Non-orthogonal Homothetic Range Partial-Sum Query on Integer Grids [Extended Abstract]

Yuan Tang[1,2(✉)] and Haibin Kan[1]

[1] School of Computer Science, School of Software,
Shanghai Key Laboratory of Intelligent Information Processing,
Fudan University, Shanghai, China
{yuantang,hbkan}@fudan.edu.cn
[2] State Key Laboratory of Computer Architecture, ICT, CAS, Beijing, China

Abstract. Algorithms for range partial sum query on high dimensional integer grids typically focus on orthogonal ranges, which by definition demands fixed right triangles between all adjacent boundary edges. We extend the algorithm to solve 2D homothetic triangular range queries in $\langle O(N\alpha(N)), O(\alpha^2(N))\rangle$ ($\langle$preprocessing bound, query bound$\rangle$ of both time and space since they are identical.), where N is the total number of grid points and $\alpha(\cdot)$ is a functional equivalence of the inverse Ackermann function. This asymmetric bound improves over the existing bound for orthogonal ranges. By the property of homotheticity, we mean that the angles between any two adjacent boundaries are arbitrarily fixed constants. The technique and bounds of our work can be extended to even higher dimensional grids.

Keywords: Range partial sum query · Triangular reduction
Non-orthogonal homothetic range

1 Introduction

In this paper, we consider the following range partial sum query problem (range query problem for short). Given a static d-dimensional integer grid A of dimension $n_1 \times n_2 \ldots \times n_d$ with each grid point[1] holding a value drawn from a semigroup $(S, \oplus)$, how fast can we answer online, if preprocessing is allowed, partial sum queries of a simple shape region Q (i.e. no self-intersection of boundary edges). For a static grid, we disallow dynamic insertion or deletion of any point values.

$$sum(Q) = \sum_{(k_1,\ldots,k_d)\in Q} A(k_1, \ldots, k_d). \tag{1}$$

Y. Tang—Supported in part by the Open Funding (No. CARCH201606).

[1] For convenience, we use the term "grid point" and "point" interchangeably in the rest of paper.

J. Chen and P. Lu (Eds.): FAW 2018, LNCS 10823, pp. 273–285, 2018.
https://doi.org/10.1007/978-3-319-78455-7_21

Our problem differs from the classic orthogonal range query problem in that orthogonal range by definition demands right triangles (90°) between any two adjacent boundary edges, while we allow arbitrarily fixed angles. We only require that all angles are known to the preprocessing algorithm and can not change in the subsequent queries. We use the general term *triangular query* to stand for *triangular shape range partial sum query in a* 2*D-grid*, *tetrahedral shape range partial sum query in a* 3*D-grid*, and/or similar extensions to higher dimensional grids. Since $(S, \oplus)$ is a semigroup, $\oplus$ operation is an associative operator, i.e. $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ holds for all x, y, z in the semigroup. No other restrictions are imposed on this semigroup as is sometimes done by similar problems in the literature. For instances, we do not assume idempotence, i.e. $a \oplus a = a, \forall a \in S$, which is required by the Berkman-Vishkin's algorithm [1]; We do not assume any partial or total ordering among elements in S as do RMQ (Range Minimum Query) algorithms; We do not assume the existence of an inverse operation of $\oplus$, otherwise a trivial algorithm exists [2].

We use the same arithmetic model for complexity analysis as in Yao [2,3], Chazelle and Rosenberg [4,5]. In the arithmetic model, the space bound is given in units of semigroup elements instead of bits. The preprocessing and query time are calculated in number of $\oplus$ operations, ignoring the time to find the proper memory cells[2]. We use the notation $\langle O(f(N)), O(g(N))\rangle$ to denote the complexity bounds of a pair of preprocessing and corresponding query algorithm, respectively. Usually, the space and time bounds of these algorithms are identical so that we do not differentiate.

Motivation

This research was motivated by our study on the Pochoir stencil compiler [8], where we have to query the properties of an arbitrary d-dimensional octagonal shape region. The octagonal shape comes from the projection of a $(d+1)$-dimensional hyper-zoid[3] onto a d-dimensional spatial grid. Apparently, a straightforward way to answer an octagonal range query is to decompose it into a set of rectangular and triangular shape queries. The range query is about partial sum because each grid point may contains various properties for the compiler to collect in order to generate an efficient kernel function for the region.

Our Contributions

1. We extend the range partial sum query problem on integer grid from orthogonal to non-orthogonal shapes. In particular, we solve the 2D triangular problem in $\langle O(N\alpha(N)), O(\alpha^2(N))\rangle$ ($\langle$preprocessing bound, query bound$\rangle$ of either

[2] In the RAM model, we can locate a proper memory cell in a recursive divide-and-conquer tree by finding the Lowest Common Ancestor (LCA) of the end vertices i of the query range. The LCA problem can in turn be solved by a ± 1 RMQ algorithm with linear preprocessing time and $O(1)$ query time [6,7].

