The use of k-best paths algorithms in clock control sequence reconstruction

Turid Herland

Master's Thesis
Master of Science in Information Security
30 ECTS
Department of Computer Science and Media Technology
Gjøvik University College, 2006

Institutt for
informatikk og medieteknikk
Høgskolen i Gjøvik
Postboks 191
2802 Gjøvik

Department of Computer Science
and Media Technology
Gjøvik University College
Box 191
N-2802 Gjøvik
Norway

Abstract

Keystream generators based on clock-controlled linear feedback shift registers are often used in stream cipher systems. Cryptanalysis of such generators can be divided into two stages: In the first stage the initial state of the LFSR is found; the second stage reconstructs the clock control sequence. A set of candidate initial states for the LFSR can be found by computing the edit distance between the sequences produced by every possible initial state and the observed ciphertex, keeping the candidates that give distances below a given threshold. The edit distance is computed recursively, and a matrix of partial edit distances is filled out during these computations. Clock control sequence reconstruction can be done by finding paths back through the edit distance matrix.

In the known plaintext scenario an optimal path, which can be found very efficiently, would give the correct clock control sequence. However, when the plaintext is not known, it behaves like noise in the cryptanalytic process, and this noise makes it necessary to also consider suboptimal paths through the matrices. This thesis uses a k-best paths algorithm to extract the k shortest paths from the matrix, in order to find the path that represents the correct clock control sequence.

It is shown, through experimental results, that the attack is successful, and that it performs better than a similar depth-first search attack when there is no noise. However, in the presence of noise, the depth-first search performs slightly better than the k-best paths attack, on average.

Samandrag

Nøkkelstraumgeneratorar som er baserte på klokkekontrollerte lineært tilbakekopla skiftregister blir ofte brukte i flytchiffer-system. Kryptoanalyse av slike generatorar kan delast inn i to etappar: Den første etappen finn kandidatar til starttilstanden (initial state) til skiftregisteret, og den andre etappen rekonstruerer klokkekontrollsekvensen. Ein kan finna kandidatar til denne starttilstanden ved å rekna ut editeringsdistansen (edit distance) mellom talsekvensen som blir produsert av kvar mogleg starttilstand, og den observerte chifferteksten, og behalda dei kandidatane som gjev ein distanse under ein viss verdi som er bestemt på førehand. Utrekninga av editeringsdistansen skjer rekursivt, ved å fylla ei matrise med partielle editeringsdistansar. Klokkekontrollsekvensen kan då rekonstruerast ved å finna stigar tilbake gjennom denne matrisa.

I eit kjent-klartekst-angrep vil ein optimal stig, som det finst effektive metodar for å finna, gje den korrekte klokkekontrollsekvensen. Derimot, når klarteksten ikkje er kjent, vert han oppfatta som støy i kryptoanalyseprosessen, og denne støyen gjer det naudsynt å vurdera suboptimale stigar gjennom matrisa, i tillegg til dei optimale stigane. Denne oppgåva brukar ein k-best-paths-algoritme til å finna dei k kortaste stigane i matrisa, i håp om at ein av dei representerer den korrekte klokkekontrollsekvensen.

Gjennom eksperimentelle resultat vert det vist at angrepet fungerer, og at det finn løysinga raskare enn eit liknande angrep, basert på eit depth-first search, når det ikkje er noko støy. Derimot så ser det ut til at depth-first-search-angrepet gjennomsnittleg er betre når støynivået aukar.

Acknowledgments

My supervisor, Professor Slobodan Petrović, has been a great resource for me. He suggested the topic for the thesis, and helped me get acquainted with it. He has always been willing to answer my questions, with a very low response time, and to meet with me whenever I felt the need. He has challenged me, and pushed me when I needed it.

I would also like to thank my husband, Johan Herland. His moral support and encouragement has been valuable, and sometimes really needed; he has also been a good partner for discussions around thesis topics. But most of all I want to thank him for lending me his programming expertise to implement much of the code that was needed for the experimental work.

Turid Herland, June 30, 2006.

Contents

List of Figures

List of Tables

1 Introduction

This thesis is concerned with cryptanalysis of stream ciphers that use clock controlled keystream generators. Such keystream generators consist of a linear feedback shift register (LFSR) and a general type subgenerator. The LFSR is clocked according to the output sequence of this subgenerator, and the output sequence of the clocked LFSR is then taken as the keystream for the stream cipher. This keystream is then XORed with the plaintext sequence to encrypt it, and get the ciphertext sequence. Conversely, to decrypt the ciphertext sequence, it should be XORed with the keystream to get the plaintext sequence.

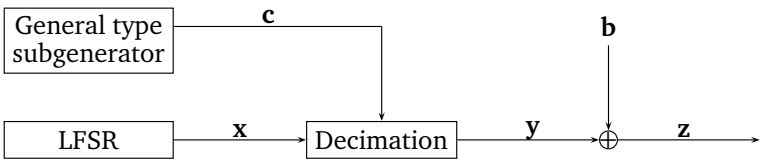


Figure 1: A stream cipher with clock controlled keystream generator. The clock control sequence **c** dictates how many times the LFSR should be clocked before its output is taken as the next bit in the keystream sequence **y**. This sequence is then XORed with the plaintext sequence **b**, to get the ciphertext sequence **z**.

The irregular clocking of the LFSR can be modelled as irregular decimation of its output, as illustrated in Figure 1. The decimation is controlled by the clock control sequence; each element of this sequence is a nonnegative integer that specifies how many positions to advance in the LFSR's output sequence before outputting the current element as the next element of the keystream sequence. That is, if the current clock control element is c, then the next $c-1$ bits of the LFSR's output sequence will be skipped, and the c^{th} bit is taken as the next output bit for the keystream sequence.

Clock controlled, or irregularly clocked, generators are often used in stream cipher systems because they produce keystreams with good cryptographic characteristics; that is they produce keystreams with long periods, high linear complexity, and good statistical properties. Popular examples of clock controlled generators are the alternating step generator and the shrinking generator. An introduction to these generators, and stream ciphers in general, can be found in [1].

Despite of the good cryptographic properties of the clock controlled generators, several methods have been developed for cryptanalyzing such generators. The goal of cryptanalysis is to decrypt the ciphertext in order to obtain the plaintext without knowledge of the secret key. In the case of stream ciphers the secret key is often the initial state of the keystream generator, and for irregularly clocked generators that means the initial states of both the LFSR and the clock control generator. Thus cryptanalysis of stream ciphers is usually concerned with cryptanalyzing the keystream generator. The different cryptanalytic approaches have used edit distances [2, 3], linear consistency tests [4], or coding theory [5] to try and cryptanalyze the generators.

The cryptanalysis scheme presented in this thesis builds on the edit distance approach used in [2] and [3]. Attacks of this kind consist of two phases: The first phase determines the candidate initial states of the LFSR by means of edit distances, and the second phase finds the clock control sequence. In the first phase all possible initial states for the LFSR are tried, and the edit distance is calculated between the LFSR's output sequence and the ciphertext sequence for each initial state, and those initial states that give the lowest edit distances are kept as candidate initial states. The edit distances in the first phase are calculated iteratively, using matrices of partial edit distances, and in [3] the second phase reconstructs the clock control sequence by performing a directed depth-first search for the optimal and suboptimal paths through the matrix. The idea in this thesis is to replace this depth-first search with a k-best paths search.

k-best paths algorithms find the k paths with the lowest total length between a pair of vertices in a given graph with weighted edges. The best known approach to this problem was proposed by Eppstein in [6], where the idea is to find the k shortest paths using a shortest path tree for the given graph and a graph structure combining heaps representing suboptimal paths in the original graph. This algorithm is used, with some minor adjustments, to find the k best paths through the edit distance matrix, in order to reconstruct the clock sequence, in the second phase of the attack in this thesis.

The main goal of this thesis project is to introduce the idea of using k-best paths algorithms for clock control reconstruction in the second phase of an attack based on edit distances, and to gain some knowledge of the performance of the proposed attack through simulations. Specifically, an attack is developed, that uses edit distances to find the candidate initial states in the first phase, and then a modified version of a k-best paths algorithm in the second phase. The number k of reconstructed paths that are needed to find the correct clock control sequence is compared to the number of reconstructed paths for the same examples when using a depth-first search instead of k-best paths.

2 Previous work

2.1 Cryptanalysis of clock controlled stream ciphers

Several different approaches have been proposed for cryptanalyzing stream ciphers based on clock controlled keystream generators. In [7], a probabilistic coding theory approach was used for the reconstruction of the clock control sequence in the shrinking generator. Another coding theory approach was used in [5], where a maximum a posteriori (MAP) decoding technique (see [8]) is used to reconstruct the initial states of both the clocked LFSR and the clock control sequence generator, based on an identified relation between the cryptanalysis problem and the decoding problem for the deletion/insertion channel.

Molland [4] uses a linear consistency test approach. The general idea is to guess the positions that the keystream bits had in the output sequence from the LFSR, and to use a linear consistency test (LCT) on the bit positions of the LFSR's output sequence that would then be known. A low weight cyclic equation that holds for any bitstream generated by the LFSR is used to test for consistency. The LCT approach was originally proposed in [9], but here the Gaussian algorithm was used to test for consistency.

The edit distance approach was introduced by Golić and Mihaljević in [2]. It finds the candidate initial states of the LFSR by calculating the edit distance between the sequence produced by the LFSR, but with regular clocking, and the ciphertext sequence, for each possible initial state. The candidate initial states are those that give an edit distance below a given threshold. The same edit distance approach is also used by Petrović and Fúster-Sabater in [3], to find the candidate initial states for the LFSR. Then the clock control sequence is reconstructed by means of a depth-first search back through the edit distance matrices that were created in the process of calculating the edit distances in the first phase. The possibility of reconstructing the clock control sequence in this way was also mentioned, but not tested, in [10].

The edit distance approach is promising because it takes into consideration the noise that the plaintext introduces into the keystream to form the ciphertext. That is, the initial states can be recovered from the ciphertext, without knowledge of the corresponding plaintext; this is known as a ciphertext only attack. A less difficult, and more common, approach is to recover the appropriate initial states directly from the keystream. This is called a known plaintext attack, and the attacks proposed in [4, 5, 7] are all of this type. It is claimed in [4] that the algorithm introduced there can be adjusted to a ciphertext only attack.

2.2 Finding the k best paths

The problem of finding the k shortest paths between two vertices in a graph with weighted edges has applications in many fields, and has thus been studied in many different contexts. In the field of biological sequence alignment, an application has arisen that is very similar to the problem of reconstructing the k best paths through the edit distance matrix, to find the clock control sequence. The edit distance between two biological sequences is often used to get an approximation of the biological alignment between them, and to

get an even better approximation, the k best alignments from the edit distance calculation are often considered. Thus algorithms for finding the k best paths in a grid graph, which can efficiently represent the matrix, have been studied in this field. Byers and Waterman [11, 12] use a simple depth-first search to find suboptimal alignments, and Naor and Brutlag combine suboptimal paths to get more suboptimal paths [13]. In [14] a more sophisticated k-best paths algorithm by Eppstein [6] is used on the biological sequences.

This algorithm proposed by Eppstein [6] finds the k shortest paths between two given nodes in a directed graph in $O(m + n \log n + k)$ time, for a graph with n vertices and m edges; for a directed acyclic graph, this bound can be reduced to $O(m + n + k)$. This time is obtained because a shortest path tree can be created in time $O(n)$ if the graph is directed or acyclic. A single destination shortest path tree, where the destination is the same as the destination node for the paths that will be searched for, is used througout the algorithm, and should be built before the search starts. The main idea in [6] is to build a heap for each vertex in the graph, representing all paths from this vertex to the destination vertex that differ from the shortest path between these nodes in at least one edge. A graph that connects these heaps together can then be searched through in a breadth-first manner to get the k best paths between a given source vertex and the destination, starting from the heap representing the source node.

Eppstein's algorithm is generally the best one proposed so far; earlier publications on the k best paths problem include [15–25]. However, some algorithms that perform better than Eppstein's in practice have been proposed by Jiménez and Marzal [26, 27]. In [26], a set of equations that generalize the Bellman equations for the single shortest path problem is used. The proposed algorithm efficiently solves these equations, and the k shortest paths are found in time $O(m + kn \log(m/n))$, assuming that a shortest path tree has already been created. [27] introduces an improvement to Eppstein's algorithm; the idea is to create heaps only for those vertices that are necessary to be able to find the k best paths between the given source and destination. Thus, in the worst case, this algorithm will construct a heap for all vertices, like in Eppstein's version, but in practice this so called lazy version will often end up doing less work than Eppstein's original algorithm.

3 Proposed attack

The main contribution from this thesis is the idea of using k-best paths algorithms to reconstruct the clock control sequence in a cryptanalysis method that uses edit distance computations to find the candidate initial states of the LFSR. This chapter describes the proposed method in detail. In short, the method is identical to the method presented in [3], but the depth first search through the edit distance matrix is replaced by a k-best paths search using the algorithm from [6].

3.1 Statistical model

The stream cipher model briefly introduced in chapter 1 is used throughout this chapter, and its figure is repeated here, in Figure 2, for convenience. A more thorough description of the model is given below.

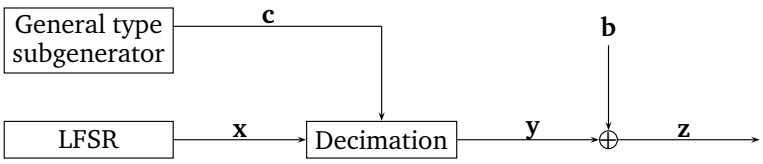


Figure 2: A stream cipher with clock controlled keystream generator. The clock control sequence **c** dictates how many times the LFSR should be clocked before its output is taken as the next bit in the keystream sequence **y**. This sequence is then XORed with the plaintext sequence **b**, to get the ciphertext sequence **z**.

The LFSR in this model is clocked regularly, and it produces a binary output sequence **x**, dependent on the LFSR's feedback polynomial and its initial state; it is assumed that the feedback polynomial is known. The general subgenerator produces the clock control sequence **c**, whose elements are positive integers. More specifically, each $c_n \in \mathbf{c}$ is taken from the set $\{a_1, a_2, \ldots, a_A\}$, where each a_n is a positive integer, and A is the number of different values the c_n can take on. It is assumed that the necessary specifics about this subgenerator are known, so that, if the initial state is recovered, its output sequence can be reproduced.

The irregular clocking of the LFSR is modelled as irregular decimation of the LFSR's output sequence **x**. This decimation is controlled by the clock control sequence **c** in the following manner: Let $k(t)$ be a clock function indicating the total sum of the clock at time t; then $y_t = x_{k(t)}$, and $k(t) = k(t-1) + c_t$, where **y** is the decimated keystream sequence. Thus the t^{th} bit in **y** is the same as bit number $k(t)$ in **x**.

The plaintext sequence **b** is modelled as a binary noise sequence, where $p = P(b_n = 1) < 0.5 \ \forall n$ is called the correlation parameter. The plaintext sequence **b** and the keystream sequence **y** are added bitwise to encrypt the plaintext sequence, and the result is the ciphertext sequence **z**. **z** can be decrypted again by performing the same bitwise addition with **y**, to recover **b**.

3.2 Finding the candidate initial states

3.2.1 The attack

The first step of the attack is to find the candidate initial states for the LFSR. This is done by calculating the edit distance between all possible output sequences of the LFSR and the intercepted ciphertext sequence.

Let $\hat{X}$ be one possible initial state for the LFSR, and let $\hat{\mathbf{x}}$ be the output sequence it produces. Then name the hypotheses

H_0 : The observed ciphertext sequence $\mathbf{z}$ is produced by the current initial state $\hat{X}$, and

H_1 : The observed ciphertext sequence $\mathbf{z}$ is not produced by the current initial state $\hat{X}$.

For each possible initial state $\hat{X}$, H_0 or H_1 is accepted based on the edit distance between $\hat{\mathbf{x}}$ and $\mathbf{z}$. If this distance d is above a given threshold t, then H_1 is accepted; if d is lower than or equal to t, then H_0 is accepted.

The threshold t is determined based on chosen probabilities for "the missing event", P_m, and for "the false alarm", P_f. Let D be the random variable representing the outcome of d. Then these probabilities are defined as $P_m = P(D > t|H_0)$ and $P_f = P(D \leq t|H_1)$. First, the probabilities P_f and P_m are selected; note that P_f determines the mathematical expectation of the cardinality of the set of candidate initial states. Next, to determine the threshold t, select $1/P_m$ initial states $\hat{X}$ for the LFSR at random, and calculate the edit distance between the corresponding output sequence $\hat{\mathbf{x}}$ and the ciphertext sequence $\mathbf{z}$. Select a threshold t that is greater than the maximum edit distance obtained in this process.

After determining t, the attack goes through all the remaining initial states $\hat{X}$ for the LFSR. For each such initial state, it generates the corresponding output sequence $\hat{\mathbf{x}}$, and calculates the edit distance d between the observed ciphertext sequence $\mathbf{z}$ and $\hat{\mathbf{x}}$. Then H_0 is accepted if $d \leq t$, and H_1 is accepted if $d > t$. The set of candidate initial states consists of those initial states for which H_0 was accepted, that is those that gave edit distances less than or equal to t.

3.2.2 Edit distance computation

The distance d should be calculated between the observed ciphertext sequence $\mathbf{z}$ of length N, and a prefix of length M of the LFSR's output sequence $\hat{\mathbf{x}}$. One cannot determine the length M of the sequence $\hat{\mathbf{x}}$ that produced the observed ciphertext sequence $\mathbf{z}$ of length N without knowledge of the clock sequence $\mathbf{c}$. Thus the best one can do is to make an informed guess. Two strategies that are mentioned in [2] are to use the mathematical expectation of M, or to use the maximum value of M. These values can be calculated based on general knowledge about the statistical properties of the clock sequence $\mathbf{c}$, like the maximum value that each element c_n can take on, or the probability of deletion, that is the probability that each c_n is greater than 1. For this thesis, the mathematical expectation of M is used.

The edit distance measure, also called Levenshtein distance, was first introduced by Levenshtein in [28]. The edit distance between two sequences is defined to be the minimum number of edit operations required to transform one of the sequences into the other, where the edit operations are substitutions, deletions, and insertions of elements. According to the statistical model described in section 3.1, it is clear that the ciphertext sequence $\mathbf{c}$ is obtained from the sequence $\mathbf{x}$ by deletion of some bits, controlled by the

clock sequence $\mathbf{c}$, and then by complementation of the remaining bits with probability p. Also, the maximum number of consecutive deletions is $E = \max_{n=1,\dots,A}\{a_n\} - 1$, where $\{a_1, \dots, a_A\}$ are all the possible values that the elements of $\mathbf{c}$ can take on.

In [29] an efficient algorithm for calculating the constrained edit distance is presented, where the constraints are related to the total number of edit operations. The constrained edit distance can be calculated iteratively by filling a matrix of partial constrained edit distances: Let e represent the number of deletions, and s the number of substitutions. Then the edit distance between the prefix $\mathbf{x_{e+s}} = \{x_i\}_{i=1}^{e+s}$ of the sequence $\mathbf{x}$ and the prefix $\mathbf{z_s} = \{z_i\}_{i=1}^{s}$ of the sequence $\mathbf{z}$ is given by:

$$\begin{aligned} W[e,s] = \min\{&W[e-e_1, s-1] + e_1 d_e + d(x_{e+s}, z_s)| \\ &\max\{0, e - \min\{N-M, (s-1)E\}\} \le e_1 \le \min\{e, E\}\}, \\ &s = 1, \dots, M \quad e = 1, \dots, \min\{N-M, sE\}, \end{aligned} \tag{3.1}$$

where d_e is the elementary edit distance associated with the deletion of one element, $d(x,z)$ is the elementary edit distance associated with substitution of the symbol x by the symbol z, and E is the maximum number of consecutive deletions; N is the length of the sequence $\mathbf{x}$, and M the length of $\mathbf{z}$. The formula (3.1) assumes that deletions are performed before substitutions, agreeing with the model described in chapter 1. This is the exact formula presented and used for finding the candidate initial states in [3].

3.3 Clock control sequence reconstruction

When the candidate initial states for the LFSR are found, the next step is to reconstruct the clock control sequence. The idea presented in [3] is to find optimal and suboptimal paths through the edit distance matrix for each candidate initial state, using a directed depth-first search. The same idea is used here, but a k-best paths search is used instead of the depth-first search. Each such search provides k paths that correspond to k different clock control sequences that are candidates for the correct clock sequence.

3.3.1 Setup

For two given sequences $\mathbf{x}$ and $\mathbf{z}$, of lengths M and N respectively, the matrix of partial edit distances, W, also called the edit distance matrix, is built up of the partial edit distances between the prefixes $\mathbf{x_{e+s}}$ and $\mathbf{z_s}$, of $\mathbf{x}$ and $\mathbf{z}$. That is, the matrix position $W[e+s, s]$ holds the edit distance d between the first $e+s$ bits of $\mathbf{x}$ and the first s bits of $\mathbf{y}$. This entry conveys that to obtain $\mathbf{z_s}$ from $\mathbf{x_{e+s}}$, one needs to perform d edit operations; more specifically one needs to perform e deletions and s substitutions. Some of these substitutions will probably substitute the symbol x with the same symbol x, but the elementary edit distance associated with such substitutions, $d(x,x)$, is usually set to zero.

An example of an edit distance matrix is given in Figure 3; it shows the edit distance matrix for the two sequences $\mathbf{x} = 1010110111$ of length $N = 10$, and $\mathbf{z} = 1101011$ of length $M = 7$; the maximum number of consecutive deletions allowed is $E = 1$. The entry $*$ means that the corresponding prefix transformation is not permitted due to the value of E.

Different paths from the matrix position $W[0,0]$ to the bottom right position, $W[N-M, M]$ represent different clock control sequences because they represent deletions at different positions in the sequence $\mathbf{x}$. Consider the path illustrated in Figure 4; it represents that a symbol has been deleted between the positions 1 and 2, 4 and 5, and 5 and

e \ s	0	1	2	3	1	5	6	7
0	0	0	1	2	3	1	1	5
1	*	2	1	1	1	2	3	3
2	*	*	1	3	2	2	2	2
3	*	*	*	6	5	1	3	3

Figure 3: Edit distance matrix for the sequences $\mathbf{x} = 1010110111$ and $\mathbf{z} = 1101011$, where the maximum allowed number of consecutive deletions is $\mathrm{E} = 1$.

