A new framework for subimage analysis

Abdullah N. Arslan¹

© Springer Nature Switzerland AG 2019

Abstract

We develop new methods for several subimage analysis problems. Images in two dimensions are represented by matrixes (higher dimensional images are represented by higher dimensional arrays). This representation yields efficient algorithms for image search and comparison problems applicable not only to images, but also to array search and comparison in any dimension two or larger. Subimage analysis based on our array-representation has limitations. For example, our methods are not rotation and scale invariant. However, they yield efficient algorithms that have practical applications. Videos and snapshots taken from videos are continuously added to the digital world where they are permanently accessible. From a snapshot, one may want to locate and watch the part of the video that contains this snapshot (e.g. moments in news-footage, soccer or olympic games). One may also want to find common parts (sequences of frames) in multiple videos for various reasons (e.g. copyright-check, false-news detection). Our algorithms are applicable in these cases. For another application, we propose representing RNA secondary structures by matrixes. Since our methods are applicable to submatrix analysis, RNA substructure search and multiple RNA structure comparison problems can be solved by using our algorithms (for exact matches).

Keywords Subimage analysis · Array analysis · Structure analysis · Video search · Video comparison · RNA secondary structure · String algorithm · Suffix tree

1 Introduction

Image analysis is a very wide area which uses many techniques and has many applications [30]. In this work, we consider three problems focusing on the following subimage analysis problems: finding given objects in a given image; finding objects that appear multiple times in an image; and finding common objects of a given size in a set of images. In particular, we study variations of these problems in which there are few host images that may include subimages (objects) of interest from a large collection. The following are some examples: in a given image, finding all matching fingerprints (iris recognition is another example) from a database of fingerprints (or from a database of biometric images); or in a given document image, finding all company logo images, characters, watermarks, or other

symbols defined in a database. Finding multiple appearances of the same fingerprint in an image, and comparing multiple images to find common fingerprints are some other applications of the problems we study in this paper.

The mentioned problems are known in the literature as the following: searching subimages [8, 28]; finding repeating subimages [11, 18]; and finding common subimages (multi-image comparison) [17]. Existing algorithms aim for near matches for the purpose of achieving computationally feasible solutions. In this paper, we develop algorithms for obtaining exact solutions. The worst-case time complexity of existing algorithms for these problems (e.g. [3, 4, 7, 16]) is not better than that of brute-force (exhaustive search) algorithms. The solutions we propose process host images first to create data structures. Subsequent problems are answered fast with the help

✉ Abdullah N. Arslan, Abdullah.Arsalan@tamuc.edu | ¹Department of Computer Science and Information Systems, Texas A&M University - Commerce, Commerce, TX 75428, USA.

of these data structures. One important example is the human brain template images which are stored for the purpose of supporting search and comparison (see [27] for a survey). Center for Geographic Analysis (<https://gis.harvard.edu>) at Harvard University developed and made available open source data for world's map and geological structures (<http://worldmap.harvard.edu/>). Image analysis on existing and new structures is a perpetual work. For another example, one may consider processing images of a crime-scene object that may contain multiple fingerprints. The fingerprint images from a database can then be run on this preprocessed image to find all matches. Similar applications are natural on processing documents that contain logo images, predefined diagram elements, human language and musical characters, or other predefined visual symbols from a very rich set of sources such as hieroglyphic alphabets.

Our interests in this paper are on images of objects that can be represented by (and stored in) multi-dimensional arrays of finite elements (pixels). For a d -dimensional (constant $d \geq 2$) such image, we use the following notation:

- the *size of the image* is described by expression $N_1 \times N_2 \times \dots \times N_d$;
- entire set of elements in the i 'th *dimension* of the image is referred to by *dimension i* , where integer $i \in [1, d]$;
- the *length (size) of dimension i* is denoted by N_i ;
- *total size of the image* is the product $N_1 N_2 \dots N_d$.

We assume without loss of generality that $N_1 = \max\{N_1, N_2, \dots, N_d\}$ (otherwise, we transpose the array storing the image: this is only needed to minimize execution time). In particular, we consider rectangular images as arrays (also called matrixes for $d = 2$) whose elements (pixels) are defined over a fixed finite alphabet Σ . Figure 1 illustrates an example for our modeling of a rectangular image by a matrix using a binary alphabet Σ .

There are limitations due to our modeling of images. Our image representation is not rotation and scale invariant. Precision of representation depends on sizes of matrixes (grid cells), and the choice of alphabet Σ . Images

are also affected by many physical parameters such as noise, blur, and intensity. However, our model is still very useful in analyzing solid objects, 3-dimensional layout of buildings, fragments of video footage. We elaborate on them in this section.

The methods we develop apply to arrays. Therefore, to refer to an image, we prefer to use the word *array*. In our representation, we further convert arrays to strings. Subimage analysis problems eventually are solved in the framework of strings.

Subimage Query A *subimage query* asks for all occurrences of a given subimage in a host image. Consider digital documents that include symbols generated from known sources in fixed format and size. There are many sets of such symbols. Figure 2 shows such example sets. Subimage searching has challenges for traditional computational methods: First, searching for a subimage using global features fails because the subimage features usually represent only a small fraction of the global image features [9]. Second, even if local regions of interest are detected, finding actual matches from the underlying database of subimages requires many image comparisons. Best methods organize the underlying database by clustering images based on (local) features (e.g. feature selection and clustering of fingerprints [5, 24]). We address these two challenges by moving this problem to the domain of strings in which we can use an efficient data structure, namely *generalized suffix tree*.

Applying tools on strings to image matching is not new. A commonly used alignment tool for biological sequences has been used in matching images [20]. In another application, sequences generated from object boundaries are used in recognizing weapons from given images [2]. In this paper, we propose a novel string representation for images which lets us create and use a suffix tree. After this representation is created for the host image, the time required by our subimage search algorithm does only depend on the size of the searched subimage, and not on the size of the host

Fig. 1 Our image representation: **a** a rectangular image; **b** a 2-dimensional grid of equal-sized binary (1: fill; 0: no fill) cells partitioning this image; **c** a matrix obtained from the cells of this grid. The origin is at the top-left corner, position (0, 0)

Fig. 2 Several symbols from different sets: **a** Emoji symbols; **b** Chinese symbols; **c** ancient Egyptian Hieroglyphics symbols

image. To the best of our knowledge, currently, no other algorithm guarantees comparably fast performance for exact subimage search.

In our model, a *subimage query* for a given subimage g in an image G becomes the problem of finding a given subarray g in array G . Therefore, we use *subarray query* for *subimage query*. Subarray search-related problems are also known in the literature (e.g. see [19, 25]).

We propose a novel method for subarray query. We process a given array G of size $N_1 \times N_2$ and build a suffix tree R_G . Subsequently, for any given array g of size $n_1 \times n_2$, our method uses R_G to find all occurrences (if any) of g (as a subarray) in G . Building R_G is done in time $O(N_1 N_2^3)$ using $O(N_1 N_2^3)$ space. The subarray query becomes the problem of finding substrings in a given string using the suffix tree R_G . For any given array g , finding all $c \geq 0$ occurrences of g in G (*subarray query*) takes time $O(n_1 n_2 + c)$, independent of the total size $|G|$ of G . Our approach is very fast when there are many subarray queries to answer on the same array G . This approach can also be generalized to d -dimensional images, where $d \geq 2$. For d -dimensional images, we use d -dimensional arrays. Figure 1 includes an example for a two-dimensional case. Figure 3 illustrates that a three-dimensional model for a three-dimensional image includes more information. The occluded additional two table-legs in part (a) are included in the model in part (b). Searching for a single leg in 3-dimensional model in Fig. 1 part (b) returns four positions shown in part (c).

For another example instance of subimage query, consider searching for a subimage in a video. The queried subimage could be a fragment originally obtained from the input video, but which frame has it in the video may not be known. Such cases often arise in real life. There are many pictures extracted from video frames in social media. Usually people would like to locate the frame of a video that is known to contain a given query subimage. A snapshot of a goal in a soccer game, or a snapshot of a winning swimmer may appear in many videos.