[3] "Hyper-zoid" is a trapezoidal analogue in a $(d+1)$-dimensional grid, where $d > 1$. The $(d+1)$-dimensional grid composes of a d-dimensional spatial grid plus a 1-dimensional temporal axis.

time or space since they are identical.), where N is the total number of grid points and $\alpha(\cdot)$ is a functional equivalence of the inverse Ackermann function [9]. This result improves over the previous result on rectangular problem, i.e. the $\langle N\alpha^2(N), \alpha^2(N)\rangle$ bounds [4,5].

2. We make the following algorithmic contributions:
   (a) We generalize the "dimension reduction" technique introduced by Chazelle and Rosenberg [4,5] to "triangular data reduction" ("triangular reduction" in short) in Sect. 3;
   (b) We show that an arbitrary triangular problem can be reduced to an Isosceles Right Triangular (IRT) problem, which can be solved similar to the square problem (Sect. 2) based on the observation that it has only one degree of freedom for scaling;
   (c) We generalize a recursive algorithmic scheme to get an $\alpha(\cdot)$ bound in the end of Sect. 2.
3. We conjecture that the optimal bounds of homothetic triangular problem in an arbitrary d-dimensional integer grid, where $d > 1$ is a constant, is $\langle O(N\alpha(N)), O(\alpha^d(N))\rangle$, in contrast to the $\langle O(N\alpha^d(N)), O(\alpha^d(N))\rangle$ bounds of orthogonal problem.

2 Square Shape Range Partial Sum Query Problem

This section considers a simpler problem where the query ranges are of square shape, i.e. the height over width must be a fixed $1:1$ aspect ratio, the study of which will serve as a warm-up for the later homothetic triangular problem to be discussed in Sect. 3. Intuitively, the square problem can be solved more efficiently because a rectangle can scale independently on either dimension, i.e. having two degrees of freedom, while a square has only one in order to keep the $1:1$ aspect ratio.

Lemma 1. *There is an algorithm of $\langle O(n_1 n_2 \log(n_1 + n_2)), O(1)\rangle$ bounds to solve the 2D square range partial sum query problem on an integer grid of dimensions $N = n_1 \times n_2$,*

Proof. We prove the lemma by construction as follows. Without loss of generality, we assume that $n_1 \geq n_2$. Referring to Fig. 1, the preprocessing algorithm works as follows.

1. We divide the $n_1 \times n_2$ grid into a 2×2 subgrids, each of dimension $n_1/2 \times n_2/2$. We store on each point p inside a subgrid g four partial sums reduced from all values contained within the maximum rectangle bounded by p and one of the four corner vertices of g. We call this procedure ***corner reduction***, respectively the partial sums ***corner values***. The corner reduction can accomplish in $O(n_1 n_2)$ time and space using dynamic programming.
2. We recursively divide each subgrid into a 2×2 array of subgrids, each of dimension $n_1/2^2 \times n_2/2^2$, and perform corner reductions for every points within every subsubgrids. This procedure continues until the size of each subgrid reaches $O(1)$.

Apparently, the cost of dynamic programming at each level of recursion is $O(n_1 n_2)$, and there are $O(\log n_2)$ levels of recursion. The total preprocessing overhead, for either time or space, thus sums up to $P_{\boxdot,0}(n_1, n_2) = O(n_1 n_2 \log n_2)$.

Given an arbitrary square query range with dimension $e_{\boxdot} \in [n_2/2^{k+1}, n_2/2^k)$ for some integer $k \in [0, \log n_2]$, i.e. $0 \leq k \leq \log n_2$, it can stride at most four intersecting subgrids at recursion level k, i.e. subgrid of dimension $n_1/2^k \times n_2/2^k$, because of the fixed $1:1$ aspect ratio. In other words, there will be exactly one corner vertex of the four intersecting subgrids sitting inside the square query range. If we assume that locating the intersecting corner vertex as well as the four neighboring subgrids is free (as is true in the arithmetic model or the RAM model, see the footnote in Sect. 1), the partial sum of the query range is then a simple summation of the four corner values stored in the vertices. $Q_{\boxdot,0}(n_1, n_2) = 3 = O(1)$. We name the procedure ***corner query***. □

Theorem 1. *There is an algorithm of* $\langle O(n_1 n_2 \alpha(n_1 + n_2)), O(\alpha^2(n_1 + n_2))\rangle$ *bounds to solve the 2D square range partial sum query problem on an integer grid of dimension* $N = n_1 \times n_2$.

Proof. Referring to Figs. 1 and 2, we prove the theorem by constructing a recursive algorithmic scheme as follows. The inputs to the recursive algorithmic scheme are:

1. A square algorithm of $\langle P_{\boxdot,k}(n_1, n_2) = O(n_1 n_2 f(n_1 + n_2)), Q_{\boxdot,k}(n_1, n_2) = O(1)\rangle$ bounds, where the subscript k indicates the k-th recursive application to the scheme, and $f(n) < n - 2$ is a function of problem dimension n.
2. A 1D algorithm of $\langle P_{-,k}(n) = O(nf(n)), Q_{-,k}(n) = O(1)\rangle$ [2,3,9].

