NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCE
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS
and

NCSR DEMOKRITOS

INSTITUTE OF INFORMATICS AND TELECOMMUNICATIONS
SOFTWARE AND KNOWLEDGE ENGINEERING LABORATORY

DIPLOMA THESIS

N-gram graph decompression

Despina - Athanasia I. Pantazi

Supervisors: George Giannakopoulos, Dr (NCSR Demokritos)
and
Panagiotis Stamatopoulos, Assistant Professor (National and Kapodistrian University of Athens)

ATHENS

SEPTEMBER 2016

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
και

ΕΚΕΦΕ ΔΗΜΟΚΡΙΤΟΣ

ΙΝΣΤΙΤΟΥΤΟ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ
ΕΡΓΑΣΤΗΡΙΟ ΤΕΧΝΟΛΟΓΙΑΣ ΓΝΩΣΕΩΝ ΚΑΙ ΛΟΓΙΣΜΙΚΟΥ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αποσυμπίεση γράφων ν-γραμμάτων

Δέσποινα - Αθανασία Ι. Πανταζή

Επιβλέποντες: Γιώργος Γιαννακόπουλος, Δρ. (ΕΚΕΦΕ Δημόκριτος)
και
Παναγιώτης Σταματόπουλος, Επίκουρος Καθηγητής (Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών)

ΑΘΗΝΑ

ΣΕΠΤΕΜΒΡΙΟΣ 2016

DIPLOMA THESIS

N-gram graph decompression

Despina - Athanasia I. Pantazi

1115 2011 00190

Supervisors: George Giannakopoulos, Dr (NCSR Demokritos)
and
Panagiotis Stamatopoulos, Assistant Professor (National and Kapodistrian University of Athens)

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αποσυμπίεση γράφου ν-γραμμάτων

Δέσποινα - Αθανασία Ι. Πανταζή

Α.Μ.: 1115 2011 00190

ΕΠΙΒΛΕΠΟΝΤΕΣ: Γιώργος Γιαννακόπουλος, Δρ. (ΕΚΕΦΕ Δημόκριτος)
και
Παναγιώτης Σταματόπουλος, Επίκουρος καθηγητής (Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών)

ABSTRACT

The current thesis examines the information represented by an n-gram graph. To achieve this task, we searched for all the strings that can be compressed to the same n-gram graph. In order to find these strings, we defined the n-gram graph decompression problem. In addition, we defined the constraint satisfaction problem formulation of the decompression problem, to optimize the latter, and we applied a set of search methods to solve it. We also designed two variable ordering heuristics to improve our time measurements. Finally, we conducted a set of experiments on certain applied search methods to retrieve the initial text from the n-gram graph. We compared the findings from our experiments and we concluded that the best results for short-length texts were achieved when Local Search was performed. For longer-length strings, the weighted degree heuristic was the one that offered the best results.

SUBJECT AREA: Artificial Intelligence
KEYWORDS: n-gram graph, decompression, local search, constraint satisfaction problem, depth first search, breadth first search, heuristics

ΠΕΡΙΛΗΨΗ

Στα πλαίσια αυτής της εργασίας, μελετήσαμε την πληροφορία που περιέχει ένας γράφος ν-γραμμάτων. Για να πραγματοποιήσουμε το παραπάνω, αναζητήσαμε όλες τις συμβολοσειρές που μπορούν να συμπιεστούν και να παράξουν τον ίδιο γράφο ν-γραμμάτων. Για να βρούμε αυτές τις συμβολοσειρές, ορίσαμε το πρόβλημα αποσυμπίεσης του γράφου ν-γραμμάτων. Επιπλέον, εξετάσαμε το πρόβλημα ως πρόβλημα ικανοποίησης περιορισμών, έτσι ώστε να το βελτιστοποιήσουμε, και εφαρμόσαμε ένα σύνολο μεθόδων αναζήτησης για να το επιλύσουμε. Παράλληλα, σχεδιάσαμε δύο ευριστικές συναρτήσεις για να καλυτερεύσουμε τους χρόνους των μετρήσεών μας. Τέλος, πραγματοποιήσαμε ένα σύνολο πειραμάτων σε συγκεκριμένους αλγορίθμους αναζήτησης, για να αποσυμπιέσουμε το κείμενο που έχει συμπιεστεί στους γράφους ν-γραμμάτων. Συγκρίνοντας τα αποτελέσματα των πειραμάτων, συμπεράναμε πως τα καλύτερα αποτελέσματα για την αποσυμπίεση μικρών συμβολοσειρών είναι αυτά που προκύπτουν με την εφαρμογή της τοπικής αναζήτησης. Οι συμβολοσειρές που περιέχουν μεγαλύτερο αριθμό γραμμάτων αποσυμπιέστηκαν γρηγορότερα με τη χρήση της ευριστικής συνάρτησης που βασίστηκε στο βάρος των κόμβων των γράφων ν-γραμμάτων.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Τεχνητή Νοημοσύνη
ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: γράφος ν-γραμμάτων, αποσυμπίεση, τοπική αναζήτηση, αναζήτηση κατά βάθος, αναζήτηση κατά πλάτος, ευριστικές συναρτήσεις

To Lucas and Jacob

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

1 INTRODUCTION

In the fields of computational linguistics and probability, n-grams are proven a useful approach to represent information with various applications, such as text classification and text summarization. What is most interesting about this tool is that we can search for all the strings that are compressed to an n-gram graph, by defining the n-gram graph decompression problem. By formulating the n-gram graph decompression problem as a constraint satisfaction problem, we are able to retrieve the initial text, which is compressed in an n-gram graph, by applying a set of search algorithms.

According to [12] an n-gram is an n-character slice of a longer string. Although in the literature the term can include any n co-occuring characters of a string s, in this work an n-gram is simply defined as a substring of a string s, where s has at least n characters. In [3], a statical, language-neutral and generic representation is defined: the n-gram graphs, where n-grams are combined with the powerful concept of the graphs. Using a simple extraction algorithm, a text T is compressed to a weighted directed graph, where nodes represent n-grams, the directed edges indicated the connection between a pair of n-grams and the weight of each edge of the graph indicates the strength of connection each pair of n-grams has. The weight of an edge <a, b>, between the n-grams a and b, is the amount of times b was found within a distance $Dmax$ of a.

The n-gram graphs offer richer information than widely used representations. They are not a lossy text representation method, as they use the whole text T, by splitting it into n-grams. We use the JINSECT toolkit[1], which is a Java-based toolkit and library that supports the use of n-gram graphs on a whole range of Natural Language Processing applications. The toolkit is a contribution to the NLP community, under the $LGPL$ licence that allows free use in both commercial and non-commercial environments.

By exploring the n-gram graph, we can extract the initial text T, which was compressed into an n-gram graph. We compress a text T to an n-gram graph G, by using the toolkit JINSECT, and focus on different algorithmic approaches to retrieve the original text T when we have as input the n-gram graph G. The attempt to retrieve the original text is called decompression of the n-gram graph. In this work, we study this decompression process of the n-gram graphs, by executing the search algorithms Local Search (LS), Depth First Search (DFS) and Breadth First Search (BFS). We defined the constraint satisfaction problem (CSP) simulation of the decompression problem, so that the experiments we performed on the above applied search methods can indicate the most optimal search algorithm to complete the decompression procedure. In addition, two variable ordering heuristics were applied to the backtracking algorithms DFS and BFS, in order to maximize the optimization of the results of the decompression problem of the n-gram graphs.

The experimental setting and evaluation of the decompression problem of the n-gram graphs were focused on strings that have fixed length, which is provided by the user. In Section 3.4, we describe that an n-gram graph can be decompressed to more that one strings. In this work, we expect to find the first solution of the n-gram graphs, so we use text that does not include any duplicate characters. The comparison of the results each search method provides, through its experiments, is based on the time the algorithm requires to find the initial text, and the method cost, which is the amount of methods each algorithm had executed, until the decompression problem was solved.

[1] JINSECT toolkit can be downloaded from: https://sourceforge.net/p/jinsect/wiki/Home/

The report has the following organization: In Section 2 we analyze the basic concepts related to our study, which include the description of the n-gram graph representation, the definition of the constraint satisfaction problems, and the search algorithms we will use for decompressing the n-gram graphs. In Section 3, we explain the modeling and solving of the search problems we use in the code of our program and we describe the constraint satisfaction problem simulation of the n-gram graph decompression problem. Furthermore, we design the heuristics we will use to try and optimize the results of our experiments, and we explain that some n-gram graphs can be decompressed to more than a unique initial text T. In Section 4, we focus on the experimental setting and evaluation of the decompression problem, by referring to the theoretical aspects of the algorithms we used, while in Section 5 we observe the practical aspects of the these algorithms. Last but not least, we conclude this thesis in Section 6, with the discussion of the results of the various decompression approaches we implemented, and we suggest some future research in this field.

2 BASIC CONCEPTS AND RELATED WORK

2.1 The N-Gram Graph Representation

In Introduction, an n-gram is described as an n-character slice of a longer string.

Example 2.1.1 Examples of n-grams from the sentence: *This is an example.*
Word unigrams: this, is, an, example
Word bigrams: this is, is an, an example
Character bigrams: th, hi, is, s_, _a, an, ...
Character 4-grams: this, his_, is_a, s_an, ...

The definition of n-gram [3], given a text (viewed as a character sequence) is given below:

Definition 2.1.1 If $n > 0, n \epsilon \mathbb{Z}$, and c_i is the i-th character of an l-length character sequence T^l = $(c_1, c_2, ..., c_l)$ (our text), then a character **n-gram** $S^n = (s_1, s_2, ..., s_n)$ is a subsequence of length n of $T^l \iff \exists\ i\ \epsilon\ [1, l-n+1] : \forall\ j\ \epsilon [1, n] : s_j = c_{i+j-1}$. We shall indicate the n-gram spanning from c_i to $c_k,\ k > i$, as $S_{i,k}$, while n-grams of length n will be indicated as S^n.

Example 2.1.2 The 4-grams extracted from the string *Extracted n-grams!* are:
S_1 = Extr, S_2 = xtra, S_3 = trac, S_4 = ract, S_5 = acte, S_6 = cted, S_7 = ted_, S_8 = ed_n, S_9 = d_n-, S_{10} = _n-g, S_{11} = n-gr, S_{12} = -gra, S_{13} = gram, S_{14} = rams, S_{15} = ams!.

Definition 2.0.2 Let $\mathbb{Q}$ be an alphabet of symbols, $\mathbb{L}$ the set of all possible n-length strings of symbols from $\mathbb{Q}$ and $\mathbb{S} = \{S_1, S_2, ..., S_{l-n+1}\}$ the set of n-grams extracted from a text T^l, made of symbols from $\mathbb{Q}$. We also define a function sf : $\mathbb{S} \to \mathbb{L}$, which assigns a n-gram a of the set $\mathbb{S}$ to a (unique) string $t = s_1, ..., s_n$ of the set $\mathbb{L}$, if the n-gram a represents the string t, i.e. $a = s_1 s_2 ... s_n$. We will say that a n-gram a is of form of a sting t, if $sf(a) = t$. Furthermore, two distinct n-grams a and b are said to be **of the same form** iff for n-grams a and b we have $sf(a) = sf(b)$.

Example 2.1.3 The 3-grams extracted from the string *Caroline's olive car* are:
$S_1 = Car,\ S_2 = aro,\ S_3 = rol,\ S_4 = oli,\ S_5 = lin,\ S_6 = ine,\ S_7 = ne',\ S_8 = e's,\ S_9 = '\ s\_,\ S_{10} = s\_o,\ S_{11} = \_ol,\ S_{12} = oli,\ S_{13} = liv,\ S_{14} = ive,\ S_{15} = ve\_,\ S_{16} = e\_c,\ S_{17} = \_ca,\ S_{18} = car$.
We see that 3-grams $S_4 = oli$ and $S_{12} = oli$ represent the same string oli, but are found in different positions in the text T^l; according to Definition 4.0.2 we shall call these n-gram to be of the same form. However, 3-grams S_1 = Car and S_{18} = car do not represent the same string; in other words, the n-gram representation we use is *case sensitive*.

The length of a n-gram *n*, is also called the *rank* of the n-gram.

2.2 Related Work

As we mentioned in the introduction, n-grams are proven a useful approach to represent information with various applications, such as text classification and text summarization. By combining n-grams with the powerful concept of the graphs, we have a statical, language-neutral and generic representation: the n-gram graphs. In the following paragraphs of this subsection, we will refer to previous works, which have used the n-gram graphs in a number of research fields.

In [4], n-gram graphs were used in the context of Sentiment Analysis over Social Media. Through their experimental study, the authors verified that it is a very suitable representation model for this task, as it allows substring matching and, second, it is a language-neutral method that makes no assumptions on the underlying languages. Furthermore, this model exhibits high classification efficiency, as well. It involves a limited number of features that solely depends on the corresponding number of classes. An n-gram graph based method was also used at a methodology for sentiment analysis of figurative language, which applies Word Sense Disambiguation. [6] The n-gram graph representation was used to assign polarity to word senses.

In [5], AutoSummENG system was presented, which is a promising novel automatic method for the evaluation of summarization systems, based on comparing the character n-gram graphs representation of the extracted summaries and a number of model summaries. It was found that statistical information related to co-occurrence of character n-grams seems to provide important information concerning the evaluation process of summary systems. The authors concluded the AutoSummENG method has proved to be a language-neutral, fully automated, context-sensitive method with competitive performance. In [7], a real, multi-document, multilingual news summarization application was developed, named NewSum. This system used the representation of n-gram graphs in a novel manner to perform sentence selection and redundancy removal for the summaries.

In the field of bioinformatics, n-gram graph representation has been a very useful method for inquiring genomic sequence composition. In [8], the authors implemented the n-gram graph approach on short vertebrate and invertebrate constrained genomic sequences of various origins and predicted functionalities. In addition, they were able to efficiently distinguish DNA sequences belonging to the same species.

According to the above work, we can conclude that the n-gram graph representation has been a useful tool for the researchers' experiments. Even though n-gram graphs have been shown to be a good representation choice for the practical aspects of the researchers' experiments, n-gram graphs were not analyzed theoretically. In this work we study the class of strings a given n-gram graph represents to better acquire insights related to the possible strengths and weaknesses of the representation.

2.3 Constraint Satisfaction Problems

2.3.1 Formal definition of a constraint satisfaction problem

A constraint satisfaction problem is a search problem, that can be defined formally by four components:

1. The **initial state** that the agent starts in.
2. A description of the possible actions available to the **agent**. The most common formulation uses a **successor function**. Given a particular state x, $SUCCESSOR - FN(x)$ returns a set of (action, successor) ordered pairs, where each action is one of the legal actions in state x and each successor is a state that can be reached from x by applying the action. Together, the initial state and the successor function implicitly define **state space** of the problem - the set of all states reachable from the initial state. A **path** in the state space is a sequence of states connected by a sequence of actions.
3. The **goal test**, which determines whether a given state is a goal state.

4. A **path cost** function that assigns a numeric cost to each path. The **step cost** of taking action a to go from state x to state y is denotes by $c(x, a, y)$.

