A Fast Algorithm to Calculate Powers of a Boolean Matrix for Diameter Computation of Random Graphs

Md. Abdur Razzaque, Choong Seon Hong, M. Abdullah-Al-Wadud, and Oksam Chae

Department of Computer Engineering, Kyung Hee University
1 Seocheon-ri, Kiheung-eup, Yongin-si, Gyonggi-do, South Korea, 449-701
m_a_razzaque@yahoo.com, cshong@khu.ac.kr,
awsujon@yahoo.com, oschae@khu.ac.kr

Abstract. In this paper, a fast algorithm is proposed to calculate k^{th} power of an $n \times n$ Boolean matrix that requires $O(kn^3p)$ addition operations, where p is the probability that an entry of the matrix is 1. The algorithm generates a single set of inference rules at the beginning. It then selects entries (specified by the same inference rule) from any matrix A^{k-1} and adds them up for calculating corresponding entries of A^k. No multiplication operation is required. A modification of the proposed algorithm[1] can compute the diameter of any graph and for a massive random graph, it requires only $O(n^2(1\text{-}p)E[q])$ operations, where q is the number of attempts required to find the first occurrence of 1 in a column in a linear search. The performance comparisons say that the proposed algorithms outperform the existing ones.

Keywords: Boolean Matrix, Random Graphs, Adjacency Matrix, Graph Diameter, Computational Complexity.

1 Introduction

Boolean matrix and its powers play a major role in mathematical research, electrical engineering, computer programming, networking, biometrics, economics, marketing and communications - the list can go on and on [1, 2, 3]. In these applications, Boolean matrices interpret the relationship among the nodes of a network, genes of different organisms, points of a circuit etc. For drawing and comparison of RNA secondary structure, [1] builds a path matrix by calculating higher powers of the input distance matrix. [2] presents an algorithm to calculate the network capacity in terms of the maximum number of k-hop paths based on the k-hop adjacency matrix of the network. [3] produces an adjacency matrix from a large database containing compiled gene ontology information for the

[1] This research was supported by the MIC, Korea, under the ITRC support program supervised by the IITA, Grant no - (IITA-2006 - (C1090-0602-0002)).

S.-i. Nakano and Md. S. Rahman (Eds.): WALCOM 2008, LNCS 4921, pp. 58–69, 2008.

genes of several model organisms. Higher powers of this matrix are then calculated to determine how closely the genes are related on biological processes and molecular functions.

Adjacency matrix is a Boolean matrix that represents the dependency among vertices of a graph. If A is the adjacency matrix of a random graph $G(n,p)$, the entries in its k^{th} power gives the number of walks of length k between each pair of vertices [4]. To find out A^k from A, $\forall k \geq 2$, is one of the most fundamental problems in graph theory. Strassen [5] made the astounding discovery that one can multiply two $n \times n$ matrices recursively in only $O(n^{2.81})$ multiplication operations, compared with $O(n^3)$ for the standard algorithm. The constant factor implied in the big O notation of this algorithm is about 4.695. A sequence of improvements have been done to Strassen's original algorithm [6]. The best one is achieved by Coppersmith and Winograd [7], which requires at most $O(n^{2.376})$ multiplication and addition operations [8]. It also improves on the constant factor in Strassen's algorithm, reducing it to 4.537. These approaches require increasingly sophisticated mathematics and are intimidating reading. Yet, viewed at a high level, the methods all rely on the same framework: for some k, they provide a way to multiply $k \times k$ matrices using $m \lll k^3$ multiplications, and apply the technique recursively to show that the exponent, $\omega < \log_k m$. The main challenge in devising these methods is in the design of the recursion for very large values of k. Moreover, the accuracy of these recursive algorithms are often worse than those of the standard algorithm [9].

