Paragliding Competition Tracklog Optimization

Ondřej Palkovský

March 1, 2010

Abstract

Paragliding competitions require finding a largest shape (triangle, broken line) in a set of points discribing a path flown. Brute-force methods of finding the optimal results have a time complexity of $O(N^5)$. Considering that the new GPS devices easily generate tracklogs exceeding 30000 points, a different approach has to be taken. This article describes an algorithm that exploits some geometric properties of the problem. The algorithm is very fast with a reasonable time and space complexity in most cases.

1 Introduction

Cross country paragliding competitions pose an interesting optimization problem; given an ordered set of points describing the path, find a set of points that would form a simple shape (triangle, broken line) and maximizes some function (circumference, length of line). The brute-force way of solving these problems is approaching time complexity of $O(N^5)$.

The typical three problems being solved in a pragliding competition are:

- Find a longest broken line composed of 5 successive points.
- Find 5 succesive points s, p_1, p_2, p_3, e that would maximize a function:

  $$c(p_1, p_2, p_3) - d(s, e)$$

  where $c(p_1, p_2, p_3)$ is a circumference of a triangle and $d(s, e)$ is the distance between the points. If

  $$d(s, p)/c(p_1, p_2, p_3)$$

  is greater then a given constant (usually 0.05 or 0.2), the function returns zero.
- So-called FAI triangle is the same as above with the added constraint that the smallest side of the triangle must not be smaller then 28% of the circumference of the triangle.

The time complexity of the problem is agravated by the fact that all coordinates are spherical. In order to compute the distance between two points, the computer must perform twice a sine function, 3 times a cosine and once arc cosine. Although a good computer today can do about 25 milions of goniometric operations per second, it takes about 50 seconds to compute distances between all points. To speed up the optimization we could theoretically store the distances in the memory; for a tracklog 30000 points we would need 3.5 gigabytes of memory. Although such amount of memory is slowly being adopted even for consumer machines, it is at least inconvenient for normal use.

In practice we can choose from 2 types of number representation - 4 byte floating point or 8 byte floating point. Although the operations on the 4 byte floating point numbers are about twice as fast, the resulting precision loss is so high that it renderes the results unusable.

The approach taken in this paper is to exploit the geometric properties of the input data to minimize the amount of goniometric operations. This allows us to cut significantly both speed and space complexity in most cases.

2 Optimization algorithms

2.1 Dynamic programming approach to broken line

The broken line problem can be solved efficiently using dynamic programming approach in $O(n^2)$ time. The dynamic programming approach is used in situations where a part of an optimal solution is itself an optimal solution. For a free flight over 3 turnpoints, the path to the first turnpoint itself is an optimal solution to the problem "longest path over zero turnpoints to this point", the path to the second turnpoint is an optimal solution to the problem "longest path over 1 turnpoint to this point" etc.

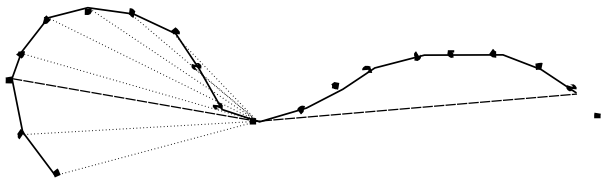

Figure 1: Part of the optimal solution is itself an optimal solution

We start with a table E_0 containing zero values. Then in every step we create a table E_s using the equation:

$$E_s[i] = \max_{1 \leq j \leq i} [E_{s-1}[j] + distance(p_i, p_j)]$$

where p_i is the i-the point of a tracklog. Element $E_1[i]$ thus contains the longest straight line leading into point p_i from some point preceeding the point p_i. Element $E_2[i]$ contains the longest broken line over one turnpoint leading into the point p_i. Element $E_5[i]$ contains the longest broken line over 3 turnpoints leading into the point p_i.

In order to compute the free flight over 3 turnpoints, we compute:

$$\max_{1 \leq i \leq n} E_5[i]$$

where n is the number of points in a tracklog.

A modification of this algorithm allows us to compute $distance(p_i, p_j)$ only once without the need to store it in memory between the iterative steps. Computing a free flight over 5 turnpoints of a tracklog of 30000 points using this more efficient method takes today about 50 seconds on a good computer.

