Software for supporting large scale data processing for High Throughput Screening

By

David Tai

Submitted to the graduate degree program in Department of Electrical Engineering and Computer Science and the Graduate Faculty of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

_____

Chairperson Jianwen Fang

_____

Jun Huan

_____

Brian Potetz

Date Defended: 6th December 2011

The Thesis Committee for David Tai

certifies that this is the approved version of the following thesis:

Software for supporting large scale data processing for High Throughput Screening

_____

Chairperson Jianwen Fang

Date approved: 6th December 2011

Abstract

High Throughput Screening for is a valuable data generation technique for data driven knowledge discovery. Because the rate of data generation is so great, it is a challenge to cope with the demands of post experiment data analysis. This thesis presents three software solutions that I implemented in an attempt to alleviate this problem. The first is K-Screen, a Laboratory Information Management System designed to handle and visualize large High Throughput Screening datasets. K-Screen is being successfully used by the University of Kansas High Throughput Screening Laboratory to better organize and visualize their data.

The next two algorithms are designed to accelerate the search times for chemical similarity searches using 1-dimensional fingerprints. The first algorithm balances information content in bit strings to attempt to find more optimal ordering and segmentation patterns for chemical fingerprints. The second algorithm eliminates redundant pruning calculations for large batch chemical similarity searches and shows a 250% improvement for the fastest current fingerprint search algorithm for large batch queries.

Acknowledgements

First and foremost, I would like to thank my advisor Dr. Jianwen Fang. He has without a doubt provided me with the help, guidance, support, and encouragement without which I would not have been able to complete this degree program. I would also like to thank Dr. Luke Huan and Dr. Brian Potetz for teaching courses valuable to my research and knowledge in addition to serving on my thesis committee. Lastly, I am grateful for the support and encouragement of my friends and family who have assisted and put up with me through the various challenges I had have to overcome during the last two years.

Contents

Chapter 1: Introduction

High throughput screening (HTS) has emerged as an important technique for allowing researchers to rapidly profile very large numbers of chemicals against drug targets. However, while development and investment of resources into HTS technologies has resulted in great advances to speed and accuracy, tools supporting HTS technologies are still trying to keep up with the increasing demands for efficient HTS data analysis. The purpose of this thesis is to describe the tools and novels algorithms I have created as part of my graduate work for better supporting HTS operations.

Chapter 2 consists of the description of K-Screen, a Laboratory Information Management System for High Throughput Screening designed primarily to efficiently and coherently present data to clients. It also includes a detailed discussion of the design and features present in both the backend server technology and the front-end user interface.

Chapter 3 discusses two novel algorithms for 1-dimensional chemical fingerprint searches. This chapter consists of a literature review of existing chemical fingerprint search algorithms, a discussion and analysis of fingerprint representation in Java, a discussion and analysis of a novel information content-based bit reordering and bit string segmentation algorithm, and a discussion of the a novel search method for chemical fingerprint searching using indexed query sets.

Chapter 2: K-Screen

2.1 Overview

As recent and future advances make HTS cheaper to perform on even larger scales, the amount of data that has to be processed, analyzed, and searched will only grow larger in size and harder for researchers to manually sift through. It is therefore an eventuality that institutions utilizing HTS technology and techniques will need to begin looking for effective solutions in the maturing area of laboratory information management systems like many other types of labs have already done.

K-Screen is one such solution. Our initial goal with K-Screen was to have an integrated application environment that supported data analysis, management, and presentation so we could efficiently perform client requested screens and searches as well as generate detailed reports on the results of those. Previously, we had attempted but failed to locate an existing software suite that sufficiently addressed all our requirements.

K-Screen is a web accessible application that offers the ability to host a large chemical structure library, process and store single-dose (primary) and dose response (secondary) screening data, perform searches based on screening results, plate coordinates, and structure, substructure and structure similarity. It uses heat maps and histograms to visualize screen or plate level statistics. Interfaces to external searches against PubChem and ZINC databases are also provided. We feel that these features make K-Screen a practical and effective alternative to other commercial or academic HTS LIMS systems.

2.2 Background

High throughput screening techniques utilize advanced robotics and computer automation technology to perform experiments designed to determine the interaction between a target such as a protein against a large collection of compounds (31). This is often used for finding candidate compounds for drug discovery research. Compounds are placed in plates usually containing 96, 384, or an even larger number of wells depending on the machine specifications. The target is then added to each well and sensitive instrumentation is then used to quantify the resulting reactions. For a given experiment, there could be tens or hundreds of thousands of compounds involved each associated with one or more interactions. The significant amount of data generated then needs to be analyzed, quality of data needs to be verified, high-level statistics need to be calculated, and all this data needs to be presented to the client in ideally the most logical and coherent fashion possible.

These types of challenges already encountered in other fields such as mass spectrometry (23), high throughput genomic sequencing (22), and crystallography (34) are widely addressed by Laboratory Information Management Systems (LIMS). LIMS are database driven software packages designed to organize and simplify a laboratory's data harvesting operations. LIMS software provides benefits such as a the ability to track a job's location in the laboratory work-flow, perform automated data analysis, reconfigure the way experimental result data are presented, and accurately log laboratory activity for the purposes of determining responsibility as well as billing. Because LIMS are designed on abstractions of specific laboratory work-flow processes, LIMS are often highly customized products either constructed as in-house solutions or contracted out to commercial consulting companies that specialize in the field. In-house

solutions are often time consuming to specify and develop while seeking a contract with a commercial vendor is often a significant monetary investment for smaller facilities.

In recent years several attempts have been made to create an open-source LIMS system for various types of laboratories. These are often in-house solution published for public use. However, because the requirements placed upon each LIMS vary from laboratory to laboratory, there is no one size fits all solution that exists at present. An effective open-source solution must therefore be built upon a flexible architecture that allows for easy modification to fit the needs of a prospective organization. It is only very recently that significant effort has been put into engineering a HTS LIMS software suite that meets these requirements. K-Screen is one such solution. Compared to similar software suites such as Harvard Screensaver (41), which was published near the end of the current development cycle for K-Screen, K-Screen is lighter-weighted and more feature rich in the realm of data analysis. It provides features such as structural similarity searches, interfaces to both PubChem (43) structure and ZINC (28) vendor compound databases, a stronger suite of cross-screen hit comparison analysis tools, and screen-centric data browsing.

When the University of Kansas High Throughput Screening Laboratory investigated the existing HTS LIMS options to support its growing information management needs in 2009, it found no solutions that fully matched its needs. Existing open-source solutions including M-Screen developed at the University of Michigan (7) were considered but not adopted due to the complex nature of the code base and some missing critical features. Ultimately the development of K-Screen was started to address both KU-HTS's needs and to fulfill the need for an adaptable free and open-source LIMS focused on data analysis.

2.3 Implementation

K-Screen is a web accessible LIMS application built with low cost, simplicity of deployment, and ease of use in mind. To accomplish these goals, K-Screen leverages numerous free and open-source tools that have proven effective and reliable. These include the popular web-development triad of Apache HTTP Server (5), MySQL (9), and PHP (10); PHP based tools such as the Yii Framework (12) and JpGraph graphing library (2); R (11) and the prada (6) and drc (1) R libraries and modified library management and search algorithms from MolDB (3) which include the use of the JME Java Applet (8) and Ghostscript (4).

2.3.1 Application Architecture. The core of K-Screen presentation layer is written in PHP using the Yii Framework. The choice to use PHP for the front end was made to simplify the setup and maintenance requirements of K-Screen. Yii Framework was similarly chosen based on the need for a simple and minimalistic structure to implement K-Screen. Yii's modern Model-View-Controller (MVC) architecture allows for the relatively quick and easy implementation of new functionality or extension of existing functionality. Like most MVC frameworks database tables have corresponding classes that automatically validate and sanitize field data. This approach largely removes the need to directly create SQL queries and error checking functions making it relatively simple to create applications that generate custom query data or tools to interface with the K-Screen's database. Conversely, this interface

between application and database also simplifies altering K-Screen's database to add additional fields.

K-Screen's web-viewable pages are constructed using Yii's modular view system that allows for numerous different pages to share and reuse common elements such as forms, menus, and figures. K-Screen also takes advantage of the user interface (UI) widgets provided by Yii that encapsulate html search forms, tables, and lists so that these types of objects can be created quickly and elegantly. K-Screen's structure thereby maximizes consistency and ease of modification while minimizing redundancy and potential errors introduced during development.

While Yii handles these basic UI and presentation tasks, K-Screen's uses the JpGraph library to handle the complex visualization tasks such as the dynamic creation of graphs, histograms, dose-response-curves, and image maps. These elements are then seamlessly integrated with control schemes embedded into each page through AJAX requests. These modern web design techniques take advantage of K-Screen's modular page layout to reduce the amount of time per http request by only loading the requisite parts of each page and give a better overall experience for staff and clients alike.

The back-end data processing for K-Screen is handled using R scripts. K-Screen launches these custom R scripts as background batch processes to offload burden from the web-server and allow the time consuming data-processing and batch jobs to be executed in parallel. This scheme also separates statistical analysis functions from the PHP presentation layer allowing for the ability to heavily modify the back end without changing the rest of the application.

2.3.2 Data Flow. The starting point with any K-Screen job is the creation of input files for the various jobs that K-Screen does. K-Screen is designed to process simple human readable and writable structure-data files (SDFs) for compound library elements and comma separated files (CSVs) to populate screening data. In the case of the library file, K-Screen uses MolDB algorithms to generate bitmaps of structure from a SDF file and statistics to be imported into the K-Screen database. These then can be immediately accessed for screening and searching.

With primary (single-dose) screens, a laboratory staff member is required to create and upload three separate files. First, the data file contains a reformatted table of screening activities with well location data. Data files can be made either from raw results or after some degree of pre-processing by the user. Since many machines generate a separate file per plate, it is of course encouraged that the most tedious part of this process, merging the files, be done automatically with the R-script provided with K-Screen or with other tools such as a Perl script. The data file is in the form of [plate number][row letter][activity for column 1] [activity for column 2]… Second, the plate configuration file lists the locations of empty wells or wells designated as positive or negative controls. Third, the plate permissions file allows laboratory staff to use a simple domain specific language to designate who is able to access what plates of the particular screen. These files are uploaded to a secure work area on the K-Screen server so the server is able to run R-scripts that generate the statistical information for each screen and plate. This data is then uploaded automatically to the database once it has been reviewed by a staff member.

In many cases a client may request some secondary (dose-response) screens to be run on compounds to verify the data collected from the primary screen. The procedure for importing secondary screens into K-Screen is to create a data file similar to the one created for the primary screen. The plate configuration file however requires additional information about concentration to be added along with data regarding the location of compounds on each plate.

After a laboratory staff member imports the screen level data, he or she is able to configure how the data will be presented to the client. With the primary screens, this involves reviewing the results and rejecting bad data and potentially running a new screen. With secondary screens, this additionally involves validating the quality of, manually refitting, or rejecting the best-fit dose-response-curves. When the staff member completes this step, the data is available for viewing by the client. The client can then use K-Screen to run various cross-screen searches to locate other potential candidate compounds to run screens against.

2.3.3 Deployment. K-Screen was designed and tested on Apache httpd servers paired with a MySQL server on both Windows XP and various flavors of Linux such as Redhat and CentOS. While a similar setup is recommended, it may not be available or it may simply be easier to use an existing configuration. To this end, K-Screen is flexible and can be configured to be used on any PHP capable web server such as Windows Server and any database supported by PHP's PDO whether remote or local. A list of PDO supported databases management systems is located at http://www.php.net/manual/en/pdo.drivers.php. Since K-Screen is built on Yii's database interface, database configuration is done through Yii.

2.4 Results

Unlike many other LIMS applications targeted at HTS laboratories, K-Screen has been designed from the beginning to fulfill a need for a web accessible data visualization, manipulation, and analysis tool. Its role is to help automate and simplify the process of analyzing results and delivering those results to clients. It is targeted specifically for academic and industrial HTS facilities seeking to better augment existing work processes through the use of web accessible tools.

K-Screen has a wide range of capabilities. At this time, K-Screen has been demonstrated to automatically generate screen and plate level statistical calculations when processing large primary screens of at least 100,000 across three hundred 384 well plates. It can automatically normalize plates using several different algorithms in addition to a raw setting that allows laboratory staff to normalize data in any way they choose. K-Screen supports searching using plate coordinates, compound names, and structure while allowing users to set search constraints based on compound activities across multiple screens. Search results can also be conveniently downloaded in both CSV and SDF formats for use in both excel and structure viewers. K-Screen allows laboratory staff to manipulate dose-response-curves from secondary screens using an intuitive web based GUI. Currently K-Screen only supports log-logistic (both four- and three-parameter) and Brain Cousen models but it can be easily expanded into any of the curves supported by the drc R library. K-Screen supports several different types of

visualization tools and browsers previews of which can be seen in Figures 2-5 under the 'Data Analysis Tools' subsection of the 'Discussion' section.

2.5 Discussion

K-Screen implements several important types of functionality. Each of the following subsections will go into detail on one of these aspects.

2.5.1 Database Model. K-Screen's database model follows standard relational database normalization conventions to provide both efficient insertions of new records and minimize the effects of scaling on querying. It is designed to reflect common high throughput screening methodologies. As such, the database model can be roughly divided into three parts excluding user data (addressed in detail in the next section) corresponding with compound library or inventory information, primary screen, and secondary screen data.

The first collection contains data for the molecular compound database. The primary table stores data of each molecular compound in the library such as identifiers, supplier information, and locations of each compound stock on the parent (or mother) plate. Each record is then associated via foreign key to its corresponding image link url, fingerprinting data, and structural data.

The second collection contains data for the primary screens. There are two main tables in this collection. The first table stores screen-level information such as the name, upload date, PI id, and client id. The second table associates each screen with plate and well level data. For each

well, this table contains references by a foreign key linking the compound and the well, columns storing the coordinate of the well in the set of child (or daughter) plates created from stock compounds from the set of parent plates, and additional columns filled with the raw activities values of the compound and the values under the various normalizations supported by K-Screen. Additionally, both these tables are linked to user visibility permissions to support K-Screen's security model.

The third collection contains data for the secondary screens. There are three main tables in this collection. The first table stores screen-level information similar to the primary screen table. The second table contains multiple concentration and activity level pairs associated by foreign key to secondary screen and compound tables. The third table contains curve information for each compound in the secondary screen run including curve formula and fitted constants. Again similar to the primary screen tables, these tables are linked to user visibility permissions to support K-Screen's security model.

2.5.2 Security Model. K-Screen's security model is based on the role based access control model (RBAC) where each user is assigned a single role associated with a set of privileges. This standard model can be easily abstracted to most organizations and adapted easily when needed. In addition to a privilege, important contact information such as telephone and location of the user's parent institution is also stored. K-Screen users are identified using a unique email which users use to login to the system. Passwords in K-Screen are stored as hashed values with an added SALT value to prevent malicious access even in event of access to the user table.

Currently K-Screen has two roles that a user can be assigned to. Users can either be clients or administrators. Clients represent the people who submit screen requests and are allowed to view processed screen data. This set of people is also allowed to search the chemical library as well use the other data analysis tools. Administrators represent the laboratory staff who perform the screens and do the data analysis work. Administrators can create, modify, or delete compounds, users, and screens from the K-Screen system.

Currently, there are two ways to add users to K-Screen. Users may either register on the main page of K-Screen or wait to be approved by an administrator or an administrator may add a user manually. The registration feature may be turned off in the settings.

