

# Fast Algorithm for Optimal Compression of Graphs

Yongwook Choi\*

## Abstract

We consider the problem of finding optimal description for general unlabeled graphs. Given a probability distribution on labeled graphs, we introduced in [4] a *structural entropy* as a lower bound for the lossless compression of such graphs. Specifically, we proved that the structural entropy for the Erdős–Rényi random graph, in which edges are added with probability  $p$ , is  $\binom{n}{2}h(p) - n \log n + O(n)$ , where  $n$  is the number of vertices and  $h(p) = -p \log p - (1-p) \log(1-p)$  is the entropy rate of a conventional memoryless binary source. In this paper, we prove the asymptotic equipartition property for such graphs. Then, we propose a faster compression algorithm that asymptotically achieves the structural entropy up to the first two leading terms *with high probability*. Our algorithm runs in  $O(n + e)$  time on average where  $e$  is the number of edges. To prove its asymptotic optimality, we introduce binary trees that one can classify as in-between tries and digital search trees. We use analytic techniques such as generating functions, Mellin transform, and poissonization to establish our findings. Our experimental results confirm theoretical results and show the usefulness of our algorithm for real-world graphs such as the Internet, biological networks, and social networks.

## 1 Introduction

Brooks argues in [3] that there is “no theory that gives us a metric for information embodied in structure.” Shannon himself alluded to it fifty years earlier in his little known 1953 paper [20]. In fact, Brooks emphasizes the importance of the quantification of information in physical structure. In computer science, however, it is more important to understand the information embodied in abstract structures that are of our particular interests in this paper. For instance, how can we quantify the amount of information in the structure of graphs such as the Internet, social networks, and biological networks? How can we understand and utilize the “structure” of non-conventional data structures such as biological data, topographical maps, medical data, and volumetric data? As the first step to understanding information in such structures, we focus on structure in graphs.

In 1984, Turan [23] raised the question of finding efficient coding method for general unlabeled graphs on  $n$  vertices, suggesting a lower bound of  $\binom{n}{2} - n \log n + O(n)$  bits.<sup>1</sup> In 1990, Naor [15] proposed such a representation that is optimal up to the first two leading terms when all unlabeled graphs are equally likely. In this paper, we solve Turan’s problem for a larger class of graphs, in particular for the Erdős–Rényi random graphs in which edges are added randomly with probability  $p$ . Naor’s result is asymptotically a special case of ours when  $p = 1/2$ .

In [4] we investigated random unlabeled graphs (or random structures) and defined the *structural entropy* as the average of logarithm of the probability of a structure. We showed that the structural entropy  $H_S$  for the Erdős–Rényi random graph  $\mathcal{G}(n, p)$  is

$$\binom{n}{2}h(p) - \log n! + o(1) = \binom{n}{2}h(p) - n \log n + O(n),$$

where  $h(p) = -p \log p - (1-p) \log(1-p)$  is the entropy rate of a conventional memoryless binary source. In this paper, we show the asymptotic equipartition property for structures  $S$  from  $\mathcal{G}(n, p)$ . That is, we prove that, for almost every structure  $S$  from  $\mathcal{G}(n, p)$ , the probability of  $S$  is very close to  $2^{-H_S}$ .

Using Shannon’s argument, it is easy to see that the structural entropy is a lower bound for lossless compression of graphical structures. Over the last decade, substantial effort was devoted to finding efficient algorithms, however without a clearly defined universal lower bound. Most known methods for structural (graph) compression are of heuristic nature. For example, Adler and Mitzenmacher [1] proposed a heuristic method for web graph compression, and similar idea has been used in [21] for compressing sparse graphs. Recently, attention has shifted to grammar compression for data structures: Peshkin [16] proposed an algorithm for a graphical extension of the one-dimensional SEQUITUR compression method. However, SEQUITUR is known not to be asymptotically optimal [18]. Therefore, the Peshkin method already lacks asymptotic optimality in the 1D case. To the best of our knowledge

\*Department of Computer Science, Purdue University, West Lafayette, IN 47907 U.S.A., [ywchoi@purdue.edu](mailto:ywchoi@purdue.edu).

<sup>1</sup>All logarithms are to the base 2 throughout this paper.

there is no provable asymptotically optimal compression scheme for graphical structures.

To fill this gap, we propose here a novel and fast algorithm that asymptotically achieves the established lower bound of the structural compression up to the first two leading terms *with high probability*. It is an improvement from the previous result [4] where we proved the optimality *on average*. In addition, our algorithm runs in  $O(n + e)$  time on average, where  $e$  is the number of edges. This is faster than  $O(n^2)$ -time algorithms (cf. [4, 15]) theoretically as well as in practice since most real-world graphs are very sparse.

The paper is organized as follows. Our algorithm is described in Section 2, where we also provide the structural entropy for  $\mathcal{G}(n, p)$  and our experimental results. The analysis is presented in Section 3, where we introduce random binary trees resembling tries and digital search trees. We use analytic techniques such as generating functions, Mellin transform, and poissonization to establish our results.

## 2 Main Results

In this section, we start with reviewing some facts and definitions from [4] to make the paper self-contained. We formally define the structural entropy of a random (unlabeled) graph model. Then, we compute the structural entropy for the Erdős–Rényi graph and describe our optimal compression algorithm. Finally, we present experimental results to show the efficiency and utility of our algorithm.

**2.1 Structural Entropy.** Given  $n$  (distinguishable) vertices, a random graph is generated by adding edges randomly. This random graph model  $\mathcal{G}$  produces a probability distribution on graphs, and the graph entropy  $H_{\mathcal{G}}$  is defined naturally as

$$H_{\mathcal{G}} = \mathbf{E}[-\log P(G)] = - \sum_{G \in \mathcal{G}} P(G) \log P(G),$$

where  $P(G)$  is the probability of a graph  $G$ . To investigate structural entropy, we introduce a *random structure model*  $\mathcal{S}$  for the unlabeled version of a random graph model  $\mathcal{G}$ . In such a model, graphs are generated in the same manner as in  $\mathcal{G}$ , but they are thought of as unlabeled graphs. That is, the vertices are indistinguishable, and the graphs having “the same structure” are considered to be the same even if their labeled versions are different. Thus, we shall use the terms *unlabeled graphs* and *structures* interchangeably. For a given structure  $S \in \mathcal{S}$ , the probability of  $S$  can be computed as  $P(S) = \sum_{G \cong S, G \in \mathcal{G}} P(G)$ . Here  $G \cong S$  means that  $G$  and  $S$  have the same structure, that is,  $S$  is *isomorphic* to  $G$ . If all isomorphic labeled graphs

have the same probability, then for any labeled graph  $G \cong S$ ,  $P(S) = N(S) \cdot P(G)$ , where  $N(S)$  is the number of different labeled graphs that have the same structure as  $S$ . The *structural entropy*  $H_{\mathcal{S}}$  of a random graph  $\mathcal{G}$  can be defined as the entropy of a random structure  $\mathcal{S}$ ,

$$H_{\mathcal{S}} = \mathbf{E}[-\log P(S)] = - \sum_{S \in \mathcal{S}} P(S) \log P(S),$$

where the summation is over all distinct structures.

**Example:** In Figure 1, we have all labeled graphs built on three vertices, and we also present all structures that can be generated by  $\mathcal{S}$  with  $N(S_1) = N(S_4) = 1$  and  $N(S_2) = N(S_3) = 3$ . ■

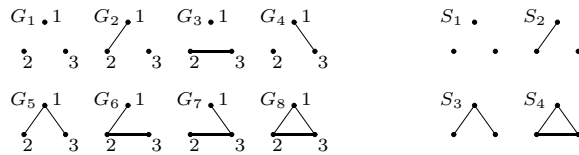


Figure 1: All different graphs (left) and structures (right) with three vertices

To estimate the number of ways to construct a given structure  $S$ , denoted as  $N(S)$ , we need to consider the automorphisms of a graph. An *automorphism* of a graph  $G$  is an adjacency preserving permutation of vertices of  $G$ . The collection  $\text{Aut}(G)$  of all automorphisms of  $G$  is called *the automorphism group* of  $G$ . In the sequel,  $\text{Aut}(S)$  of a structure  $S$  denotes  $\text{Aut}(G)$  for some labeled graph  $G$  such that  $G \cong S$ . In group theory, it is well known that  $N(S) = n! / |\text{Aut}(S)|$  [11]. With these definitions, we proved in [4] that

$$H_{\mathcal{S}} = H_{\mathcal{G}} - \log n! + \sum_{S \in \mathcal{S}} P(S) \log |\text{Aut}(S)|,$$

for any random graph  $\mathcal{G}$  and its corresponding random structure  $\mathcal{S}$ .

