CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING

DEPARTMENT OF CYBERNETICS

BACHELOR PROJECT

Evolutionary Algorithm for the Longest Common Subsequence Problem

Author: Tereza Pytelová

2012, Prague

Supervisor: Ing. Jiří Kubalík, Ph.D

Prohlášení

Prohlašuji, že jsem předloženou práci vypracovala samostatně a že jsem uvedla veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne 29.5.2012 .....

.....

Podpis autora práce

Poděkování

Děkuji především mému vedoucímu, panu Ing. Jiřímu Kubalíkovi, Ph.D, za vydatnou pomoc, trpělivost a pevné nervy.

Abstrakt

Problém nejdelší společné podsekvence je zajímavou úlohou v počítačové vědě. Na tento problém lze převést mnoho reálných úloh, jakou je například komprese dat nebo hledání vztahů mezi biologickými sekvencemi. Pro dvě vstupní sekvence je problém nejdelší společné podsekvence optimálně řešitelný pomocí dynamického programování, nicméně pro více vstupních sekvencí je problém NP-úplný a jeho řešení pomocí dynamického programování je příliš časově i paměťově náročné. Proto jsou v tomto případě využívány algoritmy heuristické jako je paprskové prohledávání stavového prostoru, optimalizace pomocí mravenčí kolonie či simulované žíhání. Tyto algoritmy nezaručují nalezení optimálního řešení, jsou však mnohem méně náročné co se týče času a paměťového prostoru. Genetické algoritmy jsou vhodné pro optimalizační problémy s velkým stavovým prostorem, jakým je právě problém hledání nejdelší společné podsekvence. V této práci je aplikován iterativní optimalizační algoritmus využívající genetický algorimus pro hledání zlepšujících kroků (POEMS - *Prototype Optimization with Evolved Improvement Steps*), který je schopen prohledat větší okolí ve stavovém prostoru než klasické konstruktivní a lokálně optimalizační algoritmy a má proto dobré předpoklady pro dosažení kvalitních výsledků. Nicméně ani jeden z navržených algoritmů neuspěl ve srovnání se současnými špičkovými algoritmy. Možné důvody neúspěchu jsou diskutovány v závěru práce.

Abstract

The Longest Common Subsequence Problem is an interesting task in computer science which has many applications namely in data compression or in molecular biology. Used to find similarities through input sequences it can be solved to optimality by means of dynamic programming for two input sequences. However for number of input sequences greater than two it becomes NP hard and dynamic programming is not very efficient as it requires too much time and space. In order to find an approximate solution in reasonable time many algorithms were proposed such as beam search, ant colony optimalization or simmulated annealing. Genetic algorithms represent another metaheuristic suitable for solving discrete optimalizing problems with large search space. Aim of this work is to take an advantages of an iterative optimization algorithm called the Prototype Optimization with Evolved Improvement Steps (POEMS) applying it to the LCSP. It has been shown that POEMS as an iterative algorithm based on genetic algorithm can obtain very good results that is attributed to its ability to search larger neighbourhood than traditional iterative local search algorithm. Nevertheless performance of both proposed algorithms was not as good as was expected. Possible reasons for such underachievement are discussed at the end of this work.

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Tereza P y t e l o v á

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Evoluční algoritmus pro řešení problému nejdelší společné podsekvence

Pokyny pro vypracování:

1. Seznamte se s problémem hledání nejdelší společné podsekvence (LCS - Longest Common Subsequence).
2. Proveďte rešerši existujících přístupů pro řešení problému LCS. Zaměřte se na přístupy využívající metaheuristiky jako jsou evoluční algoritmus, paprskové prohledávání apod.
3. Navrhněte a implementujte postup založený na evolučním algoritmu.
4. Experimentálně ověřte funkčnost a úspěšnost navrženého algoritmu.
5. Dosažené výsledky vyhodnoťte a porovnejte s ostatními známými přístupy.

Seznam odborné literatury: Dodá vedoucí práce.

Vedoucí bakalářské práce: Ing. Jiří Kubalík, Ph.D.

Platnost zadání: do konce zimního semestru 2012/2013

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

L.S.

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 9. 12. 2011

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Tereza P y t e l o v á

Study programme: Cybernetics a Robotics

Specialisation: Robotics

Title of Bachelor Project: Evolutionary Algorithm for the Longest Common Subsequence Problem

Guidelines:

1. Get acquainted with the longest common subsequence problem (LCS).
2. Review existing approaches to the LCS problem focusing on those based on metaheuristics such as evolutionary algorithms, beam search etc.
3. Propose and implement an evolutionary-based algorithm for solving the LCS problem.
4. Carry out experiments to assess a performance of the proposed algorithm.
5. Evaluate the achieved results and compare your algorithm with other existing approaches.

Bibliography/Sources: Will be provided by the supervisor.

Bachelor Project Supervisor: Ing. Jiří Kubalík, Ph.D.

Valid until: the end of the winter semester of academic year 2012/2013

L.S.

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, December 9, 2011

Table of Contents

List of Figures

List of Tables

1 Introduction

The aim of this work is to apply a genetic algorithm to the Longest Common Subsequence Problem (LCSP) trying to take the advantages of Prototype Optimization with Evolved Improvement Steps (POEMS) algorithm.

The LCSP as a classic task in computer science has wide range of applications including operating with databases, data compression or molecular biology. In all of these areas it seeks after relationship among input sequences which represent the specific problem. The LCSP can be optimally solved by dynamic programming for two input sequences while it requires $O(l_1.l_2)$ time and space, where l_1 and l_2 are the lengths of input sequences. However for arbitrary number of input sequences it becomes NP hard hence various heuristic algorithms were proposed. The classical example is beam search [5], simmulated annealing [2] or ant colony optimalization [9].

Genetic algorithms have already been used to solve this problem [8], but in their primary form they are not very efficient. Nowadays GA are usually employed in certain heuristic algorithms such as HGACO mentioned in [9]. Genetic algorithms are suitable for solving discrete optimalizing problems with large search space and for the problems, in which is not easy to define the way to the best solution. While using GA it is not necessary to understand the problem to depth, it suffices only to define characteristics of the best solution. Specially in the case of the LCSP the requirement is, that the solution has to be subsequence of all input sequences and it should be as long as possible. Another advantage of GA is their ability to change the temporary solution during entire algorithm at contrary to such algorithms as beam search which can modify only the last character. GA as well as other heuristic algorithms do not solve the problem optimally, but in such task as the LCSP where the problem specific algorithm requires excessively time and space, the non optimal solution found in reasonable time is welcome.

Prototype Optimization with Evolved Improvement Steps (POEMS) is an iterative algorithm based on genetic algorithm. Contrary to the traditional genetic algorithms, it does not evolve a population of candidate solutions to the given problem (i.e. the population of candidate common subsequences). Instead, it works with a population of so called action sequences that are used to modify a working solution. The best evolved action sequence that improves the working solution the most is applied to the working solution yiealding a new one for the next iteration. Thanks to this representation it is able to search larger

neighborhood of the current working solution than traditional local search algorithms, which use neighbourhood defined by a single action or variation operator. This is especially useful in the LCSP since the search space is very large and for only one operator it would be quite difficult to search all possible states in reasonable time. POEMS has already reached very good results applied to the Shortest Common Supersequence Problem (SCSP) [10], [11]. Since these two problems, the LCSP and the SCSP, are closely related this work tries to achieve similar competitive results using POEMS on the LCSP. As the fitness function included in POEMS, a combination of existing types of previously used fitness functions is used.

To be able to compare a new algorithm with current state-of-the-art approaches the same datasets as in [4] was choosen. The first dataset is randomly generated and contains number of various alphabets. Second dataset consists of three sets of protein and DNA sequences. The first is randomly generated as well and other two sets contain real sequences of rat and virus.

The results achieved with POEMS are not as good as expected. Its performance is worse than the compared improved beam search algorithm (IBS-LCS) presented in [4], which is the current state-of-the-art algorithm for the LCSP. On the other hand, as the size of the problem instance grows in terms of the number of input sequences its performance gets close to the IBS-LCS with respect to length of the resulting subsequence. However, the efficiency of the POEMS in terms of the computational time is much worse than that of the IBS-LCS algorithm. This can be attributed to the repeated computation of the population-based evolutionary algorithm.

This work is organised as follows: In section 2 there is a definition of the LCSP and short overview of existing approaches with focus on their fitness functions. POEMS algorithm is described in section 3 and its implementation as well as implementation for the LCSP can be found in sections 4 and 5. Section 6 contains dataset specification, algorithm setup and the results. Discussion on the problem is given in section 7, conclusion and the future work are placed at the end of this work as sections 8 and finally references are situated in section 9.