2.5.3 Data Access and Sharing. K-Screen stores relation data for primary screen and plate level permissions that are set by an administrator. Clients can request the results of their primary screens to be fully visible or with only certain plates visible to other clients. While we encourage our clients to share data and K-Screen has the capacity for data sharing, it is up to the screen owners whether and with whom they want to share all or part of their data.

Secondary screenings are associated with primary screen plate wells via identical compounds which allow K-Screen to automatically hide results for users who are not allowed to see the results of the well from the associated primary screen.

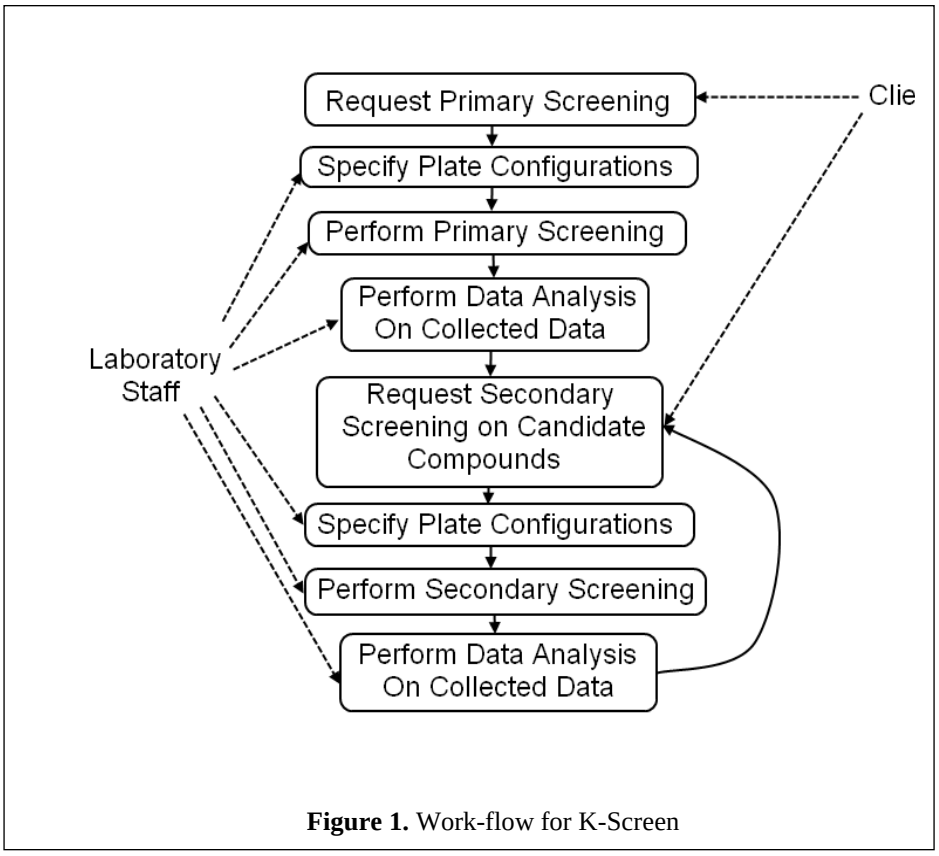


Figure 1. Work-flow for K-Screen

2.5.4 Work-flow Model. In K-Screen, laboratory staff uploads data using K-Screen's web interface (Fig 1-2). This data is quickly preprocessed by K-Screen and written to a file directory in the server. R-Scripts are then executed as background processes on the directory to generate processed data files along with screen level statistics. These screen level statistics and processed data is used by laboratory staff to assure quality and whether any plates need to be rerun. Staff then publishes the results. This sends a command to K-Screen to read data from the directory and write them into the database. Appropriate clients now have access to the finalized data. Clients then are able to run custom searches through K-Screen's web interface with text-based, file-based, structure-based, and functional group-based constraints. These searches are able to generate cross-screen data and results are available to be downloaded in CSV or SDF format.

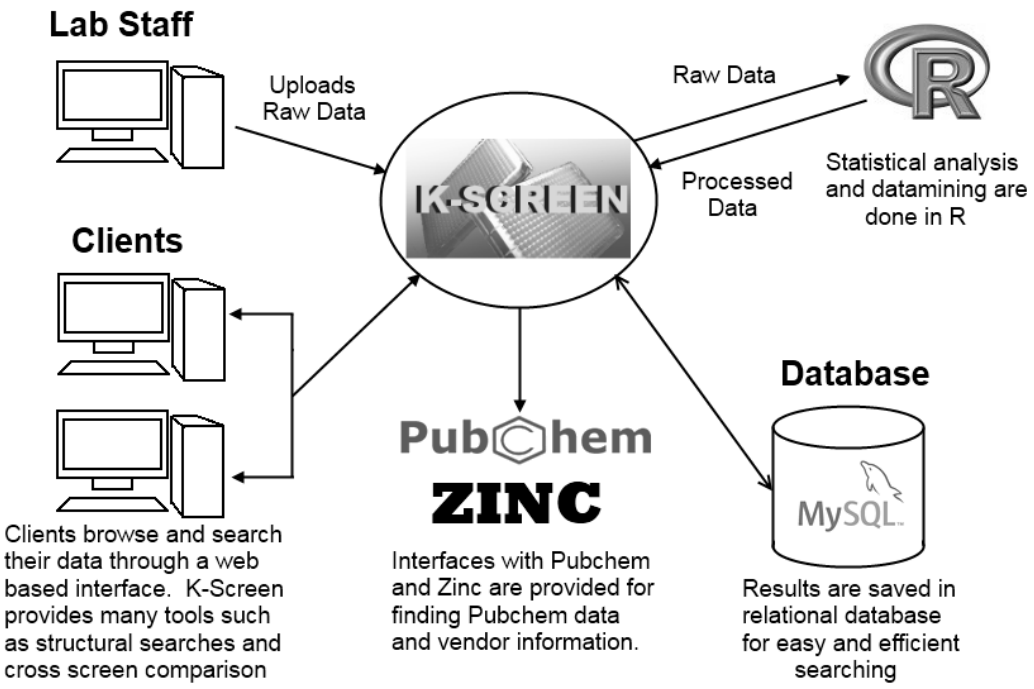


Figure 2. Division of labor for K-Screen based on the work-flow abstraction from Figure 1.

2.5.5 Data Analysis Tools. One of K-Screen's primary functions is to support advanced data analysis and data mining. K-Screen currently supports several tools to do this. All of these tools are web accessible and designed to be interactive, intuitive, user-friendly, and efficiently in communicating data to lab staff and clients. To better achieve these goals, K-Screen is designed to make use of image-based rather than traditional table and list-based data presentation. This offers more flexibility when presenting data and allows logical relationships to be used to access data. Text-based data can also be downloaded for each primary and secondary screen. These tools were originally written in PHP and have since been modified to use AJAX requests built using Yii's native support for the JQuery JavaScript library to allow for more responsive

interactions. A description of basic function and use are associated with Fig. 3 and 4. In addition, advanced data analysis and mining can be done in R platform because an R session with all data is saved for each screen.

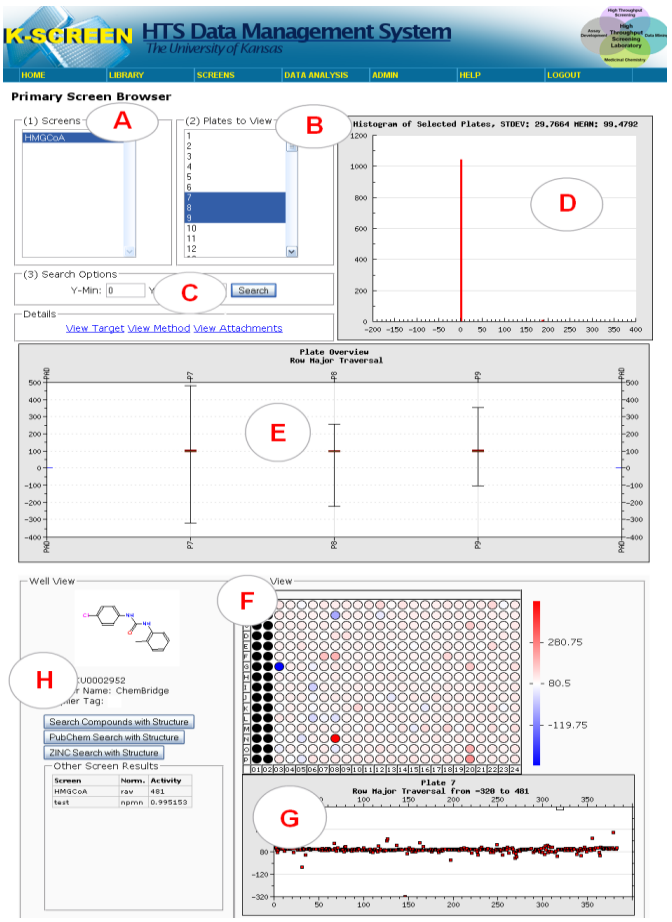


Figure 3. K-Screen's primary screen browser is a user-friendly tool used to data-mine outliers and other data points of interest. Users in addition to being able browse primary screens are able to access screen specific data including target data, measurement method data, and any attached files. Users specify a primary screen (A) and select a subset of the total plates (B) to view before executing the search (C). This dynamically generates a histogram of the selected plates (D) as well as creates a series of box-plots (E) using the distribution of data points from each plate. The box-plot gives a quick summary of how the results of each plate are distributed. Clicking on a box-plot generates an interactive heat-map representing the plate associated with the box-plot (F) as well as a plate level scatter-plot of activities (G). The wells on the heat-map and points on the scatter-plot can be clicked to display the data for the associated compound (H) including structure, if-one exists. This panel also displays values for the same compound across other accessible screens. The structure can be directly used for searching the local library, PubChem, and ZINC.

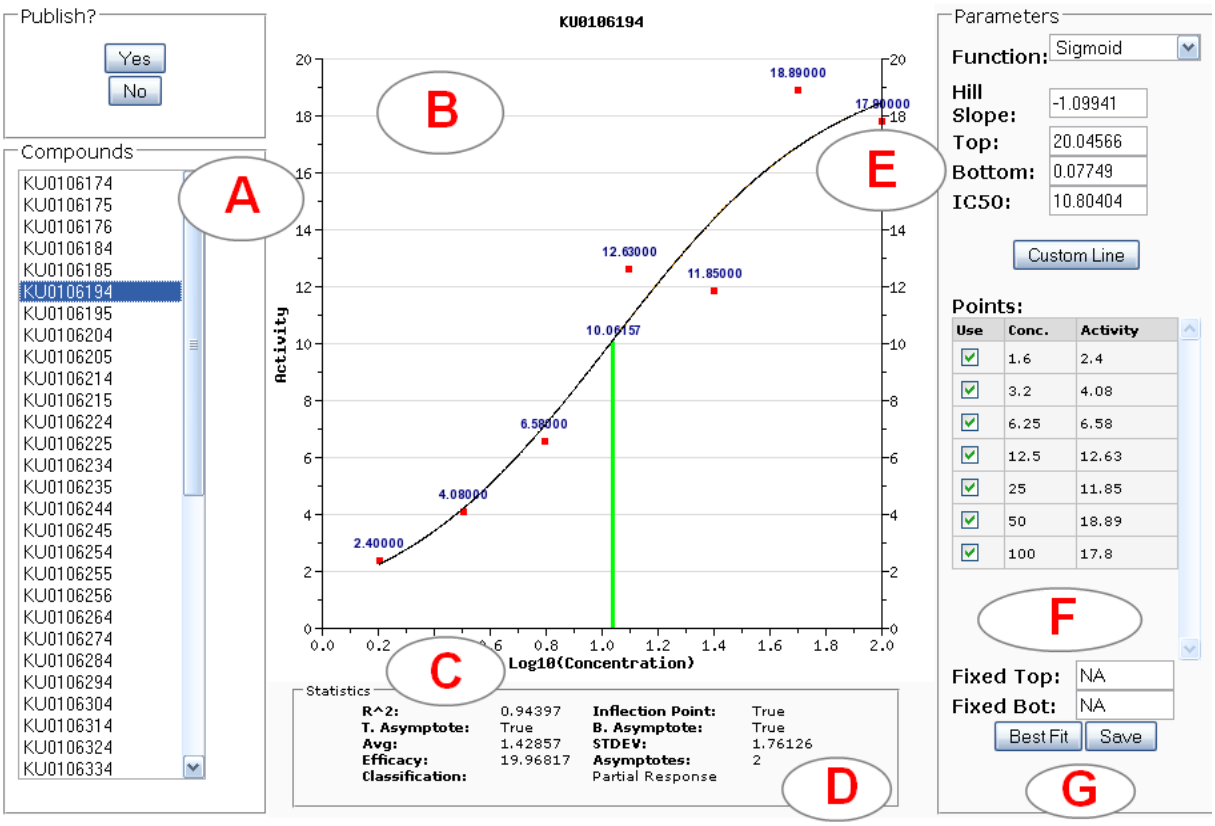


Figure 4. K-Screen's dose-response-curve editor is an easy to use tool for adjusting dose response curves. When an administrator prepares the processed secondary screen dose-response curves to be published, he or she will first select a compound (A). This will generate the best-fit curve (B) as well as the statistics associated with the curve (C) as well as the curve classification (D) according to Inglese et al(27). On the right panel, administrators can modify which function and function parameters (E) that each dose-response-curve uses. This section also is useful for reporting function parameters including half maximal inhibitory concentration (IC50). If a manual refit is not necessary then the administrator may elect to disable certain points considered to be outliers or explicitly specify the top and bottom of the curve (F) and have K-Screen refit the curve (G). Changes can be quickly saved on a curve by curve basis. A stripped down version without the ability to adjust values is available for clients.

2.5.6 Searching. K-Screen supports numerous searching features (Fig 5-7). Library search features utilize heavily modified versions of algorithms from MolDB. Each search supports the setting of screen-based activity constraints using screen and well data associated with each

compound. Text searches support searching by compound identifiers, mother plate coordinates and daughter plate coordinates.