For simplicity, we assume that $n_1 = \Theta(n_2)$ in the following analysis.

The preprocessing algorithm *recursively* partitions the input grid of dimension $n_1 \times n_2$ into a 2D $\frac{n_1}{f(n_1+n_2)} \times \frac{n_2}{f(n_1+n_2)}$ array of subgrids, each of dimension $f(n_1+n_2) \times f(n_1+n_2)$, until the size of each subgrid reaches $O(1)$. At each level of recursion, the preprocessing algorithm $P_{\boxdot,k+1}$ conducts corner reductions, line reductions, and block reductions on subgrids as follows.

1. ***Corner reduction:*** The algorithm performs corner reductions for every point with respect to the containing subgrid as in Lemma 1. The preprocessing overhead of each subgrid is $O(f^2(n_1 + n_2))$ and sums up to $O(n_1 n_2)$ over the entire input grid.
2. ***Line reduction:*** Firstly, the algorithm reduces each horizontal line segment of length $f(n_1 + n_2)$ within each subgrid into one single value by the $\oplus$ operation. That is, a horizontal line of length n_2 is reduced to a 1D array of $n_2/f(n_1+n_2)$ reduced values. The algorithm then calculates the partial sums of the $f(n_1 + n_2)$ reduced values (from horizontal line segments) within each subgrid with respect to the top and bottom boundary edges, respectively. That is, for the input grid with n_1 rows of horizontal lines, there will be $2n_1$ (one n_1 rows comes from the reduction with respect to the top edge, another

n_1 rows comes from the reduction with respect to the bottom edge) arrays, each of $n_2/f(n_1+n_2)$ ***line value***s. Thirdly, the algorithm applies the 1D preprocessing algorithm, i.e. $P_{-,k}$, to the $2n_1$ arrays of horizontal line values. Symmetrically, the algorithm applies the line reductions to all vertices line values. The horizontal line reduction takes $2n_1 \cdot P_{-,k}\left(\frac{n_2}{f(n_1+n_2)}\right) = O(n_1 n_2)$ time and space, which is asymptotically the same as the vertical line reduction. The overall line reductions thus sum up to $O(n_1 n_2)$ space and time as well.

3. ***Block reduction:*** The algorithm reduces all points within each subgrid into a single partial sum by the $\oplus$ operation and get a 2D array of $\frac{n_1}{f(n_1+n_2)} \times \frac{n_2}{f(n_1+n_2)}$ ***block value***s. We then apply the preprocessing algorithm of $P_{\square,k}$, i.e. the input square algorithm, to the array of block values. The block reduction will take $P_{\square,k}(n_1/f(n_1+n_2), n_2/f(n_1+n_2)) = O(\frac{n_1}{f(n_1+n_2)} \cdot \frac{n_2}{f(n_1+n_2)} \cdot f(\frac{n_1}{f(n_1+n_2)})) = O(n_1 n_2)$ for either time or space.

Recursive reduction: We recursively apply the above corner, line, and block reductions to all 2×2 array of subgrids of dimension $2f(n_1+n_2) \times 2f(n_1+n_2)$ with subsubgrids of dimension $f(f(n_1+n_2)) \times f(f(n_1+n_2))$. By a recursive application to all such 2×2 array of subgrids, we cover the case when a lower level (with recursion level $> k$) square query range sits between the boundary of two adjacent subgrids based on the fact that it can not stride more than four subgrids of level-k.

The preprocessing overhead of one round of above reductions can be calculated as follows. If we define $f^{(i)}(n) = n$ if $i = 0$ and $f^{(i)}(n) = f(f^{(i-1)}(n))$ if $i > 0$, it's not hard to see that the preprocessing overhead of any 2D array of dimension $2f^{(k)}(n_1+n_2) \times 2f^{(k)}(n_1+n_2)$ with subgrids of dimension $f^{(k+1)}(n_1+n_2) \times f^{(k+1)}(n_1+n_2)$ sum up to $O((f^{(k)}(n_1+n_2))^2)$. Since the entire $n_1 \times n_2$ grid has $O((\frac{n_1 n_2}{f^{(k)}(n_1+n_2)})^2)$ such 2D arrays, the overall overheads over the entire grid sum up to $O(n_1 n_2)$ for any recursion level. Since the preprocessing proceeds recursively for $f^*(n_1+n_2)$ levels[4], the overall preprocessing overheads, for either time or space, are $O(n_1 n_2 f^*(n_1+n_2))$.