2 LCSP

The Longest Common Subsequence Problem belongs to a number of well-known problems in computer science. It has wide range of applications in the areas of data compression, query optimization in databases, clustering Web

users or in pattern recognition. The LCSP is also important for molecular biology to compare DNA, RNA or amino acid sequences as through the similarity of sequences functional or structural relationship among biomolecular sequences can be found.

2.1 Definition

Goal of the LCSP is to find the longest common subsequence of the input sequences. The LCSP is defined over some finite alphabet Σ. In this task sequences of characters from Σ of finite lengths are denoted as input strings or input sequences $a_1, a_2, ..., a_k$. A subsequence of a string a is defined as a sequence obtained by deleting zero or more characters from a. Given two or more input strings $a_1, a_2, ..., a_k$ the task of the LCSP is to find the longest sequence of characters that can be found as subsequence in all input strings. For example having the input sequences ABCD, BDCA and BCDD the common subsequences are B, C, D, BC, BD and naturally the null string. Sequences BC and BD are the longest common subsequences (LCS) of this set of sequences.

Algorithms used to solve the LCSP usually work with problem defined in binary alphabet, so it is necessary to define some function f: Σ to {0, 1}. A simple binary encoding that has been proposed and used by Julstrom and Hinkemeyer [3] is defined as follows:

Let a_0 be the shortest input sequence denoted as *reference*. (In case of input sequences of the same length a_0 is considered as first of them.) Then s is sequence of {0, 1} henceforth called *mask* determining which characters from the a_0 occure in the corresponding subsequence. For example having a_0 = ABCD and s = 1001, it represents subsequence AD. Such subsequence is denoted as $c(s)$, the candidate solution.

This candidate solution is allways a subsequence of a_0, but it is not necessary a subsequence of all input strings, therefore using this representation infeasible solutions can appear.

To clarify various algorithms and its fitness functions more definitions have to be formulated. At first the length of sequence a is denote as $|a|$. The number of input sequences is marked as k and the length of the shortest one of them (or the first one of all respectively) as n. In addition let *prefix* be the shortest segment considered from left to right for which particular candidate solution is subsequence. Analogously the *suffix* is the shortest part of input sequence considered from right to left that allows c(s) to be its subsequence. At the end it is important to specify the number of input seqences for which candidate solution is

a subsequence. This number is denoted as $m(s)$. In the case of feasible solution $m(s) = n$ holds.

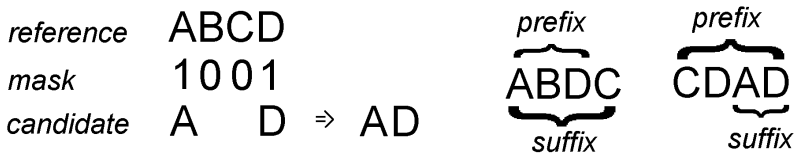

Figure 1: The mask representation

2.2 Existing Algorithms

For more than two input sequences the Longest Common Subsequence Problem is NP-hard. The problem is polynomially solvable by dynamic programming and it requires $O(l^k)$ of time and space, where l is the length of the longest input sequence and k is the number of sequences. Such algorithm is not very useful for large number of input sequences. In the case of biological sequence analysis or searching in text the length of input sequences is usually quite large. That is why approximation algorithms such as long run (LR) [13] algorithm, expansion algorithm (ExpA) [14] and best next for maximal available symbols (BNMAS) [15] have been found. These algorithms guarantee performance ratio (i.e. the ratio of the length of the optimal solution over that of the approximation) of $|\Sigma|$ that is the size of alphabet Σ of the input sequences. For DNA sequences $|\Sigma| = 4$ and for protein $|\Sigma| = 20$ holds, thus such results are not very satisfying. In order to find better approximate solutions to the Longest Common Subsequence Problem number of different heuristic algorithms have been proposed with quite good results. In addition to the current state-of-the-art Improved Algorithm for the LCSP [4] there are also algorithms based on simmulated annealing [2] or the biologically inspired ant collony optimallization (ACO) [9] which simmulates ants searching for food. Even HGACO (Hybrid GA and ACO) [16] algorithm was proposed combining two evolutionary methods: genetic algorithm and ACO. Despite the huge variety of different approaches this section is concentrated only on a few recently state-of-the-art algorithms.

2.2.1 An Enhanced ACO Algorithm with Pair Matching Strategy

This algorithm [12] combines previously proposed ACO algorithm which is inspired by ant collony foraging the food with matching pair algorithm (MPA). It takes up previously succesful algorithm HGACO which uses repeatedly alternating ACO algorithm and GA. Because GA requires lot of time, in EACO it is replaced by the matching pair algorithm (MPA) which tries to connect the best

prefixes of candidate solutions and the best *suffixes* in the order to create better solution.

The ACO uses a positive feedback realized by marks called pheromones. Ant which is going back to the nest with food leaves pheromone marks on its path. Ant searching for food will more probably use the path with larger amount of pheromones. Marks vaporate in time, thus the longer path is less used than the shorter one. Finally this algorithm discovers the shortest path from the nest to the food. MPA tries to combine the longest *prefixes* with the longest *suffixes* of input set of candidate solutions.

The heuristic algorithm is also used here but only for creating the first set of candidate solutions. Then this set of solutions is given to ACO and the result is considered as an input of MPA which tries to create longer solutions from given subsequences. Consequently ACO and MPA periodically alternates until the specified number of iterations is reached.

2.2.2 Beam search

Beam search is a constructive algorithm. It begins with an empty solution which is iteratively extended by one character from given alphabet using all possible ways. This algorithm is based on classical tree search but it allows only limited number of branches at each step. The limiting number is the so called beam width. In each iteration all possible extensions of current set of solutions which can be reached by adding exactly one character from given alphabet are created. Consequently best *beam-width* solutions are choosen from these feasible solutions and this set is used to create next generation of candidate solutions by adding one character. In the case of beam width equal to 0 it turns into greedy algorithm, if the beam width is large enough to allow all possible extensions the algorithm behaves as the basic breadth-first search. Otherwise solutions are compared together and eventually deleted with regard to their fitness function values.

Algorithm in [5] bounds not only the beam width, but also the number of maximal possible extensions controlled by input parameter of this algorithm. The value of this parameter should be between 1 and the $|\Sigma|$. If the number of possible extensions is greater than allowed only the best characters are added according to the so-called Best-next heuristic. Note, that in this algorithm solutions with diffferent lengths are conceivable.

2.2.3 An improved Beam Search algorithm

An improved BS algorithm for the LCSP (ISB-LCS) [4] is based on the same principe of BS as algorithm described above. However there are some differences, which make this algorithm beeing the best of current state-of-the-art heuristic algorithms to the LCSP. The first difference is in calculating the fitness values of candidate solutions. The fitness here is based on probability that the random string will be common subsequence of the rests of input sequences after deleting their *prefixes* related to current candidate. Another difference is that improved algorithm does not compare new candidate solution with each other but only with specified number of best of current candidate solutions. Time saved by this approach is used for enlarge the beam width. Fitness function for IBS-LCS is described in more details in section 2.3.3.

2.3 Fitness Functions for GA

In heuristic algorithms used to solve the LCSP like simmulated annealing, beam search or genetic algorithm there exist more candidate solutions. The algorithm has to decide which solution is better and which is worse and should be deleted. For this decision the fitness functions are used. The value of fitness function shows the quality of candidate solution, how "fit" it is. Hence higher fitness value means better solution. The quality of finite solution strongly depends on how much the value of fitness function corresponds with requested characteristics of an appropriate candidate solution *s*. For the right function of genetic algorithm the pertinent fitness function is essential.

Each of further named fitness functions has its own advantages and disadvantages and is tailored to certain type of algorithm. It does not mean, that it is not possible to use it in different approach, but we can use it only if we entirely understand the principe of this function. We also have to remember the conditions under which is certain type of function defined.

An overview of the most commonly used fitness functions is given in this section as they were taken as an inspiration for the evaluating function used in the POEMS algorithm.

2.3.1 JH Function

First there is a simple function which is used in evolutionary algorithm defined by Julstrom and Hinkemeyer [8] as follows:

Definition 1: $f_{JH}(s)$

$f_{JH}(s) = 3000\ (|c(s)| + 30m(s) + 50)$, if $|c(s)| = n$ and $m(s) = k$,

$= 3000\ (|c(s)| + 30m(s))$, if $|c(s)| < n$ and $m(s) = k$,

$= -1000\ (|c(s)| + 30m(s) + 50)(k - m(s))$, if $|c(s)| = n$ and $m(s) < k$,

$= -1000\ (|c(s)| + 30m(s))(k - m(s))$, if $|c(s)| < n$ and $m(s) < k$.