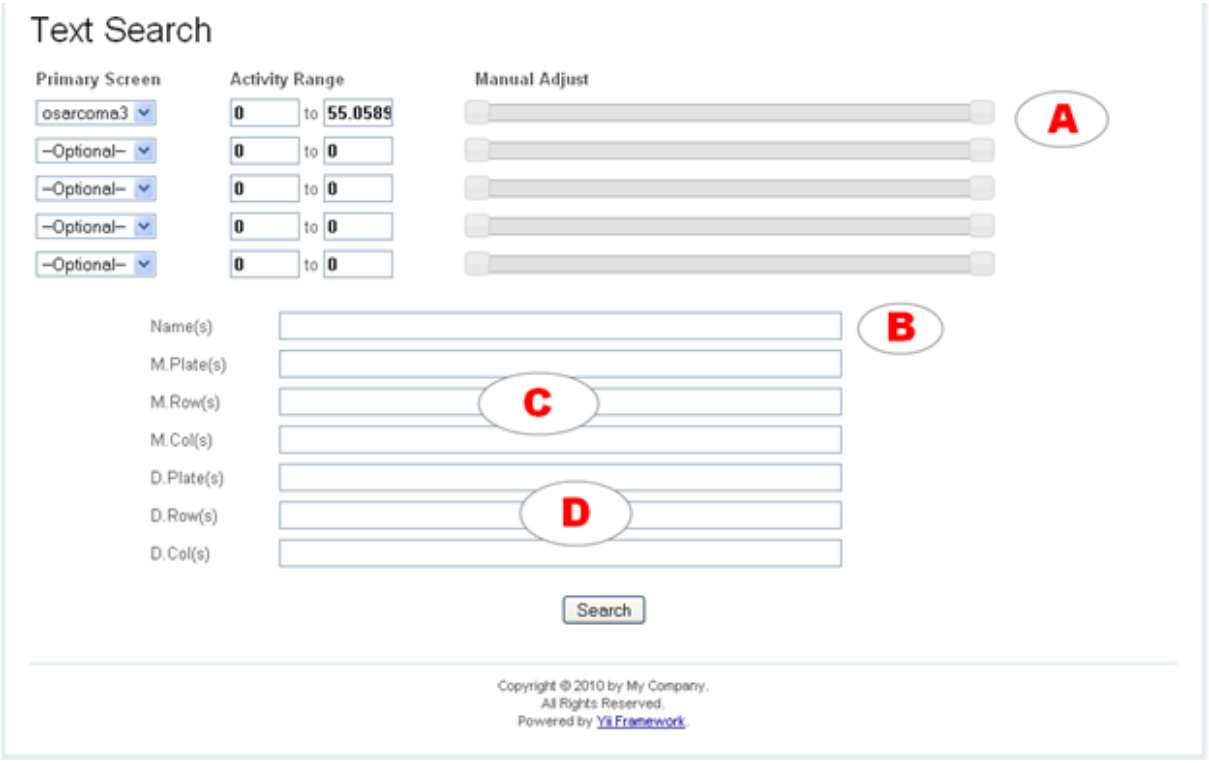


Figure 5. K-Screen's Text Search. (A) Users specify what Primary Screen they want to see and from what range. Users can then specify the chemical name (B) which searches against, Library ID, Chemical Name, and Supplier Tag. Users alternatively may want to search for specific (C) mother plate coordinates or specific daughter plate coordinates (D).

Batch text searches allow users to upload files listing compound identifiers, mother plate coordinates, and daughter plate coordinates to be searched. Functional group searches allow searches to be done over multiple families of compounds.

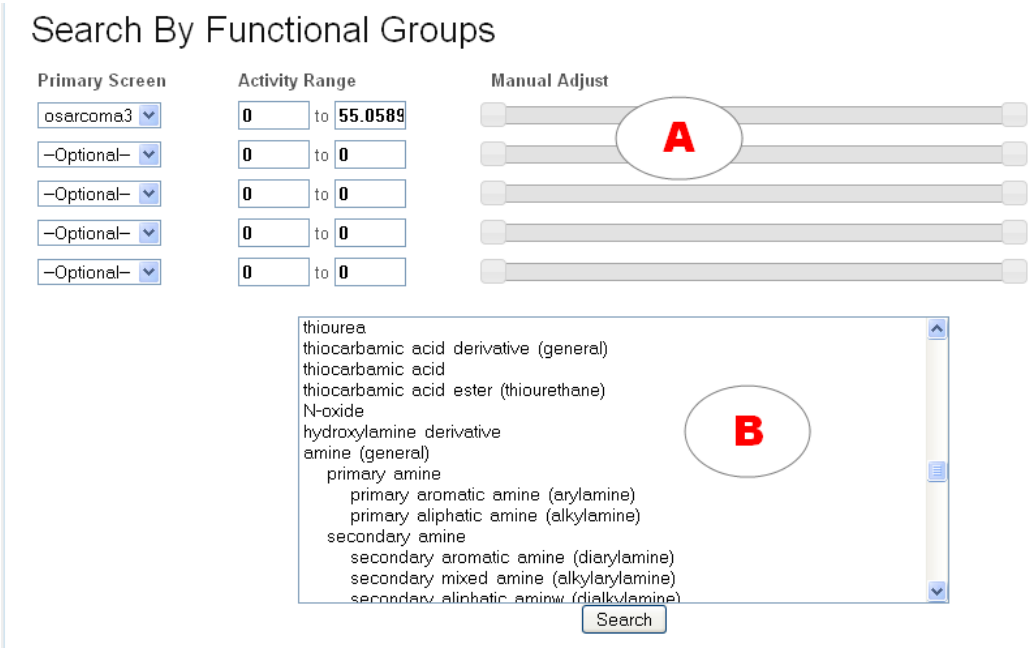


Figure 6. K-Screen's Functional Group Search based on the original implementation from MolDB. (A) Users specify what Primary Screen they want to see and from what range. Users then specify up to 5 Functional Groups to search (B).

Structural searches allow users to draw or input a structure into the JME java applet to be searched by minimal Tanimoto similarity score, exact structure, or substructure. Searches lead to a common result page that allow results to sorted by various fields including coordinates and activity, paged through, and downloaded as CSV or SDF files. K-Screen also includes a set of simple but powerful browsers-based on the Yii GridView object which allow the large lists of library compounds, primary screens, and secondary screens to be sorted, searched, and filtered based on almost any visible field in the database. These include table specific fields such as identifiers, plate coordinates, last modified date, screen targets, screen methods, and other fields.

Figure 7. K-Screen's Structure and Substructure Search based on the original implementation from MolDB. Users specify what Primary Screen they want to see and from what range. Users then draw the structure in the JME applet (A). Users next select which type of search they desire to perform (B) with which options enabled (C).

2.6 Future Plan

K-Screen is an extensible open-source software suite for a HTS LIMS. K-Screen is being actively used by KU-HTS and its clients. Development of the K-Screen HTS LIMS system will continue to support KU-HTS operations as well as those who adopt it. Future developments include integration of more data analysis tools, better integration with PubChem, support for searching ChemSpider, integration of R-Serve to better manage background R processes, the creation of a

single library to encapsulate all K-Screen R features, automated outlier removal from dose response curves, and plate-based quality control reporting. The software will be released to public at http://kscreen.org as open source software and we invite other people join us to make the software better and more comprehensive.

Chapter 3: Algorithms for Chemical Fingerprint Searches Using Bit-Strings

3.1 Overview

This section discusses research into the development of efficient algorithms for chemical fingerprint searches, specifically similarity searches over 1-dimensional bit string fingerprints. Sections 3.1 to 3.3 contain supporting information for these projects. Section 3.4 describes and analyzes the development and implementation of a Shannon Entropy-based heuristic for better fingerprint segmentation, a novel algorithm for improving the speed and accuracy of chemical fingerprint searches by balancing information when segmenting strings. Section 3.5 describes and analyzes the development and implementation of a search algorithm specifically designed to support the execution of faster batch querying for existing pruning indexes.

Each algorithm tested is implemented in Java using a similar object oriented structure. This allows for a fair comparison between each algorithm tested.

In summary, Chapter 3 will:

- Give a brief description of the importance of chemical fingerprint searches for High Throughput Screening.

- Provide an overview of existing algorithms for chemical fingerprint searches in literature.
- Describe each of the novel algorithms proposed in this section and experimentally analyze how each compares to existing algorithms.

3.2 Chemical Similarity Search Background

The sizes of chemical databases are becoming increasingly large as new chemicals are catalogued and new data is generated. The popular database PubChem in 2011 at this paper's time of writing contains 53 million chemical IDs, which represents nearly a 300% growth since 2007 with growth expected to continue steadily (19). PubChem only represents one of several online chemical databases and the countless chemical databases maintained by public and private institutions. Searching these databases has major applications in many fields but especially in the field of drug discovery where High Throughput Screening (HTS) is used to discover candidate drugs using large chemical libraries (31). Similarity searches are used to create custom libraries, eliminate redundant chemicals, and to find chemicals similar to promising drug candidates.

Chemical fingerprints are often used to conduct these types of searches efficiently. There are many methods of chemical fingerprint searches each with particular strengths and weaknesses (35). Most of these methods rely on generating fingerprints by performing a transformation to reduce the dimensionality or amount of information stored for a chemical to a more manageable size by discarding less useful information. There are many types of fingerprints and popular representations which include graphical (37, 42) and string representations (25).

For this thesis, the popular 1-dimensional fingerprinting method is used. This simple method converts chemical representations to a one dimensional feature string by describing only the presence of absence interesting properties of a chemical. These are often represented as fixed length bit strings where each bit is set either on or off, refer to as 1 and 0 respectively, based on whether or not a feature is present. These generated chemical fingerprints can then be compared against each other in relatively fast searches.

3.2.1 Tanimoto Similarity Metric. Chemical fingerprints are compared using various similarity metrics which usually employ bit-wise 'and' and 'or' operations (also referred to as intersection and union). There are a large variety (29) of similarity measures that exist for various purposes. *Holliday et al.* (26) investigates 22 such indices and the uses of each.

For the similarity metric used in this thesis, the Tanimoto similarity metric which depending on application may also be called the Jaccard Index was chosen. Not only is this one of the most popular metrics, it seems to have become by convention the de-facto standard for benchmarking chemical similarity search indices (14-16, 30, 32, 36, 39). A great deal of research has been devoted to figuring out new methods of exploiting this metric (15, 18) and thus we feel no desire to break with tradition. The Tanimoto similarity S_t is expressed as the similarity of chemical fingerprints A and B:

$$s_t = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

Where |x| represents cardinality (number of 1's bits) and ∩ and ∪ represent intersection and union respectively. For similarity searches, the objective is to only find pairs of chemical fingerprints with similarity above some threshold T for some query fingerprint similarity a and

all members b_i of database B. To achieve this, the database must be *pruned* or have all b_i removed from the results for which the following is false:

$$T \leq \frac{|a \cap b_i|}{|a \cup b_i|} = \frac{|a \cap b_i|}{|a| + |b_i| - |a \cap b_i|}$$

However, calculating the pair-wise similarity of a and ever b_i in a brute force fashion is inefficient. Thus, a simple upper bounding trick discovered by *Baldi et al.* (39) and used extensively in their work (15, 16, 18, 32, 39) can be used. For any two given bit strings, an upper bound can be calculated assuming that the shorter bit string is a subset of the longer one - that is the maximum intersection of bits is the same as the shorter bit string and the minimum union is the same as the longer bit string:

$$\min(|a|, |b_i|) \geq |a \cap b_i|$$

$$\max(|a|, |b_i|) \leq |a \cup b_i|$$

From this, it follows that:

$$T \leq \frac{\min(|a|, |b_i|)}{\max(|a|, |b_i|)}$$

This upper bound check is the basis for nearly all search indexes for faster similarity searches between chemical fingerprints. The upper bound is used as a first pass to prune all b_i which cannot have a similarity above T to reduce the total number of expensive similarity calculations. Another useful property of this upper bound is that given$|a|, |b_i|$ can be constrained by breaking down the upper bound into two cases.

Case 1: $|a| \leq |b_i|$

$$T \leq \frac{|a|}{|b_i|}$$

$$|b_i| \leq \frac{|a|}{T}$$

Case 2: $|a| \geq |b_i|$

$$T \leq \frac{|b_i|}{|a|}$$

$$T * |a| \leq |b_i|$$

A data structure or *index* can be built using the cardinality of a fingerprint as a *hash-key* to access all fingerprints with that particular statistic. In this case, an array implementation is simplest and fastest (O(n) = 1 for access times).

3.2.2 Existing Algorithms. There are several state of the art algorithms for chemical fingerprint similarity searches using bit strings. This section will serve as an introduction and literature review of existing algorithms

3.2.2.1 UCI Algorithms. The team associated with Pierre Baldi based at University of California Irvine(UCI) has published much research related to bit string similarity searches. They have published a lineage of state of the art algorithms since 2007 each improving upon the previous along with a great deal of research on chemical fingerprints (16, 17, 20, 38). The first algorithm published was the *bit bound* algorithm based on the fingerprint cardinality bounding described in section 3.2.1 (39) which yields a 1200% speed up compared to brute force for a Tanimoto similarity value of .9 for their database.

In 2008, this group published an algorithm which will be referred to XOR based upon the XOR (denoted by $\oplus$) formulation of intersection and union for statistically tighter pruning over compressed XOR-Modulus (or XOR-folded) fingerprints than simple bit bound (16). XOR-Folding is defined as calculating XOR of all bits with the same modulus congruency or all bits modulus a value sharing the same remainder. The UCI paper gives the example:

> "…the folding process with a binary vector of length *N* = 16 (1 1 0 0 1 0 0 1 0 1 0 0 1 0 0 0) [is] folded into a binary vector of length *n* = 4 (1 0 0 1), modulo 4 using the XOR operator."

The union and intersection identities of the fingerprints A and B are:

$$|A \cup B| = \frac{1}{2} * (|A| + |B| + |A \oplus B|)$$

$$|A \cap B| = \frac{1}{2} * (|A| + |B| - |A \oplus B|)$$

The useful result of both the XOR folding and the new union and intersection identities taken together is that the XOR folded fingerprints A_{xor} and B_{xor} when XOR together are bounded by:

$$|A \oplus B| \geq |A_{xor} \oplus B_{xor}| \geq abs(|A_{xor}| - |B_{xor}|)$$

Where abs is the absolute value function. Thus a new bound is generated when this is substituted back into the union and intersection identities and further back into the Tanimoto similarity.

$$|A \cup B| = \frac{1}{2} * (|A| + |B| + |A \oplus B|) \geq \frac{1}{2} * (|A| + |B| + |A_{xor} \oplus B_{xor}|)$$

$$\geq \frac{1}{2} * (|A| + |B| + abs(|A_{xor}| - |B_{xor}|))$$

$$|A \cap B| = \frac{1}{2} * (|A| + |B| - |A \oplus B|) \leq \frac{1}{2} * (|A| + |B| - |A_{xor} \oplus B_{xor}|)$$

$$\leq \frac{1}{2} * (|A| + |B| - abs(|A_{xor}| - |B_{xor}|))$$

$$s_t = \frac{|A \cap B|}{|A \cup B|} \leq T_{\oplus}(A, B) = \frac{|A| + |B| - |A_{xor} \oplus B_{xor}|}{|A| + |B| + |A_{xor} \oplus B_{xor}|} \leq \frac{|A| + |B| - abs(|A_{xor}| - |B_{xor}|)}{|A| + |B| + abs(|A_{xor}| - |B_{xor}|)}$$

Thus $T_{\oplus}(A, B)$ serves as an upper bound. If $T_{\oplus}(A, B) \leq s_t$ then B is not above the similarity threshold to A. B_{xor} is stored with the corresponding fingerprint and used to short circuit or cancel the similarity calculation. This method using a bit bound pruning index followed by modulus compression value of 128 results in a 250% speed up compared to bit bound alone.

Most recently in 2010, Baldi's group published a third algorithm which will be referred to as modulus based on dividing a bit string into discrete segments (*segmentation)* and storing the cardinality of each segment to help prune fingerprints. The published method uses a modulus *m* operation to divide a bit string into *m* segments based on the same modulus congruency as with the XOR folding trick. The cardinality of each segment is taken and used as an index hash-key. For example:

Let X = 0011001010101001 divided by modulus 2 congruency creates 2 segments which when summed result in 2. In this case segments X_1 = 01011110 and X_2 = 01000001 and X can be described by the ordered pair (5, 2) much like the how an XOR fold is used to describe the fingerprint in the previous algorithm.