Below we present the typical definition of constraint satisfaction problems (CSP) as given in [1]:

Definition 2.2.1.1 *Let $X_1,\ X_2,\ ..., X_n$ be a set of variables and $C_1,\ C_2,\ ..., C_m$ be a set of constraints. Each variable X_i has a nonempty domain D_i of possible values. Each constraint C_i involves some subset of the variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, {$X_i = v_i,\ X_j = v_j, ...$}. An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints.*

Example 2.2.1.1 One of the most common CSP problems is the N-Queen problem. For this example we will model the 8-Queen Problem by following the *definition 2.0.1*.
Let $X_1,\ X_2,\ ..., X_8$ be the set of the variables, where n in X_n is the position of the queen on the chessboard. The domain {$0\ ...\ n^2 - 1$} or in our case {$0\ ...\ 63$} includes all the possible values a variable can have. Given $R_1,\ R_2$ and $C_1,\ C_2$ of two queens' positions the following constraints must be satisfied:

1. One queen per row: $R_1 \neq R_2$
2. One queen per column: $C_1 \neq C_2$
3. One queen per diagonal: $R_1 - R_2\ \neq C_1 - C_2$ and $R_1 - R_2\ \neq C_2 - C_1$

2.3.2 Local Search For Constraint Satisfaction Problems

According to [1], local-search algorithms turn out to be very effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the successor function usually works by iteratively changing this assignment by improving steps, by taking random steps, or by restarting with another complete assignment [9]. A wide variety of local search techniques has been proposed. Understanding when these techniques work for different problems forms the focus of a number of research communities, including those from both operations research and AI.

The generic form of Local Search algorithm is given below:

Algorithm 1 LOCAL SEARCH

```
1: procedure LOCALSEARCH(V, dom, C)
2:     Inputs
3:         V: a set of variables
4:         dom: a function such that dom(X) is the domain of variable X
5:         C: set of constraints to be satisfied
6:     Outputs
7:         complete assignment that satisfies the constraints
8:     Local
9:         A[V] an array of values indexed by V
10:    repeat
11:    for each variable X do
12:        A[X] ← a random value in dom(X);
13:    end for
14:    while (stopping criterion not met &A is not a satisfying assignment) do
15:        Select a variable Y and a value VϵDom(Y)
16:        Set A[Y] ← V
17:    end while
18:    if (A is a satisfying assignment) then
19:        return A
20:    end if
21:    until termination
22: end procedure
```

A specifies an assignment of a value to each variable. The first $for\ each$ loop assigns a random value to each variable. The first time it is executed is called a **random initialization**. Each iteration of the outer loop is called a **try**. A common way to implement a new try is to do a **random restart**. An alternative to random initialization is to use a construction heuristic that guesses a solution, which is then iteratively improved.

The while loop does a **local search**, or a **walk**, through the assignment space. It maintains a current assignment S, considers a set of **neighbors** of the current assignment, and selects one to be the next current assignment. In the above algorithm, the neighbors of a total assignment are those assignments that differ in the assignment of a single variable. Alternate sets of neighbors can also be used and will result in different search algorithms.

This walk through assignments continues until either a satisfying assignment is found and returned or some stopping criterion is satisfied. The stopping criterion is used to decide when to stop the current local search and do a random restart, starting again with a new assignment. A stopping criterion could be as simple as stopping after a certain number of steps.

This algorithm is not guaranteed to halt. In particular, it goes on forever if there is no solution, and it is possible to get trapped in some region of the search space. An algorithm is **complete** if it finds an answer whenever there is one. This algorithm is incomplete.

One instance of this algorithm is **random sampling**. In this algorithm, the stopping criterion is always true so that the while loop is never executed. Random sampling keeps picking random assignments until it finds one that satisfies the constraints, and otherwise it does not halt. Random sampling is complete in the sense that, given enough time, it guarantees that a solution will be found if one exists, but there is no upper bound on the time it may take. It is very slow. The efficiency depends only on the product of the domain sizes and how many solutions exist.

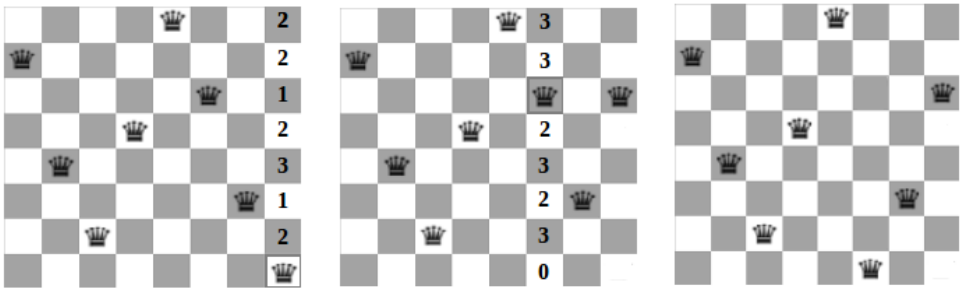
Another instance is a **random walk**. In this algorithm the while loop is only exited when it has found a satisfying assignment (i.e., the stopping criterion is always false and there are no random restarts). In the while loop it selects a variable and a value at random. Random walk is also complete in the same sense as random sampling. Each step takes less time than resampling all variables, but it can take more steps than random sampling, depending on how the solutions are distributed. Variants of this algorithm are applicable when the domain sizes of the variables differ; a random walk algorithm can either select a variable at random and then a value at random, or select a variable-value pair at random. The latter is more likely to select a variable when it has a larger domain.

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables - the **min-conflicts** heuristic. The algorithm is given below:

Algorithm 2 MIN-CONFLICTS

```
1:  procedure MIN-CONFLICTS(csp: a constraint satisfaction problem, max-steps: the number
    of steps allowed before giving up)
2:      current ← an initial complete assignment for csp
3:      for i = 1 to max-steps do
4:          if current is solution for csp then then return current
5:              var ← a randomly chosen, conflicted variable from VARIABLES[csp]
6:              value ← the value,v for var that minimizes CONFLICTS(var, v, current, csp)
7:              set var = value in current
8:          end if
9:      end for
10:     return failure
11: end procedure
```

Example 2.2.2.1 Min-Conflicts solves N-Queens Problem, which is described in *example 2.2.1.1*, by randomly selecting a column from the chess board for queen reassignment. The algorithm searches each potential move for the number of conflicts (number of attacking queens), shown in each square. The algorithm moves the queen to the square with the minimum number of conflicts, breaking ties randomly. Below, the 8-Queens Problem is solved in 2 moves. Note that the number in each tie is the number of conflicts a queen will have if we move it on it.



Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems. For example, a week's airline schedule may involve thousands of flights, but bad weather at one airport can render the schedule infeasible. We would be able to repair the schedule by minimizing the number of

changes. This can be easily done with a local search algorithm starting from the current schedule.

2.3.3 Depth First Search

Depth First Search (DFS) is an algorithm for traversing or searching tree or graph data structures. In order to apply the algorithm to a graph, we select a random node of the graph to be the root and we expand the deepest expanded vertex. If a dead end is reached, the algorithm backtracks to expand the vertices of the shallower levels. To be able to backtrack, we need to keep track of the vertices that have already been visited, by using a list for example. Depth first graph search has a recursive and a non-recursive implementation, where a stack data structure is demanded. These two implementations are given below:

Algorithm 3 DFSNR - Depth First Search Non Recursive

INPUT: A graph G and the vertex v from where the algorithm is started

```
1:  procedure DFSNR(G, v)
2:      Stack S is empty
3:      List visitedList is empty
4:      S.push(v)
5:      while S is not empty do
6:          u = S.pop()
7:          if u is not in visitedList then
8:              visitedList.add(u)
9:              for each neighbor w of u do
10:                 if w is not in visitedList then
11:                     S.push(w)
12:                 end if
13:             end for
14:         end if
15:     end while
16: end procedure
```

Algorithm 4 DFSR - Depth First Search Recursive

```
1:  procedure DFSR(G, v)
2:      List visitedList is empty
3:      visitedList.add(v)
4:      for each neighbor w of v do
5:          if w is not in visitedList then
6:              DFSR(G, w)
7:          end if
8:      end for
9:  end procedure
```

2.3.4 Depth First Search and Constraints

A generic form of Depth First Search algorithm for a constraint satisfaction problem, which contains a graph, is given below:

Algorithm 5 DFSC - Depth First Search and Constraints

INPUT: V: the set of variables from the graph
dom: a function such that $dom(x)$ is the domain of variable x
C: set of constraints to be satisfied
G: the graph
OUTPUT: The complete assignment that satisfies the constraints, in the $result$ list, or an empty list, otherwise
LOCAL: $result$ is a list that contains the assignment

```
1: procedure DFSC(V, dom, C, G)
2:     result is empty
3:     v ← a random variable from V, given by the user
4:         v indicates the starting vertex of the search
5:     DFSCmain(G, result, v, dom)
6:     if (result is not a satisfying assignment according to C) then
7:         empty result
8:     end if
9:     return result
10: end procedure
```

Algorithm 6 DFSCmain - The recursive procedure

```
1: procedure DFSCmain(G, result, v, dom)
2:     RV ← a random value in dom(v)
3:     result.add(RV)
4:     label v as visited
5:     for each neighbor w of v do
6:         if vertex w is not labeled as visited then
7:             DFSCmain(G, result, w, dom)
8:         end if
9:     end for
10: end procedure
```

2.3.5 Breadth First Search

Breadth First Search (BFS) is another algorithm for traversing or searching tree or graph data structures. It starts from a selected node or vertex and it explores the neighbor nodes first, before going to the lowest level neighbors. It is similar to the DFS algorithm, that was explained in the precious Section, but a queue data structure is replacing the stack data structure. A non-recursive implementation of the BFS algorithm is given below:

Algorithm 7 BFSNR - Breadth First Search Non Recursive

INPUT: A graph G and the vertex v from where the algorithm is started

```
1:  procedure BFSNR(G, v)
2:      Queue Q is empty
3:      List visitedList is empty
4:      Q.add(v)
5:      while Q is not empty do
6:          u = Q.remove()
7:          if u is not in visitedList then
8:              visitedList.add(u)
9:              for each neighbor w of u do
10:                 if w is not in visitedList then
11:                     Q.add(w)
12:                 end if
13:             end for
14:         end if
15:     end while
16: end procedure
```

2.3.6 Breadth First Search and Constraints

A generic form of Breadth First Search algorithm for a constraint satisfaction problem, which contains a graph, is given below:

Algorithm 8 BFSC - Breadth First Search and Constraints

INPUT: V: the set of variables from the graph
dom: a function such that $dom(x)$ is the domain of variable x
C: set of constraints to be satisfied
G: the graph
OUTPUT: The complete assignment that satisfies the constraints, in the $result$ list, or an empty list, otherwise
LOCAL: $result$ is a list that contains the assignment

```
1:  procedure BFSC(V, dom, C, G)
2:      result is empty
3:      repeat
4:      v ← a random variable from V, given by the user
5:          v indicates the starting vertex of the search
6:      BFSCmain(G, result, v, dom)
7:      if (result is not a satisfying assignment according to C) then
8:          empty result
9:      end if
10:     return result
11: end procedure
```

Algorithm 9 BFSCmain - The BFS procedure

```
1:  procedure BFSCmain(G, result, v, dom)
2:      List visitedList is empty
3:      Queue Q is empty
4:      Q.add(v)
5:      while Q is not empty do
6:          u = Q.remove()
7:          if u is not in visitedList then
8:              visitedList.add(u)
9:              RV ← a random value in dom(v)
10:             result.add(RV)
11:             for each neighbor w of u do
12:                 if w is not in visitedList then
13:                     Q.add(w)
14:                 end if
15:             end for
16:         end if
17:     end while
18: end procedure
```

2.3.7 Heuristics for Backtracking Algorithms

According to [10], when solving a constraint satisfaction problem using backtracking search, we have to decide which variable will be initiated next or which value will be given to a specific variable. These two decisions are referred to as the variable ordering and the value ordering correspondingly. A great amount of experiments, that were performed during the past decades, show that for many problems, the choice of the next variable or value can be crucial to effectively solving the problem.

Given a CSP [10] and a backtracking search algorithm, a variable ordering or value ordering is said to be optimal if the search visits the fewer number of nodes over all possible orderings when finding a solution, or showing that a solution does not exist. In this thesis, the DFS and BFS backtracking algorithms are implemented to solve the CSP simulation of the decompression problem and we will try to design two variable ordering heuristics. According to [13], the decompression problem has at least one solution. As a result, our goal is to determine whether the application of some heuristics can provide optimal orderings, even if the field of constraint programming has so far mainly focused on heuristics which have no formal guarantees [10].

Suppose that the backtracking search is attempting to extent a node n. The task of the variable ordering heuristic is to decide the next variable v to be branched to [10]. Many variable ordering heuristics have been proposed and evaluated in the literature. These heuristics can be divided in two categories: heuristics that are based on the structure of the CSP and heuristics that are based on the domain size of the variables. The two heuristics that will be presented in this thesis depend on the structure of the decompression problem and will be analyzed in chapter 3, as firstly, we need to define the CSP simulation of the decompression problem.

3 PROPOSED APPROACH

The n-gram graph decompression can be formulated as a search problem. Our purpose is to try and find a sequence of steps that will provide the initial text, that was compressed in the n-gram graph. To achieve our goal, we will model the search problem in the subsection 3.1. In addition, we will define the CSP version of the decompression problem in the subsection 3.2. As promised in the previous chapter, the two new heuristics that will be added to the decompression problem will be explained in the subsection 3.3, so that we can perform the experimental evaluation in the chapter 4.

3.1 Modeling And Solving Search Problems

In this paper, search problems are implemented according to the following assumptions:

1. Every search problem is expressed as a **Problem**. The Problem will try to find a solution by using a tree structure - the **Problem Tree**, and the given search algorithm - the **Search Algorithm**. Each Problem Tree is constructed by **Problem Tree Nodes**, that represent the possible states of our search problem.
2. The Problem needs to decide whether a Problem Tree Node is a solution of the search problem. In addition, it checks whether a problem state is valid and it estimates the distance from a solution. Last but not least, the Problem can get the next states of the problem, based on the current state. There are no other methods the Problem implements, except from the above four.
3. The Search Algorithm accepts a Problem as a parameter and tries to find a solution, by executing the algorithm.

3.2 N-Gram Graph Decompression Problem as a Constraint Satisfaction Problem

Let T^l be an l-length (in characters) initial text that can generate the n-gram graph. The T was divided in l uni-grams. Let $G = \{V^G, E^G, L, W\}$ be a n-gram graph, where V^G is the set of the variables, L is a function that labels each vertex of the graph and W is a function that declares the weight of each edge. Each vertex represents a n-gram of the initial text T. Given the graph G we search for all possible texts $\mathbb{T}$, that can generate the n-gram graph. We expect that $T \in \mathbb{T}$. We say that all texts in $\mathbb{T}$ are solutions to our problem and we represent such a solution as $S =< n_1, n_2, ..., n_l >$.

We claim that the problem of finding $\mathbb{T}$ can be modeled as a constraint satisfaction problem, as follows.

Variables The variables of the problem are the letters n_i of S.

Domain The non-empty domain of each variable is the set of the labels of the unigram graph vertices $L(V)$.

Constraints The set of constraints is the following:

1. Every label $l \in L(V)$ should appear at least once in S.
2. Every label $l \in L(V)$ should appear at most a number of times equal to the *weighted degree* of the vertex v that is labeled by l. We define as *weighted degree* of a vertex v the sum of the weights of the edges where v appears.

