Autonomous Pod-Based UAV Trajectory Prediction For Expedited Heat Seeking

Sargis S Yonan

syonan@ucsc.edu

Computer Engineering 198F

Independent Study

Advising Director Patrick E. Mantey

Spring Quarter 2017

University of California, Santa Cruz

Abstract

Using an autonomous pod of unmanned aerial vehicles, we can take advantage of the fact that the UAVs can share visual data with each other in the air to better understand an area of land. If the sensors placed on board each UAV in a pod detected an intensity of heat for example, we can mark these intensity levels in a matrix, where the location of each entry in the matrix represented some earth coordinate. We can then run some learning algorithm on the entries of the matrix to better predict where more hotspots (or any general point of interest) are located, if the hotspots tend to be near each other (in the case of a fire for example).

Case of Use Example

A team of firefighters could use such a framework equipped with infrared cameras to scan a forest during a wildfire to examine where the fire currently is and where the fire is spreading to via a live relayed overlay on top of a map. Due to the nature of this kind of problem, we need to use Learning to avoid “dumb-scanning” an area (zigzagging or sweeping through some area with no intention of finding something of interest). Hot spots need to be scanned sooner than later in the case of an emergency fire, and using a learning paradigm should be used to motivate the direction of travel for the UAVs in the direction of areas that are predicted to be hot. This way, the hot spots are scanned first, and in the case of the firefighting example, the fire can be extinguished there sooner.

Methods

Data Set

If we have N UAVs in a pod, we can have the drones independently gather aerial hotspot data and form packets of time, Earth coordinates, and interest level in real time. They can then share these packets with one another, and independently form their own “hot spot” maps. They can then use this information to decide on which areas to scan next.
Our feature list is a matrix of “hotspots”. The data can be displayed in a matrix of size $n \times m$. Each entry (area) on the matrix represents a scanned or unscanned area in the scan perimeter, equal to the area of the map pixel, M_{ij}, representing it.

M_{11}	$\cdots$	M_{1m}
$\vdots$	$\ddots$	$\vdots$
M_{n1}	$\cdots$	M_{nm}

Initially the M matrix will be a $-1_{n\times m}$ matrix of dimension $n \times m$. When the drones begin scanning the area, the M entries will be populated with a level of interest from 0 to K, where 0 means nothing of interest was detected, and K is the maximum possible value of interest. We can approximate the width of each entry to be approximately the same latitudinal distance (since the curvature of the Earth is negligible for our relatively small UAV scannable regions), and the length of each entry can represent some longitudinal distance.

Implementation & Methods

I used a K-Nearest Neighbors method to predict whether our unscanned -1 entries (which correspond to some real area with a bound of longitudinal and latitudinal coordinates), are "hotspots" or not. If some unscanned area is predicted to be a hotspot, the drones will have more weight in choosing to move in that direction.
I sought to determine if the KNN method would be a successful method for predicting where hotspots lie, and whether or not it could predict and assist the drones that helped the algorithm. I used Python (and the MatPlotLib and NumPy libraries) to assist me in the task of creating a multi-drone simulation framework, and graphing the results live.

Assumptions

Since this is a pod of drones flying together with the ability to share data packets in the air, we will take advantage of the fact that the UAVs in the pod can help each other quickly scan an area by sharing data about what they see, namely the coordinate locations (from GPS) of the points of interest (from a sensor pointing down to the ground).
We will then assume that a packet transmission protocol is on-board each UAV, and that they can send, receive, and parse these packets and store the points and interest-levels in an internal map in memory.

Algorithm

Initially, the drones know nothing of the field. Before they set off for flight, I initially set a bloom of high intensity flame on the map, and the drones then went off on flight. The UAVs initially linearly scan an area (sweeping left to right, up to down) until one of them found a hotspot and shared the data. The drones then computed a KNN (here, $K = max(n, m)$ in order to take the whole map into account) to decide on which direction to move to next (up, down, left, right, and the diagonals). I then plotted the prediction map, along with the drone point of view map. The collective map of the scanned area of the drones is plotted live in the simulation.

Every iteration of the scan, each drone will independently calculate the total map to predict the weights.

Let M be an $n \times m$ matrix held in the program memory of the algorithm, which represents a map of predictions of hotspots, where each entry represents a prediction of how "interesting" a coordinate is. Initially, we set $M =- 1_{n\times m}$, a matrix of all -1 valued entries. We then allow the drones to "dumb scan" an area (sweeping left, right, up, and down along the perimeter of the map), while they scan the area with whatever sensor they have on-board. The drones scan a sub-area (the ij-th entry of the matrix M) each iteration of the algorithm and find that that sub-area has an interest level l between 0 and K, $\eta_{ij} = l,\ l \in [0, K]$.

