POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica

Algorithms for limited-buffer shortest path problems in communication-restricted environments

Relatore: Prof. Francesco Amigoni
Correlatori: Ing. Alessandro Riva, Ing. Jacopo Banfi

Tesi di Laurea Magistrale di:
Arlind Rufi, matricola 836735

Anno Accademico 2016-2017

Alla mia famiglia.

Sommario

In diverse applicazioni, un robot, muovendosi da un posizione di partenza ad una finale, deve raccogliere dati durante il suo percorso (per esempio, un video in un scenario di monitoraggio). Il robot può avere a disposizione solo una quantità limitata di memoria per salvare i dati raccolti, per tenere i costi bassi o per motivi di sicurezza (per evitare che dati sensibili cadano nelle mani di un aggressore). Questo pone la necessità di trasmettere periodicamente i dati ad una Base Station (BS) tramite un'infrastruttura di comunicazione, che, in generale, non è disponibile ovunque. In questa tesi, studiamo questo scenario considerando una variante dello shortest path problem (che dimostriamo essere NP-hard) dove il robot acquisisce informazioni lungo il percorso, li salva in una memoria limitata e si assicura che nessuna informazione venga persa, trasmettendo periodicamente alla BS. Presentiamo e valutiamo, un algoritmo ottimale, un efficiente test di feasibility e un algoritmo euristico.

Abstract

In several applications, a robot moving from a start to a goal location is required to gather data along its path (e.g., a video feed in a monitoring scenario). The robot can have at its disposal only a limited amount of memory to store the collected data, in order to contain costs or to avoid that sensible data fall into the hands of an attacker. This poses the need of periodically delivering the data to a Base Station (BS) through a deployed communication infrastructure that, in general, is not available everywhere. In this thesis, we study this scenario by considering a variant of the shortest path problem (which we prove to be NP-hard) where the robot acquires information along its path, stores it into a limited memory buffer, and ensures that no information is lost by periodically communicating data to the BS. We present and evaluate an optimal algorithm, an efficient feasibility test, and a polynomial time heuristic algorithm.

Ringraziamenti

Desidero ringraziare tutti coloro che mi hanno aiutato nella stesura con suggerimenti, critiche ed osservazioni: a loro va la mia gratitudine, anche se a me spetta la responsabilità per ogni errore contenuto in questo lavoro.

Ringrazio anzitutto il professor Francesco Amigoni, Relatore, e gli ingegneri Alessandro Riva e Jacopo Banfi, Co-relatori: senza il loro supporto e la loro guida sapiente questa tesi non esisterebbe.

Vorrei infine ringraziare le persone a me più care: i miei amici e la mia famiglia.

Contents

Sommario I

Abstract III

Ringraziamenti V

1 Introduction 1

2 State of Art 5

2.1 Mobile Robot Navigation 5

2.1.1 Environment Representation 6

2.1.1.1 Occupancy Grid Map 6

2.1.1.2 Visibility Graph 7

2.1.1.3 Potential Fields 7

2.1.1.4 Sample Based 8

2.1.2 Self Localization 9

2.1.3 Shortest Path Planning 11

2.1.3.1 BFS 11

2.1.3.2 DFS 12

2.1.3.3 Dijkstra 13

2.1.3.4 A* 13

2.1.3.5 Theta* 15

2.1.4 Robot Motion Planning 17

2.1.4.1 DWA and Trajectory Planner 19

2.2 Constraint Shortest Path Problem and its Variants 20

2.2.1 Integer Liner Programming 20

2.2.2 Dynamic programming 21

2.2.3 The Pulse Algorithm 22

2.2.4 Acceleration strategies for the weight constrained shortest path problem with replenishment 24

2.2.5 Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles . . . 24

3 Problem Setting **27**

3.1 NP-hardness . . . 30

4 Algorithms **33**

4.1 Optimal Algorithm . . . 33

4.2 Feasibility Test . . . 35

4.3 A Heuristic Algorithm . . . 36

5 Implementation **39**

5.1 Algorithms Implementation . . . 39

5.2 Implementation as a ROS Planner . . . 40

5.2.1 What is ROS? . . . 41

5.2.1.1 Navigation Stack . . . 41

5.2.2 Algorithms implementation in ROS . . . 45

6 Experiments **47**

6.1 Experiments Settings . . . 47

6.2 Experiment Set 1 . . . 50

6.3 Experiment Set 2 . . . 54

6.4 ROS Experiments . . . 58

6.4.1 ROS used Tools . . . 58

6.4.2 Experiments . . . 60

7 Conclusions **63**

A Class Diagram **65**

B ROS complete list packages and ROS configuration **67**

List of Algorithms

2.1 BFS pseudocode 12
2.2 DFS pseudocode(recursive implementation) 13
2.3 Dijkstra pseudocode 14
2.4 A* pseudocode(recursive implementation) 16
2.5 Theta* pseudocode (only lines different from A*) 18
2.6 Pulse Algorithm 23
2.7 Pulse function 23

4.1 Graph Transformation 34
4.2 Find Times 38

List of Figures

2.1 Grid example: **Black** real obstacles, **White** free cells, **Grey** occupied Cells. 6

2.2 Visibility graph example. 7

2.3 Potential field example: *Obst* produces a repelling field and *Goal* an attracting field. 8

2.4 Sampling algorithm example: RRT. 9

2.5 GPS trilateration. 10

2.6 BFS Example 11

2.7 DFS example 12

2.8 Dijkstra example. 14

2.9 Theta* vs A*: On the left, in red, the path produced by A*, on the right, in blue, the path produced by Theta*. 17

2.10 DWA and trajectory planner example. 19

3.1 Example of an environment. Red vertices represent the *transmission vertices*, r represents the *transmission-rate function* of each node, and $\bar{r}$ represents the information unstacked from the buffer if the robots moves on the edge 28

3.2 Example of a feasible solution and infeasible solution. Starting from the green vertex with an empty buffer and going to g the blue vertex. Buffer size of the robot is B=5. Following the green edges and stopping for two time steps at node C_{33} to transmit the robot ends at the blue node with a full buffer. Following the purple edges we have a total accumulation of information of 6 which makes the path infeasible. 29

4.1 Example of a Graph transfomation with buffer B=4 considering the graph presented in Figure 3.1. (To keep the figure simple only a few nodes are considered and not all the edges are shown). After building the transformed graph applying Dijkstra's algorithm the path in green would be the optimal solution. 35
4.2 Example G_2 based on the graph presented in Figure 3.1 with a buffer size 4. In this case there exist a solution because there exists a path connecting start and goal nodes. 36
4.3 Example of feasible but not optimal path, found using the feasibility approach, shown on G_1 (that is build based on the graph presented in Figure 3.1). The feasible path starts from the cell in green, ends at the cell in blue and is composed by the edges in purple and yellow. The optimal path would be the path composed by edges in purple and in green. 37

5.1 Result of the pre-transformation of the graph in Figure 3.1. 40
5.2 Simple example of a publish-subscribe and service invocation architecture. 41
5.3 A common navigation stack setup. 42

6.1 Experimental environment (size 400×300 m). Red discs represent the communication zones (e.g, areas covered by RF transceivers). 48
6.2 Communication map with buffer discretization 1. 48
6.3 Communication map with buffer discretization 0.25. 49
6.4 Example of path generated by our optimal algorithm using as local planner Theta* (in blue) and using as a local planner 4-connected grid with Dijkstra's (in green). 51
6.5 First set of Experiments Start and Goal positions 51
6.6 Results of the experiments with buffer discretization 1. 52
6.7 Results of the experiments with buffer discretization 0.25. . . 53
6.8 Example paths of the first set of experiments. (a) First set (blue: $B = 21$, green: $B = 29$, brown: $B = 41$); (b) Second set (blue: $B = 17$, green: $B = 26$, brown: $B = 40$). 55
6.9 Results of the second set of experiments on 100 randomly selected pairs of start-goal locations. Feasible solutions depending on the buffer size. 56

6.10 Results of the second set of experiments on 100 randomly selected pairs of start-goal locations. Heuristic solution worsening in comparison with the optimal algorithm. 57
6.11 Stage view of the environment 59
6.12 Rviz view of the environment. 60
6.13 Result of the ROS experiments, optimal algorithm. 61
6.14 Result of the ROS experiments, heuristic algorithm. 62

A.1 ER Diagram of the C++ implementation. (Baseline is implementation of the heuristic algorithm.) 66

Chapter 1

Introduction

In several applications, a robot moving from a start to a goal location may be required to gather data along its path. This happens, for instance, in some monitoring applications, where the robot acquires a video feed to be later processed at a Base Station (BS) [1] or in those civilian or military settings where *a posteriori* processing of log files is required to ensure that the robot has not been hijacked by a malicious attacker [2].

The robot can have at its disposal only a limited amount of memory to store the collected information. In civilian settings, this could be motivated by the need to reduce costs, while, in military settings, this could be enforced to avoid that a large amount of sensible data fall into the hands of a malicious attacker. This memory limitation poses the need of periodically transmitting data to a BS in order to not overfill the available memory buffer.

In most application settings, it is unrealistic to assume the presence of a robust communication infrastructure able to uniformly cover the environment with the same (high) transmission rate. Typical settings can instead rely only on the presence of a limited number of "communication zones" from where robots can reliably communicate with the BS [1]. Moreover, such zones could display a significant variability in the data transfer rate, also according to the distance from the communication device [3]. The communication paradigm we use is introduced in [1] in the context of multirobot patrolling. It assumes the presence of a number of 'communication zones' that robots can exploit to communicate with the BS. We further refine the model by associating different zones with (possibly) different data transmission rates. The combination of requirements imposed by incremental data acquisition, limited memory, and restricted communication define a challenging path planning scenario.

In this thesis, we study a variant of the shortest path planning problem with the goal of planning high-level paths (defined in terms of waypoints to traverse) under the constraints imposed by the above scenarios. Specifically, the problem we consider is to compute the shortest path between given start and goal locations in a graph representing the environment, while (a) accounting for the storage of the gathered information into a buffer of limited size and (b) ensuring that no information is lost through periodic transmissions of data from communication zones to the BS.

In literature, different works try to incorporate in the path planning problem some additional (possibly application-dependent) constraints. For instance, [4] is one of the first works to propose a path planning algorithm (inspired to A*) able to cope with additional constraints, such as time, risk, energy, and uncertainty. An example of a more recent work is [5], which studies path planning for a solar-powered robot subject to time and energy constraints. However, communication issues in path planning have been mainly investigated for multirobot systems, especially in the context of maintaining (more or less periodically) the team of robots globally connected through a multi-hop network while pursuing a primary mission objective (see, for instance, the solution devised in [6] for informative path planning under periodic connectivity constraints). Our problem shares similarities with variants of the Constrained (or restricted) Shortest Path problem (CSP) [7]. The work in [8] investigates a generalization of the CSP where edges may be associated to binary indicators able to reset the accumulated weight when the corresponding edge is traversed. This model is not applicable in our case, since we have to deal with the possibility of transmitting only a portion of the accumulated data. Authors of [9], instead, propose a model that partially generalizes ours in the context of planning shortest paths with charging stops for electric vehicles (the filling of our buffer can be naively thought as dual to the draining of the battery). However, in our case we must consider cycles with negative cumulative weight.

After formalizing the problem and proving its NP-hardness, we present three original contributions:

1. An optimal exponential time solving algorithm.
2. An efficient feasibility test that can be applied to all problem instances to check if they admit a solution.
3. A polynomial time heuristic algorithm based on the refinement of a "raw" feasible solution.

To evaluate and validate our algorithms we designed three types of experiments, each with a different purpose. In particular, we consider a fixed environment and repeat the first two sets with different discretization parameters. The idea behind the first set of experiments is to evaluate how the path changes in dependence of the memory of the robot, if start and goal are fixed points. The idea behind the second set, instead, is to evaluate the computational time, and the solution cost, chosing random start and goal points. The aim of the third set of experiments is to further validate our algorithms in more real scenarios implementing them using ROS [10] (Robot Operating System).

This thesis is structured as follows:

- Chapter 2 (**State of Art**). In this chapter we give a general description of the mobile robot navigation problem, of the Constraint Shortest Path problem (CSP) and describe some of the existing algorithms used to solve problems similar to ours.
- Chapter 3 (**Problem Setting**). In this chapter we formally define our environment and problem setting. We also prove that our problem is NP-hard.
- Chapter 4 (**Algorithms**). In this chapter we describe the proposed algorithms, along with their pseudocode and execution example.
- Chapter 5 (**Implementation**). In this chapter we describe the implementation of our algorithms in C++ and their adaption to the ROS architecture.
- Chapter 6 (**Experiments**). In this chapter we describe our three sets of experiments and their results.
- Chapter 7 (**Conclusion**). In this chapter we do a quick recapitulation of the thesis and describe few ideas about possible future research.

Chapter 2

State of Art

The problem we address in this thesis shares some similarities with some optimization problems studied both in robotics and in other research fields. Planning high-level paths is a fundamental ability that a mobile robot needs to possess [11]. Without constraints, this problem becomes the classical path planning problem of finding the shortest path between two points. So this chapter begins by giving a general description of the mobile robot navigation. We start by describing some of the main techniques used to represent a given environmnet and to locate the robot in the environment. Furthermore, we present some of the main algorithms to find the shortest path without constraints in the environment.

At an algorithmic level, the problem we investigate can be framed in the class of the *shortest path problems with resource constraints* [12]. In particular it shares similarities with variants of the Constraint Shortest Path problem (CSP). Therefore, we then present the CSP and some of its variants, which are closest to our model, studied in the literature.

2.1 Mobile Robot Navigation

In order to be able to plan and follow a path, a robot needs a way to perceive the environment where it is put into, a way to locate itself in this environment, a planner to plan the high-level path using different algorithms depending on the environment representation, and a motion planner to follow the produced path. So, in this section, we describe some of the main solutions for each of these problems.

2.1.1 Environment Representation

The environment is usually represented as a state space. The state space for motion planning is a set of possible states that could be reached by the robot. This will be referred to as the configuration space (C_{space}). $C_{obstacle}$ is a subset of the configuration space that includes obstacles and unobtainable configurations (subset of free states which the robot can not reach for different reasons e.g., physical limitations). C_{free} is the remaining subset of C_{space}. There are four main techniques to build the C_{space} [11].

2.1.1.1 Occupancy Grid Map

Occupancy grid map represents the environment as an array of cells. The grid cells can be occupied, when a part of an obstacle is in the cell, or free. The C_{space} is built using the array of the cells. $C_{obstacle}$ corresponds to the occupied cells (usually colored in black/grey) and the C_{free} corresponds to the free cells (usually colored in white). Figure 2.1 shows a possible grid-based discretization of a simple environment.

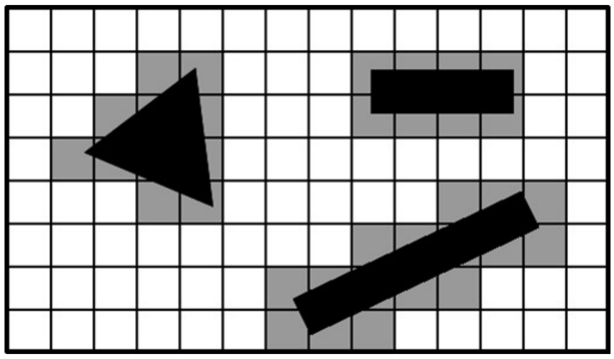


Figure 2.1: Grid example: **Black** *real obstacles,* **White** *free cells,* **Grey** *occupied Cells.*

Occupancy grid maps depend on the resolution chosen. A fine grained occupancy grid uses large amounts of memory and planning in these grids is slower. Coarse grained occupancy grid uses less memory and planning in these grids is faster but some environment features (e.g., narrow paths) may be lost. So careful consideration must be made when choosing the resolution.

2.1.1.2 Visibility Graph

Visibility graph is a graph of intervisible locations, typically build based on a set of points and obstacles in the Euclidean plane. Each node in the graph represents a point location, and each edge represents a visible connection between them. That is, if the line segment connecting two locations does not pass through any obstacle, an edge is drawn between them in the graph. Visibility graphs are used to find the Euclidean shortest path [13]. By the definition of the visibility graph, in the configuration space we only have C_{free}. Figure 2.2 shows a simple visibility graph.

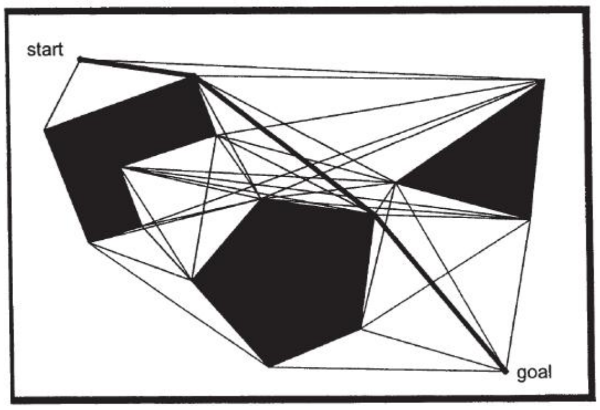


Figure 2.2: Visibility graph example.

The visibility graph can have a quadratic number of edges. There exist different algorithms to build the visibility graph which run from $O(n^3)$, for the most naive one that compares every pair of nodes in the set of obstacles and checks if they intersect with any edges of obstacles, to more faster ones that can go to $O(n^2 \log n)$ [13].

