ETH zürich

Department of Computer Science
Computer Vision and Geometry Group

Graph Based Navigation on Resource Constrained Systems

Samuel Sadok

Bachelor's Thesis

2016

Supervisors: Prof. Marc Pollefeys
Lorenz Meier

Abstract

For a successful Return-To-Land operation, it is desirable for the vehicle, to only move along known-good paths, i. e. paths that were flown before. For that, a flight graph must be stored on the vehicle. On some drone platforms, memory is very limited and storing the complete flight graph the naïve way is not possible.
This thesis develops a compact representation of the flight graph, together with several methods to achieve a very small memory footprint, making it feasible for the most resource constrained platforms. The methods are implemented as part of the PX4 autopilot.
To conclude the work, the performance of the implementation is evaluated using sample flight paths.

Contents

Chapter 1

Introduction

1.1 The PX4 Project

The PX4 autopilot [1] is a modular software solution for unmanned aerial vehicles (UAVs), commonly known as drones. It is used on many types of drones and missions and has a diverse user base.

The software is divided into modules, where each module can be considered as a separate, single-threaded process. Modules are started on system startup or on demand. The system ensures that high priority tasks, such as the attitude control loop, get sufficient CPU time.

Even though modules may share a single address space on some platforms, this is not guaranteed. Inter-module-communication is managed by the Object-Request-Broker (`ORB`). Modules can use the `ORB` to publish or subscribe to values of a certain type.

1.2 Motivation

This thesis is motivated by an issue with the Return-To-Land (`RTL`) feature of PX4. Return-To-Land can be requested by the vehicle commander. It is especially useful, if the user can no longer ensure safe operation of the vehicle due to far distance or obstruction of the sight contact. It is also commonly used for convenience, after a mission is completed. Moreover, it is invoked automatically if the radio link to the ground station is lost.

Currently, the `RTL` mode operates as follows:

1. Fly straight up to a predefined altitude.
2. Fly in a straight line to directly above the home position while maintaining the altitude.
3. Descend and loiter or land, depending on configuration.

There will obviously arise problems if there is a tall building between the vehicle and the home position, or if the vehicle is below some obstacle, such as a tree,

when the `RTL` mode is invoked. The obvious solution would be to record the entire flight path in an array of floating point numbers and fly back along the path when requested. There are two major problems with this approach: First, the `RTL` mode may be invoked after a long flight around the same location. This would mean that the vehicle may take a return path that is much longer than it has to be. This could be mitigated by detecting close passings of the flight path to itself and viewing the flight path as a graph.

The second problem is a fundamental technical issue: The PX4 autopilot is designed to run on a multitude of different hardware platforms, some of them having very limited memory resources. On such platforms, it is not possible to store the entire flight path in a naïve way. Consider, for example, a tracker that records a new position after every meter of covered flight path. After one kilometer of flight, the memory usage would already be 12kB, if a position consists of three 32-bit floating point numbers. Moreover, the tracker must obey a statically known memory limit, otherwise there is a risk of using up all available resources and potentially jeopardizing the system integrity.

Looking at the example, it seems plausible that such a tracker can be optimized, for instance by reducing the recording fidelity or by detecting straight lines in the flight path. This thesis looks at various methods to reduce the memory requirements for graph based navigation. The focus of the thesis is an implementation of a tracker which uses drastically less memory than the naïve approach, and thus enables safe Return-To-Land operation, even on the most resource constrained systems.

1.3 Constraints

For the implementation, several constraints were layed down:

- The memory usage shall not exceed a limit of 10kB.
- Memory usage (both heap and stack) must be deterministic.
- The CPU usage is not considered critical. If some graph operations take several hundred milliseconds, this is still acceptable as the graph is not part of the real time control loop. It should be ensured though, that the CPU usage is not too excessive.
- The quality of the PX4 autopilot is supposed to be monotonically increasing with time. Therefore, any modifications of the RTL mode must perform at least as conveniently and safely as the existing implementation.

1.4 Approach

- A simple compact linear path storage was designed.
- The concept was extended to store the complete graph. Some representations were rejected based on considerations about complexity and maintainability.

- Multiple heuristics were implemented to reduce memory usage, based on assumptions about common usage scenarios.
- The implementation was validated using unit tests, as well as SITL (software-in-the-loop) testing.
- The implementation was tested on actual constrained hardware to verify its feasibility in terms of computational effort and memory usage.

Chapter 2

Implementation

The code for the implementation can be downloaded at
https://github.com/samuelsadok/Firmware/tree/3cfc0190fcfe6495caed3065b98efc7b03cea487

This link points to the latest Git commit which should be considered part of the thesis. The descriptions in this chapter attempt to summarize all important implementation aspects and design considerations . If more detail on a particular feature is desired, the reader is encouraged to consult the actual implementation code and the comments therein. The most relevant files are:

- `src/modules/navigator/tracker.cpp`: Implementation of most of the graph-based navigation
- `src/modules/navigator/tracker.h`: Declarations and descriptions of the functions in `tracker.cpp`
- `src/modules/navigator/rtl_advanced.cpp`: Code for the graph-based Return-To-Land flight mode

Some of the descriptions here differ slightly from the implementation in order to omit uninteresting details and simplify notation. However, important differences are mentioned when they do occur.

2.1 System Operation

Most of the functionality described in this thesis is implemented in the `Tracker` class, which constitutes a component of the `Navigator` module of the PX4 autopilot. The `Navigator` subscribes to changes of the vehicle position as well as the home position, and forwards them to an instance of the `Tracker` class. The home position is a special position that is determined by the system. It usually equals the location of takeoff and is close to the user.

The `Navigator` can be in one of multiple available flight modes. A mode switch occurs either upon user input or automatically. Two of these modes rely on path recommendations of the `Tracker`. Likewise, the `Tracker` maintains two distinct path

storages, each targeted at a specific flight mode.

The **flight graph** (Section 2.3) attempts to record the entire flight path, possibly trading off some of the precision. It's main purpose is to support the Return-To-Land flight mode (Section 2.1.1). As shown in Section 2.7, several heuristics help in managing memory usage.

The **recent path** (Section 2.2) only stores the most recent positions that the vehicle visited. It supports the `RCRecover` flight mode (Section 2.1.2). In contrary to the graph, its precision is constant.

2.1.1 Return-To-Land

Once the `RTL` mode is invoked, it requests a path finding context from the tracker (Section 2.6). It then uses this context to iteratively obtain new position recommendations and turn them into waypoints for the vehicle, until the home position is reached. It is the job of the `Tracker` to ensure that these recommendations make up a sensible path and do not diverge too far from previously visited positions. Once the vehicle is close enough to the home position, landing is initiated.

Despite rigorous testing, the graph implementation must be expected to have bugs. For this reason, the `RTL` mode surveils the vehicle progress. If no new waypoint can be set up within a predefined time span, it falls back to the legacy `RTL` mode. Furthermore, the same action is taken if any internal sanity checks of the `Tracker` fail. These safeguards only protect against a specific class of missbehaviour, but it is conceivable that they can be extended if found to be neccessary.

2.1.2 RC Recovery

An `RCRecover` mode is available to handle loss of radio control (RC) signal appropriately. It operates in a similar way as the `RTL` mode, in that it asks the tracker for position recommendations. However, in contrary to `RTL`, the goal is not to fly to a specific destination, but only to return to a good signal.