For an $M \in \mathbb{R}^{n\times m}$ that represents a predicted map where each entry is a prediction or measured value of interest intensity:

$$M_{ab} = \sum_{i=1}^{n} \sum_{j=1}^{m} \eta_{ij} \|\phi_{ij} - \rho_{ab}\|_2$$

Where η_{ij} is the interest point intensity of the ij-th vesicle (entry in the matrix) on the map. ϕ_{ij} is a vector that has the i,j coordinates in a 2-vector i.e. $\phi_{ij} = [i\ j]^T$. ρ_{ab} is the vesicle on the map the weights are being calculated into.

Now, if the a UAV is currently at the point $[a\ b]^T$, the UAV will move to the most weighted position in the set:
$B = \left\{[a+1\ b]^T,\ [a\ b+1]^T,\ [a+1\ b+1]^T,\ [a-1\ b]^T,\ [a\ b-1]^T,\ [a-1\ b-1]^T\right\}$, as long as the point is in the map and not previously traversed upon.

Since the UAV, in this algorithm, is limited to moving in 6 different positions (|B|), M_{ab} needs to be calculated only 6 times to see which direction has the most weight.

Results

Apart from getting the machine learning part to work, I had to first create the drone flying over fire test simulation framework. I also wanted to make sure that the drones scanned the entire field, and not starve parts of a map in favor of hotspots that were already found. This required using hysteresis on already traveled-to coordinates, and putting less weight on moving to spots that were already traveled to.
For KNN, I set K equal to the square of the size of the grid. In the end, I was impressed with the agility of the algorithm to find the hotspots initially, and still scan the rest of the map.
KNN worked well for giving us relative probabilities for every point on the map, where unsupervised learning was not necessary. Images of the results are below, and animations along with the source code of the simulation can be found at: `https://github.com/SargisYonan/uav_ml`

The simulation can be run on your own machine by executing the following commands in a shell (dependencies: Python, matplotlib, numpy, git for cloning the repository):

```
$ git clone https://github.com/SargisYonan/uav_ml
$ cd uav_ml
$ python main.py
```

The points of the fake bloom of high interest level, the number of UAVs in the pod, and the size of the map are all configurable parameters in main.py.

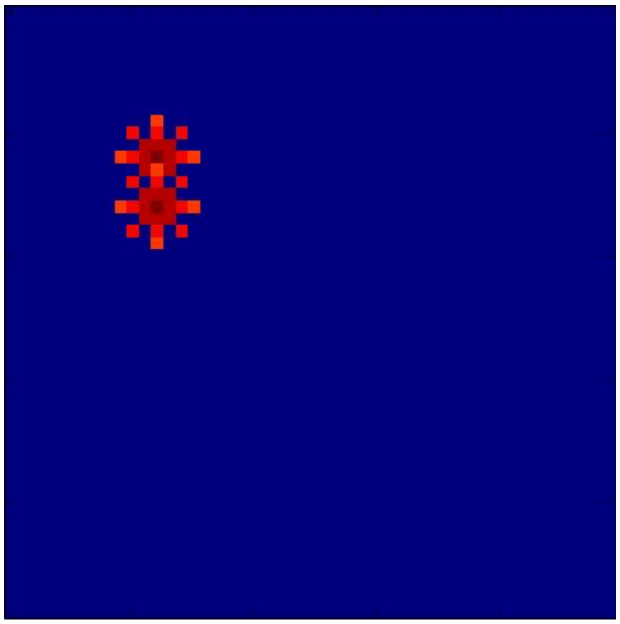

Figure 1 *The master map of where the fire is located. This information is unknown to the UAVs initially until the map is completely scanned.*

