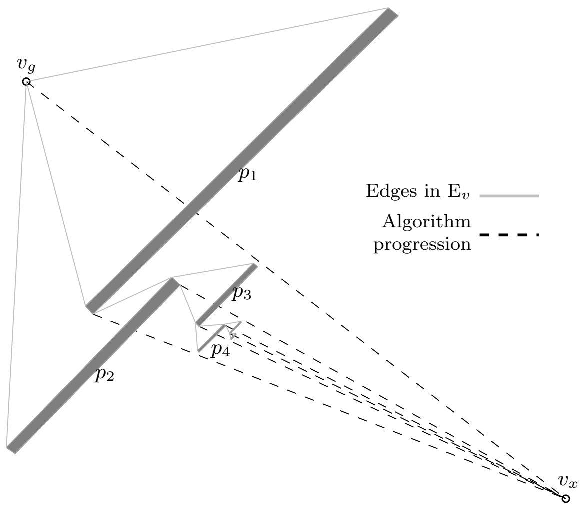


Figure 3.11: A worst-case scenario for the OVG algorithm. One new blocking polygon is discovered per iteration in the outer loop of the edge addition algorithm. Eventually, all polygons have individually been found through discovery (indicated by dashed lines) and the source vertex v_x is added to the graph.

environment, and so in a sense, our approach is "clutter dependent". This dependency is a direct result of our knowledge of a static goal point combined with the use of adding two edges per polygon.

3.4.1.1 Programming Complexity

An important observation about the edge addition algorithm is its low "programming complexity" – the complexity of the tools needed to implement the algorithm. The sets of vertices, edges and polygons require a structure that supports constant-time `contains`, `get` and `put` methods (such as a HashSet – a standard *Collections* class in Java) and the ability to compute the intersection of two lines. The only other functions required by the edge addition algorithm are `visible` and `blockers` which accept a line segment and a set of polygons as input and outputs whether an intersection occurs or what polygons intersect the segment, respectively. These functions follow directly from computing the intersection of two line segments.

Notably absent from our approach is the need to do any kind on sorting on vertices or edges. This, along with triangulation, is a key component of the classic static reduced visibility graph algorithms. Sorting, which may lead to better worst-case running time, can be expensive in terms of computational cost incurred during execution. On the other hand, our primary geometric primitive – detecting the intersection of two lines – can be computed by the calculation of the determinant of a 3×3 matrix [46]. The intersection of an edge with an obstacle can be computed by first checking if the rectangular bounding-box of the polygon intersects the edge. This method leads to both an easy-to-code algorithm and an implementation that has very low constant factors.

3.4.2 Suboptimality

3.4.2.1 Suboptimality in Convex Polygons

Figure 3.12 shows a simple environment where there exist only two polygonal obstacles. The four dash-dotted lines indicate where the oriented visibility graph edges would be added. The four dotted lines show where edges would be added (in addition to the OVG edges).

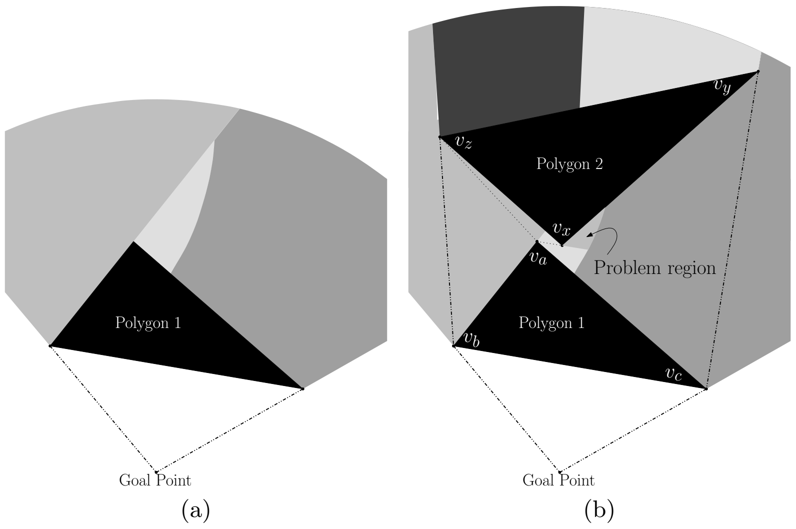


Figure 3.12: Example of a geometry that results in suboptimal paths over the corresponding oriented visibility graph. The problem region in (b) identifies the free space that points to the wrong vertex in the graph. The edges added via the OVG point this region to vertex v_c. The RVG (correctly) points this region to vertex v_x.

3.4.2.2 Suboptimality in Non-Convex Polygons

The two-edge principle fails for some geometric configurations of non-convex polygons. Figure 3.13 illustrates such an example. Recall that path costs are propagated around the boundary of the polygon, seeded by the path costs at the shadow-casting vertices. In the case of Figure 3.13, the propagation of path costs through the concave region induces the wrong bifurcation of the shadow region "behind" the polygon and the path costs at those vertices is overestimated.

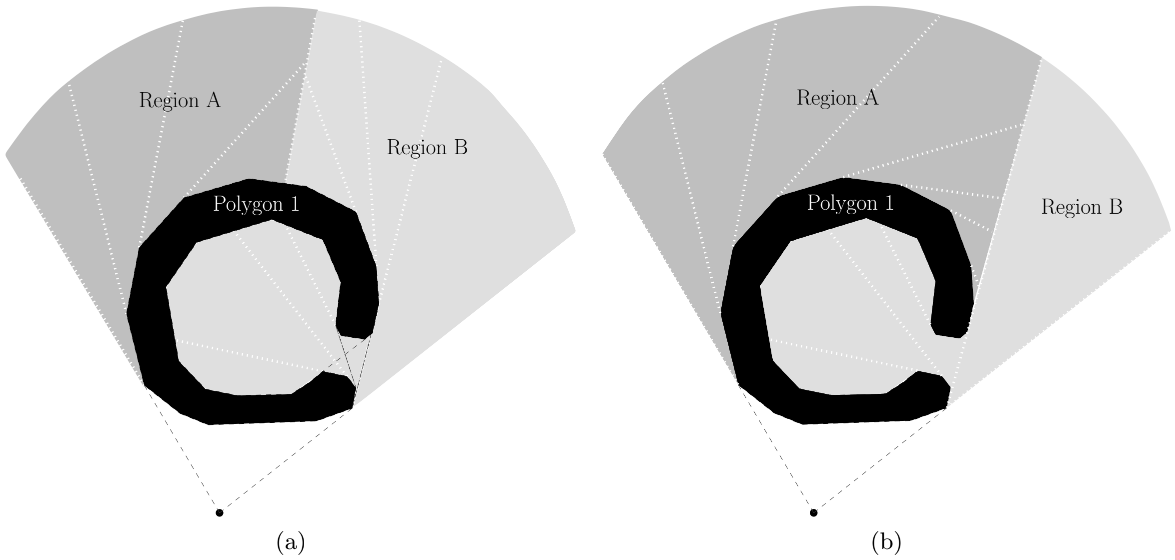


Figure 3.13: Example of suboptimality caused by simple non-convexity of a polygon. (a) Diagram of region bifurcation after optimal edge assignment (e.g. from an RVG algorithm). (b) Diagram of bifurcation after edge assignment from an OVG algorithm.

3.4.2.3 The Banana Problem

The oriented visibility graph is a suboptimal roadmap due to a third reason: the mutual intersection of two polygons' convex hulls. Consider Figure 3.14. In this example, three

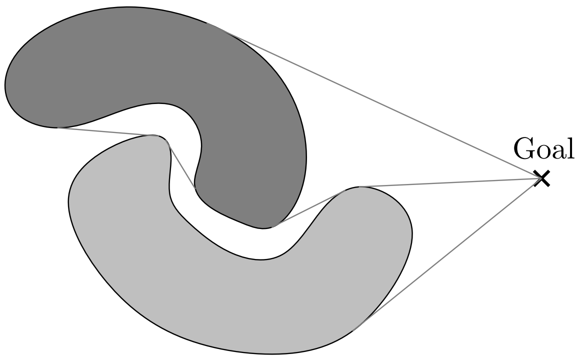


Figure 3.14: Illustration of the "banana" problem – two interlocking concave polygons may violate the two-edge principle. Both polygons require 3 edges to provide an optimal roadmap over C_{free}.

edges are needed for both polygons to produce the optimal roadmap to v_g. Of course, the OVG edge additional algorithm is designed to add only two edges per polygon. As a result, a feasible roadmap is produced (where all points in C_{free} can be connected to a valid vertex in $\mathcal{G}$), it is not optimal. Similar to how shadow regions were directed to the wrong side of the polygon in Section 3.4.2.2, the same may be true here, where the roadmap produced points portions of the shadow region to the wrong side of the union of the two polygons.

3.5 Application

In this section, we describe in brief how we have applied our Oriented Visibility Graph to the navigation of a ground robot that is given a fixed goal point in GPS coordinates, and through a GPS receiver knows approximately its own location. It is expected to traverse the outdoor environment and reach the goal in as little time as possible. Information about its surroundings is gathered through only small cameras (two stereo pairs) and a bump sensor. These cameras produce stereo maps of 4-6 meters in maximum depth, which are used in turn to accumulate an elevation map of the terrain. Also, a traversability map is produced from the raw images and combined with elevation to determine where the robot may travel to reach the goal.

Navigation commands are provided to the robot through a planner based on an OVG. This planner listens to the elevation and traversability map datastreams, and updates its graph accordingly.

The two stereo pairs generate stereo disparity maps at 4Hz each. This information runs through the process described in Section 3.5.1 and polygon updates are handed to the planner. The planner directly produces motor commands for the robot, and runs between 4 and 20 Hz. (The robot *must* receive motor commands at above 2Hz or otherwise behaves undesirably.) Our planner's average cycle time is above 5Hz.

The robot has been tested in outdoor courses with total distances over 100 meters, in open terrain, on paths through woods, and under tree canopy without trails. It is given three runs to attempt the same course, starting from about the same location. At the end of the first and second run, the robot saves its graph so that it may be re-loaded at the initiation of the second and third runs. It is the planner's job to find its way out of cul-de-sacs as it discovers them, and avoid them all together if it returns to them. The outdoor environment contains both natural and man-made cul-de-sacs and non-convex polygons to challenge the robot.

3.5.1 Generating Polygons from Sensor Data

We consider only data streams that correspond to Cartesian image maps, inasmuch as our visibility graph is presented here as strictly 2-D. With each stream, the graph is informed about the likely presence of some object type at a specific location, and the variance estimate of that likelihood. These multiple likelihood and variance maps are subsequently combined via a function $c(...)$ into a single likelihood-of-obstacle image L. An example of c, appropriate for the block diagram in Figure 3.15, is

$$c(s, v_s, t, v_t) = \begin{cases} s, & \text{if } v_s < \theta_{v_s},\ t < \theta_t,\ v_t < \theta_{v_t} \\ 0, & \text{otherwise,} \end{cases}$$

where s refers to a sort of first derivative of elevation, t is the computed traversability computed for a pixel location, and v_s and v_t refer to the variance of measurements of s and t. The various θ_i refer to user-defined threshold parameters for t, v_t, and v_t. We compute s at a pixel location by the well-known Sobel operator [32].

Now, L must be transformed via a binary decision-making function $d(L(i,j))$, identifying which pixels the robot can traverse. The simplest non-trivial decision function is naturally

$$d(L(i,j)) = \begin{cases} \textit{obstacle}, & \text{if } L(i,j) > \theta_l \\ \text{not } \textit{obstacle}, & \text{otherwise} \end{cases}$$

where θ_l is some threshold on L.

Hence, let T be the mapping from the real-valued map L to the binary obstacle map M,

$$T : \mathbb{R}^{nxm} \rightarrow \mathbb{B}^{nxm}. \tag{13}$$

By applying d at each location of L, we transform L into M.

Obstacle points in M are segregated and labelled based on any typical segmentation technique.

