Planning semi-autonomous drone photo missions in Google Earth

Fredrik Nilsson
Abstract
This report covers an investigation of the methods and algorithms required to plan and perform semi-autonomous photo missions on Apple iPad devices using data exported from Google Earth. Flight time was to be minimized, taking wind velocity and aircraft performance into account. Google Earth was used both to define what photos to take, and to define the allowable mission area for the aircraft. A benchmark mission was created containing 30 photo operations in a 250 by 500 m area containing several no-fly-areas. The report demonstrates that photos taken in Google Earth can be reproduced in reality with good visual resemblance. High quality paths between all possible photo operation pairs in the benchmark mission could be found in seconds using the Theta* algorithm in a 3D grid representation with six-edge connectivity (Up, Down, North, South, East, West). Smoothing the path in a post-processing step was shown to further increase the quality of the path at a very low computational cost. An optimal route between the operations in the benchmark mission, using the paths found by Theta*, could be found in less than half a minute using a Branch-and-Bound algorithm. It was however also found that prematurely terminating the algorithm after five seconds yielded a route that was close enough to optimal not to warrant running the algorithm to completion.

Keywords: Google Earth, Robotics, Pathfinding, A*, Theta*, Path smoothing, Traveling Salesman Problem, Branch-and-Bound, Swift, C++
# Table of Contents

Abstract.............................................................................................................. iii

Terminology...................................................................................................... vii

1 Introduction............................................................................................... 1
  1.1 Background and problem motivation...................................................... 1
  1.2 Overall aim............................................................................................ 1
  1.3 Scope........................................................................................................ 1
  1.4 Concrete and verifiable goals.................................................................. 2
    1.4.1 Calculating photo parameters from exported KML-data............... 3
    1.4.2 Determining feasibility of implementing algorithms in Swift......... 3
    1.4.3 Selecting and implementing path finding algorithm...................... 3
    1.4.4 Selecting and implementing route optimization algorithm.......... 3
  1.5 Outline..................................................................................................... 4
  1.6 Contributions.......................................................................................... 4

2 Theory........................................................................................................ 6
  2.1 KML....................................................................................................... 6
  2.2 Google Earth........................................................................................ 7
  2.3 DJI SDK................................................................................................. 7
  2.4 IOS Development with Xcode and Swift............................................. 8
  2.5 Path-finding........................................................................................... 8
    2.5.1 World representation...................................................................... 8
    2.5.2 Uniform Cost Search and A*......................................................... 10
    2.5.3 Non-optimality of grid constrained paths...................................... 11
    2.5.4 Path smoothing............................................................................... 12
    2.5.5 Theta*............................................................................................ 12
  2.6 Route optimization and Traveling Salesman Problem.......................... 13
    2.6.1 Nearest Neighbor Algorithm......................................................... 13
    2.6.2 Branch-and-bound algorithms...................................................... 14

3 Methodology............................................................................................... 15
  3.1 Establishing project scope and requirements....................................... 15
  3.2 Measuring algorithm runtime............................................................... 15
  3.3 Calculating photo parameters from exported KML-data........................ 15
  3.4 Determining feasibility of implementing algorithms in Swift.............. 16
  3.5 Selecting and implementing path finding algorithm............................ 16
  3.6 Selecting and implementing route optimization algorithm.................. 17
  3.7 Development environment..................................................................... 17

4 Implementation.......................................................................................... 18
  4.1 KML Mission Data................................................................................ 18
  4.2 Calculating photo parameters from KML-data..................................... 18
  4.3 Swift/C++ interoperability................................................................. 18
  4.4 Determining feasibility of implementing algorithms in Swift............... 19
    4.4.1 Path finding search problem model............................................ 19
Appendix A: Mission Definition in KML

References

5 Results

6 Discussion

References

Appendix A: Mission Definition in KML
# Terminology

## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT</td>
<td>Altitude</td>
</tr>
<tr>
<td>aTSP</td>
<td>Asymmetric Traveling Salesman Problem</td>
</tr>
<tr>
<td>BB</td>
<td>Branch-and-Bound</td>
</tr>
<tr>
<td>DFBB</td>
<td>Depth-first Branch-and-Bound</td>
</tr>
<tr>
<td>DJI</td>
<td>A multi-rotor aircraft manufacturer</td>
</tr>
<tr>
<td>FOV</td>
<td>Field of View</td>
</tr>
<tr>
<td>HDG</td>
<td>Heading</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>JPS</td>
<td>Jump Point Search (Modification to the A* algorithm – a search algorithm used for path finding in this project)</td>
</tr>
<tr>
<td>KML</td>
<td>Keyhole Markup Language</td>
</tr>
<tr>
<td>LOS</td>
<td>Line-of-sight</td>
</tr>
<tr>
<td>NN</td>
<td>Nearest Neighbor</td>
</tr>
<tr>
<td>OGC</td>
<td>Open Geospatial Consortium</td>
</tr>
<tr>
<td>PS</td>
<td>Path Smoothing</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>sTSP</td>
<td>Symmetric Traveling Salesman Problem</td>
</tr>
<tr>
<td>TSP</td>
<td>Traveling Salesman Problem</td>
</tr>
<tr>
<td>UCS</td>
<td>Uniform Cost Search</td>
</tr>
</tbody>
</table>

## Mathematical notation

\[ O(n) \quad \text{Asymptotic behavior with respect to } n \]
\[ g(n) \quad \text{Accumulated path cost to node } n \]
$h(n)$  Heuristic path cost from $n$ to $n_{goal}$

$f(n)$  $g(n) + h(n)$
1 Introduction

1.1 Background and problem motivation

Arrowcam AB is a company performing aerial photography using drones. Typical customers include real estate agents, construction companies, hotels and golf courses. Maps and existing photos are used to discuss the composition of the pictures to be taken with the customer. During the flight, the pilot/photographer has to spend significant time setting up the drone/camera position and orientation for each individual photo being taken as agreed with the customer. When the flight is complete, there is still a risk of the pictures not matching the customer's expectation, requiring supplementary flights to deliver the required photos. By planning photo jobs using an earth browser, such as Google Earth, it may become easier to elicit what the customer expects from each picture in terms of composition. The pictures defined in the earth browser could then be translated to the position and orientation of the camera necessary to take each picture. Position data could in turn be used to create an efficient routing between each individual photo operation. Finally, the drone could automatically fly the route, and take the pictures, resulting in a significantly shorter flight times compared to manually flying and setting up for each picture.

1.2 Overall aim

A prototype tool will be created enabling the user to use data exported from Google Earth to automatically create a near optimal routing between photo operations, taking into account user defined areas where the drone may and may not fly, including minimum altitudes over certain areas. Wind shall also be taken into account when generating the routing. The tool will be usable with the company's currently available hardware, which contains of Mac computers, iPads and DJI drones (in particular, the DJI Inspire 1). See Appendix I: Existing Software and Hardware at Arrowcam AB for a full listing.

To create the prototype tool, suitable algorithms must be found for finding the required camera position and orientation for photos, finding routes between photo operations, and to find a near optimal sequence of photo operations. The work covered in this report aims to evaluate and decide on algorithms to be used in the prototype tool mentioned in the earlier paragraph.

1.3 Scope

This report will cover the selection and evaluation of algorithms required to fulfill the tasks described in 1.2. The final prototype tool itself will not be presented or tested.
The project will not be a comprehensive study in search and optimization algorithms, but rather to develop one possible combination of algorithm implementations that satisfies the performance goals given in 1.4.

In the course of developing path finding algorithms, classes representing geometry and performing topological operations will be required. In the end, an application specific implementation was made in order not to make topological operations a bottleneck. Details of this implementation will be completely omitted in this report as it is deemed this will excessively expand the scope.

Development will be limited to iOS for use on iPad devices. This limitation is due to a strong desire from Arrowcam AB to use hardware they already own and are familiar with.

Simplifications will be made to the flight dynamics used in the path finding and optimization algorithms. These simplifications include:

- Vertical and horizontal velocity components of the aircraft are independent ie:  \( \left| \vec{v}_{\text{max}} \right| = \left| \vec{v}_{\text{max} \, XY} \right| + \left| \vec{v}_{\text{max} \, Z} \right| \)
- For comparison with manually calculated flight paths, the performance model used allows horizontal speed of 10 m/s and vertical speed of 5 m/s
- Only a single global wind velocity is used
- Operations are limited between latitudes N87/S87
- Operations across the E/W180 boundary are not supported

In the prototype operation, only single photos will be supported. However, Arrowcam AB requested the algorithms to support operations where the start position and end position of an operation do not coincide. Thus to fully enumerate all paths between operation pairs, paths will be calculated in both directions for each operation pair.

1.4 Concrete and verifiable goals

In achieving the overall aim of section 1.2, a number of subproblems have to be solved. A sample mission was created in cooperation with Arrowcam AB to create a realistic but challenging benchmark. The protocol used to define missions is discussed in Appendix A: Mission Definition in KML and the sample mission itself is described in more detail in Appendix B: Sample Mission.

Subproblems with completion criteria follow:
1.4.1 **Calculating photo parameters from exported KML-data**

A formula for determining aircraft/camera position and orientation will have to be developed, as the KML file from Google Earth does not provide this. This is further discussed in Appendix A: Mission Definition in KML.

**Completion criteria:** Three photos will be taken using actual aircraft, from data calculated with the formula, from photos defined in Google Earth. The actual photos will be compared to the photos defined in Google Earth, and should have good visual resemblance (“good visual resemblance” being subject to Arrowcam AB’s judgement).

1.4.2 **Determining feasibility of implementing algorithms in Swift**

Native iOS development is possible in Swift and Objective-C, with bridging possible to C/++ (see section 2.4). Is Swift fast enough to implement algorithms required by later subproblems, with a reasonable chance of meeting performance requirements for said subproblems (below), or is resorting to C++ required?

**Completion criteria:** Select a search algorithm for comparison. Implement algorithm in both Swift and C++. Measure performance, and determine feasibility of using Swift for performance critical algorithms considering both performance and code maintainability.

1.4.3 **Selecting and implementing path finding algorithm**

Select and implement path finding algorithm that is fast and near optimal.

**Completion criteria:** Paths between all operation pairs in sample mission should be found in less than 30 seconds on an iPad Air 32 GB (model MD792FD/B). 1% of the operation pairs should be randomly selected to be compared against manually calculated optimal paths. The mean path finding algorithm results should be no longer than 110% of manually calculated paths. Paths should also not penetrate forbidden areas.

1.4.4 **Selecting and implementing route optimization algorithm**

Select and implement an algorithm optimizing the order in which operations are performed, with path between the operations as found by the path finding algorithm in 1.4.3, taking local wind velocity into account.

**Completion criteria:** The calculated route must be no longer than the route calculated by the Nearest Neighbor algorithm (see section 2.6.1) for an arbitrarily chosen starting point. The algorithm must run in less than 30 seconds.
1.5 Outline

Chapter 1 provides an introduction to the project. The various subproblems that have to be solved during the course of the project are presented, together with goal completion criteria.

Chapter 2 presents background information that will be helpful in understanding the implementations provided, and will also be used to motivate why certain decisions were made. It begins with an overview of the KML file format and its relation to Google Earth. An overview of the target platform is given by describing the Software Development Kit (SDK) provided by DJI (the aircraft manufacturer), followed by a description of the Integrated Development Environment (IDE) and the languages that will be used. An overview of the path finding problem is then given, followed by a few possible ways of solving it. Finally, the route optimization problem is presented, together with a method of solving it.