The query algorithm works as follows. Assuming that the input square query range is of dimension $e_\square \in [f^{(k+1)}(n_1+n_2), f^{(k)}(n_1+n_2))$, i.e. the square is of dimension $e_\square \times e_\square$, the key observation is that it can stride at most one 2×2 array of subgrids of dimension $2f^{(k)}(n_1+n_2) \times 2f^{(k)}(n_1+n_2)$. Since we have preprocessed all such 2×2 array of subgrids, we can answer the query at recursion level k by that specific 2×2 array of subgrids. More specifically, it can be answered by an "***inter-block query***", i.e. a partial sum on the results returned from the following queries: one square query on the data structure preprocessed by "block reduction", two horizontal line queries (1D queries), two vertical line queries (1D queries) by "line reduction", and four corner queries by "corner reduction". Since we assume that we can locate a proper recursion level and the specific 2×2 array of subgrids in $O(1)$ time and the query time of all

[4] We define $f^*(n) = 0$ if $n \leq 1$, otherwise $f^*(n) = 1 + f^*(f(n))$.

input algorithms at any recursion level is $O(1)$, the total query overhead sums up to $O(1)$ as well.

This completes one round of application to the recursive algorithmic scheme for the square problem. The resulting algorithm is of $\langle O(n_1 n_2 f^*(n_1+n_2)), O(1)\rangle$ bounds. If we keep supplying the resulting more advanced algorithm into the above recursive algorithmic scheme, we eventually will get the optimal $\langle O(n_1 n_2 \alpha(n_1+n_2)), O(\alpha^2(n_1+n_2)\rangle$ algorithm, where $\alpha(n) = \min\{k | \log^{\overbrace{**\cdots*}^{k}}(n) \leq 2\}$ is a functional equivalence of the inverse Ackermann function [9].

We need a bit more explanation on the query bound. The recurrence of query time is $Q_{\boxdot,k+1}(n_1, n_2) = Q_{\boxdot,k}(n_1/f(n_1+n_2), n_2/f(n_1+n_2)) + 4Q_{-,k}(n_2/f(n_1+n_2)) + 3$ according to the above "inter-block query" procedure. Since the query overhead of 1D algorithm $Q_{-,\alpha(n)} = O(\alpha(n))$ and the recursive application can continue up to $\alpha(n_1+n_2)$ rounds, we have $Q_{\boxdot,\alpha(n_1+n_2)}(n_1, n_2) = 4\sum_{k=0}^{\alpha(n_1+n_2)} Q_{-,k}(n_1, n_2) + 3\alpha(n_1+n_2) = O(\alpha^2(n_1+n_2))$. □

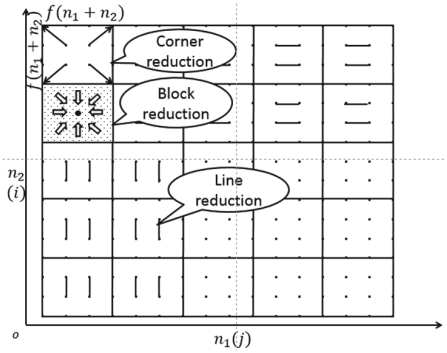

Fig. 1. This diagram depicts the corner, line, and block reductions.

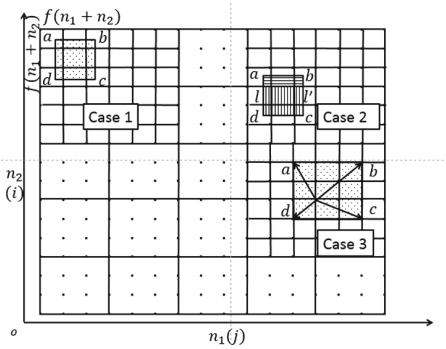

Fig. 2. This diagram demonstrates several possible square shape queries

In summary, our approach, namely the "***recursive algorithmic scheme***", has the following general framework.

1. We design an initial algorithm $\mathcal{I}$ that solves the problem correctly, but not necessarily very efficiently.
2. We develop a recursive algorithmic scheme $\mathcal{M}$, into which we plug $\mathcal{I}$ and make the resulting algorithm $\mathcal{M}(\mathcal{I})$ behave functionally identical to $\mathcal{I}$, but with asymptotically different (ideally improved) complexity bound. $\mathcal{M}$ remains oblivious of the internal structure of $\mathcal{I}$, but the knowledge of the asymptotic bounds of $\mathcal{I}$ may help.
3. We repeat the above two steps for several rounds where in any given iteration $i > 0$ the input algorithm $\mathcal{M}^{(i-1)}(\mathcal{I})$ is the resulting algorithm from the $(i-1)$-th application of the recursive algorithmic scheme $\mathcal{M}$. The goal of repeated self-application is to reach even better bounds (ideally an $\alpha(\cdot)$ bound).

3 Triangular Range Query Problem

This section discusses the 2D triangular problem. We assume that all angles of the triangular shape are fixed constants and known to the preprocessing algorithm. We argue that this is not a weaker version of the classic orthogonal range query problem because an orthogonal range by definition requires fixed right angles between all adjacent boundaries.

