CHALMERS

A method for multi-agent exploration planning

Master's Thesis in Computer Science: Algorithms, Languages and Logic

ALEXANDER CHEKANOV

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2012

The Author grants to Chalmers University of Technology the non-exclusive right to publish the Work electronically and in a non-commercial purpose make it accessible on the Internet.

The Author warrants that he/she is the author to the Work, and warrants that the Work does no contain text, pictures or other material that violates copyright law.

The Author shall, when transferring the rights of the Work to a third party (for example a publisher or a company), acknowledge the third party about this agreement. If the Author has signed a copyright agreement with a third party regarding the Work, the Author warrants hereby that he/she has obtained any necessary permission from this third party to let Chalmers University of Technology store the Work electronically and make it accessible on the Internet.

A method for multi-agent exploration planning
ALEXANDER CHEKANOV

© ALEXANDER CHEKANOV, 2012

Supervisors: K. V. S. Prasad, Igor V. Rudakov
Examiner: Graham Kemp

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46(0)31-772 1000

Abstract

This project is concerned with the problem of exploration planning, which arises in systems of one or several mobile robots set with a task of exploring the map of their environment. Today the most popular algorithm that deals with this problem is the naive (greedy) approach, which is very simple and usually shows reasonably good results. This algorithm performs poorly in the worst case with several robots however. To cope with this problem, a new approach is developed and analyzed. This approach, called *coordinated breadth-first search*, is shown to guarantee linear scaling and to be asymptotically more efficient than the naive method in the worst case. To test these findings a computer simulation was developed which admits both algorithms and arbitrary maps. Finally, a comparison between the algorithms is made and further improvements are suggested.

Acknowledgements

I would like to thank my supervisors K.V.S. Prasad and Igor V. Rudakov and my examiner Graham Kemp for the advice and suggestions that I received from them while working on this project. I would also like to thank a Chalmers graduate Jonas Einarsson for developing a LaTeX template for the Master thesis, which I used to write this report. Finally, I would like to thank my friend Evegeniy Baranov for valuable discussion of this project and for testing the computer simulation that I developed.

Alexander Chekanov
Gothenburg, Sweden
June 17, 2012

Contents

1 Introduction **1**

1.1 Motivation . . . 2

1.2 Project goals . . . 3

1.3 Thesis outline . . . 3

2 Background theory **4**

2.1 Map representation . . . 5

2.2 Simultaneous localization and mapping . . . 6

2.3 Map correspondence . . . 8

2.4 Navigation . . . 8

2.5 Exploration . . . 9

2.5.1 Naive approach . . . 10

2.5.2 Improvements and other methods . . . 11

2.5.3 Analysis . . . 12

3 Coordinated breadth-first search **15**

3.1 Requirements . . . 15

3.2 The idea . . . 17

3.3 Data structures and algorithms . . . 20

3.3.1 Frontier tracking . . . 20

3.3.2 Junction tree . . . 21

3.3.3 Synchronization . . . 23

3.4 Algorithm details 24
3.5 Analysis 27
3.5.1 Graph and tree correspondence 27
3.5.2 Running time complexity 29
3.5.3 Scaling with multiple robots 31

4 Computer simulation **33**
4.1 Development platform 33
4.2 Model of the environment 34
4.3 Assumptions 35
4.4 Data structures implementation 37
4.5 Interface 39

5 Discussion **42**

Bibliography **44**

1

Introduction

ROBOT mapping is a problem that has received much attention over the past two decades. Almost any robot that moves in the environment needs a map to plan its route and navigate. This includes robots that guide tourists in museums [1], autonomous vacuum cleaners [2] and others. A special place among them is occupied by the robots that have mapping as their primary objective — they make maps of abandoned mines [3], polar regions [4] and underwater zones [5].

The problem of making maps is known as *simultaneous localization and mapping* (SLAM) and is considered to be one of the most important topics in mobile robotics. Various solutions to this problem have been proposed and successfully implemented and the problem is considered to be solved on the conceptual level [6]. Today mobile robots can use existing SLAM methods to localize themselves in their environments and to build maps as needed.

For the case when making maps is the goal, several systems that combine SLAM algorithms with navigation and exploration planning methods have been developed. Among them there is a notable group of systems that use multiple robots to achieve the goal [7, 8, 9]. This approach has several important benefits. Firstly, the exploration time with several robots can be much lower due to their collaboration and map sharing. Multiple robots also introduce redundancy into the system, making it a lot more fault-tolerant. Finally, using several smaller robots

can be simply much cheaper than using a big and expensive one.

To illustrate this, consider the following example. There is an abandoned mine that is hard and/or dangerous to explore manually. The mine is deep, so GPS or other wireless connections with the surface are unavailable. If we send the robots into the mine, we want them to produce a feature-rich and accurate map, while taking as little time (and other resources, like fuel) as possible to do that. To complete this objective, the robots must have a plan on how to choose directions to maximize their efficiency. This particular part of the robotic system that uses multiple robots is the focus of this thesis project.

1.1 Motivation

The general idea behind multi-robot exploration approach is simple — several robots are put into the environment and explore it as if they were independent. When two robots enter communication range, they transmit their obtained map data to each other and synchronize their maps. Then they agree on a further exploration plan and head out once more, reducing the workload for each robot by a good deal.

Despite sharing the same basic idea, systems that use multi-robot exploration vary greatly in their details. Some of them, like the initial paper by Burgard et al. [7], don't take into account the limited communication range and focus on choosing the most efficient exploration destinations for the robots. Their further work [10] improves in this regard, but still focuses on the same aspect of exploration. The work by Thrun et al. [8] largely focuses on a SLAM method and then extends it to work with multiple robots. Another work by Thrun et al. [11] focuses on the map correspondence problem. Finally, a more recent work by Zhou et al. [9] is also focused on choosing optimal destinations after performing the synchronization procedure.

All of these systems have one thing in common — they sidestep the question "when are the robots going to meet?" and either don't consider the communication range limit at all or assume that robots will meet at some point. This is begging the question — is there a way to efficiently organize the meetings for the robots to reduce the time they spend exploring? Are there cases when the robots don't

meet at all while following these directions? Intuition suggests that failing to meet may be a significant issue, and taking it into account can improve the efficiency of the exploration.

1.2 Project goals

This project aims to develop an exploration planning method that perform consistently while showing results that are comparable to currently existing approaches. This includes building a computer simulation model with a fully functional multi-robot exploration system to test the methods.

More formally, the project has the following goals:

1. identify the requirements for an exploration planning method;
2. review existing exploration planning methods;
3. analyze the naive approach to exploration planning;
4. develop an exploration planning method that allows robots to explore the territory with consistent results;
5. develop simulation software to experimentally test the exploration planning methods;
6. compare and discuss the results and provide conclusions on the efficiency and applicability of the developed method.

1.3 Thesis outline

The second chapter reviews the parts of a multi-robotic exploration system and discusses the existing approaches to the exploration planning problem. The third chapter introduces the developed exploration planning method and performs an analysis of its shortcomings. The fourth chapter goes into detail about computer simulation, including its features and limitations. Finally, the fifth chapter discusses the results, draws conclusions and summarizes the work.

2

Background theory

FOR a multi-robotic exploration system to be operational, it must be able to perform several basic functions. Every robot must be able to localize itself within the environment, make maps based on its sensor data input, merge map data obtained from other robots into its own map, plan its exploration route and navigate to the chosen destination. All of these functions require and impose several constraints on formats of input and output data and various uncertainty parameters in the algorithms. This chapter identifies these constraints and explains the choice of algorithms that are used in the simulated system.

Before going into detail about theory, a few important assumptions must be made. The goal of this project is to develop and test an exploration planning method, so all other functions are auxiliary to it. Since the testing is done by the means of simulation, it provides an excellent opportunity to replace the computationally intensive and (relatively) difficult to implement SLAM and map correspondence methods with much simpler placeholders. This can be done by giving robots perfect sensors and actuators, as well as simplifying the territory to fit the selected map discretization well. The justification of these simplifications is given in description of specific algorithms in this chapter and is summarized in chapter 4.

2.1 Map representation

The main objective of this system is to make maps, so it is only natural that reducing the continuous and very irregular environment to a discrete map will influence many features of all other parts of the system.

The map used in this system must be possible to produce concurrently with any popular SLAM method and must allow easy path planning and navigation. At the same time, it should preserve enough features of the environment to satisfy the mapping goal.

An approach called "occupancy grid map" [12] satisfies these requirements. It performs well when used for exploration and path planning [13, 14], works with typical SLAM methods and many different kinds of sensors [15], supports both 2D and 3D environments [15] and produces feature-rich maps.

The idea behind occupancy grid is to divide the map into a rectangular grid, where each cell contains a random variable indicating probability whether this cell has an obstacle or not. These probabilities are updated as the robot estimates the obstacles around him from different angles, resulting in a high-fidelity map. This approach also works well with perfect sensors — in that case the grid variables are limited to values of 0 (no obstacle), 1 (obstacle found) and 0.5 (cell has not been explored yet).

A notable disadvantage of occupancy grid maps is that they require good precision of robot localization method to work well. It has been shown that this is possible to achieve with regular SLAM methods [14], and in case of perfect sensors and actuators it is not a concern at all.

Finally, an occupancy grid map can be easily converted to and from a simple grayscale bitmap, making it a perfect candidate for map format in this system.

A simple occupancy grid map fragment is shown in figure 2.1.

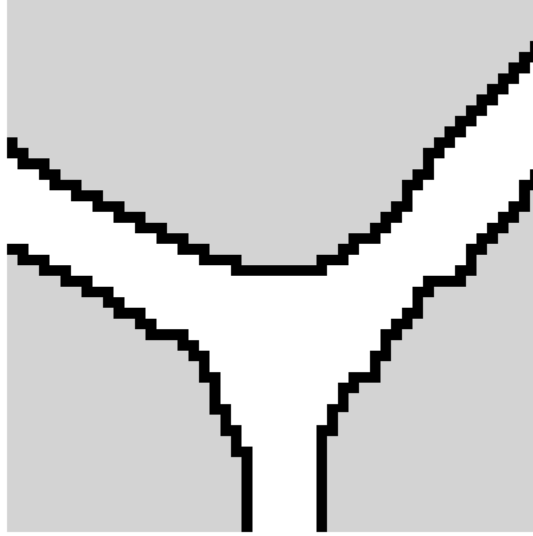


Figure 2.1: Occupancy grid map fragment. White cells correspond to the passable terrain, black cells correspond to the unpassable terrain, and gray cells correspond to the unknown parts of the map.

