DUJASE Vol. 3(1) 55-60, 2015 (January)

ACM: An Efficient Exact Algorithm for Finding Planted Motif in Biological Sequences.

S. M. Iqbal Morshed[1] and Saifuddin Md. Tareeq[1]

[1]*Department of Computer Science and Engineering, University of Dhaka, Dhaka 1000, Bangladesh*

E-mail: iqbal.bgd@gmail.com and smtareeq@cse.univdhaka.edu

Received on 16. 02. 2014. Accepted for publication on 14.07. 2014.

Abstract

Motifs are patterns found in biological sequences that are significant for understanding gene function, human disease, drug design, etc. Finding motif is one of the most important and challenging tasks in bioinformatics. In this paper a new algorithm named ATGC Counter Map (ACM) is proposed to solve motif finding problem. The proposed algorithm provides several features. First, a new data structure named ATGC Counter Map is proposed to reduce the search space. Second, as an exact algorithm it guarantees to find motif from input sequences. And third, experimental results show that the algorithm outperforms three of the well-known exact algorithms by (i) solving challenging instances that these existing algorithms failed to solve and by (ii) reducing run time for larger motif sequences.

Keywords: ATGC counter Map, Planted Motif Search, Clique Finding, Profile Matrix, Consensus Motif.

1. Introduction

Many significant biological problems have been solved by discovery of patterns in DNA and protein sequences. For instance, identification of patterns in nucleic acid sequences has helped in determining open reading frames, identifying promoter elements of genes, identifying intron/exon splicing sites, locating RNA degradation signals etc. In case of protein sequence, pattern identification helped in domain identification, protease cleavage site location, signal peptide identification, protein degradation element identification, short functional motif discovery etc. Motifs are patterns found in biological sequences which are important for understanding many biological subjects like gene function, human disease, drug design etc. As a result, motif identification plays a significant role in biological studies. The motif search problem has attracted many researchers over last two decades. In literature, many versions of the motif finding problem have been enumerated. Examples include Simple Motif Search (SMS), Planted Motif Search (PMS)-also known as (l, d) motif search, and Edit-distance-based Motif Search (EMS). In this paper, focus will be on the PMS problem. 'D'

PMS is stated as follows. Given n sequences and two integers, motif length l and mismatches (substitutions) allowed d as input. For simplicity, it is assumed that the length of each sequence is m. The problem is to identify all strings M of length l such that M occurs in each of the n sequences with at most d mismatches. Formally, string M has to satisfy the following constraint: there exists a string M_i of length l in sequence i, for every $(1 \leq i \leq n)$, such that the number of mismatches between M and M_i is less than or equal to d. String M is called a *motif*. For example, if the input sequences are GCGCGAT, CAGGTGA and CGATGCC; $l = 3$ and $d = 1$, then GAT and GTG are some of the $(3,1)$ motifs.

PMS is a well-studied problem and it has been shown to be NP-hard. As a result, all known exact algorithms for PMS take exponential time in some of the parameters in the worst case. Two kinds of algorithms have been proposed in the literature for PMS: exact and approximate. While an exact algorithm always finds the correct motif, an approximate algorithm may not always find the correct motif. Typically, approximate algorithms tend to be faster than exact algorithms. Some approximate algorithms are due to Bailey and Elkan [1], Buhler and Tompa [2], Lawrence *et. al.* [10], Pevzner and Sze [11]. These algorithms are based on local search techniques such as Gibbs sampling and expectation maximization. Some other approximate algorithms are: PROJECTION [2], MULTIPROFILER [8], Pattern Branching [12], CONSENSUS [7], GibbsDNA [10], MEME [1], and ProfileBranching [12].

Although approximate algorithms are acceptable in some cases in practice, exact algorithms are preferable since they are guaranteed to report correct (l, d) motifs. For biologists, correct motifs found by an algorithm could be much more important than its run time. Considering this, in this paper, focus is given on efficient exact algorithms. Some exact algorithms are due to Dinh *et. al.* [4][5], Davila *et. al.* [3], Rajasekaran *et. al.* [13], Eskin and Pevzner [6]. Some efficient exact algorithms are MITRA-count [6], MITRA-graph [6] and PMS2 [13].

