Time Range Queries for Hereditary Properties

6.854 Final Project

Arsen Mamikonyan, Hayk Saribekyan

December 13, 2016

Abstract

Time range queries are important for analysing large datasets with timestamps. Such datasets are common in computational geometry, where a sequence of points with timestamps often corresponds to a trajectory of an object. Time range queries in this case check a certain property of section of the trajectory.

In this paper we review recent general frameworks that can be used to handle property testing queries for time ranges, and discuss their implementations for solving certain problems in computational geometry. The results are limited to *hereditary* properties, which cover a large variety of interesting problems.

For two different hereditary properties, we compared the performance of efficient an algorithm and a naive one in practice. We observed that the efficient algorithms require more than ten times the amount of code in certain cases, but the large constant factor associated with it is compensated when working with large datasets.

1 Introduction

With the abundance of GPS and other movement tracking sensors there is a large amount of timestamped location data. The movement of an object in space can be modelled by a sequence of points $S = s_1, \ldots, s_n$. To analyse the trajectory of the movement, one may want to check a certain property P for portions $S[i, j] = s_i, \ldots, s_j$ of S for given values of i and j. This paper discusses how to efficiently address such queries when P satisfies the following two properties:

- P is boolean.
- P is hereditary. This means that for a given sequence S, if $P(S)$ is true, then $P(S')$ is true for any continuous subsequence S' of S.

In this paper we discuss and implement two properties for a sequence of points $S = s_1, \ldots, s_n$:

- *Monotonicity*: S is *monotone* if there is a direction vector v such that TODO. In terms of trajectories, monotonicity shows whether the object travelled more or less in the same direction.

- *Closeness*: S is *close* if any pair of points in S is at most of distance 1. One could use closeness queries to detect if a moving object mostly stayed in the same surrounding.

For a property P satisfying the restrictions above, let $j^*(i)$ be the largest index j such that P holds for $S[i, j^*(i)] = s_i, \ldots, s_{j^*(i)}$. Now notice, that since P is a hereditary property $P(S[i, j]) = true$ if and only if $j \leq j^*(i)$. Therefore, if we construct $j^*(i)$ we can answer property testing queries in $O(1)$. Therefore, for the rest of the paper our goal will be to efficiently compute $j^*(i)$.

Bokal et al. [1] propose an algorithms that compute $j^*(i)$ in $O(n)$ time for monotonicity, and in $O(n \log^2 n)$ time for closeness. Chan and Pratt [2] describe a different algorithm to achieve the same result for closeness, but also improve it to $O(n \log n)$ using fractional cascading.

The rest of the paper is organized as follows: Section 2 describes the naive algorithms we developed to solve the problems we selected; in Section 3 we review the general framework that is set in [1] for solving time range query problems, and show how it is applied to computer $j^*(i)$ for monotonicity. Section 4 reviews the framework by [2] and its application for the closeness property. We then give some implementation details in Section [**?**], and describe the experimental setup and our tests in Section [**?**].

2 Naive Algorithms

Suppose that for a sequence of length n a property P can be checked in $T_P(n)$ time. Then, one can compute $j^*(i)$ using binary search for each i. On a step of a binary search concerning a range $[l, r]$ the algorithm would spend $T_P(r - l)$ time. The total runtime of this simple algorithm is $O(nT_P(n) \log n)$. Call this algorithm SN (stands for super-naive). For the problems of monotonicity and closeness we can do better.

2.1 Monotonicity

To detect whether a sequence is monotone or no, we can process the points while keeping a set of polar angles (field-of-view or FoV) that a direction vector v can have. Notice that the FoV is always an interval. When a new point arrives we can update the FoV in constant time. If the FoV ever becomes empty, then the given sequence is not monotone. This algorithm runs in $O(n)$ time for a sequence of length n. Using this submodule for SN we would get a $O(n^2 \log n)$ algorithm.

However, doing a binary search for monotonicity is redundant, since we will do the same computation many times. Instead, for each i proceed with the

FoV computation until it is empty, which will happen precisely when we reach $j^*(i)+1$. The resulting algorithm runs in $O(n^2)$ time.