6 in the sequence $\mathbf{z}$. These deletions can be represented in the clock control sequence $\mathbf{c} = 1211221$.

e \ s	0	1	2	3	4	5	4	7
0	0 →	0	1	2	3	4	4	5
1	*	2	1 →	1 →	1	2	3	3
2	*	*	4	3	2	2	2	2
3	*	*	*	4	5	4	3 →	3

Figure 4: The same edit distance matrix as in Figure 3, now with an example path.

The different paths through the matrix also represent different ways to calculate the edit distance between $\mathbf{x}$ and $\mathbf{z}$. Equation (3.1) calculates $W[e, s]$ based on $w[e - e_1, s - 1]$ for the allowed value of e_1 that gives the minimum result. Thus the paths through the matrix represent different routes that can be taken in the edit distance computation. The paths that satisfy equation (3.1), that is the paths that give the minimum partial edit distances at all the positions they go through, are optimal paths.

However, one can also create suboptimal paths by allowing partial edit distances that are not optimal. For example, if $W[3, 6]$ in the example in Figures 3 and 4 was calculated based on $W[3, 5]$ instead of $W[2, 5]$, this would give a suboptimal partial edit distance in the position $W[3, 6]$. Paths that include the transition from $W[3, 5]$ to $W[3, 6]$ would then be suboptimal paths.

Suboptimal paths are needed in the search for the correct clock control sequence. This is because of the noise introduced when the plaintext is added to the keystream. The ciphertext sequence $\mathbf{z}$ is not the sequence produced by decimating $\mathbf{x}$ according to the clock control sequence, it is rather this sequence with added noise. However, the edit distance computations are performed between the ciphertext sequence and $\mathbf{x}$. Therefore the correct clock control sequence that transformed $\mathbf{x}$ to the keystream sequence $\mathbf{y}$ is not necessarily represented by an optimal path in the edit distance matrix for the sequences $\mathbf{x}$ and $\mathbf{z}$. However, that matrix is the only edit distance matrix available to the cryptanalyst.

The reconstruction of the clock control sequence is done by backtracking paths through the matrix, starting from $W[\mathrm{N} - \mathrm{M}, \mathrm{M}]$, and ending at $W[0, 0]$. However, it might not be

necessary to reconstruct the clock sequence for the full length of the intercepted ciphertext sequence. Let $\mathtt{pl}$ be the number of elements of $\mathbf{c}$ that are needed to be able to reconstruct the rest of the sequence using the generator, that is the length of the sequence that is needed to reconstruct the generator's initial state. For example, if the generator is an LFSR then $\mathtt{pl}$ would be the length of the LFSR, the linear complexity of the sequence. Then only the parts of the paths that go from $W[0,0]$ to the column $\mathtt{pl}$ need to be reconstructed. Since the paths are backtracked, this means that the search for the path that represents the clock control sequence should start in the column $\mathtt{pl}$, and everything to the right of this column can be disregarded in the search.

To keep track of the possible paths, and the different partial edit distances associated with each position in the graph based on different paths, some extra information should be stored for each matrix position. In [3] four vectors are associated with each entry in the matrix; these vectors will also be used here. Thus the following vectors are associated with each position $W[e,s]$:

1. The vector of primary pointers vp to the cells $W[vp[1],s-1],\ldots,W[vp[k],s-1]$ from which it is possible to arrive to the cell $W[e,s]$ with the minimum weight increment; $k \leq E+2$.
2. The vector of updated pointers vu to the cells $W[vu[1],pl],\ldots,W[vu[l],pl]$, through which it is possible to arrive to $W[e,s]$ with the minimum weight increment; $l \leq \min\{N-M+1, E(1+pl)\}$. Only cells in columns greater than $\mathtt{pl}$ (to the right of $\mathtt{pl}$ in the matrix) have this vector vu.
3. The vector of pointers ve to the cells $W[ve[1],s-1],\ldots,W[ve[j],s-1]$ from which it is possible to arrive to the cell $W[e,s]$ regardless of weight increment; $j \leq E+2$.
4. The vector of values vj of the edit distances corresponding to the elements of the vector vj; the cardinality of vj is j.

These vectors should be updated simultaneously with the partial edit distance computations. That is, for each position in the matrix, the partial edit distance should be calculated according to (3.1), and then information from that computation should be used to fill the vectors vp, vu, ve, and vj.

e \ s	0	1	2	3	4	$\mathtt{pl}=5$	6	7
0	0_0	0_0	1_0	2_0	3_0	4_0	$4_{0;\mathbf{0}}$	$5_{0;\mathbf{0}}$
1	$*$	2_0	$1;2_{0;1}$	$1;2_{1;0}$	$1;3_{1;0}$	$2;5_{1;0}$	$3;6_{1;0;\mathbf{1}}$	$3;5_{1;0;\mathbf{1}}$
2	$*$	$*$	4_1	$3;5_{1;2}$	$2;3_{1;2}$	$2_{1,2}$	$2;3_{2;1;\mathbf{2}}$	$2;4_{2;1;\mathbf{2}}$
3	$*$	$*$	$*$	6_2	$5;7_{2;3}$	$4;6_{2;3}$	$3;4_{2;3;\mathbf{2}}$	$3_{2,3;\mathbf{2}}$

Figure 5: The same edit distance matrix as in Figure 3, now with $\mathtt{pl}$ set to 5, and the associated vectors vp, ve, and vj for each entry, and also vu for columns 6 and 7. The subscript entries are the vectors vp and ve, to the left and right of the semicolon, respectively; the bold subscript entries to the far right in columns 6 and 7 are the vectors vu. The fullsize entries are the minimum partial edit distance to the left, and the vector vj to the right.

Figure 5 shows the edit distance matrix from Figure 4, but with the information from the vectors in addition to the optimal partial edit distance entry. The subscript to the left of the semicolon is the pointer vp, and the fullsized number to the left of the fullsize semicolon is the optimal partial edit distance. The subscript entry to the right of the semicolon is the pointer ve, and the fullsize entry to the left of the fullsize semicolon is the vector vj. The bold subscript entry to the right of the rightmost semicolon in the columns to the right of $\mathtt{pl}$ is vu. The different elements in one vector are separated by commas. If a semicolon is missing, then the entry to the right of that semicolon is empty. For example, in the cell $W[3,7]$ in the figure, the vectors ve and vj are empty, thus the semicolons to the left of these entries are left out; however, the vector vp has two entries in this cell, namely 2 and 3.

To be able to perform a k-best paths search on the matrix, it should be converted into a graph. The graph is created in the following way: A vertex is created in the graph for each cell in the matrix that belongs to the columns $\{0,\ldots,\mathtt{pl}\}$. Then, for each cell $W[e,s]$ that has a corresponding vertex, an edge is drawn from that vertex to each of the vertices corresponding to the cells that the vector $W[e,s].vp$ points to. The weight on each edge is the difference between the optimal partial edit distance entry in $W[e,s]$, and the optimal partial edit distance entry in the cell corresponding to the destination vertex. Also, an edge should be drawn between the vertex corresponding to $W[e,s]$, and each vertex corresponding to the cells pointed to by $W[e,s].ve$; the weight on each of these edges should be the difference between the corresponding entry in $W[e,s].vj$, and the optimal partial edit distance entry in the cell corresponding to the destination vertex.

A graph based on the matrix in Figure 5 is depicted in Figure 6. In the figure, $\mathtt{pl}$ is set to 5, and therefore the columns right of the column 5 in the matrix are not included in the graph. From the position $[3,5]$, there is an edge to the cell in row number 2 in the preceding column, with weight $5-3=2$, according to the pointer $W[3,5].vp$ and the minimum partial edit distance entries. There is also an edge from $[3,5]$ to row number 3 in the preceding column, with weight $7-6=1$, according to the pointer $W[3,5].ve$, the corresponding entry in $W[3,5].vj$, and the minimum partial edit distance of the destination cell.

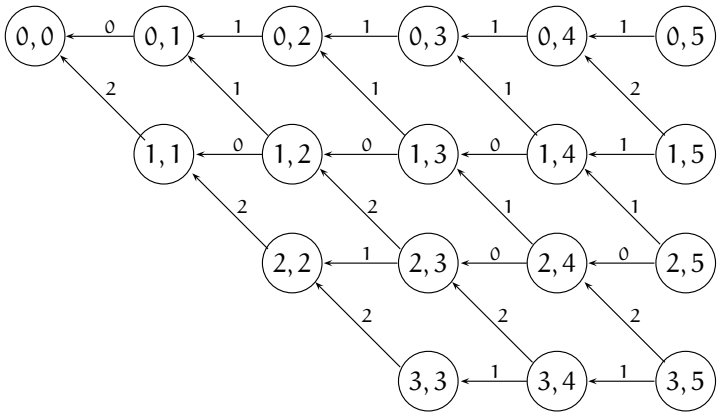


Figure 6: The graph generated from the matrix in Figure 5, with $\mathtt{pl}=5$.

A search for the path that represents the correct clock control sequence should start from each of the cells in $\mathtt{pl}$ that the correct path between $[0,0]$ and the bottom right

position can possibly pass through. These cells are those that correspond to vertices that would be reachable from the vertex representing the bottom right matrix position, if a graph were created for the whole matrix instead of just for the leftmost part of it. Let Δ be the difference between the far right column, M, and pl, that is $\Delta = M - pl$. Then the topmost cell in pl that is reachable from the bottom right position, $W[N - M, M]$, in the matrix is in the row $(N - M) - \Delta E$, where E is the maximum number of consecutive deletions. All the cells below this one are also reachable, provided that they have a valid partial edit distance entry, that is provided that the transformation from $\mathbf{x}_{e+s}$ to $\mathbf{z}_s$ is legal for the given E.

To set up for the k-best paths search, two extra vertices are added to the graph. These are the two source vertices. The first source vertex, called Src_1, has an edge going from itself to each of the vertices representing cells pointed to by $W[N - M, M].vu$. The second source vertex, called Src_2, has one outgoing edge to each of the remaining vertices representing cells in pl, that is those that are not pointed to by the edges going from Src_1. The weight on each of these edges is the difference between the optimal edit distance entry in $W[N - M, M]$ and the cell that the head of the given edge represents. If this difference is negative, then the weight is set to zero.

A k-best paths search that finds the k shortest paths between Src_1 and the vertex representing $W[0, 0]$ actually finds the k best paths from any of the cells in pl pointed to by $W[N - M, M].vu$, and $W[0, 0]$. Similarly, a k-best paths search from Src_2 to the vertex that represents $W[0, 0]$ finds the k shortest paths from any cell in pl that is not pointed to by $W[N - M, M].vu$, and $W[0, 0]$. The search for the correct clock control sequence should start by searching for paths from Src_1 to the vertex that represents $W[0, 0]$, and if there are less than k different paths between these vertices, the search should continue by finding the best paths from Src_2 to $[0, 0]$, until a total of k paths are found.

This prioritizing of cells in pl makes the search directed, and not blind. The search is directed like this because results experiments performed with the algorithm presented in [3] suggest that most of the correct clock control sequences are represented by paths that pass through the cells pointed to by vu.

Further, the different weights that are added to the edges going from Src_2 to the vertices representing the remaining cells in pl are a heuristic attempt to make up for the fact that some paths between $W[N - M, M]$ and $[0, 0]$ may have very little weight between pl and $[0, 0]$, but still be long paths because they can have high weight between $W[N - M, M]$ and pl. The weight the incoming edge from Src_2 is less than or equal to the length of the shortest path between $W[N - M, M]$ and the cell that that edge's head vertex represents.

In Figure 7 the graph from Figure 6 is shown, but with the added source vertices and their edges. Note that the edge from Src_2 to $[3, 5]$ has weight 0, since the difference between the partial edit distance entries in $W[3, 7]$ and $W[3, 5]$ is negative. It is now clear that the vertex $[0, 5]$ will not be in any of the desired paths, and thus it does not need to be part of the graph at all. This can of course be recognized when the graph is created, and then it will not be necessary to create vertices for those cells, both in pl and in the columns to the left of pl, if any, that are not reachable from the bottom right position.

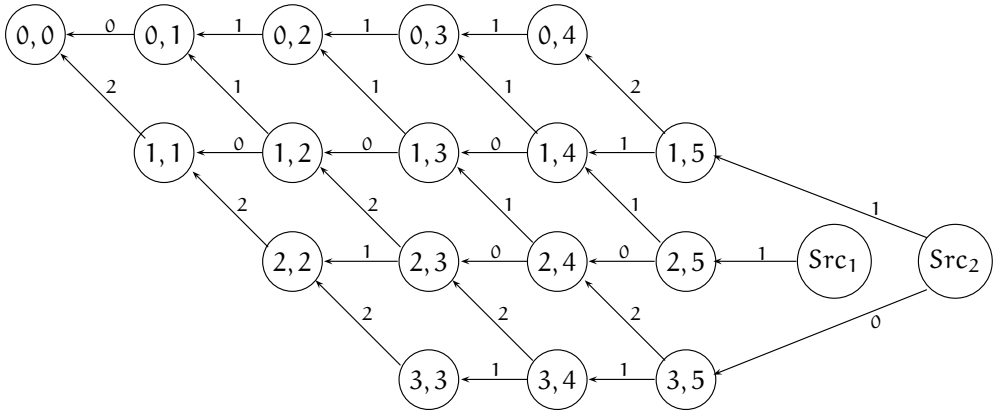


Figure 7: The same graph as in Figure 6, but with added source vertices. The vertex $[0,5]$ has been removed since none of the desired paths will pass through it.

3.3.2 Finding the k best paths

For each candidate initial state of the LFSR, a k-best paths search is performed through the graph created, in the manner described above, from the corresponding edit distance matrix. The k-best paths search follows Eppstein's algorithm, presented in [6]. This is an algorithm that finds the k shortest paths between a given source and destination in a graph with weighted edges.

Eppstein's algorithm is generally the best algorithm that is currently known for finding the k best paths. However some improvements have been suggested in [27], that often save some work in practice, but still give the same worst case complexity as Eppstein's original version. These improvements are not considered here, because the main goal of this project is to measure how many paths need to be reconstructed before the correct clock control sequence is found, not how fast each path can be found. For the same reason, the basic version of Eppstein's algorithm is used, even though improvements are described in the same paper that can speed up this basic version.

Given a graph G with n vertices and m weighted edges, the objective is to find the k paths between the source vertex s and the destination vertex t that has the lowest possible total weight. The first step is to build a single destination shortest path tree (SPT) for G, with t as destination. A shortest path tree for a graph G is a graph that shows the shortest path from each vertex in G to the given destination vertex. The shortest path between two vertices need not be unique, so there can exist several shortest path trees for each graph-destination pair. For the purpose of finding the k shortest paths, any SPT for the given graph and destination will do. A shortest path tree for the graph in Figure 7 is shown in Figure 8.

A shortest path tree can be built using Dijkstra's algorithm [30], in time $O(m + n \log n)$. However, since the graphs that are to be searched are all directed and acyclic, it is also possible to build the shortest path tree using topological search and dynamic programming; this can be done in time $O(m + n)$, and a good description of this algorithm can be found in [31]. The basic version of Eppstein's algorithm runs in time $O(m + n \log n + k \log k)$, so it doesn't matter for the overall complexity of the procedure which SPT algorithm is used. However, with the improvements suggested in the second part of [6], the k best paths can be found in time $O(m+n+k)$, given a shortest path tree.

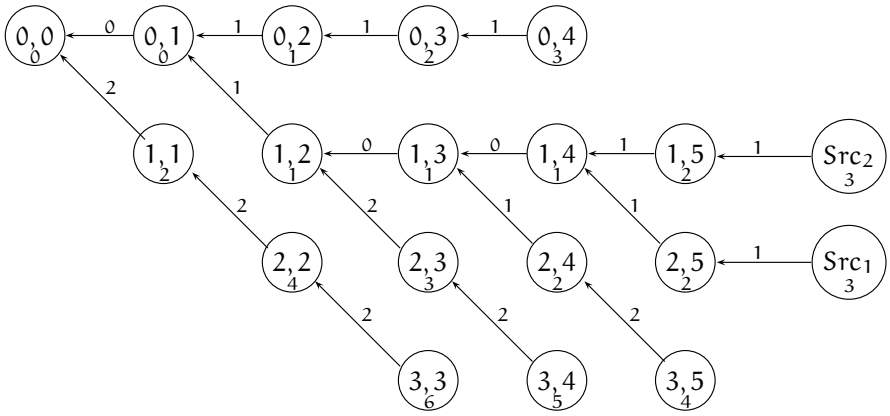


Figure 8: Shortest path tree for the graph as in Figure 7, with destination vertex $(0,0)$. Each vertex is labelled with its corresponding matrix position, and also the length of the shortest path from itself to the destination vertex.

Thus if the improved k-best paths algorithm is used, it is favourable to use the topological search method to build the SPT.

Given the shortest path tree for G, it is easy to construct a graph consisting of all the vertices in G, but containing only those edges that are not in the SPT; let the set of these edges be called $\mathit{sidetracks}(G)$. Each edge $e \in \mathit{sidetracks}(G)$ has an associated value $\delta(e)$, that measures how much weight is "lost" by being sidetracked along e instead of taking a shortest path to the destination t. $\delta(e)$ is given by

$$\delta(e) = l(e) + d(\mathit{head}(e), t) - d(\mathit{tail}(e), t), \tag{3.2}$$

where $l(e)$ is the weight associated with e, and $d(v_1, v_2)$ is the distance between the vertices v_1 and v_2, that is the length of the shortest path between v_1 and v_2; $\mathit{head}(e)$ and $\mathit{tail}(e)$ are the vertices at the head and tail of the edge e, respectively. The graph of sidetracks, with δ-values, corresponding to the SPT in Figure 8, is shown in Figure 9.

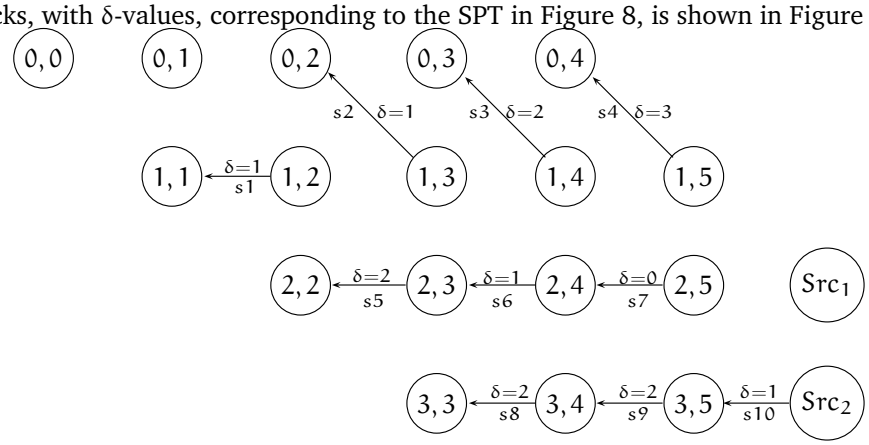


Figure 9: Graph of sidetracks corresponding to the SPT in Figure 8. The edges are labelled with unique identifiers for later use, and their δ-values, calculated according to (3.2).

Given any path p from s to t in G, the path forms a sequence of edges, where some of

the edges may be in the shortest path tree, and other edges might be in $sidetracks(G)$. The path is uniquely determined by its subsequence of edges in $sidetracks(G)$; let this subsequence be called $sidetracks(p)$. The shortest path from s to t is then represented by the empty sequence.

The main idea in Eppstein's algorithm is to create a heap for each vertex $v \in G$, of all the possible paths from v to t that are not in the SPT, and to use these heaps to find the k shortest paths. The first step in the process of creating these heaps is to, for each vertex v in G, collect all edges e that are in $sidetracks(G)$ and have tails at v in a set $out(v)$. Next, organize $out(v)$ in a heap $H_{out}(v)$, ordered by $\delta(e)$. $H_{out}(v)$ is a 2-heap, but it has the additional constraint that the root should only have one child. For the graph in Figure 7, $H_{out}(v)$ will have zero or one element for all vertices.

The next step is to create heaps $H_T(v)$ for each vertex v; these heaps will share some common structure with each other. The construction of these heaps starts at the destination vertex t, and then either a depth-first traversal or a breadth-first traversal of the shortest path tree, where $H_T(v)$ is created when v is visited, will result in correct construction of the remaining heaps. Each $H_T(v)$ is a min-heap sorted by the δ-values of its elements.

The heaps $H_T(v)$ consist of the root of $H_{out}(v)$, and of the elements of $H_T(w)$ for all vertices w on the shortest path from v to t, given by the shortest path tree. That is, the elements in $H_T(v)$ are the roots of $H_{out}(w)$ for all vertices w on the shortest path from v to t. To create $H_T(v)$, add the root of $H_{out}(v)$ to $H_T(u)$, where u is the next vertex on the shortest path from v to t in the SPT. The insertion procedure creates new copies of the nodes on the path in $H_T(u)$ that is updated due to this insertion, and the elements of $H_T(v)$ point to these new copies instead of the originals in $H_T(u)$. This way $H_T(u)$ is not changed in the creation of $H_T(v)$. Those elements of $H_T(u)$ that are not affected by the insertion need not be copied, and the elements of $H_T(v)$ can point directly to these unchanged elements of $H_T(u)$.

In Figure 10, $H_T(v)$ is shown for all vertices v in the graph in Figure 7. The heap $H_T(2,5)$ in the figure was created by inserting the sidetrack $s7$ into $H_T(1,4)$. $\delta(s7) = 0$, which places $s7$ at the root level of $H_T(2,5)$. This insertion changes the path to the elements $s1$ and $s2$ in $H_T(1,4)$, and therefore new copies are made of these elements for $H_T(2,5)$; the corresponding elements of $H_T(1,4)$ are left untouched. The insertion did not alter the path to $s3$, and therefore $H_T(2,5)$ does not need a new copy of this element; instead $s7 \in H_T(2,5)$ points to $s3 \in H_T(1,4)$. The unstarred element $(s3,2)$ in $H_T(2,5)$ in the figure is only meant as an empty pointer to the $s3$ in $H_T(1,4)$. Src_1 does not have any outgoing sidetracks, and thus $H_{out}(Src_1)$ is empty. Therefor $H_T(Src_1)$ is the same as $H_T(2,5)$; hence none of the elements of $H_T(Src_1)$ have stars.

