DESIGN AND IMPLEMENTATION OF HIGH PERFORMANCE ALGORITHMS FOR THE (N, K)-UNIVERSAL SET PROBLEM

A Thesis

by

PING LUO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2009

Major Subject: Computer Science

DESIGN AND IMPLEMENTATION OF HIGH PERFORMANCE ALGORITHMS FOR THE (N, K)-UNIVERSAL SET PROBLEM

A Thesis

by

PING LUO

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2009

Major Subject: Computer Science

ABSTRACT

Design and Implementation of High Performance Algorithms

for the (n, k)-Universal Set Problem. (August 2009)

Ping Luo, B.S., University of Electronic Science and Technology of China;

M.S., Indiana State University

Chair of Advisory Committee: Dr. Jianer Chen

The k-PATH problem is to find a simple path of length k. This problem is NP-complete and has applications in bioinformatics for detecting signaling pathways in protein interaction networks and for biological subnetwork matching. There are algorithms implemented to solve the problem for k up to 13. The fastest implementation has running time $O^*(4.32^k)$, which is slower than the best known algorithm of running time $O^*(4^k)$. To implement the best known algorithm for the k-PATH problem, we need to construct (n, k)-universal set.

In this thesis, we study the practical algorithms for constructing the (n, k)-universal set problem. We propose six algorithm variants to handle the increasing computational time and memory space needed for $k = 3, 4, \ldots, 8$. We propose two major empirical techniques that cut the time and space tremendously, yet generate good results. For the case $k = 7$, the size of the universal set found by our algorithm is 1576, and is 4611 for the case $k = 8$.

We implement the proposed algorithms with the OpenMP parallel interface and construct universal sets for $k = 3, 4, \ldots, 8$. Our experiments show that our algorithms for the (n, k)-universal set problem exhibit very good parallelism and hence shed light on its MPI implementation.

Ours is the first implementation effort for the (n, k)-universal set problem. We share the effort by proposing an extensible universal set construction and retrieval system. This system integrates universal set construction algorithms and the universal sets constructed. The sets are stored in a centralized database and an interface is provided to access the database easily.

The (n, k)-universal set have been applied to many other NP-complete problems such as the SET SPLITTING problems and the MATCHING and PACKING problems. The small (n, k)-universal set constructed by us will reduce significantly the time to solve those problems.

To my family

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Jianer Chen, for his guidance and support. Without him, this work would not be possible. I would also like to thank my committee members, Dr. Donald Friesen and Dr. Sing-Hoi Sze, for their support of this research.

Thanks to my friends and colleagues at the Supercomputing Facility of Texas A&M: Spiros Vellas, Michael Thomadakis, Francis Dang, Greta Thomas, Xiandong Meng, Keith Jackson, Raffaele Montuoro, and Taesung Kim. They have provided valuable advice for me to finish this work while working full time. I also want to extend my gratitude to the staff and faculty of the Department of Computer Science and Engineering, especially Dr. Eun Jung Kim, Dr. Anxiao (Andrew) Jiang, and Dr. Valerie E. Taylor, for making my time at Texas A&M a great experience.

Thanks also go to my mother and father for their unconditional love and my sisters for their encouragement.

Finally, I would like to thank my husband, Yang, for his patience and love.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION . . . 1
- A. (n, k)-Universal Set . . . 1
- B. High Performance Computing . . . 3
- C. Contributions . . . 4
- D. Outline of the Thesis . . . 5

II THEORETICAL BACKGROUND . . . 6
- A. Basic Probability and Algebra Terminologies . . . 6
- B. Construction of n k-wise Independent Random Variables . . . 8
- C. Construction of Small (n, k)-universal Sets . . . 10
- D. Mapping n to $k(k-1)+1$ . . . 11

III ALGORITHM VARIANTS . . . 14
- A. Algorithm Variant 1 . . . 14
- B. Algorithm Variant 2: Random Pick Initialization . . . 15
- C. Algorithm Variants 3 . . . 19
- D. Algorithm Variants for $k \geq 8$ . . . 19

IV DESIGN OF UNIVERSAL SET CONSTRUCTION AND RETRIEVAL SYSTEM . . . 24
- A. Design Goals . . . 24
- B. Universal Set Construction and Retrieval System Structure 24
  - 1. Core Component . . . 26
    - a. Parallel Search Engines . . . 26
    - b. Efficient Memory Management Module . . . 27
  - 2. Parity Check Files . . . 28
  - 3. USDB . . . 29
  - 4. Mapper . . . 29
- C. USDB Read and Write Procedures . . . 30

V EXPERIMENTS . . . 31
- A. Computing Platforms . . . 31
- B. Experiments . . . 31

CHAPTER Page

1. A Glimpse of Constructed Universal Sets 33
2. Initial Size for Random Pick Initialization 34
3. Effectiveness of Progressive Construction 36
4. Scalability 38
5. Is Randomness Alone Enough? 39

VI CONCLUSIONS AND FUTURE WORK 42

A. Conclusions 42
B. Future Work 42

REFERENCES 44

VITA 47

LIST OF TABLES

TABLE Page

I Size of the bipartite graph 15

II Estimated wall clock time for $k = 7$ 17

III Size of $(n, 4)$-universal sets for $n = 4, 7, \ldots, 53$ 22

IV Hydra system configuration 32

V Linux workstation testbed system configuration 32

VI Size of constructed universal sets 33

VII Step by step construction of a (57,8)-universal set 37

VIII Performance comparison of algorithm variant 4, 5, and 6 38

LIST OF FIGURES

FIGURE Page

1 Algorithm for constructing n k-wise independent random variables . 9

2 Algorithm for constructing (n, k)-universal sets 11

3 Faster algorithm for constructing (n, k)-universal sets 12

4 Algorithm variant 1 for the (n, k)-universal set problem 16

5 Algorithm variant 2 for the (n, k)-universal set problem 18

6 Algorithm variant 4 for the (n, k)-universal set problem 20

7 Algorithm variant 5 for the (n, k)-universal set problem 21

8 Algorithm variant 6 for the (n, k)-universal set problem 23

9 Universal set construction and retrieval system architecture 25

10 A sample matlab script for finding GF(128) and its odd powers . . . 29

11 Number of partitions not implemented 35

12 Percent of partitions not implemented 35

13 Comparison of randomly generated splitting function and randomly picked splitting function on the number of partitions that are not implemented 36

14 Comparison of randomly generated splitting function and randomly picked splitting function on percent of partitions that are not implemented 37

15 Scalability of the OpenMP program on 2, 4, 8, and 16 processors . . 39

FIGURE Page

16 Size of (21,5)-universal sets constructed from randomly generated splitting function sets, as compared with the size of the (21,5)-universal set constructed from the random space universal set. The graph shows results for 50 tests 40

17 Size of (21,5)-universal sets constructed from randomly generated splitting function sets. Let m be the size of a randomly generated splitting function set. (a) $m = 1500$. (b) $m = 2000$. (c) $m = 2500$ (d) $m = 3000$. Each subgraph contains 10 test results 41

CHAPTER I

INTRODUCTION

Given a graph G, the k-PATH problem is to find a simple path of length k. This problem has applications in bioinformatics for detecting signaling pathways in protein interaction networks [15] and for biological subnetwork matchings [11]. The complexity of the k-PATH problem has been intensively studied [2, 4, 7, 13, 17]. The best randomized algorithm has running time $O^*(2^k)$ [17]. In this thesis, we use $O^*(f)$ to refer to $O(f poly(n))$ where $poly(n)$ is a polynomial of n. So $O^*(2^k)$ means $O(2^k poly(n)$. The best deterministic algorithm has running time $O^*(4^k)$ [7]. There are some experimental studies showing that the k-PATH problem in those applications can be solved for k up to 13 [9, 15]. The best known implementation is based on color-coding and dynamic programming, and has running time $O^*(4.32^k)$ [9], which is not better than the best known deterministic algorithm of time $O^*(4^k)$ [7]. To implement the best deterministic algorithm, we need construct the (n, k)-universal set.

A. (n, k)-Universal Set

A *partition* of a set S is denoted by (S_1, S_2) such that $S_1 \cup S_2 = S$, and $S_1 \cap S_2 = \emptyset$. A *splitting function* f over a set S is a mapping from S to $\{0, 1\}$. A splitting function f *implements* a partition (S_1, S_2) if $\forall x \in S_1$, $f(x) = 0$, and $\forall y \in S2$, $f(y) = 1$.

Let n, k be integers, and Z_n be the set $\{0, 1, \ldots, n-1\}$. An (n, k)-*universal set* P is a set of splitting functions such that for any k-subset S of Z_n and any partition (S_1, S_2) of S, there is a splitting function f in P that implements (S_1, S_2). The *size* of an (n, k)-universal set P is the number of splitting functions in P. In the rest of this

The journal model is *IEEE Transactions on Automatic Control.*

thesis, *universal set* refers to (n, k)-*universal set* if no ambiguity within the context.

It is slow to construct (n, k)-universal set even when k is small (CITE). This will increase greatly the running time of the best deterministic algorithm for the k-PATH problem. However, the size of (n, k)-universal set is small for k is small. Then we can construct the (n, k)-universal set, store it, and use it whenever we run the algorithm for the k-PATH problem. By this way, we only need construct the (n, k)-universal set once, thus reducing significantly the running time of the algorithm for the k-PATH problem.

Recently, the (n, k)-universal sets have recently been used to develop efficient parameterized algorithms for a variety of other problems, including the weighted SET SPLITTING problem [6], k-PATH, MATCHING, and PACKING[7, 12]. Similarly, once the (n, k)-universal sets are constructed, they can be used to solve those problems.

Due to these applications of the (n, k)-universal sets, it is critical to construct (n, k)-universal sets of small sizes. With small (n, k)-universal sets, we can reduce greatly the time to solve the SET SPLITTING, PATH, MATCHING, and PACKING problems.

Hence our goals for the research in this thesis include:

- Construct universal sets for some k's. we want k to be as large as possible and the universal set size as small as possible.
- Build a database to store the universal sets constructed and share it with other researchers so that they don't have to reinvent the wheel.