Fig. 3 **a** A table's view in 2-dimension in certain perspective; **b** the same table in a 3-dimensional array; **c** positions of four identical legs of the table

Figure 4 includes an example snapshot used as a query. In this case, since a video is a sequence of 2-dimensional frames, it is stored in a 3-dimensional array.

The time and space required to build suffix tree R_G^d for array G of size $N_1 \times N_2 \times \dots \times N_d$ is $O(N_1 N_2^3 \dots N_d^3)$. A subarray query for any given array g of size $n_1 \times n_2 \times \dots \times n_d$ can be answered in time $O(n_1 n_2 \dots n_d + c)$, where c is the number of occurrences of g in G . In comparison, one may relate 2-dimensional exact image matching to the 2-dimensional exact string matching problem [4, 7]. Although algorithms for good average-case performance have been proposed (e.g. [16]), the worst-case time complexity of existing algorithms for this problem has remained $O(N_1^1 N_2^1 N_1^2 N_2^2)$ for matching two images of sizes $N_1^1 \times N_2^1$, and $N_1^2 \times N_2^2$. This worst-case complexity can be achieved by a naive algorithm that uses brute force. Generalizing this to any dimension two or larger, a naive (brute-force) algorithm for d -dimensional exact matching would take in the worst-case $\Theta(N_1 N_2 \dots N_d n_1 n_2 \dots n_d)$ time since there are $N_1 N_2 \dots N_d$ possible subarray positions and checking each one for an exact match takes time $\Omega(n_1 n_2 \dots n_d)$.

Repeating Subarrays Query We consider the problem of finding, for any given integer $z > 0$, all subimages g each with total size $|g| \geq z$ appearing more than once in host image G . We call this problem the *Repeating Subarrays Query*. In a digital document that contains symbols (such as shown in Fig. 2) generated by a known source, a symbol may appear many times. In Fig. 3, the same leg appears four times in the 3-dimensional model of the table (two legs appear in the 2-dimensional model of the same table). These cases are some examples for the repeating subarray query problem. We present a solution that answers this problem in time $O(N_2^2)$ (independent of z), where G is an image of size $N_1 \times N_2$. We note that a naive (brute-force) algorithm that checks all pairs of positions in G for a possible match would take $\Omega(z N_1^2 N_2^2)$ time.

All Common Subarrays Query We also consider a multiple image comparison problem whose objective is to find the subimages common to all given input images. Digital world is flooded with videos. The same video clips are used by many agents (news agents are showing the same video by adding their logos on a corner). Comparison of videos can be used to make sure that shared number of frames is within the allowed sizes by copyright rules. They do not falsely present (some fake news were known to have been generated by taking videos from computer games). Figure 5 includes an example for a shared video-segment. In this case, two videos from NASA share frames in illustrating the landing animation of Curiosity on Mars. There are eas-

Fig. 4 **a** Query subimage; **b** a youtube video titled “Curiosity Has Landed” (<https://www.youtube.com/watch?v=N9hXqzkH7YA>). Queried subimage is obtained from a fragment in a frame in this video.

Such instances often arise in practice. The queried subimage is found in a frame pointed by an arrow from the query in part (a)

Fig. 5 Two videos from NASA illustrate animation of landing of Curiosity on Mars. Video 1: “Curiosity Has Landed” (<https://www.youtube.com/watch?v=N9hXqzkH7YA>); Video 2: “NASA Mars Science Laboratory (Curiosity Rover) Mission Animation [HD x 1280]” (https://www.youtube.com/watch?v=gwinFP8_qIM). Frames 1–4 are shared in both videos

In a set of images $\{G_1, G_2, \dots, G_K\}$, where each G_i in this set is an image of size $N_1 \times N_2$, all subimages g each with total size $|g| \geq z$ (for a given integer $z > 0$) common to all these K images (*All Common Subarrays Query*) can be found in time $O(KN^2)$ using the generalized suffix tree created from $G_1, G_2, \dots, G_K$. A naive (brute-force) algorithm for this problem considers and compares all possible positions pairwise from G_1 and all $G_i, i \in [2, d]$ (based on transitivity of equality for subarrays), and it would take time $\Omega(zKN^2N^2)$.

ily more than 10 videos in social media sharing these fragments.

When all videos of two people are compared, this requires a comparison in 4 dimensions since the 4th dimension is required to store all 3-dimensional videos of an individual. Similarly, when the hierarchy grows in size (e.g. from countries to groups, and then to individuals), the needed dimensions grow in the same way.

In the literature, there are algorithms on strings that use suffix trees for solving problems that have some similarity to the problems we tackle. These other problems are about identifying all repeats (repeated substrings in strings) [15, 26], and finding longest common repeats [23] with applications in biological sequence analysis. The algorithms for these problems are not readily applicable to the problems we define in this paper. This is because in our framework, strings are not ordinary sequences of characters; they include start and end markers with which substrings of certain structures correspond to subarrays. There are also differences in objectives of the defined problems. Our framework uses strings for modeling arrays in two or higher dimensions. Additionally, our results for

two-dimensional cases also generalize to d -dimensional cases where $d > 2$ not only for *Subarray Query*, but also for *Repeating Subarrays Query* and *All Common Subarrays Query*. These generalizations are significant because image analysis in high dimensions has important application areas. For example, medical imaging applications use objects in 3 and 4 dimensions (e.g. brain template image). Example surveys on such applications can be found in the literature [22, 27].

We introduce our problems in the domain of images. However, since our algorithms are developed for objects modeled on arrays, they have much wider impact. For example, they are applicable for the analysis of lattice based models of crystal structures (see [21] for lattice models of crystals), and RNA secondary structures (see [1] for some RNA secondary structure tools).

The outline of this paper is the following: We give basic definitions and describe our notation in Sect. 2. In Sect. 3, we propose a method that processes an array to create an efficient data structure representation. We present our algorithm for *Subarray Query* in Sect. 4. In Sect. 5, we generalize our definitions and results for *Subarray Query* for d -dimensional ($d \geq 2$) images. We discuss *Repeating Subarrays Query* in Sect. 6, and *All Common Subarrays Query* in Sect. 7. We discuss applications of our algorithms on RNA secondary structure analysis, and give remarks on additional applications in Sect. 8. We summarize limitations and contributions of our methods in Sect. 9. We conclude and give pointers for future work in Sect. 10.

2 Basic definitions and notation

Let Σ be a fixed finite alphabet. We consider rectangular images as arrays whose elements (pixels) are defined over Σ . That is, $|\Sigma|$ is constant, and each element in Σ has a constant-size representation. Each pixel in an image has a position, and attributes (e.g. color) determined from the assigned symbol in Σ . For example, a black and white rectangular image can be represented by an array whose elements are in $\{0, 1\}$. We add to Σ the symbol $\#$, not originally in Σ for a special purpose.

A *suffix tree* R for a string s is a tree in which every suffix of s appears on a branch in R . In a generalized suffix tree, all suffixes of a given set of strings appear as labels on the branches [14]. There are many applications of generalized suffix trees (for example, see [6]).

Figure 6 includes a generalized suffix tree for three strings, $S_1 = \text{paint}$, $S_2 = \text{rain}$, and $S_3 = \text{train}$. The strings are appended by a $\$$ to mark suffixes' end. Every leaf node u corresponds to a suffix that is read on the branch from the root until this leaf u . Every leaf node contains a list of pairs (n, i) , where n is the identifier for the host string S_n that has

Fig. 6 A generalized suffix tree for strings $S_1 = \text{paint}$, $S_2 = \text{rain}$, and $S_3 = \text{train}$

the corresponding suffix to this branch, and i is the starting index of this suffix in string S_n . For example, the suffix $\text{in\$}$ appears on a branch which ends in a leaf node that includes $(S_2, 3)$, and $(S_3, 4)$. We note that $\text{in\$}$ starts at position 3 in $S_2 = \text{rain\$}$, and at position 4 in $S_3 = \text{train\$}$. In the actual implementation, the labels of the edges are not the substrings but beginning and ending positions of the substrings in a host string. This saves a lot of space since each label is actually a pair of integers. There are algorithms that construct a generalized suffix tree in time and space linear in the total length of the input strings (e.g. [31]).