2.1.1.3 Potential Fields

The idea behind this method is to build potential fields in the C_{space} so that the point that represents the robot is attracted to the goal and repelled by the $C_{obstacle}$ zone. Representing the environment this way has the advantage that producing a trajectory takes little computation. Motion planning in potential fields is done by computing the artificial force at the current configuration, moving by a small step at the direction of the force and repeating the process [14]. Figure 2.3 (taken from [15]) shows a simple environment, containing just the goal and start nodes, represented by potential fields method.

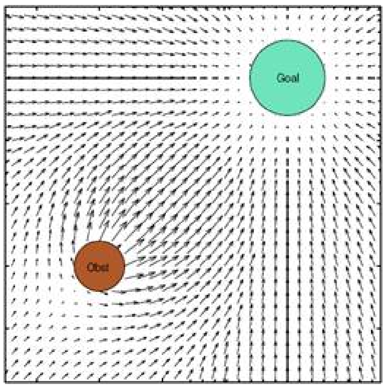


Figure 2.3: Potential field example: $Obst$ produces a repelling field and $Goal$ an attracting field.

However, potential field method can create local minima of the potential field where the robot gets traped and fails to find a path. Hence motion planning based on artificial potential fields is not complete. There are different algorithms in literature, that deal with this problem, like for example Random Motions [14] where, when blocked at a local minima, we take a random step towards another configuration.

2.1.1.4 Sample Based

Sampling-based algorithms represent the configuration space with a roadmap of sampled configurations. A basic algorithm samples N configurations in the configuration space C_{space}, and retains those in the free space C_{free} to use as milestones. A roadmap is then constructed that connects two milestones P and Q, if the line segment PQ is completely in the free space. Again, collision detection is used to test inclusion in the free space. To find a path that connects start and goal, they are added to the roadmap. If a path in the roadmap links start and goal, the planner succeeds, and returns the path. If a path that connects start and goal is not found, the reason is not definitive: either there is no path in the free space, or the planner did not sample enough milestones. One algorithm that uses sampling is RRT [16]. Figure 2.4 (taken from [17]) shows an excecution of RRT.

These algorithms work well with high-dimensional configuration spaces, because unlike combinatorial algorithms, their running time is not (explicitly) exponentially dependent on the dimension of configuration space. They are also (generally) substantially easier to implement. They are probabilistically complete, meaning the probability that they will produce a solution

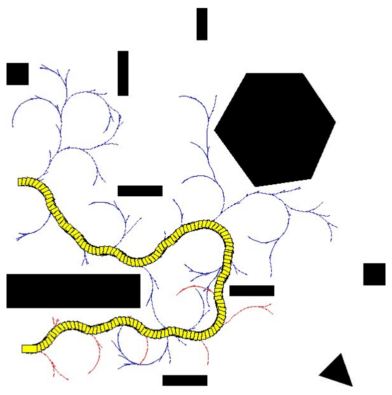


Figure 2.4: Sampling algorithm example: RRT.

approaches 1 as more time is spent. However, they cannot determine if no solution exists.

2.1.2 Self Localization

Robot localization denotes the robot's ability to establish its own position and orientation within a given frame of reference. Methods in locating the position of a robot can be divided in two groups: relative methods and absolute ones.

Relative methods consist in providing a change in position, to a previously known position. Acquiring relative measurements, dead reckoning, is the process of estimating the position of a robot based on the speed, the direction of travel, and the time passed since the last known position. Since the position estimates are based on earlier positions, the error in the estimates increases over time. Some relative methods are:

- Odometry [18] - is a navigation technique which uses the rotation of the wheels to track the position of a robot. Odometry gives good short-term accuracy, is inexpensive, and allows for very high sampling rates but, due to slippage of the wheels, terrain sensivity, or differences in wheel diameter, errors can occur and with time they accumulate and cause large position errors.

- Inertial navigation [19] - is a self-contained navigation technique in which measurements provided by accelerometers and gyroscopes are used to track the position and orientation of an object relative to a known starting point. Gyroscopes give a measurement of rate of rotation of the object, and accelerometers measure linear accelerations along the x or y axis of the object. Like odometry, position estimates drift over time, and thus the errors increase without bound.

Absolute methods consist in supplying information about the location of the robot directly from one measurement. Some absolute methods are :

- GPS - The Global Positioning System is a global navigation satellite system that provides geolocation information to a GPS receiver anywhere on or near Earth where there is an unobstructed line of sight to three or more GPS satellites [20]. There are few problems with GPS because of the fact that transmissions are line-of-sight and signals from satellites can be refracted or other signals can interfere. Figure 2.5 (taken from [21]) shows how GPS works using trilateration.

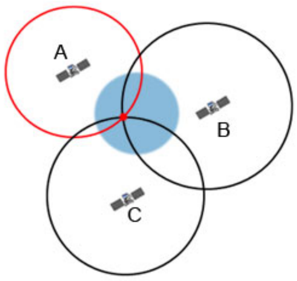


Figure 2.5: GPS trilateration.

- Landmarks or Beacons - Active landmarks (beacons) send out the position information, and the location is obtained with triangulation or trilateration. Passive landmarks have to be detected, for instance with vision.

- Map based - Map based positioning uses geometric features (lines that describe walls, ...) of the environment to compute the location of the robot. Sensor output is then matched with these features. Model matching can be used to update a global map. It needs enough sensor information and it requires large amounts of processing power. An interesting work about map based localization can be found in [22].

2.1.3 Shortest Path Planning

In graph theory, the shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. The problem of finding the shortest path between two intersections on a road map (the graph's vertices correspond to intersections and the edges correspond to road segments, each weighted by the length of its road segment) may be modeled by a special case of the shortest path problem in graphs. There exist different planning algorithms used to find the shortest path between two nodes. We present some of them. A planning algorithm is complete if it can find a path in finite time, if it exists. Similarly, a planning algorithm is optimal if it can always find an optimal path.

2.1.3.1 BFS

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at some arbitrary node of a graph and explores the neighbor nodes first, before moving to the next level neighbors [23]. A simple node expansion using breadth-first search strategy is shown on Figure 2.6 (taken from [24]).

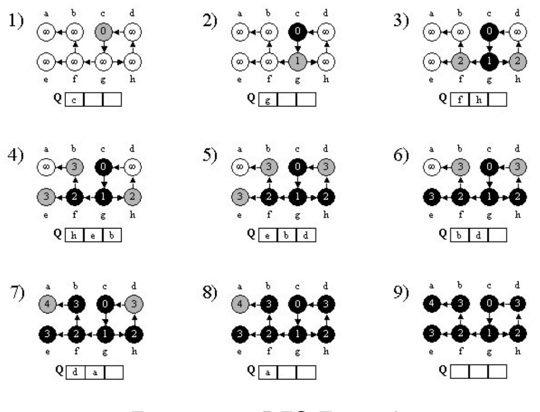


Figure 2.6: BFS Example

The time complexity can be expressed as $O(|V|+|E|)$. $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. Note that $O(|E|)$ may vary between $O(1)$ and $O(|V|^2)$, depending on how sparse the input graph is [25]. The pseudocode of the BFS algorithm is shown in Algorithm 2.1.

Breadth-first search is complete and optimal.

Algorithm 2.1: BFS pseudocode

```
Input: A graph $G$ and a starting vertex $root$ of $G$
Output: Goal state. The $parent$ links trace the shortest path back to $root$
function BFS($G, root$)
    $Q$(queue)$\leftarrow root$
    while $Q \neq \emptyset$ do
        $current \leftarrow Q.dequeue()$
        if $current = goal$ then
            return $current$
        foreach $n$ adjecent to current do
            if $n$ is not discovered then
                label $n$ as discovered
                $n.parent \leftarrow current$
                $Q.enqueue(n)$
```

2.1.3.2 DFS

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. One starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. A simple node expansion using the depth-first search strategy is shown in Figure 2.7 (taken from [26]).

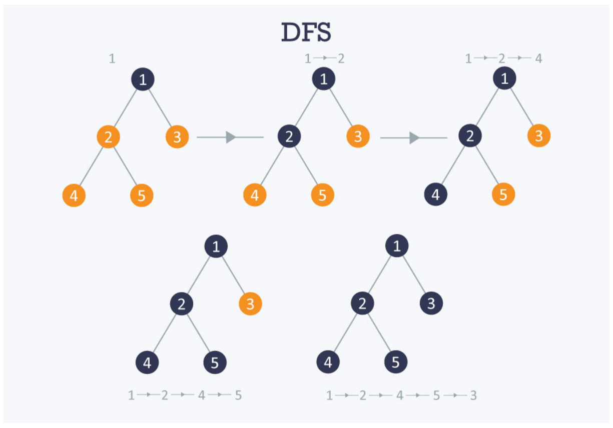


Figure 2.7: DFS example

DFS may suffer from non-termination meaning that it may go without finding a solution for an infinite time when there are loops in the graph.

Complexity of the DFS depends on the way the graph is implemented. It will be $O(|E| + |V|)$ if the graph is given in the form of adjacency list but if the graph is in the form of adjacency matrix then the complexity is $O(E^2)$, as we have to traverse through the whole row until we find an edge [25]. The pseudocode of DFS is given in Algorithm 2.2.

Algorithm 2.2: DFS pseudocode(recursive implementation)

```
Input: A graph $G$ and a starting vertex $root$ of $G$
Output: All vertices reachable from $root$ labeled as discovered
1 function DFS($G, root$)
2     label $root$ as discovered
3     foreach $root$ to $w$ in $G.adjacentEdges(w)$ do
4         if $w$ is not discovered then
5             $DFS(G, w)$
```

2.1.3.3 Dijkstra

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph [27]. For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. A simple node expansion using Dijkstra's algorithm is shown in Figure 2.8 (taken from [28]).

Dijkstra's original algorithm does not use a min-priority queue and runs in time $O(|V|^2)$. The implementation based on a min-priority queue implemented by a Fibonacci heap runs in $O(|E| + |V| \log |V|)$ [29]. Dijkstra's algorithm is complete and correct. The pseudocode of Dijkstra's algorithm is given in Algorithm 2.3. (If we are only interested in a shortest path between two nodes source and target, we can terminate the search after line 15, of Algorithm 2.3, if u = target.)

2.1.3.4 A*

A* is an informed search algorithm, or a best-first search, meaning that it solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted

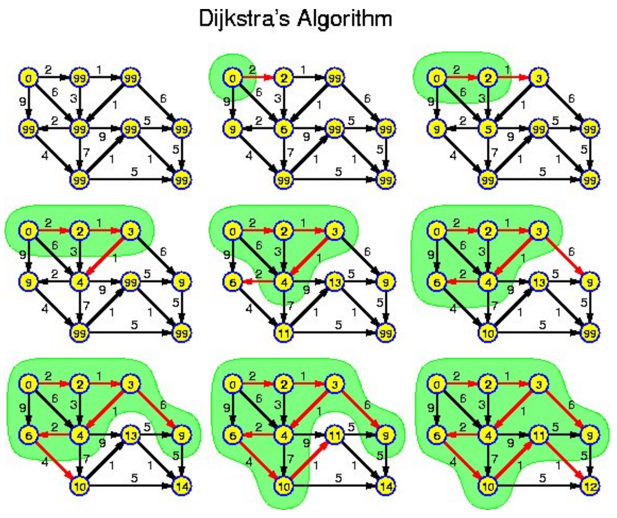


Figure 2.8: Dijkstra example.

Algorithm 2.3: Dijkstra pseudocode

Input: A graph G and a starting vertex *source*

Output: Shortest path from *source* to all the other vertex

```
function Dijkstra($G, source$)
    create vertex set $Q$
    foreach $v$ in $G$ do
        $dist[v] \leftarrow \infty$
        $prev[v] \leftarrow \emptyset$
        $Q.add(v)$
    $dist[s] = 0$
    while $Q \neq \emptyset$ do
        $u \leftarrow$vertex in $Q$ with min $dist[u]$
        $Q.remove(u)$
        foreach neighbor $n$ of $u$ do
            $alt \leftarrow dist[u] + lenght(u, n)$
            if $alt < dist[n]$ then
                $dist[n] \leftarrow alt$
                $prev[n] \leftarrow u$
    return $dist[], prev[]$
```

graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node. At each iteration of its main loop, A* needs to determine which of its partial paths to expand into one or more longer paths. It does so based on an estimate of the cost (total weight) still to go to the goal node. Specifically, A* selects the path that minimizes

$$f(n) = g(n) + h(n)$$

where n is the last node on the path, $g(n)$ is the cost of the path from the start node to n, and $h(n)$ is a heuristic that estimates the cost of the cheapest path from n to the goal. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic function must be admissible, meaning that it never overestimates the actual cost to get to the nearest goal node. A*, with an admissible heuristic, considers fewer nodes than any other search algorithm with the same heuristic.

A* is commonly used for the common pathfinding problem in applications such as games, but was originally designed as a general graph traversal algorithm. What sets A* apart from a greedy best-first search algorithm is that it takes the cost/distance already traveled, $g(n)$, into account. There exist many different variations of A* which try to speed up the search like weighted A* which expands states in the order of $f = g + \epsilon h$ values, (where $\epsilon > 1$ and is bias towards states that are closer to goal). Weighted A* trades off optimality for speed. There exist other algorithms that produce more realistic looking paths on a grid environment. One of them, Theta*, is introduced in the next section.

The time complexity of A* depends on the heuristic. In the worst case of an unbounded search space, the number of nodes expanded is exponential in the depth of the solution (the shortest path) d: $O(b^d)$, where b is the branching factor (the average number of successors per state) [30]. A* pseudocode is shown in Algorithm 2.4.

2.1.3.5 Theta*

Theta* [31] is a variation of A*, it propagates information along grid edges (to achieve a short runtime) without constraining the paths to grid edges (to find “any-angle” paths). The key difference between Theta* and A* is that Theta* allows the parent of a vertex to be any vertex, unlike A* where the parent must be a successor. Theta* is simple, fast and finds short and realistic looking paths but it can not be applied to grids whose cells have different sizes and traversal costs. On Figure 2.9 (taken from [31])

Algorithm 2.4: A* pseudocode(recursive implementation)

```
Input: A graph $G$ and a starting vertex $start$ and a goal vertex $goal$
Output: Shortest path from $source$ to $goal$
function A*($start, goal$)
    $closedSet \leftarrow \emptyset$
    $openSet \leftarrow start$
    foreach $v$ $in$ $G$ do
        $gScore[v] \leftarrow \infty$
        $fScore[v] \leftarrow \infty$
        $prev[v] \leftarrow \emptyset$
    $gScore[start] = 0$
    $fScore[start] = heuristic(start, goal)$
    while $openSet \neq \emptyset$ do
        $u \leftarrow v$ in $openSet$ with min $fScore[]$
        if $u = goal$ then
            return $reconstrunctPath(prev, u)$
        $openSet.remove(u)$
        $closedSet.add(u)$
        foreach $neighbor$ $n$ $of$ $u$ do
            if $n \in closedSet$ then
                $continue$
            if $n \notin openSet$ then
                $openSet.add(n)$
            $tentativeScore \leftarrow gScore[u] + dist[u, n]$
            if $tentativeScore \geq dist[n]$ then
                $continue$
            $prev[n] \leftarrow u$
            $gScore[n] \leftarrow tentativeScore$
            $fScore[n] \leftarrow gScore[n] + heuristic(n, goal)$
    return $failure$
```

a comparison between a path produced using A* and a path produced by Theta* is shown, it is clear that the one produced using Theta* is more realistic looking. Theta* finds a path from the start vertex to the goal vertex if such a path exists.

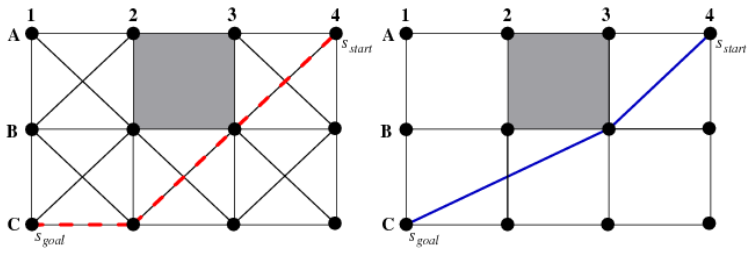


Figure 2.9: Theta vs A*: On the left, in red, the path produced by A*, on the right, in blue, the path produced by Theta*.*

Theta* exists in two versions, Basic Theta* and Angle-Propagation Theta* where the main difference is that Angle-Propagation, by calculating and maintaining angle ranges, is much faster. At algorithimic level Theta* is very similar to A* where the only difference are the lines from 21 to 26 (shown in Algorithm 2.5) from the Algorithm 2.4 of A* presented above.

2.1.4 Robot Motion Planning

Motion planning (also known as the navigation problem or the piano mover's problem) consists of the process of breaking down a desired movement task into discrete motions that satisfy movement constraints and possibly optimize some aspect of the movement. It is effectively an extension of localisation, in that it requires the determination of the robot's current position and a position of a goal location, both within the same frame of reference or coordinates. A motion planning algorithm takes a description of the environment and a goal state as input, and produces the speed and turning commands sent to the robot's wheels. Motion planning algorithms might address robots with a larger number of joints (e.g., industrial manipulators), more complex tasks (e.g., manipulation of objects), different constraints (e.g., a car that can only drive forward), and uncertainty (e.g., imperfect models of the environment or robot). The main algorithms for solving motion planning are DWA [32] and trajectory planner [33] which share a lot of similitaries.

