Institutionen för datavetenskap
Department of Computer and Information Science

Final thesis

Resource allocation of drones flown in a simulated environment

by

Anders Wikström

LIU-IDA/LITH-EX-G--14/003—SE

2014-03-07

Linköpings universitet
SE-581 83 Linköping, Sweden

Linköpings universitet
581 83 Linköping

Linköping University
Department of Computer and Information Science

Final Thesis

Resource allocation of drones flown in a simulated environment

by

Anders Wikström

LIU-IDA/LITH-EX-G--14/003—SE

2014-03-07

Supervisor: Alexander Kleiner
Examiner: Alexander Kleiner

Abstract

In this report we compare three different assignment algorithms in how they can be used to assign a set of drones to get to a set of goal locations in an as resource efficient way as possible. An experiment is set up to compare how these algorithms perform in a somewhat realistic simulated environment. The Robot Operating system (ROS) is used to create the experimental environment. We found that by introducing a threshold for the Hungarian algorithm we could reduce the total time it takes to complete the problem while only sightly increasing total distance traversed by the drones.

Contents

List of Figures

1. Introduction

Something likely to become more common in the future are computer controlled systems assisting with such operations like distributing aid in the case of natural disasters, finding missing people or survey an area for damage. In scenarios like natural disasters you might first have to locate areas that have been damaged or find people that are in need of help and send in aid.

You might have limited resources. These resources can be be physical goods such as fuel and food, but also time. If you can not distribute the aid fast enough, people might get desperate and cause secondary damage like looting, or in case of distributing medicine, disease might spread further. It then becomes a trade off between spending your fuel as efficiently as possible and reaching everyone quickly.

As an example, you could imagine a case where you have a number of hospitals, each with a limited number of ambulances (or helicopters), and a number of patients with time critical conditions. In this case, you would prioritise reaching each patient as soon as possible, and while reducing the fuel cost desired, it is not critical.

This report evaluates three different assignment algorithms for efficient resource allocation when planing movement of a group of agents, e.g. ambulances, rescue helicopters, flying drones or human rescue personnel. The first two algorithms are previously existing ones while the third is a to my knowledge new combination of two algorithms. The algorithms are tested in a simulated environment with randomized goals and starting locations, with varied number of agents and several different environments. The goal of this work is to see if we can find an algorithm that can reduce both the time taken to complete the overall goal and the resources needed to reach the goal locations.

In an attempt to bridge the theoretical aspects of these assignment algorithms to a real world practical result the experiment is conducted in an as realistic environment as possible. While running the experiment with real drones would be preferred it wouldn't be practical. To get statistically significant results while testing the assignment algorithms a lot of runs of the experiment are required. Thus a simulated environment is used in the experiment can be automatically run as many time needed.

For the experimental evaluation the Robot Operating System (ROS) [5]

is used. ROS offers a unified platform for creating multi-agent control systems suitable for the scenarios at hand. It has a simulator that can simulate close to real world drones and various packages with control interfaces to the drones that are similar to a real drones, like the AR.Drone 2.0 Parrot shown in fig 1.1.

Figure 1.1: Two AR.Drone 2.0 Parrot with different protective shells.

1.1 Problem Description

The fundamental problem studied in this report is an assignment problem. An assignment problem is about assigning a set of tasks to agents based on a cost for each agent to complete each task. The goal of assignment problems is usually to minimize the total cost for all the agents to complete their task. But if you are interested in minimizing the cost of the agent with the highest cost the problem falls in the category of bottleneck assignment problems.

If the cost associated with each task is independent of each other the problem is considered linear. Other types of assignment problems are the generalized assignment problem and quadric assignment problem involving more complex relations between the assignments and the costs of completing the tasks.

In this work we have a mix of both the linear assignment problem and the linear bottleneck assignment problem since we are interested in minimizing both the total cost and the maximum cost.

For a small number of agents it would be possible to solve assignment problems by just trying all combination and pick the one with total cost. But this soon becomes infeasible because of the way the number of combinations

scale with the number of agents. The number of possible combinations are $n!$ where n is the number of agents and tasks. That means that for $n = 4$ the number of combinations are 24, these could with some patience be tried by hand. For $n = 10$ the number of combinations are 3628800, still possible with a computer, but with $n = 30$ the number of combinations become in order of magnitude 10^{32} and it would not be feasible to try every combination even with a computer.

This means we need smarter way find the solution. For different classes of assignment problems there exists different algorithms to solve each much faster [2].

1.1.1 Linear Assignment Problem

When people speak of the "assignment problem" without any extra qualifications you generally mean the the linear assignment problem (LAP).The goal of this problem is to find a solution where the sum of the cost for each agents to complete each task is as low as possible.

LAP requires that the number of agents are equal to the number of tasks. But if you have a situation with more agents than tasks you can add extra dummy tasks with a zero cost to satisfy the requirement. LAP also requires that the costs for each agent to complete each task is know and is independent of the costs for other agents.

Currently, the best algorithm for this problem is the Hungarian algorithm [3]. This algorithm was created by Kuhn [7], but was based on work by a Hungarian called Egervary, thus the name the Hungarian algorithm.

1.1.2 Linear Bottleneck Assignment problem

The linear bottleneck assignment problem (LBAP) is closely related to LAP. But instead of minimizing the sum of all costs, LBAP minimize the maximum cost over all agents. There exists several algorithms able to solve this problem [6]. The algorithm used in this work is the Threshold algorithm since I found it the easiest to implement.