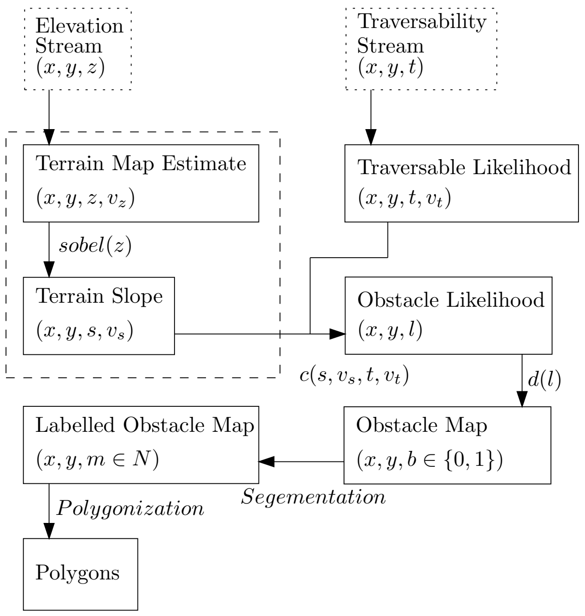

Elevation Stream (x, y, z)
Traversability Stream (x, y, t)
Terrain Map Estimate (x, y, z, v_z)
Traversable Likelihood (x, y, t, v_t)
$sobel(z)$
Terrain Slope (x, y, s, v_s)
Obstacle Likelihood (x, y, l)
$c(s, v_s, t, v_t)$
$d(l)$
Labelled Obstacle Map $(x, y, m \in N)$
Obstacle Map $(x, y, b \in \{0, 1\})$
$Segementation$
$Polygonization$
Polygons

Figure 3.15: Traversability Streams to Polygons.

Of course, all the operations in Figure 3.15 are incremental. So, updates are passed in the form of individual pixel modifications, and operations like segmentation are performed on a pixel-by-pixel basis.

3.5.2 Polygonization

The labelled obstacle map of Figure 3.15 is polygonized for input to the OVG-based planner. Polygonization is performed according to the following steps:

1. A mathematical-morphology dilation operation with a circular structuring element is applied to the labelled obstacle points for each obstacle (e.g. Figure 3.16). This dilation accommodates the physical geometry of the robot, allowing it to maintain an appropriate distance between it and obstacles.

2. By starting at any point on the boundary of the dilation from Step 1, the closed-contour set of pixels can be generated by iteratively stepping from one pixel to the next.

3. The boundary walk of Step 2 produces more pixels than necessary to accurately represent the obstacle; a reduction of these vertexes can be performed. Let δ_i be distance from a vertex v_i to the line formed by its two neighbors along the boundary. By removing those vertexes with δ less than some threshold, a representation of the obstacle is found which has fewer vertexes. Of course, fewer vertexes per polygon implies decreased running time, but tends to misrepresent the obstacles that the robot is to avoid.

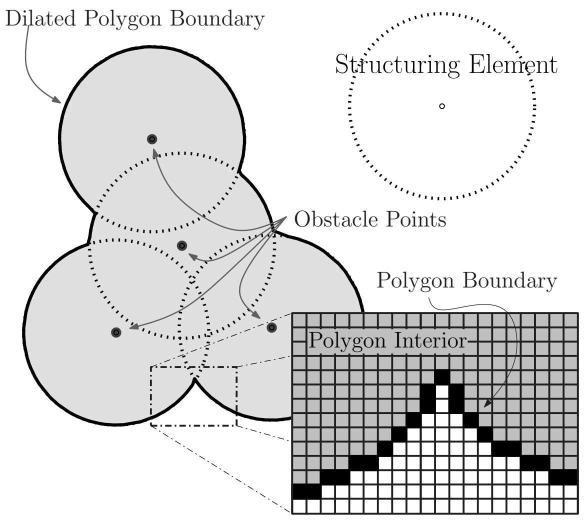


Figure 3.16: Sample Polygonization with Circular Structuring Element.

3.5.3 Samples from Application

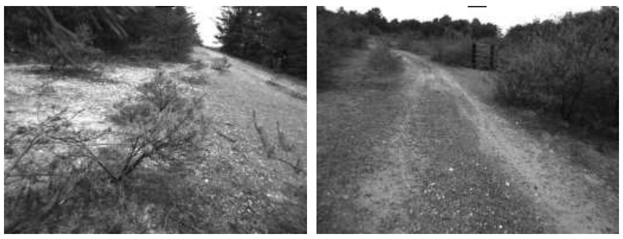


Figure 3.17: Sample stereo images.

Images such as in Figure 3.17 are used to form stereo disparity maps and the elevation stream for Figure 3.15. Figure 3.18 illustrates the graph structure overlaid on the elevation map of a test run. Polygons are shown in white, and edges are black. Here, each pixel

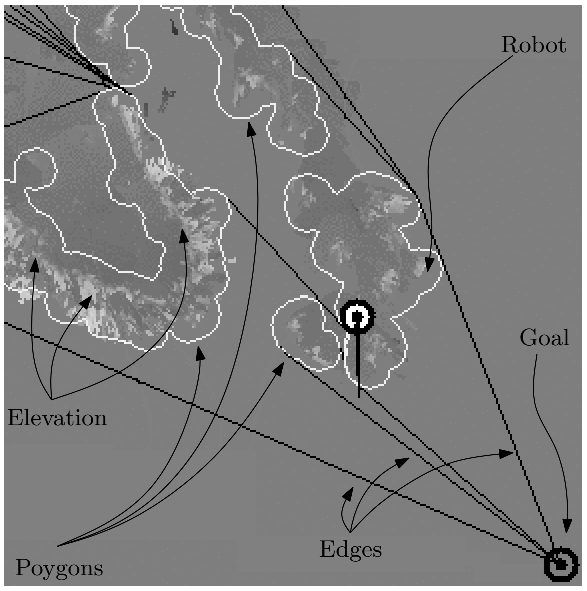


Figure 3.18: Sample terrain and graph.

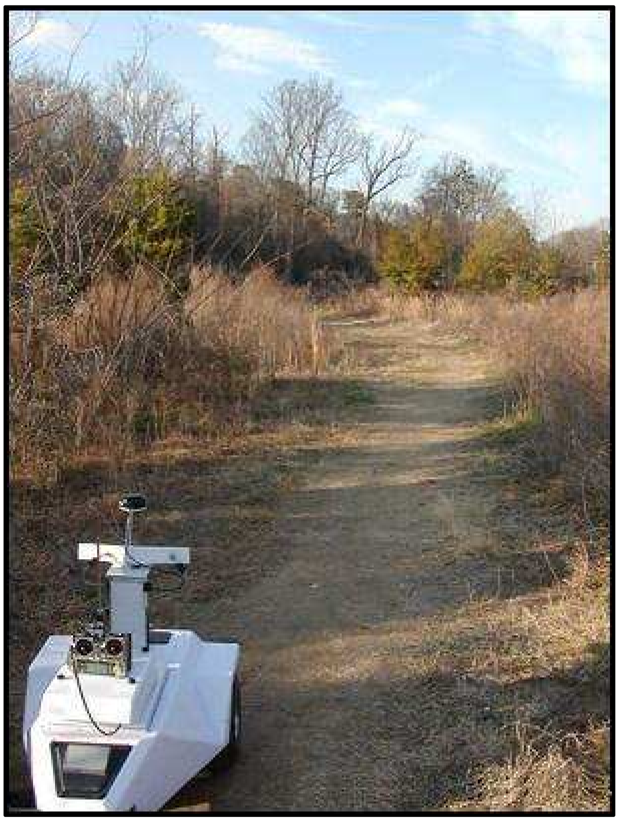


Figure 3.19: Sample Image of robot and terrain.

represents a $0.1m \times 0.1m$ square. Graphs typically contain as many as 100 polygons of various sizes, are composed of thousands of vertexes, and cover more than 100 meters from start to finish. Even with the naive Algorithms presented above, the planner still operates fast enough for our real-time system.

3.6 Summary

In this chapter, we derive a visibility graph based roadmap construction for unstructured polygonal environments known as the oriented visibility graph. The unique quality of this construction stems from the fact that we insist on a given, fixed goal point and allow for possibly suboptimal paths in the resulting roadmap. Real-world experiments illustrate the usefulness of the proposed method in time-critical outdoor applications where the perception is based solely on stereo-based elevation maps. These maps are polygonized in order to support the use of the planner. By saving the graph between runs, dynamic update rules (for adding, removing, or changing polygons) enable the robot to improve its performance over runs.

CHAPTER IV

HIERARCHICAL ORIENTED VISIBILITY GRAPHS

4.1 Introduction

This chapter describes a hierarchical variation on the oriented visibility graph. Our approach is to compose groups of polygonal obstacles based on the intersection of their convex hulls. Within a group s, edges are added among the polygons of s to form a reduced visibility graph. Subsequently, among the groups, two edges are added per polygon in a way similar to that which was done for the pure oriented visibility graph.

4.2 Motivation and Approach

Consider an oriented visibility graph where $\mathcal{P} = \{p\}$ and p is the non-convex polygon shown in Figure 4.2a. The shaded regions in the figure indicate the vertex that the free space in that region points to on its path to the goal. Path costs are propagated around the boundary of p, causing some regions to suboptimality point to the left side of the polygon instead of the right.

However, if the edges that define the convex hull of p, $\mathcal{E}_c(p)$, are added to the graph (as shown in Figure 4.2b), the graph becomes an optimal roadmap. This is because path costs are propagated not just over the polygon boundary, but also over $\mathcal{E}_c(p)$. In general, more than just the edges of the convex hull are needed to provide an optimal roadmap. The reduced visibility edges for the isolated polygon p contain the edges that form the convex hull of p.

Let $C_V(p)$ denote the ordered set of vertices that form the convex hull of the polygon p and let $C_E(p)$ denote the corresponding ordered set of edges. Let $RV(p)$ denote the set of edges that correspond to the reduced visibility graph edges where the input graph is simply p. It follows then that $C_E(p) \subseteq RV(p)$.

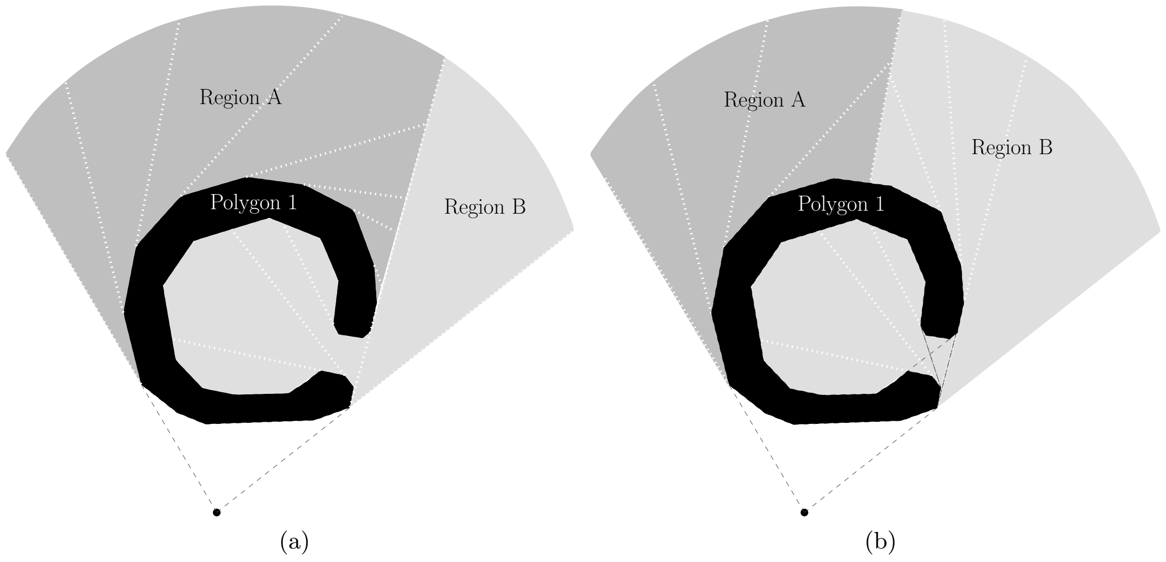


Figure 4.1: An OVG with a single polygon.

