Efficient Path Planning Methods for UAVs Inspecting Power Lines

[1]Vytautas Rafanavicius, [1]Piotras Cimmperman, [1]Vladas Taluntis, [2]Ka Lok Man, [1]Gintaras Volkvicius, [1]Martynas Jurkynas, [1]Justas Bezaras
[1]*Baltic Institute of Advanced Technology, Lithuania, vytautas.rafanavicius@bpti.lt, piotras@bpti.lt*
[2]*Xi'an Jiaotong-Liverpool University, China*

Abstract - *Power line inspection is one of the most difficult and time consuming steps in power line maintenance. Even for a sizeable group of workers it takes months to inspect all of them, especially when they are not visible from the road and must be inspected on foot or with an aerial vehicle. That problem is even more prominent when the inspection must be done as fast as possible when the power cuts out in certain regions after natural disaster. To save time and reduce expenditure Unmanned Aerial Vehicles (UAV) could be used to film the power lines and automatically find problems (e.g. a broken cable or a tree branch too close to the a line). Our research focuses on planning the route for the survey.*

Keywords*: UAV, power line, path planning, graph*

1. Introduction

The power lines inspection by its importance is divided into two:

Periodic. A team is sent to inspect electricity lines and to look for defects and objects too close to the lines [1],[2],[3]. They report problems to headquarters where they plan further actions and send a team to fix the issues.

Critical. In a state of emergency, when a line is broken after a storm or other events, a helicopter is sent to find the faulty power line (if the exact position of the fault is not reported by the local people). They report problems to the headquarters where they make a decision to send teams to reported locations in order to fix the issues.

Before any UAV is could be used, a path with GPS coordinates must be loaded in its board computer. However, the structure of the power line is not just a line, in most cases it has a number of branches. If UAV relies on fuel or batteries it is a demand to have the most efficient path to cover as many kilometres of the power line as possible. To plan a path the structure of electricity poles and lines have to be simplified to one that could be easily processed by computer to do any calculations needed. The easiest way to simplify the structure is to make it a graph. In mathematics, and more specifically in graph theory, a graph is a representation of a set of objects where some pairs of objects are connected by links. The interconnected objects are represented by mathematical abstractions called vertices (also called nodes or points), and the links that connect some pairs of vertices are called edges (also called arcs or lines). Electricity poles could be vertices and the lines that connect them could be edges. Planning a path for UAV is easier than for any ground vehicle, because there is much lower possibly of obstructions in air space than on ground, so there is no need to take account on any obstructions.

Depth-first search algorithm. Depth-first search is an algorithm for traversing or searching tree or graph data structures. One starts at the root (searching some arbitrary vertex as the root in the case of a graph) and explores as far as possible along each branch before backtracking. In this case, search would have to go through all graph's vertices (electricity poles). First it is necessary to mark all vertices from 1 to n (n – is the number of vertices in the graph). Then start trip from the vertex number one, mark it as visited and continue trip to the vertex of last visited vertex′s environment, which has the lowest number and is not visited. If trip ends up in vertex that all his environment vertices are visited, search goes back until it finds not visited one. In the end, search ends up at the first vertex [4]. The pseudo code of the recursive type of this algorithm:

```
procedure DFS-iterative(G,v):
    let S be a stack
    S.push(v)
    while S is not empty
        v = S.pop()
        if v is not labeled as discovered:
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Shortest Path Search in Unweighted Graph. This algorithm finds the shortest path from the start vertex to other vertices. It uses Breadth-First search (BFS) algorithm. BFS is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary vertex of a graph) and explores the neighbour nodes first, before moving to the next level neighbours [5]. In overall, this algorithm is great and fast solution for finding shortest paths from one point to another, which includes all points and lines in its way. In addition, this algorithm does not consider distances between vertices, so it is simpler and less time consuming than the other ″shortest paths″ algorithms that will be described next.