This function generally allows infeasible solutions (when $m(s) < k$) whose fitness is set to negative values. Algorithms which use this function have to calculate with the fact that such sequence can not be considered as finite solution. In special occurence of $|c(s)| = n$ (candidate solution has the maximal possible length) this function is increased by 50. The reason for such advantage (or disadvantage in case of infeasible solution) is not obvious, solution with the maximal possible length will be better (worse) than the shorter one even without this benefit. It is another disadvantage of this fitness function that it is not clear why are its four constants set to just these values as they are and what will happen, when we change them. It would be more understandable if these constants depend on the length of input sequences or on their number.

2.3.2 fMAX

For clarify function fMAX(s) [2] it is necessary to define $c(s)_{(d)}$ as the longest *prefix* of candidate solution, which is subsequence of all input sequences. Its length is denoted here as d. Function fMAX(s) is counted as the length of *prefix* $c(s)_{(d)}$ decreased by the length of the rest of this candidate solution. Thus the longer the *prefix* is the higher is the value of fitness function. The maximal length of *prefix* denoted as function MAX($c(s), a_1, a_2, ..., a_k$) and the fitness function fMAX(s) are defined in following way:

Definition 2: MAX($c(s), a_1, a_2, ..., a_k$)

MAX($c(s), a_1, a_2, ..., a_k$)

$= \min \{\max \{d \mid c(s)_{(d)}$ is a subsequence of $a_i\} \mid i \in \{1, ..., k\}\}$.

Definition 3: fMAX(s)

fMAX(s) = MAX ($c(s), a_1, a_2, ..., a_k$) – ($|c(s)|$ – MAX ($c(s), a_1, a_2, ..., a_k$)).

After simplification we obtain following formula:

fMAX(s) = 2MAX ($c(s), a_1, a_2, ..., a_k$) - $|c(s)|$.

Infeasible solutions are also allowed here, but they do not necessarily take the negative values. Important property of this function is that deleting a character which can not be mapped to the feasible solution increases the fitness value.

2.3.3 Probabilistic Heuristic

The probabilistic heuristic defined in [4] is based on the probability, that random string y of length z is subsequence of the rest $r_i(c(s))$ of each input sequence. Set of these rests is denoted as $R(c(s))$ and it represents input sequences deprived of characters from candidate solution.

The heuristic function $h_z(c(s))$ is defined as follows:

Definition 4: $h_z(c(s))$

$h_z(c(s))$ = Pr (y is subsequence of $R(c(s))$)

Under assumption of independance the input strings it can be defined:

Pr (y is subsequence of $R(c(s))$) = Π Pr (y is subsequence of r_i ($c(s)$)

for $i \in \{1, ..., k\}$.

Function Pr (y is subsequence of r_i ($c(s)$)) for $i \in \{1, ... ,k\}$ can be determined via dynamic programming which works with probability of that the first characters are the same or not. It should not be forgotten, that $h_z(c(s))$ depends not only on $c(s)$ but also on z. It means, that z has to be the same for all $c(s)$. How to find the best value for z is still not clear.

2.3.4 The Upper Bound Function (UB)

The UB function used in [5] counts the maximal possible extension of current $c(s)$. It is important to remember, that we have to count only with feasible solutions. The UB function is defined in the following way:

Definition 5: UB(s)

$UB(s) = |c(s)| + \Sigma \min \{|a_i^B|c \mid i = 1, ..., n\}$ for all c from Σ.

Here a_i^B denotes the maximal rest of input sequence a_i obtained from this sequence by deleting its shortest *prefix*. $|a|_c$ stands for the maximum of the occurences of character c in sequence a. For better insight here is an example:

Lets have input sequences from the previous instance: ABCD, BDCA and BCDD. Remind that null, B, C, D, BC and BD are all possible common subsequences of this set. Consider the subsequence $c(s)$ = B. The maximal rest a_1^B of the first sequence after deleting the shortest *suffix* corresponding with the $c(s)$ is CD, the same rest of the second sequence is DCA and of the last it is CDD. Now go through the alphabet trying to find minimum occurences of choosen character in all rests. For A and B character there is no common occurence. The C appears in all rests at least once hence its contribution to sum in fitness function is

equal to 1. The same holds for D therefore the sum in fitness function for this solution will be equal to 2.

Note that 2 is number of characters which are ideally possible to add. This number is usually not reachable in real. In this case if C will be added, D wont and vice versa. Thus this fitness shows the maximal estimation for extension of current subsequence which probably wont be reached.

This function is used in beam search algorithm and can not be used in genetic algorithms without changes because it prefers null solutions. It is caused by the fact, that empty solutions have the maximal possible extension which is not really obtainable.

3 POEMS

Prototype Optimalization with Improvement Steps as is described in [7] is an algorithm which iteratively improves a temporary solution called "prototype". Modifications on prototype are made by sequences of primitive problemspecific actions which enable POEMS to search larger search space. The sequence of actions is for further description denoted as AS.

Algorithm 1: Prototype Optimization with Evolved Improvement Steps

```
1    Prototype = GeneratePrototype()
2    i = 1
3    while (i <= nIterations) do
4         BestSequence = RunEA(Prototype)
5         Candidate = ApplyTo(BestSequence, Prototype)
6         if IsBetterThan(Candidate, Prototype) then
7              Prototype = Candidate
8         end if
9         i = i + 1
10   end while
11   return Prototype
```

The algorithm works as follows. At first the start prototype is initialized. Then at the beginning of each iteration POEMS randomly generates a set of AS named the AS population and then runs *nGenerations* evolutionary cycles. Each cycle tries to find the best AS with respect to the current prototype. AS are compared using fitness function of the solution they generate from the prototype considering

how much they improve it. In evolutionary cycle there are *nTournament* selections in which the best "AS-parents" for "children" are choosen. During the crossover from two parents two children are created having the properties inherited from both parents. Children are then put through mutation which can influent ther characteristics but only a little. If they are not the worst (according to their fitness), children are added to the AS population to the detriment of worse ones. In the population remain such ASs that cause the biggest improvement of the prototype. At the end of the iteration the best AS from the AS population is checked if it do not worsen the prototype. If do, the prototype stay unchanged. If the modification makes the prototype better or the same quality as it was, the AS is applied on prototype and the result is considered as the input for the new iteration.

Algorithm 2: Genetic Algorithm for POEMS

```
1     ASPopulation = GenerateASPopulation()
2     i = 1
3     j = 1
4     while (i <= nGenerations) do
5           while (j <= nTournament) do
6                 Parents = BestOf(ASPopulation)
7                 j = j + 1
8           end while
9           Children = Crossover(Parents)
10          Children = Mutation(Children)
11          if NotTheWorst(ASPopulation, Children) then
12                add(ASPopulation, Children)
13          end if
14          i = i + 1
15    end while
16    return BestOf(ASPopulation)
```

Population of action sequences is initialized at the begining of each iteration. ASs are randomly filled by actions until fixed length called *maxGenes*. Types of primitive actions are defined in problem specific part of algorithm but probability of selection of each type is the same. In addition each action can be "switched on" or "swithched off". It means that action can be marked as "NOP"

action and such action is inactive. As the consequence the real number of active actions in AS is always less or equal to its length. However NOP action still remembers they own type and can be set as active again by mutation.

Other problem occurs, if there is a local optimum. It is probable, that AS with greater number of active actions will be considered as worse and they will disappear from the population. AS with only a few active actions wont be able to get the prototype out of the local optimum that would cause a stagnation of the search process. To avoid this state the drawers were created as they are described in following paragraph.

Population of AS is devided into drawers indexed from 0 to *maxGenes*. In each drawer there are *sizeDrawer* AS. The rule for randomly generated AS is following: in each drawer can be only AS with the number of active actions the same or greater than the index of the drawer. The reason is that in the last drawer will be AS with at least *maxGenes* - 1 active actions and the diversity in number of active actions is preserved. In addition to avoid deadlocks the AS which do not modify the prototype at all is considered as the worst and its fitness is set to the least possible value.

This algorithm works with only one temporary solution called prototype. This is the difference between POEMS and beam search which remembers up to *beam-widtht* candidate solutions throw entire algorithm. Although in comparison with beam search POEMS do not have the advantage of remembering more than one candidate solution, it is capable of modifying not only the last character, but can affect all of the characters in temporary subsequence. It is limited only by number of changes that can be realized throw one AS. This number is called AS size and it can reach up to the value of *maxGenes*. This constraint is responsible for a little disadvantage of this algorithm. If it is necessary to change more bits in *mask* than *maxGenes* to achieve better solution from local maximum, improvement is practicaly unreachable. So in such cases the diversity of the possible candidate solution is influented negatively.