Figure 4.2: Removal of suboptimality in C_{free} by adding the reduced visibility edges (for the case of non-intersecting convex hulls).

The suboptimality of planned paths in an oriented visibility graph as caused by non-convexity of polygonal obstacles can be characterized as occurring either inside or outside a polygon's convex hull. The suboptimality in the exterior of all convex hulls can trivially be removed by augmenting $\mathcal{E}_v$ with $RV(p)$ for all $p \in \mathcal{P}$ in the case where

$$\{p \in \mathcal{P} \mid C_E(p) \cap C_E(q) \; \forall q \in \mathcal{P}, q \neq p\} = \emptyset. \tag{14}$$

That is, in case where the convex hull of no polygon in $\mathcal{P}$ intersect the convex hull of any other polygon in $\mathcal{P}$. Adding the reduced visibility edges (that is, not just the convex hull) removes suboptimality in both in interior and exterior of the polygon.

By lifting the assumption of Eq. 14, the problem of removing suboptimality becomes nontrivial. Our approach to solving this problem will be that of detecting the hull intersection of polygons and grouping them into "super-polygons". Within these groups, we can guarantee that we produce an optimal roadmap by adding the reduced visibility graph edges. Then, the polygons are connected to the rest of the graph (i.e. polygons not belonging to the same super-polygon) in a way similar to that of the original oriented visibility graph.

4.3 Algorithms

Just as in Chapter 3, the input to the algorithm we present below is an unordered list of polygon updates $\mathcal{P}_u$. Recall the graph used to describe the oriented visibility graph was $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P})$. We let the graph for the HOVG be $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{P}, \mathcal{S})$ where $\mathcal{S}$ is a set of "meta" or "super" polygons.

Each super-polygon $s \in \mathcal{S}$ is a subset of $\mathcal{P}$. The set of super-polygons $\mathcal{S}$ is a *partition* of $\mathcal{P}$. Let $vert(s)$ denote the set of vertices contained in the elements (i.e. the polygons) of $s \in \mathcal{S}$. Let $hull(s)$ denote the convex hull of $vert(s)$ and let $poly(s) \subset \mathcal{P}$ denote the set of polygons belonging to s. As a partition, $\mathcal{S}$ must satisfy three rules:

1. No element of $\mathcal{S}$ is empty.

$$s \neq \emptyset, \forall s \in \mathcal{S} \tag{15}$$

2. The elements of $\mathcal{S}$ *cover* $\mathcal{P}$. That is,

$$\bigcup_{s \in \mathcal{S}} poly(s) = \mathcal{P} \tag{16}$$

3. The elements of $\mathcal{S}$ are pairwise disjoint.

$$s_1 \cap s_2 = \emptyset, \forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2 \tag{17}$$

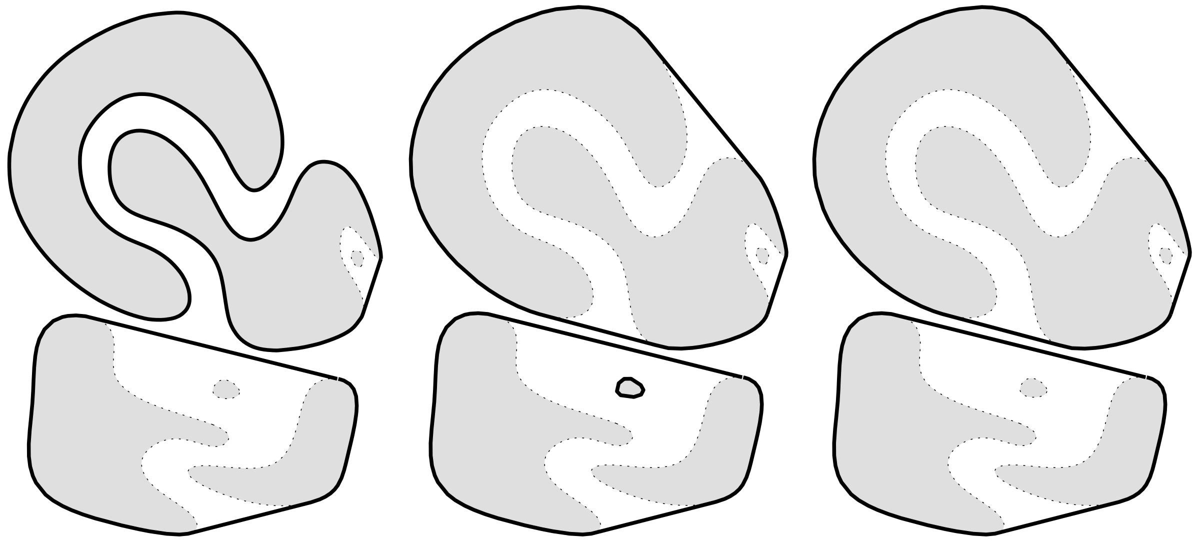
Let $\mathcal{S}$ satisfy the following rule

$$hull(s_1) \cap hull(s_2) = \emptyset, \forall s_1, s_2 \in \mathcal{S}, s_1 \neq s_2 \tag{18}$$

for which there is a unique $\mathcal{S}$ (given $\mathcal{P}$). An illustration of a super-polygon partitioning is shown in Figure 4.3.

For the OVG, the main geometric primitive that we needed to be able to compute was the intersection of two lines (and as a derivative, the intersection of a line and a polygon). For the HOVG, we will need two more. We will need to compute the convex hull of a polygon and of a set of polygons, and we will need to determine if two convex polygons intersect.

As before, an element of the input list is either



(a) Not a good partition (b) Not a good partition (c) A good partition

Figure 4.3: Illustration of good and bad super-polygon partitions of a set of polygons. Super-polygons are shown as black outlines of the gray polygons. (a) Bad because the top two polygons' hulls overlap. (b) Bad because the tiny hull intersects (is contained within) the bottom super-polygon. (c) A good partition.

1. a new polygon to be added to $\mathcal{P}$,
2. an existing polygon to be removed from $\mathcal{P}$, or
3. an existing $p \in \mathcal{P}$ with modified structure (i.e. its vertex list has changed).

The overall HOVG algorithm is described in Algorithm 4.1. It can be summarized as a five step process: remove, add, partition, prune, and add-edges. First, `remove` cuts the vertices, edges, and polygons in the current update from the graph. Second, `add` inserts the new/updated vertices, edges, and polygons. For the polygons that are removed or retracted, `partition` re-evaluates the $\mathcal{S}$ partition of $\mathcal{P}$. For the polygons that have grown, `prune` checks existing edges in $\mathcal{E}_v$ for intersection with these polygons. Finally, `add-edges` adds the inter-polygonal edges for the polygons modified in steps 1-4. The individual functions for appending polygons to super-polygons, partitioning disjoint super-polygons, and adding inter-polygonal connectivity is described in the following subsections.

Algorithm 4.1: Hierarchical Oriented Visibility Graph Algorithm.

```
1. Remove all updated polygons:
   · for $p \in \mathcal{P}_u$, remove $p$ from $\mathcal{P}$.

2. Add new/modified polygons:
   · for nonempty $p \in \mathcal{P}_u$, create vertexes $vert(p)$, add to $\mathcal{V}$, and add $p$ to $\mathcal{P}$. ·
   find-and-unify-super($p$)

3. Partition super obstacles if induced by a deletion/modification:
   · partition-super

4. Prune blocked edges of unmodified polygons:
   · for $p \in \mathcal{P}_u$, for each $e \in \mathcal{E}$, remove $e$ if $blocks(p, e)$.

5. Add Edges to modified super obstacles.
   · add-edges
```

4.3.1 The `add` function

The first three steps of this operation are the same as for the OVG algorithm: the vertices of the new polygon p are added to $\mathcal{V}$. The edges that define the polygon boundary are added to $\mathcal{E}$. The polygon p is added to $\mathcal{P}$. What remains is for the new polygon to be associated with a super-polygon. This is performed by the `fetch` function (see below), which either creates a new super-polygon to add to $\mathcal{S}$ or returns a list of existing super-polygons in $\mathcal{S}$ that the convex hull of p intersects.

So, if a new polygon induces a new super-polygon, it must simply be added to the set $\mathcal{S}$. Otherwise, the returned list must be unified into a single super-polygon (and the obsolete super-polygons removed).

The convex hull of the new super-polygon s must be computed. In the case where s contains a single polygon p, the convex hull can be computed in $O(m)$ time (with m the number of vertices of p). If s contains multiple polygons, the hull can be computed (e.g. by Graham's scan) in $O(m\ log(m))$ time.

4.3.2 The `fetch` function

A result from [73] and [54] is that the intersection of two polygons of size x and y can be computed in time $O(x+y)$. It follows then that the intersection of a convex polygon p with

m vertices with each of the k polygons in $\mathcal{P}$, a set of disjoint polygons with a grand total of n vertices, can be answered in $O(km + n)$ time with a simple algorithm. As previously stated, we expect to find in practice that the number of vertices per polygon is relatively constant, which implies that $O(km) = O(n)$, and hence that the detection of an intersection of the hull of a polygon p with every other "super obstacle" can be found in $O(n)$ time.

Algorithm 4.2 describes the `fetch` operation.

Algorithm 4.2: The `fetch` function.

```
R ← S;
h ← the convex hull of the input polygon ;
done ← false ;
Q ← ∅;
while not done do
    O ← {q ∈ R | q ∩ h};
    R ← R \ O;
    Q ← Q ∪ O ;
    h ← the convex hull of the union of h and O;
    done ← (O == ∅);
end
```

Worst-case Analysis: The `while`-loop may cycle at most $k = card(\mathcal{S})$ times (where each check for intersection inside the loop returns exactly one more super-polygon). Overall, this algorithm takes $O(kn)$ time. The lower-bound running time, which occurs in the case where the intersection of h (the convex hull of the input polygon) with all the super-polygons is established in the first call to the polygon intersection function, is $\Omega(n)$.

4.3.3 The `partition` function

The removal or retraction of a polygon p from $\mathcal{G}$ may induce a new $\mathcal{S}$ partition of $\mathcal{P}$. More specifically, s, the super-polygon to which p formally belonged may need to be "split" into two or more super-polygons. In other words, when the boundary of p in super-polygon s is retracted, its convex hull may no longer intersect the convex hull of $q, \forall q \in poly(s)$. An example of this is illustrated in Figure 4.3.3.

There are a number of ways of determining if and how s should be re-partitioned. (1) There is the static method, that of rebuilding the s from $poly(s)$ and checking if the result is more than one super-polygon. (2) A graph $G_s = (V_s, E_s)$ can be maintained and associated

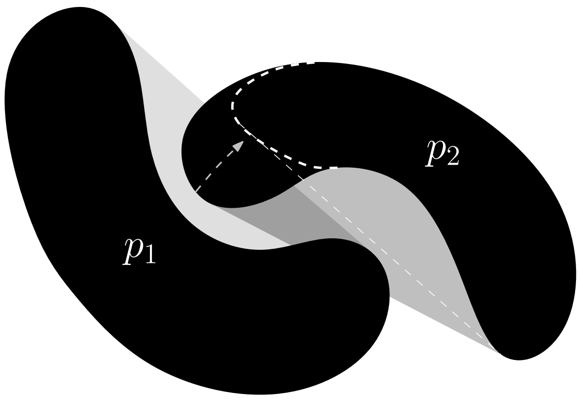


Figure 4.4: An example polygon modification that leads to a partitioning of a super-polygon. Polygon p_2 is retracted, leading to an empty intersection of the convex hulls of p_1 and p_2. What was one super-polygon must be partitioned into two.

with the super-polygon s where $V_s \subset \mathcal{V}$ is a set of vertices of s and $E_s \subset \mathcal{E}$ is a set of edges connecting V_s as defined by the convex hull intersections of each $p \in s$. When a polygon is removed or modified, the connectivity around the corresponding vertex in G_s is updated. The connectedness of G_s identifies the partitioning of s, which can be determined by, for example, a breath-first search.

