Computer System and Robotic Laboratory (CORO)

Electrical Engineering Department

Federal University of Minas Gerais (UFMG)

Av. Antônio Carlos, 6627, CEP 31270-901, Belo Horizonte, MG, Brasil

Phone: +55 3499-3470

New Strategies for Multi-Robot Coordination in Optimal Deployment Problems

Reza Javanmard Alitappeh

PhD thesis submitted to the Graduate Program in Electrical Engineering of the Federal University of Minas Gerais in fulfillment of the requirements for the degree of Doctor of Electrical Engineering.

Advisor: Prof. Dr. Luciano Cunha de Araújo Pimenta
Department of Electronic Engineering at UFMG

Co-advisor: Prof. Dr. Luiz Chaimowicz
Computer Science Department at UFMG

Belo Horizonte, MG, Brazil
April-2016

Dedication

I would like to dedicate this to my dear wife *Kossar*. Thank you very much for all of your support and encouragement while I wrote this document. It is also dedicated to my devoted *father and mother*, who taught me to be patient, and to have perseverance to accomplish the most challenging task of all my life.

Acknowledgment

First, I am merciful of God, for giving me strength to finish this work.

I would like to express my sincere gratitude to my advisor Prof. *Luciano Pimenta* who has been the ideal thesis supervisor. His sage advice, motivation, insightful criticisms, and patient encouragement aided the writing of this thesis in innumerable ways. Also my co-advisor Prof. *Luiz Chamowicz*, who guides me to find the right line in my PhD.

Besides my advisor and co-advisor, thanks to Prof. *Guilherme Pereira* who was always available to answer my inquiries and offered me devices and equipments of robotics lab (CORO). Moreover, many thanks for his help in development of the theory that allows us to deploy multi-robot system in topological map (Chapter 6). Also one of his undergrad student *Arthur Araujo* who developed the robot control package that was used in our simulation and real robot experiment in Chapter 6.

My sincere thanks also goes to Prof. *Frederico Guimaraes* for all his supports. As a kind friend, he made me forget loneliness in my living time in Brazil.

Many thanks to the committee members in my defense: Prof. *Alexandre Ramos*, again Prof. *Guilherme Pereira*, Prof. *Bruno Adorno*, and Prof. *Vinicius Mariano*.

We gratefully acknowledge support from *CNPq*, *CAPES*, and *FAPEMIG*.

Last but not least, I thank my colleagues in CORO, VerLab and MACRO and friends in Brazil for all their support.

Contents

Contents vii

Abstract xi

List of Figures xiii

List of Tables xv

List of Algorithms xvii

List of Symbols xix

List of Abbreviations xxiii

1 Introduction 1

1.1 Motivation and Contributions 2

1.2 Document Organization 5

2 Background 7

2.1 Graph (Dijkstra Algorithm) 7

2.2 Generalized Voronoi Diagram (GVD) 8

2.3 Voronoi Tessellation 10

2.4 Locational Optimization Based Deployment 11

2.4.1 Problem Definition 11

2.4.2 p-Median Problem 15

2.5 GPU and CUDA 16

2.5.1 Software Model: 16
2.5.2 Hardware Model: 21
2.5.3 New CUDA Hardware Model 22

3 Related Work **23**
3.1 Multi-Robot Deployment 23
3.1.1 Multi-Robot Deployment in Continuous Space 25
3.1.2 Multi-Robot Deployment in Discrete Space 28
3.2 GPU Based Graph Search and CVT 29

4 Robot safe deployment **33**
4.1 Introduction 33
4.2 Safe Deployment 33
4.2.1 New Metric 34
4.2.2 Collision Avoidance Between Robots 36
4.3 Proposed Solution 38
4.3.1 Discrete Approximation 38
4.3.2 Distributed Algorithm 41
4.4 Results 46
4.4.1 Simple Map 46
4.4.2 Office-like Environment 48
4.5 Conclusion 50

5 New Multi-Robot Deployment Algorithm **53**
5.1 Introduction 53
5.2 Proposed Method 53
5.2.1 Discrete Approximation and Graph Representation 54
5.2.2 Deployment Function 57
5.2.3 Robot Node Neighbors 58
5.2.4 Distributed Algorithm 60
5.3 Discrete Implementation on GPU 67
5.3.1 GPU Based Multi-Source Dijkstra 67

5.3.2 Parallel MSSP algorithm 70
5.3.3 CPU VS GPU 76
5.4 Implementation Result 77
5.4.1 Simple Map 77
5.4.2 Office-like Environment 79
5.4.3 Real Robot Experiment 83
5.5 Conclusion 86

6 Deployment in Topological Map **87**

6.1 Introduction 87
6.2 Problem Statement 88
6.3 Methodology 89
6.3.1 Topological Map Representation 90
6.3.2 Multi-Robot Deployment 94
6.3.3 Analysis 97
6.3.4 Robot Control 102
6.4 Implementation Results 105
6.4.1 Simulation Results 105
6.4.2 Real Robot Experiments 109
6.5 Conclusion 111

7 Conclusion and Future Directions **113**

Bibliography **117**

Abstract

In recent years, multi-robot systems have played an increasing role in many real world applications thanks to its flexibility, robustness, and reduced cost when compared to single-robot solutions. One of the important challenges in this field is to efficiently distribute and coordinate a team of robots over the environment, so that desired tasks can be properly performed by this team. In this work we extend previous methods on multi-robot deployment in order to improve safety, convergence, applicability and computational time. In the first extension, we propose a new strategy based on the locational optimization framework. Our approach models the optimal deployment problem as a constrained optimization problem with inequality and equality constraints. In order to consider the generation of safe paths during the deployment or in future excursions through the environment, this optimization model is built by incorporating: the classical Generalized Voronoi Diagram (GVD); and a new metric to compute distance in the environment. GVD is commonly used as a safe roadmap in the context of path planning, and the new metric induces a new Voronoi partition of the environment. Furthermore, inspired by the classical Dijkstra algorithm, we present a novel efficient distributed algorithm to compute solutions in complicated environments. A new distributed multi-robot deployment algorithm is proposed as the second extension. By relying on the novel strategy of continuous movement in a discrete approximation of the environment, the convergence of the algorithm is proven. Furthermore, as our third extension, we present a new implementation of the proposed deployment algorithm. When the number of robots is large or the region corresponding to each robot is large, the computational time of the locational optimization framework might be high. Thus, a new algorithm is proposed, which is able to run in parallel setup. CUDA is used as a platform for running the proposed algorithm. In our fourth extension, we propose a new discrete deployment strategy which properly works on a topological framework. This framework represents environments as a topological map, which transforms the original two or three-dimensional problem into a one-dimensional, simplified problem, thus reducing the computational cost of the solution. It also makes the new deployment model suitable for the environments that can be represented by a topological map, such as block-shape cities or corridor based buildings i.e. departments in universities, hospitals, governmental offices, etc. It is important to mention that this combination of our discrete deployment with the

topological framework is appropriate for the scenarios, where the map is large and the response must be fast. All the extensions are validated in simulations or actual robots experiments.

List of Figures

1.1 Comparison of centralized and decentralized system. 2
1.2 Deployment problem in a real application. 3

2.1 Example of GVD (green line). 9
2.2 A set of sites ($\mathbf{p}_i$) with corresponding Voronoi regions(V_i). 11
2.3 Euclidean and geodesic metric. 12
2.4 Example of density function in a 2D environment. 13
2.5 Example of CVT with 10 points as sites or seeds. 14
2.6 An example of p-median problem 15
2.7 Threads in memory (NVIDIA, 2007). 17
2.8 CUDA architecture. 18
2.9 CUDA Hardware Model, (NVIDIA, 2007). 20
2.10 New architecture in CUDA 6, NVIDIA (2014). 22

3.1 Deployment of four robots 25

4.1 New metric. 35
4.2 Difference between geodesic and new metric. 36
4.3 Difference between geodesic and new metric Voronoi tessellation. . . . 37
4.4 Equality and inequality constrains. 38
4.5 Discretization and graph representation. 39
4.6 Graph representation. 40
4.7 Robots motion. 42
4.8 Discrete Voronoi tessellation and robots movement. 45
4.9 Input map and density function. 47
4.10 Running the proposed algorithm for five robots. 47
4.11 $\mathcal{H}^*$ function for the first example. 47
4.12 Simulation with a different density function. 48
4.13 Office-like map with GVD 48
4.14 Density function. 49
4.15 Snapshots of running the algorithm 49
4.16 Result of running the algorithm on office-like map. 50

5.1 Discrete approximation of the environment. 55
5.2 Edge modification in neighbor cell. 56
5.3 Assumption on obstacles. 57
5.4 Computing Voronoi region on a dynamic graph. 58
5.5 Graph neighbor nodes of cell s_i. 59
5.6 Neighboring in the new strategy. 59
5.7 Projecting gradient vector. 62
5.8 Thread pool. 67
5.9 Compact adjacency list and the pattern. 69
5.10 Customized compact adjacency list. 70
5.11 Mask array. 72
5.12 Conflict in parallel implementation. 74
5.13 Running parallel wavefront algorithm. 75
5.14 Input descretized map. 78
5.15 Snapshot of a simulation run. 80
5.16 Final trajectory and cost function. 81
5.17 Input map and corresponding density function. 81
5.18 Iterations of running the new deployment method. 82
5.19 Final trajectory and cost function in Office-like Map. 82
5.20 E-puck robot and robot model. 83
5.21 Initial location of robots in the map. 84
5.22 Robots motion during the experiment. 85

6.1 A sketched map 89
6.2 Discretizated map and directions. 91
6.3 Corresponding graph of the map in Fig. 6.2. 92
6.4 Moving between nodes. 92
6.5 Density function represented in graph. 93
6.6 Voronoi subgraph. 98
6.7 Robot model. 103
6.8 Robot system in high-level. 106
6.9 The map of a neighborhood in New York 106
6.10 The density function. 107
6.11 Result of partitioning the input map by the proposed method. 108
6.12 Simulation result. 108
6.13 The map used in real robot experiment. 110
6.14 Deployment of 3 robots in a real scenario. 110

List of Tables

3.1 Different classes of deployment approach. 24

5.1 Notations. 54
5.2 Result of first test. 76
5.3 Result of second test on two graphs. 77
5.4 Result of running Dijkstra algorithm on grid graph. 77

6.1 Set of commands and costs in the simulation. 107
6.2 Comparison of $\mathcal{H}$ in different methods. 107
6.3 Commands applied on robots in the real experiment. 111

List of Algorithms

1 Dijkstra algorithm. 8

2 Distributed algorithm of robot i 41
3 Modified Dijkstra 44

4 Deployment algorithm 61
5 $MSSP()$ 65
6 MSSP_CUDA 71
7 KERNEL1 73
8 KERNEL2 75

9 Distributed controller 95
10 Function $Find_next_best_node()$. 97

List of Symbols

$\Pi_i(GVD)$	Projection of the point p_i onto the GVD
η	First vertex neighbor nodes of robot
$\nabla\mathcal{H}$	Gradient of $\mathcal{H}$ function
Ω	Representing the environment
ω	Angular velocity
$\partial\Omega$	Boundary of the environment
∂V_i	Boundary of the Voronoi region i
ϕ	Distribution density function
a_i	Radius of disk robot i
Co	The cost map between robots and all nodes in a graph
C and $\mathcal{C}$	Cost/weight matrix or cost function in a graph
$\mathcal{C}_s$	Cost/weight of static nodes
$\mathcal{C}_d$	Cost/weight of dynamic nodes
$c(i,j)$ and c_{ij}	Cost between the centers of the cells i and j or an edge
$d(i,j)$	Distance between two node i and j
$d^*(i,j)$	Minimum distance between two node i and j
$\mathcal{E}$	Set of edges in a graph
$\mathcal{E}_s$	Set of static edges
$\mathcal{E}_d$	Set of dynamic edges
$\mathcal{F}_{V_i}$	Free Voronoi region i

G	Graph
$GGVD_i^*$	Corresponds to the set of grid cells so that $d^*(\mathbf{P}(q),\mathbf{P}(p_i))$ is less than $d^*(\mathbf{P}(q),\mathbf{P}(p_j))$, $\forall j \neq i$
$g(q,p_i,G)$	Geodesic distance between nodes q and p_i in the graph G
g_i	Voronoi subgraph
h	Equality constraint
$\mathcal{H},\mathcal{H}^*,\mathcal{H}^0$	Deployment function (cost function)
$\mathcal{I}$	Set of robot commands that links two edges
$\mathcal{L}_{i,j}$	Bisector between two Voronoi region i and j
M	Number of source (robot)
Ma	A mask array for controlling wavefront algorithm
n_p	Number of Search Points
N and $\|\mathcal{V}\|$	Number of node in graph
$\mathcal{N}_G$	A set of graph neighbor nodes
$\mathcal{N}_G(p_i)$	Graph neighbor nodes of node p_i
nid	Index of the neighbor of a node
o	Voronoi tessellation map
$P = \{\mathbf{p}_1,\cdots,\mathbf{p}_M\}$	Configuration of M robots
$\mathbf{p}_i$	Configuration of robot i (vector)
p_i	Node i of graph or robot position in graph
$\mathbf{p}_i^*$	Centroid of a Voronoi cell
$\dot{\mathbf{p}}_i$	Velocity
p_i^*	Next best node to go for robot i
$Path_{Init_To_GVD}$	Path from the initial point to GVD
$Path_{GVD_To_GVD}$	Path from a point on GVD to another point on GVD
$Path_{GVD_To_Goal}$	Path from GVD to the goal point
$\mathbf{QO}$	Set of obstacle
$\mathbf{Q}_{free}$	Free configuration space
$\mathcal{QO}_i$	Configuration Obstacle i

$\mathbf{q}$	Configuration of a point in space
$\mathbf{q}_c$	Center of distribution function
$R = \{r_1, r_2, \cdots, r_M\}$	Index of M robots
$\mathcal{R} = \{r_1, r_2, \cdots, r_k\}$	A subset of R ($\mathcal{R} \subseteq R$), in which robots are neighbors
$\mathcal{R}_i$	A set of neighbor robots with shared Voronoi boundaries V_i
$\mathcal{R}^i_{\mathcal{V}GVD}$	Position of neighbor robots which already lye on $\mathcal{V}GVD$
r_i	Index of robot i
s	Starting vertex in a path
Uc	Temporary memory for cost updating
$\mathbf{u}_i$	Control input
$\mathcal{V}$	Set of nodes in a graph
$\mathcal{V}_s$	Static nodes
$\mathcal{V}_d$	Dynamic nodes
V_i	Voronoi region i
v	Linear velocity
v_s	Source node
$\mathcal{V}GVD$	Set of grid cells that contain GVD curves
$\mathcal{V}$	Set of vertices (nodes)
W_i	Tessellation region i
$\mathcal{W}_1$ and $\mathcal{W}_2$	Coefficient of the new metric
$\bar{w}$	A constant related to the integral element of area
x_0 and y_0	Center of Gaussian function
X_{ij}	Binary variable for edge $[ij]$ in MILP model
$[xy]$	The edge between nodes x and y
y	Inequality constraint
$\mathbf{z}_{p_i,q}$	A vector with direction given by the first edge of the shortest path between p_i and q

List of Abbreviations

APSP	*All-Pair Shortest Path*
AUV	*Autonomous Underwater Vehicle*
CUDA	*Compute Unified Device Architecture*
CVT	*Centroidal Voronoi Tessellation*
GPU	*Graphics Processing Unit*
GPGPU	*General Purpose Graphic Processing Unit*
GVD	*Generalized Voronoi Diagram*
GGVD	*Geodesic Generalized Voronoi Diagram*
MSSP	*Multi Source Shortest Path*
MILP	*Mixed Integer Linear Programming*
MW	*Multiplicative Weighted*
SSSP	*Single Source Shortest Path*
tid	*Thread ID*
TSP	*Traveling Salesman Problem*
UAV	*Unmanned Aerial Vehicle*
VD	*Voronoi Diagram*
VMW	*Visibility-aware Multiplicative Weighted*
WSN	*Wireless Sensor Network*

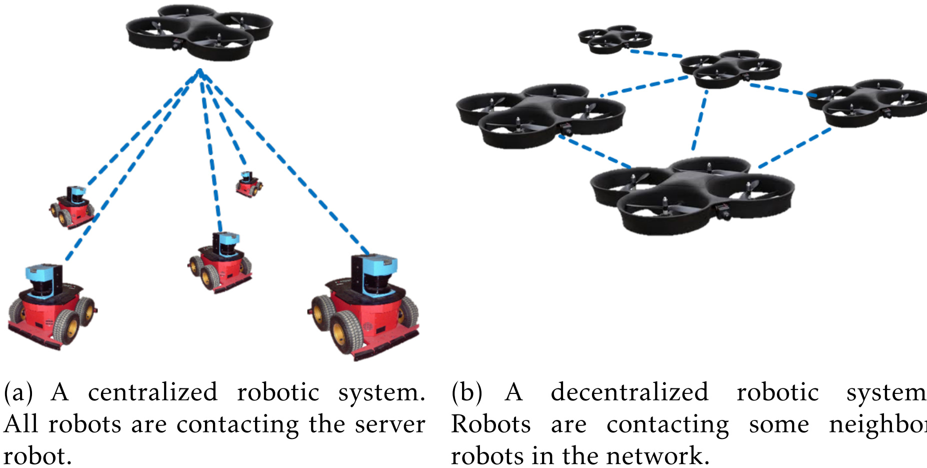
Chapter 1

Introduction

A set of autonomous sensors, which is a networked system that can communicate and use sensing data gathered mutually by each spatially distributed sensor node, is known as a Wireless Sensor Network (WSN) (Suzuki and Kawabata, 2008). Moreover, according to (Bullo et al., 2009), a system composed of a group of robots that sense their own positions, exchange messages following a communication topology, process information, and control their motion is called a robotic network. One can find several applications for this type of system such as surveillance, sensing coverage, environment monitoring, search and rescue, etc. In a multi-robot system, agents might cooperate to accomplish a single task much more efficiently than a single-robot system. Also, in the case of a complex task, usually a group of robots can reach the result much faster. Furthermore, when robots are able to sense the environment independently, after fusing the captured data the effect of a failure in some data from any individual will not have a big impact due to the probability of having redundant information provided by other agents.

There are two common types of strategies to coordinate a group of robots. First, when a leading robot or central computer collects the state information of all the other robots, the tasks and the environment to determine the appropriate motion of each individual robot, this is called centralized strategy. In this case, the difficulty is in adapting to the dynamic environment and to accommodate a large number of robots. The second strategy is called distributed or decentralized in which each robot can compute its own decision according to its states, its local environment and its interactions with nearby robots and other entities. One benefit of this distributed system is the possibility of reduction of the complexity of the involved algorithms such as in the case of path planning. Moreover, the system may be robust to single-robot failure. Fig.

1.1 indicates the communication of robots in both strategies.



(a) A centralized robotic system. All robots are contacting the server robot.

(b) A decentralized robotic system. Robots are contacting some neighbor robots in the network.

Figure 1.1: Comparison of centralized and decentralized system.

In some applications of robotic networks, an important question to be answered is: where should each robot be placed in the environment? In the present work we show distributed solutions to this problem which is referred as the deployment problem by Bullo et al. (2009).

In this topic, the solution is distributed in the sense that each agent depends only on information from a small set of other agents, called neighbors, to compute its actions. Besides, this set of neighbors is dynamic since it might change as the system evolves. As pointed by Cortes et al. (2004), this allows for scalability and robustness.

1.1 Motivation and Contributions

Due to the vast number of applications of multi-robot systems and particularly the usage of autonomous deployment strategies, in this thesis we investigate different aspects of this research topic.

We are interested in finding optimal deployment configurations for a group of robots. A configuration can be considered as optimal if it is a minimizer of a functional encoding the quality of the deployment. The quality of deployment is usually related to the time of response of the network after an event that needs servicing happens in the environment. This time is a function of the distance of the agents from the

event and the agent capabilities (speed, sensor field of view, etc). In order to minimize the distance between agents and events, our approach applies the idea of partitioning the environment into subregions which are then assigned to specific agents. Therefore, each agent is responsible for attending the events in its corresponding subregion.

Figure 1.2: In this example three firefighter cars must be located over the map in such a way that in case of fire in their dedicated region, they respond in minimum time.

An example in Fig. 1.2 considers an application of determining the best station for rescue cars in a city such that they reach the intended area in minimum time. In locational problems, we usually have the objective of finding the best positions to place entities such as: cars, robots or any kind of facility. Given the importance of multi-robot deployment problem and also the fact that it is a new research topic, several researchers are currently interested in developing new efficient strategies which are amenable to real world implementation. In this thesis we tried to address shortages found in previous methods and also extend some works in the literature.

Contribution

In this work, our contributions are divided into 3 main parts which are listed below:

- First, we extend the previous works on multi-robot deployment by considering safety in robots motion. In the new technique GVD is employed as a roadmap, so that robots are forced to keep moving over the GVD. To apply this constraint in robots path planning, we develop a new metric, which computes the shortest path between two points over the GVD. This extension generates safe routes for

the robots during deployment. Moreover, in order to compute next actions of the robots, we developed a new efficient algorithm. Simulation results validates the applicability of the proposed method.

- A new method for deploying a team of robots by considering a continuous movement of robots in a discrete representation of the environment is proposed. This work defines a new strategy to choose the next action, such that the cost function never increases. It makes possible to address the problem of convergence of previously proposed methods such as (Bhattacharya et al., 2013c). We also propose a new efficient parallel implementation of our algorithms. In particular, CUDA based discrete implementation is applied to speed up the graph search algorithm. Simulation and real robot experiments testify the performance of this extension in Chapter 5.

- While the cited deployment techniques focused on environments with two or three dimensions represented by geometric maps, we derived a discrete setup for deployment on a one-dimensional topological map. The topological map represents the robot's workspace as a graph, in our case a direct graph. The idea of this topological map on how to move a single robot in this space is developed by Araujo et al. (2015), where a robot uses human-like commands to move in structured environments. In fact, humans do not need to have a precise metric localization to reach a destination. A simple sequence of directions such as "turn right", "turn left", and "go straight" may be enough for a person to reach its destination in an urban environments or office building. This methodology uses graph search strategies to generate such sequences of commands and deploy the robot team in the environment. The main advantage of topological representation is to eliminate the need for precise localization of the robots. According to these advantages, our new multi-robot deployment setup upon this framework can be considered as a milestone in the literature of deployment. In such a way, we do not need a specific metric for the computation (as for the humans, metric localization is also not necessary). Given a robot that is able to move without the need of localization (by following walls or driveways, for example), high speed

motion and fast response can be achievable. Arguably, this method is suitable for emergency responses like chasing an evader or responding to accidents. It is important to remark that this topological based framework does not need a complete and precise map as input. Thus our method will work in scenarios in which the input map is partially known or just an image (without metric information) is available.

Our simulation results compare our proposed method with the global solution obtained by solving the p-median problem on the same input map. We also executed the algorithm on a team of real robots.

1.2 Document Organization

The document is organized as follows: in the next chapter, we explain some definitions and tools that will be used in the rest of the document; In Chapter 3, a review of related papers in continuous and discrete deployment is presented; The proposed decentralized multi-robot safe deployment is discussed in Chapter 4. In Chapter 5, our proposed deployment algorithm with convergence proof and new implementation will be presented;

Chapter 6 is dedicated to deploying robots in topological maps. This chapter also contains: discrete topological map representation, optimal deployment in this representation, the proof of convergence and implementation results; Chapter 7 concludes this text with a summary, list of limitations, and future directions to be pursued. We also show the list of publications derived during the development of this work in this chapter.

Chapter 2

Background

In this chapter we present some of the tools that will be useful in the techniques proposed in this work.

2.1 Graph (Dijkstra Algorithm)

Searching in a graph to find a path between two locations is one of the useful operations in a routing problem. In this section, we define the Single Source Shortest Path (SSSP) problem and review the Dijkstra algorithm (Dijkstra, 1959) as its solution.

Let $G = \{\mathcal{V},\mathcal{E},\mathcal{C}\}$ be a weighted graph, where $\mathcal{V}$ is the set of nodes/vertices, $\mathcal{E}$ is the set of edges and finally $\mathcal{C}$ is the set of non-negative weights of edges between nodes. The SSSP problem is the one of finding the path between a source vertex s and a destination vertex t which has minimum total weight. An efficient algorithm to solve this problem is the so-called Dijkstra algorithm. This algorithm receives the graph G and the source vertex s as inputs and returns the distance map, d, and array *pre* which contains the pointers to the "next-hop" nodes in the shortest routes to the source (s). Algorithm 1 shows the original Dijkstra algorithm.

In this algorithm, vertices are classified into two main categories: visited nodes and unvisited nodes. Thus, in each iteration, the closest node (q) from the unvisited nodes, in the priority queue (Q), is removed and added to the visited part. In this way, q's neighbors are investigated to see whether the new path from q is closer to them or not. Such that in lines 13 and 14 the distance map d and array *pre* will be updated to the new values (this is also called “relaxation”). The evolution of this process is usually associated to a wavefront propagating from the source node. The complexity

Algorithm 1: Dijkstra algorithm.

```
   Input: G, s
   Output: d, pre
 1 d(s) ← 0 // Distance from source to source
 2 foreach v ∈ V do // Initialization of distance
 3     if v ≠ s then
 4         d(v) ← ∞
 5         pre(v) ← undefined // Previous node in optimal path from source
 6     Q ← v // Add all v in the list Q
 7 while ( Q ≠ ∅ ) do
 8     q ← argmin_{q'∈Q} d(q') // return vertex in Q with minimum distance to source
 9     Q ← Q \ q // Remove q from Q
10     foreach w ∈ N_G(q) do // For each graph neighbor node of q
11         d' ← d(w) + C(w,q)
12         if d' < d(q) then // Relaxation
13             d(q) ← d' // Update the cost of neighbor nodes of q
14             pre(q) ← w // by considering new cost from source to q
```

of a naive implementation is $O(|\mathcal{V}|^2)$, where $|\mathcal{V}|$ is the number of nodes, but other implementations using a *fibonachi heap* (Cormen et al., 2001), have a much better result $O(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)$, where $|\mathcal{E}|$ is the number of edges.

2.2 Generalized Voronoi Diagram (GVD)

Consider the set of obstacles $\mathbf{QO} = \{\mathcal{QO}_1, \cdots, \mathcal{QO}_n\}$ defined in a planar configuration space. This set induces a structure called Generalized Voronoi Diagram (GVD). A set of points in the free configuration space ($\mathcal{Q}_{free}$) is defined as the Voronoi region (V_i) of the obstacle $\mathcal{QO}_i$, if these points are closer to $\mathcal{QO}_i$ than to all the other sites, where a site is the same as an obstacle in this case (Choset et al., 2005).

$$V_i = \{\mathbf{q} \in \mathcal{Q}_{free} |\ d(\mathbf{q}, \mathcal{QO}_i) \leq d(\mathbf{q}, \mathcal{QO}_j), \forall j \neq i\}, \tag{2.1}$$

where $d(\mathbf{q}, \mathcal{QO}_i)$ is the minimum distance between $\mathcal{QO}_i$ and $\mathbf{q}$. The two-equidistant surjective surface, $\mathcal{L}_{i,j}$ is the set of points equidistant to two obstacles $\mathcal{QO}_i$ and $\mathcal{QO}_j$

with distinct gradient vectors:

$$\mathcal{L}_{i,j} = \{\mathbf{q} \in \mathcal{Q} | d(\mathbf{q}, \mathcal{QO}_i) = d(\mathbf{q}, \mathcal{QO}_j) \text{ and } \nabla d(\mathbf{q}, \mathcal{QO}_i) \neq \nabla d(\mathbf{q}, \mathcal{QO}_j), j \neq i\}. \tag{2.2}$$

The points in $\mathcal{L}_{i,j}$ (that are part of the GVD) are those in which $\mathcal{QO}_i$ and $\mathcal{QO}_j$ are the closest obstacles. Therefore we can define the set:

$$V_{i,j} = \{\mathbf{q} \in \mathcal{L}_{i,j} | \; \forall h, \; d(\mathbf{q}, \mathcal{QO}_i) \leq d(\mathbf{q}, \mathcal{QO}_h)\}. \tag{2.3}$$

This last definition allows us to formally define the GVD:

$$GVD = \bigcup_i \bigcup_j V_{i,j}. \tag{2.4}$$

In Fig. 2.1 an example of definition given by Eq. (2.4) is shown.

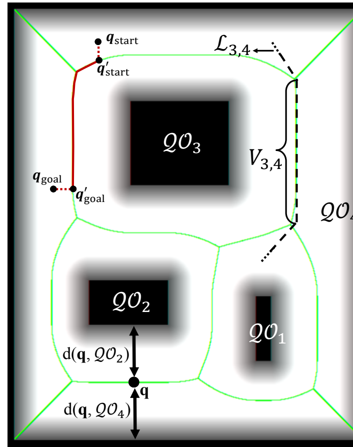


Figure 2.1: Example of GVD (green line).$\mathcal{QO}_1$, $\mathcal{QO}_2$, $\mathcal{QO}_3$ and $\mathcal{QO}_4$ are the obstacles or sites. $\mathbf{q}$ is an equidistant point between sites $\mathcal{QO}_2$ and $\mathcal{QO}_4$ that belongs to the GVD. The line $\mathcal{L}_{3,4}$ is a bisector between sites $\mathcal{QO}_3$ and $\mathcal{QO}_4$. $V_{3,4}$ is the part of line $\mathcal{L}_{3,4}$ which is a portion of the GVD.

An interesting feature of the GVD is that it can be used as a roadmap for path planning. This means that it is always possible to plan a path from a point $\mathbf{q}_{start} \in \mathcal{Q}_{free}$ to a point $\mathbf{q}_{goal} \in \mathcal{Q}_{free}$, if a solution exists, in such a way that the robot moves first to some point $\mathbf{q}'_{start} \in GVD$, moves along the GVD to some point $\mathbf{q}'_{goal} \in GVD$ in the sequence, and finally reaches the goal by moving from $\mathbf{q}'_{goal}$ to $\mathbf{q}_{goal}$ (Choset et al.,

2005) (see Fig. 2.1). Given the fact that the GVD is formed by points equidistant from the closest obstacles, in general it provides a safe route for the motion. Therefore, if several excursions in the environment are demanded from robots and a map of this environment is given, it is worth computing the GVD and then use it as a roadmap for such excursions. Given a GVD, planning a path is a matter of running a simple graph search. In the present work we use GVDs to incorporate safety in a multi-robot motion and simplify the workspace.

2.3 Voronoi Tessellation

In this section, to explain Voronoi tessellation we use the same definitions in the context of GVD (Section 2.2). In the free configuration space ($\mathcal{Q}_{free}$) of an environment, Voronoi tessellation is defined as regions associated to a set of points. If we consider to $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_M\}$ be the *sites* or points distributed over $\mathcal{Q}_{free}$, instead of obstacles in GVD, where $\mathbf{p}_i \in \mathcal{Q}_{free}$ indicates the position of point i, the set $V = \{V_1, \cdots, V_M\}$ is the corresponding Voronoi tessellation given by:

$$V_i = \{\mathbf{q} \in \mathcal{Q}_{free} | \, d(\mathbf{q}, \mathbf{p}_i) \leq d(\mathbf{q}, \mathbf{p}_j), \forall j \neq i\}, \tag{2.5}$$

where, d is the distance function, so that if we consider d as Euclidean distance it can be written: $d(\mathbf{q}, \mathbf{p}_i) = \|\mathbf{q} - \mathbf{p}_i\|$. Moreover, the boundary between two Voronoi regions V_i and V_j is:

$$\mathcal{L}_{ij} = \{\mathbf{q} \in \mathcal{Q}_{free} | \, d(\mathbf{q}, \mathbf{p}_i) = d(\mathbf{q}, \mathbf{p}_j), \, j \neq i\}. \tag{2.6}$$

If we denote the boundary of a Voronoi region V_i by ∂V_i, a neighbor site of $\mathbf{p}_i$ can be defined as:

$$\mathcal{R}_i = \{\mathbf{p}_j \in P \mid \partial V_i \cap \partial V_j \neq \emptyset, \, i \neq j\}. \tag{2.7}$$

An example of environment with four sites and their corresponding Voronoi cells (regions) are depicted in Fig. 2.2. A convex environment is partitioned to four Voronoi

cells which are associated to each point $\mathbf{p}_i$ (based on Euclidean distance). Also the boundaries between regions are shown by lines. It should be noticed that in a Voronoi tessellation we have: $\bigcup_{i=1}^{M} V_i = \mathcal{Q}_{free}$, and also $I(V_i) \bigcap I(V_j) = \emptyset$, $\forall i \neq j$, where $I(V_i)$ returns the interior of Voronoi region i.

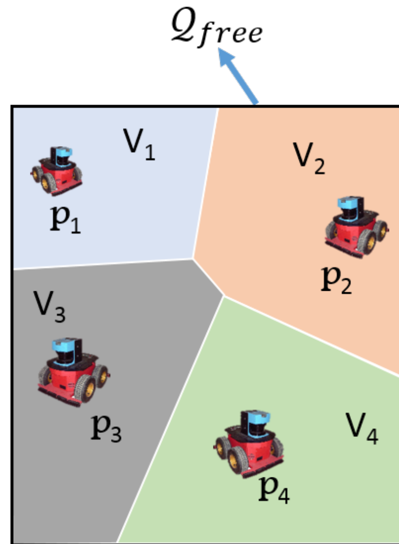


Figure 2.2: A set of sites ($\mathbf{p}_i$) with corresponding Voronoi regions(V_i).

2.4 Locational Optimization Based Deployment

In this section we present the general problem of deployment considering the locational optimization framework proposed by Cortes et al. (2004), which is the basis to our work.

2.4.1 Problem Definition

Consider a team of M robots has to be distributed on a bounded environment $\Omega \in \mathbb{R}^2$. The configuration of robots is defined by $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_M\}$, where $\mathbf{p}_i \in \Omega$. In this setup, robots must cover the whole environment, so that each of them is responsible to a Voronoi cell V_i, where $V = \{V_1, \cdots, V_M\}$, and $\bigcup_{i=1}^{M} V_i = \Omega$. The quality of this deployment can be measured by following equation:

$$\mathcal{H}(P,V) = \sum_{i=1}^{M} \mathcal{H}(\mathbf{p}_i, V_i), \tag{2.8}$$

and,

$$\mathcal{H}(\mathbf{p}_i, V_i) = \int_{V_i} d(\mathbf{q}, \mathbf{p}_i)^2 \phi(\mathbf{q}) d\mathbf{q}, \tag{2.9}$$

where d denotes the distance function between the robots and points. Different metrics can be used to compute the distance in an environment. For instance, in Fig 2.3 a comparison between Euclidean and geodesic distance in a non-convex environment is shown. While Euclidean distance between $\mathbf{p}$ and $\mathbf{q}$ is shorter and suitable for convex environment, geodesic distance is more realistic in a non-convex scenario (dashed line), in the sense of considering obstacles.