Algorithm 2.5: Theta* pseudocode (only lines different from A*)

```
function PartTheta*($start, goal$)
    if $lineofsight(prev[u], n)$ then
        if $gScore[prev[u]] + dist(prev[u], n) < gScore[n]$ then
            $gScore[n] \leftarrow gScore[prev[u]] + dist(prev[u], n)$
            $prev[n] \leftarrow prev[u]$
        if $n \in openSet$ then
            $openSet.remove(n)$
        $openSet.add(u)$
        $fScore[n] \leftarrow gScore[n] + heuristig(n, goal)$
    else
        if $gScore[u] + dist(u, n) < gScore[n]$ then
            $gScore[n] \leftarrow gScore[u] + dist[n, u]$
            $prev[n] \leftarrow u$
        if $n \in openSet$ then
            $openSet.remove(n)$
        $openSet.add(u)$
        $fScore[n] \leftarrow gScore[n] + heuristig(n, goal)$
```

2.1.4.1 DWA and Trajectory Planner

The basic idea of both the Trajectory Rollout and Dynamic Window Approach (DWA) algorithms is as follows [34]:

1. Discretely sample in the robot's control space (dx,dy,dtheta).
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

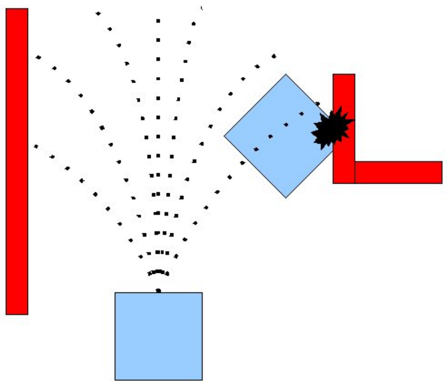


Figure 2.10: DWA and trajectory planner example.

DWA differs from Trajectory Rollout in how the robot's control space is sampled. Trajectory Rollout samples from the set of achievable velocities over the entire forward simulation period given the acceleration limits of the robot, while DWA samples from the set of achievable velocities for just one simulation step given the acceleration limits of the robot. This means that DWA is a more efficient algorithm because it samples a smaller space, but may be outperformed by Trajectory Rollout for robots with low acceleration limits because DWA does not forward simulate constant accelerations. In practice however, DWA and Trajectory Rollout perform comparably in all tests [34] and DWA is recommended for its efficiency gains.

2.2 Constraint Shortest Path Problem and its Variants

The constrained shortest path (CSP) is a well known NP-hard problem [35]. CSP is a generalization of the shortest path problem on graphs in which each edge is associated not only with a distance, but also with an additional weight, and the objective is to find the shortest path between given start and target locations, subject to a constraint on the total weight accumulated along the path. Solution strategies for the CSP can be classified into one of two main categories: Dynamic Programming (DP) and algorithms used to solve integer linear problems (like columns generation). Methods based on DP are also known as label-setting or label-correcting algorithms. In this section we describe some of the main algorithms to solve CSP and different works in different settings extending and improving the run time of the algorithms.

2.2.1 Integer Liner Programming

An integer linear programming (ILP) problem is a problem in which some or all of the variables are restricted to be integers and in which the objective function and the constraints (other than the integer constraints) are linear. Formally ILP is defined as

$$min \quad \underline{c}^T \underline{x}$$

$$A\underline{x} \geq \underline{b}$$

$$\underline{x} \geq \underline{0} \quad with \quad \underline{x} \in Z^n$$

where $\underline{x}$ represents the vector of variables (to be determined), $\underline{c}$ and $\underline{b}$ are vectors of (known) coefficients, A is a (known) matrix of coefficients, and Z^n represents the set of dimensional vectors n having integer components.

One of the algorithms used to solve integer linear problems is Column Generation [36]. The overarching idea is that many linear programs are too large to consider all the variables explicitly. Since most of the variables will be non-basic and assume a value of zero in the optimal solution, only a subset of variables need to be considered when solving the problem. Column generation leverages this idea to generate only the variables which have the potential to improve the objective function, that is to find variables with negative reduced cost (assuming without loss of generality that the problem is a minimization problem).

The problem being solved is split into two, different smaller problems: the master problem and the subproblem. The master problem is the original problem with only a subset of variables being considered. The subproblem is a new problem created to identify a new variable. The objective function of the subproblem is the reduced cost of the new variable with respect to the current dual variables, and the constraints require that the variable obey the naturally occurring constraints.

The process works as follows. The master problem is solved, from this solution we are able to obtain dual prices for each of the constraints in the master problem. This information is then utilized in the objective function of the subproblem. The subproblem is solved. If the objective value of the subproblem is negative, a variable with negative reduced cost has been identified. This variable is then added to the master problem, and the master problem is re-solved. Re-solving the master problem will generate a new set of dual values, and the process is repeated until no negative reduced cost variables are identified. If the subproblem returns a solution with non-negative reduced cost, we can conclude that the solution to the master problem is optimal.

2.2.2 Dynamic programming

Dynamic programming is a method for solving complex problems by breaking them down into collections of simpler subproblems and solving them just once. After each of the subproblems is solved the solution is stored so the next time the same subproblem occurs, instead of recomputing it, the solution is simply looked up. This way some computational time is saved in expense of a expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called “memoization”.

A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub-problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub-problems. Such optimal substructures are usually described by means of recursion.

Overlapping sub-problems means that the space of sub-problems must be small, that is, any recursive algorithm solving the problem should solve the same sub-problems over and over, rather than generating new sub-problems.

2.2.3 The Pulse Algorithm

The work in [37], with the aim to expand the body knowledge of the CSP, introduces the pulse algorithm, an exact solution method for CSP capable of handling large-scale networks in a reasonable amount of time. This algorithm consistently outperforms the methods presented in Sections **2.2.1** and **2.2.2**.

The idea behind the pulse algorithm is very simple, almost naive, yet very powerful. The algorithm is based on the idea of propagating pulses through a network from a start node $v_s \in N$ to an end node $v_e \in N$. As a pulse traverses the network from node to node, it builds a partial path P including the nodes already visited, the cumulative objective function $c(P)$ and the cumulative resource consumption $t(P)$. Each pulse that reaches the final node v_e contains all the information for a feasible path P from v_s to v_e. If nothing prevents the pulses from propagating, the algorithm completely enumerates all possible paths from v_s to v_e, ensuring that the optimal path P^* is always found. At the core of the algorithm lies the ability to (effectively and aggressively) prune pulses (i.e., prevent their propagation), without jeopardizing the optimal path. This idea is shared with other algorithms like branch and bound, where an implicit enumeration is performed with relative efficiency. Similarly, the strength of the pulse algorithm depends on the pruning strategies. The pseudocode for pulse algorithm is shown on Algorithm 2.6 with the respective pruning function shown on Algorithm 2.7.

Algorithm 2.6: Pulse Algorithm

```
Input: A graph $G$ starting vertex $v_s$ goal vertex $v_e$ maximum resourse
       consumtion $T$
Output: Optimal path $P$
1 function Pulse($G, v_s, v_e, T$)
2     $P \leftarrow \emptyset$
3     $c^0 \leftarrow 0$/* the initial objective function                    */
4     $t^0 \leftarrow 0$/* the initial resource consumption                  */
5     $initialization(G,T)$/* $initialization(G,T)$:one-to-all shortest
          path algorithm to find the minimum resource consumption
          for every node                                                   */
6     $pulse(v_s, c^0, t^0, P)$
7     return $P$
```

Algorithm 2.7: Pulse function

```
1 function pulse($v_k, c, t, P$)
2     if $!checkDominance(v_k, c, t)$ then
3         if $checkFeasibility(v_k, t)$ then
4             if $!checkBounds(v_k, c)$ then
5                 foreach $v_i \in N(v_k)$ do
6                     $c' \leftarrow c + c_{ki}$
7                     $t' \leftarrow t + t_{ki}$
8                     $pulse(v_i, c', t', P')$
```

There are three core pruning strategies presented in [37] *infeasibility pruning*, *dominance pruning*, *bounds pruning.* The *infeasibility pruning* strategy discards a partial path P_{si} (path from a starting node v_s to another node v_i) when it is not possible to reach the end node without exceeding the resource constraint. The *dominance pruning* P_1 dominates P_2 if $c(P_1) \leq c(P_2)$ and $w(P_1) < w(P_2)$ or $c(P_1) < c(P_2)$ and $w(P_1) \leq w(P_2)$ where P_1 and P_2 are two partial paths at a given node $v_i \in N$. The *bounds pruning* uses a primal bound $\bar{c}$ that is updated with the value of the best solution found so far so if $c(P_{si} + c(P^c_{ie}) \geq \bar{c})$ then path P can be safely pruned.

2.2.4 Acceleration strategies for the weight constrained shortest path problem with replenishment

The weight constrained shortest path problem with replenishment (WCSPP-R) is a generalization of CSP. The paper [8] investigates a generalization of the CSP where edges may be associated to binary indicators able to reset the accumulated weight when the corresponding edge is traversed.

The WCSPP-R consists of finding the minimum cost path between a start node $v_s \in N$ and an end node $v_e \in N$ without exceeding a resource constraint W. The WCSPP-R considers replenishment arcs that reset the value of the consumed resource to zero at the tail node, that is, just before traversing the replenishment arc. A feasible path P in the WCSPP-R is an ordered sequence of nodes that satisfies W everywhere.

They extend the pulse algorithm presented in Section **2.2.3** for the CSP adding three new acceleration strategies, namely, *path completion*, *pulse queueing*, and *best-promise exploration* order. A major modification to the original pulse algorithm is the inclusion of a pulse queue denoted by Q. When the depth of a partial path P reaches a maximum allowed value δ, its exploration pauses and the corresponding pulse is stored in Q, saving the partial path P, the node where the pulse was paused $n(P)$, the cumulative cost $c(P)$, and the resource consumption $w(P)$. The algorithm stops when the queue Q is empty.

Path completion adds a minimun cost path P^c_{ie} from v_i to v_e to a given partial path P_{si} and checks is the completed path P_{se} is feasible and if $c(P_{se})$ is less than the primal bound $\bar{c}$. If one of the conditions is not true the path can be pruned. *Pulse queueing* means performing a DFS from the start node to a maximum depth δ. When the pulse reaches that depth the partial path is stored in the pulse queue Q following a queue discipline. Once there are no more active pulses the queued pulses are processed until Q is empty. $Best - promise\ exploration$ order is defined by the queue discipline for Q.

This model is not applicable in our case, since we have to deal with the possibility of transmitting only a portion of the accumulated data.

2.2.5 Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles

Authors of [9], propose a model that partially generalizes ours in the context of planning shortest paths with charging stops for electric vehicles (EV) (the filling of our buffer can be naively thought as dual to the draining of the battery).

They extend CSP problems for EVs with realistic models of charging stops, including varying charging power and battery swapping stations so their model is able to cope with continuous, increasing, and concave functions describing how batteries recharge, even when going downhill. In particular, charging times are not independent of the state of battery when arriving at a charging station. Additionally, the charging process can be interrupted as soon as further charging would increase the arrival time at the target. While the resulting problem is NP-hard, they propose a combination of algorithmic techniques to achieve good performance in practice.

Their basic algorithm, charging function propagation (CFP), generalizes the bicriteria variant of Dijkstra's algorithm [38] extending labels to represent (infinite, continuous) sets of nondominated solutions. Label $l = (\tau_t, \beta_u, u, f_{[u,\ldots,v]})$ at a vertex v consists of the trip time τ_t of the path from s to v (including charging time on every previous charging station except u), the SoC (state of charge of the battery) β_u when reaching u, the last charging station u, and the consumption profile $f_{[u,\ldots,v]}$ of the subpath from u to v. The consumtion profile($f_{[u,\ldots,v]}$) is defined as $f_P : [0, M] \to [-M, M] \cup \infty$ representing the change of the SoC of the battery when going throught path P where ∞ value means an infeasible path.

They also present techniques based on A* Search, CH [39], and a combination of them, to reduce the running times of the basic approach, CFP.

In this problem variant, the possibility of having cycles with negative cumulative weight, namely the possibility of traveling along cycles while indefinitely recharging the battery, is clearly ruled out by the laws of physics. In our case, we must also contemplate such a possibility (think of a robot which takes a detour towards a transmission area with an increasingly higher transmission rate, and then goes back to its original path).

Chapter 3

Problem Setting

We model the environment as a connected, simple, weighted graph $G = (V, E)$, where V represents physical locations the robot can occupy and E represents the connections between those locations. Edges are associated with a weight function $t : E \to \mathbb{N}^+$, called *time function* and representing their traveling time. We assume that time evolves in discrete steps $\mathbb{N}^+$, as well. Between two subsequent time steps, the robot can either *stay still* at its current vertex or *move along a graph edge*. In the latter case, the robot is not allowed to interrupt an edge traversal once started, but it must reach the destination vertex before making another decision. In any case, at each time step, the robot stacks 1 unit of information (or, equivalently, any constant amount) into a buffer of size $B \in \mathbb{N}^+$.

The environment has some communication zones modeled as a set of *transmission vertices* $V_T \subseteq V$. To formalize the transmission of data, we define a *transmission-rate function* $r : V^2 \to \mathbb{Q}$ that describes the amount of information a robot can send to the BS *between two consecutive time steps* when moving from a vertex to another one or staying at a vertex. To consistently model transmissions according to reasonable assumptions about communication and legal moves in G, we impose some constraints on $r()$:

1. (**vertex transmission capability**): $r(v, v) > 0 \iff v \in V_T$;
2. (**legal moves**): $r(u, v) = 0$ for each $u \neq v$ s.t. $(u, v) \notin E$;
3. (**edge conservativeness**): for each $(u, v) \in E$, $r(u, v) \leq \max\{r(u, u), r(v, v)\}$.

To compact the notation, we also define the net amount of information unstacked from the buffer in a time step:

$$\bar{r}(u, v) = r(u, v) - 1.$$

If $\bar{r}(u, v) > 0$ the amount of information transmitted exceeds the amount of information stacked in a time step and, thus, the robot is able to unfill information from the buffer while moving from u to v. An example of a simple environment is given of Figure 3.1.

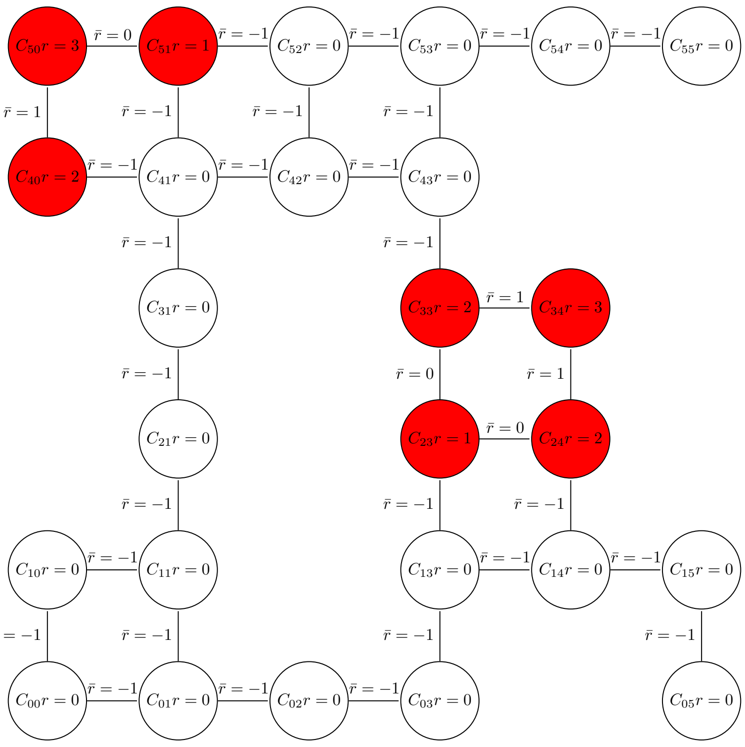


Figure 3.1: Example of an environment. Red vertices represent the transmission vertices, r represents the transmission-rate function of each node, and $\bar{r}$ represents the information unstacked from the buffer if the robots moves on the edge

The robot must move from a start vertex $s \in V$ to a goal vertex $g \in V$, without any constraints on the initial and final amount of data contained into the buffer. A *solution* S of our problem consists of a sequence of k pairs $p_i \in V \times \mathbb{N}$, representing the number of steps in which the robot remains still at a vertex. A solution starts from s and ends in g:

$$S = [p_1 = (s, t_s), p_2, \dots, p_k = (g, 0)],$$

where two subsequent pairs $p_i = (v_i, t_i)$ and $p_{i+1} = (v_{i+1}, t_{i+1})$ implicitly define the traversal of the graph edge $(v_i, v_{i+1}) \in E$ after having remained still for t_i steps at v_i. This solution encoding defines a sequence of pairs of values $[(b_1^I, b_1^O), (b_2^I, b_2^O), \dots, (b_k^I, b_k^O)]$ representing the amount of data present into the buffer when arriving (I) and leaving (O) from each of the k vertices composing a solution.