4.3.4 The `add-edges` function

Edges are added as a two step process. Step 1 is to add all the bitangent edges among the polygons in each new or modified super-polygon. Using one of the traditional reduced visibility graph algorithms [31,47,57], this can be done in $O(m^2)$ time, where m is the number of vertices in the super-polygon. Recall from Chapter 1 that allowing for dynamic polygon updates precludes a dynamic RVG algorithm. By partitioning $\mathcal{P}$ into $\mathcal{S}$, we recompute a local RVG within each modified super-polygon. Given that the number of modified polygons is small and the number of polygons in each modified super-polygon is small (where small is relative to the cardinality of $\mathcal{P}$), The time used for a k polygon updates is $O(km^2)$. Here, "small" implies $o(\frac{m}{n})= 0$ and that $O(km^2)$ is negligible compared to a static RVG: $O(n^2)$.

For Step 2 inter-super-polygonal edges are generated through Algorithm 4.3. This is based on the edge addition performed on an OVG, Algorithm 3.3. The first main difference here is that instead of searching over $\mathcal{P}$, edges discovered by searching over $\mathcal{S}$. The second main difference is that out of the list if blocking obstacles $\mathcal{P}_b$, an edge is only allowed to be created between vertices belonging to *different super-polygons*. An edge not satisfying this restriction would be redundant, as the local RVG produced in Step 1 would necessarily already contain such an edge.

Algorithm 4.3: HOVG Edge Addition Algorithm.

```
1  foreach $p \in \mathcal{P}_{upstream}$ do
2      set $g = v_g$ ;
3      find the shadow casting vertexes:
4        $v_p^{c+} = \arg\max_{v \in p}(angle(v, g, p))$ ;
5        $v_p^{c-} = \arg\min_{v \in p}(angle(v, g, p))$ ;
6      foreach $c \in \{v_p^{c+}, v_p^{c-}\}$ do
7          set $\mathcal{O} = \mathcal{S}$ ;
8          set $\mathcal{Q} = \emptyset$ ;
9          $\mathcal{S}_b = blockers(c, g, \mathcal{O})$ ;
10         $\mathcal{O} = \mathcal{O} \setminus \mathcal{S}_b$ ;
11         repeat
12             set $\mathcal{V}_q = \emptyset$ ;
13             foreach $q \in \mathcal{S}_b$ do
14                 $w = \{x \in V(q) \mid visible(x, c, \mathcal{S}_b)\}$ ;
15                 $v_q = \arg\min_{x \in w}(\mathcal{C}(c) + \mathcal{W}(x, c))$ ;
16                 append $v_q$ to $\mathcal{V}_q$ ;
17             end
18             $g = \arg\min_{v \in \mathcal{V}_q}(\mathcal{C}(v))$ ;
19             recompute the shadow caster $c$ ;
20             recompute $\mathcal{S}_b = \mathcal{P}_b \cup blockers(c, g, \mathcal{O})$ ;
21             set $\mathcal{O} = \mathcal{O} \setminus \mathcal{S}_b$ ;
22         until $c$ and $\mathcal{S}_b$ stabilize ;
23         add the edge between $s$ and $g$ ;
24     end
25     assign path costs for $v \in \mathcal{V}_p$ ;
26 end
```

4.3.5 Complexity Analysis

Here, we provide a complexity analysis of the HOVG algorithm.

The `remove` operation is $O(1)$ per updated polygon. The `add` operation is best-case $\Omega(n)$ and worst-case $O(kn)$, with k the size of $\mathcal{P}$. The `partition` function can be answered in $O(e + m\ log(m))$, where e is the number of edges in the corresponding super-polygon s and is $O(m^2)$ and m is the number of vertices in s. The `prune` operation is (as in the case of the OVG) $O(card(\mathcal{E}_v))$ per update. Finally, the `add-edges` operation has the same running time properties as for the OVG.

Overall, the algorithm is dominated by the `add-edges` function. Hence, the same notion of clutter in the environment determining the running time of the algorithm apply to the HOVG as well. Inasmuch as the `add-edges` function should behavior linearly in many environments, the quadratic terms associated with the `add` and `partition` functions can also be significant.

4.4 Implementation

Our research has led to the production of a software package written in Java being used by the GRITS and BORG labs (and the larger Robotics and Intelligent Machines initiative at Georgia Tech). This package is deployed on Georgia Tech's team for the LAGR project (entering its third year in January), is being used for testing on the Magellan Pro robots in the GRITSLab, and is in use on Team Sting, the autonomous robot racing team at Georgia Tech, competing in the Urban Grand Challenge. As a part of this effort, we have contributed a number of components:

- implementation of OVG
- implementation of HOVG
- implementation of Dijkstra's Algorithm, A^*, D^*-lite
- implementation of Full and Reduced Visibility graph
- implementation of Lee's algorithm
- implementation of Fibonacci Heap, Binomial Heep
- implementation of Minimum spanning tree algorithms
- implementation of online polygonization from stereo data
- implementation of global mapping of 3D terrain

- implementation of interactive GUI
- implementation of HTML and EPS based display
- implementation of log-file playback
- implementation of XML import/export
- implementation of optimal planning over colored graphs
- integration with GT behavior control software, including player/stage/gazebo simulation
- installation on LAGR robot for competitive use
- installation on GRITSlab iRobot Magellan Pro robots
- installation on GT Urban Grand Challenge robot *Sting* (in development)

It would be difficult to describe in detail the functionality of this rather large software package. Two components, however, should be highlighted. First, the OVG and HOVG have been implemented and integrated with the GT software package in use for the LAGR project. This represents the entire *deliberative* layer for the overall system (see Figure 2.1). In addition to running online on the robot, this system also includes an offline playback of logged data, as well as simulation of the robot in the player/stage/gazebo environment.

The second highlighted component of our work is the graphical user interface. This interface always such interaction as adding, modifying, and removing polygons through, for example, mouse click-and-drag. In addition to a configurable display, the graph can be export and imported from XML and written to EPS and image formats.

Figure 4.6 contains the EPS output from a graph represented in our software package. This figure highlights one of the main differences between our approaches, the OVG and HOVG, and the traditional RVG. The RVG is densely filled with edges joining polygons. The OVG is very sparse, with only two edges per polygon. The HOVG is a kind of middle-ground between the two, using the RVG approach in regions of the graph, and joining

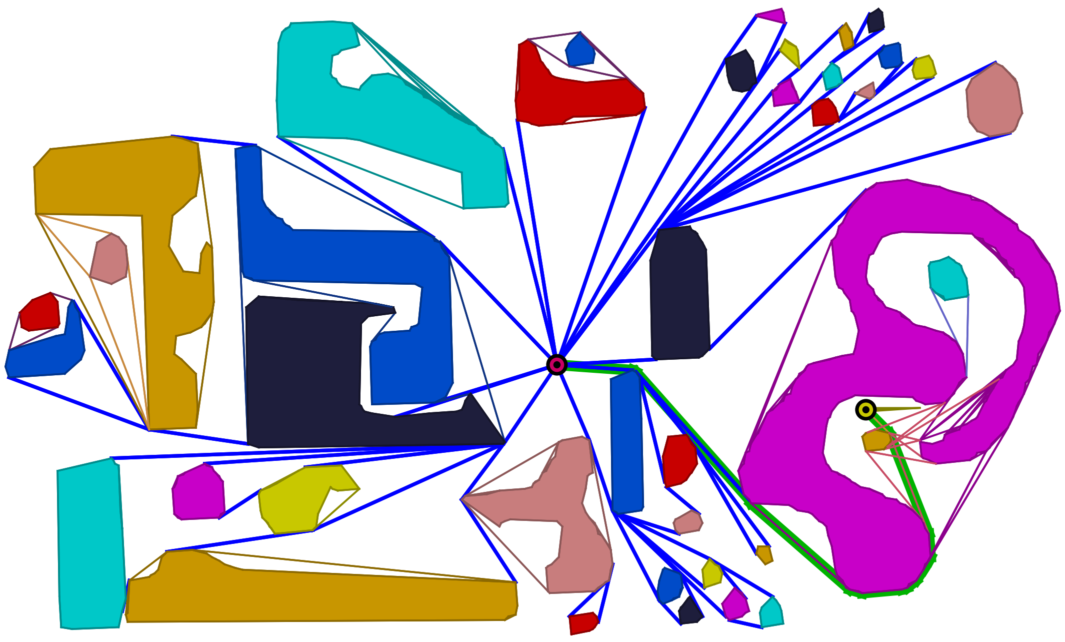


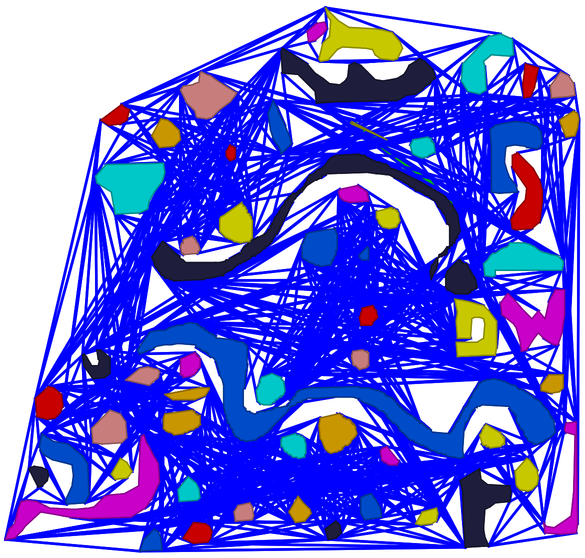
Figure 4.5: Encapsulated Postscript exported from our software package. The goal vertex is located in the center of the figure. Each polygon was generated through user interaction with the GUI.

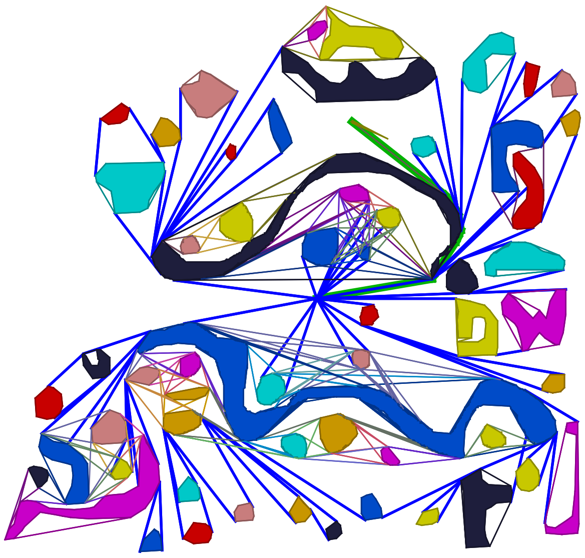
these regions using the OVG approach. Our sparse representation enables us to provide a dynamic algorithm that can run very quickly on the various robotic systems this work has been installed on.

4.5 Summary

The hierarchical oriented visibility graph is a powerful tool for path planning in dynamic environments. While still benefiting from the sparseness of the original oriented visibility graph, much of the suboptimality of that method is removed. This improvement of the roadmap comes at the cost of computing the intersection of convex hulls of simple polygonal obstacles.