2.2 Branch and Bound Optimization

Although the previous method is not particularly slow, it cannot be adapted to the triangle problems. A simple branch and bound method seems apropriate. The branch and bound

algorithm uses branching method to split a set of candidate solutions and a bounding method to find out upper or lower bound of a solution. Based on the upper or lower bound the algorithm discards a whole set of candidates without needing to inspect every one in detail.

In our case the bound function will return the maximum *score* function for the shape, in a given set of candidates. The branch and bound algorithm we will use is:

1. Find a candidate set with heighest maximum score.
2. If the candidate set contains exactly 1 solution, return it.
3. Otherwise branch the candidate set into smaller sets.
4. Repeat.

In order to efficiently find the candidate set with the heighest maximum score, we will maintain a binary tree with certain properties:

- candidate sets have no children
- node is either a candidate set (0 children) or it has 2 children
- $score(leftchild) > score(rightchild)$
- $score(parent) = score(leftchild)$

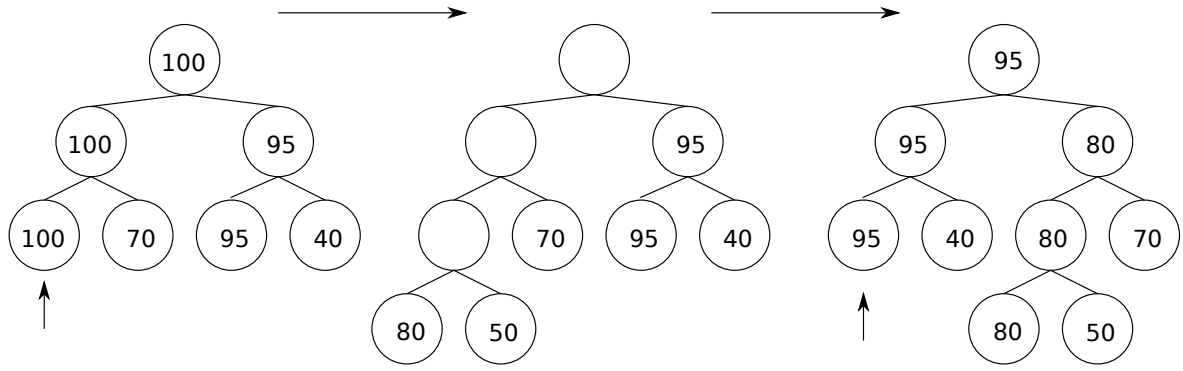

Figure 2: Example of an optimization tree and one branching step

Although the tree will not be balanced most of the time, the search time for the candidate set with highest maximum score will be reasonably limited, especially if we choose a good branching function.

In order to reduce complexity, we will immediately collapse the tree if the *score* function reports that no solution exists in the given candidate set. This usually happens when we search for a triangle and it is clear that the candidate set cannot contain any triangle fulfilling the criteria. Alternatively, if we find some solution e.g. during a broken line search, we can safely discard all solutions that have lower score then the one we already found.

Because searching for maximum triangles may be a very difficult task, we first compute a free flight score using the dynamic programming algorithm on a reduced set of points. Such a solution is a valid solution, albeit probably not a maximum solution. The result is then used as a cutoff point for subsequent problems; all candidate sets with lower maximum score are immediately discarded.

3 Branch and Bound

3.1 Rectangle Sets

The basic required operations between two sets of points are maximum and minimum distance. Although we could compute it by comparing distances between all points ($O(N^2)$), the main requirement for the bound method is speed; we need to compute it fast.

The first idea is to compute a reasonably small circle that would contain all points from a given set. This can be computed in $O(N)$, the longest distance is simply distance between the centers plus radius, minimum distance is distance between the centers minus radius. However, we claim that computing a bounding rectnagle has significant advantages over the circle:

- The time complexity of computing the rectangle is $O(N)$, but this time without any use of slow goniometric functions.
- It is easily additive; given bounding rectangles of two sets, the precise rectangle of the union can be computed without inspecting the individual points.
- As most flight tracklogs contain straight lines, the rectangle is significantly more precise.
- It is friendly to the precomputed sine and cosine of individual points; the vertices of the rectangle can be computed by combining the values of individual points without any need to use goniometric functions.
- It is easier to create more precise score functions.