From Chapter 3 and onwards, the branching into the various subproblems start becoming noticeable. From this chapter onwards, it is not recommended to read the report chronologically, but rather chronologically, subproblem by subproblem, as results and conclusions from one subproblem in later chapters may be relevant to another subproblem in an earlier chapter.

Chapter 3 contains a description of how each subproblem will be solved, what algorithms were selected for evaluation, and what criteria will be used to evaluate them by (in addition to the goal completion criteria in 1.4).

Chapter 4 presents the implementations of solutions to the various subproblems. It also touches on how interoperability between Swift and C++ is accomplished. Source code is mostly deferred to the appendices, however, essential public interfaces are described by providing parts of header-files.

Chapter 5 contains the results of each subproblem in order in table and graph-form. Result images are also shown for Calculating photo parameters from exported KML-data.

Chapter 6 concludes the report by discussing the results in Chapter 5 subproblem by subproblem. The discussions motivate the choice in algorithm for solving each subproblem. Finally, a section on ethical considerations applied to the use of the final product that the work covered in this report will be used in.

1.6 Contributions

This report and all published code has been created in its entirety by Fredrik Nilsson (the author). Experimental results have been collected by the author. Rikard Tyllström from Arrowcam AB has assisted as aircraft operator to collect the results in 5.1: Calculating photo parameters from exported KML-data. Inventors of algorithms used are attributed as the algorithms are discussed in Chapter 2: Theory. Some of the images contained in the report are attributed to
Google Earth, and their data suppliers. This is indicated in or close to these images as they appear in the report.
2 Theory

2.1 KML

*Keyhole Markup Language* (KML) is an XML-language for geographic visualization. KML was developed by Google, and later adopted by the Open Geospatial Consortium (OGC), a global standards organization for geospatial data [1].

Camera views can be defined in KML directly by using the `<Camera>` element, or indirectly using the `<LookAt>` element. The `<Camera>` element specifies the position and the orientation of a virtual camera as illustrated by the following example:

```xml
<Camera>
    <longitude>12.764683816258813</longitude>
    <latitude>56.053165621029741</latitude>
    <altitude>127.73</altitude>
    <heading>270</heading>
    <tilt>45</tilt>
    <altitudeMode>absolute</altitudeMode>
</Camera>
```

The `<LookAt>` element, specifies a point for the camera too look at, together with the relative position of the camera given by vertical/azimuth bearing and the distance between the point and the camera. This is illustrated by the following listing:

```xml
<LookAt>
    <longitude>12.76339337010659</longitude>
    <latitude>56.05316561419531</latitude>
    <altitude>47</altitude>
    <heading>-90</heading>
    <tilt>45</tilt>
    <range>113.4469857719943</range>
    <altitudeMode>absolute</altitudeMode>
</LookAt>
```

Both of the above listings result in the same view: see Figure 1 below, in which camera field of view (FOV) also has been defined using the `<horizFov>` element to emulate a 12 mm lens.
Geometric features, such as points, polygons and lines can also be defined in KML. For an exhaustive listing of KML features, see OGC KML v2.2 [1].

2.2 Google Earth

Google Earth is an earth browser allowing the user to explore a 3D model of the earth, combining satellite imagery, aerial photography and geospatial data [2]. Google Earth can be used to define and annotate views, and points of interests (Placemarks), and save/load these to/from the KML-format [3].

2.3 DJI SDK

DJI is a company that manufactures aerial photography platforms (drones) [4]. Three categories of Software Development Kits (SDK’s) have been made public through the DJI Developer Program: Guidance SDK, for stand-alone flight guidance modules not currently integrated in any DJI aircraft [4][5], Onboard SDK for allowing a dedicated on board computer interact with the aircraft [6] and the Mobile SDK.

The Mobile SDK allows the developer to write an application for either the iOS 8.0+ or the Android 4.2.2+ platform [7], that can communicate with the drone either directly via USB or WiFi, or through the DJI Remote Controller, which uses the proprietary Lightbridge wireless link to relay data between the application and the drone [8]. Important functions that can be accessed in the drone flight controller includes flight control (attitude, position, velocity), waypoint sequencing, camera and gimbal control and live video feed.
2.4 IOS Development with Xcode and Swift

Xcode is Apple's Integrated Development Environment (IDE) that can be used for building applications for Apple's various operating systems [9]. Xcode features compilers for C++, Objective-C and Swift [10]. It has a built in unit testing framework (XCTest) and comes with various profiling tools (Instruments).

Swift is an open-source, general purpose, programming language [11] of which the development is led by Apple [12]. Swift can be used for developing for iOS and other Apple operating systems [11]. The language is advertised as being a modern replacement for C-based languages, with comparable performance. Swift also brings several safety features, such as overflow checking, automatic memory management (with manual memory management still being possible using structures in the “Unsafe”-family [13]), and preventing null-pointers to objects. Swift is interoperable with Objective-C [14]. As Objective-C has interoperability with C++ [15], interoperability between Swift and C++ is achievable through an Objective-C layer.

Swift's standard library includes a few data structures, such as Array, Dictionary and Set [13], but is significantly less comprehensive than the C++ Standard Template Library (STL). Notably for this project, Swift lacks an implementation of a priority queue, whereas the C++ STL provides one [16]. However, a vast amount (31'000+ in the package manager CocoaPods at the time of writing [17]) of community libraries for Swift exist. SwiftPriorityQueue by David Kopec being one of them, claims to be a classic binary heap implementation of a priority queue released under the MIT license [18].

For compiling, Xcode uses the LLVM compiler [19] (not an acronym [20]) as a backend to create optimized machine code from intermediate code generated by clang for C-based languages, or from the Swift compiler for Swift-code [21].

2.5 Path-finding

To find an efficient route between two points, in a world with obstructions, a search algorithm can be used. More generally, a search algorithm finds a solution to a problem, formulated as a sequence of actions required to reach a goal state from an initial state [22]. Such a solution is said to be optimal if it is the “cheapest” possible solution, given some cost function. If an algorithm is guaranteed to eventually find a solution, provided it exists, such an algorithm is said to be complete.

2.5.1 World representation

The search algorithms discussed here assume the world representation is discrete. Two discrete world representations will be presented below.

A grid based world representation divides the world into a grid in \( n \) dimensions [23]. A grid node can then be traversable or non-traversable by the search algorithm, given by some function. The memory requirement and the number of
nodes the search algorithm is required to explore will be largely dependent on the grid resolution, and may become unnecessarily large for worlds with sparse complexity as shown in Figure 2.

![Figure 2: 20x10 grid representing world with 19 vertices](image)

With a visibility graph-based world representation, nodes represent navigable points. In 2D, these points are typically vertices in polygons. This comes as a consequence of the triangle inequality (the shortest path between two points in 2-space separated by a polygon will be via one or more of its vertices). Nodes that have unobstructed line of sight between an other will have an edge connecting them [24]. See Figure 3 for an example resulting in significantly fewer nodes and edges than in Figure 2. On large visibility graphs however, the number of edges may become excessive, as they can be up to quadratic in the number of vertices [25].

![Figure 3: Visibility graph representation](image)

Extending the visibility graph representation to 3-space or higher is non-trivial, as visibility now extends from being between polygon lines to being between polyhedron planes [26]. Consider finding the shortest path from the blue point to the green point in Figure 4 below. The shortest path is no longer via the discrete vertices (marked as red points) of the mid cuboid, but via points on the continuous lines (marked in red) connecting them. This nature of 3-space (and higher) makes the entire family of general Euclidian shortest path problems intractable for larger instances. An easier approximation of the problem can be created by adding “artificial” vertices in the visibility graph nodes along the continuous lines of the obstacles.
2.5.2 Uniform Cost Search and A*

Uniform Cost Search (UCS) (logically equivalent to Dijkstra's algorithm [27]) searches for the optimal path from an initial node in a graph/tree to a goal node [22]. From the initial node, all reachable neighbors are added to a priority queue, for expansion. The priority queue is ordered by minimum cost, given by some cost function $f(n)$, (typically distance, in an Euclidean space path problem) to reach the candidate node, from the initial node. When all neighbors have been expanded, the next node is popped from the front of the priority queue, and the process is repeated, until the goal node has been found. UCS is guaranteed to be optimal and complete. An illustration of a UCS run is given in Figure 5.

Since the search is expanded omnidirectionally, in a uniform weight grid, both time and space complexity is $O(b^d)$, where $b$ is the branching factor, the number of neighbors to each node, and $d$ is the depth of the shallowest solution [22]. In Figure 5 $d$ is 12, and $b$ is upper bound by 8. Enlarging the area to be searched or increasing the resolution of the grid, increases the time and space complexity exponentially. Adding more dimensions (such as searching in 3-space instead of 2-space) increases the branching factor, since each node has more neighbors.

The $A^*$ algorithm [22] builds on the UCS algorithm by letting:
\[ g(n) = \text{cost}_\text{initial to candidate} \]
\[ h(n) = \text{estimated cost}_\text{candidate to goal} \]
\[ f(n) = g(n) + h(n) \]

The latter term, \( h(n) \) is known as a \textit{heuristic} function. The purpose of the heuristic function is to estimate (at a low computational cost) the remaining cost from \( n \). The resulting search graph will be more beam shaped (see Figure 6) towards the goal, meaning a smaller number of nodes expanded [22].

A heuristic is \textit{consistent} if it never overestimates the cost from a node to the goal node, and obeys the triangle inequality, meaning:

\[ h(n) \leq \text{cost from } n \text{ to } n' + h(n') \text{ where } n' \text{ is a successor of } n \]

If a consistent heuristic is used, A* retains the completeness and optimality guaranteed by UCS, but with a significantly reduced search space, and thus lower time/space complexity [22]. Improvements of A* concentrate on reducing the search space further. One notable effort is the Jump Point Search (JPS) system by Harabor and Grastien [28]. JPS attempts to reduce the search space by pruning redundant neighbors of each node, breaking the symmetry that exists in open areas of uniform weight grids (see Figure 7).

\[ \text{Figure 6: A* search illustration} \]

\[ \text{Figure 7: Multiple symmetrical paths in a 4-connected grid exist that are all optimal in the grid constrained world} \]

2.5.3 \textbf{Non-optimality of grid constrained paths}

If the search is constrained to a grid, resulting paths may be optimal with respect to the grid, but suboptimal with respect to reality (consider the solutions displayed in Figure 7). Nash, Koenig and Tovey [29] show in their paper that shortest paths of 8-connected 2D grids and 26-connected 3D grids can be longer...
than the shortest paths in a continuous environment by about 8% and 13% respectively.

2.5.4 Path smoothing

One way to mitigate the non-optimality mentioned in 2.5.3 is to apply Path Smoothing (PS) to the paths generated by the search algorithm. In “Programming Game AI by Example” [30], Mat Buckland suggests the following algorithm (see Figure 8 for an example in an 8-connected grid):

1) Begin with selecting the start node.

2) From the selected node, find the node closest to the goal node which has unobstructed line of sight with the selected node.

3) Mark the found node as the new neighbor of the selected node.