Each element in $H_T(v)$ is the root of $H_{out}(w)$ for some w on the shortest path from v to t in G. Now connect the rest of $H_{out}(w)$ to its root in all the heaps $H_T(v)$ that this root is represented in. It is only necessary with one copy of each of these subtrees; all copies of its root can point to the same instance of the rest of the heap. The addition of these subtrees results in a new heap, $H_G(v)$, for each vertex $v \in G$.

Let the collection of the heaps $H_G(v)$, for all $v \in G$, be called $D(G)$. Also, let there be a map from vertices $v \in G$ to elements $h(v) \in D(G)$, where $h(v)$ takes v to the root of $H_G(v)$. Then the vertices reachable from $h(v)$ in $D(G)$ are exactly the vertices of $H_G(v)$, and these correspond to edges in $sidetracks(G)$ with tails on the shortest path from v to

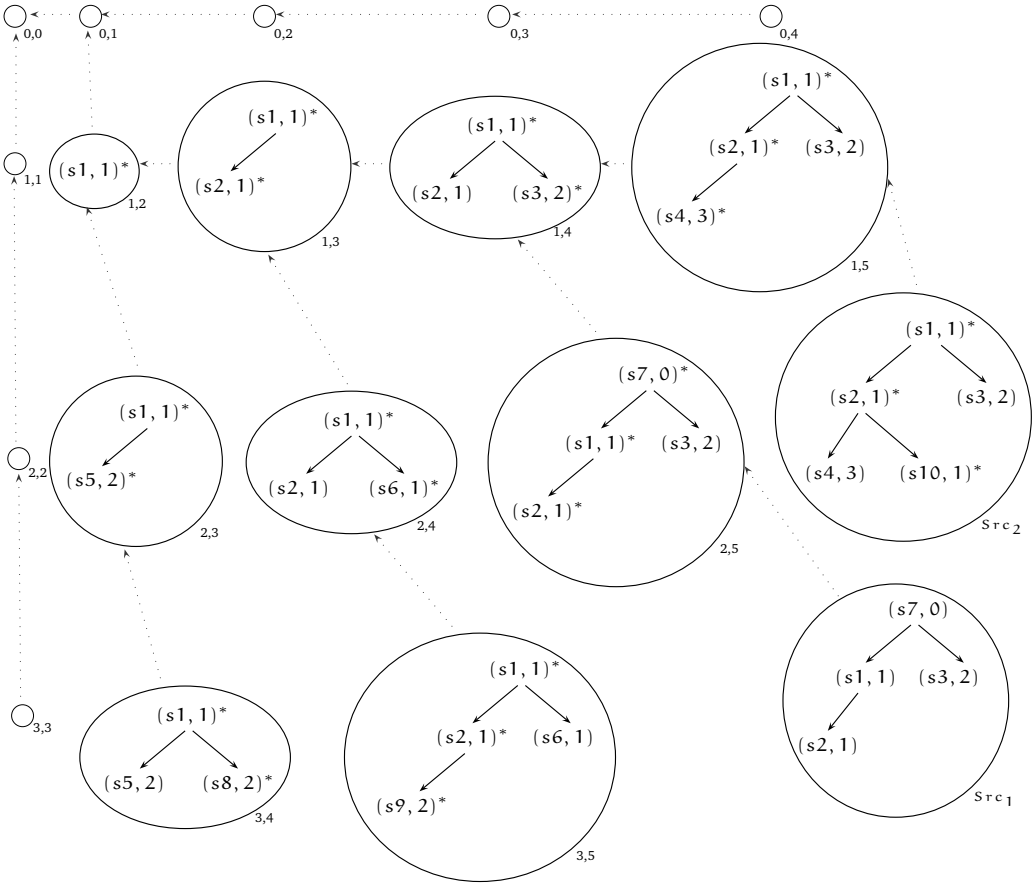


Figure 10: $H_T(v)$ for all vertices v in the graph in Figure 7. The nodes are labelled with the corresponding sidetrack label to the left of the comma, and with that sidetrack's δ-value to the right of the comma; a star symbolizes that a new copy was made of the element for that heap. The vertex each heap corresponds to in the graph in Figure 7 is noted at the bottom right of each heap, outside the ellipse. The dotted arrows show the edges in the SPT.

t. Hence the vertices reachable from $h(s)$ represent paths from s to t that differ from the shortest path by one single edge from $\mathtt{sidetracks}(G)$, and then the shortest path from the head of this sidetrack to t.

$D(G)$ for the graph in Figure 7 is shown in Figure 11. Here the shared copies of some elements in $H_T(v)$ and $H_T(u)$ for $v \neq u$ are more obvious than in the separate drawings of $H_T(v)$ in Figure 10. Note that $h(\mathrm{Src}_1) = h(2,5)$, since $H_T(\mathrm{Src}_1) = H_T(2,5)$.

$D(G)$ can be augmented, with additional edges, to form a *path graph* $P(G)$ that can represent all paths from s to t, not just those that differ from the shortest path by one sidetrack edge. To construct $P(G)$, start with $D(G)$, and add one additional vertex: the root vertex $r(s)$. Keep all the vertices and edges from $D(G)$, and assign weights to the edges in the following way: The directed edge from u to v should have the weight $\delta(v) - \delta(u)$; these edges are called *heap edges*. Now, for each vertex $v \in P(G)$ that corresponds to an edge in $\mathtt{sidetracks}(G)$ that connects the vertices u and w in G, create a new edge from v to $h(w)$ in $P(G)$, and give it weight $\delta(h(w))$. Let these edges be called *cross edges*. Finally, create an *initial edge* from $r(s)$ to $h(s)$, with weight $\delta(h(s))$. The path graph for

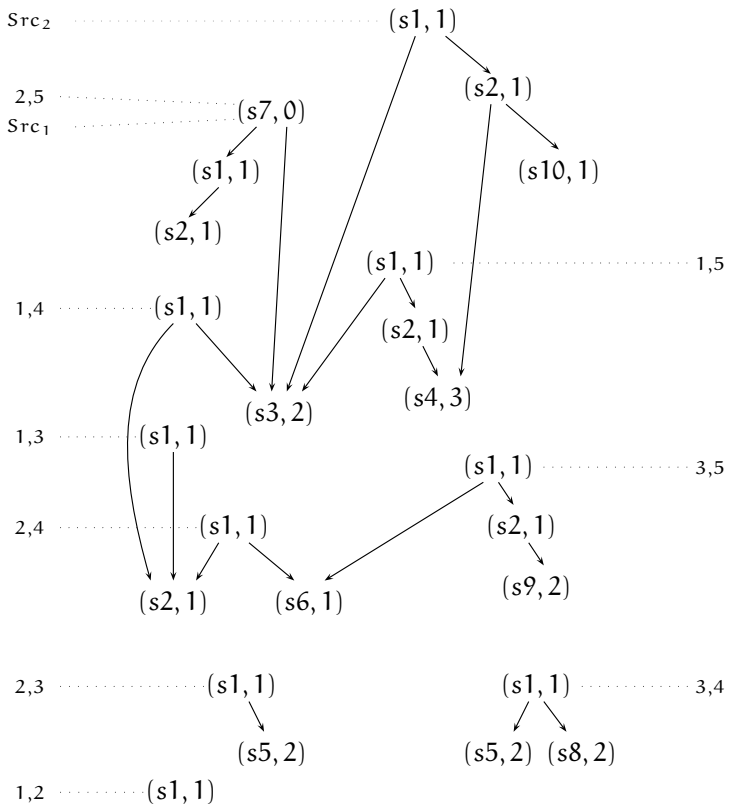


Figure 11: $D(G)$ for the graph in Figure 7. The mapping h is represented by dotted lines going from labels for v at both sides and to the elements $h(v)$ in D(G).

Figure 7, representing paths between Src_1 and $(0, 0)$, is shown in Figure 12.

Now, to find the k shortest paths from s to t in G, simply perform a breadth first search of $P(G)$. The k shortest paths of $P(G)$ will correspond to the k shortest paths between s and t in G, and a path p' in $P(G)$ can be translated into an s-t path p in the following way: The tails of the crossedges in p' and the destination vertex of p' correspond to the edges in G that form the sequence $sidetracks(p)$. If p' consists of $r(s)$ only, then $sidetracks(p)$ is empty, and p' represents the shortest path between s and t. If $sidetracks(p)$ is not empty, then the path p can be constructed by connecting the edges in $sidetracks(p)$ using the shortest paths between two successive sidetracks, and by taking the shortest path from s to the tail of the first sidetrack, and the shortest path from the head of the last sidetrack to t. The length of the path p is the length of the shortest path between s and t, plus the length of the path $p' \in P(G)$.

For example, the 5 shortest paths in the path graph in Figure 12 are:

p'_1 : $r(Src_1)$
p'_2 : $r(Src_1) \longrightarrow s7$
p'_3 : $r(Src_1) \longrightarrow s7 \dashrightarrow s1$
p'_4 : $r(Src_1) \longrightarrow s7 \longrightarrow s1$
p'_5 : $r(Src_1) \longrightarrow s7 \dashrightarrow s1 \longrightarrow s6$,

where the different arrows represent the different types of edges in the same way as in Figure 12. p'_1, p'_2, p'_3 have lengths 0, and p'_4, p'_5 have lengths 1. There are several more

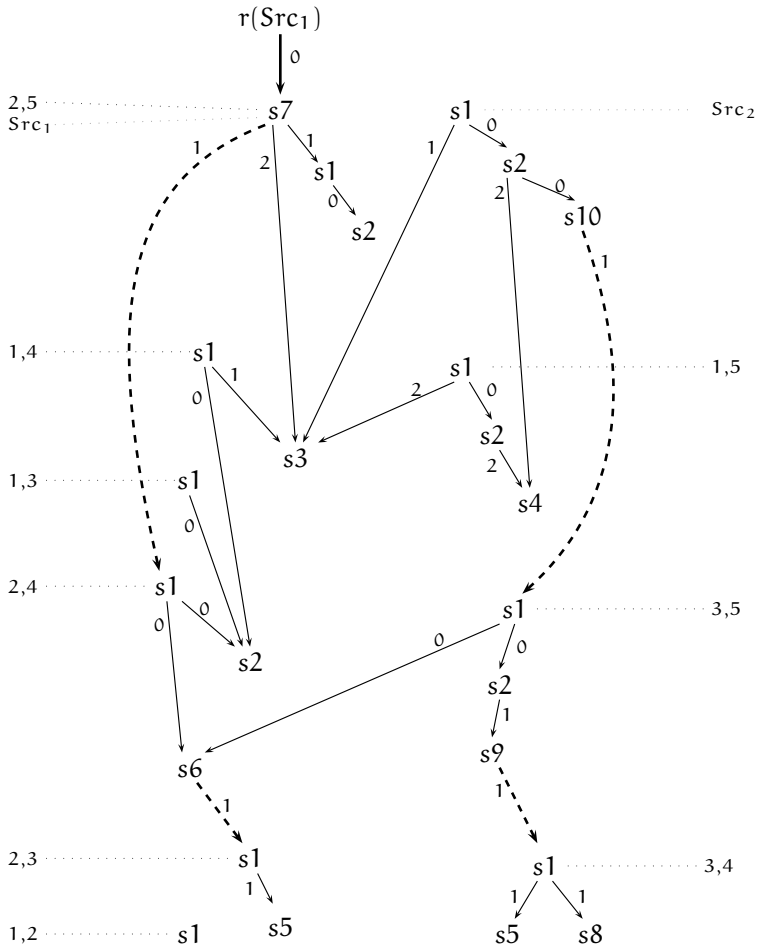


Figure 12: $P(G)$ for the graph in Figure 7, with source Src_1 and destination $(0,0)$. Heap edges are represented by solid arrows, cross edges by bold dashed arrows, and the initial edge by a solid bold arrow. Each edge is labelled with its weight.

paths in the path graph that have length 1, and it does not matter which two of these are picked to be among the 5 best paths; the breadth-first search terminates when it is certain that it will not find any paths of shorter length than the longest path included in its current set of 5 shortest paths. The paths $p'_1, \ldots, p'_5$ translate into these five paths from the graph in Figure 7:

p_1 : $Src_1 \longrightarrow [2,5] \longrightarrow [1,4] \longrightarrow [1,3] \longrightarrow [1,2] \longrightarrow [0,1] \longrightarrow [0,0]$
p_2 : $Src_1 \longrightarrow [2,5] \longrightarrow [2,4] \longrightarrow [1,3] \longrightarrow [1,2] \longrightarrow [0,1] \longrightarrow [0,0]$
p_3 : $Src_1 \longrightarrow [2,5] \longrightarrow [2,4] \longrightarrow [1,3] \longrightarrow [1,2] \longrightarrow [1,1] \longrightarrow [0,0]$
p_4 : $Src_1 \longrightarrow [2,5] \longrightarrow [1,4] \longrightarrow [1,3] \longrightarrow [1,2] \longrightarrow [1,1] \longrightarrow [0,0]$
p_5 : $Src_1 \longrightarrow [2,5] \longrightarrow [2,4] \longrightarrow [2,3] \longrightarrow [1,2] \longrightarrow [0,1] \longrightarrow [0,0]$.

The paths p_1, p_2, p_3 have length 3, and p_4, p_5 have length 4. The 5 shortest paths from the column pl to the matrix position $[0,0]$ are the paths $p_1, \ldots, p_5$, but starting from pl instead of the source vertex.

Note that the source vertex for the search is not specified until the root vertex is added in $P(G)$. Thus it is very easy to switch source vertex: just remove the outgoing edge from

the root node, and add a new edge, going to the heap representing the new source vertex in G. Thus, when switching source vertex from Src_1 to Src_2, one should simply redirect the root edge to point to the root of $H_T(Src_2)$ instead of $H_T(Src_1)$.

The algorithm described here finds the k best paths in a graph with m vertices and n edges in time $O(m + n \log n + k \log k)$; this is Eppstein's basic algorithm presented in [6]. Eppstein's improved version runs in time $O(m + n + k)$, given a shortest path tree in the input. The improvements are namely faster heap selection, and faster heap construction.

Faster heap selection speeds up the part of the algorithm that selects the k shortest paths from $P(G)$. Instead of a breadth first search, a method due to Frederickson is used [32]; this is a method that finds the k smallest weight vertices in any heap in time $O(k)$.

Faster heap construction speeds up the construction time for $H_T(v)$. This improvement actually addresses the abstract problem, given a tree T with weighted nodes, of constructing a heap $H_T(v)$ for each vertex v of the other nodes on the path from v to the root of T. By the introduction of dummy nodes with large weights, the assumptions that T is a binary tree and that its root t has indegree one, hold without loss of generality. It can also be assumed that all vertex weights in T are distinct, by use of a suitable tie-breaking rule. Now, based on these assumptions, some further techniques by Frederickson, introduced in [33], can be used to construct the heaps $H_T(v)$ in total time $O(n)$.

An improvement was also suggested by Jiménez and Marzal in [27]. The idea here is to only build $H_T(v)$ for those v that are needed to find the k best paths between the two given nodes. This improved version, called lazy version, has the same worst-case complexity as Eppstein's version, since in the worst case the lazy version will end up building $H_T(v)$ for all v. However, it is often not necessary to build all the heaps, and thus the lazy version often runs faster than Eppstein's version in practice.

3.4 Attack summary

This section briefly summarized the attack described in this chapter.

Before the attack can start, one must have the following information about the system that is to be attacked:

- The feedback polynomial and tap positions for the LFSR.
- Enough information about the general subgenerator that generates the clock control sequence, so that the full clock control sequence can be reconstructed if its initial state is recovered. That is, if the generator is an LFSR, the feedback polynomial should be known, and also the specifics about how the output from the LFSR is converted into a clock control sequence.
- An intercepted ciphertext sequence, produced by the target system, of length $M \geq pl$. pl is given by the specifics of the general subgenerator.

Once the above information is collected, one can perform the attack in the following manner:

1. Estimate N, the length of the LFSR's output sequence that produced the intercepted ciphertext sequence; N is set to its mathematical expectation, based on M and E, the maximum number of consecutive deletions. E is also given by the specifics about the general subgenerator.
2. Determine the threshold t that is used to accept the candidate initial states for the

LFSR. t is determined by first selecting values for the probabilities P_m of 'missing the event' and P_f of 'false alarm', and then calculating the edit distances between the ciphertext sequence and the output sequences produced by $1/P_m$ initial states of the LFSR; t is set to be greater than all the edit distances calculated in this process.

3. Find the candidate initial states. Calculate the edit distance between the output sequences produced by all the initial states for the LFSR that were not used in the previous step; if the edit distance is less than t, then include the initial state in the set of candidate initial states. All the initial states that were used to determine t should also be in the set of candidate initial states.
4. Reconstruct clock control sequences. For each candidate initial state:
   - Generate the graph corresponding to its edit distance matrix.
   - Find the k best paths from the source vertex Src_1 to the vertex corresponding to the matrix cell $W[0,0]$; if only $n < k$ paths exist between Src_1 and $[0,0]$, then continue by finding the $k - n$ best paths from Src_2 to $[0,0]$, to get a total of k paths.
   - Convert the recovered paths into clock control sequences.

These steps will give a set of candidate initial states for the LFSR, and for each of those a set of candidate initial states for the clock control sequence generator. To find the pair of initial states that gives the correct solution, one can either try all the candidate pairs, or run the attack again on other ciphertext segments. If none of the candidate pairs give the correct solution, then one can return to step 4, increase k and run the k-best paths algorithm again, until the correct solution is found.

4 Experimental work

The purpose of this thesis is to introduce the idea of using k-best paths algorithms in the second phase of an edit distance attack against irregularly clocked stream ciphers, and to gain knowledge about the performance of the proposed k-best paths algorithm in such an attack. The attack procedure was introduced in Chapter 3, and in this chapter experiments are used to gain knowledge about the performance of that attack. The exact purpose of the experiments is outlined in section 4.1, and the experimental strategy is introduced in section 4.2; in section 4.3 the experimental procedure is explained, and the results are reported in section 4.4.

4.1 Purpose

The main purpose of the experimental work is to gain knowledge about the performance of the proposed attack method, and especially the k-best paths algorithm in such a setting. More specifically, the questions that the experiments should answer are:

1. Is the proposed attack able to cryptanalyze the desired class of stream ciphers; that is does it find the solution?
2. How does the number of paths that the k-best paths algorithm extracts before it finds the correct clock control sequence depend on pl and the correlation parameter p?
3. How does the proposed k-best paths algorithm perform in the attack, compared to the depth-first search introduced in [3]?

The importance of question 1 should be obvious; if the proposed attack does not find the solution it has failed miserably, and the performance of the k-best paths algorithm does not matter at all. Question 2 is important, because the feasibility of the attack is very closely tied to the number of paths that need to be extracted before the solution is found. The third question is based on the fact that the proposed attack is the same as the attack proposed in [3], except for the k-best paths algorithm, that replaces the depth-first search in [3]. Thus it is very important to know how this new version of the attack performs compared to the original one.

4.2 Strategy

The strategy used to answer the questions in the previous section is to simulate the operation of a stream cipher that uses a clock controlled keystream generator, and then to perform the attack on the ciphertext that the simulation produced.

To answer question 2, experiments that vary the parameters pl and p, and that measure the number of extracted paths for each pair of parameters were performed. pl took on the values $\{10, 15, 20\}$; larger values were not feasible because the experiments would then take too much time. The experiments were performed on regular office computers, since such computers were the only computers available for this project. The values of p that were used are $\{0.00, 0.05, \ldots, 0.45\}$, that is all values between 0.00 and 0.45, in increments of 0.05. Since $0 \leq p < 0.5$, the experiments basically covered the whole range

of p, quantized into 10 values.

Question 3 was addressed by performing the depth-first search described in [3] in addition to the k-best paths search, for all the experiments that are outlined in the previous paragraph. The number of paths extracted by this depth-first search was reported, so that the number of paths extracted by the two algorithms can be compared.

The first question is answered by all the experiments outlined above, by just confirming that the attack procedure finds the correct solution, that is the clock control sequence that was produced by the simulation of the stream cipher.

The simulations generated a random initial state for both the subgenerators of the keystream generator for each experiment. These initial states were picked at random both to provide a realistic simulation of how such a stream cipher operates, and to introduce control over these parameters in the experiment, by randomizing them. The plaintext sequence was also generated at random, but weighted by the correlation parameter p.

To increase the significance of the results, as many experiments as possible should be performed. 100 experiments were conducted for each pair of parameters pl, p; this was as many experiments as the time constraints allowed.

4.3 Procedure

The experiments were performed as simulations of the system depicted in Figure 13. The clock control sequence generator in this system is an LFSR that generates a binary sequence. A '0' in this sequence corresponds to a '1 in the clock control sequence **c**, and a '1' in the binary sequence corresponds to a '2' in **c**. All the experiments were done with $E = 1$, so a binary representation of the clock control sequence was suitable.

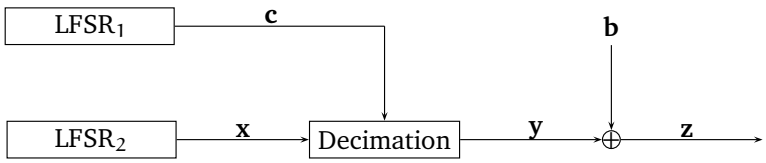


Figure 13: The system that was simulated. $LFSR_1$ produces a binary sequence, where a '0' means that the decimation function should step 1 step, and a '1' means that it should step 2 steps in **x** before outputting the next bit of **y**.

$LFSR_1$ was simulated as an LFSR of length 4, with feedback polynomial $1+D+D^4$, and the output taken from position 1. A model of this generator is shown in Figure 14. The LFSR that was used for $LFSR_2$ had feedback polynomial $1 + D^2 + D^{11} + D^{12} + D^{15}$, thus it had length 15, and the feedback was taken from positions 2, 11, 12, 15. The output from $LFSR_2$ was also taken from position 1. **b** was generated as a random binary sequence, but weighted by the given correlation parameter p, so that $P(b_n = 1) = p$ for all bits $b_n \in \mathbf{b}$.

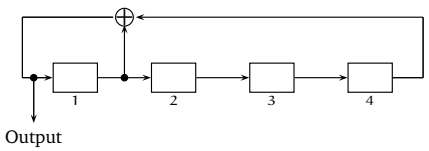


Figure 14: Model of $LFSR_1$ as it was used in the experiments.