The rectangle sets seem to exhibit more favourable behaviour then the circle sets. In order to eliminate recomputing the rectangle when the set of candidate is branched, we will build the rectangle sets in the beginning. During the optimization, the branching step then simply means using the content of a bounding rectangle (figure 3).

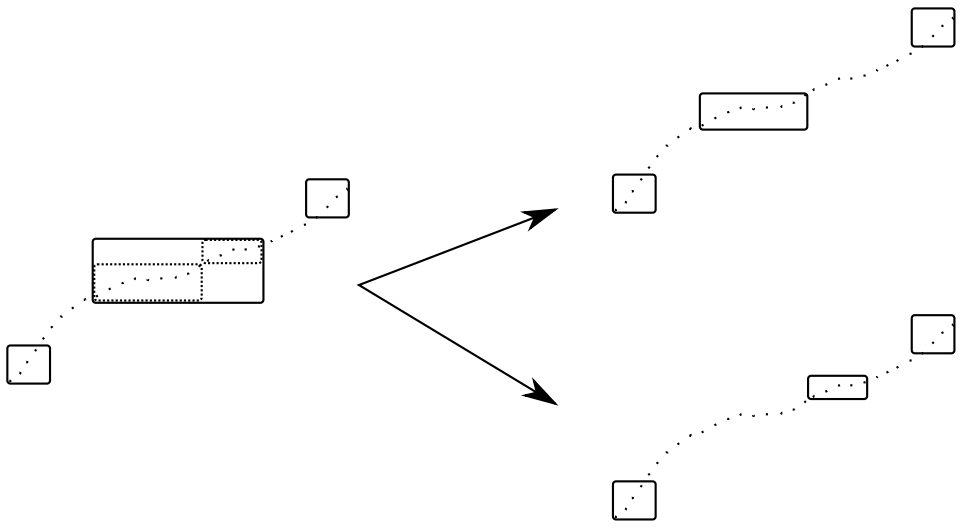

Figure 3: Branching using rectangle sets

The maximum and minimum distance are slightly more complex. The longest distance between two rectangles is equal to maximum distance between the vertices. In order to compute the longest distance we thus find the maximum path over all possible combinations of vertices of the two rectangles.

The minimum distance is similar; if the rectangles overlap, return 0. If they overlap neither on latitude nor longitude axes, return the minimum over all combination of distances of the vertices. If they overlap over longitude axis (one is northern of the other), compute the distance along the latitude axis. If they overlap over the latitude axis (one is western from the other), compute minimu longitude distance between opposing vertices of the rectangles.

3.2 Informal Proof of Corectness of Rectangle Sets

"The longest distance between two rectangles is equal to the maximum distance between the vertices."

3.2.1 Euclidian Geometry

The term longest distance between sets of points S_A and S_B denotes:

$$m_{S_A,S_B} = \sup_{\substack{A \in S_A \\ B \in S_B}} |AB|$$

Theorem: given a point A, the longest distance between a point and a rectangular set of points can be computed as follows:

$$\sup_{B \in S_B} |AB| = \max_{B \in \text{vertices}(S_B)} |AB|$$

Proof:

Lemma: given a point A, the longest distance between a point and a rectangular set of points can include only points on the edge of the rectangle. (without proof)

When computing a distance in euclidian space, we will use a right-angle triangle (figure 4)

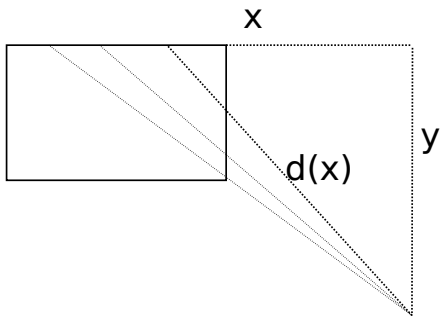

Figure 4: Euclidian distance from a rectangle

$$d(x) = \sqrt{x^2 + y^2}$$

This is a function of a distance between a point and horizontal line in a 2D space with axes X and Y. We have fixed the Y axes and move along the X axes. The function $d(x)$ has total minimum in $x = 0$, it is decreasing in negative numbers and increasing in positive. Thus, the maximum of this function (longest distance) over a horizontal line segment lies on the edges of the domain — in the vertices of the line segment. Similarly for vertical edge of a rectangle.