1.2 Notations

GF - Greedy first algorithm.
HA - Hungarian algorithm, also referred to as Munkres algorithm.
HWT - Hungarian algorithm with threshold.
Total distance or total cost - sum of all paths the drones have to travel.
Max time - The time it takes until all drones have reached their goal.
ROS - the Robot Operating System.

2. Algorithms

Three different algorithms are used. All algorithms assume you have a cost matrix with rows representing agents (or drones) and columns representing tasks(or goal locations). The cost for one drone to reach a goal location is $c_{i,j}$ where i are drones and j are goals.

A cost matrix with four drone and four goals.

$$\begin{pmatrix} c_{1,1} & c_{1,2} & c_{1,3} & c_{1,4} \\ c_{2,1} & c_{2,2} & c_{2,3} & c_{2,4} \\ c_{3,1} & c_{3,2} & c_{3,3} & c_{3,4} \\ c_{4,1} & c_{4,2} & c_{4,3} & c_{4,4} \end{pmatrix}$$

2.1 Greedy first

This is a naïve approach to assigning the goals. First find the minimum task cost for each drone. Then sort the drones based on this cost, and finally for each drone pick the task with lowest cost that has not been picked by a previous drone. A problem with this approach is that the last drone could end up with its worst possible goal. The complexity of this algorithm is $O(n^2)$.

2.2 Hungarian algorithm

To solve the problem of Greedy First algorithm we can use the Hungarian algorithm. The Hungarian algorithm can solve the problem of assigning a goal to each drone where the total cost for all drones are minimized. For the Hungarian algorithm to work, all costs must be independent of each other. For this experiment this condition is not entirely true and thus an estimated cost is used. In general this kind of problems are refereed to as "the linear assignment problem". The Hungarian algorithm is sometimes refereed to as Munkres algorithm. The Hungarian algorithm uses the fact that adding or

subtracting from entire rows or column of the cost matrix does not change the final solution. The complexity of the implementation used is $O(n^3)$.

Step 1

Subtract the minimum cost $\min(c_{i,1\ldots n})$ for each row from each row and then do the same for each column.

Step 2

Cover all zeros in the matrix with as few horizontal or vertical lines as possible. If the number of lines are less than the dimension of the matrix, continue to step 3.

If the number of lines are equal to the dimension of the matrix an optimal solution can be found by pairing rows and columns where they share zeros.

Step 3

Find the minimum of all uncovered numbers and subtract the minimum from all uncovered values and add the minimum to all values covered by two lines. Repeat step 2 and 3 until a solution is found.

2.3 Hungarian with threshold

While the Hungarian algorithm can minimize the total distance this is often not what you are most interested in. In most cases you want to minimize the time it takes to complete the entire problem. Since the drones move at an approximately constant speed the max time is proportional to the max cost. What is done in this case is to use the threshold algorithm to first exclude drone/goal combinations above a computed threshold before running the Hungarian algorithm on the remaining set of drones and goals. The threshold is calculated with the Threshold Algorithm. In short the threshold is calculated by repeated guesses based on the median of the values between a lower and upper bound until you find the minimum value for the threshold. Steps 1-4 below describes the Threshold algorithm in more detail.

A sub-problem required for step 3 and 5 for the threshold algorithm is determining if a perfect matching exists for a bipartite graph. An algorithm able to do this in polynomial time is described in [1]. The threshold algorithm can be run with a complexity of $O(n^{2.5}/\sqrt{\log n})$[6, p. 174]. Since this value is lower than HA complexity of $O(n^3)$ the HA complexity dominates HWT resulting in the the HWT having the complexity of $O(n^3)$.

Step 1

First you need to keep track of a lower bound $c_l = \min_{i,j}(c_{i,j})$ and an upper bound $c_h = \max_{i,j}(c_{i,j})$ for the threshold.

Step 2

Find the median c_m of all values in the cost matrix that are above c_l and lower than c_h.

Step 3

Check if a perfect bipartite matching is possible by pairing rows and columns where $c_{i,j} < c_h$. If it is possible to create pairs for all rows and columns set $c_h = c_m$. If not, set $c_l = c_m$

Step 4

If there exist a value between the lower and upper bound,$c_l < c_{i,j} < c_h$, go back to step 2. If not check for a perfect bipartite matching where the threshold c_T is c_l. If a perfect bipartite matching exists c_T is set to c_l otherwise it is set to c_h.

Step 5

Now set all values in the cost matrix above the threshold c_T to ∞ and feed the resulting matrix to the Hungarian algorithm described in the previous section.

An example of a cost matrix with made up numbers.

$$\begin{pmatrix} 8 & 2 & 3 & 3 \\ 2 & 7 & 5 & 8 \\ 0 & 9 & 8 & 4 \\ 2 & 5 & 6 & 3 \end{pmatrix}$$

Cost matrix after Threshold, threshold values is 6.

$$\begin{pmatrix} \infty & 2 & 3 & 3 \\ 2 & \infty & 5 & \infty \\ 0 & \infty & \infty & 4 \\ 2 & 5 & \infty & 3 \end{pmatrix}$$

Solved matrix after Hungarian Algorithm

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Final solution drones $1-4$ are matched with tasks $(2, 3, 1, 4)$.

3. Implementation

This chapter describes how the experimental setup is implemented. The implementation is based on the ROS framework that handles most of communication between the different part of the experiment.