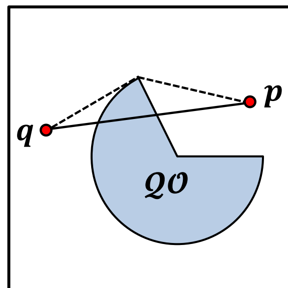


Figure 2.3: Euclidean and geodesic (dashed line) distance in a non-convex environment.

In Eq. (2.9), for each point $\mathbf{q} \in \Omega$, the function $\phi(\mathbf{q})$ denotes the importance of the point. We call this function $\phi : \Omega \rightarrow \mathbb{R}^+$, a distribution density function. In other words, ϕ shows the priority of servicing events in the environment by robots, hence the regions with higher priority have higher weights. In Fig. 2.4, an example of Gaussian function, as a density function, in a planar environment is shown, in which the center of environment (corresponding to the center of function) needs more priority to be covered by robots. Throughout our text we assume that robots have access to the information of density function before starting the deployment process. However, when the robots do not know this function, it can be learned online by techniques developed in (Schwager et al., 2009) based on sensor data.

Given the above explanation, in general in locational based multi-robot deploy-

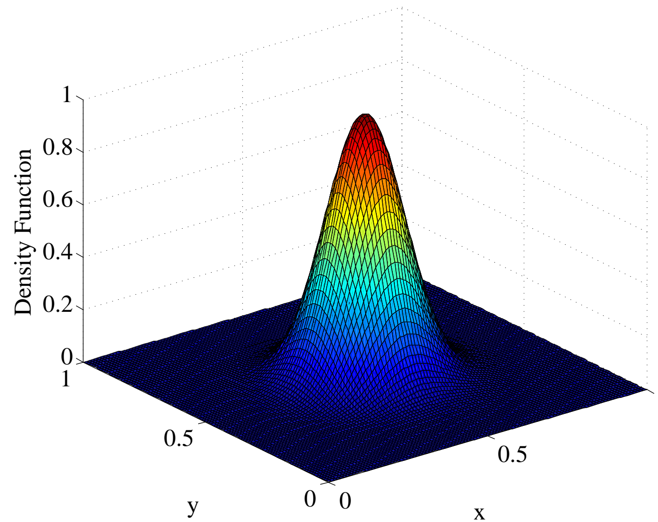


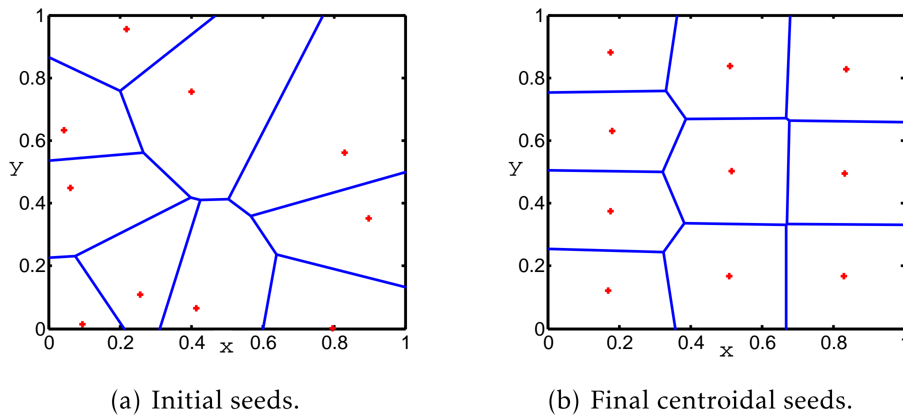
Figure 2.4: Example of density function in a 2D environment. The center of Gaussian function is placed at the point $\mathbf{q}_c = [0.5, 0.5]^T$ and its support (spread of the blob in x and y) is equal to $||\mathbf{q} - \mathbf{q}_c|| < 0.3$.

ment, a team of robots cover an area to optimize the cost function ($\mathcal{H}$), which directly depends on the distance between robots and points. Therefore, the better is the distribution of robots over the environment, the lower is the value of $\mathcal{H}$. In this way, our deployment problem is translated to a minimization problem.

According to the work by Cortes et al. (2004), if we consider a convex environment with Euclidean distance function d, it is easy to show that if robots are located on *centroid* of Voronoi partitions, the minimum value of $\mathcal{H}$ function will be obtained. The centroid can be defined by:

$$\mathbf{p}_i^* = \frac{\int_{V_i} \mathbf{q}\phi(\mathbf{q})d\mathbf{q}}{\int_{V_i} \phi(\mathbf{q})d\mathbf{q}}. \tag{2.10}$$

Therefore, to minimize the function in Eq. (2.8), robots must be driven to the centroid of their Voronoi tessellation. In such a way, the result partitions are known as *Centroidal Voronoi Tessellation* (CVT). Lloyd's algorithm is a well-known iterative and discrete approach for CVT which is proposed by Lloyd (1982). By having an environment with some points as sites, in each iteration of this algorithm the Voronoi partitions and their centroids are computed, later the points will be moved to the centroids.



(a) Initial seeds. (b) Final centroidal seeds.

Figure 2.5: Example of CVT with 10 points as sites or seeds.

This process will stop when the current sites (points) corresponding to the centroids of the next iteration. An example of this method is shown in Fig. 2.5. In this thesis, the most part of our works is based on the continuous version of this algorithm which is proposed by Cortes et al. (2004). In their work, the kinematic model of a robot is defined as:

$$\dot{\mathbf{p}}_i = \mathbf{u}_i, \tag{2.11}$$

where $\mathbf{u}_i$ is the robot control input. Also in a continuous setup, the following control law guarantees that the system converges to a CVT:

$$\mathbf{u}_i = -k(\mathbf{p}_i - \mathbf{p}_i^*), \tag{2.12}$$

where k is a positive weight. According to Eqs. (2.8) and (2.9), finally, the gradient-descent control law that leads the robots to CVT, in Euclidean distance (d) and $f(x) = x^2$, can be computed as (Cortes et al., 2004):

$$\frac{\partial \mathcal{H}}{\partial \mathbf{p}_i} = 2 \int_{V_i} \phi(\mathbf{q})(\mathbf{p}_i - \mathbf{p}_i^*) d\mathbf{q}. \tag{2.13}$$

It is important to notice that $\mathcal{H}$ is a non-convex function, which implies that the system will in general converge to a CVT that corresponds to a local minimum.

2.4.2 p-Median Problem

The locational optimization problem in Section 2.4.1 was defined in a continuous setup, but if we discretize the input map into cells and represent it as a graph, then the deployment problem can be considered as a p-median problem. In p-median problem the main objective is to find location of p facilities (or medians) relative to a set of users or customers, in which the sum of the shortest demanded distance from customers to facilities is minimized. The p-median problem was initially proposed by Hakimi (1964). This problem is classified as NP-hard, therefore many heuristic and meta-heuristic methods have been proposed to solve it (Mladenovic et al., 2007; Rolland et al., 1996). The mathematical model of this problem can be defined as follows.

Consider a set of possible locations for facilities $S = \{1,\ldots,M\}$, and a set of customers $Y = \{1,\ldots,N\}$. d is a distance function, in which $d(i,j)$ indicates the shortest distance from customer i to facility j ($i \in Y$ and $j \in J$, where J is a subset of S). Thus, the objective is to minimize the given function:

$$F = \sum_{i \in Y} \min_{j \in J} d(i,j), \tag{2.14}$$

where $J \subseteq S$ and $|J| = p$. An example of this problem with 7 customers and 2 facilities is shown in Fig. 2.6.

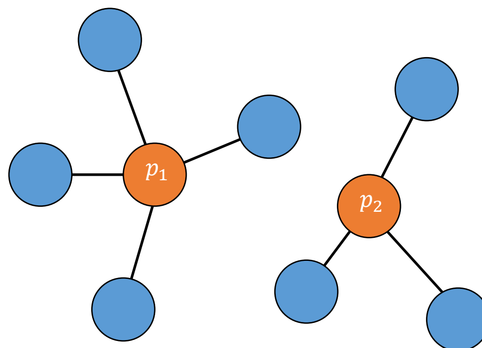


Figure 2.6: A solution of p-median problem with 2 facilities and 7 customers.

According to the above explanation, in this thesis, to validate the efficiency of our proposed deployment algorithm in a discrete setup (in Chapter 6), we compared with

the optimum result achieved by solving its corresponding p-median problem. Such that, in p-median problem, N points (vertices or customers) are assigned to M robots (facilities).

2.5 GPU (Graphics Processing Unit) and Compute Unified Device Architecture (CUDA)

GPGPU (General Purpose Computing on Graphics Processing Unit) has become a hot topic nowadays in several fields that demand high performance hardware. This is a method of using the GPU (Graphics Processing Unit) to perform computations for tasks other than graphics, such as astrophysical modeling, signal/image processing, electromagnetic field computation, etc. The great advantage of this strategy is the parallel nature of graphics processing.

Compute Unified Device Architecture (CUDA) was developed by NVIDIA as a parallel computing solution which covers both parallel software and hardware architecture. It can execute thousands of threads at the same time, thus CPU sees a CUDA device as a multi-core co-processor.

2.5.1 Software Model:

The execution in CUDA is based on threads. Hence programmers must define the number of threads for their usage. A group of threads creates a block. Inside a block, threads can cooperate together by efficiently sharing data through some fast shared memory and synchronizing their execution to coordinate memory accesses. Each thread is identified by its *thread ID*, which is the thread number within the block. A block can be defined as a 2 or 3 dimension array to simplify complex addressing. For example, in the case of 2-D block with size (S_x,S_y), the thread ID of a thread with index (x,y) is $x + y * S_x$.

The maximum number of threads within a block is limited, but blocks with the same size and dimension can be joined into a grid (See Fig. 2.7). A grid can be compiled once with all its threads, so that the maximum number of threads which can be

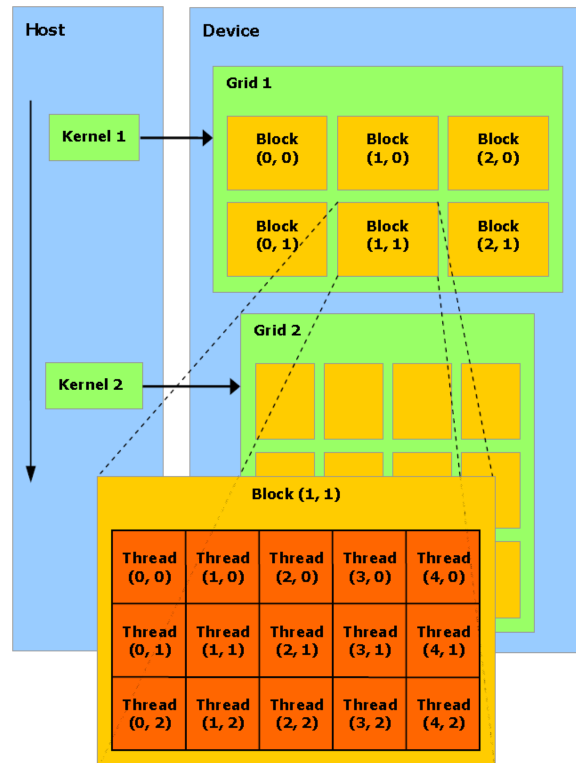


Figure 2.7: Threads in memory (NVIDIA, 2007).

launched at the same time is very large. However, it must be noticed that threads in different thread blocks from the same grid cannot communicate and synchronize with each other.

As illustrated in Fig. 2.8, the software architecture is composed of several layers. Developers use the CUDA API (Application Programming Interface) in their applications and also CUDA runtime and CUDA driver to run and launch their application on GPU.

CUDA as a programming interface provides the ability to contact and use GPU for users familiar with programming language. In fact, CUDA runtime library contains functions to control and access *host* (GPU) and *guest* (CPU) space.

Kernels

An important step in CUDA programming is defining a C function, which is called *kernel*. In contrast to regular C function, when a kernel is launched, it will be executed

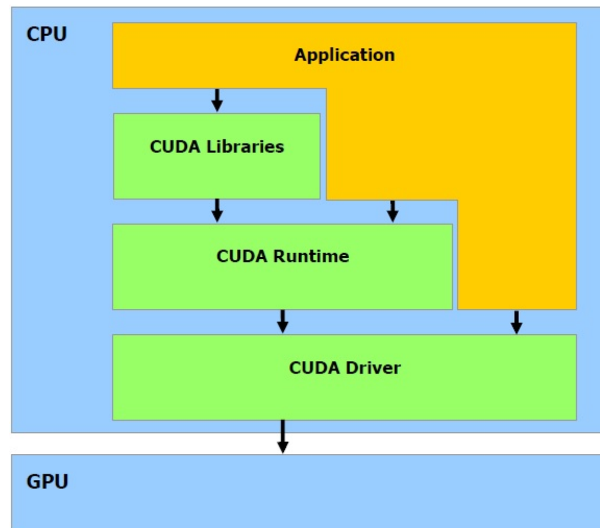


Figure 2.8: Compute Unified Device Architecture Software Stack (NVIDIA, 2007).

N times in parallel by N different CUDA threads. The user can define a kernel by writing the keyword *__global__* at the beginning of the function declaration. Furthermore, when calling it, the number of blocks and threads, which will be used in the GPU, is set between " $<<<\cdots>>>$ " after the name of the function. A simple program for adding two vectors is defined in the code below (Sanders and Kandrot, 2010). In the first code, the function for summing two vectors in CPU and storing the result into vector C is shown. Inside the code, a loop is applied to repeat the sum operator N times, where N is the size of the input vectors.

```
1 void VecAdd( int *a, int *b, int *c )
2 {
3   for (i=0; i < N; i++)
4   {
5       c[i] = a[i] + b[i];
6   }
7 }
```

In the case of CUDA, the same function, for adding two vectors, will be defined as the following code (Sanders and Kandrot, 2010):

```
1 void VecAdd( int *a, int *b, int *c )
2 {
3   int tid = threadIdx.x;
```

```
4      if (tid < N)
5          c[tid] = a[tid] + b[tid];
6  }
```

Differently from the function which was defined in CPU, here there is no loop. This means the line inside the function runs N times in N threads in a parallel way. Each thread that executes the kernel is given a unique thread ID that is accessible within the kernel through the built-in $threadIdx$ variable. The Code below shows how the number of threads and other input parameters are passed (Sanders and Kandrot, 2010):

```
1  #include <iostream>
2  #define N 10
3  int main( void )
4  {
5      int a[N], b[N], c[N]; //Input array from CPU
6      int *dev_a, *dev_b, *dev_c; // variables for GPU
7          Initit(a,b,c);//Initialize a,b and c
8      // allocate the memory on the GPU
9      cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
10     cudaMalloc( (void**)&dev_b, N * sizeof(int) ) ;
11     cudaMalloc( (void**)&dev_c, N * sizeof(int) ) ;
12     // fill the arrays 'a' and 'b' on the CPU
13         cudaMemcpy( dev_a, a, N* sizeof(int), cudaMemcpyHostToDevice);
14         cudaMemcpy( dev_b, b, N* sizeof(int), cudaMemcpyHostToDevice);
15         VecAdd<<<1,N>>>(dev_a, dev_b, dev_c);
16         cudaMemcpy( c, dev_c, N* sizeof(int), cudaMemcpyDeviceToHost );
17         for (int i=0; i<N; i++)
18                 printf("\%d+\%d=\%d \n",a[i],b[i],c[i]);
19         cudaFree(dev_a);
20         cudaFree(dev_b);
21         cudaFree(dev_c);
22         return 0;
23 }
```

The number of threads is directly related to the problem which is defined by the developer. Here in our example, 1 block is used and also the number of threads in a

block is equal to the size of the input vector or $N = 10$. On current GPUs, a thread block may contain up to 1024 threads.

It can be seen in the code that before using the variables in GPU they should be allocated by *cudaMalloc*(). This function behaves very similarly to the standard C call *malloc*(), but it tells the CUDA runtime to allocate the memory in the device. Another useful function is *cudaMemcpy*, similarly to the standard C function *memcpy*. This function copies data from host to guest and the other way around (Sanders and Kandrot, 2010).

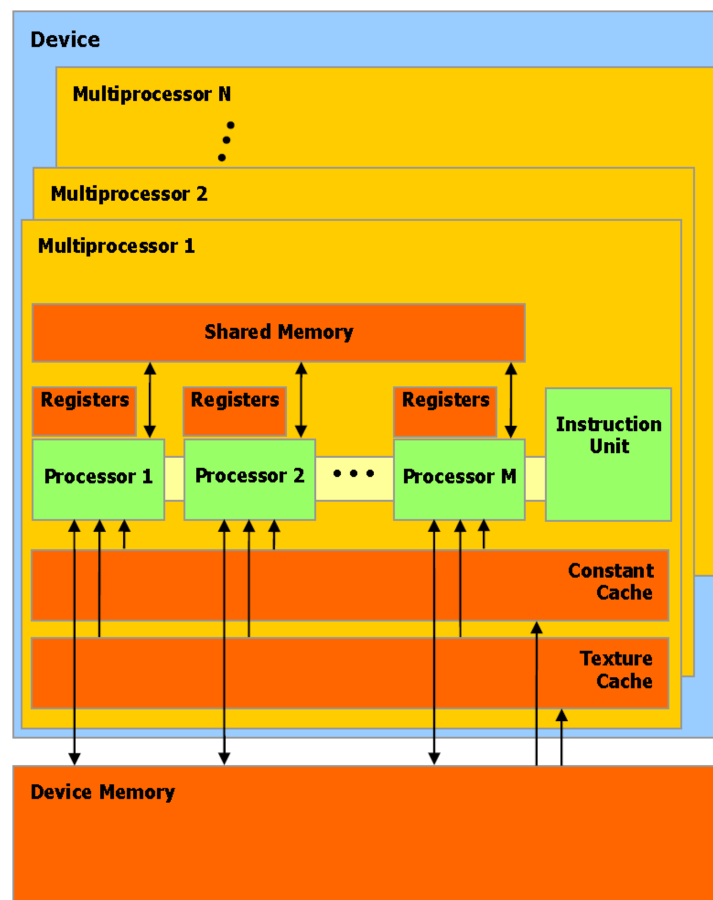


Figure 2.9: CUDA Hardware Model, (NVIDIA, 2007).

2.5.2 Hardware Model:

CUDA has two main taxonomies: the GPU is called the *device* and the CPU is called the *host*. It also consists of a set of multiprocessors (See Fig. 2.9). Each multiprocessor has access to the four following memory types (NVIDIA, 2011):

- *shared memory*, which is common to all the n processors,
- A set of 32-bit registers per processor,
- A read-only *Constant cache* that is shared between all processors,
- A read-only *Texture cache* that is also shared by all processors.

The two last types of cache memory speed up reading from constant and texture memory space. The device memory is accessible from all multiprocessors, hence they can communicate to each other via this memory.

Atomic operators:

In multi-threading applications, a common problem which must be avoided is race condition. It happens when two or more threads want to operate on the same shared memory concurrently. Atomic operations are often used to prevent race conditions. An atomic operation is capable of reading, modifying, and writing a value back to memory without the interference of any other threads, which guarantees that a race condition would not occur. Atomic operations in CUDA generally work for both shared memory and global memory. For example:

```
int atomicAdd(int* address, int val);
```

This *atomicAdd* function can be called within a kernel. When a thread executes this operation, a memory address is read, has the value of "val" added to it, and the result is written back to memory. There are other operators for the main mathematical and logical operations.

2.5.3 New CUDA Hardware Model

Because of the complexity of managing these different kinds of memories, NVIDIA proposed a new version of CUDA (NVIDIA, 2014). In CUDA 6, they introduced *uniform memory* to have a shared data between CPU and GPU. Therefore, this memory can be accessed by using a single pointer. Fig. 2.10 illustrates this new architecture.

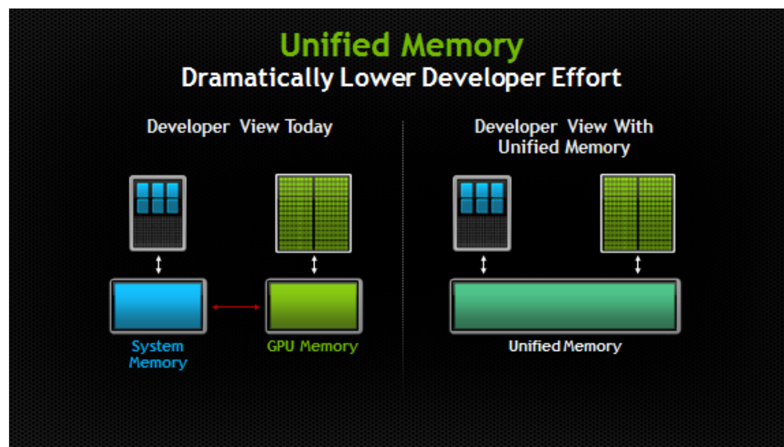


Figure 2.10: New architecture in CUDA 6, NVIDIA (2014).

In Chapter 5, we are going to show a version of the proposed distributed algorithm compatible with CUDA.

Chapter 3

Related Work

3.1 Multi-Robot Deployment

Multi-robot deployment problems have attracted attention of several groups due to the great applicability of robotic networks. A major category in multi-robot deployment control schemes is the one based on artificial potential fields or force fields: Parker (2002) developed a distributed algorithm to coordinate a group of robots to monitor multiple moving targets. This strategy was based on the use of attractive and repulsive force fields. Reif and Wang (1999) proposed artificial force laws between pairs of robots or robot groups to control Very Large Scale Robotic (VLSR) systems. Since the force laws may reflect the "social relations" among robots, this method was named *Social Potential Field*. Similarly, Howard et al. (2002) presented a potential field based method in which the final force for each robot is achieved by summing both the repulsive forces from obstacles and the repulsive forces from other robots. The objective in this case is to maximize the environment coverage. Poduri and Sukhatme (2004); Popa et al. (2004); Ji and Egerstedt (2007) also used potential field based controllers for coverage, formation control and preservation the connectivity of agents in an ad hoc network.

Another important category is the one built upon the *Locational Optimization Framework* (Okabe et al., 1992). This was initially proposed by Cortes et al. (2004). The authors in (Cortes et al., 2004) present a distributed and asynchronous approach for optimally deploying a uniform robotic network in a domain based on a framework for optimized quantization derived by Lloyd (1982). Each agent (robot) follows a control law, which is a gradient descent algorithm that minimizes the functional encoding the quality of the deployment. Furthermore, this control law depends only on the infor-

Table 3.1: Different classes of deployment approach.

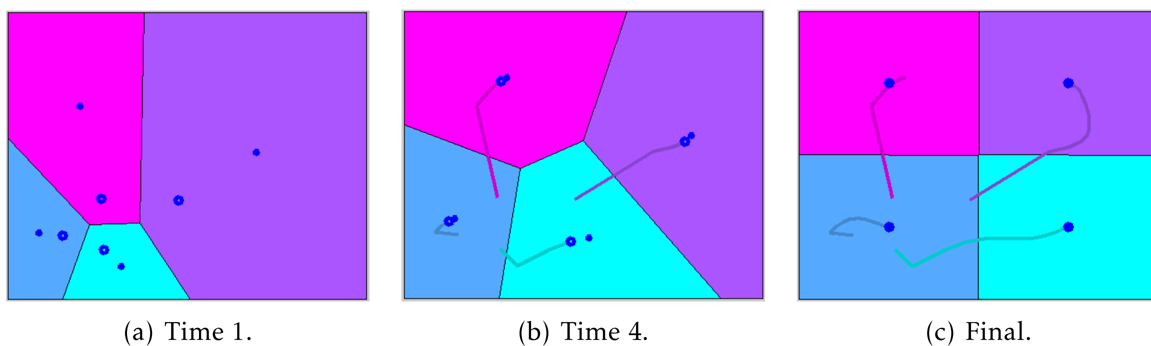
Methodology	NC[1]	NE[2]	Con[3]	Het[4]	GB[5]	CP[6]	Dec[7]
Cortes et al. (2004)	-	-	✓	-	-	✓	✓
Pimenta et al. (2008)	✓	✓	✓	✓	-	✓	✓
Schwager et al. (2009)	-	-	✓	-	-	✓	✓
Breitenmoser (2010)	✓	✓	✓	-	-	✓	✓
Stergiopoulos and Tzes (2011)	-	-	✓	✓	-	-	✓
Schwager et al. (2011)	✓	-	✓	-	-	✓	✓
Mahboubi and Sharifi (2012)	-	-	✓	-	-	-	✓
Durham and Carli (2012)	✓	✓	-	✓	✓	✓	✓
Yun and Rus (2013)	✓	✓	-	-	✓	✓	✓
Bhattacharya et al. (2013b)	✓	✓	✓	-	✓	✓	✓
Sharifi et al. (2014)	-	-	✓	-	-	✓	✓
Sharifi et al. (2015), Pierson et al. (2015)	-	-	✓	✓	-	✓	✓

[1] NC: Non-Convex [2] NE: non-Euclidean [3] Con: Continuous
[4] Het: Heterogeneous [5] GB: Grid-Based [6] CP: Convergence-Proof
[7] Dec: Decentralized

mation about the position of the robot and of its immediate neighbors. Neighbors are defined to be those robots that are located in neighboring Voronoi cells. Besides, these control laws are computed without the requirement of global synchronization. As we explained in Section 2.4, the functional also use a *distribution density function* which weighs points or areas in the environment that are more important than others. Thus, it is possible to specify areas where a higher density of agents is required. This is important if events happen in the environment with different probabilities in different points. Furthermore, this technique is adaptive due to its ability to address changing environments, tasks, and network topology.

Coverage in the sense of (Cortes et al., 2004) is illustrated in Fig. 3.1. In this figure four robots are deployed in a convex environment. The density function used is uniform, so all points of the environment have identical priority. At the beginning, the environment is divided in four Voronoi regions, one for each robot. The technique proposed in Cortes et al. (2004) generates velocity vectors for each robot so that they continuously move towards the centroids of their respective Voronoi regions, which, as a consequence, change the region itself, as show in Fig. 3.1. As the time evolves, it can

be proved that the multi-robot system converges to a configuration where the environment is equally distributed among the robots or, in the case of non-uniform density functions, is distributed according to this function. It is important to notice that, even in convex workspaces, real world implementations of this methodology require global and precise metric localization for the robots and non-linear control strategies to follow the velocity vectors.



(a) Time 1. (b) Time 4. (c) Final.

Figure 3.1: Deployment of 4 robots in a convex environment using the approach proposed by Cortes et al. (2004). In (a) and (b) the “*" sign indicates the center of the Voronoi regions, which represent the current goal position for each robot. Voronoi regions are highlighted with different colors.

Different extensions of the framework devised by Cortes et al. (2004) have been proposed in the literature. Table 3.1 lists some of the articles with their properties: convex or non-convex, Euclidean or non-Euclidean, continuous or discrete, heterogeneous or homogeneous teams, grid-based or geometric, with convergence-proof or without, and decentralized or centralized. In this work we focus on two main schemes: *continuous* and *discrete*. Hence, in the following we are going to review the literature of both schemes separately.

3.1.1 Multi-Robot Deployment in Continuous Space

After the initial work by Cortes et al. (2004), which can only be applied to convex and static environments, Pimenta et al. (2008) applied geodesic distance for deploying a team of heterogeneous robots in non-convex environments. The problem of considering time-varying distribution density functions was studied by Pimenta

et al. (2010) to solve a task of simultaneous coverage and intruders tracking. For a similar problem, Schwager et al. (2009) developed an adaptive controller, such that robots learn the distribution of sensory information (density function) during the deployment. Another approach for deploying multiple robot in non-convex environment was presented by Breitenmoser (2010). In order to avoid collision with obstacles, the authors combined the deployment control law with a local planner (Tangent Bug).

Most of the previously cited works can be classified as Voronoi-based coverage strategies since they use the centroid of the current Voronoi region in their controller. However, there are other papers that provide alternative partitioning techniques. Stergiopoulos and Tzes (2011) propose a method for area-coverage in the case where the sensors are heterogeneously range-varied. In this method, differently from what we call CVT (Centroidal Voronoi Tessellation (Cortes et al., 2004)), as a standard method for tessellation, they performed a different space-partitioning scheme, which was proposed previously by Tzes and Stergiopoulos (2010). In this work, they developed a coverage-oriented modified Voronoi, based on different sensing areas from the set of heterogeneous nodes (agents) and preserved the convexity of the Voronoi regions. In their simulation result, they claim that the proposed method, in the case of heterogeneous networks, can increase the total sensed area in the same time as the standard tessellation method.

Multi-robot deployment with the use of multiplicative weighted Voronoi (MW-Voronoi) diagram in an obstacle free environment is discussed by Mahboubi and Sharifi (2012). In this work, the authors consider different weights for each robot in the process of constructing the Voronoi tessellation. Moreover, in the presence of obstacles, the tessellation is done by applying a visibility-aware multiplicatively weighted Voronoi (VMW-Voronoi) diagram. Therefore, it is possible to have an uncovered region, where it is not only invisible from the robots viewpoint, but also will not be considered in the Voronoi tessellation.

A deployment strategy for multi-agent system that considered communication delay and sensors effectiveness variation is presented by Sharifi et al. (2014). In this work a new partitioning technique is developed in order to address variation in sensors be-

havior, which is called Guaranteed Multiplicative Weighted (GMW) Voronoi. Agents with different types of dynamics are taken into account by the same authors in (Sharifi et al., 2015). They used MV-Voronoi partitioning approach to find the corresponding region for each robot based on their dynamics.

Caicedo-Nunez and Zefran (2008a) and Caicedo-Nunez and Zefran (2008b) considered non-convex environments by constructing a diffeomorphism to convex regions with isolated obstacles in its interior, in which regular Voronoi coverage can be applied. As they mentioned besides significant computational challenges, the generated solution may differ from the corresponding optimal coverage solution in the original space. They addressed these shortages in the work by Caicedo-Nunez and Zefran (2008a) by characterizing a set of stationary points for the Lloydś algorithm in general regions.

The DisCoverage [1] algorithm in a convex environment for multi-robot exploration, based on the work of Cortes et al. (2004), is proposed by Haumann et al. (2010). Moreover, an extension of this algorithm for non-convex environments in the specific context of exploration, by including the idea of using geodesic distance as the work of Pimenta et al. (2008), is done by Haumann et al. (2011). This method transforms non-convex environments into star-shaped domains and use these new domains to apply the DisCoverage algorithm.

Ny and Pappas (2013) consider three related classes of problems: coverage control, spatial partitioning, and dynamic vehicle routing. They proposed an adaptive algorithm which was based on stochastic gradient algorithms that optimize utility functions in the absence of a priori knowledge of event location distribution.

The present work further extends the works by Pimenta et al. (2008) and Bhattacharya et al. (2013c). In the first part of our work, we incorporate safety as a parameter into the deployment problem. By merging different Voronoi diagrams, including the well known Generalized Voronoi Diagram (GVD) (Choset et al., 2005) and by considering a constrained optimization problem in the context of the Locational Optimization Framework, we generate safe routes for the robots during deployment and also after deployment when servicing a given point of the environment. We propose

[1] In (Haumann et al., 2010), using Voronoi partition-based coverage algorithm in multi-robot exploration is called *DisCoverage*.

a new Voronoi Diagram which is built according to a new metric that takes into account shortest paths that traverse the GVD. Moreover, in order to consider real world environments, we devise a new efficient algorithm to compute the next actions of the robots in the same spirit of the one proposed by Bhattacharya et al. (2013c).

In the second part, we will develop a new algorithm to address the problem of convergence of previously proposed methods such as (Bhattacharya et al., 2013c) and (Bhattacharya et al., 2013b). This work defines a new strategy to choose the next action, such that the cost function never increases.

We also contribute in the proposition of new efficient implementation methods of our algorithms. We propose a GPU (Graphics Processing Unit) based implementation.

3.1.2 Multi-Robot Deployment in Discrete Space

While our contributions are mostly based on the continuous setup of Pimenta et al. (2008) and Bhattacharya et al. (2013c), in the implementation part we applied a discrete scheme, graph representation, to present the map and robots motion. Thus we will briefly survey some of the discrete techniques in the literature of multi-robot deployment. For a chronological order and characteristic of each method presented, please refer to Table 3.1.

Yun and Rus (2013) describes a decentralized algorithm for locational optimization on graphs. The technique creates an undirected graph from the map and computes a Voronoi tessellation on the graph. Finally, based on the optimization framework, robots move to the desired nodes.

Some works also inspired by the locational optimization framework considered the discretization of the environment by grid cells to facilitate computation in complex environments. Durham and Carli (2012) considers a discrete partitioning and coverage optimization algorithm for robots with short-range communication. In this case a discrete setup was presented, in which a discrete deployment functional is defined. The authors proved that their algorithm converges to a subset of the set of centroidal Voronoi Tessellations (CVT) in discrete formulation, named pairwise-optimal parti-

tion. Gossip communication[2] was used to allow information exchange among the agents. In the work by Bhattacharya et al. (2013c), the environment was also discretized to allow the numerical computation of the environment partition (geodesic Voronoi diagram), but in that case the context was the one of generating an approximation to the continuous setup. In the same spirit of approximating the continuous setup, Bhattacharya et al. (2013b) discretized the environment and used a graph-based approach inspired by Dijkstra algorithm (Dijkstra, 1959) to directly compute the proposed control law in an efficient manner in general Riemannian manifolds with boundaries.

Among all previous approaches surveyed, the most similar to the one presented in this work is Yun and Rus (2013). The authors computed the Voronoi partitions upon an undirected graph that topologically encodes the environment. However, the authors still use the original 2D metric map to control the robots, what, similarly to all other previous strategies, makes the method dependent on precise localization. The approach proposed in the present work improves on that point since all steps of the method execute on the topological map. This highly simplifies the actual robot implementation, as will be shown in Chapter 6.