2.2 Simultaneous localization and mapping

The problem of simultaneous localization and mapping asks if it is possible for a mobile robot in an unknown environment to incrementally build a consistent map while simultaneously determining its location within that map [6]. This actually contains two problems — incremental mapping problem and localization problem. The reason why they are studied together is that they are dependent on each other — to build a map, robot needs to know where it is, but to know that it needs a map. To make matters worse, both sensors and actuators of the robot tend to be noisy, which makes processing their results a difficult problem.

To define the problem formally, the following notation is often used in SLAM literature:

- $\mathbf{m} = \{\mathbf{m}_0, \mathbf{m}_1, \ldots, \mathbf{m}_n\}$ — the set of all landmarks on the territory;
- $\mathbf{x}^t = \{\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k\}$ — the history of robot locations for time from 0 to k;
- $\mathbf{u}^t = \{\mathbf{u}_1, \mathbf{u}_2, \ldots, \mathbf{u}_k\}$ — the history of robot movement commands;
- $\mathbf{z}^t = \{\mathbf{z}_0, \mathbf{z}_1, \ldots, \mathbf{z}_k\}$ — the history of sensor readings.

Using this, we can write the probability distribution

$$p(\mathbf{x}_k,\mathbf{m}|\mathbf{z}^t,\mathbf{u}^t,\mathbf{x}_0), \tag{2.1}$$

computing which for every point of time solves the SLAM problem [16]. This definition is known as the probabilistic form of the SLAM problem and is commonly used today.

A number of solutions were proposed to solve the this problem. Some of the more notable among them are:

- EKF-SLAM — an approach using Extended Kalman Filter, which was first introduced in 1986 [17];
- FastSLAM — an approach using particle filters [16];
- SEIF-SLAM — an approach using extended information filters [18], which is easily extendable to the multi-robot case [11].

With these methods at hand, the SLAM problem is considered to be solved on theoretical and conceptual level today [6]. That means that its solutions provide consistent maps and robot location with reasonable precision.

This fact makes the SLAM part of this project much easier to implement. Since the small variation between results of these methods does not matter for a higher-level problem of path planning, the choice of particular method does not really matter, as long as the interface of the system conforms to the definition given above. That provides an opportunity to obtain the value of posterior probability directly by the virtue of perfect sensors and actuators in the simulation without any noticeable difference for the path planning algorithm.

There are two major benefits of this simplification. SLAM algorithms use relatively complex mathematical models, which are not trivial to understand and implement correctly. The other benefit comes from much reduced computation difficulty during the actual simulation — SLAM algorithms scale at least linearly with the potentially large amount of landmarks on the field. With many robots simultaneously making their way through the field, the simulation could take a long time to finish. Obtaining data using perfect sensors prevents that.

2.3 Map correspondence

The map correspondence problem deals with merging maps obtained from two robots. To solve this problem, two tasks must be completed [11]: the coordinate transformation (including rotation angle and translation vector in 2D case) must be found and the correspondence between the two sets of landmarks must be established.

The main difficulty with this problem is once again the noise in the input data, in this case in the maps obtained by the robots. Even with the best available SLAM algorithms, the maps remain slightly imprecise. While their precision is good enough for robot navigation and human purposes, it causes issues for the merging process. If the same landmark is expressed differently on two maps, it is possible to miss that correspondence and to have a duplicate landmark on the merged map. This can result in a mess once this map is merged a sufficient number of times between different robots.

There are two different major approaches to this problem. Some authors [7, 10, 19] require that robots identify their relative positions before merging maps. Others [9, 11, 20] focus on finding correspondence between landmarks using probabilistic methods, lifting that requirement. The drawback of the second approach is that the merger may be impossible if the maps have no overlap at all.

By the virtue of perfect sensors in the simulation, map correspondence problem becomes trivial. In this project we assume that robots start in the same place, giving them a known initial correspondence point. This point can be used to merge the maps easily by simply adding all unknown points from the received map. The case of more than two robots meeting at once is also simple — every robot can transmit its map to all other robots, which will merge these maps into their own individually. Since the maps are perfect, there is no risk of increasing the error by performing the merger multiple times.

2.4 Navigation

Navigation is the problem of choosing movement commands that will get the robot to his selected destination. Using the occupancy grid map, this problem is essen-

tially reduced to finding the shortest path through unoccupied spaces in the grid. The grid can be naturally represented as a graph where all unoccupied cells are connected to other unoccupied cells next to them by edges of unit cost.

Shortest path problem on a graph has many well-known solutions. In robotics, as well as in similar simulation problems (like video games), algorithms like A* and its many derivatives are used.

A* is a best-first search algorithm that uses a heuristic function $h(n)$ defined for every cell in the grid [21]. A* starts at the origin node and then at every step expands one node n for which sum of the cost to get there $g(n)$ and the heuristic value $h(n)$ is the smallest. The algorithm terminates once it gets to the goal. In order to find an optimal path, A* must use a heuristic function that never overestimates the cost to the goal. In case of the 2D grid map with no diagonal movement allowed, a simple admissible heuristic can be $\Delta x + \Delta y$, i.e. the sum of coordinate differences between search origin and the goal. This metric is also known as Manhattan distance.

There are two main disadvantages of A*. First one is that it takes a lot of space to keep all visited nodes in memory. An approach called "iterative deepening" (and algorithm IDA*) aims to solve this issue. It reduces the space requirement, but takes more time to find the path. For the purpose of this simulation, space is less important than computation speed, since robots are not likely to run out of memory while processing relatively small maps.

The other disadvantage is that A* produces discrete unnatural movements that do not look like anything a real robot would do. A number of modifications for A* (like Hybrid A* [22]) provide solutions to this problem, producing natural continuous paths. However, for this simulation discrete movements between centers of the cells are more than enough.

2.5 Exploration

The exploration planning problem deals with choosing the next target for exploration. Formally, a solution to this problem must provide the next target to the navigation system at any moment of time, given the currently known map, the position of robot in it, all available and relevant knowledge about other robots (their

destinations, positions, explored regions), which can be precise or estimated, and any other possibly useful data. This target must be chosen so that the total time spent exploring is minimized.

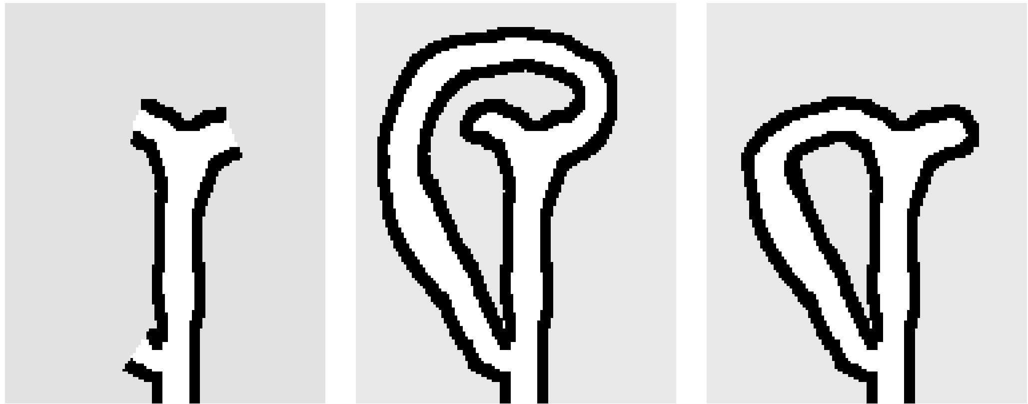
In this project, the exploration planning algorithm is called exactly once per discrete time interval and is given the occupancy grid map state, the robot's own position and the positions of other robots, if they are in range. It must perform all necessary estimations based on the input data and any state that it saved from previous calls. It can also be assumed that the robots start the exploration process from the same spot.

This problem is difficult. Even for the fully explored map, finding the shortest path that checks every corner of the map is already NP-complete (it is known as a "traveling tourist problem"). If the map is known only partially (and there is no way to precisely predict the structure of the unknown parts), an optimal solution for the general case simply does not exist.

This is easy to prove. Refer to the figure 2.2. Subfigure (a) shows the state of the world known to the robot. Subfigures (b) and (c) show two possible world states that can be uncovered from that known state. In either case, for the robot located at the top junction, it would be better to explore the small dead-end first and then follow the loop back to the beginning. But he has no way to know which choice leads to that dead-end (if any), so for any choice that he decides on there exists a map for which that choice would be suboptimal. Obviously, the same reasoning can be applied to the robot which choses the left path in the first junction. Thus no algorithm that can find an optimal solution for every map exists.

2.5.1 Naive approach

A simple greedy approach to this problem exists [7, 23, 24]. It suggests the robot to choose the closest unexplored point and head there. This algorithm will terminate when there are no reachable unexplored points, thus exploring the whole map. It also tends to produce very good results in the case when there is just one robot — if the robot can't deduce anything about the feasibility of its possible paths, then the best choice is to pick the closest one. Intuitively, this algorithm is very similar



(a) State known to the robot (b) One possible true state (c) Another possible true state

Figure 2.2: Exploration planning problem does not have a general solution

to a depth-first search algorithm of a graph.

The multi-robot version of this algorithm just makes the robots who are in the communication range of each other and at the junction pick different branches of the map to explore. This seems to be a reasonable extension and one might expect it to work just as well for multiple robots. It does work well, but has a very notable problem, which will be discussed later in this section.

2.5.2 Improvements and other methods

There has been surprisingly little interest in this specific problem in the scientific community. There is a number of works that focus on choosing the navigation target, but few of them concern themselves with minimizing the exploration time. Many of these works [25, 26] propose that the robots should stay together to reduce sensor errors , which obviously is not very useful today considering the progress in the SLAM field.

A notably interesting approach was suggested by [27]. It assumes that the robots start in different (and mutually unknown) positions on the map and then try to solve the rendezvous problem [28] while exploring the territory. The downside is that this method is far from optimal if the robots have already met once. Since this project considers the case when the robots start from the same spot, this

method will not be considered in the following chapters.

A solution that was strictly focused on minimizing exploration time was proposed by Burgard et al. in [7]. The initial version of this algorithm was not very useful by itself — it considered the communication range to be infinite and infallible. An extension to this method was provided later by the same team [10], which solved this issue. This method is also a greedy algorithm like the naive approach, but instead of simply choosing the closest unexplored area, it introduces a special utility function. This utility is based on the robots' beliefs (or precise knowledge, as in the earlier paper) about the positions and known world states of the other robots, so that the areas which are likely to be explored by others are given lower priority. Then, based on this utility and the distance to available unknown points, all robots in the communication range are assigned targets.