2.2 Closeness

The simple method of closeness-check takes $O(n^2)$ for a sequence of n points, because one has to check all pairs of points. This results in SN runtime of $O(n^3 \log n)$, which is super-slow. We can improve the closeness-check to $O(n \log n)$ using furthest-point Voronoi diagrams, but that would not qualify for SN and would also take $O(n^2 \log^2 n)$ time to compute, which is also not fast.

Define $k^*(i)$ as the largest index such that $d(s_i, s_j) \leq 1$ for all $j \leq k*(i)$. Clearly, $k^*(i) \geq j^*(i)$. Also, notice that k^* can be computed in $O(n^2)$ for all points.

Claim TODO - formatting For $i < n$ $j^*(i) = \min(j^*(i+1), k^*(i))$

Proof obvious (should we even prove this?

This claim gives us an $O(n^2)$ SN construction of $j^*(i)$ for the closeness property.

3 something and Monotonicity TODO

Bokal et al. introduced an elegant framework to deal with range queries for hereditary problems. The key idea in their work is to greedily split the given set of points into ranges, defined by anchor points, and solve for each using a divide and conquer. To simplify the reasoning we first define a property matrix A_P such that

$$A_P(i,j) = \begin{cases} 1 & P(S[i,j]) = true \\ 0 & P(S[i,j]) = false \end{cases}$$

Suppose that there is an algorithm J_P that, for a given i, finds $j^*(i)$ (notice that even if such J_P exists, we are not happy to run it n times for each i). We use J_P to find *anchor* points in S as follows. Let a_k be the index of k^{th} anchor point in S. We define $a_1 = s_1$, $a_k = j^*(a_{k-1})$ for $k > 1$. Thus, the anchor points can be found using J_P. Figure 3 shows the anchor points on A (leftmost image).

As shown in Figure 3 the anchor points form rectangles. If we find the frontier (the curve in A separating 1's from 0's) for each rectangle, then we will be done. This is where divide and conquer strategy is used. Consider a rectangle R with rows $[a, a+h]$. We can set $m = a + [h/2]$ and using J_P find $j^*(m)$.

We know that if $m \leq i \leq a+h$ then $j^*(i) \geq j^*(m)$, and if $a \leq i \leq m$ then $j^*(i) \leq j^*(m)$. Therefore, we can recursively solve for smaller rectangles that are in the upper-left and lower-right corners of R.

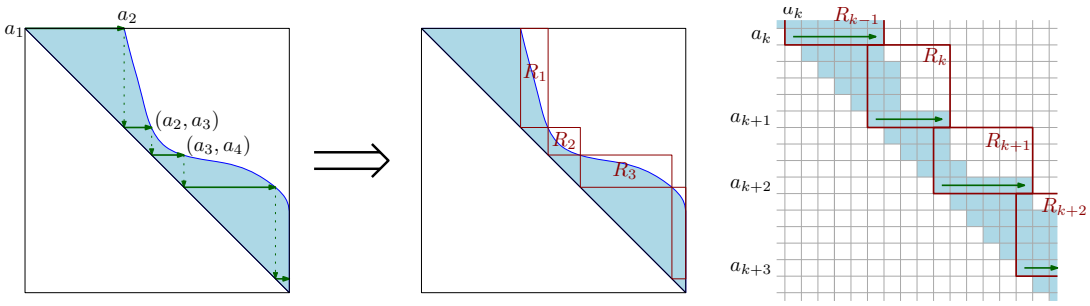

Figure 1: The property matrix A_P. The left image illustrates the anchor points. The middle one shows the greedily constructed rectangles from the anchor points. On the right side is the detailed view of the property matrix.

4 something and Closeness TODO

5 Implementation Details

- decided to implement chan's algorithm because the other one had voronoi and persistent data structure (or something similar) for point location in voronoi
    - chan's algorithm is closer to what we've done in class (envelopes)

6 Experiments

- mention that we have used random walk to mimic reality. in that case the good algorithm for monotonicity is not that good. but for the worst case, it's great.
    -

7 Conclusion

We will describe their algorithm for monotonicity property, which due to its simplicity does not require a divide and conquer in the second stage of the algorithm.