RC loss often occurs when the vehicle flies behind a large obstacle, such as a building. When RC loss is detected, flying back in the reverse direction usually quickly leads to recovery of a good signal. All of this might happen in close proximity to the offending obstacle. Thus, the vehicle should fly at high precision during RC recovery.

The flight graph supporting the `RTL` mode trades off precision for the sake of coverage. This seems unsuitable for use in an RC loss scenario, especially since this tends to occur far from the home position, where precision is reduced more aggressively. This motivates the implementation of the recent-path-buffer.

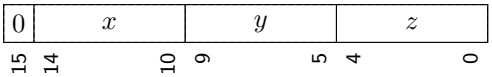

Figure 2.1. Layout of a compact delta element (`delta_item_t`)

2.2 Recent Path Buffer

The recent-path-buffer (`recent_path`) can be seen as a ring buffer of positions. It has configurable capacity, but is supposed to be small (e. g. 16 positions).

Each position $\mathbf{p}_i$ is stored relative to the preceding position $\mathbf{p}_{i-1}$. The according delta is obtained by

$$\Delta_i := \underbrace{\operatorname{round}\left(\frac{1}{u} \cdot \mathbf{p}_i\right)}_{\widetilde{\mathbf{p}}_i} - \underbrace{\operatorname{round}\left(\frac{1}{u} \cdot \mathbf{p}_{i-1}\right)}_{\widetilde{\mathbf{p}}_{i-1}}$$

where u is the unit size (usually 1 meter), and *round* applies element-wise rounding-to-closest-integer with tie-breaking away from zero. The rounding is required due to the delta representation in memory (see Figure 2.1). The notation $\mathbf{p}$ is from now on used instead of $\widetilde{\mathbf{p}}$ to denote a normalized and rounded position, as virtually all of the implementation operates on such integer positions.

The most recent position $\mathbf{h}$ (the **path head**, `recent_path_head`) is stored in absolute coordinates, so each earlier position $\mathbf{p}_i$ can be obtained by $\mathbf{p}_i = \mathbf{h} - \sum_{j=i+1}^{n} \Delta_j$, where n is the total number of positions in the buffer.

During recording, a new position is only appended when its distance to $\mathbf{h}$ is large enough. If the new position is close to any other position $\mathbf{p}_i$ already in the path, the buffer is rolled back up to that position, such that $\mathbf{h} \leftarrow \mathbf{p}_i$ and $n \leftarrow i$. This way, it is ensured that no two positions in the buffer are too close to each other.

The memory representation of each delta is shown in Figure 2.1. A 5-bit signed integer is allocated for each axis, so the representable range is $\Delta \in \{-16, \ldots, +15\}^3$. Position updates are registered at a frequency on the order of a hundred times a second, so a violation of this range seems very unlikely.

2.3 Flight Graph Data Structure

The representation of the flight graph mainly consists of two data structures:

- Position information is stored as a sequence of delta items, similar to the recent-path-buffer (Section 2.2).
- Intersections on the flight path, stored as a list of nodes. If two lines in the flight path pass close to each other, they are usually linked by a node.

To ensure a predictable impact of the flight graph on the overall resource usage of the system, it is desirable that both of these lists are allocated statically. However,

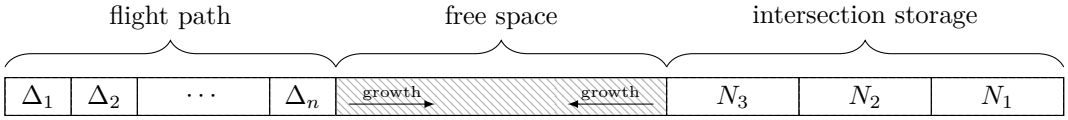

Figure 2.2. Layout of the graph buffer

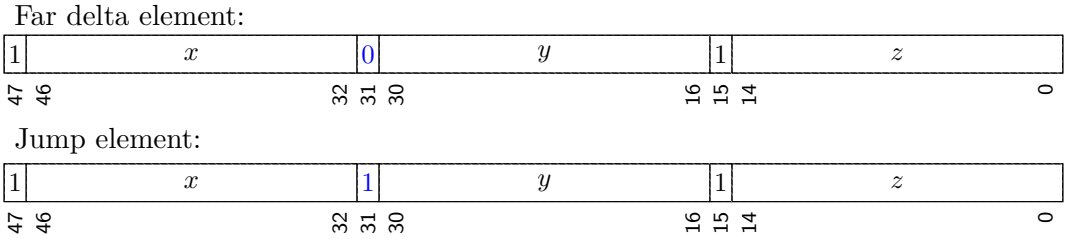

Figure 2.3. Additional delta element types. The bits 47, 31 and 15 enable parsing of the stored path in both directions.

the ratio of delta elements to nodes depends on the flight pattern and cannot be known in advance. This means that in the general case, when using two separate buffers, one buffer may become full while there is still unused space in the other buffer.

To mitigate this problem, both of the lists share the same, statically allocated memory area (`graph`). More specifically, the delta sequence is stored at the beginning of this area and grows upwards (in terms of address), while the node list is stored at the end of the area and grows downwards (Figure 2.2). The graph is considered full when both lists collide. When this happens, certain heuristics are employed to attempt to reduce memory usage (see Section 2.7).

2.3.1 Path Storage

The flight path is stored as a sequence of delta elements, similar to the recent-path-buffer. However, in addition to the compact delta element (Figure 2.1), two more delta representations are introduced (Figure 2.3).

Compact delta element Most of the path is stored in compact delta elements, which occupy only two bytes of memory. Their memory layout is shown Figure 2.1. As discussed in Section 2.2, the representable range is $\Delta \in \{-16, \ldots, +15\}^3$. A compact delta item correspondsto a line connecting two vertices. If a line becomes too long to be stored as a compact delta element, it is either split into multiple compact elements or stored as a far delta element, whichever uses less memory.

Far delta element Large position changes can be stored as far delta elements. They occupy six bytes of memory, storing a 15-bit signed integer for each axis. The lowest possible value (`0xC000`) is reserved (Section 2.7.2). The resulting range is $\Delta \in \{-16'383, \ldots, +16'383\}^3$. When using a unit size of 1 meter, this corresponds to a distance of $\sim$16km. There are currently no mechanisms in place to handle larger deltas.

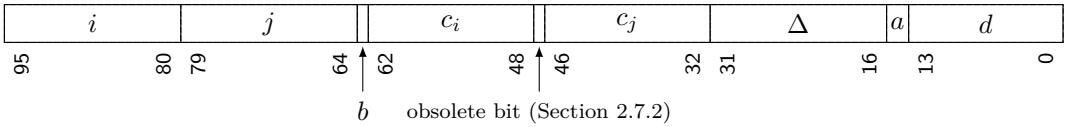

Figure 2.4. Node structure layout (`node_t`)

Jump element Jump elements allow to represent discontinuities in the flight path. Their representation differs from far delta elements only in one bit, which reflects the distinct meaning.

2.3.2 Position Representation

Any position on the flight path can be written as a tuple (i, c).

$i \in \{1, \ldots, n\}$ is an index pointing to a delta element Δ_i. Let the line represented by Δ_i start on $\mathbf{p}_{i-1}$ and end on $\mathbf{p}_i$, i. e. $\mathbf{p}_i = \mathbf{p}_{i-1} + \Delta_i$. Note that in the actual implementation, the indexing is slightly different. For instance, if the delta element occupies multiple slots in the buffer, the index always points to the last slot used by the element.

