Range Queries over Untangled Chains*

Francisco Claude, J. Ian Munro, and Patrick K. Nicholson

David R. Cheriton School of Computer Science, University of Waterloo, Canada.
{fclaude,imunro,p3nichol}@cs.uwaterloo.ca.

Abstract. We present a practical implementation of the first adaptive data structure for orthogonal range queries in 2D [Arroyuelo et al., ISAAC 2009]. The structure is static, requires only linear space for its representation, and can even be made implicit. The running time for a query is $O(k \lg n + m)$, where k is the number of non-crossing monotonic chains in which we can partition the set of points, and m is the size of the output. The space consumption of our implementation is $2n + o(n)$ words. The experimental results show that this structure is competitive with the state of the art. We also present an alternative construction algorithm for our structure, which in practice outperforms the original proposal by orders of magnitude.

1 Introduction

Imagine that you are driving and would like to find a nearby gas station using the Global Positioning System (GPS) in your car. You might instruct the GPS to draw markers on the map to indicate gas stations. Internally, the GPS would compute which gas stations are located within the rectangle shown on the screen. Scenarios like the one just described are special because the set of elements being queried do not change very often: you might update the database of your GPS once every few months. In this paper, we will discuss a recent data structure that is well suited for these kinds of scenarios.

Consider a set of two-dimensional points, $\mathcal{P}$, where $n = |\mathcal{P}|$. An *Orthogonal Range Query*, $\mathcal{R}(x_1, y_1, x_2, y_2)$, where $x_1 \leq x_2$ and $y_1 \leq y_2$, corresponds to a maximal subset $\mathcal{A} \subseteq \mathcal{P}$ such that $\forall (x, y) \in \mathcal{A}$, $x_1 \leq x \leq x_2$, $y_1 \leq y \leq y_2$. Many solutions have been proposed for answering orthogonal range queries, such as:

- R-trees [11]: a tree decomposition of $\mathcal{P}$, similar to B-trees, where a set of nodes represents a set of (possibly overlapping) rectangles. The structure takes linear space and its worst case searching time is $O(n)$. However, in practice they tend to perform better.
- Kd-trees [13]: recursively divide the d-dimensional space with hyperplanes. In our case, $d = 2$, the kd-tree uses lines for dividing the set of points. As with R-trees, kd-trees require linear space, but the query time improves to $O(\sqrt{n} + m)$, which is optimal for a structure that requires linear space [12].

* This work was supported in part by NSERC Canada, the Canada Research Chairs Programme, the Go-Bell and David R. Cheriton Scholarships Program, and an Ontario Graduate Scholarship

– Range trees [6]: a generalization of a binary search tree for multiple dimensions. The query time of this structure is $O(\lg n + m)$ time (when combined with fractional cascading [7]), but as a trade-off it requires $O(n \lg n)$ space.

In recent work [3], a static structure for orthogonal range queries was presented. This structure achieves the same complexity as kd-trees in the worst case, but has the advantage that it is adaptive: for *easy* data sets it can improve upon the worst-case lower bound. We will call this structure *u-chains* as a shortened form of *untangled-chains*.

The structure of the paper is organized as follows. First we present a more detailed description of the u-chains structure. Then we describe the implementation decisions taken during the experimental setup. Finally we present the experimental results and conclude about the work, pointing to interesting open problems.

2 The U-Chains Structure

The u-chains structure is constructed by partitioning the set of points into monotonic ascending and descending chains, where each point belongs to one and only one chain.

The problem of splitting a two-dimensional point set into two classes (descending and ascending) so that we get the minimum number of monotonic chains is NP-hard [8]. The maximum number of chains we can obtain is bounded by $O(\sqrt{n})$. Fomin et al. presented an algorithm that achieves a constant factor approximation in $O(n^3)$ time, as well as a greedy algorithm which achieves a logarithmic factor approximation in $O(n\sqrt{n}\log n)$ time [10]. Yang et al. [15] improved the Yehuda-Fogel method [4], giving an algorithm that does not guarantee an approximation factor, but generates at most $\lfloor\sqrt{2n + 1/4} - 1/2\rfloor$ chains in $O(n\sqrt{n})$ time.

If the direction is fixed, Supowit proposed an algorithm that runs in optimal $O(n \lg n)$ time and obtains the optimal number of chains [14]. (See Algorithm 1.)

