Haskell versus Rust: Benchmarking Efficient Concurrent Pathfinding in a Purely-Functional Programming Language

Prince Bernie B. Colis

Bachelor of Science in Computer Science

John Kenneth S. Lesaba

Bachelor of Science in Computer Science

Jon Ariel N. Maravilla

Bachelor of Science in Computer Science

Karl Frederick R. Roldan

Bachelor of Science in Computer Science

Senior project submitted to the faculty of the
Department of Computer Science
College of Computer Studies, Ateneo de Naga University
in partial fulfillment of the requirements for their respective
Bachelor of Science degrees

Project Advisor: Adrian Leo Pajarillo
First Panel Member
Second Panel Member
Third Panel Member

Month Day, 2020
Naga City, Philippines

Keywords: parallel programming, functional programming, graph theory

The Senior Project entitled

Haskell versus Rust: Benchmarking Efficient Concurrent Pathfinding in a Purely-Functional Programming Language

developed by

Prince Bernie B. Colis
Bachelor of Science in Computer Science

John Kenneth S. Lesaba
Bachelor of Science in Computer Science

Jon Ariel N. Maravilla
Bachelor of Science in Computer Science

Karl Frederick R. Roldan
Bachelor of Science in Computer Science

and submitted in partial fulfillment of the requirements of their respective Bachelor of Science degrees has been rigorously examined and recommended for approval and acceptance.

First Panel Member	**Second Panel Member**
Panel Member	Panel Member
Date signed: ______________________	Date signed: ______________________
Third Panel Member	**Adrian Leo Pajarillo**
Panel Member	Project Advisor
Date signed: ______________________	Date signed: ______________________

The Senior Project entitled

Haskell versus Rust: Benchmarking Efficient Concurrent Pathfinding in a Purely-Functional Programming Language

developed by

Prince Bernie B. Colis
Bachelor of Science in Computer Science

John Kenneth S. Lesaba
Bachelor of Science in Computer Science

Jon Ariel N. Maravilla
Bachelor of Science in Computer Science

Karl Frederick R. Roldan
Bachelor of Science in Computer Science

and submitted in partial fulfillment of the requirements of their respective Bachelor of Science degrees is hereby approved and accepted by the Department of Computer Science, College of Computer Studies, Ateneo de Naga University.

Marianne P. Ang, MS
Chair, Department of Computer Science
Date signed: ______________________

Joshua C. Martinez, MIT
Dean, College of Computer Studies
Date signed: ______________________

Declaration of Original Work

We declare that the Senior Project entitled

Haskell versus Rust: Benchmarking Efficient Concurrent Pathfinding in a Purely-Functional Programming Language

which we submitted to the faculty of the

Department of Computer Science, Ateneo de Naga University

is our own work. To the best of our knowledge, it does not contain materials published or written by another person, except where due citation and acknowledgement is made in our senior project documentation. The contributions of other people whom we worked with to complete this senior project are explicitly cited and acknowledged in our senior project documentation.

We also declare that the intellectual content of this senior project is the product of our own work. We conceptualized, designed, encoded, and debugged the source code of the core programs in our senior project. The source code of third party APIs and library functions used in my program are explicitly cited and acknowledged in our senior project documentation. Also duly acknowledged are the assistance of others in minor details of editing and reproduction of the documentation.

In our honor, we declare that we did not pass off as our own the work done by another person. We are the only persons who encoded the source code of our software. We understand that we may get a failing mark if the source code of our program is in fact the work of another person.

Prince Bernie B. Colis
3 - Bachelor of Science in Computer Science

John Kenneth S. Lesaba
3 - Bachelor of Science in Computer Science

Jon Ariel N. Maravilla
3 - Bachelor of Science in Computer Science

Karl Frederick R. Roldan
3 - Bachelor of Science in Computer Science

This declaration is witnessed by:

Adrian Leo Pajarillo
Project Advisor

Haskell versus Rust: Benchmarking Efficient Concurrent Pathfinding in a Purely-Functional Programming Language

by

Prince Bernie B. Colis, John Kenneth S. Lesaba, Jon Ariel N. Maravilla, and Karl Frederick R. Roldan

Project Advisor: Adrian Leo Pajarillo

Department of Computer Science

EXECUTIVE SUMMARY

To be filled in later

I dedicate this research work to all of humanity.