4 POEMS for LCSP

In this section two variants of the POEMS algorithm for solving the LCSP are described. In Figure 2 there is a flowchart of both proposed algorithm which can help to better understand the principe of optimalizing process. Each algorithm also uses its own fitness function which is defined in section 4.3.

4.1 Representation

As in most of recent approaches an indirect "mask" representation has been used. It means that the algorithm takes one input sequence as a *reference* one, which is either the shortest input sequence of the set of input sequences or the first one if all input sequences are of the same length. An array of 0s and 1s of the same length as the reference sequence is then used as a *mask* whose values indicate, which characters and in which order will constitute the candidate subsequence. In particular, a value 1 at j-th position in the *mask* means that the j-th character of the reference sequence appears in the candidate subsequence while 0 means that the j-th character of the reference sequence does not. Thus, the space of all possible subsequences is determined by the reference sequence since only the characters present in the reference sequence can appear in the generated subsequence, preserving their relative order in the reference sequence.

Such easy representation we can afford since the solution has to be a subsequence of all input sequences including the *reference*, hence character which is not in the reference sequence has no chance to be in the subsequence of all input sequences.

An alternative representation (which is not used here) would be to determine the solution by the list of characters which appears in the subsequence. In comparison with such approach using mask representation may cause that changes in subsequence take place slower and it can take more evolutionary cycles to find the way to better solution. Differences between mask and list representation is discussed in section 7.

4.2 Actions

Changes in subsequence are realized by basic "Actions". Because of mask representation the "SwitchOnOff" action would be enough to obtain all possible subsequences. This action swaps bits in *mask* from 0 to 1 and vice versa thus only one parametr is necessary. It is an integer number from 0 to length of the reference subsequence decreased by one which is denoted as *j*. This parametr define on which position the change will be applied. The result after action application depends on the initial state of the *mask*. If *mask*[*j*] = 0 action "SwitchOnOff" with parameter *j* swaps it to 1. It means that the character on *j*-th position in the *reference* will be added on the corresponding place in the subsequence. If *mask*[*j*] is equal to 1, after action application it will change to 0 thus the corresponding character will be deleted.

However there is a problem especially on the beginning of the algorithm. When the *mask* contains only 0s more "switchOn" actions are required. Using only one type of action the on effect depends on action parameter (eventually on the *mask*). Better performance can be achieved if actions will be divided into two types as mentioned above: "switchOn" and "switchOff" action. These actions have still one parameter determining on which position the action will be applied but this parameter can be derived only from corresponding positions in the *mask*. "SwitchOn" action chooses the parametr only from the off-bites, "switchOff" action only from on-bites. This separation will cause that at the begninning of algorithm more "switchOn" actions will stay in AS population to fill defaultly empty subsequence faster than using only one type of action.

4.3 Fitness functions

Two different fitness functions are described in this section. Each of them is used in one of two here proposed algorithms. Fitness functions are different, but they both check if the solution is feasible or not. The demand on feasibility is much stronger than other requirements therefore feasible solutions will be always better than infeasible ones. This is what causes the invariance of the length of the solution at the end of the S-algorithm. Relevance of the requirements in fitness functions is given by multipliers. More important requirement has to be multiplied by number large enough so that it is not impacted by any value of the less important one. This number is usually equal to the maximal possible value of the next less important requirement.

4.3.1 nF Fitness Function

The fitness function for the S-algorithm is quite simple. It do not need the mechanism for elongation the solution as it is done by algorithm itself. Thus it is enough to demand the current solution to have the required length. Solutions with different length are disadvantaged accordingly to the difference. NF fitness function is then defined as follows:

Definition 6: nF(*s*)

$nF(s) = l + (l - |ds|)$, if(reallyFeasible),

$= (l - |ds|)$, otherwise,

where l is maximal length of sequences and $|ds|$ is the difference between actual length of the current solution and required length. It is clear that to the $(l - |ds|)$ is in case of feasible solution added 1 multiplied by the maximal possible value of that bracket (it means by the l).

4.3.2 RRm Fitness Function

RRm fitness function is used by W-algorithm. It consist of three parts. The most important requirement is (as well as in the previous fitness) that the solution have to be feasible. Then the maximal possible length is demanded because elongation of the prototype have to be caused by this fitness function due to initialization to empty string at the beginning of each algorithm. The least resolver is the sum of *prefixes* (denoted here as sP) plus length of the longest *prefix* (maxP) which is multiplied by number of input sequences to have the same weight as the mentioned sum. By both these numbers the number $2kl$ is decreased so neither the third part of this fitness function is never less than 0. Accordingly to the beginning of this section the maximal value of this part ($2kl$) has to multiplie the length of current solution. The third and most important requirement has to be multiplied by $2kl^2$ to not to be impacted by other parts of this fitness. RRm fitness function and further needed definitions follows.

Definition 7: RRm(*s*)

$\mathrm{RRm}(s) = 2kl^2 + |c(s)|\ 2kl + (2kl - (\mathrm{sP} + \mathrm{maxP}k))$, if(reallyFeasible),

$= |c(s)|\ 2kl + (2kl - (\mathrm{sP} + \mathrm{maxP}k))$, otherwise.

Definition 8: maxP($c(s)$, a_1, a_2, ..., a_k)

$\mathrm{maxP}(c(s), a_1, a_2, ..., a_k) = \max \{p_i \mid i \in \{1, ..., k\}\}$.

Definition 9: sP($c(s)$, a_1, a_2, ..., a_k)

$\mathrm{sP}(c(s), a_1, a_2, ..., a_k) = \Sigma \{p_i \mid i \in \{1, ..., k\}\}$.

Remind that the p_i denotes the shortest *prefix* of a_i for which $c(s)$ is a subsequence.

4.4 Algorithms for LCSP

The accent was put on the length of resulting subsequence and not on time so the first proposed algorithm called **"S-algorithm"** is quite time-consuming. It simulates the behavior of beam search algorithm when forces candidate subsequences to have exactly the given length. At the beginning the length is equal to 0 - it means that the algorithm starts with null string. Then the length is increased by one in each iteration and through POEMS the prototype is improved. For better understanding this algorithm see Figure 2. The aim of this method is to take an advantage of iterative extension of candidate solution. Algorithm stops when the length of candidate solution is not modified during 20 iterations despite of enforced elongation. This state is caused by here used fitness function which prefers feasible solutions. Infeasible solutions are allowed, but the feasible ones

are advantaged by much greater fitness. Therefore it is practically impossible to replace them to infeasible ones. By reason of initialization to the empty sequence (which is surely the common subsequence) algorithm never ends with infeasible solution.

The second proposed algorithm combines POEMS with the fitness function based on minimalization of *prefixes* of input sequences as they are defined in section 2.2. Fitness function for this algorithm named **"W-algorithm"** is described more precisely in following paragraphs. Infeasible solutions are allowed here as well as in the previous algorithm and they are disadvantaged in the same way. There is one parameter called *window* which is used as "window to input sequences". W-algorithm also works iteratively but in this case it gradually enlarge the length of input sequences. It means that in the first iteration only first *window* characters from input sequences are considered to create a subsequence by POEMS. In next iteration the length is increased by *window* again. The algorithm stops when the end of input sequences is reached. The flowchart of this algorithm is showed in Figure 2.