To develop further the idea of structural entropy, we focus on the Erdős–Rényi random graph [2]. In this model  $\mathcal{G}(n, p)$ , graphs are generated on  $n$  vertices with edges added independently with probability  $p$ . Let  $\mathcal{S}(n, p)$  be the random structure model corresponding to  $\mathcal{G}(n, p)$ . We compute the structural entropy and prove the asymptotic equipartition property (AEP), that is, typical probability of a structure  $S$ . This structural entropy is a fundamental lower bound on the lossless compression of structures from  $\mathcal{S}(n, p)$  by Shannon’s source coding theorem. In the sequel, we write  $a_n \ll b_n$  to mean  $a_n = o(b_n)$  when  $n \rightarrow \infty$ .

**THEOREM 2.1.** *For large  $n$  and all  $p$  satisfying  $\frac{\ln n}{n} \ll p$  and  $1 - p \gg \frac{\ln n}{n}$ , the following holds:*

(i) The structural entropy  $H_S$  of  $\mathcal{G}(n, p)$  is

$$H_S = \binom{n}{2} h - \log n! + o(1)$$

$$= \binom{n}{2} h(p) - n \log n + n \log e - \frac{1}{2} \log n - \frac{1}{2} \log(2\pi) + o(1),$$

where  $h(p) = -p \log p - (1-p) \log(1-p)$ .

(ii) (AEP) For a structure  $S \in \mathcal{S}(n, p)$ ,

$$P \left( \left| -\frac{1}{\binom{n}{2}} \log P(S) - h(p) + \frac{\log n!}{\binom{n}{2}} \right| < \epsilon \right) > 1 - 2\epsilon. \quad (2.1)$$

*Proof.* The part (i) is already established in [4]. The key observation is the *asymmetry* of the Erdős–Rényi graphs (for almost every graph  $G$  from  $\mathcal{G}(n, p)$ , the identity is the only automorphism of  $G$ , that is,  $|\text{Aut}(G)| = 1$ .) To prove the part (ii), we define the *typical set*  $T_\epsilon^n$  as the set of structures  $S$  on  $n$  vertices with the following two properties: (a)  $S$  is asymmetric; (b) for  $G \cong S$ ,

$$2^{-\binom{n}{2}(h(p)+\epsilon)} \leq P(G) \leq 2^{-\binom{n}{2}(h(p)-\epsilon)}.$$

Let  $T_1$  and  $T_2$  be the sets of structures satisfying the properties (a) and (b), respectively. Then,  $T_\epsilon^n = T_1 \cap T_2$ . By the asymmetry of  $\mathcal{G}(n, p)$ , we know that  $P(T_1) > 1 - \epsilon$ , for large  $n$ . There are  $\binom{n}{2}$  distinct potential edges in  $\mathcal{G}(n, p)$ . That is, a labeled graph  $G$  can be viewed as a binary sequence of length  $\binom{n}{2}$ . Thus, by the property (b) and the AEP for binary sequences, we also know that  $P(T_2) > 1 - \epsilon$ , for large  $n$ . Thus,  $P(T_\epsilon^n) = 1 - P(\overline{T_1} \cup \overline{T_2}) > 1 - 2\epsilon$ . Now let us compute  $P(S)$  for  $S$  in  $T_\epsilon^n$ . By the property (a),  $P(S) = n!P(G)$  for any  $G \cong S$ . By this and the property (b), we can see that any structure  $S$  in  $T_\epsilon^n$  satisfies the condition in (2.1). This completes the proof.

**2.2 Compression Algorithm.** Our algorithm is a compression scheme for unlabeled graphs. In other words, given a labeled graph  $G$ , our algorithm compresses  $G$  into a code, from which one can construct a graph  $G'$  that is isomorphic to  $G$ . Our algorithm is a two-stage algorithm. For a given graph  $G$ , the algorithm first encodes  $G$  into two binary sequences and then compresses them using an arithmetic encoder. Given a graph from  $\mathcal{G}(n, p)$ , our algorithm achieves the structural entropy up to the first two leading terms with high probability as shown in the theorem below, proved in Section 3.

**THEOREM 2.2.** *Let  $L(S)$  be the length of the code generated by our algorithm for all graphs  $G$  from  $\mathcal{G}(n, p)$  that*

*are isomorphic to a structure  $S$ . The following holds:*

(i) For large  $n$ ,

$$\mathbf{E}[L(S)] \leq \binom{n}{2} h - n \log n + (c + \Phi(\log n)) n + o(n),$$

where  $h := h(p)$ ,  $c$  is an explicitly computable constant, and  $\Phi(\log n)$  is a fluctuating function with a small amplitude.

(ii) Furthermore, for any  $\epsilon > 0$ ,

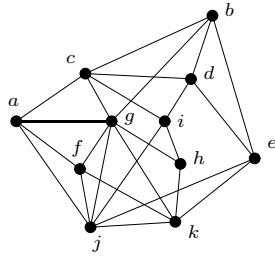
$$P(L(S) - \mathbf{E}[L(S)] \leq \epsilon n \log n) \geq 1 - o(1).$$

(iii) Finally, our algorithm runs in  $O(n+e)$  on average, where  $e$  is the number of edges.

We next describe the algorithm in some details, starting with a general framework and then proposing some useful data structures that allow us to reduce the time complexity to  $O(n+e)$ .

**2.2.1 General Framework.** First we need some definitions and notations. An *ordered partition* of a set  $X$  is a sequence of nonempty subsets of  $X$  such that every element in  $X$  is in exactly one of these subsets. For example, one ordered partition of  $\{a, b, c, d, e\}$  is  $\{a, b\}, \{e\}, \{c, d\}$  that is denoted by  $ab/e/cd$ . It is equivalent to  $ba/e/dc$ , but distinct from  $e/ab/cd$ . Given an ordered partition  $P$  of a set  $X$ , we also define an order on the elements of  $X$  as follows:  $a < b$  in  $P$  if the subset containing  $a$  precedes the subset containing  $b$  in  $P$ . For example,  $a < c$  and  $e < c$  in  $P = ab/e/cd$ , but  $e \not< a$ . An ordered partition  $P_1$  of a set  $X$  is called *finer* than ordered partition  $P_2$  of  $X$  if the following two conditions hold: (1) every element (i.e., subset of  $X$ ) of  $P_1$  is a subset of some element of  $P_2$ , and (2) for all  $a, b \in X$ ,  $a < b$  in  $P_1$  if  $a < b$  in  $P_2$ . For example, both  $a/b/e/cd$  and  $ab/e/d/c$  are finer than  $ab/e/cd$ . Finally, a subtraction of an element from an ordered partition gives us another ordered partition (e.g., for  $P = ab/e/cd$  we find that  $P-c$  and  $P-e$  are  $ab/e/d$  and  $ab/cd$ , respectively).

The first stage of our algorithm consists of  $n$  steps, and one ordered partition  $P$  of a subset of  $V(G)$  is maintained. Let  $P_i$  be the partition after the  $i$ -th step. At the beginning,  $P_0 = V(G)$ . In the  $i$ -th step, any one vertex  $v$  is chosen to be removed from the first subset in  $P_{i-1}$ . Then, for each subset  $U$  in  $P_{i-1} - v$  (in its order), we encode the *number of neighbors* of  $v$  in  $U$  using  $\lceil \log(|U| + 1) \rceil$  bits. After that,  $P_{i-1} - v$  becomes a finer partition  $P_i$  such that for each subset  $U$  in  $P_{i-1} - v$ ,  $U$  is divided into two smaller subsets  $U_1$  and  $U_2$ , and  $U_1$  precedes  $U_2$  in  $P_i$  where  $U_1$  is the set of all neighbors of  $v$  in  $U$  and  $U_2$  is the set of all non-neighbors of  $v$  in  $U$ . These steps are repeated until  $P$  becomes empty.



$\ell$	$v$	$P_{\ell-1} - v$	encoding	$P_\ell$
0				<i>abcdefghijkl</i>
1	<i>a</i>	<i>bcdefghijk</i>	<i>0100</i>	<i>cfgj/bdehik</i>
2	<i>c</i>	<i>fgj/bdehik</i>	<i>01, 011</i>	<i>g/fj/bdi/ehk</i>
3	<i>g</i>	<i>fj/bdi/ehk</i>	<i>10, 01, 10</i>	<i>fj/b/di/hk/e</i>
4	<i>f</i>	<i>j/b/di/hk/e</i>	<i>1, 0, 00, 01, 0</i>	<i>j/b/di/k/h/e</i>
5	<i>j</i>	<i>b/di/k/h/e</i>	<i>0, 01, 1, 0, 1</i>	<i>b/i/d/k/h/e</i>
6	<i>b</i>	<i>i/d/k/h/e</i>	<i>0, 1, 0, 0, 1</i>	<i>i/d/k/h/e</i>
7	<i>i</i>	<i>d/k/h/e</i>	<i>1, 0, 1, 0</i>	<i>d/k/h/e</i>
8	<i>d</i>	<i>k/h/e</i>	<i>0, 0, 1</i>	<i>k/h/e</i>
9	<i>k</i>	<i>h/e</i>	<i>1, 1</i>	<i>h/e</i>
10	<i>h</i>	<i>e</i>	<i>0</i>	<i>e</i>
11	<i>e</i>			

Figure 2: An example for our algorithm, given the graph on the left

While the algorithm is running, the binary encodings of the number of neighbors are concatenated in the order they are generated. During the course of the algorithm, we separately maintain two types of encodings - those of length more than one bits (i.e., for subsets  $|U| > 1$ ) and those of length exactly one bit (i.e., for subsets  $|U| = 1$ ). The former type of encodings are appended to a binary sequence  $B_1$ . Similarly, the latter type of encodings form a binary sequence  $B_2$ .