3. Within a distance $D_{win} = 1$ of a n-gram there are at most D_{win} neighboring n-grams.
4. For every two neighboring n-grams n_i, n_j of the text S, there is a directed edge $e_i = \{v_x, v_y\}$, where $1 \leq i, j, x, y \leq l$. The amount of times an n-gram n_i is the neighbor of the n-gram n_j is the weight w_i of the edge e_i.
5. There are no other vertices and edges in the graph G but those that are described in the constraints (i) and (ii).

Our goal is the initialization of all the variables while all the above constraints are satisfied.

3.3 Heuristics for the n-gram graph decompression CSP

As we promised in Section 2, the two heuristics that will be presented depend on the structure of the n-gram graph decompression problem. The task of the first heuristic we will define depends on the weighted degree of the vertices of the n-gram graph. The variable we will be branched to is the one with the minimum weighted degree. As we mentioned in the Introduction [1], we compress text that does not include any duplicate characters. By choosing the variable with the minimum weight, we try to maximize the chance to select the correct first variable of our solution. If we succeed, we will reduce the amount of times we backtrack while performing DFS and BFS.

The second heuristic we will present is a probabilistic one. We will calculate a factor θ for every vertex of our n-gram graph, which will be explained below, by taking into account the weighted degree of the particular vertex. So far, the two heuristics depend on the weighted degree of each vertex of the n-gram graph. By applying the second heuristic to our backtracking search method, we will have to generate a random number per variable, between 0 and 1. This number will be the probability of each variable. We will branch on the variable that has its probability number closest to θ. By defining this heuristic, we try to decompress our text by combining the knowledge of the structure of our problem and the recognized optimization randomized search heuristics can provide [14].

The difference of the two heuristics is that by choosing the first one, we will rely on the structure of our problem completely, while choosing the second heuristic will include a randomization factor to the decompression procedure. In the subsection 3.2, we defined the n-gram graph decompression problem as a constraint satisfaction problem, modeled as a search problem, so now we can explain the new heuristics.

3.3.1 Weighted Degree Variable Ordering Heuristic

The first heuristic we will define depends on the weighted degree of the vertices of the n-gram graph. Our goal is to sort the variables of the decompression CSP based on the weighted degree each vertex has.

Example 3.3.1 Assuming we have the n-gram graph G, which is initialized by the string $S : queen$. According to subsection 3.2, the variables V^G of the problem are the letters of the S, so:

$V = \{q, u, e, n\}$

the weighted degree wd of these vertices is:

$wd = \{q = 1, u = 2, e = 3, n = 1\}$

which if seen as sorted as:

$wd = \{q = 1, n = 1, u = 2, e = 3\}$

By adding this heuristic to our decompression problem, the vertices will be ordered. The next variable that will be chosen to be branched on will be the one with the minimum weighted degree. When a variable is selected, its weighted degree is reduced by one. If we backtrack while performing the search, the changes that were applied to the variables that are no longer a part of the current solution will have their weighted degree increased.

3.3.2 Probabilistic Variable Ordering Heuristic

The second heuristic we will define is a probabilistic one and will be defined below. Firstly we will describe the basic concepts of the probability theory.

Definition 3.3.1 Probability is the measure of the likelihood that an event will occur.[11] The modern definition starts with a finite or countable set called the sample space, which relates to the set of all possible outcomes in classical sense, denoted by Ω. It is then assumed that for each element $x \in \Omega$, a "probability" value $f(x)$, is attached, which satisfies the following properties:

1. $f(x) \in [0, 1]$ for all $x \in \Omega$
2. $\sum_{x \in \Omega} f(x) = 1$

That is, the probability function $f(x)$ lies between zero and one for every value of x in the sample space Ω, and the sum of $f(x)$ over all values x in the sample space Ω is equal to 1. An event is defined as any subset E of the sample space Ω. The probability of the event E, is defined as:

$P(E) = \sum_{x \in E} f(x)$

In our case every vertex v of the n-gram graph G is an event V and will have a probability $P(V)$. According to the definition $3.3.1$, the probability value $f(v)$ will satisfy the above two properties. We define the probability $P(V)$ as:

(3.3.1): $P(V) = \frac{1}{w(V)}\theta$

where θ is a factor which will be calculated according to the second equation of the 3.3.1 definition, and wd is the weighted degree of the particular vertex of the n-gram graph. For the set of the n-gram graph vertices V^G, we have:

(3.3.2): $\sum_{v \in V^G} P(v) = 1 \iff \sum_{v \in V^G} \frac{1}{wd(v)}\theta = 1 \iff \theta = \frac{1}{\sum_{v \in V^G} \frac{1}{wd(v)}}$

After calculating the factor θ, we will generate one random number per variable, between 0 and 1. The next variable that will be chosen to be branched on will be the one that has its probability

number closest to θ. When a variable is selected, the weighted degree of it is reduced by one, as we did on the first heuristic in the previous Section. If we backtrack while performing the search, the changes that were applied to the variables that are no longer a part of the current solution will have their weighted degree increased.

Example 3.3.2 Assuming we have the n-gram graph G, which is created by the string $S : red$. According to Section 3.2, the variables V^G of the problem are the letters of the S, so:

$V = \{r, e, d\}$

the weighted degree wd of these vertices is:

$wd = \{r = 1, e = 2, d = 1\}$

θ will be calculated according to equation 3.3.2:

$\theta = \frac{1}{\sum_{v \in V^G} \frac{1}{wd(v)}} \iff \theta = \frac{1}{\frac{1}{1}+\frac{1}{2}+\frac{1}{1}} \iff \theta = 0.4$

Then, we generate a probability for each variable of the CSP problem, so:

$P(V_1) = 0.05$, $P(V_2) = 0.23$, and $P(V_3) = 0.68$

These probabilities will be sorted, and we will have the following order:

$sortedProbabilities = \{P(V_2), P(V_3), P(V_1)\}.$

According to the probabilistic heuristic, the next variable that will be chosen to be branched on will be the V_2. The weighted degree of it will be reduced by one.

3.4 Amount of solutions for a single N-Gram Graph

According to [13], every n-gram graph G that was constructed from a text T^l has at least one solution. In graph theory, an arborescence is a directed graph in which, for a vertex q called the root and any other vertex v, there is exactly one directed path from q to v. [] Assuming $m(\{v, q\})$ is the degree of multiple edges of an ordered pair of vertices $\{v, q\}$, $AR(G)$ is the set of arborescences of G, D_{win} is equal to 1, and the outdegree of a vertex v is denoted as $d_{out}(v)$. The following proposition is provided by the author F. Monachou [13]:

Proposition 3.4.1 The number of solutions to a n-gram graph $G = (V, E)$ that was constructed from text T^l is equal to

$$S(G) = |AR(G)| \cdot \frac{\prod^{\forall v \in V}(d_{out}(v)-1)!}{\prod^{\forall \{v,q\}, v,q \in V} m(\{v,q\})!}$$

In the following example we will show a case where an n-gram graph G can have more than one solution.

Example 3.4.1 Let text T_1 be the text we get by decompressing the following n-gram graph $G = \{V^G, E^G, L, W\}$.

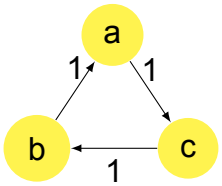


Figure 1: G_1 compressed n-gram graph for the text T_1 of the 3.4.1 example

T_1 is constructed by the uni-grams $\{a, b, c\}$. In addition, we have the set of edges $E = \{\{b, a\}, \{c, b\}, \{a, c\}\}$. Each of the three edges has weight equal to 1, so the weighted degree of each vertex equals to 2. By satisfying the constraints we defined for the n-gram graph decompression problem(Section 3.2), we decompress the G and we can get the text $T_1 : abca$. The same graph G can also be decompressed to the text $T_1 : bcab$.
Hence, we can conclude that the same n-gram graph can have more than one solution. In this work, we use text that does not include any duplicate characters. As a result, we expect to find at least one solution for every n-gram graph that is decompressed.

4 THEORETICAL STUDY

In chapter 2, it was explained how Local Search, Depth First Search and Breadth First Search can solve a CSP. In this chapter we will try to analyze the theoretical aspects each search method has.

4.1 Local Search

1. Can this search method find all the solutions of a CSP?
   According to [9], Local Search does not systematically search the whole search space but it is designed to find solutions quickly on average. It does not guarantee that a solution will be found even if one exists, and so it is not able to prove that no solution exists. It is often the method of choice for applications where solutions are known to exist or are very likely to exist. This algorithm is incomplete, as it is not guaranteed to halt. In particular, it goes on forever if there is no solution, and it is possible to get trapped in some region of the search space.

2. Algorithmic aspect of this search method
   Local Search algorithm operates using a single current state and generally moves only to neighbors of that state.

4.2 Depth First Search

1. Can this search method find all the solutions of a CSP?
   Depth-first search can get trapped on infinite branches and never find a solution, even if one exists, for infinite graphs or for graphs with loops. The implementation of the depth-first search in this thesis is based on the construction of a finite tree and we expect that all the solutions of the CSP will be found.

2. Algorithmic aspect of this search method
   For Depth-first search [9], the space used is linear to the depth of the path length from the start to a node; it considers only those elements on the path, along with their siblings. It is already mentioned that, depth-first search can get trapped on infinite branches and never find a solution, even if one exists, for infinite graphs or for graphs with loops. In this case, the time worst-case complexity is infinite. If the graph is a finite tree, with the forward branching factor bounded by **b** and depth **n**, the time worst-case complexity is $O(b^n)$ and the space complexity is $O(bn)$, as we need to store only the bm nodes of the solution path.

4.3 Breadth First Search

1. Can this search method find all the solutions of a CSP?
   Breadth-first search is a complete search algorithm, thus it will find all the solutions of a CSP. Contrary to depth-first search, which may get trapped in case of an infinity graph input, breadth-first search will eventually find the goal states.

2. Algorithmic aspect of this search method
   Let's assume that the graph is a finite tree, with the forward branching factor bounded by **b** and depth **n**. For a search to be completed, the maximum number of nodes that need to be expanded is b^n, thus, the space complexity is $O(b^n)$. The time complexity [9] is the same as the space complexity, because all the leaf nodes need to be stored in the memory at the same time.

5 EXPERIMENTAL SETTING AND EVALUATION

This Section presents the experimental results and comparisons on a number of synthetic datasets. All experiments are conducted on a processor at 2.3 GHz with 4 GB memory. For the datagrams, we used the boxplot feature of the language R[2]. Firstly, the Local Search for CSP and the simple form of the DFS and the BFS algorithms are compared. The next comparisons are between the DFS and the BFS algorithms for CSP. Last but not least, we include the comparisons between the DFS and the BFS algorithms for CSP, by applying two different heuristics on each search method to maximize the efficiency of the results.

The experiments are conducted so that we can compare the time measurements and the method cost measurements of the search algorithms we implemented to solve the n-gram graph decompression problem. After getting a string as an input, we compress it into an n-gram graph, by using the toolkit JINSECT. We ask the user to choose the search method we will execute in order to decompress the n-gram graph, and we get a file as an output, which includes the decompressed string, the execution time, the solutions we found and the method cost. After collecting the output files of the different search algorithms we executed, we use the language R to compare the results and to decide the search method that gives us the best results.

We chose the boxplot tool to express the results of our experiments, as it is a standard technique for presenting a summary of the distribution of a dataset. [15] The typical construction of the box plot partitions a data distribution into quartiles, that is, four subsets with equal size. A box is used to indicate the positions of the upper and lower quartiles; the interior of this box indicates the inner quartile range, which is the area between the upper and lower quartiles and consists of 50% of the distribution. Lines (sometimes referred to as whiskers) are extended to the minimum and maximum values in the dataset. Often, outliers are represented individually by symbols. Last but not least, the box is intersected by a crossbar drawn at the median of the dataset.

5.1 Datasets

In the main source file, the defined String $CHARS$ contains the following 100 characters:

$$0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz!@\#$$

Each time a search is executed, the user is asked to provide:

1. the number of the strings that will be generated
2. the length of each string
3. the probability of repetition for the current set of strings that will be produced.

Furthermore, the user can choose to load their own datasets, by providing the name of their file, at the beginning of the execution of the program.

In this thesis, the probability for all the following tests is 0%. As a result, all the characters each generated string will have cannot be repeated. In addition, we have an upper limit for the amount of executions while performing the backtracking algorithms. This limit is the 100 million attempts to check if the string we have constructed by that time is the solution of the algorithm. For comparability reasons, the datasets that are used for all the following experiments are the same.

[2]We used the version 3.0.2 of R. It can be downloaded from: https://cran.r-project.org/mirrors.html

5.2 Time measurements

In this Section, we present the time measurements of the software mentioned in the introduction of this chapter. We executed our search algorithms in order to decompress character strings, that are consisted of four, five, six and seven different characters. Each test file includes 1000 different strings.

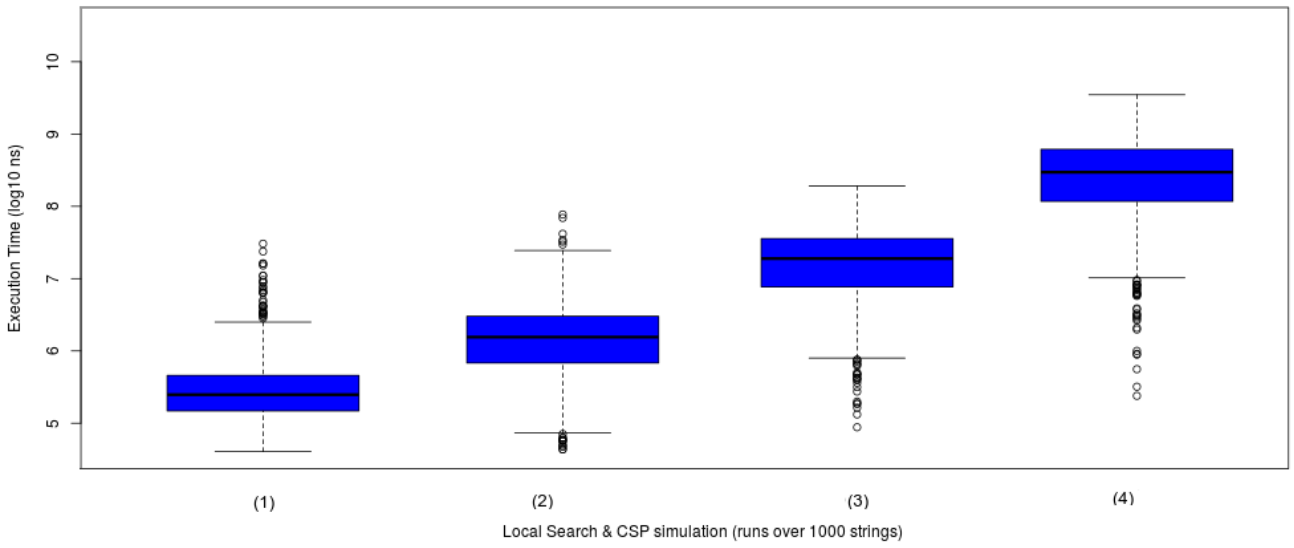


Figure 2: Time: Local Search for the decompression CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