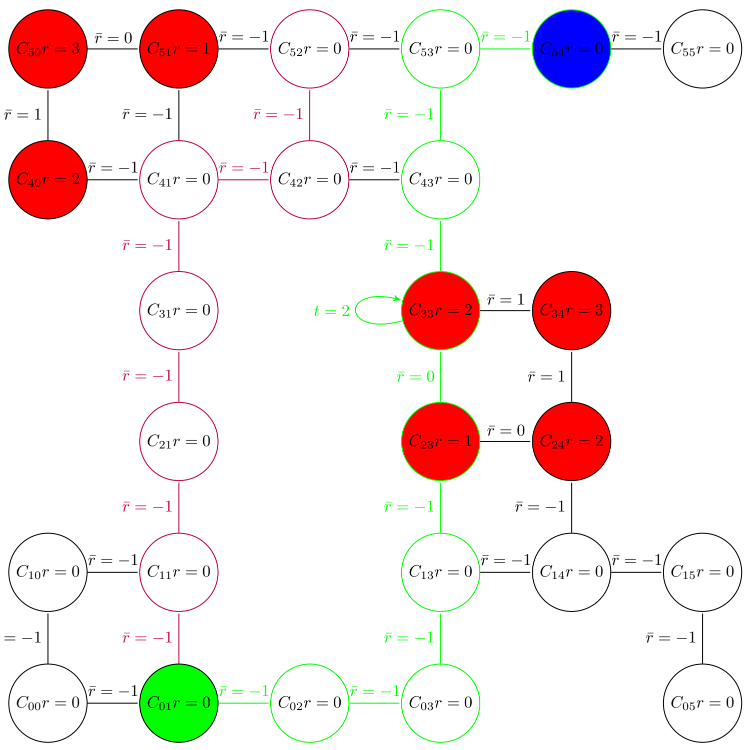


Figure 3.2: Example of a feasible solution and infeasible solution. Starting from the green vertex with an empty buffer and going to g the blue vertex. Buffer size of the robot is B=5. Following the green edges and stopping for two time steps at node C_{33} to transmit the robot ends at the blue node with a full buffer. Following the purple edges we have a total accumulation of information of 6 which makes the path infeasible.

We say that a solution S is *feasible* iff $b_i^I, b_i^O \leq B$ for each $i = 1, 2, \ldots, k$. Figure 3.2 showns an example of a feasible solution and an example of an infeasible solution. The objective is to reach the goal in the least possible time, i.e., to minimize:

$$T = \sum_{i=1}^{k-1} \left[t_i + t(v_i, v_{i+1}) \right]. \tag{3.1}$$

We call this problem *Limited-Buffer Shortest Path problem* (*LBSP*).

3.1 NP-hardness

We now give strong evidence to the fact that LBSP is a hard problem, by proving that the corresponding decision version, called LBSP-D, is NP-hard. In LBSP-D, the aim is to decide whether a given instance of LBSP admits a feasible solution with total time less than a given $\overline{T}$, with $\overline{T} \in \mathbb{N}^+$. To this aim, we construct a reduction from the decision version of CSP, which is NP-complete [35]:

CSP-D

INSTANCE: a graph $\hat{G} = (\hat{V}, \hat{E})$, an edge length function $l : \hat{E} \to \mathbb{N}^+$, an edge weight function $w : \hat{E} \to \mathbb{N}^+$, start and goal vertices $\hat{s}, \hat{g} \in V$, positive integers $L, W \in \mathbb{N}^+$.

QUESTION: is there a path from $\hat{s}$ to $\hat{g}$ in $\hat{G}$ that has total length at most L and total weight at most W?

Without loss of generality, we consider only CSP-D instances in which $l(u, v) \geq w(u, v)$, $\forall (u, v) \in \hat{E}$. Indeed, any problem instance can be turned into an instance satisfying such a constraint by simply multiplying all the lengths by a proper constant. For the reduction, we set $G = \hat{G}$ (implying $V = \hat{V}$ and $E = \hat{E}$), $V_T = V$, $B = W$, and $\overline{T} = L$. The edge time function t of LBSP-D is set equal to the edge length function l of CSP-D. The rate function, for each $u \neq v \in V$, is defined as:

$$r(u, v) = \begin{cases} 1 - \dfrac{w(u, v)}{l(u, v)} & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

Also, for each $v \in V_T$, the rate function is defined as:

$$r(v, v) = \max \{ r(u, v) \mid (u, v) \in E \} .$$

Notice that all the transmission rates are lower than 1. This means that there is no advantage in staying still at any vertex, since the buffer value would not decrease. More formally, if there exists a solution of LBSP-D whose stop time t_i on a vertex is not 0, there also exists a not worse solution whose stop time t_i is 0. Also, because $l(u, v) \geq w(u, v)$, $\forall (u, v) \in E$ there are no "negative cycles", i.e., cyclic paths allowing to decrease the buffer value when traveled. More generally, if there exists a solution where the robot travels along a cycle, then there exists a not worse solution without cycles.

Given what said above, it is straightforward to check that the constructed LBSP-D instance admits a *yes* answer iff the original CSP-D instance admits a *yes* answer.

Chapter 4

Algorithms

4.1 Optimal Algorithm

We now present an exponential algorithm for solving to optimality the LBSP defined in the previous section.

Let us notice that, despite transmission rates are rational numbers $\mathbb{Q}$, any problem instance can be turned into an equivalent instance where all the possible *buffer states* are – arbitrarily large – integer numbers. In particular, let M be the least common multiple of all the transmission rate's denominators. From now on, we assume that all the values of the time function t and the buffer size B are defined as multiple of M and thus only integer buffer states are allowed (the original time values can be obtained, once a solution is found, dividing by M).

The algorithm leverages a transformation of the input graph G, whose pseudo-code is reported in Algorithm 4.1. The *directed* graph obtained, G_B, is an expanded version of G in which the state of the buffer is explicitly represented for each vertex through a set of "buffer-expanded" vertices. An optimal solution to the LBSP on G is then obtained by simply finding a shortest path on the new graph G_B.

In order to simplify the pseudo-code, it is assumed that, for each $u, v \in V$, $t(u, v) = \infty$ if $(u, v) \notin E$, and $t(v, v) = M$. The algorithm starts by creating $B+1$ vertices $\{v^0, v^1, \ldots, v^B\}$ for each vertex in V (lines 4-5): these vertices univocally identify the state of the robot, i.e., v^i means that the robot is at v with buffer state i.

The algorithm then builds the set of arcs, which are of two types: those connecting vertices of G_B corresponding to the same vertex of G, but with different buffer states (i.e., connecting v^i and v^j), and those connecting vertices of G_B corresponding to different vertices in G (and possibly different

Algorithm 4.1: Graph Transformation

```
Input: A simple undirected graph $G = (V, E)$, a buffer size $B$, the rate
       function $r()$, the time function $t()$
Output: A weighted directed graph $G_B = (V_B, A_B, w)$
function transformGraph($G, B, r, t$)
    $V_B \leftarrow \{\}$
    $A_B \leftarrow \{\}$
    foreach $v \in V$ do
        $V_B \leftarrow V_B \cup \{v^0, v^1, \ldots, v^B\}$
    foreach $u, v \in V$ do
        for $b = 0$ to $B$ do
            $x \leftarrow \max\{0, b - t(u,v)\bar{r}(u,v)\}$
            if $x \leq B$ then
                $A_B \leftarrow A_B \cup \{(u^b, v^x)\}$
                $w(u^b, v^x) \leftarrow t(u,v)$
    return $G_B = (V_B, A_B, w)$
```

buffer states). In the pseudo-code, when $u = v$, the first case is handled, otherwise, the second case is covered (lines 6-11). These arcs correspond to temporal transitions in the system state. Following an arc, the robot can either stay still on a vertex of G and change its current buffer state, or move to another vertex of G (possibly changing his buffer state too). Notice that, for each vertex in V_B, there is exactly one arc of the first type and at most $|V| - 1$ arcs of the second type. This means that $|A_B|$ is lower than or equal to $MB|V|^2$ (where B is the buffer size).

Once the graph G_B is returned, a shortest path algorithm is applied, e.g., Dijkstra's algorithm, to find a shortest path from s^0 to any g^x, where s and g are the start and the goal vertices on G, respectively. (x could be restricted to a set of values, if additional constraints to the final state of the buffer are imposed.) Figure 4.1 shows part of the transfomed graph if we would apply Algorithm 4.1 to the graph shown on Figure 3.1.

The correctness of the algorithm follows from the fact that we are explicitly representing all the possible buffer states. The whole computing time of the transformation and the shortest path seeking is clearly exponential, and both upper-bounded by $O(MB|V|^2)$ w.r.t. the parameters B and M, where B denotes the original size of the buffer.

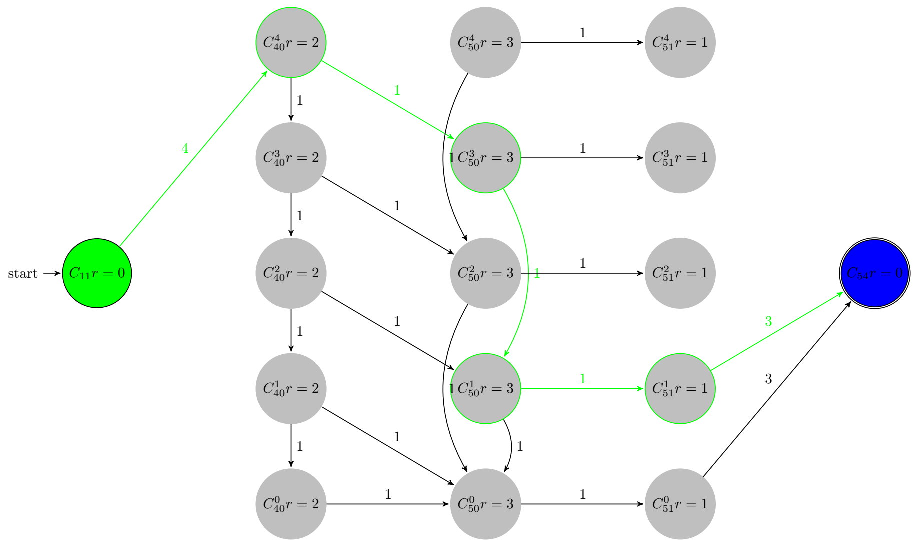


Figure 4.1: Example of a Graph transfomation with buffer B=4 considering the graph presented in Figure 3.1. (To keep the figure simple only a few nodes are considered and not all the edges are shown). After building the transformed graph applying Dijkstra's algorithm the path in green would be the optimal solution.

4.2 Feasibility Test

To decide the feasibility of a given problem instance, one could apply Algorithm 4.1 and check whether the obtained graph G_B contains at least an (s, g)-path. If not, the instance does not admit any feasible solution. However, since the computing time of such a procedure could be large (recall the above complexity bound), it could be useful to have at hand a faster method to decide feasibility.

We now present a simple method that leverages two additional graphs $G_1 = (V_1, E_1, w_1)$ (weighted) and $G_2 = (V_2, E_2)$ (unweighted). To obtain G_1, we set $V_1 = V$ and $E_1 = E$. Then, for each $(u, v) \in E_1$, we set $w_1(u, v) = \max\{0, -t(u, v)\bar{r}(u, v)\}$. The weights of G_1 represent the amount of stacked data (i.e., the amount of increase of the buffer value, if any) the robot attains when traversing (u, v). For what concerns G_2, the set V_2 is the set of vertices of G whose transmission rate is strictly greater than 1, plus s and g. We add an edge (u, v) to E_2, with $u, v \in V_2$, iff the length of the shortest path from u to v in G_1 (with weights w_1) is less than or equal to B. This is equivalent to say that there exists a u-v path in G such that it is always possible to travel from u to v without overfilling the buffer (for a sufficiently low buffer state in u).

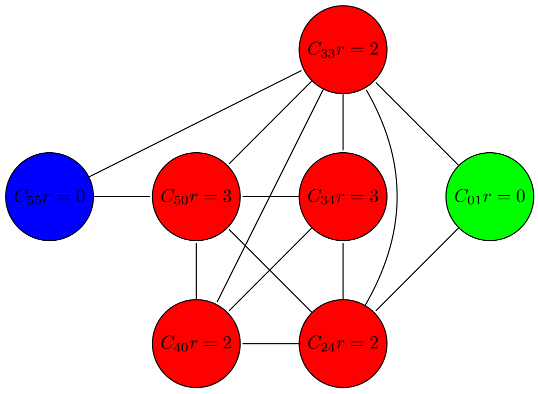


Figure 4.2: Example G_2 based on the graph presented in Figure 3.1 with a buffer size 4. In this case there exist a solution because there exists a path connecting start and goal nodes.

Given the graph G_2 constructed as above, it can be easily shown that an (s, g)-path in G_2 exists if and only if the problem instance admits at least a feasible solution. Figure 4.2 shows an example of G_2 built from the graph shown in Figure 3.1. Figure 4.3 shows an example of a path found using the method described above and the graph G_2 shown in Figure 4.2. The graph shown in Figure 4.3 is G_1 built based on the graph of Figure 3.1. The path found using the feasibility method is not optimal as shown in Figure 4.3 because it does not consider the transmission rates of the nodes, if they are strictly greater than 1. The computing time of this procedure is bounded by the complexity of finding a shortest path between each pair of vertices in G_1 (to compute E_2), that is, $O(|V|^3)$.

4.3 A Heuristic Algorithm

We now present a heuristic algorithm based on the refinement of an initial solution, which can be obtained from a weighted variant of the graph G_2 used above for the feasibility test. In particular, we start by constructing the "skeleton" of the heuristic solution as the walk (a sequence of possibly-repeated vertices) $\mathbf{w} = [s = v_1, v_2, \ldots, v_k = g]$ associated to the shortest (s, g)-path on G_2 according to a set of weights w_2. Specifically, the weight $w_2(u, v)$ of an edge $(u, v) \in E_2$ is defined as the traveling time of the path associated to the satisfaction of the buffer constraint (i.e., the traveling time of the path obtained by minimizing the weights w_1 defined previously). We now present a method to compute the *sequence of stopping times* for $\mathbf{w}$. (In fact, this method allows to compute the stopping times for any given

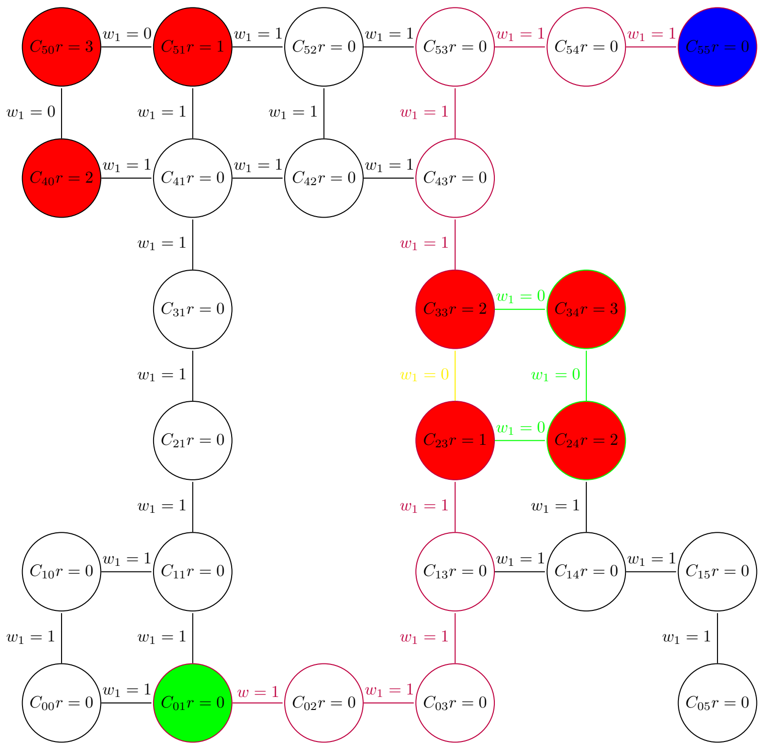


Figure 4.3: Example of feasible but not optimal path, found using the feasibility approach, shown on G_1 (that is build based on the graph presented in Figure 3.1). The feasible path starts from the cell in green, ends at the cell in blue and is composed by the edges in purple and yellow. The optimal path would be the path composed by edges in purple and in green.

sequence of vertices underlying a feasible solution.)

Formally, in order to obtain a solution $S = [p_1 = (s, t_s), p_2, \dots, p_k = (g, 0)]$, we have to find a sequence of times $T = [t_1 = t_s, t_2, \dots, t_k = 0]$ such that the buffer constraint is satisfied and the objective function (3.1) is minimized. To this aim, we use the procedure of Algorithm 4.2.

The algorithm iterates through the vertices of the walk, iteratively updating the buffer state b^I, b^O at each vertex. The idea is that, at a given vertex v_i, if the robot cannot reach the next vertex v_{i+1} in the walk, it has to necessarily transmit a certain amount of information to proceed further. Such an amount q is transmitted by the updateTimes() function, which iteratively tries to exploit the vertices with the higher transmission rates. Each transmission is subject to two constraints: (1) the amount of information transmitted cannot exceed that stacked in the buffer, (2) the transmission cannot take place farther than B time steps from the vertex where the robot

Algorithm 4.2: Find Times

```
Input: A feasible walk $\mathbf{w} = [v_1, \dots, v_k]$ on $G$, a buffer size $B$, the rate
       function $r()$, the time function $t()$
Output: A sequence of times $T$
1 function findTimes($\mathbf{w}, B, r, t$)
2     $b^I, b^O, T \leftarrow k$-length array initialized to 0
3     for $i = 0$ to $k - 1$ do
4         $b_i^O \leftarrow \max\{0, b_i^I - T_i \bar{r}(v_i, v_i)\}$
5         $b_{i+1}^I \leftarrow \max\{0, b_i^O - t(v_i, v_{i+1}) \bar{r}(v_i, v_{i+1})\}$
6         if $b_{i+1}^I > B$ then
7             updateTimes($T, b^I, b^O, i$)
8             updateBuffers($T, b^I, b^O, i$)
9     return $T$
```

got stuck. More precisely, the algorithm makes use of two functions, which have access to the input data B, r, t.