Essentially this algorithm uses dimensionality reduction from the number of bits to chosen value of *m* much like how bit bound reduces dimensionality to 1. This key can then be used to create a bound tighter than the bit bounding algorithm for the special case of modulus 2. For this algorithm, union and intersection are again defined as:

$$|A \cap B| \leq \min(|A_1|, |B_1|) + \min(|A_2|, |B_2|) \leq \min(|A|, |B|)$$

$$|A \cup B| \geq \max(|A_1|, |B_1|) + \max(|A_2|, |B_2|) \leq \max(|A|, |B|)$$

This gives a new Tanimoto similarity formulation of:

$$T = \frac{|A \cap B|}{|A \cup B|} \leq \frac{\min(|A_1|, |B_1|) + \min(|A_2|, |B_2|)}{\max(|A_1|, |B_1|) + \max(|A_2|, |B_2|)} \leq \frac{\min(|A|, |B|)}{\max(|A|, |B|)}$$

This formulation can equivalently be stated as:

$$T = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \leq \frac{\min(|A_1|, |B_1|) + \min(|A_2|, |B_2|)}{\max(|A_1|, |B_1|) + \max(|A_2|, |B_2|)}$$

$$= \frac{\min(|A_1|, |B_1|) + \min(|A_2|, |B_2|)}{|A| + |B| - \min(|A_1|, |B_1|) + \min(|A_2|, |B_2|)}$$

The two min's can then be isolated to one side:

$$\min(|A_1|, |B_1|) + \min(|A_2|, |B_2|) = \frac{T}{T+1}(|A| + |B|)$$

This gives 4 cases:

- $|A_1| + |A_2| \leq \frac{T}{T+1}(|A| + |B|)$, which is equivalent to bit bound;
- $|B_1| + |B_2| \leq \frac{T}{T+1}(|A| + |B|)$, which is equivalent to bit bound;
- $|A_1| + |B_2| \leq \frac{T}{T+1}(|A| + |B|)$, which yields $|B_2| \leq \frac{T}{T+1}(|A| + |B|) - |A_1|$
- $|B_1| + |A_2| \leq \frac{T}{T+1}(|A| + |B|)$, which yields $|B_1| \leq \frac{T}{T+1}(|A| + |B|) - |A_2|$

For the general modulus *m* > 2, an intersection inequality check must be used since there are many unknowns:

$$T = \frac{|A \cap B|}{|A \cup B|} \leq \frac{\sum_{n=1}^{m} \min(|A_n|, |B_n|)}{\sum_{n=1}^{m} \max(|A_n|, |B_n|)}$$

The authors reported that this indexing scheme is 300% faster than the bit bound and XOR combination previously developed and about 500% faster than bit bound alone.

3.2.2.1 Other Algorithms. Five other algorithms have been published on this subject. First, in 2009 Smellie (36) proposed a binary tree which will be referred to as bit tree to store binary strings as paths and a path-based pruning algorithm for comparisons. Each node of the tree root represents a bit position. Inserting a bit string is a recursive function. The string is inserted at the root. At each node, if the bit at the corresponding position is 0, then make the recursive call on the left child, if the corresponding position is 1, then make the recursive call on the right child. When all bits in the bit string are exhausted, then a reference to the bit string is stored in a leaf node. To execute similarity searches in this tree for the *i*-th node with length *N*:

$$T = \frac{|A \cap B|}{|A \cup B|} \leq \frac{\sum_{n=1}^{i} |A_n \cap B_n| + \sum_{n=i+1}^{N} |A_n|}{\sum_{n=1}^{i} |A_n \cup B_n| + \sum_{n=i+1}^{N} |A_n|}$$

Since searching the bit strings start with the first bit, the summations are accumulated as the tree is traversed. If this upper bound falls below the threshold, the algorithm backtracks and all strings along the path are pruned. Smellie also developed a version that removed runs of 0 bits. The bit tree's algorithm's was only 400-500% faster than exhaustive search and the compressed version 250% faster than the uncompressed version. Thus Smellie's bit tree is comparable to bit bound alone and worse than bit bound + XOR or modulus.

Second, in 2010 Aung and Ng (14) proposed a bit string segmentation algorithm called ChemDex very similar to the UCI modulus algorithm. The ChemDex algorithm divides a bit string into equally sized segments after reordering the bits so bits more likely to be 1 are at the start of the bit string. Compared to the modulus algorithm, ChemDex algorithm contains two major differences when calculating similarity. First, ChemDex exclusively uses the intersection threshold rather than the more efficient bound used by the modulus algorithm for $m = 2$. This method is used for the general case of $m > 2$. Second, ChemDex contains an additional pruning function for segments during the similarity calculation. For all bit strings A and B with S segments, sum all segments 1 … s … S and then iteratively replace s by the actual intersection and union for that segment and do a threshold check:

$$T = \frac{|A \cap B|}{|A \cup B|} \leq \frac{\sum_{s=1}^{i} |A_s \cap B_s| + \sum_{s=i+1}^{S} \min(|A_s|, |B_s|)}{\sum_{s=1}^{i} |A_s \cup B_s| + \sum_{s=i+1}^{S} \max(|A_s|, |B_s|)}$$

The bit reordering mentioned above serves a greedy algorithm that attempts to minimize the number of checks. This algorithm was benchmarked to be about 200% faster than bit bound alone.

The last three algorithms were published in *Kristensen et al*. (30) in 2010. The first of these algorithms is *k*D-grid, another string segmentation algorithm where *k* denotes the number of segments the bit string is broken into. *KD-grid* can simply be thought of as the bound used by Smellie's bit tree applied to segmented bit strings:

$$T = \frac{|A \cap B|}{|A \cup B|} \leq \frac{\sum_{s=1}^{i-1} \min(|A_s|, |B_s|) + \min(|A_i|, |B_i|) + \sum_{s=i+1}^{S} |A_s|}{\sum_{s=1}^{i-1} \max(|A_s|, |B_s|) + \max(|A_i|, |B_i|) + \sum_{s=i+1}^{N} |A_s|}$$

This strategy is very similar to the bit tree. The bit string is segmented and the number of bits in each segment is added up. An n-ary tree is used to index the bit strings, where n is the number of bits in each segment where the level (number of nodes between a node and root + 1) of the node in the tree corresponds with the segment number. This algorithm performs somewhat faster than bit bound and XOR.

The last two algorithms published by *Kristensen et al*. are the related single and multi-bit trees which will be referred to as split-bit tree and match-bits tree to avoid confusion with Smellie's work. Both of these algorithms prune based on the same bit-wise and threshold developed and proved in their paper. The split-bit tree is a binary tree where each node represents a *split-bit* which is used to divide the library of strings into two groups. This algorithm however was not very successful even when used with *k*D-grid. Unlike either of Smellie's bit trees, not enough of the tree is stored to effectively prune long bit strings. The match-bits tree was created to

address the short comings of the single bit tree. It saves all bits which are identical at each node before splitting the library at the node. If no bits are shared between all the fingerprints, then a leaf node is generated to store that portion of the bit string library. Thus, instead of only being able to prune based on depth number of bits, the match-bit tree can store data on a potentially large portion of a bit string. When used along with bit bound pruning, this algorithm performs very quick searches faster than most other algorithms though no concrete number is provided by *Kristensen et al*.

3.3 Efficient Representations for Chemical Fingerprints in Java

The first steps for developing more efficient algorithms for chemical fingerprint similarity searches is obtaining chemical data, generating fingerprints, and picking an efficient representation for the chemical fingerprints. For the chemicals, a database containing the first 1.5 million PubChem chemical ID's that had been used to generate a demo database for K-Screen was used. Because not all PubChem ID's are used in the PubChem database, the actual number of chemicals present was actually 1,307,109. Since PubChem automatically generates a 881 bit fingerprint whose specification can be found on the website, (13) this fingerprint was used as the primary fingerprints in the experiments. In addition the first 100,000 166 bit MACC were fingerprinted using Molecular Operation Environment (MOE, version 2009.10, Chemical Computing Group, Inc. Montreal, Canada) to have a second set of very different fingerprints to use for testing

The issue of a picking the best fingerprint representation for Java was more difficult. There are many considerations. The main consideration is the architecture of the machine the tests will be run on. The test will be done on a Windows 7 machine with an i5-2500K 64bit processor and 8gb of memory. The standard representation for a bit string on a 64 bit machine is an array of 64 bit integers to exploit the ability to perform what essentially amounts to a bit-wise operation over 64 bits per instruction. However Java provides for a several ways of representing this:

1. Java array of objects
2. 2D Java style array of arrays
3. 1D C++ style array accessed as a 2D array

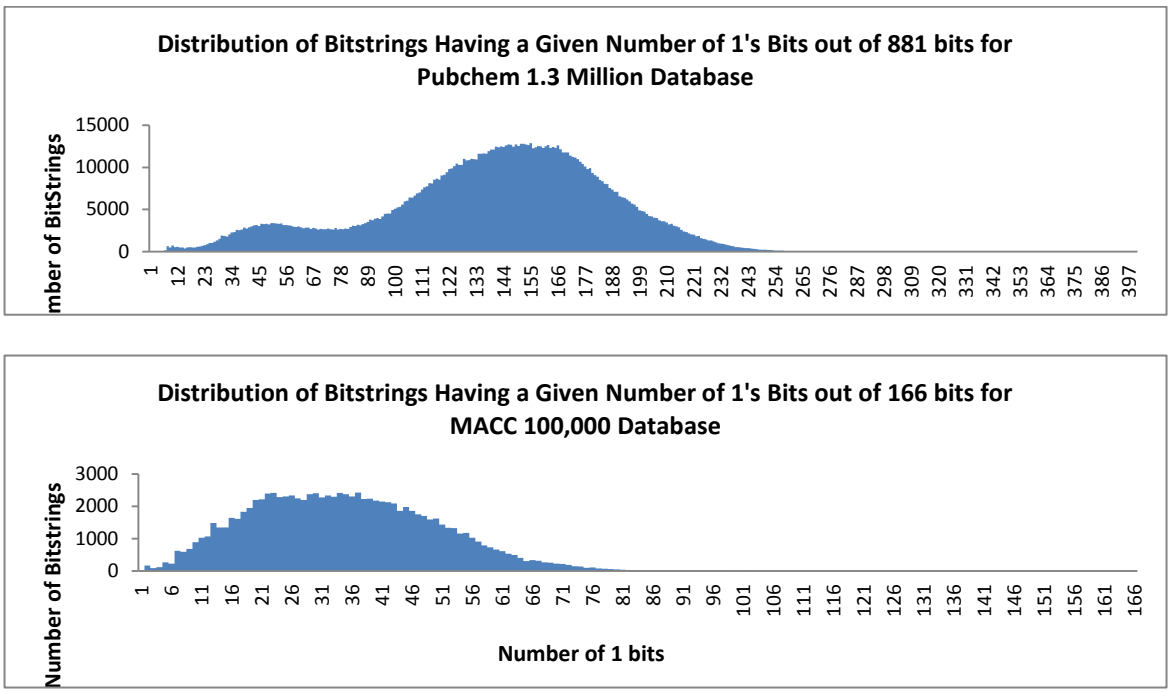


Figure 8. The number of 1 bits in each fingerprint for both fingerprint databases used for the experiments.

The implementation of 2 and 3 are straight-forward arrays. For 1, there were two choices. Java contains a dynamically resizable BitSet object. However, this black boxed implementation

potentially contains unwanted bounds checking and checking for dynamic resize. Thus, it was decided to implement a BitString object based on a fixed array of longs implementing all operations needed to calculate Tanimoto similarity including bit-wise and, or, xor, and popcount.

In addition, after calculating the statistics of the bit strings which can be seen in Figure 8, both sets of fingerprints were found to be mostly sparse. The average number of 1 bits in the PubChem fingerprints is 139 or 15.7% of the total length and the average number of 1 bits in the MACC fingerprints is 34 or 20.4% of the total length. Thus a representation that can exploit this amount of sparseness could potentially be more efficient than the intuitive array representations. To test this theory, a sparse matrix representation using compressed row format (CSR) (33) was also tested. An example of this format can be seen in Figure 9.

$$Matrix = \begin{bmatrix} 10 & 0 & 0 & 0 \\ 0 & 11 & 12 & 0 \\ 0 & 13 & 0 & 0 \\ 14 & 0 & 0 & 15 \end{bmatrix}$$

Values = [10, 11, 12, 13, 14, 15]

Columns = [0, 1, 2, 1, 0, 3]

Rows = [0, 1, 3, 4]

Figure 9. An example of compressed row format. Compressed row format stores a matrix as three arrays. The value array stores the values of the matrix, the column array stores the column index of each value, and the row array stores the index of the first value from the value matrix for each row.

Since 64 bit instructions operate over 64 contiguous bits, matrix sparseness is only useful when there is a *run* 64 0's (64 0's in row). In an attempt to maximize the amount of sparseness useful

for 64 bit calculations when using compressed row format, a preprocessing method is used. This method referred to as *rotate* attempts to align the longest run of 0's to the beginning of a 64 bit integer to maximize run length. This implementation also required rotated copies of query fingerprints which are pre-calculated for efficient lookup.

To find the fastest fingerprint database representation, the 881 bit PubChem fingerprints since each of those fingerprints is encoded using fourteen 64 bit integers and would provide a more accurate representation of typical fingerprints which are often longer compared to testing shorter fingerprints. Each MACC fingerprint by comparison only uses three 64 bit integers to encode 166 bits and thus would give high variance results depending on the sample of fingerprints used since eliminating the calculation of even a single integer would have a disproportionately large overall impact. Before running the test, the average number of 64 bit operations saved for both the standard CSR and CSR rotation representations was done. It was found that the average number of integers used was 11.83 and 11.80 respectively which represent a 15.5% and 15.7% reduction in number of calculations for Tanimoto similarity calculation.

In addition, the simple bit bound pruning algorithm for this test as opposed to a brute force pair-wise comparison of all fingerprints of a query array and all elements of a single large database fingerprint array. This is because bit bound breaks the database into many smaller databases which is a much better approximation for other pruning algorithms which also break the database into many small arrays as opposed to maintaining the database as a single large

array. In addition the number of queries is varied in testing as efficiency for large query sets is also of importance to the SimDex algorithm described later.

In my testing summarized in Figure 10, the 2D Java array of arrays was found to perform the best. The BitString object performed the worst. CSR Rotate performed somewhat better than standard CSR but both performed worse than the 1D C++ array. This is likely due to the combination of the number of memory accesses that the CSR implementation requires for a single operation over a single integer and an insufficient sparseness to offset the overhead of memory access. Thus while CSR may potentially be a more efficient for fingerprints with longer 0 run lengths (24), it is not efficient for this particular set of fingerprints. The 1D C++ array performed slightly worse than the 2D Java array of arrays. In separate testing not included in this paper, 1D C++ array outperformed the 2D Java array of arrays in brute force pair-wise comparison which confirms the results of Gundersen and Seihaug's research into Java matrix representations (24). The only explanation as all other things are equal is that there are hidden overhead associated with manipulating few large continuous blocks of memory on the Java Heap as opposed to many very small blocks of memory. Thus 2D Java array of arrays is used as the representation for the rest of my experiments.