**Example:** Figure 2 shows the progress of our algorithm step by step. Here  $\ell$  denotes the step number, and  $v$  denotes the chosen vertex in each step. All encodings whose length is larger than one (denoted by *italic font*) are appended to  $B_1$ . The other encodings (those of length one) form  $B_2$ . After eleven steps,  $B_1$  and  $B_2$  are *010001011100110000101* and *1000101010011010001110*, respectively. ■

In the second stage,  $B_1$  and  $B_2$  are compressed to  $\hat{B}_1$  and  $\hat{B}_2$  by a binary arithmetic encoder [6]. Thus, the encoding of  $G$  consists of  $n$ ,  $\hat{B}_1$ , and  $\hat{B}_2$ . The general framework of decoding algorithm is very similar to the above (cf. [4]).

In a naive implementation of this general framework, the time complexity is  $O(n^2)$  as follows. In each step of the first stage, we need to count the number of neighbors in each disjoint subset in  $P$  and split it into two smaller subsets. This can be done in  $O(n)$  time by scanning all remaining vertices in  $P$ . Thus the first stage takes  $O(n^2)$  time in total. In the second stage, a linear-time arithmetic encoder takes  $O(n^2)$  time since the length of  $B_2$  is  $\Theta(n^2)$ , which will be proved in Section 3.

We use the following three novel techniques to reduce the time complexity to  $O(n + e)$  on average. First, we use efficient data structures to maintain the partition  $P$  and encode the number of neighbors in each subset. Second, in the arithmetic encoding, we process the intermediate sequence  $B_2$  not in bitwise manner, but instead we process a run of consecutive zeroes in one

step. Third, when outputting the code in the arithmetic encoder, we use a greedy outputting method proposed in [14].

**2.2.2 Data structures.** To describe our data structure, we define the *position* of a vertex  $v$  in a partition  $P$  as the number of vertices on the right side of  $v$  in  $P$ . Similarly, we define the *rank* of a subset  $U$  and all vertices  $v \in U$  as the number of vertices on the right side of  $U$  in  $P$  (i.e., the position of the rightmost vertex in  $U$ ).

The partition  $P$  of a subset of  $V(G)$  is maintained by the following five arrays, each of which is of size  $n$ . Firstly,  $pos[v]$  and  $rank[v]$  store the position and the rank of a vertex  $v$  in  $P$ , respectively. An array  $vid[i]$  stores the vertex at position  $i$  (i.e.,  $pos[vid[i]] = i$ ). An array  $size[r]$  stores the size of a subset whose rank is  $r$ . Lastly, for  $r$  such that  $size[r] > 1$ ,  $next[r]$  stores the largest rank  $r'$  such that  $r > r'$  and  $size[r'] > 1$ . We also have a variable *head* containing the largest rank  $r$  such that  $size[r] > 1$ . These arrays are updated while  $P$  becomes smaller and finer in each step. For example, when  $P = P_3 = fj/b/di/hk/e$  in our previous example, the arrays are as follows.

	k	j	i	h	g	f	e	d	c	b	a	
pos	1	6	3	2	-	7	0	4	-	5	-	
rank	1	6	3	1	-	6	0	3	-	5	-	
vid	10	9	8	7	6	5	4	3	2	1	0	
size	-	-	-	-	f	j	b	d	i	h	k	e
size	0	0	0	0	2	1	0	2	0	2	1	
next	-	-	-	-	3	-	-	1	-	-	-	

We observe the following properties: (1) The vertices with the same rank are in the same subset in  $P$ ; (2) The division of a subset  $U$  does not affect the ranks of vertices outside  $U$  (it affects only the rank of vertices in  $U$  that are neighbors of the chosen vertex); (3) Once the size of a subset becomes one, its rank is the same as its position and does not change until the end; (4) Using *head* and *next*, one can traverse only the subsets

whose size is larger than one.

**2.2.3 Algorithm.** Now we describe our algorithm in more detail. The first stage consists of  $n$  steps. Let  $P_i$  be the ordered partition after the  $i$ -th step, which is maintained implicitly by the arrays described previously. Here we assume that the input graph is given as an adjacency list and  $N(v)$  denotes the list of neighbors of a vertex  $v$ . We also have a temporary array  $\mathcal{B}$  of size  $n$ , which stores a stack in each element. An array *count* is used for counting the number of neighbors. In the  $i$ -th step, the algorithm works as follows:

1. Remove the leftmost vertex  $v$  from  $P_{i-1}$  and update arrays accordingly.
2. While traversing  $N(v)$ , for each neighbor  $u$  that is still in  $P_{i-1}$  (let  $r = \text{rank}[u]$ ),
  - 2.1. If  $\text{size}[r] > 1$ , increase  $\text{count}[r]$  by one.
  - 2.2. If  $\text{size}[r] = 1$ , mark its position by pushing the step number  $i$  to the stack in  $\mathcal{B}[r]$ .
3. While traversing subsets  $U$  such that  $|U| > 1$  using *head* and *next* (let  $r$  be the rank of  $U$ ),
  - 3.1. Encode the number of neighbors in  $U$  (stored in  $\text{count}[r]$ ) using  $\lceil \log(\text{size}[r] + 1) \rceil$  bits, and output to  $B_1$ .
  - 3.2. Mark the position of  $U$  (i.e., the positions of both ends of  $U$ ) by pushing  $-i$  to the stacks in  $\mathcal{B}[r]$  and  $\mathcal{B}[r + \text{size}[r] - 1]$ .
  - 3.3. Update *size* and *next* arrays accordingly reflecting the division of  $U$ .
4. While traversing  $N(v)$ , for each neighbor  $u$  that is still in  $P_{i-1}$ ,
  - 4.1. Update the rank of  $u$ .
  - 4.2. Move  $u$  to the right position by updating *pos* and *vid* (i.e., swap  $u$  and the vertex at the position which  $u$  moves to.)

After repeating the above steps until  $P$  becomes empty, we extract  $B_2$  from  $\mathcal{B}$  as a form of run length codes – a sequence of the length of the run of zeroes between each two consecutive ‘1’s (including both ends). For example,  $B_2 = 1000101010011010001110$  is encoded as 0,3,1,1,2,0,1,3,0,0,1. The time complexity of the construction of  $B_2$  is analyzed in the following lemma, which will be used later to analyze the overall time complexity of our algorithm.

LEMMA 2.1.  $B_2$  can be constructed by scanning  $\mathcal{B}$  once, and it takes  $O(n + \ell)$  time, where  $\ell$  is the total number of elements inserted in  $\mathcal{B}$ .

*Proof.* In the  $i$ -th step, there are  $n - i$  vertices in  $P$ , and their positions are from 0 to  $n - i - 1$ . In procedure 3.2,

the position of each subset of size larger than one in  $P$  is marked. Thus, one can infer the number of subsets of size one (i.e., singleton sets) in  $P$  and the positions of those subsets. In procedure 2.2, the position of each singleton set containing a neighbor is marked. Each of these marked positions contributes a bit ‘1’ and each of the rest contributes a bit ‘0’. Thus the concatenation  $c_i$  of the bits in decreasing order of position is the contribution to  $B_2$  in the  $i$ -th step. Therefore,  $B_2$  is nothing but  $c_1 c_2 \cdots c_{n-1}$ . Clearly, for each  $c_i$ , the number of zeroes between each two consecutive ‘1’s can be computed in one scan of  $\mathcal{B}$ . This can be done for all  $i$ ’s in parallel. After that, the concatenation of  $c_i$ ’s takes  $O(n)$  time.