978-1-5090-5140-3/17/$31.00 ©2017 IEEE

Dijkstra′s algorithm. It is an algorithm for finding the shortest paths between vertices in a graph, which may represent, for example, road networks or electricity pole networks. This algorithm in principal is very similar to the previous one, it's just that weights of edges are put in the distance array. Also algorithm replaces existing value in array distance array if it calculated the shorter path by going the another way [6]. In this algorithm, closer vertices has higher priority. In addition, instead of queue that is used in BFS algorithm, this algorithm use another array that contains information if it already have visited vertex or not. The search ends after it visits all vertices [7].

This algorithm is perfect if distances between the points are different. In case all distances between certain points are known, this algorithm can find the shortest paths. If the distance between the points are the same or varies slightly, it is better to use Shortest path search in unweighted graph algorithm for finding shortest paths, because it is less time consuming and has better performance.

Fleury′s Algorithm. In graph theory, an Eulerian path is a trail in a graph which visits every edge exactly once. Similarly, an Eulerian circuit or Eulerian cycle is an Eulerian trail that starts and ends on the same vertex. Leonhard Euler first discussed it while solving the famous Seven Bridges of Konigsberg problem in 1736. The main condition for the graph to have either an Eulerian path or Eulerian cycle is that either every single of its vertices have the even degree (a number of edges touching the vertex) or just two of them have odd degree.

The algorithm for finding either an Eulerian path or Eulerian cycle for the graph that meets the condition is the Fleury′s algorithm. With this algorithm, consider a graph known to have all edges in the same component and at most two vertices of odd degree. The algorithm starts at a vertex of odd degree, or, if the graph has none, it starts with an arbitrarily chosen vertex. At each step it chooses the next edge in the path to be one whose deletion would not disconnect the graph, unless there is no such edge, in which case it picks the remaining edge left at the current vertex. It then moves to the other endpoint of that vertex and deletes the chosen edge. At the end of the algorithm, there are no edges left, and the sequence from which the edges were chosen forms an Eulerian cycle if the graph has no vertices of odd degree, or an Eulerian trail if there are exactly two vertices of odd degree [8].

Hierholzer′s algorithm specializes in finding an Eulerian cycle in the graph, so the graph has to meet the condition of even vertices degrees to work. The sequence of the algorithm is as follows: any starting vertex *v* is chosen and a trail of edges from that vertex until returning to *v* is computed. It is not possible to get stuck at any vertex other than *v*, because the even degree of all vertices ensures that, when the trail enters another vertex *w* there must be an unused edge leaving *w*. The tour formed in this way is a closed tour, but may not cover all the vertices and edges of the initial graph. As long as there exists a vertex *u* that belongs to the current tour but that has adjacent edges is not part of the that tour, it has to start another trail from *u*, following unused edges until returning to *u*, and join the tour formed in this way to the previous tour [9].

To sum up, this algorithm has far better performance rate for finding an Eulerian cycle.

2. Methodology

Data and Methods for Path Planning. In this project, open source software *Mission Planner* was used, in which route could be planned by simply selecting electrical poles one by one. Exists a few ways to plan the route in *Mission Planner*. One way is to simply select points on the map in order in which they should be visited. The other way is to upload a file with the coordinates of the poles. This file contains lines with points that will be visited. In first line, there is first point with its coordinate, in second line second point and so on. This way is much more efficient.

Reading data of electricity poles. First, some sort of file is needed that would contain each electricity pole′s coordinates and information about which poles are adjacent to it. To be efficient, create some sort of data structure to instantly put all poles in while reading them from a file. Two simple data structures could help. First data structure, call it *Position*, will have just two attributes − two doubles for containing coordinates. The second data structure, call it *Pole*, will have three attributes − two doubles for containing coordinates and a vector list of *Position* data's structure objects, that will contain all adjacent poles for the *Pole* object. To even more simplify reading data from file, create a file that would already contains GPS coordinates of the poles and the adjacent ones, so that there would be no need to convert them from any other format.

These two data structures are good for reading data from coordinate file, but not very good for searching for paths. As there was mentioned in path planning methods section − when searching for the path, a graph with two arrays has to be defined: *L[]* and *\lst[]*. First, *\L[]* array will contain adjacent poles from *Pole* structure, and each member of *lst[]* array will contain the number of adjacent poles from current *Pole* structure plus the previous array number. For example, there is 5 electricity poles as in Fig. 1, each named from 1 to 5.