Usually, to find out A^k from A, k-1 number of matrix multiplications is required, but the use of doubling trick reduces the number approximately to $\lceil log_2 k \rceil$. For instance, using doubling trick, A^{16} can be computed by 4 matrix multiplications only, $A \rightarrow A^2 \rightarrow A^4 \rightarrow A^8 \rightarrow A^{16}$ and A^{40} can be computed by 6 matrix multiplications only, $A \rightarrow A^2 \rightarrow A^4 \rightarrow A^8 \rightarrow A^{16} \rightarrow A^{32} \rightarrow A^{40}(A^{32} \times A^8)$. Hence, Coppersmith and Winograd's algorithm along with doubling trick necessitates almost $O(\lceil \log_2 k \rceil n^{2.376})$ multiplications.

In this paper, we propose a fast algorithm for finding out the higher powers, k, of a Boolean matrix A that runs in $O(kn^3p)$ addition operations, faster than that of [7] even with doubling trick. We observe that 0 entries in the matrix have no contribution in this computation and therefore we exclude this overhead and develop inference rules using 1 entries. The proposed algorithm selects entries (specified by the inference rule) from the matrix A^{k-1} and adds them up only for updating corresponding entries of A^k. Consequently, our algorithm completely avoids multiplication operations and executes faster. Our second contribution is graph diameter computation algorithm. Let us consider, $G(n, p)$ is a random graph of n vertices in which a pair of vertices appears as an edge with probability p. Diameter of $G(n, p)$ is the longest of the shortest walks in between any two vertices of the graph [10]. In other words, the graph diameter is the maximum number of nodes traversed along an optimal path connecting two arbitrary nodes. Floyd-Warshall's shortest path algorithm [11, 12] gives the shortest paths in between each pair of vertices both for directed and undirected graphs. By finding out the longest one of them, we can solve diameter computation problem.

The time complexity of this algorithm is $O(n^3)$. To the best of our knowledge, there has been no further improvement in computation time required for computing exact diameter of a graph, even though there is rich literature for estimating diameter of random graphs [10, 13, 14, 15, 16]. The proposed graph diameter computation algorithm is much faster, for a massive random graph [14] having diameter d, our algorithm requires $O(n^2(1\text{-}p)E[q])$ operations, where q is the number of attempts required for finding out the first occurrence of 1 in a column.

The rest of the paper is organized as follows. Section 2 introduces the algorithm for computing higher powers of a Boolean matrix and section 3 describes the diameter computation algorithm. Correctness proof and complexity analysis of the algorithms are presented in section 4 and 5 respectively. Section 6 carries out the performance comparisons and the paper is concluded in section 7.

2 Higher Powers of Boolean Matrix

In this section, we describe an algorithm that takes a Boolean matrix A as input and produces the powers of it. The proposed algorithm works as follows. At first, it produces rules (we call it inference rule) for all rows of A. We define an $n \times n$ integer array *Irule*, which stores the column indices having 1 entries in the respective rows of A. The inference rule generated for a particular row is used to update all entries of that row to calculate any matrix A^k from A^{k-1}. Therefore, the procedure **GenerateIrule()** in Algorithm 1 is called only once for a matrix.

Algorithm 1. Inference rule generation

```
Procedure GenerateIrule(A, n, Irule)
/* r_m is the number of 1's in r^th row */
1. for r := 0 to n − 1 do /* for each row, r */
2.     r_m := 0;
3.     for c := 0 to n − 1 do /* for each column, c */
4.         if A(r, c) = 1 then
5.             Irule(r, r_m) = c;
6.             r_m = r_m + 1;
7.         endif
8.     end for
9. end for
```

Once the inference rules for a matrix A is generated, *Irule* contains the column indices having 1 entries in each row. Now, we calculate each entry of higher power matrix $A^2(r,c)$ from A following (1).

$$A^2(r,c) = \sum_{i=0}^{r_m-1} A(Irule(r,i),c) \tag{1}$$

The interesting fact is that the inference rules, we have already developed, to get A^2 from A, can also be applied to get A^3 from A^2, A^4 from A^3 and so on. Hence, each entry of A^k is updated as follows

$$A^k(r,c) = \sum_{i=0}^{r_m-1} A^{k-1}(Irule(r,i),c), \forall k \geq 2 \tag{2}$$

The procedure **BoolMatMul()** presented in Algorithm 2 calculates A^k from A^{k-1}.