3.1 Experiment description

The experiment consists of a group of simulated drones that fly in a simulated environment. They are supposed to move from their starting position to their goal location in a map with obstacles. The drones all fly at the same height and thus can not fly over the same location at the same time (In an indoor environment the drone would interfere with each other if flown over the same location due to air turbulence). This is a slight violation of one of the condition of linear assignment problems since the real distances (costs) are not entirely independent of each other. Instead approximated distances are used as input for the assignment algorithms.

The maps are a grids of 1x1 meter squares that are either open space or obstacles, this simple map format is used to make finding valid goal and starting locations easy.

3.2 Introduction to ROS - Robot Operating System

Sometimes you end up having to solve an unexpected problem where you don't have an existing solution ready to be used. Solutions for different parts of the problem may have been solved in different systems but they can't talk with each other since all have different interfaces and different methods for communication.

This is a problem the Robot Operating System (ROS) [5] is trying to solve. It offers a unified platform for creating so called nodes that each solve a part of a problem, and where each node talk the same language used for communication. ROS also provides a lot of ready-made nodes that are easy to utilize when building a new system.

ROS is in it self not bound to any particular programming language. Core parts (Most ROS nodes) are written in C++, Python and LISP. There is some support for Java and Lua. The Operating systems with best support are currently Ubuntu and MAC OS/X but can be run on any Unix based operating system.

ROS nodes

A ROS application consists of ROS nodes. Each ROS-node is an independent program taking care of a specific task needed for the robot application. Different nodes have different tasks, some run an algorithm, some provide visualization of data and some provide services for other nodes needed perform their tasks.

ROS message

ROS-messages are packages with language independent data that is used for communications between different ROS-nodes. ROS-messages are based on few base data types (like int, string and array) can then be hierarchically structured into more complicated types in a similar way to C-structs. ROS-messages are compiled along with ROS-nodes into different languages and can then be used for communications between different ROS-nodes. Fig 3.1 shows the structure of a ROS-message called gemoetry_msgs/Twist. It contains two gemoetry_msgs/Vector3s which in turn contains 3 float64s each.

ROS topic

ROS-topics are used by nodes to relay information asynchronously to other nodes that may choose to subscribe to the topics if they need the information provided. All data on one topic can only consist of data of one ROS-message type, this makes sure all data transmitted are of the type expected. Fig 3.2 shows a ROS-message of the type gemoetry_msgs/Twist being sent on the ROS-topic cmd_vel (command velocity) which tells the drones at what speed they should move.

```
geometry_msgs/Vector3 linear
  float64 x
  float64 y
  float64 z
geometry_msgs/Vector3 angular
  float64 x
  float64 y
  float64 z
```

Figure 3.1: Definition of a ROS-message type.

```
linear:
  x: -0.814936391259
  y: 0.0193206766152
  z: 0.0
angular:
  x: 0.0
  y: 0.0
  z: 0.00333481658148
```

Figure 3.2: A message from the ROS-topic cmd_vel.

ROS service

Nodes can also provide services for other nodes. The data types of parameters and returned data are ROS-messages and are defined in a srv(service) file and is compiled with the ROS package into a class for all languages that may use it. The program calling the service will block until the result is returned from the service or until a specified timeout time is reached.

ROS parameter

ROS provides a parameter server where nodes can store information that can be shared with different nodes. This is useful for storing settings that are shared by several nodes.

Tf

Tf (transformation) is a framework for performing transformations between different coordinate systems. Information about how to perform transformations are published in the ROS-topic called tf. ROS-messages contains information about what coordinate frames the transformation can transform between and the translation and rotation needed to perform each transformation. For example the odom(odometry) coordinate frame to base_footprint as in fig 3.4. Odometry frame here is the drones internal coordinate system and base_footprint is base for drone structure where different parts of the drone like sonar_link can have an offset to. Tf allows building chains of transformations where each link can be handled by a different node. Fig 3.3 is an example of a transformation tree used in this work.

Roslaunch

Roslaunch is a utility program for ROS used to launch groups of nodes with a single command. You run roslaunch with a .launch file that can launch ROS-nodes with specific parameters, load parameters to the ROS-parameter server or include other .launch files. This way you can create hierarchies of .launch files to avoid having to duplicate entire launch files if you just need a slightly modified version . It can also monitor if some node crashes and either restart it or abort and close all other nodes.

3.3 Gazebo

Gazebo is the simulator used for this experiment. It is a standalone program but is well integrated with ROS in the form of plugins for Gazebo.

In this experiment a empty world for gazebo is used. The only thing simulated are the drones. The reason an empty world is used is to reduce the CPU load. Collision detection with obstacles other then drones is not needed for this experiment since all the plans made by the global planner

