An explorative study on applying functional language for approximation algorithms

Muhammad Akram[1], & Muhammad Imran Shafi[2]
[1]*Abdul Razaq Institute of Modern Languages and Computer Sciences, Mirpur AK*
[2]*Nackademin YRKESHÖGSKOLA Stockhlom, Sweden*
akram.moghal@gmail.com, cancerbyname@hotmail.com

Abstract: Approximation algorithms are widely used for problems related to computational geometry, complex optimization problems, discrete min-max problems and NP-hard and space hard problems. Due to the complex nature of such problems, imperative languages are perhaps not the best-suited solution when it comes to their actual implementation. Functional languages like Haskell could be a good candidate for the aforementioned mentioned issues. Haskell is used in industries as well as in commercial applications, e.g., concurrent applications, statistics, symbolic math and financial analysis. Several approximation algorithms have been proposed for different problems that naturally arise in the DNA clone classifications. In this paper, we have performed an initial and explorative study on applying functional languages for approximation algorithms. Specifically, we have implemented a well known approximate clustering algorithm both in Haskell and in Java and we discuss the suitability of applying functional languages for the implementation of approximation algorithms, in particular for graph theoretical approximate clustering problems with applications in DNA clone classification. Basic purpose of study is to get the proof of idea. We also further explore the characteristics of Haskell that makes it suitable for solving certain classes of problems that are hard to implement using imperative languages.

Keywords: Approximation algorithms, functional languages, imperative languages, Bipartite graph, Haskell.

1. Introduction

Clustering problems have received a lot of attention recently (see e.g., [1, 2, 3, 4,5,6,7, and 8]). Generally clustering problems refer to a set of problem in which we intend to divide our data (e.g., text, numbers, pictures, nodes, people, etc) in different groups or clusters. Formation of cluster and populating any cluster may depend upon the problem underlying. [2] Grouping (clustering) criteria may depend upon some "similarity test" for data items. Data items that are "similar" to each other may be kept into one cluster or even totally opposite.

In this paper, we will consider a specific subset of clustering problems that has been proven to be NP-hard. Specifically, we will consider the problem of clustering binarized fingerprints with at most p missing values (CMV(p) for short) which arises very naturally in the problem of characterizing DNA clone libraries, especially in the so called oligonucleotide fingerprinting method [1]. CMV(p) is a combinatorial optimization problem where one tries to identify clusters and resolve the missing values in the

fingerprints simultaneously. The objective is to minimize the cardinality of the partition and the motivation behind is the minimum description length (MDL) principle (or Occam's razor) which makes it natural to consider the problem of partitioning the fingerprints into the smallest number of clusters, each consisting of similar fingerprint vectors. Furthermore, this approach is also consistent with the hypothesis that bio-molecular diversity is a precious resource [9]. The CMV(p) problem was first considered in [9] where it was shown to be NP-hard for $p \geq 3$ and polynomially solvable for $p = 1$. In [1] it was shown to be NP-hard also for $p = 2$. Furthermore, a factor $min(1 + \ln n, 2 + p \ln l)$ approximation algorithm for the CMV(p) problem was proposed in [1]. The aforementioned approximation algorithm runs in time O(nl2^p) , where n is the number of binarized fingerprints, and l is the length of the fingerprint vector [1]. Note that for $p = O(\log n)$, the aforementioned algorithm runs in polynomial time.

Hence, one possible technique to attack the aforementioned clustering problem is by designing approximation algorithms [3, 4, 5, 6, 7, and 8]. By using approximation algorithms it could be possible to get a near optimal solution for a computational "hard" problem. By using approximation algorithm, one will guarantee to solve the problem but solution may differ from optimal solution with not a great degree [10].

The main aim of our paper is to conduct an explorative study on advantages in using functional programming implementing approximation algorithms; in particular we will consider approximation algorithms for approximate clustering problems. Pure functional languages not only provide the general computational solutions, but also well because of their purity [11]. These languages encourage the mathematical thinking to its users. We have implemented the approximation algorithm proposed in [1] for clustering problem in Haskell. Because Haskell is a general purpose and non-strict purely functional language that can be used to implement the approximation algorithms and these algorithms are proposed for the different problems that naturally arise in the DNA clone classification.

