Indexed Geometric Jumbled Pattern Matching

S. Durocher[1] R. Fraser[1,2] *T. Gagie*[3]
D. Mondal[1] M. Skala[1] S. Thankachan[4]

[1]University of Manitoba

[2]Google

[3]University of Helsinki,
Helsinki Institute for Information Technology

[4]University of Waterloo

CPM '14

Background

Suppose we have a string $S[1..n]$ over an alphabet $\{a_1, \ldots, a_\sigma\}$ and we want to build an index such that later, given a vector $\langle c_1, \ldots, c_\sigma \rangle$, we can quickly determine whether there is a substring in which each a_i occurs c_i times.

Cicalese and Lipták (2009) gave an $\mathcal{O}(n)$-space index with $\mathcal{O}(1)$ query time for the case $\sigma = 2$, and Kociumaka et al. (2013) gave an $o(n^2)$-space index with $o(n)$ query time for the general case.

We generalize both results to two dimensions but we discuss here only our generalization of Cicalese and Lipták's result.

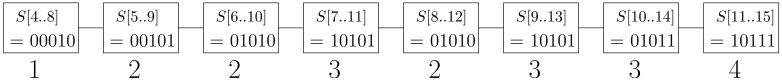
CL's Index

Suppose $S = 111000101010 1110$. The 5-bit substring with the least 1s is $S[4..8]$ with 1, and the one with the most is $S[11..15]$ with 4. It follows that $S[4..15]$ contains 5-bit substrings with 1, 2, 3 and 4 copies of 1, but S doesn't contain any 5-bit substrings with 0 or 5 copies of 1.

For each possible substring length ℓ, if we store the minimum and maximum number of 1s in a substring of length ℓ, then we can *detect* a substring matching a query vector. If we store S in a data structure supporting fast rank queries, then we can use binary search in $S[4..13]$ to *locate* such a substring.

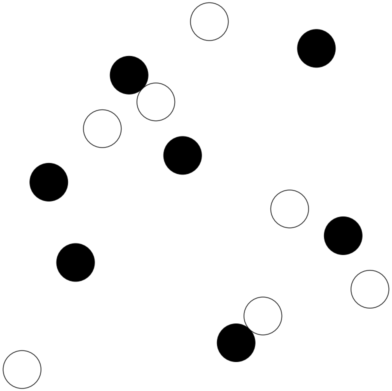
Path of Substrings

For locating, we store a rank data structure for S and use binary search on the path of consecutive ℓ-tuples between one with the minimum number of 1s and one with the maximum number of 1s.



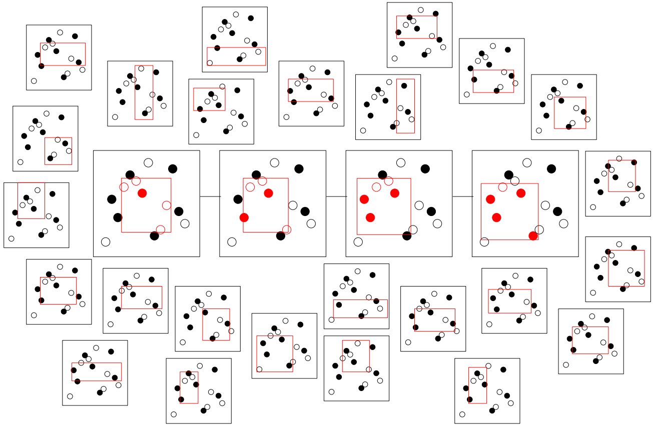
Point Sets

Now suppose we want to build an index for a set of black-or-white points in general position (i.e., with distinct x- and y-coordinates) that lets us quickly find axis-aligned rectangles containing specified numbers of black and white points.



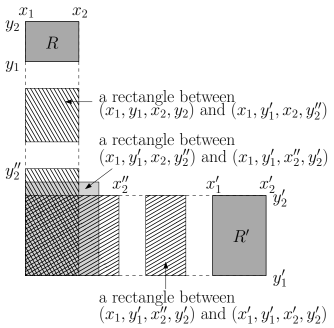
Path of Rectangles

Since the points are in general position, there *exists* a path of ℓ-point rectangles between one with the minimum number of black points and one with the maximum number of black points, but how do we binary search in it?

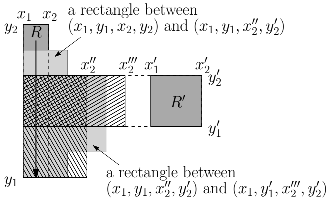


"Amoeboid Motion"

We can move the rectangle like an amoeba — stretching out on one side and contracting on the opposite side — changing direction at most once.