Let R be a generalized suffix tree of a set of strings in S . Given a string s , determining if s is a substring of any string in S can be done in time $O(|s|)$ by using R [14]. More precisely, substring s is searched in prefixes of all suffixes of all strings in S appearing on (more precisely, achievable via) branches of R . The fact that this takes time $O(|s|)$, and the number of nodes, edges, and the total size of the information stored in labels of the constructed tree R are linear in the input strings' total length [14, 31] are fundamental for efficiency of our algorithms in this paper.

In the rest of the paper, d denotes a constant integer two or larger. We use d to refer to images' number of dimensions.

3 Proposed preprocessing for images

We imagine a rectangular image G of size $N_1 \times N_2$ as an array of pixels $G = (t_{ij})$ of size $N_1 \times N_2$ as shown in Fig. 7a. We assume without loss of generality that $N_1 = \max\{N_1, N_2\}$ (otherwise we transpose G).

We represent array G as a string S_G , where each t_{ij} is considered as an element in a fixed finite alphabet Σ , and S_G is obtained from the rows of G such that $S_G = \#_1 \prod_{i=1}^{N_1} t_{i,1} t_{i,2} \dots t_{i,N_2} \#_1$, where $\prod$ denotes the concatenation operation. For the example in Fig. 7b, $S_G = \#_1 001100\#_1 010010\#_1 111111\#_1 100001\#_1 100001\#_1$ (0, 1, respectively, denotes the white, black pixels).

Let $b_{j,k}$ denote a slab (a rectangular subarray) of array G such that it has the top left corner $(1, j)$, bottom right

Fig. 7 **a** An array model of a rectangular image G ; and **b** an example image: arrows indicate the order of elements traversed in generating the string representation

Fig. 8 Slab $b_{j,k}$ (enclosed by dashed lines) in image G , where $j, k, j+k-1 \in [1, N_2]$

corner $(N_1, j+k-1)$, and for all j, k , it is true that j, k , and $j+k-1$ are in $[1, N_2]$. In other words, slab $b_{j,k}$ is a subarray of G of size $N_1 \times k$ with its top left corner at $(1, j)$. Slab $b_{j,k}$ of G is illustrated in Fig. 8.

Let $S_{b_{j,k}}$ denote a string obtained from slab $b_{j,k}$ of G . That is, $S_{b_{j,k}} = \#_1 \prod_{i=1}^{N_1} t_{i,j} t_{i,j+1} \dots t_{i,j+k-1} \#_1$. We note that all strings for images start with a $\#_1$ including S_G and all $S_{b_{j,k}}$. Let B_G be the following collection of strings $S_{b_{j,k}}$:

$$B_G = \bigcup_{\substack{\text{for all slabs } b_{j,k} \\ j, k, j+k-1 \in [1, N_2]}} S_{b_{j,k}}.$$

Two arrays g and g' are *identical* if they are of the same dimension and size, and all elements in the same positions are the same in both arrays.

A given array g of size $n_1 \times n_2$ is a *subarray* of array G at position (i', j') if the following two subarrays are identical: g and the subarray of G with top-left and bottom-right corners at positions (i', j') and $(i' + n_1 - 1, j' + n_2 - 1)$, respectively.

Definition 1 For given two arrays g and G , if there exists (i', j') such that the subarray of G at position (i', j') is identical to g , then we say that array g *appears* at position (i', j') in array G .

Lemma 1 For arrays G of size $N_1 \times N_2$, g of size $n_1 \times n_2$, and for $j' \in [1, N_2]$, $i' \in [1, N_1]$, g appears at (i', j') in G iff S_g is the prefix of length $|S_g|$ of the suffix of $S_{b_{j',k'}}$ $\in B_G$ that starts at index (position) $(i' - 1)(n_2 + 1) + 2$ in this suffix.

Proof If g is a subarray at position (i', j') in G , then for $k' = n_2$, g is a subarray of the slab $b_{j',k'}$ of G starting at row i' . Figure 9 illustrates this case on array G in part (a); and in suffix tree R_G in part (b). By the definitions of S_g and $S_{b_{j',k'}}$, S_g is the prefix of length $|S_g|$ of the suffix U of $S_{b_{j',k'}}$ $\in B_G$ that starts at position $i = (i' - 1)(n_2 + 1) + 2$ (including $\#_1$'s) in this suffix. The suffix tree R_G has a branch ending at a leaf node that contains in its list the tuple (j', k', i) .

If g is different from the subarray of size $n_1 \times n_2$ at position (i', j') in G , clearly g is not a subarray of the slab $b_{j',k'}$ of G starting at row i' , where $k' = n_2$. By the definitions of S_g and $S_{b_{j',k'}}$, S_g differs from the prefix that starts at index (position) $(i' - 1)(n_2 + 1) + 2$ in the suffix of $S_{b_{j',k'}}$ $\in B_G$. The suffix tree R_G cannot have a branch ending at a leaf node containing in its list the tuple (j', k', i) . $\square$

Proposition 1 Build a generalized suffix tree R_G that stores B_G containing all $S_{b_{j,k}}$ for all $j, k, j+k-1 \in [1, N_2]$. This takes time and space $O(N_1 N_2^3)$. There are N_2^2 slabs $b_{j,k}$ each of total size $O(N_1 N_2)$. The total length of strings is $O(N_1 N_2^3)$.

We can use a linear time and space suffix tree construction algorithm [31] in building R_G . Since the total length of the input strings is $O(N_1 N_2^3)$, this takes $O(N_1 N_2^3)$ time and space. There are $O(N_2^2)$ slabs (one string for each slab). Therefore, R_G has $O(N_2^2)$ nodes and edges.

After building R_G , we postprocess it. There is only one edge whose label starts with $\#_1$ from the root. We keep this edge and the subtree rooted at the child arrived from this edge. All subtrees rooted at other children of the root are removed. Post-processing R_G does not increase the number of nodes and edges. We use post-processed R_G .

We say that string h is a *complete prefix* if it is a prefix of string $S_{b_{j,k'}}$ for some slab $b_{j,k'}$, and h starts and ends with a $\#_1$. Throughout the entire paper, for simplicity we ignore the end marker $\$$ in all suffixes in our discussions. Let H be the set of all complete prefixes obtained from B_G . There is a one-to-one correspondence between H and set of all possible rectangular images in G (i.e. between distinct elements).

Let T be the set of complete prefixes obtained from the labels of R_G . We note that on each branch in R_G the

Fig. 9 **a** Subarray g in slab $b_{j',k'}$ of G ; **b** The subarray g is represented by a string S_g that appears as a prefix of suffix U of string $S_{b_{j',k'}}$ for slab $b_{j',k'}$. U starts in $S_{b_{j',k'}}$ at a position calculated by an expression

first label starts with a $\#_1$, and ends with a $\#_1$. There are one-to-one correspondences for all pairs of the sets H , T , and set of all possible rectangular (distinct) subimages in G .

In the leafs of R_G , each (j, k, i) is stored for the suffix starting at index i in $S_{b_{j,k}}$.

4 Subarray search

We define the problem of searching for a given subimage in a host image as a query problem on arrays.

Definition 2 *Subarray Query (SAQ)*: Given an array g of size $n_1 \times n_2$, find all positions (i', j') at which g appears in a host array G of size $N_1 \times N_2$.