In general, our construction of the (n, k)-universal sets provides a solid foundation to implement many FIXED PARAMETER TRACTABLE algorithms, which are of running time $O^*(f(k))$ where $f(k)$ is independent of the input size n. Because of many real applications, the studies of fixed parameter tractable algorithms have drawn increasing attentions [8]. There are many experimental studies of fixed parameter tractable algorithms [3, 5, 9]. However, we are the first to implement algorithms to construct the (n, k)-universal set.

B. High Performance Computing

High performance computing (HPC) refers to solving advanced computation problems quickly and reliably using the power of parallel machines. A parallel machine can either be a shared memory supercomputer or a distributed memory computer cluster. The former can efficiently exchange or share data through the shared memory, and the latter communicates via messages.

One way of using HPC on solving a problem is to program the problem with parallel programming models. There are two most commonly used parallel programming models: OpenMP and MPI. OpenMP (Open Multi-Processing) is an application programming interface for parallel program implementation on shared memory architectures. It uses directives to tell processor how to distribute data and work across the processors. MPI stands for Message Passing Interface. It is a communication protocol designed especially for distributed systems, but can also be used for shared memory architectures. It uses message passing to send/receive data between processes.

There are pros and cons for the two models. OpenMP is easy to use and debug. It is easy to convert a serial code into an OpenMP parallel code. Its downsides include: OpenMP can only be used in shared memory architectures and it is mostly used for loop parallelization. MPI runs on both shared memory architectures and distributed memory architectures. It can be used in a wider range of applications than OpenMP. Its weaknesses include: MPI requires more design and programming efforts and is hard to debug. Its performance can be affected by the communication network interconnecting the nodes.

The universal set problem, as many combinatorial problems, is a good candidate to be solved with HPC. First, the algorithms for solving the (n, k)-universal set problem require a large amount of computing time, especially when k is large. Second, the algorithms have good features for parallelization and hence can make good use of the power of HPC.

Due to our time limit, we choose OpenMP to parallelize our programs. However, the problem is also suitable for MPI.

C. Contributions

Our contributions presented in this thesis include the following five aspects:

- We construct and store universal sets for $k = 3, 4, \ldots, 8$ with $n = k(k-1)+1$). Our universal set size is small. The size for $k = 7$ is 1576 and for $k = 8$ is 4611.
- We propose a universal set construction and retrieval framework for easy sharing of our universal sets. The sets can be used to directly in the construction for splitting set, k-path, matching, and packing.
- Ours is the first implementation effort for the (n, k)-universal set problem. It can lead to implementations for a variety of parameterized problems.
- The scalability of our algorithms parallelized with OpenMP demonstrates the universal set problem has inherent parallelism. This provides us valuable insight on the possibility to parallelize the problem with MPI.
- We provide two simple but effective empirical techniques based on some nice observations both from the property of the problem and experimental results.

The techniques nicely handle larger cases ($k \geq 7$) when the computational resources (memory space and CPU time) become limited.

D. Outline of the Thesis

The thesis is organized as follows: In Chapter II, we introduce the theoretical background of our study. The main algorithms for solving the universal set problem are discussed. In Chapter III, we introduce the algorithm variants that are improved with different heuristics to handle different k's. In Chapter IV, we present a framework of universal set construction and retrieval system and discuss each component in detail. In Chapter V, experiments for examining the system are discussed and results are presented. In Chapter VI, We provide conclusions for our research and list future work for the study.

CHAPTER II

THEORETICAL BACKGROUND

In this chapter, we describe the current results on constructing the (n, k)-universal set. First we give the terminologies needed for discussions. Then we show how to construct n k-wise independent random variables and how to construct (n, k)-universal set using n k-wise independent random variables. Finally, we show how to reduce the construction of (n, k)-universal set problem to $O(n)$ constructions of $(k(k-1)+1, k)$-universal set problems and give the fastest algorithm constructing (n, k)-universal set.

A. Basic Probability and Algebra Terminologies

We take the basic probability definitions from the book by Motwani and Raghavan [14].

A *sample space* Ω is a set. The elements of a sample space are *elementary events.* A subset A of Ω is an *event.* A *probability measure* Pr is a function from subsets of Ω to the interval $[0, 1]$ such that (1) for all $A \subseteq \Omega$, $0 \leq Pr[A] \leq 1$, (2) $Pr[\Omega] = 1$, and (3) for disjoint events $A_1, A_2, \cdots$, $Pr[\cup_i A_i] = \sum_i Pr[A_i]$.

A collection of events $\{A_i | i \in I\}$ is independent if for all subsets $S \subseteq I$, we have $Pr[\cap_{i \in S} A_i] = \Pi_{i \in S} Pr[A_i]$. These events are *k-wise independent* if every subcollection of k events is independent.

A *random variable* X is a real-valued function over the sample space such that for all real value x, we have $\{\omega \in \Omega | X(\omega) \leq x\}$ is an event. Then we have that $Pr[X \leq x] = Pr[\{\omega \in \Omega | X(\omega) \leq x\}]$. A collection of n random variables $X_1, X_2, \cdots, X_n$ are

k-wise independent if for any k random variables $X'_1, X'_2, \cdots, X'_k$, we have

$$Pr[X'_1 = x_1, X'_2 = x_2, \cdots, X'_k = x_k] = Pr[X'_1 = x_1]Pr[X'_2 = x_2] \cdots Pr[X_k = x_k].$$

A *binary operation* $\star$ on a set G is a function $\star$: $G \times G \rightarrow G$. For any a, $b \in G$, we write $a \star b$ for $\star(a, b)$. A binary operation $\star$ on G is *associative* if for all a, b, $c \in G$, we have $a \star (b \star c) = (a \star b) \star c$. A binary operation $\star$ on G is *commute* if for all a, $b \in G$, $a \star b = b \star a$.

A *group* is a pair of a set G and a binary operation $\star$ such that (1) $\star$ is associative, (2) there exists an element $e \in G$, called an *identity* of G, such that for all $a \in G$, we have $a \star e = e \star a = a$, and (3) for each $a \in G$, there is an element $a^{-1} \in G$, called an *inverse* of a, such that $a \star a^{-1} = e$. A group $(G, \star)$ is an *abelian* group if for all a, $b \in G$, we have $a \star b = b \star a$, i.e., the operation $\star$ is commutative.

A *field* is a triple $(F, +, \cdot)$ such that $(F, +)$ is an abelian group with identity 0 and $(F - \{0\}, \cdot)$ is also an abelian group, and $a \cdot (b + c) = (a \cdot b) + (a \cdot c)$ for all a, b, c in G. The *order* of a field $(F, +, \cdot)$ is the cardinality of F. A *Galois field $GF(q)$* is a field of finite order q. The order q of a Galois field $GF(q)$ must be a power of a prime [16].

A Galois field $GF(q)$ can be represented using $0, \alpha, \alpha^2, \cdots, \alpha^{q-1}$ where α is a *primitive* element in $GF(q)$ such that $\alpha^i \neq \alpha^j$ when $i \neq j$ and $\alpha^{q-1} = 1$. The *order* of a primitive element is $q - 1$. A *polynomial $f(x)$ over $GF(q)$* is a polynomial $a_0 + a_1 x + \cdots + a_n x^n$ with coefficient $a_i \in GF(q)$. The addition and multiplicative operations of polynomials over $GF(q)$ are defined as:

$$\sum_{i=1}^{n} a_i x^i + \sum_{i=1}^{n} b_i x^i = \sum_{i=1}^{n} (a_i + b_i) x^i.$$

$$\sum_{i=1}^{n} a_i x^i \cdot \sum_{j=1}^{n} b_j x^j = (a_0 \cdot b_0) + [(a_0 \cdot b_1) + (a_1 \cdot b_0)]x + [(a_0 \cdot b_2) + (a_1 \cdot b_1) + (a_2 \cdot b_0)]x^2 + \cdots$$

where $a_i + b_i$ is the the addition operation in $GF(q)$ and $a_i \cdot b_j$ is the multiplicative operation in $GF(q)$.

A polynomial $f(x)$ is *irreducible* in $GF(q)$ if $f(x)$ can not be factored into a product of lower-degree polynomials over $GF(q)$. An irreducible polynomial $f(x)$ over $GF(q)$ of degree m is a *primitive* polynomial over $GF(q)$ if the smallest positive integer n for which $f(x)$ divides $x^n - 1$ is $n = q^m - 1$. It can be shown that the roots α of an mth-degree primitive polynomial $f(x)$ over $GF(q)$ have *order* $q = p^m - 1$ [16], i.e., α is a primitive element in $GF(q)$ and $GF(q)$ can be represented by $0, \alpha, \alpha^2, \cdots, \alpha^{q-1}$. Note that the order of an element in $GF(q)$ may not necessary be q, the order of the Galois field $GF(q)$.

Given a primitive polynomial $f(x)$ over $GF(p)$, let α be a root of $f(x)$. Then $GF(q = p^m)$ can be represented by polynomials of α over $GF(p)$ and those polynomials are of degree $m - 1$ or less. Those polynomials can also be represented as vectors of length m. For example, let α be a root of a primitive polynomial $x^3 + x + 1$ over $GF(2)$. Then $GF(8)$ can be represented by $0, \alpha, \alpha^2, \cdots, \alpha^7 = 1$. We use vectors $[0, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$ to represent $0, \alpha$ and α^2 respectively. Since $\alpha^3 + \alpha + 1 = 0$, we have that $\alpha^3 = \alpha + 1$ and the vector $[1, 1, 0]$ can represent α^3.

B. Construction of n k-wise Independent Random Variables

Now we show how to construct n k-wise independent random variables, which will be used to construct (n, k)-universal sets. There is a lower bound on the sample spaces which have n k-wise independent random variables.

Lemma B.1 *[1] For fixed k, any sample space with n k-wise independent random* *variables has size* $\Omega(n^{\lfloor k/2 \rfloor})$.

Also we can construct a sample space containing n k-wise independent random variables such that the sample space is of size $2(n+1)^{(k-1)/2}$.