$c \in [0, 1)$ is a coefficient, which specifies a point somewhere along the associated line. A value of 1 represents the beginning of the line $\mathbf{p}_{i-1}$, a value of 0 represents the end of the line $\mathbf{p}_i$. This value is stored as a 15-bit unsigned integer, where the theoretical value $32'768$ corresponds to the coefficient 1. Note that the value 1 cannot be represented. This is not a problem in practice, as the beginning of a line equals the end of the preceding line.

Given the tuple (i, c) and the vertex $\mathbf{p}_i$ or $\mathbf{p}_{i-1}$, we can calculate the absolute position $\mathbf{p}$ represented by the tuple:

$$\mathbf{p}(i, c) := \mathbf{p}_i - \text{round}(c \cdot \Delta_i) \tag{2.1}$$
$$= \mathbf{p}_{i-1} + \Delta_i - \text{round}(c \cdot \Delta_i) \tag{2.2}$$

Similarly, $\mathbf{p}_{i-1}$ and $\mathbf{p}_i$ can be calculated when $\mathbf{p}$ is known.

2.3.3 Intersection Storage

A node links two lines i and j of the flight path that pass close to each other. In addition to that, it stores routing information for that intersection. Thus, the nodes act as enablers of the path finding algorithm Section 2.6. Without any nodes, the path finding algorithm would not know about any intersections and therefore only suggest to linearly follow the flight path. However, deleting nodes does not result in any loss of data. The node list can always be rebuilt from the flight path, provided sufficient free memory.

A node can be seen as a tuple $((i, c_i), (j, c_j), \Delta, d, b, a)$ and is represented in the implementation by the structure `node_t`, occupying 12 bytes (Figure 2.4). (i, c_i) and

(j, c_j) are the two positions in the graph that are linked. Δ stores the delta between the two points: $\Delta := \mathbf{p}(j, c_j) - \mathbf{p}(i, c_i)$.

$d \in \{0, \ldots, 16'382, \infty\}$ stores the most recently calculated distance of the node to home, with respect to moving along the graph. If this distance is invalid or was just changed, the dirty-bit b is set.

Note that a node actually consists of two points in close proximity. The stored distance-to-home is always at least as large as the true distance of either of these points. Since d is calculated based on the surrounding nodes, the introduced error may add up. However, no matter how large the error becomes, the worst thing that can happen is that the path finding algorithm will chose a non optimal path.

The action a can be one of `line1-forward`, `line1-backward`, `line2-forward`, `line2-backward`, where `forward` denotes the direction of increasing indices. This reflects how the stored distance was calculated. For the path finding algorithm, this means that when following the specified action, it is guaranteed to reach home after covering a distance of at most d.

The current implementation only stores routing information for a single destination. Based on the initial motivation of the flight tracker, there seems to be no obvious reason to support multiple destinations simultaneously. The implementation *does* however allow to change the destination dynamically.

Beyond that, if multi-destination support is to be added, the three last fields in each node would just have to be replicated for each destination. As the number of destinations changes, the node size would also change, so the system could delete all existing nodes and then rebuild the node list from scratch.

2.4 Graph Consolidation

Each new position that arrives at the `Tracker` and is far enough from the graph head, is appended to the end of the flight path. The most recent positions make up the unconsolidated area. After the maximum consolidation debt is reached, i. e. the number of new, unconsolidated positions m_Δ has reached a predefined limit, the consolidation algorithm (`consolidate_graph`) is invoked. This algorithm consists of three passes over the unconsolidated area. At the end of the algorithm, the graph is considered fully consolidated.

2.4.1 Line Detection

The first pass attempts to compress the path without introducing too much error. In the implementation, this is done by a line detection algorithm. In theory, other methods could be used as well.

The line detection pass operates in a greedy fashion. It starts at the first unconsolidated position and tries to find the longest valid line from there. Next, it

continues from the end of that line to find the next line. A sequence of k positions $\mathbf{p}_i \to \mathbf{p}_{i+1} \to \cdots \to \mathbf{p}_{i+k}$ is replaced by a line $\mathbf{p}_i \to \mathbf{p}_{i+k}$, if all intermediate positions $\mathbf{p}_{i+1}, \ldots, \mathbf{p}_{i+k-1}$ lie close to the new line (Figure 2.5).

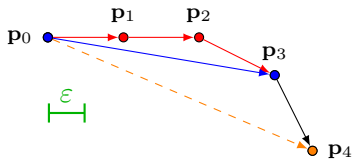

Figure 2.5. Line detection: The red lines are replaced by a single line $\mathbf{p}_0 \to \mathbf{p}_3$ because this is the longest valid line from $\mathbf{p}_0$. The line $\mathbf{p}_0 \to \mathbf{p}_4$ would be invalid, since the points $\mathbf{p}_2$ and $\mathbf{p}_3$ are no longer within the margin ε of this line. The margin is discussed in more detail in Section 2.7.1.

2.4.2 Redundancy Removal

The redundancy removal pass accounts for the fact, that a vehicle sometimes flies along a previously flown path. This is especially true during return-to-land operation, where the entire return path is close previously visited positions.

Therefore, it is attempted to remove redundant positions. A sequence of k positions $\mathbf{p}_i \to \mathbf{p}_{i+1} \to \cdots \to \mathbf{p}_{i+k}$ is replaced by a jump element $\mathbf{p}_i \to \mathbf{p}_{i+k}$ (Section 2.3.1), if all positions lie close to some preceding line (Figure 2.6). Note, that for $k < 4$, the jump element itself would require more memory than the original points. In such a case, the repacement is omitted.

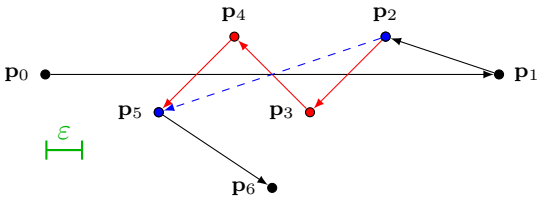

Figure 2.6. Redundancy removal: The red lines are replaced by the dotted line, stored as a jump delta element. The positions $\mathbf{p}_2, \ldots, \mathbf{p}_5$ are within the margin ε of the line $\mathbf{p}_0 \to \mathbf{p}_1$.

2.4.3 Node Generation

The node generation pass iterates through each new line, and checks if it is close to any previous line. If this is the case, a node is generated and added to the node list. While the first two passes are guaranteed *not* to increase memory usage, this last pass *does* increase memory usage. The nodes themselves are implicitly defined by the flight path, so theoretically the node generation pass adds no information. However, in practice, the path finding algorithm requires some space for each node where routing information is cached. Conceptually, the explicit node generation can be seen as reserving memory for the path finding algorithm.

2.5 Routing Data Synthesis

The routing data is generated and refreshed lazily. The `refresh_distances` algorithm is similar to the Bellman-Ford algorithm [2]. The main difference is that node updates take effect immediately and not only after the remaining nodes are visited. The algorithm looks as follows (set $t = 1$ for the initial iteration):

1. For each dirty node $N = ((i, c_i), (j, c_j), \Delta, d, b, a)$ $(b = 1)$:
   (a) Reset the dirty flag: $b \leftarrow 0$
   (b) Starting at point (i, c_i), walk forward on the path (in the direction of increasing indices), until another node N' is encountered. While doing so, calculate the length of the interval. Note that no absolute positions are required for this.
   (c) If N could achieve a shorter distance-to-home through the considered interval and N', update d accordingly. Additionally, store the action $a \leftarrow$ `line1-forward`. Analogously, if N' could achieve a shorter distance-to-home through N, update N'. Mark any updated node as dirty.
   (d) Repeat step 1b for three more intervals: backward from (i, c_i) and forward and backward from (j, c_j).
2. If there are any dirty nodes, increment t and go to step 1.