In this paper a new motif finding algorithm called ACM is proposed. This new algorithm has the following contributions. First, it presents a new data structure, named ATGC Counter Map to reduce the search space which essentially reduces the run time of the algorithm. Second, the proposed algorithm is an exact algorithm. Therefore it guarantees to detect correct motif from the sequence. Third, performance of the proposed algorithm is compared with three existing exact algorithms. The proposed algorithm outperforms these exact algorithms in terms of run time and instance solving. In the following sections the details of the proposed algorithm is discussed.

2. Proposed Methods of Motif Finding

2.1 Notations and definitions

In this section some notations and definitions are introduced that will help to describe the proposed algorithm clearly.

Definition 1. A string $x = x[1] \ldots . x[l]$ of length l is called an *l-mer*. Set of all l-mers from input sequence is denoted as S.

Definition 2. Given two strings x and y of equal length, the *Hamming distance* between x and y, denoted by $d_H(x, y)$ is the number of mismatches between them.

Definition 3. A string C_x is called *ATGC count* of an l-mer x if C_x is of the form <number of A in x>.<number of T in x>.<number of G in x>.<number of C in x>. For example, ATGC count of l-mer AATCCG is 2.1.1.2 . In general any ATGC count is denoted by C whereas for a specific l-mer x ATGC count is denoted by C_x.

Definition 4. Given two ATGC Count C_x and C_y where l is same, the *Count distance* between C_x and C_y, denoted by $d_C(C_x, C_y)$ is the total number of difference in number of A,T,G and C. For example, Let, $C_x = 1.3.2.1$ and $C_y = 2.1.2.2$, then $d_C(C_x, C_y) = (1 + 2 + 0 + 1) = 4$.

Definition 5. A set of ATGC Count C_{y_i} is called *Neighbor of an ATGC Count* C_x if $d_C(C_x, C_{y_i}) \leq 2d$ and C_{y_i} is greater than C_x. It is denoted by $N(C, d)$. For example- neighbor of C=1.1.2.1 for d=1 are {1.1.3.0, 1.2.2.0, 2.1.2.0, 1.2.1.1, 2.1.1.1, 1.2.2.1}.

Definition 6. *ATGC Counter Map* data structure is a map data structure containing *(key,value)* pair where *key* is ATGC count C, and value is list of l-mers which have same number of A,T,G,C as C's internal '.' separated value represents.

Definition 7. For a key each l-mer stored in the value is called its *member*. A member is denoted as μ. For example, members of C=1.1.2.1 can be patterns ATGGC, GGCAT, TACGG etc. where all patterns have same number of A,T,G,C as C represents. A member contains two lists. First one is *Current Neighbor*, which contains all l-mers x such that $d_H(\mu, x) \leq 2d$ and $C_x \geq C_\mu$ where $x \in S$. Second one is *Previous Neighbor*, which contains all l-mers x such that $d_H(\mu, x) \leq 2d$ and $C_x < C_\mu$ where $x \in S$

Definition 8. For a key=C, the *Neighbor of the key* are all the key which contains the value from $N(C, k)$, where $k = 1$ to $2d$. For all *neighbor key*, C is defined as *home key*.

Definition 9. For each member μ, *Member group* of μ denoted by γ contains all l-mers x such that $d_H(\mu, x) \leq 2d$ and $x \in S$. Member group contains the member itself, current neighbor and previous neighbor of the member.

Definition 10. Given a set of n strings $S_1, \ldots, S_n$ of length m each, a string M of length l is called an (l, d)-motif of the strings if there are at least n strings such that the Hamming distance between each one of them and M is no more than d. M is called an (l, d)-motif.