Algorithm 2. Boolean matrix multiplication using addition operations only

---

Procedure **BoolMatMul(*A, n, Irule, k*)**
/* r_m is the number of 1's in r^{th} row */
1. **for** $r := 0$ to $n-1$ **do** /* for each row, r */
2. **for** $c := 0$ to $n-1$ **do** /* for each column, c */
3. **for** $m := 0$ to $r_m - 1$ **do**
4. $x := Irule(r,m)$;
5. $A^k(r,c) = A^k(r,c) + A^{k-1}(x,c)$;
6. **end for**
7. **end for**
8. **end for**

At every step, we need to store only the immediate lower powered matrix in memory. The procedure **HigherPower()** in Algorithm 3 shows how it iterates the above steps for each power k of the matrix A to compute A^P, where P is any integer greater than or equal to 2.

Algorithm 3. Calculating P^{th} power of Boolean matrix A

Procedure **HigherPower(*A, n, P*)**
Irule: $n \times n$ integer array, initialized to 0
1. GenerateIrule(*A, n, Irule*);
2. **for** $k := 2$ to P **do** /* Calculates A^P */
3. BoolMatMul(*A, n, Irule, k*);
4. **end for**

The mathematical reasoning behind the proposed algorithm is not so difficult to understand. Its correctness proof is given in section 4.

3 Diameter Computation Algorithm

The adjacency matrix A of $G(n,p)$ is a Boolean matrix with n rows and n columns labeled by graph vertices having entries as follows

$$A(i,j) = \begin{cases} 1, \textit{ if there is an edge from } v_i \textit{ to } v_j \\ 0, \textit{ otherwise} \end{cases} \tag{3}$$

For a simple graph, the adjacency matrix must have 0's on the diagonal. In this section, we propose an algorithm that takes such an adjacency matrix A of a random graph and computes its diameter. For simplicity we consider connected graph here.

The entries of the matrix A^k contain the number of walks of length k between corresponding vertices. For pair of vertices having walk length other than k, or having no path connecting them, entries in A^k become zero. Therefore, if cycles are avoided in computing the walk lengths, only shortest paths are calculated here. Eventually, all entries of A^{d+1}, where $k \geq 1$, become zero if diameter of the graph is d. For solving diameter computation problem, we are concerned only with the shortest paths between pair of vertices. Therefore, we define a matrix A_d whose entries are determined using (4).

$$A_d(i,j) = \begin{cases} 1, \textit{ if dist(i,j)=d} \\ 0, \textit{ otherwise} \end{cases} \tag{4}$$

where *dist(i,j)* denotes the minimum path length between vertices v_i and v_j. We take an auxiliary Boolean matrix $PathFound_d$ of size $n \times n$, which keeps track of the paths that are already found. $PathFound_1$ is initialized to the matrix A and diagonal entries are set to 1. Accordingly, we modify the procedure **BoolMatMul(*k*)** and rename as **FindNewPaths(*d*)** in Algorithm 4. At each step, it finds a path of length d for each pair of vertices v_i and v_j, unless it is already computed or i=j (diagonal entries). Therefore, the entries of $PathFound_d$ are observed as follows

$$PathFound_d(i,j) = \begin{cases} 1, \textit{ if dist(i,j)} \leq d \\ 0, \textit{ otherwise} \end{cases} \tag{5}$$