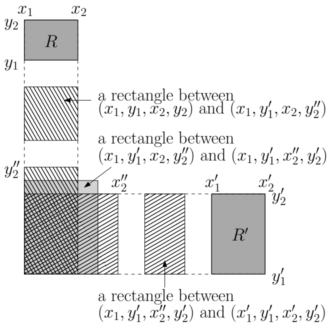
x_1
x_2
y_2
R
y_1
a rectangle between (x_1, y_1, x_2, y_2) and (x_1, y_1', x_2, y_2'')
a rectangle between (x_1, y_1', x_2, y_2'') and (x_1, y_1', x_2'', y_2')
y_2''
x_2''
x_1'
x_2'
y_2'
R'
y_1'
a rectangle between (x_1, y_1', x_2'', y_2') and (x_1', y_1', x_2', y_2')

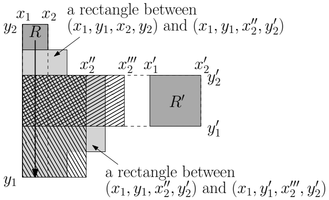


x_1
x_2
y_2
R
a rectangle between (x_1, y_1, x_2, y_2) and (x_1, y_1, x_2'', y_2')
x_2''
x_2'''
x_1'
x_2'
y_2'
R'
y_1'
y_1
a rectangle between (x_1, y_1, x_2'', y_2') and $(x_1, y_1', x_2''', y_2')$

“Amoeboid Motion”

We can move the rectangle like an amoeba — stretching out on one side and contracting on the opposite side — changing direction at most once.


x_1
x_2
y_2
R
y_1
a rectangle between (x_1, y_1, x_2, y_2) and (x_1, y'_1, x_2, y''_2)
a rectangle between (x_1, y'_1, x_2, y''_2) and (x_1, y'_1, x''_2, y'_2)
y''_2
x''_2
x'_1
x'_2
y'_2
R'
y'_1
a rectangle between (x_1, y'_1, x''_2, y'_2) and (x'_1, y'_1, x'_2, y'_2)


x_1
x_2
a rectangle between (x_1, y_1, x_2, y_2) and (x_1, y_1, x''_2, y'_2)
y_2
R
x''_2
x'''_2
x'_1
x'_2
y'_2
R'
y'_1
y_1
a rectangle between (x_1, y_1, x''_2, y'_2) and $(x_1, y'_1, x'''_2, y'_2)$

Proof by incomprehensible diagram? :o)

Range Queries

To binary search in the amoeba-rectangle's path, we use an $\mathcal{O}(n)$-space data structure by Brodal et al. (2011) that supports 3-sided range selection queries in $\mathcal{O}(\log n / \log \log n)$ time. We also store an $\mathcal{O}(n)$-space data structure that support range counting queries on the black points only.

For each step of the binary search, we decide where we want the "front" of the rectangle, then we use a range-selection query to find where the "back" should be such that it contains ℓ points. Once we know the boundaries of the rectangle we're checking, we use two range-counting queries to determine how many black points it contains.

Space and Query Time

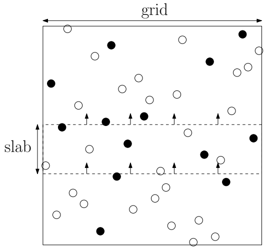
Lemma

Given n black-or-white points in general position, we can build an $\mathcal{O}(n)$-space data structure such that later, given numbers of black points and white points, in $\mathcal{O}\left(\log^2 n / \log\log n\right)$ time we can find an axis-aligned rectangle matching that query if one exists.

Construction

To build our index, we must find for each $\ell \leq n$ the ℓ-point rectangles containing the least and most black points.

We reduce to rank space and, for $\ell \leq h \leq n$, we sweep a slab of height h up the grid while keeping track of the minimum and maximum numbers of 1s in a substring of length ℓ in the binary string obtained by collapsing the slab to one dimension.



Dynamic Index for Strings

Lemma

We can maintain a dynamic binary string of up to n bits such that inserting or deleting a bit takes $\mathcal{O}(n \log n)$ time and, given ℓ, determining the minimum and maximum number of 1s in any substring of length ℓ takes $\mathcal{O}(1)$ time.

Corollary

We can build our index for points sets in $\mathcal{O}\left(n^3 \log n\right)$ time.

Summary

Theorem

Given n black-or-white points in general position, we can build an $\mathcal{O}(n)$-space data structure such that later, given numbers of black points and white points, in $\mathcal{O}\left(\log^2 n / \log\log n\right)$ time we can find an axis-aligned rectangle matching that query if one exists.

Summary

Theorem

Given n black-or-white points in general position, we can build an $\mathcal{O}(n)$-space data structure such that later, given numbers of black points and white points, in $\mathcal{O}\left(\log^2 n / \log\log n\right)$ time we can find an axis-aligned rectangle matching that query if one exists.

So, now that the community has looked at jumbled pattern matching in strings, trees, graphs and point sets,

what's next?