2.2 ATGC Counter Map Algorithm

2.2.1 Overview of the Algorithm

ACM algorithm uses ATGC Counter Map to store each l-mer. Then for each key in map it traverses the neighbor of the key. To do this proposed NeighborTraverse algorithm is used. By traversing all the neighbors, for each member in key, its member group is stored. Then a graph for each member group is created using MITRA [6] algorithm. MITRA [6] algorithm made some improvements on WINNOWER [11] algorithm for creating such graph. Finally, clique finding algorithm described in [9] is used to find clique of size n in the graph. If a clique of size greater than or equals n is found, a profile matrix for the nodes of the clique is created. From profile matrix, a consensus string σ is formed. If for all nodes of clique, hamming distance between a node and consensus string $d_H(x_i, \sigma) \leq d$, where $1 \leq i \leq n$, then this consensus string σ is our desired motif. Otherwise, there is no motif.

Here the ACM algorithm is described in three subsections. In subsection 2.2.2 ATGC Counter Map data structure is discussed. In subsection 2.2.3 NeighborTraverse algorithm is presented and how it is used to find the neighbor of a key is described. Also how a member group of a member in the key is stored is described in this section. And finally in subsection 2.2.4 how a graph is created from member group and how clique finding algorithm is implemented on the graph to find the motif is discussed.

2.2.2 ATGC Counter Map data structure

ATGC Counter Map data structure uses the map data structure to store each l-mer in S. Map data structure stores *(key, value)* pair where each key is unique. In ACM algorithm, *ATGC count* is used as key and corresponding list of l-mer (i.e. members) as value. *ATGC Counter Map* is created by the following CreateACM procedure as given in algorithm 1.

Algorithm 1 Create ATGC Counter Map

1: **procedure** CreateACM(map M)
2: **for** each l-mer ϵ S **do**
3: compute its ATGC count C.
4: **if** C is already present in the map as a key **then**
5: push l-mer at the back of the member list.
6: **else**

7: insert C in M as key
8: push the l-mer as value of key $= C$
9: **end if**
10: **end for**
11: **end procedure**

For example, insertion of the l-mers {TGAGG, ATTGC, AGTGC, AGAGC, AGTAC, TTACC, GAGTG, AGTTC} in ACM where $l = 5$ is illustrated in **Fig. 1**. As the ATGC Counter Map is sorted by key value, inserted l-mers appear in sorted order of its ATGC Count value.

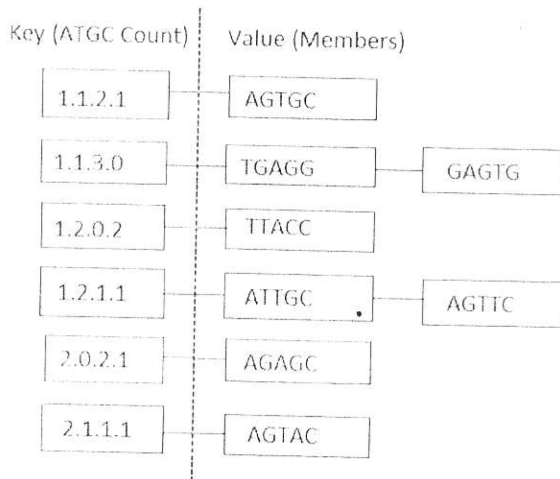

Fig. 1: ATGC Counter Map after insertion of the l-mers {TGAGG, ATTGC, AGTGC, AGAGC, AGTAC, TTACC, GAGTG, AGTTC} where $l = 5$. This map is sorted by ATGC Count Value

Complexity of insertion is same as the complexity of inserting into a map which is $O(\log K)$ where k is the size of the map which is the total number of ATGC Count value.

2.2.3 Neighbor Traversing Algorithm

After building up ATGC Counter Map, for each key (i.e. ATGC Count) its neighbors are traversed. While traversing a neighbor, for each member $\mu_i (1 \leq i \leq p)$ in home key all members $\mu_j (1 \leq j \leq q)$ in neighbor key is compared, where p and q are number of members in home and neighbor key respectively. If $d_H(\mu_i, \mu_j)$ is no more than $2d$, then μ_i stores μ_j as its current neighbor member (Definition 7) and μ_j stores μ_i as its previous neighbor member (Definition 7).