Algorithm 1 – **Supowit**($\mathbf{p_1 \ldots p_n}$)

```
1: S ← ∅
2: for i = 1...n, where x(p_i) < x(p_j) ∀i < j ≤ n do
3:     let S' = {A ∈ S, miny(A) ≥ y(p_i)}
4:     if S' ≠ ∅ then
5:         let A_0 = argmin_A{miny(A), A ∈ S'}
6:         append p_i to A_0
7:     else
8:         add p_i as a chain to S
9: return S
```

For the u-chains structure, the chains are required to be untangled, meaning that no two chains cross or *tangle*. By running an untangling algorithm on the

output of Supowit's algorithm, it is possible to obtain the minimum number of untangled chains [3]. The main property that allows the untangling algorithm to work is that all tangles generated by Supowit's algorithm are so called *v-tangles*, and when processed in the right order, they generate the corresponding untangled set.

Next we give some basic definitions to introduce the notation used for describing the algorithm for obtaining the untangled set of chains.

Definition 1 (H). *[3] Given an edge (p_i, p_j), define $H^+(p_i, p_j)$ to be the open half-plane bounded by the line through p_i and p_j and containing the point $(x(p_i)+1, y(p_i)+1)$, and $H^-(p_i, p_j)$ symmetrically.*

Using this definition we can define a v-tangle:

Definition 2 (v-tangle). *[3] Suppose we have two chains C_1 and C_2 with edges $(p_1, p_2), \ldots, (p_{\ell-1}, p_\ell) \in C_1$ and $(q_1, q_2) \in C_2$ such that $p_1 \in H^-(q_1, q_2)$, $p_\ell \in H^-(q_1, q_2)$, and $p_i \in H^+(q_1, q_2)$ for all $1 < i < \ell$. Also, (q_1, q_2) is called the upper part of the v-tangle, and $(p_1, p_2), \ldots, (p_{\ell-1}, p_\ell)$ the lower part. We define the operation of untangling a v-tangle as removing $p_2 \ldots p_{\ell-1}$ from C_1 and inserting it to C_2, between q_1 and q_2.*

Definition 3 (rv-tangle). *[2] Suppose we have two chains C_1 and C_2 with edges $(q_1, q_2) \in C_1$ and $(p_1, p_2), \ldots, (p_{\ell-1}, p_\ell) \in C_2$ such that $p_1 \in H^+(q_1, q_2)$, $p_\ell \in H^+(q_1, q_2)$, and $p_i \in H^-(q_1, q_2)$ for all $1 < i < \ell$. We call such a tangle a* reverse v-tangle*, or* rv-tangle*.*

The untangling procedure is based on a simple building block shown in Algorithm 2 [2]. The main idea is to run one Untangling-Pass and then extract the lower chain. We then recurse on the residual point set, extracting the lower chain until no points remain. This version differs from the preliminary version [3], which contained a difficulty revealed by this experimental work and corrected in a later version [2]. This procedure takes $O(kn \lg n + k^2 n)$ time, where k is the minimum number of chains.

Algorithm 2 – Untangling-Pass(P)

```
1: Run Supowit(P) to get chains C_1, ..., C_s where C_s is the uppermost chain
2: for i = s down to 1 do
3:     for j = i − 1 down to 1 do
4:         Find and untangle all v-tangles between C_i and C_j
5: Return C_1, ..., C_s
```

When the chains are untangled and sorted, searching becomes easier in the following sense:

- Searching in a chain is equivalent to doing a binary search, because we can exploit the fact that the chain is monotonic.

- When a chain C does not intersect the query, we can discard all the chains to one side of C, given that we add dummy points at the beginning and end of each chain to enforce an ordering. The dummy points add $O(\sqrt{n})$ words of extra space[1].

The search algorithm described requires $O(k \lg n + m)$ time, which can be improved to $O(\lg n + k + m)$ time using fractional cascading. In the worst case, when $k = \Theta(\sqrt{n})$, we achieve the optimal time/space trade-off: linear space and $O(\sqrt{n} + m)$ query time [12]. For easier instances (i.e., when k is small) the structure takes advantage of this fact and performs better.

3 Implementation Details

In the following subsections we describe the practical decisions taken during the implementation phase. We go through the partitioning and untangling preprocessing steps.