The connection between each pole describes this structure of adjacency:

- 1: {2,4}

- 2: {1,5}
- 3: {4, 5}
- 4: {1, 3}
- 5: {2,3}

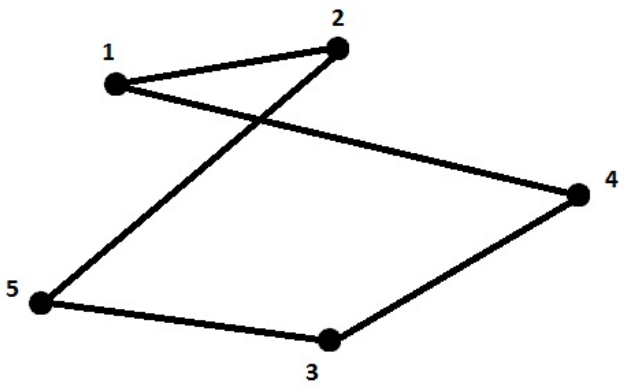

Fig.1 Example of electricity poles graph

Array *L[]* looks like this: 2 4 1 5 4 5 1 3 2 3
Array *lst[]* looks like this: 0 2 4 6 8 10

First member of *lst[]* array describes that there is a graph and this is essential because second number will be calculated as this member plus the adjacent poles for the first pole. Every other number in general is calculated as a previous member plus adjacent poles for the current pole. With this form it becomes like address array: first pole's adjacent poles from array *L[]* are in an interval *(0;2]*; second pole's adjacent poles from array *L[]* are in an interval *(2;4]* and so on.

Formatting an output file. After applying a path planning method an array will be formed, in which the poles that are needed to be visited will be written in visiting order. Now there is a need to print them in lines with any essential information needed for *Mission Planner* to read the output file correctly. Because the visiting order was calculated by the algorithm, program will just search for each member of the path array in *Pole* structure array and when the match is found will print all the information needed in *Mission Planner* syntax.

Overall program creation. An overall program creation could be shown from a programmer point of view with an UML diagram in Fig. 2.

The goal is to visit all poles and all lines, the best algorithm to choose is Hierholzer's algorithm for Eulerian cycle search. Of course, some corrections to main data should be made if it does not meet Hierholzer's algorithm requirements. For this algorithm implementation and overall program writing, choose C++ as program′s programming language and *Microsoft Visual Studio 2013* as software for creating it. For test, our own built unmanned aerial vehicle was used, which is shown in the Fig.3. It has integrated PIXHAWK autopilot system, which controls the plane. This micro-

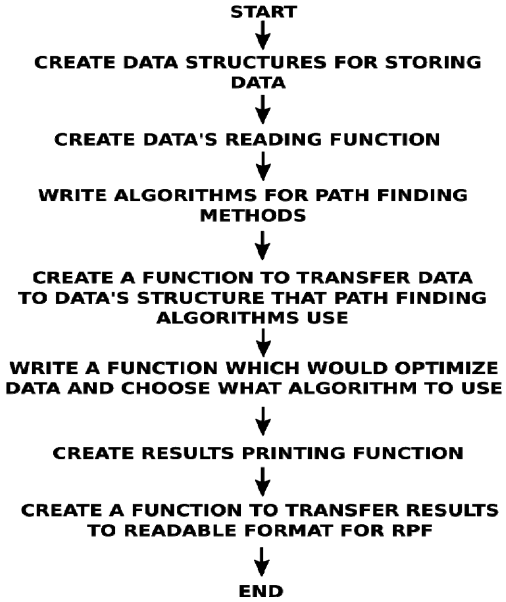

Fig.2 UML diagram of the computer software

controller has a 32-bit Cortex M4 core processor and a set of integrated sensors:

- ST Micro L3GD20H 16 bit gyroscope
- ST Micro LSM303D 14 bit accelerometer / magnetometer
- Invensense MPU 6000 3-axis accelerometer / gyroscope
- MEAS MS5611 barometer

Fig.3 Photo of Unmanned Aerial Vehicle used in this project

UAV was powered by lithium-ion battery. Battery specifications are the following:

- Capacity (mAh): 10000
- Voltage (V): 14.8
- Weight (g): 804

The plane has a gopro hero 3+ camera on board. During the test flight, camera was configured on high quality resolution: 4096 x 2160 and 12 fps video format.