- updateTimes(T, b^I, b^O, i): it starts computing the amount to transmit $q = b_{i+1}^I - B$, in order to proceed further through the walk. This function tries to transmit q units of information at each already traversed vertex (index less than $i + 1$), starting from the one with the highest transmission rate and considering only those vertices whose rate is greater than 1. At each transmission attempt, for each $j < i$, we check two constraints: (1) at vertex v_j, the robot cannot transmit more than the current value of b_j^O, (2) at vertex v_j, the robot cannot be farther than B time steps from v_{i+1}. At this point, if q units of information are completely transmitted, the sequence of times T is updated.

- updateBuffers(T, b^I, b^O, i): given an updated sequence of times T, it straightforwardly updates the buffer states b^I and b^O, up to the vertex v_{i+1}.

The algorithm has a complexity of $O(|\mathbf{w}|^2)$, since updateTimes() can be run in $O(|\mathbf{w}|)$ by pre-sorting the vertices in the walk by transmission rates and pre-computing the distances between each pair of vertices.

Chapter 5

Implementation

5.1 Algorithms Implementation

We implemented the algorithms described in the previous chapter in C++. To build the graphs and to find the shortest paths we used the LEMON graph library [40] which is an open source C++ template library for optimization tasks related to graphs and networks. It provides highly efficient implementations of common data structures and algorithms. We used the implementation of Dijkstra's algorithm and the implementation of graphs as list_DiGraphs. For the complete class diagram of the program created see **Appendix A**. The complete code can be downloaded from https://github.com/Arlind1992/TesiPlanner.git.

To keep the number of nodes and edges to a minimum, in the implementation, a pre-transformation of the initial graph G is done. In particular, a new graph $G_{pre} = (V_{pre}, E_{pre})$ is build from the original one containing only the communication nodes of the original graph. So we have $V_{pre} = V_T \subseteq V$. The edges are created applying Dijkstra's algorithm on the original graph G from each node $v \in V_T$ to every other node. We connect two nodes $\in G_{pre}$ only if the distance returned by Dijkstra's algorithm is smaller than the buffer size B. Algorithm 4.1 is then applied.

If we want to find the shortest path between two nodes that are not in the communication set, $(v_1, v_2) \notin V_T$, we need to add them to G_{pre} and then to G_B. To connect them to G_{pre} we use the same logic as above when connecting two communication nodes. To connect them with G_B we use the cost of edges created in the previous step. So we connect the new nodes with the corresponding state calculated according to the rule, same as the one used in Algorithm 4.1, $max(0, b - t(u, v_1)\bar{r}(u, v_1))$, where u is a communication node. After adding them to the graphs now we can apply Dijkstra's algorithm to find the shortest path. Figure 5.1 shows the resulting graph after the pre-transformation of the graph in Figure 3.1 (only the nodes in red are left after transformation). The start and goal nodes are added like in Figure 5.1.

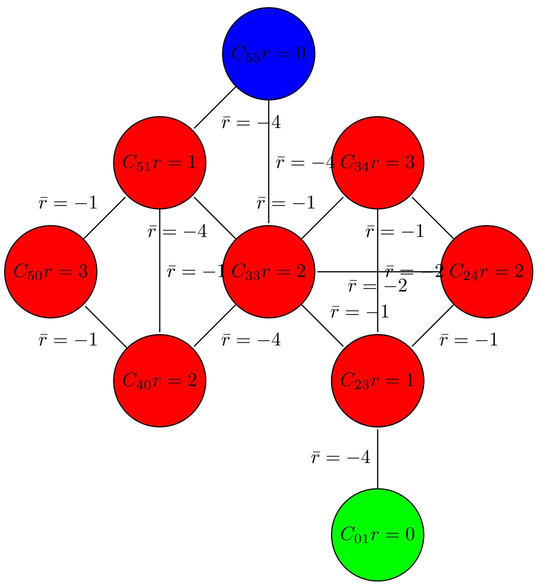


Figure 5.1: Result of the pre-transformation of the graph in Figure 3.1.

By implementing the algorithm in such a way we can save the static parts of the transformed graph in a file for faster creating times. Moreover, if we want to execute the algorithm multiple times on the same map with different start and goal nodes we apply the Algorithm 4.1 only once and then just add the different nodes to the graph and apply a shortest path algorithm (we implemented it with Dijkstra's).

5.2 Implementation as a ROS Planner

In order to further validate the feasibility of our approach we implemented the proposed optimal algorithm using ROS (an operating system used for real robots). Therefore, in this section we describe ROS. Then, we describe the implementation of the optimal algorithm using ROS's architecture, the main problems to be solved within this architecture and the proposed solutions.

5.2.1 What is ROS?

The Robot Operating System (ROS) is a flexible framework for writing robot software [10]. It is a collection of tools, libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

ROS has many packages ready to use for general purposes robots. that require little effort to be done by the user due only to a correct parameterization. The open source nature of the project allows users to customize the packages code to develop new robot behaviors while reusing other community members work.

From a technical perspective ROS is a middleware that offers an interprocess communication layer. The communication happens with a publish/subsribe mechanism in which each package works as a node that publishes messages on a Topic (i.e., an argument) and receives messages from subscribed topics; a simple example of nodes communicating through a publisher-subscribe architecture can be seen in Figure 5.2 (taken from [41]). Every node can listen for a topic and receive every message of that kind that is sent in the communication infrastructure. Coordination is regulated by a special node called *roscore* that informs other nodes of available topics and notifies the publishers when a subscription to a topic happens. It is also possible that a node needs to perform remote procedure calls, in this case services should be used [42]. Other details on how communication by topics happens in ROS are reported in [43].

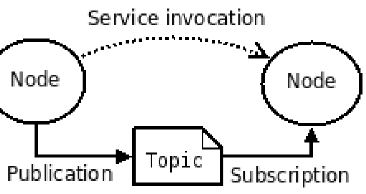


Figure 5.2: Simple example of a publish-subscribe and service invocation architecture.

5.2.1.1 Navigation Stack

One of the main tasks for robots is autonomous navigation. ROS provides a package (i.e., the navigation stack) that enables the programmer to setup a robot to navigate the environment avoiding the developing of entirely new solutions, but allowing him to customize the solution to the specific robot. The navigation stack faces some common problems in robot navigation:

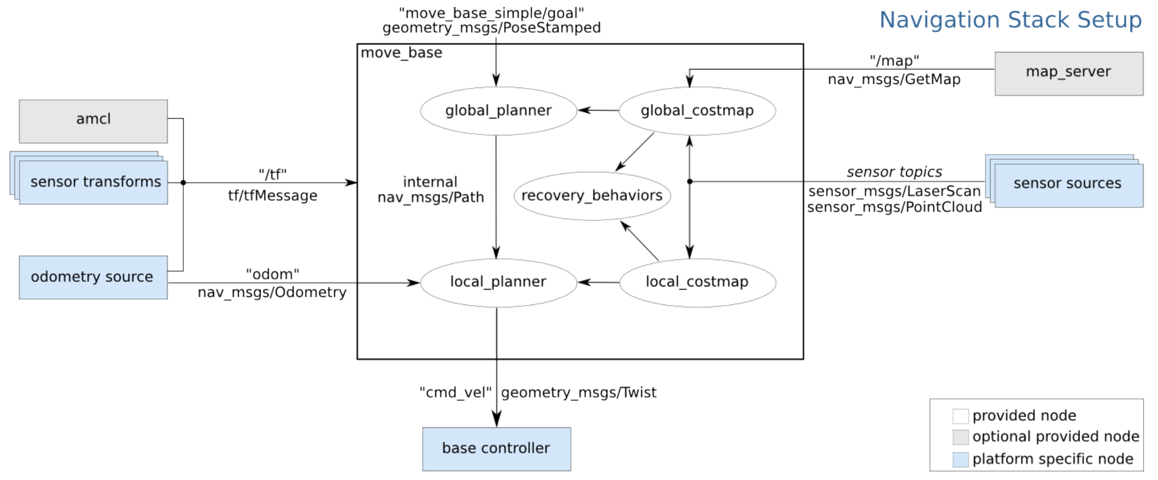


Figure 5.3: A common navigation stack setup.

Mapping The map building process. This problem is addressed mainly by the map server package [44].

Localization The capability of the robot to understand where it is in a map. The amcl package [45] taking in input a map, laser scans, and transform messages (trasformation between coordinates systems) gives in output a pose estimation for the robot in the map.

Planning The capability to plan to move in an environment. The move base package [46] handles the planning phase providing global and local planning.

A graphical representation of a common navigation stack setup is shown in Figure 5.3 (taken from [47]). Other details and tutorials on the setup of the navigation stack can be found in [48].

The move_base_package In the navigation stack, the planning part is delegated to the move base package. This package takes as input a map, a pose estimate for the robot in the map, and a goal to perform the planning phase that results in velocities commands (i.e., controls in terms of linear and angular velocities) to be sent to the mobile base. As shown in Figure 5.3 the move_base_package is composed of other packages that handle various aspects of the planning phase:

global_planner it defines the entire path from the start position to the goal position for the robot. It is possible to develop various kinds of logics for the planner and to include them in move base as plugins.

local_planner given a path to be followed and a local costmap, (where a costmap is a data structure that stores information about the environment, for example it contains information if a cell, in a grid of cells, is free or not), produces the velocity commands to be sent to the mobile base. It is possible to develop various kinds of logics for the planner including them in move base as plugins.

global_costmap builds a costmap for the entire map.

local_costmap builds a costmap for the part of the map around the robot (updated with sensors feedbacks).

recovery_behaviors is a node responsible for recovery behaviors generation that tries to clear the costmap (i.e., sensing again the surrounding area to avoid errors) when the robot is stuck.

Actually there are few logics already implemented for the global planner and for the local planner that can be chosen. The global planner has three alternatives:

carrot_planner takes a goal point, checks if the goal is inside an obstacle, and if so it walks back along the vector between the objective and the robot until an intermediate point that is not an obstacle is found. It then passes this new goal point on as a plan to the local planner. It is not a planner that ensures completeness, but could be good for few specific applications.

navfn provides an implementation of a grid-based navigation function that uses Dijkstra's algorithm in a 2-variables environment (i.e., only x and y coordinates are considered).

global_planner is the upgraded replacement of navfn with optimizations like the use of A* instead of Dijkstra's.

A more detailed description of the three packages can be found respectively in [49], [50] and [51].

The local planner has two alternatives based on the same procedure. They both apply sampled possible velocities for the robot in a few steps of forward simulations, a score is given to the resulting trajectories based on the velocity and on the proximity to obstacles, goal and global path. The highest score trajectory is then chosen for the movement sending the respective velocities to the mobile base and the process starts again. The difference between the two possible logics of the local planner are:

base_local_planner is based on the Trajectory Rollout approach [33], described in Section **2.1.4.1**. It samples from the set of possible velocities over the entire forward simulation time given the acceleration bounds of the vehicle.

dwa_local_planner is based on the Dynamic Window approach [32], described in Section **2.1.4.1**. It samples from the set of possible velocities only for one simulation step given the acceleration bounds of the vehiche.

The best choice, among the available ones and only for shortest path problems without constraints, is the one implemented in the global planner package or the navfn, while for the local planner some considerations based on the specific application should be done [52]. The above planners do not consider any constraints so they can not be directly used for our problem. Other details on the local planner possible implementations in ROS can be found in [52] and [34].

5.2.2 Algorithms implementation in ROS

There are two main problems when implementing our algorithms using ROS regarding navigation.

1. Integreating the planner of the C++ implementation with the ROS architecture.

2. Stopping the robot at communication nodes if the robot has to transmit any data.

To solve these problems we needed to create two plugins[1], one to modify the global_planner and one to modify the local_planner used by the navigation stack.

global_planner The new global planner uses the planner described in **5.1**, transforms the path generated by the planner to ROS coordinates and publishes in two topics. One the default topic (nav_msgs/Path) [51]. The second topic is a new topic created for the purpose of containing the information about the communication nodes in which the robot has to stop and the amount of time the robot needs to stop in each one of these nodes.

local_planner As a local planner one of the already implemented local planners can be used (we use base_local_planner) with some small modifications. The modifications, of the new local planner, consist in adding a subscriber to the new topic, described above. When the information, needed for the robot to move, is generated, the planner checks if the robot is in a configuration corresponding to a communication node where it needs to stop. If it is, the robot is stopped for the amount of time specified by the global planner.

The message passed by the global planner to the local planner contains the coordinates of the position, the robot needs to stop in, in terms of grid coordinates, coordinates calculated after the map goes throught discretization and the grid that is used by the planner is formed. It is done in this way because the local planner uses the Trajectory Rollout approach [33] and it is

[1]For information on planners as plugins in ROS visit http://wiki.ros.org/navigation/Tutorials/Writing A Global Path Planner As Plugin in ROS.

not guaranteed (in a reasonible amount of motion computational time) that the robot will pass through a certain exact point, but it can be configured that the robot will have to pass through every node specified by the global planner.

The topic created can be used also by another node, which can transmit the data to a base station. Moreover, constant connection times can be easily added to the robot wait time for each node.

Chapter 6

Experiments

We designed some experiments to valuate the running time and the solutions found by our algorithms. In this chapter we first descibe the settings in which we decided to test our algorithms. Then we proceed with describing and analyzing the results of three different sets of experiments. All the experiments are run on a laptop equipped with an Intel Core i7@2.70 GHz CPU and 16 GB RAM.

6.1 Experiments Settings

Environment. To test our agorithms we choose a grid representation of the environment. We start from a given map, set a discretization parameter, and build the grid. For our tests we chose the map in Figure 6.1 with a size of 400 meters long and 300 meters wide. We consider cells of 4×4 meters meters. The robot moves at a speed of 4 m/s and we consider steps of 1 second, so that, in our discretization, this will correspond to 1 cell/step.

Communication. To simulate communication three RF transceivers are represented in the environment (WiFi) with a maximum range of 80 meters (the red circles on Figure 6.1) which correspond to 20 cells. The bitrate, of each of the RF transceivers, changes depending on the distance and the discretization of the buffer (the minimum amount of information the robot can transmit). The robot collects data with a speed of 8 Mbit/s (for instance, a video feed at low resolution), which will represent our basic buffer unit (i.e., the “+1” unit of data stacked into the buffer at each time step).

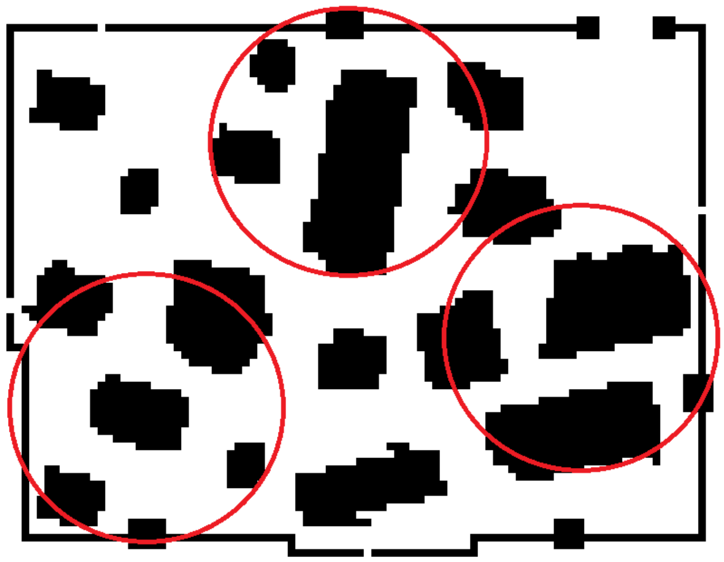


Figure 6.1: Experimental environment (size 400×300 m). Red discs represent the communication zones (e.g, areas covered by RF transceivers).

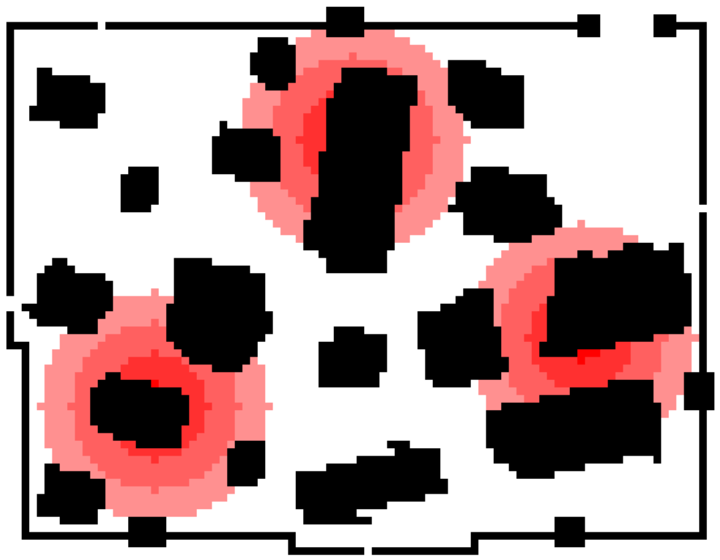


Figure 6.2: Communication map with buffer discretization 1.

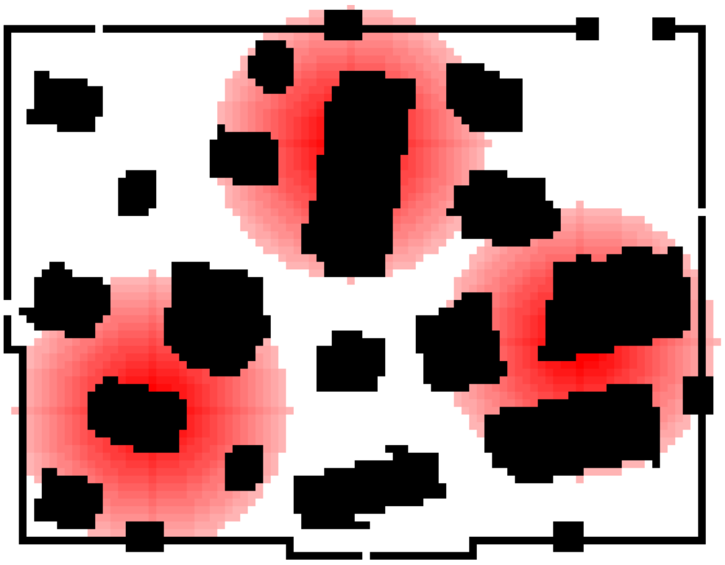


Figure 6.3: Communication map with buffer discretization 0.25.