8 Our contribution

We've implemented

- In $O(n)$ time we can find all maximal subsequences that define monotone paths in some (subpath-dependent) direction. [**?**]

- In $O(n \log^2 n)$ time time we can find all maximal subsequences with diameter at most 1. [**?**]

9 Algorithms

9.1 k^*

Let $k^*(i) = \inf_{m \geq i}\{d(i,m) > 1\}$. **Claim** $j^*(i-1) = \min(j^*(i), k^*(i-1))$. Thus after we calculate $k^*(i)$ for all elements, we can calculate $j^*(i)$ in $O(n)$ time by looping over all indices in the reverse order.

9.2 Bokal et al Overview

TODO: A page that deftly describes the algorithm we've implemented.

9.3 Chan, Prat Overview

TODO: A page that deftly describes the algorithm we've implemented.

10 Implementation Details

In this section we discuss some implementation details of the Chan, Pratt algorithm for diameter query problem. We skip the details of the monotonicity query, because it was relatively simple.

The novelty of the algorithm from Chan, Pratt is how they combine 1D range tree, with secondary structure that stores circle intersections. While the algorithm is elegant, its implementation is not. In particular, it is necessary to implement an intersection algorithm for arbitrary *polyarcs*. A polyarc is a convex object, similar to a polygon, but instead of straight edges each side is an arc of a circle. The algorithm for the intersection of polyarcs is similar to that of polygons, however there are many more edge cases to consider.

11 Experimental Results

11.1 Monotonicity

TODO: Describe Naive algorithm which is a $O(n^2)$ algorithm.

First let's look at the monotonicity resutls. Here we use the following dataset.

We have a point moving along x axis in the positive direction with speed 1 (i.e. 1 per index), we add Gaussian noise to this point to make the problem interseting. In Figure 2 you can see results for noise with standard deviations $\sigma_y = 0.05, \sigma_y = 1$. After we generate the dataset, we run Bokal et. al algorithm on the dataset and the answer boundarries of the matrix A that indicates if for points in range $[i,j]$ exists a common direction that has positive dot product with each of the displacements.

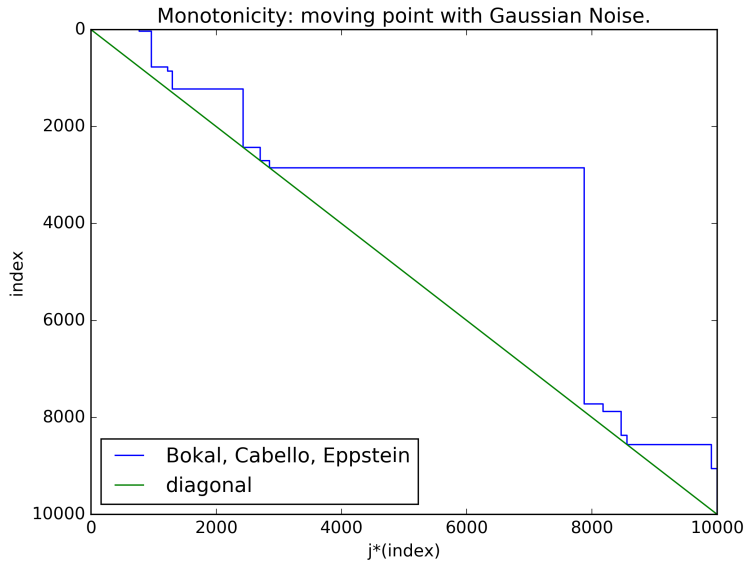

Figure 2: Monotonicity algorithm results for moving point with Gaussian noise.

Figure 3 shows runtime differences in millliseonds between Naive algorithm and algorithm we presented from Bokal et al.

Bokal et al	Naive
0.616	3.43
2.008	12.799
6.038	45.436
19.33	152.641
61.926	428.032
194.457	1454.634

Figure 3: Runtimes (in millseconds) of Naive and Bokal et. al

11.2 Diameter

Now the dataset is a random walk starting at the origin. At each step we uniform randomly pick a direction, and move 0.3 distance in that direction. We keep doing this until we have enough points for an experiment.

TODO: Describe Naive algorithm which is a $O(n^2)$ algorithm.