Now we will prove that:

$$m_{S_A,S_B} = \max_{\substack{A \in \text{vertices}(S_A) \\ B \in \text{vertices}(S_B)}} |AB|$$

Proof: let's claim otherwise. We have already proven that at least one of the points A and B will be a vertex of one of the rectangles. Suppose that B is a vertex of a rectangle and A *is not* a vertex, and $m_{S_A,S_B} = |AB|$. Then according to the previous theorem the maximum distance from B to S_A is equal to $\max_{A_v \in \text{vertices}(S_A)} |A_v B|$. But A was supposed not to be a vertex.

3.2.2 Spherical Geometry

Many things that work in euclidian geometry do not work on a sphere. However, in the next paragraphs I will try to prove that if we consider small enough distances (i.e. smaller than 90°, we can safely use the same theory).

Our spherical "rectangles" are defined by latitude and longitude. In order to prove the theorem, we will divide the options into different catagories.

Lemma: the longest line does not consists of points inside the rectangle.

First let's think about the scenario that the lemma is not valid; we will claim that there is a point B that is *not* inside (i.e. not on the edge) the rectangle S_B whose distance from some point A is higher then from any other point in a rectangle S_B. This is true if the points A and B are the opposite poles of the sphere. In all other cases we can construct a line AB and find points along the line that are further away from point A until we reach the edge of a rectangle. Therefore, in further discussion we have to prove the theorem only for the points on the edges.

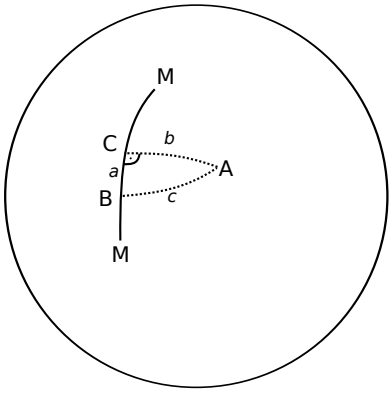

Figure 5: Right angle triangle on sphere

Let's start with the meridian edge first (figure 5). For any point A and a point B that is part of a segment of meridian line MM, we can construct a right-angled triangle, whose

hypotenuse (c) measures the $|AB|$ and one leg (a) is part of the meridian corresponding to M. In such triangle, we can use a spherical version of the Pythagorean theorem:

$$\cos c = \cos a \cos b$$

The distance function for a fixed point A, when moving along the meridian (i.e. thus varying the length of a) is:

$$d(a) = \arccos(t \cos a)$$

As long as $\cos a$ and $\cos b$ are positive (i.e. in the interval $(-\frac{\pi}{2}, \frac{\pi}{2})$), the function $d(a)$ has minimum in $a = 0$ and is decreasing in negative numbers and increasing in positive. Therefore, the longest distance can again be found at the vertices of the segment M; as long as the points are less then $\frac{\pi}{2}$ radians apart.

To prove the theorem on the latitude line, we will use the equation for measuring general distance between two points on a sphere:

$$|AB| = \arccos(\sin A_{lat} \sin B_{lat} + \cos A_{lat} \cos B_{lat} \cos(A_{lon} - B_{lon}))$$

When we fix the point A and move along the latitude line, the distance function will look like this:

$$d(l) = \arccos(c_1 + c_2 \cos l)$$

This function has minimum in $l = 0$, i.e. when the points A and B are on the same meridian. The function is decreasing in negative domain and increasing in positive, the maximum — again — is found on the vertices of the segment of the latitude line of our rectangle set.

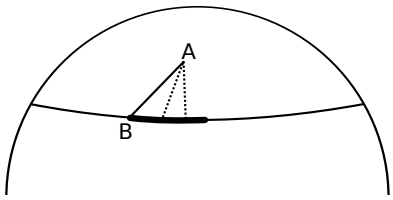

Figure 6: Distance of a point to a line segment on a latitude line

We have effectively proven that the theorem is true as long as the longiuted distance is less then 90°, which is about 3400km on 70° of latitude (north of Alaska). That seems to be appropriate for our objectives.

3.2.3 Ellipse in the Spherical Geometry

Most of the optimization algorithm hinges on a a more complex situation: Given points A and B and a rectangle S_C, is it true that:

$$\sup_{C \in S_C} (|AC| + |BC|) = \max_{C_{ver} \in \text{vertices}(S_C)} (|AC_{ver}| + |BC_{ver}|)$$

This theorem is true in Euclidian space; the set of points with the same sum of distances from 2 points is ellipse. Ellipse is convex, thus an intersection with a rectangle produces either all points inside or at least one vertex outside of the ellipse. Therefore, some vertex of a rectangle is really the point with maximum sum of distances from the 2 points.