Algorithm 4. Searching for existence of paths of length d

Function **FindNewPaths(*d*)**
Input: $PathFound_{d-1}(r,c), A_{d-1}$
Return type: Boolean
1. $flag = False$;
2. **for** $r := 0$ to $n-1$ **do** /* for each row, r */
3. $Found = False$;
4. **for** $c := 0$ to $n-1$ **do** /* for each column, c */
5. **if** $r = c$ **OR** $PathFound_{d-1}(r,c) = 1$ **then**

```
6.              Skip operations below and continue with next value of c;
7.          end if
8.          for m := 0 to r_m − 1 do /* each x in inference rule of r */
9.              x := Irule(r, m);
10.             if A_{d−1}(x, c) = 1 then
11.                 A_d(r, c) := 1;
12.                 PathFound_d(r, c) := 1;
13.                 flag := True;
14.                 Found := True;
15.                 Break loop for m;
16.             end if
17.         end for
18.         if Found = True then
19.             Break loop for c;
20.         end if
21.     end for
22. end for
23. return flag;
```

Eventually, for a connected graph, all entries of matrix $PathFound_d$ will become 1 for a certain value of d and **FindNewPaths(*d*+1)** returns false. For a disconnected graph $G(n,p)$, we have used the convention that the diameter of G is the maximum diameter of its connected components. In that case, some 0 entries will still be remaining when **FindNewPaths(*d*+1)** returns false. The function **ComputeDiameter()** in Algorithm 5 repeatedly calls **FindNewPaths(*k*)** until it returns false, which means the diameter is k-1.

Algorithm 5. Computation of diameter of a graph, G

```
Function ComputeDiameter()
Input: Adjacency matrix A of G
Return value: Diameter of G as integer
1. GenerateIrule(A, n, Irule);
2. k := 1;
3. repeat
4.         k := k + 1;
5. untill FindNewPaths(k) = false;
6. return k − 1;
```

4 Proof of Correctness

Theorem 1. *For an $n \times n$ Boolean matrix,* A*, inference rules correctly pick up necessary and sufficient entries to calculate A^k from A^{k-1}, $\forall k \geq 2$.*

Proof: Traditional multiplication algorithm uses (6) to calculate A^k

$$A^k(r,c) = \sum_{x=0}^{x=n-1} A(r,x) \times A^{k-1}(x,c) \tag{6}$$

Here for the entries $A(r,c)$=0, the multiplication as well as the addition operations cannot contribute anything to the sum. Hence our inference rules take only 1 entries of the respective rows. Again since $A(r,x) = 1, \forall x \in Irule(r)$, each multiplication simply yields $A^{k-1}(x,c)$. Therefore, avoiding multiplication operations, the (6) correctly reduces to

$$A^k(r,c) = \sum_{\forall x \in Irule(r)} A^{k-1}(x,c) \tag{7}$$

Lemma 1. *Generating a single set of inference rules is sufficient for calculating any A^k, where $k \geq 2$.*

Proof: Theorem 1 follows that inference rules produced from A picks up the entries from the other matrix, with which A is being multiplied. Any A^k can be generated by adding entries of A^{k-1}. Hence, the set of inference rules created from A can serve all necessary multiplications.

Theorem 2. ***FindNewPaths(d)*** *updates the entries so that $A_d(i,j)$=1 denotes that the length of the shortest path in between vertices v_i and v_j is d.*

Proof: We prove it by induction. $A(i,j)$=1 certainly denotes that the shortest path length between i and j is 1. Now, let us suppose that $A_{d-1}(i,j)$ denotes that the shortest path length between i and j is d-1. The inner loop of **FindNewPaths(*d*)**, statement 9, looks for an x where $A_{d-1}(x,c)$=1. *Irule*(r) guarantees that there is a path of length 1 from r to x. Hence, there exists a path of length d from r to c via x. If there exists a path between i and j whose length is h, where $h < d$, it would have been found earlier when **FindNewPaths(*h*)** was called and the matrix *PathFound* would have kept track of it.