ACKNOWLEDGEMENTS

I thank everyone who helped me finish this thesis.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

The Shortest Common Superstring (SCS) problem, known to be NP-Complete, seeks the shortest string that contains all strings from a given set. In this paper, we provide the summary of the problem and some of its characteristics.

The SCS problem has been extensively studied for its applications in string compression and DNA sequence assembly [7].

The superstring problem has applications to data storage, specifically, data (string) compression [5]. In many programming languages, a character string may be represented by a pointer to that string. The problem for the compiler is to arrange strings so that they may be "overlapped" as much as possible.

DNA sequence assembly is another problem to which an SCS algorithm is known to apply. The *sequencing* problem in molecular biology is to "read" a string of DNA, which can be viewed as a string over the alphabet {A,C,G,T}. Sequencing produces such a large number of fragments that almost all genome positions are covered by many fragments. This short fragments thus have large overlaps between other pieces. Hence, they can be given as an input to SCS algorithm. Figure 1.1 shows an overlap graph consisting DNA reads (or fragments) as nodes.

In [7]'s paper, SCS was used to analyze DNA sequence assembly using a greedy algorithm.

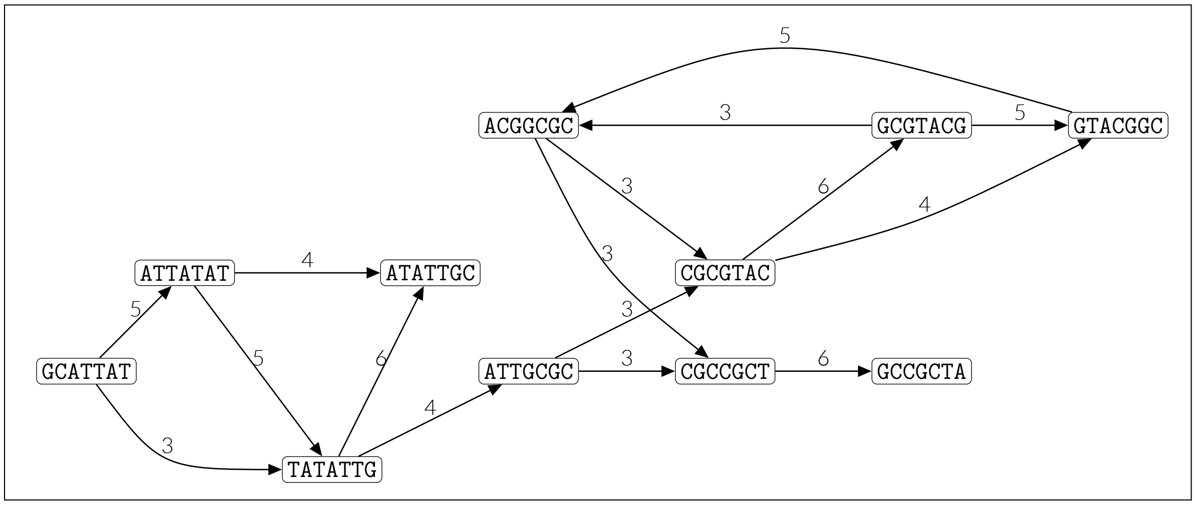

Figure 1.1: Sample overlap graph with each adjacent nodes having at least $k = 3$ overlaps. The original string is `GCATTATATATTGCGCGTACGGCGCCGCTACA`.

1.1 Project Context

1.2 Purpose and Description

1.3 Objectives

1.4 Scope and Limitations

Chapter 2

Review of Related Systems and Related Literature

Several researches have provided in depth study on approximation algorithms for the SCS problem.

Most [11, 7, 12], however, considered a greedy algorithm to solve the SCS problem. The reason why greed works for shortest common superstring problem was explored by [7]. They explained the good performance of the greedy algorithm by using the smoothed analysis. That is, for any given instance I of SCS, the average approximation ratio of the greedy algorithm on a small random perturbation of I is $1 + o(1)$.

Apart from the greedy approach, other methods including bio-inspired algorithms, have also been employed to solve lots of hard problems in computer science including the SCS problem. In [12]'s work, for example, four approaches for finding solutions to the SCS problem: a standard genetic algorithm, a novel cooperative-coevolutionary algorithm, a benchmark greedy algorithm, and a parallel coevolutionary-greedy approach have been compared. In the paper, the coevolutionary approach produced the best results.