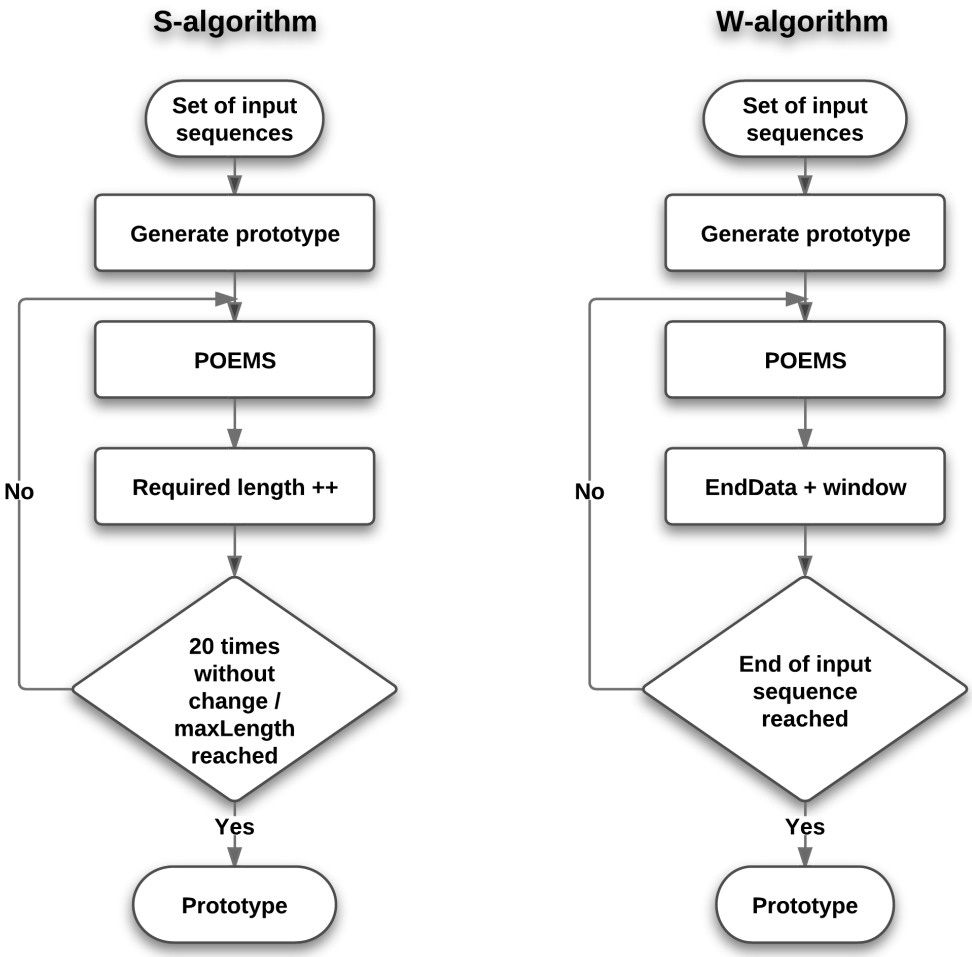

Figure 2: The flowchart of the S-algorithm and the W-algorithm

5 Implementation

In this section an implementation of entire algorithm is described in details as it was implemented in Java. The class diagram is shown in Figure 3.

5.1 Implementation of POEMS

Lets start with the class **startBatch**. In its main method it reads configuration of algorithm and loads data. Then new instance of class POEMS is created and in dependance of the type of the algorithm its method start() or runSteps() is launched. Both of these methods returns the resulting solution and class startBatch then only save results to files.

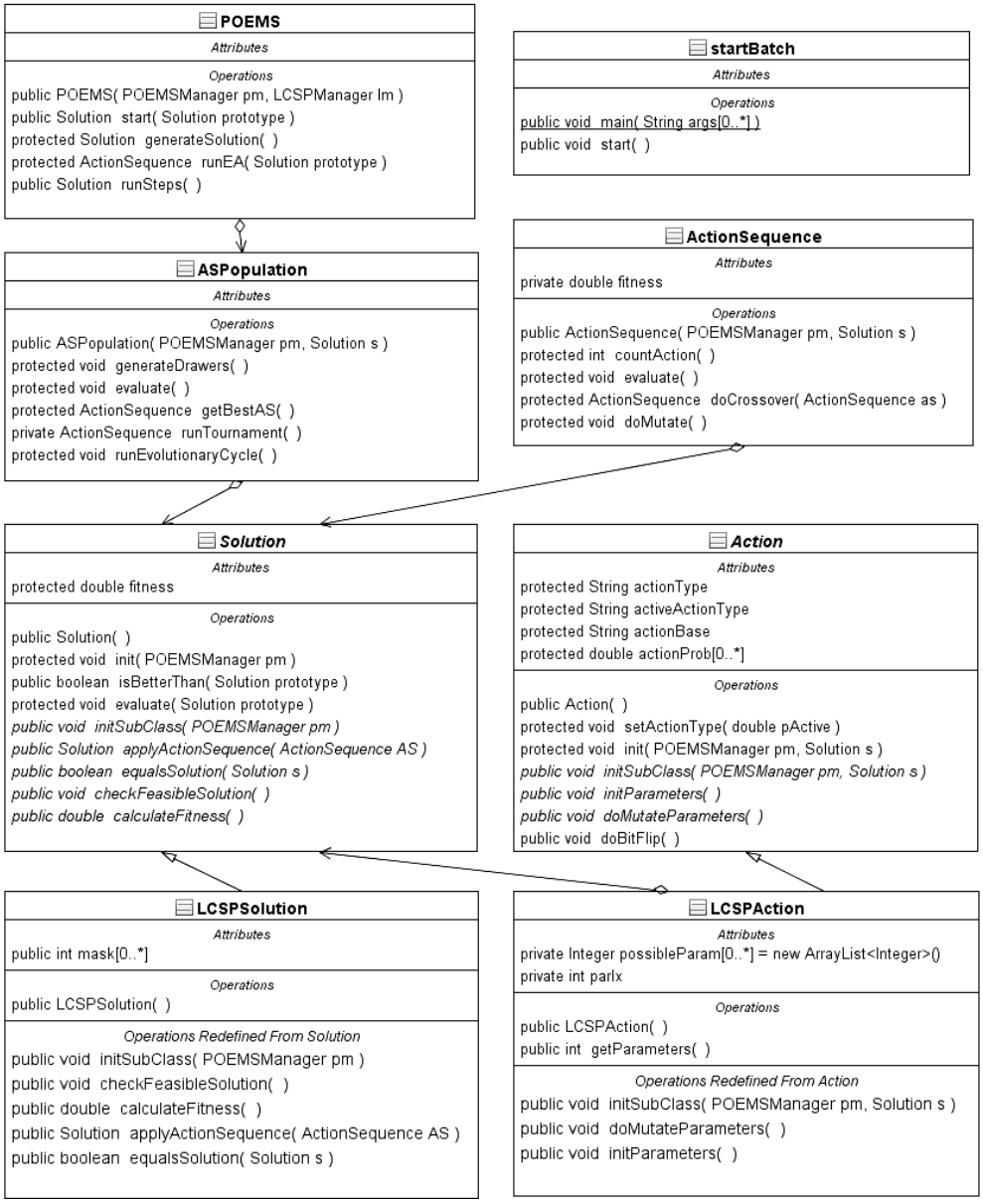

Figure 3: The class diagram of the POEMS implementation

Class **POEMS** returns the solution to given problem. At the beginning there is a method generateSolution() which generates new instance of class Solution and returns it initialized accordingly to the currently used algorithm. Method runEA(Solution prototype) runs an evolutionary algorithm on current prototype. At the beginning of each evolutionary algorithm it creates a new instance of class ASPopulation and then runs its method runEvolutionaryCycle() while the *nGenerations* cycles are not completed. At the end it returns the best AS from the ASPopulation. Method start() uses both above mentioned methods. After generating solution (namely prototype) it calls method runEA(Solution prototype) to find the best AS. If this AS does not worsen the prototype, it applies AS to the prototype using method applyActionSequence(ActionSequence AS) from class Solution. This all is done *nIterations* times and then the prototype is returned. Finally there is a method runSteps used in steps algorithm which iteratively increasses the required length of the prototype and runs method start until the terminal conditions are not reached.

Class **ASPopulation** extends an array list of ActionSequences. It contains a method generate Drawers() used during initialization, method evaluate() and method getBestAS() which returns the best AS from the list. Then there are two methods in relation with evolutionary algorithms: the first is the method runEvolutionaryCycle() where crossover and mutation take place with probabilities defined in configuration. For choosing two "parents" for "children" it uses the second method runTournament(), which *nTournament* times compares randomly choosen ASs from the list and returns the best AS for each parent. Children are added to the list when they are not the worst with regard to their fitness function.

Class **ActionSequence** which extends an array list of Actions has method doCrossover(ActionSequence as) and the method doMutate(). The first of them creates two children from two parents (the instance and the parameter). One of the children is returned by this method and by the second the current instance is replaced. The method doMutate() specifies the type of mutation of the actions in AS, concretely whether type of action or its parameters will be mutated. There is also method evaluate() used for evaluating children in evolutionary cycle. The last important method in this class is countAction() which returns the number of active actions in this AS.

There are two abstract classes in POEMS implementation which represent the problem specific classes. The first is the class **Action** containing methods relationed with initialization, namely the setActionType(double pActive) method and the method doBitFlip(). The setActionType(double pActive) method sets

the type of this action to randomly choosen actiontype from the list of possible types given by problem specific part of the algorithm. With pActive probability the action is set to active type, else it stays "NOP". Method doBitFlip() can exchange the active actiontype for NOP and vice versa. Other methods are abstract and specify the initialization and mutation for the problem specific parts.