Buffer discretization. Depending on the discretization of the buffer we have different discretizations of the communication zones. The discretization of the buffer defines the minimum, atomic, amount of information that can be unstacked by the buffer in one time step. In our experiments we use two different discretizations, one with a discretization of 1 (conservative scenario), meaning that the robot can unstack a minimum of 8 Mbit, and a discretization of 0.25 (non-conservative scenario) meaning that the robot can unstack a minimum of 2 Mbit.

Transmission speed. The transmission speed, of each communication node, needs to be in correspondence with the discretization of the buffer, so the transmission rate needs to be a multiple of the minimum amount of memory that can be unstacked from the buffer. In our experiments the transmission speed, apart from the first three cells (12 meters) where it stays constant 32 MBit/s in both discretization scenarios, descreases linearly based on an approximation model of the results of [53]. We consider two discretizations for the transmission rate function corresponding to the buffer discretizations. In the first case, we assume that the rate decreases by 8 Mbit/s each 4 cells (Figure 6.2), so that speeds will be going from 32 to $24, 16, 8$, corresponding to $4, 3, 2, 1$ w.r.t. the buffer unit. In the second case, we assume that the rate decreases by 2 Mbit/s each cell (Figure 6.3) so that speeds will go from 32 to $30, 28, 26...2$, corresponding with $4, 3.75, 3.5...0.25$

w.r.t the buffer unit. (When moving between cells having a different rate, we assume that the robot can transmit at the minimum of the two rates.)

Local path planning algorithm. As a local path planning algorithm (a shortest path algorithm to be used to calculate the shortest path between two points on the graph, without considering the communication, neccessary for the pre-elaboration step) we considered two different algorithms:

- Theta*
- Dijkstra's

Because of a few problem that we are going to explain in the next paragraph we decided to use Dijstra's algorithm for our experiments, even though the paths produced by Theta* are more realistic looking. So for all our experiments planning takes place on the vertices of a uniform 4-connected grid.

Problems with Theta*. The basic problem with Theta* is that it uses a 8-connected grid to calculate the shortest path. Using 8-connected grid makes the discretization of the world difficult, meaning that we would have approximate the time passed in an edge, losing this way the optimality of our algorithm. On Figure 6.4 an example of how the path generated using Theta* is more realistic.

Heuristic algorithm. In each of our experiments we compare the results of our optimal algorithm (algorithm 4.1) with a heuristic algorithm. The heuristic algorithm first finds a feasible path using the method described in Section **4.2** and then it calculates the transmition times using the Algorithm 4.2.

6.2 Experiment Set 1

In the first set of experiments, we keep fixed start and goal locations as shown in Figure 6.5 and, study how the solution cost (expressed as number of time steps) and the runtime of our algorithms vary as a function of the buffer size B. We test our algorithm for both our discretization scenarios described in the section above.

Figure 6.6 shows the results obtained with a buffer discretization 1. There are presented two graphs, one showing how the cost, in terms of robot time steps, varies in dependence of the buffer size for both optimal

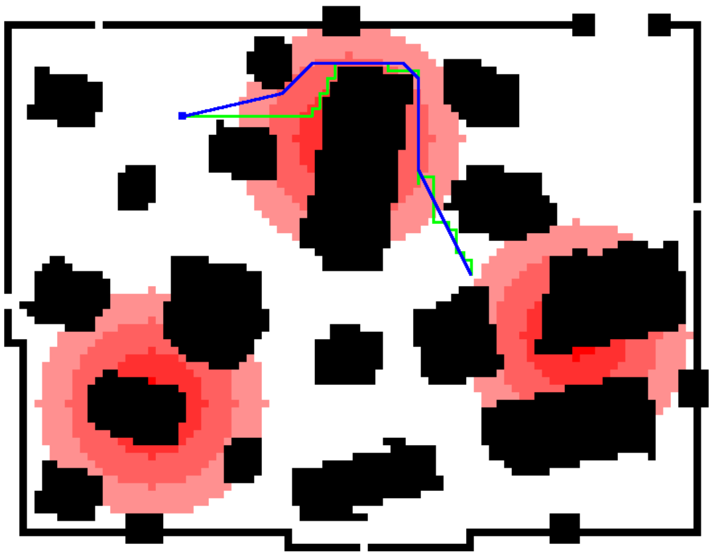


Figure 6.4: Example of path generated by our optimal algorithm using as local planner Theta (in blue) and using as a local planner 4-connected grid with Dijkstra's (in green).*

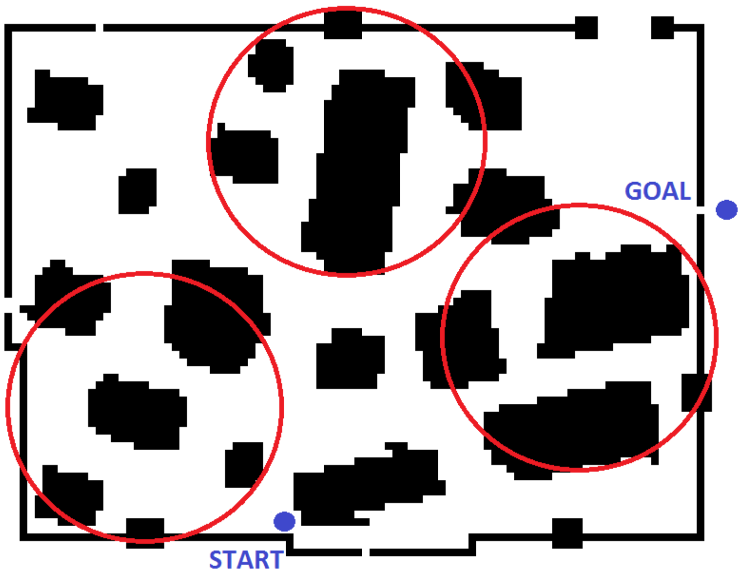
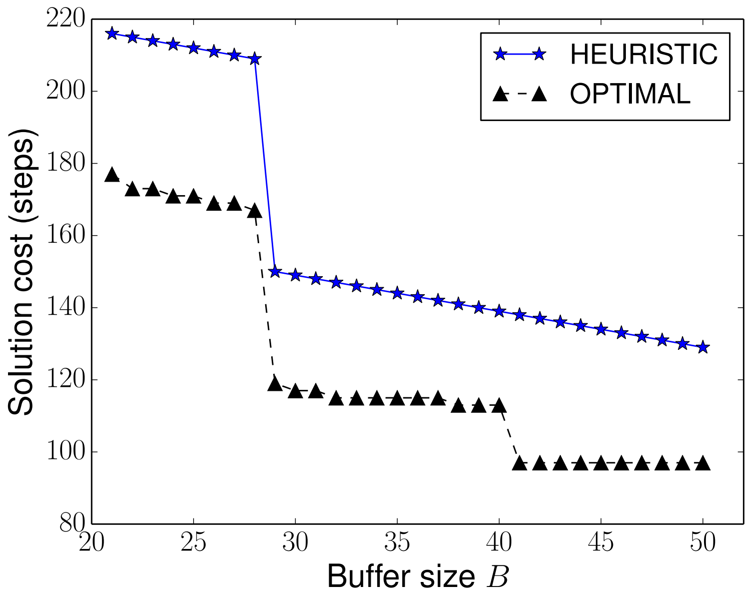
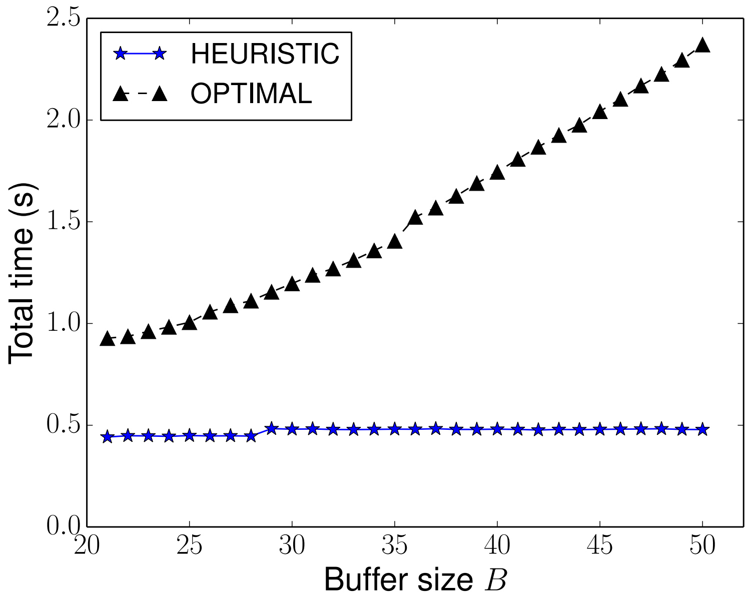


Figure 6.5: First set of Experiments Start and Goal positions

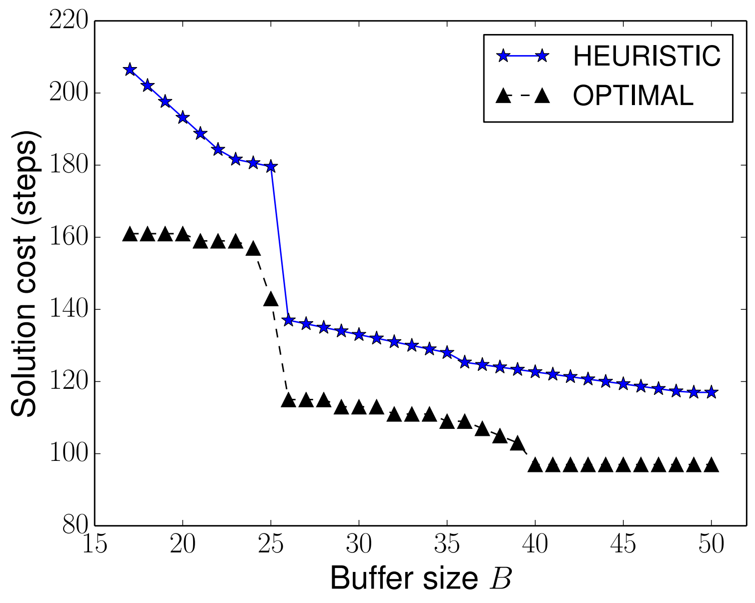


(a)

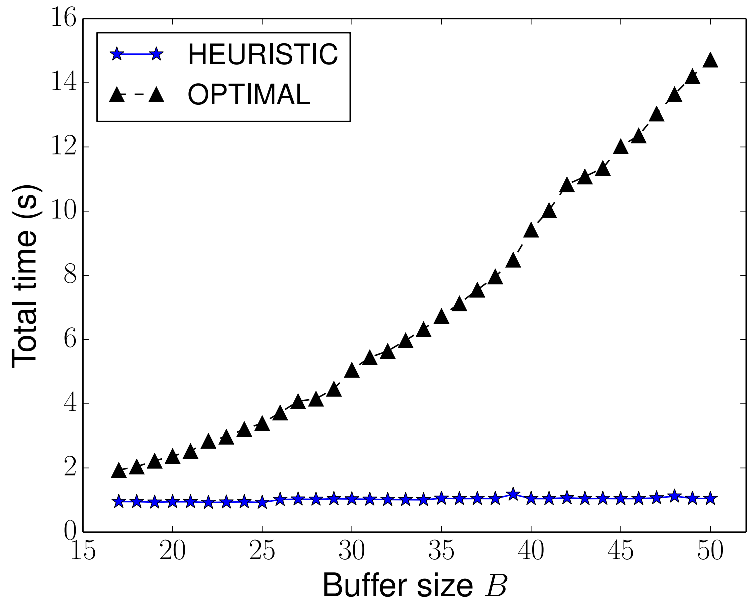


(b)

Figure 6.6: Results of the experiments with buffer discretization 1.



(a)



(b)

Figure 6.7: Results of the experiments with buffer discretization 0.25.

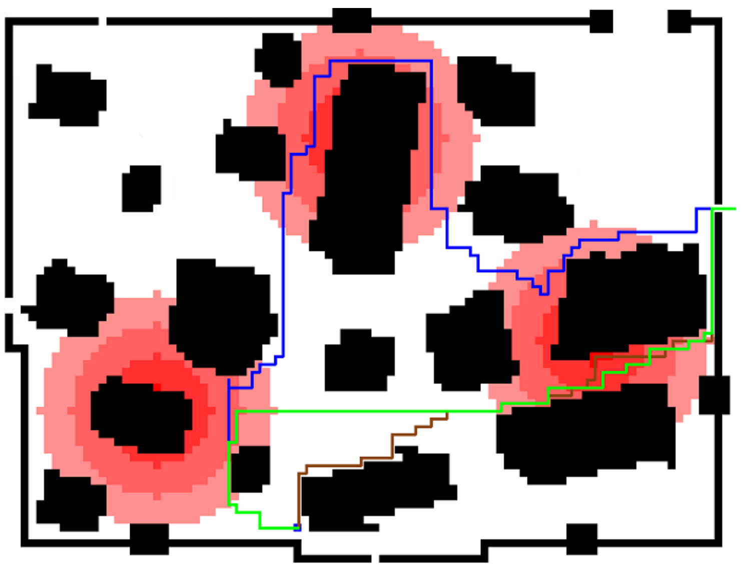
and heuristic algorithms, the other one showing the computational time it takes to calculate the path in dependence of the buffer size. Examining Figure 6.6(a), we can immediately notice how the optimal algorithm behaves significantly better than the heuristic. At the same time, note that both the algorithms are able to obtain feasible solutions from a buffer size $B = 21$. Also, in both cases, the solution cost decreases for increasingly larger buffer sizes. While it is expected for the optimal algorithm, this fact suggests the soundness of the heuristic algorithm. Figure 6.6(b) shows the computing time required by the two algorithms. Clearly, the heuristic algorithm runs faster, but its efficiency does not always compensate for the loss in solution quality. Figure 6.8(a) reports three example paths (we do not explicitly show where the robot stops in a cell for 1 or more steps to transmit data) computed by the optimal algorithm and corresponding to different buffer size values: the smallest value for which we obtain a solution ($B = 21$) and the two values corresponding to sharp changes in the solution cost ($B = 29$ and $B = 41$). Notice how these paths vary in length according to the number of different transmission regions traversed. Moreover, note that the three paths belong to three different "classes", taking routes that cannot be reduced to each other.

Figure 6.7 shows the results obtained with a buffer discretization 0.25. The graphs shown are similar to the graphs built for the buffer discretization 1 (described above). Looking at the solution costs of Figure 6.7(a), we observe the same trends of the previous case with buffer discretization 1. The graphs clearly show that the smaller the discretization of the buffer the smaller the solutions cost, for the same buffer size, is. Compared to the discretization of 1 of the buffer, the runtime of our algorithms (Figure 6.7(b)) increases, but not dramatically. Figure 6.8(b) shows three example paths computed by the optimal algorithm and corresponding to interesting buffer size values. The same considerations made before hold also in this case.

Looking at the running times for both discretizations we can notice that when we have a discretization 1 the running time of the algorithm seems to increase linearly with the size of the buffer and when we have a discretization 0.25 the running time of the algorithm seems to increase exponentialy.