We give the algorithms in detail with supporting complexity analysis of each step. This analysis leads to a worst-case cubic running time, but we show how the running time in practice may typically be closer to linear in the size of the graph. Our work is supported by a large software package implementing both the OVG and HOVG algorithms. This package is integrated with an interactive graphical suer interface as well as as the mapping and


(a) RVG


(b) HOVG

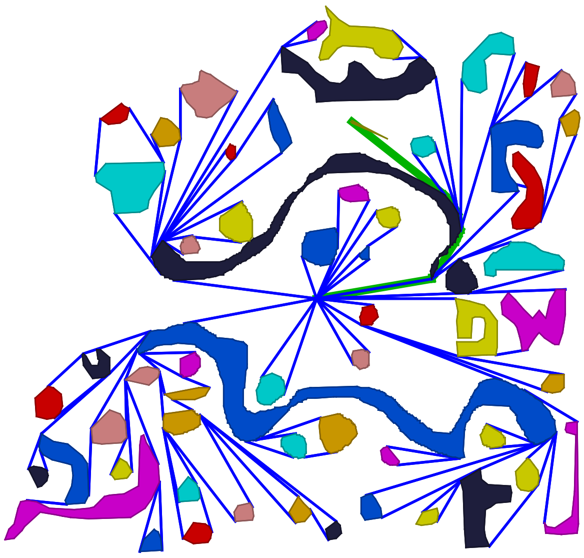

(c) OVG

Figure 4.6: Comparison of connectivity for visibility graphs. The reduced visibility graph is most dense. The hierarchical visibility graph has come regions of high edge density. The oriented visibility graph is the most sparse with exactly two edges per polygon.

planning modules on real-world robot hardware.

CHAPTER V

GLOBALLY OPTIMAL PATH PLANNING OVER WEIGHTED COLORED GRAPHS

In this chapter, we present our method from [77] for finding a globally optimal path through a colored graph. Optimal here means that, for a given path, the induced path coloring corresponds to an equivalent class. A total ordering is placed over these equivalent classes, and the edge weights are simply tie breakers within the classes. Optimality is achieved by mapping the class, or color, of each edge in combination with its weight to a real number. As a result, optimal paths can be computed using just the new weight function and standard edge relaxation methods (e.g. Dijkstra's Algorithm). The motivation for this research is the task of planning paths for mobile autonomous robots through outdoor environments with unknown and varied terrain.

5.1 Introduction

An emerging approach to handling the complexity associated with robot navigation tasks in unstructured environments is to decompose the control task into basic building-blocks [3,13, 21]. Each atomic building block corresponds to a particular control law (or mode), defined with respect to different tasks, sensory sources, or operating points [12]. The high-level control question then becomes that of concatenating these building-blocks together in order to meet the global objectives. The inherently unknown and unstructured nature of autonomous mobile robots' environments makes control particularly challenging, inasmuch as any precomputed optimal sequencing of the modes quickly becomes invalidated (suboptimal or even infeasible) as the environment or the robot's knowledge thereof changes.

The work in this chapter is based on an exploration of the different modes in the sense that by trying different mode strings, the robot builds up a high-level description of how the selection of particular modes in particular situations affects the performance of the system, similar to the idea presented in [37]. We achieve this through so-called Visual Feature Graphs, where each edge in the graph corresponds to a particular control law, and each vertex corresponds to a distinctive feature or place. Such graphs thus describe how the application of a certain control law takes the robot from one distinctive feature to the next. As an example, consider Figure 5.1. This way of structuring information about the environment is especially appealing in outdoor robotics applications, where the (distinctive) feature density can be expected to be fairly low.

Once such a Visual Feature Graph has been produced, one can plan optimal paths over them, which is the topic under investigation in this chapter. However, since some distinctive places correspond to areas where the robot is likely to get stuck (such as mud, brushy vegetation or quicksand), these places (or vertices) should be avoided. Other vertices may define places where the robot could possibly traverse successfully but which should nonetheless be avoided if more traversable paths are available. We encode these initial observations in a directed, colored graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W_{\mathcal{E}}, \mathcal{C}_{\mathcal{V}}, \mathcal{C}_{\mathcal{E}})$ where

- $\mathcal{V}$ is the set of vertices (distinctive places/features).
- $\mathcal{E} \subset \mathcal{V} \times \mathcal{V}$ is a set of ordered pairs of vertices.
- $W_{\mathcal{E}} : \mathcal{E} \to \mathbb{R}^+$ is a cost associated with each edge. This cost can for instance be interpreted as distance travelled or how long it would take to reach the edge's target vertex.
- $\mathcal{C}_{\mathcal{V}} : \mathcal{V} \to \mathcal{K}$ designates a class (i.e. color) $\mathcal{C}_{\mathcal{V}}(v)$ to each vertex in the graph. The set of classes $\mathcal{K} = \{1, \ldots, K\}$ corresponds to the different levels of traversability (or some other encoding of how suitable that distinctive place is for navigation), as discussed above.
- $\mathcal{C}_{\mathcal{E}} : \mathcal{E} \to \mathcal{K}$ designates a class $\mathcal{C}_{\mathcal{E}}(e)$ for each edge in the graph. In fact, we will let $\mathcal{C}_{\mathcal{E}}(e)$ be given by $\mathcal{C}_{\mathcal{V}}(v)$ where v is the target vertex of edge e.

The planning problem thus becomes that of finding the best path over a set of vertices, from the current vertex to the goal vertex, where "best" is loosely interpreted as minimizing the total edge-cost while avoiding vertices belonging to "bad" classes.

Some comments about the notation used below should be made. We assume that we are given $\mathcal{C}_{\mathcal{V}}$, which assigns a class to a vertex, and the class corresponds to the traversability of the terrain associated with that distinctive place. Hence, our graph is a colored graph (following the notation of [19]), but because it is possibly *improperly colored*, and a vertex's color is dictated by terrain quality rather than, for example, its connectivity, we prefer the term "class" over "color".

We moreover define $\mathcal{C}_{\mathcal{E}}$, the class of an edge, to be the class of the target vertex of the edge. The reason for this is that it is natural to assign a class to an edge matching where the edge terminates and the terrain that must be traversed to go there. (Note that $\mathcal{C}_{\mathcal{E}}$ could instead be defined based on the source vertex of an edge with little effect for the purposes of this work. It is important, however, that the definition be consistent.)

To make more concrete what we mean by "best path" and "bad classes", let us first establish the basic goal behind this research. We want to be able to use existing solutions to the `find-path` problem (e.g. Dijkstra's Algorithm), where, when given a colored graph $\mathcal{G}$ (as defined above), we are asked to plan a path over the edges between two specified vertices (for example, see [46]). Our goal here is to find a weight mapping $U_{\mathcal{E}} : \mathcal{E} \mapsto \mathbb{R}^{+}$, whereby global properties of the graph (including its constituent classes) are incorporated into $U_{\mathcal{E}}$, with the result that the simpler new weighted graph $(\mathcal{V}, \mathcal{E}, U_{\mathcal{E}})$ can be planned over without having to involve semantic symbols, i.e. the vertex classes.

This chapter presents a method for determining $U_{\mathcal{E}}$ and an optimal path over the new graph that is in fact optimal also over the original colored graph. Here, "best" does not correspond to a simple minimum edge-cost path, but rather, as already pointed out, a path that is optimal within the set of paths of the lowest "path class". The next section causally motivates the problem we solve in this work, and Section 5.3 presents a formal problem statement. The remainder of the chapter is organized as follows: In Section 5.4, we present our solution, i.e. the derivation of an appropriate the mapping $U_{\mathcal{E}}$. Sections 5.5 and 5.6 provide theorems and proofs, and Section 5.8 finalizes the discussion with conclusions.

5.2 Motivation

In this section, we present a situation where we could naturally define distinctive terrain features. This example should be thought of as simply a motivating example, rather than a realistic, full-scale robotics application. Figure 5.1 depicts an outdoor environment populated with six different terrain types (listed in decreasing traversability): roadway, pathway, field, forest, mud, and marsh. Marked in the figure are *start* and *goal* vertices. What is the best route over this environment?

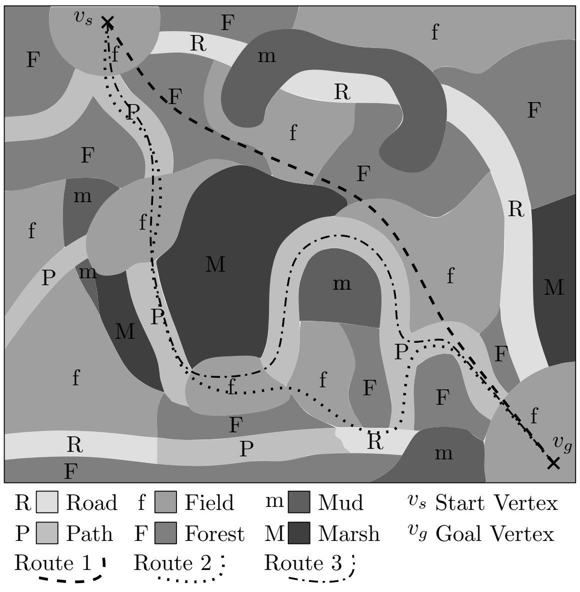


Figure 5.1: An example environment with 6 terrain classes; roads are the most easily traversed, followed by paths, fields, forests, mud, and marsh. Three feasible paths between the start and goal vertices are shown.

What we know from experience is that wheeled robots drive efficiently over roadways and pathways, and reasonably well in fields, but have trouble in forests, and are practically stuck in mud or marsh. Three paths are overlaid on the figure, showing potential routes to the goal. Route 1 is the shortest, but passes through a forest region (for specifics, see Table 1). Route 2 is longer but avoids the forest regions, passing at worst through a field. Route 3 is similar to Route 2, but is in the field regions less. We intuitively come to the conclusion that Route 3 is the best of the three.

Consider Figure 5.2, which is a graphical representation of Figure 5.1. Routes 1-3 are marked on the graph. We will use the developments in the following sections to show that

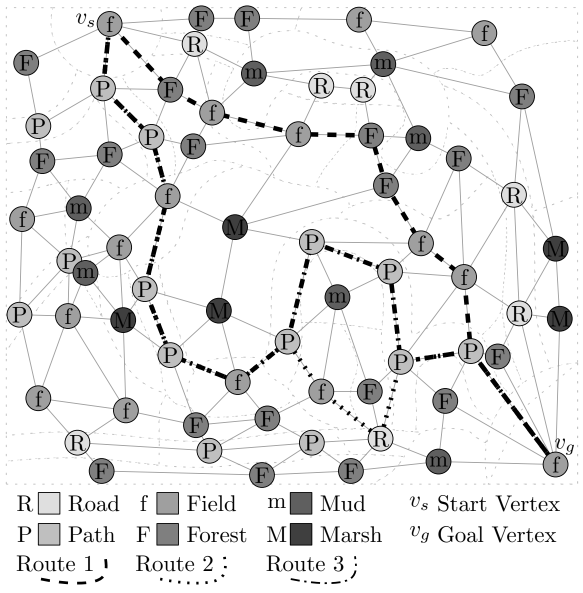


Figure 5.2: A combinatorial representation $\mathcal{G} = (\mathcal{V}, \mathcal{E}, W_{\mathcal{E}}, \mathcal{C}_{\mathcal{V}}, \mathcal{C}_{\mathcal{E}})$ of the environment from Figure 5.1. The discs represent vertices, with labels identifying the class of the vertex. Edges are depicted as undirected for the sake of clarity. Note that Paths 2 and 3 partially overlap.

Table 1: The number of edges per class for Paths 1-3 from Figure 5.2.

Route Number	Road Edges	Path Edges	Field Edges	Forest Edges	Mud Edges	Marsh Edges
1	0	1	5	3	0	0
2	1	7	4	0	0	0
3	0	9	3	0	0	0

Note how Path 1 has edges in the Forest class, while Paths 2 & 3 do not. Also, note how Path 3 has fewer Field edges than Path 2.

we can use standard edge relaxation methods on a simple graph $(\mathcal{V}, \mathcal{E}, U_{\mathcal{E}})$, and in so doing, identify Route 3 as the best path among Routes 1-3.

It is our intention within this work that the difference amongst classes should represent something that could not be reflected as a well-defined or well-posed transform on edge cost. In fact, we will impose a total ordering on the path classes and simply use the edge weights to break intra-class ties. A good example might be to define class based on what's necessary for the robot's safe operation. In regions where the robot may totally fail its mission, such situations (may) need to be completely avoided if at all possible – if any alternative route is available. This work is targeted at tractable path planning for these types of scenarios.