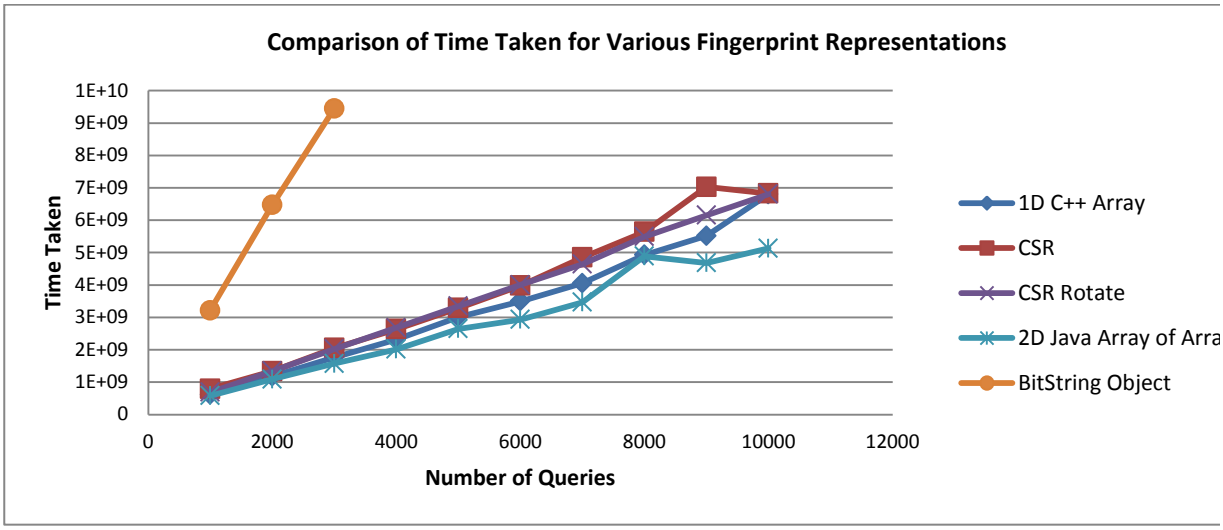


Figure 10. Results of the querying a bit bound pruning index when various fingerprint representations are used. The line representing the BitString Object is truncated due to rapid increase in time taken.

3.4 Information Content-Based Bit String Segmentation Algorithm

3.4.1 Overview. As can be seen from literature review in section 3.2.2, all pruning algorithms rely on some flavor of dimensionality reduction though such as bit string segmentation in the case of Aung and Ng (14), the UCI Algorithm (16, 33, 38), and *k*D-grid (15) or bit matching as in the case of the three bit tree algorithms previously mentioned (13, 18) to reduce high dimension bit strings into low dimension representations. This allows for large groups of similar bit strings to be pruned using a single operation. Ideally the sizes of these groups should be maximized (or equivalently, number of bins minimized) to reduce the overhead of pruning but also be minimized (or equivalently, the number of bins maximized) to increase the accuracy of the operation.

Thus creating or at least configuring similarity searching algorithms requires finding a balance between too little granularity and too much pruning overhead. Consider a chemical fingerprint

as existing in a length of string dimension space. One method is to modify algorithms to break this space down by dividing the space further such as increasing the *k* in *kD-grid*. Another method is to use a *heuristic,* an algorithm designed to increase the efficiency of other algorithms, to reorganize the divisions in this space to be more efficient. An example of this would be the split bit tree which intelligently selects bits in order of ability to divide the fingerprint space unlike Smellie's bit tree which simple divides the space without considering how well each bit divides the fingerprint space. Bit reordering and segmentation algorithms can be considered to be in this second class of heuristic algorithms as they try to maximize the pruning for a given dimensionality reduction.

This section defines such a heuristic algorithm for segmenting chemical fingerprints based on Shannon Entropy. Shannon Entropy is defined as $H(X) = \sum_{i=0}^{n} P(x_i) * log_n\left(\frac{1}{P(x_i)}\right)$ where $x_i \in X$ is an occurrence from the set of possible *n* occurrences (in this case 0 and 1 for a binary string) and $P(x_i)$ is the probability of occurrence x_i. Shannon Entropy is commonly used in compression or encoding as it gives the fewest possible bits an alphabet *X* can be encoded on based on probability of occurrence for each element. Benz *et al.* (20) has applied this recently to compressing chemical fingerprints. However, it can also be used as the measure of information as it is defined also as the expected value of information content. I use this interpretation of Shannon Entropy to segment bit strings to have bins containing the same amount of information content. This in turn will reduce the number of poor prunes.

3.4.2 Similar Approaches. There are few explicitly published reordering and segmentation heuristics for chemical similarity searches. However, there are implicit mentioned in the descriptions of several algorithms.

1. The first and most intuitive method is simply splitting a bit string into equal parts without any reordering by either selecting a sequential block of bits. This is used in *k*D-grid.

2. Similar to the first, the second method uses the modulus to segment a bit string by choosing bits a set distance apart from each other. This algorithm is based upon what Nasr *et al.* refers to as the 'exchangeability assumption' which is defined in "Hashing Algorithms and Data Structures for Rapid Searches of Fingerprint Vectors" (32) as:

> "it basically means that the validity of any formula should remain unchanged under any permutation of the fingerprint components. It is clear that in most cases real fingerprint components are not exchangeable. In fact the different components do not even have similar distributions—some features may be very common, others very rare. However, in spite of the deviations from exchangeability, previous work has shown that the exchangeability assumption leads to good global estimates of bulk properties, such as the distribution of the number *B* of 1-bits across all the molecules in a large database."

However as noted above, this assumption does not accurately reflect real world fingerprints as exchanging any two features may or may not change the result of an equation but as this paper illustrates can change the efficiency at which the final results of the equation is reached.

3. The third method is another intuitive method which is used by Aung and Ng (29) in his algorithm, read the database and reorder the bits according to which bit has the most 1's across all fingerprints. This is intuitive since both the union and intersection are affected primarily by combination of 1's and 1's and 0's and 1's but not 0's and 0's.

4. The last method is the also similar to this method and proposed by Kristensen *et al.* when describing the split bit tree (18). When choosing the bit to split, the split bit selection criteria chooses the bit that best splits the list of fingerprints at a node into equal halves. This is equivalent to saying, at each node select the bit with the highest Shannon Entropy since high Shannon Entropy implies a more uniform random distribution which in this case would be 50/50.

3.4.3 Algorithm Description. The aim of this heuristic algorithm is to define segments such that the information content in each segment as near as possible identical. Instead of making the assumption that all bits in a chemical fingerprint provide equal amounts of data as in the first intuitive method mentioned above, I test the assumption that all bits provide different amounts of information and equalizing this information content rather than the bits in each segment will generate the most efficient bins and hence reduce the overhead of pruning. The algorithm for Shannon Entropy-Based Bit String Segmentation is actually quite simple. It consists of four steps:

1. Calculate the binary entropy for $H(BIT_j) = P(0) * log_2\left(\frac{1}{P(0)}\right) + P(1) * log_2(\frac{1}{P(1)})$ and set all undefined *H(X)* to 0 since either P(0) or P(1) may equal 0.

2. Sort all bits of the bit string by increasing Shannon Entropy.

3. For a bit string of length *n*, Find the bit *i* which best divides the string such that $\left|\sum_{j=0}^{j=i-1} H(BIT_j) - \sum_{j=i}^{j=n} H(BIT_j)\right|$ is minimized. In other words divide the bit string into two continuous segments where both sections has as near as possible the same sum of entropies.

4. Optionally repeat Step 3 for each segment or some segments depending on pruning and memory considerations.

While only some sorted ordering or binning is required for Step 2, the choice of increasing entropy is made to address the observation that the number of 1's present at a given bit index across all fingerprints in a database tends to obey a power law as discovered by Benz *et al*. (21) In other words, most bits in a given fingerprint are set to 0 and therefore have low entropy and information content. Thus, the typical result of dividing a bit string into two segments is a segment with a very large amount of low entropy bits and a segment with a very small amount of high entropy bits.

For a given fingerprint database, there is a probability density function for each bit. While bits may not be independent as the features each represents may not be statistically independent from at least some of the other bits, there is still an underlying probability density function for each bit. Thus for a random fingerprint sampled from a database, the low entropy bits have a high probability of being set 1 or 0. It follow that for any two random fingerprints *A* and *B*

sampled from this database, there is a high probability that a given low entropy bit *i* will have either a high $P(A_i = 0, B_i = 0)$ or $P(A_i = 1, B_i = 1)$. This implies that a segment *s* created from only low entropy bits will more likely result in both:

$$\min(|A_s|, |B_s|) \approx A_s \cap B_s$$

$$\max(|A_s|, |B_s|) \approx A_s \cup B_s$$

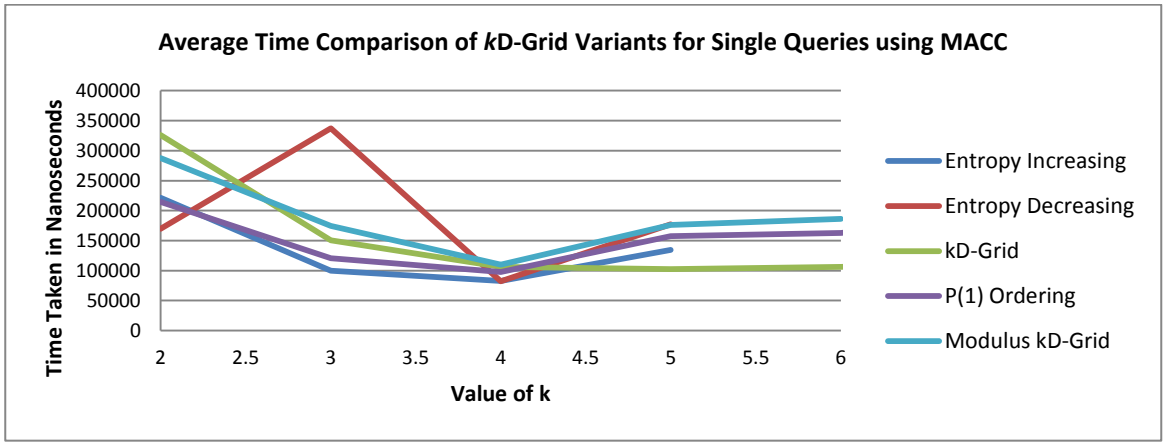
Because $P(A_i = 0, B_i = 0)$ or $P(A_i = 1, B_i = 1)$ coincides with $A \subseteq B$ or $B \subseteq A$ which is the case that maximizes $\min(|A_i|, |B_i|)$ and minimizes $\max(|A_i|, |B_i|)$. Thus it can be said that a segment created from only low entropy bits will generate a tighter bound than a segment created from a random collection of high and low entropy bits. Conversely, a segment created from only high entropy bits will more likely generate a looser bound than a segment created from a random collection of high and low entropy bits. Thus, by sorting the bit string according to entropy, the tighter pruning step can be done first which reduces the number of subsequently looser pruning steps. This is somewhat related to the match bit tree since it keeps track of all bits whose entropy value is 0 at each node.

However, it should be noted that a fingerprint database may or may not be representative of all possible fingerprints and thus each bit's probability density function is at best only an approximation of the probability of that bit for all fingerprints.

3.4.4 Methods. The full MACC and PubChem fingerprint databases are used in the following tests. Testing is conducted on a Windows 7 machine with an i5-2500K processor and 8gb of memory. The Shannon Entropy algorithm generated segmentation pattern will be tested by

implementing it as a preprocessing step for the *k*D-grid algorithm. KD-grid is the algorithm that from testing is the best algorithm for segmentation-based pruning with total number of segments k > 2 while the UCI modulus algorithm is best for k = 2. The Shannon Entropy algorithm will be tested against the approach of increasing the value of k with k bins of equal size. It will also be tested against the $P(1)$ algorithm used by Aung and Ng (29) and the modulus segmentation algorithm. These algorithms will be implemented using equal sized segments like standard *k*D-grid. The importance of ordering will also be tested by reversing the sort order so that Shannon Entropy is decreasing. In addition, the performance of the entropy segmented kD-grid algorithm will be measured against the very fast match-bit tree algorithm.

3.4.5 Results and Discussion. Figure 11 shows the average query time for standard *k*D-grid and *k*D-grid with preprocessing for various values of k. Segmentation based on information content balancing with increasing entropy performs the best in these tests for both databases. The algorithm performs better than the best search time of standard *k*D-grid and shows proportionally larger increases in performance for smaller values of k. When the entropy ordering is reversed, the algorithm only seems to perform well for k equals power of 2. After the initial segmentation into two segments, splitting only one of the two segments resulted in an average similarity search time worse than the initial segmentation into two segments. However, segmentation into 4 rather than 3 segments showed a performance boost similar to or better than increasing entropy ordering.



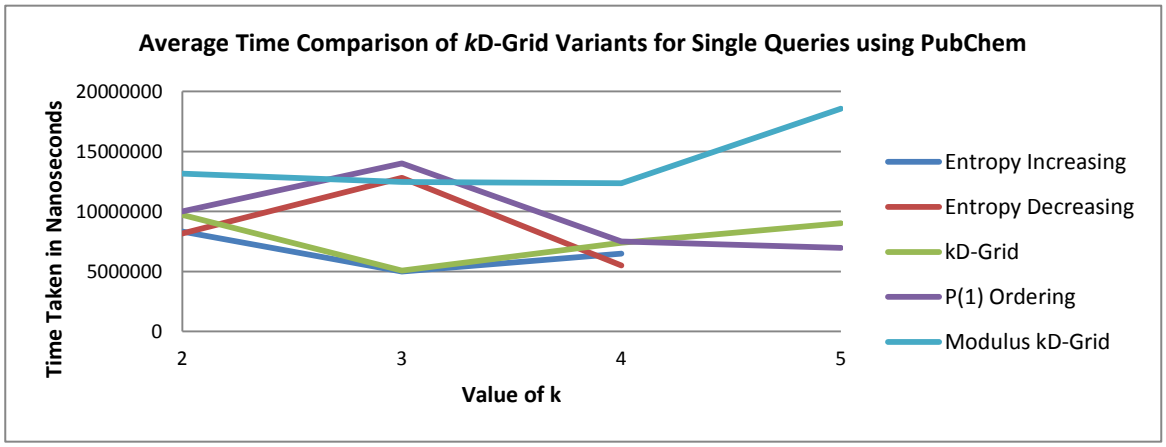


Figure 11. The average amount of time taken to run a single query from each fingerprint database is shown here. The experiment was performed 10,000 times with a 0.9 similarity threshold using a random non-repeated fingerprint for each value of k and then averaged. For values of k not powers of two, the segmentation patterns resulting in the best pruning were selected.

The P(1) decreasing ordering performed worse than the entropy orderings. It performed better than standard *k*D-grid for the MACC database but worse for the PubChem database. This discrepancy may be accounted for by the different percentage of 1 bits in each algorithm: 20.4% for MACC and 15.7% for PubChem. Modulus performed the worst out of the non-standard *k*D-grid algorithms. The reason for this poor behavior is likely the fact that fingerprints tend to cluster similar or related data near each other (13) for simpler human

comprehension. Thus selecting unlikely to be related spatially separated bits seems to be the least desirable method.