While improving on the naive approach, this algorithm does not consider the cases when the robots fail to meet at all or only do so very rarely (and merge little data as a result), which makes it suffer from the same problem. For the purpose of this thesis, the naive approach and this algorithm can be considered to be similar and analyzed together in face of that single problem. Since naive approach is easier to implement and has been analyzed for the case of a single robot [24], only it will be considered further, but all qualitative conclusions can be extended to this superior method.

2.5.3 Analysis

The single-robot version of the naive approach has been analyzed by Koenig et al. [24]. They have shown that the worst-case performance of this greedy algorithm is bounded from above by $O(|V|^{3/2})$, where $|V|$ is the number of vertices in the graph which the robot traverses. Unfortunately, they do not provide any way to construct a graph from the actual map, but from their illustrations it may seem that the graph reflects topological properties of the map. Since this algorithm can be run on any graph, for convenience in this project it will be considered that the graph is obtained from the map in the following way:

1. every junction on the map is a vertex;

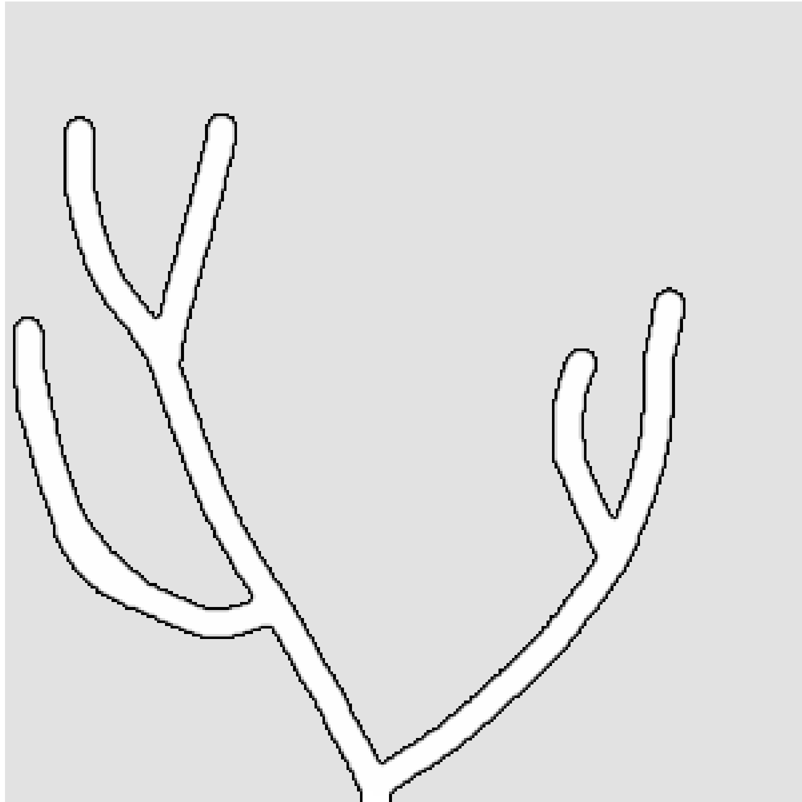


Figure 2.3: An example map where the naive algorithm fails to scale

2. every dead-end is a vertex;
3. two vertices are connected by an edge iff their respective junctions/dead-ends are connected by a stretch of a road.

The precision with which the junctions and dead-ends are determined can be arbitrary.

With a graph like this, all available paths are preserved while all unnecessary details are omitted.

This constitutes analysis for the single-robot version of the algorithm. The multi-robot version is however more tricky. Koenig states that the algorithm scales well, but provides no argument to back up his words. Unfortunately, he is only half right.

The algorithm indeed scales decently most of the time. However, there exist map configurations on which the algorithm does not scale at all — when multiple

robots perform exactly the same as a single robot. This happens when the robots fail to meet or when these meetings don't give them much new information. An example of such map for two robots is shown in figure 2.3.

While discovering the general relationship between the map topology and scaling for the naive algorithm is beyond the scope of this project, some conclusions could be derived from the experiments:

- the algorithm may fail to scale on both tree-like maps (like in figure 2.3) and maps with loops;
- for the tree-like maps, n robots can miss each other if there are n first-turn branches of similar length and they have two smaller branches in the beginning, one of which is explored by robots when they start exploring their parent branch and the other is explored on the way back. This structure allows the robots to pass each other while they explore these small branches, so that they can switch the branches without meeting.
- if the robots only meet close to the end of the exploration, the amount of shared data will be small and they will not gain much from it;
- the robots may sometimes follow each other just outside of their communication range, thus performing redundant exploration;
- some of the robots may meet and complete the exploration, while the others fail to do so and explore the whole map by themselves.

Formally, the failure to scale means that the worst case running time complexity of the naive algorithm for n robots remains $O(|V|^{3/2})$, independent of n.

3

Coordinated breadth-first search

THE deficiency of the naive and Burgard's methods shown in the previous chapter can be crippling for some applications. This chapter presents a different approach to the exploration planning problem, which was developed in this project specifically to overcome this problem.

3.1 Requirements

The issue with these algorithms comes from their approach to coordination. While they successfully decide on reasonably efficient areas to explore at every point when the robots are within communication range, they do not consider places where the robots may meet. Therefore, for an algorithm to consistently synchronize the data between multiple robots on any map configuration, it must be built around guaranteed meetings after every certain point in the exploration.

This is not enough by itself. The algorithm must scale approximately linearly with the number of robots on the map, assuming the branching factor of the map allows it. Here is an example of a bad algorithm, which schedules the meetings correctly and seems to distribute the workload evenly, but will scale poorly on many maps: *Let the robots take separate branches while they are traveling together and fully explore these branches once they are alone (by the means of some exploration plan). Once they are done exploring their branches, they must return to the start*

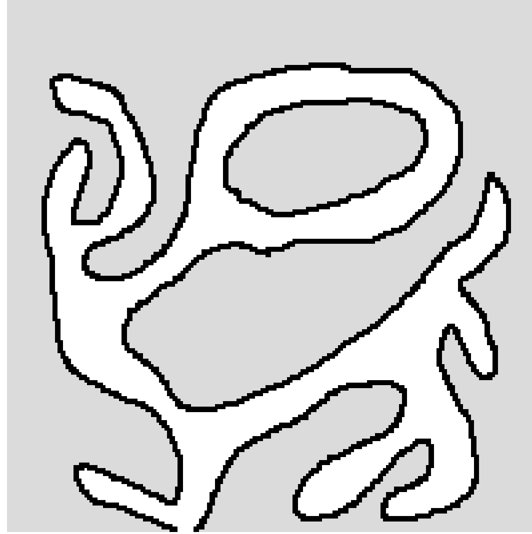


Figure 3.1: A map where planning meetings is not enough to obtain good results

point and wait there for everyone, synchronizing their findings and completing the exploration.

Why is this algorithm bad? Check the map on the figure 3.1. Using this algorithm, one of the robots will explore the left branch and stop, exploring only a small fraction of map, while the other robot does all the work. This result is obviously not desirable and produces the same kind of results as the naive worst case, but for a different reason.

This requirement includes planned meetings in it. This is obvious — if any robot fails to meet with the others, it will never transmit its data to them and its work will be wasted. Since speed increase beyond linear is impossible (n robots can cover at most n times more distance than one robot in the same time), this case will result in sub-linear improvement, which breaks the constraint.

Finally, a note needs to be made about maps with very small branching factor and a large number of robots. For some maps (like a completely linear map with no branches at all) it is impossible to satisfy this requirement. Any algorithm will perform poorly on maps like this, and therefore the only maps worth considering are those with sufficient branching factor to accommodate all robots most of the time.

3.2 The idea

As evident from the name, coordinated breadth-first search (CBFS) borrows the idea from the common graph search algorithm. Before explaining it in detail, it is necessary to define several terms:

Frontier cell
: an explored passable cell on the occupation grid map which is adjacent to one or more unexplored cells.

Frontier node
: a set of adjacent frontier cells such that no other frontier cell can be added to it.

Origin
: the starting position of robots on the map.

Junction
: a position that indicates a split in the road on the map, an intersection.

With these at hand, the CBFS will first be defined for a single robot and then adjusted to work with multiple robots.

The algorithm consists of two steps: the update step and the choice step. The update step fires every time the robot moves and keeps track of the changes in the frontier nodes and build a junction tree (with the root in origin). Every frontier node is assigned to a junction from which it was first found. When a new junction is found, the node that was split to make that junction is considered to be explored, while the nodes obtained from that split are assigned to the newfound junction. Junctions are added to the tree as they are found as children to the junction from which the robot was going when it found a new junction. Figure 3.2 shows an example junction tree (green junctions and red edges) and frontier nodes (blue) assigned to it.

The choice step is called by the robot every time when it needs to choose the next target for exploration. It simply chooses the frontier node assigned to the root junction (origin). Once origin has no more frontier nodes left, it picks one from the first level junctions (root's children in the tree) and keeps doing that until all of

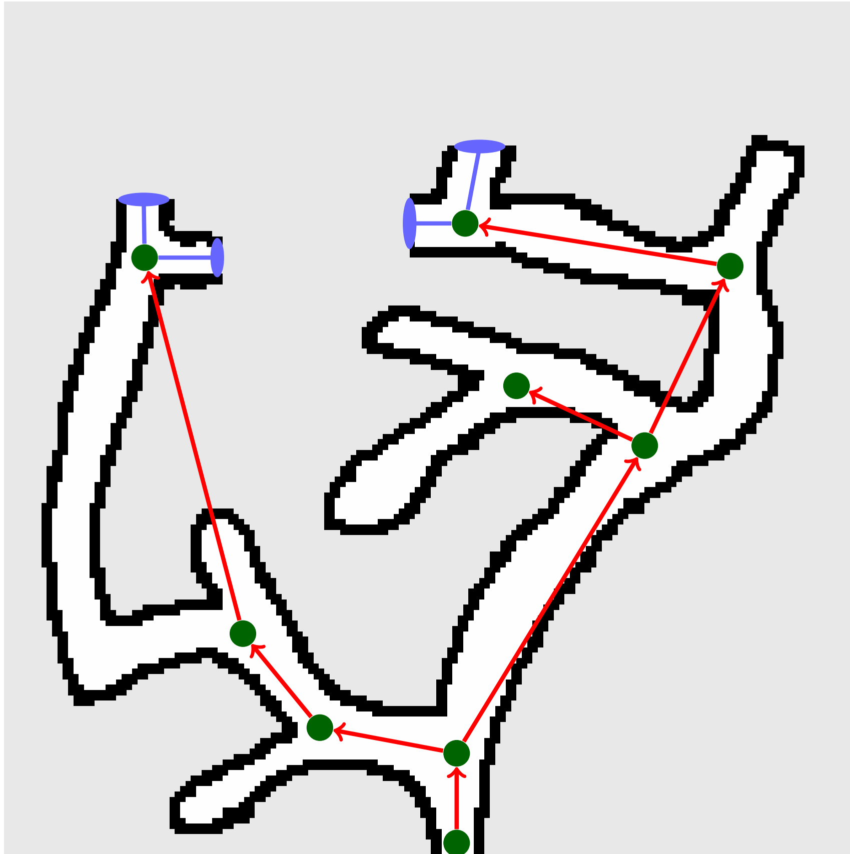


Figure 3.2: An example of junction tree with assigned frontier nodes overlaid on the map

them are exhausted. Once that happens, it goes deeper and exhausts all junctions at the next level. This process is repeated until there are no more frontier nodes left anywhere in the tree, which means that the map is fully explored. In figure 3.2 all frontier nodes belong to the fourth-level junctions and any of them can be chosen at the next choice step. For a single robot however, it is best to choose the node closest to the robot's position.