The algorithm is guaranteed to terminate.

Proof. Let $D^{(t)}$ be the sum of the distances d of all nodes at the end of iteration t. Note that node distances are stored as integers, so we have $D^{(t)} \in \mathbb{N}$.

The distance of each node is monotonically decreasing with respect to t: the update step (1b) only allows updates that improve a node distance. The sum $D^{(t)}$ is a sum of monotonically decreasing functions and thus monotonically decreasing itself.

Furthermore, node distances cannot be negative, because they start at infinity and cannot be reduced beyond zero (since interval have non-negative length). Thus, the sum $D^{(t)}$ must have a (non-negative) lower bound.

Since $D^{(t)}$ is discrete, this implies, that there is some point in time where no more progress will be made, i. e. $D^{(t)} = D^{(t-1)}$. In this iteration, we know that no node distance changed. It follows, that no node was set to dirty (1b) and moreover, all previously dirty nodes are now clean. Consequently, in step 2, the algoritm does not continue. □

At the end of the algorithm, all nodes are clean and have valid and consistent rounting information.

2.6 Path Finding

The path finding algorithm is organized such that it can be executed step by step. Each iteration relies on up-to-date routing information (Section 2.5). The state is maintained in a path finding context (`path_finding_context_t`). Advancing the state is equivalent to finding the next point on the shortest path to the desired destination (the home position).

The context can be seen as a tuple $(\mathbf{p}, (i, c_i), (z, c_z))$. (i, c_i) is the current position on the return path, $\mathbf{p}$ is the corresponding absolute position and (z, c_z) is a checkpoint. Usually, the state is initialized at the graph head ($\mathbf{p} = \mathbf{h}$), and such that $(i, c_i) = (z, c_z)$.

1. If we're not at the checkpoint, i. e. $(i, c_i) \neq (z, c_z)$, go to step 5.
2. Out of all the nodes that lie on the current checkpoint, select the one that promises the shortest distance to home. Follow the action that it recommends. If this results in a line switch, update $\mathbf{p}$. In some cases (particularly the first iteration), the checkpoint may not lie on a node. In this case, no line switch is done.
3. Prefetch the path in the recommended direction, until a node is encountered. In case the checkpoint was not at a node, both directions (forward and backward) must be considered, and the one with the shorter distance-to-home is taken. The position of the encountered node becomes the new checkpoint (z, c_z).
4. If the line switch already resulted in a position update, skip step 5.
5. If $i \neq z$ or $c_i \neq c_z$, walk on the flight path in the direction of the checkpoint (e. g. if $i < z$, walk forward). Update $\mathbf{p}$ accordingly.

2.7 Memory Management

After prolonged operation of the vehicle, the flight graph will eventually reach its memory limit. A trivial solution would be to just stop recording. However, this does not seem satisfying.

In real-world use cases, by the time the graph buffer becomes full, it is likely that some of the flight graph is no longer required at high precision or at all. This conjecture stems from the observation that users usually cannot control the vehicle accurately when it is far away from them, so they tend to maintain a larger margin to obstacles. Moreover, the main purpose of storing the graph is to find a path to a certain destination. All parts of the graph, which do *not* lie on the shortest path to this destination, can be considered non-essential.

Based on these assumptions, two heuristics were implemented that are invoked in an interleaving manner. The overall memory management operation is shown below.

p is the **memory pressure**, starting at 1. It is incremented every other time the graph becomes full.

1. Record flight with $p = 1$ until graph becomes full.
2. Increment pressure $p \leftarrow p + 1$ and consolidate entire graph to apply the new margin derived from p (Section 2.7.1).
3. Continue to record flight with increased pressure until graph becomes full again.
4. Remove non-essential parts by rewriting the graph (Section 2.7.2).
5. Continue to record flight until graph becomes full again.
6. Go to step 2.

In the actual implementation, the memory pressure is tracked by the variable `memory_pressure` and p is given by `(memory_pressure»1) + 1`.

2.7.1 Adaptive Precision

In Section 2.4, the term "close" was used without proper definition. We now define "close" to be within some margin ε. The margin ε at some point $\mathbf{p}$ is a function of that point and the memory pressure p:

$$\varepsilon(\mathbf{p}, p) := \max\Big\{ \overbrace{\left\lfloor \frac{p-1}{2} \right\rfloor + 1}^{\text{Regime 1}},\ \overbrace{p \cdot \log_2(\underbrace{\|\mathbf{h} - \mathbf{p}\|}_{\text{distance to home}} / \lambda)}^{\text{Regime 2}} \Big\}$$

The margin function was designed to satisfy the following requirements:

Regime 1 Within some range λ of the home position (e. g. $\lambda = 64$ meters), it shall be attempted to use maximum precision, even after a few increments of memory pressure. If the entire flight graph is within this range and contains no nodes, the other methods will fail to yield any memory. Therefore, we are willing to relax this requirement in extreme cases. We do this every fourth time the memory pressure increases.

Regime 2 At further distances, the precision shall decrease with the distance to home. A proportional decrease would reflect the ability of the user to control the vehicle at sight. However, other modes of control are possible, therefore a proportional decrease would be too drastic. Furthermore, the slope of the precision decrease should increase with memory pressure.

Efficiency During a consolidation run, the ε-function is evaluated many times, so it should be efficiently computable.

In the actual implementation, the function is slightly modified (denoted $\hat{\varepsilon}$) in order to improve efficiency. It's output is shown in Figure 2.7. Conceptually, it is the same function. In mathematical notation however, it looks slightly more convoluted:

$$\hat{\varepsilon}(\mathbf{p}, p) := \max\Big\{\overbrace{\Big\lfloor \frac{p-1}{2} \Big\rfloor + 1}^{\text{Regime 1}},\ \overbrace{p \cdot \Big\lfloor \Big(\max\Big\{0, \underbrace{\Big\lfloor \log_2 \overbrace{\|\mathbf{h}-\mathbf{p}\|_2^2}^{\text{squared distance to home}} \Big\rfloor}_{\text{proportional to number of bits in distance}} \Big\} - \lambda' \Big)/2 \Big\rfloor}^{\text{Regime 2}} \Big\}$$

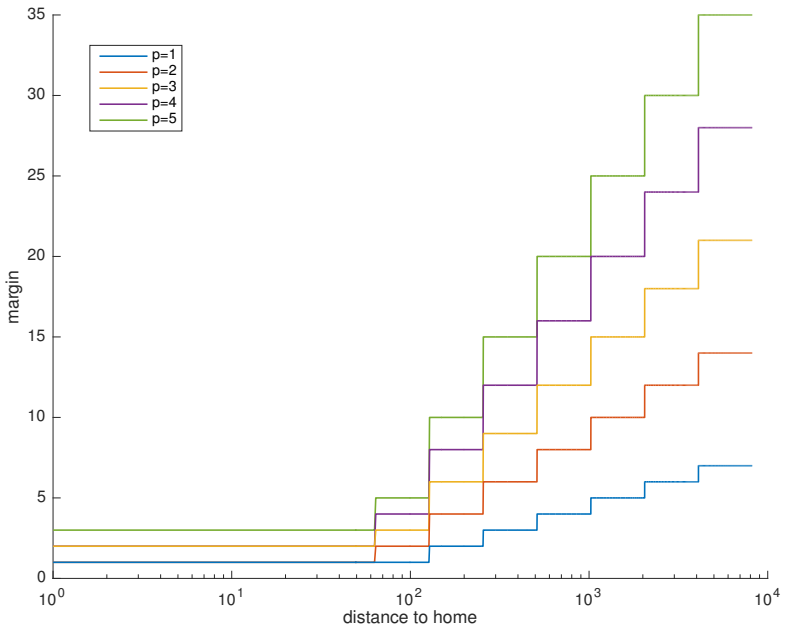

Figure 2.7. Margin $\hat{\varepsilon}(\mathbf{p}, p)$ in relation to $\|\mathbf{h}-\mathbf{p}\|$ for some values of p