The second abstract class is class **Solution**. It contains method evaluate(Solution prototype) which sets the fitness to the candidate solution with respect to the current prototype. For comparing solutions there are two methods: isBetterThan(Solution prototype) which returns true if candidate solution has the same or better fitness than prototype and equalsSolutions(Solution s) method. The second of them is abstract as well as oher methods such as importatnt applyActionSequence(ActionSequence) method, calculateFitness() used by all evaluate methods and method reallyFeasible(). Also initialization of Solution class instance is problem specific.

5.2 Implementation of LCSP

This part overrides two abstract classess from the POEMS part of entire algorithm. During the initialization of class **LCSPAction** list of possible action types called actionBase is created. For this implementation there are "switchOn" action and "swithcOff" action in this list. In dependence on the action type method initParameters() is executed choosing only feasible parameters for each type of the action. For "switchOn" action only the off-bits are possible as parameters and analogously "swithcOff" actions can switch off only on-bits. According to the same rule the mutation of the parameters can be done by method doMutateParameters() when the parameter is increased or decreased to the next possible value.

Class **LCSPSolution** represents the solution to the LCSP. It contains an array of 0s and 1s which acts as the *mask* determining the appearance of the corresponding subsequence. This array is initialized according to the type of algorithm either to all 0s or to all 1s. Method reallyFeasible() returns true if the subsequence represendted by the *mask* is subsequence of all input sequences. In this class there is also important calculateFitnes() function which returns the value of fitness depending on current type of the algorithm. Previously mentioned method applyActionSequence(ActionSequence AS) modifies the *mask* in accordance with the AS from its parameter. It goes through the list of AS's actions and in consonance with their type change matching bit in the *mask*. This class also implements method equalsSolutions(Solution s). It returns true if both

solutions represent the same subsequences. Note that to be equal Solutions do not necessary have to have the same *mask.*

6 Experiments

Here proposed algorithms were both implemented in Java on personal computer Intel(R) Pentium(R) Dual CPU 2.16GHz with 1,96GB RAM using Windows XP. The same computer was used also for running tests. At first a few experiments were designed to test varying configurations of proposed algorithms and back up some arguments in previous sections. Results of this experiments are brought together in Table 1.

The rest of the tests are focused on comparison of solutions found by current state-of-the-art algorithms (namely IBS-LCSP [4] and BS [5]) and the solution found by POEMS. Two types of BS algorithm were proposed in [5] but only the one which emphasize quality of resulting solution (BS-high-quality, BShq respectively) was considered since the length of the solution is the main priority for both W-algorithm and S-algorithm.

6.1 Datasets

Datasets used for testing most other algorithms were choosen to obtain meaningful results. On the same datasets two current state-of-the-art algorithms were tested, namely improved algorithm [4] or beam search [5]. These datasets consist of various types of alphabet. There is 0-1 alphabet in the randomly generetad ES dataset and also typical DNA alphabet consists of four letters: A, C, G and T. Alphabet of size 20 denoting types of amino acids in proteins also can be found as well as alphabets of the size from 10 to 100 characters in randomly generated ES. Datasets named ES and rand are randomly generated on contrary of Rat and Virus datasets which comes from real genomes of rat and virus.

Rat, virus and rand datasets contain only two types of alphabet. Classical DNA alphabet consists of four letters denotes the base, protein 20-characters alphabet designates the type of amino acid in this protein. In case of DNA sequences there are not only four characters in datasets, but also the "N" character appears. It represents non-recognized base, thus it could be regarded as arbitrary character from given alphabet. The same role is played by the character "X" in protein datasets where it denotes an arbitrary amino acid. Except randomly generated rand dataset all sequences originates from real DNA sequences from NCBI (National Center for Biotechnology Information) taking first 600 character from obtained sequence. To run tests only sets with number of input

sequences equal to 10, 40, 100 and 200 were choosen as they show the performance of proposed algorithms enough.

Randomly generated ES dataset contains various type of alphabets. Namely the lengths 2, 4 and 10 for length of input sequences equals to 1000, 25-characters alphabet for length of 2500 and at the end alphabet which consists of 100 character with input seguences of length 5000. Due to lack of time for tests only four datasets from ES were choosen, namely that of $|\Sigma|$ equal to 2 and 10 with combination of number of sequences in dataset equal to 10 and 50.

6.2 Setup

Rat, virus and rand dataset contain only one file per each combination of length of the alphabet and number of sequences thus all of these datasets were tested five times independantly to get reliable results. The average values for each dataset are presented in corresponding tables.

The ES dataset contains 50 different input files for each combination of the length of the alphabet and the number of input sequences. Thus the algorithm was run only once on each data file and the presented results show the average lengths of sequences among all 50 files. Even though the both proposed algorithms put stress on quality of the solution rather than at time, the average values of required execution timed are shown as well. Standard deviations of the corresponding values can be found in brackets. Aditionally the performance ratio (α) defined as (W-algorithm – IBS-LCSP)/IBS-LCSP (analogously for S-algorithm) is counted to make comparison of new algorithms and the current state-of-the-art IBS-LCSP easier.

6.3 Configuration of POEMS

Proposed algorithms were tested with folowing configuration: An *nIterations* of POEMS was set to 20 for the S-algorithm and to 50 for W-algorithm and probability of setting action to be active was considered as 50%. Genetic algorithm used in POEMS was configurated to 20 *nGenerations* and 10 *nTournaments*, maximal possible size of AS (*maxGenes*) was set to 10 and *sizeDrawer* to 20. Thus GA works with 200 AS. Probabilities was following: p. of crossover = 71%, p. of mutation = 18% and p. of switching NOP and active action = 26%.

Except the first five experiments designed for parameters setting, tests were run on algorithms as they are described in previous sections. The S-algorithm does not need any special setting whereas for the W-algorithm its only parameter

window was configurated to 20. Both algorithms start with an empty solution thus an array *mask* was initialized to all 0s.

6.4 Results

In order to briefly present the results of the first part of experiments focused on the appropriate setup of the algorithms Table 1 was created. At first the results of final setup of algorithm is for better comparison placed in the first column. Results of experiment which tries to demonstrate improvement caused by adding sum of indexes in *mask* to the current fitness function follows in the column 2. As is seen from table the difference in lengts of founded solutions is not much relevant. The third test was focused on setting the only parameter of the W-algorithm namely the *window*. For further tests the *window* = 20 and it is clear that the larger window has negative impact on the quality of solution. Also *nIterations* as an important parameter of POEMS algorithm was tested and as it is seen from the fourth column more iterations cause better performance of algorithm as it has more time to find better solution. Latterly the fourth experiment was run to show differences between the algorithm which uses two types of actions and the one which uses only one type. Additionally the difference of results while working only with feasible solutions and while infeasible solutions are allowed is presented at the end of this table.

All these tests run the W-algorithm as the S-algorithm requires a lot of time and moreover it does not need any specific configuration. For tests only two dataset of rat genome was used as the results for virus and random dataset would be similar. Test were designed to put stress on quality of the resulting sequence, thus only the average lengths of subsequences are compared in the table.

Concerning the comparison of proposed algorithm and the current state-of-the-art algorithms there are six tables showing the performance of W-algorithm on DNA and protein datasets. In the first column the number of input sequences for appropriate dataset is defined. Results of W-algorithm given in second column are calculated as average lengths of obtained sequences (LLCS) while BShq and IBS_LCSP results folows in further columns. For each algorithm the average time results calculated in seconds are placed in special columns whereas the performance ratio takes place at the end of each table beeing calculated from results of W-algorithm with regard to the IBS-LCSP.

Results of ES dataset are divided into two tables (see Table 8, 9). The first of them contains results from datasets with number of input sequences equal to 10 on which both algorithms were tested. Considering datasets with larger alphabet size the W-algorithm provides solutions of higher quality than the S-algorithm but it

would need more experiments to compare it together. On the datasets with 50 input sequences only S-algorithm was tested.

In spite of expectance obtained results were not as good as that of the IBS algorithm. Nevertheless they are quite close to them considering datasets with large number of input sequences. Especially for rat datasets of number of input sequences egual to 200 the performance ratio reaches the value of -0.03.