Outline of the high-level idea:

1. Reducing an arbitrary triangular range query problem to an isosceles right triangular (IRT for short) problem: We have an observation that there is always a smallest ***feature triangle*** of the query triangular shape, and we can tile up the entire grid by ***feature parallelogram***s composed of feature triangles and corresponding inverted. A ***feature triangle*** is a smallest triangle that has the query triangular shape and has all its vertices on the grid points. Similarly, we have the notion of ***feature parallelogram***. Referring to Fig. 5, if we reduce every feature triangle and its corresponding feature parallelogram (left-hand side) to two partial sum values (Δ and $\square$ respectively in the figure) and store them as a new grid point, we get a new reduced grid on the right-hand side. By the reduction, an arbitrary triangular range query on the original grid (left-hand side) is equivalent to an IRT query on one of the reduced grids (right-hand side) by aligning its oblique line with one of the tilings of feature parallelograms. To align with all possible input query ranges, we can have up to $O(|\Delta|)$ different reduced grids, where $|\Delta|$ is the number of points internal to a feature triangle. In the rest of paper, we use the term "triangle" to stand for IRT unless otherwise specified.
2. Extending the notion of data reduction to triangular prefix/suffix reduction.
3. The key observation is then an input IRT range of dimension e_Δ will stride at most one 3×3 array of subgrids of dimension $3(e_\Delta/2) \times 3(e_\Delta/2)$, where $e_\Delta/2$ is the dimension of the IRT's largest embedded square.

Organization: We first address how to preprocess all triangular prefix/suffix partial sums in linear time and space by Lemma 2; We then construct an initial algorithm in Theorem 2 and a recursive algorithmic scheme in Theorem 3, respectively.

Definitions and Conventions throughout the section:

1. All coordinate systems in the figures of this paper have horizontal axis indexed from left to right (small coordinate on the left and large on the right), and vertical axis from bottom to top (small coordinate on the bottom and large on the top).
2. Referring to Fig. 3, for columns to the left of L_v, we define the triangular prefix "$v(i,j).\Delta$ prefix" as the partial sum of the largest IRT bounded between the bottom-left vertex (i,j) and partition line L_v, where i is the horizontal coordinate and j is the vertical coordinate, respectively.

3. Referring to Fig. 4, for columns to the right of L_v, we define the triangular suffix "$v(i,j).\Delta$ suffix" as the partial sum of the largest IRT bounded between the top-right vertex (i,j) and partition line L_v.
4. Note that it depends on the orientations of IRT and the partition line whether the reduction to the left, right, upper, lower side of the partition line is for triangular prefix or suffix.
5. We define an IRT's ***core length*** ($e_\square$) as the dimension of its largest embedded square. We denote an IRT's dimension, i.e. the horizontal/vertical base, by e_Δ. Clearly, $e_\Delta = 2e_\square$.

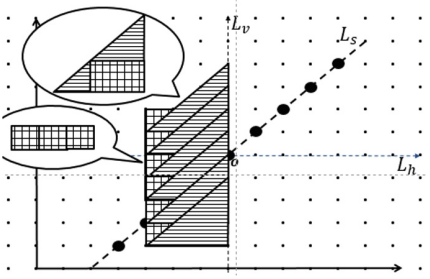

Fig. 3. Triangular prefix

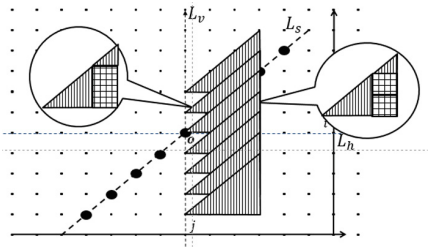

Fig. 4. Triangular suffix

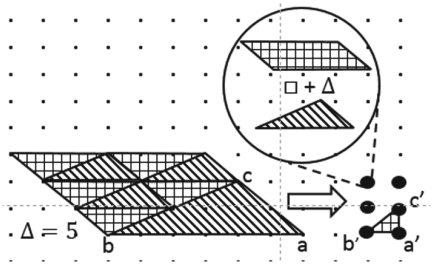

Fig. 5. This diagram shows how to reduce an arbitrary triangular problem to an IRT problem. In this diagram, every grid point on the right-hand side holds two values, i.e. the partial sum of a feature triangle (Δ) and of its feature parallelogram ($\square$).

Lemma 2. *For an IRT range query on a reduced grid, we can preprocess all triangular prefixes and suffixes with respect to a given partition line in time and space linear to the size of grid.*

Proof. Referring to Figs. 3 and 4, we assume without loss of generality a vertical partition line L_v. Initially, each grid point on the reduced grid stores two values, one is the partial sum of a feature triangle (Δ) and the other is the partial sum of the corresponding feature parallelogram ($\square$). We compute the triangular prefixes and suffixes by dynamic programming with respect to L_v as follows.

1. We can compute all triangular prefixes to the left of L_v column by column by the recurrences of (2) and (3).

$$v(i,j).\Delta\,\text{prefix} = v(i,j).\Delta + v(i+1,j+1).\Delta\,\text{prefix} + v(i+1,j).\boxdot \tag{2}$$

$$v(i,j).\boxdot = v(i+1,j).\boxdot + v(i,j).\square \tag{3}$$

In the recurrences, $v(i,j).\Delta$ and $v(i,j).\square$ is the partial sum of feature triangle and corresponding feature parallelogram stored at cell (i,j), and $v(i,j).\boxdot$ denotes the partial sum of one horizontal line segment bounded between coordinate (i,j) and L_v. By the dynamic programming recurrences, we can compute all triangular prefixes in time and space linear to the size of grid.