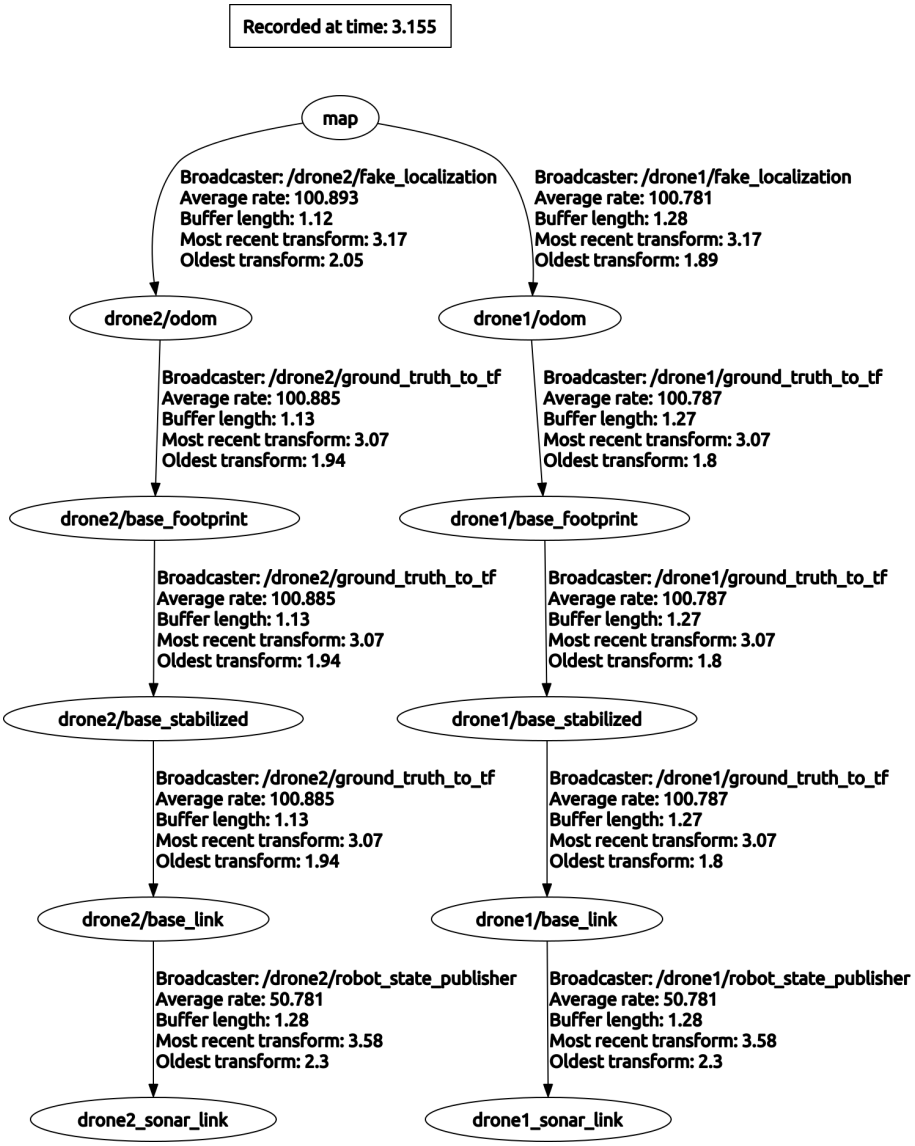

Figure 3.3: A tf transformation tree with two drones. The circles are coordinate frames, and different ROS-nodes publishes information on how to transform between them.

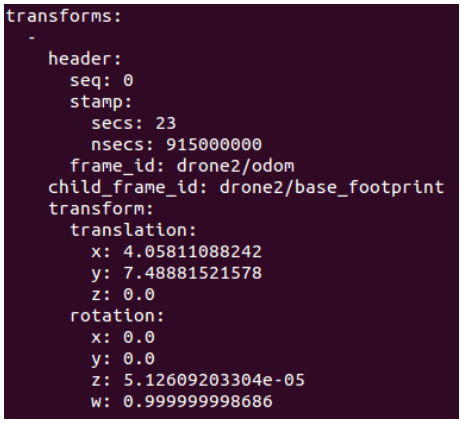

Figure 3.4: An example of a tf ROS-message containing information about how to transform between the odom and base_footlink coordinate frame for drone2.

make plans that stays clear of all obstacles. (Since the global planner creates plans that are collision free with the terrain)

3.4 Global Planner

After drones have been assigned their goals the global planner is called. The input is a set of initial positions for the drones and a goal for each drone. This planner created paths the drones can follow to without colliding with each other on their way to the goal.

The algorithm run by the global planner is prioritized planning [8]. The basis for prioritized planning is assign a priority to all agents, in this case, drones. Then find plans for all drones with a single agent motion planning method where drones with running the motion planning algorithm first are treated as moving obstacles for later drones.

3.5 Used Ros Nodes

Rviz Rviz is data visualization tool for ROS. Since most ROS-nodes does not have a GUI to show their current state, you can use Rviz to gather data sent on different ROS-topics to visualize what is happening.

Figure 3.5 and 3.6 show the map, the drones and the paths the drones follow. Notice that the drones in the two examples have the same starting locations but different goals since different assignment algorithms are used. The drone with the black path gets a goal closer to its starting location due the threshold used in Hungarian with threshold.