4) Select the found node, and go to 2, repeat until the goal node is reached.

![Figure 8: Paths before (blue) and after smoothing (purple)](image)

2.5.5 Theta*

While path smoothing may shorten paths generated by grid constrained search, the quality of the path is still subject to the input path to the path smoother. Grid constrained search may generate paths that are difficult to shorten. Nash, Daniel, Koenig and Felner [25] give examples of this, and present Theta* search. Theta* lifts the inherent limitation of A* that a child node in the search can only be connected to its immediate predecessor. This is accomplished by checking if unobstructed line of sight to the grandparent exists for each node explored. If so, the node is noted as being connected to its grandparent, instead of its parent. See Figure 9 for an example.

From experimental results in 2D-grids [25] Theta* is seen to produce shorter paths than both A* and A* with PS (an example of the latter illustrated by the last few nodes in Figures 8 and 9). This however comes with the increased computational cost of performing line-of-sight checks, which manifests itself as longer run times than A* in the experimental results. For 2D-problems, Nash, Daniel, Koenig and Felner [25] suggest calculating upper and lower angular bounds for when a node has line of sight, to reduce the number of line-of-sight checks.
checks (AP Theta*) [25]. For 3D-problems, Nash, Koenig and Tovey, in another paper [29], suggest deferring the line-of-sight check until the latest possible time, to possibly avoid performing some unnecessary checks (Lazy Theta*).

---

2.6 Route optimization and Traveling Salesman Problem

The algorithms presented in section 2.5 can be used to solve the problem of finding the cheapest path between two points. Another problem to be solved is to find the cheapest route, visiting at least all points once. This problem is a variant of the **Traveling Salesman Problem** (TSP) [31]:

“Given a set of cities along with the cost of travel between each pair of them, the *traveling salesman problem*, or TSP for short, is to find the cheapest way of visiting all the cities and returning to the starting point...” “...the ordering is called a *tour*...[32]”

This is how Applegate et. al defines the TSP. More generally, the cities can be viewed as nodes in a weighted graph, where the weight of the nodes' edges correspond to the distance between the cities. If the weight between two nodes is equal regardless of which direction they are traversed, the TSP is symmetric (sTSP), otherwise is asymmetric (aTSP). The problem is then to find the Hamiltonian cycle (a path through the graph visiting all nodes at least once) with the minimum weight [31].

Solving the problem by exhaustively enumerating all possible tours has a time complexity of $O(n!)$ [31][32] and is therefore considered impractical even for relatively small problems.

2.6.1 Nearest Neighbor Algorithm

A simple algorithm for finding an approximative solution for TSP is the **Nearest Neighbor** (NN) algorithm. It works by building a tour from a starting node, continuously selecting the next node by lowest cost from the list of unvisited neighboring nodes [33]. This approach produces acceptable results very quickly for Euclidean problems, but very poorly for the general TSP, both aTSP and sTSP. [34]. In fact, it has been shown that TSP:s can be constructed where the NN algorithm will select the worst tour [34].
2.6.2 Branch-and-bound algorithms

The family of Branch-and-bound (BB) strategies attempt to search for the best solution to a search problem in by dividing (branching) it into a set of smaller, and easier, sub-problems [32]. A lower bound of the cost of the final solution for the branch is calculated from the subproblem. This lower bound is compared to a known upper bound (cost of the best known complete solution) to decide if it is worth pursuing the branch further, or if it should be pruned.

The algorithm that coined the term “Branch-and-bound” [32] was presented by Little, Murty, Sweeney and Karel in 1963 [35]. In their experiments, they calculate random 30-city aTSP:s in about a minute on average on an IBM7090. The experimental results of the original paper suggests the average time complexity of the algorithm as being exponential in the number of cities.

In the original paper [35], describing Little et. al's algorithm alone takes up eight pages, thus a detailed description is omitted in this report and the reader is referred to the original paper [35], which also includes a worked example.

The original algorithm has a breadth-first approach, resulting in a minimum number of nodes being explored, however in the same paper [35], Little, Murty, Sweeney and Karel suggest that a depth-first approach may be even more feasible for a few reasons: First, the nature of how bounds are calculated in the algorithm make this more computationally advantageous. Second, memory usage will be considerably smaller than with a breadth-first approach, as only one branch will need to be held in memory at any given time. Third, the algorithm will “bee-line” to feasible solutions. This quickly gives an approximative solution that helps discarding unfeasible branches, and also enables the best known tour to be used as an approximative solution, if the algorithm is stopped prematurely. This last property is especially helpful. In fact, experimental results by Zhang [36] in 2000 indicate that Depth-First Branch-and-Bound (DFBB) outperform the, according to Zhang, best known approximative solution for the aTSP: the Kanellakis-Papadimitrou local search algorithm. This is both in terms of execution time, and also in terms of solution quality when the DFBB algorithm is terminated prematurely.

For exact solutions to larger problems, more sophisticated methods exist. See Applegate, Bixby, Chvatal, Cook for a broad study [32].
3 Methodology

As the project contains several sub-problems to be solved, methodology for each sub-problem will be discussed individually.

In general, an iterative approach has been applied, where each subproblem has been explored only to the degree required to complete the relevant goal.

3.1 Establishing project scope and requirements

To establish Arrowcam AB's requirements, two interviews were conducted with them. An unstructured interview, to determine the required capabilities of the application and later a structured interview to determine their software/hardware environment. A follow-up unstructured interview was also conducted to establish the sample-mission mentioned in section 1.4 as a realistic but, for a human pilot, challenging mission (detailed in Appendix B: Sample Mission).

3.2 Measuring algorithm runtime

To measure algorithm runtime, a minimal application was written, which would run the algorithm a number of times on the iPad. It measured the runtime using calls to CACurrentMediaTime() [37], and then printed the results to the display of the iPad. The application is shown in Appendix C: Application for testing algorithm performance. The reason for choosing this method, instead of using the performance-measuring facility of Xcode's unit testing framework XCTest, was to avoid any interference from foreign code/interaction between the iPad and the computer.

3.3 Calculating photo parameters from exported KML-data

Through experimentation with Google Earth, it was discovered that Google Earth does not produce views using the direct <Camera>-element, but with the <LookAt>-element (see section 2.1 for a description). The camera position was required to calculate a camera position based on vertical/horizontal bearing and distance from the focus point. When the formula for doing so was completed, it was tested by defining photos in Google Earth that would allow for comparing them with actual photos. The photos were taken with the aircraft, manually inputting the data from the formula, and compared with the Google Earth-photos to determine their resemblance. The Google Earth images are not presented here, but together with the result images in 5.1.2 for easy comparison. A mathematical comparison was never made, thus “resemblance” is not quantified, but a subjective measure by Arrowcam AB.
3.4 Determining feasibility of implementing algorithms in Swift

Researching development for the iOS platform during Swift, anecdotes of poor performance in the early versions of Swift were encountered. To determine if Swift 3 was fast enough to use for implementing the algorithms, three similar versions of the A* algorithm (see section 2.5.2) were developed: The first version was in Swift, with a community provided implementation of a priority queue. The second version was in Swift, but used the C++ STL priority queue implementation, through an Objective-C wrapper. The third version used “pure” C++, and was accessed from Swift through an Objective-C wrapper. A* was chosen as it was one of the candidate algorithms to use for path finding, and would test both CPU and memory performance. The results for other algorithms may not be analog to the results obtained for A* due to differing characteristics. The main performance measurement was total runtime to find optimal solutions in a 5m grid for all operation pairs in the sample mission. In choosing whether to implement in Swift or C++, code readability and maintainability was also considered.

3.5 Selecting and implementing path finding algorithm

In order to maintain the scope of the project, the selection of algorithms to be evaluated had to be severely restricted. It was determined that only the number of algorithms required to meet the goal set in 1.4.3 would be evaluated. The criteria of selecting algorithms for evaluation was:

- Completeness – The algorithm had to be able to find a solution if one existed
- Optimality – The algorithm had to find an optimal or near optimal solution
- Ease of understanding – The algorithm had to be simple enough to be understood by an undergraduate student during the course of the project
- Ease of implementation – The algorithm had to be easy to implement in Swift or C++.

Grid based A* was selected over visibility-graph based A* as the grid based representation required less pre-processing since it does not suffer from the same problems in 3D. Grid based A* would also allow further improvement through JPS if required. Several more sophisticated A* variants were rejected due to their increased complexity for a problem that did not require more performance. The second algorithm chosen for evaluation was Theta*, as the results from the paper [25] seemed promising, and since it would require very little work to derive it from an existing A* implementation.
The performance measure for selecting path finding algorithm was shortest total path cost of all pairs in the sample mission, still meeting the runtime requirement in the goal criteria in 1.4.3. To ensure the optimality requirement is met, 1% of all paths were calculated by hand, and compared with the generated algorithms. To ensure paths were not penetrating forbidden areas, paths were exported to KML and 10% of the paths generated for each algorithm were visually examined for validity.

3.6 Selecting and implementing route optimization algorithm

The same restrictions and criteria as in 3.5 were applied on selecting what route optimization algorithms should be evaluated. The problem is very widely studied, and as such, some very sophisticated algorithms exist to solve it, however, for its ease of understanding and implementation, Branch-and-Bound was selected for evaluation. Nearest Neighbour was also selected to use as a goal-completion reference, as mentioned in 1.4.4.

Due to Zhang's [36] findings (see section 2.6.2), local search algorithms were not evaluated, as they would likely be inferior to a Depth First Branch-and-Bound method. More sophisticated Branch-and-Bound-based algorithms such as Truncating Branch-and-Bound[36], or Branch-and-Cut [32] were not selected due to their increased complexity. With reference to the results from the original paper on the Branch-and-Bound algorithm [35], it was thought that the algorithm should be sufficient to meet the requirement of 30 operations.

Since wind should be taken into account, and since start positions and end positions of operations can not be assumed to coincide (see 1.4), the problem is asymmetric, and does not fulfill the triangle inequality, thus specialized algorithms to solve Euclidean/Metric TSP:s are thus rejected.

The performance measure for selecting route optimization algorithm was the runtime required to meet the optimality requirement in the goal criteria in 1.4.4.

3.7 Development environment

Development has been performed in macOS Sierra using Xcode 8.3.2. Xcode's own unit testing framework XCTest was used for unit testing.

The cocoapods package manager [17] was used for managing libraries.

For all performance testing (described in 3.2), code was compiled with flags -Os (Fastest, Smallest) for LLVM-generated code (i.e. C++), and -O -whole-module-optimization for the Swift-compiler generated code. Note that both represent the highest “Optimization level” selectable. All other compiler settings were left to defaults.

The target aircraft has been DJI Inspire 1.
4 Implementation

4.1 KML Mission Data
A simple protocol for how to define mission data (mission bounds and operations) in Google Earth was agreed on with Arrowcam AB. A description of the protocol, and a description of the sample mission mentioned in 1.4 can be found in Appendix A: Mission Definition in KML and Appendix B: Sample Mission respectively. A trivial parser for the KML format was also written, with details omitted for brevity.