2. We can compute all triangular suffixes to the right of L_v column by column by the recurrences of (4).

$$v(i,j).\Delta\,\text{suffix} = v(i,j).\Delta + v(i-1,j-1).\Delta\,\text{suffix} + v(i,j-1).\sqsubset\sqsupset \tag{4}$$

In the recurrence, $v(i,j-1).\sqsubset\sqsupset$ stands for the partial sum of one vertical line segment bounded between cell $(i,j-1)$ and the horizontal base of the largest IRT bounded between (i,j) and L_v. In Fig. 4, $v(i,j-1).\sqsubset\sqsupset$ is the vertical line segment shaded by mini-grid.

Observing that the length of $v(i,j).\sqsubset\sqsupset$ is always $i-1$ for all $\sqsubset\sqsupset$ on the same column i, we denote it by $d_i = i-1$, $i \geq 1$. We can then compute all $v(i,j).\sqsubset\sqsupset$ values column by column by the following simple dynamic programming algorithm.

For the i-th column to the right of L_v, we partition it into segments of length $d_i = i-1$, $i \geq 1$. We then compute by dynamic programming for every point on the column the partial sum from itself to the top and bottom vertices of the holding segment and store them as $v(i,j).\sqsubset$ and $v(i,j).\sqsupset$, respectively. Now, it's clear that any $v(i,j).\sqsubset\sqsupset$ can be computed in $O(1)$ time by summing up one $\sqsubset$ and one $\sqsupset$ value of cell (i,j) with respect to the top and bottom vertices of the holding segment.

Since the preprocessing time for $\sqsubset\sqsupset$ values are amortized $O(1)$ for every grid point, it's not hard to see that the entire preprocessing overhead of triangular suffixes are linear as well by (4). □

Theorem 2. *There exists an algorithm that can solve the 2D triangular range partial sum query on a grid of dimension $n_1 \times n_2$ in bounds of $\langle O(n_1 n_2 \log(n_1 + n_2)), O(1)\rangle$.*

Proof. To simplify the discussion, we assume without loss of generality that $n_1 \geq n_2$ and $n_1 = \Theta(n_2)$. Referring to Figs. 6 and 7, we construct a preprocessing algorithm for 2D triangular range query problem as follows:

1. We divide the $n_1 \times n_2$ grid into a 2×2 array of subgrids, each of dimension $n_1/2 \times n_2/2$, ignoring the at most one (1) row or column difference between subgrids to simplify the discussion. More generally, referring to

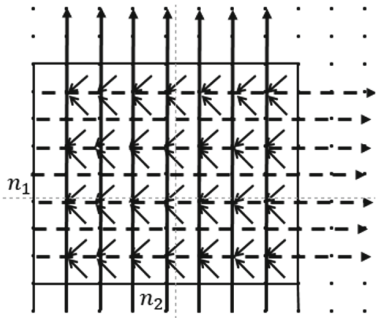

Fig. 6. Perform corner reduction (up to 2 directions) in initial triangular algorithm

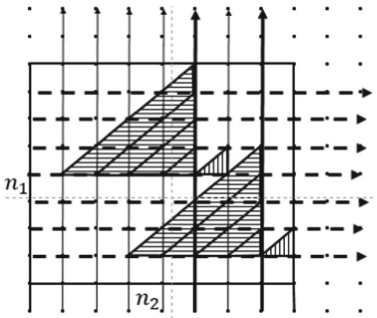

Fig. 7. Perform triangular prefix/suffix (up to 2x) in initial triangular algorithm

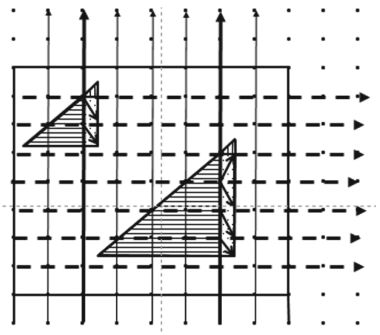

Fig. 8. Query in initial triangular algorithm

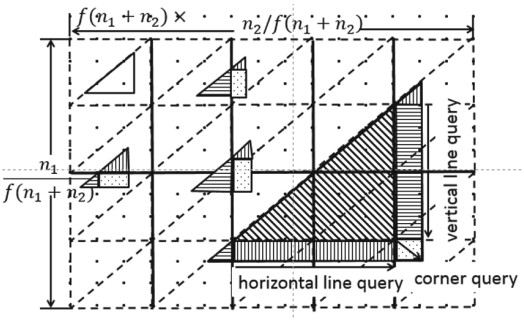

Fig. 9. Query in recursive algorithmic scheme for triangles

Fig. 7, at each recursion level k, where $k \in [0, \log n_2]$, we divide each grid of dimension $n_1/2^k \times n_2/2^k$ into a 2×2 array of subgrids, each of dimension $n_1/2^{k+1} \times n_2/2^{k+1}$. Within each subgrid, we perform corner reductions for all points with respect to two vertices, depending on the orientation of the query triangular shape; We also compute triangular prefixes and suffixes for all grid points up to 2x of the subgrid's dimension with respect to every vertical boundaries. The reason behind the 2x triangular prefixes and suffixes computation will become clear in the query algorithm. Since the preprocessing complexity for both corner reduction and triangular prefixes/suffixes are linear, the overall overhead for any recursion level is clearly linear.