Alternatively, consider another example of where this work is applicable. The problem is to plan a path where vertices are classified according to the nature of locomotion employed. That is, a vertex may represent an airport, a bus stop, a taxi stand, a ferry, or a canoe put-in. In this scenario, taking a plane may be for the user unacceptable, regardless the additional cost incurred by travelling via any other route. The number of take-offs and landings may be, for example, most unacceptable. Given this kind of classification of vertices, the approach we describe in this chapter provides an optimal solution.

Our task is to solve an explicitly global objective for which a tractable local solution is not known. That is, a modification of edge weights based on local information is insufficient for solving our global problem. Moreover, we consider our knowledge of path class to be deterministic. Hence, we are motivated not towards the learning problem of identifying the vertex classification mapping, but towards finding an optimal path through $\mathcal{G}$ given C_V.

5.3 Problem Definition

Define an edge $e_{ij} \in \mathcal{E}$ as the ordered pair (v_i, v_j) with $v_i,\ v_j \in \mathcal{V}$; the edge passes in the direction from $v_i \longmapsto v_j$. Let the path p be a string of edges over G which connects some v_0 to v_F, the first and last vertices of the string. The edges of p are ordered such that the n-th edge passes from vertex v_n to v_{n+1}, with $n \in \{1, \ldots, length(p)\}$ and $length(p)$ is the length of p.

Partition the set of vertices $\mathcal{V}$into subsets based on class, such that

$$\mathcal{V} = \bigcup_{k=1}^{K} \mathcal{V}_k, \tag{19}$$

where

$$\mathcal{V}_k = \Big\{ v \in \mathcal{V} \mid \mathcal{C}_{\mathcal{V}}(v) = k \Big\} \tag{20}$$

and where K is the number of classes. Similarly, partition the set of edges $\mathcal{E}$, such that

$$\mathcal{E} = \bigcup_{k=1}^{K} \mathcal{E}_k, \tag{21}$$

where

$$\mathcal{E}_k = \Big\{ e_{ij} = (v_i, v_j) \in \mathcal{E} \;\Big|\; v_j \in \mathcal{V}_k \Big\}. \tag{22}$$

The set of all acyclic paths connecting v_s to v_g (the start and goal vertices) is denoted by Π. (We could more precisely call this $\Pi(v_s, v_g)$, but omit the arguments for the sake of clarity.) Let Π_k be the set of paths where the highest vertex class included in each path within Π_k is exactly k. In other words,

$$\Pi_k = \Big\{ p \in \Pi \;\Big|\; N(p,k) > 0 \;\&\; N(p,l) = 0,\ \forall l > k \Big\}, \tag{23}$$

with

$$N(p,k) = \text{card}(\{e \in p \;\big|\; \mathcal{C}_{\mathcal{E}}(e) = k\}) \tag{24}$$

being the number of edges of the path p that are of class k. (Here, card$(\dot{})$ denotes cardinality.) Furthermore, let

$$\Pi_k^i = \Big\{ p \in \Pi_k \;\Big|\; N(p,k) = i \Big\} \tag{25}$$

be the set of k-class paths with exactly i edges of class k.

Now, when asked to plan a path between v_s and v_g, we define the (not necessarily unique) optimal path p^* as one satisfying the following criteria:

1. Let

   $$\kappa = \min_{k \in \{1,\dots,K\}} (\Pi_k \neq \emptyset), \tag{26}$$

   and hence Π_κ is the set of all paths with the minimum maximum-vertex-class in Π. The optimal path p^* is in Π_κ.

2. Let

   $$\mathcal{P}_\kappa = \arg\min_{p \in \Pi_\kappa} N(p, \kappa). \tag{27}$$

   So, $\mathcal{P}_\kappa$ is the set of paths in Π_κ with the fewest number of κ-class vertices. The optimal path p^* is in $\mathcal{P}_\kappa$.

3. Let

   $$\mathcal{P}_{\kappa-1} = \arg\min_{p \in \mathcal{P}_\kappa} N(p, \kappa - 1). \tag{28}$$

   $\mathcal{P}_{\kappa-1}$ is the set of paths in $\mathcal{P}_\kappa$ with the fewest number of $(\kappa-1)$-class vertices.

4. Similarly,
$$\mathcal{P}_i = \underset{p \in \mathcal{P}_{i+1}}{\arg\min}\, N(p, i), \tag{29}$$
where $i = \{1, \ldots, \kappa - 1\}$. Path p^* is in all $\mathcal{P}_i$.

5. Finally, if $card(\mathcal{P}_1) > 1$, we wish that
$$p^* = \min_{p \in \mathcal{P}_1} \big(\sum_{e \in p} W_{\mathcal{E}}(e) \big). \tag{30}$$

Consequently, we have a recursive problem definition, with the optimal path belonging to recursively dependant sets of paths, based on edge classes. What we desire is a formulation of a new edge-weight function $U_{\mathcal{E}}$ that encodes this semantic information about the edge classes and incorporates the original weight function $W_{\mathcal{E}}$. In finding it, we will obtain a p^* satisfying Criteria 1-5, without having to worry about vertex classes. In fact, one can think of this construction as imposing a total order on the path classes, with $\Pi_k^i < \Pi_l^j$ for all $l > k$, and $\Pi_k^i < \Pi_k^j$ for all $j > i$, and so on.

Criteria 1-5 take into account all paths in the graph between the start and goal vertices. Hence, a selection of $U_{\mathcal{E}}$ finding p^* with respect to these specifications accounts for *all* such paths, and consequently, accounts for global properties of the graph's vertices and each vertex's class. Note that some modification of $W_{\mathcal{E}}(e)$ based only on the immediate neighbors of $e \in \mathcal{E}$ would be insufficient to guarantee the solution of p^* via an algorithm like Dijkstra's Algorithm. That is, locally modifying $W_{\mathcal{E}}$ does not incorporate the (global) information needed to satisfy criteria 1-5.

Now, we break the task of finding p^* into three subproblems:

Subproblem 1 *Find $U_{\mathcal{E}} : \mathcal{E} \mapsto \mathbb{R}^+$ such that, corresponding to Criteria 1, the cost of paths belonging to class k (i.e. $p \in \Pi_k$) should always be greater than the cost of paths belonging to class $k-1$. By the cost of a path p, we mean*
$$c(p) = \sum_{e \in p} U_{\mathcal{E}}(e). \tag{31}$$

Subproblem 2 *Find $U_{\mathcal{E}} : \mathcal{E} \mapsto \mathbb{R}^+$ such that, corresponding to Criteria 4, the cost should be higher for the path which has more edges in class k than other paths which have the same number of edges for all classes greater than k.*

Subproblem 3 *Find $U_{\mathcal{E}} : \mathcal{E} \mapsto \mathbb{R}^+$ such that, corresponding to Criteria 5, for paths that have the same number of edges in all classes, the optimal path minimizes $\sum_{e \in p} W_{\mathcal{E}}(e)$. In other words, paths that are equivalent according to class are distinguished based on the original edge weighting.*

This leads us to the main problem under consideration in this chapter, namely:

Problem 1 *Find $U_{\mathcal{E}} : \mathcal{E} \mapsto \mathbb{R}^+$ such that the optimal path p^* over the weighted graph $(\mathcal{V}, \mathcal{E}, U_{\mathcal{E}})$ is also optimal over the original colored graph according to the optimality Criteria 1-5.*

The overall implication of these subproblems is three-fold:

1. Subproblem 1 implies that the passage of a path through a k-class vertex makes that path less desirable than every path that visits vertices of class strictly less than k, without regard to how long or costly such paths may be or how many vertices they include; vertex classification bears strong meaning.

2. Subproblem 2 implies that the passage between adjacent vertices of the same class has more meaning than simply the weight of the edge joining them. The number of hops in a vertex class along a path has more impact than the length of the path.

3. The last subproblem implies that while vertex classification is the primary factor in determining an optimal path, the regular edge cost $W_{\mathcal{E}}$ is still required where vertex classification of two paths is identical.

5.4 The New Edge-weight Function, $U_\mathcal{E}$

Define the modified weight function

$$U_\mathcal{E}(e) = W_\mathcal{E}(e) + \sigma_i + \gamma_i, \tag{32}$$

with

$$\sigma_i = \sum_{e \in \mathcal{E}_i} W_\mathcal{E}(e), \tag{33}$$

$$\gamma_i = (1 + n_{i-1})(\sigma_{i-1} + \gamma_{i-1}), \tag{34}$$

and

$$\gamma_1 = 0, \tag{35}$$

where $n_i = card(\mathcal{E}_i)$ denotes the number of edges in

E_i, Finally, let C_{Π_k} be

$$C_{\Pi_k} = \Big\{ c(p) \;\Big|\; p \in \Pi_k \Big\}, \tag{36}$$

the set of costs for the paths in Π_k, where, as before,

$$c(p) = \sum_{e \in p} U_\mathcal{E}(e). \tag{37}$$

We shall demonstrate in the following sections that this recursive definition of γ_i is just what we need to solve Problem 1.

5.5 Lemmas

Two lemmas will prove useful for the next section.

Lemma 5.5.1 The maximum cost of Π_k is less than γ_{k+1}.

Proof:

For an acyclic path which never visits any vertex v with $\mathcal{C}_\mathcal{V}(v) > k$, the highest possible cost of the path corresponds to one that includes every edge $e \in \mathcal{E}$ where $\mathcal{C}_\mathcal{E}(e) \leq k$ (though it may be that no such path exists). That is,

$$\begin{array}{rl}
\multicolumn{2}{l}{max(C_{\Pi_k})} \\
\leq & \displaystyle\sum_{i=1}^{k}\sum_{e\in\mathcal{E}_i} U_{\mathcal{E}}(e) \\
= & \displaystyle\sum_{e\in\mathcal{E}_1} U_{\mathcal{E}}(e) + \sum_{e\in\mathcal{E}_2} U_{\mathcal{E}}(e) + \ldots + \sum_{e\in\mathcal{E}_k} U_{\mathcal{E}}(e) \\
= & \displaystyle(\sum_{e\in\mathcal{E}_1} W_{\mathcal{E}}(e) + n_1(\sigma_1+\gamma_1)) + \\
 & \displaystyle+(\sum_{e\in\mathcal{E}_2} W_{\mathcal{E}}(e) + n_2(\sigma_2+\gamma_2)) + \\
 & \displaystyle+\ldots+(\sum_{e\in\mathcal{E}_k} W_{\mathcal{E}}(e) + n_k(\sigma_k+\gamma_k)) \\
= & (\sigma_1 + n_1(\sigma_1+\gamma_1)) + (\sigma_2 + n_2(\sigma_2+\gamma_2)) + \\
 & +\ldots+(\sigma_k + n_k(\sigma_k+\gamma_k)) \\
= & (1+n_1)(\sigma_1+\gamma_1) + (\sigma_2 + n_2(\sigma_2+\gamma_2)) + \\
 & +\ldots+(\sigma_k + n_k(\sigma_k+\gamma_k)) \\
= & (\gamma_2) + (\sigma_2 + n_2(\sigma_2+\gamma_2)) + \\
 & +\ldots+(\sigma_k + n_k(\sigma_k+\gamma_k)) \\
= & (\gamma_3) + (\sigma_3 + n_3(\sigma_3+\gamma_3)) + \\
 & +\ldots+(\sigma_k + n_k(\sigma_k+\gamma_k)) \\
\ldots & \\
= & (\gamma_{k-1}) + (\sigma_{k-1} + n_{k-1}(\sigma_{k-1}+\gamma_{k-1})) + \\
 & +(\sigma_k + n_k(\sigma_k+\gamma_k)) \\
= & (\gamma_k) + (\sigma_k + n_k(\sigma_k+\gamma_k)) \\
= & \gamma_{k+1}
\end{array}$$

■

Lemma 5.5.2 The minimum cost of Π_k is greater than $\sigma_k + \gamma_k$.