5 Complexity Analysis

Let n be the number of rows or columns of the matrix and p be the probability that an arbitrary entry of the matrix is 1. Then the number of operations required for finding out the inference rules for n rows of the matrix is n^2. If X represents the number of 1's that occur in one row (or column), then X is said to be a binomial random variable with parameters (n,p) [17]. Therefore, the expected number of 1's in a row of the matrix A is given by

$$m = E[X] = \sum_{i=0}^{n} i \binom{n}{i} p^i (1-p)^{n-i} = np \tag{8}$$

5.1 Complexity for Calculating Higher Powers of Boolean Matrix

The proposed algorithm requires m addition operations, given by (8), for calculating a single entry of the next higher-powered matrix A^2. Consequently, the complexity for computing all entries of A^2 is given by $O(n^2 + n \times n \times m) = O(n^2(1+m)) = O(n^2 m)$ (constant part is omitted). By replacing m by np, we get the complexity $O(n^3 p)$ and hence for getting A^k from A requires $O(kn^3 p)$ addition operations. No multiplication is required here.

5.2 Complexity for Diameter Computation

In this case, we update only the 0 entries of the target matrix. For each of the (n-m) 0 entries of a row, the function **FindNewPaths(*d*)** searches for the first occurrence of 1 in m positions. The probabilistic distribution of getting first success (occurrence of 1) in repeated finite number of trials (m) is truncated geometric random [18]. Let q represent the number of attempts required for getting the first 1 in a column. Using [18], we get the conditional probability mass function (pmf) of q, with the condition that the procedure gets a 1, as

$$P(q) = \frac{(1-p)^{q-1}p}{1-(1-p)^m} \tag{9}$$

Replacing the value of m from (8) and using geometric series equations [19], we find the expected number of attempts required, $E[q]$, as in (10).

$$E[q] = \frac{1-(1-p)^{np}(1+np^2)}{p(1-(1-p)^{np})} \tag{10}$$

Consequently, the number of search operations required for updating a single row is (n-m)$E[q]$, which follows that getting A_2 involves n(n-m)$E[q]$ computations. The conditional probability that a single 0 entry of the matrix is updated is given by [18] as follows

$$C_p = \sum_{k=1}^{m} P(q) = \frac{1-(1-p)^{np-1}}{1-(1-p)^{np}} \tag{11}$$

As our diameter computation algorithm finds new paths only, after each step of calculating A_d from A_{d-1}, the number of 0's need to update will be reduced. After the first step, i.e., when A_2 is calculated from A, in each row (n-m)C_p number of 0's is updated and the estimated number of 0's remaining is $(n - m) - (n - m)C_p = (n - m)(1 - C_p)$. Similarly, after the second step, i.e., A_3 is calculated from A_2, $n - m)(1 - C_p)C_p$ number of 0's of each row is updated and the number of 0's left is given by $(n - m)(1 - C_p) - (n - m)(1 - C_p)C_p = (n - m)(1 - C_p)^2$.

Therefore, for a graph with diameter d, the expected number of search operations required by our diameter computation algorithm is

$$n^2 + n(n-m)E[q](1 + (1-C_p) + (1-C_p)^2 + ... + (1-C_p)^d) \tag{12}$$

$$= n^2 + n(n-m)E[q]\frac{1-(1-C_p)^d}{1-(1-C_p)},\ using[18] \tag{13}$$

$$= n^2(1-p)E[q]\acute{C_p},\ where, \acute{C_p} = \frac{1-(1-C_p)^d}{C_p} \tag{14}$$

(14) gives the complexity for diameter computation for any random graph. For massive random graphs, we have seen that the value of $\acute{C_p}$ is computed as even less than $\sqrt{d}$, which is insignificant as compared to n^2. Hence, we can write the complexity of our graph diameter computation algorithm as $O(n^2(1-p)E[q])$.