In the second stage, both  $B_1$  and  $B_2$  are compressed by a binary arithmetic encoder, but  $B_2$  is compressed by a modified arithmetic encoder using the greedy outputting method as described in [14]. We first briefly describe a general (non-adaptive) binary arithmetic encoder and then describe our modified arithmetic encoder. Given a probability  $p$  for a bit ‘1’, the encoder starts with an initial interval  $[0, N)$  where  $N$  is a large positive integer. For each bit, it first calculates the new interval from the current interval. Then, it outputs code bits from the newly calculated interval so that its length is greater than a predefined threshold. If we use this encoder, the complexity would be  $O(n^2)$  since the length of  $B_2$  is  $\Theta(n^2)$  in bits. Thus, in the modified encoder, we process a run of zeroes in one step. When we extract  $B_2$  from  $\mathcal{B}$ , we estimate the probability  $p$  of having ‘1’ in  $B_2$  and also precompute in a table the probability of a run of  $k$  zeroes, which is  $(1 - p)^k$  for  $k = 1, 2, \dots$ . We recall that  $B_2$  stores lengths of runs of zeroes. When the encoder gets a number from  $B_2$ , it calculates the new interval for a run of zeroes in constant time by looking up this precomputed table. After this, it outputs code bits in constant time using the greedy outputting method in [14]. After that, it processes a bit ‘1’ in usual way. Here we need one restriction on  $k$  since for very large  $k$  the probability  $(1 - p)^k$  becomes too small to represent the new interval precisely. Thus, we set  $k_{max} = \lceil 1/p \rceil$ , and if  $k > k_{max}$ , then we process only the first  $k_{max}$  zeroes in every step until all  $k$  zeroes are exhausted.

**2.3 Experimental Results.** To test our algorithm, we applied it to the Erdős–Rényi random graphs and real-world networks including biological, social, and technological networks. Table 1 summarizes the results for the real-world networks. For comparison, we list the lengths of three other encodings of graphs, namely, the usual implementations of adjacency matrix of  $\binom{n}{2}$  bits, and adjacency list of at least  $e \lceil \log n \rceil$  bits (nor-

Table 1: The average code length and running time for real-world networks

Networks	# of nodes	# of edges	Code length (bits)				CPU time (secs)	
			our algo.	adj. mat. $\binom{n}{2}$	adj. list $e \lceil \log n \rceil$	arithmetic coding	$O(n+e)$ algo.	$O(n^2)$ algo.
US Airports	332	2,126	8,108	54,946	19,134	12,947	<0.01	<0.01
Protein interaction(Yeast)	2,329	6,646	46,853	2,785,980	79,752	67,063	0.11	0.12
Collaboration(Geometry)	6,167	21,535	113,684	19,012,861	279,955	241,549	0.47	0.68
Collaboration(Erdős)	6,934	11,857	60,263	24,043,645	154,141	147,121	1.02	1.08
Genetic interaction(Human)	8,595	26,066	221,226	37,018,710	364,924	310,459	1.22	1.54
Internet(AS level)	25,881	52,407	301,463	334,900,140	786,105	737,851	12.97	13.81

mally,  $2e \lceil \log n \rceil$  bits) where  $e$  is the number of edges. Finally, we applied an arithmetic encoder to the adjacency matrix, which can achieve  $\binom{n}{2} h(p)$  bits. For many real-world networks, our algorithm achieves twice better compression than the standard arithmetic encoder. For comparison, we also implemented  $O(n^2)$ -time algorithm in [4]. For all of the real-world test data, our  $O(n+e)$ -time algorithm is faster than  $O(n^2)$ -time algorithm. We measured CPU time on a machine equipped with Pentium D 3.0GHz processor and 2GB of RAM, running Linux. All the numbers are averages over 100 measurements.

Figure 3 shows the results for  $\mathcal{G}(n, p)$  graphs. In (a) and (b), we plot the gain of our encoding against arithmetic encoding, that is, the difference between two encodings. We plot it for a fixed  $p$  in (a) and for a fixed  $n$  in (b). The plots confirm our analysis that the gain is asymptotically close to  $n \log n$ . In (c) and (d), we plot the CPU time consumed by  $O(n+e)$ -time and  $O(n^2)$ -time algorithms, and it shows that  $O(n+e)$ -time algorithm is faster unless the graph is too dense.

### 3 Analysis

In this section, we analyze the compression performance and time complexity of our algorithm, proving Theorem 2.2.

**3.1 Binary Trees  $T_n$  and  $T_{n,d}$ .** To analyze our algorithm, we conveniently introduce a binary tree that better captures the progress of the algorithm. Given a graph  $G$  on  $n$  vertices, the binary tree  $T_n$  is built as follows. At the beginning, the root node contains all  $n$  graph vertices,  $V(G)$ , that one can also visualize as  $n$  balls. Then a graph vertex (ball)  $v$  is removed from the root node. The other  $n-1$  graph vertices move down to the left or right depending whether they are adjacent vertices in  $G$  to  $v$  or not; adjacent vertices go to the left child node and the others go to the right child. We create a new child node in  $T_n$  if there is at least one graph vertex in that node. At this point, the tree is of height 1 with  $n-1$  vertices in the nodes at level 1.

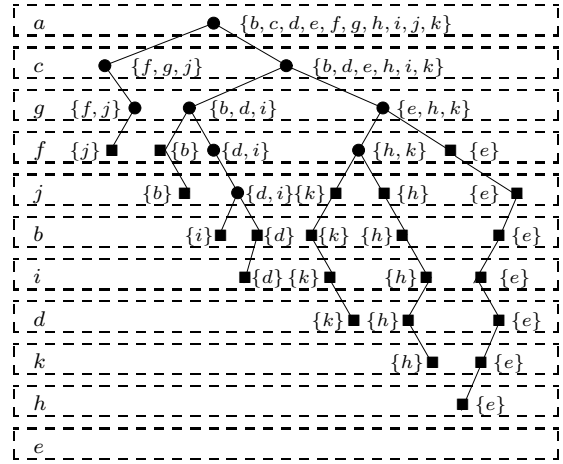
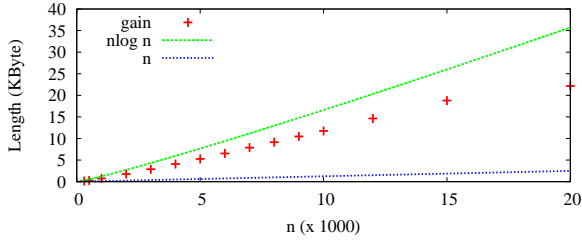


Figure 4: A binary tree  $T_n$

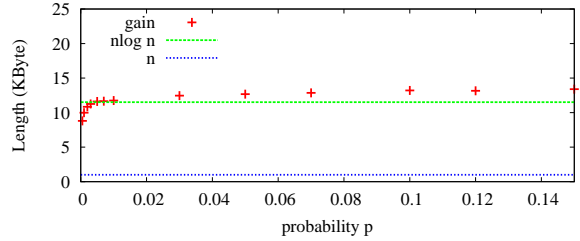
Similarly, in the  $i$ -th step, we remove one graph vertex (ball)  $v$  from the (level-wise) leftmost node at level  $i-1$ . The other graph vertices at level  $i-1$  move down to the left or right depending whether they are adjacent to  $v$  or not. We repeat these steps until all graph vertices are removed (i.e., after  $n$  steps).

Given the example graph in Figure 2, the construction of the tree  $T_n$  and the progress of the algorithm are presented in Figure 4. The selected graph vertices are shown on the left. At each level, the subsets of graph vertices (after removing the chosen vertex) are shown next to the nodes. In this example, the same vertices are selected as in Figure 2. We observe that the subsets at each level (from left to right) are the same as the subsets in each step of our algorithm in Figure 2.

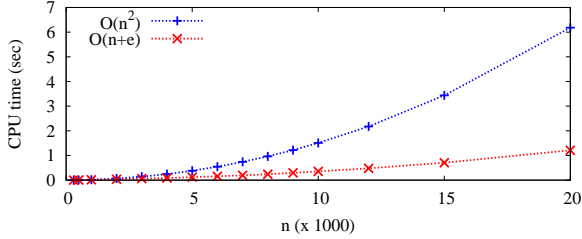
Let  $N_x$  denote the number of graph vertices that pass through node  $x$  in  $T_n$  (excluding the graph vertex removed at  $x$ , if any). In Figure 4, for example,  $N_x$  is the number of graph vertices shown next to the node  $x$ . Our algorithm needs to encode, for each node  $x$  in  $T_n$ , the number of neighbors (of the removed graph vertex) among  $N_x$  vertices. This requires  $\lceil \log(N_x + 1) \rceil$  bits. Let  $L(B_1)$  and  $L(B_2)$  be the lengths of  $B_1$  and  $B_2$ ,



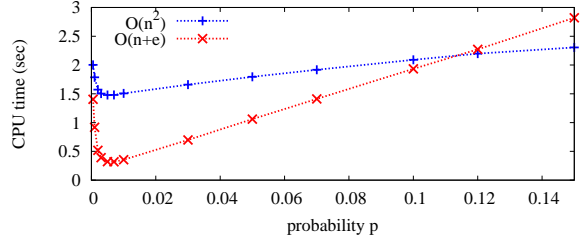
(a) gain in code length against arithmetic coding ( $p=0.01$ )



(b) gain in code length against arithmetic coding ( $n=10000$ )



(c) running time ( $p=0.01$ )



(d) running time ( $n=10000$ )

Figure 3: The average gain in code length and running time for the Erdős-Rényi random graphs.

respectively. Then, by the construction

$$L(B_1) = \sum_{x \in T_n \text{ and } N_x > 1} \lceil \log(N_x + 1) \rceil,$$

and

$$L(B_2) = \sum_{x \in T_n \text{ and } N_x = 1} \lceil \log(N_x + 1) \rceil = \sum_{x \in T_n \text{ and } N_x = 1} 1.$$

In Figure 4, the summations for  $L(B_1)$  and  $L(B_2)$  are over all circle-shaped nodes and over all square-shaped nodes, respectively. Here we can observe an important property of  $B_2$ . That is, given a graph from  $\mathcal{G}(n, p)$ , the sequence  $B_2$  constructed by our algorithm is probabilistically equivalent to a binary sequence generated by a memoryless source( $p$ ) with  $p$  being the probability of generating a ‘1’.