Lemma B.2 *[1] Let* $n = 2^d - 1$ *for an integer* d *and* $k \leq n$ *be an odd number. There is an algorithm of running time* $O(n(n+1)^{(k-1)/2})$ *that constructs a uniform probability space* Ω *of size* $2(n+1)^{(k-1)/2}$ *and a group of* n *k-wise independent* $0,1$-*random variables* $\xi_1, \ldots, \xi_n$ *over* Ω *such that* $Pr(\xi_i = 1) = 1/2$ *for all* $1 \leq i \leq n$.

The proof of Lemma-B.2 includes the construction of a sample space of n k-wise independent random variables. We list the construction in Fig. 1.

```
Algorithm 1
input : $n$, $k$. ($n = 2^d - 1$ and $d$ is an integer. $k$ is an odd number and $k \leq n$)
output: a (0,1)-matrix $A$ with $2(n+1)^{(k-1)/2}$ rows and $n$ columns,
        the random space $\Omega = \{1, 2, \ldots, 2(n+1)^{(k-1)/2}\}$,
        and the random variable $\xi_j$ defined as $\xi_j(i) = a_{ij}$.

1. Let $x_1, \ldots, x_n$ be the $n$ nonzero elements of $F = GF(n+1)$,
   represented as column-vectors of length $d$ over $GF(2)$.
2. $H = \begin{pmatrix} 1 & 1 & 1 & \ldots & 1 \\ x_1 & x_2 & x_3 & & x_n \\ x_1^3 & x_2^3 & x_3^3 & & x_n^3 \\ \vdots & & & \vdots & \vdots \\ x_1^{k-2} & x_2^{k-2} & x_3^{k-2} & \ldots & x_n^{k-2} \end{pmatrix}$
3. Create a (0,1)-matrix $A = (a_{ij})(i \in \Omega, 1 \leq j \leq n)$ whose
   $2(n+1)^{(k-1)/2}$ rows are all linear combinations of the rows of $H$.
```

Fig. 1. Algorithm for constructing n k-wise independent random variables

Note that the operation x_i^j in the algorithm 1 is the multiplicative operation $\cdot$ in the filed $GF(n+1)$.

The output matrix A of algorithm 1 contains only elements 0 and 1. Let S be a set of n elements. Without loss of generality, let S be $\xi_1, \cdots, \xi_n$. Then every row is a splitting function of S. Moreover, $Pr(\xi_{h_1} = b_{h_1}\xi_{h_2} = b_{h_2} \ldots \xi_{h_k} = b_{h_k}) = Pr(\xi_{h_1} = b_{h_1})Pr(\xi_{h_2} = b_{h_2}) \ldots Pr(\xi_{h_k} = b_{h_k}) = 1/2^k > 0$, since $\xi_1, \cdots, \xi_n$ are k-wise independent random variables. Thus for any k subset $\{\xi_{h_1}, \cdots, \xi_{h_k}\}$ of S and any partition $\xi_{h_1} = b_{h_1}, \cdots, \xi_{h_k} = b_{h_k}$ where h_i is either 0 or 1, there is a row implements that partition. So A is a (n, k)-universal set.

The (n, k)-universal set A generated in algorithm 1 is of size $\mathcal{D} = 2(n+1)^{(k-1)/2}$, which is quite large. We discuss how to construct a (n, k)-universal set of smaller size.

C. Construction of Small (n, k)-universal Sets

Using the matrix A produced by algorithm 1, we can construct a (n, k)-universal set of size much smaller than $\mathcal{D} = 2(n+1)^{(k-1)/2}$, as stated by Lemma C.1.

Lemma C.1 *[6] Let k be an odd number, $k \leq n$. There is an (n, k)-universal set of size bounded by $2ek2^k \log n$, which can be constructed in time $O(\binom{n}{k}k2^k(2n)^{(k-1)/2})$, where e is the base of the natural logarithm.*

The algorithm to construct a small (n, k)-universal set from A is listed in Fig. 2.

The (n, k)-universal set constructed by algorithm 2 is small. But the running time of algorithm 2 is too large. We will show how to construct the (n, k)-universal set faster by reducing the (n, k)-universal set problem to n $(k(k-1)+1, k)$-universal set problems.

```
Algorithm 2
input : The (0,1)-matrix $A$ from Algorithm 1
output: An $(n,k)$-universal set

1. Construct a bipartite graph $(U,V)$ where vertices in $U$ are rows in $A$ and vertices
   in $V$ are all partitions of all $k$-subsets of $S = \{\xi_1, \cdots, \xi_n\}$. There is an edge from
   $u \in U$ to $v \in V$ if and only if the corresponding row implements the corresponding
   partition of $v$.
2. Search a vertex $u \in U$ whose degree $d$ is at least $|V|/2^k$.
       Mark $u$ as selected.
       Delete all vertices $v \in V$ adjacent to $u$.
3. Repeat step 2 until $V = \emptyset$.
4. Output all the selected $u$s.
```

Fig. 2. Algorithm for constructing (n,k)-universal sets

D. Mapping n to $k(k-1)+1$

Without loss of generality, we assume that the set S is $\{1, \cdots, n\}$.

Lemma D.1 *[10] Let p be a prime such that $n \leq p < 2n$, and let W be a k-subset in Z_n. Then there is an integer z, $0 \leq z < p$, such that the function g_z over Z_n, defined as $g_z(a) = (az \mod p) \mod k^2$, is injective from W.*

While n can be very large, Lemma D.1 enables us to concentrate on (k^2,k)-universal sets since the composition of any (k^2,k)-universal set and the family of g_z for all $0 \leq z < p$ gives an (n,k)-universal set. Thus we reduce the (n,k)-universal set problem to $O(n)$ (k^2,k)-universal set problems. The total running time becomes $O(n\binom{k^2}{k}k2^k(2k)^{(k-1)})$, which is much faster than constructing the (n,k)-universal set directly by algorithm 2.

By analyzing the proof of Lemma D.1, we further discover that for any k subset of Z_n, there is a mapping that is injective from S to $Z_{k(k-1)+1}$, as stated by Lemma

D.2.

Lemma D.2 *[10] Let p be a prime such that $n \le p < 2n$, and let W be a k-subset in Z_n. Then there is an integer z, $0 \le z < p$, such that the function g_z over Z_n, defined as $g_z(a) = (az \mod p) \mod (k(k-1)+1)$, is injective from W.*

Therefore, our focus is on constructing $(k(k-1)+1, k)$-universal sets. The improvement from k^2 to $k(k-1)+1$ has reduced greatly the running time of our program. The algorithm for the reduction is given in Fig. 3.

```
Algorithm 3
input : $n$, $k$
output: $(n,k)$-universal set $US$

1. Find the smallest prime $p$, $n \le p < 2n$
2. $US = \emptyset$
3. For $z = 1$ to $p$
   $US' = \emptyset$
      For $i = 1$ to $n$
         $f(i) = (iz \mod p) \mod k(k-1)+1$
      Let $S$ be the set of $f(1), \cdots, f(n)$
      $US' \leftarrow$ the $(k(k-1)+1, k)$-universal set (for $S$) found by algorithm 1
      $US \leftarrow US \cup$(the concatenation of $f(i)$ and US')
4. Output $US$
```

Fig. 3. Faster algorithm for constructing (n, k)-universal sets

Combining all discussions above, we have an algorithm to construct an (n, k)-universal set, as stated in Theorem D.3.

Theorem D.3 *[6] Let n, k be integers and $k \le n$. There is an algorithm with running time $O(n2^{k+12log^2k+12logk}) = O(n2^{k+o(k)})$ that constructs an (n,k)-universal set of size bounded by $n2^{k+12log^2k+12logk+6} = n2^{k+o(k)}$.*

In the rest of this thesis, we will consistently use A for the matrix produced by algorithm 1, called *random space matrix*, U for the collection of splitting functions represented by the rows of A, and V for the collection of all partitions of all k subsets.

CHAPTER III

ALGORITHM VARIANTS

We design six algorithm variants to handle different k's to take full advantage of a system's memory space and computing power. They are parallelized using the OpenMP paradigm for shared memory architectures.

A. Algorithm Variant 1

Algorithm 2 is the main algorithm based on which we solve the universal set problem for small k's. How small is k to be considered solvable? Let's check the memory requirement for algorithm 2. The algorithm constructs a bipartite graph which takes time $O(k|U||V|) = O(k\binom{n}{k}2^k(2n)^{(k-1)/2})$ and assumes the bipartite graph is stored in a $|U| \times |V|$ matrix. Suppose storing each element of the matrix requires only 1 bit. The memory size required for the bipartite graph is shown in Table I.

Obviously, only $k = 3$ and 5 are practical for algorithm 2. So our first task is to avoid storing the bipartite graph. In addition, we modify algorithm 2 a bit so that it can find a smaller universal set. Algorithm 2 searches for a u that has the degree no less than $\mathcal{D}'/2^k$, where $\mathcal{D}'$ is the size of V after some vertices have been deleted. The improved algorithm searches for the u that has the largest degree.

Now let's see how to parallelize the algorithm. Obviously, the performance bottleneck for algorithm 2 is the loop from step 2 to step 3, and that is where we should focus. When searching for the v that has the largest degree, i.e., the splitting function that implements most of the partitions, we can divide V into p subsets and allocate them to p processors. Each processor takes one subset and calculate the degree of each vertex in that subset. Since calculating the degree of a vertex is independent to any other vertices, all processors can perform their assigned calculations in parallel.

Table I. Size of the bipartite graph

k	n	n'	$\|U\| =$ $2(n'+1)^{(k-1)/2}$	$\|V\| =$ $\binom{n}{k}2^k$	Memory(GB) for $(U \cup V, E)$
3	7	7	16	6720	0.000012517
5	21	31	2048	651168	0.155250549
7	43	63	524288	4124686592	251750.890625
9	73	127	536870912	49705994990080	3106624686880

A parallel version of algorithm 2 is shown in Fig. 4. In this algorithm, U is the set of all available splitting functions from A. $U(tid)$ is a subset of U that is allocated to the processor *tid*. V is stored as an $\binom{n}{k}2^n \times k$ two dimensional array, with the second dimension holding which k of the $1, \ldots, n$ are selected. *implemented* is a boolean array of size $|V|$ to indicate whether a vertex in V has been implemented by a splitting function currently available in US. *degree* is an integer array of size $|U|$. It is used to accumulate the degree of $u \in U$ that has not been selected into US in each iteration of the loop 3. Two types of comments are inserted in the algorithm to indicate which part will run in serial and which part will run in parallel.