The multi-agent extension of this algorithm is fairly simple. When two robots enter the communication range, they synchronize their maps, frontier nodes and junction trees and split the frontier nodes belonging to the top-most tree level among themselves. To split the nodes, they are sorted (following the priority system which is explained in detail in section 3.4) and assigned as follows:

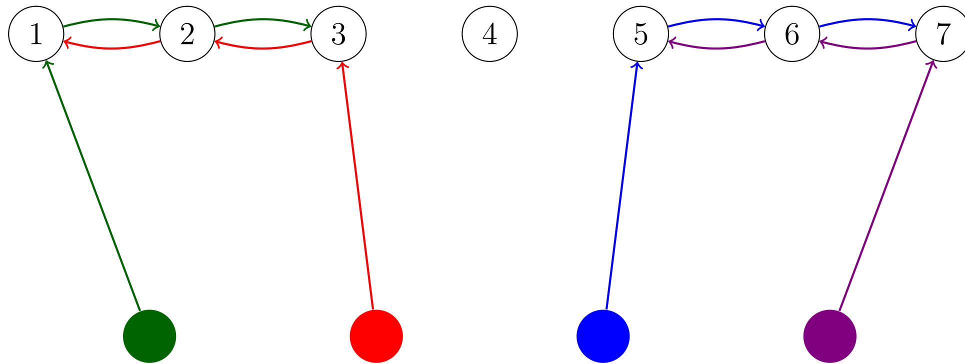


Figure 3.3: The order of traversal of several top-level frontier nodes when there are fewer robots than nodes

- if the number of robots equals the number of nodes, every robot gets one node;
- if there are more robots than nodes, then the remaining robots take the nodes one level lower and split them between themselves;
- if there are more nodes than robots, then the robots try to cover the nodes with equal intervals (e.g. in case there are 4 robots and 7 nodes, the robots will take nodes 1, 3, 5 and 7).

Each robot remembers to what position in the list it was assigned and once it is done exploring his assigned node (either finding a new junction or a dead end), it goes to the next node in the list towards the robot closest to it so that they can make pairs (see figure 3.3). Once any pair of robots meet, they forget the previous directions and perform the split again on the remaining top-level frontier nodes. In figure 3.3 it might seem that the robots don't tend to the node №4, but they will do it as soon as either group meets up.

All of this put together constitutes the basic idea behind CBFS. The analysis section (3.5) will provide some more considerations on why it was designed that way and will try to assess its possible deficiencies.

3.3 Data structures and algorithms

3.3.1 Frontier tracking

Occupancy grid map has no concept of frontier — it is just a bitmap of values. CBFS operates with much higher-level concept of frontier nodes, and for it to work, a module that tracks changes in the occupancy grid and provides access to the frontier nodes is necessary.

The first operation supported by this module is extracting the frontier cell set from the occupancy grid. Recall that a frontier cell is a cell that is known to be passable and is adjacent to one or more unexplored cells. Finding all such cells can be done by simply iterating over the whole occupancy grid and checking every cell to fit this definition. This operation has the cost of $O(h * w)$ where h and w are height and width of the map respectively. Since this update needs to be run every time robot receives new scan info or update from other robots, and the size of the map can get very large, this operation becomes too costly.

Fortunately, the frontier cell set can be updated dynamically as well. Every time a robot receives new scan info or data from other robots, it can check every updated cell if it fits the frontier cell definition, and check the existing frontier cells if they no longer fit it (this happens when their adjacent cells are explored). Scanner range is usually much smaller than map dimensions (otherwise, there wouldn't be any problem at all) and average frontier size is on the order of $O(h+w)$, which is a lot more manageable.

Once the frontier cell set is known, it must be used to produce the frontier node cell set. This is quite simple to do. The frontier cell set can be treated as a graph, where each cell is a vertex and adjacent (on the occupancy grid) cells are joined by an edge. Frontier nodes by definition are connected components on this graph and can be retrieved using a simple depth-first of breadth-first search [29].

These two operations are enough to retrieve frontier nodes, but managing them is much harder. The main difficulty arises from the need to know if a node N_{t+1} at time $t+1$ corresponds to some node N_t at time t. This correspondence is necessary to reassign this new node to the same junction that its predecessor belonged to.

There are several possibilities here. First, N_{t+1} can exactly match (by a set

membership test) some N_t. This happens when the node wasn't updated on this turn and constitutes an obvious match to N_t. Second, a N_{t+1} can partially match some N_t. This can happen when the node was updated partially (which constitutes a match with N_t) or when a node was split in two and a new junction must be made (which must be treated separately). Finally, N_{t+1} can have no overlap with any nodes from time t at all. This happens when the robot travels a narrow corridor and a scanner update takes away every cell in the node in one tick. It can also happen, although rarely, that this node was made as a result of a split and a new junction must be made. Either way, since there is no node from which the parent junction can be retrieved, N_{t+1} must be assigned to the current junction.

All that remains is to address the situation when the split happens. This can be easily detected by comparing the node counts at time t and $t+1$. If this count is increased, then there definitely was a split and a new junction must be made. The previous paragraph assigns both nodes that may come from the split to current junction (or to the parent of their predecessor, which will be current junction at that point). If a new junction is made, this can be used to quickly determine these nodes and assign them to the new junction. The simplest way to do this is to find all nodes belonging to current junction within the scanner range of robot position and assign them to the new junction. Even if some of the nodes that get reassigned that way change their owner, it does not matter, since they are so close to this new junction and might as well be assigned to it.

3.3.2 Junction tree

Since the formation of a new junction is detected during the frontier node set update, junction tree can be implemented as a fairly simple structure with not too much code behind it.

In its essence, the junction tree is just a simple tree of junctions and frontier nodes assigned to these junctions. When a new junction is added, it is assigned as a child to the current junction. Keeping up current junction is not difficult — a new junction is always set as current, and if the robot decides to explore a frontier node that belongs to a junction that is not current, current is set to that junction.

There are several practical issues that arise with the use of junction tree. The

first and the most notable is that current implementation of frontier node set spawns many small frontier nodes (of only 1-3 cells), which cause a lot of problems. These nodes spawn new junctions in places where there shouldn't be any, which increases the tree depth (and as will be shown later, this is very bad for the performance of CBFS). These nodes also count as valid targets for choice step of CBFS, and since they are often located very close to the other, real, nodes, this can make the multi-robot assignment algorithm perform much worse. There is little that can be done with the latter at this point (but it will be addressed later in the priority system), but the former can be fixed.

There are two general ways to solve this problem. One is to simply disregard small nodes when comparing the node counts between current and previous turns. In practice, it worked rather well with a threshold of 10 cells for the small node, but it is possible to miss legitimate nodes this way sometimes. Another approach is to track the unpassable regions between the nodes and consider (for the purpose of counting) the nodes that are disconnected by a small region to be just one node. This will cause the splits to be detected later (since just one unpassable cell will not make a new junction anymore), but will not miss any legitimate split. A combination of these two methods almost completely eliminates the problem of extra junctions.

Another consideration that can be implemented is trimming the junction tree. Trimming in this context means removing junctions that have no more frontier nodes assigned to them. This can provide two improvements — the robots using CBFS will run back and forth a bit less (in the rare cases when the part of the next level gets dumped into current level) and merging junction trees during synchronization becomes much easier. There is no concrete proof that this will never cause the robots to meet, so it can be considered a dangerous feature, but in the experiments that has never happened.

There is one more issue that prevents the robot from detecting large road splits properly. A large split in this case is a junction where a single road splits into 3 or more new roads. Large halls can be classified as such splits as well. The problem is that the robot detects several smaller (1 into 2) splits and places several junctions where there should be one. This is partially mitigated by tracking unpassable regions between nodes, but it solves the problem only in few smaller cases. A

much better solution is to merge junctions that are located close to each other, with everything assigned to the child junction being reassigned to the parent. The coordinates of the merged junction can either be inherited from the parent (as implemented in this project) or recalculated to the middle of the path between them.

3.3.3 Synchronization

Synchronization between robots happens when they stay inside each other's communication range. There are five steps to the synchronization process that must be performed in order:

1. Occupancy grid maps must be merged;
2. Frontier node sets must be merged or rebuilt;
3. Junction trees must be merged;
4. Frontier nodes must be reassigned to the merged tree;
5. Robots must agree on their next destinations;

Occupancy grid map merging is a well developed topic in modern robotics and was previously discussed in section 2.3. The other four parts of this process are unique to CBFS and need to be explored in detail.

As it was shown in section 3.3.1, the easiest way to merge frontier node sets is by rebuilding them from scratch using the updated occupancy grid map. Since synchronization does not happen very often (compared to performing sensor scans), and merging occupancy grids is at least $O(h * w)$ anyway, rebuilding the frontier does not have any significant effect on the running time and is much easier to perform.

Merging junction trees may seem somewhat tricky, but ultimately it is not very difficult. They possess a very useful property — even though different robots may mark same road splits as different junctions, the coordinates of these junctions will always be nearby. This makes merging fairly simple — junctions with exactly matching coordinates are merged together, while everything else is simply dumped

into the new tree while preserving parent-child relationships. Once this is done, junctions in close proximity are merged together (as described in section 3.3.2, except that they will often be on the same depth in the tree rather than have a parent-child relationship).

The next step is to reassign frontier nodes to the merged junction tree. This is done similarly to the update process described in section 3.3.2, but the nodes N^r_{t+1} must be compared with the own nodes of the robot N^r_t as well as the nodes of the other robot $N^{r'}_t$. This adjustment is simple — match with the node in the either list is good enough to determine the new result. The situation when there is no match at all is impossible, because for the node to exist it must have existed before in one of the robots' maps. Partial matches with both robots are extremely rare, but can be treated like matches with either of them without any further problems.