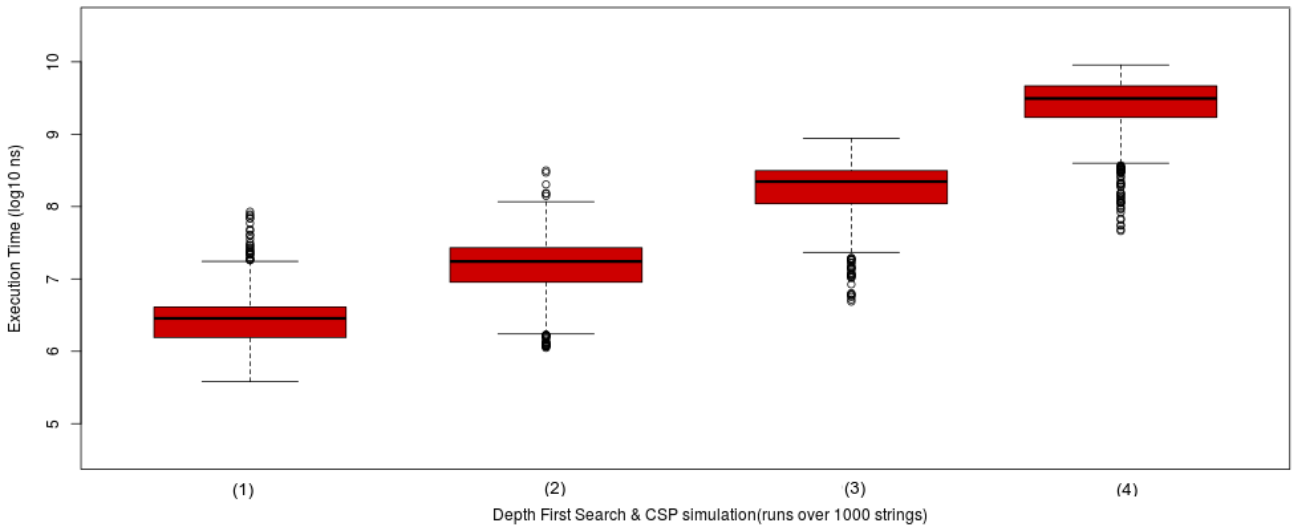


Figure 3: Time: DFS for the decompression CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

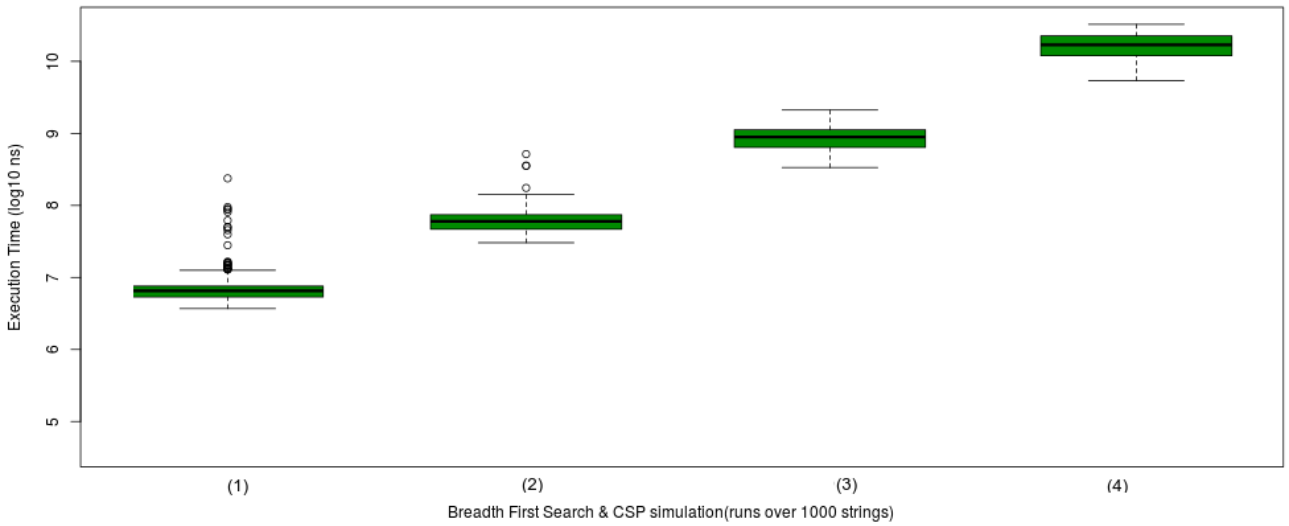


Figure 4: Time: BFS for the decompression CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

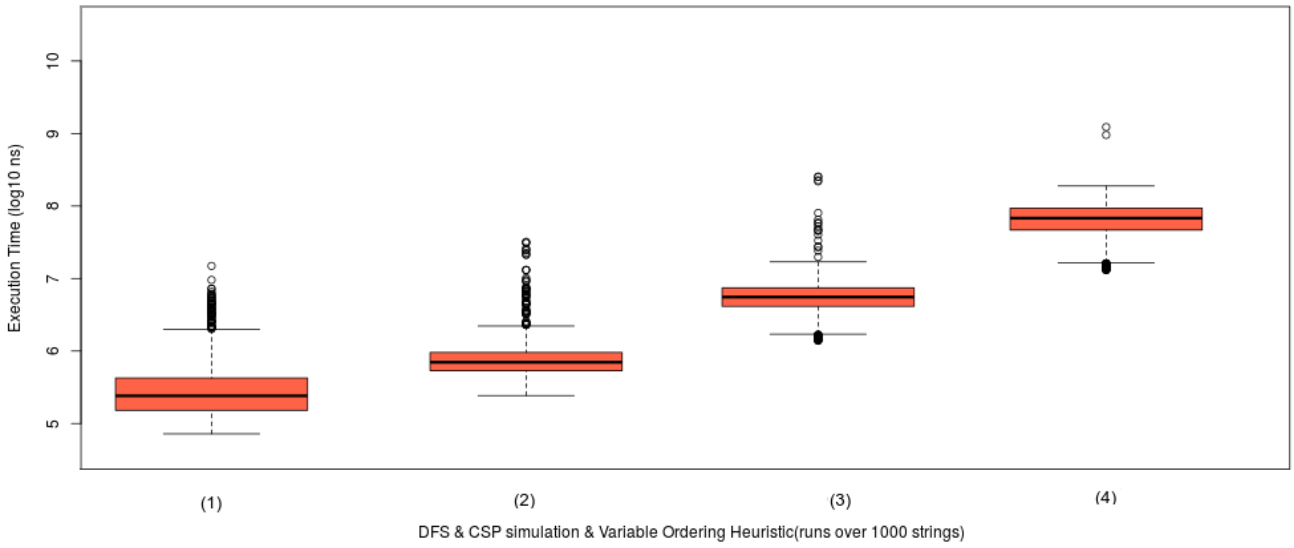


Figure 5: Time: The weighted degree heuristic with DFS for the CSP - Multiple string lengths.
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

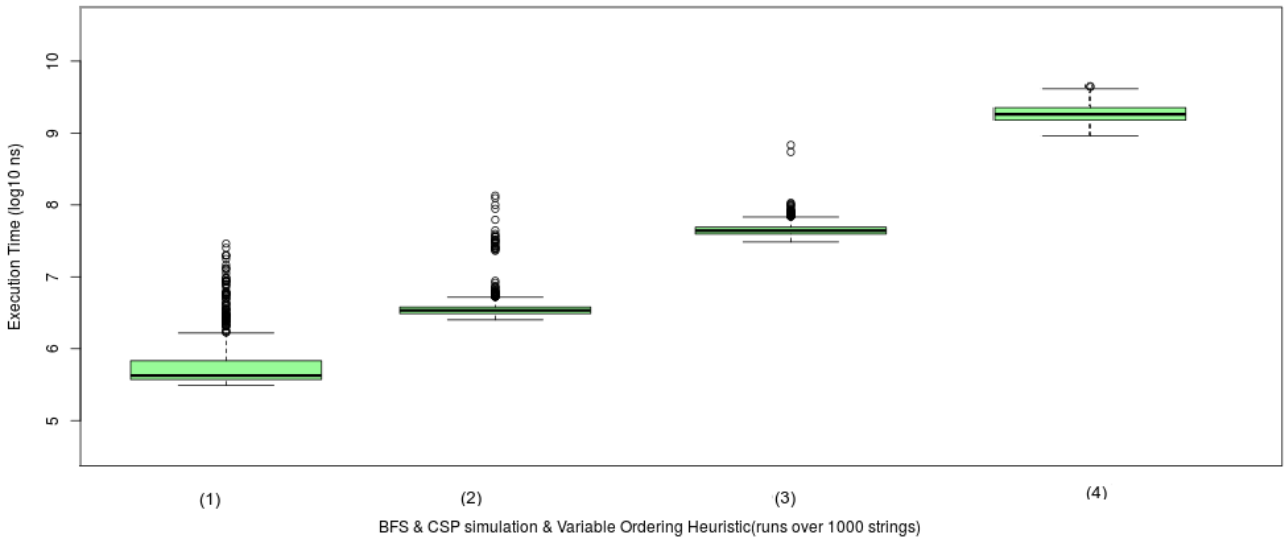


Figure 6: Time: The weighted degree heuristic with BFS for the CSP - Multiple string lengths The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

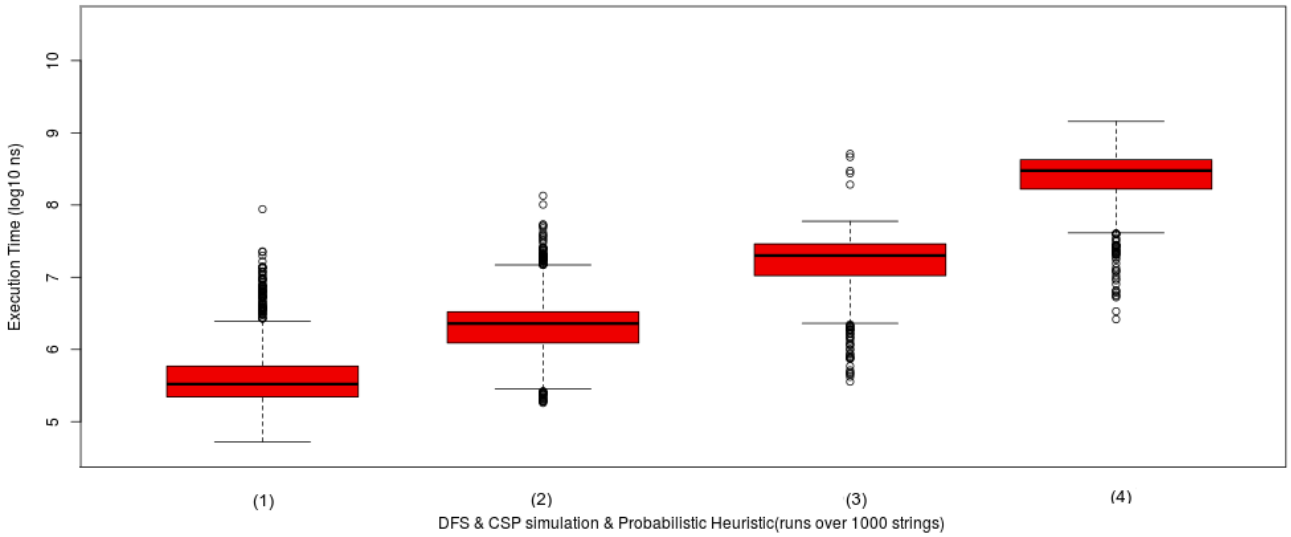


Figure 7: Time: The probabilistic heuristic with DFS for the CSP - Multiple string lengths The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

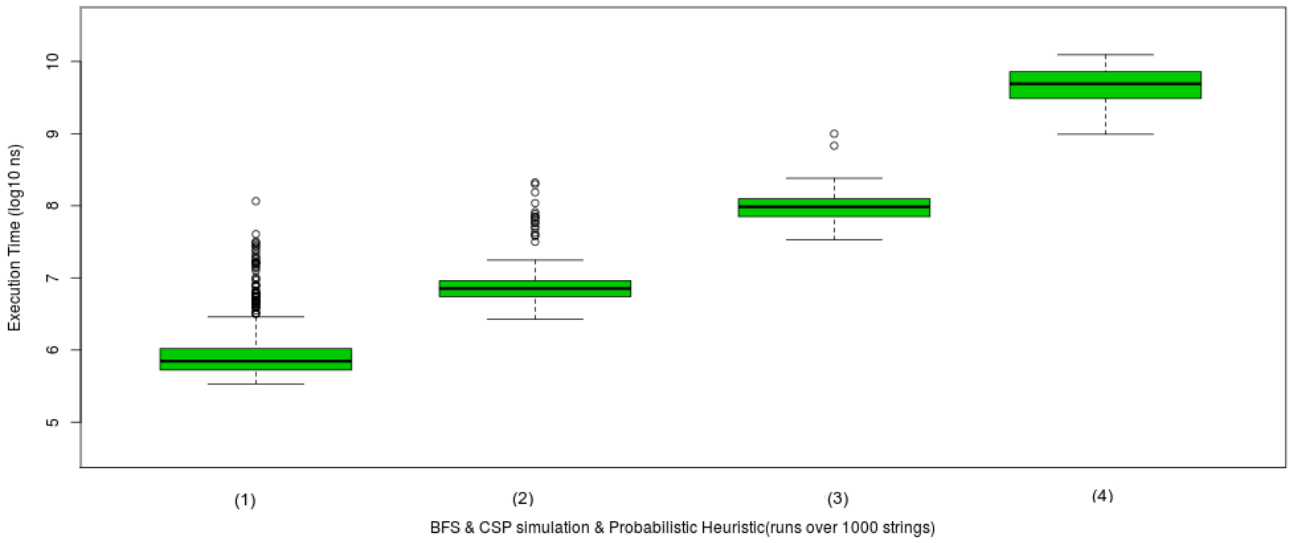


Figure 8: Time: The probabilistic heuristic with BFS for the CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

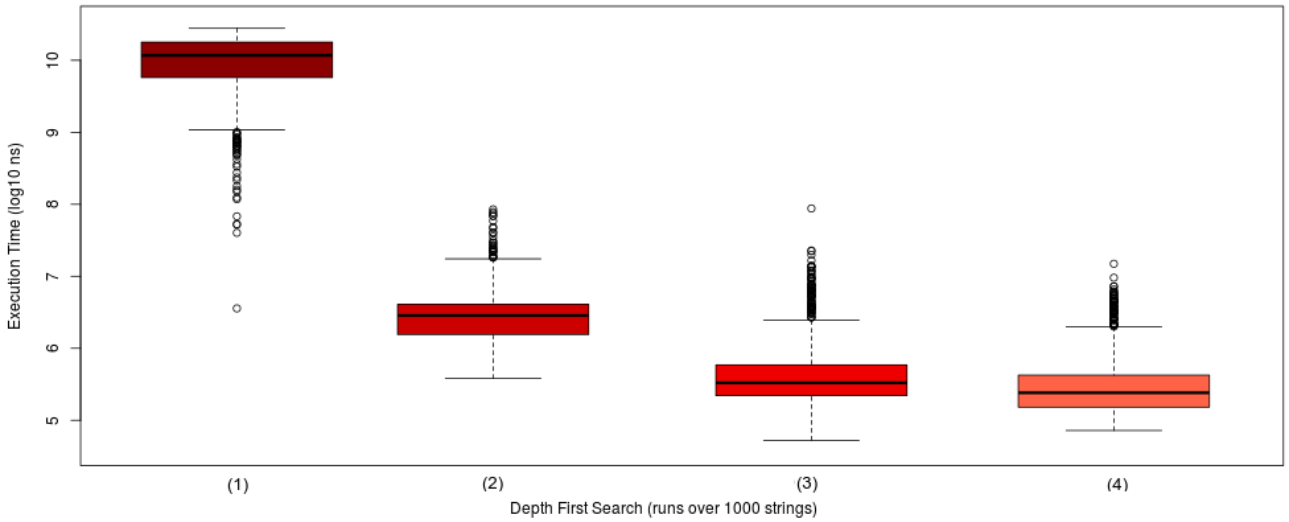


Figure 9: Time: DFS for the decompression problem for 4 character length strings
(1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