3.2 GPU Based Graph Search and CVT

Graph is one of the most useful approaches in representing topology of a system in different fields. Finding the shortest path between nodes in a graph is a fundamental problem. The problem is called Single Source Shortest Path (SSSP) when the distance between one node (vertex) to all the other nodes is considered in a weighted graph. The most popular sequential algorithm for solving this problem is Dijkstra (Dijkstra, 1959) which can run on non-negative edge weight graphs. The simplest version of the Dijkstra algorithm uses an array structure to maintain the graph, thus the complexity of it is $O(|\mathcal{V}|^2)$, where $|\mathcal{V}|$ indicates the number of nodes in the graph. Different versions of the data structure have been proposed in order to speed up the algorithm. By using

[2] A short-range communication with asynchronous and unreliable communication between nearby robots

Fibonacci heap (Cormen et al., 2001) its running time is bounded by $O(|\mathcal{V}|\log|\mathcal{V}|+|\mathcal{E}|)$, where $|\mathcal{E}|$ is the number of edges. Meyer and Sanders (2013) introduced $\Delta Step$ which ordered nodes by using bucket representation with size Δ, and each bucket may be processed in parallel. They also did a review in detail of another formulation of parallel SSSP. Madduri et al. (2007) do an efficient implementation of this algorithm for multi-thread parallel computer (Cray MTA-2). By improving hardware for parallel processing, General Purpose Graphical Processing Unit (GPGPU) has become a popular topic being a solution not only in Graphics, but also in different applications in different areas. It has high power of parallel processing with lots of threads and low price. Moreover, it was introduced a SDK, which makes programming easy for developers. CUDA was one of the most famous parallel purpose hardware (graphic card) introduced by NVIDIA (we explained the architecture in Section 2.5). Some researchers have been working on SSSP in order to implement it on CUDA. Harish and Narayanan (2007) used compact adjacency list to represent the graph and presented a fast implementation of graph search algorithms such as breadth-first search, SSSP and All-Pairs Shortest Path (APSP) in CUDA. Martin et al. (2009) proposed several solutions for SSSP based on Dijkstra. They compared Dijkstra algorithm implemented on CUDA with adjacency list and CPU, based on Fibonacci structure on different random graphs. Also, they claim that their algorithm overcomes the problem in (Harish and Narayanan, 2007) for not using the atomic function.

APSP (All Pairs Shortest Path) is another challenge in graph search, in which paths between all pairs must be found. In general, parallel algorithms for solving this problem can be classified in two categories: methods that run SSSP iteratively and methods based on Floyd-Warshall algorithm (Floyd, 1962). An example of the first category is the work of Harish and Narayanan (2007). Usually, this type of approach requires redundant computation and also large memory space. A good literature review of APSP is discussed in (Katz and Kider, 2008). Moreover, Katz and Kider (2008) adapted the original Floyd-Warshall algorithm to become a hierarchically parallel method. They run the proposed method on large graphs on a single GPU with multiple processors and multiple GPUs. The most interesting approach found so far has been proposed

by Okuyama et al. (2012). In that work, Okuyama et al. provide a parallel APSP algorithm based on the techniques proposed by Harish and Narayanan (2007) for SSSP problem. They defined N tasks, where N is the number of sources (vertex), and tried to speedup the computation by sharing the graph data between different tasks with shared memory. The present thesis proposes a modified version of SSSP based on (Harish and Narayanan, 2007). Differently from Okuyama's work, which uses N times more memory space to provide dedicated arrays to each of the N problems, our algorithm uses an array with the same size of Harish's work. The other difference is that instead of having N sources for computing the cost, we have M sources, where M is the number of robots. Thus the new method is designed for Multi-Source (robot) Shortest Path (MSSP) problem. And finally, besides computing shortest paths from sources to all nodes in the graph, we compute the Voronoi region associated to each robot. A Voronoi diagram (VD) is a fundamental geometrical structure which has been used in different applications such as: Motion planning (Sud et al., 2008), Collision detection (Sud et al., 2006), shape modeling (Alliez et al., 2008) and so on. In the following a brief review of previous works that compute VD in parallel is presented.

Hoff et al. (1999) developed an algorithm which could run on interpolation-based polygon rasterization hardware to compute discrete approximation of 2D and 3D Voronoi diagram. The necessary computation of Voronoi diagram has been reduced to finding a 3D polygonal mesh approximation to the distance function of a Voronoi site over a planar 2D rectangular grid of point samples (the space was uniformly sampled by points), where the distance function for a site gives, for any point, the distance to that site. As reported by Rong and Tan (2007), they worked on a jump flooding algorithm (JFA) for computing Voronoi Diagram in discrete 2D space. However, because of jumping behavior, it cannot be used when the connectivity of the neighbor nodes is needed in the computation. Rong et al. (2011) reviewed the literature of parallel Voronoi computation and CVT and implemented different methods in 2D by applying CUDA and tried to compare their speed. A popular method for constructing Voronoi diagrams is the sweep line algorithm (Fortune, 1987), but in spite of its simplicity and popularity, there is no effective technique to parallelize it. In a recent paper, Xin et al.

(2013) aimed to tackle this challenge by proposing a new algorithm, called untransformed sweepcircle, for computing a 2D Voronoi diagram. They showed that the new technique is more flexible and general than sweep line algorithm due to its parallel nature. This algorithm sweeps the circle by increasing its radius across the plane. At any time during the sweeping process, each site inside the sweep circle defines an ellipse composed of points equidistant from that point and from the sweep circle. The union of all ellipses forms the Voronoi diagram.

In contrast to the reports found in the literature, our algorithm presented in this work is able to work in non-convex environments. Furthermore, other approaches cannot be applied in our graph based representation because of the need of connectivity. Later, we will show how our proposed method merges graph searching and VD computation.

Chapter 4

Robot safe deployment

4.1 Introduction

As mentioned before, safety can be an important property in multi-robot deployment. This chapter further extends the works of Pimenta et al. (2008) and Bhattacharya et al. (2013c) to include this property in the problem. This is done by modeling the safe deployment as a constrained optimization problem. In this case, the robots are enforced to move along the environmental GVD as this is a roadmap commonly used to allow safe motion in path planning literature. Next section presents the proposed safe deployment modeling.

4.2 Safe Deployment

In this section we propose the modeling of the safe deployment problem by means of an optimization problem. Consider the bounded free configuration space $\mathcal{Q}_{free} \subset \mathbb{R}^2$. Let $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_M\}$ be the configuration of M robots with indexes $R = \{r_1, \cdots, r_M\}$, where $\mathbf{p}_i \subset \mathcal{Q}_{free}$. The problem to be solved is the one of finding distributed robotic actions, in the sense that only robots in the neighborhood of robot i will be taken into account, which leads the system to a local solution of the minimization problem given

by:

$$\min_{\mathbf{p}} \mathcal{H}(P, GGVD) \tag{4.1}$$

s.t.

$$\begin{cases} y_{i1}(\mathbf{p}_i) \leq 0, \ldots, y_{im}(\mathbf{p}_i) \leq 0 \\ h(\mathbf{p}_i) = 0. \end{cases}$$

where, $y_{im}(\mathbf{p}_i)$ and $h(\mathbf{p}_i)$ indicate to: the inequality constraint (for collision avoidance); and quality constrain (to keep robot moving on the GVD) respectively. The next subsections will explain the meaning of the terms used in the defined problem and from this explanation it should be clear how safety is then incorporated in the locational optimization framework.

4.2.1 New Metric

The Geodesic distance is a metric which is more realistic than Euclidean distance in non-convex environments, as explained in Subsection 2.4.1. This distance is used in the deployment functional presented by Pimenta et al. (2008) as the general function d, as defined in Chapter 2. In this case, the induced Voronoi Tessellation is the so-called geodesic Voronoi Tessellation. Now, we propose a further extension on this metric which will be called *Geodesic Distance Based on GVD*. This distance function corresponds to the length of the shortest path from two points when using a GVD as a roadmap. In Fig. 4.1, this shortest path between two points in $\mathcal{Q}_{free}$ is shown.

In general, we can divide this path into three parts: a path from the initial point to GVD ($Path_{Init_To_GVD}$), a path from a point on GVD to another point on GVD ($Path_{GVD_To_GVD}$), and a path from GVD to the goal point ($Path_{GVD_To_Goal}$). The *Geodesic Distance Based on GVD* is then defined as:

$$\begin{aligned} d(\mathbf{p}_i, \mathbf{p}_j) = \quad & \mathcal{W}_1.\|\mathbf{p}_i - \Pi_i(GVD)\| + \mathcal{W}_2.\, g(\Pi_i(GVD), \Pi_j(GVD)) \\ + \quad & \mathcal{W}_1.\|\mathbf{p}_j - \Pi_j(GVD)\|, \end{aligned} \tag{4.2}$$

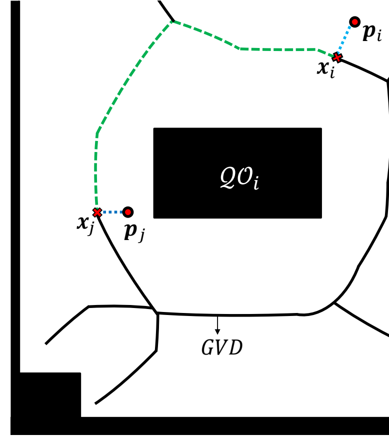


Figure 4.1: Blue dashed line is the portions of path: $Path_{Init_To_GVD}$ and $Path_{GVD_To_Goal}$. Green dash line belongs to GVD, $Path_{GVD_To_GVD}$.

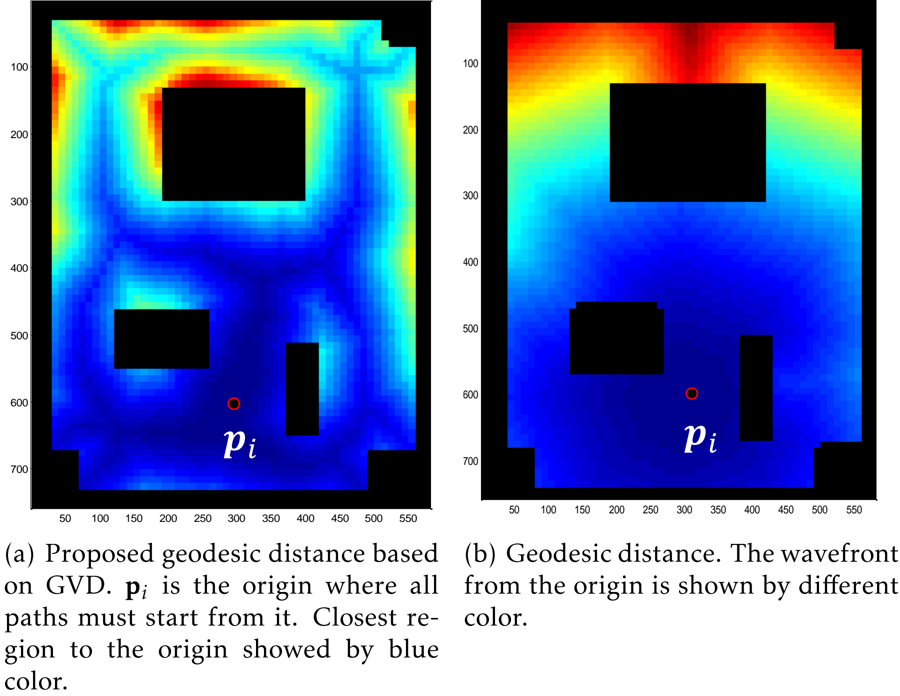
where $g(x_i, x_j)$ gives the shortest distance between two points x_i and x_j on the GVD, if the motion is constrained to remain on the GVD, $\Pi_i(GVD)$ represents the projection of the point $\mathbf{p}_i$ onto the GVD which corresponds to the closest point on the GVD to $\mathbf{p}_i$, and $\mathcal{W}_1$ and $\mathcal{W}_2$ are the weights of each part of the path. For example, by assigning a large value to $\mathcal{W}_1$ the cost of $Path_{Init_To_GVD}$ or $Path_{GVD_To_Goal}$ can be increased. These weights help to adjust the cost of two portions of the path so that it is worth first moving to the GVD as soon as possible and perform most of the motion traversing it. Fig. 4.2 shows the difference between geodesic distance and our new metric. The cost between source point $\mathbf{p}_i$ and points on/close to GVD is lower than the cost of path to other points.

As safety regarding the existing obstacles is related to the distance the robot keeps from them and the GVD provides a roadmap which keeps equidistance from the closest obstacles, we can say that this metric can introduce safety in the deployment solution. In the minimization problem defined in Eq. (4.1) the cost function is defined according to the new metric, d:

$$\mathcal{H}(P, GGVD) = \sum_{i=1}^{M} \int_{GGVD_i} d(\mathbf{q}, \mathbf{p}_i)^2 \phi(\mathbf{q}) d\mathbf{q}, \tag{4.3}$$

where $GGVD$ will be defined as the Voronoi tessellation induced by the new metric

and $\mathcal{W}_1 >> \mathcal{W}_2$. For the sake of clarity, Fig. 4.3 contains the result of computing



(a) Proposed geodesic distance based on GVD. $\mathbf{p}_i$ is the origin where all paths must start from it. Closest region to the origin showed by blue color.

(b) Geodesic distance. The wavefront from the origin is shown by different color.

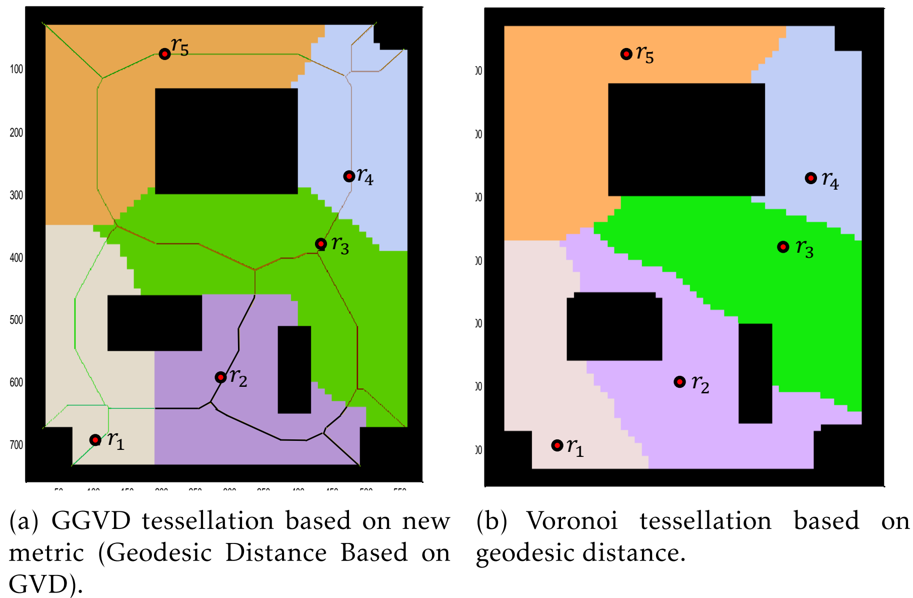
Figure 4.2: Difference between geodesic and new metric.

two different Voronoi tessellations (based on new metric and geodesic distance) on the same input map with five robots at the same positions.

Now, we can also describe the equality constraint $h(\mathbf{p}_i) = 0$ in Eq. (4.1). This function is defined as the difference between the distance functions $d(\mathbf{p}_i, \mathcal{QO}_i)$ and $d(\mathbf{p}_i, \mathcal{QO}_j)$ in which $\mathcal{QO}_i$ and $\mathcal{QO}_j$ are the closest obstacles to robot i. Thus, this means the robots must be deployed along the GVD.

4.2.2 Collision Avoidance Between Robots

Since the focus of this part of the work is on safety, besides the static obstacles we should also take into account the possible collisions between robots. A practical problem of the unconstrained minimization executed by the pure gradient-descent law by Cortes et al. (2004) is that actual robots are not point-robots. Thus, we propose to use here the same strategy presented by Pimenta et al. (2008). Basically, in this work the results for point robots are extended to robots that can be modeled as circular disks,



(a) GGVD tessellation based on new metric (Geodesic Distance Based on GVD).

(b) Voronoi tessellation based on geodesic distance.

Figure 4.3: Difference between geodesic and new metric Voronoi tessellation. Five robots are placed on arbitrary places.

each one with radius $a_i = a_j, \forall i,j$. In the following lines we will revisit the concepts defined by Pimenta et al. (2008). Consider the ordinary Voronoi diagram induced by the Euclidean distance considering the robot positions as sites. Let $\mathcal{F}_{V_i}$ be the *free Voronoi region* defined by the set of points:

$$\mathcal{F}_{V_i} = \{\mathbf{q} \in V_i \mid \|\mathbf{q} - \mathbf{q}_{\partial V_i}\| \geq a_i, \forall \mathbf{q}_{\partial V_i}\}, \tag{4.4}$$

where $\|\cdot\|$ is the Euclidean norm and $\mathbf{q}_{\partial V_i}$ is a point at the boundary of the ordinary Voronoi region induced by the Euclidean metric, ∂V_i. In this case, the boundaries of the free Voronoi regions, $\partial \mathcal{F}_{V_i}$, are hyperplanes parallel to the hyperplanes that define the boundaries of V_i, located at a distance a_i from ∂V_i. These hyperplanes ($y_{ik} = 0$) will define the linear constraints, used in the problem in Eq. (4.1). It should be clear that if all the robots have the same value of radius then safety is guaranteed as long as they remain inside their own free Voronoi region, *i.e.*, $y_{ik} \leq 0$. In Fig. 4.4, free Voronoi region $\mathcal{F}_{V_i}$, where inequality constraint is satisfied, is highlighted. Robot i also satisfies the equality constraint by lying on the GVD.

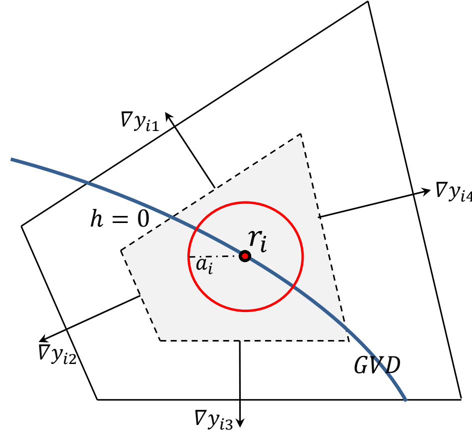


Figure 4.4: Highlighted region ($\mathcal{F}_{V_i}$) satisfies inequality constraint $y_i(\mathbf{p}_i) \leq 0$, moreover in this region blue bold line satisfies equality constraint $h(\mathbf{p}_i) = 0$.

4.3 Proposed Solution

In order to solve the safe deployment problem in an efficient manner we first present a discrete approximation of the continuous setup shown in the last section and then propose an algorithm to solve the discrete problem. The discrete setup and the proposed algorithm builds upon the work by Bhattacharya et al. (2013c), which presents a modified Dijkstra algorithm able to compute simultaneously, at each iteration, the geodesic Voronoi diagram and the robot next actions in the case of deployment on Riemannian manifolds [1] with boundaries.

4.3.1 Discrete Approximation

Consider the uniform square tiling of the 2-dimensional Euclidean configuration space. The graph $G = \{\mathcal{V}, \mathcal{E}, \mathcal{C}\}$ is induced from the uniform square tiling of the configuration space by considering an 8-connectivity neighborhood. Fig. 4.5 illustrates a graph obtained from an uniform square tiling of the free configuration space. The set of vertices (nodes) is given by $\mathcal{V}$, the set of edges by $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, and a cost function is denoted by $\mathcal{C} : \mathcal{V} \rightarrow \mathbb{R}^{+}$. It is important to mention that a node of the graph is placed in grid cells located inside $\mathcal{Q}_{free}$. Moreover, the cost of each edge is computed based

[1] A Riemannian manifold is a smooth manifold equipped with a Riemannian metric.

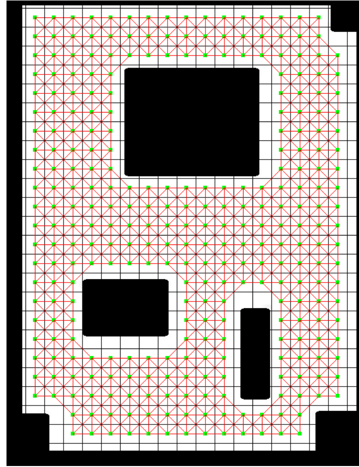


Figure 4.5: Discretization and graph representation.

on the defined new metric (Eq. (4.6)) as will be clarified later. We will also use the notation p_i to denote the node that contains the position of robot i, $\mathbf{p}_i$, and the operator $\mathbf{P}(s)$ to return the position of the center of the grid cell s. Therefore, $\mathbf{P}(p_i)$ returns the center of the cell that contains robot i. Furthermore, we will use $\mathcal{N}_G(p_i)$ as the set of graph neighbor nodes: $\mathcal{N}_G(p_i) = \{q \in \mathcal{V} \mid [p_i, q] \in \mathcal{E}\}$. Given a discrete grid based map, different techniques for computing the GVD have been proposed such as the *Brushfire* algorithm (Choset et al., 2005). In this work, we have considered a different approach which requires the input map be given as an image. The GVD is then computed by applying a skeleton operator (Santiago et al., 2011). One of the main advantages of this method is the high precision of the GVD. We compute the GVD before discretizing the environment into cells, allowing the GVD to be independent from the discretization resolution. The GVD is embedded in our graph by labeling the set of grid cells that contain a piece of the GVD as the approximate GVD, $\mathcal{V}GVD$. Now, we can define the edge cost function:

$$\mathcal{C}(i,j) = \begin{cases} \mathcal{W}_2 \cdot c(i,j), & \text{if } i\,,j\, \in \mathcal{V}GVD \\ \mathcal{W}_1 \cdot c(i,j), & \text{otherwise}, \end{cases} \tag{4.5}$$

where $c(i,j)$ is given by the Euclidean distance between the centers of the cells i and j. Since it is our objective to deploy and also move the robots along the GVD we will use $\mathcal{W}1 >> \mathcal{W}2$. For instance, we will consider $\mathcal{W}_1 = c$ and $\mathcal{W}_2 = 1$, where c is a fixed large

number. See an example in Fig. 4.6, where $c = 1000$.

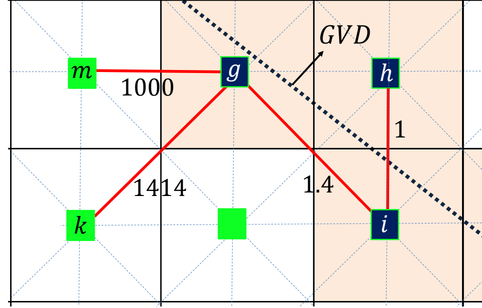


Figure 4.6: New Graph representation with 8-connectivity. The nodes g, h, and i are in the set $\mathcal{V}GVD$.

The shortest path between two vertices s and q corresponds to the sequence of nodes (consequently edges), $\{s, v_1, v_2, \ldots, v_m, p\}$, connecting this pair such that the sum of the edge costs is minimum. We will define this minimum cost sum as $d^*(\mathbf{P}(s), \mathbf{P}(q))$:

$$d^*(\mathbf{P}(s), \mathbf{P}(q)) = \mathcal{C}(s, v_1) + \mathcal{C}(v_1, v_2) + \ldots + \mathcal{C}(v_m, q), \tag{4.6}$$

This allows us to define the discrete version of the deployment functional:

$$\mathcal{H}^* = \sum_{i=1}^{M} \sum_{q \in GGVD_i^*} d^*(\mathbf{P}(q), \mathbf{P}(p_i))^2 \phi(\mathbf{P}(q)) \bar{w}, \tag{4.7}$$

where $GGVD_i^*$ corresponds to the set of grid cells so that $d^*(\mathbf{P}(q), \mathbf{P}(p_i))$ is less than $d^*(\mathbf{P}(q), \mathbf{P}(p_j))$, $\forall j \neq i$, and $\bar{w}$ is a constant related to the integral element of area. $\phi(\mathbf{P}(q))$ indicates the value of density function on the specific point q (projection of q on grid map). It should be noticed that we assume all the robots have access to this information initially. Assuming that the robots are located at the center of the grid cells, *i.e.* $\mathbf{P}(p_i) = \mathbf{p}_i$, we can compute the gradient of $\mathcal{H}^*$:

$$\frac{\partial \mathcal{H}^*}{\partial \mathbf{p}_i} = \sum_{q \in GGVD_i^*} 2\mathbf{z}_{p_i,q} d^*(\mathbf{P}(q), \mathbf{P}(p_i)) \phi(\mathbf{P}(q)) \bar{w}, \tag{4.8}$$

where $\mathbf{z}_{p_i,q}$ is the unit vector with direction given by the first edge of the shortest path between p_i and q, *i.e.* the direction of $\mathbf{P}(p_i) - \mathbf{P}(v_1)$, and magnitude given by $\mathcal{W}_2$ if

p_i , $v_1 \in \mathcal{V}GVD$ or $\mathcal{W}_1$ otherwise. Based on last equation we propose a gradient descent approach in the next subsection.

Algorithm 2: Distributed main algorithm running in robot i.

Input: G,$\mathcal{V}GVD$,ϕ,p_i
//G is the graph, $\mathcal{V}GVD$ is the approximate generalized Voronoi diagram, ϕ is the density function, $p_i \in \mathcal{V}$ is robot i initial location (graph node).
Output: p_i, o
//p_i is robot i final location and $o : \mathcal{V} \rightarrow \{1,2,\ldots,n\}$ is the discrete tessellation map $GGVD^*$ as computed by robot i.

```
1 while (Termination criteria is not met) do
2     Broadcast position p_i //Robot i sends its position to other robots.
      {p_j} ← Receive_Position_and_Neighborhood() //Receive locational
      information of neighbor robots, R_i ⊆ R, j ∈ R_i
3     R^i_VGVD ← {p_j} ∩ VGVD // Set of neighbor robots that are already on the
      VGVD.
4     if p_i ∉ VGVD then //check if the robot is not on the VGVD.
          Set the current direction of motion as the one towards the closest cell in
          VGVD which is not occupied by another robot)
      else
          Call Modified_Dijkstra(G,VGVD,φ,p_i,R^i_VGVD) //Compute both the next
          action (cell) p'_i and the GGVD* as seen by robot i.
          Set the current direction of motion to reach p'_i
5     if (There is no active inequality constraint) OR (There is an active inequality
      constraint AND current direction of motion is not obstructed by another robot
      then //collision avoidance constraint.
6         Move according to the current direction of motion
      else
7         Stop
```

4.3.2 Distributed Algorithm

In the last subsection we presented a discrete approximation (Eq. (4.7)) of the deployment functional defined in Eq. (4.3). In order to minimize this discrete version of the functional we propose to use the distributed gradient in Eq. (4.8) to determine the next action of a given robot. The gradient is distributed in the sense that only the information provided by neighbor robots is necessary.

The problem defined in Eq. (4.1) has some constraints, which means that our so-

lution should also take these constraints into account to define the next action. The collision avoidance inequalities are implemented by first verifying if any of these constraints are active, *i.e.*, $y_{ik} = 0$ for some k. If this is the case, it means there are at least two robots in the imminence of a collision, thus the involved robots will be allowed to move only if the desired direction of motion is orthogonal or has a negative projection onto the segment joining the two robot centers.

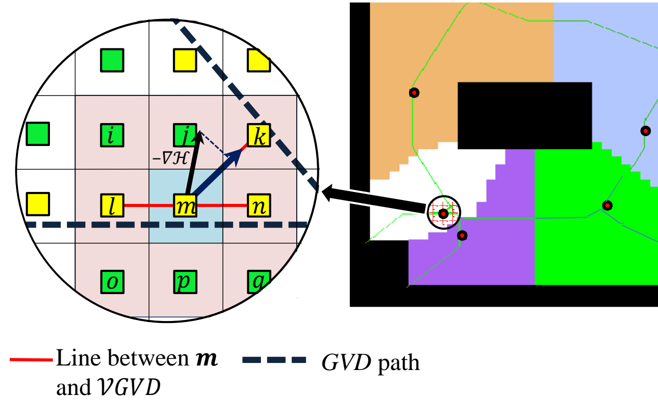


Figure 4.7: Robot is placed in the center of blue cell (m), yellow squares belong to $\mathcal{V}GVD$, thus the set $\{k,l,n\}$ corresponds to the neighbors of the robot and also part of the $\mathcal{V}GVD$.

In Algorithm 2, we assume that robots have communication with their neighbor robots in order to send and receive the locational information. Since our technique is based on a discretized grid graph representation, neighbor robots ($\mathcal{R} \subseteq R$) can be defined by:

$$\mathcal{R}_i = \{j \in R \mid \exists [xy] \in \mathcal{E}, x \in V_i, y \in V_j, i \neq j\}$$

The equality constraint which enforces the robots to be deployed along the GVD is imposed in our solution by means of two steps. As can be seen in Algorithm 2, if a robot is not in a cell that is part of the $\mathcal{V}GVD$ the next action for this robot is to move towards the closest cell in $\mathcal{V}GVD$ which is not occupied by any other robot at that time. This can be considered as the first step of the proposed approach. The second step is activated when the robot enters a cell which is part of the $\mathcal{V}GVD$. Now, the next action of this robot is a motion along a straight line from the current grid cell to a neighbor cell which is also part of the $\mathcal{V}GVD$.

The next cell is computed based on the gradient descent direction given by the negative of the expression in Eq. (4.8). In fact, we determine the next cell as the neighbor one (according to the 8-connectivity) which is also part of the $\mathcal{V}GVD$ so that the projection between the computed gradient descent direction and the direction determined by the segment joining the center of this neighbor cell and the center of the current cell is maximized (See Fig. 4.7).

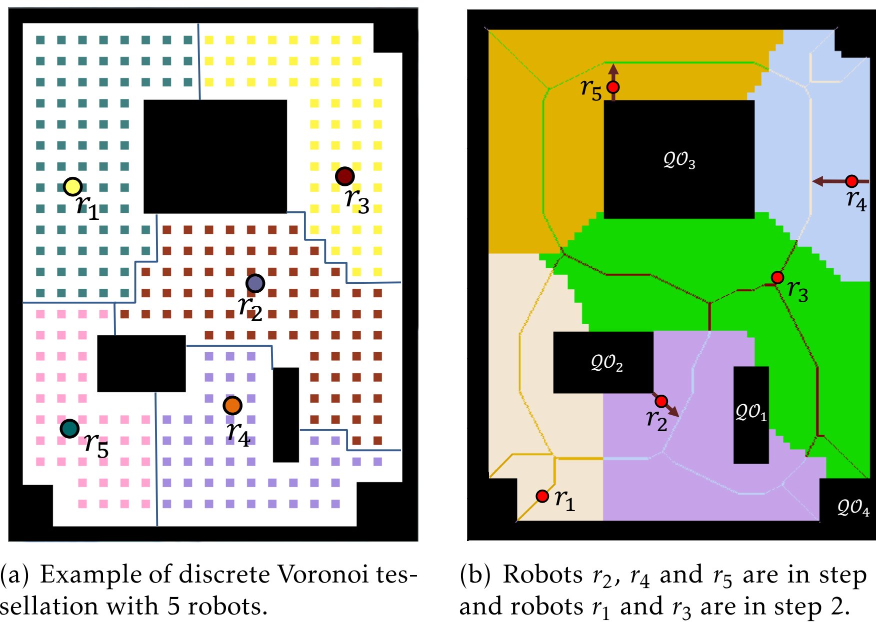
As proposed by Bhattacharya et al. (2013c), we present Algorithm 3, that computes the gradient descent direction and the Voronoi tessellation, $GGVD^*$, simultaneously in every time-step by means of a wavefront propagation procedure similarly to the process in Dijkstra's algorithm (Dijkstra, 1959). The wavefront in a given iteration represents the set of points equidistant to the start node also called source. In our case we consider wavefronts emanating from multiple sources (given by the locations of the robots). (See Fig. 4.8(a)) As the wavefronts propagate two operations are executed: (i) graph vertices in the wavefronts are associated to robots (sources) at shortest distance (according to the proposed metric) giving rise to the Voronoi regions; and (ii) terms of the summation in Eq. (4.8) associated to vertices in the wavefronts are added to a variable responsible to store the gradient descent direction. The places where the wavefronts collide determine the Voronoi boundaries. Given the asynchronous nature of the distributed implementation different robots might be in different steps at a given moment. This is not an issue since it can be easily accommodated in our approach.

Algorithm 3: Modified Dijkstra

Input: G,$\mathcal{V}GVD$,ϕ,p_i,$\mathcal{R}^i_{\mathcal{V}GVD}$ //G is the graph, $\mathcal{V}GVD$ is the approximate generalized Voronoi diagram, ϕ is the density function, $p_i \in \mathcal{V}$ is the current location of robot i (graph node), and $\mathcal{R}^i_{\mathcal{V}GVD}$ is the locations of neighbor robots that lie on the $\mathcal{V}GVD$.

Output: p_i', o // p_i' is the next cell for robot i and $o : \mathcal{V}_G \rightarrow \{1,2,\ldots,n\}$ is the discrete tessellation map $GGVD^*$ as computed by robot i.

1 Initiate d^*: $d^*(v) \leftarrow \infty$, for all $v \in \mathcal{V}$ // New metric distance.
2 Initiate o: $o(v) \leftarrow -1$, $\forall v \in \mathcal{V}$ // Voronoi tessellation.
3 Initiate η: $\eta(v) \leftarrow \emptyset$, $\forall v \in \mathcal{V}$ // robot graph vertex neighbor. $\eta : \mathcal{V} \rightarrow \mathcal{V}$
4 $\mathbf{I}_i \leftarrow \mathbf{0}$ // The gradient descent of the discrete functional.
5 **foreach** $i \in p_i \cup \mathcal{R}^i_{\mathcal{V}GVD}$ **do**
6 $\quad d^*(p_i) \leftarrow 0$
7 $\quad o(p_i) \leftarrow i$
8 $\quad$**foreach** $q \in \mathcal{N}_G(p_i)$ **do** // For each graph vertex neighbor of p_i
9 $\quad\quad \eta(q) \leftarrow q$
10 $Q \leftarrow \mathcal{V}$ // Set of unvisited nodes.
11 **while** $(Q \neq \emptyset)$ **do**
12 $\quad q \leftarrow \arg\min_{q' \in Q} d^*(q')$ // Maintained by a heap data-structure.
13 $\quad Q \leftarrow Q \setminus q$ // Remove q from Q
14 $\quad k \leftarrow o(q)$
15 $\quad s \leftarrow \eta(q)$ // The direction of one component of the gradient related to q.
16 $\quad$**if** $(s\ != \emptyset)$ AND $(k == p_i)$ **then** // Equivalently, q is not a vertex occupied by a robot and $q \in GGVD^*_i$.
17 $\quad\quad \mathbf{I}_i \leftarrow \mathbf{I}_i + \phi(q) \times d^*(q) \cdot (\mathbf{P}(s) - \mathbf{P}(p_i))$
18 $\quad$**foreach** $w \in \mathcal{N}_G(q)$ **do** // For each graph node neighbor of q
19 $\quad\quad d' \leftarrow d^*(q) + \mathcal{C}(q,w)$ //relaxation.
20 $\quad\quad$**if** $d' < d^*(w)$ **then** // If the new path to w from q is shorter than
21 $\quad\quad\quad d^*(w) \leftarrow d'$ // the previous value ($d^*(w)$),
22 $\quad\quad\quad o(w) \leftarrow k$ // update the cost value and the owner to the new one.
23 $\quad\quad\quad$**if** $(s\ != \emptyset)$ **then**
24 $\quad\quad\quad\quad \eta(w) = s$
25 $p_i' \leftarrow \arg\max_{u \in \mathcal{N}_G(p_i) \cap \mathcal{V}GVD} \frac{(\mathbf{P}(u) - \mathbf{P}(p_i))}{\|\mathbf{P}(u) - \mathbf{P}(p_i)\|} \cdot \mathbf{I}_i$ // Choose next action as the cell in $\mathcal{V}GVD$ best aligned with the gradient descent direction.