The algorithm is optimized by defining neighbors as those values which are greater than home key. This optimization works because the ATGC Counter map is sorted in ascending order of key value. Therefore, a neighbor key in current operation will be home key in any of the next operation. So, when a neighbor with greater value is traversed, member of the home key is stored as the previous neighbor member of the neighbor key where conditions are fulfilled. So, home key doesn't need to traverse smaller neighbors as smaller neighbors members are already stored in previous neighbor member of the home key.

To create neighbor $N(C, d)$ like the example provided in definition 5, it needs to reduce (or subtract) amount d from position i of the home key, where $2 \leq i \leq 4$ in all possible way and then distribute (or add) d in position(s) j of home key,where $1 \leq j \leq 4$ and $j \neq i$, in all possible way.

To traverse neighbors, a new recursive algorithm is proposed. Pseudocode of the NeighborTraverse algorithm is provided in the Algorithm 2. In procedure NeighborTraverse home key C_x and ATGC Counter Map M is provided as parameter.

Algorithm 2 Traversing neighbor key of the home key
1: **procedure** NeighborTraverse(C_x, M)
2: $z = C_x$
3: **for** all possible Reduction of amount from position i of z,where $2 \leq i \leq 4$ **do**
4: **for** all possible Distribution of amount at position j of z, where $1 \leq j \leq 4$ and $j \neq i$ **do**
5: Compute a new ATGC Count C_y
6: **if** $C_y \geq C_x$ and C_y is a key in M **then** CompareMembers(C_x, C_y)
7: **end if**
8: **end for**
9: **end for**
10: **end procedure**
11: **procedure** CompareMembers(C_x, C_y)
12: **for** all members $\mu_i \epsilon\ C_x$ **do**
13: **for** all members $\mu_j \epsilon\ C_y$ **do**
14: **if** $(d_H(\mu_i, \mu_j) \leq 2d)$ **then**
15: μ_i.Cureent Neighbor Member $= \mu_j$
16: μ_j.Previous Neighbor Member $= \mu_i$
17: **end if**
18: **end for**
19: **end for**
20: **end procedure**

For example, to traverse the neighbor of 1.1.2.1 with $d = 1$, all possible reduction from position $i, 2 \leq i \leq 4$ would be {1.1.2.0, 1.1.1.1, 1.0.2.1}. Now for each possible reduction there are different possible distributions. For {1.1.2.0}, 3 possible distributions, preserving the condition mentioned in line 4, are {1.1.3.0, 1.2.2.0, 2.1.2.0}. Similarly, {1.2.1.1, 2.1.1.1, 1.1.1.2} and {2.0.2.1, 1.0.3.1, 1.0.2.2} are the possible distribution of {1.1.1.1} and {1.0.2.1} respectively, preserving the conditions.

So, for the home key {1.1.2.1}, 9 neighbor keys {1.1.3.0, 1.2.2.0, 2.1.2.0 1.2.1.1, 2.1.1.1, 1.1.1.2, 2.0.2.1, 1.0.3.1, 1.0.3.2} are obtained. According to the condition in line 6, 3 neighbors {1.1.1.2, 1.0.3.1, 1.0.2.2} which are less than home key are pruned. And finally there remain 6 neighbors {1.1.3.0, 1.2.2.0, 2.1.2.0, 1.2.1.1, 2.1.1.1, 2.0.2.1}.

From ATGC Counter Map created in **Fig. 1**, the neighbor keys of 1.1.2.1 are found which is shown in **Fig. 2(a)**. Here {1.2.2.0, 2.1.2.0} are not included because they are not key in ATGC Counter Map illustrated in **Fig. 1**. In **Fig. 2(b)** the current neighbor members of the member `AGTGC' is shown which is stored during the traversal of neighbor keys in **Fig. 2(a)**.