To set up precise recurrence relations for our analysis, we need to define a random binary tree  $T_{n,d}$  for integers  $n \geq 0$  and  $d \geq 0$ , which is generated similarly to  $T_n$  as follows. If  $n = 0$ , then it is just an empty tree. For  $n > 0$ , we create a root node, in which we put  $n$  balls. In each step, all balls independently move down to the left (with probability  $p$ ) or right (with probability  $1 - p$ ). We create a new node if there is at least one ball in that node. Thus, after  $i$ -th step, the balls will be at level  $i$ . If the balls are at level  $d$  or greater, then we remove one ball from the leftmost node before the balls move down to the next level. These steps are repeated until all balls are removed (i.e., after  $n + d$  steps.) We observe that, if  $T_n$  is generated by a graph from  $\mathcal{S}(n, p)$ ,

$T_n$  is nothing but the random binary tree  $T_{n,0}$ . Thus, by analyzing  $T_{n,0}$ , we can compute both  $L(B_1)$  and  $L(B_2)$ .

**3.2 Average Performance.** In this section, we prove part (i) of our main result, that is, we derive the average length of the compressed string representing graphical structure.

Let us first estimate  $L(B_1)$ . As before,  $N_x$  denotes the number of balls that pass through node  $x$  (excluding the ball removed at  $x$  if any). Let

$$A_{n,d} = \sum_{x \in T_{n,d} \text{ and } N_x > 1} \lceil \log(N_x + 1) \rceil,$$

and  $a_{n,d} = \mathbf{E}[A_{n,d}]$ . Then  $\mathbf{E}[L(B_1)] = a_{n,0}$ . Clearly,  $a_{0,d} = a_{1,d} = 0$  and  $a_{2,0} = 0$ . For  $n \geq 2$  and  $d = 0$ , we observe that

$$a_{n+1,0} = \lceil \log(n + 1) \rceil + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (a_{k,0} + a_{n-k,k}). \quad (3.2)$$

This follows from the fact that starting with  $n + 1$  balls in the root node, and removing one ball we are left with  $n$  balls passing through the root node. This contributes  $\lceil \log(n + 1) \rceil$ . Then, those  $n$  balls move down to the left or right subtrees. Let us assume  $k$  balls move down to the left subtree (the other  $n - k$  balls must move down to the right subtree, and this happens with probability  $\binom{n}{k} p^k q^{n-k}$ .) At level one, one ball is removed from those  $k$  balls in the root of the left subtree. This contributes  $a_{k,0}$ . There will be no removal among  $n - k$  balls in the right subtree until all  $k$  balls in the left subtree are

removed. This contributes  $a_{n-k,k}$ . Similarly, for  $d > 0$ , we can see that

$$(3.3) \quad a_{n,d} = \lceil \log(n+1) \rceil + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (a_{k,d-1} + a_{n-k,k+d-1}).$$

This recurrence is quite complex, but we only need a good upper bound that is presented in the next lemma.

LEMMA 3.1. *For all integers  $n \geq 0$  and  $d \geq 0$ ,*

$$a_{n,d} \leq x_n$$

such that  $x_n$  satisfies  $x_0 = x_1 = 0$  and for  $n \geq 2$

$$(3.4) \quad x_n = \lceil \log(n+1) \rceil + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (x_k + x_{n-k}).$$

*Proof.* We use induction on both  $n$  and  $d$ . Clearly,  $a_{n,d} \leq x_n$  for  $n = 0$  or  $1$  ( $d \geq 0$ ). For  $n = 2$  and  $d = 0$ ,  $a_{2,0} \leq x_2$  since  $a_{2,0} = 0$  and  $x_2 \geq 2$ . For other cases ( $n = 2$  and  $d > 0$ , or  $n > 2$ ), we assume that  $a_{i,j} \leq x_i$  holds for  $i < n$ , and for  $i = n$  and  $j < d$ . Now we want to show that  $a_{n,d} \leq x_n$ . We divide it into two cases.

(i) When  $d = 0$ . We observe that

$$\begin{aligned} a_{n,0} &\leq a_{n+1,0} = \lceil \log(n+1) \rceil \\ &+ \sum_{k=1}^{n-1} \binom{n}{k} p^k q^{n-k} (a_{k,0} + a_{n-k,k}) + q^n a_{n,0} + p^n a_{n,0}. \end{aligned}$$

Thus,

$$(3.5) \quad \begin{aligned} &(1 - p^n - q^n) a_{n,0} \\ &\leq \lceil \log(n+1) \rceil + \sum_{k=1}^{n-1} \binom{n}{k} p^k q^{n-k} (a_{k,0} + a_{n-k,k}). \end{aligned}$$

Similarly, from (3.4), we get

$$(3.6) \quad \begin{aligned} &(1 - p^n - q^n) x_n \\ &= \lceil \log(n+1) \rceil + \sum_{k=1}^{n-1} \binom{n}{k} p^k q^{n-k} (x_k + x_{n-k}). \end{aligned}$$

Therefore,

$$\begin{aligned} &(1 - p^n - q^n) a_{n,0} && \text{(by (3.5))} \\ &\leq \lceil \log(n+1) \rceil + \sum_{k=1}^{n-1} \binom{n}{k} p^k q^{n-k} (a_{k,0} + a_{n-k,k}) \\ &\leq \lceil \log(n+1) \rceil + \sum_{k=1}^{n-1} \binom{n}{k} p^k q^{n-k} (x_k + x_{n-k}) \\ &= (1 - p^n - q^n) x_n. && \text{(by (3.6))} \end{aligned}$$

(ii) When  $d > 0$ . By (3.3) and induction hypothesis,

$$a_{n,d} \leq \lceil \log(n+1) \rceil + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (x_k + x_{n-k}) = x_n.$$

This completes the proof.

The next step involves solving asymptotically recurrence (3.4). This is a standard recurrence that can be solved by using generating functions, Mellin transform, and poissonization [22]. The proof of the following lemma can be found in our journal version of this paper [5].

LEMMA 3.2. *Consider the following recurrence for  $x_n$  with  $x_0 = x_1 = 0$  and for  $n \geq 2$*

$$x_n = a_n + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (x_k + x_{n-k}),$$

where  $a_n = \lceil \log(n+1) \rceil$  for  $n \geq 2$  and  $a_0 = a_1 = 0$ . Then:

(i) *If  $\log p / \log q$  is irrational, then*

$$x_n = \frac{n}{h} A^*(-1) \log e + o(n),$$

where

$$A^*(-1) = \sum_{b \geq 2} \frac{\lceil \log(b+1) \rceil}{b(b-1)}.$$

(ii) *If  $\log p / \log q = r/d$  (rational) with  $\gcd(r, d) = 1$ , then*

$$x_n = \frac{n}{h} (A^*(-1) + \Phi(\log_p n)) \log e + O(n^{1-\eta})$$

for some  $\eta > 0$ , where

$$\Phi(x) = \sum_{k \neq 0} A^*(-1 + 2k\pi r i / \log p) \exp(2k\pi r x i)$$

is a fluctuating function with a small amplitude.

Finally, the average length of  $L(B_1)$  can be derived. We present it in the next theorem.

THEOREM 3.1. *For large  $n$ ,*

$$\mathbf{E}[L(B_1)] \leq \frac{n}{h} (\beta + \Phi_1(\log n)) + o(n),$$

where  $h := h(p)$ ,

$$\beta = \log e \cdot \sum_{b \geq 2} \frac{\lceil \log(b+1) \rceil}{b(b-1)} = 3.760 \dots,$$

and  $\Phi_1(\log n)$  is a fluctuating function for  $\log p / \log q$  rational with small amplitude and asymptotically zero otherwise.