3.1 Partitioning the Points

For our experiments, we implemented two algorithms for partitioning the set of points into monotonic descending and ascending chains. The first was the algorithm by Yang et al. [15]. The basic idea of the algorithm is to start with all of the points in descending chains called *layers*, and to repeatedly identify and extract ascending chains. It is important to note that the number of chains generated by the algorithm can be different if we start with ascending layers and extract descending chains. Although the algorithm can be implemented to run in $O(n\sqrt{n})$ time using optimizations described in [4], we instead opted to implement the simplified version of the algorithm which runs in $O(n\sqrt{n}\log n)$ time. Even with the extra $\log n$ factor, the processing time was not prohibitive for data sets of over a million points.

The second algorithm was the greedy method of Fomin et al. [10]. Since they describe the greedy algorithm in terms of a graph co-coloring problem, we will provide a brief description. We start with two empty sets, one for ascending chains, and one for descending chains. We take the set of points and compute the maximal length ascending chain, as well as the maximal length descending chain. The longer of the two chains is then added to the appropriate set, and we repeat this process on the residual point set until no points remain. Each pass of our implementation of the greedy algorithm takes $O(n \log n)$ time [4], and it generates a number of chains that is within a logarithmic factor of optimal [10]. Therefore the overall running time of our implementation is $O(n\sqrt{n}\log^2(n))$, but can be improved to $O(n\sqrt{n}\log n)$ using the techniques from [4].

Once we have determined which points belong to which set, we discard the extra information about the chains provided by the partitioning algorithms. In

[1] We need two dummy points for each chain, but we are also spending this amount of space for the pointers to the chains.

Algorithm 3 – **Untangling(P)**

```
1: Run Supowit(P) to get chains C_1, ..., C_k where C_k is the uppermost chain
2: done ← false
3: A_l ← 1
4: A_u ← k
5: while A_l < A_u AND ¬done do
6:    for i = A_u down to A_l do
7:       for j = i − 1 down to A_l do
8:          Find and untangle all v-tangles between C_i and C_j
9:    Apply transformation of lemma 2 to C_{A_l} ... C_{A_u}
10:   for i = A_l to A_u do
11:      for j = i + 1 to A_u do
12:         Find and untangle all v-tangles between C_i and C_j
13:   Apply transformation of lemma 2 to C_{A_l} ... C_{A_u}
14:   A_l ← A_l + 1
15:   A_u ← A_u − 1
16:   if C_1, ..., C_k are untangled then
17:      done ← true
18: Return C_1, ..., C_k
```

order for our subsequent untangling algorithm to work in a provably correct manner we need to run Supowit's algorithm on both the ascending and descending sets of points.

3.2 Untangling Chains

In this section we propose an alternative method for untangling the chains. As we will see in the experimental results, this alternative approach allows for a much faster construction in practice. However, we first describe how to improve the running time of the previous algorithm [2].

Lemma 1. *The algorithm by Arroyuelo et al. [2] for finding the minimum number of monotonic untangled chains can be adapted to run in $O(n \lg n + k^2 n)$ time.*

Proof. The improvement comes from running Supowit's algorithm k times in $O(n \lg n + kn \lg k)$ time. The main observation is that we only need to sort the points once, after which we can use markers to keep track of which points are still participating (i.e., do not belong to chains that are removed). We can then generate the set of the participating points, sorted, in $O(n)$ time. This happens only k times, so we spend $O(nk + n \lg n)$ time for the sorting phase in the k passes. The construction of the chains itself is easier, since we never have more than k chains in A, each point added requires at most $O(\lg k)$ time, and again we do this for every point at most k times, obtaining the remaining $O(nk \lg k)$.

Adding the cost of the k runs of Supowit's algorithm to the k runs of the Untangling-Pass, we obtain $O(n \lg n + k^2 n)$ time in the worst case. ☐

The idea of the alternative algorithm is inspired by the previous proposal [2], but is less dependent on the properties of Supowit's algorithm. The following simple lemma is key for our practical variation. It allows us to transform a set of chains having only rv-tangles into a set of chains that has only v-tangles.