Zaritsky et al's [12] work served as the foundation of this paper. Hence, the following algorithms, which have been explored and tested in the latter, have been extracted and validated whether they are indeed generating good approximate solution to the SCS problem.

2.0.1 Greedy-SCS

The core idea of the greedy algorithm is to repeatedly merge pairs of distinct strings with maximum overlap until only one remains. It has been conjectured by [8] that the superstring produced by the greedy algorithm is at most two times the optimal. Below is a pseudocode for the greedy algorithm:

Algorithm 1 GREEDY(R)

```
Input: $R$ - set of strings
Output: superstring of set $R$
1: while ——$R$—— ¿ 1 do
2:     choose $x_1 \neq x_2 \in R$ such that $overlap(x_1, x_2)$ is maximal
3:     $R \leftarrow (R - \{x_1, x_2\}) \cup \{merge(x_1, x_2)\}$
4: end while
5: return remaining string in $R$
```

The greedy algorithm is probably the most widely used heuristic used in DNA-sequencing due to its simplicity and constant performance guarantee which is most likely to be factor-2.

2.0.2 Genetic Algorithm

In this paper, we follow the standard model for the genetic algorithm (see Algorithm 2). That is, the algorithm generates an initial population of random candidate solutions, then the process of selection based on a defined fitness function, crossover, and mutation are used to evolve the next generation, of which each individual is then evaluated and assigned a fitness value. These steps are repeated a predefined number of times or until the solution is satisfactory.

Algorithm 2 GENERIC GA()

```
1:  $t \leftarrow 0$ {generation number}
2:  initialize Population$_t$
3:  evaluate(Population$_t$)
4:  while termination condition not met do
5:      select individuals from Population$_t$
6:      recombine individuals
7:      mutate individuals
8:      Population$_{t+1} \leftarrow$ newly created individuals
9:      $t \leftarrow t + 1$
10:     evaluate(Population$_t$)
11: end while
12: return solution derived from the best individual in Population$_t$
```

Chapter 3

Technical Background

A superstring is simply a string over some alphabet for which given a set of string from the same alphabet, the latter's members are all substrings of the former. To understand the SCS problem, it is best to assume some important concepts related to the problem itself. First, let us assume some set $R = \{x_1, \ldots, x_n\}$ as a set of strings (or blocks) over some alphabet Σ. Formally, we define a *superstring* of R as a string w containing each $x_i \in R$, as a substring.

Following are the elementary operations associated with the construction of a superstring. The $overlap(u, v)$ of two strings u and v is the maximum overlap between u and v. That is to say, the longest suffix of u (in terms of characters) that is a prefix of v. The $prefix(u, v)$ of u and v is the prefix of u obtained by removing its overlap with v. Lastly, $merge(u, v)$ is the concatenation of u and v with the overlap appearing only once.

As an example, say we have, $\Sigma = \{a, b, c\}$ and $R = \{cbcaca, cacac\}$. The string *cacacbcaca* is a superstring of R while the string *cbcacac* is the shortest common superstring. The following relations also hold:

$overlap(cbcaca, cacac) = caca,$

$overlap(cacac, cbcaca) = c\ ,$

$prefix(cbcaca, cacac) = cb,$

$merge(cbcaca, cacac) = cbcacac.$

A *superstring* w is also defined as the string $prefix(x_1, x_2) \cdot prefix(x_2, x_3) \cdots prefix(x_n, x_1) \cdot overlap(x_n, x_1)$. That is, a *superstring* is simply a concatenation of all the strings "minus" the overlapping duplicates. Apparently, each superstring of a set of strings defines a permutation of the set's elements. Conversely, every permutation of the set's elements corresponds to a single superstring

Our interest however lies on defining the SCS problem. Essentially, the SCS problem is: Given a set of strings R and a positive integer K, does R have a superstring of length K?

Figure 3.1 shows the SCS Decision Problem following the template provided by Garey and Johnson [6].

INSTANCE:
Finite alphabet Σ, finite set R of strings from Σ^* and a positive integer K.
QUESTION:
Is there a string $w \in \Sigma^*$ with $|w| \leq K$ such that each string $x \in R$ is a substring of w, i.e. $w = w_0 x w_1$ where $w_0, w_1 \in \Sigma^*$?