For each experiment the values of M, pl, and p were given at the start of the experiment. The clock control generator that was used, $LFSR_1$, gives $pl = 4$. However, the only real function the value of pl has in the experiments is to specify how many elements of the clock control sequence should be reconstructed, so therefore the value of pl was varied as explained in section 4.2 even though the clock control generator was always $LFSR_1$.

Given those three values, and the feedback polynomials and tap positions of the LFSRs, each experiment first generated random initial states for the LFSRs. Then these initial states were used together with the specifics about the LFSRs to produce the sequences **x** and **c**, decimate **x** according to **c**, and then bitwise add the resulting keystream sequence **y** to the plaintext sequence **b**; **b** was generated according to the correlation parameter p. M bits of the resulting ciphertext sequence **z** were captured to be used as the intercepted ciphertext.

The goal of this thesis was to study the use of k-best paths algorithms in clock control reconstruction; the procedure of determining the candidate initial states for $LFSR_2$ has been thorougly studied before, for example in [2]. Therefore only the second phase of the attack described in chapter 3 was performed in the experiments. Each experiment thus assumed that the correct initial state for $LFSR_2$ was already found, and it simply received the first N bits of the correct sequence **x** from the simulation of $LFSR_2$, where $N = 3M/2$ is the mathematical expectation for N.

The first step in the experiments was to fill the edit distance matrix, including the vectors vp, vu, ve, vj, based on the edit distance between the first N bits of **x** and the first M bits of **z**. Next, the graph corresponding to the pl first columns of that matrix was created, with its two source vertices, as explained in section 3.3.1. Then the k-best paths algorithm was used to extract one path at a time, first from Src_1, and then from Src_2, until the path corresponding to the correct clock control sequence was found. The number of extracted paths, k, was reported for each experiment. Source code for the implementation of the k-best paths algorithm is listed in Appendix A.

To be able to efficiently extract one path at a time, always extracting the shortest path that had not yet been extracted, a priority queue was used in the breadth-first search of $P(G)$. This priority queue held paths of $P(G)$, keyed by path length. It was initialized to hold only one element, namely the root path, representing the shortest path from source to destination. Then, to extract the shortest path that had not yet been extracted, the minimum element from the priority queue was extracted, and all its children in $P(G)$ were added to the queue. Each path that was extracted was directly converted to its corresponding clock control sequence, and checked against the correct clock control sequence. When the solution was found, the number of extracted paths was reported, and the search terminated.

The depth-first search described in [3] was also performed on all the edit distance matrices, and the number of extracted paths was reported for each matrix.

4.4 Results

The proposed attack found the correct clock control sequence in all the experiments that were performed. More detailed results are reported below.

4.4.1 Dependence of k on p and pl

It is desirable for the cryptanalyst that the number of paths k that are extracted before the correct path is found is as small as possible. The number of extracted paths was reported for each of the experiments described above, and for each pair of parameters pl, p, the average number of extracted paths was calculated. The results are presented in Table 1.

p	$pl = 10$	$pl = 15$	$pl = 20$
0.00	11	52	200
0.05	35	623	13588
0.10	71	1168	30470
0.15	117	2383	51728
0.20	162	3318	90681
0.25	203	4680	136302
0.30	273	6946	198791
0.35	321	9009	271941
0.40	394	10696	331923
0.45	444	14182	427830

Table 1: Average number of extracted paths for the performed experiments.

Figures 15, 16, and 17 show the dependence of k on p for $pl = 10, 15, 20$, respectively. For $pl = 10$, the dependence of k on p is very close to linear; the average number of extracted paths is close to $1000p$ for all of the p-values that were used in the experiments. For $pl = 15$ and $pl = 20$ on the other hand, it looks more like the dependence of k on p is polynomial, of degree 2. However, the graphs are not quite steep enough to give a clear second-degree impression, so these dependencies are also quite close to linear.

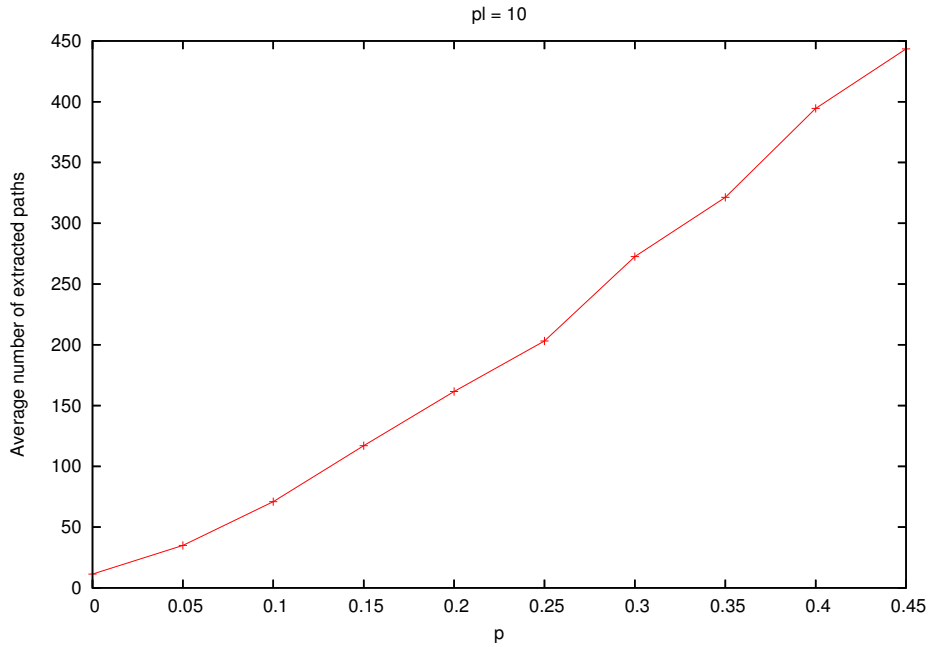


Figure 15: Average number of extracted paths k for the different values of the correlation parameter p that were tested; $pl = 10$

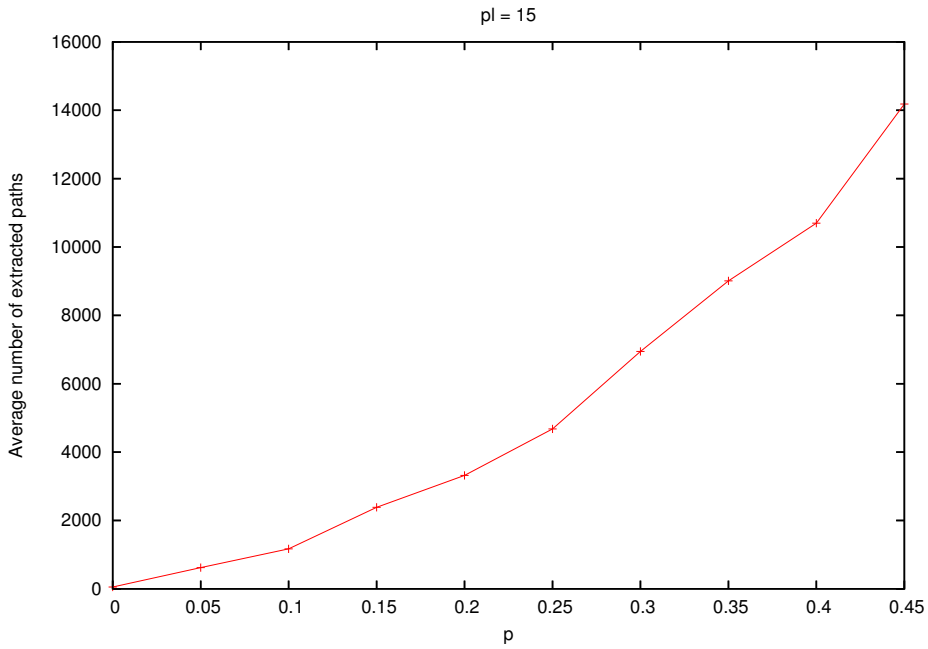


Figure 16: Average number of extracted paths k for the different values of the correlation parameter p that were tested; $\mathtt{pl} = 15$

The dependence of k on pl is presented in Figure 18. With only tree different values of pl one cannot draw too many conclusions about how k depends on pl, but for the values that were tested there is a clear indication that the dependency is exponential. The scale in Figure 18 is logarithmic, and for most of the tested values of p, the results line up on almost completely straight lines, especially for high values of p.

4.4.2 Comparison of k-best paths and depth-first search

For each of the experiments, the depth-first search described in [3] was also performed, and the number of paths it extracted before finding the correct one was reported together with the number of extracted paths for the k-best paths algorithm. Table 2 shows the results for both the algorithms together.

The dependency on p can be compared for the two algorithms in Figures 19, 20, and 21. The three graphs show that the k-best paths approach goes through more paths than the depth-first search on average, before the correct solution is found. However, the gap between the two algorithms is relatively small, and the number of extracted paths seems to depend on p in very much the same way for both algorithms. Also, the k-best paths approach performs better than the depth-first search for $\mathtt{p} = 0$ for all pl, and for $\mathtt{pl} = 10$ and low values of p. The general trend seems to be that the gap between the two algorithms, favouring the depth-first search, increases slightly when p increases.

In Figure 22, the average number of extracted paths is plotted vs. pl for both methods. Again, the depth-first search performs somewhat better than the k-best paths search, except for when $\mathtt{p} = 0$, but the graphs for the two algorithms are very similar in form, and the distance between them is low.

The depth-first search from [3] starts by searching through all optimal paths, and if the solution is not found, it searches through all paths whose length is the length of the optimal paths plus one, and then plus two, and so on, until the correct solution is found.

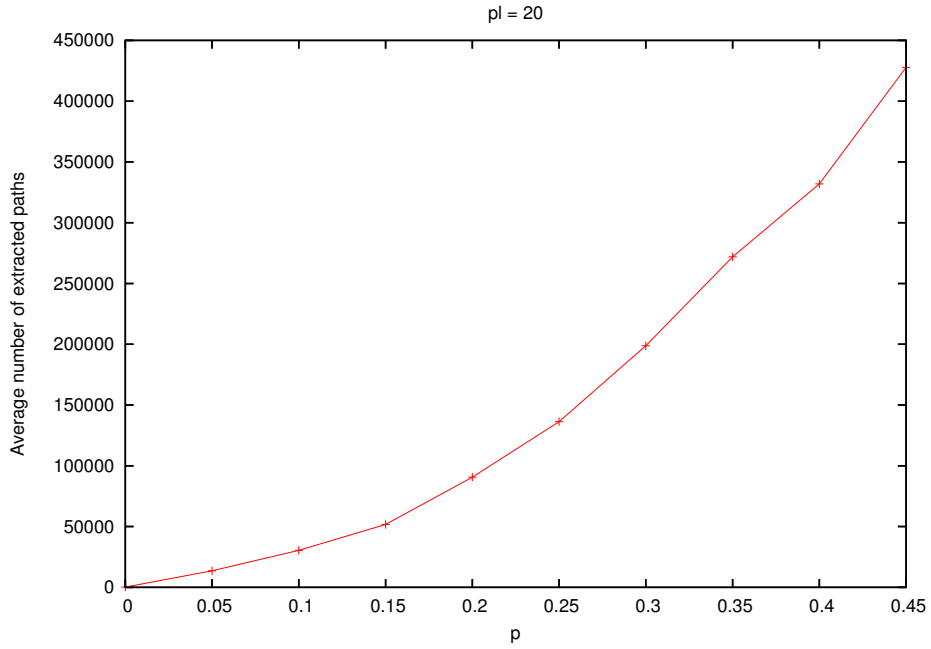


Figure 17: Average number of extracted paths k for the different values of the correlation parameter p that were tested; pl = 20

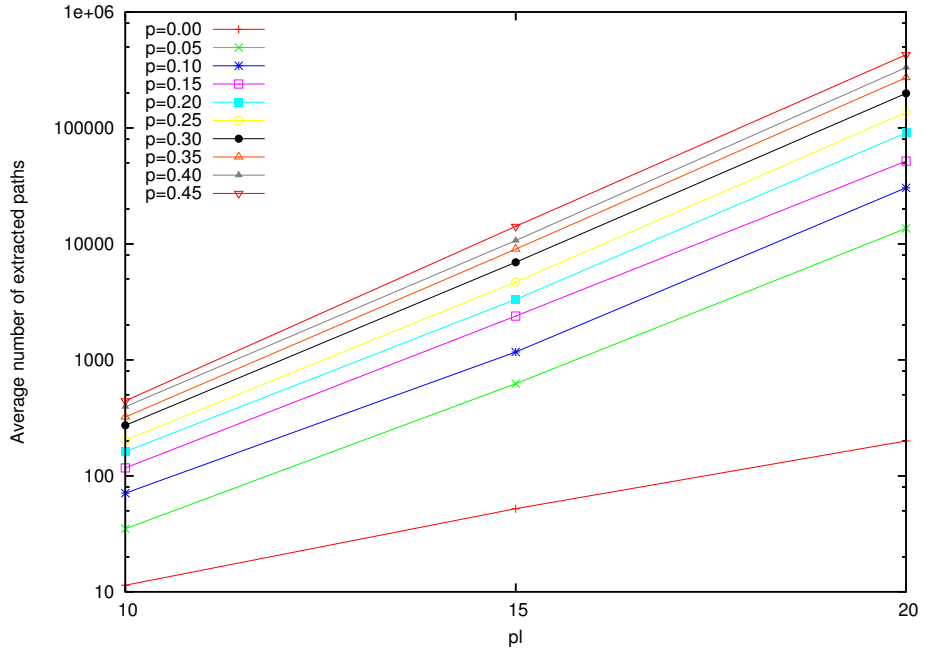


Figure 18: Average number of extracted paths k for the different values of pl that were tested. Note that the scale on the vertical axis is logarithmic.

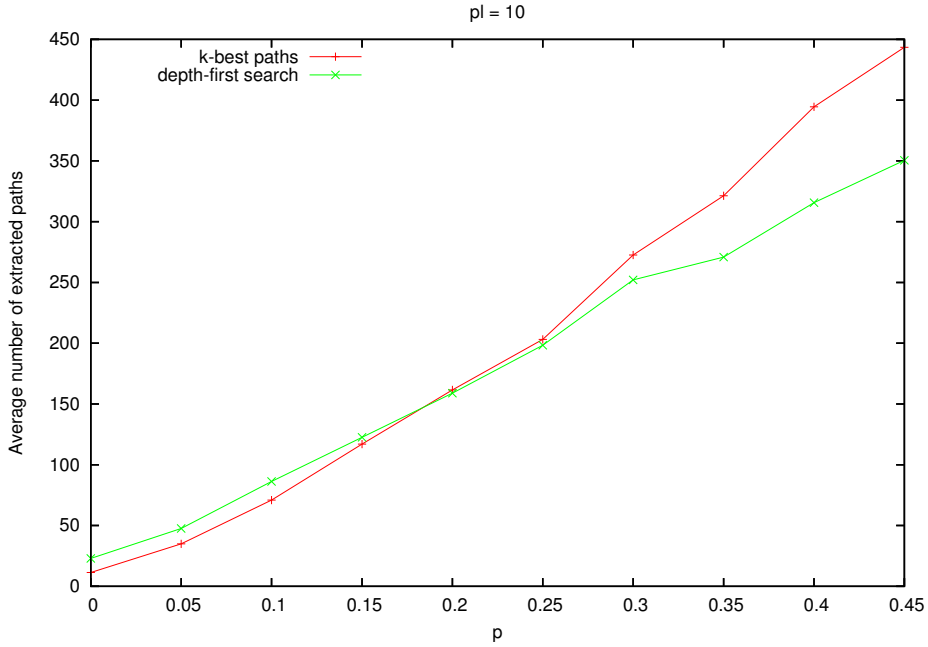


Figure 19: Average number of extracted paths for both k-best paths and depth-first search; $\mathtt{pl} = 10$

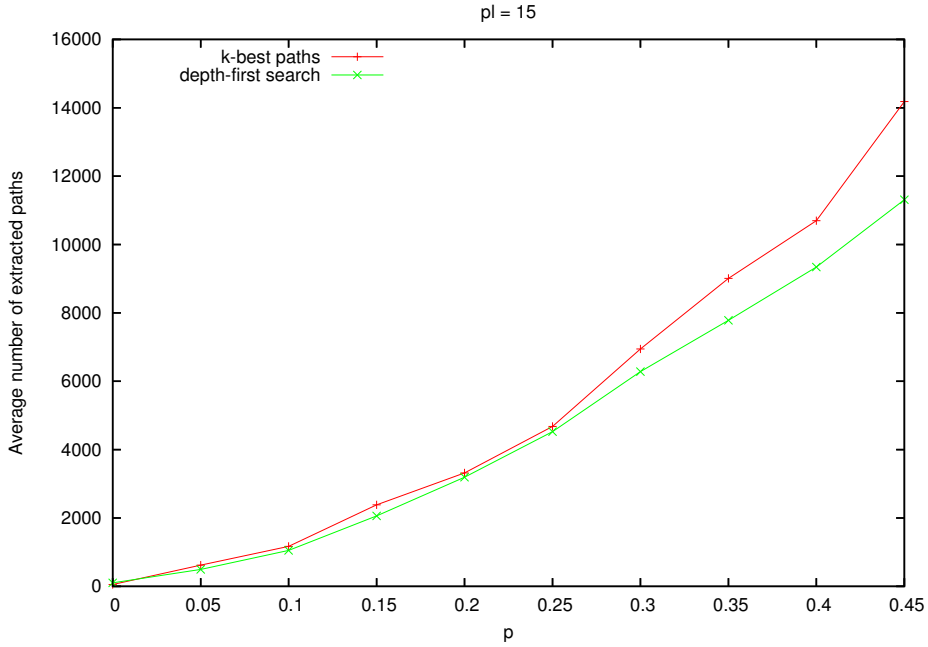


Figure 20: Average number of extracted paths for both k-best paths and depth-first search; $\mathtt{pl} = 15$

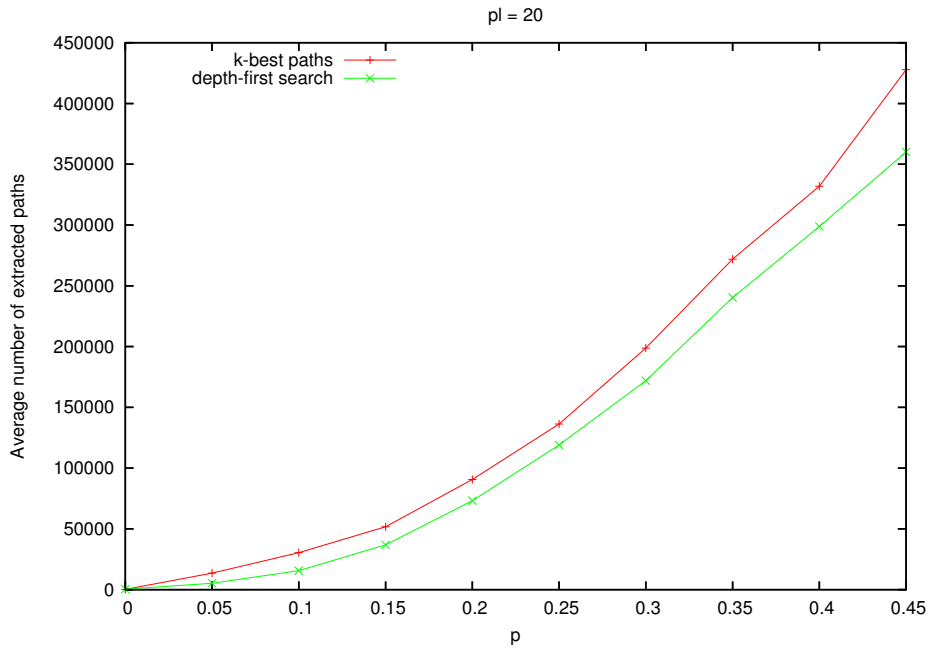


Figure 21: Average number of extracted paths for both k-best paths and depth-first search; $\mathtt{pl} = 20$

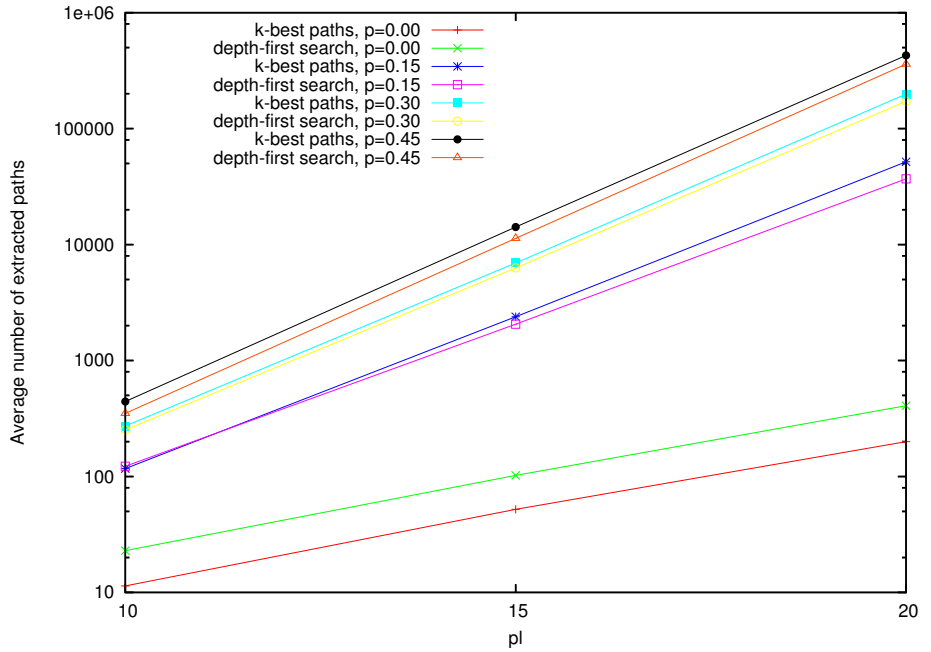


Figure 22: Average number of extracted paths for $\mathtt{p} = 0.00, 0.20, 0.40$, vs. $\mathtt{pl}$ for both algorithms. Note that the scale on the vertical axis is logarithmic.