Proof:

Conceptually, the path in Π_k with minimum cost visits the fewest vertices possible and the weight of the edges on this path are minimal. So such a path would, in principle, visit exactly one k-class edge and a minimum of other edges. There are two possible cases:

Case 1: If $\mathcal{C}_{\mathcal{V}}(v_g) = k$, (where v_g is the goal vertex) then the minimal path would be just $v_s \rightarrow v_g$ (if such a path existed). So,

$$\begin{aligned}
&min(C_{\Pi_k}) \\
&\quad \geq \; U_{\mathcal{E}}(e_{sg}) \\
&\quad = \; W_{\mathcal{E}}(e_{sg}) + \sigma_k + \gamma_k \\
&\quad > \; \sigma_k + \gamma_k
\end{aligned}$$

Case 2: If $\mathcal{C}_{\mathcal{V}}(v_g) \neq k$, then the minimal path would be $v_s \rightarrow v_j \rightarrow v_g$, where $v_j \in \mathcal{V}_k$. So,

$$\begin{aligned}
&min(C_{\Pi_k}) \\
&\quad \geq \; U_{\mathcal{E}}(e_{sj}) + U_{\mathcal{E}}(e_{jg}) \\
&\quad = \; (W_{\mathcal{E}}(e_{sj}) + \sigma_k + \gamma_k) + \\
&\qquad\;\; +(W_{\mathcal{E}}(e_{jg}) + \sigma_{\mathcal{C}_{\mathcal{V}}(v_g)} + \gamma_{\mathcal{C}_{\mathcal{V}}(v_g)}) \\
&\quad > \; \sigma_k + \gamma_k
\end{aligned}$$

In both cases, we find

$$min(C_{\Pi_k}) > \sigma_k + \gamma_k. \tag{38}$$

■

5.6 Theorems

Theorem 5.6.1

$$max(C_{\Pi_{k-1}}) < min(C_{\Pi_k}). \tag{39}$$

Proof:

Simply taking Lemmas 5.5.1 and 5.5.2 together, we find

$$max(C_{\Pi_{k-1}}) \leq \gamma_k < \sigma_k + \gamma_k < min(C_{\Pi_k}) \tag{40}$$

■

Theorem 5.6.2 *The cost of traversing a k-class path with j k-class vertices is always less than a k-class path with (j + 1) k-class vertices, where k is the highest class on the two paths. In other words,*

$$max(C_{\Pi_k^j}) < min(C_{\Pi_k^{j+1}}).$$

Proof: First, we assume $card(\mathcal{E}_k) > j$.

$$\begin{aligned}
& max(C_{\Pi_k^j}) \\
\leq \; & \sum_{i=1}^{k-1} \sum_{e \in \mathcal{E}_i} U_{\mathcal{E}}(e) + \text{worst } j\text{-length path in } \mathcal{E}_k \\
< \; & \sum_{i=1}^{k-1} \sum_{e \in \mathcal{E}_i} U_{\mathcal{E}}(e) + j(\sigma_k + \gamma_k) + \sum_{e \in \mathcal{E}_k} W_{\mathcal{E}}(e) \\
= \; & \gamma_k + j(\sigma_k + \gamma_k) + \sigma_k \\
= \; & (j+1)(\sigma_k + \gamma_k)
\end{aligned}$$

$$\begin{aligned}
& min(C_{\Pi_k^{j+1}}) \\
\geq \; & \text{best } (j+1)\text{-length path in } \mathcal{E}_k \\
> \; & (j+1)(\sigma_k + \gamma_k)
\end{aligned}$$

Hence,

$$max(C_{\Pi_k^j}) < min(C_{\Pi_k^{j+1}}). \tag{41}$$

This proves the theorem.

■

Theorem 5.6.1 establishes that $U_{\mathcal{E}}$ solves Subproblem 1. In other words, a path in Π_k always has lower cost than a path in Π_{k+1}. Theorem 5.6.2 establishes that $U_{\mathcal{E}}$ solves Subproblem 2. $U_{\mathcal{E}}$ also solves Subproblem 3, by noting that for a path p

$$\begin{aligned} c(p) &= \sum_{e \in p} U_{\mathcal{E}}(e) && (42) \\ &= \sum_{e \in p} (W_{\mathcal{E}}(e) + \sigma_{\mathcal{C}_{\mathcal{E}}(e)} + \gamma_{\mathcal{C}_{\mathcal{E}}(e)}), && (43) \end{aligned}$$

and that for two paths p_1 and p_2 with the same numbers of edges over all the classes, the difference between the cost of the two paths reduces to the difference of the sum of their edge weights. That is,

$$c(p_1) - c(p_2) = \sum_{e \in p_1} W_{\mathcal{E}}(e) - \sum_{e \in p_2} W_{\mathcal{E}}(e). \tag{44}$$

By solving each of the three subproblems, $U_{\mathcal{E}}$ solves Problem 1.

5.7 Implementation

Figure 5.3 shows a screenshot taken from the GRITSlab [1] graph planning software library corresponding to the graph depicted in Figure 5.2. Here, edge weights have been assigned according to $U_{\mathcal{E}}$, and Dijkstra's algorithm has been run over the resulting graph. The dark line pointing out of each node indicates the next node to be taken on the optimal path to the goal (which is located in the lower right-hand corner).

The thick line starting from the robot (upper left-hand corner) and ending at the goal correctly identifies Route 3 as the optimal path over this weighted colored graph. The code needed to implement $U_{\mathcal{E}}$ is straight-forward and follows directly from equations 32-35.

5.8 Summary

We are able to find optimal paths over a directed colored graph $G = (\mathcal{V}, \mathcal{E}, W_{\mathcal{E}}, \mathcal{C}_{\mathcal{V}}, \mathcal{C}_{\mathcal{E}})$, by transforming it into $(\mathcal{V}, \mathcal{E}, U_{\mathcal{E}})$, and applying standard edge relaxation algorithms. The resulting optimal path over $(\mathcal{V}, \mathcal{E}, U_{\mathcal{E}})$ is also optimal over G in the sense that it matches our criteria specified in Criteria III:1-5. This transformation has been motivated by the need to navigate a mobile robot through an outdoor environment populated with regions of relatively distinct and identifiable terrain.

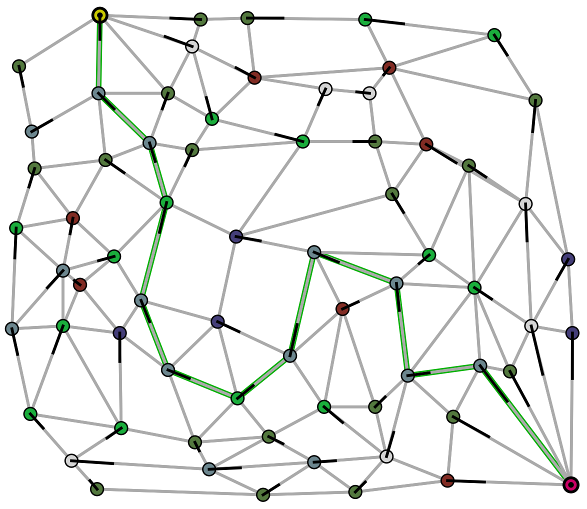


Figure 5.3: Screenshot from the GRITSlab [1] graph planning software library of the example from Figure 5.2.

CHAPTER VI

CONCLUSIONS

Path planning is, informally, the process of deciding how to get from here to there when you do not know what will be along the way. It is a fundamental problem of robotics. This thesis addresses this problem, narrowly at the level of dynamic path planning in the plane, as well as at a higher-level, over colored graphs, and most generally within the context of the overall robot control architecture. Our motivation throughout has been to provide solutions to the path planning problem with actual application to field robots, where notions of theoretical performance are as important as effectiveness in non-trivial outdoor environments.

Simultaneous control and mapping

We introduce a new feedback mechanism upward through the canonical two-layer control architecture. This enables information derived within the low-level controllers to be passed to the global planner, and incorporated into a global map. This approach is presented as a part of the control system used to drive Georgia Tech's robot for the LAGR project, and experiments highlight the its utility.

The oriented visibility graph

Dealing with unknown unstructured environments is both a challenging problem and key feature of any robust field robot. Our work in this area has been implemented on large-scale systems in exactly these kinds of environments. Our representation of the environment, through the oriented visibility graph and hierarchical oriented visibility graph, is sparse, which leads to our ability to implement these approaches on actual systems where the perception of the environment is constantly changing.

The hierarchical oriented visibility graph

The hierarchical extension of the oriented visibility graph is an interleaving of the reduced visibility graph in small regions of the graph with these regions are interconnected by a oriented visibility graph. This directly addresses the loss of guaranteed optimality for the OVG, while providing the possibility of efficient dynamic updates.

More than just a black-box planning module that has been installed in competitive robot systems, our work is supported by a mature software package. This package contains implementations of our approaches and classic approaches such as A^*, D^*, and the reduced visibility graph. It is a part of an interactive graphical user interface and integrated with both the control/behavior software developed in Georgia Tech's BORG lab, as well as the simulation environment player/stage/gazebo. This implementation is a valuable feature of our work, and lends support to the strength of the methods we have presented.

Path planning over colored graphs

Finally, our work on planning over colored graphs has led to a globally optimal solution to finding paths where the vertices in the graph have been assigned a class. This class imposes a total ordering on the vertices (and edges), and we derive an edge-weight function that incorporates this class information along with the standard edge weight. The solution to the path planning problem is then computed using Dijkstra's Algorithm.

REFERENCES

[1] "Georgia robotics and intelligent systems laboratory." http://gritslab.ece.gatech.edu.

[2] "Learning applied to ground robots (LAGR) proposer information pamphlet," May 2004. BAA # 04-25.

[3] Arkin, R., *Behavior-Based Robotics.* The MIT Press, 1998.

[4] Arkin, R. and Balch, T., "Aura: Principles and practice in review," *Journal of Experimental and Theoretical Artificial Intelligence*, vol. 9, 1997.

[5] Aronov, Guibas, Teichmann, and Zhang, "Visibility queries in simple polygons and applications," in *ISAAC: 9th International Symposium on Algorithms and Computation (formerly SIGAL International Symposium on Algorithms), Organized by Special Interest Group on Algorithms (SIGAL) of the Information Processing Society of Japan (IPSJ) and the Technical Group on Theoretical Foundation of Computing of the Institute of Electronics, Information and Communication Engineers (IEICE))*, 1998.

[6] Asano, T., Asano, T., Guibas, L. J., Hershberger, J., and Imai, H., "Visibility of disjoint polygons," *Algorithmica*, vol. 1, 1986.

[7] Balch, T. and Arkin, R., "Behavior-based formation control for multi-robot teams," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, pp. 926–939, 1998.

[8] Balch, T. R., *Behavioral Diversity in Learning Robot Teams.* PhD thesis, Georgia Institute of Technology, Atlanta, Georgia, USA, 1998.

[9] Bohlin, R. and Kavraki, L., "Path planning using lazy PRM," in *IEEE Int. Conf. on Robotics and Automation*, vol. 1, pp. 521–528, 2000.

[10] Brock, O. and Khatib, O., "High-speed navigation using the global dynamic window approach," in *Proc. Int. Conf. on Robotics and Automation*, 1999.

[11] Brock, O. and Khatib, O., "Elastic strips: A framework for motion generation in human environments," *International Journal of Robotics Research*, vol. 21, pp. 1031–1052, 2002.

[12] Brockett, R., "On the computer control of movement," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, vol. 1, pp. 534–40, 1988.

[13] Brooks, R., "A robust layered control system for a mobile robot," *IEEE Journal of Robotics and Automation*, vol. RA-2, pp. 14–23, Mar. 1986.

[14] Brooks, R. A., "Elephants don't play chess," *Robotics and Autonomous Systems*, vol. 6, pp. 3–15, June 1990.

[15] Canny, J., *The Complexity of Robot Motion Planning.* Cambridge, MA: MIT Press, 1987.