Figure 3.1: SCS Decision Problem

In the Compendium of NP Optimization problems published by [3] and in the list of NP-Complete Problems published by [6] the SCS problem appears under under Storage and Retrieval (SR).

Chapter 4

Methodology

Methodology stuff will appear here.

4.1 Systems Analysis

4.2 Systems Design

4.3 Requirements Specification

4.4 Development and Testing

Chapter 5

Results

The researcher conducted an experiment to validate the results in [12]'s work. Following is the description of the experiment done.

The input strings used in the experiment were generated in a manner similar to the one used in DNA sequencing. That is, a random string is generated, duplicated a predetermined number of times, and the copies are randomly divided into blocks of a given size. The set of all these blocks is the input to the SCS problem. The reasons for choosing this type of input has been extensively explained in [12]. Since our random string is a sequence of characters `A`,`C`,`G`, and `T`, we must transform this into an equivalent binary string. To do this, we simply assign each letter a unique two-bit representation, that is, `A = 00`, `C = 01`, `G = 11`, and `T = 10`.

The parameters used in the input generation is as follows:

Size of random string: 250 bases or characters

Minimal block size: 20 characters

Maximal block size: 30 characters

Number of duplicates created from the random string: 5

Let l denote the length of the derived string in the genetic algorithm case. Let m denote the number of blocks not covered by the derived string, and let b denote the maximal block size (30, in our case). The fitness value, f , of an individual is computed as follows:

$$f = \frac{1}{l + m * b}$$

This fitness function above drives evolution towards shorter superstrings covering as many blocks as

Number of Generations	Average Superstring Length (Rounded to Nearest Units)	Average Runtime (in Seconds)
100	760	34.57
250	596	74.20
500	526	139.70
1000	482	267.09
5000	421	1038.73
10000	413	1943.55
20000	409	3700.57

Table 5.1: Average superstring length and runtime of our Genetic Algorithm on 50 randomly generated problem instances. Each input set contains 50 blocks.

possible.

We performed 50 randomly generated problem instances. On each problem instance each type of genetic algorithm was executed twice and the better run of the two was used for statistical purpose. The results are summarized in Table 5.1 and is presented graphically in Figure 5.1.

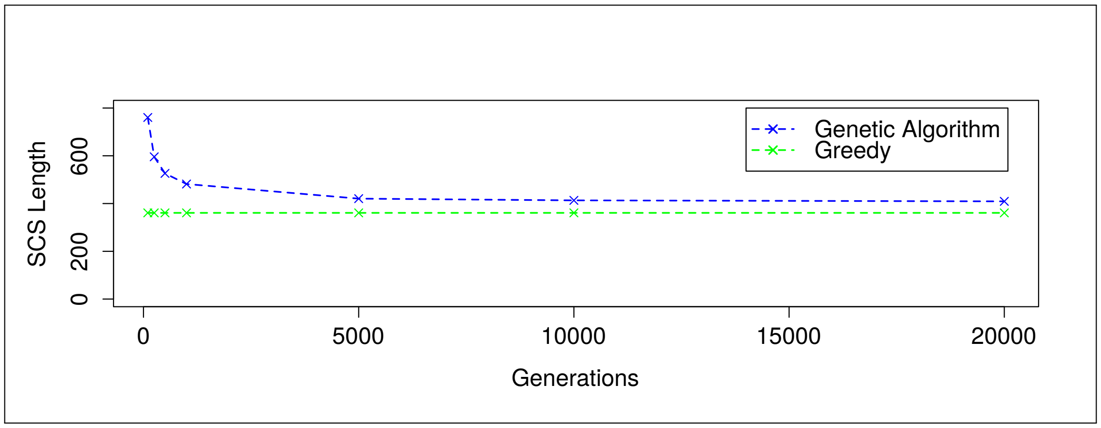

Figure 5.1: Best superstring as a function of generations. Each point in the figure represents the average of 50 runs on 50 different randomly generated problem instances. For each such instance, two runs were performed, the better of which was considered for statistical purposes

Note that our results seem to disagree with that of [12]. In contrast with the latter's findings, our tests shows that the greedy algorithm outperformed the genetic algorithm both in terms of finding superstrings with minimal length and the the time it takes to do so.