Moreover, the approximate clustering algorithm in proposed in [1] is also implemented in an imperative language, more specifically, we choose Java here. The purpose of implementing the solution in Java was to compare the difference of thinking and modeling the problem because as we have read before that Haskell gives the natural and mathematical way of thinking.

Specifically, our paper is an effort to investigate the suitability of functional languages for solution of approximation problems and implementation of some approximation algorithms using functional language. We are going to explore characteristics of functional language (e.g., Haskell), that make it suitable for solving certain classes of problems and implementing some problems that are hard to implement using imperative languages.

2. Background & Related Work

Much work has been done in the fields of functional programming languages, approximation algorithms, DNA clone clustering problems and greedy algorithms [7,11,12,13,14,15, and16] so the problem and problem solving method is not new for computer science researchers. Functional programming languages have existed in the field of computer science for a while and proven their relative advantages.

According to [12], Haskell is a functional programming language that has been developed on the principles of "do less, get more" providing higher level of abstraction for programmers and software architects, its structure that makes it suitable for solving many Artificial Intelligence related problems, its strength for developing problem prototype, its connection with mature and well understood computer science theory. Haskell provides a different way of thinking for programmers and this way is suitable for many computer science problems, especially for many problems that cannot be solved efficiently using imperative languages like C, C++, Java etc. Haskell promotes and encourages mathematical thinking and avoids complicated details of object oriented methodology that researchers find hard to understand easily. Haskell has been used by researchers and programmers for different projects and have shown its relative benefits.

DNA fingerprints are nothing new to researchers and there exist many problems that are based of study of DNA fingerprints. This technique has been used to study genetic material [16] and has been proven very beneficial to solve many problems including guilt of accused, ethnicity issues, immigration arguments and creation of high quality plants/animals. DNA clone clustering technique is used in forensic medicine and many other research problems. Many algorithms provide solutions of DNA clone clustering problem including the one that we have implemented which was provided by Figueroa et al. in [1].

Greedy algorithms are very famous method for solving a certain type of problems and have been used many famous problems including money counting, greedy scheduling, 1/0 knapsack, greedy shortest path finding in graphs and many more. Greedy algorithms are useful in solving typical kind of iterative decision making problems. Greedy approach takes best possible choice for each step (iteration) without taking care of its affect of overall solution (hoping that local optimum solution will lead to global optimum solution that is not the case for always).

A lot of study has been done in field of Approximation Algorithms. Many NP hard problems that do cannot be solved in polynomial using other approaches, approximation algorithms come with an optimum or near optimum solution for them [17]. Approximation algorithms do not guarantee to provide optimal solution but there is mostly little compromise that makes it suitable for solving such problems.

3. Our Contribution

Underlying problem (DNA Clone Clustering with Missing Values) is solved using greedy approximation algorithm by using a functional programming language (Haskell) and an imperative language (Java). Emphasis is to investigate the characteristics and differences of implementation using imperative and functional languages with respect to the aforementioned features, namely syntax, way of thinking, list/set operation, pattern matching etc. These features were selected by us during the implementation of our approximation algorithm in imperative and functional languages. We observe these features during the implementation of under studied problem.

The idea behind this explorative study is to find suitability and relative advantages of a kind of programming technique (object oriented or functional) for a typical graph problem. Furthermore this study explores the suitability of two commonly used programming paradigms for researchers in applied mathematics.

The Greedy Clustering algorithm below is proposed by Figueroa et al. in [1]. The Construction of H = (A;B;E) algorithm below is also proposed by Figueroa et al. in [1]. For more details on the algorithms, please see [1].

Algorithm Construction of H = (A;B;E) (Figureoa et al. [1])
1 A := Ø;
2 B := F
3 E := Ø;
4 for all x ϵ B do
4.1 for all y ϵ res(x) do
4.1.1 if y (not ϵ) A then
4.1.1.1 Insert(y, A)
endif
4.1.2 Insert(E, xy)
endfor
endfor
End Construction of H = (A;B;E)

Algorithm Greedy Clustering (Figueroa et al. [1])
1 for i := 1 to n do
1.1 Q_i:= Ø;
endfor
2 for all x ϵ A do
2.1 Insert(x,Qdeg(x))
endfor
3 for i := n to 1 do
3.1 while Q_i is not empty do
3.1.1 x :=Delete(Q_i)
3.1.2 Begin reporting a new cluster
3.1.3 for all y neighbor of x do
3.1.3.1 Report(y)
3.1.3.2 Delete(y)
endfor
3.1.4 Delete(x)
endwhile
endfor
End Greedy Clustering

Approach to solve the problem is based on greedy strategy. In this strategy, an iterative approach is taken. On every decision level, the best possible option is chosen (largest possible cluster in this case). Greedy clustering process does not guarantee to always provide optimal solution but we have test this approach and often it comes up with optimal or a near optimal solution.