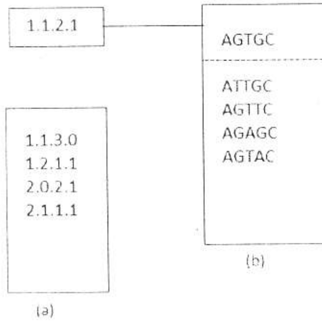

Fig. 2: (a) shows the neighbor keys of 1.1.2.1 and (b) shows the current neighbor members of `AGTGC' which is stored by traversing neighbor keys in (a)

2.2.4 Graph Creation and Clique Finding

During the traversal of neighbor key, each member μ of the home key stores its current neighbor member such that for each l-mer in current neighbor member x_i, $d_H(x_i, \mu) \leq 2d$, where $1 \leq i \leq r$ and r is the size of the current neighbor member. Each member μ also has previous neighbor member which is stored during the traversal process of any previous home key smaller than current home key. For each l-mer in previous neighbor member y_i, $d_H(y_i, \mu) \leq 2d, 1 \leq i \leq t$, where t is the size of the previous neighbor.

For each member, a member group (definition 9) is formed by fulfilling current neighbor and previous neighbor. In figure 2(b), member group of AGTGC is its current neighbor members because it has no previous neighbor member.

For each member group a graph using WINNOWER [11] algorithm is created. According to WINNOWER each l-mer in the member group is a vertex. An edge connects two vertices if the corresponding l-mers have no more than $2d$ mismatches and they are from different sequence.

For example, a graph from the member group depicted in **Fig. 2(b)** is created. The member group is redefined as in **Fig. 3(a)**. If it is assumed, for the sake of simplicity, that in **Fig. 3(a)** no two l-mers in member group are from same sequence and $2d = 1$, a graph illustrated in **Fig. 3(b)** can be constructed. Here, a number inside each node represents corresponding l-mer of the member group of AGTGC shown in **Fig. 3(a)**. There is an edge between two nodes if and only if hamming distance between two nodes is not more than 1.

Improvements done by MITRA [6] algorithm is also incorporated in ACM algorithm. MITRA made improvement by removing spurious edges from the graph. At each node of the tree, it removes edges by computing the degree of each vertex. If the degree of the vertex is less than n, it can remove all edges that lead to the vertex since it is not part of a clique. It repeats this procedure until it cannot remove any more edges. If the number of edges remaining is less than the minimum number of edges in the clique, the existence of a clique is ruled out.

If number of edges remaining is greater than or equal to n clique finding algorithm is applied as proposed in [9]. Applying clique finding algorithm in **Fig. 3(b)**, the clique in **Fig. 3(c)** is obtained. By definition of clique, this is the maximal complete subgraph of graph in **Fig. 3(b)**.

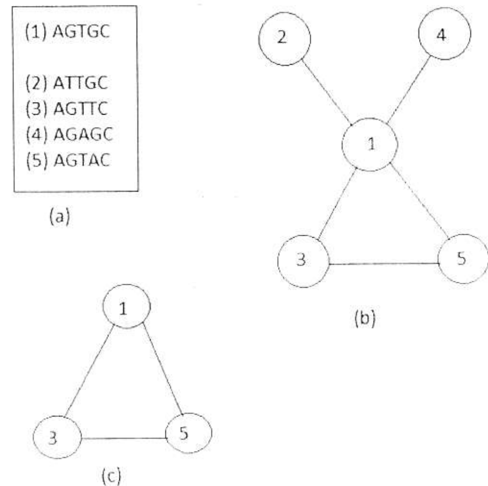

Fig. 3: (a) shows the member groups of AGTGC. (b) shows the graph created from (a). (c) shows the clique found from graph in (b)}

If a clique of size greater than or equals to n is found, a profile matrix for the nodes of the clique is created. From profile matrix, a consensus string σ is formed. If for all nodes (i.e. l-mer) of clique x_i, hamming distance between a node and consensus string $d_H(x_i, \sigma) \leq d$, then this consensus string σ is the desired motif M.