Lemma 2. *Given a set of ordered chains over $\mathcal{P}$, where all tangles among two chains are rv-tangles, we can map this problem to one with ordered chains and only v-tangles.*

Proof. Let $m_x = \max\{x \mid \exists y,\ (x,y) \in \mathcal{P}\}$ and $m_y = \max\{y \mid \exists x,\ (x,y) \in \mathcal{P}\}$. We compute the new set $\bar{\mathcal{P}} = \{(m_x - x, m_y - y) \mid (x,y) \in \mathcal{P}\}$, maintaining the chains. The order of the chains is reversed. □

This observation allows to do one untangling pass, transform the points, and then repeat the procedure over the remaining set of chains, without discarding information about the chains. This idea is formalized in Algorithm 3.

We currently do not have a proof that Algorithm 3 will yield untangled chains after it finishes. However, we have found no case in practice where Algorithm 3 fails to untangle the chains. In fact, for all of the data sets we have tried in practice, the $O(n)$ time check on line 16 of the algorithm succeeds after a small constant number of passes. We conjecture that in the worst case k passes are required. We end this section by noting that we can run Algorithm 3 for k passes, and, in the advent of a failure, detect that the chains have not been untangled. After such a failure, we can then run the previous algorithm [2]. Since both algorithms have the same running time by Lemma 1, running Algorithm 3 does not affect the overall running time asymptotically. However, as we will see in the next section, there is a clear separation in practice between the two algorithms.

4 Experimental Results

The machine used for the experiments has an AMD Athlon(tm) 64 X2 Dual Core Processor 5600+, core speed 2900MHz, L1 Cache size 256KB, and L2 Cache size 1024KB. It has 4GB of main memory of speed 800MHz. The operating system installed is GNU/Linux – Ubuntu 9.10, with kernel 2.6.31-17-generic running in 64bits mode and the compiler installed in the system is GNU/g++ version 4.4.

Our implementation for preprocessing the data set was compiled with flags `-O2 -Wall`, the code for searching was compiled with `-O2 -frounding-math -Wall`.

We used data sets provided by the Georgia Tech TSP Web page[2]. We included the following sets: Italy, China, LRB744710 and World.

4.1 Partitioning

We first tried the different partitioning methods: `aa` for all chains ascending, `ad` for all descending, `gr` for the greedy approach [10], `pa` for Yang et al.'s

[2] http://www.tsp.gatech.edu

Data Set	Nr. Points	Partitioning Method	Nr. Chains	Avg. Nr. of Pts/Chain	Std. Dev. of Pts/Chain
Italy	16862	aa	285	59.16	31.78
		ad	207	81.46	47.38
		pa	173	97.47	62.69
		pd	160	105.39	57.17
		gr	141	119.59	56.87
China	71009	aa	520	136.56	51.36
		ad	697	101.88	53.87
		pa	367	193.49	68.57
		pd	373	190.37	103.96
		gr	324	219.16	57.65
LRB	744710	ad	1203	619.04	227.76
		aa	1129	659.62	271.43
		pd	1074	693.40	262.35
		pa	1006	740.27	262.79
		gr	1065	699.26	228.38
World	1904711	aa	4167	457.09	266.50
		ad	3046	625.32	314.56
		pa	1865	1021.29	574.67
		pd	1843	1033.48	558.83
		gr	1608	1184.52	424.15

Table 1. Results of different partition methods.

method [15] starting with all points in ascending layers, and finally `pd` for Yang et al.'s method starting with all points in descending layers.

Table 1 shows the number of chains obtained with each method for the data sets considered in this work. We also include the average number of points per chain and the standard deviation for the expected number of points per chain.

The results show that considering the points in a fixed direction provides a competitive result with respect to the other partitioning methods. Yet, in most cases (except `LRB`), the `gr` approach obtains the minimum number of chains.

4.2 Construction

We implemented the two untangling algorithms. Table 2 shows the results for the small data set and gives a representative idea on the running time of this new version. For the `World` data set, it takes about one day to untangle the chains using the original method, while the new algorithm takes about twenty minutes.

Data Set	Partitioning Method	Time (sec) Original [2]	Time (sec) This paper
Italy	aa	293.75	1.53
	ad	254.34	1.10
	pa	125.34	0.48
	pd	220.72	0.68
	gr	116.78	0.41

Table 2. Results of the two untangling methods.