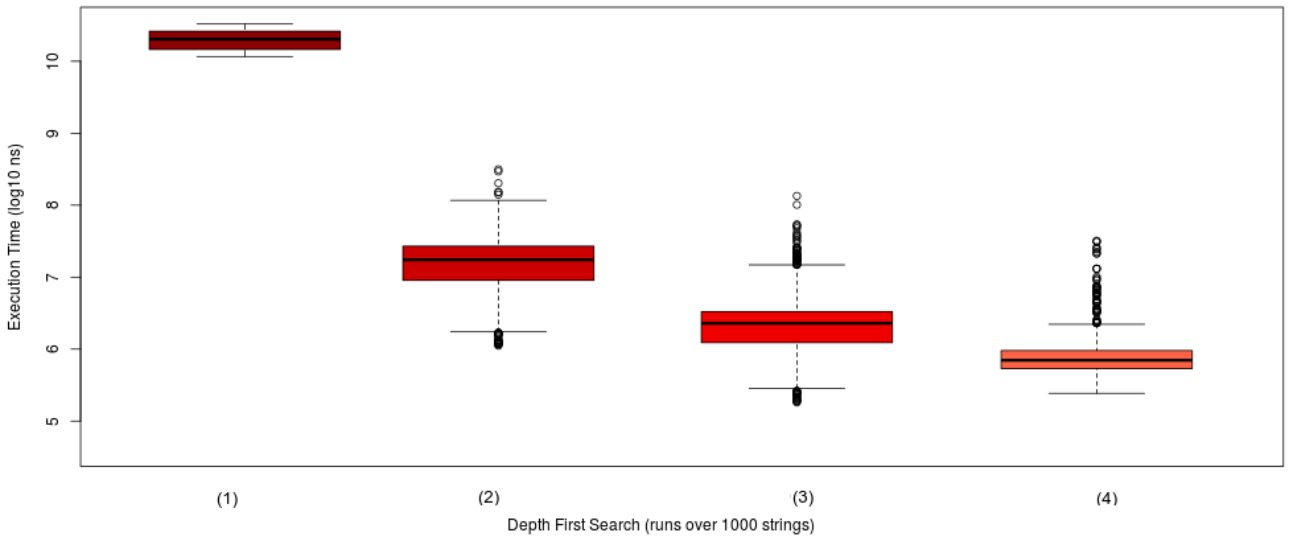


Figure 10: Time: DFS for the decompression problem for 5 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

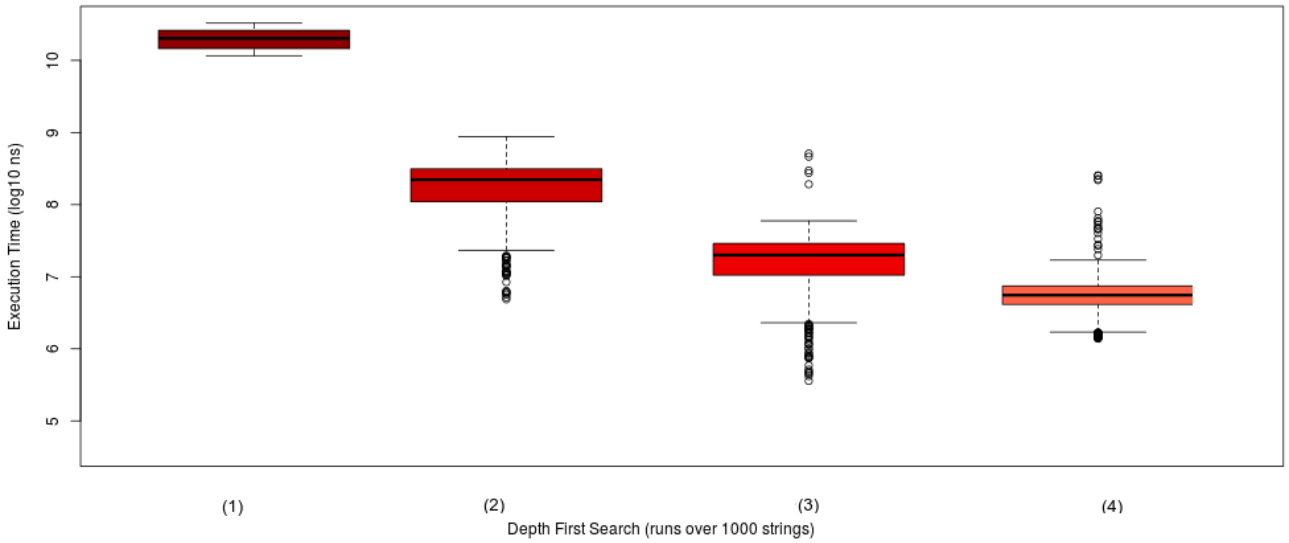


Figure 11: Time: DFS for the decompression problem for 6 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

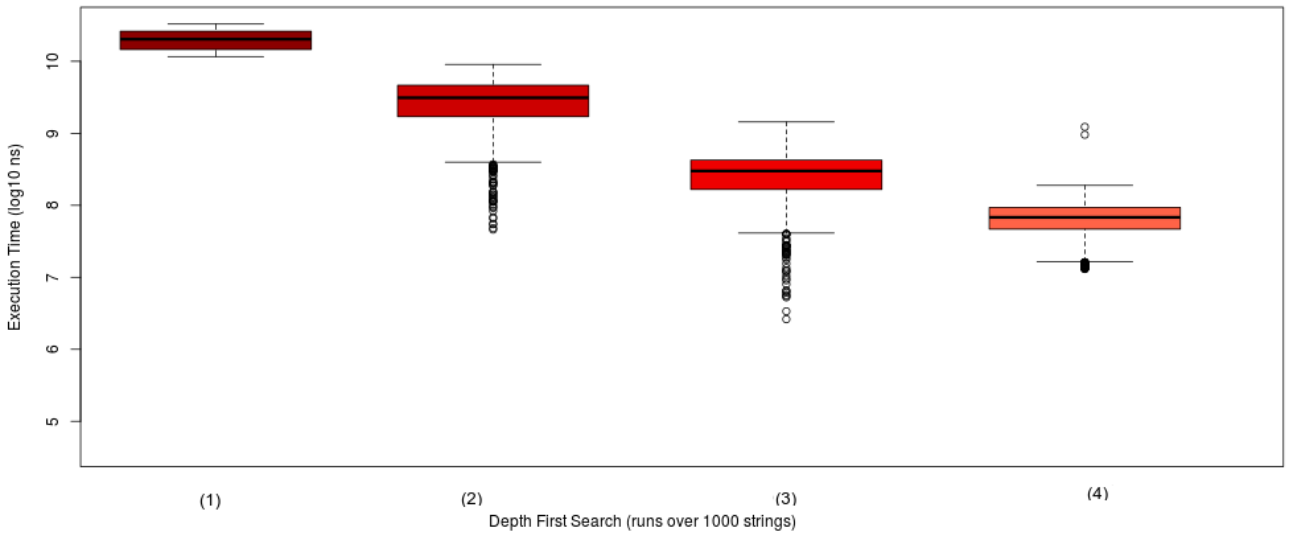


Figure 12: Time: DFS for the decompression problem for 7 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

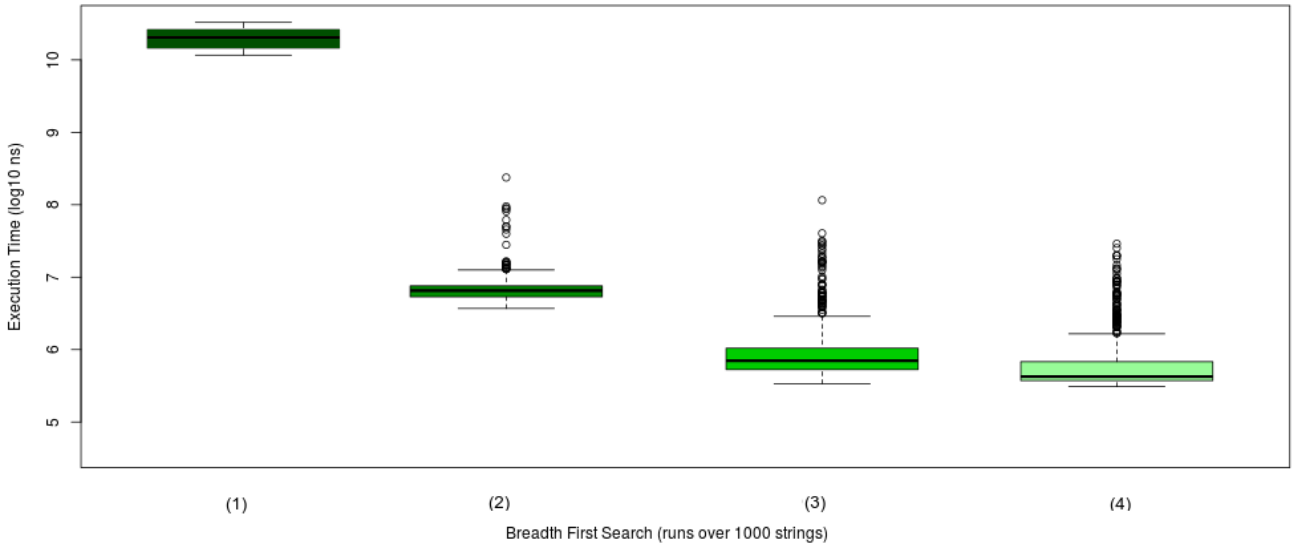


Figure 13: Time: BFS for the decompression problem for 4 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

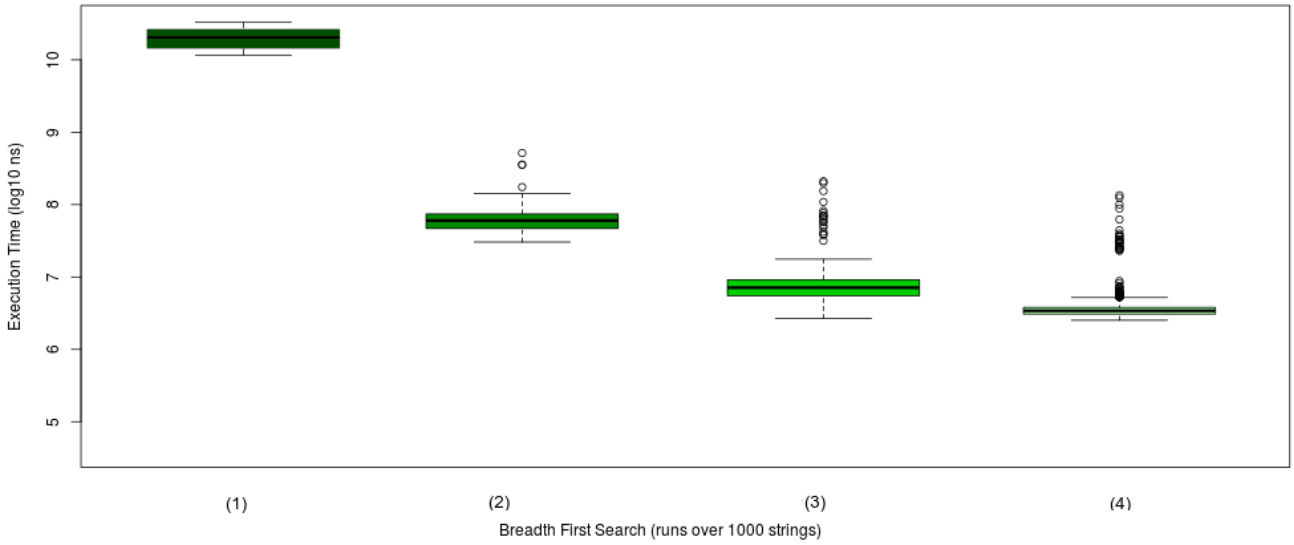


Figure 14: Time: BFS for the decompression problem for 5 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

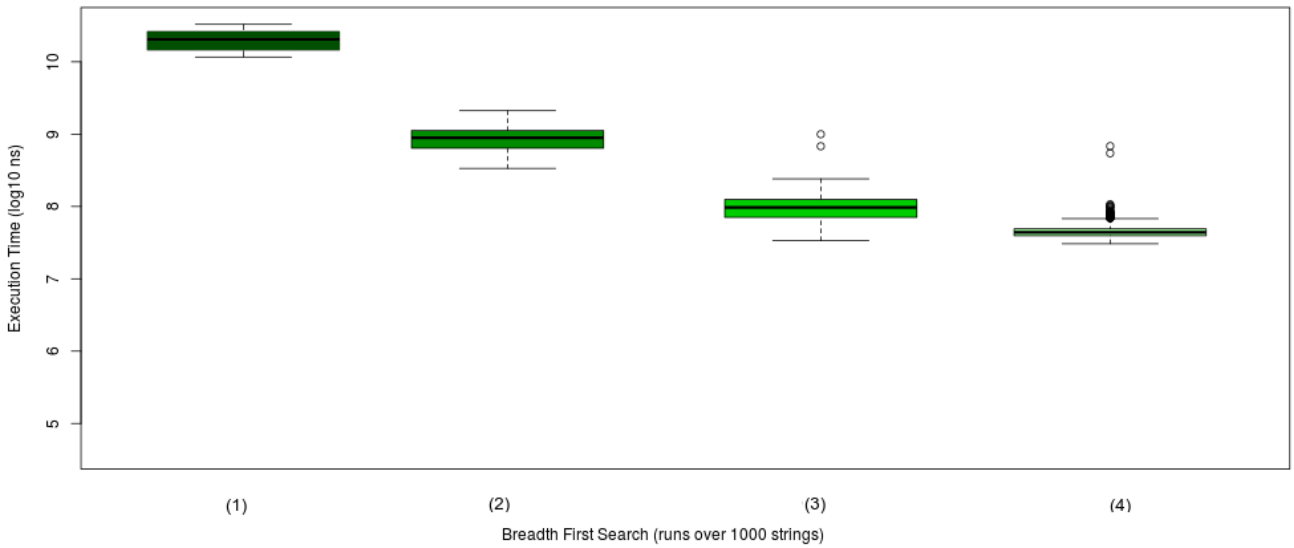


Figure 15: Time: BFS for the decompression problem for 6 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