2. Summing up the overhead over all $\log n_2$ recursion levels and $O(|\Delta|)$ different tilings yields the claimed preprocessing bound, where $|\Delta|$ is a constant for any given query triangular shape.

The query algorithm works as follows. Given an arbitrary input triangular query range, we firstly align its oblique line to one of the $O(|\Delta|)$ tilings, thus

reduce it to an IRT range query. Analogous to the case of square range query (Lemma 1), we have an observation that an IRT range query with core length $e_\square \in [n_2/2^{k+1}, n_2/2^k)$ can stride at most one 3×3 array of subgrids with dimension $3(n_1/2^k) \times 3(n_2/2^k)$, thus can be answered by summing up one triangular prefix (up to 2x of the subgrid's dimension), one triangular suffix, and up to three (3) corner queries as shown in Fig. 8. □

Theorem 3. *There exists a recursive algorithmic scheme that can solve the* $2D$ *triangular range partial sum query on a grid of dimension* $n_1 \times n_2$ *in bounds of* $\langle O(n_1 n_2 \alpha(n_1 + n_2)), O(\alpha^2(n_1 + n_2))\rangle$.

Proof. Referring to Fig. 9, we construct a recursive algorithmic scheme as follows.

We assume an input triangular algorithm with $\langle P_{\Delta,i}(n_1, n_2) = O(n_1 n_2 f(n_1 + n_2)), Q_{\Delta,i}(n_1, n_2) = O(1)\rangle$ bounds, and an input 1D algorithm with $\langle P_{-,i}(n) = O(nf(n)), Q_{-,i}(n) = O(1)\rangle$ bounds, where $f(n) < n - 2$ is a function of the input problem's dimension and subscript i specifies the rounds of repeated self-application to the recursive algorithmic scheme as in the case of square problem (proof of Theorem 1). Again, we assume $n_1 = \Theta(n_2)$ to simplify the discussion.

Analogous to the recursive algorithmic scheme for the square problem (proof of Theorem 1), at recursion level k, the preprocessing algorithm $P_{\Delta,i}$ partitions an $f^{(k)}(n_1 + n_2) \times f^{(k)}(n_1 + n_2)$ grid into a $\frac{f^{(k)}(n_1+n_2)}{f^{(k+1)}(n_1+n_2)} \times \frac{f^{(k)}(n_1+n_2)}{f^{(k+1)}(n_1+n_2)}$ array of subgrids, each of dimension $f^{(k+1)}(n_1 + n_2) \times f^{(k+1)}(n_1 + n_2)$, and conducts corner reductions, line reductions, triangular prefix/suffix reduction, and block reductions on all subgrids. Recursively, it preprocesses all 3×3 subgrids of dimension $3f^{(k+1)}(n_1 + n_2) \times 3f^{(k+1)}(n_1 + n_2)$ with subsubgrids of dimension $f^{(k+2)}(n_1+n_2) \times f^{(k+2)}(n_1+n_2)$, and so on. In order to align the oblique line of input (reduced) IRT query range with one of the 2D arrays, we have to preprocess for $f^{(k+1)}(n_1+n_2)$ differently aligned 2D arrays of subgrids at level k. Since the preprocessing overhead for one such array is $O((f^{(k)}(n_1+n_2))^2/f^{(k+1)}(n_1+n_2))$, the overhead over all differently aligned arrays is then $O((f^{(k)}(n_1 + n_2))^2)$, and sum up to $O(n_1 n_2)$ over the entire grid.

The query algorithm works as follows. Assuming that the input (reduced) IRT range has core length $e_\square \in [f^{(k+1)}(n_1 + n_2), f^{(k)}(n_1 + n_2))$, the key observation is that the range can stride at most one 3×3 array of subgrids of dimension $3f^{(k)}(n_1 + n_2) \times 3f^{(k)}(n_1 + n_2)$. Since we have preprocessed all such 3×3 subgrids, we can answer the query by summing up at most the following results, i.e. one triangular query on the data structure preprocessed by "block reduction", one horizontal and one vertical line queries (1D queries), one corner query, one triangular prefix and one triangular suffix query. The query time is $O(1)$ following a similar argument as in the proof of Theorem 1.