4.2 Calculating photo parameters from KML-data
As mentioned in Appendix A: Mission Definition in KML the KML file exported from Google Earth not contain the explicit camera position and orientation, instead this had to be calculated. This was done by first calculating the horizontal and the vertical travel distance from the <tilt> and <range> elements in the <LookAt>-element (see 2.2), and then calculating the latitude and longitude from the distance travelled along the the reciprocal to the heading in the <heading>-element. Altitude was calculated by simply summing the vertical travel distance and the contents of the <altitude>-element. Camera tilt and heading were finally derived from the <tilt> and <heading>-elements respectively.

To validate the algorithm, test photos were defined in Google Earth (shown together with the resulting photos in 5.1.2). The algorithm was used to manually calculate camera position/orientation for each test photo. The positions/orientations were provided to an application that could fly the drone into position and take the photo. This was done for all photos. Finally, photos would be compared for visual resemblance.

4.3 Swift/C++ interoperability

Swift interoperability with C++ is achieved through Objective-C and C++ as shown in Figure 10. The Objective-C wrapper has an Objective-C-only header file (.h), which Swift is interoperable with, but uses Objective-C++ in its body file (.mm), which in turn can call C++ code. Since Objective-C can be directly called from Swift, the Swift-Wrapper is not necessary per se, but this structure was still chosen to allow the SwiftWrapper to marshal data in and out of un-
managed memory, making memory management transparent to the client code. For an example of this in action, see Appendix D: Swift A* with C++ STL Priority Queue.

4.4 Determining feasibility of implementing algorithms in Swift

4.4.1 Path finding search problem model

The model chosen for the search problem is conceptually described in Figure 11. (This model is somewhat revised later in 4.5) The structure roughly follows the outline suggested for a search problem suggested by Russel and Norvig in [22].

![Figure 11: Search problem model](image)

The search problem would be initialized with: a bounds object (containing the bounds-geometry), a grid map translating between real-world coordinates and grid coordinates and a list of all allowable actions (determining the connectivity of the grid). The search algorithm would call GridProblem::actionsFor(GridState), then GridProblem would call the Bounds-object to see what actions would result in hitting an obstructed grid node, and returning these actions. The bounds class uses a trivial point-in-polygon algorithm to determine whether positions are obstructed or not. A grid size of 5 m was used as this was thought to be a good compromise between achieving sufficient granularity of the world-representation, and keeping the search space small.

The implementation used was naive in several ways (did only consider the midpoint of each cuboid for determining if a cuboid was obstructed or not, lacking possible performance optimizations etc). The reason for this was to minimize development time, and because it was thought that despite the naive implementation, the result from testing the algorithms would still give sufficient indication on what technique (Swift/Swift & C++ STL/C++) should be used for future development.
4.4.2 A* Implementation

The implementation of A* followed the outline suggested for a search algorithm by Russel and Norvig in [22] and the description in 2.5.2. The algorithm was implemented in one Swift-only version (using a community-provided version of the priority queue data structure), one C++-only version (with wrappers to enable it being called from Swift) and one hybrid version where Swift was used for the algorithm, but wrappers were written to enable the Swift-version to use the C++ STL priority queue implementation. Source code for the last version is included in Appendix D: Swift A* with C++ STL Priority Queue. Small variations from the model described in 4.4.1 exist, but the implementation conceptually agrees with the model. The two other implementations (Swift-only and C++-only) are functionally equal to the version displayed in the appendix and their source code is omitted. At a late stage, a flaw was discovered in the implementation: A node already in the frontier-set would not be updated if another, shorter path was found to it. It was decided that while this may deteriorate path quality, the results of the algorithm run-time tests would still be valid. This would however need correcting before proceeding with the next subproblem (4.5).

4.5 Evaluating different search algorithms

The subproblem of determining implementation language 1.4.2 was fully investigated before choosing implementation language for succeeding subproblems. This investigation resulted in the decision to use C++ for the problem solving algorithms. For a discussion of this decisions, see section 6.2.

4.5.1 Problem structure

To optimize performance and to improve path quality, some improvements were made from the structure presented in 4.4.1. Significant changes were:

- The action cost could now be different for horizontal movement, upwards movement and downwards movement. This better reflected the performance characteristics of the actual aircraft.

- The search algorithm became independent from what actions are available, instead available successor states together with their costs were immediately available to the search algorithm.

- The Bounds/GridMap objects in Figure 11 was replaced with a grid-referenced height map which was pre-calculated before search begun.

- The height map now considered the entire cuboid for each grid node when determining obstacle clearance, preventing the search algorithm from “shortcutting” through corners of the bounds-environment.
• A line-of-sight check was added so that a goal successor would be immediately generated if unobstructed line of sight existed between the current node and the goal node.

• Problem specific knowledge was used to prevent successors of “useless” actions to be generated (such as trying to descend below the goal altitude – no-fly zones can not be underflown, or trying to climb above the goal altitude, if the search is already above the highest no-fly zone that can be overflown). The intention with this was to prune infeasible states from the search space.

• The problem object would use pre-allocated buffers to pass data back to the search algorithm instead of allocating/deallocating memory each time the search algorithm asked for a list of succeeding states.

For reasons mentioned in 4.4.1, a grid size of 5 m was used.

4.5.2 A*

In order to maximize the benefit from the similarity between Theta* and A*, A* was implemented as described by the A* pseudo-code in the original paper on Theta* by Nash et. al [25]. In order to update nodes already existing in the “open” set (as described in 4.4.2), the data structure for the open-set was changed to an ordered set instead of a priority queue. This was to allow for fast random search/deletion. A rudimentary memory pool was also created to somewhat decrease the number of memory allocations required. Most of the search algorithm was the same in both A* and Theta* and is fully listed in: Appendix E: Search Algorithm Base for A* and Theta*.

The only difference in the A* implementation and the Theta* implementation (which was subclassed from the A* class) was the updateVertex() method (which was used to record the lowest cost path for an explored node, see the appendix for context), listed below for A*:

```cpp
void AStar::updateVertex(AStarSearchNode* parent, AStarSearchNode* child, double stepCost, std::set<AStarSearchNode*>& openSet, const Grid3DProblem& problem, std::set<AStarSearchNode*>::const_iterator childInOpenSetIter) {
    // Is the path shorter through this parent to this child? (always, for new children)
    if(parent->pathCost + stepCost < child->pathCost) {
        // Since we will update the child with a shorter path, delete the current entry in the open-set first if it exists
        if(childInOpenSetIter != nullptr) {
            openSet.erase(*childInOpenSetIter);
            openNodesUpdated++;
        }
        child->pathCost = parent->pathCost + stepCost;
        child->parent = parent;
        child->heuristicValue = problem.heuristicValue(child->gridPosition);
        openSet.insert(child);
    }
```
This method would ensure that the path to any node found was always the shortest path found so far.

4.5.3 **Theta***

As mentioned in sections 2.5.5 and 4.5.2, very few changes needed to be made to A* to derive Theta*. Theta* was subclassed from the A* implementation. Since the Theta* needed to make line-of-sight checks, it also required a reference to the bounds-object. The updateVertex() method was overridden as follows:

```cpp
void Theta::updateVertex(AStarSearchNode* parent, AStarSearchNode* child, double stepCost, std::set<AStarSearchNode*>& openSet, const Grid3DProblem& problem, std::set<AStarSearchNode*>::const_iterator childInOpenSetIter)
{
    //Is there line of sight to the grandparent?
    if(parent->parent != nullptr && gridBounds.hasEuclideanLineOfSight(parent->parent->gridPosition, child->gridPosition)) {
        const auto costFromGrandParent = parent->parent->pathCost + problem.straightLineCost(parent->parent->gridPosition, child->gridPosition);
        if(costFromGrandParent < child->pathCost) {
            //Since we will update the child with a shorter path, delete the current entry in the open-set first if it exists
            if(childInOpenSetIter != nullptr) {
                openSet.erase(*childInOpenSetIter);
            }
            child->pathCost = costFromGrandParent;
            child->parent = parent->parent;
            child->heuristicValue = problem.heuristicValue(child->gridPosition);
            openSet.insert(child);
        }
    } else {
        //Treat like regular AStar node (no parent-skipping)
        AStar::updateVertex(parent, child, stepCost, openSet, problem, childInOpenSetIter);
    }
}
```

4.5.4 **Path smoothing**

A path smoothing algorithm was implemented based on the ideas in section 2.5.4 with the following modifications:

- Prior to smoothing, extra nodes would be injected along the path between the existing nodes at a minimum of a given interval. This was to give the path smoother more nodes to use for smoothing, and it was thought that this could result in shorter paths.

- Instead of performing a forward linear search from all nodes between the node being smoothed from to the goal node, a binary search was im-
4.5.5 **Swift Path Finding Interface**

A Swift interface to the path finding algorithms was written with the following public interface:

```swift
func findPathFrom(_ from: inout Position3D, to: inout Position3D, within bounds: CPPBounds, usingStrategy strategy: Strategy, optimizedFor performanceProfile: AircraftPerformanceProfile) -> Path?
```

```swift
enum Strategy {
    case Astar
    case Theta
    case AstarPS
    case ThetaPS
}
```

The performance profile-argument would cause the z-weights to be set in the search problem to optimize the path for a particular aircraft.

Since the path smoothing code has already written, it was decided to include a strategy using Theta* with path smoothing as well, since this came at virtually zero additional development cost.

Some of the complexity of configuring the search algorithms was abstracted away by providing default parameters in the Path Finding interface. One such parameter was grid connectivity. It was decided that a 26-connected grid (diagonal movement possible in all directions simultaneously) would be used with A* to produce high quality paths, while Theta* would use a 6-connected grid (movement only possible in a single direction at a time). The reason for this was that it was thought that Theta* still could produce high quality paths in a 6-connected grid with the added benefit that each node would have a maximum of 6 neighbors to explore rather than 26.

The interface would retrieve the path-finding result from the search algorithms through the relevant Objective-C-wrappers, call code for path smoothing, if desired, and marshal the results into Swift-managed memory. For a full source code listing, see: Appendix G: Swift path finding interface.

4.6 **Route optimization algorithms**

4.6.1 **Cost Matrix**

To represent city pair distances, a cost matrix was implemented, where the rows would represent the from-city, and columns would represent the to-city. Relevant parts of the public interface for the Cost Matrix is listed below:

```cpp
class CostMatrix {
public:
    void setCost(unsigned row, unsigned col, double cost);
    double getCost(unsigned row, unsigned col) const;
```
double getMinInRow(unsigned row, unsigned& colOut) const;
void markDeleted(unsigned row, unsigned col);
void markDeleted(unsigned* rows, unsigned* cols, size_t n);
bool findFirstLessThan(double value, unsigned& rowOut, unsigned& colOut) const;

double getMaxTheta(unsigned& rowOut, unsigned& colOut) const;

double littleReduce();
};

Some of the methods were specific to the branch-and-bound algorithm used (getMaxTheta(), littleReduce(), findFirstLessThan()), but were still implemented as part of the Cost Matrix to allow optimization requiring access to the Cost Matrix's private data members.

When marking a row and column combination in the matrix as deleted (effectively removing the possibility to go from (row) and to (column) a particular city again), the internal storage is re-allocated to cater for the smaller data requirement, while a mapping is retained, allowing the client code to use the same column and row numbers as in the original matrix. Costs were represented by the double-datatype, which is 8 byte wide in the 64-bit iOS runtime [38]. For a 30*30 cost matrix, the storage of the costs alone would thus require 7kB. For this reason, saving memory was thought to be important, as algorithms may require keeping several cost matrices in memory simultaneously.