(a) Example of discrete Voronoi tessellation with 5 robots.

(b) Robots r_2, r_4 and r_5 are in step 1 and robots r_1 and r_3 are in step 2.

Figure 4.8: Discrete Voronoi tessellation and robots movement in safe deployment strategy.

Reaching the $\mathcal{VGVD}$ is the only objective of the robots which are out of the $\mathcal{VGVD}$. Thus, these robots execute step 1 until the objective is reached while the others will execute step 2 without considering the ones in step 1 in the computation of their next actions. The actions in steps 1 and 2 will be executed as long as the desired motion does not conflict with the allowed motion directions, due to the active inequality constraints. A snapshot of robots movements in a given time is presented in Fig. 4.8(b). The ideas previously discussed are organized in the form of the Algorithms 2 and 3. We consider these are the algorithms running in every robot individually.

Termination :

The commands in the while loop of Algorithm 2 are executed until termination criteria are met. We consider two criteria: (i) maximum number of iterations is reached; and (ii) a measure of convergence is less than a pre-specified threshold. This measure of convergence could be, for example, the variation in the positions of robots over the most recent m time-steps.

Complexity :

It is clear that the bottleneck of our iterative algorithm is the function described in Algorithm 3. Since this function runs exactly in the same format of the Dijkstra algorithm, the graph vertices have a constant degree, and a heap is maintained as a priority queue to store the unvisited nodes, the running time is given by $O(|\mathcal{V}|\log(|\mathcal{V}|))$ (where $|\mathcal{V}|$ is the number of vertices in the graph).

4.4 Results

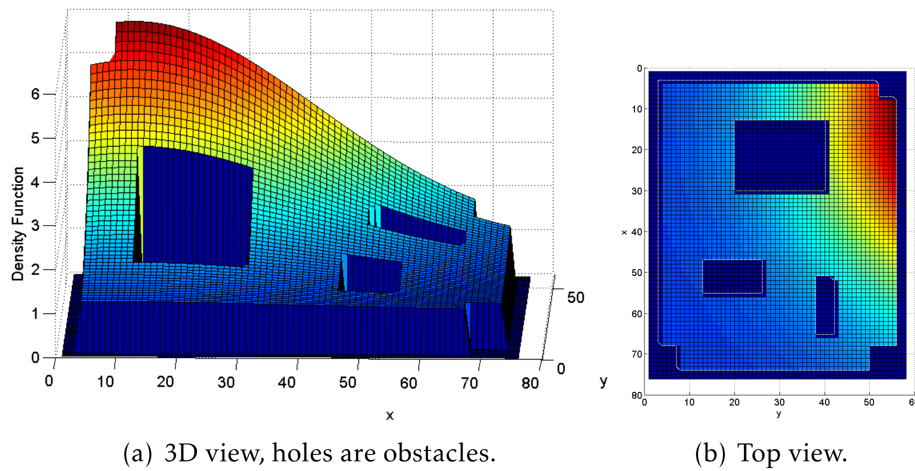
In this section we illustrate our approach by simulating the deployment of robots in two different environments. Videos are available at:
http://www.cpdee.ufmg.br//~coro/movies/RezaThesis/.

4.4.1 Simple Map

A simple room with some obstacles is the environment of our first experiment. The size of the input map is 3.79×2.91 meters, represented by an image with 728×582 pixels. By using a discretization resolution equal to 10 pixels, we have a grid map with 72×58 cells. One criterion to choose discretization rate is the size of the robot (which is 10 pixel here). The map and the density function are shown in Fig. 4.9. In this experiment five robots are considered. Fig. 4.10 shows the system evolution.

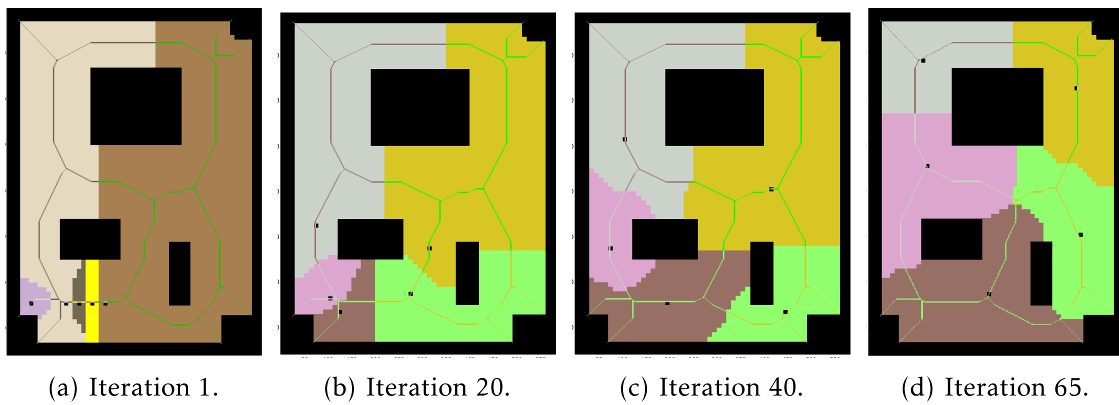
By observing robots' movements during deployment, it is evident at the beginning that two robots have a large Voronoi region. After some iterations the decrease/increase of size of large/small Voronoi regions contribute to minimize the cost function. In iteration 65 the system converges (see Fig. 4.11).

In order to show how the density function affects the final deployment we repeat the simulation with a different density function in Fig. 4.12(a). The result of this simulation with the same parameters is shown in Fig. 4.12.



(a) 3D view, holes are obstacles. (b) Top view.

Figure 4.9: Input map and density function. Color scheme denotes the density of regions, in which dark red is the center of the density function.



(a) Iteration 1. (b) Iteration 20. (c) Iteration 40. (d) Iteration 65.

Figure 4.10: Snapshots when running the proposed algorithm for five robots.

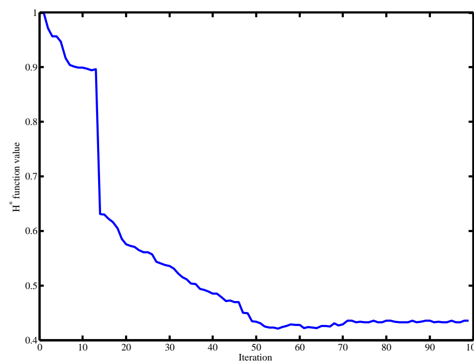
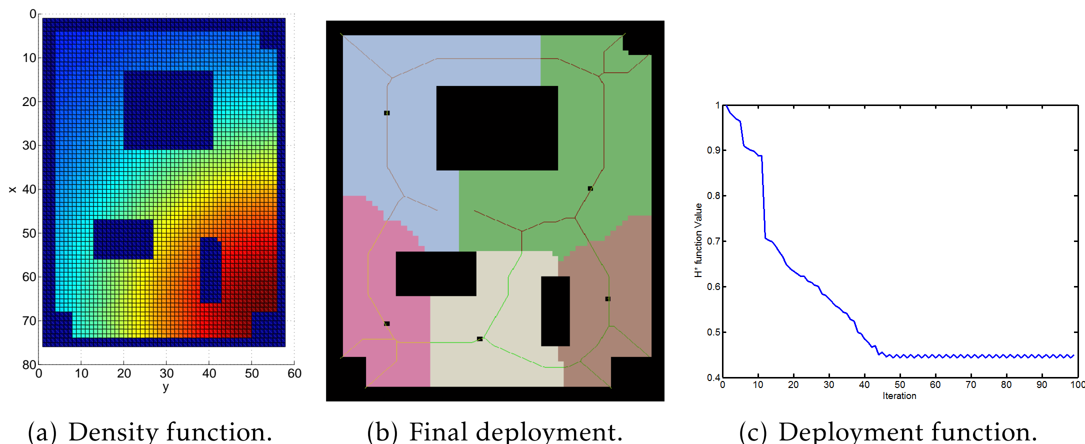


Figure 4.11: $\mathcal{H}^*$ function for the first example.



(a) Density function. (b) Final deployment. (c) Deployment function.

Figure 4.12: Simulation result with a different density function.

4.4.2 Office-like Environment

In the second experiment, the method was tested on a more complicated map with size 40.0×60.0 meters and grid graph size of 80×120 (Fig. 4.13). Initially, some of the robots are on the GVD and others are not. We set the center of the density function at the center of the map according to the top view shown in Fig. 4.14. Because of the large input map, we consider three groups of two robots. They start their movement from three different parts of the map. Fig. 4.15(a) outlines the initial positions in which robots 2, 4, 5 and 6 are on the GVD, whereas robots 1 and 3 are not. Fig. 4.15(f) shows the final positions. Fig. 4.16 shows robot trajectories and the evolution of the deployment functional, which is minimized as desired.

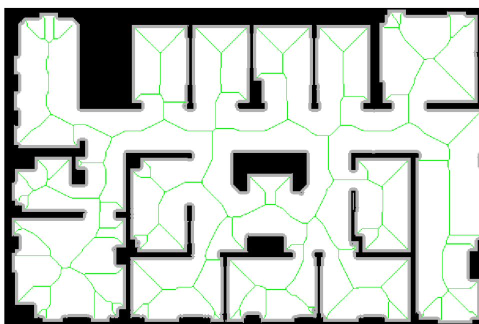
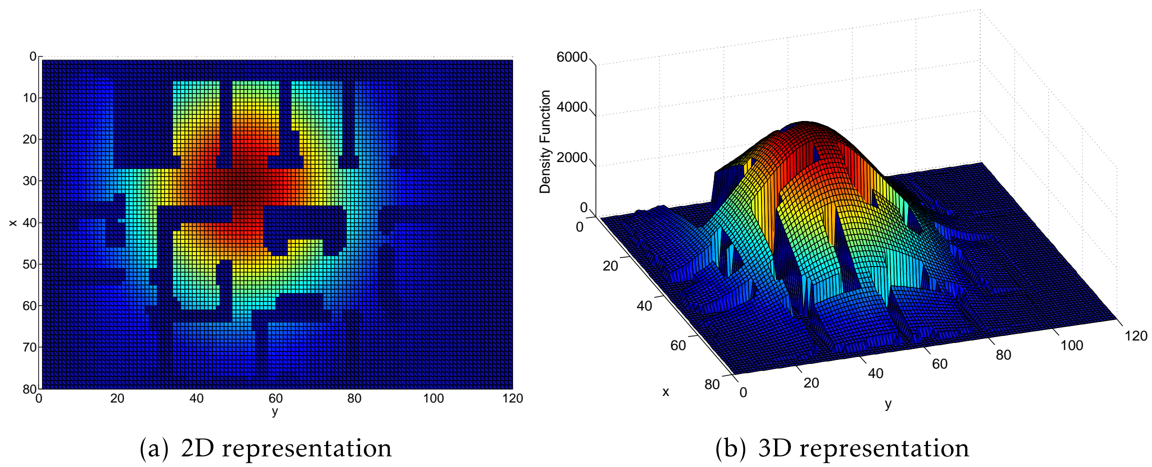


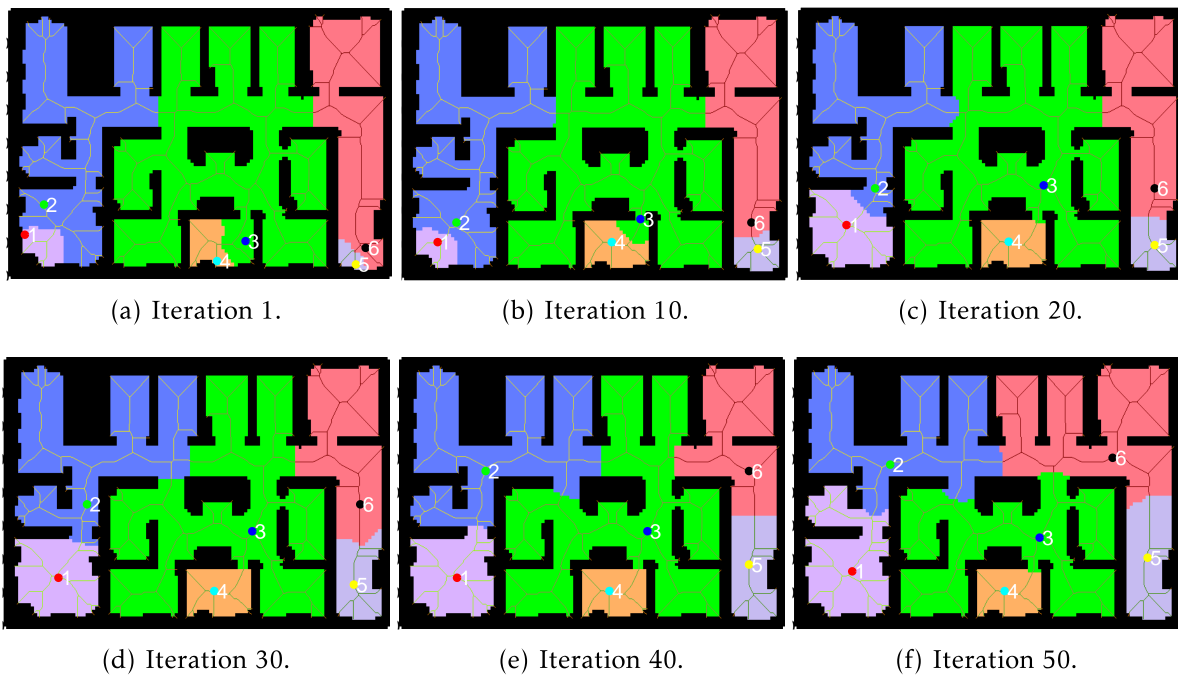
Figure 4.13: Office-like map with GVD (green lines) in free configuration space.



(a) 2D representation

(b) 3D representation

Figure 4.14: Density function has higher value at the center of the map.



(a) Iteration 1.

(b) Iteration 10.

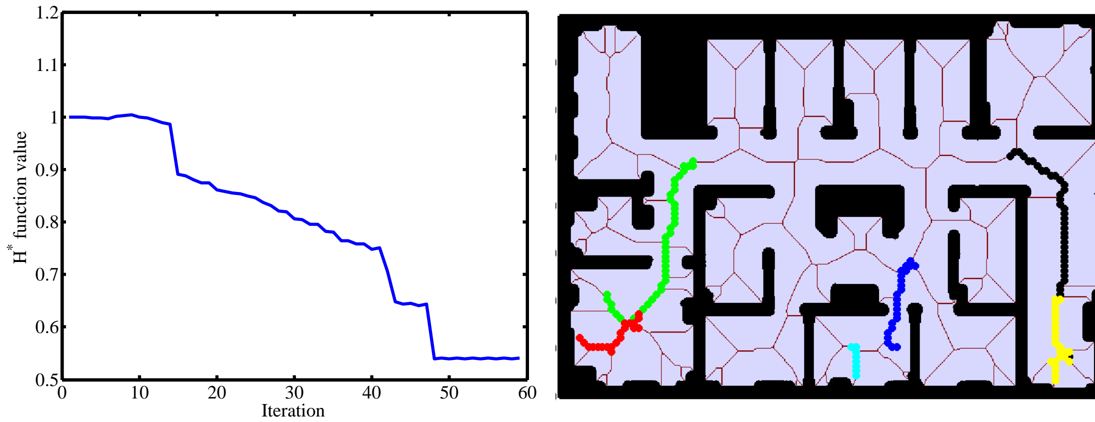
(c) Iteration 20.

(d) Iteration 30.

(e) Iteration 40.

(f) Iteration 50.

Figure 4.15: Snapshots of running the algorithm by 6 robots in 3 groups.



(a) $\mathcal{H}^*$ function converged after 50 iterations. (b) Robot trajectories in the office-like environment.

Figure 4.16: Result of running the algorithm on office-like map.

4.5 Conclusion

We addressed the problem of deriving optimal distributed control laws to deploy robotic networks in complex environments safely. The deployment problem is translated to a constrained optimization problem so that a deployment functional defined with the use of a new distance function must be minimized while satisfying constraints of two types: (i) inequality constraints for inter-robot collision avoidance, and (ii) an equality constraint to enforce the robots to be deployed at the generalized Voronoi diagram of the environment for maximizing distance from static obstacles. It is also interesting to mention that the proposed framework can also be used with other roadmaps different from the GVD. We presented a distributed algorithm which allows for efficient computation of a discrete solution for the discrete approximation of the problem. Distributed in this case means that each robot needs only the information provided by its neighbors. It is known that a discrete approximation implies in certain deviation from the real solution. However, as long as a reasonable discretization resolution is used, the loss of accuracy is compensated by the gain in computational efficiency. The proposed algorithm is based on wavefront propagation as in the Dijkstra Algorithm. The running time complexity of the proposed algorithm is given by

$O(|\mathcal{V}|\log(|\mathcal{V}|))$, where $|\mathcal{V}|$ is the number of discretization cells. The main limitation of the proposed strategy stems from the fact that neighbor robots are the ones that share Voronoi boundaries in the tessellation, independently of distance. If the necessary information between neighbors is exchanged by direct communication between robots, performance might degrade if such robots are too far one from another. Moreover, we did not prove the convergence of the proposed safe deployment, which is addressed in the next chapter.

Chapter 5

New Multi-Robot Deployment Algorithm

5.1 Introduction

As we mentioned before, one of the challenges in multi-robot deployment is to prove convergence of the deployment algorithm. In this chapter, we are going to show a new algorithm which is possible to prove its convergence based on the locational optimization framework (defined in Section 2.4). For the sake of simplicity we are just going to show the algorithm in a scenario in which we are not considering the safety as in the last chapter. However, it is easy also to use the result of the last chapter as long as we include the same constraints used in that chapter also in this problem. To guarantee the convergence, the new technique is defined in a continuous setup, even though the representation of the environment is approximated in a discrete form. In our continuous setup, robots follow the gradient vector of deployment function which leads them to converge to a local minimum. Besides convergence proof, a new parallel implementation of the algorithm is also presented which is much faster than previous ones. Further, actual robot experiments shows the applicability of the proposed algorithm in real applications.

5.2 Proposed Method

Similar to Chapter 4, in order to solve the deployment problem in an efficient manner we first present a discrete approximation of the environment, and then present the

new algorithm. The discrete setup and the proposed algorithm is built upon the work of Bhattacharya et al. (2013c) which presents a modified Dijkstra algorithm. This algorithm is able to compute the geodesic Voronoi diagram and the next robot action at the same time in each iteration. A key limitation of some of the algorithms in the literature is the lack of convergence proof. Thus, we aim to propose an algorithm with guaranteed convergence. In this chapter, we show our algorithm, its proof, and a discrete parallel implementation on CUDA.

5.2.1 Discrete Approximation and Graph Representation

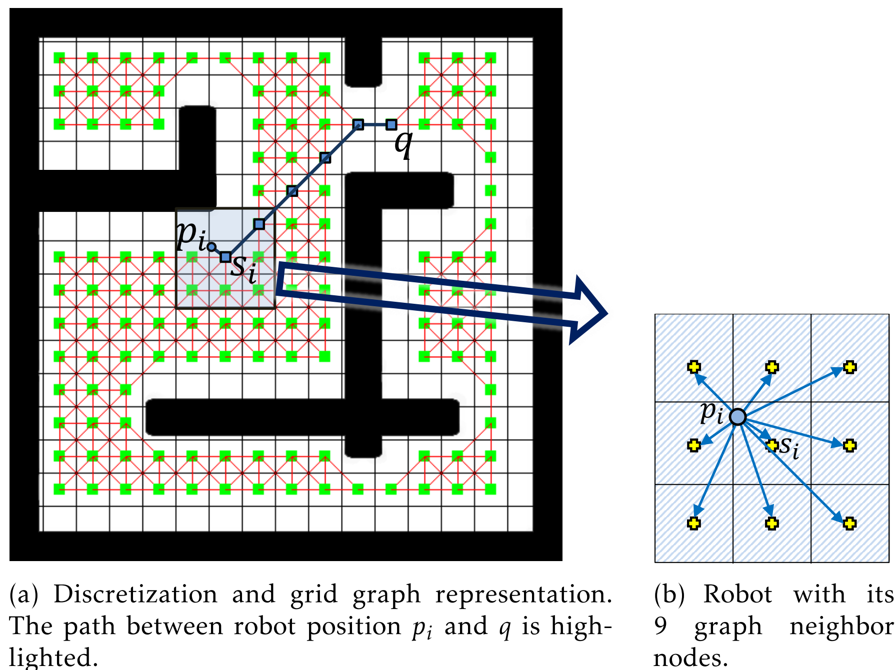
Although same notations of Chapter 4 are considered in this part, for the convenience of the reader, we have collected in Table 5.1 some of the key symbols.

Symbol	Notation
p_i	indicates the dynamic node (location of robot i) in the graph $p_i \in G$
$\mathbf{p}_i$	shows the position vector of node p_i in the environment, $\mathbf{p}_i \in \Omega$
$\mathbf{P}(s)$	gives the exact position of node $s \in G$ in the environment (center of cell)
$\mathcal{N}_G(s)$	an operator that returns graph neighbor nodes of node s

Table 5.1: Notations that will be used in this chapter.

In a bounded environment $\Omega \subseteq \mathbb{R}^2$, consider again the uniform square tiling of the 2-dimensional configuration space explained in Subsection 4.3.1. We have M robots with index set $R = \{r_1, \cdots, r_M\}$, and position $P = \{\mathbf{p}_1, \cdots, \mathbf{p}_M\}$, where $\mathbf{p}_i \in \Omega$ indicates the position of r_i. The graph $G = \{\mathcal{V}, \mathcal{E}, \mathcal{C}\}$ is built by considering the uniform square tiling of the configuration space with 8-connectivity neighborhood. However, different from similar methods in the literature, the graph G consists of two node sets: static node $\mathcal{V}_s$; and dynamic nodes $\mathcal{V}_d$, thus $\mathcal{V} = \mathcal{V}_s \cup \mathcal{V}_d$. Similarly, the set of edge and cost are defined as: $\mathcal{E} = \mathcal{E}_s \cup \mathcal{E}_d$, and $\mathcal{C} = \mathcal{C}_s \cup \mathcal{C}_d$. These dynamic nodes are associated to the *robots*, such that by moving the robots over the environment, the sets $\mathcal{V}_d$, $\mathcal{E}_d$ and $\mathcal{C}_d$ of the graph must be updated. All the other nodes ($\mathcal{V}_s$) that represent the *environment* are static. An example of this new graph representation is shown in Fig. 5.1 (a), where the position and graph neighbor nodes of a robot is depicted in Fig. 5.1 (b). It should be mentioned that in the discretization process cells that are partially occupied by obstacles are not

considered as free cells.



(a) Discretization and grid graph representation. The path between robot position p_i and q is highlighted.

(b) Robot with its 9 graph neighbor nodes.

Figure 5.1: Discrete approximation of the environment.

In general, as we mentioned graph G contains a set of vertices (nodes) given by $\mathcal{V}$ (that consists of static and dynamic nodes), the set of edges by $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$, and weights of the edges $\mathcal{C} : \mathcal{V} \rightarrow \mathbb{R}^{+}$. The weight of each edge is assigned based on the Euclidean distance between two neighbor nodes.

While in this chapter the notation $p_i \in \mathcal{V}_d$ will be used to denote the dynamic node, which is associated to robot i, the real position of robot i in the environment is posed by $\mathbf{p}_i$. The operator $\mathbf{P}(s_i)$ is used to return the position of the center of a grid cell, where s_i denotes to the corresponding node in that cell. As an example in Fig 5.1, a dynamic node p_i has the node s_i, corresponding to the center of the cell where robot i is located, and also 8 other static neighbor nodes of s_i as its own neighbor nodes. The shortest path between robot dynamic node p_i and another node of the graph, q, is defined based on a sequence of nodes (consequently edges), connecting this pair such that the sum of the edge costs is minimized (Fig. 5.1(a)). In this figure, the minimum

cost (geodesic distance) is given by $d(p_i,q,G)$:

$$d(p_i,q,G) = \mathcal{C}(p_i,s_i) + \mathcal{C}(s_i,v_1) + \ldots + \mathcal{C}(v_m,q), \tag{5.1}$$

where $v_1,v_m \in \mathcal{V}_s$, are the static nodes in the graph.

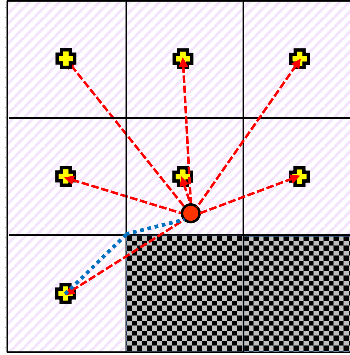


Figure 5.2: In free configuration space, red point is the robot, its neighbors are directed by red vectors. Blue dash line is the correction of geodesic distance. Checker board pattern illustrates cells with obstacle.

In our new technique due to having a dynamic node which does not follow grid based structure, a modification step is needed when the robot is very close to an obstacle. Thus the edges of the dynamic node might pass through the obstacle. This situation is presented in Fig. 5.2, where the blue dash line is the modification of the invalid edge. Since in our graph the information of nodes and weights of edges is stored, in this modification we change the weight of the corresponding edge by considering a detour around the obstacle.

Before presenting our cost function, we make an assumption about the input environment:

Assumption A. After discretization, there should not be two diagonal neighbor cells included in an obstacle, while the other two common adjacent neighbors are free. This can be achieved by not having very thin obstacles (like a line), or the size of the robots must be larger than a cell. Fig. 5.3 shows some of the valid and invalid situations.

In Fig. 5.3 (left-up) clearly the robot cannot move through the obstacle (from cell with star to the cell with plus sign). Thus, this assumption facilitates the avoidance of

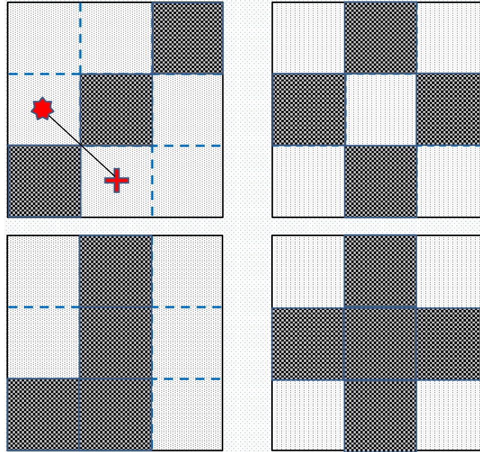


Figure 5.3: First row shows invalid obstacle shape and second row shows valid situation. Checker board pattern illustrates cells with obstacle.

considering the star cell as a neighbor of plus sign cell during graph construction.

5.2.2 Deployment Function

The deployment functional considered in this chapter is given by:

$$\mathcal{H}(P,V) = \sum_{i=1}^{M} \sum_{q \in V_i} d(q,p_i,G)^2 \phi(q), \tag{5.2}$$

where q is a node in robot i's Voronoi region V_i, $\phi(q)$ denotes the density of point q (explained in Subsection 2.4.1), and $d(q,p_i,G)$ is a function that indicates the shortest distance between q and p_i in G (Eq. (5.1)). It should be noticed that according to our new dynamic graph representation and the continuous distance function $d()$, the deployment function (Eq. (5.2)) is a continuous function, for a fixed edge set. In this case for robot i we have:

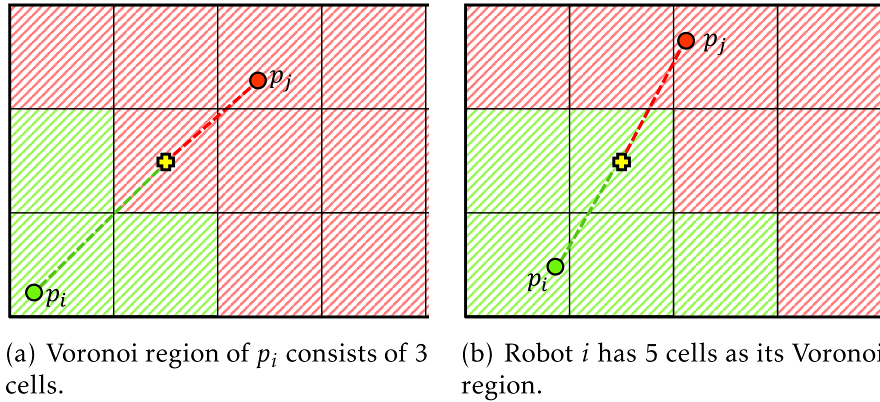
$$\mathcal{H}_i(p_i,V_i) = \sum_{q \in V_i} d(q,p_i,G)^2 \phi(q). \tag{5.3}$$

Now, we can compute the gradient of $\mathcal{H}_i$ as following:

$$\frac{\partial \mathcal{H}(P,V)}{\partial \mathbf{p}_i} = \sum_{q \in V_i} 2\mathbf{z}_{p_i,q} d(q,p_i,G)\phi(q), \tag{5.4}$$

where $\mathbf{z}_{p_i,q}$ is the unit vector with direction given by the first edge of the shortest path between p_i and q, *i.e.* the direction of $\mathbf{p}_i - \mathbf{P}(v_1)$, where $v_1 \in \mathcal{V}_s$ is a static node.

As we mentioned, V_i denotes the Voronoi region corresponding to robot i. To compute the Voronoi region the same technique of Chapter 4 is applied, so that Dijkstra algorithm is executed on the graph G to find the shortest path from the robots positions to all other nodes in the graph. But differently in the new technique the Voronoi region might change when the robot moves even inside a single cell. Fig. 5.4 contains an example of computing Voronoi regions for two robots before and after their movement. In this figure, the Voronoi regions achieved by our technique is different, and also a better approximation of continuous Voronoi tessellation (defined in Section 2.3). In other words, the technique based on discrete distance function of static nodes (similar to Chapter 5.4) will yield equal Voronoi regions for both cases in Fig. 5.4, although robots' positions have changed.



(a) Voronoi region of p_i consists of 3 cells.

(b) Robot i has 5 cells as its Voronoi region.

Figure 5.4: An example of computing Voronoi region in our new dynamic graph representation, when robot moves in the same cell.

In fact, the main objective of this chapter is to find robots configurations which minimize the deployment function in Eq. (5.2). Next section shows robots node neighbors in different conditions.

5.2.3 Robot Node Neighbors

As we explained in Subsection 5.2.1, as robots (or dynamic nodes $\mathcal{V}_d$) move over the environment, their neighbor nodes might change. Thus, we define a function

$GetCell(p_i)$ which returns the node corresponding to the center of the current cell (s_i), where the robot lies inside. Also as an operator, $\mathcal{N}_G(s_i)$ is responsible to maintain the neighbor nodes of s_i. In Fig. 5.5, an example illustrates the cells s_i corresponding to robot p_i ($GetCell(p_i) = s_i$), and its 8 neighbor nodes, $\mathcal{N}_G(s_i) = \mathcal{N}_G^{s_i}$, where $\mathcal{N}_G^{s_i}$ is the variable containing neighbor nodes of s_i.

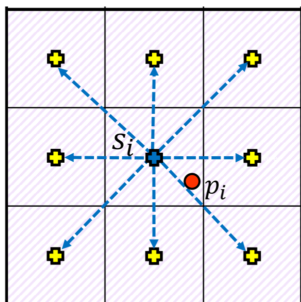
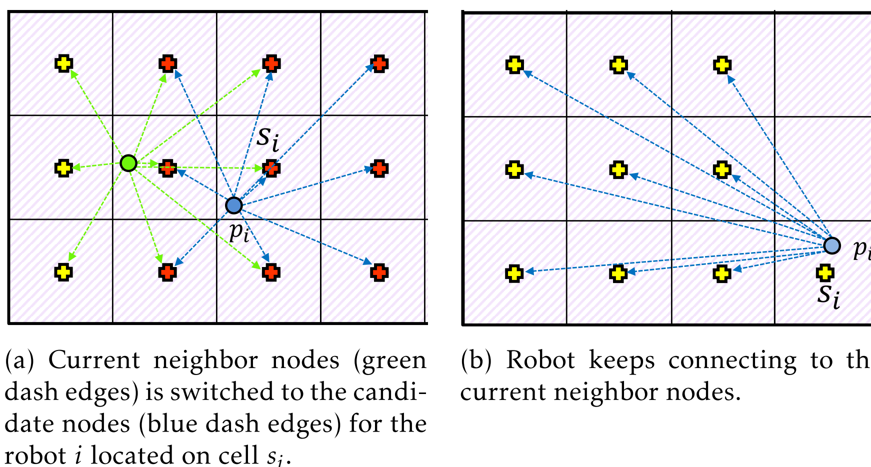


Figure 5.5: The node s_i indicates the cell where robot i is placed. It has 8 neighbor nodes obtained by $\mathcal{N}_G(s_i) = \mathcal{N}_G^{s_i}$.



(a) Current neighbor nodes (green dash edges) is switched to the candidate nodes (blue dash edges) for the robot i located on cell s_i.

(b) Robot keeps connecting to the current neighbor nodes.

Figure 5.6: Two different neighbor nodes set in the proposed technique.

Obviously, the neighbor nodes of the robots might change when they enter into new cells. In our proposed algorithm robots can have two different set of neighbor nodes when they move: current ($\mathcal{N}_{Gcurr}$) or candidate ($\mathcal{N}_{Gcand}$) set. These two situations are depicted in Fig. 5.6, where in (a) robot i (with its dynamic node p_i) enters into a new cell (s_i) and switches its neighbor nodes from *current* (green edges) to *candidate* (blue edges). In other words, after the switch the current neighbor nodes will change, and its

new neighborhood will be given by the one that were candidate before the switch. Thus in this case, the current neighbors of robot in Fig. 5.6 (a) will be $\mathcal{N}_{G^{curr}} = \mathcal{N}_G(s_i) \cup s_i$. In contrast, as the second case in Fig. 5.6 (b) robot i does not switch its neighbor nodes and keeps the current neighbors even after entering into cell s_i. Robot i decides to switch or keeps its current neighbor set, if this switching makes the value of the cost function $\mathcal{H}_i$ decreases. This condition is due to the main intention of the proposed approach which is to decrease the cost function, and is explained in the next section in detail.

5.2.4 Distributed Algorithm

In comparison to the work of Bhattacharya et al. (2013c), in the proposed algorithm, instead of using static nodes and approximating positions of the robots, we offered a continuous movement by considering dynamic nodes. We also apply a gradient descent approach–based on locational optimization framework introduced by Cortes et al. (2004)–to move the robots in the direction of minimizing the cost function.