3.1. Findings of Study

While implementing our problem using Java as well has Haskell, we made different observations and we are putting these observations in this document. It should not be taken as comparison of two languages. These two languages (Java and Haskell) are entirely different languages and entirely different approach is adopted in any of two languages while solving a typical problem.

3.1.1 Syntax

Syntax of Haskell is totally different from imperative languages like Java, C/C++ and it takes a while to be comfortable with its syntax for a person coming from C/C++ background but it suits best for a person with mathematical or research background. Once Haskell syntax is understood, it is a matter of fun. At the same time java syntax is purely object oriented. So it takes a lot of efforts from a person not having background of object oriented programming.

User defined type in Java typically contains class declaration, data member in class, constructors to initialize objects and member functions. A new (user-defined) type in Java can typically be defined as:

```
public class ClassTypeName {
ClassDataMambers ….
ClassProcedures
}
```

Haskell defines data type simply with only type structure declaration.

```
type TypeName = Type-Structure
```

e.g.

```
type GraphNode = [Int]
```

Data type related procedures are defined independent of type declaration. This way of defining data types is simpler for researchers since it is easier to understand as fingerprint node is a set or list of integers.

3.1.2 Way of Thinking

Haskell takes a pretty simple conceptual approach for solving problems. While solving a problem, problem solver have to take care of structures required, structures combinations and operations. Function or procedure is a key to problem solution. Design approach starts from identification of high level functions then intermediate level functions are defined and at the end low level functions that are actually solving small portions of a problem. High level functions are more about combining intermediate and low level functions in a way that solves the problem under discussion. Structures of data are defined only on "needed" basis. A structure is not defined until it is really "needed".

Haskell approach:

1. High level functions identifications
2. Intermediate functions identification to facilitate high level function to structure the solution
3. Identification of structures necessary to solve the problem
4. Identifying and implementing low level functions for solving typical small parts of complete problem

This approach suits best to those who are conducting since they do not come from a programming background and it is hard for them to following pure object oriented programming way of thinking. If one observes in mathematical terms and Haskell supports this way.

Java way of thinking for solution of a problem evolves in terms of objects and classes. Java programmers identify different user-defined types (classes), methods to perform necessary operations and generality of solution is also kept in mind (Java types are independent reusable codes). So Java code is a lot extendable, general purpose and containing many details.

Java Style:

1. Objects identification
2. Objects interaction
3. Reusability of code
4. Emphasis on being general purpose (no emphasis on any typical instance of problem)

5. Containing details (exception handling, things to facilitate general purpose solutions)

This approach suits to a programmer with object oriented background because they are habitual of thinking in terms of objects and interfaces.

3.1.3 Approximate Algorithms

Algorithms that cannot be solved using imperative languages are often solved using approximation algorithms without compromising efficiency significantly.
Haskell solution for underlying approximation algorithm, i.e. CMV(p) took less effort and supported better way of thinking. Code was so simple and was close to actual way of solving the problem e.g. a list is a set of integers, an edge is a pair of two nodes etc.
While implementing same problem with Java, problem faced was that way of thinking for problem solution and way of actual implementation were different. Problem was first thought and understood in its real meanings (sets, pairs etc) and later it was transformed in object oriented way to get things work.

3.1.4 List Comprehension

List comprehension is a phenomenon for defining lists using existing lists.
Since Haskell is very rich language for list operations, it has also very efficient and comprehensive ways for list comprehension. It is similar to set builder notations in mathematics.
e.g.

```
list :: [Int]
list = [1,2,3,4,5,6,7,8,9,10]
newlist :: [Int]
newlist = [x*x |  x <- list] ---- newlist is defined from existing list and contains values as square of values of existing list.
```

It has been declared earlier that Java does not facilitate programmer directly in terms of lists and list operations but Java facilitates greatly to define user defined lists and list operations. Not much built-in operations are available for this purpose.