On a sphere, we will first construct a reflection of the point B along the meridian. The shortest path between points A and B' is going through the point C_s. The length $|AC|+|CB'|$ (which is the same as $|AC|+|CB|$) is obviously (see appendix) monotonally increasing on both sides of the point C_s along the CC line segment, therefore we can find again the maximum in the vertices (figure 7).

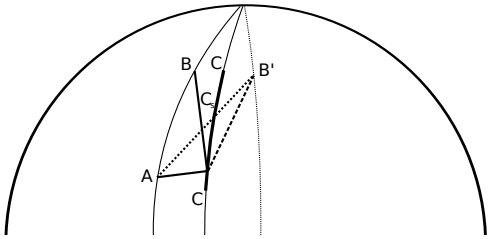

Figure 7: Longest distance from 2 points is through the vertices of a line segment CC

However, it is clearly not true for the intersection of the latitude line and an ellipse. As can be seen on figure 8, the distance $|AD| + |DB|$ is longer than the distance $|AC| + |CB|$. If a rectangle has one side defined by latitude segment CX, measuring the distance from the points A and B only over the vertices (i.e. points C and X) would not return a correct longest distance.

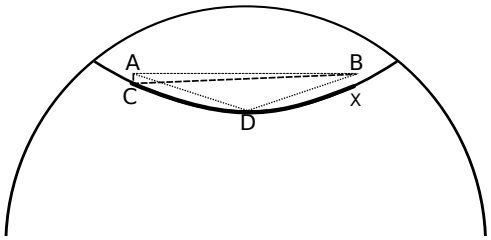

Figure 8: Distance from two points is longer through the point D instead of C

Because the mathematics gets quite complex, some numerical tests were done. The deviation of the theorem depends on height ($|AC|$), width ($|AB|$) and latitude of the area. 2 triangles were measured (the ACB and ADB as on figure 8). The difference between the circumference of these triangles measures the *deviation* of the algorithm from the correct result. When the area gets higher, the algorithm yields better results. When the area is wider or we move nearer to the pole, the algorithm deviation is higer.

However, the deviation for an area 300km wide (corresponding to 600km flat triangle), 100 m high located on the 70 latitude line (north of Alaska) is 56 meters. The difference disappears when the area is only 160m high. The break-even height of a more realistic 200km flat triangle located on the 55 latitude line is 1.7 meters.

There are 2 options to correct for the deviation; one is to find a formula that can characterize the deviation and use it to correct the results. Other is to recompute all coordinates into a different system, e.g. use a meridian-type system instead of latitude. This would probably require significantly more complex computations, however it would be needed only to find out the correct bounding "rectangles". Once the "rectangles" are found, all other computations

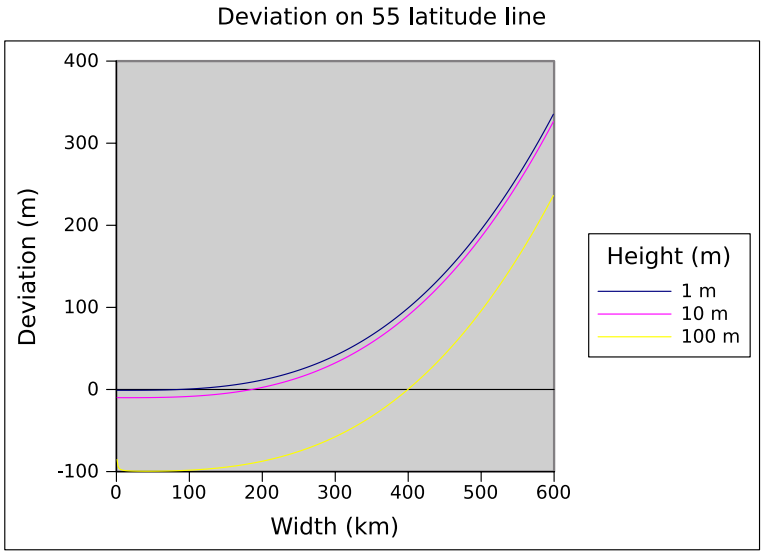

Figure 9: Deviation from correct longest distance

could continue in the usual coordinate system.