B. Algorithm Variant 2: Random Pick Initialization

Among the data structures used in algorithm variant 1, the two dimensional array for V requires the largest memory space. We have been vague on how it is stored in previous section. Now let's check its storage in detail. Considering an integer type in a modern computer is 32 bits long, $t = n/32$ integers (or $t = n/32 + 1$ if n mod $32 > 0$) will be enough for the second dimension of V. The t integers with total

```
Algorithm Variant 1 (parallel)
input : $n$, $k$, the $(0,1)$-matrix $A$ from Algorithm 1, and $p$ processors
output: $(n,k)$-universal set $US$

(Serial region)
1. Let $US = \emptyset$, $\mathcal{D}' = |V|$
2. For each $v \in V$
2.1     $implemented(v) = FALSE$
(Parallel region)
3. For each $u \in U(tid)$
3.1.    $degree(u) = 0$
3.2.    If $u \neg\in US$ then
3.2.1.      For each $v \in V$
3.2.1.1.        If $implemented(v) == FALSE$ and $u$ implements $v$ then $degree(u)++$
(Serial region)
4. Find $u'$ that has the largest degree
5. $US = US \cup \{u'\}$
6. For each $v \in V$
6.1.    If $u$ implements $v$ then $implemented(v) = TRUE$
7. $\mathcal{D}' = \mathcal{D}' - degree(u)$
8. Repeat steps 3 to 7 until $\mathcal{D}' = 0$
9. Output $US$
```

Fig. 4. Algorithm variant 1 for the (n,k)-universal set problem

$t * 32$ bits will be treated as one long binary string. Its first n bits will be considered. One such a long binary string represents a partition of a k subset of Z_n.

With this storage method, storing information for V when $k \geq 7$ is a problem. When $k = 7$, V alone needs 30.7313G memory. Since most current SMP clusters are equipped with up to 32G memory per node, storing V and other data in the memory in one node will exceed the memory capacity.

On the other hand, time will also be a problem. We estimate how long it will take to construct a $(43,7)$-universal set based on our construction for a $(21,5)$-universal set on the supercomputer at Texas A&M University. We will give its detail configuration

Table II. Estimated wall clock time for $k = 7$

k	n	n'	$\|U\| = 2(n'+1)^{(k-1)/2}$	$\|V\| = \binom{n}{k}2^k$	Wall Clock Time
5	21	31	2048	651168	7 seconds
7	43	63	524288	4124686592	$>$ 130 days (estimated)

in Chapter V. The estimation is listed in Table II. We will need about 130 days to just finish $k = 7$. Certainly it is unacceptable.

To solve $k = 7$, we need to find ways to reduce the memory requirement and reduce the running time. We observe that if we randomly pick some vertices in U, only a small percent of vertices in V are not implemented by any of the splitting functions selected. For example, we randomly pick 200 vertices in U when $k = 7$, only about 20% partitions in V are not implemented by any of the 200 vertices. This observation helps to fully utilize 32G main memory in a node and to solve $k = 7$ quickly.

We incorporate this observation into algorithm variant 1. The resulted algorithm is listed in Fig. 5. The main improvement of algorithm variant 2 is that it initializes US with m randomly selected splitting functions from U. These splitting functions will implement part of the partitions in V. The rest of the vertices that are not implemented, small in number, will be stored in the main memory. The algorithm then searches for the vertex in U that has the largest degree in each loop iteration and includes it in US. This second half of the algorithm is same as algorithm variant 1.

The value of m is very important and is system dependent. On one hand, we don't want it to be too large to make our optimized selection less effective. One the

```
**Algorithm Variant 2 (parallel)**
input : $n$, $k$, $p$ processors, $m$, and the matrix $A$ from Algorithm 1
output: $(n, k)$-universal set $US$
(Serial region)
1. Let $US = \emptyset$, $\mathcal{D}' = |V|$
2. Randomly selected $m$ elements $U$ and put them into $US$
3. Enumerate $\binom{n}{k}$ combinations of $n$ and store in set $L$
(Parallel region)
4. for each $k$-subset $l \in L(tid)$
4.1.    Let $P$ be the set of $2^k$ possible splittings of the $k$ elements in $l$
4.2.    For each $p \in P$
4.2.1       Construct $v$ from $l$ and $p$
4.2.2       $is\_implemented = FALSE$
4.2.3       For each $u \in US$
4.2.3.1         If $u$ implements $v$ then $is\_implemented = TRUE$; break
4.2.4       If $is\_implemented == FALSE$
4.2.4.1         store $v$ in $unimplemented\_verts$
5. For each $v \in unimplemented\_verts$
5.1.    $implemented(v) = FALSE$
(Parallel region)
6. For each $u \in U$
6.1.    $degree(u)(tid) = 0$
6.2.    If $u\neg \in US$ then
6.2.1.      For each $v \in unimplemented\_verts$
6.2.1.1.        If $implemented(v) == FALSE$ and $u$ implements $v$ then $degree(u)++$
(Serial region)
7. For $i = 1$ to $p$
7.1.    $tot\_degree(u) = tot\_degree(u) + degree(u)(tid)$
8 Find $u'$ that has the largest degree from $tot\_degree(u)$
9. $US = US \cup \{u'\}$
(Parallel region)
10. for each $k$-subset $l \in L(tid)$
10.1.   Let $P$ be the set of $2^k$ possible splittings of the $k$ elements in $l$
10.2.   For each $p \in P$
10.2.1.     Construct $v$ from $l$ and $p$
10.2.2.     If $u$ implements $v$ then $implemented(v) = TRUE$
(Serial Region)
11. $\mathcal{D}' = \mathcal{D}' - tot\_degree(u)$
12. Repeat step 5 until $\mathcal{D}' = 0$
13. Output $US$
```

Fig. 5. Algorithm variant 2 for the (n, k)-universal set problem

other hand, we don't want it to be too small so that the un-implemented partitions in V cannot be stored in the main memory.

C. Algorithm Variants 3

Algorithm variant 1 and 2 can only work for odd k's since they use the random space matrix A as their base universal set where an optimized universal set will be selected. We also design a third variant that is almost the same as variant 1 except that it generates a base set of splitting functions R randomly. The same optimization technique is used to search R, i.e., looking for the vertex that with the largest degree to include in the universal set in construction. Unlike algorithm variant 1 where A is already a universal set, R may or may not be. So there are two search outcomes: a universal set is found from R and the program stops, or some $v \in V$ are not implemented. In the second case, the algorithm repeatedly generates a new random splitting function set and optimizes it until all partitions are implemented. The advantage of algorithm variant 3 as compared to variant 1 and 2 is that it does not require the parity of k and can be used to construct universal set for $k = 4$ and 6.

D. Algorithm Variants for $k \geq 8$

None of above algorithms can be used for $k \geq 8$. Over the course of our research, we have designed three algorithms to handle this case, with much improvement from the first one to the last one. We will provide the results in chapter V.

The first one is a simplified version of algorithm 3. It divides k into almost two even parts: k_1 and k_2. The product of an (n_1, k_1)-universal set and an (n_2, k_2)-universal set are calculated for all $k_1 \leq n_1 \leq n - k_2 + 1$ and $n_2 = n - n_1$. The union of all the production forms an (n, k)-universal set. The algorithm is listed in Fig. 6.

It uses the mapping algorithm listed in Fig. 3.

```
Algorithm Variant 4
input : $n$, $k$
output: $(n,k)$-universal set $US$

1. Let $US = \emptyset$
2. Divide $k$ into $k_1$ and $k_2$, $|k_2 - k_1| \leq 1$
3. $US_1 = us(k_1(k_1-1)+1, k_1)$; $US_2 = us(k_2(k_2-1)+1, k_2)$
4. For $n' = k_1$ to $n - k_2 + 1$
4.1      If $n' \leq k_1(k_1-1)+1$ then
4.1.1        $US_1' = US_1$ (choose the first $n'$ bits of each element in $US_1$)
4.2      Else $US_1' = Mapping(n', k_1, US_1)$
4.3      If $(n-n') \leq k_2(k_2-1)+1$ then
4.3.1        $US_2' = US_2$ (choose the first $(n-n')$ bits of each element in $US_2$)
4.4      Else $US_2' = Mapping((n-n'), k_2, US_2)$
4.5      $US = US \cup (US_1' \times US_2')$
5. Output $US$
```

Fig. 6. Algorithm variant 4 for the (n,k)-universal set problem

The universal set found by algorithm variant 4 is very big. We improve it by avoid mapping used in step 4.2 and 4.4 since mapping will generate a much larger universal set than constructing it from scratch. Fig 7. lists the slightly improved algorithm.

Algorithm variant 5 can generate smaller universal sets than variant 4. For example, it decreases the size of an $(57,8)$-universal set from 60,750,322 to 30,358,244. However, the size is still too large. Our last algorithm is designed to build an $(57,8)$-universal set that will be 10,000 times smaller.

The idea for the algorithm starts with our observation while using algorithm variant 5 to calculate a $(57,8)$-universal set. In that construction, we use $(n,4)$-universal

```
Algorithm Variant 5
input : $n$, $k$
output: $(n, k)$-universal set $US$

1. Let $US = \emptyset$
2. Divide $k$ into $k_1$ and $k_2$, $|k_2 - k_1| \leq 1$
3. For $n' = k_1$ to $n - k_2 + 1$
3.1     $US_1 = (n', k_1)$-universal set
3.2     $US_2 = ((n - n'), k_2)$-universal set
3.3     $US = US \cup (US_1' \times US_2')$
4. Output $US$
```

Fig. 7. Algorithm variant 5 for the (n, k)-universal set problem

sets to synthesize $(57, 8)$-universal set. As a preparation, we need to construct an $(n, 4)$-universal set for $n = 4, \ldots, 53$. Table III lists the size of some of the resulted universal sets.