Since the system is distributed, robots do not know about the position of other robots, except for their neighbor robots. In fact we assume that a robot has communication with its Voronoi neighbor robots, hence it can compute its Voronoi region based on its neighbor robots locations (we will not discuss about the network architecture here). Neighbor robots of robot i are defined as: $\mathcal{R}_i = \{j \in R \mid \exists [xy] \in \mathcal{E}, x \in V_i, y \in V_j, i \neq j\}$ ($\mathcal{R}_i \subseteq R$). Since the environment is discretized, the boundary between two Voronoi regions are those cells (nodes) which are neighbors of the nodes of other Voronoi regions. For the boundary nodes of two Voronoi regions V_i and V_j, if $d(q,p_i) = d(q,p_j)$, these nodes will be assigned to robot i with minimum index, in which $r_i = Min(r_i, r_j)$. After constructing the Voronoi region of robot i, V_i, gradient, and cost function of its current position, the robot follows the direction of the gradient descent vector. At the same time, it checks whether the value of cost function may decrease if the graph neighborhood changes. Such that, if the candidate neighbor nodes yield a smaller cost, robot exchanges the current neighbor nodes to the candidate one. This technique leads the algorithm to always decrease the $\mathcal{H}$ function. This property will also be used to prove

Algorithm 4: Main function in the new deployment algorithm for robot i

Input: G,ϕ,p_i, // where G is the graph, ϕ is the density function, and p_i is robot current node position.

1 $s_i \leftarrow GetCell(p_i)$ // Return the node corresponding to the cell in which robot i is inside

2 $\mathcal{N}^{p_i}_{Gcurr} \leftarrow \mathcal{N}_G(s_i) \cup s_i$ // Initial robot's first 9 current neighbor nodes

3 **while** $(True)$ **do**

4 $(\{p_j\},\{\mathcal{N}^{p_j}_G\}) \leftarrow Receive_Position_Neighbor()\ \forall j \in \mathcal{R}_i$ // Receive information from neighbor robots $(\mathcal{R}_i)$

5 $G \leftarrow Compute_graph(\mathcal{N}^{p_i}_{Gcurr} \cup \{\mathcal{N}^{p_j}_G\}, p_i \cup \{p_j\})$ // Compute the graph G based on location of neighbor robots and nodes

6 $(\nabla\mathcal{H}_i, o_i) \leftarrow MSSP(G, p_i, \{p_j\}, \mathcal{N}^{p_i}_{Gcurr}, \{N^{p_j}_G\})$// Compute gradient vector $\nabla\mathcal{H}_i$ and tessellation o_i.

7 $\mathcal{H}_i \leftarrow \sum_{q\in o_i} d(q,p_i,G)^2\phi(q)$ // Compute the cost function of the current position

8 $s_i \leftarrow GetCell(p_i)$

9 $\mathcal{N}^{p_i}_{Gcand} \leftarrow \mathcal{N}_G(s_i) \cup s_i$ // Update candidate neighbor nodes

10 $G' \leftarrow Compute_graph(\mathcal{N}^{p_i}_{Gcand} \cup \{\mathcal{N}^{p_j}_G\}, p_i \cup \{p_j\})$ // Create graph G'

11 $\mathcal{H}^*_i \leftarrow \sum_{q\in o_i} d(q,p_i,G')^2\phi(q)$ // Compute the cost function of the new graph

12 **if** $(\mathcal{H}^*_i < \mathcal{H}_i)$ **then** //Switching the neighbors

13 $\mathcal{N}^{p_i}_{Gcurr} \leftarrow \mathcal{N}^{p_i}_{Gcand}$

14 $G \leftarrow Compute_graph(\mathcal{N}^{p_i}_{Gcurr} \cup \{\mathcal{N}^{p_j}_G\}, p_i \cup \{p_j\})$ // Create graph G

15 $(\nabla\mathcal{H}_i, o_i) \leftarrow MSSP(G, p_i, \{p_j\}, \mathcal{N}^{p_i}_{Gcurr}, \{N^{p_j}_G\})$// Compute $\nabla\mathcal{H}_i$ and o_i based on updated neighbor nodes.

16 $\dot{\mathbf{p}}_i \leftarrow k.feasible_projection(-\nabla\mathcal{H}_i)$ // Find a feasible projection of gradient vector

the convergence of our algorithm.

The algorithm that runs on robot i is shown in Algorithm 4. In general the whole process in this algorithm can be divided in two main parts: computing the graph and cost function $\mathcal{H}$ based on robot's current neighbor nodes; and the same computation for the candidate neighbor nodes. Before the loop in line 2, robot's current neighbor nodes $(\mathcal{N}^{p_i}_{Gcurr})$ is initialized. Later, after receiving the locational information of robot's neighbor robots, it computes the graph G and cost value $\mathcal{H}_i$ in lines 5 and 7. Lines 8 to 11 are for computing the graph G' and cost $\mathcal{H}^*$ for the candidate neighbor nodes. Besides the difference of these two sets of computations based on the neighbor nodes, the

same Voronoi tessellation is used for both. Multiple Source Shortest Path algorithm (MSSP) is applied to compute both Voronoi tessellation (o_i) and the gradient vector ($\nabla\mathcal{H}_i$) of r_i. It is important to note that both outputs of the function $MSSP()$ are computed at the same single run of Dijkstra exploration on the graph; this is explained in Algorithm 5. The function "*Compute_Graph*()" plays a vital role due to creation of the graph based on the motion of the robot and its neighbors. In this way, it considers edges and weights of the dynamic nodes based on its current position and neighbor nodes in graph G.

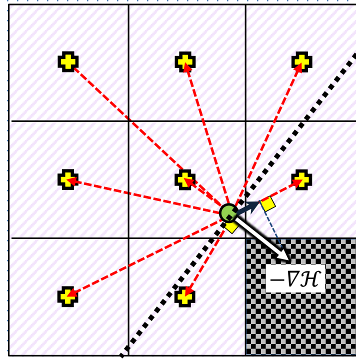


Figure 5.7: Green point is the robot, its neighbors are directed by red vectors. Black dash line is the half plane which defines the valid region for projection. Dark blue vector shows the projected $-\nabla\mathcal{H}$. Checker board pattern illustrates cells with obstacle.

To guarantee that robot motion always decreases the $\mathcal{H}$ function, we finally check if $\mathcal{H}_i^*$ is less than the last $\mathcal{H}_i$ (in line 12), and exchange the current neighbor nodes to the candidate one if the condition is accomplished (line 13).

Before robot i moves based on the gradient vector, in line 15 the new Voronoi partition and gradient vector is computed based on the updated neighbor nodes. Finally, in line 16, the feasibility of the gradient vector are checked. If the computed gradient direction guides the robot to move into a cell which is occupied by an obstacle, a modification in the direction must be imposed to avoid collision. We project the vector $-\nabla\mathcal{H}$ onto the closest valid vector which connects the robot to its graph neighbor (See Fig. 5.7). It should be mentioned that this vector is called valid only if it has non-negative projection onto $-\nabla\mathcal{H}$. This constraint guarantees that the cost function will not be increased. If non-point robots are considered then we just need to incorpo-

rate also the same linear constraints we had in Chapter 4. When two robots are about to collide we must consider only the motion directions which are orthogonal to the two robots. As before, this should be a direction with non-negative projection onto $-\nabla\mathcal{H}$. If this is not possible the robot will stop. In general, we assume that function "$feasible_projection()$" finds a valid direction to be followed, i.e., directions that avoid collision and also decrease the cost function. Remember that variable p_i always indicates the corresponding dynamic node of robot i, which is located on exactly the real position of robot $\mathbf{p}_i$. So it is always changed based on the robot movement. However the velocity vector $\dot{\mathbf{p}}_i$ will be applied on the robot and makes the robot to move in the direction of the gradient descent vector.

In order to implement *MSSP*, a technique similar to the one proposed by Bhattacharya et al. (2013a) and applied in the previous section is used. In their implementation, the key idea was to make a basic modification to Dijkstra algorithm that enables them to create a geodesic Voronoi tessellation. Furthermore, for speeding up a priority queue is applied. The complete algorithm of $MSSP$ is shown in Algorithm 5. Next, we present the convergence proof.

Proof of Convergence

At first, we show some lemmas and definitions.

Lemma 5.1. By considering the deployment function $\mathcal{H}$ from Eq. (5.2), we redefine $\mathcal{H}$ on an arbitrary partition W which is not Voronoi tessellation (Subsection 2.3) as:

$$\mathcal{H}(P,W) = \sum_{i=1}^{M} \mathcal{H}_i(p_i,W_i), \tag{5.5}$$

and for a component i we have:

$$\mathcal{H}_i(p_i,W_i) = \sum_{q\in W_i} d(p_i,q,G)\phi(q).$$

The following inequality is valid:

$$\mathcal{H}(P,V) \leq \mathcal{H}(P,W).$$

Proof. In order to prove this lemma, we refer to the Proposition 3.1 in (Du et al., 1999). According to the definition of $\mathcal{H}(P,V)$ in Eq. (5.2) and $\mathcal{H}(P,W)$ in Eq. (5.5), if we apply our new distance function from Eq. (5.1) in the definition of Voronoi tessellation (Eq. (2.5)), we can write:

$$d(p_i,q)\phi(q) \leq d(p_j,q)\phi(q),$$

where $q \in V_i$ (V_i is a Voronoi partition), and $q \in W_j$ is the same point, where W_j is a non Voronoi partition, which leads us to have:

$$\mathcal{H}(P,V) = \sum_{i=1}^{M} \sum_{q \in V_i} d(p_i,q)\phi(q) \leq \sum_{i=1}^{M} \sum_{q \in W_i} d(p_i,q)\phi(q),$$

And finally:

$$\mathcal{H}(P,V) \leq \mathcal{H}(P,W).$$

■

Definition 5.1. In the continuous multi-robot system described in this chapter, let $S(G,P)$ be our state pair where G is the graph and P is:

$$P = \begin{bmatrix} p_x^1 \\ p_y^1 \\ \vdots \\ p_x^M \\ p_y^M \end{bmatrix},$$

where p_x^i is the x component of robot i position.

Algorithm 5: $MSSP()$

Input: G,p_i,$\{p_j\}$,$\mathcal{N}_G^{p_i}$,$\{N_G^{p_j}\}$ // G is the updated graph, p_i denotes the dynamic node of robot i, and $\{p_j\}$ contains the node position of neighbor robots. $\mathcal{N}_G^{p_i}$,$\{N_G^{p_j}\}$ are the corresponding graph neighbor nodes.
Output: $\nabla\mathcal{H}_i$ and o // $\nabla\mathcal{H}_i$ is the gradient vector, and o is the discrete Voronoi tessellation.

```
1  Initiate C: C(v) ← ∞,    for all v ∈ 𝒱  // Geodesic distance map.
2  Initiate o: o(v) ← −1, ∀v ∈ 𝒱  // Tessellation.
3  Initiate η: η(v) ← ∅,  ∀v ∈ 𝒱 // robot graph neighbor nodes. η : 𝒱 → 𝒱
4  ∇ℋ_i ← 0  // The gradient vector of the discrete functional.
5  foreach  k ∈ p_i ∪ {p_j} do
6      C(p_k) ← 0
7      o(p_k) ← k
8      foreach  q ∈ 𝒩_G^{p_k}  do  // For each graph vertex neighbor of p_k.
9          η(q) ← q
10 Q ← 𝒱_s  // Set of unvisited static nodes.
11 while (Q ≠ ∅) do
12     q ← argmin_{q'∈Q} C(q')  // Maintained by a heap data-structure.
13     Q ← Q \ q  // Remove q from Q
14     k ← o(q)
15     s ← η(q) // The direction of one component of the gradient related to q.
16     if (s != ∅) then  // Equivalently, q is not a vertex occupied by a robot.
17         ∇ℋ_i ← ∇ℋ_i + ϕ(q) × C(q) · (p_i − P(s)) // p_i is the position vector.
18     foreach w ∈ 𝒩_G^q do  // For each graph vertex neighbor of q
19         C' ← C(q) + 𝒞(q,w)  //relaxation.
20         if C' < C(w) then
21             C(w) ← C'
22             o(w) ← k
23             if  (s != ∅) then
24                 η(w) = s
```

Theorem 5.1. Given a multi-robot system as described in this section with robots running Algorithm 4, the state pair S (defined in Definition 5.1) will converge to a local minimum of the cost function $\mathcal{H}$ (Eq. (5.2)).

Proof. According to Algorithm 4 the value of $\mathcal{H}$ changes due to three reasons: i) motion given by a gradient descent controller; ii) switch of the graph edges associated with the dynamic nodes which represent the robots; iii) computation of a new Voronoi tessellation. Thus, based on Lemma 5.1 and lines 12-15 of the algorithm we can state that $\mathcal{H}$ is a decreasing function. Furthermore, $\mathcal{H}$ is lower bounded ($\mathcal{H} > 0$) and piecewise continuous, as for a fixed graph G (fixed edge set for the dynamic nodes) the function is continuous, and it might have discontinuity when the graph edges switch.

As $\mathcal{H}(t)$ is decreasing and lower bounded, we can conclude that $\mathcal{H}$ converges to a value $\mathcal{H}_{min}$ as $t \to \infty$. Since $\mathcal{H}$ is continuous in p_i when G is fixed and G changes only if the switch causes a decrease in the value of $\mathcal{H}$ ($\mathcal{H}_i^* \leq \mathcal{H}_i$ in Algorithm 4), then there will exist a finite time T so that G remains constant $\forall t > T$. If this was not true we would conclude that $\mathcal{H}$ would not converge which would be a contradiction. Therefore, after T, our algorithm behaves exactly as a traditional gradient descent algorithm which allows us to conclude that the multi-robot system will converge to a local minimum of $\mathcal{H}$.

■

Computational Complexity

In Algorithm 4, the major burden of our computation is on $MSSP()$ function. Similar to the complexity of Dijkstra algorithm implemented on priority queue, the complexity of our method is given by $O(|\mathcal{V}|\log|\mathcal{V}|)$. Furthermore, we have some additional computation for computing $\mathcal{H}_i$ (and updating the graph), in which we can consider a linear complexity or $O(|\mathcal{V}|)$. By considering these two parts we have $O(|\mathcal{V}|\log|\mathcal{V}|)+O(|\mathcal{V}|)$, which can be considered as $O(|\mathcal{V}|\log|\mathcal{V}|)$, in general.

In order to improve the computational time of our method, we implement this it in a parallel way on CUDA hardware. In the next section our proposed implementation is explained.

5.3 Discrete Implementation on GPU

Since the function *MSSP()* is the bottleneck of the proposed approach, we decided to implement it in an efficient manner. This section explains how to implement it in a GPU. The main objective of parallel implementation is to increase speed.

5.3.1 GPU Based Multi-Source Dijkstra

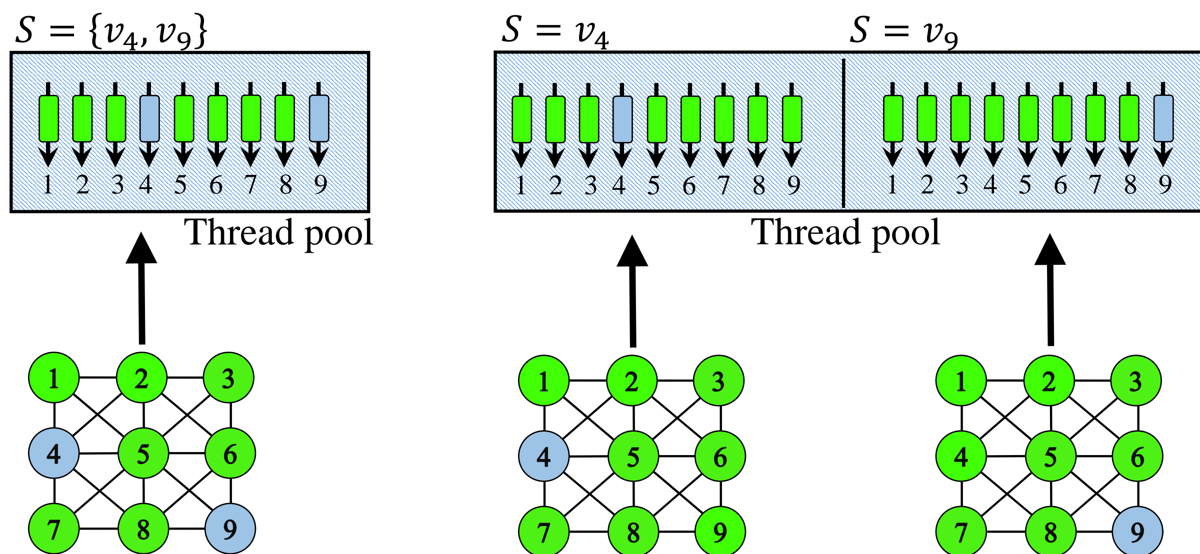


Figure 5.8: Left: thread pool of proposed method with 2 sources. Right: thread pool of method proposed in Okuyama et al. (2012) with 2 sources.

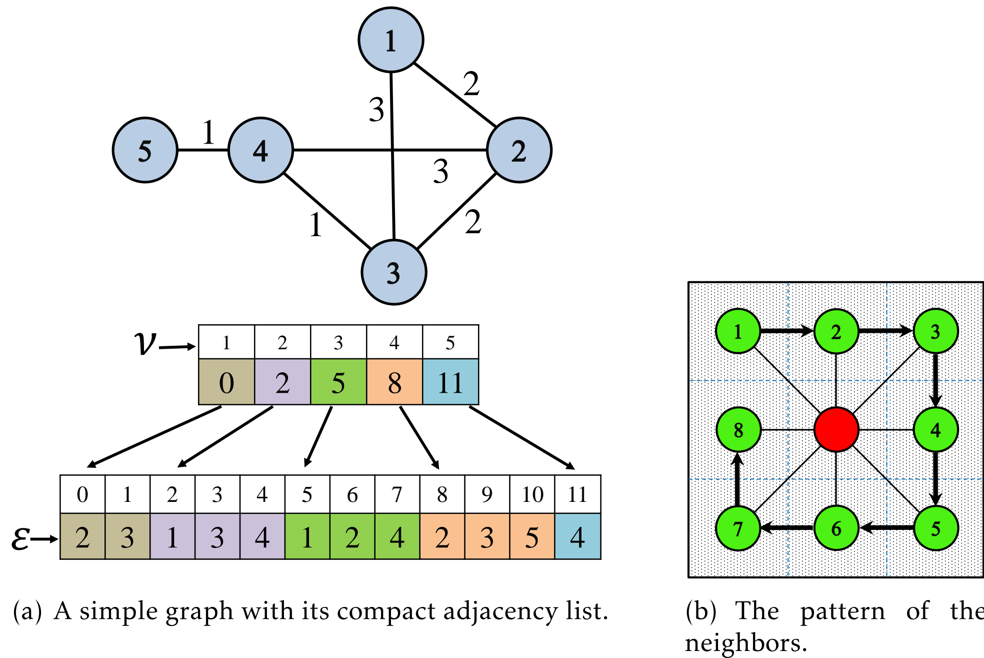
In this work a modified version of the parallel Single Source Shortest Path (SSSP) Dijkstra's algorithm is proposed. The new parallel method is designed for Multi Source (robot) Shortest Path (MSSP) problem in undirected graphs and is able to run on CUDA. In general, this algorithm performs three important tasks: finding shortest distance map from the sources to all points on the map, computing the control integral to obtain the gradient vector (Eq. (5.4)), and also label cells of the map to show the corresponding Voronoi tessellation (o_i) of each robot.

We have reviewed the literature of SSSP and APSP in Chapter 3, Section 3.2, and have discussed why none of them can be used directly in our problem. Our modified algorithm is based on SSSP which was proposed firstly by Harish and Narayanan

(2007). Differently, in our application, there are more than one source (robot), thus the naive method can run SSSP algorithm M times, where M is the number of sources. The main disadvantage of this method is the high complexity that is $M \times O(N \log(N))$ (N is number of nodes).

In comparison to this simple method, our proposed algorithm runs SSSP once with the same complexity and memory space of the original parallel SSSP algorithm by Harish and Narayanan (2007). In the recent work, a new implementation of APSP (All Pair Shortest Path) by Okuyama et al. (2012), which is based on (Harish and Narayanan, 2007), on directed graph, is proposed. The major drawback of this approach is the memory space and the number of threads, $N \cdot |\mathcal{V}|$, where N is the number of SSSP problems that have to be solved. Therefore, their kernel cannot deal with larger graphs as compared to the original kernel even though it has an advantage over the parallel implemented Floyd-Warshall algorithm by Venkataraman and Mukhopadhyaya (2003). The output of this algorithm is the distance to each node from nodes $1, \cdots, N$. Therefore, the size of output distance array will be $N \times N$. This makes a difference, in comparison to our application, because here a single distance map with size N is needed to show which source has the minimum distance to discrete cells (nodes in graph). Another straightforward solution to our problem can be the method of Okuyama et al. (2012). However, instead of running N sources, we set M sources (robots). Also, after running the algorithm we need one extra operation to merge all distance maps together to make a unique map, which is the desired output in our application. But it still needs $M \cdot |V|$ threads and memory space. To overcome this problem, in our new method, the size of memory and the number of threads which we need to compute the distance map is the same as SSSP illustrated in the original work of Harish and Narayanan (2007), $|V|$ (See Fig. 5.8). In Fig. 5.8, nodes 4 and 9 are the sources. As shown, the new method needs much less memory and threads than previous method.

The graph $G = \{\mathcal{V}, \mathcal{E}, \mathcal{C}\}$ is represented by means of a compact adjacency list which is very useful in the parallel algorithm. In Fig. 5.9(a), we show a good example of a simple graph with the corresponding structure. Each entry in the vertex array $\mathcal{V}$ shows the starting index of its adjacency list in the edge array $\mathcal{E}$.



(a) A simple graph with its compact adjacency list.

(b) The pattern of the neighbors.

Figure 5.9: Compact adjacency list and the pattern.

Based on our grid graph representation, we modified this compact adjacency list in a well structured way. The map is discretized into cells, thus for each node inside the cell there are at most eight neighbor nodes. It is remarkable to mention that, to have a uniform structure we consider 8 cells for all nodes (even nodes with less than 8 neighbors). We define a pattern which represents the order of neighbors when they convert to the adjacency compact list (See Fig. 5.9(b)). Indeed, we redefined the structure to be applicable in multi-source problem. It should be mentioned that, in the case of a cell where the robot is placed in it, we will have 9 vertex neighbors for this dynamic node (robot). However, our new parallel implementation gives an advantage of having a fixed graph when running the algorithm. Hence, we always have 8 neighbors for each node in the graph and adjacency list. In the next subsection we will show how the new algorithm carries this benefit.

Fig. 5.10 shows an example related to our representation. Nodes 1,2 and 4 belong to an obstacle, but due to the uniform array, those nodes are considered with weight '-1' on their connected edges to other nodes (3,5,6,7,8). In other words, the length of the weight array will be 8 $\times|\mathcal{V}|$, where $\mathcal{V}$ is the number of nodes or cells in the map.

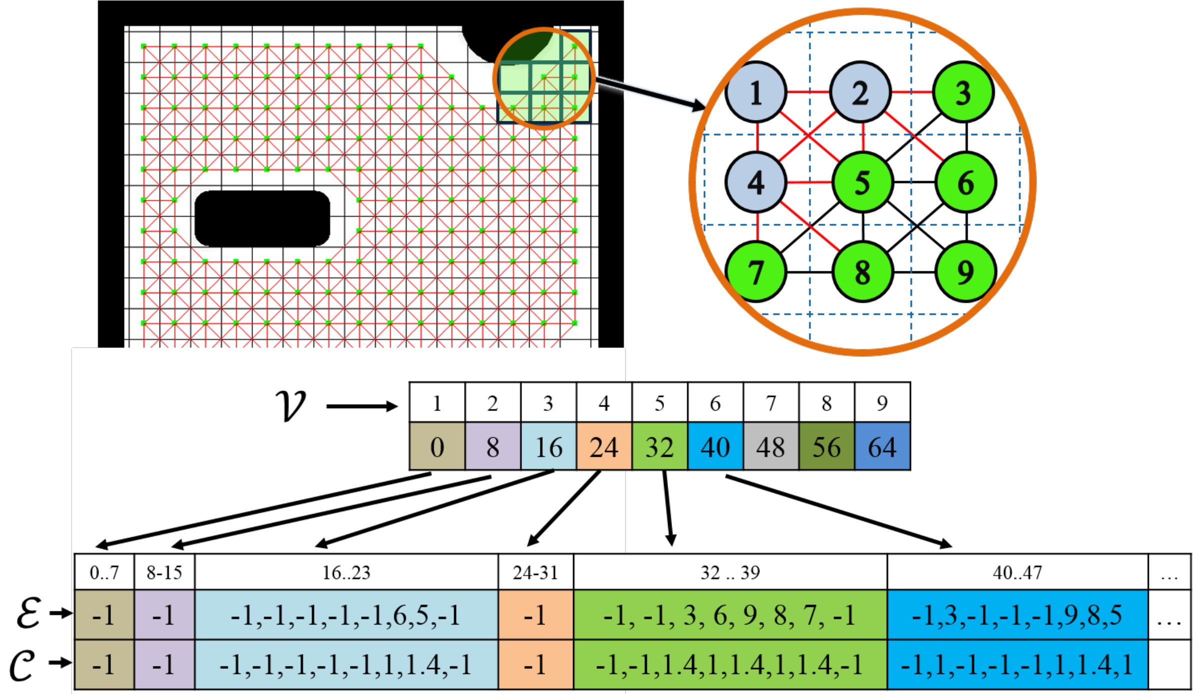


Figure 5.10: Customized compact adjacency list. The red edges have -1 as weights

Suppose there is a discretized map with N ($rows \times cols$) cells and M robots. Therefore, graph $G = \{\mathcal{V},\mathcal{E},\mathcal{C}\}$ has three components, $\mathcal{V}$ of size N, $\mathcal{E}$ being the edges between nodes and $\mathcal{C}$ being the cost between nodes with the size of $|\mathcal{E}|$. After constructing the required arrays, in order to visit all nodes in our grid graph, Dijkstra algorithm and wavefront technique are used by starting from robots' initial positions. In the next section we will show how the parallel algorithm is implemented.

5.3.2 Parallel MSSP algorithm

We use CUDA based Dijkstra Single Source Shortest Path (SSSP) algorithm, as the basis of MSSP Algorithm 6 shows the pseudo-code. In comparison to Algorithm 5, in the parallel version first we initialized the corresponding value of the node that robot i is placed in its cells (lines 7-10). Also in lines 12 to 16 the mask array (Ma), Cost array Co, tessellation map (o_i) and the first vector of gradient (η) are set. In the loop in line 17 to 20, kernel 1 and 2 refer to the functions that will return the gradient and tessellation, and will be running in the device memory in a parallel way.

Algorithm 6: MSSP_CUDA

Input: G,ϕ,p_i,$\{p_j\}$ // G is the updated graph, ϕ is density function, p_i denotes the position of robot i, and $\{p_j\}$ contains the position of neighbor robots.

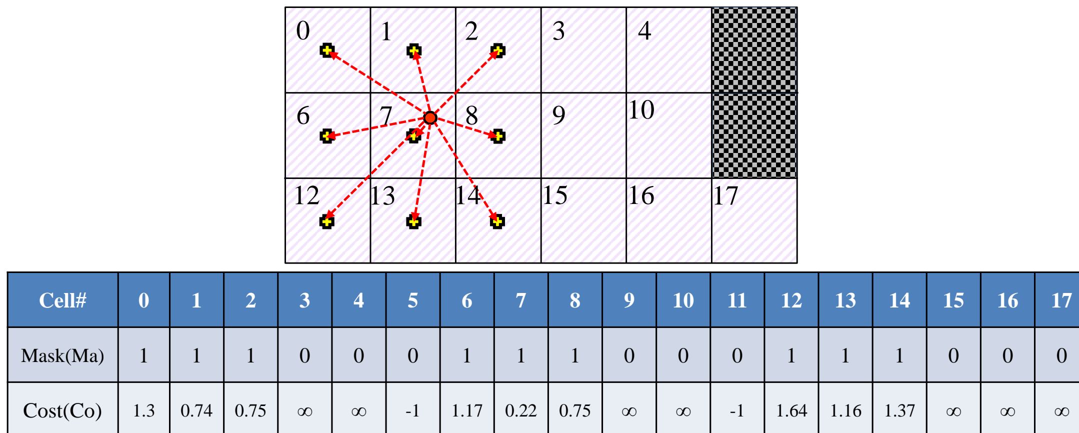
Output: $\mathbf{I}$ and o.

1 Initiate $Ma : Ma(v) \leftarrow$ false, $\forall v \in \mathcal{V}$ // mask array
2 Initiate $Co : Co(v) \leftarrow \infty$, $\forall v \in \mathcal{V}$, // Cost array
3 Initiate $Uc : Uc(v) \leftarrow \infty$, $\forall v \in \mathcal{V}$, // Updating Cost array
4 Initiate $o : o(v) \leftarrow -1$, $\forall v \in \mathcal{V}$, // Owner of the nodes, equal to the corresponding tessellation
5 Initiate $\eta : \eta \leftarrow -1$, $\forall v \in \mathcal{V}$, // The direction of the gradient vector
6 **foreach** $p_k \in p_i \cup \{p_j\}$ **do**
7 $s_k \leftarrow GetCell(p_k)$ // $GetCell()$ returns the graph node index where robot is placed in it
8 $Ma(s_k) \leftarrow true$
9 $Co(s_k) \leftarrow Cost(p_k,s_k)$// Returns Euclidean distance between two inputs
10 $o(s_k) \leftarrow k$
// The node belongs to the region of robot k
11 $\mathbf{I}_k \leftarrow \mathbf{0}$ // The control integral
12 **foreach** $q \in \mathcal{N}_G(s_k)$ **do** // For each neighbor of s_k
13 $Ma(q) \leftarrow true$
14 $Co(q) \leftarrow Cost(p_k,q)$
15 $o(q) \leftarrow k$
16 $\eta(q) \leftarrow q$
17 **while** $Ma \neq \emptyset$ **do** //Until no more unexplored node remains
18 **foreach** $\forall v \in \mathcal{V}$ *in parallel* **do**
19 KERNEL1(G,Ma,Co,Uc,η,ϕ,o)
20 KERNEL2(Ma,Co,Uc)

In general, the algorithm starts checking the nodes of the current positions of the robots and their neighbor nodes in lines 6 to 16, and continue exploring the graph by propagating to other nodes in parallel (lines 17 to 20). In this algorithm, the mentioned new variables are defined as following:

- Co as the cost of the distance (cost map) between robots and all nodes in graph with size N.
- Uc is a temporary memory for cost updating.
- Ma is a mask of size N for controlling wavefront that starts from robots' position and propagates to other cells.

For a better understanding of the usage of Mask array, Ma, Fig. 5.11 shows a robot which is placed inside cell number 7. As it can be seen, it has 9 neighbors ({0,1,2,6,7,8,12,13,14}). To avoid changing our graph representation which has a fixed size $8 \times N$ (8 neighbors for each node), Mask array (Ma) is used. This array is initialized with "1" in its elements corresponding to 9 neighbor nodes of the robot. Therefore, the dynamic node with all its neighbors can be represented in our fixed graph. Besides that, the cost of the first 9 neighbors is computed too. Based on this initialization, for the purpose of launching MSSP function in CUDA, we just need the adjacency list which was calculated in the discretization step. Kernel 1 and kernel 2 are called in a loop by using these input parameters.



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Mask(Ma)	1	1	1	0	0	0	1	1	1	0	0	0	1	1	1	0	0	0
Cost(Co)	1.3	0.74	0.75	∞	∞	-1	1.17	0.22	0.75	∞	∞	-1	1.64	1.16	1.37	∞	∞	∞

Figure 5.11: Red point is robot with 9 vertex neighbors. Correspondingly, in mask (Ma) 9 elements are set to 1 with their costs.

The termination condition is met when there is no more element in the mask array (Ma) with value '1', to continue the propagation.

As mentioned in Section 2.5, kernels will be launched in GPU with all codes inside in parallel way. In this implementation, the number of threads which is needed for each launching, is equal to the number of '1' elements in mask array. In Algorithm 7, it is shown how the parallel multi-source shortest path is implemented.

In line 2, the logical command "if" allows those tids to continue, if their corresponding value in Mask array is equal to '1'. For example, at first call, just tids with index of the neighbor nodes of the robot will be passed to the remaining part of the

Algorithm 7: KERNEL1 (G,Ma,Co,Uc,η,ϕ,o)

```
1  tid ← getThreadID;
2  if (Ma(tid)) then
3      foreach nid ∈ N_G^tid do // For each neighbor of tid
4          if (nid != o(tid) and nid != −1) then
5              if (Uc(nid) > Co(tid) + C_G(nid)) then //Atomic operator
6                  Uc(nid) ← Co(tid) + C_G(nid);
7                  o(nid) ← o(tid);
8                  if (η(nid) ≠ −1) then // Modifiying I, if the node is being visited
                   for the second time
9                      I_o(nid) −= I^n_o(nid) ;
10                 η(nid) ← η(tid);
11                 I^n_o(nid) = Uc(nid) ∗ φ(nid) ∗ (Pos(o(nid)) − P(η(nid))) ;
12                 I_o(nid) += I^n_o(nid) ;
       Ma(tid) ← false;
```

code. Line 4 plays a crucial role to avoid considering the visited nodes which were ancestor of *tid*, and also neighbor nodes that occupied by obstacle; these node have $nid = -1$.

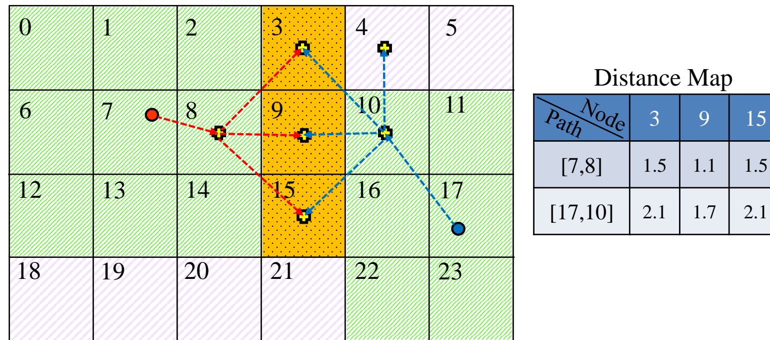
One of the challenges in parallel programming is to avoid *interference*. It occurs when read/write operators are applied on a shared memory concurrently by more than one thread. Such a case can happen in our implementation when two nodes reach to the same node at the same time (See Fig. 5.12). In this figure, nodes 8 and 10 read the cost(Co) of node 9 (in line 5 of kernel1), at the same time. The value at memory location corresponding to node 9 is ∞. If threads from node 8 (t_8) and 10 (t_{10}) both want to check and change the value at this location at the same time, each thread will first have to read the value. Depending on when the reading occur, it is possible that both t_8 and t_{10} will read a value of ∞. After checking in line 5, that is established for both threads, t_8 writes the new cost value "1.1" (See distance map in 5.12), then t_{10} writes its own cost value "1.5" into the memory location, which is not correct. The value, ∞, should have been updated to "1.1" (by t_8), but instead, the value was updated by t_{10}.