Once the memory pressure has increased, the consolidation algorithm (Section 2.4) is rerun across the entire graph. Since the precision is now lower, the consolidation algorithm will tend to find more lines and redundancies and thus result in reduced memory usage. The yield of this consolidation depends on the path. If no memory can be freed, the memory pressure continues to increase and will at some point neccessarily result in a successful compression.

This has not been pushed to the most extreme cases, where technical issues would arise. In particular, intersections where both lines have a distance of more than 15 units along any axis cannot be stored in the node structure.

2.7.2 Rewrite Pass

The rewrite pass (`rewrite_graph`) reorganizes the entire graph, so that any non-essential parts (i. e. parts that do not lie on the shortest path to home) are removed. After this rewrite, the graph consists of just one single path that leads to the home position and contains no nodes (Figure 2.8).

For the implementation, it was important that there would be no notable memory overhead for this operation. The entire graph rewrite happens in-situ in the graph buffer and is supported only by a small cache (e. g. 18 bytes) on the stack. The

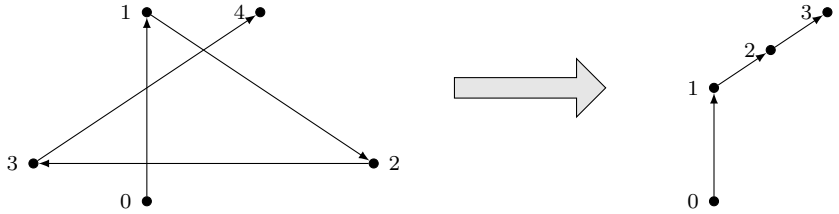

Figure 2.8. Graph rewrite operation: only the shortest path to home is retained.

Figure 2.9. Graph buffer layout during rewrite pass

path is rebuilt in a temporary area, the **rewrite area**, which is located in the graph buffer between the delta element list and the node list (Figure 2.9).

The algorithm is implemented in the function `rewrite_graph` and is outlined here:

1. Start at the graph head to initialize a path finding context (Section 2.6).
2. Request the next position on the return path, and append the negative delta to the rewrite cache (except if the delta is $(0, 0, 0)$).
3. At the most recent point on the path, there may be several directions, that the path finding algorithm did *not* choose. For instance, if the path finding context was at a node and the return path continues in the forward direction (direction of increasing indices) on line 1 (the first member of the node), the backward direction from line 1 and both directions on line 2 must have been rejected. For these intervals, we know for certain that they will not be part of the shortest path to home. Therefore, mark all rejected intervals as obsolete, that is, replace the delta items with the obsolete-placeholder (`0xC000`) until a node or another obsolete item is encountered. The delta item that is currently referenced by the path finding context must be retained.
4. If the path finding context was at a node, mark this node as obsolete as well.
5. If in step 2, the destination was reached or the cache write operation failed, flush the cache to the rewrite area:
   (a) If there is enough space in front of the rewrite area to flush the entire rewrite cache, go to step 5c. Otherwise, step 5b will create the neccessary space.
   (b) Shift the remaining delta items of the regular path to the beginning of the graph buffer, so that the obsolete items are overwritten. Likewise, shift the remaining nodes to the end of the graph buffer, so that the obsolete nodes are overwritten. Also, shift the rewrite area so that its end remains at the beginning of the node list. The shift operations are illustrated in Figure 2.10.
   (c) Dequeue the delta elements from the beginning of the rewrite cache and copy them to the front of the rewrite area.

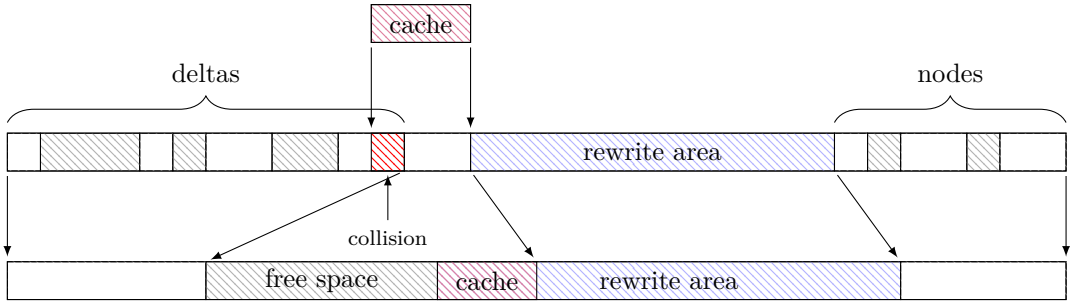

Figure 2.10. This figure shows a situation where there is not enough space to flush the rewrite cache to the rewrite area. The buffer contents are shifted to collect the freed space of obsolete delta items and nodes (shown in gray).

6. If the cache-write failed previously, it should be empty now. Retry and then go to step 2.
7. If the path finding algorithm did not reach the destination yet, go to step 2.
8. To complete the rewrite, shift the rebuilt path to the beginning of the graph buffer and delete all remaining nodes.

By the time the cache must be flushed, it is guaranteed that enough space was freed to make room for the cache contents. This claim was not investigated in detail, but in essence, the reasoning goes like this: The path finding algorithm does not emit more positions than the number of lines it visits, unless there are nodes on a line. In this case however, the nodes are deleted, yielding even more space than if a delta item was deleted.

Since the free space may not become available immediately (e. g. in the first iteration) and since the path finding algorithm may change in the future, it is still safer to maintain a small rewrite cache, rather than to write directly to the rewrite area.

Chapter 3

Evaluation

3.1 Computational Complexity

In this section, we derive the computational complexities of the main algorithms. These runtimes algorithms are then put into context of the overall system operation. We consider the length of the graph buffer unbounded in this section.

3.1.1 Consolidation

All three passes of the graph consolidation (Section 2.4) operate on the unconsolidated area at the end of the graph. Let n_Δ be the number of consolidated delta items and m_Δ the initial number of unconsolidated delta items.