Theorem 1 *Let G be an array processed as described in Proposition 1. For a given subarray g of size $n_1 \times n_2$, all c distinct positions (i, j) at which g appears (all c occurrences of g) in G can be found in time and space $O(n_1 n_2 + c)$. That is, the SAQ problem can be solved within this specified time and space complexity.*

Proof By corollary of Lemma 1, we see that the subarray query SAQ can be answered by solving a substring search problem as the following: For a given g , construct S_g ; search S_g in R_G ; if no such S_g is found then return that g does not appear in G . Otherwise, return all positions (j, k, i) , where S_g is a prefix of the suffix starting at position i in string $S_{b_{j,k}}$ for slab $b_{j,k}$.

that involves the row number i' . U appears on a branch in B_G . This branch ends with a leaf that has a list containing the triplet (j', k', i)

Constructing S_g takes time $O(|g|) = O(n_1 n_2)$. If g appears in G , the traversal for searching S_g in R_G arrives at a node u after visiting at most $O(|g|)$ nodes. The traversal reaches at most c leafs in the subtree rooted at node u . In the lists of these leaf nodes collectively c triplets (one for each position) of (j, k, i) are obtained. Each found distinct (j, k, i) corresponds to a distinct appearance of g in some slab $b_{j,k}$ starting at row i' in G (including also $\#_1$'s in S_g) such that $i = (i' - 1)(k + 1) + 2 = (i' - 1)(n_2 + 1) + 2$ (or $i' = (i - 2)/(k + 1) + 1$). If g does not appear in G , the search will discover this after examining $O(|g|)$ nodes during the traversal. Therefore, the total time required for finding all $c \geq 0$ occurrences of g in G is $O(n_1 n_2 + c)$. $\square$

5 Searching for d -dimensional ($d \geq 2$) objects

The result expressed in Theorem 1 generalizes to d -dimensional ($d \geq 2$) images. For this purpose, we first extend our definitions. For a given d , let $\#_1, \#_2, \dots, \#_{d-1}$ be symbols added to Σ . We imagine a d -dimensional ($d \geq 2$) image G of size $N_1 \times N_2 \times \dots \times N_d$ as an array of elements $G = (t_{j_1 j_2 \dots j_d})$. We assume without loss of generality that $N_1 = \max\{N_1, N_2, \dots, N_d\}$ (otherwise we transpose G).

Next, let slab $D' = b_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}$ denote a d -dimensional ($d \geq 2$) subarray of G such that D' is of size $N_1 \times k_2 \dots k_3 \dots k_d$, and the lexicographically smallest corner (the top-left corner if $d = 2$) of slab D' in G is $(1, j_2, j_3, \dots, j_d)$.

Consider that we traverse the elements of D' in lexicographical order of dimension numbers $j_2, j_3, \dots, j_d$. The index values and elements on these indexes appear in lexicographical order of indexes. In other words, these elements are sorted in dimension 1, and then in dimension 2 within the same dimension-1 value, and so on until dimension d (i.e. in lexicographical order). Let $S_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}$ be the string obtained in this way (i.e. let the traversal in lexicographical order yields the string $S_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}$). In $S_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}$ the first and last elements are $\#_i$; and immediately after a sequence of elements at dimension i , the symbol $\#_i$ appears before a sequence of elements at dimension $i + 1$ starts. Let B_G^d be the collection of strings obtained from all slabs D' of G . That is,

$$B_G^d = \bigcup_{\substack{\text{for all slabs } b_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)} \text{ such that} \\ \text{for all } d' \in [2, d], j_{d'}, k_{d'}, j_d' + k_{d'} - 1 \in [1, N_{d'}]}} S_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)} \quad (1)$$

For example, consider a 3-dimensional image G described by array $T[i, j, k] = 0$ for all $i \in [1, 4], j \in [1, 3], z \in [1, 2]$, except that $A[4, 3, 2] = 1$ (a 3-dimensional array of size $4 \times 3 \times 2$ composed entirely of 0's except for one corner cell $A[4, 3, 2] = 1$). This example is illustrated in Fig. 10.

In this case, $S_G = \#_1 00\#_2 00\#_2 00\#_2 \#_1 00\#_2 00\#_2 00\#_2 \#_1 00\#_2 00\#_2 01\#_2 \#_1$ using the lexicographical order of indexes $(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (1, 3, 1), (1, 3, 2), \dots, (4, 1, 1), (4, 1, 2), (4, 2, 1), (4, 2, 2), (4, 3, 1), (4, 3, 2)$. For slab $b_{(2,2), (2,1)}$, the corresponding string is $S_{(2,2), (2,1)} = \#_1 0\#_2 0\#_2 \#_1 0\#_2 0\#_2 \#_1 0\#_2 0\#_2 \#_1 0\#_2 1\#_2 \#_1$ based on the lexicographical order of corresponding indexes $(1, 2, 2), (1, 3, 2), (2, 2, 2), (2, 3, 2), (3, 2, 2), (3, 3, 2), (4, 2, 2), (4, 3, 2)$ in this slab.

Proposition 2 Build a generalized suffix tree R_G^d that stores B_G^d defined in Eq. 1. This takes time and space $O(N_1 N_2^3 \dots N_d^3)$, where $N_1 = \max\{N_1, N_2, \dots, N_d\}$. In G , there are $N_2^2 \dots N_d^2$ slabs $b_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}$ each of total size $O(N_1 N_3 \dots N_d)$. The total length of strings is $O(N_1 N_2^3 \dots N_d^3)$.

Suffix tree R_G^d stores suffixes of $N_2^2 \dots N_d^2$ strings (one string for each slab). Therefore, it has $O(N_2^2 \dots N_d^2)$ nodes and edges.

We post-process R_G^d with root node r in a similar way described in Proposition 1, and obtain essential properties for our results. More specifically, we keep only the subtree rooted at r 's child to which the label starts with $\#_1$. After this post-processing, there are one-to-one pairwise correspondences among the sets of all complete prefixes obtained from the branches of R_G^d , complete prefixes obtained from set of all slabs, and set of all d -dimensional subimages in G (the correspondences are among distinct elements because these sets are not multi-sets).

In the leaves of R_G^d each element $(j_1, (j_2, k_2), (j_3, k_3) \dots (j_d, k_d))$ is stored for the suffix starting in position j_1 in string $S_{b_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}}$.

A given array g of size $n_1 \times n_2 \times \dots \times n_d$ is a subarray of d -dimensional array G at position $(j'_1, j'_2, \dots, j'_d)$ if g is identical to the subarray of G which has a corner at position $(p_1^\pi, p_2^\pi, \dots, p_d^\pi)$ for all $\pi \in 2^n$ such that $\pi_1 \pi_2 \dots \pi_k \dots \pi_d$ is the binary string representation of π , where for all $k \in [1, d]$, if $\pi_k = 0$ then $p_k^\pi = j'_k$; else (if $\pi_k = 1$) then $p_k^\pi = j'_k + n_k - 1$.

Definition 3 Given two d -dimensional ($d \geq 2$) arrays g and G , if there exists a position $(j'_1, j'_2, \dots, j'_d)$ in G such that the subarray of G at position $(j'_1, j'_2, \dots, j'_d)$ is identical to g , we say that array g appears at position $(j'_1, j'_2, \dots, j'_d)$ in array G .

Lemma 2 For d -dimensional arrays G of size $N_1 \times N_2 \times \dots \times N_d$, g of size $n_1 \times n_2 \times \dots \times n_d$, and for integers $j'_d \in [1, N_d]$, g is an array that appears at $(j'_1, j'_2, \dots, j'_d)$ in G iff S_g is the prefix of length $|S_g|$ that starts at index (position) $(j'_1 - 1)(n_2 + 1)(n_3 + 1) \dots (n_d + 1) + 2$ in string $S_{(j'_2, k_2), (j'_3, k_3), \dots, (j'_d, k_d)} \in B_G^d$.

Fig. 10 A 3-dimensional image used for simple illustration of definitions. The top-left corner is at position $(0, 0, 0)$. The filled cell at position $(4, 3, 2)$ is the only cell that has color code 1. All other cells have color code 0

Proof The proof of Lemma 2 is a generalization of that of Lemma 1. In this case, if g is a subarray at position $(j'_1, j'_2, \dots, j'_d)$ in G , then g is a subarray of the slab $b_{(j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d)}$ of G starting at row j'_1 (in dimension 1). By the definitions of S_g and $S_{b_{(j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d)}}$, S_g is the prefix of length $|S_g|$ of the suffix U of $S_{b_{(j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d)}} \in B_G^d$ that starts at position $(j'_1 - 1)(n_2 + 1)(n_3 + 1) \dots (n_d + 1) + 2$ (including $\#_1$ s) in this suffix. The suffix tree R_G^d has a branch ending at a leaf node that contains in its list the tuple $(j'_1, (j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d))$.