This complexity is valid for any $p < 1$. For $p = 1$, we need to have a different look. In such a case the matrix will contain 1 entries only. Hence the loop at step 8 of Algorithm 4 will always terminate at the very first iteration, and $FindNewPath(k)$ will be called only once and demonstrates a complexity of $O(n)$. Here, the overall complexity of the proposed diameter computation algorithm will be $O(n^2)$.

The space/time trade-off of the algorithm is stated as follows. The proposed algorithm needs an $n \times n$ array to store the inference rules. However, it may be cut down by using n lists. Then it will need n^2 values to be stored in the worst case. However, this extreme case will occur when $p = 1$, which gives a time complexity of only $O(n^2)$, i.e., a great reduction in computation time complexity is gained with the compensation of space overhead.

6 Performance Comparisons

For performance comparison, we have carried out experiments on Boolean matrices with various densities of 0's and 1's. The experiments are carried out in a Pentium IV PC (2.8GHz, 1GB RAM). The values plotted in all the graphs are mean of 5 individual experiments.

Fig. 1(a) shows the ratio of computation time required by Coppersmith and Winograd's algorithm [7] (T_c) to that of our proposed algorithm (T_p) for computing A^2 from A. As the number of rows (or columns), n, of a matrix increases, the value of $R_t(=\frac{T_c}{T_p})$ slowly increases for all values of p. R_t decreases as p increases. This is because, in our proposed algorithm, the number of addition operations required for updating a single entry of a row is directly proportional to the number of 1's in that row. However, Fig. 1(a) also depicts that even for matrices with all 1 entries ($p = 1.0$), is greater than 1.2, which shows the superiority of the proposed method.

In Fig. 1(b), the computation time ratio R_t is shown for calculating higher powers of A. R_t decreases with higher values of k. This is because of using doubling trick with [7], which reduces the computation time significantly for large values of k. However, even for k=100, R_t is found to be at least 2. Both Fig. 1(b) and Fig. 1(c) show that R_t decreases as p increases, the cause is explained above. Fig. 1(c) also points out that n has very less effect on R_t.