Figure 3.5: Rviz example Hungarian Algorithm

Figure 3.6: Rviz example Hungarian With Threshold

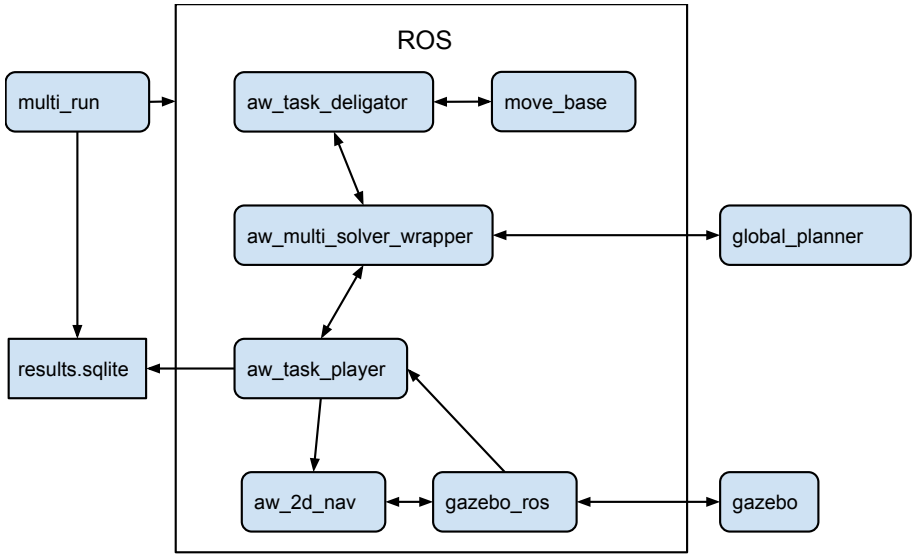

Figure 3.7: Graph of the most important parts of the implementation

Move_base: Move_base is navigation module. It can be used to navigate a map and avoid both static and dynamic obstacles. For this experiment only the static map planning functionality is used. For this experiment move_base is used to estimate the distances for all drones to each goal.

This node is much more complex than what is needed for this experiment, a simpler node running the A* algorithm would suffice.

Gazebo_ros: This is a plugin in gazebo that behaves like a ROS node to provide ROS integration for gazebo. This publishes information about the state of the drones flown in the gazebo simulator.

Map_server: Publishes a map image and specifies parameters like resolution, size and origin. The map is a is a grayscale image where color values below a set threshold is considered occupied space. The map is fixed in the map tf coordinate frame on the top of Fig 3.3.

Fake_localization: Provides tf-transformation between the map coordinate frame and the drone base_link coordinate frame.

Ground_truth_to_tf: This node is used for providing the true simulation position of the drones to other nodes. Publishes transformations to the tf topic.

Robotstate_publisher: This publishes the internal tf-transformations of different parts on the drone. In this case sonar_link, used to by the drones to know how high above the ground they fly.

3.5.1 ROS-nodes create for this experiment

Below are the ROS-nodes written for this experiment.

Aw_2d_nav: This node calculates the distance from the current position and the goal and outputs the distance as the velocity towards the goal with a max velocity to prevent the drone from flying to fast. This is a very simple approach but works for this experiment since the the goals are always nearby and are planned so no obstacles are in the way.

Aw_task_deligator: This node contains the logic of all the assignment algorithms. It provides a ROS-service that provides a list of starting locations and their goals.

Aw_multiplanner_wapper: To run several drones at the same time and make sure they don't collide, a global planner is used. But the global planner used is not integrated with ROS so this node is used to provide and ROS interface to the planner.

multi_run: This is not a ROS-node but a Python script used to automatically run the experiment multiple time without manually calling roslaunch with the correct settings. You can give it a list of different number of drones, a list with the assignment algorithms you want to test and the map you want to run on. Then it generates starting and goal locations on free locations on the map and runs all combinations of drone numbers and assignment algorithms on that set of starting locations and goals. You can also tell the script to repeat this process as may times as you want.

3.5.2 Execution

This section describes how a run of the experiment is done.

Initialization: A run of the experiment is initialized with a call to ros_launch with the desired initial condition. A small Python program (multi_run) is used to repeatedly do this with different initial condition and settings.

Cost estimation: When all parts of the experimental system have started the cost estimation is done. The ros-node move_base is used to create path from each drone to all potential goals, the length of these path are then used as cost estimates.

Goal allocation: When the cost matrix is done for all the drones, the assignment algorithm is run, (GF, HA or HWT depending on launch settings). The result of this step is drone/goal pairs.

Global planner: Now when we know were all drones need to go the global planner can be run. The global planner creates collision free path for all drones to reach their respective goals. This step can fail, if for example two drones need to move in opposing directions in a tight corridor. If this step fails the experiment run terminates.

Plan execution: The plan we get from the global planner is a list of waypoints for each drone. Before the next waypoint is sent to each drone all drone have to be at their previous waypoint. This is done to preserve the collision free nature of the path, otherwise a fast and a slow drone could collide if they reach a point in the map where they both have to pass. The experiment run terminates when all drone have reached their final goal.

3.5.3 Simulation

Three different test maps are used. Each map is run with 4,6,8 and 10 drones. Each map is also run with three different algorithms, Hungarian algorithm, Hungarian algorithm with threshold and Greedy first. Each combination of these, map, drone count and algorithm, is run 100 times each for a total of $3 * 4 * 3 * 100 = 3600$ runs.

All starting and goals locations are randomly placed on the map. The starting and goals locations are placed on a 1x1 meter grid with 0.5 meter offset in x and y-coordinates compared to the map so all goals are placed in the center of the pixels of the map image.

Maze3

Maze3 is a maze with three rooms, each with several entrances and exits.

Maze4

Maze4 is also a maze. It is more difficult for the global planner than Maze3 since there are less open space and less alternative routes.

LiuB2

This map is based on one of the buildings in Linköpings University. Its a small part of the B-building. Its slightly upscaled to allow more detail since the walls are a minimum of one meters wide. This map is easy for the global planner to make plans for since there are a lot of open areas.