If g is different from the subarray of size $n_1 \times n_2 \times \dots \times n_d$ at position $(j'_1, j'_2, \dots, j'_d)$ in G , clearly g is not a subarray of the slab $b_{(j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d)}$ of G starting at row j'_1 (in dimension 1). By the definitions of $S_{b_{(j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d)}}$ and S_g , S_g differs from the prefix that starts at index (position) $(j'_1 - 1)(n_2 + 1)(n_3 + 1) \dots (n_d + 1) + 2$ in the suffix of $S_{b_{(j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d)}} \in B_G^d$. The suffix tree R_G^d cannot have a branch ending at a leaf node containing in its list the tuple $(j'_1, (j'_2, n_2), (j'_3, n_3), \dots, (j'_d, n_d))$. $\square$

Definition 4 *Subarray Query (SA^dQ)*: Given an array g of size $n_1 \times n_2 \times \dots \times n_d$, find all positions $(j'_1, j'_2, \dots, j'_d)$ for integers $j'_d \in [1, N_d]$ such that g appears at position $(j'_1, j'_2, \dots, j'_d)$ in a host array G of size $N_1 \times N_2 \times \dots \times N_d$.

Theorem 2 Let G be a d -dimensional array ($d \geq 2$) processed as described in Proposition 2. For a given subarray g of size $n_1 \times n_2 \times \dots \times n_d$, all c distinct positions $(j'_1, j'_2, \dots, j'_d)$ at which g appears (all c occurrences of g) in G can be found in time and space $O(n_1 n_2 \dots n_d + c)$.

Proof By corollary of Lemma 2, we see that SA^dQ can be reduced to a substring search problem. The proof of this statement is a generalization of Theorem 1. Constructing S_g takes time $O(|g|) = O(n_1 n_2 \dots n_d)$. There is only one node u arrived from the root by following S_g in R_G^d . On this path until u there are at most $O(|g|)$ nodes, and the subtree rooted at u has at most c leaves. These leaves include c different elements of (tuple) $(j_1, (j_2, k_2), (j_3, k_3), \dots, (j_d, k_d))$. From each found distinct tuple $(j_1, (j_2, k_2), (j_3, k_3), \dots, (j_d, k_d))$, there is a corresponding slab with lexicographically smallest corner $(j'_1, j'_2, j'_3, \dots, j'_d)$ in which j'_1 is obtained from j_1 . If g is not a subarray of G then after examining at most $O(|g|)$ nodes, the search will conclude that searching for S_g fails, and therefore g does not appear in G . Therefore, the time required for searching and finding all $c \geq 0$ occurrences of g is $O(n_1 n_2 \dots n_d + c)$. $\square$

A naive (brute-force) algorithm for the SA^dQ problem ($d \geq 2$) would check every position in G for a possible

match, and it would take $\Theta(N_1 N_2 \dots N_d n_1 n_2 \dots n_d)$ time. Our algorithm is significantly faster.

In the problems in the rest of the paper we do not consider two-dimensional images separately; we consider d -dimensional images for any constant $d \geq 2$ with the initial problem definition. Without loss of generality, in image's d -dimensional array representation, we assume that $N_1 = \max\{N_1, N_2, \dots, N_d\}$ (otherwise the array can be transposed to satisfy this assumption).

6 Finding repeating subarrays

Definition 5 *Repeating Subarrays Query (RA^dQ)* ($d \geq 2$): Given an integer $z > 0$, find all subarrays g of total size $|g| \geq z$ appearing more than once in a d -dimensional (pre-processed) host array G of size $N_1 \times N_2 \times \dots \times N_d$.

Theorem 3 Let G be a d -dimensional array ($d \geq 2$). There exists a suffix tree representation for G such that after building it, every instance of problem RA^dQ can be solved in time $O(N_2^2 \dots N_d^2)$.

Proof Let G be a d -dimensional array ($d \geq 2$) processed first as described in Proposition 2 yielding the suffix tree R_G^d after post-processing. We further process R_G^d only once for the purpose of developing an algorithm that solves all subsequent RA^dQ problems. That is, with the help of the resulting tree, the RA^dQ problem is solved for any given value of z in the time complexity described in Theorem 3.

We introduce additional definitions for the needed steps in further processing of the suffix tree R_G^d . For every edge (u, v) in the suffix tree R_G^d , let $s_{u,v}$ be the substring that corresponds to the segment of the suffix on the branch containing (u, v) . For example, in Fig. 6, let r be the root, w be the rightmost child of r , and c be the rightmost child of w , then $s_{r,w} = t$, and $s_{w,c} = \text{rain}\$$.

In suffix tree R_G^d , consider the labels on a branch from the root to a leaf. For such a branch, the corresponding string from the first $\#_1$ to the last $\#_1$ indicates a d -dimensional subarray (equivalently, a complete prefix). Since every suffix in B_G^d starts and ends with a $\#_1$, the first and last symbols on any branch is a $\#_1$ (recall that suffix end-marker $\$$ is ignored). For efficient search processing in a later step, at every node v we count the symbols that have been seen in a complete prefix (d -dimensional subarray) on branch labels at arriving v . We also count the symbols growing toward a complete prefix later. These values help with a traversal step (performed later for solving the RA^dQ problem) in identifying nodes at which the length bound is achieved for a given z . The problem then becomes outputting all suffixes obtained from the leaves in the subtrees rooted at such identified nodes.

We explain how we modify the suffix tree R_G^d in details. We have the following definitions: Let $\#_j$ symbol denote any $\#_j$ for some $j \in [1, d-1]$. On the path arriving at node v from the root,

- e_v is the total number of non $\#$ symbols; and
- f_v is the total number of non $\#$ symbols until the last seen $\#_1$ on this path before v .

We note that at node v , f_v is the size of the largest complete d -dimensional subarray observed; and e_v is the size of a growing d -dimensional subarray not necessarily complete yet, but will be completed later (at a leaf node at the latest). For the root node, $f_{root} = e_{root} = 0$. On traversing an edge (u, v) ,

- if $s_{u,v}$ does not include $\#_1$
 - set $f_v = f_u$; and set $e_v = e_u + |s_{u,v}|$;
- else
 - if the last symbol in $s_{u,v}$ is a $\#_1$
 - set $e_v = f_v = e_u + \text{number of non } \# \text{ symbols in } s_{u,v}$;
 - else let $s_{u,v} = p\ell q$, where ℓ is the last $\#_1$ in $s_{u,v}$
 - set $f_v = e_u + \text{number of non } \# \text{ symbols in } p$; and
 - set $e_v = f_v + \text{number of non } \# \text{ symbols in } q$;

All these calculations for added attributes can be done by using traversal on R_G^d . Let M_G^d be the resulting suffix tree modified from R_G^d as described.

We do an additional tree traversal in order to add information to the suffix tree about all *common subarray sizes* shared by two or more subarrays. We recursively traverse M_G^d , starting at the root node *root*. A *common subarray size* $m_u > 0$ is achievable by a branch passing through node u

if the following is true: m_u is the maximum value of f such that the subtree rooted at u has (at least) two leafs x and y such that $f_x = f_y = m_u$. That is, m_u is the maximum value of f shared by any two leafs in the subtree rooted at u . Calculation of m for all nodes can be done by recursively traversing M_G^d in depth-first (or iteratively in a bottom-up) manner based on all f values which were calculated in the previous traversal.

Consider the example illustrated in Fig. 11. The suffix tree R_G^d in part (a) includes a set of branches ending in a number of leafs in subtree B rooted at node v'' that yield a common subarray. Our modification algorithm calculates e , f and m values and adds them to R_G^d and generates the suffix tree M_G^d shown in part (b). In this case, the total lengths of all labels (excluding $\#_1$ symbols) are $f_{v''}$ until node v'' , and maximum $m_{v''}$ until leaf nodes (to at least two leaf nodes). For any given positive value of z , the traversal we propose visits node v'' and discovers that the size (length) lower-bound is satisfied at this node if $f_{v''} \geq z$. Further, if $m_{v''} \geq z$ then it traverses the subtree rooted at node v'' and outputs all subarrays obtained at the leafs of this subtree B since there are at least two leafs in B meaning that the common sufficiently large subarray until v'' is shared by multiple arrays (i.e. all arrays corresponding to the leafs in B). We note that M_G^d is built only once from G to answer all subsequent RA^dQ problems on G for any $z > 0$.