It is obvious that the deviation of our algorithm from the correct values is small enough to be ignored. The deviation would only manifest in very large and very flat triangles or very long and extremly straight east-west free flights and even then it would be very small. Additionally, the optimization algorithm practically guarantees that these deviations will have zero impact on the results of optimization of triangles and makes it very unlikely for free flights; considering the shape of the deviation function, the total deviation will not exceed the numbers depicted in figure 9 when we use width as a total distance of a free flight.

3.2.4 Meridian-like system for latitude

In order to correct for the deviations described in the previous chapter we could compute the bounding rectangles in a different coordinate system (figure 10); afterwards we would convert the rectangle vertices back into the classical latitude/longitude system and use the algorithms described in the following sections. This algorithm was not implemented as the gain is negligible.

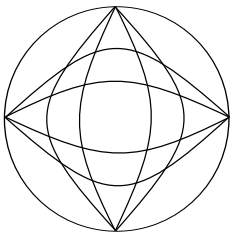

Figure 10: A different meridian-like coordinate system for latitude

To convert latitude into a meridian type coordinate with a center on the (0,0) coordinates,

the following formula can be used:

$$\tan P_m = \frac{\tan P_{lat}}{\sin P_{lon}}$$

3.3 Broken-line Bound function

The simplest way to compute a maximum score for a broken line is to add the maximum distances between the sets corresponding to each point (figure 11).

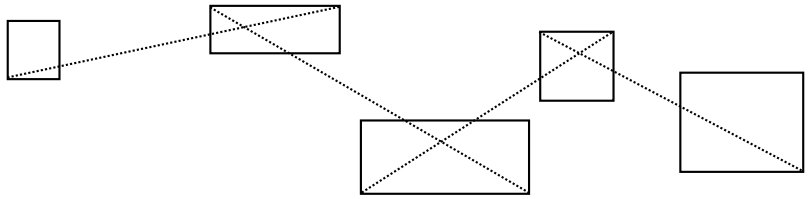

Figure 11: Simple algorithm for longest broken line

A better approach comes from an insight that the maximum path traverses always through the vertices (or, through the intersection with the equator). We can easily build a significantly more precise score by using a dynamic programming algorithm to find out the longest path over the vertices of the different rectangles without sacrificing any speed (figure 12).

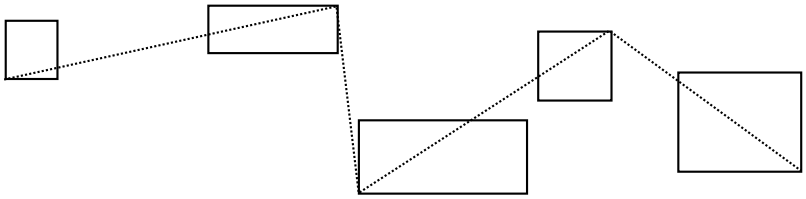

Figure 12: More precise algorithm for longest broken line

3.3.1 Vertex Elimination

Under certain conditions we can eliminate vertices on the edges of the broken line from the computation. If the first two or last two rectangles do not overlap, we will discard the vertices on the adjacent side.

3.3.2 Straight Line Detection

Simulation of the optimization algorithm revealed that some patterns significantly overestimate the expected score of a candidate set and reduce the ability of the optimization algorithm to prune the search tree of bad solutions. This is particularly visible when multiple vertices share one set of points.

When 2 candidate solutions sharing the same set of points are detected, temporary branching of this area is performed and the highest score is returned. As most of the tracklogs actually consists of straight lines, this optimization lets the score function converge significantly