An observation from Table III is that while n increases, the size of the universal set for the same k increases too, but not dramatically. The larger the n, the slower the size increases. So, if we can construct an (n', k)-universal set $(n' < n)$ using any of the algorithm variants 1, 2, or 3, we can have a good guess on the size of an (n, k)-universal set. For example, we have constructed a $(24, 8)$-universal set of size 2100 using algorithm variant 3, we speculate a $(57, 8)$-universal set will not exceed 10,000.

Now the critical question is how to construct an (n'', k)-universal set from an (n', k)-universal set given $n'' > n'$. Since $\binom{n'}{k}$ is a subset of $\binom{n''}{k}$, we want the information contained in the (n', k)-universal set to be fully utilized. Our method satisfying this requirement is to extend the length of each binary string in the (n', k)-universal set from n' bits to n'' bits. The first n' bits are kept the same and the extended $n'' - n'$

Table III. Size of $(n, 4)$-universal sets for $n = 4, 7, \ldots, 53$

n	size	n	size	n	size
4	16	22	74	40	111
7	34	25	81	43	116
10	37	28	86	46	119
13	50	31	93	49	126
16	58	34	100	52	128
19	68	37	105	53	130

bits are filled in randomly.

The idea is implemented in algorithm variant 6 listed in Fig. 8. It is parallelized with OpenMP. But for simplicity, we omit the details on how it is parallelized as it is similar to the approach in algorithm variant 2.

This algorithm has two new input parameters: n_{start} is provided to control the initial universal set and *step_size* is used to control the construction time. Its effectiveness will be checked in Chapter V. Since the method in algorithm variant 6 progressively constructs a larger universal set from a smaller and easy to build universal set, we name it *progressive construction*.

```
**Algorithm Variant 6 (parallel)**
input : $n$, $n_{start}$, $step\_size$, $k$, $p$ processors
output: $(n, k)$-universal set $US$.
1. $n' = n_{start}$
2. While $n' < n$ do
2.1      $US = (n', k)$-universal set
2.2      For each binary string $b = b_1 \ldots b_{n'} \in US$
2.2.1        Extend $b$ to $b_1 \ldots b_{n'} \ldots b_{n'+step_size}$,
             the extra $step\_size$ bits are randomly filled with 0 or 1.
2.3      Check against $US$ to find $Q$ contains all $v \in V$ that are not implemented.
2.4      Randomly generate and optimize a splitting functions set $R$ for $Q$.
2.5      $US = US \cup R$
2.6      $n' = n' + step\_size$
3. return $US$
```

Fig. 8. Algorithm variant 6 for the (n, k)-universal set problem

CHAPTER IV

DESIGN OF UNIVERSAL SET CONSTRUCTION AND RETRIEVAL SYSTEM

We have introduced in previous chapter the algorithm variants used to construct universal sets for various k's. In this chapter, we first describe our goals for designing a universal set construction and retrieval system, and then describe its components in detail.

A. Design Goals

Our design for the universal set construction and retrieval system is driven by four goals. First and the most important, we want that the size of (n, k)-universal sets constructed by our system is small. By small, we mean it is less than the theoretical upper bound $2k2^k \ln(n + 1)$. Second, we want our system to run fast so that it can find an (n, k)-universal set for a relatively large k in a reasonable amount of time. Third, our design should be extensible to use new universal set construction algorithms. Lastly, Our design should provide convenient interface for others to access our universal sets.

B. Universal Set Construction and Retrieval System Structure

We design a scalable universal set construction and retrieval system (USCRS) based on the four goals stated in previous section. The system can build (n, k)-universal sets for $k = 3, \ldots, 8$, stores them into a database called universal sets database (USDB), and provides an interface for universal set retrieval. The system architecture is shown in Fig. 9.

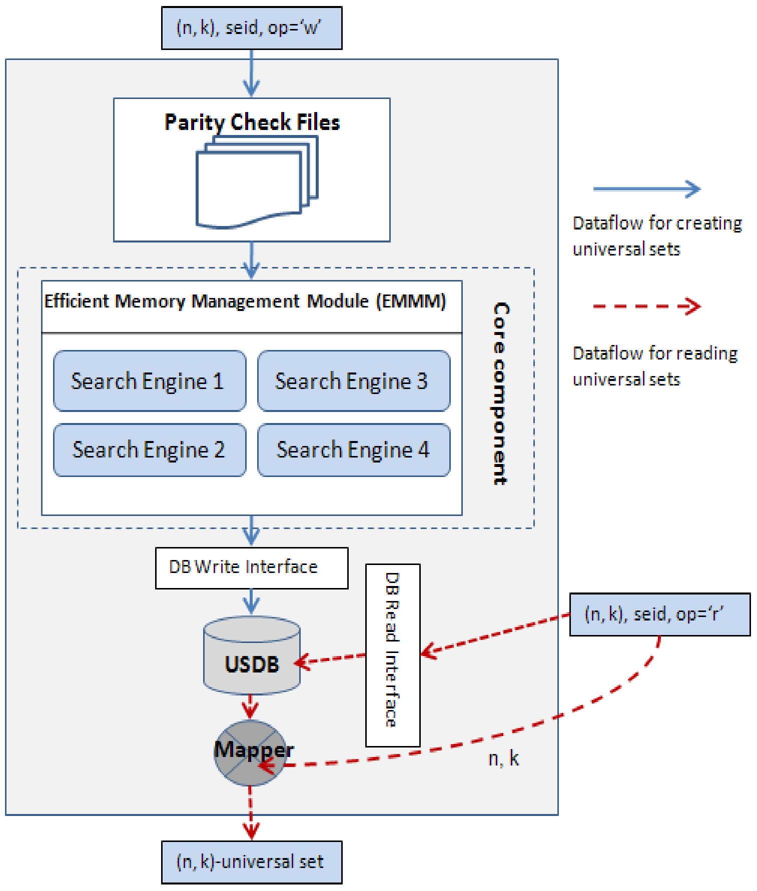


Fig. 9. Universal set construction and retrieval system architecture

1. Core Component

The core of our design includes an efficient memory management module and four universal set parallel search engines.

a. Parallel Search Engines

As k increases from 3 to 8, the amount of time and space required for algorithm 2 to build a universal set increases dramatically. It is more efficient to design different algorithms to handle different cases. In chapter 3, we presented six algorithms for different k's. Four of them (algorithm variant 1, 2, 3, and 6) are used in USCRS.

Each algorithm is implemented as a standalone subroutine called a search engine. The name comes from the face that in each algorithm, a universal set is constructed by **searching** a splitting function set, which is either a random space universal set or a randomly generated universal set. The searching process of each search engine is described below:

1. **Search engine 1** implements algorithm variant 1 and is invoked when $k = 3$ or 5 and $seid = 1$. It constructs a $(7, 3)$-universal set or a $(21, 5)$-universal set in a loop. In each loop iteration, it searches the random space universal set the splitting function that satisfies two criteria: 1. it is not currently selected in the universal set in construction; 2. it implements the most number of partitions. The splitting function is selected to the universal set and partitions implemented by it are deleted. The loop stops when no partitions are left.

2. **Search engine 2** implements algorithm variant 2 and is invoked when $k = 7$ and $seid = 2$ to build a $(43, 7)$-universal set. It first randomly picks 200 splitting functions from the random space universal set. It then checks all the partitions for all 7-subset of Z_{43} and stores the partitions that are not implemented by

any of the 200 splitting functions in the main memory. The rest of its operation is same as search engine 1. The initial size 200 is picked empirically based on our experiment that will discussed in chapter V.

3. **Search engine 3** implements algorithm variant 3 and is invoked when $k = 4$ or 6 and $seid = 3$. It first randomly generates a splitting function set and then do similar operation as search engine 1. The main difference between search engine 3 and 1 is that 3 deals with even k's and searches a randomly generated universal set.

4. **Search engine 4** implements algorithm variant 6 and is invoked when $k = 8$ and $seid = 4$. It first calls search engine 3 to construct a (21,8)-universal set, followed by constructing of (26,8)-, (31,8)-, (36,8)-, (47,8)-, and (57,8)-universal set through the use of *progressive construction*.

This part of the core can be easily extended by adding new search engines that implements new algorithms. We allow different search engines to construct universal sets for the same k. They are differentiated by *seid*.

b. Efficient Memory Management Module

As we discussed in chapter III, when $k \geq 7$, we should be careful on memory usage. For this reason, we store each binary using one bit in the memory, instead of using a bool type. This will save us about 1/8 of the required memory. Representing a binary string in this condensed form makes equality comparison between two binary strings a comparison between two integers. This makes our program run a lot faster since the searching process consists of a large number of equality comparisons. But since bit is not a natural data type in a computer system, it adds programming complexity.

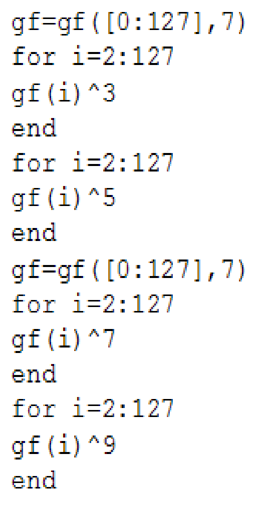
EMMM is responsible to handle the added complexity among other tasks related to efficient memory usage. These include:

- Allocates and de-allocates memory space.
- Dynamically enlarges a table size when necessary.
- Constructs H from a parity check file.
- Constructs M out of H.
- Stores a $0, 1$-matrix into binary strings.
- Converts between a binary string and its bit representation.
- Extends the length of a bit-represented binary string and fills the extended spots with random bits.