	$pl = 10$		$pl = 15$		$pl = 20$	
p	DF	K	DF	K	DF	K
0.00	23	11	102	52	408	200
0.05	48	35	495	623	5214	13588
0.10	86	71	1050	1168	15661	30470
0.15	123	117	2060	2383	36892	51728
0.20	159	162	3191	3318	73218	90681
0.25	198	203	4524	4680	119019	136302
0.30	252	273	6283	6946	172017	198791
0.35	271	321	7784	9009	240318	271941
0.40	316	394	9343	10696	298816	331923
0.45	350	444	11312	14182	360129	427830

Table 2: Average number of extracted paths for the depth-first search (DF) and k-best paths (K) algorithms.

The k-best paths method used in this thesis searches through all paths that start in vu first, and then it starts a new search for paths that start outside of vu if the solution was not found in the first search. In the second search the starting points for the paths are weighted based on the difference between the optimal partial edit distance of each possible starting point and the bottom right position in the matrix.

The results presented above indicate that the depth-first search finds the correct solution faster in the general case, but the difference in the order the paths are searched through in the two methods suggest that the k-best paths approach should find the solution faster than the depth-first search when the solution is represented by a path that starts from a cell in vu.

Figures 23, 24, and 25 show the average number of paths the two algorithms went through whenever the solution was represented by a path starting from vu, and Figures 26, 27, and 28 present results for solution paths starting outside of vu. As expected, the k-best paths algorithm performs better than the depth-first search for paths starting from cells in vu, and the depth-first search performs better for paths starting form cells not in vu.

The percentage of solution paths that start from a cell in vu, and the corresponding percentage of solutions that do not start from a vu cell is listed in Table 3. The overall percentage of solution paths that start from vu is actually greater than 50%, thus the k-best paths algorithm performs better than the depth-first search in more than 50% of the experiments; still the depth-first search has a better overall performance than the k-best paths.

All comparisons and figures so far have been based on the computed average of extracted paths for each pair of parameters pl, p, for the two algorithms. Now Figures 29, 30, 31, and 32 show the actual number of extracted paths for each of the 100 experiments that were performed for $pl = 20$ and $p = 0.00, 0.15, 0.30, 0.45$, respectively; similar graphs for all the performed experiments are presented in Appendix B. Note that the main difference between the two algorithms is that the peaks for the k-best paths algorithm are higher, and come more often than those for the depth-first search.

The k-best paths algorithm even performs better than the depth-first search in a sur-

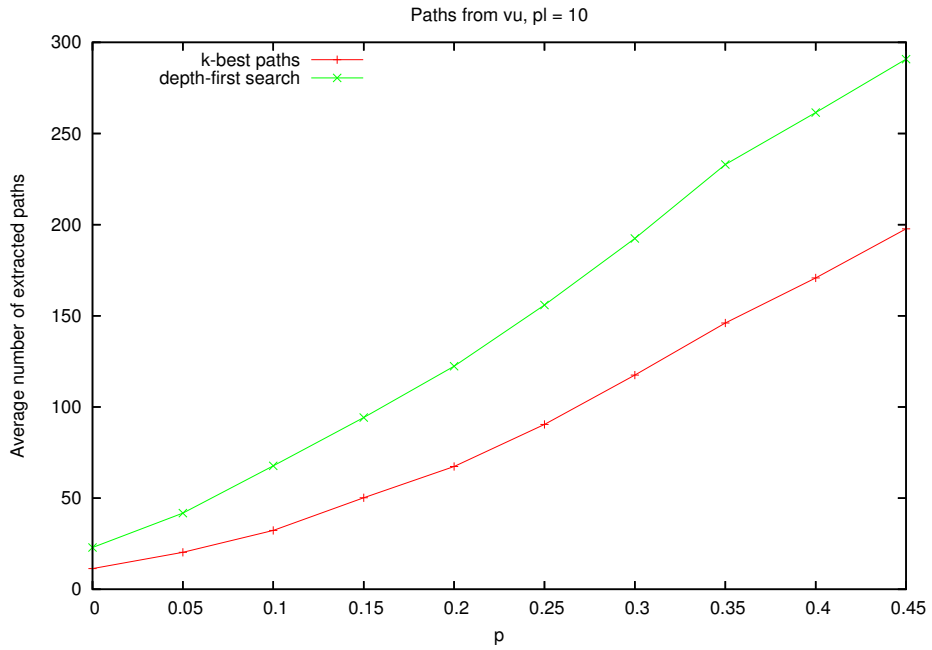


Figure 23: Average number of extracted paths for paths starting from cells in vu; $\mathtt{pl} = 10$

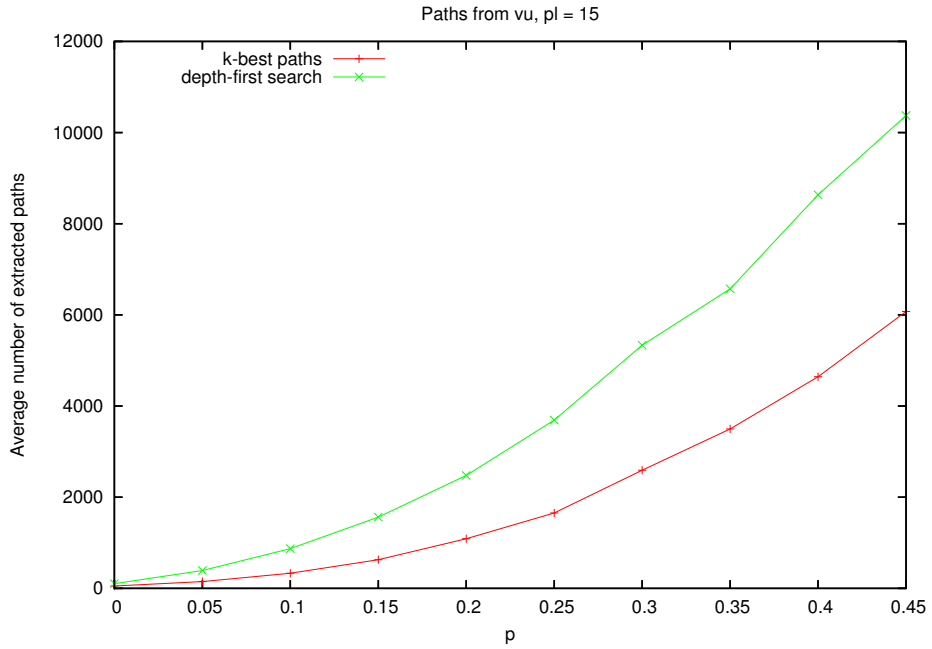


Figure 24: Average number of extracted paths for paths starting from cells in vu; $\mathtt{pl} = 15$

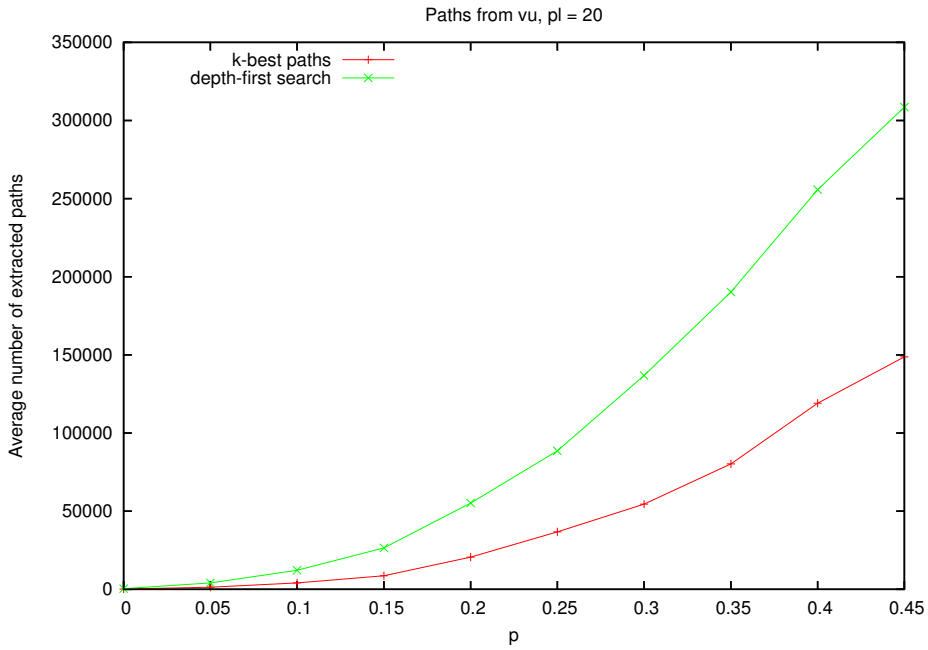


Figure 25: Average number of extracted paths for paths starting from cells in vu; $\mathrm{pl} = 20$

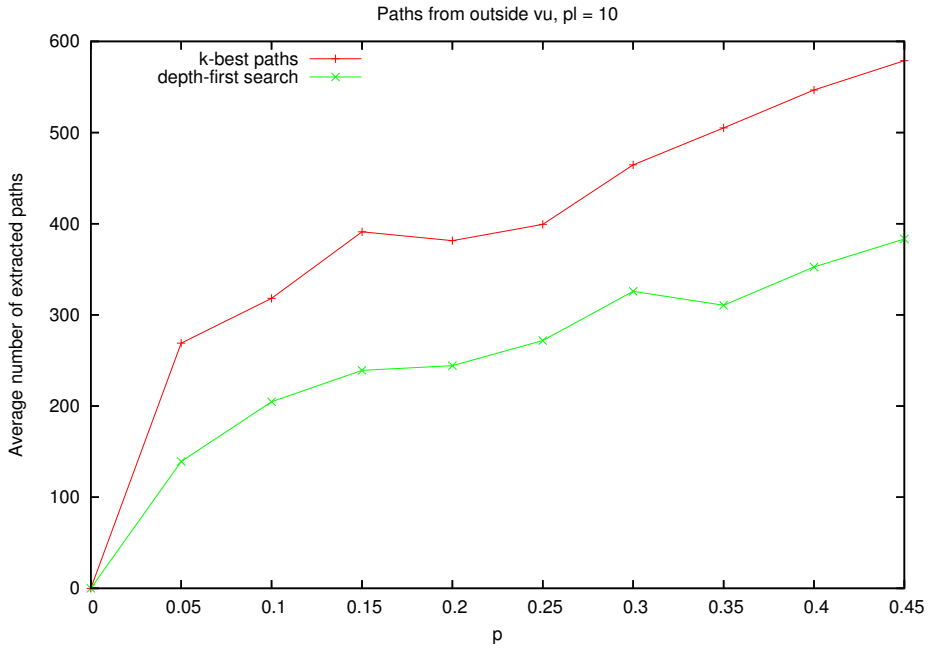


Figure 26: Average number of extracted paths for paths starting from cells outside of vu; $\mathrm{pl} = 10$

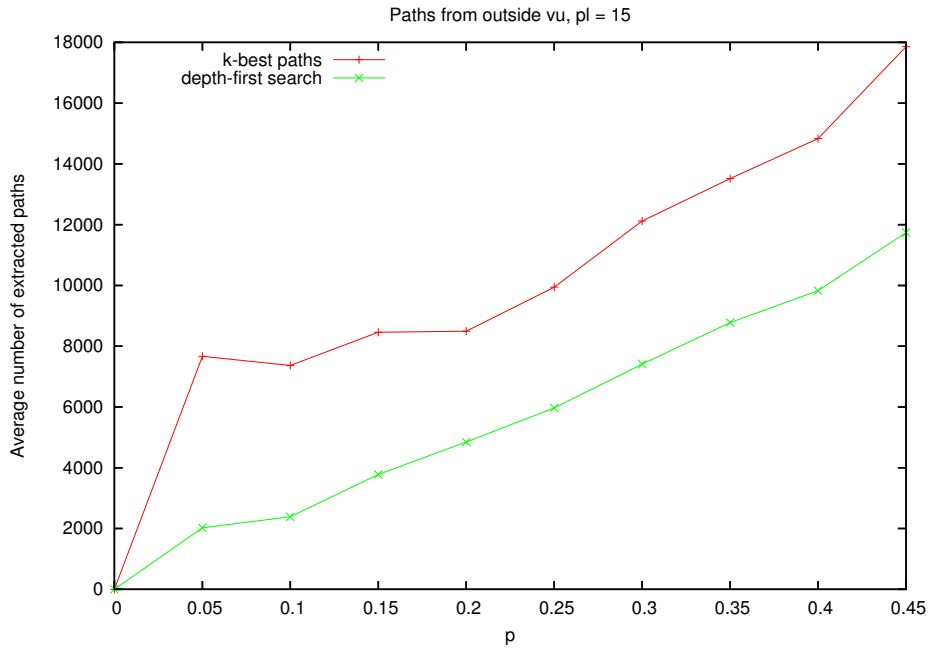


Figure 27: Average number of extracted paths for paths starting from cells outside of vu; $\mathtt{pl} = 15$

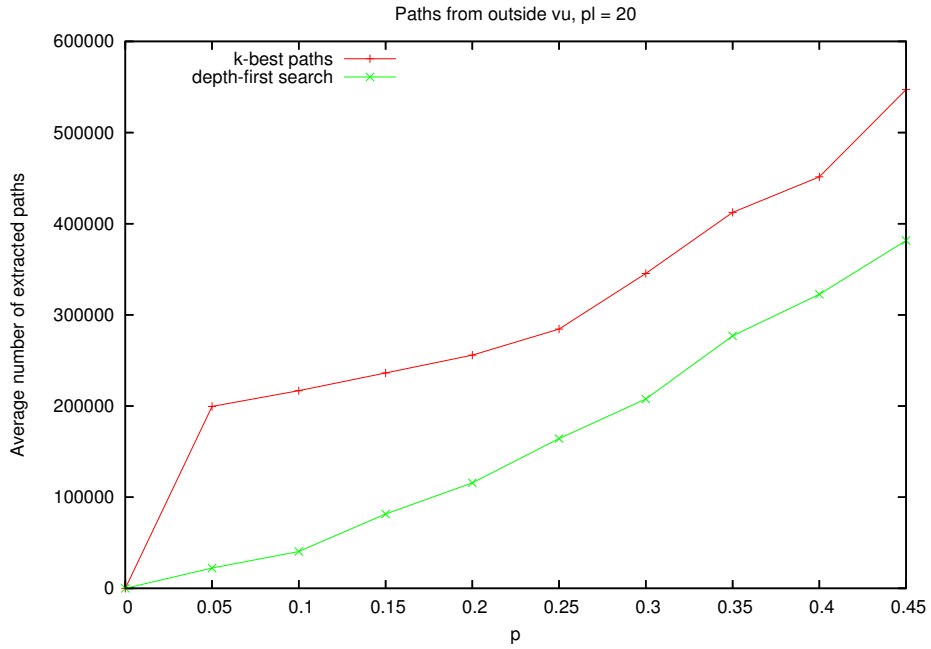


Figure 28: Average number of extracted paths for paths starting from cells outside of vu; $\mathtt{pl} = 20$

p	Solutions from v_u	Solutions not from v_u
0.00	100%	0%
0.05	94%	6%
0.10	87%	13%
0.15	80%	20%
0.20	70%	30%
0.25	62%	38%
0.30	53%	47%
0.35	46%	54%
0.40	39%	61%
0.45	32%	68%
All p	66%	34%

Table 3: The percentage of solution paths that start from v_u, and the corresponding percentage of solution paths that do not start from v_u for each tested value of p.

prisingly high number of experiments. However, the "peak experiments", where the k-best paths goes through a considerably higher number of paths than the depth-first search introduces such a big difference between the two algorithms that the average number of extracted paths always becomes lower for the depth-first search.

The exception is of course when $p = 0$, and low values of p when $pl = 10$, where the k-best paths algorithm performed better than the depth-first search. The general trend seems to be that the two algorithms get very similar results.

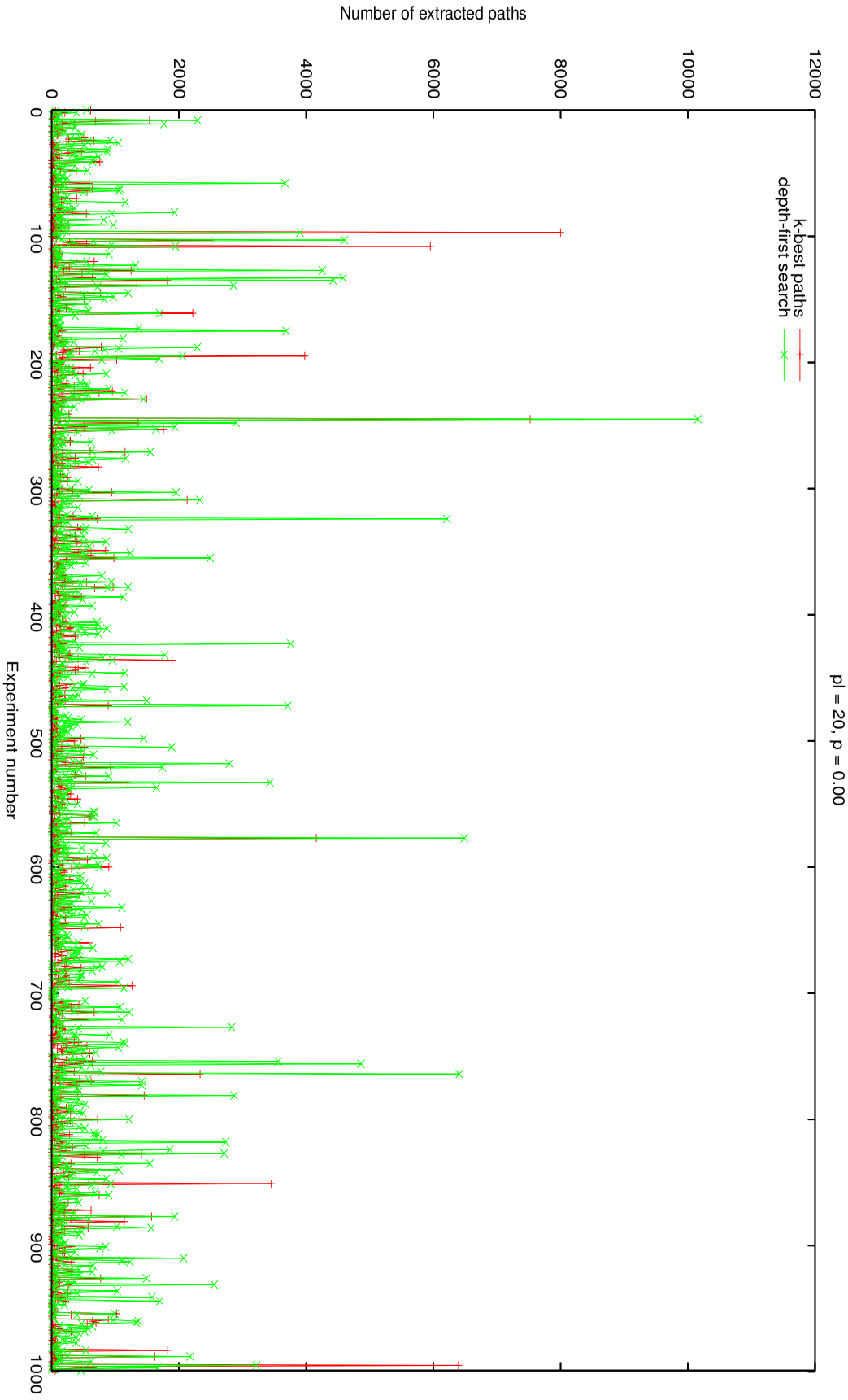


Figure 29: The actual number of extracted paths for each performed experiment for $pl = 20$ and $p = 0.00$.

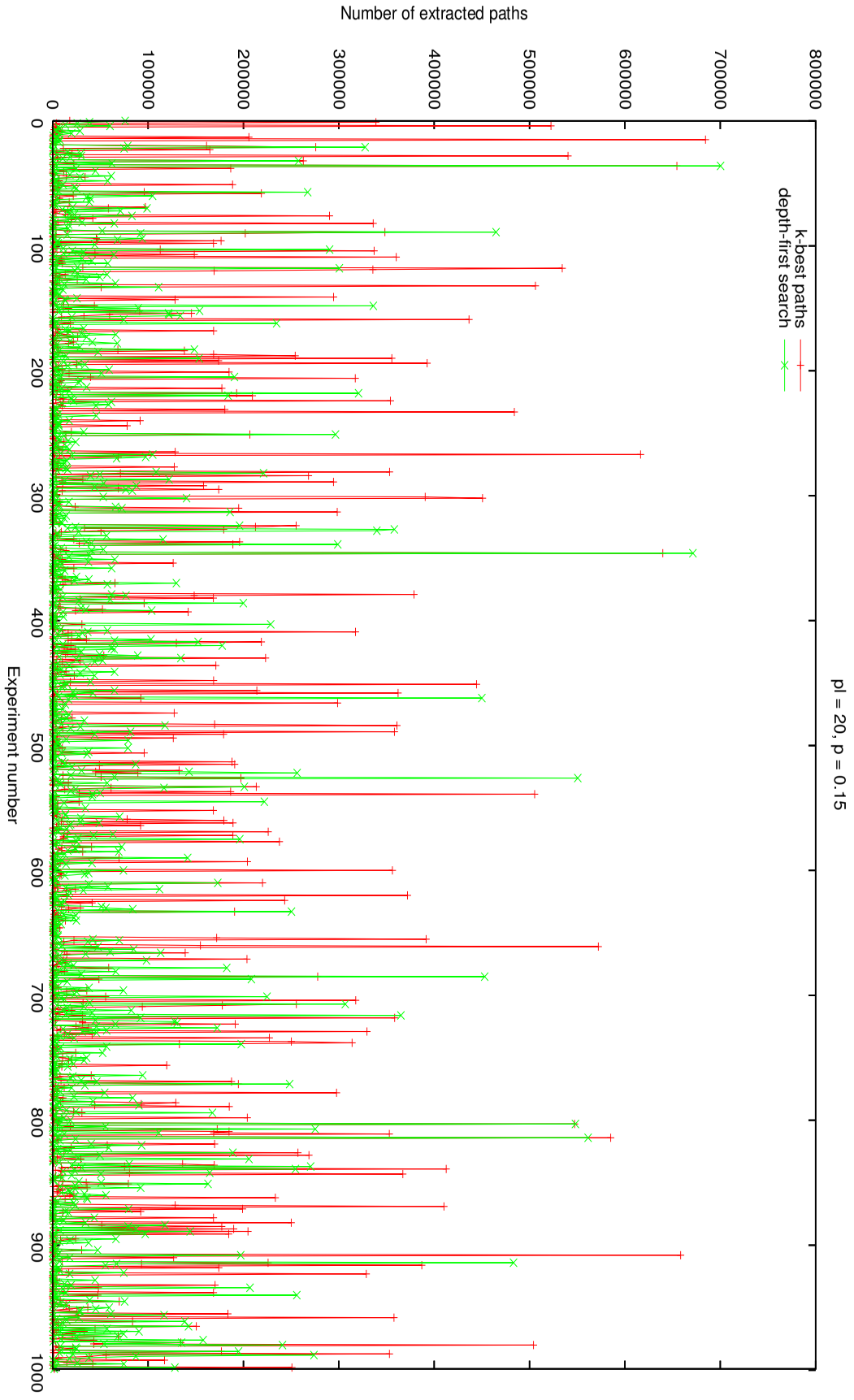


Figure 30: The actual number of extracted paths for each performed experiment for $\mathtt{pl} = 20$ and $\mathtt{p} = 0.15$.