As explained in Section 2.5.2, this kind of problem can be solved by using *atomic*

functions. Fortunately, in this version of CUDA that we used, atomic operators are already implemented. Thus, in line 5 an *atomic add operator* is used in order to avoid interference by assigning exclusive rights to one thread at a time.

Besides the situation which has been explained, another case can happen when two threads reach the same node in different moments. To distinguish this condition, in line 8, we check the value of variable $\eta(nid)$ if it is not equal to -1 (initialized value), which means that the node nid has been visited beforehand. Furthermore, by passing if operator in line 5, we already know that the new path (from current node tid to nid) is smaller than old value in $Uc(nid)$. It means the old longer path and owner of node nid must be replaced by the new one. Also, we have to modify Integral $\mathbf{I}$ by subtraction from old integral value, $\mathbf{I}^{n}_{o(nid)}$. This is done in line 9. Since the algorithm is based on node indexes (instead of position vector), in line 11 to compute the gradient vector, $\mathbf{Pos}(o(nid))$ is applied to access the position vector of the robot. $o(nid)$ indicates to the dynamic node (robot) where the node nid belongs to its Voronoi region.

And finally in lines 11 and 12 the integral of node nid is added to $\mathbf{I}_{o(nid)}$, by using cost, Uc, density value, ϕ and first unit vector of node nid.



Path \ Node	3	9	15
[7,8]	1.5	1.1	1.5
[17,10]	2.1	1.7	2.1

Figure 5.12: Red and blue points are robots, conflict can happen in orange pattern cells.

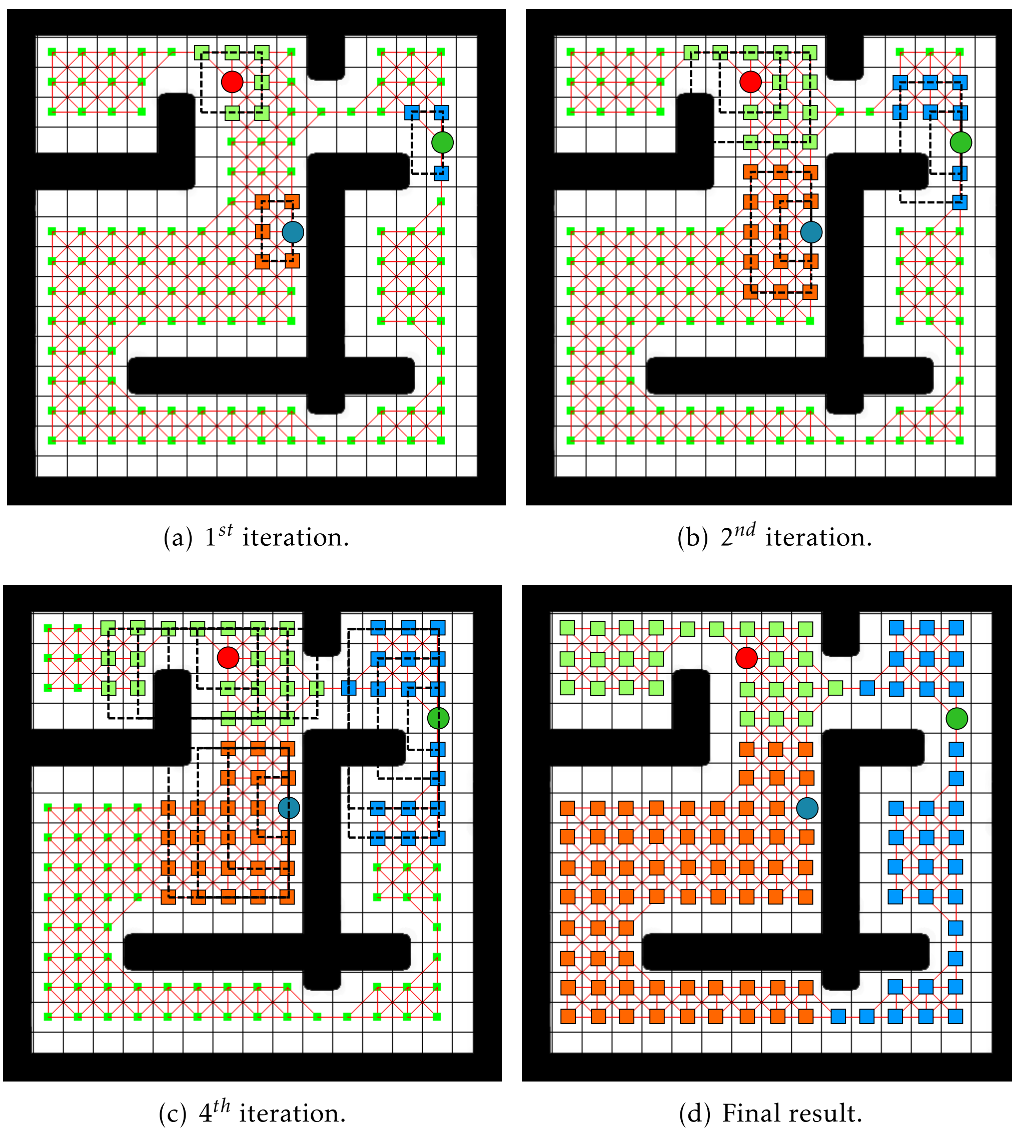
After execution of the first kernel, a second kernel (Algorithm 8) updates the cost value of elements in Co. By comparing to Uc, those elements in Co which have a bigger value than Uc, will be replaced by equivalent elements from Uc. Also, the mask array will be updated based on this comparison too (line 4).

Fig. 5.13 (a-d) shows the wavefront computed by the proposed method. It can be

Algorithm 8: KERNEL2 (Ma,Co,Uc)

```
1 tid ← getThreadID
2 if (Co(tid) > Uc(tid)) then
3     Co(tid) ← Uc(tid) //Update Co by Uc
4     Ma(tid) ← true //to keep evolving the wavefront
5 Uc(tid) ← Co(tid)
```

seen that the number of threads inside kernel increases during the construction of the Voronoi region.



(a) 1st iteration.

(b) 2nd iteration.

(c) 4th iteration.

(d) Final result.

Figure 5.13: Running parallel wavefront algorithm. Robots are circles.

5.3.3 CPU VS GPU

In order to validate the speed of running Dijkstra algorithm in the new implementation on CUDA, we compare the results of CPU and GPU. In the case of CPU, we used one of the best libraries, which is called **Boost C++ Library** (Sutter and Alexandrescu, 1998). It has already SSSP Dijkstra algorithm implemented inside. To have a fair result in all of the simulation, we report the mean of 50 times running the algorithm on the same computer.

In first simulation we used a benchmark, USA Road graph [1], the (undirected) road networks of the 50 US States and the district of Columbia, with around 24 million nodes and 29 million edges. The distance between cities are the weight on each edge.

Method	Time (sec)
CPU	6.49
GPU(CUDA)	49.83

Table 5.2: Result of first test.

We got an unexpected result in this implementation (See Table 5.2), as Boost library was much faster than CUDA. Based on this implementation we understand that the vertex degree of the graph has an effective influence on running SSSP on CUDA. Indeed, when the nodes have degree of 2 or 3 like in this graph, in kernel launching, in each iteration the number of threads for each node is 2 or 3. This case is similar to a sequential running, thus to use the ability of parallelism of CUDA, the input graph must have much more edges than nodes. In the first test the ratio between number of edges and nodes is 1.2. Therefore, in our second test an undirected graph with 5 nodes and 20 edges and 6 nodes and 22 edges, (with ratio 4 and 3.6 consequently) are used. Table 5.3 illustrates that GPU's execution time is half of the time that should be spent for CPU.

If the ratio is more than 2, CUDA can run faster than CPU. By increasing this ratio the difference of running time between CPU and GPU will increase too. According to our working space, in the last experiment we convert a map to a grid graph with

[1] ftp://ftp.cc.gatech.edu/pub/people/bader/CSE6140/USA-road-d.USA.gr.gz

Method	Time (sec) G1	Time (sec) G2
CPU	0.0021	0.0043
GPU(CUDA)	0.0012	0.00198

Table 5.3: Result of second test on two graphs with 5 ($G1$) and 6 ($G2$) nodes.

2,400 nodes and 11,035 edges (with the ratio 4.59). The result in Table 5.4 validates our above explanation.

Method	Time (sec)
CPU	0.354
GPU(CUDA)	0.022

Table 5.4: Result of running Dijkstra algorithm on grid graph.

Although in Table 5.4 similar to GPU, the running time by CPU is less than a second, but if we consider a large map with many robots then the difference between the running times will be substantial.

5.4 Implementation Result

To demonstrate the performance of our distributed algorithm, we conducted two experiments using a simple real size map with 4 robots and a large office-like map with 6 robots. All simulations were run on a laptop with processor Intel (R) Core (TM) i7-3520M 2.90 GHz with 6 GB RAM. Moreover, the GPU was NVIDIA Geforce GT 640 M LE with CUDA 5.5 SDK in C++ programming language. By using the same hardware and software configuration we did some experiments on real robots.

5.4.1 Simple Map

In the first simulation a map with size of 6.4×5.3 meters is used. By applying a discretization factor of 11 cm the map is divided into 58×48 cells. It means our graph representation has N=2784 nodes. As we mentioned in Subsection 5.3.1, each node is connected to its 8 neighbors, hence in our representation the number of demanded

memory cell is 22272 ($8N$). The maximum number of threads which is used in each iteration in the CUDA kernel can be equal to N, 2784.

The input map is depicted in Fig. 5.14. The center of density function is defined at down-left ($x_0 = 4.1$, $y_0 = 1.8$) and it was used a Gaussian function with standard deviation (1.5,2) for the x and y axis.

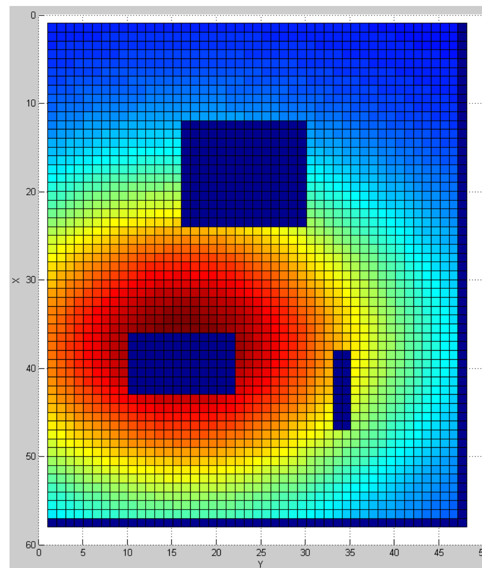


Figure 5.14: Input descretized map.

Consider Fig. 5.15, which plots the progress of the deployment during one trial run. Fig. 5.15(a) is the initial configuration of robots and 5.15(c) is the final location of the robots which is concluded by meeting the termination criteria. Furthermore, the final regions show that the map is well distributed between two robots.

Fig. 5.16(a) outlines the trajectory of robots from start to end configuration. From this figure, it can be seen that robots' movement is oriented to the center of the density function. After 85 iterations the algorithm reaches the final solution. Fig. 5.16(b) presents the decreasing of cost function during the deployment process.

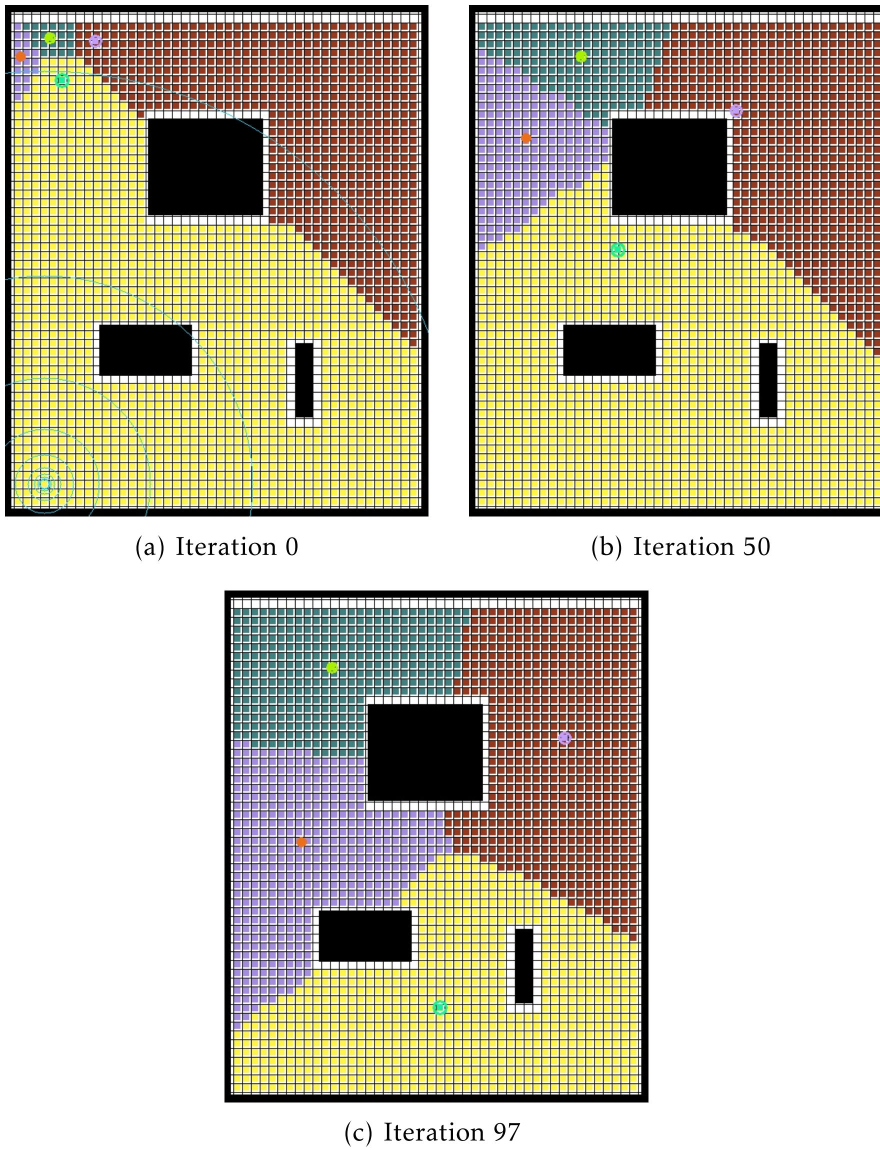
The video of this simulation is available at http://www.cpdee.ufmg.br//~coro/movies/RezaThesis/.

5.4.2 Office-like Environment

In the second simulation, a more complicated map is considered. The real size of the office-like map is 40×60 meters (Fig 5.17(a)). The discretization factor is 80 cm, hence in our grid representation there are 50×75 cells and N=3750 nodes, correspondingly. Fig. 5.17(b) shows the density function. Its parameters are defined as follows: $x_0 = 55.6$, $y_0 = 37$ and $\sigma^2 = 20$.

We divide 6 robots into 3 groups with 2 robots in each one. Such case is depicted in the following figure. In Fig. 5.18 (a-c) the initial location and the result of running our algorithm is shown. We obtain the final configuration after 180 iterations.

The path of the movement from start to end points are presented in Fig. 5.19(a). As it can be seen in Fig. 5.19(b), the cost function has decreased during deployment as expected.

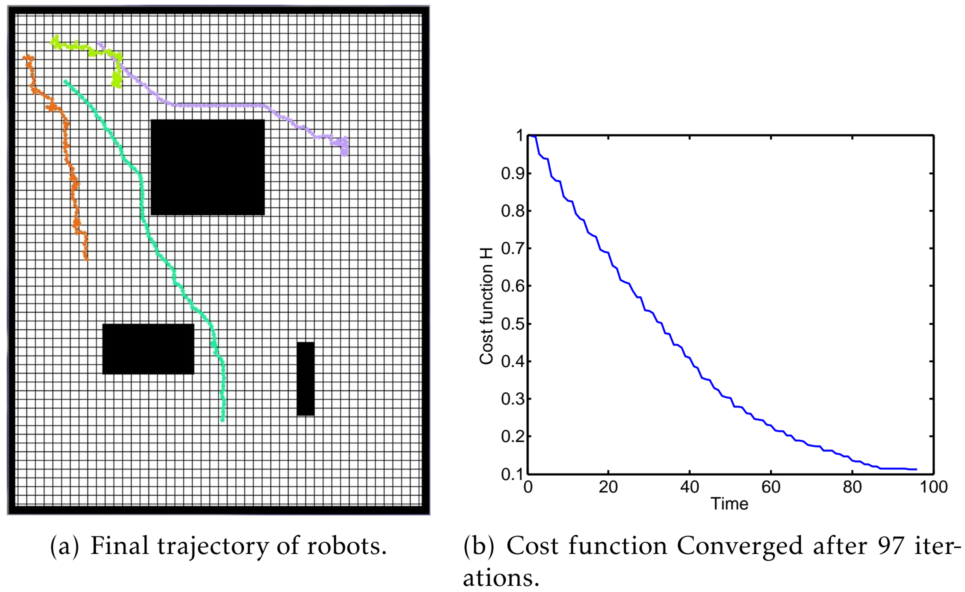


(a) Iteration 0

(b) Iteration 50

(c) Iteration 97

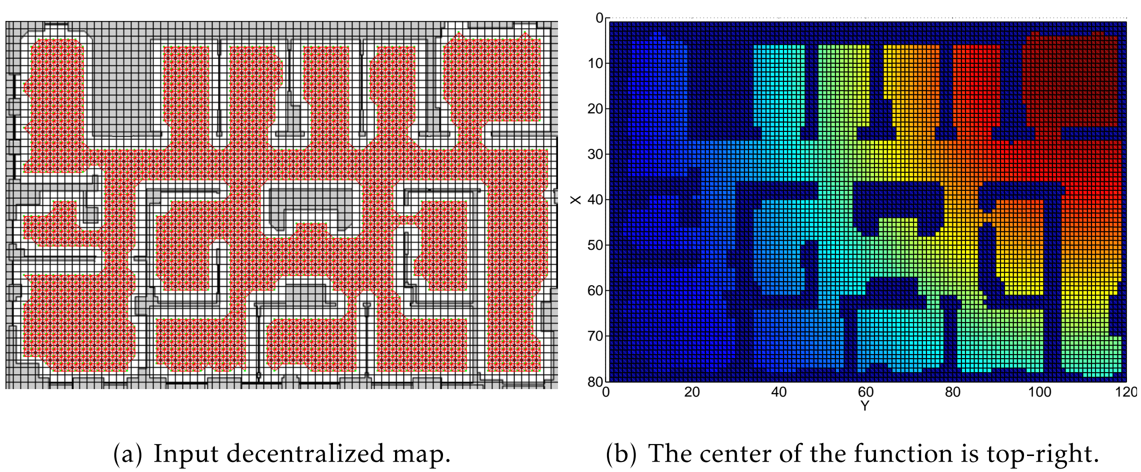
Figure 5.15: Snapshot of a simulation run.



(a) Final trajectory of robots.

(b) Cost function Converged after 97 iterations.

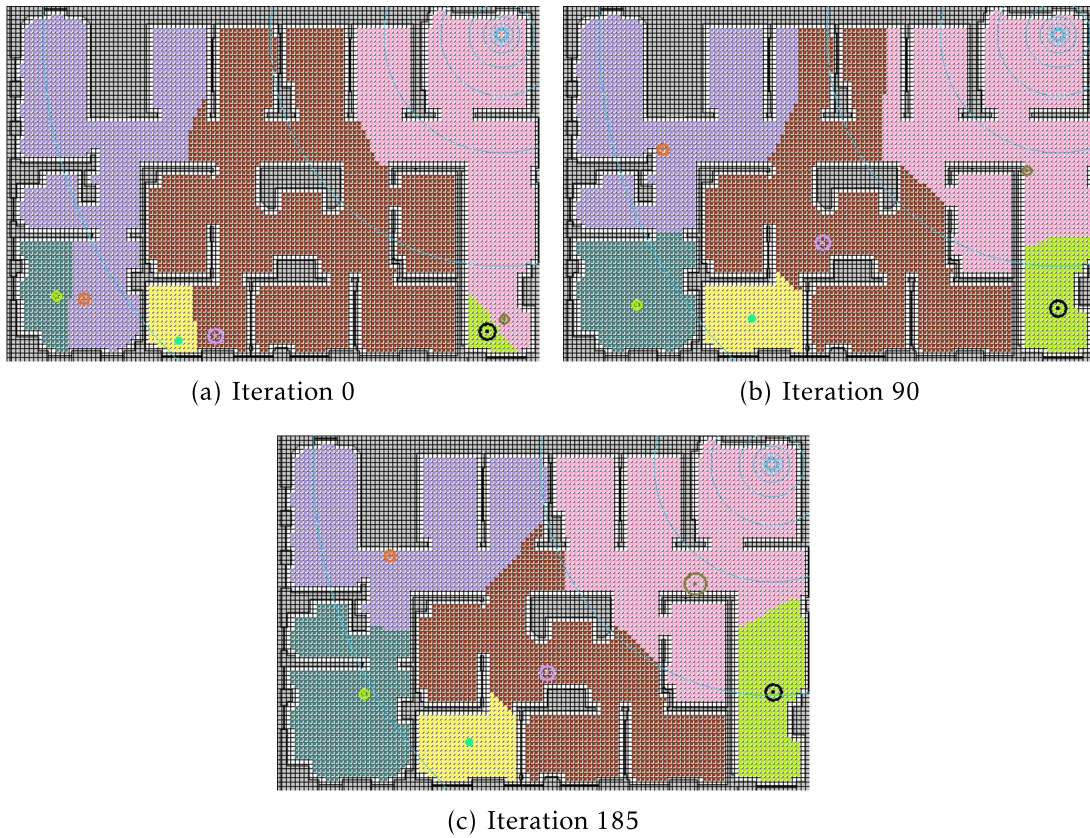
Figure 5.16: Final trajectory of two robots and the evolution of cost function over the time.



(a) Input decentralized map.

(b) The center of the function is top-right.

Figure 5.17: Input map and corresponding density function.

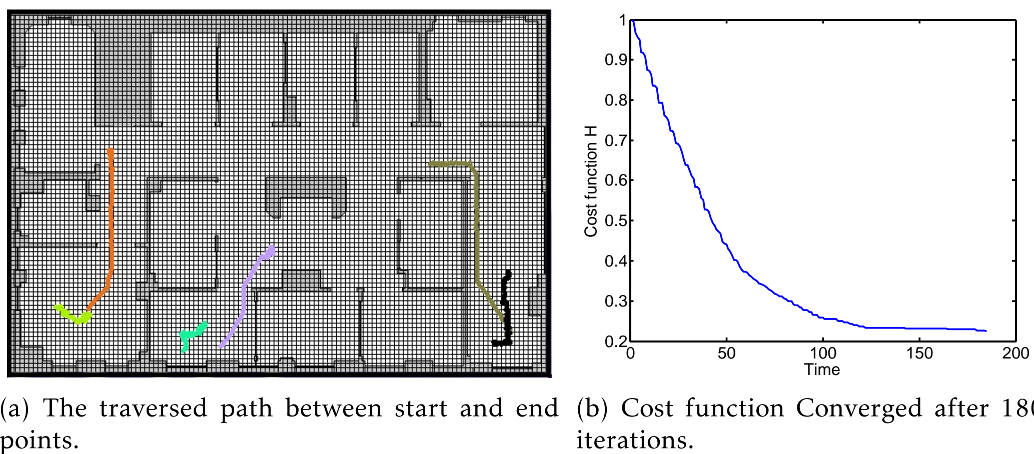


(a) Iteration 0

(b) Iteration 90

(c) Iteration 185

Figure 5.18: Iterations of running the new deployment method.



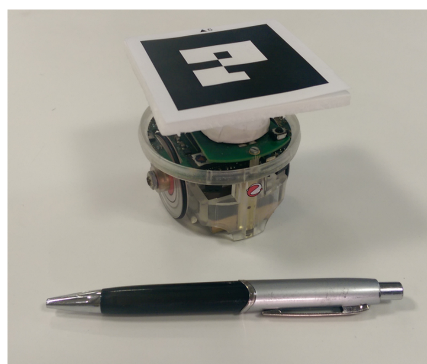
(a) The traversed path between start and end points.

(b) Cost function Converged after 180 iterations.

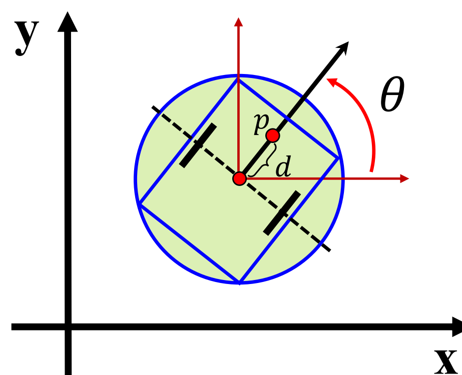
Figure 5.19: Final trajectory of 6 robots and the evolution of cost function over the time.

5.4.3 Real Robot Experiment

To evaluate the behavior of our algorithm in real world, we have applied it on real robots in the presence of external noise such as localization and communication errors. In these experiments a group of E-pucks[2] is used. Each E-puck is a differential drive platform equipped with camera, blue-tooth communication, and IR sensor (See Fig. 5.20(a)). The localization of the robots was provided by a global tracking system. This system is composed of an overhead camera and tracking patterns fixed at the top of the robots. At software level, for tracking robots' position and sending velocity commands to the robots we used the Robot Operation System (ROS) (Quigley et al., 2009).



(a) E-puck robot is equipped with a camera, blue-tooth communication, IR sensor, etc.



(b) A circular robot.

Figure 5.20: E-puck robot and robot model.

As previously mentioned, each robot is a differential drive platform which means that it is subjected to a non holonomic non-slip constraint. Moreover, by means of ROS it is possible to send linear (v) and angular (ω) velocity commands to this robot. However, the controller based on Eq. (5.4) is only suitable for holonomic fully actuated robots in which it is possible to directly define the robot velocity. To address this issue we use the same strategy presented in (Pimenta et al., 2013). First, we apply the gradient vector in Eq. (5.4) to generate a corresponding velocity vector, $[\dot{x}_i \ \dot{y}_i]^T$, where $\dot{x}_i$ and $\dot{y}_i$ are the desired velocity coordinates with respect to a global frame. Second,

[2]http://www.e-puck.org/

we use this vector as an input in a static feedback linearization scheme (Desai et al., 1998). The robot command is then given by :

$$\begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) \\ \frac{-\sin(\theta)}{d} & \frac{\cos(\theta)}{d} \end{bmatrix} \begin{bmatrix} \dot{x}_i \\ \dot{y}_i \end{bmatrix}, \tag{5.6}$$

where θ is the robot orientation with respect to the global frame and d is a parameter to be tuned related to the position of a reference point p. This reference point is indeed the point that is used as the robot position and d corresponds to the offset from the wheels axle (See Fig. 5.20(b)).

In our experiments, 5 robots started at 2 corners of the map (See Fig. 5.21). We have run the algorithm on a laptop equipped with CUDA processor. The map size is: 1.6×1.3 meter, discretized in a grid of 30×38 cells, with a Gaussian density function, illustrated in Fig. 5.22. Moreover (a,b) shows some snapshots of the tessellation for one trial run. And (d) depicts the perfectly descending evolution of $\mathcal{H}$. We can observe from the figures that the agents converge to an optimal deployment that is biased towards the peak of the density function. The video of this experiment is available at http://www.cpdee.ufmg.br//~coro/movies/RezaThesis/.

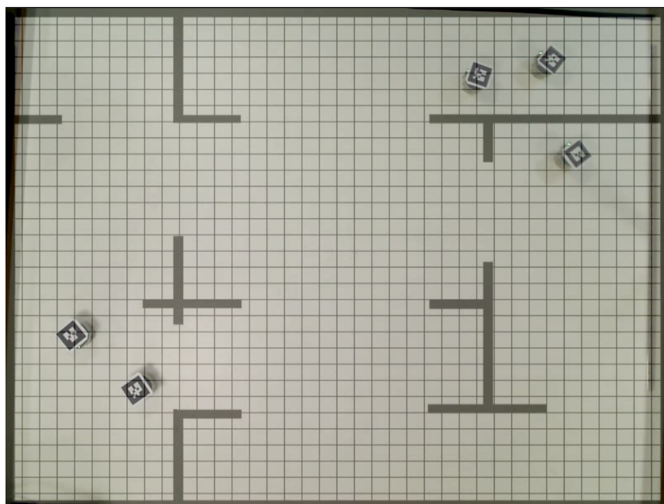
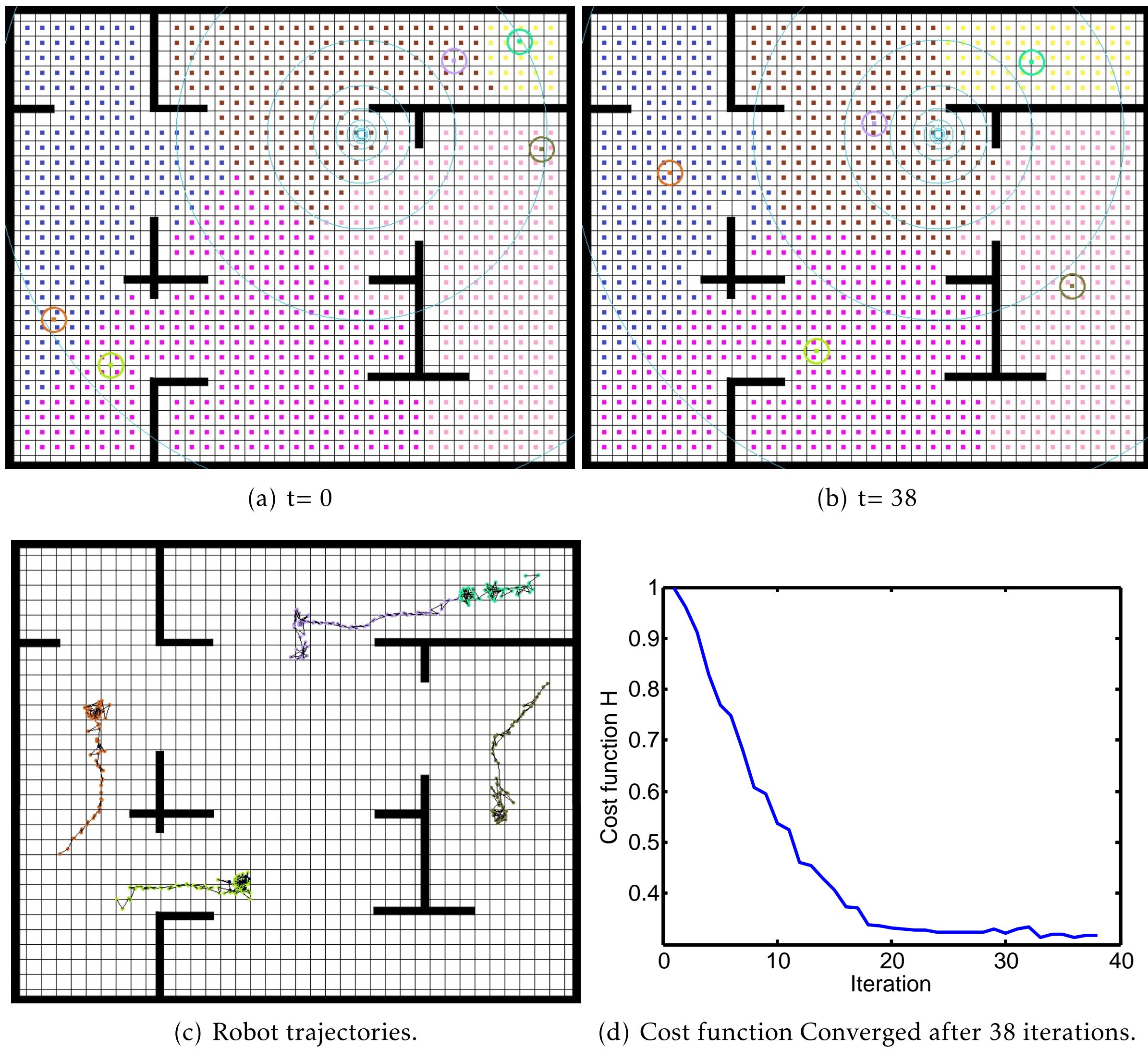


Figure 5.21: Robots are placed at initial configuration. Top view photo is taken by the camera located at height 1.5 meter from the robots.



(a) t= 0

(b) t= 38

(c) Robot trajectories.

(d) Cost function Converged after 38 iterations.

Figure 5.22: Snapshots of tessellation during experiment. The center of density function is the up right in the map (center of circles).

5.5 Conclusion

In this chapter, we presented a new multi-robot deployment algorithm.

The new algorithm controls the movement of robots in such a way that the cost function is never increased. By relying on this property, we ensured convergence of the proposed method. A discrete representation of the map and also a graph search algorithm were used based on the method by Bhattacharya et al. (2013c). The main difference is that our scheme used dynamic nodes to represent the robots in the graph instead of approximating their positions based on the center of the grid cell where they are located. Thus in this new mixed continuous and discrete scheme convergence can be achieved.

We also presented a new implementation using the parallel technology of GPU, CUDA. In this implementation, the well defined technique of representation based on the compact adjacency list was used. We propose a new multi-source parallel graph search algorithm (MSSP) based on SSSP introduced by Harish and Narayanan (2007), which is fast and also needs the same memory size and thread number of the original work. Moreover, different from other methods, both distance, control integral and Voronoi tessellation map are computed at the same time in our parallel implementation. Simulations on different maps and actual experiments illustrate the efficiency of the algorithm.

Chapter 6

Deployment in Topological Map

6.1 Introduction

In this chapter we propose an efficient deployment strategy to optimally distribute a team of robots in environments that can be represented by topological maps. Among the several applications for our solution are sensing and coverage of large corridor-based buildings, such as hospitals and schools, and the optimal placement of service vehicles in the streets of a big city.

As we mentioned, our new multi-robot deployment setup works based on the framework developed for a single robot by Araujo et al. (2015). In that paper, the authors used graph based topological maps to model the robot's workspace. Independently of the dimension of the original workspace, this strategy transforms the problem into a one dimensional problem, which highly increases the computational efficiency of the method, thus allowing for the deployment of large teams of robots in very large workspaces. Since we derived our deployment model according to this framework, we first review some of the main aspects of the framework. However, more detail can be found in (Araujo et al., 2015). After this review we then present our contribution for multi-robot deployment.