For the **line detection pass**, for each line, it is first attempted to aggregate 2 positions, then 3 positions, and so on, until the resulting line would no longer be valid. For each candidate length i, there are $i-1$ point-to-line checks. Therefore, for each aggeregated line of length k, $\sum_{i=1}^{k} i = \frac{(k+1)k}{2}$ point-to-line checks are performed. Let the line detection result in l lines with the lengths $k_1, \ldots, k_l$. Note that $k_1 + \cdots + k_l = m_\Delta$, and every line has positive length. The resulting number of point-to-line checks is:

$$N = \sum_{i=1}^{l} \left(\frac{(k_i+1)k_i}{2} \right) \tag{3.1}$$

$$= \frac{1}{2} \sum_{i=1}^{l} (k_i+1)k_i \tag{3.2}$$

$$= \frac{1}{2} \left(\sum_{i=1}^{l} k_i^2 + \sum_{i=1}^{l} k_i \right) \tag{3.3}$$

$$\leq \frac{1}{2} \left(\left(\sum_{i=1}^{l} k_i \right)^2 + \sum_{i=1}^{l} k_i \right) \tag{3.4}$$

$$= \frac{1}{2} \left(m_\Delta^2 + m_\Delta \right) \tag{3.5}$$

In step 3.4, we observe that all k_i are positive and use that $a^2 + b^2 \leq (a+b)^2$ holds for positive a and b. The equality in particular holds for $l = 1$. In step 3.5, we use the initial constraint $\sum_{i=1}^{l} k_i = m_\Delta$. Point-to-line checks have constant runtime, thus the line detection runs in $\mathcal{O}(m_\Delta^2)$.

For the **redundancy removal**, each unconsolidated position is checked against every preceeding line (both consolidated and unconsolidated lines). In the worst case, there are still m_Δ unconsolidated positions, if the preceding line detection did not result in any compression. The resulting number of point-to-line checks is therefore

$$\sum_{i=0}^{m_\Delta} (n_\Delta + i) = m_\Delta \cdot \left(n_\Delta + \frac{m_\Delta}{2} \right)$$

The runtime of the redundancy removal pass is thus in $\mathcal{O}(m_\Delta n_\Delta + m_\Delta^2)$.

The **node detection** can be analyzed analogously to the redundancy removal, except that line-to-line checks are used instead of point-to-line checks: Each unconsolidated line is checked against every preceeding line. Furhermore, we must consider the complexity of adding a node: for each new node, it's closeness to all existing nodes is verified. If there are n_N nodes initially, and m_N new nodes are added, this results in $m_N(n_N + m_N/2)$ closeness checks. The resulting runtime of the node detection pass is therefore $\mathcal{O}(m_\Delta n_\Delta + m_\Delta^2 + m_N n_N + m_N^2)$.

To obtain a usable overall estimate, we make the following observations, without further proofs:

(i) All three passes run in $\mathcal{O}(m_\Delta n_\Delta + m_\Delta^2 + m_N n_N + m_N^2)$.

(ii) The number of added nodes m_N is at most the number of line-to-line checks: $m_N \leq m_\Delta n_\Delta + \frac{m_\Delta^2}{2}$

(iii) Of the existing lines, each line i yielded at most $(i-1)$ nodes, therefore: $n_N \leq \sum_{i=1}^{n_\Delta} (i-1) \leq n_\Delta^2$

(iv) The number of unconsolidated positions is usually bounded by a constant (`MAX_CONSOLIDATION_DEBT`).

(v) The total number of delta elements and nodes is bounded by the graph size n (`GRAPH_LENGTH`): $n_\Delta \leq n,\ m_\Delta \leq n,\ n_N \leq n,\ m_N \leq n$

Based on this, we give three bounds for the overall runtime:

- **Naïve estimation**: Substituting the bounds from (ii) and (iii) in (i), we obtain the estimation

$$\mathcal{O}(m_\Delta n_\Delta^3 + m_\Delta^2 n_\Delta^2 + m_\Delta^3 n_\Delta + m_\Delta^4) \tag{3.6}$$

- **Constant** m_Δ: For a bounded unconsolidated area size, from the naïve bound we obtain

$$\mathcal{O}(n_\Delta^3) \tag{3.7}$$

- **With respect to graph size**: Using (v), we can express the runtime from (i) with respect to the buffer size and obtain

$$\mathcal{O}(n^2) \tag{3.8}$$

3.1.2 Routing Data Synthesis

The routing data synthesis (Section 2.5) is comparable to Bellman-Ford [2]. Its main structure consists of two nested loops: The inner loop iterates over each dirty node and potentially leads to new dirty nodes. The outer loop terminates when there are no more dirty nodes at all.

The outer loop can be analyzed analogously to the Bellman-Ford algorithm: Since no two nodes can lie further appart than $n_N - 1$ edges, information about the shortest path takes at most this many iterations to propagate through the entire path. The last iteration results in no updates and therefore terminates the loop.

In the inner loop, at each dirty node, up to four adjacent intervals are measured (`walk_to_node`). An interval is a sequence of consecutive delta elements, and is bounded by jump elements or nodes. Any given interval is thus measured at most twice (if it is bounded by two dirty nodes). Measuring an interval consists of scanning the node list (to find the interval bounds) and walking along the interval. Measuring an interval of length k therefore runs in $\mathcal{O}(k + n_N)$ and measuring all (at most $4n_N$) intervals twice, runs in $\mathcal{O}(2(k_1 + k_2 + \ldots) + 8n_\Delta^2) = \mathcal{O}(n_\Delta + n_N^2)$. In addition to that, for each dirty node, up to four nodes are updated. Updating a node consists of scanning the node list and runs in $\mathcal{O}(n_N) \subseteq \mathcal{O}(n_\Delta + n_N^2)$.

Combining the results and applying the observation (v) from Section 3.1.1, we obtain an overall complexity of

$$\mathcal{O}(n_N n_\Delta + n_N^3) \tag{3.9}$$
$$= \mathcal{O}(n^3) \tag{3.10}$$

3.1.3 Path Finding

Assuming that all routing data is up to date, the reasoning goes similar to Section 3.1.2. Path finding (Section 2.6) consists of walking along the shortest path (which is at most the entire graph) and special handling at each encountered node (which is at most every node). In particular, at each node, the node list is scanned and up to two intervals are measured. As discussed in Section 3.1.2, this runs in $\mathcal{O}(n_\Delta + n_N^2)$. Dumping the entire shortest path has therefore the same complexity as generating the routing data, i. e. $\mathcal{O}(n_N n_\Delta + n_N^3) = \mathcal{O}(n^3)$.

3.1.4 Rewrite Pass

In the beginning of the graph rewrite (Section 2.7.2), the graph is consolidatated ($\mathcal{O}(n^2)$) and the routing data is generated ($\mathcal{O}(n^3)$). After that, the entire return path is fetched ($\mathcal{O}(n^3)$) and written to the buffer.

Over the course of the rewrite, the data in the graph buffer is shifted multiple times. The worst possible shift pattern looks like this: In each iteration, only the first item is removed and the entire remainder is shifted. Such a shift pattern has a complexity of $\mathcal{O}(n^2)$. This pattern is considered for both the delta list and the node list. Similarly, shifting the growing rewrite area over the course of the rewrite has the same complexity. In addition to that, each shift pass requires a scan over the entire graph. There are at most n rewrite iterations, thus the scans run in $\mathcal{O}(n^2)$ as well.

The rewrite pass therefore runs in $\mathcal{O}(n^3)$.

3.1.5 Overall Runtimes

From the runtimes of the separate algorithms, we can determine the runtimes of the operations that are relevant for the overall system operation.

Position Update Usually, only a single delta item is appended to the flight path. However, sometimes, the graph is consolidated (with constant m_Δ). The complexity is therefore $\mathcal{O}(n_\Delta^3)$.

Graph Overflow When the graph becomes full, either the entire graph is consolidated or a rewrite is conducted. The latter has the dominant runtime of $\mathcal{O}(n^3)$.

Path Finding Initialization The path finding initialization is required for instance once the RTL mode is invoked (see Section 2.1.1). It consists of calculating the routing information and therefore runs in $\mathcal{O}(n^3)$.