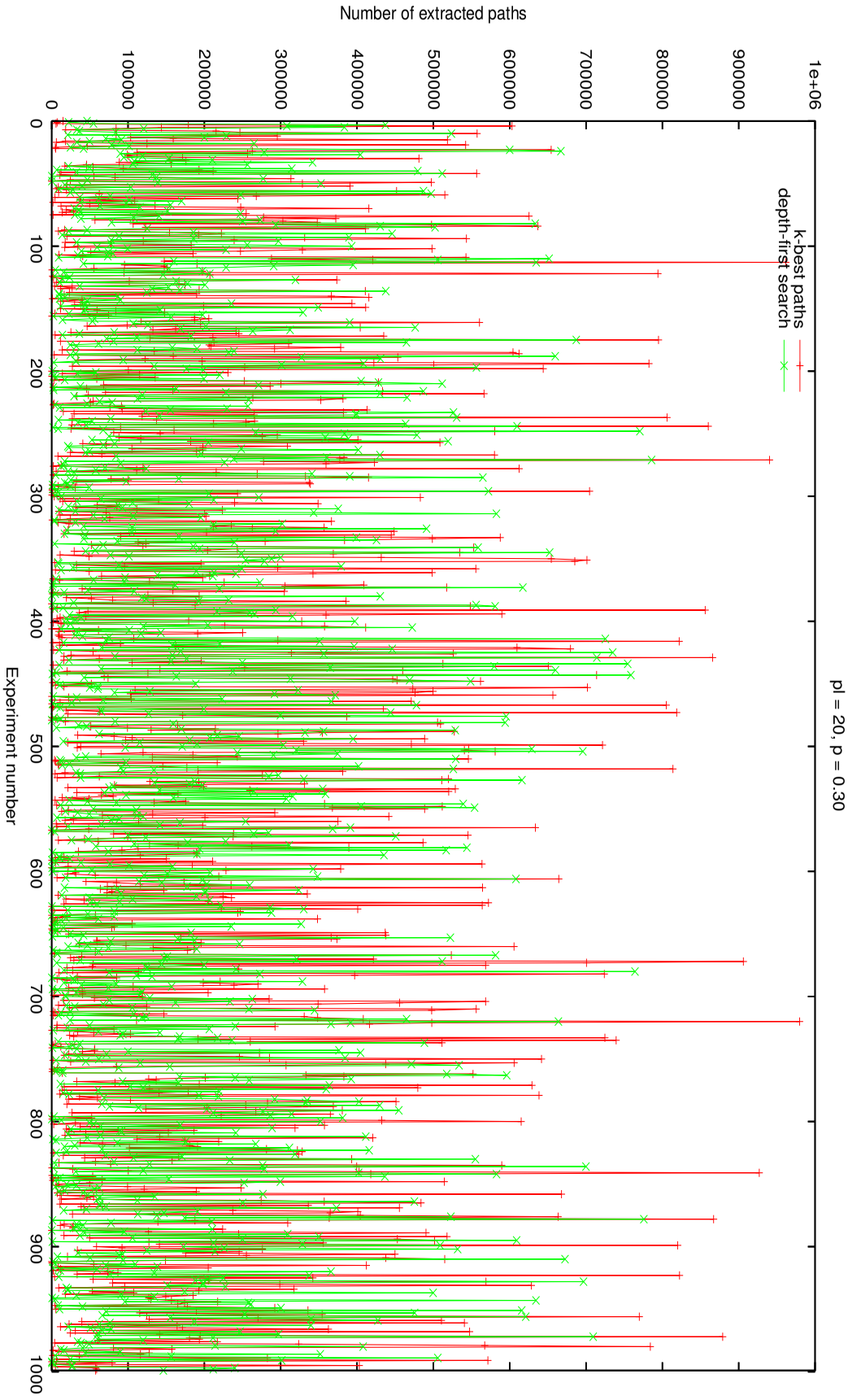


Figure 31: The actual number of extracted paths for each performed experiment for $pl = 20$ and $p = 0.30$.

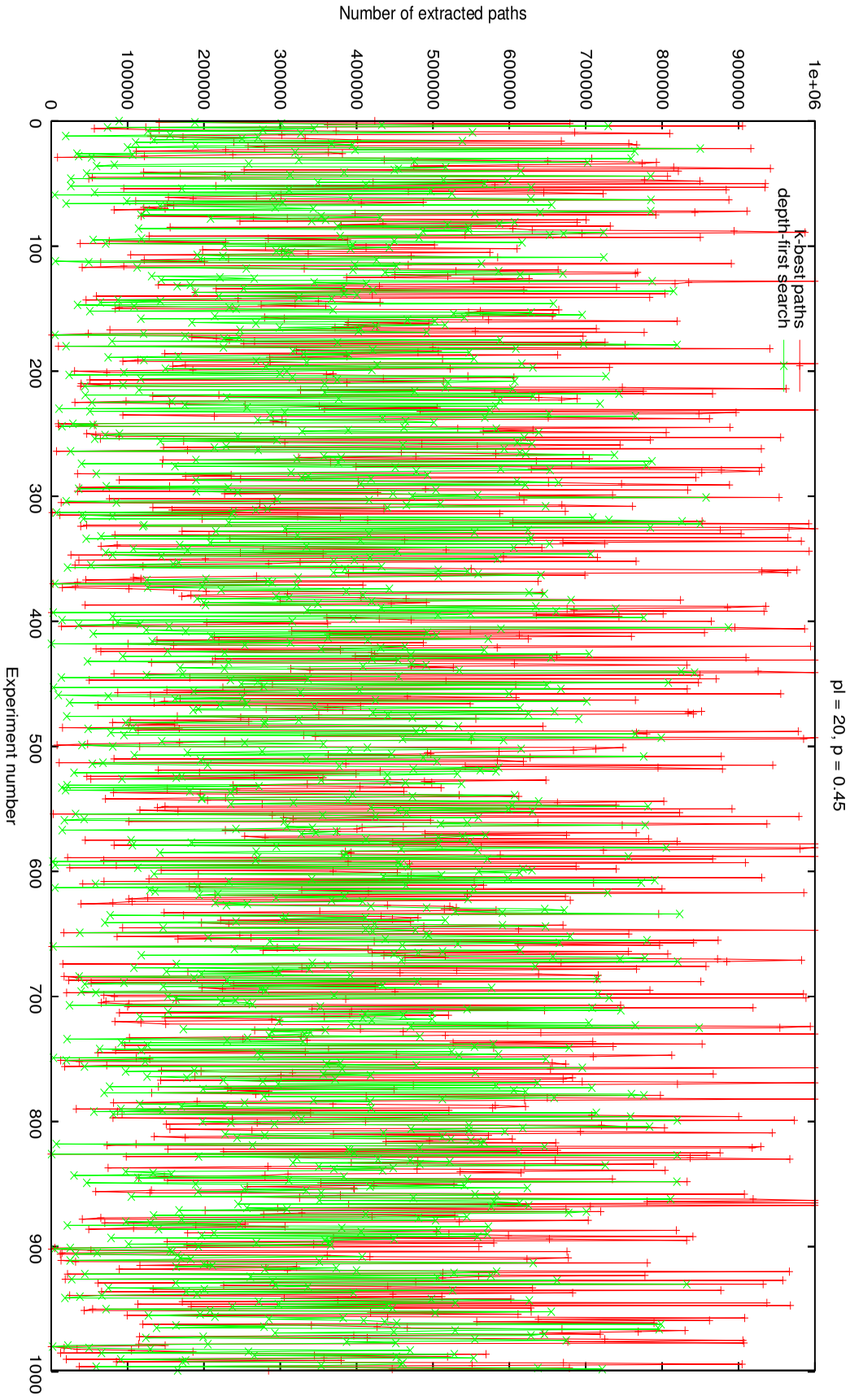


Figure 32: The actual number of extracted paths for each performed experiment for $\mathtt{pl} = 20$ and $\mathtt{p} = 0.45$.

5 Discussion

5.1 The general purpose of the algorithm

The first question that was posed in Section 4.1 was answered in a very convincing way by all the experiments; the proposed attack does work, it found the correct clock control sequence in all the performed experiments. Hence it can be concluded that the proposed attack can be used to cryptanalyze stream ciphers using clock controlled keystream generators.

5.2 The dependency of k on p and pl

The correlation parameter p is a measure for the probability of a '1' in the plaintext sequence, that is the probability of a substitution in the transformation from the LFSR's output sequence **x** to the intercepted ciphertext sequence **z**. If $p = 0$, that means that there were no substitutions, and thus the only edit operations that were used to transform **x** to **z** were deletions. This again means that the clock control sequence describes the only transformation needed to convert **x** to **z**. Thus, if $p = 0$, the correct clock control sequence will always be represented by an optimal path in the edit distance matrix.

However, when $p > 0$, some substitutions will be introduced after the deletions have taken place according to the clock control sequence. Thus the clock control sequence will describe the edit operations necessary to transform **x** to the keystream sequence **y**, but the edit distance matrix will represent edit distances between **x** and **z** $\neq$ **y**. It must therefore be expected that the clock control sequence might be represented by a suboptimal path if $p > 0$.

Because of this it cannot be expected that the number of extracted paths, k, does not increase when p increases. The experimental results showed that the average number of extracted paths, both for the k-best paths algorithm and the depth-first search, depends linearly on p. Nevertheless, the depth-first search performed better than the k-best paths algorithm on average; that is its linear factor is less than that of the k-best paths.

The number of possible clock control sequences increases exponentially with pl. Therefore it is not unexpected that k also seems to increase exponentially with pl. Again, the depth-first search and the k-best paths search behaved similarly, but the depth-first search gave somewhat lower averages than the k-best paths algorithm, except for when $p = 0$.

5.3 Comparison of k-best paths and depth-first search

The results indicate that the depth-first search generally performs better than the k-best paths search in the clock control sequence reconstruction, except for when there is no noise. In the latter case, the solution path is always optimal, and thus it starts from a cell in v_u; it is therefore expected that the k-best paths algorithm finds the solution faster than the depth-first search in this case. The graphs show that the average number of extracted paths for the two algorithms follows approximately the same curve, but as p increases, this average increases faster for k-best paths than for the depth-first search. The depth-first search performs better than the k-best paths in most cases where $p > 0$,

and this tendency seems to become clearer as pl increases.

The results for individual experiments show that the main difference between the two algorithms is that the k-best paths algorithm peaks more often in the number of extracted paths than the depth-first search does. That is, the k-best paths search ends up extracting an extremely high number of paths, compared to most of the experiments, more often than the depth-first search; in other words k-best paths has a higher number of "bad" experiments, where the performance is way worse than normal. Also, these peaks are often higher for the k-best paths than for the depth-first search.

The differences between the procedures in the two algorithms suggest that the k-best paths peaks come when the clock control sequence is represented by a path that has a relatively high cost, that is there are many paths between pl and $[0,0]$ that have lower cost than the solution path. This is because the depth-first search extracts all paths with cost less than or equal to the optimal cost plus a given discrepancy, in the order that the depth-first search finds them, which is not necessarily in the order of smaller cost before larger cost. The k-best paths search on the other hand, always finds all paths with cost lower than the cost of the solution path before it finds the solution path.

The weights that are introduced on the edges from the source vertex Src_2 are meant to make up for the fact that sometimes a path has a very low cost between pl and $[0,0]$, but the shortest path from $[N-M,M]$ to the cell in pl where the path started might have a high cost. Thus the cost of the full length of such paths can be relatively high, even though the parts of the paths that are included in the search have low costs. Without these extra weights there would probably be more short paths between pl and $[0,0]$ that the k-best paths search would have to go through before it found the correct solution.

However, the weights that are added to the edges are computed heuristically, and thus they only form a lower bound of the length of the shortest path between $[N-M,M]$ and the cells represented by the destination vertices. The number of extracted paths might be reduced further if the actual length of the shortest path between $[N-M,M]$ and the current cell were used.

If the actual path lengths were to be used, one would have to invest some extra work to determine these lengths. One way to do this is to compute a single source shortest path tree for the right part of the matrix, that is for the columns $pl,\ldots,N-M$, with the bottom right position, $[N-M,M]$, as the source. Another method is to keep track of the length of the shortest path to each cell in pl, for each matrix position in columns to the right of pl, during the edit distance computation. This information could be kept in another vector associated with each matrix position, just like the existing vectors vp, vu, ve, vj.

The fact that the k-best paths algorithm performs better than the depth-first search whenever the solution path starts from a cell in vu, and the relatively high percentage of such solution paths, also indicate that the problem with the k-best paths algorithm versus the depth-first search is the "peak experiments". The k-best paths algorithm does actually perform better than the depth-first search, on average, in 66% of the experiments. Nevertheless the overall performance is still better for the depth-first search, except for when $p=0$.

The main problem with the performance of the k-best paths algorithm is these peak experiments, thus this algorithm would probably perform better if several attacks were run in parallel, on different ciphertext segments. In such a setup it would not matter if a few of the parallel experiments were peak experiments, as long as the solution could be

found relatively quickly by some of the other experiments.

6 Conclusion

This thesis has introduced a new method for cryptanalyzing stream ciphers based on clock controlled keystream generators. This method uses edit distance to find the candidate initial states of the clocked subgenerator, and then a directed k-best paths search through the edit distance matrix is used to find the correct clock control sequence. This attack is based on the similar attack from [3], where a depth-first search is used instead of the k-best paths algorithm.

Experiments have demonstrated that the attack is able to reconstruct the initial state of the correct clock control sequence, given a sufficiently long intercepted ciphertext sequence and specifics about the keystream generator. The experiments also showed that the average number of paths that the attack extracts from the edit distance matrix before the correct clock control sequence is found, depends linearly on the correlation parameter p, and exponentially on pl, the number of elements of the clock control sequence that are needed to reconstruct the initial state of its generator.

The performance of the proposed attack is very similar to that of the attack in [3]. When there is no noise, the k-best paths algorithm goes through fewer paths on average than the depth-first search from [3] to find the correct clock control sequence; that is, the k-best paths approach seems to be faster than depth-first search in the known-plaintext scenario. However, when $p > 0$, the depth-first search extracts slightly fewer paths on average than the k-best paths, before the solution is found. This difference in average performance is mostly due to some isolated experiments where the k-best paths search extracts a very high number of paths compared to the average case. The k-best paths attack has more of these "peak experiments" than the depth-first attack has, and the k-best paths peaks are also often higher.

7 Future work

This thesis has proposed a new attack on stream ciphers using clock controlled keystream generators, where a k-best paths algorithm is used to reconstruct the clock control sequence from an edit distance matrix. Experimental work has shown that, in certain isolated experiments, the k-best paths algorithm extracts a relatively high number of paths before the correct solution is found, compared to the average case. If the number of extracted paths in these experiments could be reduced, then the proposed attack would have a much better average performane.

In particular, one attempt to reduce the number of extracted paths would be to use a better algorithm to compute the weights for the edges going out of the vertex Src_2 in the graph created from the edit distance matrix. In this thesis these weights are computed as the difference between the optimal partial edit distances of the lower right corner of the matrix and the cells that the destination vertices represent. Thus the weight on each edge is a lower bound for the length of the shortest path from $[N-M, M]$ to cell that the head of the edge represents. One suggestion is to use the actual length of this shortest path instead.

It would also be interesting to perform experiments where several attacks are run in parallel, on different ciphertext segments. Since the k-best paths algorithm has a greater problem with peak experiments than the depth-first search, it can be expected that parallel attacks can lead to a greater improvement in the performance of the k-best paths attack compared to the attack that uses depth-first search. Thus it would be particularly interesting to see if the depth-first search still performs better than the k-best paths attack on average when parallel attacks are used, or if the k-best paths algorithm can outperform the depth-first search in such a setup.

Bibliography

[1] Menezes, A. J., Vanstone, S. A., & Oorschot, P. C. V. 1996. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA.

[2] Golić, J. D. & Mihaljević, M. J. 1991. A generalized correlation attack on a class of stream ciphers based on the levenshtein distance. *Journal of Cryptology*, 3(3), 201–212.

[3] Petrović, S. & Fúster-Sabater, A. 2004. Clock control sequence reconstruction in the ciphertext only attack scenario. In *ICICS*, 427–439.

[4] Molland, H. 2004. Improved linear consistency attack on irregular clocked keystream generators. In *Lecture Notes in Computer Science*, Bimal Roy, W. M., ed, volume 3017, 109–126. Springer-Verlag.

[5] Johansson, T. 1998. Reduced complexity correlation attacks on two clock-controlled generators. In *Lecture Notes in Computer Science*, K. Ohta, D. P., ed, volume 1514, 342–356. Springer-Verlag.

[6] Eppstein, D. 1997. Finding the *k* shortest paths.

[7] Chambers, W. G. & Golić, J. D. September 2002. Fast reconstruction of clock-control sequence. *Electronics Letters*, 38(20), 1174–1175.

[8] Bahl, L., Cocke, J., Jelinek, F., & Raviv, J. March 1974. Optimal decoding of linear codes for minimizing symbol error rate. *IEEE Transactions on Information Theory*, 20(2), 284–287.

[9] Zeng, K., Yang, C. H., & Rao, T. R. N. 1989. On the linear consistency test (lct) in cryptanalysis with applications. In *CRYPTO '89: Proceedings on Advances in cryptology*, 164–174, New York, NY, USA. Springer-Verlag New York, Inc.

[10] Golić, J. & Menicocci, R. 1997. Edit distance correlation attack on the alternating step generator. In *Advances in Cryptology - CRYPTO '97*, Lecture Notes in Computer Science, 499–512. Springer-Verlag.

[11] Byers, T. & Waterman, M. 1984. Determining all optimal and nearoptimal solutions when solving shortest path problems by dynamic programming. *Operations Research*, 32, 1381–1384.

[12] Waterman, M. l983. Sequence alignments in the neighborhood of the optimum with general application to dynamic programming. In *Proc. Natl. Acad. Sci. USA*, volume 80, 3123–3124.

[13] Naor, D. & Brutlag, D. L. 1993. On suboptimal alignments of biological sequences. In *CPM '93: Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, 179–196, London, UK. Springer-Verlag.

[14] Shibuya, T. & Imai, H. January 1997. Enumerating suboptimal alignments of multiple biological sequences efficiently. In *Proc. 2nd Pacific Symp. Biocomputing*, 409–420.

[15] Azevedo, J. A., Santos Costa, M. E. O., Silvestre Madeira, J. J. E. R., & de Queirós Vieira Martins, E. 1993. An algorithm for the ranking of shortest paths. *Eur. J. Operational Research*, 69, 97–106.

[16] Dreyfus, S. E. 1968. An appraisal of some shortest path algorithms. In *ORSA/TIMS Joint National Mtg.*, volume 16, 166.

[17] Fox, B. L. 1973. Calculating kth shortest paths. *INFOR–Canad. J. Op. Res. & Inf. Proc.*, 11, 66–70.

[18] Fox, B. L. May 1975. More on kth shortest paths. *Commun. Assoc. Comput. Mach.*, 18(5), 279.

[19] Fox, B. L. 1975. k-th shortest paths and applications to the probabilistic networks. In *ORSA/TIMS Joint National Mtg.*, volume 23, B263.

[20] de Queirós Vieira Martins, E. 1984. An algorithm for ranking paths that may contain cycles. *Eur. J. Operational Research*, 18(1), 123–130.

[21] Miaou, S.-P. & Chin, S.-M. 1991. Computing k-shortest path for nuclear spent fuel highway transportation. *Eur. J. Operational Research*, 53, 64–80.

[22] Shier, D. R. 1979. On algorithms for finding the k shortest paths in a network. *Networks*, 9(3), 195–214.

[23] Shier, D. R. 1976. Iterative methods for determining the k shortest paths in a network. *Networks*, 6(3), 205–229.

[24] Skicism, C. C. & Golden, B. L. 1987. Computing k-shortest path lengths in Euclidean networks. *Networks*, 17(3), 341–352.

[25] Skicism, C. C. & Golden, B. L. 1989. Solving k-shortest and constrained shortest path problems efficiently. In *Network Optimization and Applications*, Shetty, B., ed, number 20 in Annals of Operations Research, 249–282. Baltzer Science Publishers.

[26] Jiménez, V. M. & Marzal, A. 1999. Computing the k shortest paths: A new algorithm and an experimental comparison. In *WAE'99*, Vitter, J. & Zaroliagis, C., eds, Lecture Notes in Computer Science, 15–29. Springer-Verlag.

[27] Jiménez, V. M. & Marzal, A. 2003. A lazy version of eppstein's k shortest paths algorithm. In *WEA 2003*, et al., K. J., ed, Lecture Notes in Computer Science, 179–191. Springer-Verlag.

[28] Levenshtein, A. 1966. Binary codes capable of correcting deletions, insertions and reversals. In *Soviet Physics-Doklandy*, volume 10, 707–710.

[29] Oommen, B. J. 1986. Constrained string editing. *Information Sciences*, 40(3), 267–284.

[30] Dijkstra, E. W. December 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), 269 – 271.

[31] Manber, U. 1989. *Introduction to Algorithms: a Creative Approach*. Addison-Wesley.

[32] Frederickson, G. N. 1993. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2), 197–214.

[33] Frederickson, G. N. 1991. Ambivalent data structures for dynamic 2-edge-connectivity and k smallest spanning trees. In *IEEE Symposium on Foundations of Computer Science*, 632–641.

A Source code for the k-best paths implementation

The k-best paths code that was used in the experiments is listed here. The code was written in C++, and the program was named `pathfinder`. In section A.1, the header file is listed; this file also contains comments that explain the purpose of each function. Section A.2 lists the C++ implementation of the functions from the header file.