[16] Chen, Szczerba, and Uhran, "Planning conditional shortest paths through an unknown environment: A framed-quadtree approach," in *Proc. of the IEEE Int'l Conf. on Robotics and Automation*, May 1995.

[17] Cormen, T., Leiserson, C., Rivest, R., and Stein, C., *Introduction to Algorithms, Second Edition.* MIT Press and McGraw-Hill, 2001.

[18] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., *Computational Geometry: Algorithms and Applications, 2nd Ed.* Springer-Verlag, 2000.

[19] Diestel, R., *Graph Theory.* Springer-Verlag, third ed., 1997.

[20] Edelsbruner, H., Guibas, L., and Stolfi, J., "Optimal point location in a monotone subdivision," Tech. Rep. 2, DEC systems Research Center, 1984.

[21] Egerstedt, M., Johansson, K., Lygeros, J., and Sastry, S., "Behavior based robotics using regularized hybrid automata," in *Proc. IEEE Conf. on Decision and Control*, vol. 4, pp. 3400–5, 1999.

[22] Egerstedt, M., *Motion Planning and Control of Mobile Robots.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2000.

[23] Ferguson, D. personal communication, May 2006.

[24] Ferguson, D. and Stentz, A., "Multi-resolution field D*," in *Proceedings of the International Conference on Intelligent Autonomous Systems (IAS)*, March 2006.

[25] Ferguson, D. and Stentz, A., "Replanning with RRTs," in *IEEE International Conference on Robotics and Automation*, 2006.

[26] Fox, D., Burgard, W., and Thrun, S., "The dynamic window approach to collision avoidance.," *IEEE Robotics and Automation*, vol. 4, no. 1, 1997.

[27] Fox, D., Burgard, W., Thrun, S., and Cremers, A., "A hybrid collision avoidance method for mobile robots," in *Proc. of the IEEE International Conference on Robotics & Automation*, 1998.

[28] Gat, E., "Integrating planning and reacting in a heterogeneous asynchronous architecture for mobile robots," in *SIGART Bulletin*, vol. 2, pp. 70–74, 1991.

[29] Gat, E., "On three-layer architectures," 1997.

[30] Gat, E., "Three layered architectures," 1998.

[31] Ghosh, S. and Mount, D., "An output sensitive algorithm for computing visibility graphs," *SIAM Journal on Computing*, vol. 20, pp. 888–910, 1991.

[32] Gonzalez, R. and Woods, R., *Digital Image Processing.* Addison-Wesley, 1992.

[33] Hart, P., Nilsson, N., and Raphael, B., "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, pp. 100–107, 1968.

[34] Hefferman, P. and Mitchell, J. S. B., "An optimal algorithm for computing visibility in the plane," in *2nd Workshop WADS91*, (Ottawa), pp. 437–448, 1991.

[35] Hershberger, J. and Suri, S., "An optimal algorithm for Euclidean shortest paths in the plane," *SIAM Journal on Computing*, vol. 28, no. 6, pp. 2215–2256, 1999.

[36] Howard, A., Seraji, H., and Werger, B., "Global and regional path planners for integrated planning and navigation," *Journal of Robotics Systems*, vol. 12, 2005.

[37] Hristu-Varsakelis, D. and Andersson, S., "Directed graphs and motion description languages for robot navigation," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, vol. 3, pp. 2689–94, 2002.

[38] Hsu, D., Kavraki, L., Latombe, J., Motwani, R., and Sorkin, S., "On finding narrow passages with probabilistic roadmap planners," in *Int. Workshop on Algorithmic Foundations of Robotics*, 1998.

[39] Kavraki, L., Svestka, P., Latombe, J.-C., and Overmars, M., "Probabilistic roadmaps for path planning in high-dimensional configuration spaces," *IEEE Trans. on Robotics and Automation*, vol. 12, pp. 566–580, 1996.

[40] Kim, D., Sun, J., Oh, S. M., Rehg, J., and Bobick, A., "Traversability classification using unsupervised on-line visual learning for outdoor robot navigation," in *IEEE Int'l Conf. on Robotics and Automation*, 2006.

[41] Koenig, S. and Likhachev, M., "Fast replanning for navigation in unknown terrain," *Transactions on Robotics and Automation*, 2005.

[42] Koenig, S. and Likhachev, M., "Real-time adaptive a*," in *Int. Joint Conf. on Autonomous Agents and Multiagent Systems*, 2006.

[43] Konolige, K., Agrawal, M., Bolles, R. C., Cowan, C., Fischler, M., and Gerkey, B. P., "Outdoor mapping and navigation using stereo vision," in *Intl. Symp. on Experimental Robotics*, July 2006.

[44] Latombe, J., *Robot Motion Planning.* Kluwer Academic Publishers, 1991.

[45] LaValle, S., "Rapidly-exploring random trees: A new tool for path planning," 1998.

[46] LaValle, S., *Planning Algorithms.* Cambridge University Press, 2006. Also available at http://msl.cs.uiuc.edu/planning/.

[47] Lee, D., *Proximity and Reachability in the Plane.* PhD thesis, University of Illinois, Urbana Champagne, 1978.

[48] Likhachev, M., *Search-based Planning for Large Dynamic Environments.* PhD thesis, Carnegie Mellon University, 2005.

[49] Likhachev, M., Ferguson, D., Gordon, G., Stentz, A., , and Thrun, S., "Anytime dynamic a*: An anytime, replanning algorithm," in *Int. Conf. on Automated Planning and Scheduling*, 2005.

[50] Lozano-Perez, T., "A simple motion-planning algorithm for general robot manipulators," *IEEE Journal of Robotics and Automation*, vol. 3, 1987.

[51] Lozano-Prez, T. and Wesley, M. A., “An algorithm for planning collision-free paths among polyhedral obstacles,” *Communications of the ACM archive*, vol. 22, pp. 560–570, 1979.

[52] Lyons, D. and Hendriks, A., “Planning as incremental adaptation of a reactive system,” *Robotics and Autonomous Systems*, vol. 14, no. 4, pp. 255–228, 1995.

[53] Nilsson, N., “Mobile automation: An application of artificial intelligence techniques,” in *Proc. 1st Int. Joint Conf. Artificial Intelligence*, pp. 509–520, 1969.

[54] O’Rourke, J., Chien, C.-B., Olson, T., and Naddor, D., “A new linear algorithm for intersecting convex polygons,” *Comput. Graphics Image Process.*, vol. 19, pp. 384–391, 1982.

[55] ORourke, J. and Streinu, I., “The vertex-edge visibility graph of a polygon,” *Computational Geometry: Theory and Applications*, vol. 10, pp. 105–120, May 1998.

[56] Overmars, M., “A random approach to motion planning,” tech. rep., Utrecht University, Oct. 1992.

[57] Overmars, M. H. and Welzl, E., “New methods for computing visibility graphs,” in *Symposium on Computational Geometry*, pp. 164–171, 1988.

[58] Papadimitriou, C., “An algorithm for shortest-path motion in three dimensions,” *Inform. Process. Letters*, vol. 20, pp. 259–263, 1985.

[59] Pocchiola, M. and Vegter, G., “The visibility complex,” *Int. J. Comput. Geom. Appl.*, vol. 6, pp. 279–308, 1996.

[60] Quinlan, S. and Khatib, O., “Elastic bands: connecting planning and control,” in *IEEE Conf. on Robotics and Automation*, 1993.

[61] Ranganathan, A. and Koenig, S., “A reactive robot architecture with planning on demand,” in *Proc. of the IEEE Int’l Conf. on Intell. Robots and Systems*, 2003.

[62] Reif, J. and Storer, J., “A single-exponential upper bound for finding shortest paths in three dimensions,” *Journal of ACM*, vol. 41, pp. 1013–1019, 1994.

[63] Rimon, E. and Koditschek, D. E., “Exact robot navigation using artificial potential fields,” *IEEE Transactions on Robotics and Automation*, vol. 8, pp. 501–518, oct 1992.

[64] Riviere, S., “Dynamic visibility in polygonal scenes with the visibility complex,” in *Symposium on Computational Geometry*, pp. 421–423, 1997.

[65] Roos, T., *Dynamic Voronoi Diagrams.* PhD thesis, University of Wurzburg, 1991.

[66] Rosenblatt, J., “Damn: A distributed architecture for mobile navigation,” *Journal of Experimental and Theoretical Artificial Intell.*, vol. 9, no. 2, pp. 339–60, 1997.

[67] Rosenblatt, J., “Maximizing expected utility for optimal action selection under uncertainty,” *Autonomous Robots*, vol. 9, no. 1, pp. 17–25, 2000.

[68] Sharir, M., *Handbook of Discrete and Computational Geometry, 2nd Ed.*, ch. Algorithmic Motion Planning, pp. 1037–1064. New York: Chapman and Hall/CRC Press, 2004. J. E. Goodman and J. O’Rourke, editors.

[69] Stentz, A., "Optimal and efficient path planning for partially-known environments," in *Proc. IEEE Int. Conf. Robot. & Autom.*, pp. 3310–3317, 1994.

[70] Stentz, A., "The focussed D* algorithm for real-time replanning," in *Proc. of the Int'l Joint Conf. on Artificial Intelligence*, Aug. 1995.

[71] Sukthankar, R., Pomerleau, D., and Thorpe, C., "A distributed tactical reasoning framework for intelligent vehicles," in *Intelligent Systems and Manufacturing*, 1997.

[72] Sun, J., Mehta, T., Wooden, D., Powers, M., Regh, J., Balch, T., and Egerstedt, M., "Learning from examples in unstructured, outdoor environments," *Journal of Field Robotics*, 2006.

[73] Toussaint, G., "A simple linear algorithm for intersecting convex polygons," *The Visual Computer*, vol. 1, no. 4, pp. 118–123, 1985.

[74] Walter, W. G., *The Living Brain.* Gerald Duckworth and Co., Ltd, 1953.

[75] Williams, S., Newman, P., Rosenblatt, J., Dissanayake, G., and Durrant-Whyte, H., "Autonomous underwater navigation and control," *Robotica*, vol. 19, no. 5, pp. 481–496, 2001.

[76] Wooden, D., "A guide to vision-based mapping," *IEEE Robotics and Automation Magazine*, June 2006.

[77] Wooden, D. and Egerstedt, M., "On finding globally optimal paths through weighted colored graphs," in *IEEE Conference on Decision and Control*, (San Diego, CA), Dec. 2006.

[78] Wooden, D. and Egerstedt, M., "Oriented visibility graphs: Low-complexity planning in real-time environments," in *IEEE Confernce on Robotics and Automation*, June 2006.

[79] Wooden, D., Egerstedt, M., and Ghosh, B.K., "Quantized Principal Component Analysis with Applications to Low-Bandwidth Image Compression and Communication," in *ISCIE Symposium on Stochastic Systems Theory and Its Applications*, "Saitama, Japan," Nov. 2004.

[80] Wooden, D., Egerstedt, M., and Ghosh, B.K., "Quantized Principal Component Analysis with Applications to Low-Bandwidth Image Compression and Communication," *Int. Journal of Innovative Computing, Information and Control*, v. 1, n. 3, pp. 479-492, 2005.

[81] Wooden, D., Powers, M., MacKenzie, D., Balch, T., and Egerstedt, M., "SCAM: Layered hybrid control with feedback between layers," in *IEEE Int'l conf. on Robotics and Automation*, 2007. Submitted to IEEE Int'l conf. on Robotics and Automation.

[82] Yahja, A., Stentz, A., Singh, S., and Brumitt, B., "Framed-quadtree path planning for mobile robots operating in sparse environments," in *Proc. IEEE Int'l Conf. on Robotics and Automation*, pp. 650–655, 1998.

[83] Yang, Y. and Brock, O., “Elastic roadmaps: Globally task-consistent motion for autonomous mobile manipulation,” in *Robotics: Science and Systems*, 2006.

[84] Yershova, A., Jaillet, L., Simeon, T., and LaValle, S., “Dynamic-domain RRTs: Efficient exploration by controlling the sampling domain,” in *IEEE Int. Conf. Robotics and Automation*, 2005.