3.1.5 Pattern Matching

Pattern matching is a phenomenon in which one tries to identify the presence of a particular pattern within given data. Pattern matching can be used to know the relevance of data with a given pattern, identification of structures and replace / remove matching parts from the data.
Pattern matching is a greater strength of Haskell. It offers clear syntax and flexible options for pattern matching. In the example given below, a function (reportOneCluster) has been defined and is provided to match with three given patterns. Whichever pattern matches best when function is called, will be executed. This thing gives programmer greater strength to define different patterns for different instances of same set of problem independently from each other.
e.g.

```
reportOneCluster :: [Queue]->[(GraphNode,GraphNode)]->[GraphNode]
```

```
reportOneCluster [] _                = []
reportOneCluster _ []                = []
reportOneCluster (q:qs) elist
            |(queNodes q) == []   = reportOneCluster qs elist
            | otherwise              = (cluster (head (queNodes q)) elist)
            where
            maxDegreeNode = head (queNodes q)
```

Java does not facilitate us for pattern matching in this way. In java, different instances of same problem (when solved using one function) are handled through conditional structures (if, if-else, switch statement). There is not built-in support for direct pattern matching in Java.

3.1.6 Algebraic Data Type

Algebraic data types are a kind of composite types. Parts of algebraic data types are made up of other data types. One can define all constructors or an algebraic data type while defining it.

e.g.

```
data Maybe t1 = Changed t1 | Original t1| Nothing
```

Algebraic data types (ADTs) are fundamental part of Haskell. Haskell has built-in support for ADTs. ADTs in Haskell are closed and one has to define all possible constructors while defining an ADT. It is not possible to add more constructors for an ADT at runtime. [38]

Java is an open language and if one can force (in some way) a class can have only limited number of sub-classes, algebraic data types can be implemented in Java as well. In current scenario, java does not support ADTs.

3.1.7 Lazy Evaluation

Lazy evaluation is a feature of specific set of programming language in which a computation (calculation) is delayed until it becomes "necessary" to evaluate that expression. Lazy evaluation gives significant performance increase as only desired or necessary calculations and processor's cycles are not wasted in doing unnecessary computations.

Haskell is a language that follows rule of lazy evaluation. Lazy evaluation feature enables Haskell to support infinite data structures. Haskell takes an approach for calling a function as call-by-need that means that a function is not called and is not evaluated until it becomes necessary to call it.

e.g. x = expression-to-be-evaluated.

In above statement, x is a variable in which expression-value will be stored. But variable x is in itself not very important from point of view of lazy evaluation criteria. x is important only when it is going to be used at any place later. So expression will not be evaluated until there comes a point when x is used in any further calculation or any operation. This thing happens even if program flow seems to be evaluating x before that point.

Java is not the language supporting lazy evaluation. It does not delay expressions' evaluation. Java typically program sequence and statements or expressions are evaluated in the order in which program instructions force them to.

4. Conclusion and Future Work

We have conducted an initial explorative study that makes use of approximation algorithm with greedy approach to solve the problem clustering fingerprints with at most p missing values (CMV(p) for short). Two techniques have been used for solving the same problem using Object Oriented (Java) and Functional programming (Haskell). Problem requires clustering of DNA fingerprints in a way that reduces total number of clusters formed. A greedy approach was proposed [21] to solve the problem. In [1] it has been proved that the value of any solution returned by the algorithm in [1] always is upper bounded by min(1 + ln n, 2 + p ln n) times OPT, where OPT denotes an optimal solution of the aforementioned problem.

The main aim of our study was to get the proof of idea and to find degree of suitability of programming languages, in particular functional programming languages, for approximation algorithms, in particular approximate clustering algorithms. From our literature study and our practical experiences, we found that different features of Haskell include its simple syntax, built-in support for mathematical and algebraic data structures, rich operations for lists and list comprehension, pattern matching, emphasis on abstraction to a higher degree then Java. Our implementation of the approximate clustering algorithm uses the aforementioned features, thus Haskell is an excellent choice for implementation in this case. "Haskell is a general purpose, purely functional programming language featuring static typing, higher order functions, polymorphism, type classes and monadic effects" [18]. According to our observations, features described earlier are strong enough to attract new learners as well as experienced programmers.