6.3 Experiment Set 2

In the second set of experiments, we again consider both discetizations of the buffer and we compare the solution quality of the optimal algorithm against


(a)

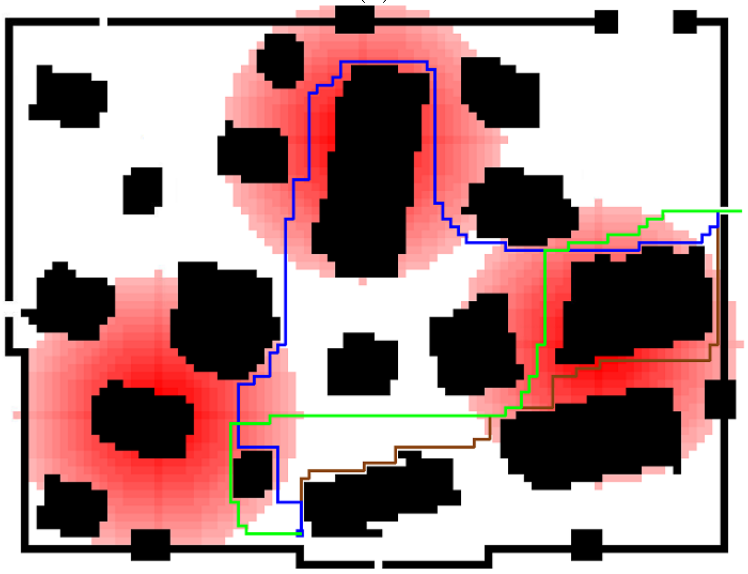
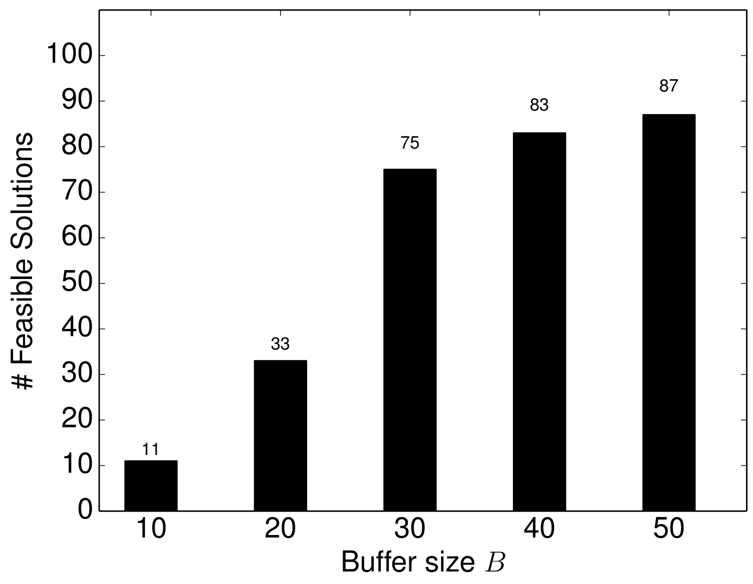
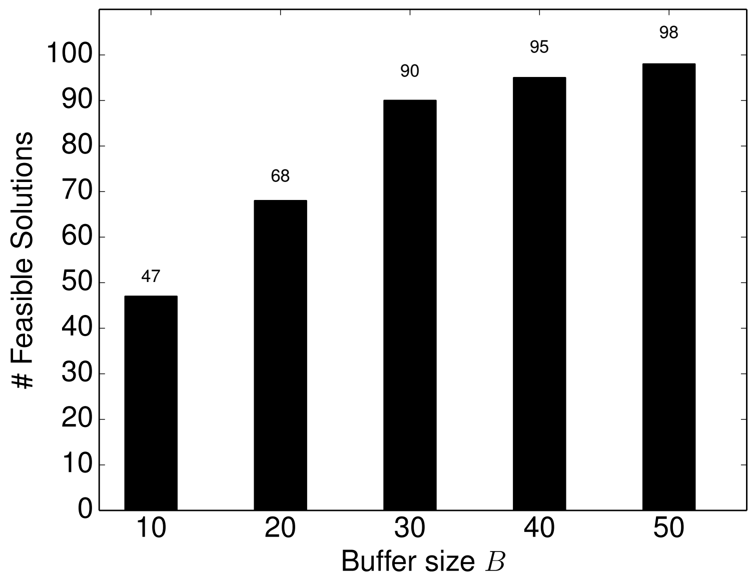

(b)

Figure 6.8: Example paths of the first set of experiments. (a) First set (blue: $B = 21$*, green:* $B = 29$*, brown:* $B = 41$*); (b) Second set (blue:* $B = 17$*, green:* $B = 26$*, brown:* $B = 40$*).*

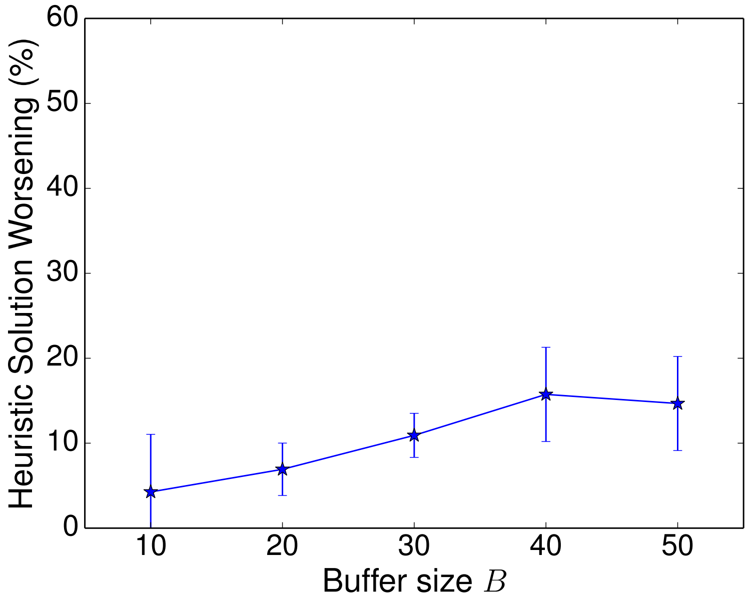


(a) Buffer discretization 1

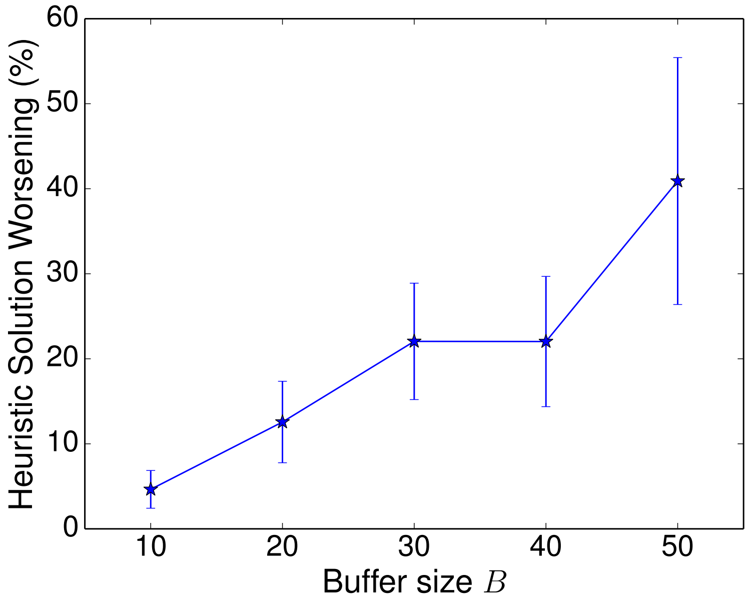


(b) Buffer discretization 0.25

Figure 6.9: Results of the second set of experiments on 100 randomly selected pairs of start-goal locations. Feasible solutions depending on the buffer size.



(a) Buffer discretization 1



(b) Buffer discretization 0.25

Figure 6.10: Results of the second set of experiments on 100 randomly selected pairs of start-goal locations. Heuristic solution worsening in comparison with the optimal algorithm.

that of the heuristic one on 100 randomly selected pairs of start-goal locations for each buffer size. For each fixed instance number, we keep fixed start and goal locations among the two rate scenarios. Figures 6.9(a)-(b) show the number of instances for which the two algorithms return a solution. (Note that, by construction, both the algorithms return a feasible solution whenever there is at least one.) In both scenarios, this number increases with the buffer size, as expected. Clearly, for a fixed buffer size, we are able to solve a larger number of instances in the non-conservative scenario. Figures 6.10(a)-(b) show the average gap (in percentage) between the solution returned by the optimal algorithm and the heuristic, plotted with 95% confidence interval bars. In both cases, the average gap increases as the buffer size increases. Also, note how the average gap of the conservative scenario is significantly smaller than the gap of the non-conservative one for a given buffer size, possibly because it represents a simpler setting.

6.4 ROS Experiments

The ROS experiments are simple experiments, with the purpose to show that our algorithms implementations in ROS work, since the solution cost and the paths found by the algorithms are equal to the ones produced by the C++ implementation. The experiments are run on a mobile robot simulator called Stage [54] and to view the produced paths a package called rviz is used (both are going to be described in the next subsection).

6.4.1 ROS used Tools

Stage Stage is a robot simulator [54]. It provides a virtual world populated by mobile robots sensors, along with various objects for the robots to sense and manipulate.

There are three ways to use Stage:

- The Stage program: a standalone robot simulation program that loads your robot control program from a library that you provide.
- The Stage plugin for Player (libstageplugin)- provides a population of virtual robots for the popular Player networked robot interface system.
- Write your own simulator: the libstage, C++ library, makes it easy to create, run and customize a Stage simulation from inside your own programs.

Stage provides several sensor and actuator models, including sonar or infrared rangers, scanning laser rangefinder, color-blob tracking, fiducial tracking, bumpers, grippers and mobile robot bases with odometric or global localization and was designed with multiagent systems in mind, so it provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity. Our environment in stage view is shown on Figure 6.11.

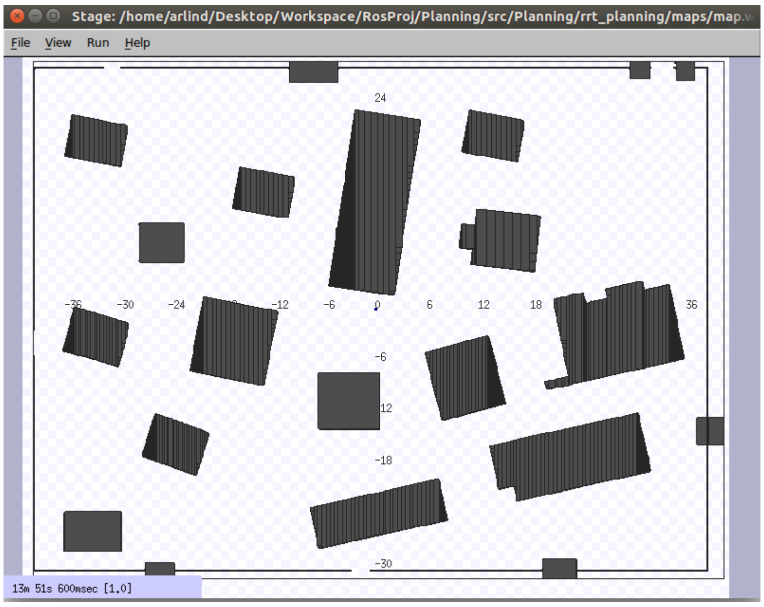
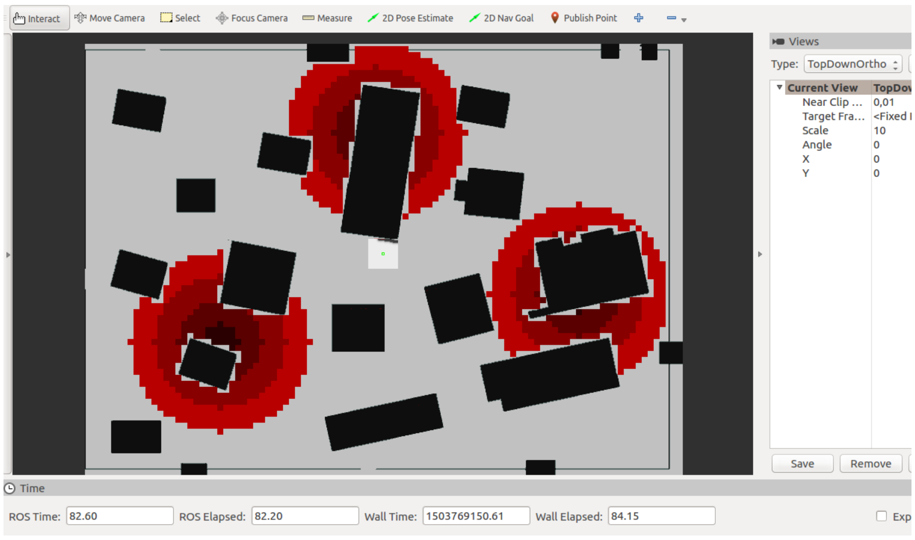

Stage: /home/arlind/Desktop/Workspace/RosProj/Planning/src/Planning/rrt_planning/maps/map.w
File
View
Run
Help
24
-36
-30
-24
-12
-6
0
6
12
18
36
-6
12
-18
-30
13m 51s 600msec [1.0]

Figure 6.11: Stage view of the environment

rviz rviz [55] is a 3D visualizer for displaying sensor data and state information from ROS. We use it to show the paths generated by the algorithms and the motion of the robot. To view the communication zones and the generated path we use markers. Markers are messages (visualization_msgs/Marker) that send basic shapes (cube, sphere, cylinder, arrow) to rviz. With rviz it is possible to set a navigation goal to simulate path generation and the robots motion. Figure 6.12 shows the environment using rviz package with the robot at the center of the environment. (The map is a little bit different from the maps viewed in the sections above because ROS uses costmaps to save the discretization of a map so rviz shows the real map not discretized version, instead in our simulations, done only in C++, in a custom viewer, we show the already discretized map.)

For a complete list of used packages and configuration check **Appendix**



Type: TopDownOrtho

Current View	TopDow
Near Clip ...	0,01
Target Fra...	<Fixed
Scale	10
Angle	0
X	0
Y	0

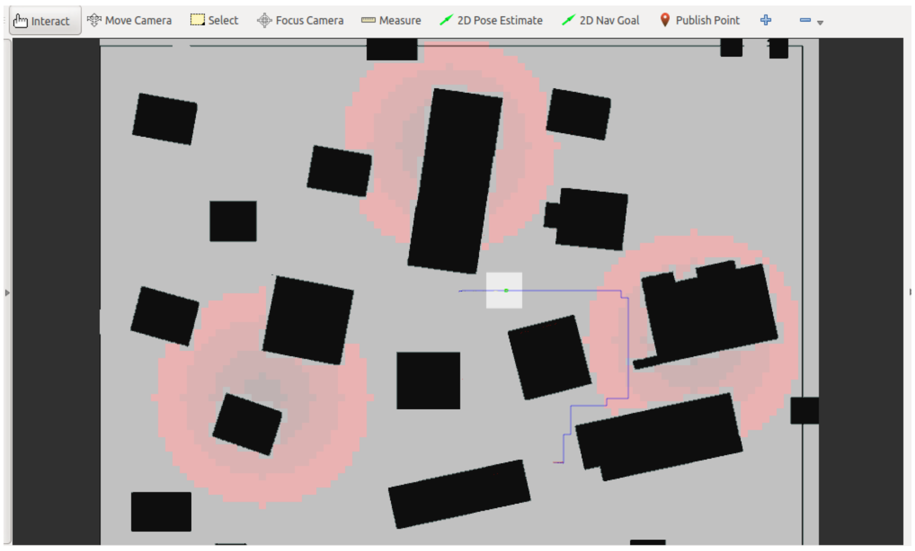
ROS Time: 82.60 ROS Elapsed: 82.20 Wall Time: 1503769150.61 Wall Elapsed: 84.15

Figure 6.12: Rviz view of the environment.

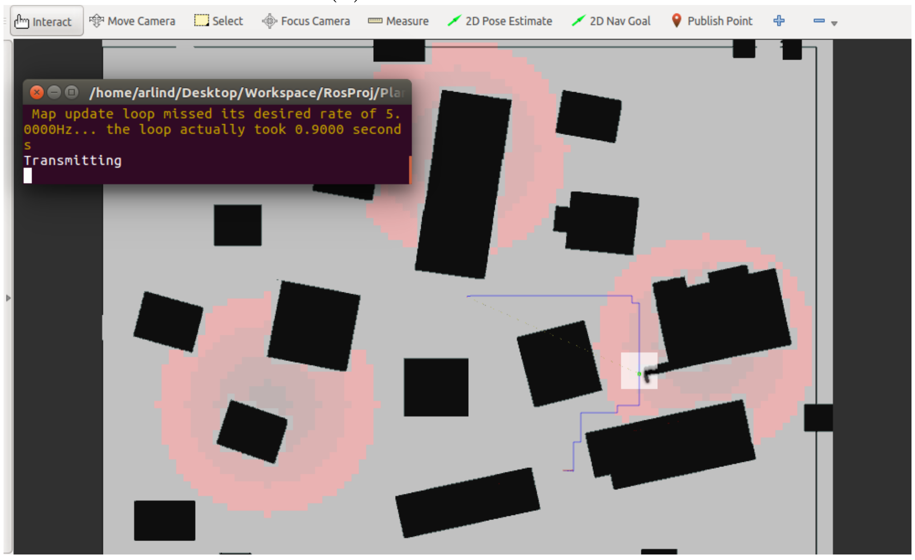
B.

6.4.2 Experiments

The experiments are done with a discretization of the buffer 1. Using rviz we give a goal location to the robot and save the path produced. Figure 6.13(a) shows the path produced by the optimal algorithm and the robot following the produced path. In Figure 6.13(b) the robot stops to transmit data, which for our experiments it means posting on the console the message *Transmitting*. Figure 6.14(a) shows the path produced by the heuristic algorithm and the robot following the produced path. In Figure 6.14(b), as in Figure 6.13(b), the robot stops at a cell to transmit.



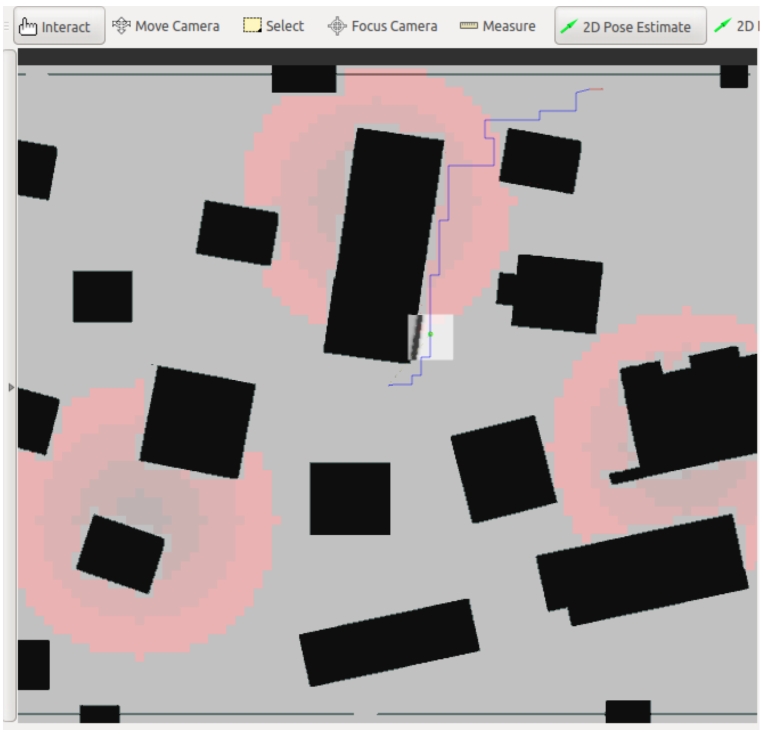
(a) Calculated Path.