4.6.2 Nearest Neighbor Algorithm

The nearest neighbor algorithm was developed to have a reference to measure the performance of the other route optimization algorithm(s) against, as described in 1.4.4. The implementation was trivial and is included in Appendix H: Nearest Neighbor.

4.6.3 Branch-and-Bound

The branch-and-bound algorithms were implemented as faithfully as possible in accordance with the original paper by Little, et. al [35]. One breadth-first variant was implemented, used a priority queue for selecting the next node to investigate. In addition, one variant was implemented using a stack, with the branch excluding a city pair always being pushed before the node assigning a city pair, leading to a depth-first behavior, as discussed in 2.6.2. A time-out facility cutting the search of prematurely was also added to the second implementation. To avoid unnecessary syscalls, the timeout facility is only polled every n nodes, meaning that there is a small delay to be expected between the lapse of the timeout and the algorithm actually returning. C++ move semantics were used to the largest extent possible to avoid costly copy-operations.

Due its the sheer size and due to it being littered with implementation specific details, the implementation source code is omitted. The methods used for calculating lower bounds for each node is exactly in accordance with the original paper [35].
5 Results

5.1 Calculating photo parameters from exported KML-data

Three test images were defined in Google Earth with a focal length of 15 mm. The images from Google Earth are shown together with the result images in this section for easy comparison.

5.1.1 Image parameters

Table 1 contains the image parameters calculated from the KML-data in Google Earth by the method presented in 4.2. These are what the aircraft used to take the resulting photos below.

<table>
<thead>
<tr>
<th>#</th>
<th>Latitude°</th>
<th>Longitude°</th>
<th>Alt. (m)</th>
<th>Hdg°</th>
<th>Tilt°</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27.806686</td>
<td>-15.581683</td>
<td>409.17</td>
<td>159.7</td>
<td>85</td>
</tr>
<tr>
<td>2</td>
<td>27.805767</td>
<td>-15.581946</td>
<td>451.14</td>
<td>101.9</td>
<td>73.36</td>
</tr>
<tr>
<td>3</td>
<td>27.805804</td>
<td>-15.582339</td>
<td>455.67</td>
<td>353.2</td>
<td>82</td>
</tr>
</tbody>
</table>

Table 1: Calculated image parameters

Result images follow on next page
5.1.2 Result images

Figure 12: Image 1 in Google Earth

Figure 13: Image 1 result
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson
2017-06-04

Figure 14: Image 2 in Google Earth

Figure 15: Image 2 result
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson

Figure 16: Image 3 in Google Earth

Figure 17: Image 3 result
5.2 Determining feasibility of implementing algorithms in Swift

5.2.1 Algorithm runtime

Table 2 and Figure 18 show the mean time in seconds required for finding paths between all 870 operation pairs in the sample mission for each implementation.

<table>
<thead>
<tr>
<th></th>
<th>Swift</th>
<th>Swift/C++ STL</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean runtime (s)</td>
<td>357.54</td>
<td>88.25</td>
<td>37.51</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>2.72%</td>
<td>0.27%</td>
<td>2.05%</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 2: Swift, Swift/C++ STL, C++ Mean runtime

![Figure 18: Swift, Swift/C++ STL, C++ Mean runtime (s)](image)

5.2.2 Profiler traces

Figures 19, 20, 21 shows output from the profiler Instruments (bundled with Xcode) during one run of finding all paths in the sample mission.

![Figure 19: Profiler trace for Swift implementation](image)
5.3 Selecting and implementing path finding algorithm

5.3.1 Path validity

10% of the paths generated by each algorithm were examined manually, and all paths examined were found to be valid.

5.3.2 Algorithm runtime

Table 3 and Figure 22 show the mean time in seconds required for finding paths between all 870 operation pairs in the sample mission for each algorithm. “PS” indicates Path Smoothing was used.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Mean runtime (s)</th>
<th>Rel. std. dev (%)</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>A*</td>
<td>5.47</td>
<td>1.94%</td>
<td>10</td>
</tr>
<tr>
<td>A* PS</td>
<td>5.52</td>
<td>1.94%</td>
<td>10</td>
</tr>
<tr>
<td>Theta*</td>
<td>3.51</td>
<td>0.39%</td>
<td>10</td>
</tr>
<tr>
<td>Theta* PS</td>
<td>3.54</td>
<td>0.43%</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 3: A*, A* PS, Theta*, Theta* PS Mean runtimes
5.3.3 Path cost vs. manually calculated path

Table 4 and Figure 23 describe the cost of the paths generated by the search algorithm (expressed as required flight time in seconds) for all 870 operation pairs in the sample mission. Paths were calculated with zero-wind conditions. Red numbers in the table exceed the performance goal set in 1.4.3.

<table>
<thead>
<tr>
<th>Path</th>
<th>From</th>
<th>To</th>
<th>Manual Time (s)</th>
<th>A* Time (s)</th>
<th>%</th>
<th>A* PS Time (s)</th>
<th>%</th>
<th>Theta* Time (s)</th>
<th>%</th>
<th>Theta* PS Time (s)</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>#1</td>
<td>Private House</td>
<td>Felspar 5</td>
<td>9.6</td>
<td>11.63</td>
<td>121</td>
<td>10.67</td>
<td>111</td>
<td>11.63</td>
<td>121</td>
<td>10.67</td>
<td>111</td>
</tr>
<tr>
<td>#2</td>
<td>Pier side view</td>
<td>Garnet 2</td>
<td>21.52</td>
<td>23.72</td>
<td>110</td>
<td>21.55</td>
<td>100</td>
<td>21.52</td>
<td>100</td>
<td>21.52</td>
<td>100</td>
</tr>
<tr>
<td>#3</td>
<td>Grand Ave</td>
<td>Felspar 11</td>
<td>25.28</td>
<td>26.8</td>
<td>106</td>
<td>25.36</td>
<td>100</td>
<td>25.38</td>
<td>100</td>
<td>25.33</td>
<td>100</td>
</tr>
<tr>
<td>#4</td>
<td>Garnet 5</td>
<td>Felspar 4</td>
<td>12.47</td>
<td>12.47</td>
<td>100</td>
<td>12.47</td>
<td>100</td>
<td>12.47</td>
<td>100</td>
<td>12.47</td>
<td>100</td>
</tr>
<tr>
<td>#5</td>
<td>Felspar 3</td>
<td>Meat market</td>
<td>23.52</td>
<td>23.52</td>
<td>100</td>
<td>23.52</td>
<td>100</td>
<td>23.52</td>
<td>100</td>
<td>23.52</td>
<td>100</td>
</tr>
<tr>
<td>#6</td>
<td>San Diego Bay</td>
<td>Beach looking N</td>
<td>23.93</td>
<td>28.71</td>
<td>120</td>
<td>26.23</td>
<td>110</td>
<td>26.79</td>
<td>112</td>
<td>26.47</td>
<td>111</td>
</tr>
<tr>
<td>#7</td>
<td>Felspar 9</td>
<td>Felspar 1</td>
<td>13.45</td>
<td>15.7</td>
<td>117</td>
<td>13.93</td>
<td>104</td>
<td>13.93</td>
<td>104</td>
<td>13.93</td>
<td>104</td>
</tr>
<tr>
<td>#8</td>
<td>Parking lot</td>
<td>Felspar 7</td>
<td>13.55</td>
<td>13.55</td>
<td>100</td>
<td>13.55</td>
<td>100</td>
<td>13.55</td>
<td>100</td>
<td>13.55</td>
<td>100</td>
</tr>
<tr>
<td>#9</td>
<td>Boulevard</td>
<td>Garnet 4</td>
<td>14.7</td>
<td>14.7</td>
<td>100</td>
<td>14.7</td>
<td>100</td>
<td>14.7</td>
<td>100</td>
<td>14.7</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>Total</td>
<td></td>
<td>158.02</td>
<td>170.8</td>
<td>108</td>
<td>161.98</td>
<td>103</td>
<td>163.49</td>
<td>103</td>
<td>162.16</td>
<td>103</td>
</tr>
</tbody>
</table>

Table 4: Path costs for paths generated by search algorithms vs manually calculated
### 5.3.4 Visualized paths

In Figures 24 and 25 below, example output from the search algorithms for the path “San Diego Bay to Beach looking N” is shown. Figure 24 Shows the paths before path smoothing is applied and Figure 25 shows the path after path smoothing is applied.

![Image](image_url)
5.3.5 Search algorithm path cost for all paths

Table 5 shows the total cost of all of the 870 paths generated from the sample mission. The column “Best path” is the total cost of all paths, if the best generated path is selected for each path. “% equals best” is the percentage of times that the search algorithm comes up with the lowest cost path. (Note that several algorithms can come up with the same, lowest cost path, all algorithms that do are then considered to “equal best”).

<table>
<thead>
<tr>
<th></th>
<th>Best path</th>
<th>A*</th>
<th>A* PS</th>
<th>Theta*</th>
<th>Theta PS*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total cost (s)</td>
<td>16049.13</td>
<td>17352.3</td>
<td>16374.34</td>
<td>16149.97</td>
<td>16077.04</td>
</tr>
<tr>
<td>% of best</td>
<td>108.12%</td>
<td>102.03%</td>
<td>100.63%</td>
<td>100.17%</td>
<td></td>
</tr>
<tr>
<td>% equals best</td>
<td>39.66%</td>
<td>64.02%</td>
<td>48.05%</td>
<td>85.40%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Path cost for all paths

5.4 Selecting and implementing route optimization algorithm

Figure 26 and Tables 6 through 10 show the cost of the best tour for each algorithm, grouped by wind condition. The tables also include mean run time. “NN” represent the Nearest Neighbor algorithm, “DFBB 5/29 s” means Depth First Branch-and-Bound with a time cutoff limit of 5 and 29 seconds respectively. “DFBB” alone means DFBB run until an exhaustive solution has been found.
“BB” means Breadth First Branch-and-Bound. In all cases except the “No wind” case, BB reached about 700 MB memory use and crashed before it found an optimal solution.