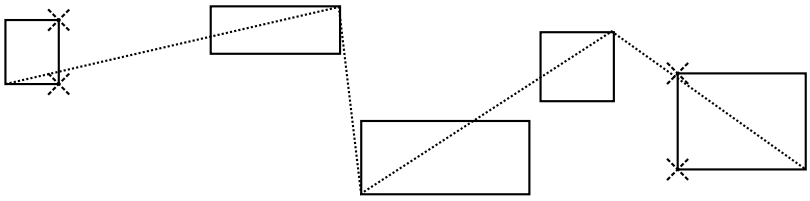

Figure 13: Elimination of some vertices on the edge rectangles from computation

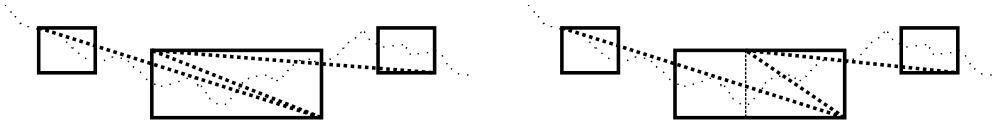

Figure 14: Longest line with 2 points sharing one set (left), better result by eager branching (right)

faster and allows the optimization algorithm to dispose of bad solutions much sooner (figure 14).

3.4 Triangle Bound Function

We use exhaustive search over all the vertices of the rectangles that define the sets corresponding to the triangle vertices. Additionally we have to check whether the triangle is closed, i.e. if there exist points s and e that are near enough, where s preceeds first point of triangle and e succeeds the last. The condition is usually a percentage of the total triangle circumference. When we define V_1, V_2 and V_3 to be sets of vertices of a bounding rectangle of a set of points corresponding to the vertices of a triangle, *mindist(S,E)* to be a a good guess of a minimum distance between sets of points for checking closing condition, the scoring function for a flat triangle becomes:

$$C_{flat} = \max_{\substack{v1 \in V_1 \\ v2 \in V_2 \\ v3 \in V_3}} [\text{triangle}(v1, v2, v3)] - \text{mindist}(S, E)$$

Additional conditions are placed on FAI triangle; first it is checked whether the triangle can be closed at all and if it is really possible to draw a FAI triangle using given coordinate sets. The FAI condition allows us to produce a different scoring function. When we define a smallest possible triangle side as:

$$e = \min_{(A,B) \in \{(V_1,V_2),(V_2,V_3),(V_3,V_1)\}} [\max_{\substack{v_1 \in A \\ v_2 \in B}} \text{distance}(v_1, v_2)]$$

the scoring function can be computed as:

$$C_{ufai} = \frac{e}{0.28} - \text{mindist}(S, E)$$

The resulting scoring function for FAI triangle then becomes:

$$C_{fai} = \min\{C_{flat}, C_{ufai}\}$$

3.4.1 Vertex Elimination

The longest triangle computation takes $3.4^3 = 192$ distance operations. Reducing the number of vertices from the computation can dramatically lower the expected time. In certain particular — although quite typical — situation we could reduce the total number of distance computations to 12.

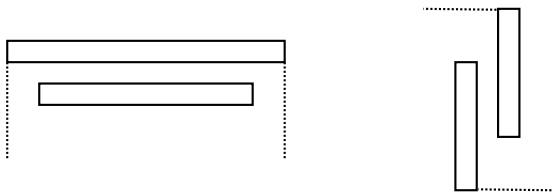

Figure 15: One set included in another over longitude axis (left), no inclusion (right)

If any candidate set is *not* completely included in any other candidate set over latitude or longitude axes (figure 15), only the 2 situations in figure 16 are possible. The longest triangle can be found among the circled points.

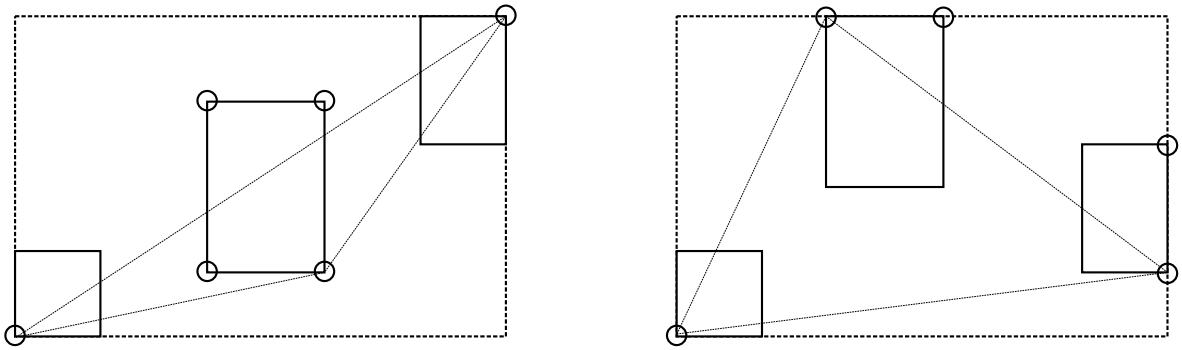

Figure 16: Two possible situations with non-overlapping sets

This optimization reduces in most typical cases the number of distance computations from 192 to 12. It reduces the total computing time by 50%.