The claimed final preprocessing and query bounds then follow similarly to that of Theorem 1. □

4 Future Work and Open Problems

We have shown an asymmetric upper bound of $\langle O(O(N\alpha(N)), O(\alpha^d(N))\rangle$ for the homothetic triangular range partial sum query problem. The reason behind

asymmetry is that we employ the 1D algorithm [2,9] in our recursive algorithmic scheme. An interesting open problem is whether this bound is tight? A more interesting problem is how to handle general non-orthogonal range partial sum query problem without losing much in complexity bounds, i.e. the angles between adjacent boundaries of input query ranges may change from query to query?

5 Related Work

To the best of our knowledge, all previous research on range partial sum query problem assumed orthogonal ranges. Yao [2] devised the first 1D partial sum algorithm with an $\langle O(N\alpha(N)), O(\alpha(N))\rangle$ bound. Seidel [9] provided an excellent graphical illustration and simplification of the algorithm. Chazelle and Rosenberg [4] later extended the algorithm to arbitrary d-dimensional grid with bounds of $\langle O(N\alpha^d(N)), O(\alpha^d(N))\rangle$ by the technique of "dimension reduction". In their later work [5], Chazelle and Rosenberg proved a lower bound of the 1D offline problem, where the queries are known ahead of time. They further conjectured that their multi-dimensional extension is optimal.

Other works in the literature studied variants of the problem, such as the dynamic version that allows insertions and deletions [10], or special cases such as RMQ (Range Minimum Query), where properties such as idempotence and ordering among elements can be utilized [11–18].

Classic orthogonal range searching problem in computational geometry [19] assumes a contiguous space and sparse points, while in our setting we assume an integer grid where every point holds a valid value.

References

1. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comput. **22**(2), 221–242 (1993)
2. Yao, A.C.: Space-time tradeoff for answering range queries (extended abstract). In: Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing, STOC 1982, pp. 128–136. ACM, New York (1982)
3. Yao, A.C.C.: On the complexity of maintaining partial sums. SIAM J. Comput. **14**(2), 277–288 (1985)
4. Chazelle, B., Rosenberg, B.: Computing partial sums in multidimensional arrays. In: Proceedings of the Fifth Annual Symposium on Computational Geometry, SCG 1989, pp. 131–139. ACM, New York (1989)
5. Chazelle, B., Rosenberg, B.: The complexity of computing partial sums off-line. Int. J. Comput. Geom. Appl. **1**(1), 33–45 (1991)
6. Demaine, E.D., Landau, G.M., Weimann, O.: On Cartesian trees and range minimum queries. In: Albers, S., Marchetti-Spaccamela, A., Matias, Y., Nikoletseas, S., Thomas, W. (eds.) ICALP 2009. LNCS, vol. 5555, pp. 341–353. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02927-1_29
7. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. J. Algorithms **57**, 2005 (2005)

8. Tang, Y., Chowdhury, R.A., Kuszmaul, B.C., Luk, C.K., Leiserson, C.E.: The Pochoir stencil compiler. In: Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures, SPAA 2011, pp. 117–128. ACM, New York (2011)
9. Seidel, R.: Understanding the inverse Ackermann function. In: Invited Talk: 22nd European Workshop on Computational Geometry, Delphi, Greece (2006)
10. Fredman, M.L.: The inherent complexity of dynamic data structures which accommodate range queries. In: FOCS, pp. 191–199 (1980)
11. Amir, A., Fischer, J., Lewenstein, M.: Two-dimensional range minimum queries. In: Ma, B., Zhang, K. (eds.) CPM 2007. LNCS, vol. 4580, pp. 286–294. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73437-6_29
12. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. **40**(2), 465–492 (2011)
13. Fischer, J., Heun, V.: Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In: Lewenstein, M., Valiente, G. (eds.) CPM 2006. LNCS, vol. 4009, pp. 36–48. Springer, Heidelberg (2006). https://doi.org/10.1007/11780441_5
14. Yuan, H., Atallah, M.J.: Data structures for range minimum queries in multidimensional arrays. In: SODA, pp. 150–160 (2010)
15. Poon, C.K.: Optimal range max datacube for fixed dimensions. Int. J. Found. Comput. Sci. **15**(5), 773–790 (2004)
16. Brodal, G.S., Davoodi, P., Lewenstein, M., Raman, R., Srinivasa Rao, S.: Two dimensional range minimum queries and Fibonacci lattices. In: Epstein, L., Ferragina, P. (eds.) ESA 2012. LNCS, vol. 7501, pp. 217–228. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33090-2_20
17. Brodal, G.S., Davoodi, P., Srinivasa Rao, S.: On space efficient two dimensional range minimum data structures. Algorithmica **63**(4), 815–830 (2012)
18. Davoodi, P., Raman, R., Satti, S.R.: Succinct representations of binary trees for range minimum queries. In: Gudmundsson, J., Mestre, J., Viglas, T. (eds.) COCOON 2012. LNCS, vol. 7434, pp. 396–407. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-32241-9_34
19. Chazelle, B., Edelsbrunner, H.: An optimal algorithm for intersecting line segments in the plane. J. ACM **39**(1), 1–54 (1992)