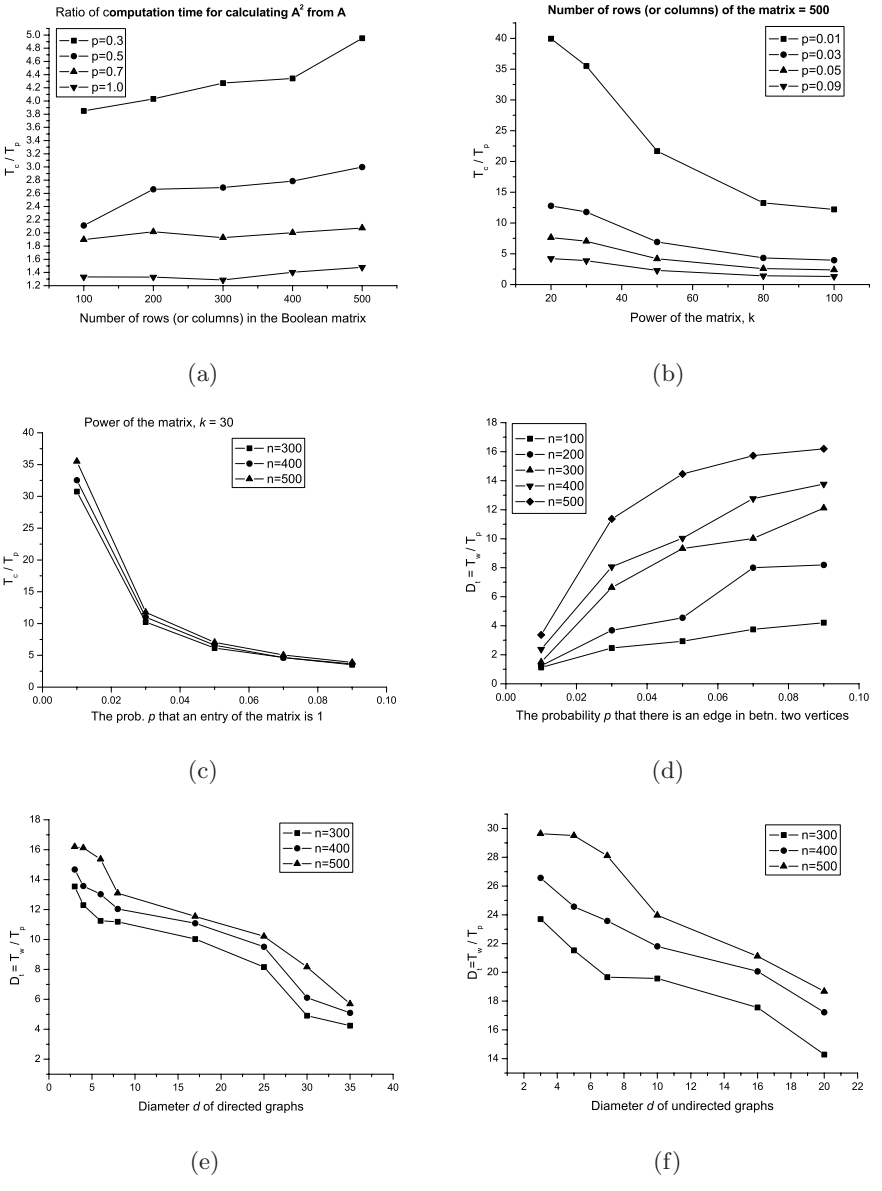

Fig. 1. Ratio of computation time for calculating (a) A^2 from A, (b) A^k from A, (c) A^{30} from A, (d) diameter of directed random graphs (p varies), (e) diameter of directed random graphs (d varies) and (f) diameter of undirected random graphs

The performance of the proposed algorithm may degrade in comparison to [7] for further large values of p and k. However, in natural applications we may seldom require to compute very high power of A. For example, let A represent

a random graph on n vertices, where n=500 and p=0.5. The diameter of this graph is approximately $\frac{\log(n)}{\log(np)}$ [20], which yields only 2. Hence, finding A^2 is sufficient to find relationships in between any pair of vertices of that graph. Again, if the diameter of a graph with 500 vertices is 100, then p should be set to approximately 0.002. This causes each vertex to have degree 1, which is not practical in most applications. Hence we may make comment that finding very high power of matrices representing graphs is not necessary in most of the cases. Therefore, we claim that our proposed method is more applicable to find practically useful powers of matrices.

For practical implementation of our graph diameter computation algorithm and its comparison with Floyd-Warshall's shortest path algorithm [11] that can compute the diameter of a graph exactly, we have randomly placed edges in between vertices ensuring at least one edge to each vertex of the graph. We have taken inputs such that there is no self loop in the graph i.e., the adjacency matrix contains 0 entries in its main diagonal.

Fig. 1(d) and Fig. 1(e) show the ratio of computation time required by Floyd-Warshall's algorithm (T_w) to that of our proposed algorithm (T_p) for computing diameter of directed random graphs. Fig. 4 depicts that the proposed algorithm provides better performance than Floyd-Warshal's algorithm for higher values of p. This is because our algorithm updates 0 entries only and as p increases, the number of 0 entries in the adjacency matrix decreases. Fig. 1(e) shows that D_t drops as d increases. However, it shows that for practical values of diameters the proposed method is still better. It is also observed that the ratio of computation time, $D_t = T_w/T_p$, is higher for larger values of n.

Fig. 1(f) shows that our algorithm provides much better performance for undirected graphs. For all the graphs, the computation time ratio, D_t, is almost double of that for directed graphs as shown in Fig. 1(e). This is because, in case of undirected graphs, we need to deal with only either half part of the main diagonal of the matrix.