Average MACC Time and Memory Usage			
Algorithm	Time Taken(ns) for Single Query	Memory (mb) for Full Database Index	Percent Unpruned
Entropy Increasing, k = 4	82600	54	1.5%
Entropy Decreasing, k = 4	82224	51	1.5%
*k*D-grid, k = 4	106432	32	2.5%
*k*D-grid, k = 5	102326	38	1.4%
Match-Bit Tree, depth = 8	123157	43	0.33%
P(1) Ordering, k = 4	97485	56	2.1%
Modulus kD-Grid, k = 4	109745	50	2.3%
Average PubChem Time and Memory Usage			
Algorithm	Time Taken(ns) for Single Query	Memory (mb) for Full Database Index	Percent Unpruned
Entropy Increasing, k = 3	4978497	918	6.4%
Entropy Decreasing, k = 3	12811339	848	19%
Entropy Decreasing, k = 4	5511096	1588	3.9%
*k*D-grid, k = 3	5073624	933	6.2%
Match-Bit Tree, depth = 7	4311615	694	8.2%
Match-Bit Tree, depth = 9	3557940	905	2.1%
P(1) Ordering, k = 3	6963861	908	13%
Modulus kD-Grid, k = 5	18575392	2308	10.3%

Figure 12. These tables show the average amount of time taken to run a single query averaged from running 10,000 queries, the amount of memory needed store the each algorithm's index of the whole database, and the average percent of the database left unpruned. For *k*D-grid algorithms, the k with the best results is shown along with selected other k values that are helpful for comparison. For match-bit tree, both the first depth value with results better than the best *k*D-grid and the best depth value are shown for the PubChem database.

Figure 12 shows memory usage for the selected fastest times across all algorithms tested. Memory usage directly corresponds to the number of bins generated by each indexing algorithm and the granularity of pruning. It is evident that higher memory usage and larger number of bins tend to produce better pruning and faster results until a point is reached when

the pruning cost exceeds the amount of time saved from performing unneeded similarity calculations. Also, evident from Figure 12 is the efficiency of the match-bit tree pruning algorithm which is able to prune more strings for a lower memory cost and have a cheap enough pruning function to benefit from higher pruning percentages.

Additionally, observe that performance and memory usage and pruning is highly dependent on the characteristics of fingerprints used. The default ordering of fingerprint bits in the MACC database for example seems rather poor and the default *k*D-grid algorithm generates very poor segmentation choices. This in turn leads to inefficient pruning steps and poor performance compared to the entropy-based segmentation approaches. The case of the PubChem fingerprints is different, *k*D-grid shows nearly identical performance to entropy segmentation for the best value of k though the entropy segmentation is able to outperform it for other values of k from Figure 12. This is likely the result of the existing arrangement of the bits being good for continuous block segmentation. The evidence points to the default ordering of the bits being good for this type of block segmentation because the modulus algorithm performed relatively poor.

MACC Search Times (ns) for 100 Queries						
Entropy of Split Point	Entropy Increasing			Entropy Decreasing		
	k=2	k=3	k=5	k=2	k=3	k=5
0	162002357	37751639	8544167	64063370	36145313	18927913
0.05	118174976	15937625	10897833	1.23E+08	33726647	19516640
0.1	102151514	15462414	10290757	1.07E+08	31804965	18076077
0.15	100994897	14243906	11174003	99859738	31219036	17667732
0.2	98674508	13199250	11860696	1.07E+08	31522885	17832252
0.25	36877102	12722484	12135622	38268837	32937324	17803950
0.3	34166093	12314448	12101411	33926933	32961582	23564646
0.35	30137680	12157081	13778337	29868975	33444257	18772412
0.4	27324042	12380070	13534821	29684551	33787915	18228158
0.45	26134458	18196435	12641934	26675292	32959716	17951676
0.5	25370015	11398858	12299831	27841239	33311770	17679860
0.55	25197720	14221514	13188365	25008630	33471626	22880752
0.6	20572182	10833144		20895314	33878105	
0.65	19584439	10361975	11091587	19690491	34125974	17622947
0.7	17766630	11036228	13337645	17395915	33806264	17976556
0.75	17509121	12458132	13846757	17357973	33705188	17939547
0.8	15607030			15334281		
0.85	15892841	11878734	12475858	15636887	36029309	18577413
0.9	20034459	10799867	11497134	21040241	41274052	20797037
0.95	35825914	12179163	12071244	35387711	45443349	32329315

Figure 13. Since it is infeasible to test every possible split value, entropy values were tested starting with 0.025 at 0.05 intervals to create bins of size 0. Values differ The bin with the predicted split value and the best empirical values are marked. Red values mark predicted split value for entropy. Yellow highlighted values mark empirically determined best split for a segment. Empirical testing was done using 100 queries different from that of the first experiment. Values were summed and not averaged to make timing differences more apparent. Since the algorithm is iterative, gray highlighted bins contain selected split values for previous values of k.

Predicted Values: 0.8382, 0.64, 0.953, 0.39, 0.79, 0.9, 0.9725

Another interesting statistic is ability to choose a good splitting point for string segmentation. Figure 13 and 14 show the ability of entropy based information content balancing to choose the

best possible segmentation pattern. Each table shows the entropy value which best divides the string into two segments with the same amount of information content and show the difference between predicted and empirically tested best splitting choices. Observe that there is a tendency for this heuristic to predict better for increasing entropy than decreasing entropy bit orderings.

PubChem Search Times (ns) for 100 Queries						
Entropy of Split Point	Entropy Increasing			Entropy Decreasing		
	k=2	k=3	k=5	k=2	k=3	k=5
0	2239669902	1.46E+09	1.76E+09	2.23E+09	1.49E+09	1.54E+09
0.05	2090859641	8.46E+08	2.39E+09	2.07E+09	2.08E+09	1.49E+09
0.1	2000643775	1.42E+09	2.15E+09	2E+09	2.08E+09	1.51E+09
0.15	1943373318	1.4E+09	2.31E+09	1.94E+09	1.47E+09	1.47E+09
0.2	1888768773	1.35E+09	2.4E+09	1.88E+09	2.08E+09	1.37E+09
0.25	1245136390	1.36E+09	2.49E+09	1.82E+09	2.01E+09	1.37E+09
0.3	1174100348	1.35E+09	2.87E+09	1.75E+09	2.09E+09	1.36E+09
0.35	1071800797	1.27E+09	2.86E+09	1.07E+09	1.96E+09	1.35E+09
0.4	998916468	1.26E+09	2.69E+09	1E+09	1.99E+09	7.59E+08
0.45	1566628206	1.30E+09		1.59E+09	1.98E+09	
0.5	973879225	7.33E+08	2.19E+09	9.74E+08	1.37E+09	7.53E+08
0.55	953671849	1.32E+09	2.77E+09	9.56E+08	1.95E+09	1.37E+09
0.6	942186533	1.37E+09	2.8E+09	1.51E+09	1.96E+09	1.41E+09
0.65	874491900	8.17E+08	2.38E+09	8.86E+08	1.33E+09	1.35E+09
0.7	868394388			8.67E+08		
0.75	1409106771	1.33E+09	2.49E+09	1.41E+09	1.34E+09	1.37E+09
0.8	871372546	4.92E+08	2.55E+09	1.41E+09	1.1E+09	1.39E+09
0.85	800612364	4.07E+08	2.91E+09	7.99E+08	1.14E+09	1.38E+09
0.9	753919407	4.65E+08		7.67E+08	1.19E+09	
0.95	1557967724	4.83E+08	3.02E+09	1E+09	1.19E+09	9.6E+08

Figure 14. Same experiment as Figure 13 for PubChem fingerpint database.

Predicted Values: 0.7, 0.485, 0.9265, 0.32, 0.613, 0.824, 0.989

Figure 13 shows that for MACC fingerprints there are 5 selected empirically tested bins which are either the same or adjacent to selected predicted bins for increasing entropy as opposed to

only 2 for the case of decreasing entropy. Figure 14 shows a less pronounced affinity with increasing entropy with 4 adjacent or same bins as opposed to 3.

3.4.5 Conclusion. The development of intelligent heuristic algorithms is one method of increasing the efficiency of existing chemical fingerprint search algorithms. This paper has shown that this novel bit string segmentation based on information content balancing is one such algorithm. It has shown that it has the ability to perform superior bit string segmentation than simply dividing the bit string to have equal sized segments, reordering the bits in a string by P(1), or using modulus to select spatially distant bits. It has shown a 24% speed up for the 166 bit MACC fingerprint with a database of 100,000 chemicals, and an average speed up compared to standard kD-Grid for 881 bit PubChem fingerprint database of 1.3 million or about 15%. The success of this algorithm shows that the development of better heuristics based on bit reordering and segmentation is a feasible way of making more efficient fingerprint similarity search algorithms.

3.5 SimDex: Query Set Indexing for Batch Queries

3.5.1 Overview. To perform similarity searches, a query fingerprint must first be *hashed* or have statistics used for pruning the search index calculated for it. Depending on how the search index prunes, one or more hashes are generated for each query fingerprint and stored as a fingerprint object's member variable or associated some other way with it (see Figure 15 for a simple example). Hashes can also be calculated on the fly as a search is executed but this incurs penalties for recalculating or checking for recalculation of a fingerprint's hashes when

using multi-tiered (an index of indices) indexing such as match-bit tree which uses bit bound to first bin fingerprints by *popcount* or the number of 1's in the bit string (14, 16, 30, 32).

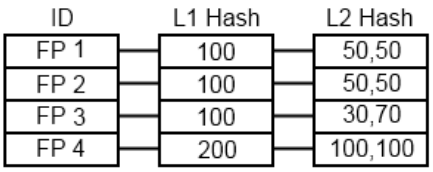


Figure 15. An example query set with associated hashes for the Nasr(32) indexing scheme which uses the Swamidass bounding index(39) to index modulus 2 indices.

Performing searches over the example bit bound followed by modulus 2 search index from Figure 16 using standard search algorithm is simple and straight forward:

- Submit FP1 for the search.
    - Prune level 1 index for hash 100.
        - Prune level 2 index for hash (50, 50).
            - Perform similarity calculations and thresholding for unpruned database.
- Submit FP2 for the search.
    - Prune level 1 index for hash 100.
        - Prune level 2 index for hash (50,50)
            - Perform similarity calculations and thresholding for unpruned database.
- Submit FP3 for the search.
    - Prune level 1 index for hash 100.
        - Prune level 2 index for hash (30, 70).

- Perform similarity calculations and thresholding for unpruned database.

- Submit FP4 for the search.
  - Prune level 1 index for hash 200.
    - Prune level 2 index for hash (100,100)
      - Perform similarity calculations and thresholding for unpruned database.

3.5.2 Algorithm Description.

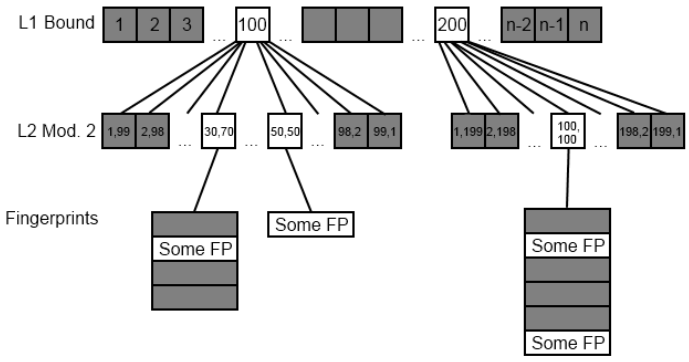


Figure 16. This shows an example search index being pruned for an exact search using the query set presented in Figure 15. Grayed out sections are pruned while white sections are unpruned. A search using single queries would take a maximum of query set size (4) multiplied by number of hashes each (2) or 4*2 = 8 pruning steps. A search using the SimDex batch query set index would only take total number of distinct hashes or 5 pruning steps.

The standard methodology is adequate for searching single queries at a time but this method is inefficient for executing batch searches. We proposed a novel search algorithm to address many of these inefficiencies called SimDex. For SimDex, instead of simply associating hash keys

by fingerprint, fingerprints are binned by hash keys as demonstrated in Figure 3. By using a structure-based on hashes instead of based on fingerprints, allows redundant pruning steps to be eliminated. Performing searches over Figure 17 is done by traversing the tree structure in a depth first fashion.

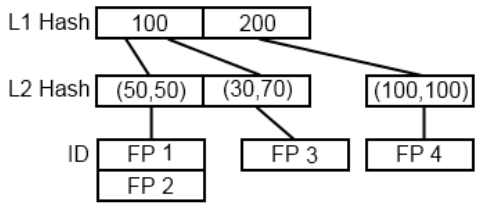


Figure 17. This shows the example query set from Figure 15 arranged in a tree structure used for the database search index instead of simply by member variable or some other type of association.

An example search would occur like so:

- Prune for level 1 hash 100.
    - Prune for level 2 hash (50,50)
        - Perform similarity calculations and thresholding using FP1 for unpruned database.
        - Perform similarity calculations and thresholding using FP2 for unpruned database.
    - Prune for level 2 hash (30,70)
        - Perform similarity calculations and thresholding using FP3 for unpruned database.
- Prune for level 1 hash 200
    - Prune for level 2 hash (100,100)

- Perform similarity calculations and thresholding using FP4 for unpruned database.

3.5.3 Performance. The total amount of time used by a search can be approximated by:

$$S_{Time} \approx H_{Time} + P_{Time} + CT_{Time}$$

where S_{Time} is the total time and H_{Time}, P_{Time}, and CT_{Time} represent the time taken for the three major parts of the search algorithms: hashing, pruning, and comparison calculation. For typical single query searches:

$$H_{Time} = N_{Query} * \sum_{t=1}^{Tiers} h(t)$$

$$P_{Time} \approx N_{Query} * \sum_{t=1}^{Tiers} p(t)$$

$$CT_{Time} \approx N_{Query} * N_{DB} * ct * \prod_{t=1}^{Tiers} p_{unpruned}(t)$$

where

- N_{Query} is the size of the query set
- $Tiers$ is the number of index levels for the search index
- $h(t)$ is the hash time for a particular level of index
- $p(t)$ is the prune time for a particular level of index
- N_{DB} is the size of the database
- ct is the compare and threshold time
- $p_{unpruned}(t)$ is the percent pruned for each level

The batch search algorithm we propose instead provides an upper bound for pruning time which as a side effect slightly increases the hash time to account for structuring the query set. The batch query searching used in SimDex as seen above only prunes per used hash instead of per query fingerprint hash. As the number of all possible hashes generated by current pruning methods is finite and relatively small, this drastically reduces the number of instances that the pruning routine is executed by limiting it to the number of used hashes. Thus two variables are changed in the time equation and CT_{Time} remains the same:

$$\overline{H}_{Time} \approx N_{Query} * \left[h_{insert} + \sum_{t=1}^{Tiers} h(t) \right]$$

$$\bar{P}_{Time} \approx \sum_{t=1}^{Tiers} h_{used}(t) * (p_{retrieve}(t) + p(t))$$

where h_{insert} is the time it takes to insert an entry into the query index, $h_{used}(t)$ is the number of used hashes on a level of index, and $p_{retrieve}(t)$ is the cost of retrieving all hashes out of a structure at a level of index. Plainly then $\bar{P}_{Time}$ gives an upper bound for pruning time based on the number of possible unique hashes per level $h_{max}(t)$:

$$\bar{P}_{Time} \leq \sum_{t=1}^{Tiers} h_{max}(t) * (p_{retrieve}(t) + p(t))$$

Thus, for sufficiently large query sets, prune time becomes constant and therefore very small compared to the overall time costs for sufficiently large searches. The trade off, as mentioned before, is that $h_{insert}$is introduced as an additional parameter in $\overline{H}_{Time}$. However, because there are many highly efficient data structures but few low cost pruning functions, h_{insert} generally is much less than $\sum_{t=1}^{Tiers} p(t)$ which gives an overall speed up for search times.