The next step is to estimate the average length of  $B_2$ . Let  $S_{n,d}$  be the total number of nodes  $x$  in  $T_{n,d}$  such that  $N_x = 1$ , that is,

$$\begin{aligned} S_{n,d} &= \sum_{x \in T_{n,d} \text{ and } N_x=1} 1 = \sum_{x \in T_{n,d} \text{ and } N_x=1} N_x \\ &= \sum_{x \in T_{n,d}} N_x - \sum_{x \in T_{n,d} \text{ and } N_x > 1} N_x. \end{aligned}$$

Let  $B_{n,d} = \sum_{x \in T_{n,d}, N_x > 1} N_x$ . We observe that

$$(3.7) \quad L(B_2) = S_{n,0} = \sum_{x \in T_{n,0}} N_x - B_{n,0} = \frac{n(n-1)}{2} - B_{n,0}.$$

The last equality follows from the fact that the sum of  $N_x$ 's for all  $x$  at level  $\ell$  in  $T_{n,0}$  is equal to  $n-1-\ell$ .

Let  $b_{n,d} = \mathbf{E}[B_{n,d}]$ . For our analysis we only need  $b_{n,0}$ . Clearly,  $b_{0,d} = b_{1,d} = 0$  and  $b_{2,0} = 0$ . For  $n \geq 2$ , we can find the following recurrence (similarly to  $a_{n,d}$ ):

$$(3.8) \quad b_{n+1,0} = n + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (b_{k,0} + b_{n-k,k}),$$

and for  $d > 0$ ,

$$(3.9) \quad b_{n,d} = n + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (b_{k,d-1} + b_{n-k,k+d-1}).$$

To prove our main result, we only need a lower bound that is established in the next lemma.

LEMMA 3.3. *For all  $n \geq 0$  and  $d \geq 0$ ,*

$$b_{n,d} \geq y_n - \frac{n}{2}$$

such that  $y_n$  satisfies  $y_0 = 0$  and for  $n \geq 0$

$$(3.10) \quad y_{n+1} = n + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} (y_k + y_{n-k}).$$

*Proof.* We prove it by induction on both  $n$  and  $d$ . Clearly,  $b_{n,d} \geq y_n - n/2$  for  $n = 0$  or  $1$  ( $d > 0$ ). For  $n = 2$  and  $d = 0$ ,  $b_{2,0} \geq y_2 - 2$  since  $b_{2,0} = 0$  and  $y_2 = 1$ . For other cases ( $n = 2$  and  $d > 0$ , or  $n > 2$ ), we assume that  $b_{i,j} \geq y_i - \frac{i}{2}$  holds for  $i < n$ , and for  $i = n$  and  $j < d$ . Now we want to show that  $b_{n,d} \geq y_n - \frac{n}{2}$ . We divide it into two cases.

(i) When  $d = 0$ . By (3.8) and induction hypothesis,

$$\begin{aligned} b_{n,0} &\geq (n-1) + \sum_{k=0}^{n-1} \binom{n-1}{k} p^k q^{n-1-k} \\ &\quad \cdot \left( y_k - \frac{k}{2} + y_{n-1-k} - \frac{n-1-k}{2} \right) \\ &= y_n - \frac{n-1}{2} > y_n - \frac{n}{2}. \quad (\text{by (3.10)}) \end{aligned}$$

(ii) When  $d > 0$ . By (3.9) and induction hypothesis,

$$\begin{aligned} b_{n,d} &\geq n + \sum_{k=0}^n \binom{n}{k} p^k q^{n-k} \left( y_k - \frac{k}{2} + y_{n-k} - \frac{n-k}{2} \right) \\ &= y_{n+1} - \frac{n}{2} \geq y_n - \frac{n}{2}. \end{aligned}$$

This completes the proof.

It is easy to see that  $y_n$  represents the expected path length in a *digital search tree* over  $n$  strings as discussed in [13, 22]. The authors of [13] proved, among others, that

$$(3.11) \quad y_n = \frac{n}{h} \left( \log n + \frac{h_2}{2h} + \gamma - 1 - \alpha + \Phi_2(\log n) \right) + \frac{1}{h} \left( \log n + \frac{h_2}{2h} - \gamma - \log p - \log q + \alpha \right) + O(1),$$

where  $h_2 = p \log^2 p + q \log^2 q$ ,  $\gamma = 0.577 \dots$  is the Euler constant, and

$$\alpha = - \sum_{k=1}^{\infty} \frac{p^{k+1} \log p + q^{k+1} \log q}{1 - p^{k+1} - q^{k+1}}.$$

In the above,  $\Phi_2(\log n)$  is a fluctuating function for  $\log p / \log q$  rational with small amplitude and zero otherwise.

In summary, by (3.7), Lemma 3.3, and the above, we arrive at our next result.

THEOREM 3.2. *For large  $n$ ,*

$$\begin{aligned} \mathbf{E}[L(B_2)] &\leq \frac{n(n-1)}{2} - \frac{n}{h} \log n \\ &+ \frac{n}{h} \left( \frac{h}{2} - \frac{h_2}{2h} - \gamma + 1 + \alpha - \Phi_2(\log n) \right) - \frac{1}{h} \log n + O(1), \end{aligned}$$

with the notations as below (3.11).

Finally, we compute  $\mathbf{E}[L(S)] = \mathbf{E}[L(\hat{B}_1) + L(\hat{B}_2)] + O(\log n)$ , proving the part (i) of Theorem 2.2. We observe that the arithmetic encoder can compress a binary sequence of length  $m$  on average up to  $mh + \frac{1}{2} \log m + O(1)$ , where  $h$  is the entropy rate of the binary source [7, 24]. Thus, by Theorem 3.1,

$$\mathbf{E}[L(\hat{B}_1)] \leq \frac{h'}{h} (\beta + \Phi_1(\log n))n + o(n),$$

where  $h$ ,  $\beta$ , and  $\Phi_1(\log n)$  are defined in Theorem 3.1, and  $h'$  is the entropy rate of the binary source that  $B_1$  is generated from. Similarly, we can compute  $\mathbf{E}[L(\hat{B}_2)]$ .

In this case, however, we know that the entropy rate for  $B_2$  is  $h := h(p)$ . Thus, by Theorem 3.2,

$$\begin{aligned} \mathbf{E}[L(\hat{B}_2)] &\leq \binom{n}{2} h - n \log n \\ &+ n \left( \frac{h}{2} - \frac{h_2}{2h} - \gamma + 1 + \alpha - \Phi_2(\log n) \right) + O(\log n), \end{aligned}$$

where  $h$ ,  $h_2$ ,  $\gamma$ ,  $\alpha$ , and  $\Phi_2(\log n)$  are defined above. This completes the part (i) of Theorem 2.2.

**3.3 Performance with High Probability.** Now we prove part (ii) of Theorem 2.2, that is, we show that  $L(S) - \mathbf{E}[L(S)] \leq \epsilon n \log n$  with high probability. Since  $L(S) = L(\hat{B}_1) + L(\hat{B}_2)$ , we need bounds for  $L(\hat{B}_1)$  and  $L(\hat{B}_2)$ . We start with  $L(\hat{B}_1)$ . By Markov's inequality,

$$P\left(L(\hat{B}_1) > \epsilon n \log n\right) < \frac{\mathbf{E}[L(\hat{B}_1)]}{\epsilon n \log n} = O\left(\frac{1}{\log n}\right), \quad \epsilon > 0. \quad (3.12)$$

Handling  $L(\hat{B}_2)$  is more complicated. In [24] it was proved that for a binary sequence  $X$  of length  $\ell$ , the code length generated by an arithmetic encoder is at most  $-\log P(X) + \frac{1}{2} \log \ell + 3$ . In our case,  $B_2 = b_1 b_2 \cdots b_{L(B_2)}$  is memoryless, and then

$$\begin{aligned} L(\hat{B}_2) &< -\log P(B_2) + \frac{1}{2} \log L(B_2) + 3 \\ &= L(B_2) \cdot \left[ -\frac{1}{L(B_2)} \sum_{i=1}^{L(B_2)} \log P(b_i) \right] + \frac{1}{2} \log L(B_2) + 3. \end{aligned} \quad (3.13)$$

Thus we need good bounds for  $L(B_2)$  and the sum of  $\log P(b_i)$ . With respect to  $L(B_2)$ , recall that  $L(B_2) = \binom{n}{2} - B_{n,0}$  where

$$B_{n,0} = \sum_{x \in T_{n,0}, N_x > 1} N_x,$$

and  $N_x$  is the number of balls that pass through node  $x$  in tree  $T_{n,0}$  (excluding the ball removed at  $x$  if any). We shall show that  $B_{n,0}$  is related to the *path lengths* in slightly modified trees that we denote as  $\hat{T}_n$  and  $\bar{T}_n$ . The tree  $\hat{T}_n$  is constructed from  $T_{n,0}$  by removing all nodes  $x$  with  $N_x = 1$  that are not direct children of nodes  $y$  with  $N_y > 1$ . Then, we put back balls into the nodes of  $\hat{T}_n$  using the following rules: we put each ball back into the node where it was removed; if such a node does not exist in  $\hat{T}_n$ , then we put the ball into the first node  $x$  with  $N_x = 1$  on its path in  $T_{n,0}$ . To construct  $\bar{T}_n$  we observe that there might be some nodes with two balls in  $\hat{T}_n$ . In such a case, we add a child node and move one ball down to the new node to eliminate

all nodes with two balls. Figure 5(a,b) illustrates the construction of  $\hat{T}_n$  and  $\bar{T}_n$  for the tree  $T_n$  (equivalently,  $T_{n,0}$ ) from Figure 4. Notice that in this figure all circle-shaped nodes and the square-shaped nodes – directly connected to these circle-shaped nodes – are the same in both  $T_n$  and  $\hat{T}_n$ .