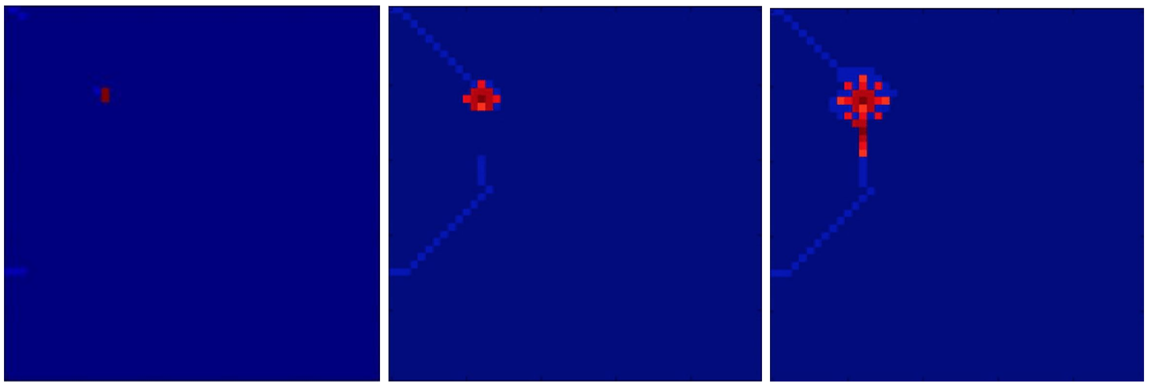

Figure 2 *Track the steps in finding the fire (3 drones)*

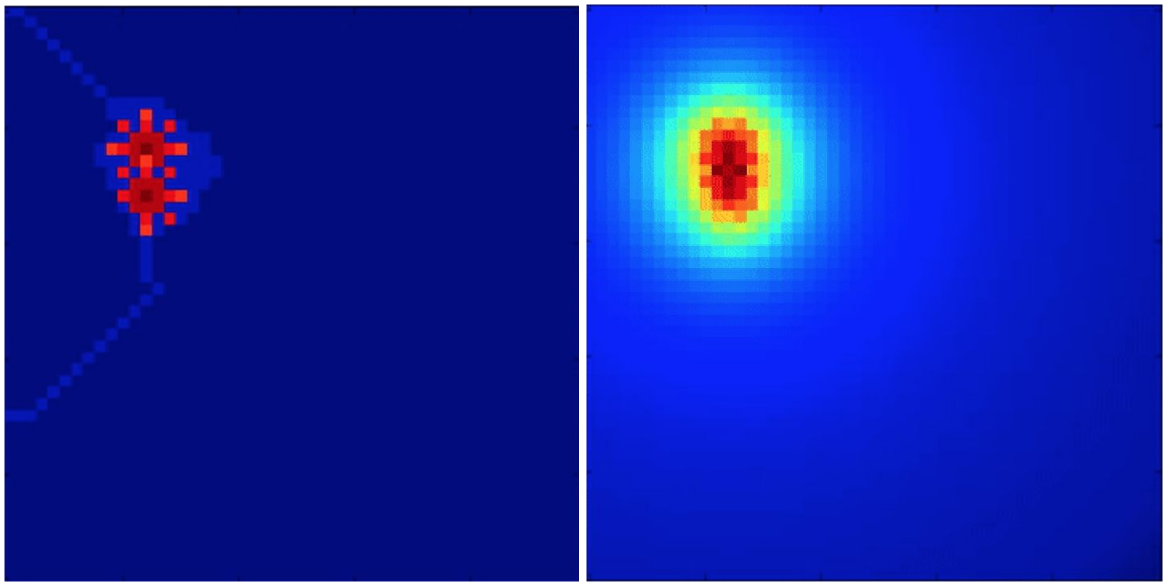

Figure 3 *Here the 3 drones found the bloom, and are moving on to scan the rest of the map (left). The KNN predicted map of the probabilities of where heat is coming from is on the right.*

Runtime Efficiency

Recalculating the map every iteration of the of algorithm is very computationally expensive. The algorithm to calculate $M \in \Theta(max(n,m)^2)$ is something on the order of magnitude that might be infeasible for a low-powered microcontroller to handle while also estimating UAV attitude and running a stabilization controller. This is why this algorithm would most likely require a more computationally capable controller on-board the UAV, or also offload the algorithm to a ground station that would then report back a coordinate to fly to.

Possible Optimizations

Currently, the algorithm runs quickly initially because my implementation takes advantage of the fact that M is a mostly sparse matrix with "undefined" scan points. The matrix is stored as a linked-list of linked-lists and simply ignores the unscanned regions until they have been scanned. As the matrix M fills up, however, the algorithm is noticeably sluggish because more points must be factored in the KNN computation.

To further optimize this algorithm, the map can be split into a sub-map and then be computed on. We can choose:

$S \in \mathbb{R}^{f\times g} \subseteq M \in \mathbb{R}^{n\times m}$, where $f \le n,\ g \le m$ (i.e. set K-Nearest-Neighbor's K to a smaller value) and then compute

$$S_{ab} = \sum_{i=1}^{f}\sum_{j=1}^{g} \eta_{ij}\|\phi_{ij} - \rho_{ab}\|_2$$

$$M \in O(max(n,m)^2) \ge S \in \Theta(max(f,g)^2)$$

Which would increase the speed of the algorithm by some factor of $(m-f) \times (n-g)$.

Future Work

I would like to include more learning abilities for pods of UAVs. Another possibility of the pod is to have the drones take off at set times in the day and night, scan the same area of land each time, and fly back home to charge and repeat. The pod would then "learn" the area for different times in the day (with possibly different sensors) during different times in the year. They would be able to alert a user of an anomaly that was detected. This is because they would have "learned" the area of land that they typically scan for different times of the day and year, and could therefore detect if something is "out of place".

Conclusion

Though I was very satisfied with the results of the algorithm, the processing times of the KNN algorithm as the epochs increased were unbearably slow for the environment in which they would ideally run on (low-powered embedded systems that would only be computing attitude estimations, PIDs for flight stabilization, and simple packet transmission/parsing). In order to make this algorithm "fly", a decent architecture would have to placed on board, and a dedicated process, controller, or thread would have to be devoted to running just this algorithm.