There is one drawback of this method that must be mentioned. At the point P_{Time} decreases to a constant, the speed up compared to non-query indexed searches will begin to decrease. This is due to the fact that the search algorithms are still O(n). The SimDex algorithm reduces the leading coefficient for a larger constant. Therefore at very large query sets, the upper bound of P_{Time} will be so small compared to H_{Time} and C_{Time} as to not have a meaningful contribution. However, query sizes are generally small enough to not encounter this phenomenon. This problem however can be addressed more concretely. Algorithms can be modified to prune better or novel algorithms with better pruning can be developed to potentially reduce C_{Time} by reducing $\prod_{t=1}^{Tiers} p_{unpruned}(t)$. As long as the upper bound of P_{Time} for those algorithms is not extraordinarily larger than current algorithms, SimDex will be able to produce good results.

3.5.4 Beneficial Cases. This naturally leads to the question of what conditions maximize the advantages of this execution structure. Intuitively this strategy shows benefits under two distinct conditions: First, this strategy would highly favor query sets consisting of a tightly grouped set of fingerprints which share identical or a very low number of hashes. We call this the *dense case*. The dense case minimizes the number of used hashes and may arise in instances where a user desires to search by a related group of chemicals. Thus this case is only conditionally useful as it largely must be encountered by design unlike the more general second case.

Second, this strategy produces benefits for query sets that contain very large number of fingerprints such that most or all hashes refer to multiple fingerprints. We call this the *full case*. The full case occurs when a query set becomes very large such that it begins to exhaust the number of possible hashes. In this case, as mentioned above, the pruning cost cannot grow any larger and becomes a constant. However, filling most hashes is unlikely to occur unless using the simplest of pruning algorithms with small hash spaces such as bit bound (39). More realistically, the full case will occur when all likely hashes are exhausted. This is due to the distribution of fingerprint features being non-uniform as can be seen in section 3.3. Thus, the derivative of the time taken to execute a search with indexed query set will monotonically decrease until reaching a constant as query set size increases. It is this case that will be the focus of the following experiments.

3.5.5 Implementation. The data structure used for the query index may vary in complexity. A query index can be intuitively and generically abstracted and implemented as a hash table of lists of fingerprints or a hash table of hash tables in the case of multi-tiered indices. A query set may also be indexed as if it were a database. For implementations used for testing in this paper, the latter option was chosen as it appears more time efficient (though perhaps less space efficient depending on implementation) as most of the published indexing schemes use a simple variations of n-ary trees (14, 16, 30, 32, 36) or arrays in the case of bit bound (39).

In addition, SimDex also has an advantage in that it allows for the comparison of extremely large databases that have pre-existing indices. Large databases must be indexed regardless to query out of them efficiently so this scheme can avoid the need to rebuild the query set index. While out of memory search implementations may be required to handle such a large

operation, it is worth considering as even a small reduction such as 10% for a search time over two large databases taking a long period of time such as two weeks is a substantial reduction in time.

3.5.6. Methods.

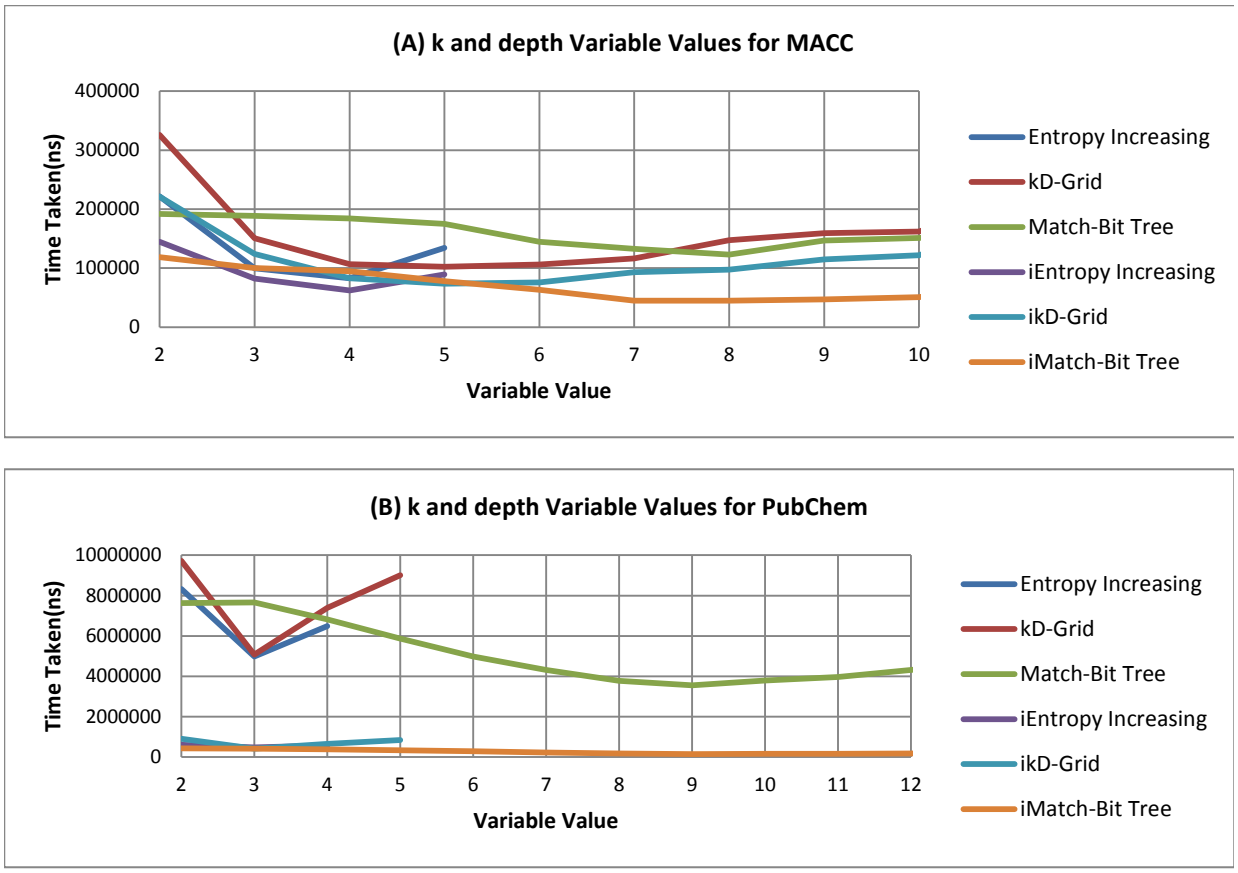


Figure 18. The average time taken for a single query for each similarity search algorithm for query set sizes of 10,000. Data series starting with 'i' denote query set indexing enabled in SimDex.

Again, the full MACC and PubChem fingerprint databases are used in the following tests. Testing are conducted on the same Windows 7 machine with an i5-2500K processor and 8gb of memory. The *k*D-grid algorithm, entropy segmentation *k*D-grid algorithm, match-bit tree, and bit bound were selected in addition to the naïve brute force pair-wise comparison algorithm for the experiments. Algorithms selected were based on speed and interesting properties. Since

all algorithms which required additional variable input for either segmentation or depth, 10,000 fingerprints were tested against each database which is shown in Figure 18. Since query set indexing upper bounds the pruning cost, this changes the prune cost to similarity calculation cost ratio. Thus Figure 18 also shows the variable input for query set index algorithms though there are no differences for the two databases tested.

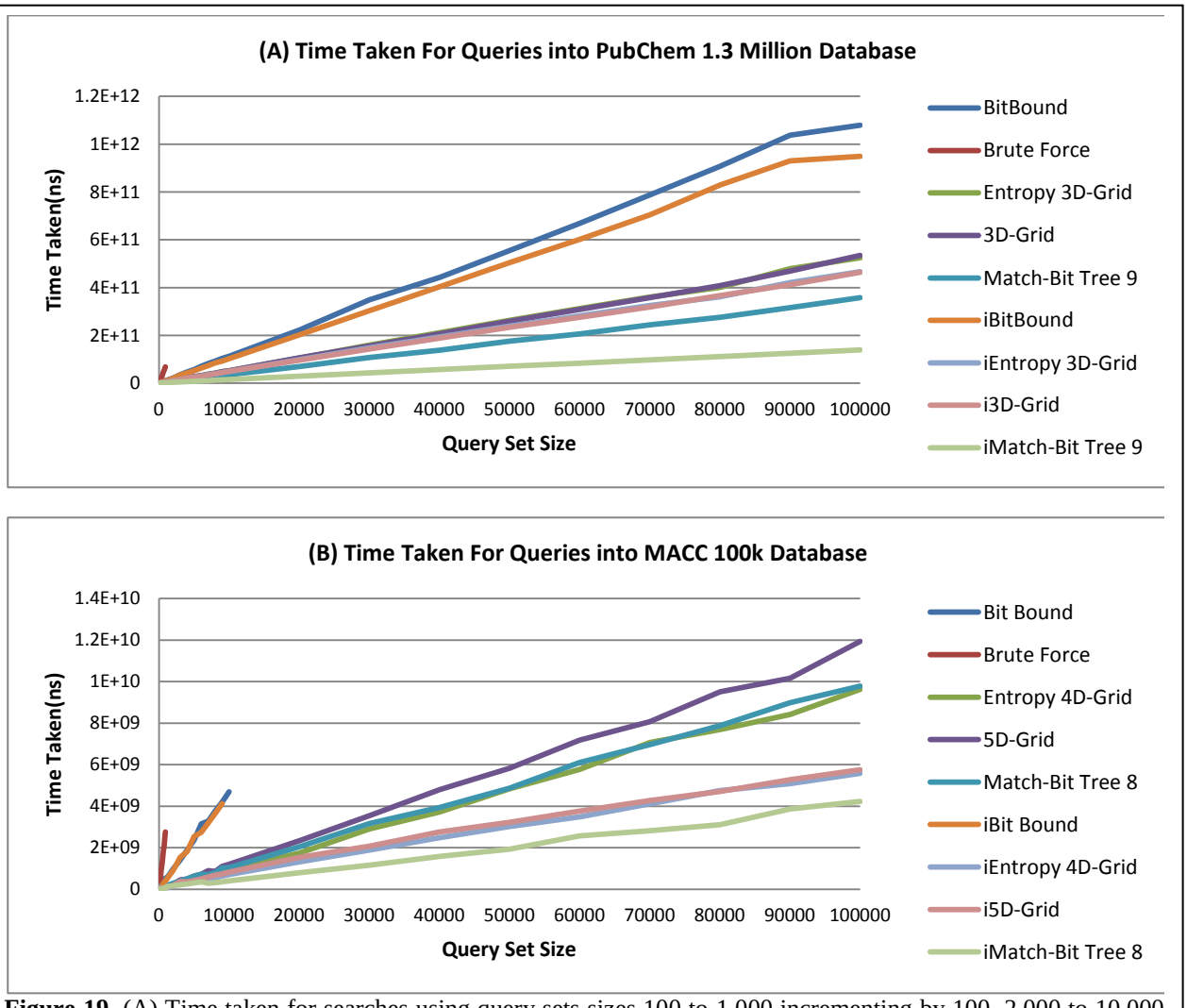


Figure 19. (A) Time taken for searches using query sets sizes 100 to 1,000 incrementing by 100, 2,000 to 10,000 incrementing by 1,000, and 20,000 to 100,000 incrementing by 10,000 for 881-bit PubChem Fingerprint. (B) Experiment was repeated for 166-bit. Time measured was the amount of time spent hashing or indexing the query set and executing the search. Brute force and bit bound are truncated to save space.

3.5.7. Results and Discussion.

Figure 19 shows how well the SimDex query set indexing approach scales with query size. As expected, searches with indexed query sets run in less time when compared with each algorithm's sequential querying counterparts due to the upper bound on pruning. The trend lines are not smooth due to the differences between each query set used because each query set is a new random sample unrelated to previous query sets. This is especially true for bit bound which has the fewest bins (the same as the number of bits in a fingerprint) so small variations in query set composition can result in large changes. All algorithms experience a significant speed up. The match-bit tree algorithm achieves a much larger amount of speed up with SimDex query set indexing compared to both the *k*D-Grid and entropy algorithms. This is especially evident for the PubChem fingerprints.

Figure 20 more clearly shows the speed up as fraction of time taken. The behavior is as predicted in section 3.5.3 as each algorithm converges towards a near constant speed up. Both the *k*D-Grid algorithms fail to reach this point, but both the match-bit tree and the bit bound algorithms do. Also observe that the match-bit tree algorithm fraction of time taken increases after reaching a minimum which is a symptom of reaching not only the full case but the full upper bound of P_{Time} for the MACC experiment. Bit-bound also reaches this point as match-bit tree uses bit-bound as the first level of pruning but it is harder to observe using these graphs compared to match-bit tree. This is actually somewhat of a surprising result as match-bit tree did not have an index for the 2nd tree tier since that tier requires a full fingerprint instead of some hash, only the 1st tier bit bound. This evidence instead points to bit bound and the loop

overhead to retrieve the second tier of indexes as being a significant cost in this java implementation.

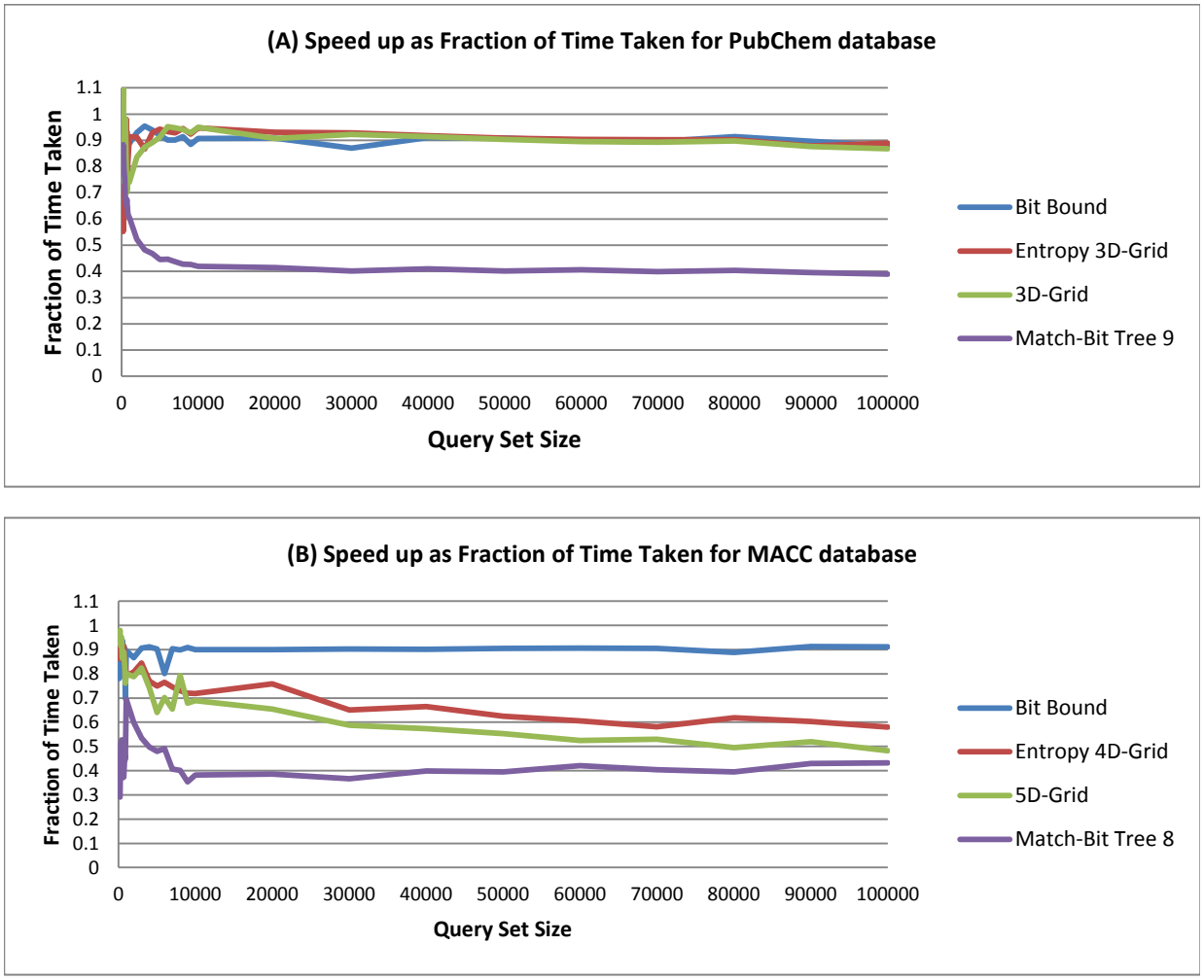


Figure 20. Speed up as fraction of time.