Once all of this is done, the robots must agree on their next destinations. The general idea how this is done was described earlier, and more details will be explained in section 3.4.

One final thing needs to be said about the unproductive synchronizations. They happen when the robots don't have anything new to share with each other — for example, when they meet each other in the middle of explored terrain and then spend some time traveling together. An event like this can (and should) be detected at the first stage of occupancy grid map synchronization, since it acts as a flag to abort synchronization at this point of time with this robot. This serves two purposes — firstly, it obviously saves some computation time which otherwise would have been wasted on useless merges; and secondly, it prevents the robots from reassigning targets between each other at every point of time. The experiments showed that without this improvement the robots can get stuck going back and forth in the same place because they switch targets when their relative position changes.

3.4 Algorithm details

It was stated earlier that CBFS selects the frontier nodes assigned to the highest available level in the junction tree, sorts them and then assigns robots to explore these nodes. This is the main idea of CBFS, but the underlying sorting and

selection process is in fact more complicated.

There are several reasons why this can't be done directly. First of all, it was mentioned earlier in section 3.3.2 that the current implementation of frontier node set tracking spawns a lot of very small (consisting of only 1-3 cells) nodes. While these nodes don't spawn junctions, they are still considered to be valid targets for the CBFS algorithm.

Why is this bad? Consider that several robots are assigning their next destinations and have to choose from 2 'real' nodes and many small nodes. If all of the small nodes are concentrated near one of the 'real' nodes, it will cause all robots except one to travel in that direction, which is far from optimal. What is even worse, these small nodes are not persistent — they often appear after one scanner tick and become fully explored right on the next one. It is simply not viable to target them specifically, since a vast majority of them will disappear while the robots explore the 'real' nodes.

However, such nodes can't be ignored entirely. Even though most of them get explored automatically, there is no guarantee that *all* of them will be gone by the time the current level of the tree is explored. If some small node was missed during exploration of the 'real' nodes, it is not very likely to be accidentally explored later when dealing with a nodes of the next level. The reason for this is that while exploring the current level, robots often traverse the tree branches of that level, but when dealing with the next level, they will often utilize shortcuts on the map that are not present on the junction tree (like loops).

To cope with this a priority system was developed. When the robots are assigning their next destinations, they have several several ranks of possible destinations, for each of which (if it is used) the CBFS choice step is applied. These ranks are as follows:

1. large nodes on the highest level of the junction tree;
2. if previous rank is empty, small nodes on the highest level of the junction tree;
3. large nodes on the second highest level of the junction tree;
4. everything else.

With this system small nodes on the current level are skipped completely if there are large nodes still remaining there.

Within every rank, the nodes are sorted by the distance from the robots with the exception of the last rank, which is first sorted by the node depth in the tree and then by distance. If there are more robots than nodes in the current rank, remaining robots explore the next rank. If there are more robots than nodes in total, remaining robots just stick together with some of the robots assigned to the first rank.

This system however is only good for assigning targets when there are multiple robots deciding how to split the targets between themselves. When a robot is exploring some path alone, he can achieve better results by following a slightly different pattern:

1. close proximity large node on the second-highest level of the junction tree;
2. close proximity small node on the second-highest level of the junction tree;
3. close proximity large node on the highest level of the junction tree;
4. close proximity small node on the highest level of the junction tree;
5. next node as was discussed with the other robots (see figure 3.3);
6. everything else.

The first two ranks in this system are called "follow-through" and may seem completely against the idea of CBFS. The idea with them is that the robot is only allowed to follow them for a very limited distance from the junction (typically 1-2 scanner ranges, not more). This is done to eliminate very short dead-ends, which may cause significant back-tracking later, but can be eliminated now at very little cost. If no such dead-end is found, the robot simply returns and proceeds with the regular CBFS routine. This improvement is purely practical and is not included in the analysis.

The other ranks are self-explanatory and strictly follow the CBFS idea. They make every robot prioritize large nodes over small as it explores its assigned branch and once it is done make it meet up with other robots. If the robot fails to meet up, it will just proceed with exploring as if it was alone.

3.5 Analysis

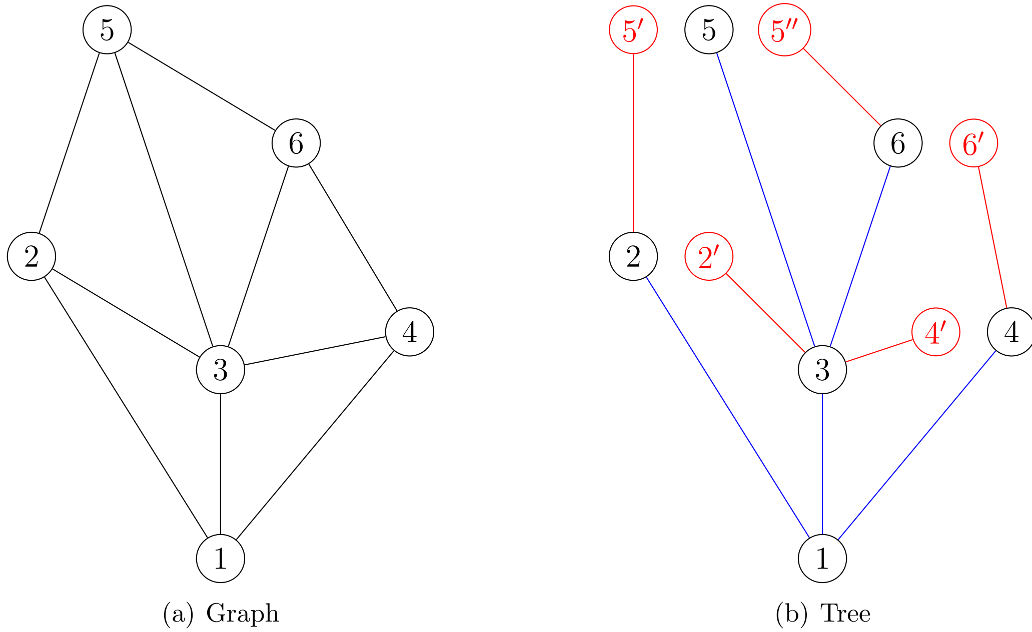
3.5.1 Graph and tree correspondence

The CBFS algorithm runs on an occupancy grid, but makes most of its decisions using the junction tree. The process of building the tree was described in section 3.3.2, but in order to compare CBFS with the naive approach, it is necessary to understand how this tree is related to the graph used in the naive algorithm analysis (section 2.5.3). The correspondence between the features on the map and the elements on the naive algorithm graph and CBFS junction tree is shown in table 3.1.

Map feature	Naive graph element	CBFS junction tree element
Road junctions	Every junction is a vertex	Every junction is a junction node
Dead-ends	Every dead-end is a vertex	Every dead-end is a frontier node
Frontier	Not reflected specifically	Frontier is represented as a set of frontier nodes
Road stretches	Every road stretch between two junctions is an edge	Road stretch that connects a junction with a dead-end is an edge between a junction node and a frontier node. Road stretch that connects two junctions that are not yet connected by any path is an edge between two junction nodes. Road stretch that connects two junctions that are already connected by a path is an edge between a junction node and a frontier node.

Table 3.1: Correspondence between map features and elements of the naive algorithm graph and CBFS junction tree

Another important distinction is that the naive algorithm graph is built strictly



(a) Graph (b) Tree

Figure 3.4: Correspondence between the naive algorithm graph and the complete junction tree

incrementally — no vertices or edges are ever deleted, while the implementation of the junction tree actively prunes them to store only currently relevant data, discarding the exhausted nodes. For the analysis the complete version of tree is of interest — a junction tree where the junctions are not pruned and frontier nodes remain as leaves even after they are explored. While a tree like this is not obtained by the process described in section 3.3.2, it shares an important property with it: if CBFS possessed this tree instead of its incrementally built one, the traversal order would have remained the same. Therefore, such tree can be used to analyze the behavior of CBFS. The intuitive difference between the naive algorithm's graph and such complete tree is shown in the figure 3.4.

More formally, the tree can be obtained from the graph in the following way:

- the root of the tree is the origin vertex;
- the graph is explored using breadth-first search;
- as new nodes are found, they are added to the tree as children to the nodes from which they were found;

- if an already known node is found, a new leaf node is created (the "prime" nodes on figure 3.4) as a child to the node from which it was reached;
- when continuing search from already known node, the initially reached version of it (without "prime") is used.

Of course, depending on the breadth-first search traversal order, different trees are possible. However, since CBFS algorithm is deterministic, and since every edge will be traversed when following any tree, the particular choice of tree does not matter for the analysis.

The naive algorithm graph is always connected, which means that if it has $|V_g|$ vertices, the number of its edges is between $O(|V_g|)$ and $O(|V_g|^2)$. The complete junction tree contains exactly the same number of edges $|E_t| = |E_g|$, but more vertices $|V_t| = |E_t| + 1$.

With this, the conversion and metric comparison is given between the graph that was used to analyze naive algorithm and the junction tree of CBFS.

3.5.2 Running time complexity

Running time complexity of a typical graph-based breadth-first search algorithm is known to be $O(|E| + |V|) = O(|E|)$ (for a tree, which is a connected graph). This holds true for a single-robot CBFS algorithm as well.

Let us define tree depth as the average depth of all frontier nodes in the tree and tree breadth as the average branching factor of all junction nodes:

$$d = \frac{\sum\limits_{v_i \in F} dist(v_i, v_0)}{|F|} \qquad b = \frac{\sum\limits_{v_i \in J} children(v_i)}{|J|} \tag{3.1}$$

where J is the set of junction nodes, F is the set of frontier nodes ($V = J + F$) and v_0 is origin.