Let  $\ell(\hat{T}_n)$  and  $\ell(\bar{T}_n)$  be the path lengths to all *balls* in  $\hat{T}_n$  and  $\bar{T}_n$ , respectively. From the construction it is clear that

$$B_{n,0} = \ell(\hat{T}_n).$$

Now let us compare  $\ell(\hat{T}_n)$  and  $\ell(\bar{T}_n)$ . Whenever we have two balls in a node of  $\hat{T}_n$ , we move one ball down in  $\bar{T}_n$  to a new node. This results in such a path in  $\bar{T}_n$  being longer by one than the corresponding path in  $\hat{T}_n$ . However, this can happen at most  $n/2$  times since there are at most  $n/2$  nodes with two balls. Thus we find<sup>2</sup>

$$\ell(\hat{T}_n) + n/2 \geq_{st} \ell(\bar{T}_n).$$

To estimate the path length  $\ell(\bar{T}_n)$ , we introduce another binary tree  $D_n$  that is probabilistically equivalent to the *digital search tree* built over  $n$  random binary strings. It is constructed as follows. If  $n = 0$ , then it is just an empty tree. For  $n > 0$ , we create a root node in which we put  $n$  balls. One ball remains in the root node, and rest of balls independently move down to the left or right. We create a new child node if there is at least one ball in that node. We recursively repeat it (i.e., we leave one ball in a node while moving others down.) Figure 5(c) illustrates this construction.

We shall next show that

$$\ell(\bar{T}_n) \geq_{st} \ell(D_n),$$

where  $\ell(D_n)$  is the path length to all balls (nodes) in  $D_n$ . For this, we consider two actual trees  $\bar{t}_n$  and  $d_n$  given the same binary choices regarding the action left/right (1/0) for the  $n$  balls. We also assume that the input to both trees is the same, that is, balls are inserted in the same order and therefore we always identify the “smallest” ball in input. Whenever a ball remains in a node during the construction of these trees, we assume that the smallest ball is left in the node. Then, in the next lemma we show that the path length in  $\bar{t}_n$  is at least the path length in  $d_n$ . Thus  $\ell(\bar{T}_n) \geq_{st} \ell(D_n)$ .

**LEMMA 3.4.** *Given binary choices for  $n$  balls, let  $\bar{t}_n$  and  $d_n$  be two tree instances of  $\bar{T}_n$  and  $D_n$ , respectively. Let  $u_t \in \bar{t}_n$  and  $u_d \in d_n$  be two corresponding nodes*

<sup>2</sup>For two real-valued random variables  $X$  and  $Y$ , we write  $X \geq_{st} Y$  if the value of  $X$  is always greater than or equal to that of  $Y$  for every event, or equivalently if  $P(X > t) \geq P(Y > t)$  for all  $t \in (-\infty, \infty)$  [17].

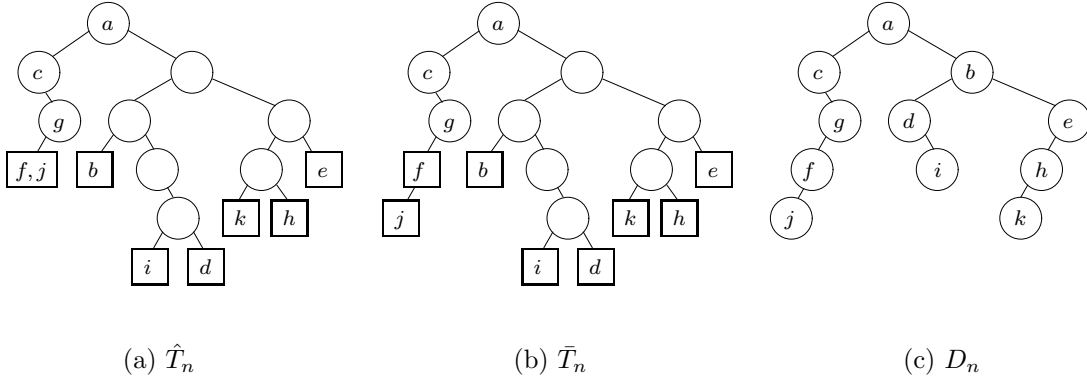


Figure 5: An example of binary trees  $\hat{T}_n$ ,  $\bar{T}_n$ , and  $D_n$ , given binary choices for eleven balls  $\{a, b, c, d, e, f, g, h, i, j, k\}$ .

in these trees (i.e., nodes that are reached by the same binary choices). We denote by  $B(u)$  the set of balls in the subtree rooted at node  $u$ . Then,  $B(u_t) \supset B(u_d)$  for any  $u_t \in \bar{t}_n$  and  $u_d \in d_n$ .

*Proof.* For the root nodes, it is trivial since both sets have the same  $n$  balls. Now it is sufficient to show that the statement is true for children if it is true for their parent nodes. Thus let us assume that  $B(u_t) \supset B(u_d)$  for  $u_t \in \bar{t}_n$  and  $u_d \in d_n$ . Let  $s_t$  and  $s_d$  be the smallest ball (in the input ordering) in  $B(u_t)$  and  $B(u_d)$ , respectively. Now we consider two sets of balls  $S_t$  and  $S_d$  that will move down from  $u_t$  and  $u_d$ , respectively. Note that  $S_d = B(u_d) - s_d$ . We shall show that  $S_t \supset S_d$  considering two cases: 1) if  $u_t$  is not the leftmost node, then  $S_t = B(u_t) \supset B(u_d) - s_d = S_d$ ; 2) if  $u_t$  is the leftmost node, then  $S_t = B(u_t) - s_t \supset B(u_d) - s_d = S_d$  since either  $s_t$  is the same ball as  $s_d$  or  $s_t$  is not in  $S_d$ . Therefore each ball  $b \in S_d$  is also in  $S_t$ , and  $b$  moves down in the same direction for both  $u_t$  and  $u_d$ . Therefore, the statement is true for both children nodes.

Now we are ready to prove a relation between  $B_{n,0}$  and the path length in a digital search tree shown in the following lemma.

LEMMA 3.5. *Let  $Y_n := \ell(D_n)$  be the path length in a digital search tree. Then,*

$$B_{n,0} + \frac{n}{2} \geq_{st} Y_n.$$

*Proof.* Given binary choices for  $n$  balls, let us consider tree instances  $\hat{t}_n$ ,  $\bar{t}_n$ , and  $d_n$ . As we observed,  $\ell(\hat{t}_n) + \frac{n}{2} \geq \ell(\bar{t}_n) \geq \ell(d_n)$ . Therefore,  $\ell(\hat{T}_n) + \frac{n}{2} \geq_{st} \ell(D_n)$ . We know that  $\ell(\hat{T}_n)$  and  $\ell(D_n)$  are equivalent to  $B_{n,0}$  and  $Y_n$ , respectively. This completes the proof.

Finally, we establish the following two lemmas.

LEMMA 3.6. *For any  $\epsilon > 0$ ,*

$$P\left(L(B_2) \leq \binom{n}{2} - y_n + \epsilon y_n\right) \geq 1 - o(1),$$

where  $y_n$  is defined in Lemma 3.3.

*Proof.* We observe that  $y_n = \mathbf{E}[Y_n]$ , where  $Y_n$  is the path length in a digital search tree. Let us compute the probability  $P_n = P\left(L(B_2) > \binom{n}{2} - y_n + \epsilon y_n\right)$  for large  $n$ . We shall prove that  $P_n \rightarrow 0$ . We have

$$\begin{aligned} P_n &= P(B_{n,0} < (1 - \epsilon)y_n) \quad (\text{by (3.7)}) \\ &\leq P\left(Y_n - \frac{n}{2} < (1 - \epsilon)y_n\right) \quad (\text{by Lemma 3.5}) \\ &= P\left(\frac{Y_n - y_n}{\sqrt{\text{Var } Y_n}} < \frac{-\epsilon y_n + n/2}{\sqrt{\text{Var } Y_n}}\right) \\ &< A\mu^k \quad (\text{by Theorem 1A of [13]}) \end{aligned}$$

for positive constants  $A$  and  $\mu < 1$ , where  $k = \frac{-\epsilon y_n + n/2}{\sqrt{\text{Var } Y_n}} = \Theta(\sqrt{n \log n})$  as proved in Theorem 1A of [13]. Thus,  $P_n$  becomes exponentially small as  $n \rightarrow \infty$ .