Problem RA^dQ asks for repeating substrings s of length $\geq z > 0$ (excluding $\#$ symbols) such that s is a complete prefix of a suffix of $S_{(j_2, k_2), (j_3, k_3), \dots, (j_d, k_d)}$. Each such substring (if it exists) corresponds to a subarray of size $\geq z$ that appears at two or more distinct positions in G . The problem can be answered by a single traversal partitioned into two levels on M_G^d in the following way: During the *outer-level* part of the traversal, on visiting edge (u, v) , if $m_v < z$, simply skip the subtree M_v rooted at v , and jump to the successor sibling of node v as subtree M_v yields no solutions

Fig. 11 **a** Suffix tree R_G^d ; **b** suffix tree M_G^d obtained by modifying R_G^d . In this example, the total lengths of all labels (excluding $\#_1$ symbols) are $f_{v''}$ until node v'' , and maximum $m_{v''}$ until two or more leafs in subtree B rooted at node v'' . This indicates that in suffix tree M_G^d , every leaf in B yields a common subarray satisfying the size lower bound

for RA^dQ . Otherwise, if $f_v \geq z > 0$ (i.e. size lower-bound is already achieved at v) and $m_v \geq z$ (and there are at least two descendant leafs achieving this lower bound) switch to an *inner-level* traversal (traversal starting at node v); otherwise, continue in the outer-level traversal in an ordinary way (not skipping M_v). In the inner-level traversal, perform a traversal starting at node v to its completion; and then return to the outer-level traversal and resume (after completing the traversal of subtree M_v rooted at v). By the inner-level traversal, collect all position elements $(j_1, (j_2, k_2), \dots, (j_d, k_d))$ from the lists L of leafs of the subtree M_v rooted at v ; if there are more than one leafs, make a note that the prefix from the root to node v of length f_v is common for all nodes in subtree M_v . We note that collectively the outer-level and inner-level traversal parts visit each node and edge for no more than twice in M_G^d . All complete prefixes found in subtrees M_v correspond to d -dimensional subarrays that satisfy the given size lower bound. This result and Lemma 2 imply the correctness of Theorem 3.

The time taken by the algorithm for the RA^dQ problem is $O(N_2^2 N_3^2 \dots N_d^2)$, which is dominated by the asymptotical traversal time based on the total number of nodes and edges in the suffix tree M_G^d . A naive (brute-force) algorithm would check all positions in G pairwise for a possible match of subarray of size z . This would take $\Omega(zN_1^2 N_2^2 \dots N_d)$ time. Our algorithm is faster than the naive algorithm by a factor of $\Omega(zN_1^2)$.

Each distinct sequence of tuples $(j_1, (j_2, k_2), (j_3, k_3), \dots, (j_d, k_d))$ found in lists stored at the leafs visited by the inner-level traversal corresponds to a distinct appearance of subarray g in slab $b_{(j_2, k_2)(j_3, k_3) \dots (j_d, k_d)}$. This appears in the suffix of $S_{(j_2, k_2)(j_3, k_3) \dots (j_d, k_d)}$ starting at index j_1 . This substring corresponds to an appearance of a subarray with the lexicographically smallest corner in dimension 1 at position j'_1 in G such that $|g| \geq z$. By also considering # symbols, the relation between j_1 and j'_1 are the following $j_1 = (j'_1 - 1)(k_2 + 1)(k_3 + 1) \dots (k_d + 1) + 2$. That is, $j'_1 = (j_1 - 2)/((k_2 + 1)(k_3 + 1) \dots (k_d + 1)) + 1$. $\square$

7 Finding all common subarrays in multiple arrays

Definition 6 *All Common Subarrays Query (CA^dQ):* Given an integer $z > 0$, find all occurrences of all subarrays each of size $\geq z$ common in all arrays $\{G_1, G_2, \dots, G_K\}$, where each G_i , $i \in [1, K]$, is of size $N_1 \times N_2 \dots \times N_d$.

Let $G_1, G_2, \dots, G_K$ be d -dimensional arrays ($d \geq 2$). Let for each $i \in [1, K]$, $B_{G_i}^d$ be the set of strings obtained using $G = G_i$ in Eq. 1. We set B_K^d to be the union of all $B_{G_i}^d$, $i \in [1, K]$.

That is, B_K^d includes strings from all the slabs of arrays in $\{G_1, G_2, \dots, G_K\}$. Let R_K^d be the generalized suffix tree created from B_K^d and $G_1, G_2, \dots, G_K$ in a similar way described in Proposition 2. One addition is that each list at a leaf in its position elements also records the host string for the suffix element (i.e. number i for G_i along with the starting position j_1 and slab information $(j_2, k_2), \dots, (j_d, k_d)$). We first process R_K^d to construct M_K^d in a similar way described (for constructing M_G^d in the proof of Theorem 3, and process it further in a bottom up fashion to color nodes that are shared by all K arrays. For this purpose we start at the very bottom (at the leafs), carry the lists L to the ancestor nodes. If a list L_v at a node v contains all array indexes in $[1, K]$ then we color v to black. That is, going up in M_K^d if a node has at least one black children then it will be colored to black; otherwise, it will be white. After completing the coloring step, we make another traversal on M_K^d and remove all non-black nodes from M_K^d in the subtrees rooted at these nodes. Let Q_K^d be the resulting suffix tree. That is, each node in Q_K^d , for every $i \in [1, K]$, leads to a leaf node that has a position element (tuple) containing i for array (the host string's number) in its list.

Compared to M_K^d , Q_K^d has $O(K)$ times more nodes and edges. Processing Q_K^d requires a few more traversals that involve collection of $O(K)$ times longer lists of position tuples at the leafs. Therefore, building suffix tree Q_K^d takes $O(K)$ times longer time than building M_K^d . We remark that this processing is done only once.

Theorem 4 *Given an integer $z > 0$, each problem CA^dQ on arrays $\{G_1, G_2, \dots, G_K\}$ can be solved in time $O(KN_2^2 \dots N_d^2)$, once suffix tree Q_K^d is built from $G_1, G_2, \dots, G_K$, where each G_i , $i \in [1, K]$, is of size $N_1 \times N_2 \dots \times N_d$.*

Proof Problem CA^dQ asks for substrings s of length $\geq z > 0$ (excluding # symbols) such that s is a complete prefix of a suffix of $S_{(j_2, k_2) \dots (j_d, k_d)}$, and s appears in all arrays $G_1, G_2, \dots, G_K$. The CA^dQ problem can be solved in a very similar way shown in the proof of Theorem 3 for the RA^dQ problem. The only differences are that the generalized suffix tree Q_K^d is used instead of M_G^d , and the common subarrays found need to be reported by including the number (identifier) of the host array. We note that there can be multiple different groups of subarrays of size $\geq z$. All such groups will be reported.

The correctness follows from Lemma 2 and by noting that, first, all nodes in Q_K^d are black (i.e. all nodes in Q_K^d yield subarrays shared by all K arrays); and second, the length lower bound is satisfied for every subarray found in the same way used in solving the RA^dQ problem. The time taken by our algorithm for the CA^dQ problem is $O(KN_2^2 N_3^2 \dots N_d^2)$. This is the time mainly spent on the traversal of the suffix tree Q_K^d with $O(KN_2^2 N_3^2 \dots N_d^2)$ nodes and edges. $\square$

naive (brute-force) algorithm for this problem would take time $\Omega(zN_1N_2 \dots N_d(N_1N_2 \dots N_d + \dots + N_1N_2 \dots N_d)) = \Omega(zKN_1^2N_2^2 \dots N_d^2)$ because it needs to consider all positions in G_1 and perform pairwise comparisons with all other arrays $G_2, \dots, G_k$ for finding exact matches. Our algorithm is faster than the naive algorithm by a factor of $\Omega(zKN_1^2)$. $\square$

We remark that the existence of common subarrays with larger total sizes implies the existence of common subarrays with smaller total sizes for the same input arrays. This observation can be used as a guide for searching common subarrays with maximal total sizes in a binary search manner.