Consider the tree on figure 3.5. The blue path indicates a possible breadth first search traversal order (forward only to avoid cluttering), while every edge is marked with how many times it was traversed while going forward. This forward constraint is included for simplicity — it simply hides the factor of 2. This factor

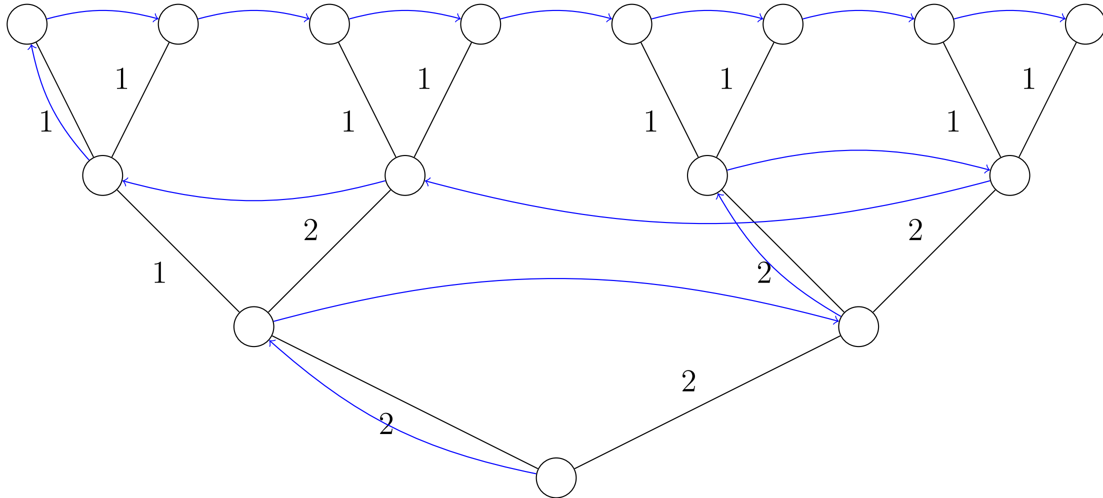


Figure 3.5: CBFS edge forward traversal count distribution

comes from the fact that for every time robot traverses an edge forward, it will have to traverse it backward at some point later.

What is the total number of edge traversals in terms of b and d? Every edge going to a terminal node is traversed once, and there are b^d such edges. Every edge on the subterminal level is traversed twice, except for one of them, which was explored last. There are b^{d-1} such edges, so the total number of forward traversals for this level is $2b^{d-1} - 1$. For every next level, the traversal count per edge is increased by one, while the total amount of traversals is decreased by one — the algorithm "saves" one edge for every level ahead of it, because it starts traversing the next level without the need of going back. The complete formula is:

$$t = b^d - 0 + 2b^{d-1} - 1 + ... + db - d + 1 = \sum_{i=1}^{d} (ib^{d-i+1} - i + 1) \tag{3.2}$$

Computing the sum in this equation yields $O(b^d)$ with a low degree polynomial of b and d hidden inside the O-notation. This value is the same as the order of edges in the tree with depth d and branching factor b, which means that $t = O(|E|)$. For the worst case of a tree that was made from a complete graph, this results in $O(|V_g|^2)$. However, this worst case is not possible in practice. The occupancy grid map can't be reduced to a complete junction graph (except for small trivial cases),

because this graph must be planar. For a planar graph, the maximum number of edges is linear in the number of vertices, so even for the worst case running time complexity remains $t = O(|E|) = O(|V_g|)$, usually with some coefficient hidden in O-notation.

3.5.3 Scaling with multiple robots

The scaling idea of the CBFS algorithm is based on two major points:

- robots distribute the workload by exploring different branches;
- on their way back the robots will meet, sharing their findings.

The first point was already explained in detail earlier. The second point follows from the fact, that while exploring the current branch, robots move through the same trajectories (since they use optimal paths and their knowledge of the map is the same at the moment of synchronization) and will always meet each other while changing branches.

Earlier in section 3.1 it was stated that the only maps that are of interest for the purpose of scaling are those with a sufficiently large branching factor. The reason for this is quite obvious — if the map consists of a single straight corridor, any algorithm will not scale. If the branching factor is equal to or greater than the number of robots, then robots will scale well by choosing different branches. The scaling will be perfect (meaning equally distributed workload) if all road segments are equal, and will be somewhat worse (but still linear) if the branches are different in size, because the robot who is finished earlier will have to follow another robot for a while until they meet.

This does not happen very often however — a large map may have a few extremely imbalanced branches on the same level, but they can be expected to be approximately equal on average. Maps where all or most of junctions have extremely imbalanced branches are statistically infeasible and unrealistic (in terms of real-world maps).

Another thing that needs a special note is a map with sufficiently large depth value and branching factor somewhat less than the robot count. CBFS can handle situations like this fairly well by sending some of the robots further into labyrinth

than the current branch level. These robots may miss some of the previous branch robots when they reach their level, doing some redundant exploration, but will successfully meet with them eventually.

4

Computer simulation

COMPUTER simulation has served several purposes in this project. It was used to confirm the weakness of the naive approach and to detect the exact shape of the map that makes this weakness evident. Other than that it was run on various maps to provide experimental data which would support the analysis performed in sections 2.5.3 and 3.5. This section describes various details related to the simulation, as well as implementation-specific on the two implemented algorithms — CBFS and naive approach.

4.1 Development platform

The simulation is implemented using C# 4.0 programming language and Microsoft.NET 4.0 framework. There are several notable advantages to this combination:

1. Microsoft.NET provides an extensive library of basic algorithms and data structures (like various collections);
2. C# and .NET allow the use of LINQ technology to greatly simplify interaction with collections, saving time and increasing the readability of code;
3. Microsoft.NET includes an easy to use, but very flexible and powerful graphical user interface development tool — Windows Presentation Foundation (WPF);

4. Microsoft Visual Studio 2010 provides unparalleled ease of development and debugging for C# projects;

5. C# and .NET have a detailed documentation as well as support of a large community, which makes solving any problems that arise during the development easier.

All of these points together allow the programmer to focus on implementing the algorithms rather than on technical details, which makes C# and Microsoft.NET a good choice for this project. No other external libraries were used.

4.2 Model of the environment

The model of the environment consists of two major components — the simulation engine and the robot.

The component diagram of the model is shown in figure 4.1. The robot component is a direct implementation of all relevant subsystems of the robot. The control block is the set of the main organizational routines of the robot and determines the order in which other components are used. It uses the scanner to get the surrounding terrain data from the simulation engine and the communication module to synchronize with any nearby robots. The updates from the sensor are stored in the map, which is later used by the exploration module to perform all specific analysis goals of CBFS and by the navigation module to find the shortest path with A^* algorithm. The exploration module is called to determine the next target, and it also directly calls the navigation module once the move direction for the current turn is established.

The simulation engine acts both as a coordinator of the model and as an environment (or the world) in which the robots exist. It uses a simple Δt approach which prompts every robot to take one action at every moment of time. The simulation is started from the user interface and ends after every robot has explored every reachable point on the map.

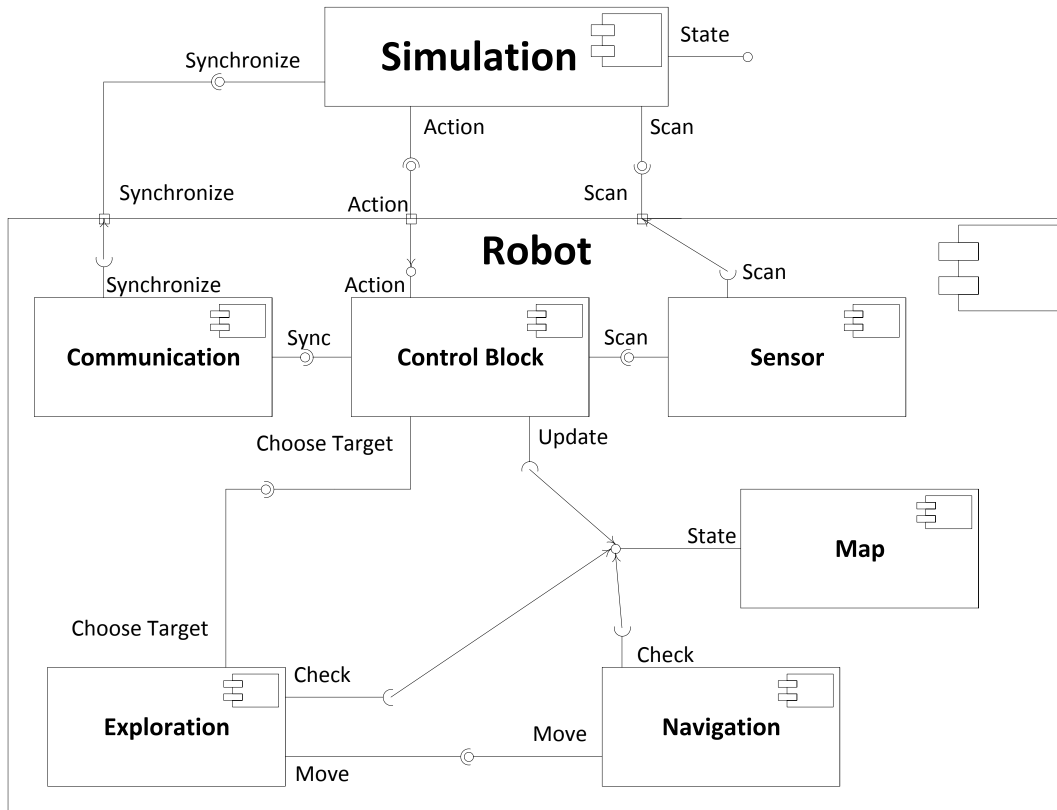


Figure 4.1: UML component diagram for the computer simulation

4.3 Assumptions

Every model is a simplification of the real world object. In this project, a number of simplifications were made to keep the model simple while preserving its integrity. Also, goal of this model is not to reproduce the world as accurately as possible, but to experimentally verify the results obtained analytically for the naive approach and the CBFS. The assumptions made in the model include:

1. The robots possess perfect sensors. The reasoning for this was already explained in detail in chapter 2. In short, the random noise in the sensors is very important for the SLAM and map correspondence algorithms, but since they are not the focus of this project, they are being used as a black box and thus don't need to have real-world input. Perfect sensors mean that every

scan that robots perform returns the true state of the world without error.

2. The robots possess perfect actuators. This point is similar to previous one and was also explained earlier. Perfect actuators allow robots to be certain that after issuing a movement command it will find itself exactly where it expected to be.

3. The world is strictly 2-dimensional with flat maps. This means that there is no height constraint used for any purpose in this project, and that the routes on the map may not pass above each other. This may give the robots an unfair opportunity to predict the end of the tunnel when they can't see it, but they are not allowed to do it. Other than that, such a map is consistent with the real world examples.

4. The sensor is approximated as a Manhattan circle ($|x|+|y| = r$) with a given radius with a robot in its centre. It is possible to think of it as of a laser range finder with a limited range and which is pointed in every direction at every moment of time. Ultimately, the shape of the sensor does not matter, as the algorithm works with anything, so this point does not affect the integrity of the model.