In the cases where a time limited DFBB-search ran to exhaustion before the time limit was up, the same result has been inferred for longer time limits. These inferred results are cursive, and n = inferred in the table.
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson
2017-06-04

<table>
<thead>
<tr>
<th></th>
<th>NN</th>
<th>BB</th>
<th>DFBB 5 s</th>
<th>DFBB 29 s</th>
<th>DFBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean tour cost (s)</td>
<td>499.41</td>
<td>-</td>
<td>404.35</td>
<td>404.35</td>
<td>404.35</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>0.00%</td>
<td>-</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mean runtime (s)</td>
<td>0.01</td>
<td>-</td>
<td>5.1</td>
<td>26.64</td>
<td>26.64</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>1.64%</td>
<td>-</td>
<td>0.70%</td>
<td>0.84%</td>
<td>0.84%</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>10</td>
<td>inferred</td>
</tr>
</tbody>
</table>

Table 8: Wind from East 8 m/s

<table>
<thead>
<tr>
<th></th>
<th>NN</th>
<th>BB</th>
<th>DFBB 5 s</th>
<th>DFBB 29 s</th>
<th>DFBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean tour cost (s)</td>
<td>463.39</td>
<td>-</td>
<td>355.78</td>
<td>355.78</td>
<td>349.96</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>0.00%</td>
<td>-</td>
<td>0.00%</td>
<td>0.00%</td>
<td>-</td>
</tr>
<tr>
<td>Mean runtime (s)</td>
<td>0.01</td>
<td>-</td>
<td>5.82</td>
<td>29.59</td>
<td>212.43</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>2.35%</td>
<td>-</td>
<td>0.16%</td>
<td>0.86%</td>
<td>-</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>10</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 9: Wind from South 8 m/s

<table>
<thead>
<tr>
<th></th>
<th>NN</th>
<th>BB</th>
<th>DFBB 5 s</th>
<th>DFBB 29 s</th>
<th>DFBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean tour cost (s)</td>
<td>491.71</td>
<td>-</td>
<td>406.45</td>
<td>404.45</td>
<td>404.45</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>0.00%</td>
<td>-</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0.00%</td>
</tr>
<tr>
<td>Mean runtime (s)</td>
<td>0.01</td>
<td>-</td>
<td>5.34</td>
<td>18.17</td>
<td>18.17</td>
</tr>
<tr>
<td>Rel. std. dev (%)</td>
<td>1.64%</td>
<td>-</td>
<td>0.15%</td>
<td>2.30%</td>
<td>2.30%</td>
</tr>
<tr>
<td>n</td>
<td>10</td>
<td>-</td>
<td>10</td>
<td>10</td>
<td>inferred</td>
</tr>
</tbody>
</table>

Table 10: Wind from West 8 m/s

5.4.1 Visualized tours

Examples of the generated tours from the tables above are included in Figure 27 and Figure 28. Each new color indicates the path between two unique operations. “A” indicates the first point in the tour, “B” indicates the second point in the tour. This is to help the reader see the direction of the tour.
Figure 27: NN algorithm, Wind from West (left in image) 8 m/s. Image attribution: Google Earth, Landsat/Copernicus

Figure 28: DFBB algorithm, Wind from West (left in image) 8 m/s. Image attribution: Google Earth, Landsat/Copernicus
6 Discussion

6.1 Calculating photo parameters from exported KML-data

According to Arrowcam AB, the resulting images in 5.1.2 produced by the method discussed in 4.2 have excellent resemblance to the images defined in Google Earth. The method is considered viable for use in the final application.

6.1.1 Goal completion

The goal is considered fully completed.

6.2 Determining feasibility of implementing algorithms in Swift

6.2.1 Performance

Comparing the three different implementations, the Swift implementation was slower than the C++ implementation by an order of magnitude. It is probably not wise to draw broad conclusions about Swift's performance from this, as it is quite possible that an implementation-specific detail is causing this poor performance. Looking at the profiler traces of the different implementations, it is rather suspicious that the pop operation from the priority queue takes up 68% of the time in the Swift implementation whereas it only takes up about 1% and 5% in the Swift/C++ STL and the C++ implementations respectively. The cause of this was not determined, and a search for the cause was abandoned.

It was noted that when using the C++ STL priority queue, the implementation was faster than the “pure” Swift implementation by a factor of 4. This furthers the suspicion that something in the integration with, or implementation of the community provided priority queue caused the poor performance of the Swift-only solution.

Even though a number of known possible optimizations were not implemented at this stage, the poor performance of the Swift implementations in nominal numbers was considered a significant risk against meeting performance goals required by later subproblems.

6.2.2 Code maintainability

When implementing in C++ it was noted to be very convenient to have direct access to the STL, without having to write wrappers for each container. It was also noted that the algorithms used lended themselves very well to manual memory management, something that was experienced as somewhat awkward in Swift, but rather natural in C++.
The biggest downside to implementing the algorithms in C++ was the extra code required to achieve interoperability with Swift. In the implementation used for this comparison 786 lines of problem-specific code was written. In addition to this, 638 lines of code were written (45% of the C++-related code) solely for interoperability with Swift. Late in the process, it was discovered that XCTest could be used to unit test C++-code through Objective-C++ directly. This, in combination with using a more narrow interface to Swift was thought to be able to improve the ratio between problem-specific code and interoperability-code. (After completing implementations of subproblems 3 (1.4.3) and 4 (1.4.4), this ratio was about 16%).

6.2.3 Subproblem summary
In summary, the benefit of direct access to the C++ STL, together with the considerably better performance of the pure C++ solution (compared to both the “pure” Swift, and the “hybrid” Swift/C++-solutions) was considered outweighing the additional code introduced by the Swift/C++ interoperability layer, and it was decided to implement algorithms in later subproblems in C++.

6.2.4 Goal completion
The goal is considered completed with a reservation. All algorithms were implemented and measurements made. For the specific implementation, C++ was clearly superior in performance, however the results are not sufficient to reliably say anything about C++ vs. Swift performance in general.

6.3 Selecting and implementing path finding algorithm
After adding line-of-sight checking and other problem specific knowledge discussed in 4.5.1, all algorithms met the runtime requirement in 1.4.3 with ease, as seen in Table 3.

All algorithms also met the optimality requirement even without smoothing as seen in Table 4. (It is however noted that the sample size comparing against manually calculated paths was made quite small due to the amount of work required to calculate optimal paths by hand). In the sample there are four paths that only require a line-of-sight (LOS) check. It can be argued whether these paths are relevant to comparing path finding algorithms, but as the paths were randomly selected from a typical environment that the algorithm would work in, and since being able to perform LOS checks is a relevant quality of a path finding algorithm, the line-of-sight paths were deemed relevant.

An interesting auxiliary conclusion from Table 5 is that when Theta* operated on a 6-connected grid (as mentioned in 4.5.5), it was superior to A*, operating on a 26-connected grid, both in runtime and in path quality. During the research phase of this project, no literature was encountered where Theta* was compared with A* with Theta* being restricted to a 6-connected grid (3D) or a 4-connected grid (2D). The same relationship exist, but with smaller difference in path quality, between Theta* PS and A* PS.
Considering the criteria in 3.5, Theta* PS was selected for path finding. It is considered to be sufficient for the application, but if further refinements were to be made, focus would go into improving memory management, possibly increasing the use of pooling and otherwise decreasing the number of memory allocations made.

The results of the optimizations made (discussed in 4.5.1) is shown by the run-time of the algorithms having decreased by a factor of 5-10. If the same improvement was simply applied to the Swift implementation tested and discussed in 6.2, it is plausible that a “pure” Swift implementation could have met the performance goals required by this subproblem, this was however not tested.

6.3.1 Goal completion

The goal is considered fully completed.

6.4 Selecting and implementing route optimization algorithm

The performance achieved by the implementations of the branch-and-bound algorithms is rather humble, considering that Little et al. [35] achieved comparable performance 50 years ago. This is probably explained by that very little effort was spent on optimizing data-structures, and sub-algorithms used for branching and bounding. As Little et al [35] states: “The efficiency of the process, however rests very strongly on the devices used to split the subsets and to find the lower bounds”. The achieved performance of the DFBB implementation was however considered satisfactory, and further performance improvements were not pursued.

The breadth first algorithm had a tendency to run out of memory for more difficult problems. In hindsight, it is believed that Little et al. [35] implicitly discards the cost matrix of a node if it is not immediately branched from. The implementation used in this project does not discard any cost matrices for non-terminal nodes. This is thought to be the reason why the algorithm quickly runs out of memory. As the DFBB implementation does not suffer from this in the same way, and as it achieves satisfactory performance, no last minute attempt to change the memory management in the breadth first implementation was made.

Looking at tables 6 through 10, it can be seen that the DFBB algorithm meets the performance requirement in 1.4.4 even with a cutoff time of only 5 seconds. In fact, Figure 26 shows that the time spent searching beyond 5 seconds does not decrease the flight time sufficiently to make the extra time spent searching worthwhile. Figure 26 also seems to supports that the nearest neighbor algorithm is a worse algorithm for aTSP:s (with wind) than for symmetric TSP:s (no wind. A curious observation from Table 6 is the short runtime for an exhaustive search, compared to the time taken for an exhaustive search in tables 7 to 10. The reason why has not been established, however it is possible that the algorithm simply finds very close upper and lower bounds very early in its run, pruning away most of the search space.
Considering the criteria in 3.6, DFBB with 5 s. cutoff was selected for route optimization. Further improvements would include optimizing subroutines for calculating lower bounds, and branching pairs. Looking into Truncated DFBB as proposed by Zhang [36] would also be considered.

6.5 Ethical considerations

6.5.1 Privacy

Use of unmanned aircraft for photographing is today controversial with regards to privacy. At the time of writing, Sweden is an example where “drone photography” is in general legally equivalent to monitoring by CCTV, and require the same permits [39]. The author of this report believes use of the final product using the algorithms implemented and evaluated in this report does not significantly affect privacy in either direction.

6.5.2 Environmental

The possibility to add restricted zones that must be circumnavigated can make it easier to avoid overflying sensitive parts of a mission area. As the flight time for a mission will likely be shorter than what a human operator could accomplish without assistance, the time spent “on site” will decrease, and also decreasing the time spent making noise, possibly disturbing neighbors and animals. The author of the report has not come up with any negative environmental factors.

6.5.3 Safety

It should be noted that the final product is not meant to assume the responsibility of a human operator. The final product is intended to be used with the operator maintaining line of sight with the aircraft, monitoring the aircraft and always being ready to take manual control if necessary.

If the final product was used to perform missions without supervision, safety would be severely compromised. Not considering strictly legal factors, no assumptions can be made about the accuracy of the data used to create the bounds for the aircraft. The algorithms implemented in this project have not been validated to the extent necessary for unsupervised operation. The reliability of the actual aircraft and its on-board software could also present a hazard.

If the final product is used as designed, it should allow the operator to spend less effort looking at a display trying to take the right photo, and more time monitoring aircraft performance and maintaining situational awareness. This could improve safety compared to fully manual operation.
References


Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson


42
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson 2017-06-04


[30]  M. Buckland, Programming Game AI by Example, 1st edt, Plano, TX, USA: Wordware Publishing Inc, 2005


Appendix A: Mission Definition in KML

Mission definition in KML

Google Earth was used to plan missions, including what photo operations were to be conducted and what areas were allowable for the aircraft to fly in. This data was exported as KML, and a parser was written to read the data. Details on the parser are omitted for brevity.

Photo operations

![Photo operation in Google Earth](image)

Photo operations were defined as “Placemarks” in Google Earth (see Figure 29). When KML data was exported from Google Earth, each placemark would have an associated <LookAt> element as discussed in 2.2. Unfortunately, no method was found to make Google Earth produce a <Camera> element instead, which made it necessary to calculate “backwards” to find the camera position for each photo. The calculation was trivial and is omitted for brevity. To allow for different types of operations, and to allow passing camera parameters etc, the “Description”-field was used. In this version, the only valid entry was “simplephoto”, indicating a regular single shot photograph.
Bounds

The allowable area for the aircraft was defined by drawing a polygon, setting its height to the lowest allowable area and putting “deck” in its description, together with the maximum allowable altitude (see Figure 30). Omitting the maximum allowable altitude defaults it to the lower bound + 120 m.