The Flight. Video was filmed from about 70 m height and pointing perpendicularly to the ground. The plane had to visit all electricity poles in its flight path. The UAV was designed for belly landing. The automated landing procedure goes as follows: after the last checkpoint of the mission the descent phase begins. The autopilot turns the plane upwind, throttles down the engine and starts the descend. An important parameter is the glide slope which is the ratio between distance and height from the last checkpoint and desired landing spot. The recommended glide slope depends on the plane model. At a certain height or time to landing threshold the plane flares to increase drag and slow itself before contact with the ground. This method of landing is most useful to us because it is already implemented in Mission Planner, to use it one just needs to specify the landing point and altitude at the end of the route as well as to tweak some settings according to the plane the altitude to flare at, the time to flare before the ground (redundancy is for safety measures) and minimum pitch target.

3. Experiment, Simulation and Analysis

Path planning software. The main goal was to create a program which could quickly plan the shortest and the most efficient route among selected electrical poles.

The program was written using *Microsoft Visual Studio 2013* software in C++ programming language. The program uses Hierholzer's algorithm to find shortest path and data's optimization algorithm to make data valid. This program creates and works with data structures called *Position*, *Pole*, *Structure* and *virtualEdges*. In *Position* the program saves each pole's GPS coordinates on the map (longitude and latitude). In *Pole* it saves each pole's position and information on what other electricity poles that it is connected to. *Structure* contains an array of *Poles* and its status (either it was visited or not). *virtualEdges* is an optimization structure.

Hierholzer's algorithm implementation. Firstly, the data is valid - all vertices degrees are even. Trail starts from first vertex that is in the structure of adjacency and continues in order of the structure of adjacency until it gets back to the beginning (all vertices degrees are even, so that it cannot stuck anywhere else). While travelling vertex by vertex, algorithm deletes edges that just have been travelled through from the structure of adjacency. Also it marks which vertices it visited and gives priority for those vertices that are not yet visited (if all vertices were visited from given vertex, algorithm travels to the first one of structure of adjacency). When it gets back to the beginning it looks if there are any unvisited vertices. If so, it starts new cycle from it and when finishes it, inserts this tour into the main tour.

With valid data, Hierholzer's algorithm can handle route planning itself. A planned route from valid data in Fig. 4.

While planning this route, algorithm first planned the main big circle and then with second, third and fourth loop planned two triangles and rectangle and inserted them in the main route.

Data optimization algorithm implementation. Implemented Hierholzer's algorithm can plan route when data is valid (all vertices degrees are even). However, in reality could happen that data will not be valid, so there is a demand to write data optimization algorithm, which could make all vertices degrees even from given structure of adjacency by connecting odd degree vertices in pairs so that the sum of new created edges length is shortest. At the start, the algorithm check if there is any odd degree vertices. If so, it recursively, one after another, find all available pairs of vertices and save just the shortest one. To calculate distance between vertices (which are defined in GPS coordinates) algorithm uses haver-sine formula, which count distance between two GPS points.

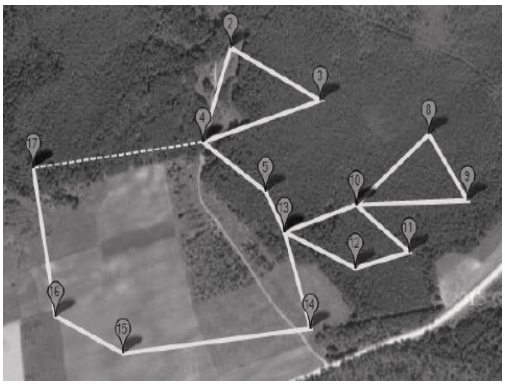

Fig.4 Planned route from valid data

When data is invalid, data's optimization algorithm adds virtual edges to connect odd degree vertices. There is a planned route on electrical poles and lines, which originally formed Y form, in Fig. 5a.