5. The sensor shoots discrete lines (using Bresenham's line algorithm [30]) from the centre to the edges of the sensor range, performing a hit test. If an obstacle (unpassable terrain cell) is found, the line does not proceed further. All points hit by the lines are included in the scan info, every other cell is reported as unknown. The problem with this approach is that Bresenham's line algorithm does not hit every cell inside a Manhattan circle for every possible range, leaving some of them out sometimes. This has not proved to cause any noticeable effect however and essentially just changes the shape of the sensor slightly.

6. The CBFS algorithm often uses a distance metric to compare various lengths. The true distance between two points on the map can only be calculated by the search algorithm (like A^*), which is a computationally costly procedure, especially for the long routes. There are ways to avoid recalculating same

or similar paths many times, but this model instead substitutes this metric by a Manhattan distance metric in some cases. This can cause noticeable problems once in a while, but since both CBFS and naive approach have to use this metric, they even out.

7. The discretization of the map and the A^* algorithm used to traverse it do not produce perfectly realistic paths. This does not affect the algorithms in any noticeable way.

8. The communication range suffers from the same issue as the sensor range. The algorithms can work with any communication range however, so it is sufficient for testing.

With all of this, it can be said that the integrity of the model is preserved given its goal.

4.4 Data structures implementation

Having to simulate several robots performing their search at the same time, this computer simulation is a computationally intensive piece of software. The choice of classes which will represent the abstract data structures required by the robots can greatly reduce the burden of computation. This section explains this choice.

Occupancy grid map. There are two ways to handle the map. In the real-world example it must be treated as a dynamically increasing two-dimensional array that bounds all explored cells. In this simulation however, the size of the map is known initially and therefore the map can be made static. In that case, the map does not need to support the operations of adding or removing new items, and must support access by coordinate. The optimal data structure for this is a simple two-dimensional array (class `List<T>`) with $O(1)$ access.

A^* **open set.** This is a set which contains the frontier of A^* search. It frequently adds nodes and frequently needs to find (and remove) the best node matching a certain condition. A doubly linked list (class `LinkedList<T>`) provides $O(1)$

addition and removal (given the position of the node in the list is known), but finding the element will require a full $O(n)$ search. To speedup the search, the list can be kept sorted. This makes addition $O(\log n)$ using binary search and retrieval $O(1)$. Unfortunately, the sorted list class provided by .NET does not support multiple entries with equal sort metric, which is required here, so instead a wrapper around `LinkedList<T>` was written.

A^* closed set. This is a set which contains exhausted cells in A^* search. It needs to support adding items and checking if an item was already added. Both of these can be done in $O(1)$ by a dictionary (class `Dictionary<K, T>`). Dictionary requires a key that comes together with every entry, which can be computed as a simple hash based on cell's coordinates ($x + 10000 * y$ was used and will not cause collisions for any map with width less than 10000). Since the cell will never be added twice there, there are no further problems.

Frontier nodes. A single frontier node is a set of frontier cells. It needs to support fast addition and membership test operations, for which dictionary is perfect as was shown earlier. All other operations (finding central cell and checking intersection with other node) require processing every single node in the dictionary and can be done either in $O(n)$ or by caching using any data structure. Therefore, dictionary has no disadvantages for frontier nodes.

Entire frontier. The frontier itself consists of a set of its nodes. Since there are typically few nodes (less than 20 most of the time), a simple dynamic array can be used for them without any issue. However, it is also necessary to keep track of all frontier cells (which are a merge of all nodes). These cells are best kept separately at the cost of some extra memory. Using dictionary, this allows for rapid ($O(1)$) membership tests, additions and removals.

Junction tree. The tree itself is stored as a classic tree with every junction having a list of children and a parent as a links. On top of that however, for merging it is necessary to have an access to all junctions in a linear way. Since junctions are essentially coordinate pairs, dictionary works great here as well. One

last thing about the tree is storing links to frontier nodes. Rather than storing them as links in the tree, it is better to store the links to parent junctions at the nodes themselves. The reason for this is simple — nodes are frequently remade from scratch, so it is better to not involve junction tree in this update process at all. Since there are rather few nodes, finding a node that belongs to a particular junction in $O(n)$ is not very troublesome, especially since this operation is done not very often.

4.5 Interface

The computer simulation includes a simple GUI which allows monitoring robot progress as well as customization of several settings. There are three window types in the GUI: settings window (figure 4.2), global view window (figure 4.3) and single robot monitor windows (figure 4.4), one for each robot.

The settings window allows the user to configure the simulation before it is started. First, the user must choose a map. The map must be in a standard graphical format (.png, .bmp and .gif are accepted among others), must be black and white only (with black corresponding to unpassable terrain and white corresponding to passable terrain) and must have some passable terrain on one of the

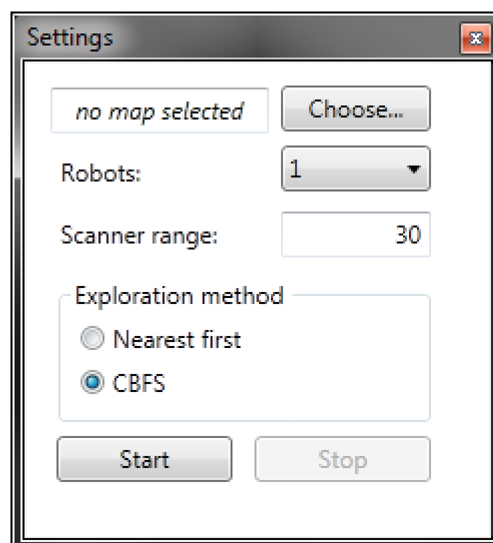


Figure 4.2: Settings window

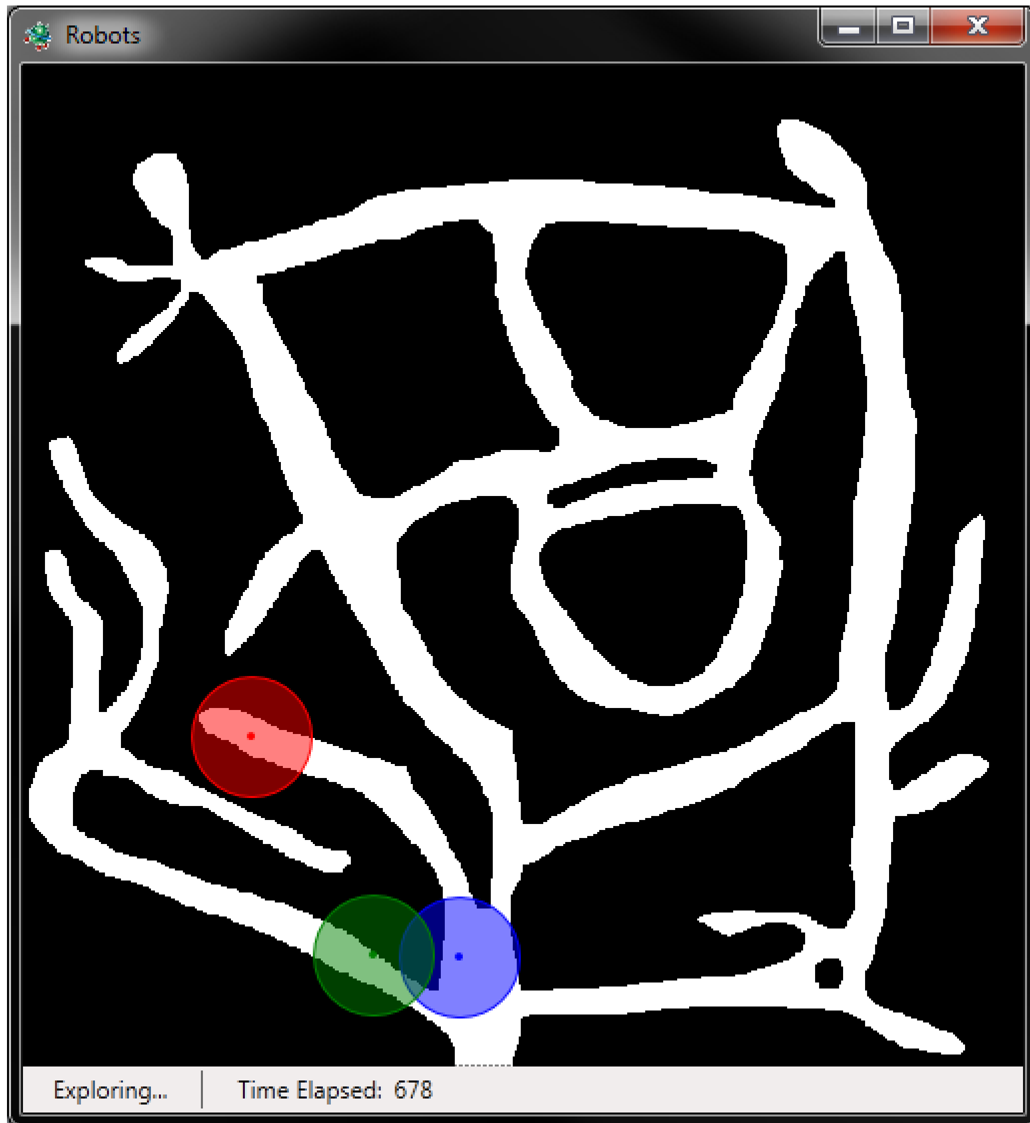


Figure 4.3: Global view window

edges (for the starting point, which will be detected automatically). The user may also specify the amount of robots (1 to 4), the scanner range (radius in pixels) and choose the exploration planning method (between naive and CBFS). Once the simulation is started, it may be paused or stopped using the settings window.

The global view window shows the whole map (revealed) and the robots' positions on it. It also reports the current status of simulation and time elapsed since started on the status bar in the bottom of the window.