With our work, we think that a new window of opportunity has been explored with more emphasis. A language (Haskell) has been explored that supports and promotes mathematicians way of thinking and provides built-in support for complicated research problems that are hard to implement in imperative languages. Its syntax is simple and once one gets comfortable with it, programming is a lot easier and well aligned with human thought process.

An interesting future work would be to conduct an observation study on undergraduate lectures in approximation algorithms, where a functional language (e.g., Haskell) is actually used for implementation of the algorithms.

5. Acknowledgement

Authors have done this work during MS study in Blekinge Institute of technology (BTH), Sweden. We are really thankful to our thesis supervisor Dr. Mia Persson, for her cooperation towards the completion of this study.

6. References:

[1] Figueroa, A., Goldstein, A., Jiang, T., Kurowski, M., Lingas, A., and Persson, M. Approximate clustering of incomplete fingerprints. Journal of Discrete Algorithms 6(1):103-108, 2008.

[2]. Sanjay Dasgupta, C.H. Papadimitriou, U.V. Vazirani, *"Algorithms"*, 2006, ISBN-13: 978-0073523408.

[3]. Bansal, N., Blum, A. & Chawla, S. (2004), 'Correlation Clustering', Machine Learning 56(1–3), 89–113.

[4]. Charikar, M., Guruswami, V. & Wirth, A., Clustering with Qualitative Information, in'Proc. 44th Annual Symposium on Foundations of Computer Science (FOCS 2003)', 2003, pp. 524–533.

[5]. Demaine, E. & Immorlica, N. Correlation Clustering with Partial Information, in 'Proc. 6th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX 2003)', 2003, pp. 1–13.

[6]. Drmanac, S., Stavropoulos, N.A., Labat, I., Vonau, J., Hauser, B., Soares, M.B. & Drmanac, R. (1996), 'Gene-representing cDNA clusters de-fined by hybridization of 57,419 clones from infant brain libraries with short oligonucleotide probes', Genomics 37, 29–40.

[7]. Figueroa, A., Borneman, J. & Jiang, T., 'Clustering binary fingerprint vectors with missing values for DNA array data analysis', Journal of Computational Biology 11(5):887–901, 2004.

[8]. Figueroa, A., Goldstein, A., Jiang, T., Kurowski, M., Lingas, A., and Persson, M. Approximate clustering of incomplete fingerprints. Journal of Discrete Algorithms 6(1):103-108, 2008.

[9]. Figueroa, A., Borneman, J. & Jiang, T., Clustering binary fingerprint vectors with missing values, to appear in Journal of Computational biology, 2004.

[10]. Rajeev Motwani, Lecture Notes on Approximation Algorithms, Volume I

[11]. Paul Hudak, "Conception, evolution, and application of functional programming languages",ACM Computing Surveys (CSUR), Volume 21, Issue 3, September 1989, ISSN: 0360-0300.

[12]. Bruce J. MacLennan, "*Functional Programming Practice and Theory*", ISBN 0-201-1344-5.

[13]. Benjamin Goldberg, "*Functional Programming Languages*", ACM Computing Surveys, Vol 28, No. 1, March 1996.

[14]. Paul Hudak, John Hughes, Simon Peyton Jones and Philip Wadler, *"A History of Haskell: being lazy with class*", The third ACM SIGPLAN History of Programming Languages Conference (HOPL-III) San Diego, California, June 9-10, 2007.

[15]. Antony J.T. Davie, *"An Introduction to Functional Programming Systems Using Haskell"*, Cambridge University Press, ISBN: 05212772478, 1992.

[16]. Lorne T. Kirby, "*DNA Fingerprinting: An Introduction"*, Oxford University Press (USA), ISBN-13: 978-0195118674

[17]. O'REILLY Lamp: The open source web platform (a home of O'Reilly's online books and web resources), http://www.onlamp.com/pub/a/onlamp/2007/05/21/an-introduction-to-haskell---part-1-why-haskell.html?page=1 , last viewed on August 15, 2008

[18]. Eyal Amir, "*Approximation Algorithms for Treewidth*", University of Illinios, USA, 2002.