On the same set of test cases, the greedy algorithm produced superstrings having an average of 361 characters, in an average of runtime 130.64 milliseconds. This is far better than the 409 average superstring length that the genetic algorithm derived in 20000 generations in more or less an hour of operation. Note that the average length of the superstring derived by the greedy algorithm is less than the conjectured factor-2 performance guarantee. Recall that the length of the initial (reference) string is 250.

Still it is good to note that the genetic algorithm approach in solving the SCS problem converges. That is, we see a better solution yield as time or generations progress.

Chapter 6

Contributions and Recommendations

6.1 Summary of Contributions

6.2 Recommendations

There could be several reasons why our tests did not yield the "same" results as that of [12]'s. The author claims that this may be due to the myriad of parameters used in evolving individuals in each step of the genetic algorithm and the large number of possible candidates or values that we can set these parameters with. For example, we may claim that the appropriate choice of the recombination technique is necessary. In [12], a two-point crossover was employed. Thinking that this may be less of a factor, the author proceeded with just one-point cross in implementing the genetic algorithm for the experiment. Another could be in choice of the probabilities associated with recombination, elitism, and mutation rates. The author did only consider what was stated in the base paper without performing tests to validate whether this values are appropriate and could still complement the change made in some of the assumptions. As such, more empirical tests could be done to appropriately estimate these parameters.

If indeed DNA sequence assembly is the task at hand, lots of other techniques are now available. One may consider for example representing the reads or DNA fragments in a de Bruijn graph as opposed to the overlap graph presented in this paper. Assembling reads in a de Bruijn graph reduces the problem to a fragment assembly problem that can be formulated as the goal to find a trail or Eularian path that visits each edge (read or fragment) in the (de Bruijn) graph exactly once. Such, somehow makes the assembly process much “easier” since Eularian path construction is known to be solvable in deterministic polynomial time.

Appendix A

Code Listing

```
/**
 * File:   scs-greedy.cpp
 * Author: glenn
 *
 * Created on May 13, 2015, 10:52 AM
 */

#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
#include <list>

using namespace std;

/**
 * Greedy SCS
 */
string find_max_overlap(string source, string sink);
void showR(list<string> R){
    for(list<string>::iterator r = R.begin(); r != R.end(); r++ ){
        cout << *r << endl;
    }
}
void greedy_scs(list<string> &R);

// remove substrings
void reduce(list<string> &R){
    list <string> S;
    for (list<string>::iterator x = R.begin(); x != R.end(); x++) {
        bool is_substr = false;
        for (list<string>::iterator y = R.begin(); y != R.end(); y++) {
            if(*x == *y ) continue;
            if((*y).find(*x) != -1){
                cout << *x << " " << *y << endl;
                is_substr = true;
            }
        }
        if(!is_substr){
            S.push_back(*x);
        }
    }
    R.clear();
    R.insert(R.end(), S.begin(), S.end());
}

int main(int argc, char** argv) {

    if(argc != 2){
        cout << "Usage: " << argv[0] << " <input-file>" << endl;
        return -1;
    }

    int accessions = 50;
    {
    list<string> R;
    string reference;
    string fragment;

    string filename = argv[1];
    ifstream fin(filename.c_str());
    if(!fin){
        cout << "Could not open file \'" << filename << "\'" << endl;
        return -1;
    }

    fin >> reference;
    while(fin >> fragment){
        R.push_back(fragment);
    }

    //~ R.push_back("AACTG");
    //~ R.push_back("AACA");
    //~ R.push_back("ATC");
    //~ R.push_back("ATCCG");
    //~ R.push_back("AAT");
    //~ R.push_back("AATA");
    //~ R.push_back("AATC");
    //~ R.push_back("AGG");

    //~ reduce(R);

    int start_s=clock();
    // the code you wish to time goes here
    greedy_scs(R);
    int stop_s=clock();
    //~ cout << "Time: " << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << endl;

    //~ cout << reference << endl;
    //~ cout << reference.length() << endl;
    //~ cout << endl;

    cout << R.front().length() << ", "
        << (stop_s-start_s)/double(CLOCKS_PER_SEC)*1000 << ", "
        << R.front() << endl;
```

```
}

    return 0;
}

string find_max_overlap(string source, string sink) {

    int k = min(source.length(), sink.length()); // search windows
    string suffix = source.substr(source.length() -
        k, source.length()); // last k characters
    string prefix = sink.substr(0, k); // first k characters
    string matched = "";
    while (true) {
        if (suffix == prefix)
            return suffix;
        else {
            k--;
            suffix = source.substr(source.length() - k, source.length());
            prefix = sink.substr(0, k);
        }

    }
    return "";
}

void greedy_scs(list<string> &R) {
    // find the maximum overlap strings

    string overlap_xy, max_overlap_x, max_overlap_y;
    int max_overlap = 0;
    int iter = 0;
    while (R.size() > 1) {

        iter++;
        max_overlap = 0;
        overlap_xy = max_overlap_x = max_overlap_y = "";
        //~ cout << "finding overlap: R = "  << R.size()  << " t = " << iter << endl;
        //~ showR(R);


        for (list<string>::iterator x = R.begin(); x != R.end(); x++) {
            for (list<string>::iterator y = R.begin(); y != R.end(); y++) {

                if (x == y)
                    continue;

                overlap_xy = find_max_overlap(*x, *y);

                if (overlap_xy.length() >= max_overlap) {
                    max_overlap = overlap_xy.length();
                    max_overlap_x = *(x);
                    max_overlap_y = *(y);

                }
            }
        }

        // cout << "max overlap: " << max_overlap << ": "
        // << max_overlap_x << ", " << max_overlap_y << endl;

        // remove the two strings

        for (list<string>::iterator it = R.begin(); it != R.end(); it++) {
            if (*it == max_overlap_x) {
                it = R.erase(it);
            }
            if (*it == max_overlap_y) {
                it = R.erase(it);
            }
        }

        // merge
        string merged_xy;
        //~ cout << "maxoverlap : " << max_overlap << endl;

        //~ cout << endl;
        //if(max_overlap != 0)
            merged_xy = max_overlap_x.substr(0, max_overlap_x.length() - max_overlap) + m
        //~ else
            //~ merged_xy = max_overlap_x + max_overlap_y;

        if (merged_xy != "")
            R.push_back(merged_xy);


    }

}
```