Table 1: Parameters setting.

rat	RRm	RRm + Σ(i)	*window* = 50	*nIter*. = 20	one action	all feasible
|Σ| = 4	174.6(3.5)	173.6(2.8)	169.4(2.1)	170.6(1.6)	174.8(1.8)	176.6(3.3)
|Σ| = 20	60.6(1.2)	59.6(2.4)	58.6(1.9)	57.2(0.4)	59.0(1.8)	61.2(1.2)

Table 2: Results for rat instance of alphabet size 4. Comparison of the W-algorithm with BS-hiqh-quality and the improved BS algorithm for rat instance of alphabet size 4.

|Σ| = 4

l = 600	W-algorithm		BShq		IBS-LCSP		α
k	LLCS	time	LLCS	time	LLCS	time	
10	175(3.5)	173.7	191	9.7	**199**	0.4	-0.12
40	141(2.4)	323.1	**146**	9.4	**146**	0.5	-0.03
100	127(1.0)	589.5	**132**	38.5	**132**	1.0	-0.03
200	117(2.1)	1050.0	**121**	69.1	120	1.6	-0.03

Table 3: Results for rat instance of alphabet size 20. Comparison of the W-algorithm with BS-hiqh-quality and the improved BS algorithm for rat instance and alphabet size 20.

|Σ| = 20

l = 600	W-algorithm		BShq		IBS-LCSP		α
k	LLCS	time	LLCS	time	LLCS	time	
10	61(1.2)	165.3(2.7)	69	27.4	**70**	0.5	-0.13
40	44(0.5)	286.2(8.2)	**49**	47.0	**49**	0.6	-0.10
100	35(1.0)	509.3(2.6)	38	64.8	**39**	1.1	-0.10
200	31(1.1)	863.6(7.4)	**33**	101.0	32	1.7	-0.03

Table 4: Results for virus instance of alphabet size 4. Comparison of the W-algorithm with BS-hiqh-quality and the improved BS algorithm for virus instance and alphabet size 4.

$|\Sigma| = 4$

l = 600	W-algorithm		BShq		IBS-LCSP		α
k	LLCS	time	LLCS	time	LLCS	time	
10	193(2.7)	176.1(3.1)	212	11.6	**225**	0.5	-0.14
40	154(2.0)	330.6(1.1)	162	21.9	**168**	0.6	-0.08
100	144(1.7)	631.1(4.0)	150	43.9	**158**	1.2	-0.09
200	144(0.7)	1140.2(1.3)	145	84.5	**154**	2.1	-0.06

Table 5: Results for virus instance of alphabet size 20. Comparison of the W-algorithm with BS-hiqh-quality and the improved BS algorithm for virus instance and alphabet size 20.

$|\Sigma| = 20$

l = 600	W-algorithm		BShq		IBS-LCSP		α
k	LLCS	time	LLCS	time	LLCS	time	
10	64(1.0)	164.1(2.7)	**75**	27.2	**75**	0.5	-0.15
40	44(0.8)	286.8(8.2)	**49**	48.4	**49**	0.6	-0.10
100	39(0.7)	523.3(2.7)	43	74.2	**44**	1.5	-0.11
200	39(0.6)	906.4(8.2)	43	140.0	**44**	2.2	-0.11

Table 6: Results for random instance of alphabet size 4. Comparison of the W-algorithm with BS-hiqh-quality and the improved BS algorithm for random instance and alphabet size 4.

$|\Sigma| = 4$

l = 600	W-algorithm		BShq		IBS-LCSP		α
k	LLCS	time	LLCS	time	LLCS	time	
10	189(2.5)	174.0	211	9.8	**218**	0.4	-0.13
40	157(0.5)	333.8	167	21.0	**172**	0.7	-0.09
100	146(0.9)	639.4	154	40.3	**158**	1.2	-0.08
200	140(1.2)	1141.7	146	74.3	**150**	2.1	-0.07

Table 7: Results for random instance of alphabet size 20. Comparison of the W-algorithm with BS-hiqh-quality and the improved BS algorithm for random instance and alphabet size 20.

$|\Sigma| = 20$

l = 600	W-algorithm		BShq		IBS-LCSP		α
k	LLCS	time	LLCS	time	LLCS	time	
10	50(1.0)	159.7	**61**	33.3	**61**	0.5	-0.18
40	33(0.7)	277.0	37	43.2	**38**	0.6	-0.13
100	28(0.6)	500.0	**31**	59.2	**31**	1.2	-0.10
200	25(0.4)	862.8	27	98.0	**28**	1.9	-0.11

Table 8: Results for sets of 10 input sequences from ES instance. Comparison of the W-algorithm and the S-algorithm with BS-hiqh-quality and the improved BS algorithm for sets of 10 input sequences from ES instance.

k = 10

l = 1000	W-algorithm		S-algorithm		BShq		IBS-LCSP		α(W)	α (S)
$\|\Sigma\|$	LLCS	time	LLCS	time	LLCS	time	LLCS	time		
2	556.7(4.7)	498.1	561.3(5.3)	1027.2	592.6	14.8	**610.2**	0.9	-0.09	-0.08
10	161.7(2.7)	3354.5	158.5(3.0)	312.5	192.2	9.4	**199.7**	0.9	-0.19	-0.21

Table 9: Results for sets of 50 input sequences from ES instance. Comparison of the S-algorithm with BS-hiqh-quality and the improved BS algorithm for sets of 50 input sequences from ES instance.

k = 50

l = 1000	S-algorithm		BShq		IBS-LCSP		α
$\|\Sigma\|$	LLCS	time	LLCS	time	LLCS	time	
2	507.3(2.0)	2695.2	521.9	43.5	**535.0**	1.5	-0.05
10	115.4(1.6)	698.9	129.6	18.8	**134.6**	1.4	-0.14

7 Discussion

In this section possible reasons for low quality of obtained sequences are discussed as well as problems which appeared within the implementation.

7.1 Feasibility of Candidate

There are two ways to assure resulting solution to be feasible (i. e. to be subsequence of all input sequences). At first we can consider only the feasible subsequences through entire algorithm. It requires to use a method which turns all infeasible solutions to the feasible ones before each calculation of fitness function. This problem can be tackled by deleting characters which stand in the way solution to be feasible (considered from left to right). Such behaviour may be considered as quite big interference with process of genetic algorithm but on the other hand it can better preserve the diversity of candidate solutions.

Alternatively the infeasible solutions are allowed, but they have much less fitness values than the feasible ones. Therefore feasible solution always has the greater fitness than infeasible, so it can not be replaced by infeasible one. Since starting from the empty sequence (which is always a common subsequence) algorithm never ends with the infeasible solution.

7.2 Initialization of Prototype

It is a big question how should the first prototype look like. Most of previously used algorithms are constructive and start with empty solution as well as here proposed algorithms. Both W-algorithm and S-algorithm begin with an empty subsequence and iteratively try to fill it with feasible characters. But still there are other possibilities how to create the first prototype. As opposition to the empty subsequence the *mask* can be initialized to all ones. It means that at the beginning of the algorithm the first prototype looks entirely like the reference sequence. Then within the algorithm infeasible characters are deleted until feasible solution is reached. Special fitness function is needed here to decide which characters should be deleted to obtain subsequence as long as possible. Other alternative is to initialize *mask* to 0s and 1s randomly, however it was realised that it is not very good method as it is too much determining. Consequently it is very difficult for AS to improve the prototype especially when it is necessary to change large number of characters.

7.3 Problem Representation

This paragraph is focused on comparison between representation of solution as *mask* on shortest input sequence (*reference*) and as a list of characters from given alphabet. For good performance the probability with that changes takes place (eventually the speed of modifying the solution) is important and it is closely associated with the representation of the problem.

For better understanding lets consider following example: First start with quite short input sequences. Having reference sequence ACCBD and other two sequences CCBDA, CCBDD it is clear that the longest common subsequence is CCBD. Lets think about the mask representation. Suppose the *mask* 11100 and corresponding subsequence ACC. It is seen, that it is necessary to remove the character A to be able to obtain longer subsequence. That can be done only by "switchOff" action with parameter 0. If we consider only "switchOff" actions this modification has probability 1/3 (one of three switched-on characters). With list representation and corresponding two types of actions the probability is the same. (There are only three characters to be switched off.) However if we want to add some character on the particular place, the probabilities will be different. Now having the *mask* 01100 and subsequence CC we want to add B to obtain longer subsequence CCB. While using mask representation the probability of required action ("switchOn" on 3) is 1/3 again (one of three switched off characters) but using list representation it is necessary to choose not only the position, but also the character which will be added. The probability of adding B is then 1/3x1/4 = 1/12. (The probability of choice of the right place and the probability of choice of the right character.)

For short sequences mask representation seems to be better. However everything change dramatically when the sequences will be much longer. Imagine the simmilar case as above with input sequences of length 100. For ACCBDAAAA ... A, CCBDABBBB ... B, CCBDDDDDD ... D the LCS will be the same (CCBD) but probabilities of modifications will change. Having *mask* 111000000 ... 0 probability of removing A stays 1/3 (one of three on-bites), but when we want to add B, with mask representation it will be more difficult. The probability of action which add B at the end of subsequence CC is now 1/98 (one character of 98 switched-off characters. But with list representation it still remains 1/12. Therefore for longer sequences list representation appears as more useful.