4.3 Query Generation

For measuring the time, we generated 6 different types of queries. For each data set we find the bounding box $[x_{min}, x_{max}] \times [y_{min}, y_{max}]$. We refer to the regular

grid partition of the bounding box as `tile`. For the other queries we generate a random point[3] $[x_i, y_i]$ for each query, and the rectangle is determined as follows:

- `rand`: we generate the second point within $[x_i, x_{max}] \times [y_i, y_{max}]$.
- `tiny`: we generate the second point at random within $[x_i, x_i + (x_{max} - x_i)/K] \times [y_i, y_i + (y_{max} - y_i)/K]$, where $K = 100$.
- `med`: same as `tiny` but using $K = 10$.
- `tall`: we generate the second point at random within $[x_i, x_i + (x_{max} - x_i)/25] \times [y_i, y_{max}]$.
- `wide`: we generate the second point at random within $[x_i, x_{max}] \times [y_i, y_i + (y_{max} - y_i)/25]$.

Figure 1 shows roughly how the six types of queries look.

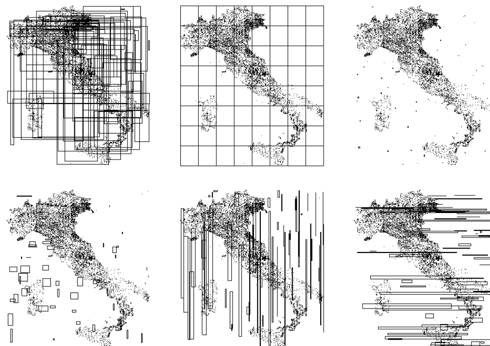

Fig. 1. Example of queries generated for `Italy`. From left to right, starting from the top row: `rand`, `tile`, `tiny`, `med`, `tall` and `wide`.

4.4 Range Searching with Different Partitioning Methods

We next measured the searching time considering the two variants: `binary` for binary searching every chain and `adaptive` for the version that binary searches a candidate chain to start, and uses the fact that the chains are untangled. Figures 2 and 3 show the results for the sets `LRB` and `World`. As we can see in Figure 2, for the `binary` variant, the running time depends on the number of chains obtained by the partitioning method, as expected. For the case of `adaptive`, the running time of `aa` and `ad` is sometimes better, since they do a binary search over all chains at once. In theory, `binary` should be better if $ad > c$, where a and d are the number of ascending and descending chains generated by the partitioning method, respectively, and c is the number of chains generated if we only consider

[3] By random we mean uniformly at random unless stated otherwise.

one direction. However, in this argument, we are ignoring the number of points in each chain, which affects the final result.

In general `gr` behaves reasonably well compared to the other partitioning methods for `binary` and `adaptive`. For this reason, we use this partitioning method for the remaining experiments.

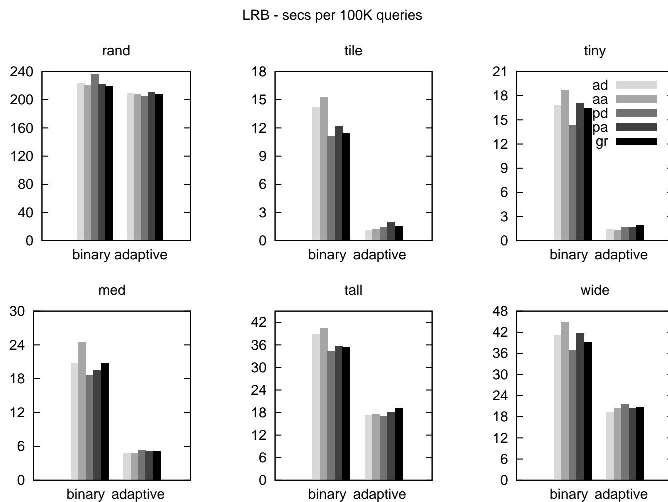

Fig. 2. Detailed time for U-Chains on `LRB`.

4.5 Comparison to Previous Work

We tested our implementations against two structures implemented in the well known CGAL library [1] (version 3.4, default Ubuntu 9.10, multiverse packaging): Kd-tree and Range-tree. The code was compiled with flags `-O2 -frounding-math -Wall`.