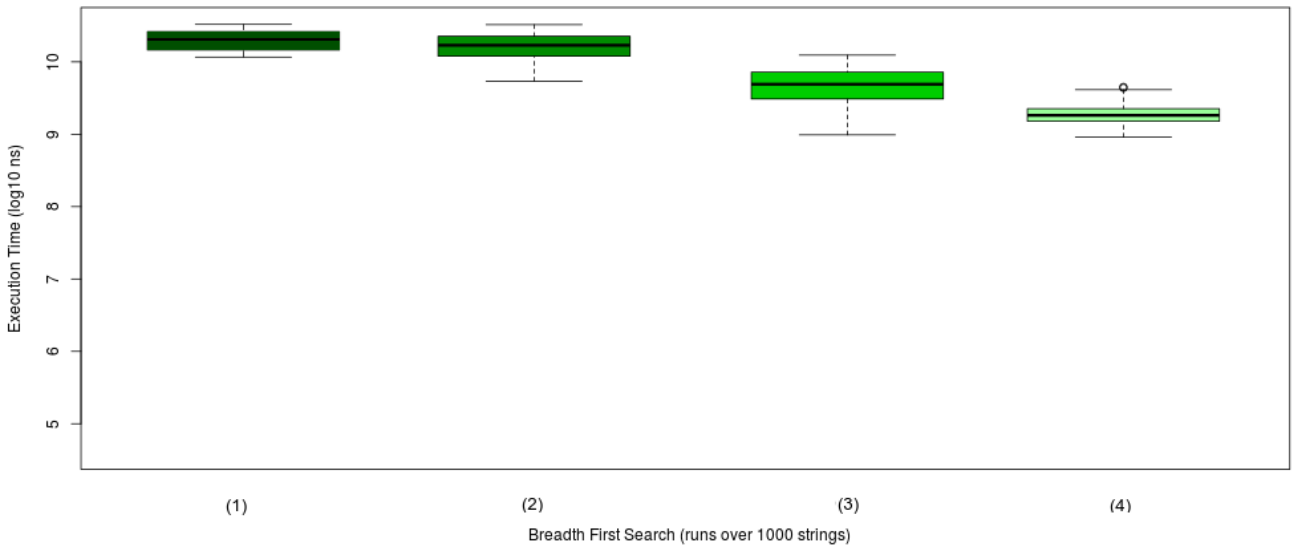


Figure 16: Time: BFS for the decompression problem for 7 character length strings
(1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters,
(4) the weighted degree heuristic with BFS for CSP.

The above diagrams (Figures 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16) show us that the Local Search is a quick search method for the decompression of n-gram graphs that have strings consisted by a few characters. As you can notice, the simple BFS diagrams are not included, because an ”out of memory” error had occurred during our executions. This behaviour is expected, due to the logic of this algorithm, when dealing with strings that do not have multiple appearances of their characters. The simple DFS diagrams for the 5, 6 and 7-character length are not included either, as the upper limit for the amount of executions while performing the backtracking algorithms (100 million attempts) was reached. When we want to decompress longer strings, we should prefer the DFS or BFS algorithm for the CSP simulation of our problem. The heuristics improve our time measurements, especially the weighted degree heuristic. The following table gives us a general view of all the search methods we used to decompress the n-gram graphs.

	length of text [chars]			
	4	5	6	7
Local Search	**0,58**	2,62	26,12	454,75
DFS simple	11.841,68	-	-	-
DFS with CSP	4,15	19,75	221,24	3.222,16
DFS with CSP & WD heuristic	0,59	**1,2**	**7,19**	**71,61**
DFS with CSP & probabilistic heuristic	1,04	3,81	22,09	303,9
BFS simple	-	-	-	-
BFS with CSP	7,51	63,08	905,6	17.343
BFS with CSP & WD heuristic	0,98	4,56	46,41	1.940,1
BFS with CSP & probabilistic heuristic	1,54	8,77	100,75	5.254,52
Best time	**0,58**	**1,2**	**7,19**	**71,61**

Table 1: General time measurement comparisons
Time in milliseconds, the total strings were 1000 for every algorithm measurement.
The dash symbol (-) indicates no results could be provided.

5.3 Method Cost measurements

In this Section, we present the method cost measurements of the software mentioned in the introduction of this chapter. The method cost indicates the amount of times the functions of each algorithm were called, until we have our solution. We find the method cost measurements interesting, as we can see the amount of steps (function calls) each search algorithm required, while trying to solve the decompression problem. The strings we decompressed are consisted of four, five, six and seven different characters. Each test file includes 1000 different strings.

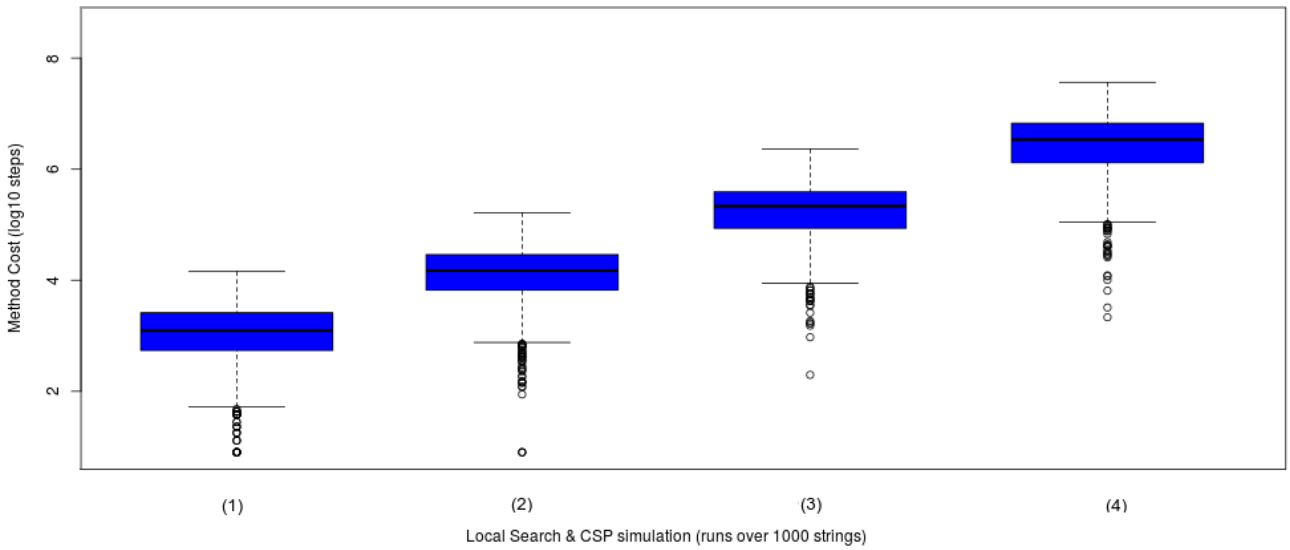


Figure 17: Method Cost: Local Search for the decompression CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

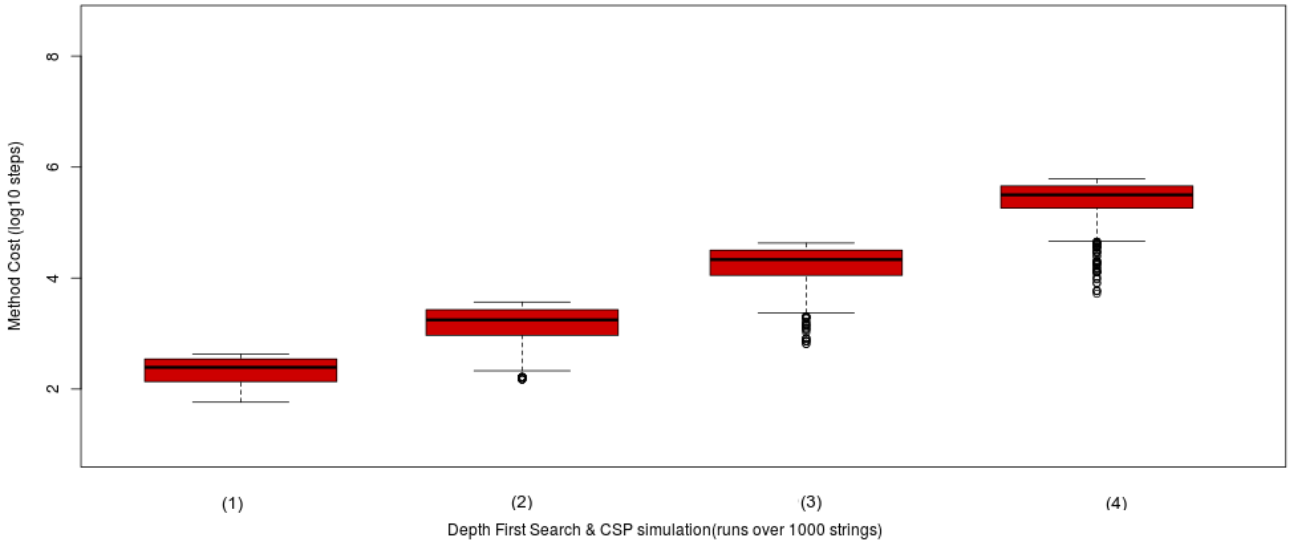


Figure 18: Method Cost: DFS for the decompression CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

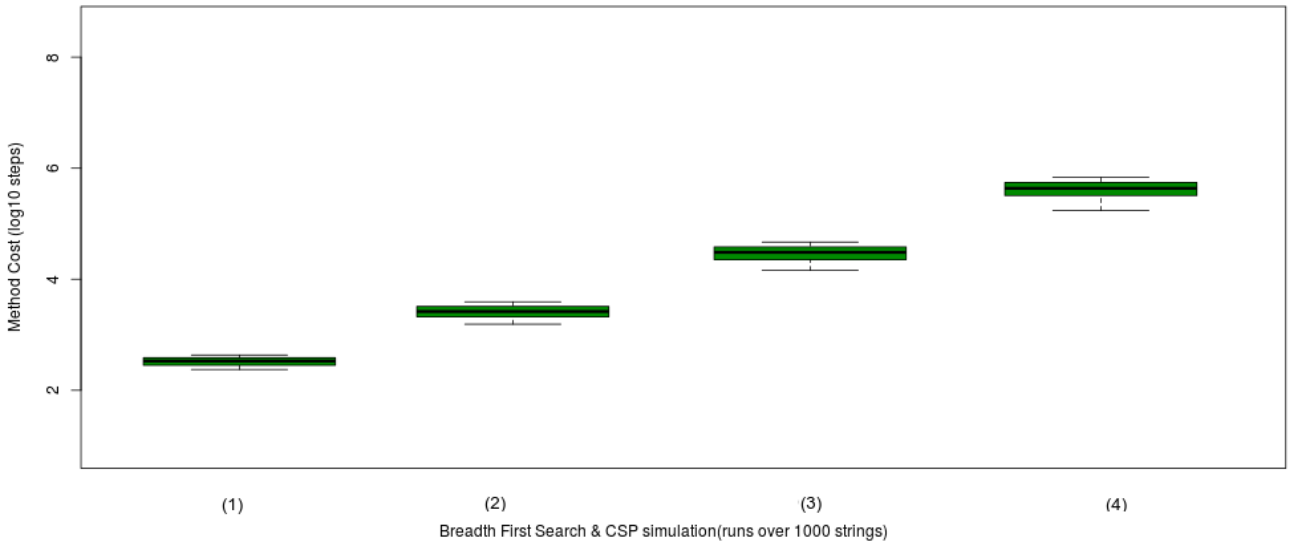


Figure 19: Method Cost: BFS for the decompression CSP - Multiple string lengths
The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

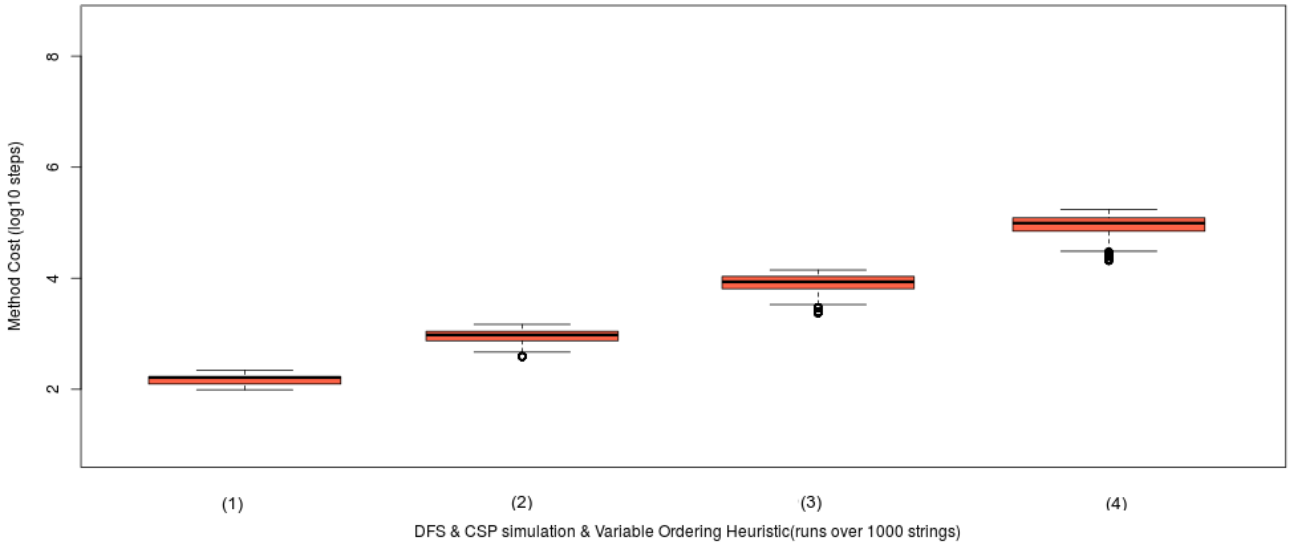


Figure 20: Method Cost: The weighted degree heuristic with DFS for the CSP - Multiple string lengths.