Due to the use of a topological map, in their proposed approach, the robots may be controlled using a sequence of human like commands, such as "turn right", "turn left" and "move straight". Also, no global metric localization is required. This approach is designed to be applied in large environments, usually non-convex workspaces, that are suitable for topological mapping. These include metropolitan regions composed of streets and intersections, pipelines and energy distribution systems with several connections and bifurcations, and large buildings with intersecting corridors.

Since in many applications of real world the robotic group cannot be controlled by a centralized system, our method is fully distributed in a way each robot needs to communicate only with its neighbors. Deriving our method upon the mentioned framework tackles the problem of applying multi-robot deployment on very large environments, which was a challenge for cited works in the literature. Also, the proposed strategy is provably correct in the sense that it is guaranteed that the robot positions converge to the optimal locations in the topological map. As a drawback, it is important to mention that, these topological locations do not necessarily correspond to optimal positions in the real workspace, which makes the quality of the deployment dependent on the map discretization.

The rest of the chapter is divided as it follows: problem statement is in Section 6.2; Section 6.3 is dedicated to present the proposed methodology, including the discretization technique, the topological map model, the methodology itself and the convergence proofs; Simulations and actual robot experiments are in Section 6.4; Finally, in Section 6.5 we conclude the work and present some ideas for future research.

6.2 Problem Statement

Given an environment and a team of mobile robots we want to distribute this team in the environment in such a way that *events of interest* over the environment can be efficiently handled by the robots. We assume that the environment is partially known and may be roughly represented by a non-proportional drawing, a sketch without metric information or even a photo (an example is shown in Fig. 6.1). Also, we consider very limited robots in terms of the sensors they carry for localization. While finding the best location for each robot is important, representing the environment in an efficient way plays a vital role to increase the performance of the deployment. Thus, in a block-shape environment, $\Omega \subset \mathbb{R}^n$ composed of corridors or driveways connected through intersections of incoming corridors or driveways (see Fig. 6.1). Assume a team of M mobile robots equipped with very simple sensors and reactive controllers so that they are only able to identify intersections, to move along corridors or driveways, and

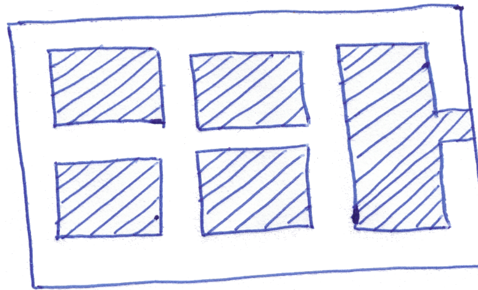


Figure 6.1: A sketched map of a building containing rooms (hatched polygons) corridors and intersections, where robots can move (white spaces). Our methodology can make use of simple, non-scaled drawing like this to deploy the robots over the environment.

to perform turning motions. Furthermore, a discrete data structure G encodes the environment without the use of any type of metric information so that this limited group of robots can use it to navigate. Thus, the objective of this chapter is to present a solution for the multi-robot deployment problems in the explained framework.

Problem 6.1 (Optimal distribution of robots)**.**
Consider the team of M mobile robots as described above with access to the data structure G provided by (Araujo et al., 2015) (explained in Section 6.3.1). Consider also that the robots have knowledge of a density function $\phi : \Omega \rightarrow \mathbb{R}^+$ that defines the relative importance of locations defined in the environment. Regions with higher values of density function are more likely to have events of interest in its interior. *Provide a deployment strategy with guaranteed convergence which is able to optimally distribute the team without the need of metric information and precise sensors for localization.*

In the next section we present a solution to the stated problem.

6.3 Methodology

In this section, we show the map representation framework proposed in the work by Araujo et al. (2015), and present a new distributed deployment algorithm along with a proof of convergence that solves Problem 6.1.

6.3.1 Topological Map Representation

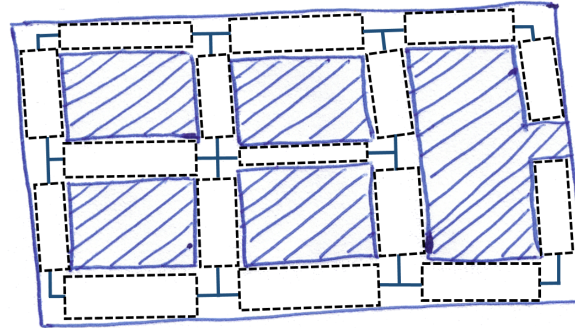
In the framework of the cited paper, to represent the environment as a topological, one dimensional map without using metric information, a *Graph* data structure is used. Thus, topological representation of an elementary representation (i.e. sketch, drawing, or photo) of a bounded environment Ω can be defined as a mapping $\Omega \rightarrow G$, that converts the environment into a graph space. In particular, this section focus on block shaped environments, such as the floors of an office building or neighborhoods of city.

A directed graph $G(\mathcal{V},\mathcal{E},\mathcal{C},\mathcal{I})$ is defined by a set of nodes $\mathcal{V}$, connected by edges $\mathcal{E}$ with specific costs $\mathcal{C}$, and robot commands $\mathcal{I}$. In this way, an edge $e \in \mathcal{E}$ denotes the link between two nodes (from x to y), $e = [xy]$ $(x,y \in \mathcal{V})$ and, $c(e) \in \mathcal{C}$ indicates the cost of a robot motion given by $I(e) \in \mathcal{I}$ between x and y. Furthermore, $\mathcal{N}_G(x)$ indicates the set of the neighbor nodes of x: $\mathcal{N}_G(x) = \{y \in \mathcal{V} \mid [xy] \in \mathcal{E}\}$.

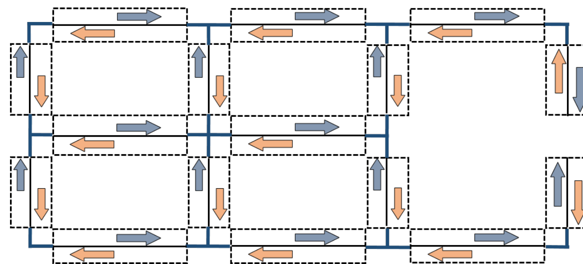
To construct the graph G, differently from previous methods (Yun and Rus (2013), Durham and Carli (2012)) that discretized the environment into grid cells, our framework makes a specific region, for example, a corridor or street, to be a cell. Since it does not consider a precise metric, the size of those cells are not necessarily equal, although we assume in this work that the cells are about the same size. Fig. 6.2(a) illustrates the division of the sketched map of Fig.6.1 in several cells.

To map real world problems where the direction of movements in some regions are constrained, such as one-way and two-way streets in big cities, a direction of movement is associated to each cell of the map. Regions that allow two-way movements are subdivided into two new regions. Each of these regions is then associated to a node in $\mathcal{V}$. Fig. 6.2(b) shows an example of this process. In this case, all regions of the graph are bi-directional. Therefore, each region has been separated into two regions, thus generating two nodes in the graph. The corresponding graph for this example is shown in Fig. 6.3. Notice that the edges of this graph represent possible movements among the nodes. If a robot can move from a node to another, we assume that these nodes are neighbors and add a correspondent edge to $\mathcal{E}$.

In a block shape symmetric map represented by a graph G, a cost between two



(a) Map discretization. Each region between two intersections is considered to be a cell.



(b) Depending on the allowed direction of movement, each original cell may be divided into two other cells. The arrows indicate the allowed motion in each region.

Figure 6.2: Discretization and definition of directions on the sketch map of Fig. 6.1.

neighbor nodes $c(x,y) \in \mathcal{C}$, $x,y \in \mathcal{V}$ and a command $I(x,y) \in \mathcal{I}$ are assigned to the edge connecting x and y ($[xy]$). This means that, to go from node x to y, a robot must execute a command $I(x,y)$ that will result in a cost $c(x,y)$, where $c : \mathcal{V} \times \mathcal{V} \to \mathbb{R}^{+}$.

Below are some properties assumed for c:

- $c(x,x) = 0$,
- $c(x,y) \geq 0$,
- $c(x,y) \leq c(x,z) + c(z,y)$,
- The graph might be asymmetric, *i.e.*, $c(x,y) \neq c(y,x)$.

Given this, *"Path"*, *"Commands"* and cost *"d"* between two arbitrary nodes (x and z) are described respectively as:

$$Path(x,z) = \{x, \cdots, y, \cdots, z\}, \;\; x,y,z \in \mathcal{V}$$

$$Commands(x,z) = \{I(x,p),\cdots,I(y,q),\cdots,I(w,z)\}\ x,p,y,q,w,z \in \mathcal{V}$$

$$d(x,z) = c(x,p) + \cdots + c(y,q) + \cdots + c(w,z)$$

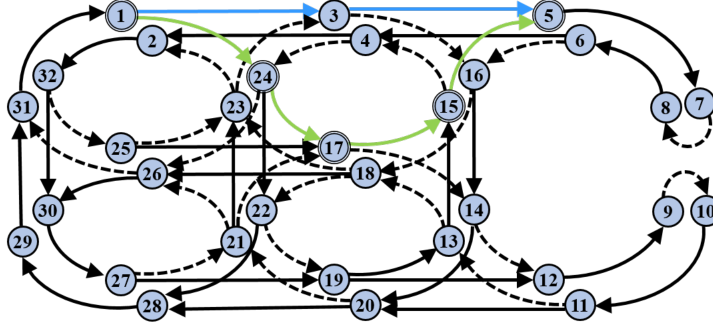


Figure 6.3: The corresponding graph of the map in Fig. 6.2 and two possible paths from node 1 to 5 with different color.

1 → 3 → 5	
$path_1$	$\{v_1, v_3, v_5\}$
$Command_1$	{Go_ Straight, Go_ Straight}
d_1	$c(v_1, v_3) + c(v_3, v_5)$=2+2=4

(a) A path between nodes 1 and 5.

1 → 24 → 17 → 15 → 5	
$path_2$	$\{v_1, v_{24}, v_{17}, v_{15}, v_5\}$
$Command_2$	{Turn_Right, Turn_Left, Turn_Left, Turn_Right}
d_2	$c(v_1, v_{24}) + c(v_{24}, v_{17}) + c(v_{17}, v_{15}) + c(v_{15}, v_5)$ =4+4+4+4=16

(b) An alternative path between the same nodes.

Figure 6.4: Moving from a node to another one in our new representation scheme. And computing the distance between nodes.

The command set is induced based on real world vehicle or human motions. An example of human-like command set is $\{Turn_lef,\ Turn_right,\ Go_Straight,\ Turn_Back\}$, with costs $c(x,y) \in \{4,4,2,8\}$. In this example, the cost of making a turn to right or left is higher that the cost of going straight, which is realistic for several robots. Moreover, the command

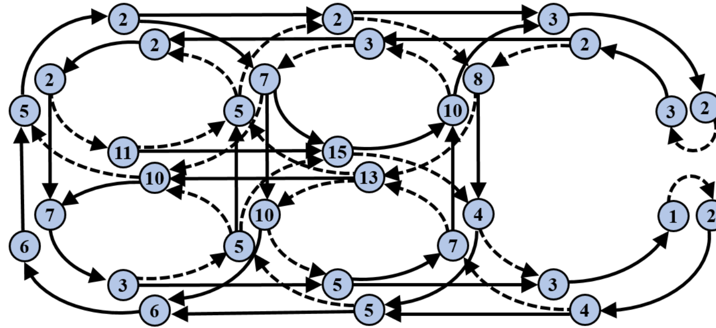


Figure 6.5: Density function indicated that the middle corridor with values 10, 11, 13 and 15 has higher priority to be serviced. The density value of each node corresponding to a corridor is written inside the node.

"turn_back" costs the maximum value for a robot. Function $d(x,z)$ is then a cost function that denotes the sum of the costs over the path from node x to y.

As an example, in Fig. 6.3 at least two paths between the left uppermost region (node 1) and the right uppermost region (node 5) exist. As highlighted in this figure using blue and green, these paths are: {1, 3, 5} and {1, 24, 17, 15, 5}. Fig. 6.4 presents the corresponding values of commands and costs for the two paths shown in Fig. 6.3. By relying on this weighting technique, the path with the smallest cost between two nodes can be found using the Dijkstra algorithm.

As in a standard deployment problem, a density function may be defined over the topological map. Assuming a continuous density function over the original environment, this maybe done by assigning a higher number to the node associated with the center of the density function, and decreasing this value as we get far from this node (in terms of number of nodes). This function can be also defined based on the frequency of events of interest for each region or a global probability function. Since the density function states the priority of a region (node) to be serviced, regions with lower *events of interest* will receive a lower number. If we consider the map in Fig. 6.2 as an office, and the number of users (or applicants) as the density function, Fig. 6.5 indicates a function where the middle corridor has more priority. In practical applications, we can assume that an automatic system monitors the traffic of users in order to obtain the density function. In this way, the values in the nodes denote the density of users in

the corridors (see Fig. 6.5).

Next section will show our decentralized solution for deploying robots in the environment using graph G and the density function defined over it.

6.3.2 Multi-Robot Deployment

Now, we show our solution to Problem 6.1. As previously stated, we assume a team of M mobile robots, $R = \{r_1, \cdots, r_M\}$, with access to the graph G and full knowledge of the density function $\phi : \mathcal{V} \to \mathbb{R}^+$. The graph node in which robot i is currently located is given by $p_i \in \mathcal{V}$, and $P = \{p_1, \cdots, p_M\}$.

Our strategy is based on the partition of the graph in such a way that, after the deployment, each robot will be responsible to respond only to the events that happen at the graph nodes assigned to that robot. Before showing our algorithm we first need to define the specific graph partitioning used in this chapter:

Definition 6.1 (Voronoi subgraph)**.** The Voronoi subgraph g_i in G is given by:

$$g_i = \{x \in \mathcal{V} \mid d(p_i, x) < d(p_j, x), \forall i \neq j\}, \tag{6.1}$$

where, $d(x,y)$ is a function $d : \mathcal{V} \times \mathcal{V} \to \mathbb{R}^+$ that denotes the cost of the shortest path between nodes x and y. If $d(p_i, x) = d(p_j, x)$, the node x is assigned to the robot with smaller index number as in Yun and Rus (2013).

The set of Voronoi subgraphs defines a Voronoi partitioning of the graph G. In the partition every node will be assigned to a robot. Also the union of all subgraphs is equal to $\mathcal{V}$ and the intersection between two different subgraphs is empty.

Our approach builds upon the work in Yun and Rus (2013) in which the problem of optimally deploying the team of robots on a graph is treated as a locational optimization problem. As in Yun and Rus (2013), we reformulate our general deployment problem as the one of minimizing the cost function:

$$\mathcal{H}(P, G) = \sum_{i=1}^{M} \mathcal{H}_i(p_i, g_i), \tag{6.2}$$

where,

$$\mathcal{H}_i(p_i, g_i) = \sum_{q \in g_i} d(p_i, q)\phi(q). \tag{6.3}$$

The discrete formulation given above is exactly the same formulation of the well known p-median problem (Reese, 2006), which is a NP-hard problem with several previously proposed centralized solutions. In this work we present a distributed solution which is used for multi-robot deployment in the same spirit of Yun and Rus (2013).

The main idea of our solution is to generate successive iterations in which the robots are relocated to different nodes in such a way $\mathcal{H}$ decreases until reaching convergence. In fact, this iterations consists in choosing a special node inside the robot subgraph and then move the robot to this node.

Our solution is presented in the form of a distributed control algorithm in Algorithm 9, similar to what is shown as Algorithm 1 in (Yun and Rus, 2013). In fact, our contribution is the proposition of a more efficient algorithm (Algorithm 10) to find the next best node, where the robot should be relocated. Thus, our Algorithm 10 is used in the place of Algorithm 2 defined in (Yun and Rus, 2013) for multi-robot deployment.

Algorithm 9: Distributed controller for robot i (similar to Yun and Rus (2013)).

```
  Input: G,p_i,p_i* = p_i // where G is the graph, p_i is the robot location, and p_i* is the
                              next best node
1 State: Compute
  {p_j*} ← Receive_Locations() // Receive locational information of neighbor
                          robots, j ∈ R_i (the current next best node of neighbor robots)
2 g_i ← Compute_Voronoi(G,p_i*,{p_j*}) // Compute Voronoi subgraph.
3 p_i* ← Find_next_best_node(g_i,p_i*)  // Call Algorithm 10.
  if (p_i ≠ p_i*) then
4 |  State ← Moving  // Switch to Moving State.
5 State: Moving
6 Move_To(p_i*)  // Explained in Section 6.3.4.
  if (p_i = p_i*) then
7 |  State ← Compute // Switch to Compute State.
```

In Algorithm 9, the robot is always in one of the two states; *Compute:* to compute the *next best node* (p_i^*); and *Moving:* to move to the node p_i^*. Before computing the next node, robot i must receive the current information from other robots called neighbors.

A set of neighbors is defined by $\mathcal{R}_i = \{j \in R \mid \exists [xy] \in \mathcal{E}, x \in g_i, y \in g_j, i \neq j\}$, which means robot j is a neighbor of robot i if they have vertices that are neighbors in graph G; it should be noticed that, in contrast to the continuous setup, here we do not have a common boundary between two Voroni regions. Instead, the vertices on the boundary of Voronoi region i are the neighbors of the corresponding vertices in Voronoi region j. We assume that robot i and its neighbor robots can communicate to each other whenever they need to exchange information. Most works based on the locational optimization framework rely on the same assumption. The information shared between the robots is the next best location and not the current location of the robots.

In *Compute* state, after obtaining $\{p_j^*\}$ (next neighbor robots location), and computing the corresponding Voronoi subgraph (g_i), the next best node (p_i^*) is found by calling Algorithm 10. The state will be switched to *Moving* if the new best node differs from the current one. Robot i moves toward p_i^* and reaches this node in finite time (assuming absence of failure in the low level controller of the robot), and changes the state to *Compute* again. Note that according to Algorithm 9, function $Find_next_best_node()$ is only called when $p_i = p_i^*$.

In Algorithm 10, the next best node is selected based on the possibility of decreasing $\mathcal{H}$ by decreasing the component of this function related to robot i, $\mathcal{H}_i$. The next node is chosen to be the direct graph neighbor node which allows for the maximum decreasing of $\mathcal{H}$ considering the current Voronoi subgraph as the partition associated to robot i.

An example of computing $\mathcal{H}_i$ for node 15 and its neighbors 4 and 5 is shown in Fig. 6.6. It should be noticed from the figure that for computing $\mathcal{H}_i$ for nodes 4 and 5, the same Voronoi subgraph that was applied to compute $\mathcal{H}_i$ for node 15, is used.

Next section presents the proof of convergence for the proposed algorithm along with its computational complexity analysis and comparison.

Algorithm 10: Function $Find_next_best_node()$.

Input: g_i, p'_i // g_i is the Voronoi subgraph of robot i and p'_i corresponds to the current best node.

Output: p^*_i; // The next best node.

1 $\mathcal{H}_i \leftarrow \sum_{q \in g_i} d(p'_i, q)\phi(q)$ // Compute the cost function of the current position of robot i.

2 **foreach** $a \in \mathcal{N}_G(p'_i)$ **do**

  $\mathcal{H}_a \leftarrow \sum_{q \in g_i} d(a, q)\phi(q)$ //Compute $\mathcal{H}_a$ value for all the neighbor nodes of p'_i.

3 $p^{min}_i \leftarrow \arg\min_{k \in \{\mathcal{N}_G(p'_i)\}} \mathcal{H}_k$ // Find the minimum $\mathcal{H}_k$ among the neighbor graph nodes

4 **if** $\mathcal{H}_k < \mathcal{H}_i$ **then** // If the minimum cost function of the neighbor nodes is less than the robot current $\mathcal{H}_i$

5   $p^*_i \leftarrow p^{min}_i$ // Set the neighbor node (with minimum cost) as the next best node.

  else

6   $p^*_i \leftarrow p'_i$ // Otherwise the current node is the best node (there is no better node out of neighbor nodes)

7 Return p^*_i

6.3.3 Analysis

The proposed algorithm works upon the following assumptions:

i) The graph (G) that represents the input map is given to all the robots in the beginning of the task.

ii) Robots have access to the next best node of their neighbors. Whenever a robot needs to compute its next best node, the latest updated information of neighbor robots is available by means of communications.

Given these assumptions, we can now prove the convergence of the proposed multi-robot system.

System Convergence

We start by presenting some lemmas and definitions.

Lemma 6.1. Let $\mathcal{H}(P, G_W) = \sum_{i=1}^{M} \mathcal{H}_i(p_i, w_i)$, where $\mathcal{H}_i(p_i, w_i) = \sum_{q \in w_i} d(p_i, q)\phi(q)$, w_i is an arbitrary partition of G which is different from the Voronoi partition defined in

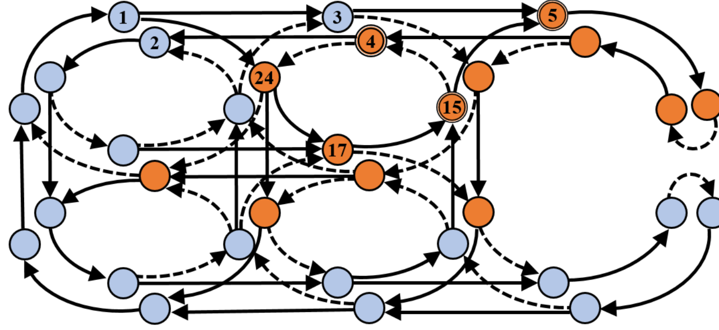


Figure 6.6: Robot is placed on node 15. Based on Algorithm 10, $\mathcal{H}$ will be computed for nodes 15, 4 and 5, over the same Voronoi subgraph (nodes with brown color).

(6.1), and $\phi : \mathcal{V} \mapsto \mathbb{R}^+$. The following inequality holds:

$$\mathcal{H}(P,G) \leq \mathcal{H}(P,G_W),$$

where $\mathcal{H}(P,G)$ is computed according to Eq. (6.2) using a Voronoi partition of the nodes of G.

Proof. The proof follows the same arguments used in the proof of Proposition 3.1 in (Du et al., 1999). According to Eqs. (6.2) and (6.3) we have that:

$$\mathcal{H}(P,G) = \sum_{i=1}^{M} \sum_{q \in g_i} d(p_i,q)\phi(q),$$

and

$$\mathcal{H}(P,G_W) = \sum_{i=1}^{M} \sum_{q \in w_i} d(p_i,q)\phi(q).$$

According to Definition 6.1 we have that $d(p_i,q)\phi(q) \leq d(p_j,q)\phi(q)$ if q is in the Voronoi subgraph of i, g_i. Thus, as w_i is not the associated Voronoi subgraph the inequality holds:

$$\mathcal{H}(P,G) = \sum_{i=1}^{M} \sum_{q \in g_i} d(p_i,q)\phi(q) \leq \sum_{i=1}^{M} \sum_{q \in w_i} d(p_i,q)\phi(q),$$

Thus:

$$\mathcal{H}(P,G) \leq \mathcal{H}(P,G_W).$$

■

Before presenting the second lemma, we will define a decreasing lower bounded sequence.

Definition 6.2 (Decreasing lower bounded sequence)**.** A sequence $\{x_n\}$ is called decreasing and lower bounded, if:

i) $x_i \leq x_{i-1} \quad \forall i \geq 1$

ii) $\exists B_0 \in \mathbb{R}$ such that $\forall n \; x_n \geq B_0$

Lemma 6.2. Every decreasing lower bounded sequence $\{x_n\}$ converges to the greatest lower bound of the set $\{x_n : n \in \mathbb{N}\}$.

Proof. This is a well known result in real analysis. ■

Definition 6.3. Let S be the set of state vectors of the multi-robot system described in Subsection 6.3.2 with robots executing algorithm 9 and 10, where state vector S is a vector of the next best nodes:

$$S = \begin{bmatrix} p_1^* \\ \vdots \\ p_M^* \end{bmatrix}.$$

Definition 6.4. A state vector S^* is called a local minimum state of the $\mathcal{H}$ function in Eq. (6.2) if :

$$\mathcal{H}(S,G) = \sum_{i=1}^{M} \mathcal{H}_i(p_i^*, g_i) \leq \mathcal{H}(S',G), \; \forall S',$$

where, S' is any state vector which differs from S in only one of its entries j, where the current next best node is replaced by a node which is a neighbor of p_j^*. The computation of $\mathcal{H}(S',G)$ is performed using the same Voronoi subgraph defined for $\mathcal{H}(S,G)$ as the associated graph partition.

Theorem 6.1. Given a multi-robot system as described in Subsection 6.3.2 with robots executing algorithms 9 and 10, its associated state vector converges to a local minimum state of $\mathcal{H}$ in (6.2) in finite time if the following assumptions are verified: (i) the robots

have access to the graph G; and (ii) the robots have access to the next best nodes of their neighbors whenever they need.

Proof. The value of $\mathcal{H}$ can only change due to two reasons: 1) the computation of a new Voronoi tessellation (line 3 in Algorithm 9); and 2) the computation of a next best value (line 4 in Algorithm 10).

If the assumptions are verified then, according to Lemma 6.1, the computation of the new Voronoi tessellation cannot increase the value of $\mathcal{H}$. Also by inspection of Algorithm 10 we can say that if there exist a state vector S' obtained by the replacement of the entry associated with the current p_i^* with a graph neighbor of p_i^* such that $\mathcal{H}(S',G) < \mathcal{H}(S,G)$, where S is the current state vector, then the system will evolve to a new state vector in finite time and this new state vector is so that $\mathcal{H}$ will decrease. Moreover, if such a vector S' does not exist then the system does not change the state vector.

Given these facts and also the fact that $\mathcal{H} \geq 0$, we can guarantee that $\mathcal{H}$ evolves according to a decreasing lower bounded sequence $\{\mathcal{H}_t\}$. Thus, according to Lemma 6.2, $\mathcal{H}$ converges to the greatest lower bounded of the set $\{H_t : t \in \mathbb{N}\}$. Since the number of state vectors in S is finite, this convergence happens in finite time.

When $\mathcal{H}$ reaches convergence, the state vector will have converged to a local minimum state as given by Definition 6.4, since this vector does not change when no feasible S' capable of decreasing the value of $\mathcal{H}$ can be found. ∎

Computational Complexity

We consider the bottleneck of our algorithm *computing the Voronoi tessellation and the cost function* $\mathcal{H}_i$ in each time step. Considering our weighted directed graph, similarly to (Bhattacharya et al., 2013a), since the algorithm is base on Dijkstra, the complexity of this computation is $O(|\mathcal{V}|\log|\mathcal{V}|)$ when an efficient data structure such as a heap is used, where $|\mathcal{V}|$ is the number of nodes in the graph. In fact, all the computations are done over the single run of Dijkstra algorithm.

To compute the complexity of the algorithm precisely, we may consider its two main parts. In the first part, the cost function (also the Vornoi tessellation) is computed

for robot located on p_i (line 1 in Algorithm 10). The second part denotes the time related to the computation of the $\mathcal{H}_i$ for the graph neighbor nodes of p_i (line 2 in Algorithm 10). As we mentioned, the complexity of computing $\mathcal{H}_i$ is $O(|\mathcal{V}|\log|\mathcal{V}|)$. Since in our discrete setup the maximum number of neighbor nodes is 4, $|\mathcal{N}_G(p_i)| \leq 4$, the complexity for all the neighbor nodes is $4 \cdot O(|\mathcal{V}|\log|\mathcal{V}|)$. However we can consider it as $O(|\mathcal{V}|\log|\mathcal{V}|)$. It should be mentioned that to compute the $\mathcal{H}_i$ for neighbor nodes, the Voronoi tessellation that was computed for p_i is used. Thus, the Voronoi tessellation is computed only once in each step.

Thus, we can write that the total complexity of our algorithm is $\mathrm{O}(|\mathcal{V}|\log|\mathcal{V}|)$.

Comparison

We compare our proposed method with the most similar works in the literature, (Durham and Carli, 2012) and (Yun and Rus, 2013). In the sense of time complexity the first and second algorithm need $O(|\mathcal{V}|^3)$ and $O(|\mathcal{V}|^2)$ respectively. While the technique proposed by Yun and Rus (2013) looks for the best next node for robot i among the nodes in its entire assigned Voronoi subgraph g_i (computing Eq. (6.2) for many nodes), in (Durham and Carli, 2012) robots move according to a random destination in their dedicated subgraph after meet other robots and find the generalized centroids based on pairwise partitioning rule. In addition, the former and latter techniques need a time to compute the shortest path from the robot's current position to: the next best node; or the random sample point in the corresponding subgraph respectively. This can be done by *BFS* (Breast-First Search) algorithm with $O(|\mathcal{V}|)$ or Dijkstra algorithm with complexity $O(|\mathcal{V}|\log|\mathcal{V}|)$. In contrast, we compute the value of $\mathcal{H}$ in the robot's next possible movement to neighbor nodes to see whether it is worth or not. Hence robots are able to move to one of their neighbor nodes in each iteration. This modification which is applicable in any kind of graph representation, decreases substantially the complexity, and also makes the algorithm provable in the sense of convergence.

Another shortage of these works implies that they demand a network with high data transferring capacity which is not necessary in our method. Whereas Durham

and Carli (2012)'s work performs in short-range "gossip[1]" communication networks, robots i and j need $O(|V_i| + |V_j|)$ (where V_i is Voronoi partition of robot i), for communication bandwidth. This happens when they meet each other to compute their pairwise Voronoi partitions. Also, in Yun and Rus (2013)'s two-hop communication is needed, the information of neighbor robots and neighbors' neighbor is needed to be transferred over the communication network. Although we assumed to establish a reliable network communication between neighbor robots, we do not need high bandwidth network, since just neighbor robots exchange their locational information (a single node number).

And finally, in contrast to these works, our proposed method works without having the need of a precise metric and map for "next best point computation".

6.3.4 Robot Control

In this section we review the work done by Araujo et al. (2015) for controlling a robot in a topological map. In fact we used the same technique to control the movement of a team of robots in our deployment problem.

The solution for the multi-robot deployment problem shown in the previous section gives, for each robot at each time interval, the node of graph G where the robot must go after living the current node. Based on the way G was defined, remember that each node represents a portion of the environment, which could be a corridor or a street block, for example. Also, to each edge of the graph, there was associated a command for the robot, whose cost of execution was used in our algorithm to help deciding the best path. Therefore, assuming that the robot is currently at node $x \in \mathcal{V}$ and the next node computed by the algorithm is $y \in \mathcal{V}$, a controller needs to be designed so the robot follow edge $[xy]$ by executing command $I(x,y)$. As mentioned before, the set of commands used in this work contain "human-like" instructions which are executed when the robot leaves the current node. If a robot is in an indoor environment, for example, a command will be executed when an intersection of corridors is found. Thus,

[1]A short-range communication with asynchronous and unreliable communication between nearby robots

two controllers are necessary: one to drive the robots inside the node and another to follow command associated to the edge.

For the first controller, remember that the workspaces considered in this chapter are defined by regions similar to corridors or street blocks. These regions are usually defined by constrained areas surrounded by some sort of structure, for example, walls in a building, or the sidewalks on the streets. These structures will be a constant reference for the robot inside a node and can be used to aid its guidance and control. In this work, similar to the technique applied in (Araujo et al., 2015), we use such structures to define a vector field to guide the robot. Without loss of generality and to facilitate the explanation of our vector field generation method, we will call the structure that delimits the node region by *wall* in the rest of the section.

To compute the vector field, we create a reference frame on the closest point from the robot to the wall on its right side, as shown in Fig. 6.7. In this frame, the X axis is tangent to the wall, pointing in the forward direction, and the Y axis is orthogonal to it, pointing to the left wall. It is assumed that the robot is always in coordinate $X = 0$, so the frame moves with the robot.

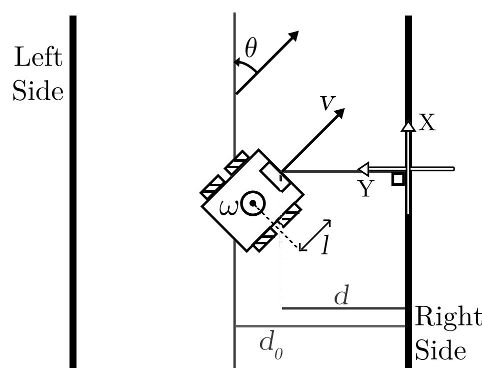


Figure 6.7: Geometry involved in the vector field generation and robot control. From the reference frame, distance d_0 is the expected distance while d is the current distance of the robot. θ indicates the robot's relative orientation and l, the distance from the center of the robot to its control point.

To make the robot move along the corridor by keeping itself parallel to the corridor's right wall at a distance d_0 from it, a planar velocity vector field $\mathbf{u} = [u_X, u_Y]^T$ is created. While the field component along the corridor, u_X, have a constant value, the

component along Y-axis, u_Y, is proportional to the error between the robot current distance to the wall, d, and the desired distance d_0. The vector is then normalized and scaled to the desired velocity of the robot.

To avoid possible obstacles in the corridor, such as people and other robots, vector field $\mathbf{u}$ can be summed with a repulsion vector field designed to avoid obstacles. Although this approach seems to be simple, it can easily create local minima in the field, which would stop the robot. Since the authors believe that the definition of obstacle avoidance vector fields and the solution to the problems related to it are out of the scope of this study, the reader is refereed to the works by Lam et al. (2011), which presents a vector field that allows the robot to present human-friendly behaviors in the presence of people and work of Araujo et al. (2015), which adds an obstacle/people avoidance solution to our vector field.

To track the vector field with a nonholonomic robot, we transform each vector into the inputs of the robot using a static feedback linearization controller (d'Andréa Novel et al., 1995). Assuming that the robot inputs are linear velocity v and angular velocity ω we have (similart to Eq. 5.6):

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ -sin(\theta)/l & cos(\theta)/l \end{bmatrix} \begin{bmatrix} u_X \\ u_Y \end{bmatrix} \tag{6.4}$$

where l is the distance from the center of the robot to its control point and θ is the angle between the robot and the reference frame, as shown in Fig. 6.7.

This previous controller drives the robot inside the nodes of the topological map until it finds itself in an intersection, which represents a node change. Because of the block-shaped regions of the map, it is possible to perceive that the robot is approximating an intersection by detecting that the structures of the corridor are shaping up to the opening of an intersection. If the robot is using a laser to follow the wall, for example, it is possible to detect that the segment that represents the wall in the laser scan is decreasing in length. When an intersection is detected, the robot must follow the command associated to the edge that connects the current node with the next one computed by the planner. Since the number of possible instructions is small, one

simple controller can be developed for each instruction. For example, a "Turn Left" instruction would lead to the activation of a proportional controller that would turn the robot by 90 degrees to the left, so it can move forward and enter the region of the new node, after the intersection. Once the controller finishes its task, the corridor follower, vector field based controller is switched back to move the robot inside the new node.