7 Conclusions

To the best of the author's knowledge, this is the ever first algorithm for computing higher powers of a matrix that does not require any multiplication operation at all. Our proposed graph diameter computation algorithm neither uses BFS nor dominating sets, needs it only to search the existence of new paths. Our algorithms are easy to implement and have the desired property of being combinatorial in nature and the hidden constants in the running time bound are fairly small.

The applicability of our algorithm, or similar approach, to the problem of diameter computation of weighted graphs, transitive closure and all-pair of shortest paths needs further investigation and analysis. We conjecture that our approach, or simple modification of it, can help solve these problems within $O(n^2(1\text{-}p)E[q])$ operations. We leave this as our future work.

References

1. Auber, D., Delest, M., Domenger, J., Dulucq, S.: Efficient drawing of RNA secondary structure. Journal of Graph Algorithms and Applications 10(2) (2006)
2. Li, N., Guo, Y., Zheng, S., Tian, C., Zheng, J.: A matrix based fast calculation algorithm for estimating network capacity of MANETS. In: Proceedings of the 2005 Systems Communications (ICW), IEEE Computer Society, Los Alamitos (2005)
3. Zhou, M., Cui, Y.: Constructing and visualizing gene relation networks, An International Journal on Computational Molecular Biology (2004) ISBN In Silico Biology, 4, 0026
4. Lipschutz, S.: Data Structures and Algorithm. In: Schaum's outline series, McGraw Hill, New York (2000)
5. Strassen, V.: Gaussian elimination is not optimal. Numerical Mathematics 13, 354–356 (1969)
6. Burgisser, P., Clausen, M., Shokrollahi, M.A.: Algebric complexity theory, Grundlehren der Mathematischen Wissenschaften, vol. 315. Springer, Heidelberg (1997)
7. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. J. of Symbolic Computation 9, 251–280 (1990)
8. Bilardi, G., D'Alberto, P., Nicolau, A.: Fractal matrix multiplication: A case study on probability of cache performance. In: Brodal, G.S., Frigioni, D., Marchetti-Spaccamela, A. (eds.) WAE 2001. LNCS, vol. 2141, pp. 26–38. Springer, Heidelberg (2001)
9. Robinson, S.: Toward an optimal algorithm for matrix multiplication. SIAM news 38(9) (2005)
10. Bollobas, B.: The diameter of random graphs. IEEE Trans. Inform. Theory 36(2), 285–288 (1990)
11. Floyd, Robert, W.: Algorithm 97: shortest path. Communications of the ACM 5(6), 345 (1962)
12. Cormen, T.H., Leiserson, C.E., Ronald, L.R.: Introduction to Algorithms, 2nd edn. The MIT press and McGraw-Hill book company, Cambridge (2001)
13. Bollobas, B.: The Evolution of Sparse Graphs. In: Graph Theory and Combinatorics, pp. 35–57. Academic Press, London-New York (1984)
14. Aiello, W., Chung, F., Lu, L.: Random Evolution of Massive Graphs. In: Handbook of Massive Data Sets, pp. 97–122. Kluwer, Dordrecht (2002)
15. Lu, L.: The diameter of random massive graphs. In: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, pp.912–921. ACM/SIAM Press (2000)
16. Marinari, E., Semerjian, G.: On the number of circuits in random graphs, J. of Statistical Mechanics: Theory and Experiment, P06019 (2006) DOI:10.1088/1742 5468/2006/06/P06019
17. Ross, S.M.: Introduction to Probability Models, 8th edn. Academic Press, London (2002)
18. Bertsekas, D., Gallager, R.: Data Netwroks, ex. 2.13, 2nd edn. Prentice Hall, New Jersey (1992)
19. Yates, R.D., Goodman, D.J.: Probability and Stochastic Processes, B.6 (2005)
20. Chung, F., Lu, L.: The diameter of sparse random graph. J. of Advances in Applied Mathematics 26(4), 257–279 (2001)