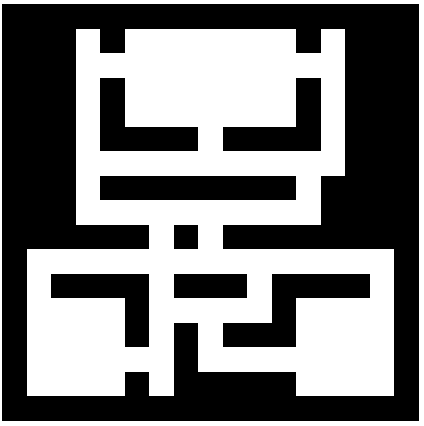

Figure 3.8: Maze3 map

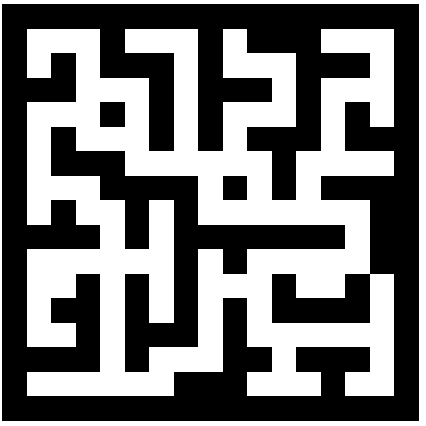

Figure 3.9: Maze4 map

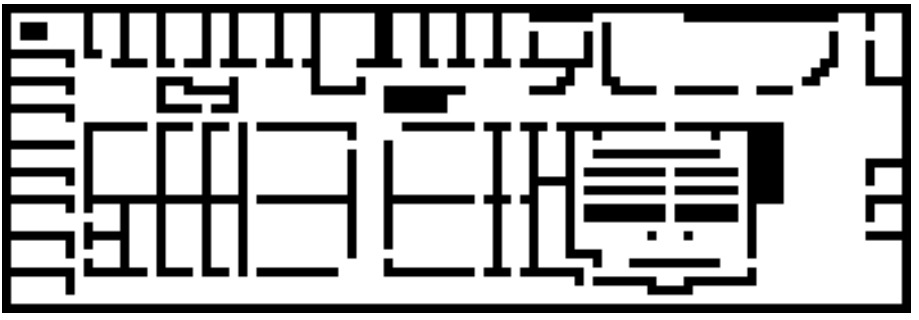

Figure 3.10: LiuB2 map

4. Results

The results presented in this chapter are the result of test run with the gazebo simulator. Each combination of map and algorithm and number of drones has been run 100 times. This is done to make sure the results presented are valid for a broad range of environments and see how the algorithms behave with various number of drones. All error bars in the figures below show the standard error of the mean.

Success rate

Three different metrics are used to compare the algorithms. First is the success rate, this is how often the global planner manage to create collision free paths for all the drones to follow. The global planner is not perfect and might not find a solution even when one is possible. The most important factor causing the planner to fail is cases where drones have to pass each other in a tight corridor. This can in some sense be viewed as a metric for how easy problems the algorithms generate, but it is highly dependant on the global planner used and the environment.

In fig 4.1 we can see that the assignments that HWT creates are easiest for the global planner to make plans for. We can also observe that the success rate depends on the complexity of the environment. Maze4 with its tight corridors makes it hard while the open spaces in liuB2 allows the planner to find plans most of the time.

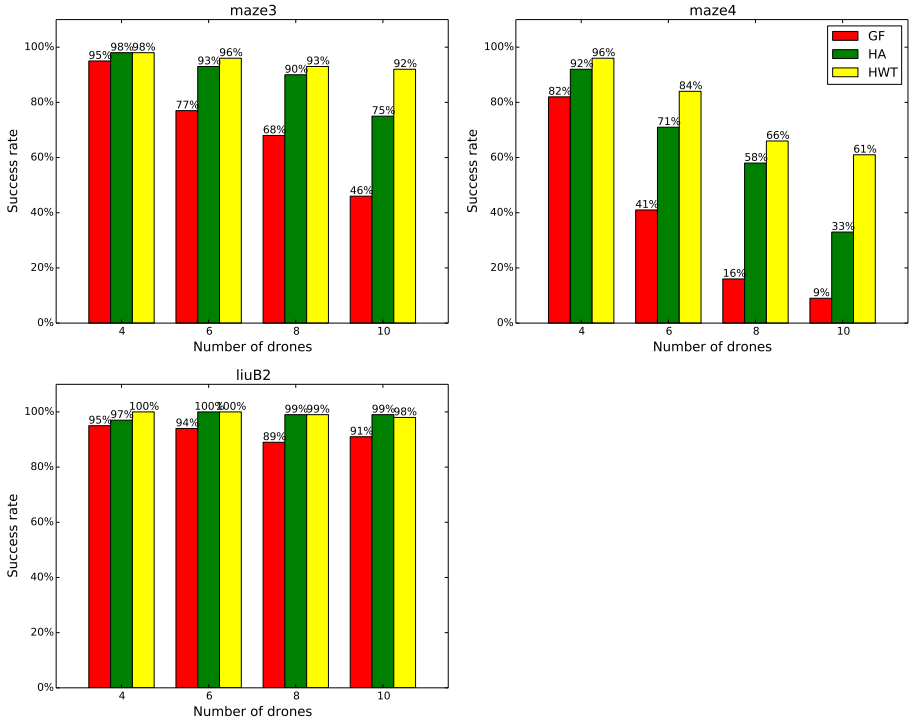

Figure 4.1: Planner success rates

Max Time

The second metric compared is, max time, the time it takes until all drones have reached their goals. The graphs to the left in figure 4.2 to 4.4 shows the average improvement compared to the GF algorithm. Only simulation setups where all algorithms assignments let the global planner make a plan are included. Since the difficulty of Maze4 for large number of drones we can see only a few samples remain and the uncertainty become very high but the difference in time is still large enough for the results to be statistically significant.