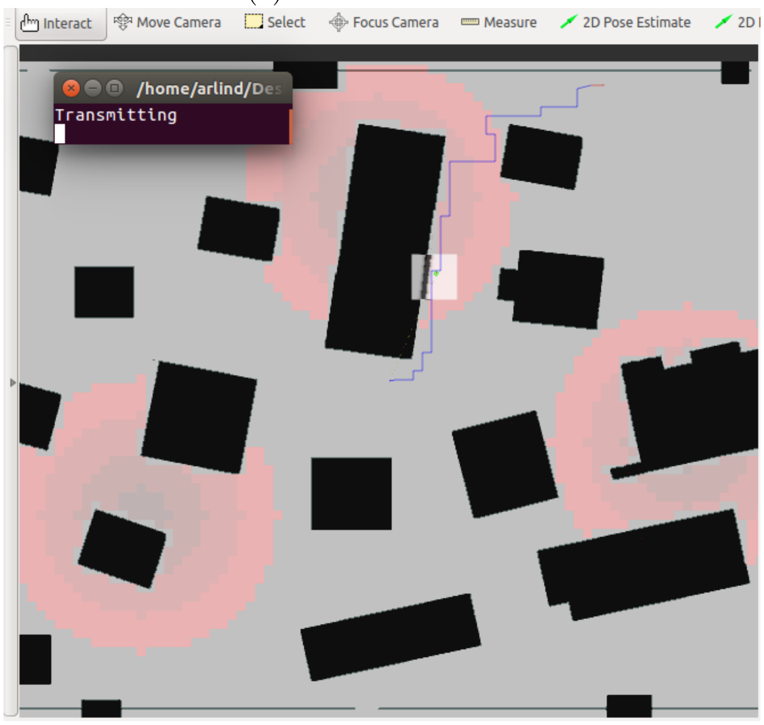
```
 Map update loop missed its desired rate of 5.
0000Hz... the loop actually took 0.9000 second
s
Transmitting
```

(b) Stopping cell to transmit and Transmitting message on the console.

Figure 6.13: Result of the ROS experiments, optimal algorithm.



(a) Calculated Path .



```
Transmitting
```

(b) Stopping cell to transmit and Transmitting message on the console.

Figure 6.14: Result of the ROS experiments, heuristic algorithm.

Chapter 7

Conclusions

In this thesis we considered the problem of finding shortest paths under a limited-buffer constraint, for a robot that acquires data while the time evolves and can transmit them only from some communication zones. We called this problem LBSP. We proposed three algorithms:

- An optimal algorithm that first transforms the given graph and then applies a common shortest path algorithm (Dijkstra's in our case). The complexity of our optimal algorithm is $O(MB|V|^2)$, where B is the buffer size, V is the number of nodes, and M is $1/D$ with D being the discretization parameter of the buffer.

- A feasibility test that, given a problem setting, determines in a fast way if there exists a feasible solution. The complexity of our fesibility test is $O(|V|^3)$, where V is the number of nodes.

- A heuristic algorithm that, given a feasible path, calculates the optimal stopping times at each communication node. The complexity of our heuristic algorithm is $O(|\mathbf{w}|)$, where $\mathbf{w}$ is the feasible path and $|\mathbf{w}|$ is its size.

We implemented the algorithms in C++ and tested them using two sets of experiments with two different discretizations of the environment. In the experiments, the cost of the solutions returned by the heuristic algorithm is comparable with that of solutions found by the optimal algorithm, but the gap increases with the complexity of the setting. The computing time of the heuristic algorithm is consistently shorter than that of the optimal one. Depending on the buffer size, the paths returned by the algorithms can be very different from one another. We also noted, as expected, that the smaller the discretization of the envirenmnet the higher the computing times and the smaller the solution costs of the optimal algorithm.

To further validate our algorithms we implemented them for ROS architecture, and tested them using Stage simulation tool.

Some interesting directions for future research are:

- The investigation of the applicability of the approach proposed by [56] to our constrained path planning problem.
- The Theta* discretizzation of the environmnet and loss of optimality. How would the solutions change and how much time would be lost or gained considering the more realistic paths of Theta*.
- Finding pruning strategies to reduce the number of nodes produced by the optimal algorithm.
- The implementation of communicating nodes in a ROS architecture and studying the impact these nodes would have in the total transmitting time, considering that we would have to add connection latency to the times calculated by the algorithms.
- From a theoretical point of view, it would also be worth it to investigate the NP-membership of the LBSP.

Appendix A

Class Diagram

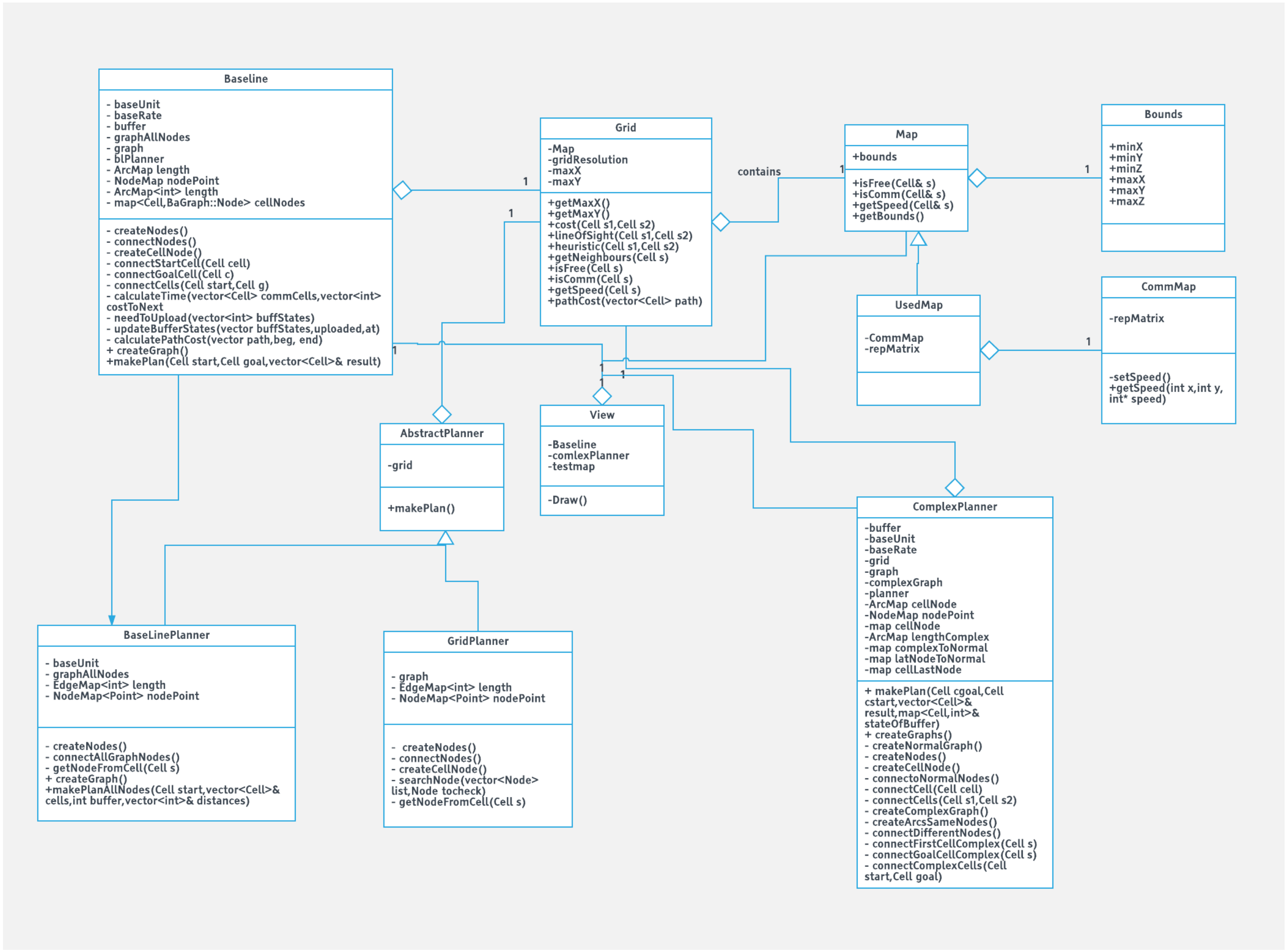


Figure A.1: ER Diagram of the C++ implementation. (Baseline is implementation of the heuristic algorithm.)

Appendix B

ROS complete list packages and ROS configuration

The created ROS package depends on:

- **costmap_2d**
- **nav_core**
- **geometry_msgs**
- **message_generation**
- **message_runtime**

It is build dependent on:

- **catkin**
- **message_generation**

The launch configuration file launches the following ROS nodes:

- **stage_ros**
- **rviz**
- **map_server**
- **move_base**
- **tf**

To be able to build and run the package you need to have all the above packages.
The ROS algorithms implementation code can be found in
https://github.com/Arlind1992/ROStesi. After installing the program using the instructions in the README.md file, to run the program use the following commands:

- source devel/setup.bash - to register the new packages.
- roslaunch rrt_planning heuristic_planner.launch - to launch the planner using the heuristic algorithm.
- roslaunch rrt_planning tesi_planner.launch - to launch the planner using the optimal algorithm.

Bibliography

[1] J. Banfi, N. Basilico, and F. Amigoni, "Minimizing communication latency in mutirobot situation-aware patrolling," in *Proc. IROS*, pp. 616–622, 2015.

[2] M. Hooper, Y. Tian, R. Zhou, B. Cao, A. Lauf, L. Watkins, W. Robinson, and W. Alexis, "Securing commercial wifi-based uavs from common security attacks," in *Proc. MILCOM*, 2016.

[3] J. Jansons and T. Dorins, "Analyzing IEEE 802.11 n standard: outdoor performance," in *Proc. ICDIPC*, pp. 26–30, 2012.

[4] O. Causse and J. Crowley, "Navigation with constraints for an autonomous mobile robot," in *Proc. IROS*, vol. 3, pp. 1899–1905, 1994.

[5] P. Plonski, P. Tokekar, and V. Isler, "Energy-efficient path planning for solar-powered mobile robots," *Journal Field Robot*, vol. 30, no. 4, pp. 583–601, 2013.

[6] G. Hollinger and S. Singh, "Multirobot coordination with periodic connectivity: theory and experiments," *IEEE Transactions on Robotics*, vol. 28, no. 4, pp. 967–973, 2012.

[7] R. Hassin, "Approximation schemes for the restricted shortest path problem," *Mathematics of Operations Research*, vol. 17, no. 1, pp. 36–42, 1992.

[8] M. Bolívar, L. Lozano, and A. Medaglia, "Acceleration strategies for the weight constrained shortest path problem with replenishment," *Optimization Letters*, vol. 8, no. 8, pp. 2155–2172, 2014.

[9] M. Baum, J. Dibbelt, A. Gemsa, D. Wagner, and T. Zündorf, "Shortest feasible paths with charging stops for battery electric vehicles," in *Proc. ACM SIGSPATIAL GIS*, 2015. Paper 44.

[10] "ros.org - about ROS." http://wiki.ros.org/about-ros. Accessed: 2017-07-29.

[11] S. LaValle, *Planning Algorithms.* Cambridge University Press, 2006.

[12] S. Irnich and G. Desaulniers, "Shortest path problems with resource constraints," in *Column generation* (G. Desaulniers, J. Desrosiers, and M. Solomon, eds.), pp. 33–65, Springer, 2005.

[13] M. Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry (2nd ed.).* Springer-Verlag, 2000.

[14] J.-C. Latombe, *Robot Motion Planning.* Springer Science Business Media, LLC, 1991.

[15] "Mcgill - local path planning using virtual potential field." http://www.cs.mcgill.ca/ hsafad/robotics/. Accessed: 2017-08-30.

[16] S. LaValle, "Rapidly-exploring random trees: A new tool for path planning," vol. 264, p. 4, 1998.

[17] "Rrt page - planning for a forward-only car-like robot." http://msl.cs.uiuc.edu/rrt/gallery_carforward.html. Accessed: 2017-08-30.

[18] G. Gini and V. Caglioti, *Robotica.* Zanichelli, 2003.

[19] O. J. Woodman, "An introduction to inertial navigation," 2007.

[20] "GPS - what is GPS?." http://www.loc.gov/rr/scitech/mysteries/global.html. Accessed: 2017-08-27.

[21] "GPS - how does GPS work?." http://www.physics.org/article-questions.asp?id=55. Accessed: 2017-08-27.

[22] M. C. Andreu, *Map-base localization for urban service mobile robotics.* PhD thesis, Universitat Politecnica De Catalunya, 2011.

[23] E. F. Moore, "The shortest path through a maze," *Harvard University Press*, 1959.

[24] "Algorithms - breadth first search." http://alumni.cs.ucr.edu/ tmauch/old_web/cs141/cs141_pages/breadth_first_search.html. Accessed: 2017-08-30.

[25] C. Thomas, L. Charles, R. Ronald, and S. Clifford, *Introduction to Algorithm.* MIT Press and McGraw-Hill., 1990.

[26] "Algorithms - deapth first search." https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/. Accessed: 2017-08-30.

[27] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, pp. 269–271, Dec. 1959.

[28] "Graphs - graph algorithms." http://users.informatik.uni-halle.de/ jopsi/dssea/chap8.shtml. Accessed: 2017-08-27.

[29] F. Michael, L. Tarjan, and E. Robert, "Fibonacci heaps and their uses in improved network optimization algorithms," *Journal of the ACM*, vol. 34, no. 3, pp. 596–615, 1987.

[30] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall., 1995.

[31] K. Daniel, A. Nash, S. Koenig, and A. Felner, "Theta*: Any-angle path planning on grids," *Journal of Artificial Intelligence Research*, vol. 39, pp. 533–579, 2010.

[32] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics and Automation Magazine*, vol. 4, pp. 23–33, 3 1997.

[33] B. P. Gerkey and K. Konolige, "Planning and control in unstructured terrain," in *Proc. ICRA*, 2008.

[34] "dwa_local_planner - ROS wiki." http://wiki.ros.org/dwa_local_planner. Accessed: 2017-07-29.

[35] M. Garey and D. Johnson, *Computers and intractability: A guide to the theory of NP-completeness.* W. H. Freeman, 1979.

[36] J. Desrosiers and M. E. Lübbecke, *Column Generation.* Boston, MA: Springer US, 2005.

[37] L. Lozano and A. L. Medaglia, "On an exact method for the constrained shortest path problem," *Computers Operations Research*, vol. 40, no. 1, pp. 378 – 384, 2013.

[38] E. Martins, "On a multicriteria shortest path problem.," *European Journal of Operational Research*, vol. 16, no. 1, pp. 236–245, 1984.

[39] R. Geisberger, P. Sanders, D. Schultes, and C. Vetter, "Exact routing in large road networks using contraction hierarchies.," *Transportation Science*, vol. 46, no. 3, pp. 388–404, 2012.

[40] B. Dezső, A. Jüttner, and P. Kovács, "LEMON - An open source C++ graph template library," *Electronic Notes in Theoretical Computer Science*, vol. 264, no. 5, pp. 23–45, 2011.

[41] "ROS basic concepts - ROS wiki." http://wiki.ros.org/custom/images/wiki/ROS_basic_concepts.png. Accessed: 2017-07-29.

[42] "Services - ROS wiki." http://wiki.ros.org/Services. Accessed: 2017-07-29.

[43] "Topics - ROS wiki." http://wiki.ros.org/Topics. Accessed: 2017-07-29.

[44] "map_server - ROS wiki." http://wiki.ros.org/map_server. Accessed: 2017-07-29.

[45] "amcl - ROS wiki." http://wiki.ros.org/amcl. Accessed: 2017-07-29.

[46] "move_base - ROS wiki." http://wiki.ros.org/move_base. Accessed: 2017-07-29.

[47] "Navigation stack - ROS wiki." http://wiki.ros.org/navigation/Tutorials/RobotSetup. Accessed: 2017-07-29.

[48] "map_server - ROS wiki." http://wiki.ros.org/navigation. Accessed: 2017-07-29.

[49] "carrot_planner - ROS wiki." http://wiki.ros.org/carrot_planner. Accessed: 2017-07-29.

[50] "navfn - ros wiki." http://wiki.ros.org/navfn. Accessed: 2017-07-29.

[51] "global_planner - ROS wiki." http://wiki.ros.org/globalPlanner. Accessed: 2017-07-29.

[52] "map_server - ROS wiki." http://wiki.ros.org/base_local_planner. Accessed: 2017-07-29.

[53] J. Jansons and T. Dorins, "Analyzing ieee 802.11n standard: outdoor performanace," in *Proc. ICDIPC*, pp. 26–30, July 2012.

[54] "Stage - the robot simulator." http://rtv.github.io/Stage/. Accessed: 2017-08-30.

[55] "ROS wiki - rviz." http://wiki.ros.org/rviz. Accessed: 2017-08-30.

[56] S. Bhattacharya and V. Kumar, "Persistent homology for path planning in uncertain environments," *IEEE Transactions on Robotics*, vol. 31, no. 3, pp. 578–590, 2015.