In view of (3.13) and Lemma 3.6, we need to find a bound for  $\sum_{i=1}^{L(B_2)} \log P(b_i)$  which we present next.

LEMMA 3.7. *For any  $\epsilon > 0$ ,*

$$P\left(-\frac{1}{L(B_2)} \sum_{i=1}^{L(B_2)} \log P(b_i) \leq h + \epsilon \frac{\log n}{n}\right) \geq 1 - o(1),$$

where  $h := h(p)$ .

*Proof.* Let  $F_m(X_1, \dots, X_m) = -\log P(X_1, \dots, X_m) - mh$ , where  $X_i$ 's are binary independent random variables with  $p$  being the probability of '1' and  $q = 1 - p$ . Denoting by  $\hat{X}_i$  an independent copy of  $X_i$  (with the same distribution as  $X_i$ ), we have

$$|F_m(X_1, \dots, X_i, \dots, X_m) - F_m(X_1, \dots, \hat{X}_i, \dots, X_m)| \leq |\log P(X_i) - \log P(\hat{X}_i)| \leq c,$$

where  $c = \max\{\log p/q, \log q/p\}$ . Thus, by Azuma's inequality [22]

$$P(-\log P(X_1, \dots, X_m) - mh \geq \epsilon' n \log n) \leq \exp\left(-\frac{\epsilon'^2 n^2 \log^2 n}{2mc^2}\right) = o(1)$$

provided that  $m = O(n^2)$ . Since  $L(B_2) = O(n^2)$ , this completes the proof.

By the above two lemmas, after some algebra we conclude that, with probability  $1 - o(1)$ ,

$$L(\hat{B}_2) < \binom{n}{2} h - n \log n + \epsilon n \log n.$$

This and (3.12) complete the part (ii) of Theorem 2.2.

**3.4 Time Complexity.** In this section, we prove part (iii) of Theorem 2.2, that is, we show that the time complexity of our algorithm in Section 2.2.3 is  $O(n + e)$  on average. Let us first analyze the first stage, which consists of  $n$  steps. Clearly, the procedure 1 takes constant time in each step, and thus it takes  $O(n)$  time in total. The procedures 2 and 4 take  $O(|N(v)|)$  time in each step since each of operations inside the loop takes constant time. Thus, they take  $\sum_{v \in V(G)} O(|N(v)|) = O(e)$  time in total. In the  $i$ -th step, the procedure 3 takes  $O(s_i)$  time, where  $s_i$  is the number of subsets in  $P_{i-1} - v$  whose size is larger than one. Thus, in total, it takes  $O(s)$  time where  $s = \sum_{i=1}^n s_i$ , which is the total number of nodes  $x$  in  $T_{n,0}$  with  $N_x > 1$ . In Figure 4, for example,  $s$  is the number of circle-shaped nodes in  $T_n$ . By the same analysis as in Lemma 3.2 (in this case,  $a_n = 1$ ), we can prove that the expected value of  $s$  is at most  $O(n)$ .

Finally, by Lemma 2.1, the construction of  $B_2$  from  $\mathcal{B}$  takes  $O(n + \ell)$  time where  $\ell$  is the number of elements inserted in  $\mathcal{B}$ . We can see that  $\ell = O(e + s)$  as follows. The number of elements inserted in procedure 2.2 is bounded by  $e$  since every insertion corresponds to a distinct edge. Clearly, the number of elements inserted in procedure 3.2 is bounded by  $O(s)$ . Therefore, the first stage takes  $O(n + e)$  time on average.

In the second stage,  $B_1$  and  $B_2$  are compressed by an arithmetic encoder. Clearly,  $B_1$  can be compressed in  $O(n)$  time since the length of  $B_1$  is  $O(n)$ . The number of elements in the run length form of  $B_2$  is at most  $e + 1$ . Thus, the time complexity of the compression of  $B_2$  would be  $O(e)$  except that there could be long runs of zeroes, which are compressed in multiple steps. Let  $n_0$  and  $n_1$  be the number of '0's and '1's in  $B_2$ , respectively. Thus,  $p = n_1/(n_0 + n_1)$ . The number of additional steps to process  $k_{max} = \lceil 1/p \rceil$  zeroes is bounded by

$$\frac{n_0}{\lceil 1/p \rceil} \leq \frac{n_0}{1/p} = \frac{n_0 n_1}{n_0 + n_1} \leq n_1 \leq e.$$

Therefore, the second stage takes  $O(n + e)$  time. This completes the proof.

### Acknowledgement

The author would like to thank Wojciech Szpankowski for numerous discussions and his invaluable comments.

### References

- [1] M. Adler and M. Mitzenmacher, Towards compressing web graphs, *In Proc. of the IEEE Data Compression Conference*, 203–212, 2001.
- [2] B. Bollobas, *Random Graphs*, Cambridge University Press, Cambridge, 2001.
- [3] F.P. Brooks Jr, Three great challenges for half-century-old computer science, *Journal of the ACM*, 50(1), 25–26, 2003.
- [4] Y. Choi and W. Szpankowski, Compression of graphical structures, *IEEE International Symposium on Information Theory*, Seoul, 364–368, 2009.
- [5] Y. Choi and W. Szpankowski, Compression of Graphical Structures: Fundamental Limits, Algorithms, and Experiments, In submission to *IEEE Transactions on Information Theory*, available at <http://www.cs.purdue.edu/~spa/papers/structure.pdf>.
- [6] T.M. Cover and J.A. Thomas, *Elements of Information Theory*, John Wiley & Sons, New York, 2006.
- [7] M. Drmota, H.-K. Hwang, and W. Szpankowski, Precise average redundancy of an idealized arithmetic coding, *Proc. Data Compression Conference*, 222–231, 2002.
- [8] M. Drmota, Y. Reznik, and W. Szpankowski, Tunstall Code, Khodak Variations, and Random Walks, preprint 2009.
- [9] G. Fayolle, P. Flajolet, and M. Hofri, On a Functional Equation Arising in the Analysis of a Protocol for a Multi-Access Broadcast Channel, *Advances in Applied Probability*, 18(2), 441–472, 1986.
- [10] P. Flajolet and R. Sedgewick, *Analytic Combinatorics*, Cambridge University Press, Cambridge, 2008.
- [11] F. Harary and E.M. Palmer, *Graphical Enumeration*, Academic Press, 1973.

- [12] P. Jacquet, and W. Szpankowski, Analytical depoissonization and its applications, *Theoretical Computer Science*, 201, 1–62, 1998.
- [13] P. Jacquet and W. Szpankowski, Asymptotic behavior of the Lempel-Ziv parsing scheme and digital search trees, *Theoretical Computer Science*, 144(1&2), 161–197, 1995.
- [14] Y. Jia, E.-H. Yang, D.-K. He, and S. Chan, A greedy renormalization method for arithmetic coding, *IEEE Transactions on Communications*, 55(8):1494–1503, 2007.
- [15] M. Naor, Succinct representation of general unlabeled graphs, *Discrete Applied Mathematics*, 28(3), 303–307, 1990.
- [16] L. Peshkin, Structure induction by lossless graph compression, *In Proc. of the IEEE Data Compression Conference*, 53–62, 2007.
- [17] S. Ross, *Stochastic Processes*, John Wiley & Sons, New York, 1983.
- [18] S.A. Savari, Compression of words over a partially commutative alphabet, *IEEE Trans. on Information Theory*, 50, 1425–1441, 2004.
- [19] W. Schachinger, Limiting distributions for the costs of partial match retrievals in multidimensional tries. *Random Structures and Algorithms*, 17(3-4), 428–459, 2000.
- [20] C. Shannon. The lattice theory of information. *IEEE Transaction on Information Theory*, 1:105–107, 1953.
- [21] J. Sun, E.M. Bollt, and D. Ben-Avraham, Graph compression—save information by exploiting redundancy, *Journal of Statistical Mechanics: Theory and Experiment*, P06001, 2008.
- [22] W. Szpankowski, *Average Case Analysis of Algorithms on Sequences*, John Wiley & Sons, New York, 2001.
- [23] Gy. Turan, On the succinct representation of graphs, *Discrete Applied Mathematics*, 8(3), 289–294, 1984.
- [24] F.M.J. Willems, Y.M. Shtarkov, and T.J. Tjalkens, The Context Tree Weighting Method: Basic Properties, *IEEE Transactions on Information Theory*, 41, 653–664, 1995.