Before planning this route, data's optimization algorithm connected 5^{th} with 4^{th} and 7^{th} and 8^{th} vertices correct data. While first connection can be clearly seen, the second one is on the path in the map that already exists. As one can see in Fig. 5b, when path is moved a little bit, the clear route appears.

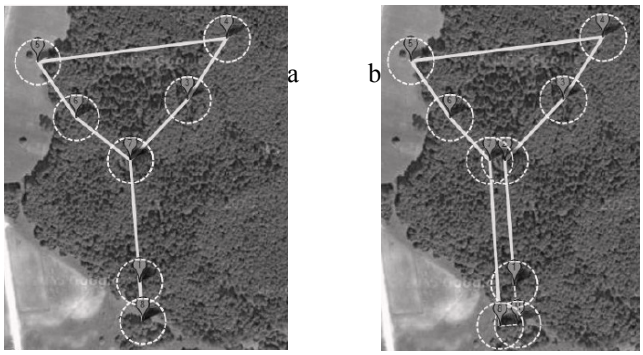

Fig.5 Planned route using not valid data (a), moved planned route using valid data (b)

This program is a good tool for planning shortest routes. Although program's performance rate is $O(n^3)$, it could upgradeable to as much as $O(n^2)$. Another problem could be that there is no special algorithm in case a given structure of adjacency indicates multi-part graph, so it is possible that program would crash or would not plan the route fully when this kind of data is given.

Path optimization in HIL. Before real test, UAV was tested in Mission Planner auto pilot to find optimal settings for flight by running a hardware-in-the-loop (HIL) simulation over "X-Plane" software. The goal was to find the way-point parameters and auto-pilot controller settings that would give the most accurate and stable flight right over intended path. After a test flight with the default Mission planner settings it was found that they are accurate enough to fly over well-placed way-points set in Mission Planner not deviating too far from electricity lines. In the simulation the plane flew over straight paths perfectly and over way-points placed about 200 meters apart with no sharp turns it flew with minimal error as seen in Fig. 6. On the other hand, the plane deviated a lot from the optimal course on sharp turns taking over 250 meters to get back on course as seen in Fig. 7.

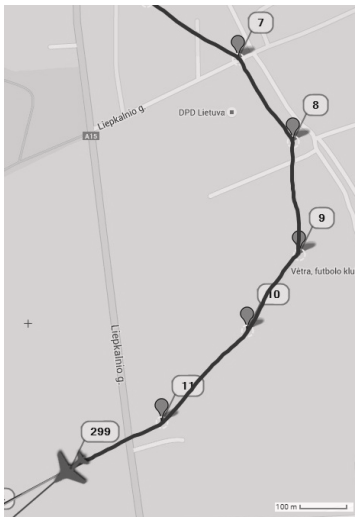

Fig.6 Straight flight and low-angle turnIn these illustrations the red plane icon represents UAV's position at the time the picture was take and the number next to it shows it's altitude. Thin lines coming out from the front of the plane represent some flight data. Red line is current flight direction, orange is pointing at the following checkpoint and black, pink, green are turning guidelines. Green bubbles are way-points and their order is represented by their numbers. Way-points are connected by a straight bright yellow line. And purple line is the simulated flight path. The occasional sharp edges on the flight path are tracking errors due to map shifting and they do not represent actual flight path at these out-of-place points. In these pictures the flight path did not go over some way-points exactly because it was allowed for the plane to miss the way-point by 20 meters. When radius is set too small, UAV has sometimes to turn around due to missing a checkpoint, especially when they are placed too close to one another.

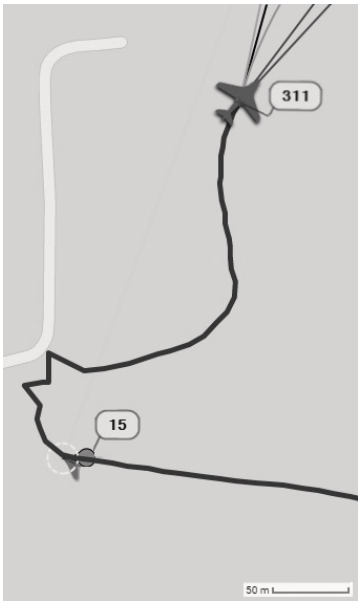

Fig.7 Flight path after a sharp turn.