The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

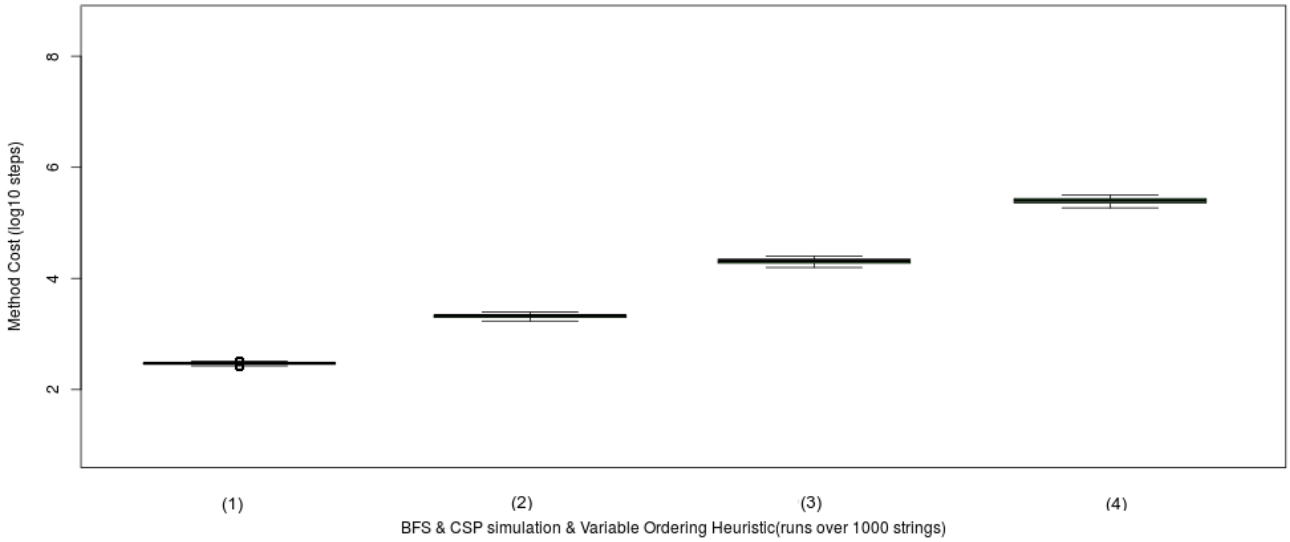


Figure 21: Method Cost: The weighted degree heuristic with BFS for the CSP - Multiple string lengths

The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

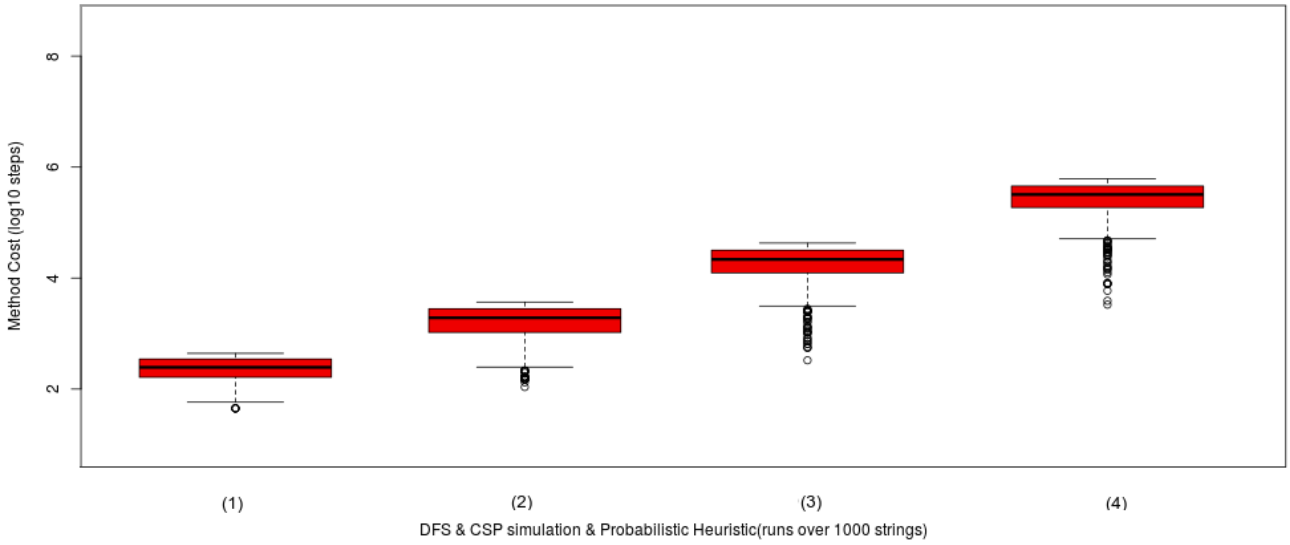


Figure 22: Method Cost: The probabilistic heuristic with DFS for the CSP - Multiple string lengths The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

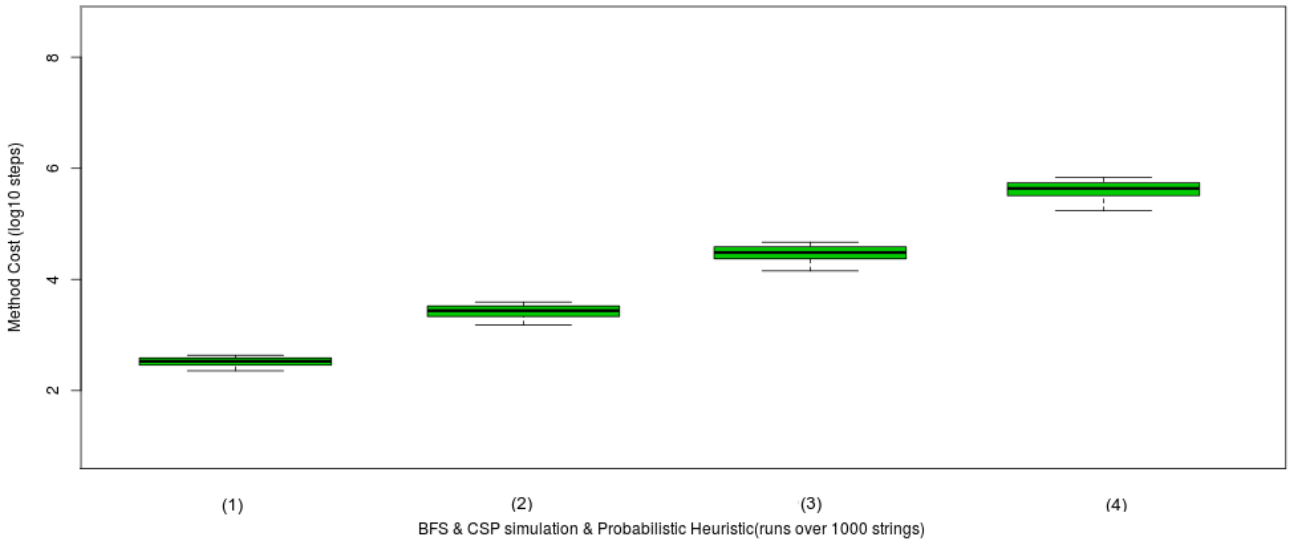


Figure 23: Method Cost: The probabilistic heuristic with BFS for the CSP - Multiple string lengths The strings consist of (1) four characters, (2) five characters, (3) six characters, (4) seven characters.

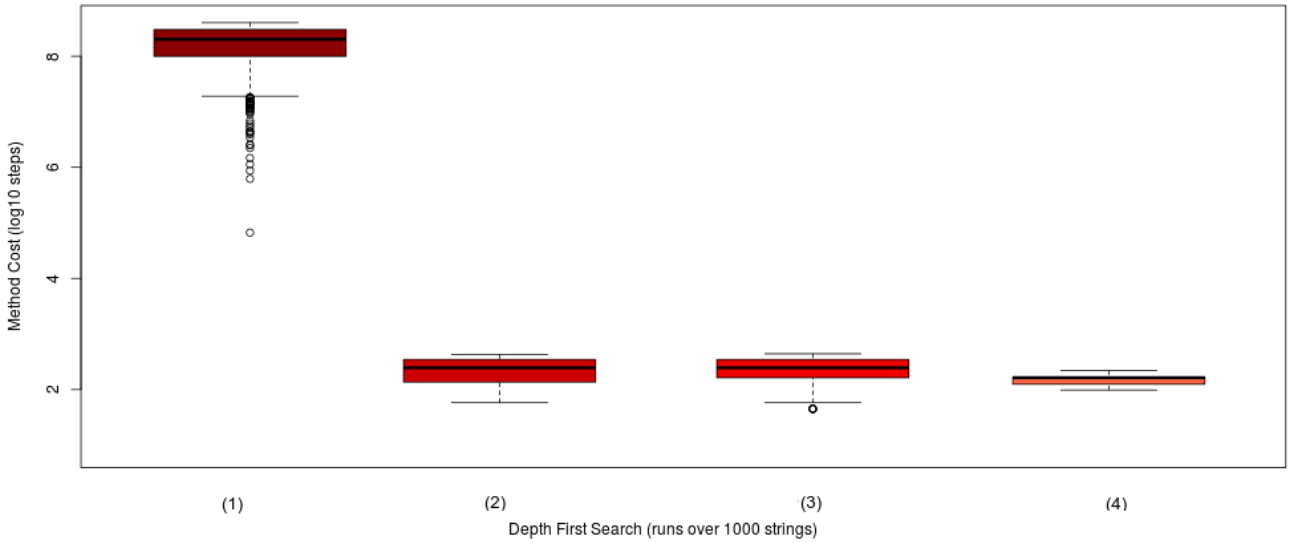


Figure 24: Method Cost: DFS for the decompression problem for 4 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

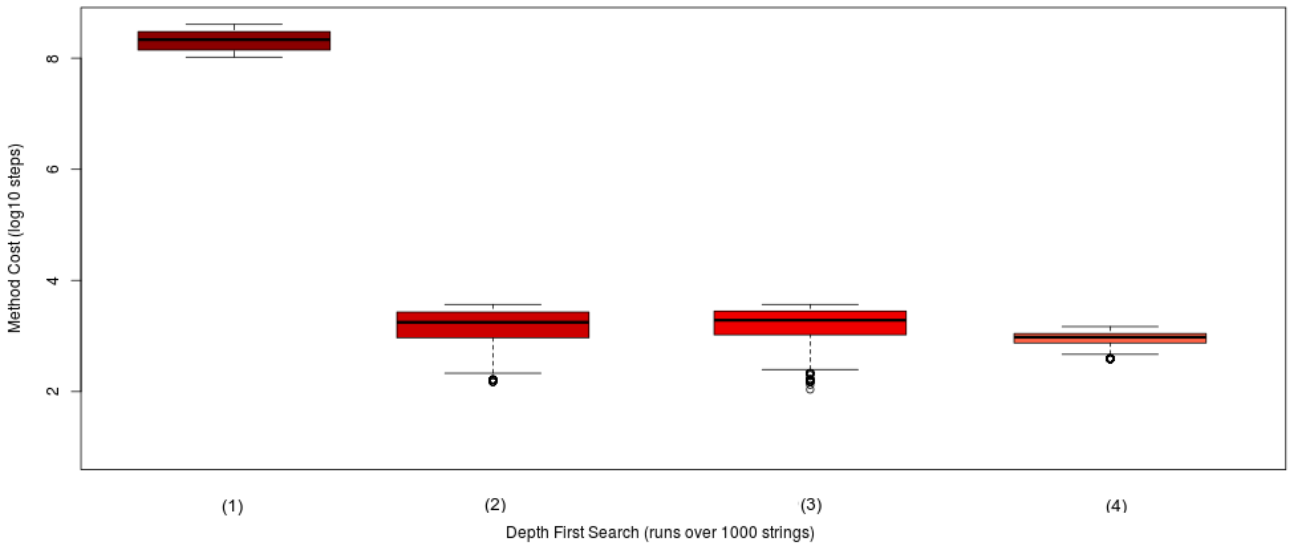


Figure 25: Method Cost: DFS for the decompression problem for 5 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

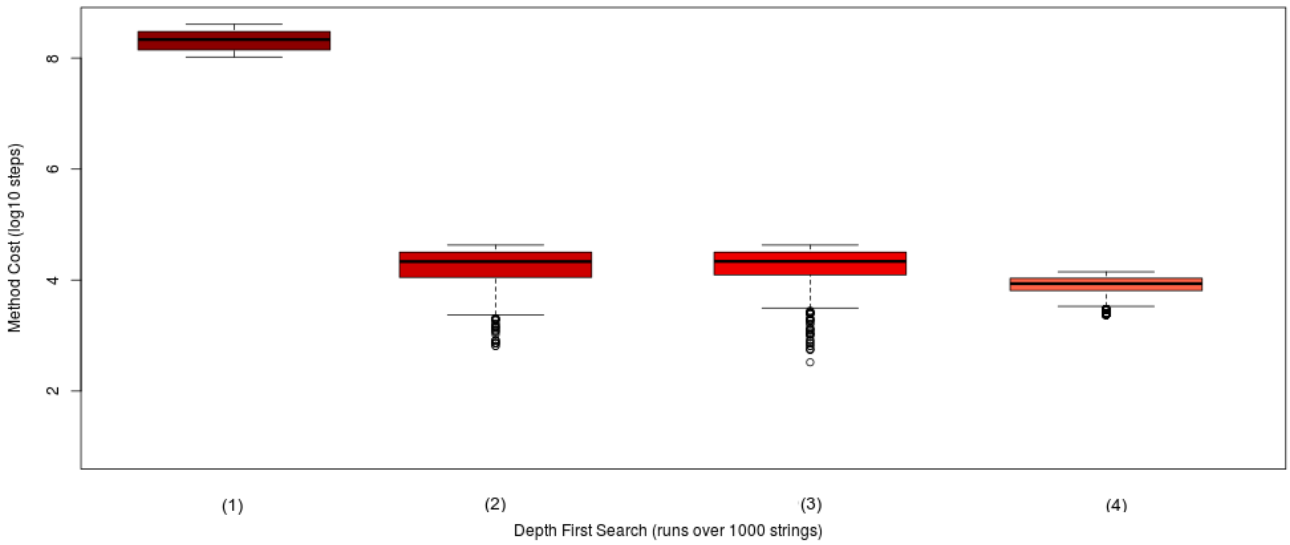


Figure 26: Method Cost: DFS for the decompression problem for 6 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

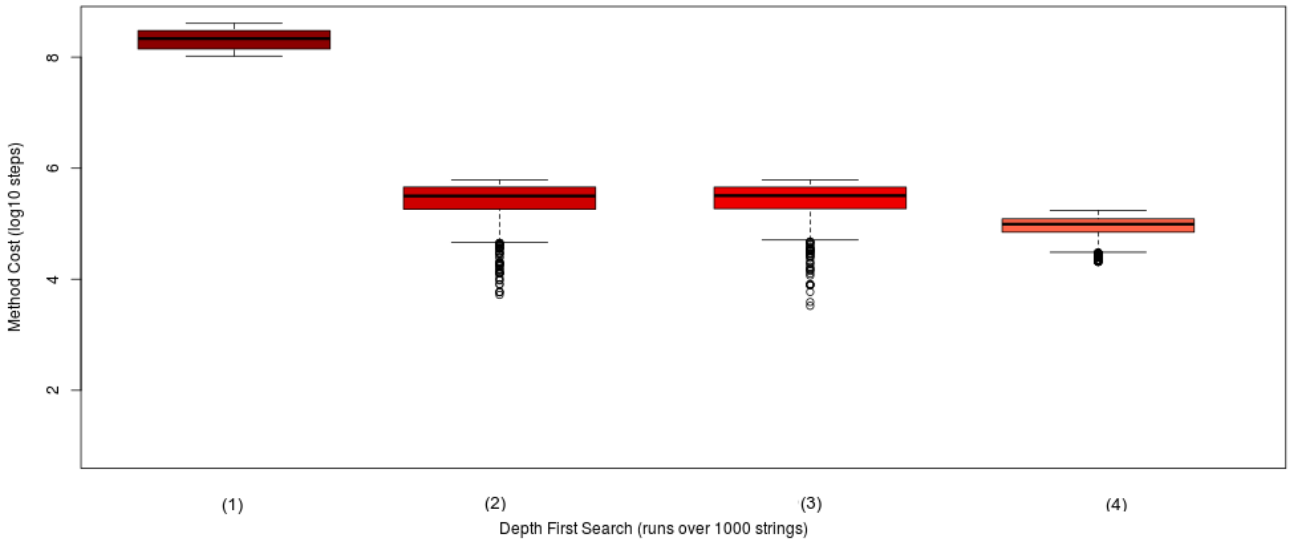


Figure 27: Method Cost: DFS for the decompression problem for 7 character length strings (1) simple DFS, (2) DFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with DFS for CSP.