Further restrictions could be added by adding additional polygons, and putting “nofly” in the description. The altitude of the polygon would serve as the minimum altitude to overfly the polygon. Adding “unlimited” would completely exclude the lateral limits of the polygon from the allowable area.
Appendix B: Sample Mission

A sample mission was developed together with Arrowcam AB to use for evaluating the performance of the algorithms under test. The sample mission was chosen to be realistic, but challenging. The sample mission is presented in the following figures. Figure 32 displays the bounds for the mission. The green plane represents the deck and is approximately 250 m * 500 m. Yellow areas are no-fly zones up to their respective ceiling altitude. Red boxes are no-fly zones with unlimited altitude. The vertical limits of the bounds is between 31 and 150 m.

![Figure 32: Bounds](image)

Figure 33 displays the location and altitude of all photo locations. Note that the positions indicated are the positions required by the aircraft when taking each photo, and not the point of interest for the photo itself. Finally Figure 34 displays bounds and photo locations together.
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson
2017-06-04

Figure 33: Photo locations

Figure 34: Bounds and photo locations
Appendix C: Application for testing algorithm performance

```swift
import UIKit

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()
    }
    @IBOutlet weak var statusLabel: UILabel!
    @IBOutlet weak var resultsOutput: UILabel!

    @IBAction func runPressed(_ sender: Any) {
        statusLabel.text = "Running..."
        profile()
        statusLabel.text = "Done"
    }

    override func didReceiveMemoryWarning() {
        super.didReceiveMemoryWarning()
        print("Memory warning")
    }

    private func profile() {
        // Setup for test here
        var timing = [Double]()
        for _ in 1...samples {
            let testStarted = CACurrentMediaTime() // Test algorithm here
            let testFinished = CACurrentMediaTime() // Record any other test results here
            timing.append(testFinished - testStarted)
        }
        self.resultsOutput.text = "Output timing + other data here"
    }
}
```
Appendix D: Swift A* with C++ STL Priority Queue

AstarSTL.swift

//
//  AStarSTL.swift
//  Falco
//
//  Created by Fredrik Nilsson on 2017-04-08.
//  Copyright © 2017 Fredrik Nilsson (frni1203). All rights reserved.
//
import Foundation
import QuartzCore

class AStarSTL<State: Hashable, Action>: SearchAlgorithm {
    typealias PState = State
    typealias PAction = Action
    typealias _SearchNode = SearchNode<State, Action>

    private let heuristicFunction: (State) -> Double

    init(withHeuristic heuristicFunction: @escaping (State) -> Double) {
        self.heuristicFunction = heuristicFunction
    }

    func solve<P: SearchProblem>(_ problem: P) -> [(Action?, State)]? where P.Action == PAction, P.State == PState {
        var explored = Set<State>()
        var frontierQueue = STLPriorityQueue<_SearchNode>()
        frontierQueue.push(problem.initialNode, withValue: heuristicFunction(problem.initialNode.state))
        var frontierStateSet = Set<State>()
        frontierStateSet.insert(problem.initialNode.state)
        while !frontierQueue.isEmpty {
            let exploring = frontierQueue.pop()
            frontierStateSet.remove(exploring.state)
            if problem.isGoal(exploring) {
                return AStarSTL<State, Action>.buildSolutionTo(exploring)
            }
            explored.insert(exploring.state)
            let successors = problem.successorsOf(exploring)
            for successor in successors {
                if !explored.contains(successor.result.state) && !frontierStateSet.contains(successor.result.state) {
                    frontierQueue.push(successor.result, withValue: (successor.result.pathCost + heuristicFunction(successor.result.state)))
                    frontierStateSet.insert(successor.result.state)
                }
            }
        }
    }
}
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson

private static func buildSolutionTo(_ goalNode: _SearchNode) -> [(Action?, State)] {
    var stateSequence = [(Action?, State)]()
    var node: _SearchNode! = goalNode
    repeat {
        stateSequence.append(node.action, node.state)
        node = node?.parent
    } while node != nil
    stateSequence.reverse()
    return stateSequence
}

STLPriorityQueue.swift

// STLPriorityQueue.swift
// Falco
// Created by Fredrik Nilsson on 2017-04-08.
// Copyright © 2017 Fredrik Nilsson (frni1203). All rights reserved.

import Foundation

class STLPriorityQueue<T> {
    private var wrappedQueue: STLPriorityQueueWrapperWrapper

    public var isEmpty: Bool {
        get {
            return self.wrappedQueue.empty()
        }
    }

    public var count: UInt {
        get {
            return self.wrappedQueue.count()
        }
    }

    init() {
        self.wrappedQueue = STLPriorityQueueWrapperWrapper()
    }

    func push(_ object: T, withValue value: Double) {
        
    50
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson

```swift
let objectPtr = UnsafeMutablePointer<T>.allocate(capacity: 1)
objectPtr.initialize(to: object)
wrappedQueue.pushObject(objectPtr, withValue: value)
```

```swift
func pop() -> T {
    let objectPtr = wrappedQueue.pop()!.assumingMemoryBound(to: T.self)
    let object = objectPtr.pointee
    objectPtr.deinitialize()
    objectPtr.deallocate(capacity: 1)
    return object
}
```

```swift
dinit {
    while !wrappedQueue.empty() {
        let objectPtr = wrappedQueue.pop().assumingMemoryBound(to: T.self)
        objectPtr.deinitialize()
        objectPtr.deallocate(capacity: 1)
    }
}
```

---

**STLPriorityQueue.h**

// STLPriorityQueue.h
// Falco
//
// Created by Fredrik Nilsson on 2017-04-07.
// Copyright © 2017 Fredrik Nilsson (frnii203). All rights reserved.
//
#ifndef STLPriorityQueue_h
#define STLPriorityQueue_h

#import <Foundation/Foundation.h>

struct STLPriorityQueueWrapper;

@interface STLPriorityQueueWrapperWrapper : NSObject {
    struct STLPriorityQueueWrapper *queueWrapper;
} - (id) init;
- (void) dealloc;
- (void) pushObject:(void*)object withValue:(double)value;
- (void*) pop;
- (BOOL) empty;
- (unsigned long) count;
@end

51
STLPriorityQueue.mm

struct PriorityQueueNode {
    void* objectPointer;
    double queueValue;
    bool operator<(PriorityQueueNode other) const {
        return queueValue > other.queueValue;
    }
};

struct STLPriorityQueueWrapper {
    std::priority_queue<PriorityQueueNode> *queuePtr;
};

@implementation STLPriorityQueueWrapperWrapper

- (id)init {
    if (self = [super init]) {
        queueWrapper = new STLPriorityQueueWrapper();
        queueWrapper->queuePtr = new std::priority_queue<PriorityQueueNode>();
    }
    return self;
}

- (void)dealloc {
    delete queueWrapper->queuePtr;
    delete queueWrapper;
}

- (void)pushObject:(void*)object withValue:(double)value {
    PriorityQueueNode node = {object, value};
    queueWrapper->queuePtr->push(node);
}

- (void*)pop {
    auto node = queueWrapper->queuePtr->top();
    queueWrapper->queuePtr->pop();
    return node.objectPointer;
}


Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson

```c

}  
- (BOOL) empty {  
    return queueWrapper->queuePtr->empty();  
}  
- (unsigned long) count {  
    return queueWrapper->queuePtr->size();  
}

@end
```
Appendix E: Search Algorithm Base for A* and Theta*

```cpp
size_t AStar::findSolution(const Grid3DProblem& problem, Grid3DNode** resultPath) {
    AStarNodeMemoryPool memoryPool;
    std::unordered_map<Grid3DNode, AStarSearchNode*> seenNodes;
    std::set<AStarSearchNode*> open;
    // The "closed" set in the original formulation of A* is the set difference seen \ open
    ProblemSuccessor* successorBuffer = new ProblemSuccessor[problem.maxSuccessorCount()];
    auto initialSuccessor = problem.getInitial();
    auto initialNode = memoryPool.addNode(AStarSearchNode(nullptr, initialSuccessor.node, 0, 0));
    seenNodes[initialNode->gridPosition] = initialNode;
    open.insert(initialNode);

    while(!open.empty()) {
        auto exploringIter = open.begin(); auto exploring = *exploringIter;
        open.erase(exploringIter);
        if(problem.isGoal(exploring->gridPosition)) {
            return buildPath(initialNode, exploring, resultPath);
        }
        // Get successors
        auto nSuccessors = problem.getSuccessors(exploring->gridPosition, successorBuffer);
        for(decltype(nSuccessors) i = 0; i < nSuccessors; i++) {
            const auto& successor = successorBuffer[i];
            auto existingIter = seenNodes.find(successor.node);
            if(existingIter == seenNodes.end()) {
                // Undiscovered node so far, add it and update vertexes
                auto addedNode = memoryPool.addNode(AStarSearchNode(nullptr, successor.node, std::numeric_limits<double>::infinity(), std::numeric_limits<double>::infinity()));
                seenNodes[addedNode->gridPosition] = addedNode;
                updateVertex(exploring, addedNode, successor.stepCost, open, problem, nullptr);
            } else {
                auto nodeInOpenSet = open.find(existingIter->second);
                if(nodeInOpenSet != open.end()) {
                    // Node is discovered, but still not closed, update vertices
                    as we might still have found a shorter way
                    updateVertex(exploring, *nodeInOpenSet, successor.stepCost, open, problem, &nodeInOpenSet);
                }
            }
        }
    }
}
```
Planning semi-autonomous drone photo missions in Google Earth
Fredrik Nilsson

}  
}  

// No solution found
return 0;
}
Appendix F: Path Smoothing

```cpp
size_t BPathSmoother::smoothPath(const Position3D* inputPath, size_t inputSize, Position3D** outputPath, double minResolution) const {
    // Begin by injecting new points every n metres if necessary to give the
    // smoother more points to smooth to
    std::vector<Position3D> densePath;
    densePath.push_back(inputPath[0]);
    decltype(inputSize) i = 1; i < inputSize; i++) {
        // Add intermediary points until we have at least the desired resolution
        while (Line3D(densePath.back(), inputPath[i]).flatEarthLength() > minResolution) {
            densePath.push_back(Line3D(densePath.back(), inputPath[i]).intermediaryPoint(minResolution));
        }
        // Add the next point
        densePath.push_back(inputPath[i]);
    }

    // Proceed with the smoothing
    std::vector<Position3D> newPathPoints;
    newPathPoints.push_back(densePath.front());
    const auto goalPoint = densePath.back();
    auto currentPoint = newPathPoints.back();
    size_t currentPointIndex = 0; // Indexing original array
    const size_t originalPathCount = densePath.size() - 1;
    size_t lowerSearchBound = 1;
    size_t upperSearchBound = originalPathCount;
    while (currentPoint != goalPoint) {
        // Classic binary search strategy for finding the furthest visible point
        // New upper and lower search bounds are set when this search is completed
        while (lowerSearchBound <= upperSearchBound) {
            const auto candidatePointIndex = lowerSearchBound + (upperSearchBound - lowerSearchBound) / 2;
            const auto candidatePoint = densePath[candidatePointIndex];
            const auto line = Line3D(currentPoint, candidatePoint);
            if (bounds.contains(line)) {
                // Point is reachable, search further
                lowerSearchBound = candidatePointIndex + 1;
            } else {
                // Point is unreachable, search closer
                upperSearchBound = candidatePointIndex - 1;
            }
        }
        // The furthest reachable point will be at the index of upperBound
        // (since the search will terminate when upperBound < lowerBound,
```
// which will occur when they have crossed over the target
// if upperBound = -1, no points were reachable, which could indicate a
// bug in the path finding algorithm
if(upperSearchBound <= currentPointIndex) {
    // Was not able to optimize this point, just trust the path finding
    // algo
    // in that the next point is still reachable
    upperSearchBound = currentPointIndex + 1;
}
currentPoint = densePath[upperSearchBound];
currentPointIndex = upperSearchBound;
newPathPoints.push_back(currentPoint);
lowerSearchBound = currentPointIndex + 1;
upperSearchBound = originalPathCount;
}