Overall the results are similar for all maps, HWT shows an significant improvement over HA and GF. The relative improvement is greater for simulation runs with larger amount of drones. In the figures to the right we can observe that the max time for HWT stays about the same regardless of number of drones. This can be explained be two opposing influences on the result. As more drones are added on the map the more likely there is that there is a goal close to the drones resulting in lower max times. The second opposing effect stems from the costs used as input to the algorithm are estimates and assumes all drones can go where ever they need without

considering each other the global planner will introduce an unpredictable influence on the results. As the number drones increases the cost estimates are more likely to be off leading to increased max time.

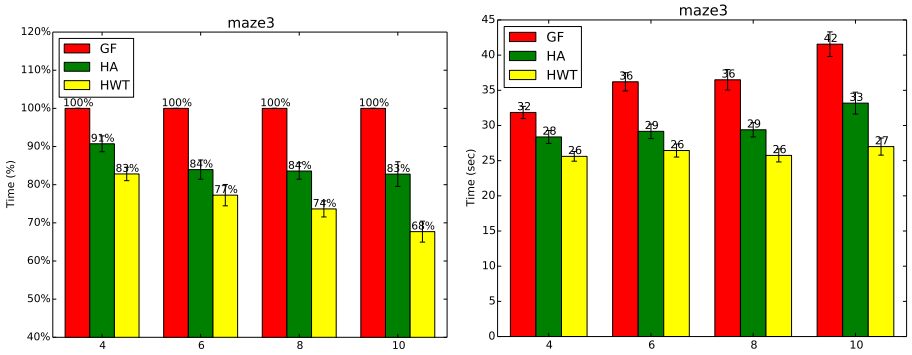

Figure 4.2: Maze3 max time

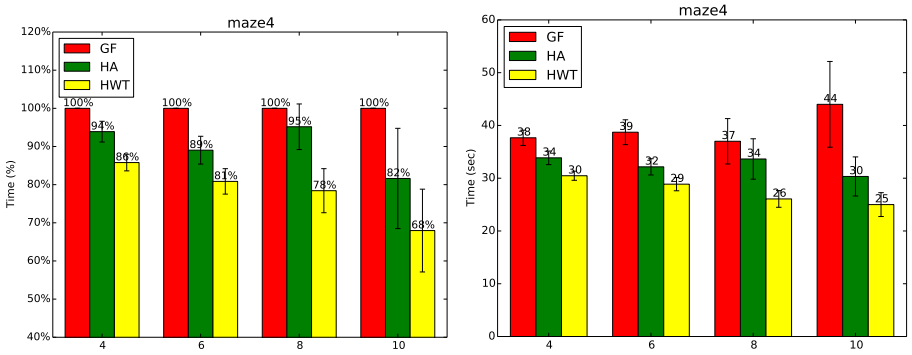

Figure 4.3: Maze4 max time

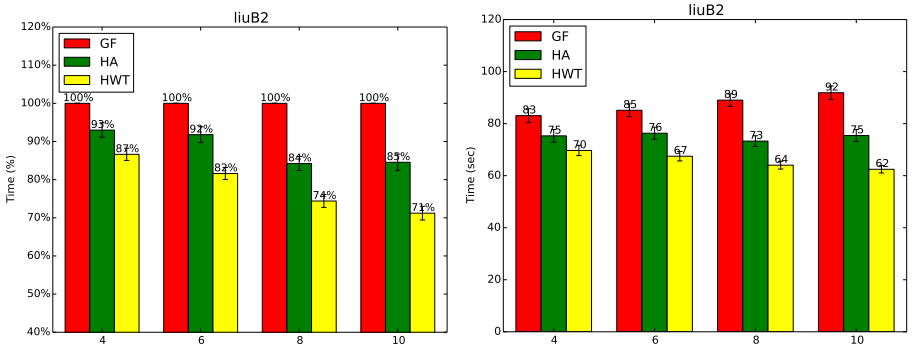

Figure 4.4: Liu2B max time

Total distance

The third metric is the total distance the drones have to travel to reach their goals. As seen in figure 4.5 to 4.7 HA and HWT gives similar results, both show significantly lower total distances than GF. In 4.7 we can see that HA gives slight better results than HWT. This is expected since what HA does is minimizing the total cost, and the cost used for input to the algorithms are estimates of the distances from drones to goals.