Path Finding Step At each waypoint, an iteration of the path finding algorithm is done. Since the entire algorithm runs in $\mathcal{O}(n^3)$ and there are at most n steps, the amortized runtime of a path finding step is $\mathcal{O}(n^2)$.

It shall be reiterated here, that low computational complexity was not an objective of this thesis (as noted in Chapter 1). Instead, the objective for the optimizations was the size of the graph buffer. Since in practice, all algorithms have a complexity bounded by the buffer size, a small buffer automatically leads to a predictable maximum runtime.

3.2 Memory Usage

The utilization of the memory available to the graph buffer was measured as follows:

1. A path was created via mission planning in the drone commander App QGroundControl [3].
2. The parameters `GRAPH_LENGTH` and `MAX_CONSOLIDATION_DEBT` were both set to a large value, so that no optimizations would be applied and the unaltered (albeit rounded) positions would be retained.
3. The mission was flown virtually using SITL (software-in-the-loop) simulation.

4. The graph contents (i. e. the unaltered positions) were dumped into a file (e. g. `long_path1.path` in the `navigator_tests` folder).

5. The parameters were reset: `GRAPH_LENGTH = 256`, `MAX_CONSOLIDATION_DEBT = 64`. Note that the actual graph buffer size is 512 Bytes.

6. The dumped path was statically compiled into a unit test.

7. In the unit test, the path was loaded into the `Tracker`.

8. On every instance of graph buffer overflow, the graph buffer utilization was recorded before and after the compression action. Recall that the compression action is either an increase of memory pressure combined with consolidation of the whole graph, or a graph rewrite.

Note that the graph buffer utilization is equal on all architectures, as long as the compiler yields the structure layouts described in Chapter 2. Appart from the graph buffer, the implementation was determined to have a constant overhead of 640 Bytes.

The described measurements were conducted on two different paths. Figure 3.1 is mostly a network of routes within the vicinity of the home position. Figure 3.2 contains a convoluted flight path near the home position, provoking many intersections, and a path that reaches out for more than 2 kilometers (using unit size 1). The results are shown in Figure 3.3 and Figure 3.4. The following observations can be made:

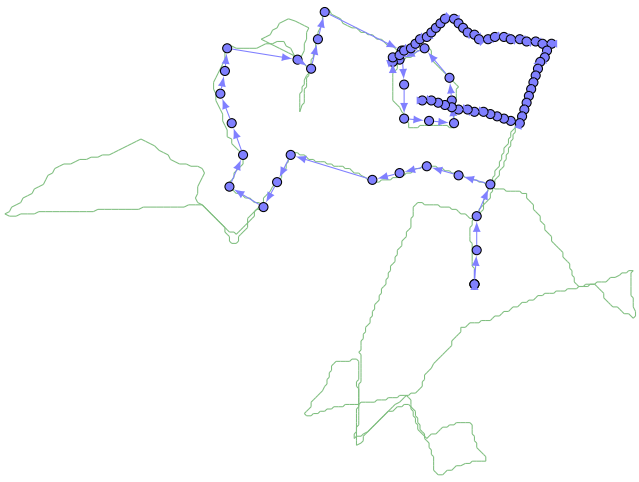

Figure 3.1. Path A used for the memory and CPU tests. The blue path depicts the path as it is stored in the buffer at the end of recording.

- On path A, the increase of memory pressure (i. e. the consolidation) is quite effective, in that it compresses the graph to half of its original size.

- The first consolidation of path B, where the ratio of nodes to delta elements is higher, is not as effective as on path A.

- In both consolidation passes on path B, there seem to be difficulties in reducing the number of nodes.
- The node storage uses a significant portion of the graph buffer. This can be explained by the large memory footprint of a single node (12 Bytes).
- The rewrite is very effective. Note that a single node remains in the buffer after a rewrite. This is a special node that represents the home position.

It should be noted that these handcrafted paths do not neccessarily reflect a real world scenario. Both memory usage and CPU usage (Section 3.3) depend significantly on the flight pattern (i. e. how often intersections occur). To obtain a statistically relevant estimate, it is therefore neccessary to use sufficiently large amounts of logs that were recorded during actual flight. However, the results shown here are valuable as a rough estimation of performance. As such, they indicate that the memory usage should be small enough for resource constrained hardware, even for larger paths.

3.3 CPU Load

To measure the CPU usage of the graph implementation, a setup similar to the one in Section 3.2 was used. In particular, the same paths were used. The main difference was, that the unit tests were executed on the PX4FMU-v4, a flight controller for small drones, powered by a Cortex-M4F processor running at 168 MHz [4]. For each run, the time spent for each compression action was recorded. The results are shown in Table 3.1. Note that the compression actions have the most critical runtimes of all algorithms, since they invoke all other algorithms.

Path, Buffer Size	Consolidation 1	Rewrite	Consolidation 2
Path A, 512 Bytes	85ms	44ms	—
Path B, 512 Bytes	102ms	56ms	42ms
Path B, 1024 Bytes	296ms	97ms	—

Table 3.1. Runtimes of critical algorithms. The second consolidation was only triggered for in one scenario.

As in Section 3.2, these results do not accurately reflect real world scenarios. However, they indicate that the runtimes are still acceptable for the chosen paths, the bottleneck being the first memory pressure increment. For larger paths however, it may be neccessary to put some work into runtime optimizations.