For example, from the clique in **Fig. 3(c)**, a profile matrix and consensus string as in **Fig. 4** is created assuming $n = 3$ and $d = 1$. For all nodes of clique, hamming distance between a node and consensus string is no more than d (where $d = 1$). Therefore this consensus string is the motif.

Clique		A	G	T	G	C
		A	G	T	T	C
		A	G	T	A	C
Profile Matrix	A	3	0	0	1	0
	T	0	0	3	1	0
	G	0	3	0	1	0
	C	0	0	0	0	3
Consensus String		A	G	T	G	C

Fig. 4: From clique nodes, profile matrix is created. From profile matrix, consensus string is created.

3. Results and Discussion

In this section performance of ACM algorithm is compared with three other well-known exact algorithms - PMS2 [13], MITRA Graph [6], MITRA Count [6]. Algorithms for motif search typically tested on random input data sets. Any such data set will consist of 20 random strings each of length 600 ($n = 20, m = 600$). A random motif of length l is planted at a random position in each of the strings, mutating it in exactly d positions. The algorithms are tested for varying l and d values. In particular, the following instances: (11,2), (12,3), (13,3), (15,4), (17,4), (19,5), (21,5) and (28,8) are employed.

Table 1: Run time in seconds of MITRA Count, MITRA Graph and ACM algorithms on different instances all in a Pentium III, 750 MHz and 1GB RAM machine

	MITRA Count [6] PIII, 750MHz, 1GB RAM	MITRA Graph [6] PIII, 750MHz, 1GB RAM	ACM PIII, 750MHz, 1GB RAM
(11,2)	60	60	52
(12,3)	60	240	1223
(13,3)	120	120	199
(15,4)	300	300	615
(17,4)	-	-	155
(19,5)	-	-	512
(21,5)	-	-	44
(28,8)	-	240	134

Table 2: Run time in seconds of PMS2 and ACM algorithms on different instances in a Pentium 4, 2.4 GHz and 1GB RAM machine and also ACM in a Intel Core i5, 2.5 GHz and 4GB RAM machine

	PMS2 [13] (P4, 2.4GHz, 1GB RAM)	ACM (P4, 2.4GHz, 1GB RAM	ACM (Corei5,2.5GHz,4GB RAM)
(11,2)	0.78	45	12
(12,3)	15.5	860	250
(13,3)	20.98	154	42
(15,4)	217	413	126
(17,4)	216	110	33
(19,5)	-	398	109
(21,5)	-	30	8
(28,8)	-	128	38

The ACM algorithm is run on two machines with different specifications to compare algorithms. To compare with PSM2 [13] which run on a 2.4 GHz Intel Pentium 4 processor having 1GB RAM, the ACM is run on a machine with 2.4 GHz Intel Pentium 4 processor having 1GB RAM. To compare with MITRA-Graph [6] and MITRA-Count [6] which run on a machine with 750 MHz Intel Pentium III processor having 1GB RAM, the ACM is run on a machine with 750 MHz Intel Pentium III processor having 1GB RAM. And hence the running time, taken from the respective papers, is comparable as they are run on machine with same specifications.

Table 1 and 2 shows the performance of the algorithms on challenging instances. In the tables as mentioned in [4], the

letter '-' means that corresponding algorithm cannot solve the challenging instance in the experimental settings.

The algorithms are compared in terms of run-time and instance solving. For run time, as shown in Table 1 and Table 2 for $l \leq 15$ MITRA-Graph [6], MITRA-Count [6] and PSM2 [13] performs better than the ACM. But as the length l increases (i.e. for $l > 15$) ACM outperforms all these algorithms by almost a factor of 2. This is because, as the value of l increases, number of edges in graph created on member groups decreases. Therefore clique finding algorithm can perform faster.