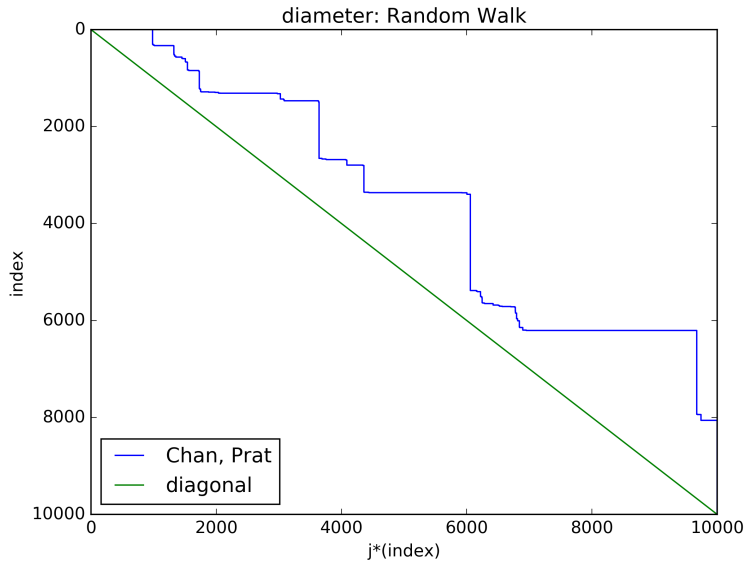

Figure 4: Caption...

Figure 4 shows boundaries of points that satisfy all points in range $[i, j]$ fall into some circle with radius 1.

Figure 5 will show results comparing runtimes of Naive algorithm with algorithm we presented from Bokal et al.

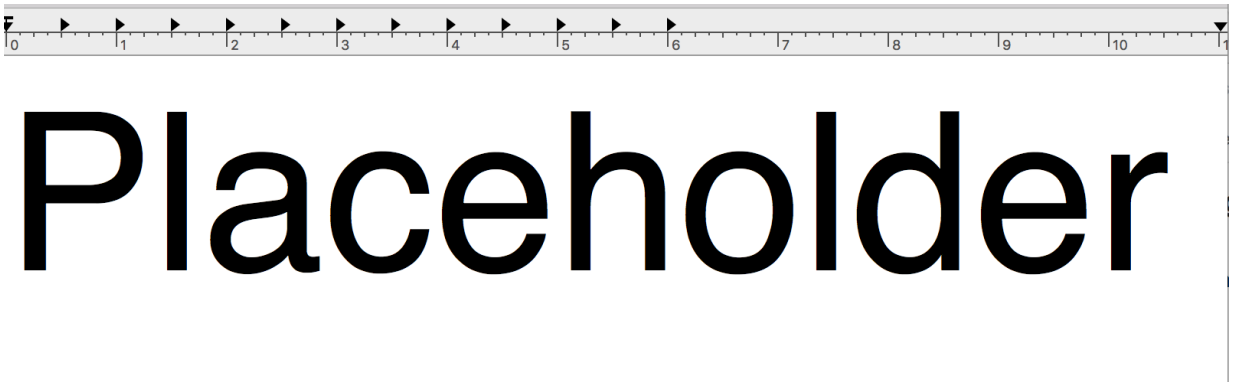

Figure 5: Runtimes (in millseconds) of Naive and Chan, Prat [Similar to Figure 3

12 Conclusion

It was really challanging to do an implementation project in computational geometry, some sentences in the papers take 100+ lines of code to implement. Thus, for small datasets, added complexity is not worth the speedup gained by the algorithm.

This was a great project otherwise, we've implemented couple of algorithms that we have seen during the lecture - sweep line, range tree, intersection of convex polygons using envelopes. Also couple that we had not found in Bokal et al and Chan, Prat.

References

[1] Drago Bokal, Sergio Cabello, and David Eppstein. Finding All Maximal Subsequences with Hereditary Properties. In Lars Arge and János Pach, editors, *31st International Symposium on Computational Geometry (SoCG 2015)*, volume 34 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 240–254, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[2] Timothy M. Chan and Simon Pratt. Two Approaches to Building Time-Windowed Geometric Data Structures. In Sándor Fekete and Anna Lubiw, editors, *32nd International Symposium on Computational Geometry (SoCG 2016)*, volume 51 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 28:1–28:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.