8 Remarks on applications of subarray analysis

Our algorithms are applicable to analyzing objects that are stored in arrays. We can consider images of fixed locations (e.g. geographical maps in <https://gis.harvard.edu>), arrangement of various types of atoms (e.g. cubic lattice models of crystals [21]), arrangement of various types of cells (e.g. brain topology [22, 27]). In these cases, types of elements in arrays are defined in an alphabet that may also include “void” to represent a gap, or “wildcard” to represent a don’t-care-type element. Under such settings an adjacency matrix defines a local context in a large global body; and searching for substructures (sub-formations), repeating substructures, and common substructures in multiple objects are computable by our algorithms.

If certain attributes of elements are implied by their positions in an array, while these attributes are identified by the positions, an additional attribute can be stored in the array. For example, consider an adjacency matrix G for a graph. A given row number (or a column number) in G identifies a node, and position (i, j) in G stores edge (i, j) . Each submatrix g of G sufficiently represents a subgraph of this graph. We elaborate the use of this feature of adjacency matrices for graphs on an RNA substructure analysis method we propose in this section.

RNA is a polymer of four nucleotides namely adenine (A), cytosine (C), guanine (G), and uracil (U). A linear RNA sequence of nucleotides (a sequence of A, C, G, U) folds into a secondary structure in which nucleotides form base-pairs by making hydrogen bonds [12]. In each such structure, a number of different types of substructures are observed. These types are namely, unstructured single strand, bulge loop, hairpin loop, interior loop, stem, multi-branched loop, and pseudoknot. RNA secondary structure is shown in a 2D picture in which the linear nucleotide

sequence can be followed on the boundary from the terminals 5' to 3' (see [12] for details).

An RNA secondary structure can be represented by a graph (see [1] for example). We propose some slight modifications on such a graph to apply our new methods for analysis. The order of the nucleotides, and base-pairs are shown by edges. In Fig. 12 part (a) we illustrate an RNA secondary structure for a hypothetical example whose nucleotide sequence is CC...UGACA...UGACA...G. For a simply illustrative example, we only show two hairpin loops, and a few nucleotides at the beginning and end. We use the directed edges to indicate the nucleotide order, and undirected (doubly-directed) edges to indicate the base-pairs.

A recent work in RNA substructure analysis presents algorithms for searching for a given substructure (an RNA segment with a given folding), and for comparing RNA structures to find common substructures [1].

Our methods offer algorithms for RNA substructure analysis based on a graph representation of RNA secondary structures. We propose to use an adjacency matrix representation for RNA secondary structures. For a given RNA sequence S of length n , let the set of nodes of the graph representing it be the positions $\{1, 2, \dots, n\}$. Let $S[i]$ denote the nucleotide in the RNA sequence at index i . The node i in the graph has nucleotide value $S[i]$. That is, the position implies the nucleotide. Matrix elements store edges. There is an edge $(i, i+1)$ for each position $i \in [1, n-1]$, except that edge $(n, 1)$ does not exist. If nucleotides at positions i, j make a bond (base-pair) then edges (i, j) and (j, i) are present in the graph (therefore in the matrix). In this setting, the adjacency matrix for the edges of the graph can be used in substructure analysis.

In [1], several RNA secondary substructure-related problems are defined. Substructure search, multiple RNA structure comparison problems reduce to string problems which eventually are solved using suffix arrays. In this work, we reduce a set of problems defined on arrays to string problems. We solve the resulting problems using suffix trees. Both suffix trees and suffix arrays are efficient data structures used for similar objectives. There are trade-offs between them [13]. Suffix trees in our problems in this paper suite better for efficiency and ease of explanation.

Our work in this paper offers a general method on similar problems if analyzed objects can be modeled by using arrays. In the case of RNA substructure analysis, we show that RNA secondary structure can be represented by an adjacency matrix. RNA substructure search reduces to the SAQ problem and multiple RNA structure comparison reduces to the CAQ problem. The reductions are easy. The RNA structure problem instances can be efficiently converted to instances of the corresponding SAQ or CAQ

Fig. 12 **a** RNA secondary structure with repeating hairpin loop for a hypothetical RNA; **b** Adjacency matrix g for the subgraph representing the hairpin loop shown in part **(a)** and the adjacency matrix G for the entire RNA molecule. The row (or column) positions identify the nucleotides. For better illustration, instead of index (row and column numbers) we write the nucleotide at the index next to each row and column. We only show a few select edges, not all edges in G

problems, and solutions can be obtained by our algorithms efficiently.

Our search algorithm for SAQ finds for any given submatrix g constructed from a given substructure in a matrix G constructed once from a given RNA secondary structure. Our algorithm for RAQ finds all occurrences of g in G if size of interest z is at least $|g|$. Similarly, our algorithm for CAQ finds all g of size at least z in a set of matrixes G_i 's (combined into a single suffix tree) in all of which g is a common submatrix. For example, for the RNA secondary structure illustrated in Fig. 12, algorithm for SAQ finds the hairpin loop shown in the figure, and algorithm for RAQ reports this hairpin loop if the given size of interest for the query is less than 5 (the size of this hairpin loop). If there is a family of RNAs all of which contain this hairpin loop, algorithm for CAQ finds this hairpin loop if its size 5 meets the given size of interest value z for this query.

9 Limitations and contributions

Our image representation uses arrays. All subimages are generated as subarrays which are encoded in sequences. Although the number of sequences is large, these sequences are efficiently compressed into a generalized suffix tree. The most significant advantage of our approach is that it yields fast search and comparison algorithms. The superiority of our methods over naive (brute-force) methods is mainly thanks to suffix tree representation. For strings in two or higher dimensions, to the best of our knowledge, there does not exist any suffix-tree based

matching algorithm. The worst-case time complexity of existing algorithms for these problems (e.g.[3, 4, 7, 16]) is not better than that of brute-force. Since exact solutions require impractical execution time, many proposed approaches use similarity features for certain objects or classes of objects. A comprehensive study on this topic can be found in [29] which reports that average precision in many cases is around 70%. Face-recognition has attracted a special attention. It is found that 3D information helps with the face recognition and the precision of 93% is achieved on tested datasets [10].

Our suffix tree representation for all subarrays; search and comparison methods based-on this representation are novel ideas that have not appeared before in the literature. To the best of our knowledge, video comparison is a problem that was never introduced before. Our suffix tree-based approach makes it a tractable problem if frames are partitioned into grids of relatively larger cells.

The limitations of our matching approach are that they are not rotation and scale invariant. This is because all subimage information stored in the suffix tree is obtained from host image(s) at certain size and orientation. They are also affected by noise and changes at pixel (cell) level. However, our approach has still very useful applications in the following domains:

- 2-dimensional digital documents that contain symbols from a known source (size and orientation are fixed): in documents that include symbols such as those shown in Fig. 2, our search and comparison algorithms work efficiently and precisely.

- video: our algorithms for searching video by query-image (using extracted images from known sources, e.g. Fig. 4), and for comparing multiple videos (e.g. Fig. 5) work efficiently and precisely. Video-searching is an application in 3 dimensions since videos are arrays of 2-dimensional frames. Videos that belong to groups of people are arrays in 4 dimensions. The dimensions would grow with the hierarchy (e.g. country → group → individual videos). Our algorithms apply for all dimensions, and they have many practical use. One may want to locate most interesting snapshots of sports games and match those moments (e.g. goals in soccer games, finishing moments of runners and swimmers in olympic games, funny and unbelievable moments of human experiences). Video-clips are shared in many different videos. It is important to identify shared parts because of copyright verification, and detecting/avoiding fake-news and falsely-produced news for speculation or political gain (it has been reported that fake-news were generated from video-segments taken from computer games).
- arrays in dimensions two or larger: our algorithms apply to objects modeled by arrays. They work for 3-d objects (e.g. Fig. 3), and RNA secondary structures (e.g. Fig. 12).

10 Conclusion and future work

We present fast methods for three subimage analysis problems in a framework of strings. In this framework, finding subimages, repeating subimages, and common subimages in a single and multiple images reduce to string problems. We achieve fast solutions for these problems. These solutions are enabled by preprocessing images and building a generalized suffix tree from them. All subsequent instances of subimage analysis problems can be answered significantly faster in comparison to naive (brute-force) algorithms for these problems. Since our algorithms define the subimage problems on arrays, they are also applicable to the corresponding array problems, namely subarray search, finding repeating subarrays, and finding common subarrays in multiple arrays. All our solutions also generalize to dimensions higher than two. Our algorithms are applicable to substructure analysis of objects whose structures can be modeled by arrays.