Situation changes not favourably for list representation in case of larger alphabet. If the alphabet of this particular problem will be of size 50 the probability of adding B will change to 1/3x1/50 = 1/150 while with the mask representation it stays the same since type of character is choosen with place. The same case comes on with elongation the sequences when the number of possible places grows.

Considering the same number and types of active actions following formulae for probability of adding the right character on the particular place hold:

for mask representation:

$P = 1/(n - |c(s)|)$,

for list representation:

$P = 1/((|c(s)| + 1)|\Sigma|)$,

where n is the length of the *reference* and $|c(s)|$ denotes the length of candidate solution.

If we match these two formulae together it can be deduced following:

$|c(s)| = (l - |\Sigma|)/(|\Sigma| + 1)$.

If this formula holds, probabilities for mask and list representation are the same. Length of candidate solution greater than this boundary will make the mask representation more convenient, in case of shorter candidate solution it would be better to choose the list representation.

It is clear that the boundary value depends on length of input sequences and on used alphabet. Thus we are able to count the boundary values for datasets used in this work to sum up which representation would have better performance. For DNA sequences of length 600 the boundary is (600-4)/(4+1) = 119,2, for proteins datasets it can be calculated as (600-20)/(20+1) = 27,5. These numbers are quite close to lengths of resulting sequences thus it would be considered that it is not important which representation is used. Nevertheless at the beginning when actions are applied on very short candidate solutions the list representation always win.

7.4 Further Actions

To change representation of problem is not the only way to improve performance of proposed algorithm. Also adding some specific actions can cause faster modification of the prototype and shorten the way to the better solution. Excepting "switchOn" action (on position) and "switchOff"action (on position) there can be also action, which switch two characters together or change the character to another from given alphabet. More types of actions can be useful in both representation the mask and the list.

Number of actions needed to perform some modifications:

	add char.	remove char.	switch 2 char.	change char.
sw.On/sw.Off	1 action	1 action	2 actions	2 actions

If each two actions which is needed to switch two characters and change character to other from given alphabet will be replaced by one special "switchTwo" and "changeCharacter" action, the modification of prototype could be easier.

7.5 Characters Specified by the Mask

The last problem which was found using mask representation in relation with S-algorithm is not easy to describe therefore use the example again. Suppose the S-algorithm as is described in section 4 and start with subsequence with required length 1. Imagine, that character T is in all sequences at the first position. Therefore the best subsequence is with no doubt T and actions whose result is T will have the best fitness. The problem occures if T appears also somewhere at the end of the reference sequence e.g. on the last position. Consequently the action which switches on the last bit in *mask* can be considered as the best, however for the rest of the algorithm there is no chance to add some character after T. Characters can be added only before T, but if it is required to have T as the first character adding anything before it is not desired. This problem is caused by the fact, that the choosen T is not whatever T, but exactly the T on the last position in *reference*. While evaluating T it is not controlled, where the T is, and algorithm considers as T the first T which appears in each subsequence. To deal with this problem the T at the end of the reference sequence should has not the same fitness as T choosen from the first position.

First attempt at improvement of the algorithm preformance was adding the "least-index" request to fitness function. Sum of indexes in *mask* should be the least possible, means that whan we want to add T, it should be such T in *reference*, which has the least index. Fitness function then may looks as follows:

(previous fitness)(*maxI*) + $\Sigma\{mask[i] \mid i = 1, ..., n\}$

where *maxI* is maximal possible value of addded sum and it is equal to $l(l - 1)/2$.

Regrettably it turned out that it is not much helpful due to problem mentioned in the first paragraph. It is very small probability that the action will pitch upon the "best T", it means the T at the beginning of the reference sequence. Concretely if we consider the sequence of length of 600 characters, the probability of choosing the right T is 1/600 to switch on required T. When the right T is not choosen at first, the probability changes to 1/360000 (1/600 to delete currently choosen T x 1/600 to add the right one). There wont be this problem with the list representation where it is enough to choose the best character from the alphabet.

In this work the algorithm deals with this problem by special attitude to the reference sequence, while it consideres not the first found charecter but exactly the one specified by the *mask*. Nevertheless it is still not enough to move "T"s to the beginning of *reference*.

8 Conclusions and Future Work

In this work two algorithms employing POEMS algorithm were proposed to deal with the LCSP for arbitrary number of input sequences. However obtained results are not as good as was expected especially for less numbers of input sequences. This fact is attributed to several problems discussed in previous section. Nevertheless for large sets of input sequences results are quite close to the current state-of-the-art algorithms with regard to the solution quality. However, the POEMS is much worse than both compared state-of-the-art algorithms in terms of the computational time. This is probably caused by repeated computation of the population-based evolutionary algorithm.

In the future it would be interesting to run proposed algorithms with combination of the list representation or with more types of actions. Further experiments can also investigate the dependance of solution quality on *window* size in case of W-algorithm.

9 References

[1] T. Jansen and D. Weyland, "Analysis of evolutionary algorithms for the longest common subsequence problem". In Proceedings of the 9th annual conference on Genetic and evolutionary computation. New York: ACM, 2007, p. 939-946. ISBN: 978-1-59593-697-4.

[2] D. Weyland, "Simulated annealing, its parameter settings and the longest common subsequence problem". In Proceedings of the 10th annual conference on Genetic and evolutionary computation. New York: ACM, 2008, p. 803-810. ISBN: 978-1-60558-130-9.

[3] B. A. Julstrom and B. Hinkemeyer, "Starting from scratch: Growing longest common subsequences with evolution". In Proceedings of the 9th international conference on Parallel Problem Solving from Nature. Springer-Verlag Berlin, Heidelberg: 2006, p. 930-938. ISBN: 978-3-540-38990-3.

[4] S. R. Mousavi and F. Tabataba, "An improved algorithm for the longest common subsequence problem". Computers and Operations Research 2012, 39(3), p. 512-520.

[5] Ch. Blum, M. J.Blesa, M. López-Ibáñez, "Beam search for the longest common subsequence problem". Computers and Operations Research 2009, 36(12) p. 3178–86.

[6] Ch. Blum and M. J. Blesa, "Probabilistic Beam Search for the Longest Common Subsequence Problem". Lecture Notes in Computer Science 2007, p. 150-161. ISBN: 978-3-540-74446-7.

[7] J. Kubalík, "Solving the DNA Fragment Assembly Problem Efficiently Using Iterative Optimization with Evolved Hypermutations". In Proceedings of the 12th annual conference on Genetic and evolutionary computation. New York: ACM, 2010, p. 213-214. ISBN: 978-1-4503-0072-8.

[8] B. Hinkemeyer and B. A. Julstrom. "A Genetic Algorithm for the Longest Common Subsequence Problem". In Proceedings of the 8th annual conference on Genetic and evolutionary computation. Seattle: ACM, 2006, p. 609-610. ISBN:1-59593-186-4.

[9] S. J. Shyu and C.-Y. Tsai, " Finding the longest common subsequence for multiple biological sequences by ant colony optimization". Computers & Operations Research 2009, 36(1), p. 73-91.

[10] J. Kubalík, "Efficient stochastic local search algorithm for solving the shortest common supersequence problem". In Proceedings of the 12th annual conference comp on Genetic and evolutionary computation [CD-ROM]. New York: ACM, 2010, p. 249-256. ISBN 978-1-4503-0073-5.

[11] J. Kubalík, "Evolutionary-Based Iterative Local Search Algorithm for the Shortest Common Supersequence Problem". In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation [CD-ROM]. New York: ACM, 2011, p. 315-322. ISBN 978-1-4503-0557-0.

[12] H.-Y. Wenga, S.-H. Shiaub, C.-B. Yanga, Y.-H. Penga and K.-S. Huanga, "An Enhanced ACO Algorithm with Pair Matching Strategy for the Longest Common Subsequence Problem".

[13] T. Jiang and M. Li, "On the approximation of shortest common supersequences and longest common subsequences". SIAM Journal on Computing 1995, 24(5), p. 1122–39.

[14] P. Bonizzoni, G. D. Vedova, G. Mauri, "Experimenting an approximation algorithm for the LCS". Discrete Applied Mathematics 2001, 110, p. 13–24.

[15] K. S. Huang, C. B. Yang, K. T. Tseng, "Fast algorithms for finding the common subsequence of multiple sequences". In Proceedings of international computer symposium. Taipei, Taiwan: 2004. p. 90–95.

[16] H.-Y. Weng, S.-H. Shiau, K.-S. Huang, and C.-B. Yang, “Hybrid algorithm for the longest common subsequence problem”. In Proceedings of the 26th Workshop on Combinatorial Mathematics and Computation Theory. Chiayi, Taiwan: 2009, p. 122–129.