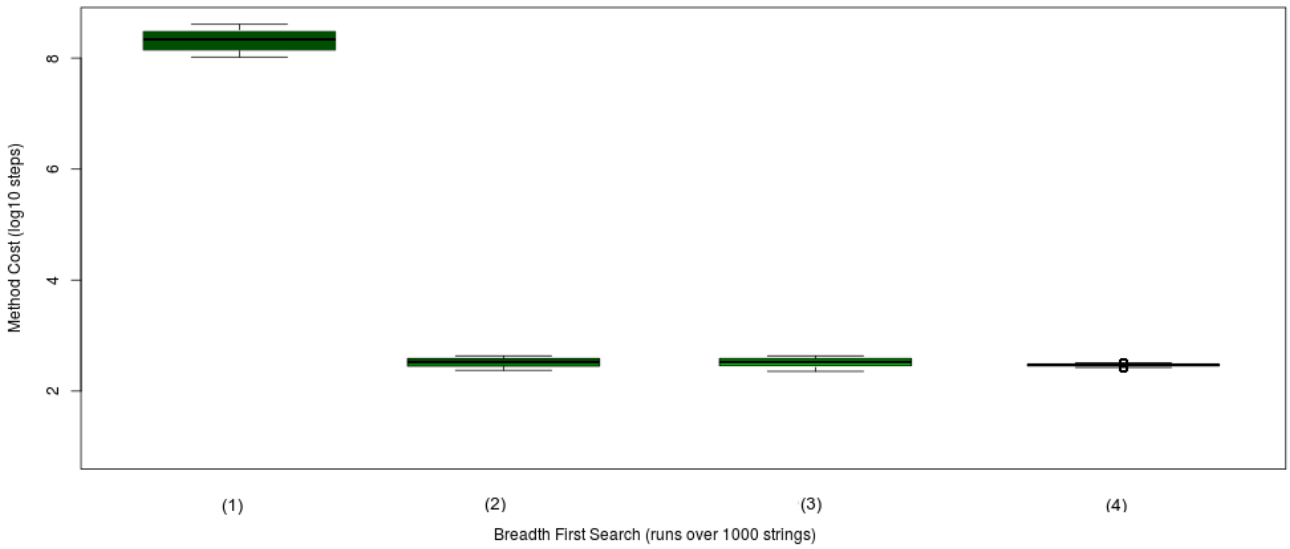


Figure 28: Method Cost: BFS for the decompression problem for 4 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

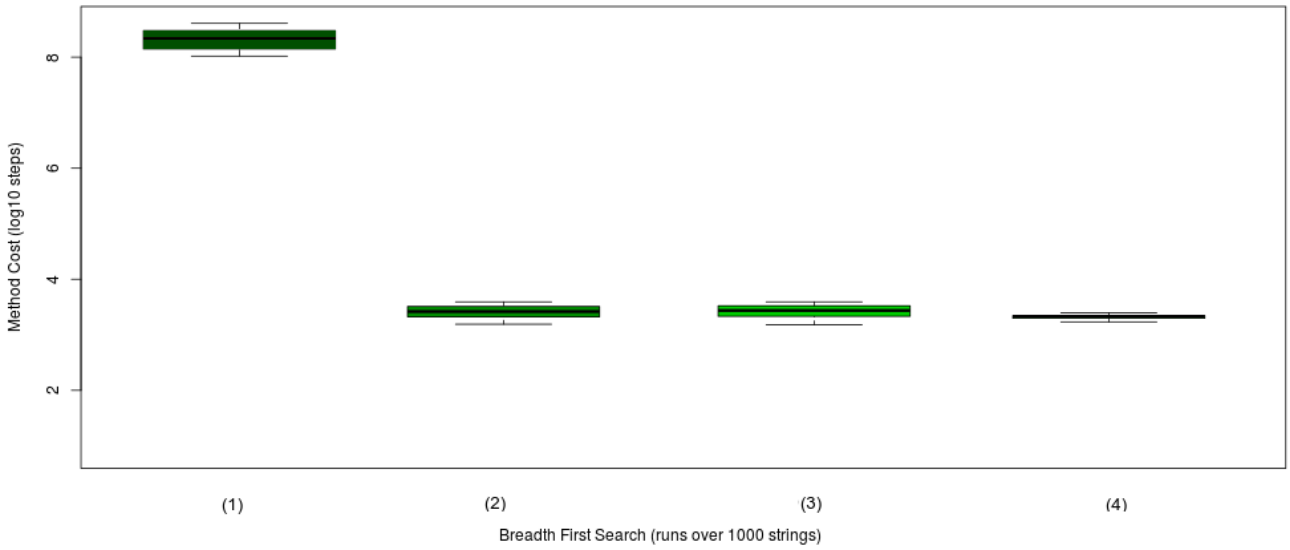


Figure 29: Method Cost: BFS for the decompression problem for 5 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

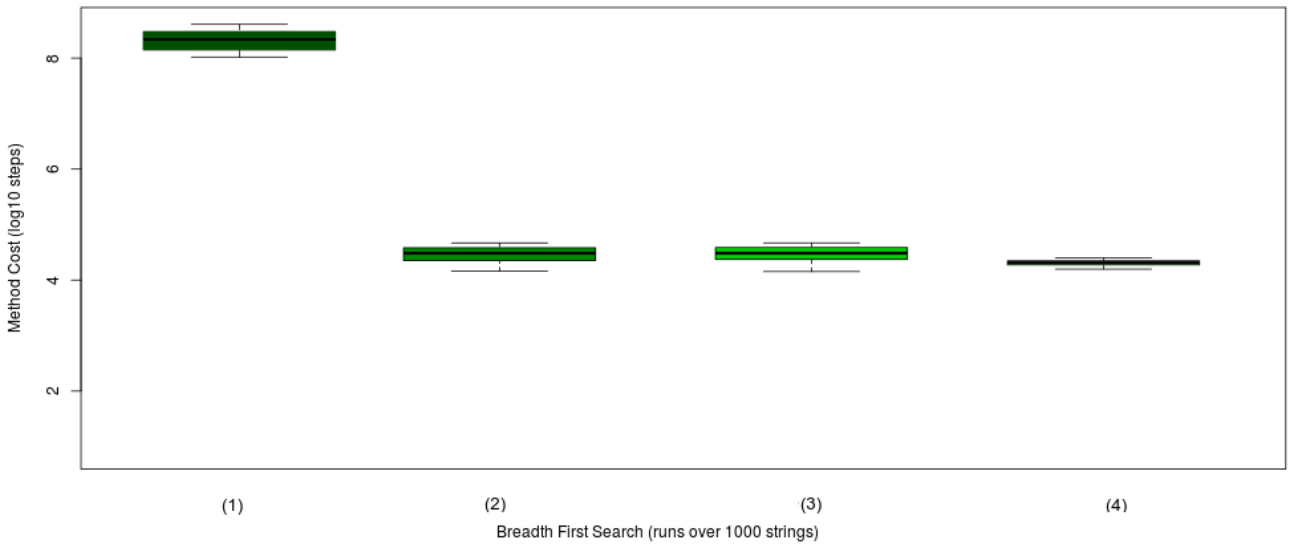


Figure 30: Method Cost: BFS for the decompression problem for 6 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

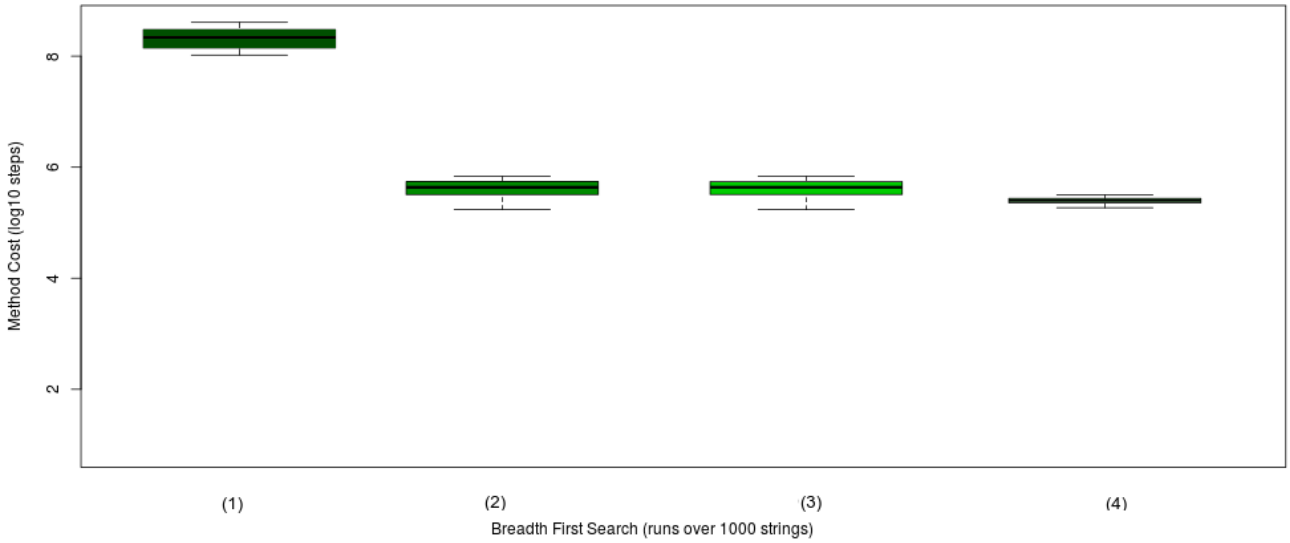


Figure 31: Method Cost: BFS for the decompression problem for 7 character length strings (1) simple BFS, (2) BFS for CSP, (3) the probabilistic heuristic with DFS for CSP characters, (4) the weighted degree heuristic with BFS for CSP.

The above diagrams (Figures 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31) show us that the DFS algorithmic approaches are better that the other search methods for the decompression of n-gram graphs, when we are interested in the amount of the functions that are executing in each algorithm. As we have mentioned in the previous subsection, while discussing the time measurements of our experiments, the simple BFS diagrams are not included, because an ”out of memory” error had occurred during our executions. The simple DFS diagrams for the

5, 6 and 7-character length are not included either, as the upper limit for the amount of executions while performing the backtracking algorithms (100 million attempts) was reached. The weighted degree heuristic for the DFS gave us the least method costs for all the different lengths of the strings we decompressed. The following table gives us a general view of all the search methods we used to decompress the n-gram graphs.

	length of text [chars]			
	4	5	6	7
Local Search	1,8 K	21,4 K	288,7 K	4.975,7K
DFS simple	201.803,2 K	-	-	-
DFS with CSP	0,25 K	1,83 K	21,7 K	314,3 K
DFS with CSP & WD heuristic	**0,16 K**	**0,93 K**	**8,35 K**	**97,36 K**
DFS with CSP & probabilistic heuristic	0,25 K	1,93 K	21,91 K	319,73 K
BFS simple	-	-	-	-
BFS with CSP	0,34 K	2,68 K	30,57 K	432,34 K
BFS with CSP & WD heuristic	0,3 K	2,1 K	20,46 K	252,26 K
BFS with CSP & probabilistic heuristic	0,34 K	2,74 K	30,81 K	435,3 K
Best time	**0,16 K**	**0,93 K**	**8,35 K**	**97,36 K**

Table 2: General method cost measurement comparisons
Cost is measured in steps (times each function of the search problem is executed), the total strings were 1000 for every algorithm measurement.
The dash symbol (-) indicates no results could be provided.

6 CONCLUSION

In this thesis, we have implemented and performed experiments on a set of search algorithms, in order to solve the decompression problem of the n-gram graphs. We defined the constraint satisfaction problem formulation of the decompression problem, in order to optimize it, and we designed two variable ordering heuristics to improve our time measurements. The experimental results presented judged one class of character strings to be decompressed, these which do not have multiple character appearances. The best results for short-length text were achieved when Local Search was performed. For longer-length strings, the weighted degree heuristic was the one that offered the best results.

The results presented above are encouraging and can be improved. The parallelization of the search algorithms we analyzed could offer betters time measurements. Furthermore, new variable ordering heuristics and value ordering heuristics can be designed and implemented, so that we can determine which of them can provide better experimental results for the decompression problem. Future work on the decompression problem of the n-gram graphs can include experiments focused on strings that have multiple appearances of the same character.

ACRONYMS

AI	Artificial Intelligence
CSP	Constraint Satisfaction Problem
NLP	Natural Language Processing
LS	Local Search
DFS	Depth First Search
BFS	Breadth First Search
WD	Weighted Degree

REFERENCES

[1] S. Russell, P. Norvig. (2003), *Artificial Intelligence - A modern approach*, Second Edition, Prentice Hall, pages 179-200.

[2] Kenneth H Rosen. (2011), *Discrete mathematics and its applications*, Seventh Edition, McGraw-Hill New York.

[3] George Giannakopoulos. (2009), *Automatic summarization from multiple documents*, Citeseer.

[4] F.Aisopos, G.Papadakis, T.Varvarigou (2011), *Sentiment Analysis of Social Media Content Using N-Gram Graphs*, Proceedings of the 3rd ACM SIGMM international workshop on Social media.

[5] G. Giannakopoulos, V. Karkaletsis, G. Vouros, P, Stamatopoulos (2008), *Summarization System Evaluation Revisited: N-Gram Graphs*, ACM Trans. Speech Lang. Process.

[6] V. Rentoumi, G. Giannakopoulos, V. Karkaletsis, G. Vouros (2009), *Sentiment Analysis of Figurative Language using a Word Sense Disambiguation Approach*, Borovets, Bulgaria

[7] G. Giannakopoulos, G. Kioumourtzis, V. Karkaletsis (2014), *NewSum: "N-Gram Graph"-Based Summarization in the Real World*, Innovative Document Summarization Techniques: Revolutionizing Knowledge Understanding, IGI

[8] D. Polychronopoulos, A. Krithara, C. Nikolaou, G. Paliouras, Y. Almirantis, G. Giannakopoulos (2014), *Analysis and Classification of Constrained DNA Elements with N-gram Graphs and Genomic Signatures*, Algorithms for Computational Biology, Springer International Publishing

[9] D. Poole, A. Mackworth. (2010), *Artificial Intelligence: Foundations of Computational Agents*, Cambridge University Press, 4.8

[10] F. Rossi, P. van Beek, T.Walsh (2006), *Handbook of Constraint Programming*, Elsevier B.V., 4.6

[11] G and C Merriam, Webster's Revised Unabridged Dictionary (1913)

[12] William B Cavnar, John M Trenkle, et al. *N-gram-based text cat- egorization*. Ann Arbor MI, 48113(2):161–175, 1994.

[13] Faidra Monachou, *Designed and implemented two n-gram graph decompression algorithms in Java using the JInsect Toolkit*, 2013, Personal communication with supervisor.

[14] A. Auger, B. Doerr. (2011) *Theory of Randomized Search Heuristics: Foundations and Recent Developments*, preface

[15] K. Potter (2006) *Methods for Presenting Statistical Information: The Box Plot*