For instance solving, again from Table 1 and Table 2 it is seen that ACM can solve instances in case of $d \geq 5$ where PSM2 [13] fail to solve these instances. And for ACM with larger d if the ratio of mismatches is lower ((19,5) and (21,5)) then it took less time. This is because as the mismatches increase the neighbor finding procedure of ACM algorithm performs steadily while PSM2 fail to handle this situation. In case of (18,6) where the allowed mismatches are too high (66.66%), for a single l-mer, number of possible neighbor l-mer increases exponentially. As the neighbor increases, both the vertices and corresponding edges increase exponentially in a neighbor group. Consequently, clique finding algorithm fails and consequently ACM also fails in this situation.

4. Conclusion

In this paper a new algorithm named ACM is proposed. ACM algorithm uses a new data structure named ATGC counter map which efficiently reduces the search space for neighbor of an l-mer. We also developed a novel algorithm named NeighborTraverse to traverse the neighbor of an l-mer on ATGC Counter Map. The proposed ACM algorithm outperforms existing algorithm in terms of run-time for larger instances ($d \geq 15$). It also solves instances where number of mismatches $d \geq 5$ which previous algorithm PMS2 failed to solve. Performance and accuracy are main issues in biological sequence analysis and average performance of the proposed algorithm is much better than existing ones. Future scope will be to allow insertion and deletion in the motif.

References

1. T. L. Bailey and C. Elkan. Fitting a mixture model by expectation maximization to discover motifs in biopolymers. Proceedings of the Second International Conference on Intelligent Systems for Molecular Biology, 2:28-36.
2. J. Buhler and M. Tompa. Finding motifs using random projections. Journal of computational biology, 9(2):225-242, Jan. 2002
3. J. Davila, S. Balla, and S. Rajasekaran. Pampa: An improved branch and bound algorithm for planted (l, d) motif search. Technical report, 2007.
4. H. Dinh, S. Rajasekaran, and J. Davila. qPMS7: a fast algorithm for finding (l, d)-motifs in DNA and protein sequences. PloS one, 7(7):e41425, Jan. 2012.
5. H. Dinh, S. Rajasekaran, and V. K. Kundeti. PMS5: an effcient exact algorithm for the (l, d)-motif finding problem. BMC bioinformatics, 12(1):410, Jan. 2011.
6. E. Eskin and P. A. Pevzner. Finding composite regulatory patterns in DNA sequences. Bioinformatics, 18(1):354-363, 2002.
7. G. Z. Hertz and G. D. Stormo. Identifcation of consensus patterns in unaligned DNA and protein sequences: a large-deviation statistical basis for penalizing gaps. Proceedings of the Third International Conference on Bioinformatics and Genome Research, pages 201-216, 1995.
8. U. Keich and P. A. Pevzner. Finding motifs in the twilight zone. Bioinformatics, 18(10):1374-1381, Oct. 2002.
9. J. Konc and D. Janezic. An improved branch and bound algorithm for the maximum clique problem. MATCH Communications in Mathematical and in Computer Chemistry, 58:569-590, 2007.
10. C. E. Lawrence, S. F. Altschul, M. S. Boguski, J. S. Liu, a. F. Neuwald, and J. C. Wootton. Detecting subtle sequence signals: a Gibbs sampling strategy for multiple alignment. Science (New York, N.Y.), 262(5131):208-214, Oct. 1993.
11. P. A. Pevzner and S.-H. Sze. Combinatorial approaches to finding subtle signals in DNA sequences. Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology. AAAI Press, San Diego, pages 269-278, 2000.
12. A. Price, S. Ramabhadran, and P. A. Pevzner. Finding subtle motifs by branching from sample strings. Bioinformatics, 19(Suppl 2):ii149-ii155, Oct. 2003.
13. S. Rajasekaran, S. Balla, and C.-H. Huang. Exact algorithms for planted motif problems. Journal of computational biology, 12(8):1117-28, Oct. 2005.