Figure 4 shows the results obtained for the four data sets. We do not include Range-trees for `World` since the process required more than $7GB$ of RAM. For the larger data sets, `adaptive` dominates for all types of queries. In the two smaller data sets `adaptive` is competitive with the two other approaches considered, yet it does not present a clear time advantage with respect to Range-trees. The main advantage is the space consumption, when range-trees take more than $7GB$ of RAM for `World`, adaptive requires only about $30MB$. The u-chains structure offers this interesting trade-off, but does not allow updates.

We include the table of values for `LRB`, we highlight every value that is within 5% of the minimum obtained.

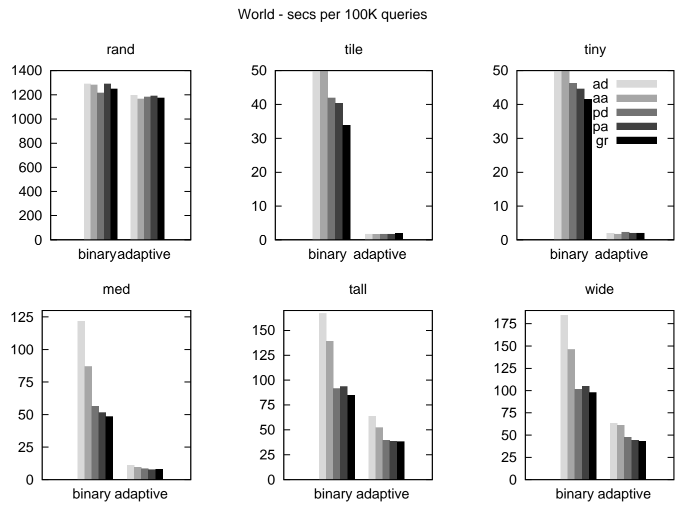

Fig. 3. Detailed time for U-Chains on `World`.

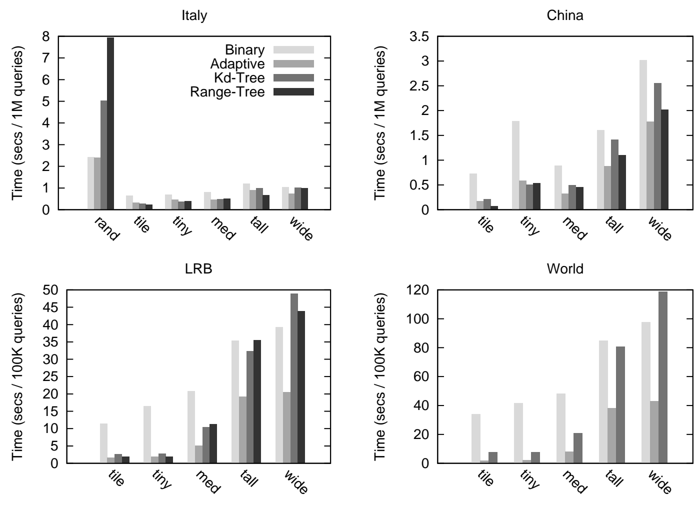

Fig. 4. Time for range queries in the data sets.

Query Type	Partitioning Method	Time (sec) Binary	Time (sec) Adaptive	Time (sec) Kd-Tree	Time (sec) Range-tree
rand	aa	223.62	**208.90**	757.38	1144.92
	ad	221.03	**208.24**		
	gr	235.65	**205.06**		
	pa	222.15	**210.22**		
	pd	219.46	**207.13**		
tile	aa	14.21	**1.12**	2.60	1.87
	ad	15.26	1.17		
	gr	11.16	1.44		
	pa	12.21	1.90		
	pd	11.39	1.58		
tiny	aa	16.85	**1.36**	2.72	1.88
	ad	18.69	**1.32**		
	gr	14.27	1.62		
	pa	17.10	1.71		
	pd	16.48	1.92		
med	aa	20.82	**4.70**	10.43	11.23
	ad	24.46	**4.82**		
	gr	18.57	5.25		
	pa	19.42	5.11		
	pd	20.82	5.07		
tall	aa	38.71	**17.17**	32.35	35.49
	ad	40.32	**17.46**		
	gr	34.20	**16.87**		
	pa	35.60	17.96		
	pd	35.42	19.19		
wide	aa	41.00	**19.39**	48.99	43.85
	ad	44.86	20.42		
	gr	36.72	21.53		
	pa	41.65	20.53		
	pd	39.21	20.57		