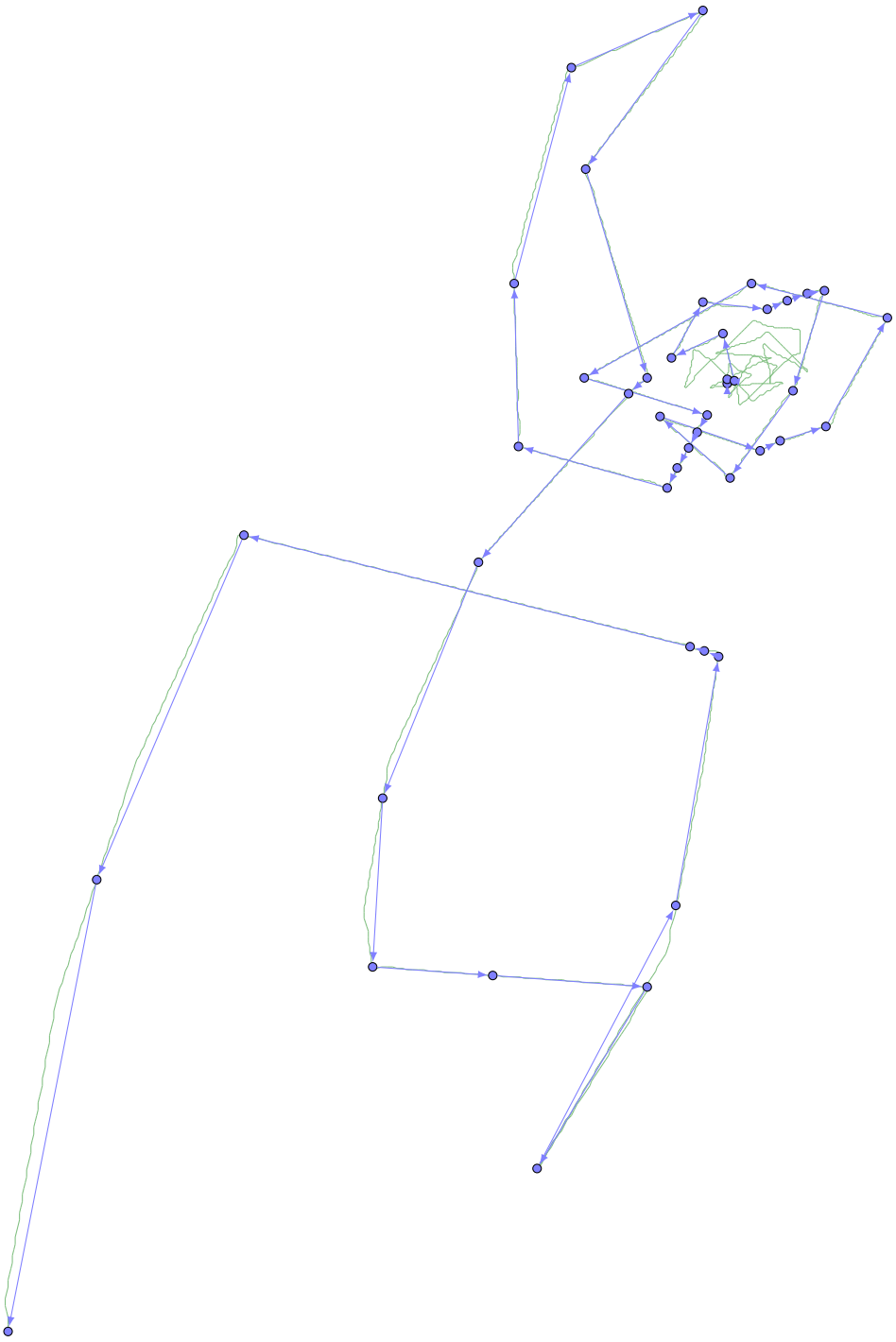

Figure 3.2. Part of path B used for the memory and CPU tests (the path is not shown completely)

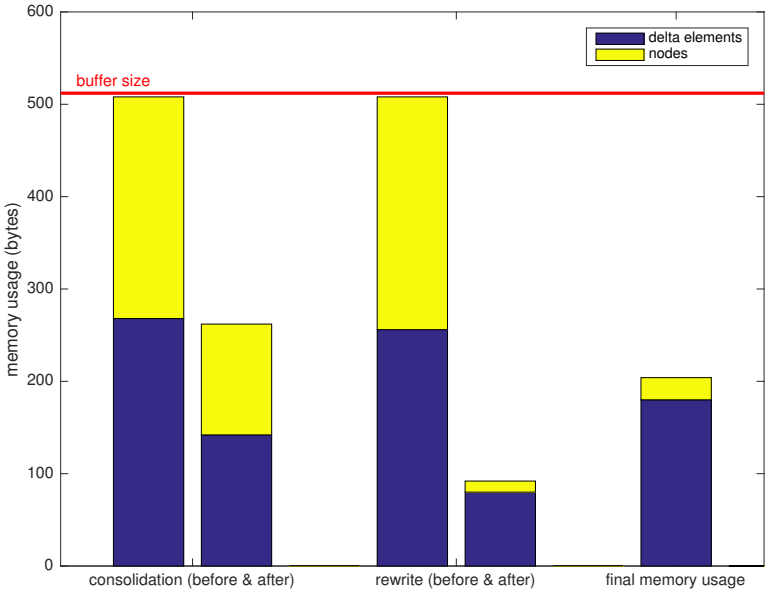

Figure 3.3. Graph buffer utilization during recording of path A

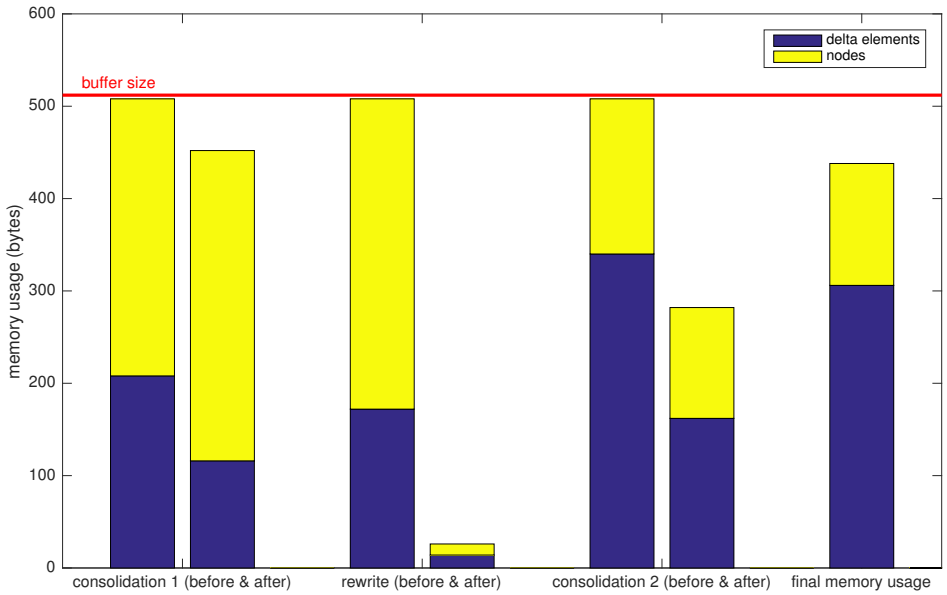

Figure 3.4. Graph buffer utilization during recording of path B

Chapter 4

Conclusion

An implementation was presented which stores the covered flight path in a memory efficient format. The implementation compresses straight lines and removes redundancies and adapts the recording precision heuristically. Whenever the buffer becomes full, the graph is either rewritten or the precision is reduced.

To evaluate memory and CPU performance of the implementation, two hand crafted paths were loaded into the graph buffer, while monitoring memory utilization and the runtime of critical algorithms. The obtained results indicate that the current implementation should be suitable for real world usage, especially in terms of memory efficiency. However CPU usage may still need to be optimized.

4.1 Future Work

Even though the implementation was verified thoroughly, flight tests must still be conducted to test the implementation in a real world environment. The work can then be adapted for merger into the main repository of the PX4 autopilot.

Further work could be done in improving path compression ratio, by augmenting or replacing the line detection by a more advanced compression method. However, given the test results, this does not seem to be neccessary, as the current implementation already enables a decently economic memory usage. If evaluating the implementation on real flight data reveals a significantly higher average node count than anticipated, a more compact node storage should be implemented. Also, as mentioned before, optimizations regarding CPU efficiency may still be required.

Bibliography

[1] Drone software solution - px4 pro open source autopilot. http://px4.io/technology/px4-software-stack/, accessed on 2016-09-02.

[2] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[3] Qgc - qgroundcontrol - drone control. http://qgroundcontrol.com/, accessed on 2016-09-02.

[4] Px4fmu autopilot / flight management unit - pixhawk flight controller hardware project. https://pixhawk.org/modules/px4fmu, accessed on 2016-09-02.

Eigenständigkeitserklärung

Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben. Davon ausgeschlossen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuer und die Betreuerinnen der Arbeit.

Titel der Arbeit: Graph Based Navigation on Resource Constrained Systems

Verfasst von: Samuel Sadok

Ich bestätige mit meiner Unterschrift

- Ich habe keine im Merkblatt "Zitier-Knigge" beschriebene Form des Plagiats begangen.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu dokumentiert.
- Ich habe keine Daten manipuliert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Plagiate überprüft werden kann.

Zürich, 2. September 2016