A.1 Header file

```
#ifndef PATHFINDER_H
#define PATHFINDER_H

#include "graph.h"                    // Graph, Vertex, Edge, Path, etc.
#include "heap/binary_heap.h"         // BinaryHeap
#include "heap/binary_marked_heap.h" // BinaryMarkedHeap

#include <list>                       // std::list
#include <set>                        // std::set
#include <vector>                     // std::vector
#include <map>                        // std::map


/// Forward declarations.
class BinaryHeap2Graph;
class BinaryMarkedHeap2Graph;


/// Encapsulation of the k best paths algorithm.
class PathFinder {

public:
    /** Constructor
     *
     * @param graph The graph in which to find paths.
     * @param source The paths must start at this vertex in g.
     * @param target The paths must end at this vertex in g.
     * @return A PathFinder object which can be used to retrieve the
     *         paths from 'source' to 'target' in sorted order (least
     *         weight first)
     */
    PathFinder (const Graph& graph, const Vertex& source,
                const Vertex& target);

    /// Destructor
    ~PathFinder ();

    /// Return true iff this object has been initialized correctly
    /// and is ready for usage
    bool valid () const { return isValid; }

    /// Reset the path retrieval pointer. The next path retrieved will
    /// be the first/best path.
    void reset () { resetPos = true; }

    /// Retrieve the next path found.
    Path next ();

    /** Retrieve the next k paths.
     * @param k The number of paths to retrieve starting at the current
     *          path retrieval pointer. If k paths cannot be found, the
     *          actual number of paths found will be stored into k.
     * @return A list with the k next paths in sorted order from src
     *         to target in g. If it is not possible to find k paths
     *         from src to target in g, the number of paths returned
     *         will be less that k, and the parameter k will be
     *         rewritten to the actual number of paths found.
     */
```

```
    std::list<Path> getPaths (unsigned int& k);

    /** Change the source vertex, i.e. the vertex from which paths are
     *  found.
     *
     * @param source The new source vertex in g from which paths are
     *               found.
     */
    void changeSource (const Vertex& source);

private:
    /// Initialize this PathFinder object, i.e. run most of Eppstein's
    /// k best paths algorithm
    void init ();

    /// Return the next best path in P(g)
    Path getNextPath ();

    /// Convert the given path in P(g) to its corresponding path in g.
    Path fromP2G (const Path& inP);

    /** Fill the outRoots map. This method calls itself recursively.
     *
     * @param outRoots Mapping from any vertex v in g to the binary
     *                 marked heap of root sidetracks for vertices
     *                 between v and target t. Keyed by v.index().
     *                 This mapping is recursively built by this
     *                 method.
     * @param rootMap Mapping from vertices in g to root sidetracks
     *                in H_out(v).
     * @param current Vertex in g to be processed by this call to the
     *                method.
     * @param prev Vertex in g which was processed previous to this
     *             invocation. This parameter is only useful on recursive
     *             invocations of this method. On the first invocation,
     *             this parameter should not be specified, as the default
     *             value (an empty/invalid Vertex) suffices.
     */
    void fillOutRoots (
        std::vector<BinaryMarkedHeap2Graph>& outRoots,
        const std::vector<Edge>& rootMap,
        const Vertex& current,
        const Vertex& prev = Vertex()
    );

    /** Build D(g). This method calls itself recursively.
     *
     * @param h Mapping from any vertex v in g to its root sidetrack
     *          node in D(g). This mapping is built and used by this
     *          method.
     * @param outRoots Mapping from any vertex v in g to the binary
     *                 marked heap of root sidetracks for vertices
     *                 between v and target t. Keyed by v.index().
     * @param rootMap Mapping from any vertex v in g to its root
     *                sidetrack (in sidetracks). Keyed by v.index().
     * @param heapMap Mapping from any vertex v in g to the binary
     *                heap of sidetracks (in sidetracks graph)
     *                connected to v's root sidetrack. Keyed by
```

```
     *               v.index().
     * @param current Current vertex in g which is to be processed.
     * @param stMap Mapping from any sidetrack edge e in sidetracks to
     *              its currently (changes while traversing SPT(g))
     *              corresponding vertex in D(g). This parameter should
     *              be specified as an empty mapping by the initial
     *              caller. Recursive calls use this parameter to pass
     *              necessary state between invocations.
     */
    void buildD (
        const std::vector<BinaryMarkedHeap2Graph>& outRoots,
        const std::vector<Edge>& rootMap,
        const std::vector<BinaryHeap2Graph>& heapMap,
        const Vertex& current,
        std::map<Edge, Vertex> stMap
    );

    /// Convert D(g) to P(g)
    void buildP ();


    /// Set to true when this object has been correctly initialized
    /// and is ready for usage.
    bool isValid;

    /// The graph in which this object generates paths.
    const Graph& g;

    /// The source vertex from which paths start.
    Vertex s;

    /// The target vertex where paths end.
    Vertex t;

    /// Sorted list of currently computed paths in P(g).
    std::list<Path> paths;

    /** Path retrieval iterator. Points to the last Path in p returned
     *  from this object.
     *
     * If no appropriate previous path exists (i.e. before the first
     * Path is retrieved, or after reset() has been called), the
     * accompanying boolean flag will be false.
     */
    std::list<Path>::const_iterator pos;
    bool resetPos;

    /// SPT(g), sidetracks(g) and P(g) graphs, all derived from g.
    Graph spt, sidetracks, p;

    /// Root vertex in P(g). This vertex has exactly one outgoing edge
    /// to the vertex representing the source vertex s.
    Vertex p_root;

    /// Binary heap of paths in P(g) currently being processed. Ordered
    /// by least path length.
    BinaryHeap<weight_t, Path> inProcess;
```

```
    /// Set of cross edges in P(g).
    std::set<Edge> crossEdges;

    /// Mapping from vertices in P(g) to corresponding edges in
    /// sidetracks.
    std::vector<Edge> p2st;

    /// Mapping from any vertex v in g to its root sidetrack node in
    /// P(g).
    std::vector<Vertex> g2p;

};

/// Simple extension of BinaryHeap for converting a heap into a
/// corresponding graph.
class BinaryHeap2Graph : public BinaryHeap<weight_t, Edge> {
public:
    /// Constructor. Create a new and empty binary heap.
    BinaryHeap2Graph () : BinaryHeap<weight_t, Edge>() {}

    /// Copy-Constructor. Create a new binary marked heap from the
    /// given.
    BinaryHeap2Graph (const BinaryHeap2Graph& other)
    : BinaryHeap<weight_t, Edge>(other) {}

    /// Destructor.
    virtual ~BinaryHeap2Graph () {}

    /** Extend the given graph with a graph representation of this
     *  heap.
     *
     * The internal structure of this heap is added to the given graph,
     * where each node in the heap corresponds to a vertex added in
     * the graph, and an edge from each parent to every child
     * node/vertex is added. The vertices and edges added will
     * constitute a subgraph disconnected from the existing contents of
     * the given graph.
     *
     * @param g Graph to which this heap's graph representation will be
     *          added.
     * @param d2st Mapping from vertex index  in graph g to Edge in
     *             sidetracks graph. For each vertex added to graph g
     *             by this method, a corresponding entry will be added
     *             to d2st.
     * @returns The index of the vertex corresponding to the heap's
     *          root node.
     */
    virtual unsigned int addToGraph (Graph& g,
                                     std::vector<Edge>& d2st)const;
};


/// Simple extension of BinaryMarkedHeap for converting a heap into a
/// corresponding graph.
class BinaryMarkedHeap2Graph : public BinaryMarkedHeap<weight_t, Edge> {
public:
    /// Contructor. Create a new and empty binary heap.
    BinaryMarkedHeap2Graph () : BinaryMarkedHeap<weight_t, Edge>() {}
```

```
    /// Copy-Contructor. Create a new binary marked heap from the given.
    BinaryMarkedHeap2Graph (const BinaryMarkedHeap2Graph& other)
    : BinaryMarkedHeap<weight_t, Edge>(other) {}

    /// Destructor.
    virtual ~BinaryMarkedHeap2Graph () {}

    /** Extend the given graph with a graph representation of this heap.
     *
     * The internal structure of this heap is added to the given graph,
     * where each marked node in the heap corresponds to a vertex added
     * in the graph, and an edge from each parent to every child
     * node/vertex is added. For unmarked nodes no new vertices are
     * added to the graph. Instead the unmarked node is represented by
     * a vertex already present in g found by mapping the node (edge
     * object) through the given stMap parameter. All edges connected to
     * unmarked nodes will be connected to the existing vertices found
     * by mapping through stMap.
     *
     * All vertices added are also registered in stMap.
     *
     * @param g Graph to which this heap's graph representation will
     *          be added.
     * @param d2st Mapping from vertex index  in graph g to Edge in
     *             sidetracks graph. For each vertex added to graph
     *             g by this method, a corresponding entry will be
     *
     * @param stMap Mapping from Edge objects in sidetracks graph their
     *              corresponding Vertex objects in graph g. This map is
     *              used to resolve unmarked unmarked nodes, and is
     *              updated whenever adding another vertex to graph g.
     * @returns The vertex index in g of the root vertex the first
     *          vertex added (the first marked node encountered) is
     *          located.
     */
    virtual unsigned int addToGraph (
        Graph& g,
        std::vector<Edge>& d2st,
        std::map<Edge, Vertex>& stMap
    ) const;
};

#endif // PATHFINDER_H
```

A.2 C++ source code

```
#include "pathfinder.h"              // PathFinder
#include "utils.h"                   // ASSERT()

#include <iostream>                  // cout, endl
#include <fstream>                   // ofstream
#include <utility>                   // std::pair


using namespace std;
using namespace boost;


/*** PathFinder implementation ***/

PathFinder::PathFinder (const Graph& graph, const Vertex& source,
const Vertex& target) : isValid(false), g(graph), s(source),
t(target), paths(), pos(), resetPos(true),
  spt(), sidetracks(), p(), inProcess() {
    ASSERT(s.valid());
    ASSERT(t.valid());
    ASSERT(g.contains(s));
    ASSERT(g.contains(t));

    init();

    changeSource(source);

    // All necessary initial calculations are complete. 'paths' contains
    // exactly one path (the shortest) (provided that s and t are not
    // disconnected), and inProcess contains the next path (if existing).
    ASSERT(paths.size() <= 1);
    ASSERT(inProcess.numElements() <= 1);
    ASSERT(resetPos);
    isValid = true;
}

PathFinder::~PathFinder () {
    isValid = false;
    resetPos = true;
    inProcess.clear();
    paths.clear();
}

Path PathFinder::next () {
    ASSERT(valid());
    if (resetPos) {
        if (paths.size() == 0) { // There exists no path between s and t.
            return Path(g); // Return null path to signify no more paths.
        }
        pos = paths.begin();
        ASSERT(pos != paths.end());
        resetPos = false;
    }
    else {
        list<Path>::const_iterator nextpos = pos;
        if (++nextpos == paths.end()) { // Must retrieve another path
                                        // from inProcess
            Path next = getNextPath();
```

```
            if (next.empty()) { // There are no more paths to be found.
                resetPos = true;
                return Path(g); // Return null path to signify no more
                                // paths.
            }
            paths.push_back(next);
        }
        ASSERT(pos != paths.end());
        pos++;
        ASSERT(pos != paths.end());
    }

    return fromP2G(*pos);
}

list<Path> PathFinder::getPaths (unsigned int& k) {
    ASSERT(valid());
    list<Path> kpaths;
    for (unsigned int i = 0; i < k; ++i) {
        Path c = next();
        if (c.empty()) break;
        kpaths.push_back(c);
    }
    k = kpaths.size();
    return kpaths;
}

void PathFinder::changeSource (const Vertex& source) {
    ASSERT(g.contains(source));
    ASSERT(p_root.valid());
    ASSERT(p.contains(p_root));

    // Reset path retrieval pointer, empty paths and inProcess
    reset();
    paths.clear();
    inProcess.clear();

    // Remove outgoing edge from p_root if existing.
    p_root.clear();

    s = source;

    // If there are no outgoing edges from s in SPT, there are no
    // possible paths from s to t, and we can therefore return
    // immediately.
    if (s.in(spt).numOutEdges() <= 0) {
#ifdef DEBUG_CODE
        cout << "There is no path from source (" << s.getLabel() <<
        ") to target (" << t.getLabel() << ")! Aborting." << endl;
#endif // DEBUG_CODE
        return;
    }

    // Add root edge if src has a corresponding sidetrack in P(g).
    if (g2p[s.index()].valid()) Edge re(p_root, g2p[s.index()],
        g2p[s.index()].getWeight(), "__root_edge__");

    // Add empty path to list of computed paths. This is the path
```

```
    // representing the shortest possible path between the source
    // and the target.
    Path null(p);
    paths.push_back(null);

    if (p_root.numOutEdges() > 0) { // There are more paths in g than
                                    // the shortest path
        // Create first path in inProcess, consisting of the root
        // edge only.
        ASSERT(p_root.numOutEdges() == 1);
        Path root(null);
        Edge root_edge = *(p_root.outEdges().first);
        ASSERT(root_edge.valid());
        root.add(root_edge);

        // Start processing from this path.
        ASSERT(inProcess.empty());
        inProcess.insert(root.cost(), root);
    }

    // All necessary initial calculations are complete. 'paths' contains
    // exactly one path (the shortest), and inProcess contains the next
    // path (if applicable).
    ASSERT(paths.size() == 1);
    ASSERT(inProcess.numElements() <= 1);
    ASSERT(resetPos);
}

void PathFinder::init () {
#ifdef DEBUG_CODE
    g.writeDot("/tmp/g.dot");
#endif // DEBUG_CODE

    // Create SPT and Sidetracks from g.
    tie(spt, sidetracks) = g.SPT(t);

#ifdef DEBUG_CODE
    spt.writeDot("/tmp/spt_g.dot");
    sidetracks.writeDot("/tmp/sidetracks_g.dot");
#endif // DEBUG_CODE

    // Create H_out(v) for each vertex v in g
    /// map from any vertex v in g to its root sidetrack (in sidetracks
    /// graph). Keyed by v.index().
    vector<Edge> rootMap(g.numVertices(), Edge());
    /// map from any vertex v in g to the binary heap of sidetracks (in
    /// sidetracks graph) connected to v's root sidetrack. Keyed by
    /// v.index().
    vector<BinaryHeap2Graph> heapMap(g.numVertices(), BinaryHeap2Graph());

    // Fill rootMap and heapMap: For each vertex v, insert all outgoing
    // sidetracks into heapMap, then pull the minimum sidetrack into
    // rootMap.
    VertexIt vit, vend;
    for (tie(vit, vend) = g.vertices(); vit != vend; ++vit) {
        Vertex v = *vit;
        ASSERT(v.valid());
        ASSERT(heapMap.size() > v.index()); // Ensure that heapMap is
```

```
                                                // big enough
            heapMap[v.index()] = BinaryHeap2Graph(); // Initialize empty heap
                                                    // in heapMap.

            // Insert outgoing sidetracks into heapMap.
            OutEdgeIt oit, oend;
            for (tie(oit, oend) = v.in(sidetracks).outEdges();
                 oit != oend; ++oit) {
                heapMap[v.index()].insert(oit->getWeight(), *oit);
            }

            // Pull minimum sidetrack from heapMap into rootMap.
            if (heapMap[v.index()].numElements() > 0) {
                rootMap[v.index()] = heapMap[v.index()].extractMinimum().second;
            }
        }

    #ifdef DEBUG_CODE
        cout << "Printing rootMap and heapMap:" << endl;
        for (unsigned int i = 0; i < rootMap.size(); ++i) {
            cout << "\t" << Vertex(i, g).getLabel() << ":\troot = ";
            if (rootMap[i].valid()) cout << rootMap[i].getWeight();
            else                    cout << "N/A";
            cout << ",\theap = " << heapMap[i].toString() << endl;
        }
    #endif // DEBUG_CODE


        // Create H_T(v) for each vertex v in g.
        /// map from any vertex v in g to the binary marked heap of root
        /// sidetracks for vertices between v and target. Keyed by v.index().
        vector<BinaryMarkedHeap2Graph> outRoots(g.numVertices(),
                                                BinaryMarkedHeap2Graph());
        fillOutRoots(outRoots, rootMap, t); // calls itself recursively to for
                                            // all vertices v in g.

    #ifdef DEBUG_CODE
        cout << "Printing outRoots:" << endl;
        for (unsigned int i = 0; i < outRoots.size(); ++i) {
            cout << "\t" << Vertex(i, g).getLabel() << ":\t"
            << outRoots[i].toString() << endl;
        }
    #endif // DEBUG_CODE


        // Create D(g), the conglomeration of H_out(v) and H_T(v) for all
        // vertices v.
        g2p = vector<Vertex>(g.numVertices(), Vertex());
        // if no sidetracks can be reached from vertex v, then h[v] will be
        // an invalid/empty Vertex.
        p = Graph("P(" + g.getLabel() + ")");
        buildD(outRoots, rootMap, heapMap, t, map<Edge, Vertex>());
        ASSERT(p2st.size() == p.numVertices());

    #ifdef DEBUG_CODE
        p.writeDot("/tmp/D_g.dot");
        cout << "D(g) has " << p.numVertices() << " vertices." << endl;
        cout << "Dumping g2p map:" << endl;
```

```
    ASSERT(g.numVertices() == g2p.size());
    for (unsigned int i = 0; i < g2p.size(); ++i) {
        cout << "\th[" << Vertex(i, g) << "]   \t= ";
        if (g2p[i].valid()) cout << g2p[i] << endl;
        else cout << "<*** EMPTY VERTEX*** >" << endl;
    }
#endif // DEBUG_CODE


    // Convert D(g) into P(g), i.e. compute different edge weight, add c
    // ross edges and a root vertex with an edge to vertex s.
    buildP();

#ifdef DEBUG_CODE
    p.writeDot("/tmp/P_g.dot");
    cout << "Cross Edges in P(g):" << endl;
    for (set<Edge>::const_iterator it = crossEdges.begin();
         it != crossEdges.end(); ++it) cout << "\t" << *it << endl;
#endif // DEBUG_CODE

    // Add root vertex to P(g)
    p_root = Vertex(p, 0, "__root__");
}

Path PathFinder::getNextPath () {
    ASSERT(valid());
    if (inProcess.empty()) { // There are no more paths to process.
        return Path(p); // Return null path.
    }

    Path current = inProcess.extractMinimum().second;
#ifdef DEBUG_CODE
//  cout << "Current Path is " << current << endl;
#endif // DEBUG_CODE

    // Add current path's children paths to toBeProcessed
#ifdef DEBUG_CODE
//  cout << "Adding children of " << current << endl;
#endif // DEBUG_CODE
    OutEdgeIt child, cend;
    for (tie(child, cend) = current.to().outEdges(); child != cend;
         ++child) {
        Path child_path(current);
        child_path.add(*child);
        ASSERT(child_path.continuous());
        inProcess.insert(child_path.cost(), child_path);
    }

    return current;
}

Path PathFinder::fromP2G (const Path& inP) {
    ASSERT(valid());
    // Convert p to a list of sidetrack edges (in g) that must be
    // incorporated into final path.
    Path st_edges(g);
    unsigned int pathCost = s.in(spt).getWeight();
    ASSERT(inP.continuous());
```

```
// Add sidetracks taken (which is represented by the vertex from
// which a cross edge starts)
for (Path::const_iterator pit = inP.begin(); pit != inP.end();
     ++pit) {
    if (crossEdges.count(*pit) == 0) continue; // Only add sidetrack
                                               // if at the start of
                                               // a cross edge.
    ASSERT(p2st.size() > pit->from().index());
    Edge st = p2st[pit->from().index()];
    ASSERT(st.valid());
    ASSERT(sidetracks.contains(st));
    pathCost += st.getWeight();
    st = st.in(g, true, false);
    ASSERT(st.valid());
    ASSERT(g.contains(st));
    st_edges.add(st);
}
// Also add the very last vertex in p as this represents the last
// sidetrack taken.
if (inP.size() > 0) {
    ASSERT(p2st.size() > inP.to().index());
    Edge st = p2st[inP.to().index()];
    ASSERT(st.valid());
    ASSERT(sidetracks.contains(st));
    pathCost += st.getWeight();
    st = st.in(g, true, false);
    ASSERT(st.valid());
    ASSERT(g.contains(st));
    st_edges.add(st);
}

// Create path in g following SPT from src to target except for
// following edges in st_edges.
Path result(g);
Vertex current = s; // Current vertex (will trace path from src to
                    // target)
while (current != t) {
    Edge e;
    if (st_edges.size() > 0 && st_edges.front().from() == current) {
        // We must follow a sidetrack from this vertex.
        e = st_edges.front();
        st_edges.pop_front();
    }
    else { // We must follow SPT from this vertex.
        ASSERT(current.in(spt).numOutEdges() == 1);
        e = current.in(spt).outEdges().first->in(g);
    }
    ASSERT(e.valid());
    ASSERT(g.contains(e));
    result.add(e);
    current = e.to();
}

ASSERT(st_edges.empty());
ASSERT(pathCost == result.cost());
ASSERT(result.continuous());
ASSERT(result.from() == s);
ASSERT(result.to() == t);
```

```
    return result;
}

void PathFinder::fillOutRoots (
    vector<BinaryMarkedHeap2Graph>& outRoots,
    const vector<Edge>& rootMap,
    const Vertex& current,
    const Vertex& prev
) {
    // Vertices current and prev are in graph g (prev may be invalid).
    // rootMap is the mapping from vertex indices to root sidetracks in
    // H_out(v) for all vertices v in g.

    // Set up binary heap for current vertex. This is a copy of the
    // previous vertex's heap with no marked nodes.
    if (prev.valid()) {
        outRoots[current.index()] =
            BinaryMarkedHeap2Graph(outRoots[prev.index()]);
    }
    outRoots[current.index()].clearMarked();

    // Add the root sidetrack of the current vertex to the outRoots map.
    Edge currentRootST = rootMap[current.index()];
    if (currentRootST.valid()) {
        outRoots[current.index()].insert(currentRootST.getWeight(),
                                         currentRootST);
    }

    InvAdjVertexIt it, end;
    for (tie(it, end) = current.in(spt).invAdjVertices(); it != end;
         ++it) {
        ASSERT(it->valid());
        Vertex next = it->in(g);
        ASSERT(next.valid());
        fillOutRoots(outRoots, rootMap, next, current);
    }
}

void PathFinder::buildD (
    const vector<BinaryMarkedHeap2Graph>& outRoots,
    const vector<Edge>& rootMap,
    const vector<BinaryHeap2Graph>& heapMap,
    const Vertex& current,
    map<Edge, Vertex> stMap
) {
    // Add graph representation of binary heap in H_out(v) to D(g) for
    // each vertex v in g
    if (heapMap[current.index()].numElements() > 0) {
        unsigned int cur_d_heap_root_index =
            heapMap[current.index()].addToGraph(p, p2st);
        ASSERT(p2st.size() == p.numVertices());
        ASSERT(cur_d_heap_root_index >= 0 &&
               cur_d_heap_root_index < p.numVertices());
        Vertex cur_d_heap_root(cur_d_heap_root_index, p); // Get vertex
                                                          // representing
                                                          // root of heap.
        ASSERT(cur_d_heap_root.valid());
        ASSERT(p.contains(cur_d_heap_root));
```