Table 3. Results for LRB

5 Conclusions

In this paper we presented a practical implementation of the u-chains data structure. The experimental results show that this structure is efficient and suitable for two-dimensional orthogonal range queries over static data sets. In particular, for our two largest data sets, the adaptive search method on the u-chains structure significantly outperformed both range-trees and kd-trees. Although both of these structures are dynamic, we note that there are likely further optimizations that can be applied to our search methods to improve performance. A secondary advantage of the u-chains structure is that it only occupies $2n+O(\sqrt{n})$ words. Unlike range-trees and kd-trees, which require many pointers per node, each chain in the u-chains structure only stores its ordered set of points and its length. This small footprint certainly does not hurt performance, as it is likely to have better cache locality than the other data structures.

We included a new construction method (untangling algorithm) that can handle much larger data sets than the original proposal. The main advantage of the new algorithm is that it often terminates in much less than k passes, and, in the cases studied here, k is usually close to $\sqrt{n}$.

Our results leave many interesting open problems related to the u-chains:

- It would be an interesting challenge to make this structure dynamic, allowing for insertions and deletions.
- The performance of the structure could be improved by implementing fractional cascading, or any method that re-uses computational effort when jumping from chain to chain. It would be interesting to see how much fractional cascading would improve the search time.

– We could implement different methods to search the chains, like interpolation search or galloping search [5]. The impact of those variations will depend on the data sets considered, and a study of their impact would be of interest.
– Finally, we could look at ways to improve cache locality when searching the chains. For instance, considering the van Emde Boas layout for representing each chain [9].

References

1. CGAL, Computational Geometry Algorithms Library, http://www.cgal.org
2. Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Munro, J.I., Nicholson, P.K., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search, available at `http://www.recoded.cl/docs/untangling.pdf`
3. Arroyuelo, D., Claude, F., Dorrigiv, R., Durocher, S., He, M., López-Ortiz, A., Munro, J.I., Nicholson, P.K., Salinger, A., Skala, M.: Untangled monotonic chains and adaptive range search. In: 20th International Symposium on Algorithms and Computation (ISAAC). pp. 203–212 (2009)
4. Bar Yehuda, R., Fogel, S.: Partitioning a sequence into few monotone subsequences. Acta Informatica 35(5), 421–440 (1998)
5. Barbay, J., López-Ortiz, A., Lu, T., Salinger, A.: An experimental investigation of set intersection algorithms for text searching. J. Exp. Algorithmics 14, 3.7–3.24 (2009)
6. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM 18(9), 509–517 (1975)
7. Chazelle, B., Guibas, L.J.: Fractional Cascading: I. A Data Structuring Technique. Algorithmica 1(2), 133–162 (1986)
8. Di Stefano, G., Krause, S., Lübbecke, M.E., Zimmermann, U.T.: On minimum k-modal partitions of permutations. Journal of Discrete Algorithms 6(3), 381–392 (2008)
9. van Emde Boas, P.: Preserving order in a forest in less than logarithmic time. In: 16th Annual Symposium on Foundations of Computer Science (FOCS). pp. 75–84 (1975)
10. Fomin, F.V., Kratsch, D., Novelli, J.C.: Approximating minimum cocolorings. Information Processing Letters 84(5), 285–290 (Dec 2002)
11. Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: 1984 ACM SIGMOD international conference on Management of data (SIGMOD). pp. 47–57. ACM, New York, NY, USA (1984)
12. Kanth, K.V.R., Singh, A.K.: Optimal dynamic range searching in non-replicating index structures. In: 7th International Conference on Database Theory (ICDT). pp. 257–276. Springer-Verlag, London, UK (1999)
13. Lueker, G.S.: A data structure for orthogonal range queries. In: 19th Annual Symposium on Foundations of Computer Science (SFCS). pp. 28–34. IEEE Computer Society, Washington, DC, USA (1978)
14. Supowit, K.J.: Decomposing a set of points into chains, with applications to permutation and circle graphs. Information Processing Letters 21(5), 249–252 (1985)
15. Yang, B., Chen, J., Lu, E., Zheng, S.: Design and Performance Evaluation of Sequence Partition Algorithms. Journal of Computer Science and Technology 23(5), 711–718 (2008)