For future work, we plan to address image analysis applications of our algorithms further. In these applications, we will aim to improve complexity, and achieve scale and rotation invariant search and comparison. We will

revisit the array representation for images. As can be seen in Fig. 1, the numbers of rows and columns used affect the precision of the image-representation and the storage size. Instead of storing each pixel, we can partition the image into a grid of cells with predefined size. Grid cells can be clustered based on a similarity definition. All grid cells in each cluster can be mapped to and represented by a symbol in an alphabet. We can represent the grid by a matrix that uses symbols from this alphabet. The numbers of rows and columns in the matrix can be significantly smaller compared to the numbers of pixels in the input image. This will also reduce the space and time requirement of our algorithms.

Another approach for matrix representation for images would compute and use as reference the points of interest (e.g. corners) for images. For a given image, this approach stores in a matrix the points of interest and a subset chosen from non-interest points. The benefit of this new approach is that not only it reduces space and time requirement, but also improves scale and rotation invariant feature of the resulting image analysis algorithms. If points of interest are defined based on the geometric features of images, they will be scale invariant. For rotation invariant feature, we propose the following: In a given matrix, starting at a given element p , let a *spiral ordering-based traversal* be defined as a curve on a plane that winds around p by treating p as a fixed center point and by passing through the matrix elements that are at continuously increasing distance from p . For a given matrix, for every point p in this matrix, we propose that we generate a string (the *spiral string* for p) up to a given length by following the spiral order starting at p . Let H be a given host matrix. A generalized suffix tree P can be generated from spiral strings for p (up to a given length) for all possible p in H . Let h be another given matrix, and s be the spiral string for c , where c is the "center" point in h . String s can be searched in a generalized suffix tree P . Due to the cyclic nature of suffixes of generated spiral strings by following spiral order, we expect to find a sufficiently long suffix-match for s in P if and only if the query matrix h appears in the host matrix H as a submatrix in a rotation invariant manner. We plan to study these and other similar ideas in future work.

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

1. Arslan AN, Anandan J, Fry E, Monschke K, Ganneboina N, Bowerman J (2017) Efficient RNA structure comparison algorithms. *Spec Issue J Bioinform Comput Biol World Sci* 15(6):17400009. <https://doi.org/10.1142/S02197200177400091>
2. Arslan AN, Hempelmann CF, Attardo S, Blount GP, Sirakov NM (2015) Threat assessment using visual hierarchy and conceptual firearms ontology. *Opt Eng* 54(5):053109. <https://doi.org/10.1117/1.OE.54.5.053109>
3. Baeza-Yates R, Rgnier M (1990) Fast algorithms for two dimensional and multiple pattern matching. In: Gilbert JR, Karlsson R (eds) *SWAT 90 1990. Lecture notes in computer science*, vol 447. Springer, Berlin, pp 332–347
4. Baker T (1978) A technique for extending rapid exact string matching to arrays of more than one dimension. *SIAM J. Comput.* 7:533–541
5. Bhuyan MH, Bhattacharyya DK (2009) An effective fingerprint classification and search method. *IJCSNS Int. J. Comput. Sci. Netw. Secur.* 9(11):39–48
6. Bieganski P, Riedl J, Carlis J, Retzel EF (1994) Generalized suffix trees for biological sequence data. In: *Proceedings of the twenty-seventh Hawaii international conference on biotechnology computing*, pp 35–44
7. Bird R (1977) Two dimensional pattern matching. *Inf Proc Lett* 6:168–170
8. Biradar V, Sarojadevi H (2012) Image analysis techniques for fingerprint recognition. *Int J Comput Eng Res* 2(3):606–615
9. Bosch P, van Ballegooij A, de Vries AP, Kersten M (2001) Exact matching in image databases. In: *IEEE international conference on multimedia and expo, ICME 2001*, pp 513–516. <https://doi.org/10.1109/ICME.2001.1237739>
10. Crawford M (2011) *Facial recognition progress report*. SPIE Newsroom, Published Sep. 28
11. Dong W, Wang Z, Charikar M, Li K (2012) High-confidence near-duplicate image detection. In: *ACM international conference on multimedia retrieval*, Hong Kong, June 5–8
12. Durbin R, Eddy S, Krogh A, Michison G (1998) *Biological sequence analysis*. Cambridge University Press, Cambridge
13. Fogh JOS (2014) *Pattern matching using suffix trays, arrays and trees*. Thesis, Aarhus University, October 2014
14. Gusfield D (1997) *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge University Press, Cambridge ISBN 0-521-58519-8
15. Huo H, Wang X, Stojkovic V (2009) Repeats identification using improved suffix trees. *Int J Comput Biol Drug Des* 2(3):265–277
16. Karkkinen J, Ukkonen E (1994) Two and higher dimensional pattern matching in optimal expected time. In: *Proceedings of SODA'94, SIAM*, pp 715–723
17. Katukam R, Sindhoora P (2015) Image comparison methods and tools: a review. In: *1st national conference on emerging trends in information technology [ETIT], 28th–29th December 2015*, pp 35–42
18. Ke Y, Sukthankar R, Huston L (2004) Efficient near-duplicate detection and sub-image retrieval. In: *ACM Multimedia*, October 10–16
19. Khorsheed OK (2014) A review search bitmap image for sub image and the padding problem. *Int J Adv Eng Technol* 7(3):684–691 ISSN: 22311963
20. Kim H-S, Chang H-W, Liu H, Lee J, Lee D (2009) BIM: image matching using biological gene sequence alignment. In: *16th IEEE international conference on image processing (ICIP)*, Cairo, Egypt. <https://doi.org/10.1109/ICIP.2009.5414214>
21. Kosevich AM (2006) *The crystal lattice: phonons, solitons, dislocations, superlattices*, 2nd edn. Wiley VCH Verlag GmbH & Co, KGaA. ISBN 9783527405084
22. Kumar A, Kim J, Cai W, Fulham M, Feng D (2013) Content-based medical image retrieval: a survey of applications to multidimensional and multimodality data. *J Digit Imaging* 26(6):1025–1039
23. Lee I, Iliopoulos CS, Park K (2007) Linear time algorithm for the longest common repeat problem. *J Discrete Algorithms* 5(2):243–249. <https://doi.org/10.1016/j.jda.2006.03.019>
24. Liu M, Jiang X, Kot AC (2007) Efficient fingerprint search based on database clustering. *Pattern Recognit* 40:1793–1803
25. Liu B, Zhu Y, Wang C, Li M, Shi W, Mao Y (2016) Finding all-one hyper-submatrix of an incidence matrix. In: *IEEE 18th international conference on high performance computing and communications; IEEE 14th international conference on smart city; IEEE 2nd international conference on data science and systems (HPCC/SmartCity/DSS)*, Sydney, NSW, Australia. <https://doi.org/10.1109/HPCC-SmartCity-DSS.2016.0149>
26. Main MG, Lorentz RJ (1984) An $O(n \log n)$ algorithm for finding all repetitions in a string. *J Algorithms* 5(3):422–432. [https://doi.org/10.1016/0196-6774\(84\)90021-X](https://doi.org/10.1016/0196-6774(84)90021-X)
27. Oishi K, Chang L, Hao Huang H (2019) Baby brain atlases. *NeuroImage* 185:865–880. <https://doi.org/10.1016/j.neuroimage.2018.04.003>
28. Sebe N, Lew MS, Huijsmans DP (1999) Multi-scale sub-image search. In: *Proceedings of the seventh ACM international conference on Multimedia 1999 (Part 2)*, Orlando, Florida, pp 79–82. <https://doi.org/10.1145/319878.319901>
29. Sivic J (2006) *Efficient visual search of images and videos*. PhD Thesis, Robotics Research Group, Department of Engineering Science, University of Oxford
30. Sonka M, Hlavac V, Boyle R (2015) *Image Processing, analysis, and machine vision*, 4th edn. Cengage Learning ISBN-13: 978-1133593607
31. Ukkonen E (1995) On-line construction of suffix trees. *Algorithmica* 14(3):249–260. <https://doi.org/10.1007/BF01206331>

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.