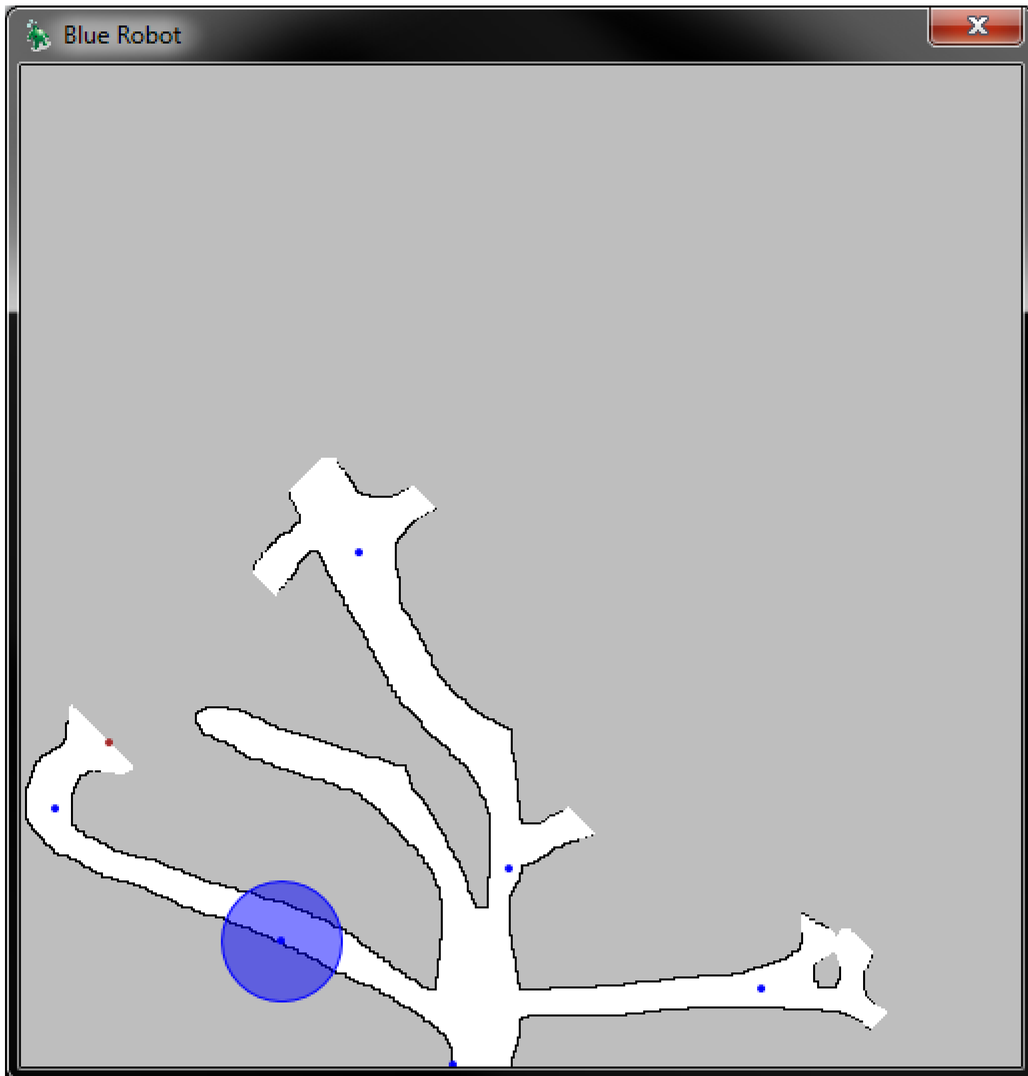


Figure 4.4: Single robot monitor window

The single robot monitor windows appear after the simulation is started. This window shows the map as it is known by a robot. It also reports the robot's position on it, its current target (brown point) and all junctions in its junction tree (blue points).

5

Discussion

The purpose of this project was to study the problem of exploration planning in multi-robot systems and to improve the performance of currently used solutions. The typical solution for this problem employs the naive approach, which performs well on average, but has poor worst case running time and does not scale in many realistic situations. To counter these problems a new method called *coordinated breadth first search* was developed. The comparison of running time and scaling factor for these two methods is shown in the table 5.1.

Parameter	Naive approach	Coordinated breadth first search
Best case running time	$O(\|V\|)$	$O(\|V\|)$
Worst case running time	$O(\|V\|^{3/2})$	$O(\|V\|)$
Best case scaling	$O(1/n)$	$O(1/n)$
Worst case scaling	1	$O(1/n)$

Table 5.1: Performance comparison results

CBFS provides consistent results by design and guarantees scaling and linear exploration time. It is however not perfect. The constant hidden in the O-notation for running time is heavily dependent on the map (more precisely, the distances between vertices on the junction graph) and for most cases is worse for CBFS. Likewise, the constant hidden in the O-notation for scaling factor is also dependent

on the map. In practice it means for many (although far from all) maps naive approach performs better than CBFS. The exact nature of these dependencies is out of scope of this project.

Apart from these analysis results, this project has done an attempt to formalize and study the problem of multi-robot exploration planning, which is heavily overlooked by the scientific community. The results (both the theory and the developed computer simulation) can be used to further study this problem and to develop better solutions.

While CBFS in its current state does not provide decisive increase in performance over the naive approach, it has shown that the deficiencies of this approach can be overcome. CBFS can be further improved to implement features from other methods (like an utility function similar to Burgard's in place of the priority system) and extended to reduce the overhead travel of breadth-first search by using junction graph instead of tree. Finally, if a complete statistical analysis of possible maps was performed, it could be used to further analyze the performance of both CBFS and the naive approach, and its conclusions could be used to greatly improve either method.

Formally, in this project the following goals were achieved:

- two existing approaches to exploration planning problem were reviewed;
- the naive approach was analyzed and its weakness was identified;
- a new method to cope with this problem was developed and analyzed;
- a computer simulation software was developed to test these methods;
- the new method was compared with the naive method and conclusions were drawn;
- further ways to improve the developed method were suggested.

Bibliography

[1] W. Burgard, A. Cremers, D. Fox, D. Hähnel, G. Lakemeyer, D. Schulz, W. Steiner, S. Thrun, The interactive museum tour-guide robot, Proceedings of the National Conference on Artificial Intelligence (1998) 11–18.

[2] I. Ulrich, F. Mondada, J. Nicoud, Autonomous vacuum cleaner, Robotics and autonomous systems 19 (3-4) (1997) 233–245.

[3] D. Ferguson, A. Morris, D. Hähnel, C. Baker, Z. Omohundro, C. Reverte, S. Thayer, C. Whittaker, W. Whittaker, W. Burgard, S. Thrun, An autonomous robotic system for mapping abandoned mines, Advances in Neural Information Processing Systems (NIPS 03).

[4] D. Apostolopoulos, L. Pedersen, B. Shamah, K. Shillcutt, M. Wagner, W. Whittaker, Robotic antarctic meteorite search: Outcomes, Proceedings of IEEE International Conference on Robotics and Automation 4 (2001) 4174–4179.

[5] A. Bennett, J. Leonard, A behavior-based approach to adaptive feature detection and following with autonomous underwater vehicles, IEEE Journal of Oceanic Engineering 25 (2) (2000) 213–226.

[6] H. Durrant-Whyte, T. Bailey, Simultaneous localization and mapping: part I, Robotics & Automation Magazine, IEEE 13 (2) (2006) 99–110.

[7] W. Burgard, M. Moors, D. Fox, R. Simmons, S. Thrun, Collaborative multi-

robot exploration, Proceedings of IEEE International Conference on Robotics and Automation, 2000 1 (2000) 476–481 vol. 1.

[8] S. Thrun, A probabilistic on-line mapping algorithm for teams of mobile robots, The International Journal of Robotics Research 20 (5) (2001) 335–363.

[9] X. Zhou, S. Roumeliotis, Multi-robot SLAM with unknown initial correspondence: The robot rendezvous case, International Conference on Intelligent Robots and Systems, 2006 IEEE/RSJ (2006) 1785–1792.

[10] W. Burgard, M. Moors, C. Stachniss, F. Schneider, Coordinated multi-robot exploration, IEEE Transactions on Robotics 21 (3) (2005) 376–386.

[11] S. Thrun, Y. Liu, Multi-robot SLAM with sparse extended information filers, Robotics Research (2005) 254–266.

[12] A. Elfes, Using occupancy grids for mobile robot perception and navigation, Computer 22 (6) (1989) 46–57.

[13] A. Diosi, G. Taylor, L. Kleeman, Interactive SLAM using laser and advanced sonar, Proceedings of the 2005 IEEE International Conference on Robotics and Automation (2005) 1103–1108.

[14] F. Bourgault, A. Makarenko, S. Williams, B. Grocholsky, H. Durrant-Whyte, Information based adaptive robotic exploration, Intelligent Robots and Systems, 2002. IEEE/RSJ International Conference on 1 (2002) 540–545 vol. 1.

[15] S. Thrun, Learning occupancy grid maps with forward sensor models, Autonomous robots 15 (2) (2003) 111–127.

[16] M. Montermerlo, FastSLAM: a factored solution to the simultaneous localization and mapping problem with unknown data association, Ph.D. thesis, Carnegie Mellon University (2003).

[17] R. Smith, P. Cheeseman, On the representation and estimation of spatial uncertainty, The International Journal of Robotics Research 5 (4) (1986) 56–68.

[18] S. Thrun, Y. Liu, D. Koller, A. Ng, Z. Ghahramani, H. Durrant-Whyte, Simultaneous localization and mapping with sparse extended information filters, The International Journal of Robotics Research 23 (7-8) (2004) 693–716.

[19] K. Konolige, D. Fox, B. Limketkai, J. Ko, B. Stewart, Map merging for distributed robot navigation, Proceedings of 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems 1 (2003) 212–217 vol. 1.

[20] A. Birk, S. Carpin, Merging occupancy grid maps from multiple robots, Proceedings of the IEEE 94 (7) (2006) 1384–1397.

[21] S. J. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition, Pearson College Division, 2010.

[22] D. Dolgov, S. Thrun, M. Montemerlo, J. Diebel, Practical search techniques in path planning for autonomous driving, Ann Arbor 1001 (2008) 48105.

[23] B. Yamauchi, Frontier-based exploration using multiple robots, Proceedings of the second international conference on Autonomous agents (1998) 47–53.

[24] S. Koenig, C. Tovey, W. Halliburton, Greedy mapping of terrain, Proceedings of 2001 IEEE International Conference on Robots and Automation 4 (2001) 3594–3599 vol. 4.

[25] I. Rekleitis, G. Dudek, E. Milios, Multi-robot exploration of an unknown environment, efficiently reducing the odometry error, International Joint Conference on Artificial Intelligence 15 (1997) 1340–1345.

[26] R. Kurazume, S. Nagata, S. Hirose, Cooperative positioning with multiple robots, Proceedings of IEEE International Conference on Robotics and Automation (1994) 1250–1257 vol. 2.

[27] N. Roy, G. Dudek, Collaborative robot exploration and rendezvous: Algorithms, performance bounds and observations, Autonomous robots 11 (2) (2001) 117–136.

[28] E. Anderson, R. Weber, The rendezvous problem on discrete locations, Journal of Applied Probability (1990) 839–851.

[29] J. Hopcroft, R. Tarjan, Algorithm 447: efficient algorithms for graph manipulation, Commun. ACM 16 (6) (1973) 372–378.

[30] E. Angel, Interactive Computer Graphics, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.