2. Parity Check Files

Given n and k ($k \leq n$, and k is odd), the construction of the parity check matrix H in algorithm 1 is non-trivial. Let the field $F = GF(n' + 1)$ ($n' = 2^d - 1$, where d is the smallest integer satisfying $n \leq n'$). We need to calculate x_i^t over the field F, where $t = 1, 3, \ldots, k - 2$, for every $x_i \in F$ and $x_i \neq 0$. Fortunately, Matlab handles the calculation easily. Figure 10 is a matlab script for calculating GF(128) and its powers for all non-zero members in the field.

The output from matlab is in decimal. We only store the first n numbers and their powers into a file called a parity check file. The file serves as a prototype for creating a parity check matrix H that is used in the algorithms for universal set construction. For each odd k from 3 to 15, we create a parity check file. One file will be selected according to the input values of k.



```
gf=gf([0:127],7)
for i=2:127
gf(i)^3
end
for i=2:127
gf(i)^5
end
gf=gf([0:127],7)
for i=2:127
gf(i)^7
end
for i=2:127
gf(i)^9
end
```

Fig. 10. A sample matlab script for finding GF(128) and its odd powers

3. USDB

Each (n, k)-universal set is stored in a text file whose name is indexed by the values of n, k, and *seid* of the search engine that constructs the universal set. Our USDB is a centralized place holder for all the text files for the universal sets. We choose text files to store our universal sets since our data size is small and it is easy to provide web based interface for accessing any universal set for given n and k.

4. Mapper

For given n and k, the search engines can only provide a $(k(k-1)+1, k)$-universal set. When $n < (k(k-1)+1, k)$, the first n bits of each binary string of the $(k(k-1)+1, k)$-universal set are returned to form an (n, k)-universal set. When $n > (k(k-1)+1, k)$, things become a little more complicated. The component *mapper* is used to handle this complexity. There are two reasons for not providing universal sets for an arbitrarily large n directly. First, as n increases, the computational time (n, k)-universal set increases quickly. Therefore, we cannot afford for an arbitrarily large

n. second, Lemma 3 and 4 guarantee to find an (n, k)-universal set whose size is no more than $2n$ times the size of the $(k(k-1)+1, k)$-universal set. *mapper* implements the mapping algorithms listed in Fig. 3.

C. USDB Read and Write Procedures

USCRS takes four inputs from a user: n, k, *seid* for search engine id, *op* for operation type that can be 'w' for write or 'r' for read.

When *op*='w', the universal set construction process begins. If k is odd, a parity check file is picked and passed to the core; otherwise, a null handler will be passed to the core. Next, EMMM checks the parity check file handler. If the handler is not null, EMMM reads the file and generates the parity check matrix H and then generates the random space universal set M. A search engine will be invoked according to the value of *seid*. The search engine will first check if k belongs its cases. If no, USCRS returns with an error message. If yes, the search engine with *seid* is invoked to construct an (n, k)-universal set. Through the DB write interface, the universal set is written into a file stored in USDB.

When *op*='r', the universal set retrieval process starts. The parameters are passed to the DB read interface, which finds the universal set in the USDB for the given k, and *seid*. The (n', k)-universal set $(n' = (k(k-1)+1)$, if exists, is sent to the mapper which will create an (n, k)-universal set as output. *seid* can be '*' when *op*='r', which means returns any universal sets for the given n and k.

CHAPTER V

EXPERIMENTS

In this charter, we present our test results on the algorithms we proposed and implemented.

A. Computing Platforms

Most of our experiments are done on hydra, currently the most powerful university supercomputer serving the Texas A&M research community. Hydra is an IBM power5+ cluster of 52 p5-575 nodes. Each node is an SMP (Shared-memory Multi Processor) machine with 16 IBM Power5+ processors running at 1.9GHz and sharing 32G or 64G memory. 48 of the nodes are interconnected via the high performance communication switch (HPS) that enables fast intro-node communication. Our parallel search engines are implemented using OpenMP and thus can run on up to 16 CPUs at the same time.

A much detailed system configuration is listed in Table IV.

The last two steps in the construction of a $(57, 8)$-universal set are done on a dual core Linux workstation. Its system configuration is listed in Table V.

B. Experiments

In this section, we explore answers to the following questions:

- What is the wall time for constructing universal sets for $k = 3, \dots, 8$ using our algorithms? What is the size of the universal sets?
- How to select the *initial size* for *random pick initialization*?
- How effective is *progressive construction*?

Table IV. Hydra system configuration

Operating system	AIX5.3 with parallel environment
Number of nodes	52
Total number of processors	832 1.9GHz Power5+
Cores/node	16
Memory/node	32G*
Cache configuration	64K instruction and 32K data L1/core, 1.875M unified L2/core, 39M unified L3/dual-cores

*Three nodes have 64G memory

Table V. Linux workstation testbed system configuration

Operating system	SUSE Linux 10.2
Number of nodes	1
Total number of processors	2 Intel core(TM)2 E8400 3.0GHz
Cores/node	2
Memory/node	4G
Cache configuration	6M L2

Table VI. Size of constructed universal sets

k	n	actual size	theoretical size	wall clock time (seconds)	algorithm
3	7	14	99	< 1	variant 1
4	13	49	-	< 1	variant 3
5	21	166	974	5	variant 1
6	31	612	-	670	variant 3
7	43	1576	6781	8954	variant 2
8	57	4611	-	*	variant 6

*The construction of the (57,8)-universal set is done on both hydra and the Linux workstation. See Table VII for wall clock time consumed

- What is the scalability of our OpenMP parallel program?
- How good is randomness?

1. A Glimpse of Constructed Universal Sets

We have constructed universal sets for $k = 3, \ldots, 8$, and stored them in the USDB. The size of each universal set is listed in Table VI. The column *theoretical size* is calculated using the theoretical upper bound $2k2^k \ln(n+1)$ for odd k's. It gives some idea how good our universal sets are. The wall time in seconds is measured when running our program on 16 processors on hydra, except for the case $k = 8$, which we will discuss in more detail later. The algorithm that is used to construct each universal set is also specified in Table VI.

In general, we consider our universal sets are small as they are far less than the theoretical upper bound.

2. Initial Size for Random Pick Initialization

Algorithm variant 2 is designed to construct a $(43, 7)$-universal set. It uses *random pick initialization* to initialize the universal set with m randomly selected splitting functions from the random space universal set. m is called the initial size. In this experiment, we examine how m affects the number of partitions implemented by the m splitting functions and what is the best m for constructing a $(43, 7)$-universal set on hydra.

We test $m = 50, 100, \ldots, 300$ and record the total number of partitions that are **not** implemented by the m randomly selected splitting functions. For each m, we run our program five times.

Fig. 11 shows the total number of partitions that are not implemented for m =50, 100, 150, 200, 250, and 300 in five tests. Two things call our attention. First, for the same m, the number of un-implemented partitions in the five tests doesn't vary much. This is what we expected since the m splitting functions are randomly picked from the random space universal set. Second, tests for m =200, 250, and 300 result in bars with almost the same height. This means the randomly picked splitting functions beyond the first 200 contribute only a few implemented partitions.

In Fig. 12, the percent of partitions that are not implemented (averaged from the five tests) for each case is plotted. When m increases from 50 to 300, the percentage decreases quickly until $m = 200$. After that point, it almost stays the same.

Fig. 12 provides guidance on how to choose m. m can be any number from 0 to 200 depending on available main memory on a system. On hydra, m can be 200.

For comparison, we also conduct a similar experiment with randomly generated splitting functions. In this experiment, we randomly generate m splitting functions and record how many partitions are not implemented by the m splitting functions.