3.5 Branch Function

The whole algorithm begins with one candidate set "all vertices are in a bounding rectangle comprising all points". As the bounding rectangles form a binary tree themselves, the branching step starts with the two children sets. The branching will produce several new candidate sets, e.g. "all vertices are in the first set", "first vertex is in the first set, all the other are in the second set", "first two vertices are in the first set, the others are in the second one" etc.

During the optimization algorithm the input to the branching function is thus something like "First n_1 points are in set s_1, next n_2 points are in set s_2, etc.". The branching step chooses some set (e.g. s_2), divides it (into s_{2_1} and s_{2_2}) and produces new candidates with the points redistributed between the sets; e.g. "first n_1 points are in set s_1, next n_2 points are in

set s_{2_1}, etc.", "First n_1 points are in set s_1, next $n_2 - 1$ points are in set s_{2_1}, next 1 point is in set s_{2_2}", etc.

Several different ways to choose the set to be divided were tested - among others the number of vertices sharing one set, number of points in the set, area of the set. Surprisingly, consistently best results were obtained by choosing the set with the longest diagonal of the bounding rectangle.

4 Additional Optimizations

By tracing the input to the *distance* function we found out that only about 3% of the distance computations are unique; 97% are repeated computations of distance over the same 2 points. Several tests where made using different approaches to cache the results (Hash table, IntMap, Judy), but none was successful in speeding up the computation.

We could probably speed up the computation if we maintained a lazily computed array of distances between the points — and in our case the distances between all the vertices of the bounding rectangles. The memory requirements would come to several gigabytes. Trading space for time in this case might make some of the micro-optimizations described above obsolete.

5 Conclusion

The algorithm described in this article was implemented in Haskell, tests were done using the GHC 6.10.4 compiler on an Intel Core2 Duo L9600 running on 2.13GHz. Several tracklogs with 2000 - 30,000 points were tested. The results are very satisfactory; the great majority of tracklogs is analyzed within a fraction of a second. Free flights comprising of 30,000 points take about 11 seconds while needing $\sim$ 50MB of memory. A semi-unclosed FAI triangle (an example of a very hard to optimize flight) with about 10,000 points took 2.5 minutes and needed $\sim$ 100MB of memory.

A Is an ellipse on a sphere convex?

We have already proposed a meridian-like coordinate system to provide correct bounding rectangles. Here is the *obvious* proof that it works correctly.

First, let's define a function $f(x)$; the function defines the distance $|AX| + |XB|$ (see figure 7), where x is a distance from point C_s (e.g. positive to the north and negative to the south). If the function has local minimum in the point C_s and is decreasing in the negative numbers and increasing in the positive (for x being small enough compared to the radius of the sphere), our algorithm works properly. Incidentally, this also proves that an ellipse projected on a sphere is convex.

The proof is actually simple; as you move along the CC further away from the point C_s, the nearer triangles are completely inside the triangles with point C further from point C_s. Therefore, we have only to prove that a triangle that is inside another triangle is really smaller.

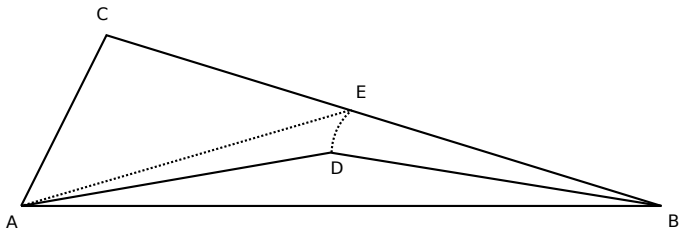

Figure 17: Small and large triangle

On figure 17 you see triangles ACB and ADB. As $|AD| > |AC|$, we have to find some other way to compare them. We will construct a point E, where $|BD| = |BE|$. Then $|AE| > |AD|$ (as we have proven in section 3.2.2 for distance from latitude line) and $|ACE| > |AE|$ (by triangle inequality).