Additionally, Figure 20 shows that nearly any sized batch query larger than 100 query fingerprints receives some sort of improvement. In the case of bit bound, its ability to prune well for very small sizes is evident as *k*D-grid descends from 1 while bit-bound and match-bit tree ascend to a maximum before descending again.

Figure 21 shows the pruning costs as it varies by query size. *k*D-grid shows a curve as it has a negative derivative because it has not reached a constant speed up while match-bit tree and bit bound reach a constant quickly. The amount of it takes for an algorithm to converge is proportional to the number of hashes filled. Bit-bound fills quickly as there are only as many hashes as there are bits in a fingerprint. KD-grid at least for the standard case uses $\left(\frac{Fingerprint\ Size}{k}\right)^k$ hashes and therefore requires large query sets before reaching any form of the full case. For the MACC fingerprint bit bound and the 5D-Grid have a total of 166 and 4,0335,776 hashes respectively. For PubChem bit bound and 3D-Grid have a total of 881 and 25,325,845 hashes respectively. For this reason, kD-grid shows limited increases in speed for

PubChem.

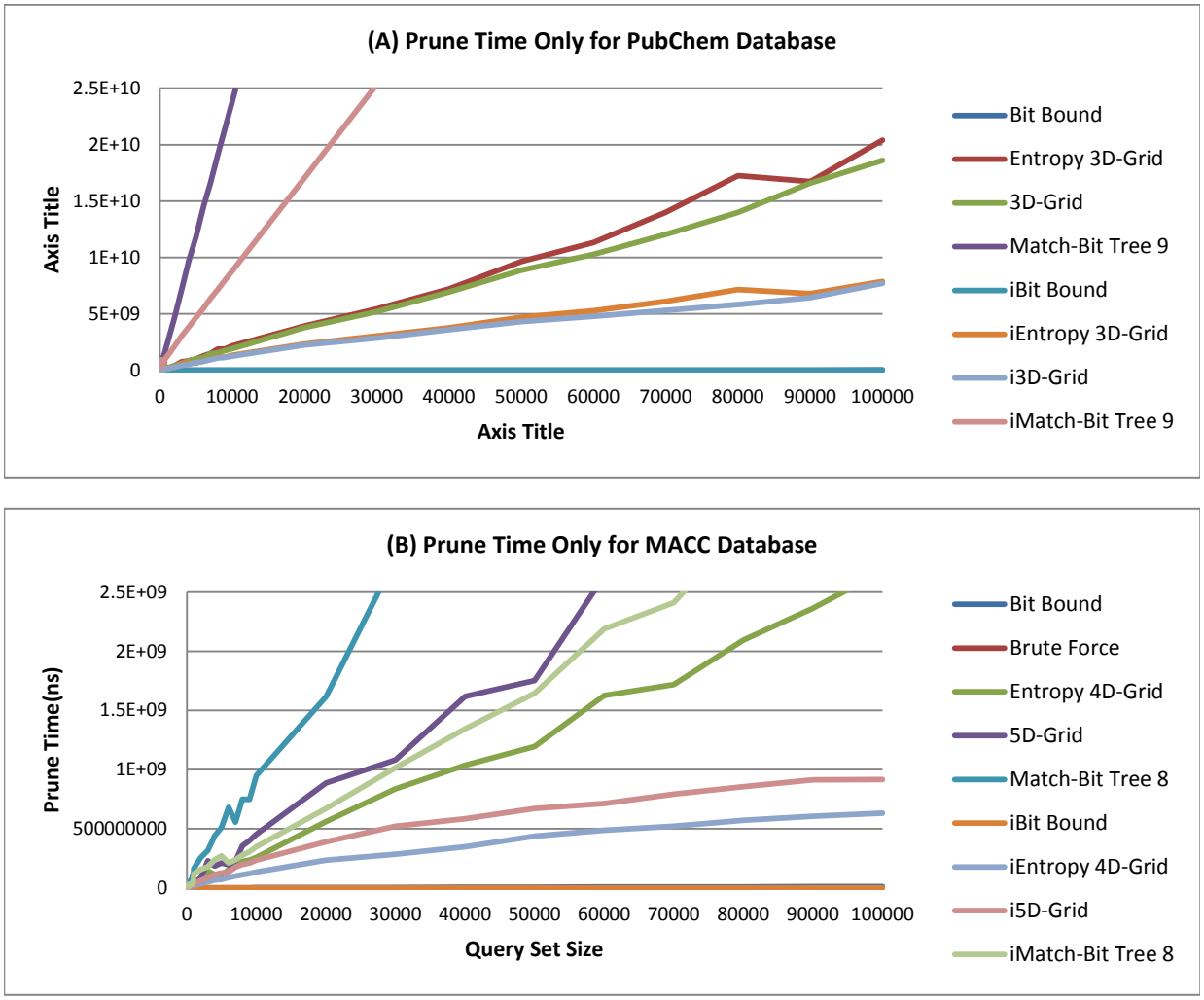


Figure 21. Pruning time only. Bit bound is extremely fast to prune and tiny compared to other algorithms.

3.5.8 Conclusion. The SimDex query set indexing algorithm shows good potential for accelerating batch searches. SimDex experimentally demonstrates performance increases typically between 111% and 250% range for the selected algorithms. The experiments also show that SimDex achieves a higher speed up after when the full case is reached but that performance increases are also present for nearly all sizes over 100 batch queries sets of at

least 100. They also support the theory that convergence occurs more quickly for smaller query sizes when the pruning index has a low number of hashes.

There are many further applications for the algorithm not explored in this paper. Extending query set indexing to other types of fingerprinting is one major application, specifically those relying on fixed length binary or multi-valued strings which can be thought of as an extension of binary string is one particularly interesting application. Additionally, since the query set indexing algorithm implemented in SimDex is independent of database indexing algorithms, it may be implemented for new indexing algorithms or indexing algorithms not addressed in this paper.

Chapter 4: Conclusion and Future Work

This thesis presents software tools and algorithms developed to better support HTS operations efficiently. K-Screen assists HTS laboratories in storing, managing, and presenting data to clients in a simple and coherent way. It is currently being used by the KU HTS laboratory and interest has been shown in potentially using it provide information management support by other laboratories. It is one of a few published (40) open-source HTS LIMS software and the only one which primarily focuses on presentation for the end-client. It can hopefully be further developed and adapted to the changing HTS environment or at least used as a basis for more advanced HTS LIMS software.

Next, the information content balancing algorithm is one of the first studies conducted on bit ordering and segmentation in chemical fingerprints. This algorithm shows improved

performance for finding more efficient segmentations of chemical fingerprint bit strings. This is a useful contribution because no algorithms for intelligent heuristic functions for bit string segmentation have been explicitly mentioned or experimentally tested in previous publications. Further development of heuristics based on reordering and segmentation of bit strings may result in better similarity search algorithms for both single dimensional bit strings and other feature based chemical representations. It could be argued that the match-bit tree shows much better potential than improved *k*D-Grid but this heuristic should be valid for any segmentation algorithm like *k*D-Grid. Thus the success of these heuristics is independent of the success of a single segmentation algorithm. Future developments in segmentation algorithms may show that segmentation algorithms can be superior to match-bit tree just like how match-bit tree showed that it was possible for bit matching algorithms can be superior to segmentation algorithms.

Lastly, the SimDex algorithm for batch query searches shows good performance for large databases and speeds up the current fastest state of the art algorithm by 2.5 times. The value of such a batch search algorithm will only increase as search requirements for HTS increase. As the datasets that are generated by the HTS process grow larger, the performance of search algorithms will need to increase to match this trend. It is also independent of search algorithm and may be adapted to indexing algorithms for other types of fingerprints such as graphical or string based ones. In addition, I have also developed a simple breakdown and estimation of similarity search times that can potentially be useful in finding bottle necks in algorithms. A paper detailing this algorithm is currently under consideration to be published by the ACS

Journal of Chemical Modeling and Information. Finally, I intend to publish the source code for SimDex to assist others in further benchmarking, developing, or creating new algorithms.

4. References

1. http://cran.r-project.org/web/packages/drc/index.html.
2. http://jpgraph.net/.
3. http://merian.pch.univie.ac.at/~nhaider/cheminf/moldb5doc.html.
4. http://pages.cs.wisc.edu/~ghost/.
5. http://www.apache.org/.
6. http://www.bioconductor.org/help/bioc-views/2.8/bioc/html/prada.html.
7. http://www.lsi.umich.edu/facultyresearch/centers/chemicalgenomics/chemoinformatics.
8. http://www.molinspiration.com/jme/.
9. http://www.mysql.com/.
10. http://www.php.net/.
11. http://www.r-project.org/.
12. http://www.yiiframework.com/.
13. *PubChem Substructure Fingerprint V1.3* Available from: ftp://ftp.ncbi.nih.gov/pubchem/data_spec/pubchem_fingerprints.txt.
14. Aung, Z. and S.-K. Ng, *An indexing scheme for fast and accurate chemical fingerprint database searching*, in *Proceedings of the 22nd international conference on Scientific and statistical database management*2010, Springer-Verlag: Heidelberg, Germany. p. 288-305.
15. Baldi, P. and D.S. Hirschberg, *An Intersection Inequality Sharper than the Tanimoto Triangle Inequality for Efficiently Searching Large Databases.* Journal of Chemical Information and Modeling, 2009. **49**(8): p. 1866-1870.
16. Baldi, P., D.S. Hirschberg, and R.J. Nasr, *Speeding Up Chemical Database Searches Using a Proximity Filter Based on the Logical Exclusive OR.* Journal of Chemical Information and Modeling, 2008. **48**(7): p. 1367-1378.
17. Baldi, P. and R. Nasr, J. Chem. Inf. Model., 2010: p. 1205.
18. Baldi, P. and R. Nasr, *When is Chemical Similarity Significant? The Statistical Distribution of Chemical Similarity Scores and Its Extreme Values.* Journal of Chemical Information and Modeling, 2010. **50**(7): p. 1205-1222.
19. Baykoucheva, S., *A New Era in Chemical Information: PubChem, DiscoveryGate, and Chemistry Central.* Online, 2007. **31**(5): p. 5p.
20. Benz, R.W., S.J. Swamidass, and P. Baldi, J. Chem. Inf. Model., 2008. **48**: p. 1138.
21. Benz, R.W., S.J. Swamidass, and P. Baldi, *Discovery of Power-Laws in Chemical Space.* Journal of Chemical Information and Modeling, 2008. **48**(6): p. 1138-1151.
22. Donofrio, N., et al., *'PACLIMS': A component LIM system for high-throughput functional genomic analysis.* Bmc Bioinformatics, 2005. **6**: p. -.
23. Droit, A., et al., *PARPs database: A LIMS systems for protein-protein interaction data mining or laboratory information management system.* Bmc Bioinformatics, 2007. **8**: p. -.
24. Gundersen, G. and T. Steihaug, *Data structures in Java for matrix computations.* Concurrency and Computation: Practice and Experience, 2004. **16**(8): p. 799-815.
25. Hettne, K.M., et al., *A dictionary to identify small molecules and drugs in free text.* Bioinformatics, 2009. **25**(22): p. 2983-2991.
26. Holliday, J.D., C.Y. Hu, and P. Willett, *Grouping of coefficients for the calculation of inter-molecular similarity and dissimilarity using 2D fragment bit-strings.* Comb Chem High Throughput Screen, 2002. **5**(2): p. 155-66.
27. Inglese, J., et al., *Quantitative high-throughput screening: a titration-based approach that efficiently identifies biological activities in large chemical libraries.* Proc Natl Acad Sci U S A, 2006. **103**(31): p. 11473-8.

28. Irwin, J.J. and B.K. Shoichet, *ZINC - A free database of commercially available compounds for virtual screening.* Journal of Chemical Information and Modeling, 2005. **45**(1): p. 177-182.
29. James, C.A., D. Weininger, and J. Delany, *Daylight Theory Manual*2004.
30. Kristensen, T.G., J. Nielsen, and C.N. Pedersen, *A tree-based method for the rapid screening of chemical fingerprints.* Algorithms Mol Biol, 2010. **5**: p. 9.
31. Mayr, L.M. and P. Fuerst, *The future of high-throughput screening.* J Biomol Screen, 2008. **13**(6): p. 443-8.
32. Nasr, R., D.S. Hirschberg, and P. Baldi, *Hashing Algorithms and Data Structures for Rapid Searches of Fingerprint Vectors.* Journal of Chemical Information and Modeling, 2010. **50**(8): p. 1358-1368.
33. Pissanetzky, S., *Sparse matrix technology*1984, London ; Orlando: Academic Press. xiii, 321 p.
34. Prilusky, J., et al., *HalX: an open-source LIMS (Laboratory Information Management System) for small- to large-scale laboratories.* Acta Crystallographica Section D-Biological Crystallography, 2005. **61**: p. 671-678.
35. Sheridan, R.P. and S.K. Kearsley, *Why do we need so many chemical similarity search methods?* Drug Discovery Today, 2002. **7**(17): p. 903-911.
36. Smellie, A., *Compressed binary bit trees: a new data structure for accelerating database searching.* J Chem Inf Model, 2009. **49**(2): p. 257-62.
37. Sun, B., P. Mitra, and C.L. Giles, *Independent informative subgraph mining for graph information retrieval*, in *Proceeding of the 18th ACM conference on Information and knowledge management*2009, ACM: Hong Kong, China. p. 563-572.
38. Swamidass, S. and P. Baldi, J. Chem. Inf. Model., 2007. **47**: p. 952.
39. Swamidass, S.J. and P. Baldi, *Bounds and Algorithms for Fast Exact Searches of Chemical Fingerprints in Linear and Sublinear Time.* Journal of Chemical Information and Modeling, 2007. **47**(2): p. 302-317.
40. Tai, D., R. Chaguturu, and J. Fang, *K-Screen: A Free Application for High Throughput Screening Data Analysis, Visualization, and Laboratory Information Management.* Combinatorial Chemistry & High Throughput Screening, 2011. **14**(9): p. 757-765.
41. Tolopko, A.N., et al., *Screensaver: an open source lab information management system (LIMS) for high throughput screening facilities.* Bmc Bioinformatics, 2010. **11**: p. -.
42. Wang, X., et al., *G-hash: towards fast kernel-based similarity search in large graph databases*, in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*2009, ACM: Saint Petersburg, Russia. p. 472-480.
43. Wang, Y.L., et al., *PubChem: a public information system for analyzing bioactivities of small molecules.* Nucleic Acids Res, 2009. **37**: p. W623-W633.