6.4 Implementation Results

In this section, the efficiency of the proposed method is investigated in both simulated and real robots experiments. Before showing the result, we explained the developed distributed architecture. The high-level system is shown in Fig. 6.8, where *Control*, *Deployment* and *Interface* modules are our three main entities. While the control module captures data and interacts with the environment, deployment module compute the next best node by relying on data received from the control module. To keep robots communication, the interface module send/receive pose information to robot i to/from other robots through a wireless connection. The deployment module receives and sends the best node information from/to the interface module periodically. Thus α_1 and α_2 are sampling rate for receiving and sending information from/to other robots. Also Δt refers to the time difference: $\Delta t = |t^{i+1} - t^i|$, $t > 0$.

In our implementations, we used C++ programming, ROS and Matlab for the different modules. According to Fig. 6.8, the Deployment module is implemented in Matlab, Interface and Control modules are developed in ROS. We used ROS toolbox in Matlab to exchange information between them.

6.4.1 Simulation Results

For the simulation we selected a real outdoor map of a neighborhood in New York City from Google Maps (See Fig. 6.9). In this map, streets and blocks have a symmetric shape, which is important for metropolitan cities in order to facilitate distributing services and urban management i.e. transportation, pipeline, electricity and so on.

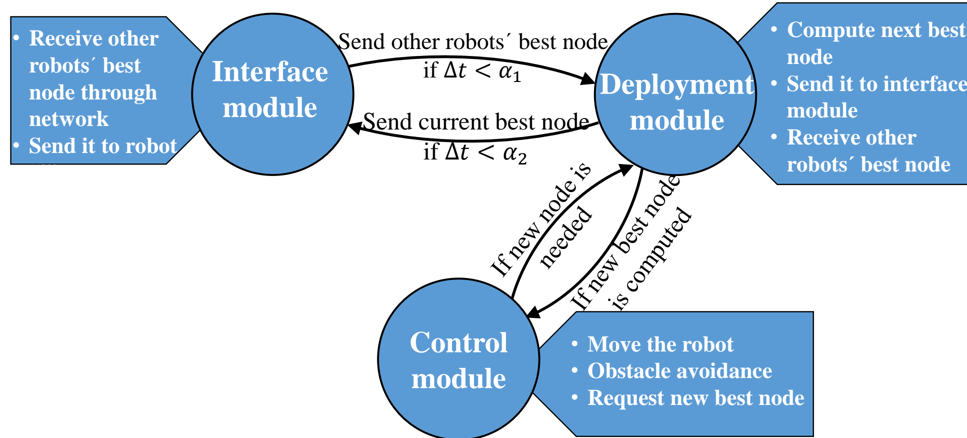
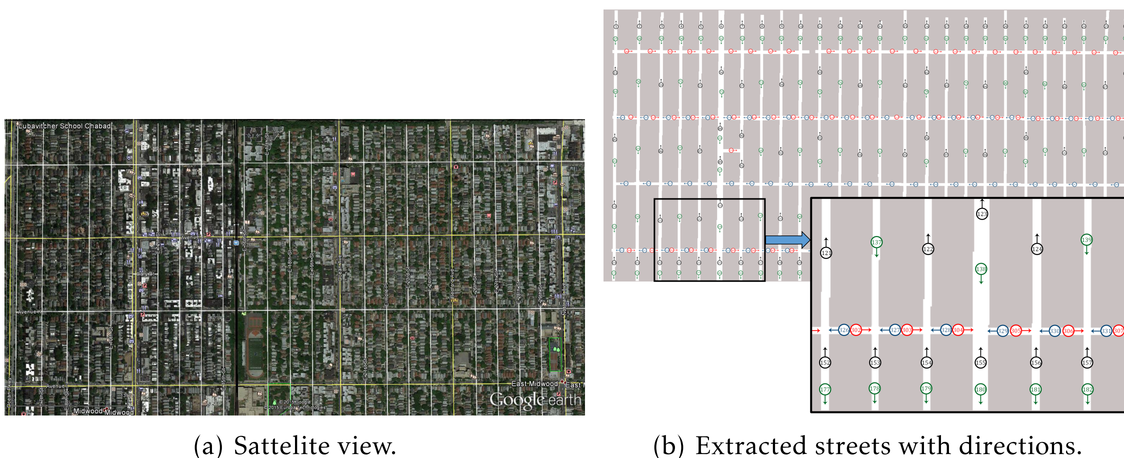


Figure 6.8: The high-level system of robot i, and the relation between three main modules in a high-level description.

Furthermore, this block-shape property gives us the ability to run our deployment algorithm without the need of precise localization. Thus, regardless the scale of the input map, first of all we find a topological representation of the map by extracting the streets as in Fig. 6.9 (b). This is done by applying morphological operators in image processing i.e. thresholding, erosion and dilation.



(a) Sattelite view. (b) Extracted streets with directions.

Figure 6.9: The map of a neighborhood in New York (grabbed from Google map).

Since the direction of the streets is available in Google Maps, a directed graph G can be constructed based on this map (the method was explained in Section 6.3.1). The command set $\mathcal{I}$ and its corresponding cost $\mathcal{C}$ used in this simulation is shown in Table

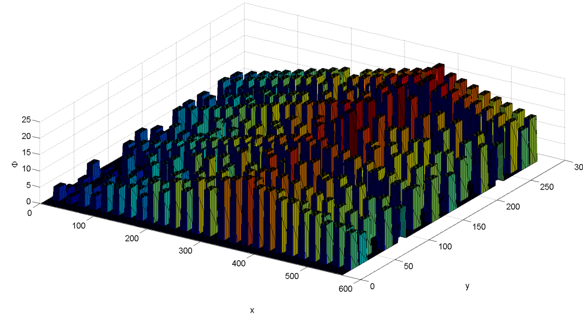


Figure 6.10: The density function defined in this scenario. The center of this function is defined at (387,152) where node 114 is placed.

Table 6.1: Set of commands and costs in the simulation.

Command#	Type	Cost
1	Turn_Left	1.5
2	Turn_Right	1.5
3	Straight	1
4	Turn_Back	2

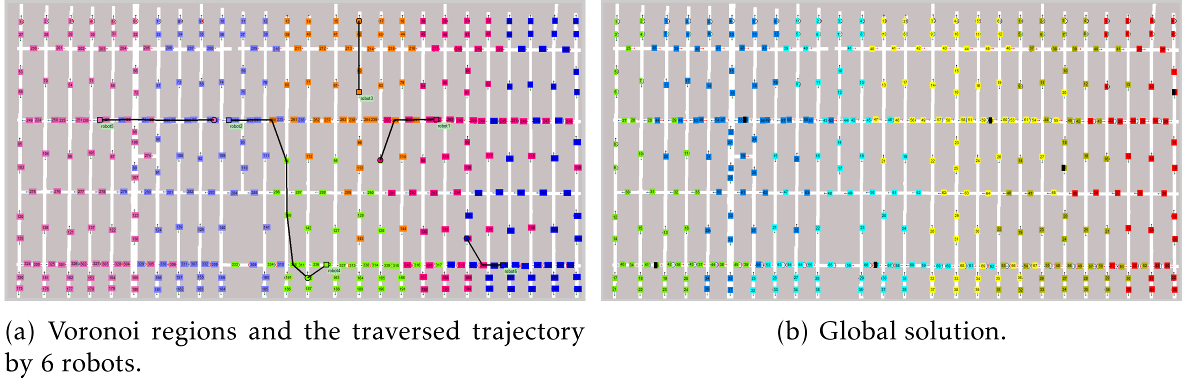
6.1. We defined the center of density function on the node 114, such that Fig. 6.10 represents the density function on this map.

After constructing graph G, we distribute 6 robots randomly over the environment. Robots are equipped with laser sensor to move through the streets by following the curbs. After applying Algorithm 9, robots' final locations, corresponding assigned regions and traversed paths are depicted in Fig. 6.11 (a) with different color. The video of this simulation is available in the following link, http://www.cpdee.ufmg.br//~coro/movies/RezaThesis/.

Table 6.2: Comparison of $\mathcal{H}$ in different methods.

Methodology	Cost($e+5$)	Time
Global Solution	1.12686	more than one hour
Proposed algorithm	1.18786	0.6797 sec

In order to investigate the performance of our on-line method, we performed an



(a) Voronoi regions and the traversed trajectory by 6 robots.

(b) Global solution.

Figure 6.11: Partitioning obtained by the proposed method and solving the MILP model of the p-median problem

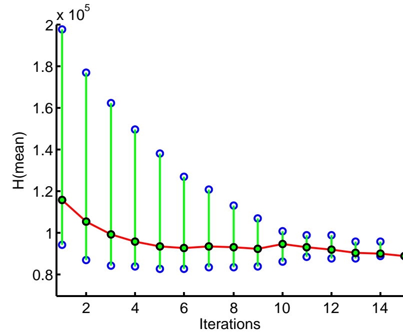


Figure 6.12: Result of simulating the proposed method 100 times on New York scenario. Each column contains mean (red line), max and min of the deployment function cost.

off-line p-median solution (Daskin and Maass, 2015) which is applicable for similar purpose on graphs. In this problem, by considering a graph with $N \times M$ nodes, the objective is to assign M facilities to N customers. One of the methods to solve this NP-hard problem is the Mixed Integer Linear Programming (MILP) which yields the global optimum solution (Sennea et al., 2005).

In this simulation, robots start their movement from the same initial locations that were used in the proposed method. After applying a solver on the MILP model which takes hours for the graph with $|\mathcal{V}| = 347$ we find a result which is shown in Fig. 6.11 (b). Moreover, the value of the $\mathcal{H}$ function and computation time of proposed technique and MILP methods are shown in Table 6.2.

In our algorithm, function $\mathcal{H}$ decreases over time. The result of 100 trial runs of

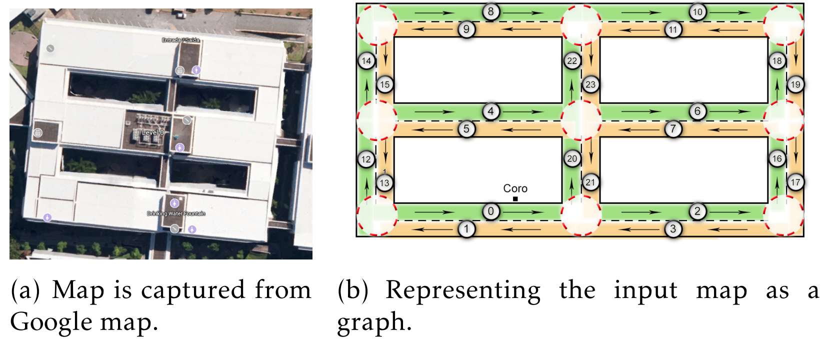
the proposed method on the New York map scenario is illustrated in Fig. 6.12. In each trial the robots are distributed randomly at initial positions, and after executing about 15 iterations, they converged about the final positions. It is important to remark that in order to move in such a big map, in our method only few human-like commands are enough to get from a point (or street) to another.

6.4.2 Real Robot Experiments

After validating the performance of the proposed method through simulations, we performed a real world experiment to verify its applicability in real world scenario. Thus, three robots were chosen to cooperate in a distributed setup. Two of the robots are based on the Pioneer P3-AT [2] mobile base. They are non-holonomic robots with four wheels, used in conjunction with a laser range sensor and odometric sensors. The first Pioneer, which we will call Robot 1, has indoors wheels and a SICK LMS100 laser[3] range sensor, while the second robot, to be called Robot 2, has outdoor wheels and a SICK LMS291 laser range sensor. The third robot, Robot 3, is a small iRobot Create mobile base [4]. It has also odometric sensors and a Hokuyo URG [5] laser range sensor.

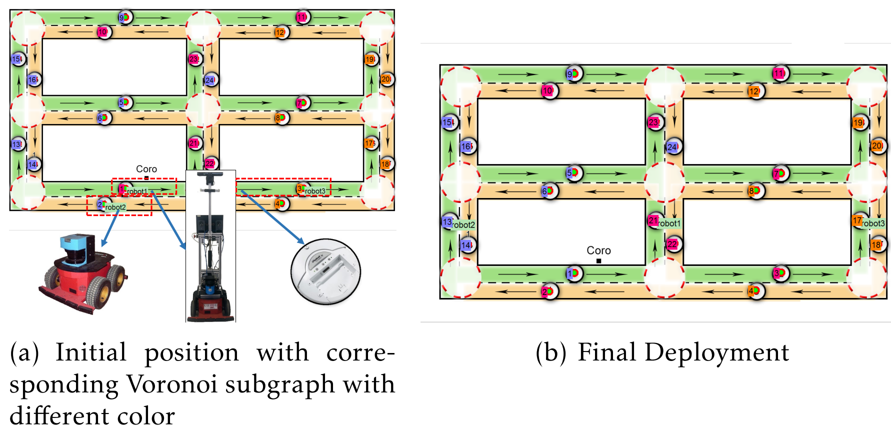
The experiments were executed in the second floor of the School of Engineering building at the Federal University of Minas Gerais [6]. It is a symmetrical building, and is composed of many corridors and intersections (See Fig 6.13 (a)). To build the corresponding graph, the corridors were represented as nodes of the graph, one node for each direction of movement, while edges model the intersections. Each edge received a label with the direction to which the robot had to turn to reach the new corridor in the intersection.

[2] http:www.mobilerobots.comResearchRobotsP3AT.aspx
[3] http:www.sick.com
[4] http:store.irobot.com
[5] https:www.hokuyo-aut.jp
[6] http://www.ufmg.br/



(a) Map is captured from Google map. (b) Representing the input map as a graph.

Figure 6.13: The map used in real robot experiment.



(a) Initial position with corresponding Voronoi subgraph with different color (b) Final Deployment

Figure 6.14: Deployment of 3 robots in a real scenario.

Fig. 6.13 (b) shows the topological map of the environment, with 24 nodes. The corridor at the middle (nodes 4, 5, 6 and 7) is defined as the center of the density function, so the robots are expected to move toward this corridor.

The initial nodes, and the final deployment locations are depicted in Fig. 6.14. Robots 1, 2, and 3 started from nodes 1 to 3 (see Fig. 6.14 (a)), and ended on nodes 21, 13 and 17 (see Fig. 6.14 (b)), respectively. In the Entire execution each robot runs a single command (See Table 6.3). The complete video of this experiment can be found in http://www.cpdee.ufmg.br//~coro/movies/RezaThesis/.

As we mentioned, the experiment was implemented in a decentralized fashion, so that each robot moves and decides individually. During robots motion, they compute their next best node, Whenever they detect a corner or intersection.

Table 6.3: Commands applied on robots in the real experiment.

Robot#	command
1	Turn_Left
2	Turn_Right
3	Turn_Left

6.5 Conclusion

In this work we proposed a distributed multi-robot deployment method based on a topological representation of the environment. A practical technique was applied in order to simplify a multi-dimensional map into a single dimension one. This technique can be used with very large maps, decreasing the dimension and computation cost relatively. Once the topological model of the input map is obtained, robots use a wall following control to move, hence no precise localization is needed. This is done by using a natural scheme of navigation, such that robots move to a location by following a sequence of human like commands. We derived our strategy upon the topological framework proposed by Araujo et al. (2015). In comparison with different techniques found in the literature, our method has a good performance in real applications because of the low computation, no need of precise localization, and no need of high bandwidth communication. These remarkable properties make our method fast enough to be executed in real world scenarios. In comparison to two similar discrete deployment methods found in the literature (Yun and Rus (2013), Durham and Carli (2012)), the proposed approach declines the computational complexity, and requires lower-bandwidth network to exchange locational information (next best node) with neighbor robots, which is usually a challenge in real experiments. As future work, we would like to perform real experiments in multi-floor maps, combining aerial and ground based robots in the same mission. Notice that even in this complex situation the problem can be considered as a 1D problem.

Chapter 7

Conclusion and Future Directions

While in the present work we deal with coordination and cooperation of multi-robot systems, the focus of the study was the *deployment problem*. In multi-robot deployment, finding a set of positions for a group of robots is the objective, such that the total time for responding to "events of interest" inside robots dedicated region is minimized. By investigating the literature, we tried to improve this task in different aspects.

In Chapter 4, we added safety to the original deployment setup. We used the safe roadmap, GVD, and proposed a new metric which forced the robots to move through the GVD. Based on the new metric on GVD we ensure that robots just move on the GVD curves during the deployment, thus safety of the robots motion is guaranteed. Simulation results validated the efficiency of the proposed method. As this method discretizes the environment into cells, in the case of large cells, it will degrade accuracy of robots motion. Also the proposed strategy suffered with the lack of convergence proof, which is addressed in Chapter 5. In this work we were not interested to deal with communication and networking issue, we assumed to have an appropriate network, which guarantees stable connection during the execution of the algorithms. However, it is possible to extend this work in order to consider connectivity maintenance of robots by applying the different techniques proposed by Popa et al. (2004); Reich et al. (2012a); Sreenath et al. (2006); Reich et al. (2012b).

Chapter 5 presents a new algorithm to address the convergence problem in the previous method. Robots move in the direction of the gradient descent vector which guarantees decreasing the cost function. By relying on this property, and also the new technique of assessing robots neighbor nodes in the graph in terms of cost function,

we are able to prove the convergence of the algorithm. Although the environment is discretized (similar to the last method), in this new technique robots move in a continuous manner, thus a more accurate movement, instead of an estimation on center of the cells, is obtained. In order to speedup the execution of the algorithm, in a new implementation based on CUDA, algorithm is run in a parallel way. However, because of tilling the environment it is still expensive in terms of complexity for very large environments.

To address the problem of high computational complexity for large environments in the latest algorithm, in Chapter 6 we extend the discrete setup of deployment problem considering a topological map-based framework (proposed by Araujo et al. (2015)). This framework simplifies the 2-dimensional space to 1D graph space, thus the computational cost declines significantly. In a simulation result we showed that the new deployment strategy resulted a very close result to the global optimum found by a centralized solution. Moreover, this method eliminates the need of accurate localization for the robots motion. Hence, the proposed method is applicable for systems that need fast response. In comparison to time complexity achieved by the most similar works in the literature (Durham and Carli, 2012) and (Yun and Rus, 2013), which were $O(|\mathcal{V}|^3)$ and $O(|\mathcal{V}|^2)$ respectively, our method declines to $O(|\mathcal{V}|\log|\mathcal{V}|)$, where $\mathcal{V}$ is the number of nodes in the graph. Furthermore in the spirit of communication between robots, while in both cited works a considerable amount of data was transferred over the communication network; in (Durham and Carli, 2012) the whole Voronoi partitions between two robots, and in (Yun and Rus, 2013), the information of two-hop communication, in our proposed algorithm robots just exchange the information of the next best node (an integer number). Although the proposed approach is suitable for block-shaped environments, the topological framework was defined in a general form and covers wide range of environments.

In general, in systems where more than one objective is important, we need to set a trade off between the objectives. As shown in this study, the trade off between *complexity* and *cost*, or *speed* and *accuracy* is considered in different scenarios.

Future Directions

- Proposing an algorithm that is able to solve the deployment problem on large environments with reasonable computational complexity.

- Applying the deployment algorithm in unknown environments together with SLAM (Simultaneously Localization And Mapping); as robots move they create the map.

- Implementing the topological map deployment algorithm for multi floor block shape building by adding new commands.

- Considering limitations in network communication in deployment problem.

Publications

Now, we show the list of papers published in the last 42 months as the result of my work and also of collaboration with other students and professors during my PhD studies.

[2016] David Saldana, R. Javanmard Alitappeh, Luciano C. A. Pimenta, Renato Assuncao, and Mario F. M. Campos, Dynamic Perimeter Surveillance with a Team of Robots, Accepted in IEEE International Conference on Robotics and Automation (ICRA 2016)

[2015] K. Jeddisaravi, R. Javanmard Alitappeh, Luciano C. A. Pimenta, Frederico G. Guimaraess, Multi-Objective Approach for Robot Motion Planning in Search Tasks, Applied Intelligence, Vol. (44), pp (1–17)

[2014] R. Javanmard Alitappeh, Luciano C. A. Pimenta, Multi Robot Safe Deployment on GVD Space, Book chapter in Springer Tracts in Advanced Robotics, Vol. (112), pp (65–77)

[2014] K. JeddiSaravi, R. Javanmard Alitappeh, Luciano C. A. Pimenta, Frederico G. Guimaraess, A Multi Objective Approach for Single Robot Environment Exploration, In Proc. of Brazilian Conference on Automation (CBA), Brasil, Belo Horizonte

In Preparation

The following papers are currently in preparation for submission:

[2016] R. Javanmard Alitappeh, Guilherme A. S. Pereira, Arthur R. Araujo, Luciano C. A. Pimenta, Multi-robot deployment using topological maps, Journal of Intelligent and Robotic systems

[2016] Vinicius Graciano Santos , R. Javanmard Alitappeh, Anderson Pires, Luciano C. A. Pimenta, Douglas G. Macharet, Luiz Chaimowicz, Segregation of Robotic Swarms, Journal of Swarm Intelligence

[2016] R. Javanmard Alitappeh, Luciano C. A. Pimenta, New Algorithm for Multi-Robot Deployment, To decide

[2016] R. Javanmard Alitappeh, K. Jeddisaravi, Luciano C. A. Pimenta, Luiz Chaimowicz, Heterogeneous Multi-Robot Exploration in Search Task, To decide

[2016] R. Javanmard Alitappeh, David Saldana, Luciano C. A. Pimenta, Motion planning and Online Adaptation for Dynamic Perimeter Surveillance with a Team of Robots,To decide.

Bibliography

Alliez, P., Ucelli, G., Gotsman, C., and Attene, M. (2008). Recent advances in remeshing of surfaces. In *Shape Analysis and Structuring*, pages 53–82.

Araujo, A. R., Caminhas, D. D., and Pereira, G. A. (2015). An architecture for navigation of service robots in human-populated office-like environments. *11th IFAC Symposium on Robot Control (SYROCO) 2015, IFAC-PapersOnLine*, 48(19):189–194.

Bhattacharya, S., Ghrist, R., and Kumar, V. (2013a). Multi-robot Coverage and Exploration in Non-Euclidean Metric Spaces. *Algorithmic Foundations of Robotics X, Springer Tracts in Advanced Robotics*, 86:245–262.

Bhattacharya, S., Ghrist, R., and Kumar, V. (2013b). Multi-robot coverage and exploration on Riemannian manifolds with boundaries. *The International Journal of Robotics Research*, 33(1):113–137.

Bhattacharya, S., Michael, N., and Kumar, V. (2013c). Distributed coverage and exploration in unknown non-convex environments. *Distributed Autonomous Robotic Systems, Springer Tracts in Advanced Robotics*, 83:61–75.

Breitenmoser, A. (2010). Voronoi coverage of non-convex environments with a group of networked robots. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4982–4989, Anchorage, Alaska.

Bullo, F., Cortés, J., and Martínez, S. (2009). *Distributed Control of Robotic Networks*. Applied Mathematics Series. Princeton University Press.

Caicedo-Nunez, C. and Zefran, M. (2008a). Performing coverage on nonconvex domains. In *Proc. of the IEEE International Conference on Control Applications (CCA)*, pages 1019–1024.

Caicedo-Nunez, C. H. and Zefran, M. (2008b). A coverage algorithm for a class of nonconvex regions. In *Proc. of the IEEE Conference on Decision and Control (DCC)*, pages 4244–4249.

Choset, H., Lynch, K. M., Hutchinson, S., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). *Principles of Robot Motion: Theory, Algorithms, and Implementation*. MIT Press, Boston.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001). *Introduction to Algorithms, third edition*. Cambridge, MA: McGraw-Hill., 2nd editio edition.

Cortes, J., Martinez, S., Karatas, T., and Bullo, F. (2004). Coverage control for mobile sensing networks. *IEEE Transaction on Robotics and Automation*, 20(2):243–255.

d'Andréa Novel, B., Campion, G., and Bastin, G. (1995). Control of nonholonomic wheeled mobile robots by state feedback linearization. *The International Journal of Robotics Research*, 14(6):543–559.

Daskin, M. and Maass, K. (2015). The p-median problem. In *Location Science*, pages 21–45. Springer International Publishing.

Desai, J., Ostrowski, J., and Kumar, V. (1998). Controlling formations of multiple mobile robots. volume 4, pages 2864–2869. IEEE.

Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische mathematik*, 1(l 959):269–271.

Du, Q., Faber, V., and Gunzburger, M. (1999). Centroidal voronoi tessellations: Applications and algorithms. *SIAM Review*, 41(4):637–676.

Durham, J. and Carli, R. (2012). Discrete partitioning and coverage control for gossiping robots. *IEEE Transactions on Robotics*, 28(2):364 – 378.

Floyd, R. W. (1962). Algorithm 97: Shortest path. *Commun. ACM*, 5(6):345–.

Fortune, S. (1987). A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1-4):153–174.

Hakimi, S. L. (1964). Optimum locations of switching centers and the absolute centers and medians of a graph. *Operations Research*, 12(3):450–459.

Harish, P. and Narayanan, P. J. (2007). Accelerating large graph algorithms on the gpu using cuda. In *Proc. of the 14th International Conference on High Performance Computing (HiPC)*, pages 197–208, Berlin, Heidelberg. Springer Berlin Heidelberg.

Haumann, D., Breitenmoser, A., Willert, V., Listmann, K., and Sieqwart, R. (2011). DisCoverage for non-convex environments with arbitrary obstacles. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 4486–4491, Shanghai, China. IEEE.

Haumann, D., Listmann, K. D., and Willert, V. (2010). DisCoverage: A new paradigm for multi-robot exploration. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 929–934. IEEE.

Hoff, K. E., Culver, T., Keyser, J., Lin, M., and Manocha, D. (1999). Fast computation of generalized Voronoi diagrams using graphics hardware. In *Proc. of the 26th annual conference on Computer graphics and interactive techniques (SIGGRAPH '99)*, pages 277–286.

Howard, A., Matari, M. J., and Sukhatme, G. S. (2002). Mobile Sensor Network Deployment using Potential Fields : A Distributed , Scalable Solution to the Area Coverage Problem. In *Distributed Autonomous Robotic Systems*, pages 299–308. Springer Japan.

Ji, M. and Egerstedt, M. (2007). Distributed coordination control of multiagent systems while preserving connectedness. *IEEE Transactions on Robotics*, 23(4):693–703.

Katz, G. and Kider, J. T. (2008). All-pairs shortest-paths for large graphs on the GPU. In *Proc. of 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55.

Lam, C.-P., Chou, C.-T., Chiang, K.-H., and Fu, L.-C. (2011). Human-centered robot navigation towards a harmoniously human-robot coexisting environment. *IEEE Transactions on Robotics*, 27(1):99–112.

Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137.

Madduri, K., Bader, D., Berry, J., and Crobak, J. (2007). An Experimental Study of A Parallel Shortest Path Algorithm for Solving Large-Scale Graph Instances. In *Ninth Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*.

Mahboubi, H. and Sharifi, F. (2012). Distributed coordination of multi-agent systems for coverage problem in presence of obstacles. In *Proc. of the American Control Conference (ACC)*, pages 5252–5257.

Martin, P. J., Torres, R., and Gavilanes, A. (2009). Cuda solutions for the sssp problem. pages 904–913.

Meyer, U. and Sanders, P. (2013). Δ-stepping: A parallel single source shortest path algorithm. *Journal of Algorithms*, 49:114–152.

Mladenovic, N., Brimberg, J., Hansen, P., and Moreno-Perez, J. A. (2007). The p-median problem: A survey of metaheuristic approaches. *European Journal of Operational Research*, 179(3):927 – 939.

NVIDIA (2007). NVIDIA CUDA Compute unified device architecture: programming guide.

NVIDIA (2014). Compute Unified Device Architecture (CUDA) 6, http://devblogs.nvidia.com/parallelforall/powerful-new-features-cuda-6/.

NVIDIA, C. (2011). Nvidia CUDA C programming guide. *NVIDIA Corporation*.

Ny, J. L. and Pappas, G. (2013). Adaptive deployment of mobile robotic networks. *IEEE Transactions on Automatic Control*, 58(3):654–666.

Okabe, A., Boots, B., and Sugihara, K. (1992). *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. John Wiley & Sons, Inc.

Okuyama, T., Ino, F., and Hagihara, K. (2012). A task parallel algorithm for finding all-pairs shortest paths using the GPU. *International Journal of High Performance Computing and Networking*, 7(2):87–98.

Parker, L. E. (2002). Distributed algorithms for multi-robot observation of multiple moving targets. *Autonomous robots*, 12(3):231–255.

Pierson, A., Figueiredo, L., Pimenta, L., and Schwager, M. (2015). Adapting to performance variations in multi-robot coverage. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 415–420.

Pimenta, L. C. A., Kumar, V., Mesquita, R. C., and Pereira, G. A. S. (2008). Sensing and coverage for a network of heterogeneous robots. In *Proc. of the IEEE Conference on Decision and Control (DCC)*, number 2, pages 3947–3952, Cancun, Mexico. IEEE.

Pimenta, L. C. a., Pereira, G. a. S., Michael, N., Mesquita, R. C., Bosque, M. M., Chaimowicz, L., and Kumar, V. (2013). Swarm coordination based on smoothed particle hydrodynamics technique. *IEEE Transactions on Robotics*, 29(2):383–399.

Pimenta, L. C. A., Schwager, M., and Lindsey, Q. (2010). Simultaneous coverage and tracking (SCAT) of moving targets with robot networks. *Algorithmic Foundation of Robotics VIII, Springer Tracts in Advanced Robotics*, 57:85–99.

Poduri, S. and Sukhatme, G. (2004). Constrained coverage for mobile sensor networks. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA)*, pages 165–171, New Orlans, LA, USA. IEEE.

Popa, D., Stephanou, H., Helm, C., and Sanderson, A. (2004). Robotic deployment of sensor networks using potential fields. In *Proc. of the IEEE International Conference on Robotics and Automation, (ICRA)*, pages 642–647, New Orlans, LA, USA. IEEE.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. (2009). ROS: an open-source Robot Operating System. In *Proc. of the IEEE International Conference on Robotics and Automation (ICRA) Workshop on Open Source Software*.

Reese, J. (2006). Solution methods for the p-median problem: An annotated bibliography. *Networks*, 48(3):125–142.

Reich, J., Misra, V., Rubenstein, D., and Zussman, G. (2012a). Connectivity maintenance in mobile wireless networks via constrained mobility. *IEEE Journal on Selected Areas in Communications*, 30(5):935–950.

Reich, J., Misra, V., Rubenstein, D., and Zussman, G. (2012b). Connectivity maintenance in mobile wireless networks via constrained mobility. *Selected Areas in Communications, IEEE Journal on*, 30(5):935–950.

Reif, J. H. and Wang, H. (1999). Social potential fields: A distributed behavioral control for autonomous robots. *Robotics and Autonomous Systems*, 27(3):171–194.

Rolland, E., Schilling, D., and Current, J. (1996). An efficient tabu search procedure for the p-median problem. *European Journal of Operational Research*, 96:329–342.

Rong, G., Liu, Y., Wang, W., and Yin, X. (2011). GPU-assisted computation of centroidal Voronoi tessellation. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):345–356.

Rong, G. and Tan, T.-S. (2007). Variants of Jump Flooding Algorithm for Computing Discrete Voronoi Diagrams. In *Proc. of the 4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, pages 176–181. IEEE.

Sanders, J. and Kandrot, E. (2010). *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley.

Santiago, G., Moreno, L., Dolores, B., and Piotr, J. (2011). Path planning for mobile robot navigation using voronoi diagram and fast marching. *International Journal of Robots and Automation(IJRA)*, 2(1):42–64.

Schwager, M., Rus, D., and Slotine, J. (2009). Decentralized, adaptive coverage control for networked robots. *International Journal of Robotics Research*, 28(3):357–375.

Schwager, M., Rus, D., and Slotine, J. J. (2011). Unifying geometric, probabilistic, and potential field approachs to multi-robot deployment. *International Journal of Robotics Research*, 30(3):371–383.

Sennea, E., Lorenab, L. A., and Pereirab, M. A. (2005). A branch-and-price approach to p-median location problems. *Computers and Operations Research*, 32:1655–1664.

Sharifi, F., Chamseddine, A., Mahboubi, H., Zhang, Y., and Aghdam, A. G. (2015). A Distributed Deployment Strategy for a Network of Cooperative Autonomous Vehicles. *IEEE Transactions on Control Systems Technology*, 23(2):737–745.

Sharifi, F., Zhang, Y., and Aghdam, A. G. (2014). A Distributed Deployment Strategy for Multi-Agent Systems Subject to Health Degradation and Communication Delays. *Journal of Intelligent and Robotic Systems*, 73(1):623–633.

Sreenath, K., Lewis, F. L., and Popa, D. O. (2006). Localization of a wireless sensor network with unattended ground sensors and some mobile robots. In *IEEE International Conference on Robotics, Automation, and Mechatronics (RAM)*, pages 1–8, Bangkok, Thailand. This paper won the IEEE RAM Conf. Best Paper Award 2006.

Stergiopoulos, Y. and Tzes, A. (2011). Coverage-oriented coordination of mobile heterogeneous networks. In *In Proc. of Mediterranian Control & Automation (MED)*, pages 175–180.

Sud, A., Andersen, E., Curtis, S., Lin, M. C., and Manocha, D. (2008). Real-time path planning in dynamic virtual environments using multiagent navigation graphs. *IEEE Transactions on Visualization and Computer Graphics*, 14(3):526–538.

Sud, A., Naga K. Govindaraju, Gayle, R., Kabul, I., and Dinesh, M. (2006). Fast proximity computation among deformable models using discrete Voronoi diagrams. *ACM Transactions on Graphics (Proceedings of ACM SIG- GRAPH 2006)*, 25(3):1144–1153.

Sutter, H. and Alexandrescu, A. (1998). http://www.boost.org/.

Suzuki, T. and Kawabata, K. (2008). Deployment of Wireless Sensor Network using Mobile Robots to Construct an Intelligent Environment in a Multi-Robot Sensor Network. In *Advances in Service Robotics*. INTECH.

Tzes, A. and Stergiopoulos, Y. (2010). Convex Voronoi-inspired space partitioning for heterogeneous networks: a coverage-oriented approach. *IET Control Theory & Applications*, 4(12):2802–2812.

Venkataraman, G. and Mukhopadhyaya, S. (2003). A blocked all-pairs shortest-paths algorithm. *Journal of Experimental Algorithmics*, 5.

Xin, S.-Q., Wang, X., Xia, J., Mueller-Wittig, W., Wang, G.-J., and He, Y. (2013). Parallel computing 2D Voronoi diagrams using untransformed sweepcircles. *Computer-Aided Design*, 45(2):483–493.

Yun, S.-k. and Rus, D. (2013). Distributed coverage with mobile robots on a graph: locational optimization and equal-mass partitioning. *Robotica*, 32(02):257–277.