To eliminate problems after sharp turns, loops around the turns were created [10]. As seen in Fig. 8, placing one way-point 50 meters behind a sharp turn (way-point 5) was not enough to make the aircraft return to the course correctly. Creating a circle of way-points (wp. 10-17) with radius of about 50 meters after a sharp turn turned out to be enough to make the UAV return to the intended path at a correct angle.

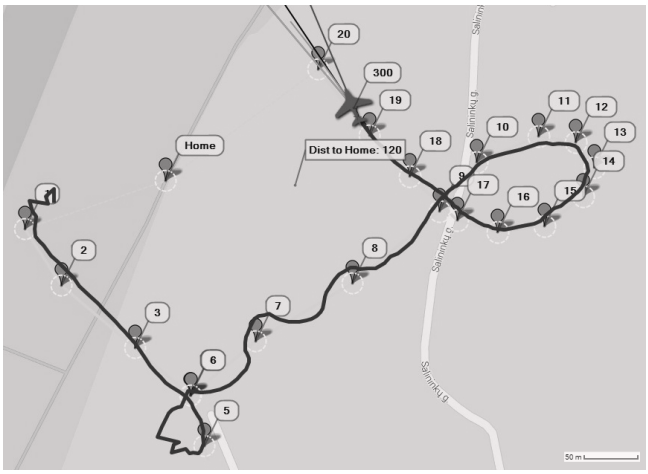

Fig.8 Solution to the Sharp turn problem.According to these results, after smoothing sharp turns with outer-circles, the UAV is able to fly right over the electricity lines with only a small error after turns. Although that will be minimal when inspecting long and straight power lines.

4. Conclusions

It was shown that UAVs could be used for automated power line path tracking with proper software, hardware and take-off / landing area. It would save work time, allows more frequent periodical power line inspections and more responsive critical time inspections. Fully developed system would require minimal human interaction and would eliminate the need for people to travel to hard-to-reach locations just to check for possible faults.

Selected References

[1] I. Golightly and D. Jones, ″Visual control of an unmanned aerial vehicle for power line inspection,″ 2005, pp. 288–295.

[2] S. J. Mills, ″Visual Guidance for Fixed-Wing Unmanned Aerial Vehicles using Feature Tracking: Application to Power Line Inspection,″ Queensland University of Technology, 2013.

[3] J. Zhang, L. Liu, B. Wang, X. Chen, Q. Wang, and T. Zheng, ″High Speed Automatic Power Line Detection and Tracking for a UAV-Based Inspection,″ 2012, pp. 266–269.

[4] R. Tarjan, ″Depth-First Search and Linear Graph Algorithms,″ SIAM J. Comput., vol. 1, no. 2, pp. 146–160, Jun. 1972.

[5] U. Brandes, ″A faster algorithm for betweenness centrality*,″ J. Math. Sociol., vol. 25, no. 2, pp. 163–177, Jun. 2001.

[6] F. Schulz, D. Wagner, and K. Weihe, ″Dijkstra′s Algorithm On-Line: An Empirical Case Study from Public Railroad Transport,″ in Algorithm Engineering, vol. 1668, J. S. Vitter and C. D. Zaroliagis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 110–123.

[7] M. Barbehenn, ″A note on the complexity of Dijkstra′s algorithm for graphs with weighted vertices,″ IEEE Trans. Comput., vol. 47, no. 2, p. 263, Feb. 1998.

[8] M. Fleury, ″Deux problèmes de Géométrie de situation,″ 1883.

[9] H. Fleischner, X.1 Algorithms for Eulerian Trails, Eulerian Graphs and Related Topics: Part 1, Volume 2, Annals of Discrete Mathematics 50. Elsevier, 1991.

[10] B. Carlo L., S. Barbara, and L. Domenico, ″Path Planning for Autonomous Vehicles by Trajectory Smoothing using Motion Primitives,″ IEEE Trans. Control Syst. Technol., vol. 16, no. 6, pp. 1152–1168, Nov. 2008.