REFERENCES

[1] C. Armen and C. Stein, *Improved length bounds for the shortest superstring problem*, in Algorithms and Data Structures, S. Akl, F. Dehne, J.-R. Sack, and N. Santoro, eds., vol. 955 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 1995, pp. 494–505.

[2] C. Armen and C. Stein, *A 223 superstring approximation algorithm*, Discrete Applied Mathematics, 88 (1998), pp. 29 – 57. Computational Molecular Biology {DAM} - {CMB} Series.

[3] G. Ausiello, P. Crescenzi, V. Kann, Marchetti-Sp, G. Gambosi, and A. M. Spaccamela, *Complexity and Approximation: Combinatorial Optimization Problems and Their Approximability Properties*, Springer, Jan. 2000.

[4] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, *Linear approximation of shortest superstrings*, J. ACM, 41 (1994), pp. 630–647.

[5] J. Gallant, D. Maier, and J. Storer, *On finding minimal length superstrings*, Journal of Computer and System Sciences, 20 (1980), pp. 50 – 58.

[6] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.

[7] B. Ma, *Why greed works for shortest common superstring problem*, Theoretical Computer Science, 410 (2009), pp. 5374 – 5381. Combinatorial Pattern Matching.

[8] D. Maier and J. Storer, *A note on the complexity of the superstring problem*, Tech. Rep. 223, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, New Jersey, 1977.

[9] M. Middendorf, *More on the complexity of common superstring and supersequence problems*, Theoretical Computer Science, 125 (1994), pp. 205 – 228.

[10] K. Plociennik, *A probabilistic {PTAS} for shortest common superstring*, Theoretical Computer Science, 522 (2014), pp. 44 – 53.

[11] J. S. Turner, *Approximation algorithms for the shortest common superstring problem*, Information and Computation, 83 (1989), pp. 1 – 20.

[12] A. Zaritsky and M. Sipper, *Coevolving solutions to the shortest common superstring problem*, Biosystems, 76 (2004), pp. 209 – 216. Papers presented at the Fifth International Workshop on Information Processing in Cells and Tissues.

VITA

JR is BS Information Technology student of the Department of Computer Science at the Ateneo de Naga University.