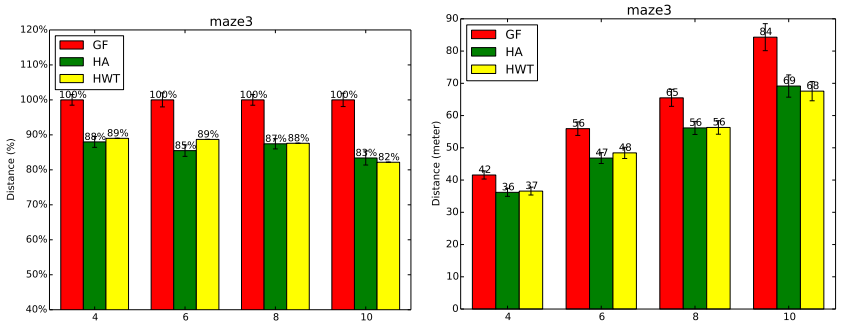

Figure 4.5: Maze3 total distance

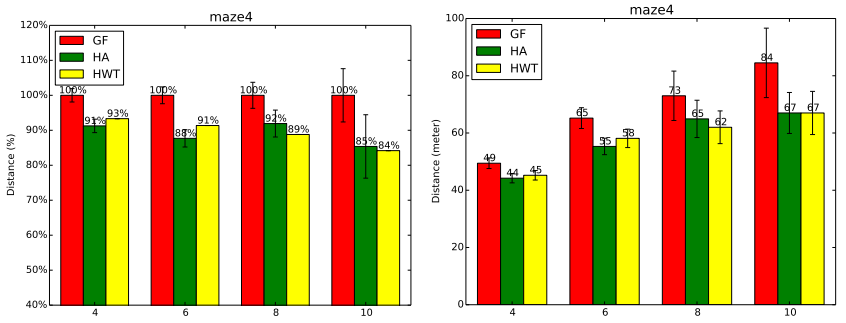

Figure 4.6: Maze4 total distance

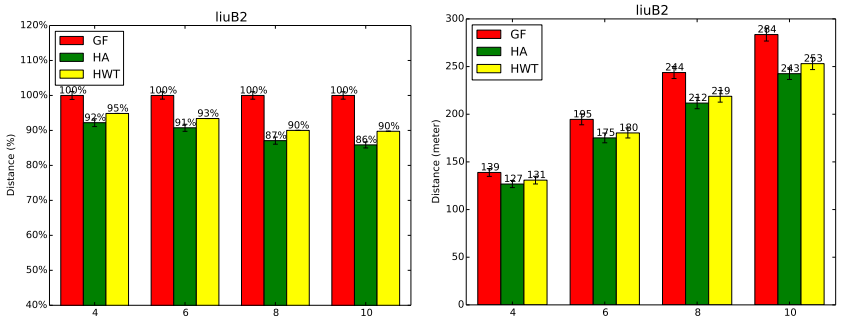

Figure 4.7: Liu2B total distance

5. Conclusion and Discussion

In this report we compared three different assignment algorithms. The Greedy First algorithm is simple to implement and has the lowest time complexity but is not good enough when it comes to the results. It generates solutions that are hard to solve for the global planner and the plans have the longest total distances and highest max time.

The Hungarian algorithm is a well documented algorithm which makes it easy to find examples on how it works and how to implement it. The Hungarian algorithm performs the best for the problem it is meant to solve, namely minimizing total distance (or total cost).

The combined algorithm Hungarian with Threshold give the lowest max time of all the algorithms. If you compare the total distances for HWT with HA, HWT is equal or marginally worse than HA, but if max time is of interest HWT is a big improvement over running HA alone. This makes HWT the best algorithm in most cases.

5.1 Possible improvement

One could test on what scales the computational complexity of the algorithms tested would become important. This is not done in this work since other parts of the experiment used could not be run with high enough number of drones to make the difference significant.

The global planner in this experiment uses a prioritized planning algorithm. This algorithm does not guarantee that optimal plan is found or may even fail to find a plan. Using another better planner here would remove some of the noise in the results. For this experiment the speed of the planner was prioritised over more optimal plans.

5.2 Related problems

In the real world, there are a lot of situations that does not fit into the problem categories of the LAP and LBAP. Even this experiment does not fully comply with the linear part of the problem since the drones can block each other, but it is still useful as an approximation. If a problem is to

complex to be represented as LAP the problems can generally not be solved optimally in polynomial time. A lot of work is done to find algorithms able to solve these kind problems. In the work done by Lantao Liu and Dylan A Shell[4] they present an algorithm for generalized bottleneck assignment problem. Their aim is not to always find the optimal solution but a solution that is close enough to the optimal solution to be useful. This is a trade off often needed when you venture beyond linear assignment problems.

Bibliography

[1] H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time o(n1.5mlog n). *Information Processing Letters*, 37(4):237 – 240, 1991.

[2] George Bernard Dantzig. *Linear programming and extensions / George B. Dantzig.* Princeton, N. J. : Princeton Univ. Press, 1963, pr. 1968, 1968.

[3] Harold W. Kuhn. The hungarian method for the assignment problem. *50 Years of Integer Programming 1958-2008*, page 29, 2010.

[4] Lantao Liu and Dylan A Shell. Physically routing robots in a multi-robot network: Flexibility through a three-dimensional matching graph. *The International Journal of Robotics Research*, 32(12):1475–1494, 2013.

[5] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[6] Silvano Martello. Rainer Burkard, Mauro Dell'Amico. Assignment problems [online resource], 2012.

[7] Alexander Schrijver. On the history of combinatorial optimization (till 1960). *Handbooks in operations research and management science*, 12:1–68, 2005.

[8] J.P. van den Berg and M.H. Overmars. Prioritized motion planning for multiple robots. Inst. of Inf. & Comput. Sci., Utrecht Univ., Netherlands, 2005.