size_t resultSize = newPathPoints.size();
*outputPath = new Position3D[resultSize];
// C++ vector is copied to a C-array for compatibility with Swift the swift
layer
for(decltype(resultSize) i = 0; i < resultSize; i++) {
    (*outputPath)[i] = newPathPoints[i];
}
return resultSize;
Appendix G: Swift path finding interface

```swift
import Foundation

class PathFinder {
    enum Strategy {
        case AStar
        case Theta
        case AStarPS
        case ThetaPS
    }
    private let smoothingResolution = 5.0
    let pathFinderWrapper = PathFinderWrapper()!

    func findPathFrom(inout from: Position3D, to: inout Position3D, within bounds: CPPBounds, usingStrategy strategy: Strategy, optimizedFor performanceProfile: AircraftPerformanceProfile) -> Path? {
        var resultSize = 0
        var resultPtr: UnsafeMutablePointer<Position3D>? = nil
        if strategy == .AStar || strategy == .AStarPS {
            resultPtr = pathFinderWrapper.aStarSearch(from: &from, to: &to, inBounds: bounds.objcWrapper, descentPenaltyFactor: descentPenaltyFactor, climbPenaltyFactor: climbPenaltyFactor, resultSizeOut: &resultSize)
        } else {
        }
        if resultPtr == nil {
            return nil
        }
    }
```
if(strategy == .AStarPS || strategy == .ThetaPS) {
    // Post-smoothing
    var smoothingResultSize = 0
    let smoothResultPtr = pathFinderWrapper.smoothenPath(resultPtr, pathSizeIn: resultSize, smoothingResolution: smoothingResolution, inBounds: bounds.objcWrapper, resultSizeOut: &smoothingResultSize)

    // Deallocate the pre-smoothing array and replace the pointer with the pointer to the smoothed path
    // so the code below is agnostic to whether the path was PS'd or not
    resultPtr!.deinitialize(count: resultSize)
    resultPtr!.deallocate(capacity: resultSize)
    resultPtr = smoothResultPtr
    resultSize = smoothingResultSize
}

// Copy the result path into a Swift array and deallocate the result
var resultArray = [Position3D]()
var currentPositionPtr = resultPtr!
for _ in 0..<resultSize {
    resultArray.append(currentPositionPtr.pointee)
    currentPositionPtr = currentPositionPtr.advanced(by: 1)
}
resultPtr!.deinitialize(count: resultSize)
resultPtr!.deallocate(capacity: resultSize)
return Path(points: resultArray)
Appendix H: Nearest Neighbor

```cpp
bool NearestNeighbor::solve(const CostMatrix& costMatrix, unsigned start, unsigned* resultPtr) const {
    // Make a working copy of the costMatrix
    CostMatrix workingMatrix(costMatrix);
    size_t i = 0;
    resultPtr[i] = start;
    // n nodes gives n - 1 edges (not including the edge that connects first
    and last nodes)
    const auto matrixOrder = workingMatrix.getOrder();
    while (i < matrixOrder - 1) {
        auto from = resultPtr[i];
        unsigned next;
        double value = workingMatrix.getMinInRow(from, next);
        // If there is no open edge left, we failed to solve this problem
        if (value == std::numeric_limits<double>::infinity()) {
            return false;
        }
        // Select the best neighbor as next for this iteration
        resultPtr[++i] = next;
        // We will not go back to "from", and we will not go from it again
        either
            workingMatrix.markDeleted(from, from);
    }
    // Close the path by going back to the start point
    resultPtr[++i] = resultPtr[0];
    return true;
}
```
Appendix I: Existing Software and Hardware at Arrowcam AB

Computers

- Macbook Pro 15”, Early 2011, OSX 10.11.2
- Macbook Pro 15”, 2016, macOS 10.12.3

Tablets

- iPad Air A1475, iOS 10.2.1
- iPad Mini 2 A1489, iOS 10.2.1

Aircraft

- DJI Inspire 1 – T600
- DJI Matrice 600 Pro
- DJI Mavic Pro

Software

- DJI GO 3.1.5
- DJI GS Pro 1.2.0
- Autopilot for DJI Phantom, Mavic, Inspire, Matrice 3.9
Beskrivning av examensarbete med start LP 4 2017

Titel: Visualizing and planning autonomous photo missions using Google Earth

1. Personuppgifter
Namn: Fredrik Nilsson (frni1203), 19870125-1533

2. Tidsperiod
Slutförs maj 2017

3. Bakgrund

På flygdagen följer kunden oftast inte med för att granska bilderna ”på plats”. Eftersom en tydlig ”skiss” av hur bilderna ska se ut saknas så blir fotografen tvungen att ta ett stort antal bilder för att försäkra sig om att flygningen åtminstone resulterar i en bild som matchar kundens krav. Eftersom fotografen heller inte på förhand vet vilken position för drönaren som ger önskad bild går tid även åt att positionera drönaren för varje bild, vilket leder till lång flygtid, speciellt om många olika bilder ska tas.

4. Problem
- Hur kan 3D-position samt gimballäge för att ”ta” samma bild i verkligheten som bilden tagen i Google Earth beräknas?
- Hur kan uppdrag automatiskt planeras för att kunna ta flera bilder samtidigt som den totala flygtiden minimeras och hinderfrihet säkerställs?
- Med hjälp av svaren på ovanstående problem: Hur designas en applikation för att ge Arrowcam möjlighet att med gott resultat planera och delvis autonomt utföra fotoupdrag?

5. Metod
Existerande hårdvaru/mjukvarumiljö undersöks hos Arrowcam för att bestämma utvecklingsplattform.
företagets önskemål.

Unit-testing och versionshantering kommer att användas där så är lämpligt.

6. Resultat
En applikation (med eventuell middleware/script), som utifrån Google Earth-data minst kan skapa och flyga en färdplan, och ta ett flertal stillbilder. Finns tid läggs funktioner även till för filmsekvenser, och mer avancerade manöversekvenser, och/eller övriga funktioner som företaget ser sig behöva allt eftersom de testar applikationen.

En rapport skrivs även i enlighet med Mittuniversitetets riktlinjer, där bl.a. teori, arbetets gång och resultat redovisas.

7. Tidsplan

**Iteration 1**
Vecka 12: Undersökning hårdvarumiljö/mjukvarumiljö. Utvecklingsmiljö, API-Orientering
Vecka 13: Fastställa arbetssätt i Google Earth, hur definiera bilder? Hur definiera tillåtet flygområde?
Vecka 14: Verktyg för att visualisera drönarpositioner och färdplaner (för att underlätta validering/utveckling av algoritmer, kan även användas i GUI:t sen)
Vecka 15: Matematik för att bestämma nödvändig drönarposition för givna bilder. Rapportstruktur klar.
Vecka 16: Algoritm för att beräkna färdplan. Denna och tidigare algoritm beskriven i rapport.
Vecka 17: Arkitektur, Utveckling av GUI
Vecka 18: Utveckling av GUI + Testning i simulator
Vecka 19: Fälttestning och feedback. Första utkast till rapport

**Iteration 2**
Vecka 20: Vidareutveckling av företagets önskade funktioner
Vecka 21: Fälttestning. Sista utkast till rapport
Vecka 22: Slutredigering av rapport. Inlämning

8. Kontakter
Rikard Tyllström, Grundare Arrowcam, rikard@arrowcam.se, 070-738 31 75

Efter godkännande är detta ett kontrakt mellan dig som student och universitetet, vilket inte ensidigt får ändras.
Opponering i kursen DT133G – Examensarbete

Rapport skriven av Fredrik Nilsson, studentid fnri1203.
Opponent: Patrik Nygren, studentid pany0900.

**Layout och disposition av rapporten**

Det saknas uppgifter om kursen, handledare, författare och ämnesområde på rapportens inledande sidor.

Innehållsförteckningen ser lite konstig ut då vissa rubriknamn sträcker sig över flera rader.

**Terminology**

Här förklaras att JPS är ”Jump Point Search (Modification to the A* algorithm)”. Men jag saknar en förklaring till vad ”A* algorithm” är, även om den förklaras långt senare i teorikapitlet.

**Introduction**

När jag läser avsnitt 1.1 och 1.2 så blir jag inte helt säker på vilket (eller om det är båda) verktyg som man ämnar inhämta flygrutt ifrån, Google Earth eller Google Maps. Båda nämns och i senare delar är det enbart Google Earth som nämns varför jag förmodar att det är denna som gäller.

Tydligt och utförligt beskrivet om vad arbetet skall åstadkomma, framförallt vad det inte skall åstadkomma. Det hade varit spännande att se den färdiga prototypen och därför känns det viktigt att det klargörs tidigt och tydligt att denna inte kommer presenteras.


Avsnittet ”1.6 Contributions” är medtaget men innehåller ingen information.

**Theory**

Välbeskrivande avsnitt om algoritmer höjer förståelsen för de problem som man syftar till att hitta en lösning på. Kanske kunde dock mer bakgrundsinformation ha givits på andra områden, som till exempel en förklaring av XML (bakgrund till vad XML är, hur det fungerar med taggar och element).
Methodology

Bra förklarat om till exempel varför vissa algoritmer inte utvärderats och jag bedömer det som att man med den givna informationen i detta kapitel kan göra om arbetet och få ett liknande resultat.

Construction

Kapitlet verkar vara väl genomtänkt med förklarande kodexempel, med samma struktur som övriga kapitel (det vill säga att man bör läsa delproblem för delproblem tvärs över de olika kapitlen.

Result

Mycket trevligt med bilder som jämför den bild som finns i Google Earth med den som faktiskt tas med drönaren. Genom att inkludera dessa får läsaren en högst påtaglig uppfattning om hur väl resultatet föll ut i detta delproblem.

Resultaten av övriga delproblem presenteras även de tydligt för jämförelse.

Discussion

Författaren beskriver sina resultat väl och varför ett delproblem anses vara löst (eller delvis löst). Men, beaktat delproblem 2, även om resultatet lite talar för sig själv (man väljer ju förmodligen den lösning som ger bäst prestanda) så saknar jag information om de lösningar som förkastats. Hade de varit dugliga i sammanhanget trots att de var långsammare? Hade värdet av att hålla sig till en ”ren” Swiftlösning uppvägt en långsammare lösning i detta avseende?

Delproblemen kopplas naturligt tillbaka till motsvarande delproblem i tidigare kapitel, bra!

Bilagor

Informationen given i de olika bilagorna kompletterar väl rapporten i övrigt. Innehållet är dessutom välplacerat då jag anser att ifall den hade infogats i rapportens huvuddel så hade det gjort rapporten mer svårläst.