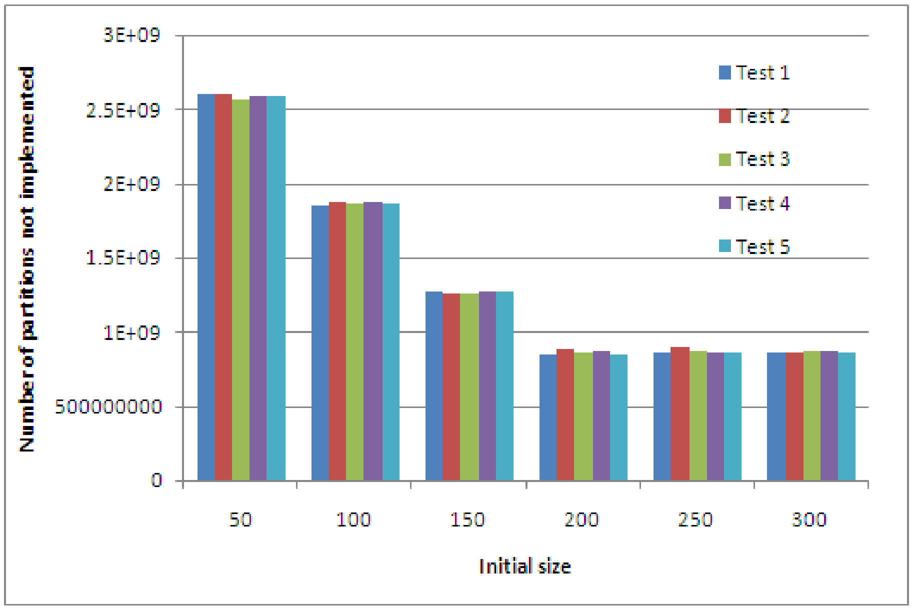


Fig. 11. Number of partitions not implemented

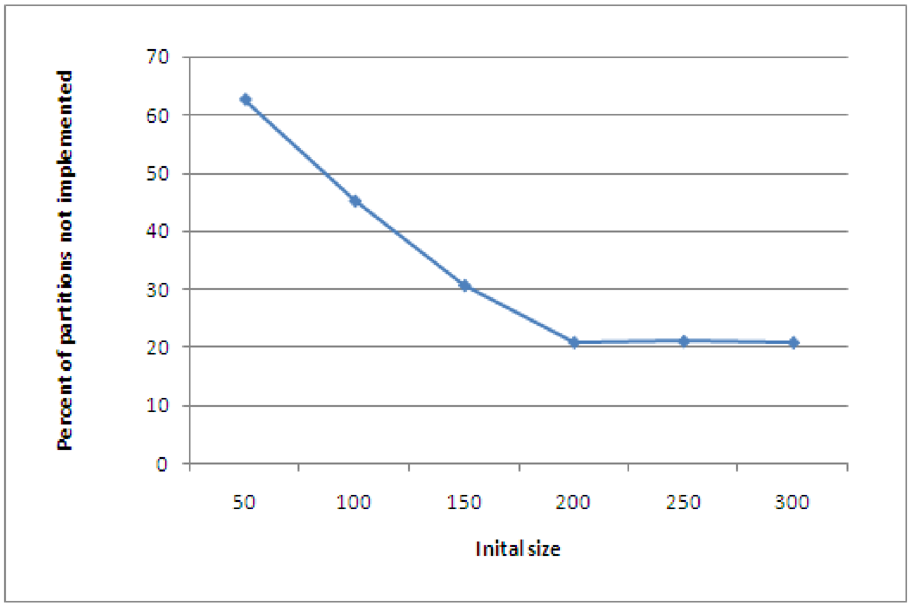


Fig. 12. Percent of partitions not implemented

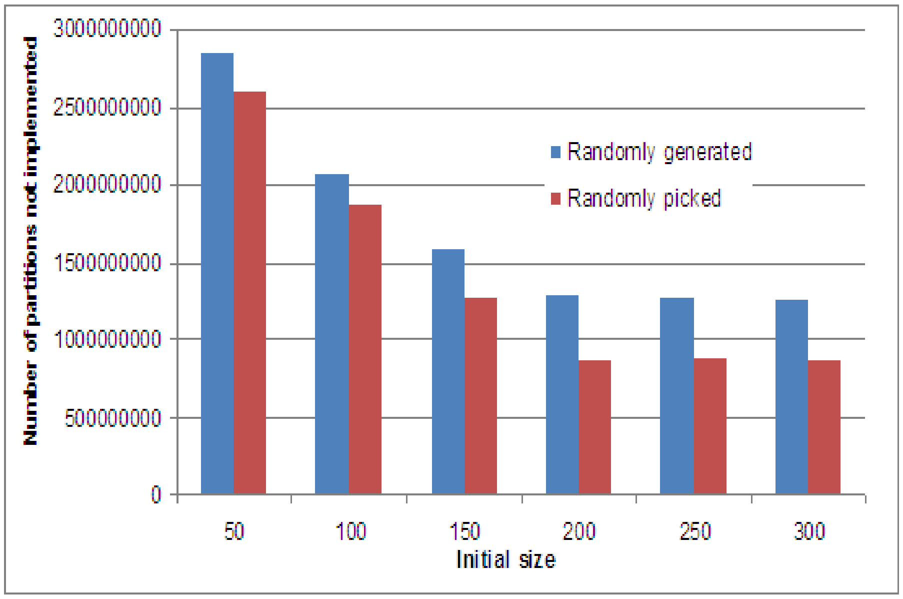


Fig. 13. Comparison of randomly generated splitting function and randomly picked splitting function on the number of partitions that are not implemented

Fig. 13 and Fig. 14 show that the number or percent of partitions that are not implemented by randomly generated splitting functions follow the same trend as Fig. 11 and Fig. 12 respectively. Moreover, randomly generated splitting functions can implement less partitions than the same number of randomly picked splitting functions.

3. Effectiveness of Progressive Construction

The *progressive construction* method used in algorithm variant 6 is by far our best algorithm to build a (57,8)-universal set. This algorithm takes six steps to finish the construction. Table VII lists the universal set size and wall clock time for each step. The first step constructs a (24,8)-universal set by using algorithm variant 3. The second step constructs a (29,8)-universal set based on the first step, and so on, until an (57,8)-universal set is constructed.

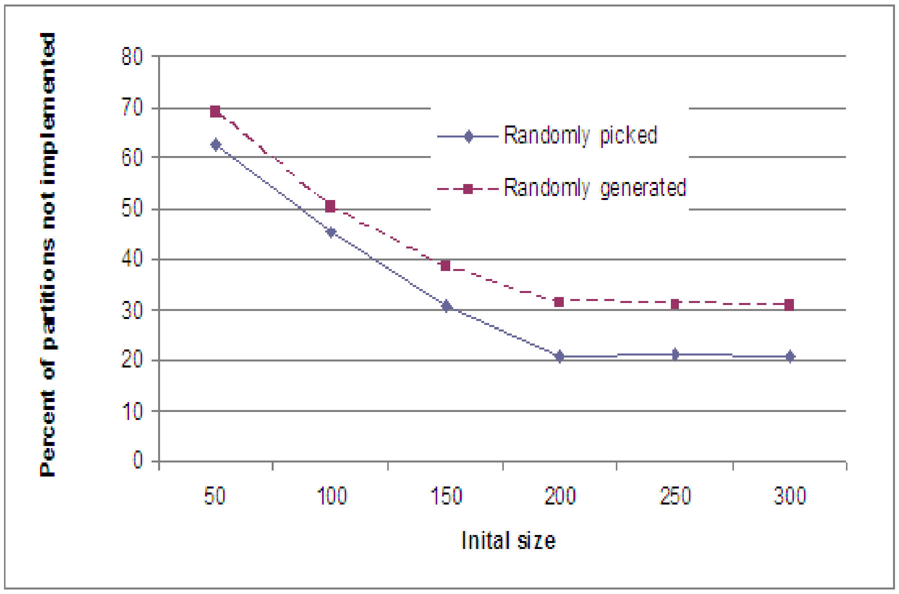


Fig. 14. Comparison of randomly generated splitting function and randomly picked splitting function on percent of partitions that are not implemented

Table VII. Step by step construction of a (57,8)-universal set

step#	n	size	wall clock time
1	24	2100	2m32s
2	29	2556	7m5s
3	34	2998	26m44s
4	39	3427	1h21m31s
5	47	4168	15h21m26s
6	57	4611	4d21h21m11s

Step 1-4 are done on hydra. Step 5 and 6 are done on the Linux workstation

Table VIII. Performance comparison of algorithm variant 4, 5, and 6

Algorithm variant	size
4	60750322
5	30358244
6	4611

This algorithm works very well for the (57,8)-universal set, as compared to algorithm variant 4 and 5. However, it also consumes a lot more time than the other two algorithms. Table VIII lists size of the universal set constructed by each algorithm.

4. Scalability

There are two major operations in our algorithms: checking for partitions that are not implemented and searching for an optimum universal set. Algorithm variant 1 and 3 contains only searching operation, while algorithm variant 2 and 8 contains both checking and searching. In all algorithms, over 99% of time is spent on doing searching or both checking and searching. We have parallelized both operations with OpenMP. In this section, we are interested in checking how well our program scales after the parallelization. We need to be careful on choosing which case for the scalability test. It shouldn't be too small to give inaccurate scalability, or too large to spent too long to get results. We choose $k = 7$ for the right amount of time needed to construct a (43,7)-universal set.

The scalability of our program is shown in Fig. 15. The program scales very well up to 16 processors.

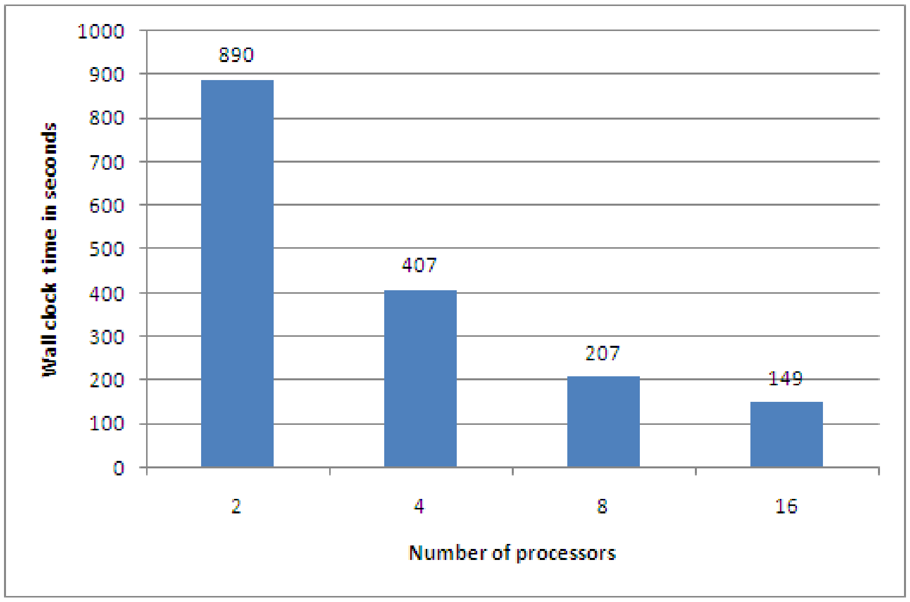


Fig. 15. Scalability of the OpenMP program on 2, 4, 8, and 16 processors

5. Is Randomness Alone Enough?

We have used randomness in different ways in our algorithms: in algorithm variant 2, we *randomly* select some splitting functions from the random space universal set; in algorithm variant 3 and 6, we *randomly* generate a universal set; and in algorithm variant 6, we fill in the extended bits with *random* bits. Randomness has done very good work.

Since algorithm variant 3 doesn't require the parity of k, we are interested to find out how the algorithm based on pure randomly generated splitting functions will work for odd k's. We pick $k = 5$ for the test. $k = 3$ is excluded because it is too small. We run the program 50 times and record the universal set sizes and plot them in Fig. 16. The smallest size we get is 175 and largest is 181. The red line represents the size from algorithm variant 1, which is 166.

The randomly generated splitting function set in previous test has a size of 1500,

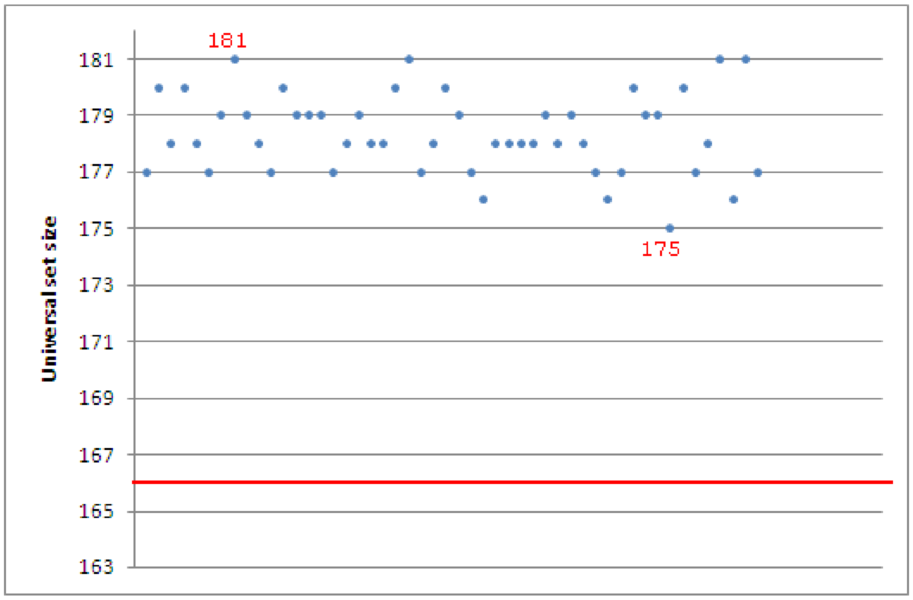


Fig. 16. Size of (21,5)-universal sets constructed from randomly generated splitting function sets, as compared with the size of the (21,5)-universal set constructed from the random space universal set. The graph shows results for 50 tests

while the random space universal set has a size of 2048. Now we change the size of the randomly generated universal set and check if this will have any influence. We test additional sizes of 2000, 2500, and 3000. For each size, we run the program 10 times. The results are plotted in Fig. 17. The graph shows, with larger search base, the algorithm has no improvement on the size of the universal set it constructs.

Obvious, the algorithm based on the random space universal set works better. This justifies the existence of algorithm variant 1.

On the other hand, we cannot eliminate algorithm based on randomly generated universal set too. This is because without this algorithm, we need to use the universal set for $k + 1$ for even k's. This method usually results in a larger universal set than constructing from a randomly generated universal set.

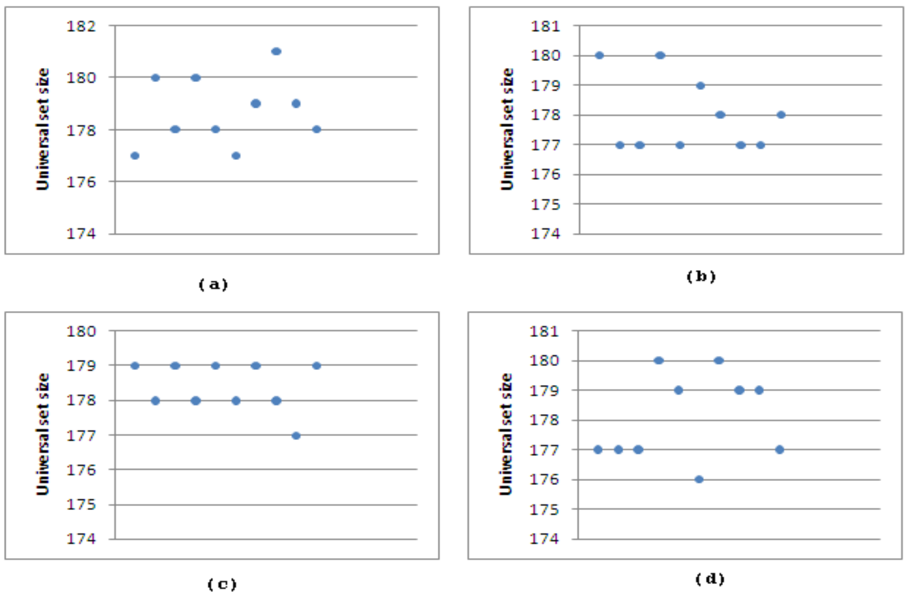


Fig. 17. Size of (21,5)-universal sets constructed from randomly generated splitting function sets. Let m be the size of a randomly generated splitting function set. (a) $m = 1500$. (b) $m = 2000$. (c) $m = 2500$ (d) $m = 3000$. Each subgraph contains 10 test results

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

A. Conclusions

In this thesis, we proposed and implemented effective algorithms for calculating universal sets for $k \leq 8$. We successfully constructed a $(43, 7)$-universal set of size 1576 and a $(57, 8)$-universal set of size 4611, which we consider small. The main heuristic techniques that help us achieve such small universal sets are called *random pick initialization* and *progressive construction*.

We demonstrated that the universal set problem is very suitable to a parallel computing environment. Our experiment shows that the problem implemented with OpenMP scales well up to 16 CPUs.

We also proposed a universal set construction and retrieval framework. It has two prominent features: first it has a scalable design and allows easy add-ons of new search engines that implement new algorithms for constructing universal sets; second, it has a centralized database that stores the constructed universal sets and provides a simple interface for retrieving the universal sets. The expandability and re-usability of our system can become a useful resource for any researchers who are interested in studying the universal set problem and who are interested in using universal sets to construct solutions for other problems that have wide applications.

B. Future Work

Our implementation, though effective, can definitely be improved. In future work, we are interested in at least two directions.

First, our parallel implementation of the algorithms uses the OpenMP interface, which is simple but cannot take the full power of a cluster. As we explained earlier, we chose OpenMP because of our time limit. Another commonly used parallel technique is MPI, which can use many more processors, depending on the scalability of the problem and the available processors. Through the OpenMP implementation, we have demonstrated the universal set problem can make very good use of as many as 16 CPUs. This shows that the problem exhibits very good parallelism. Thus, it is worthwhile to implement the universal set problem using MPI. The MPI implementation can overshadow the OpenMP implementation with two factors: more processors means more computing power; more computing nodes involved means more memory space. As a result, the MPI implementation not only enable us to solve larger k's (9 and 10), but also can improve universal set size for smaller k. For example, $k = 7$ can be solved using algorithm variant 1 to find an optimal solution.

Second, we have established a database that contains the universal sets for $k \leq 8$. We need to extend it to include $k = 9, 10$ and larger. Although algorithm 6 allows us to construct universal sets for any $k > 8$, we really don't want to do that as the universal set size is far beyond desirable. With MPI implementation, we can aim for larger k's. It is not difficult for us to construct universal sets for $k = 9, 10$ with current techniques. However, for $k > 10$, new heuristics need to be invented.

REFERENCES

[1] N. Alon, L. Babai, and A. Itai, "A fast and simple reandomized parallel algorithm for the maximal independent set problem," *Journal of Algorithms*, vol. 7, pp. 567–683, 1986.

[2] N. Alon, R. Yuster, and U. Zwick, "Color-coding," *Journal of the ACM*, vol. 42, pp. 844–856, 1995.

[3] G. Bai and H. Fernau, "Constraint bipartite vertex cover simpler exact algorithms and implementations," in *Proceedings of the 2nd Annual International Workshop on Frontiers in Algorithmics*, 2008, pp. 67–78.

[4] H. Bodlaender, "On linear time minor tests with depth-first search," *Journal of Algorithms*, vol. 14, pp. 1–23, 1993.

[5] J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon, "Solving large fpt problems on coarse-grained parallel machines," *Journal of Computer and System Sciences*, vol. 67, no. 4, pp. 691–706, 2003.

[6] J. Chen and S. Lu, "Improved parameterized set splitting algorithms: A probabilistic approach," in *Proceedings of the 13th Annual International Computing and Combinatorics Conference*, 2007, pp. 537–547.

[7] J. Chen, S. S. S. Lu, and F. Zhang, "Improved algorithms for path, matching and packing problems," in *Proceedings of the 18th Annual ACM Symposium on Discrete Algorithms*, 2007, pp. 298–307.

[8] R. G. Downey and M. R. Fellows, *Parameterized Complexity.* New York, USA: Springer, 1999.

[9] F. Hüffner and S. Wernicke and T. Zichner, "Algorithm engineering for color-coding with applications to signaling pathway detection," *Algorithmica*, vol. 52, pp. 114–132, 2008.

[10] M. Fredman, J. Komlos, and E. Szemeredi, "Storing a sparse table with $O(1)$ worst case access time," *Journal of ACM*, vol. 31, pp. 538–554, 1984.

[11] B. P. Kelley, R. Sharan, R. M. Karp, T. Sittler, D. E. Root, B. R. Stockwell, and T. Ideker, "Conserved pathways within bacteria and yeast as revealed by global protein network alignment," in *Proceedings of National Academy of Sciences USA, 100*, 2003, pp. 11 394–11 399.

[12] J. Kneis, D. Molle, S. Richter, and P. Rossmanith, "Divide-and-color," in *Lecture Notes in Computer Science*. New York, USA: Springer, 2006, vol. 4271, pp. 58–67.

[13] I. Koutis, "Faster algebraic algorithms for path and packing problems," in *Proceeding of the 35th International Colloquium on Automata, Languages and Programming*, 2008, pp. 575–586.

[14] R. Motwni and P. Raghavan, *Randomized Algorithms*. New York, USA: Cambridge University Press, 1995.

[15] J. Scott, T. Ideker, R. M. Karp, and R. Sharan, "Efficient algorithms for detecting signaling pathways in protein interaction networks," *Journal of Computational Biology*, vol. 13, pp. 133–144, 2006.

[16] S. B. Wicker, *Error Control Systems for Digital Communication Storage*. New Jersey, USA: Prentice-Hall, Inc., 1995.

[17] R. Williams, "Finding paths of length k in $O^*(2^k)$," *Information Processing Letters*, vol. 109, no. 6, pp. 315–318, 2009.

VITA

Name:	Ping Luo
Address:	Department of Computer Science
	c/o Jianer Chen
	Texas A&M University
	College Station, TX 77843
Email:	pingluo@tamu.edu
Education:	M.S. in Mathematics, Indiana State University, 2003
	B.S. in Computer Science and Engineering, University of Electronic Science and Engineering, 1997

The typist for this thesis was Ping Luo.