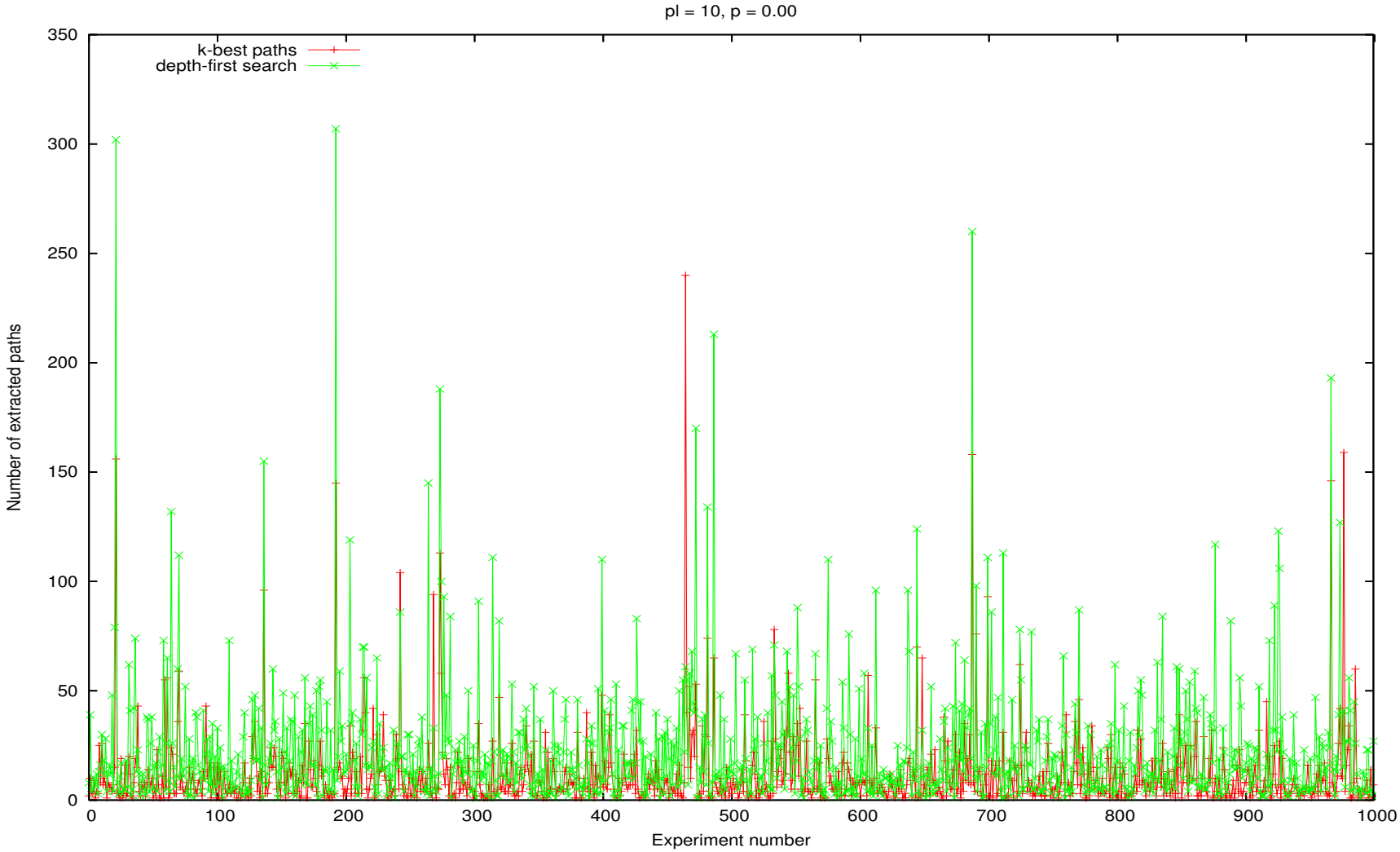
```
            stMap[heapMap[current.index()].findMinimum().second] =
                cur_d_heap_root; // Map from heap root's sidetract edge to its
                                 // corresponding vertex in D(g).
        }

        // Add graph representation of binary heap H_T(v) to D(g) for each
        // vertex v in g.
        unsigned int offset = p.numVertices();
        unsigned int root_index = outRoots[current.index()].addToGraph(p,
            p2st, stMap);
        ASSERT(p2st.size() == p.numVertices());

        // We must also add an edge from the new vertices to the vertices
        // already in D(g) representing the root of the corresponding heap
        // maps.
        for (unsigned int i = offset; i < p.numVertices(); ++i) {
            // Loop through the newly added vertices.
            Vertex d_outRoot = Vertex(i, p);
            ASSERT(d_outRoot.valid());
            d_outRoot.setLabel(current.getLabel() + " - " +
                               d_outRoot.getLabel());
            Edge st = p2st[i];
            ASSERT(st.valid());
            // Find the vertex in g to which this sidetrack belongs.
            Vertex g_from = st.from().in(g);
            ASSERT(st == rootMap[g_from.index()]);
            ASSERT(g_from.index() < heapMap.size());
            Vertex d_heapmap_root;
            if (heapMap[g_from.index()].numElements() > 0) {
                d_heapmap_root =
                    stMap[heapMap[g_from.index()].findMinimum().second];
            }
            if (d_heapmap_root.valid()) { // There exists a heap map for
                                          // g_from.
                ASSERT(!p.hasEdge(d_outRoot, d_heapmap_root));
                Edge root2heap(d_outRoot, d_heapmap_root);
            }
        }

        // Set up mapping g2p(v) from vertex v in g to v's root sidetrack
        // represented by a vertex in d.
        Vertex cur_d_root_st(root_index, p);
        if (cur_d_root_st.valid()) g2p[current.index()] = cur_d_root_st;
            // Add mapping to root sidetrack's vertex in d if it exists.

        // Progress recursively on vertices in shortest path tree
        InvAdjVertexIt it, end;
        for (tie(it, end) = current.in(spt).invAdjVertices(); it != end;
             ++it) {
            ASSERT(it->valid());
            Vertex next = it->in(g);
            ASSERT(next.valid());
            buildD(outRoots, rootMap, heapMap, next, stMap);
        }
    }

    void PathFinder::buildP () {
        // According to Eppstein, vertices in P(g) should be unweighted.
```

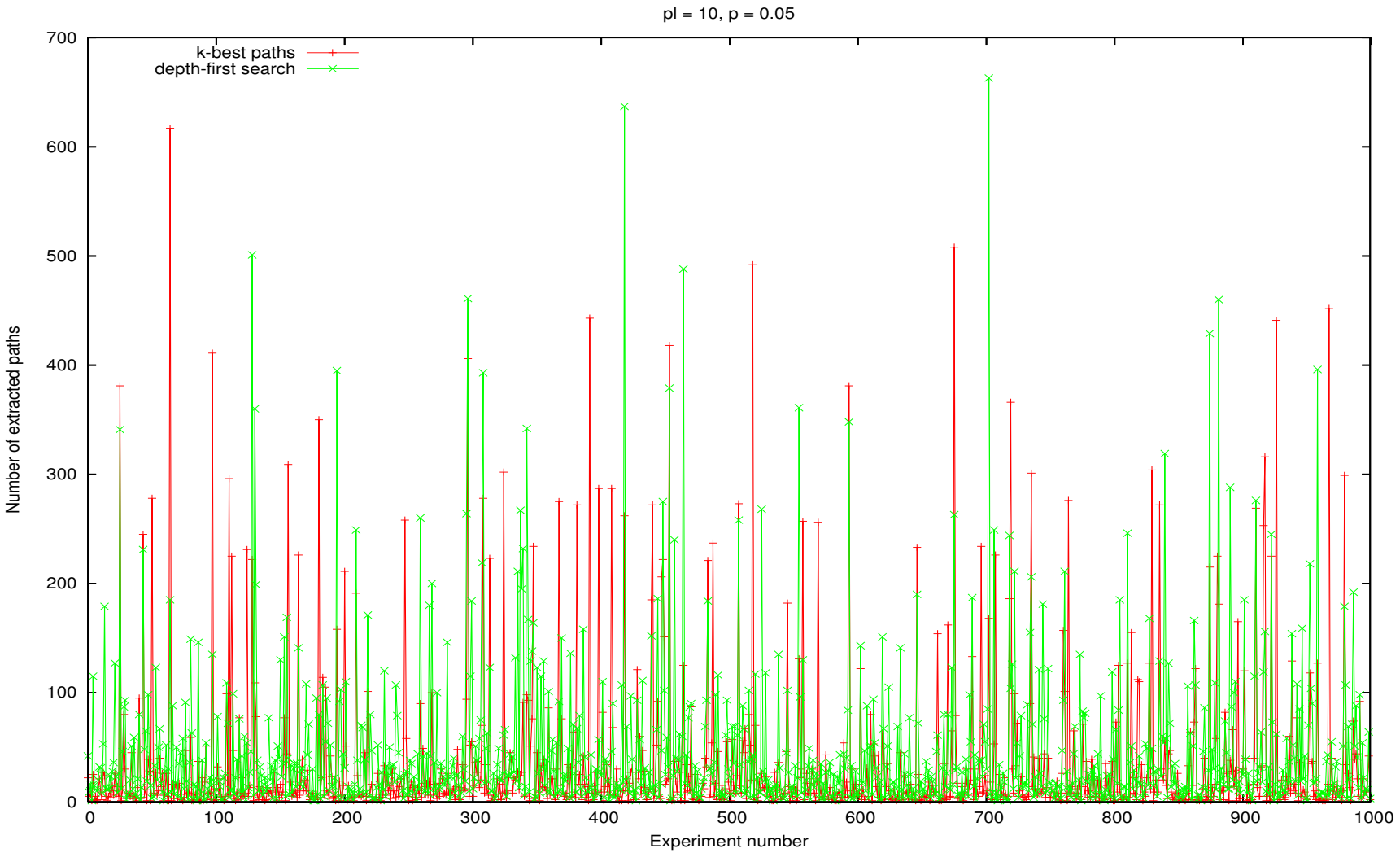
```
    // However, there is no harm in leaving in information that is
    // already there.

    // Set weights on edges in P(g)
    EdgeIt eit, eend;
    for (tie(eit, eend) = p.edges(); eit != eend; ++eit) {
        eit->setWeight(eit->to().getWeight() - eit->from().getWeight());
    }

    // Add cross edges to P(g)
    ASSERT(crossEdges.empty());
    ASSERT(p2st.size() == p.numVertices());
    VertexIt vit, vend;
    for (tie(vit, vend) = p.vertices(); vit != vend; ++vit) {
        Edge st = p2st[vit->index()];
        ASSERT(st.valid());
        Vertex g_to = st.to().in(g);
        ASSERT(g_to.valid());
        if (g2p[g_to.index()].valid()) {
            Edge cross(*vit, g2p[g_to.index()],
                       g2p[g_to.index()].getWeight(), "cross");
            crossEdges.insert(cross);
        }
    }
}


/*** BinaryHeap2Graph implementation ***/

unsigned int BinaryHeap2Graph::addToGraph (Graph& g, vector<Edge>& d2st)
const {
    unsigned int offset = g.numVertices();
    for (unsigned int i = 0; i < this->array.size(); ++i) {
        Edge e = this->array[i]->second; // Get node value; an Edge
                                         // object in the sidetracks graph.
        ASSERT(e.valid());
        Vertex v(g, e.getWeight(), e.getLabel()); // Create vertex
                                                  // representation in g.
        ASSERT(v.valid());
        ASSERT(d2st.size() == v.index());
        d2st.push_back(e);
        ASSERT(offset + i == v.index());
        if (i > 0) {
            Vertex pv(offset + parentIndex(i), g);
                // Get parent node's vertex representation in g.
            ASSERT(pv.valid());
            Edge e(pv, v); // Create edge from parent vertex to
                           // current vertex.
        }
    }
    return offset;
}


/*** BinaryMarkedHeap2Graph implementation ***/

unsigned int BinaryMarkedHeap2Graph::addToGraph (
    Graph& g,
```

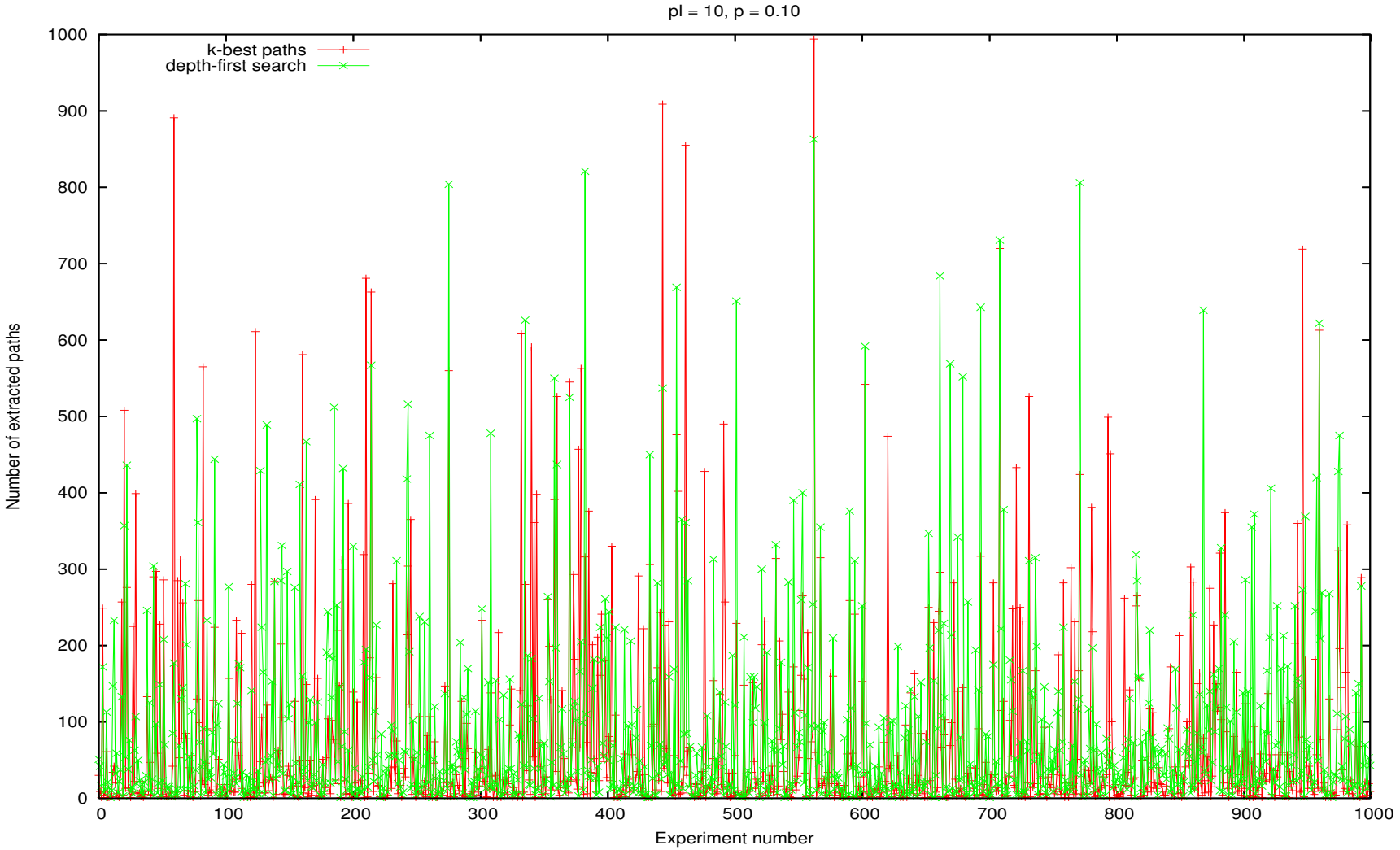
```
    vector<Edge>& d2st,
    map<Edge, Vertex>& stMap
) const {
    if (numElements() == 0) return static_cast<unsigned int>(-1);
#ifdef DEBUG_CODE
    unsigned int oldNum = g.numVertices();
#endif // DEBUG_CODE
    for (unsigned int i = 0; i < this->array.size(); ++i) {
        Edge e = this->array[i]->second; // Get node value; an Edge
                                         // object in the sidetracks graph.
        ASSERT(e.valid());
        bool m = marked[this->array[i]]; // Get flag indicating whether
                                         // or not this node is marked.
        Vertex v;
        ASSERT(!v.valid());
        if (m) { // Node is marked. We must add a new vertex to g
            v = Vertex(g, e.getWeight(), e.getLabel());
                // Create vertex representation in g.
            ASSERT(d2st.size() == v.index());
            d2st.push_back(e);
            stMap[e] = v;
            ASSERT(v.index() >= oldNum);
        }
        else { // Node is unmarked. We must use an existing vertex in g.
            ASSERT(stMap.count(e) == 1);
            v = stMap[e]; // Retrieve vertex from stMap.
            ASSERT(v.index() < oldNum);
        }
        if (i > 0) { // This is not the root node and therefore has a
                     // parent. Add edge from parent to current node's
                     // vertex.
            Edge pe = this->array[parentIndex(i)]->second;
            ASSERT(pe.valid());
            ASSERT(stMap.count(pe) == 1);
            bool pm = marked[this->array[parentIndex(i)]];
            Vertex pv = stMap[pe]; // Get parent node's vertex
                                   // representation in g.
            ASSERT(pv.valid());
            if (pm || m) Edge e(pv, v); // Either current or parent node
                                        // (or both) is marked. Add an
                                        // edge from pv to v.
            else ASSERT(g.hasEdge(pv, v)); // There should already be an
                                           // edge between pv and v
        }
    }
    ASSERT(g.numVertices() >= oldNum);
    return stMap[this->array[0]->second].index();
}
```

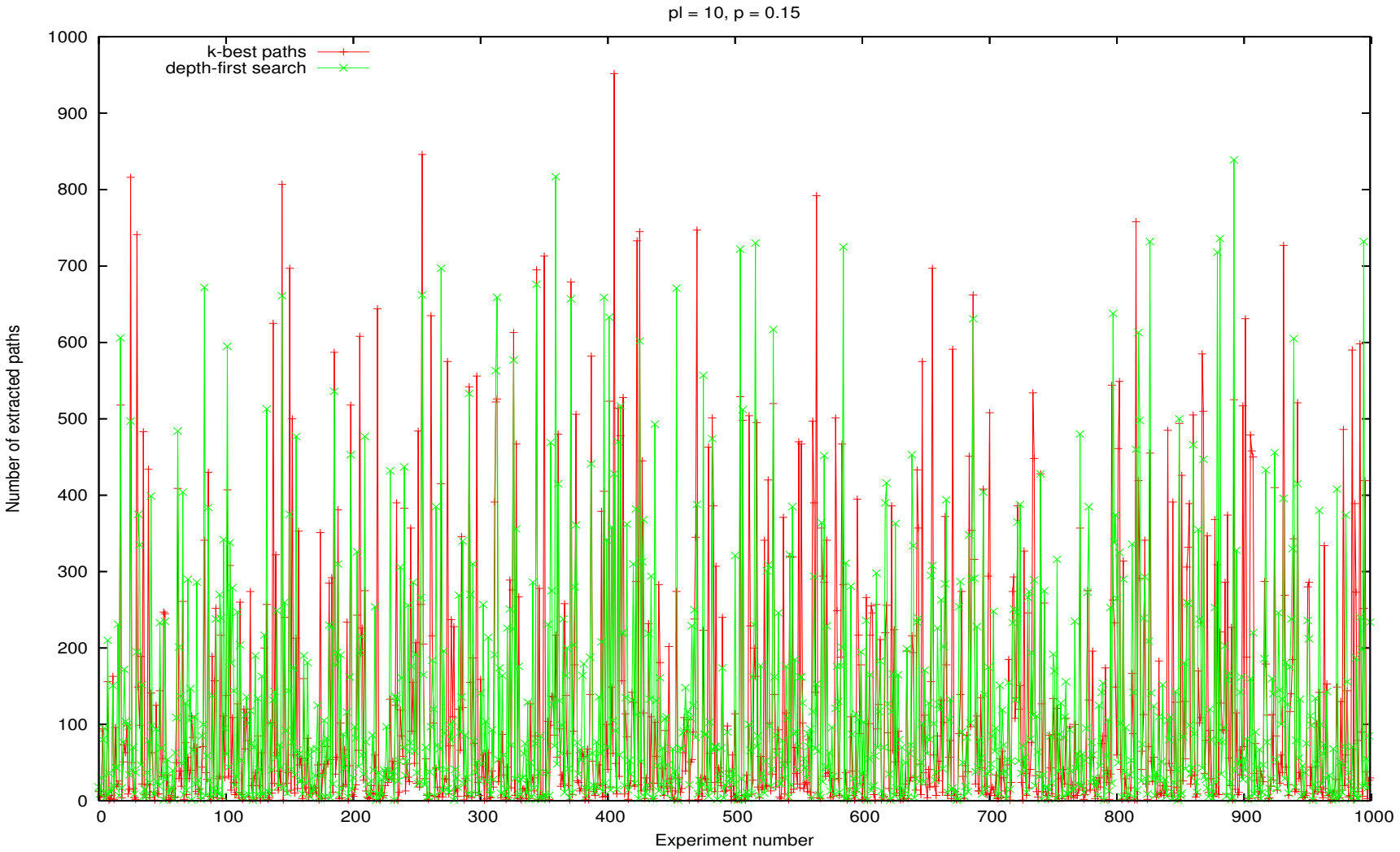
B Experimental results

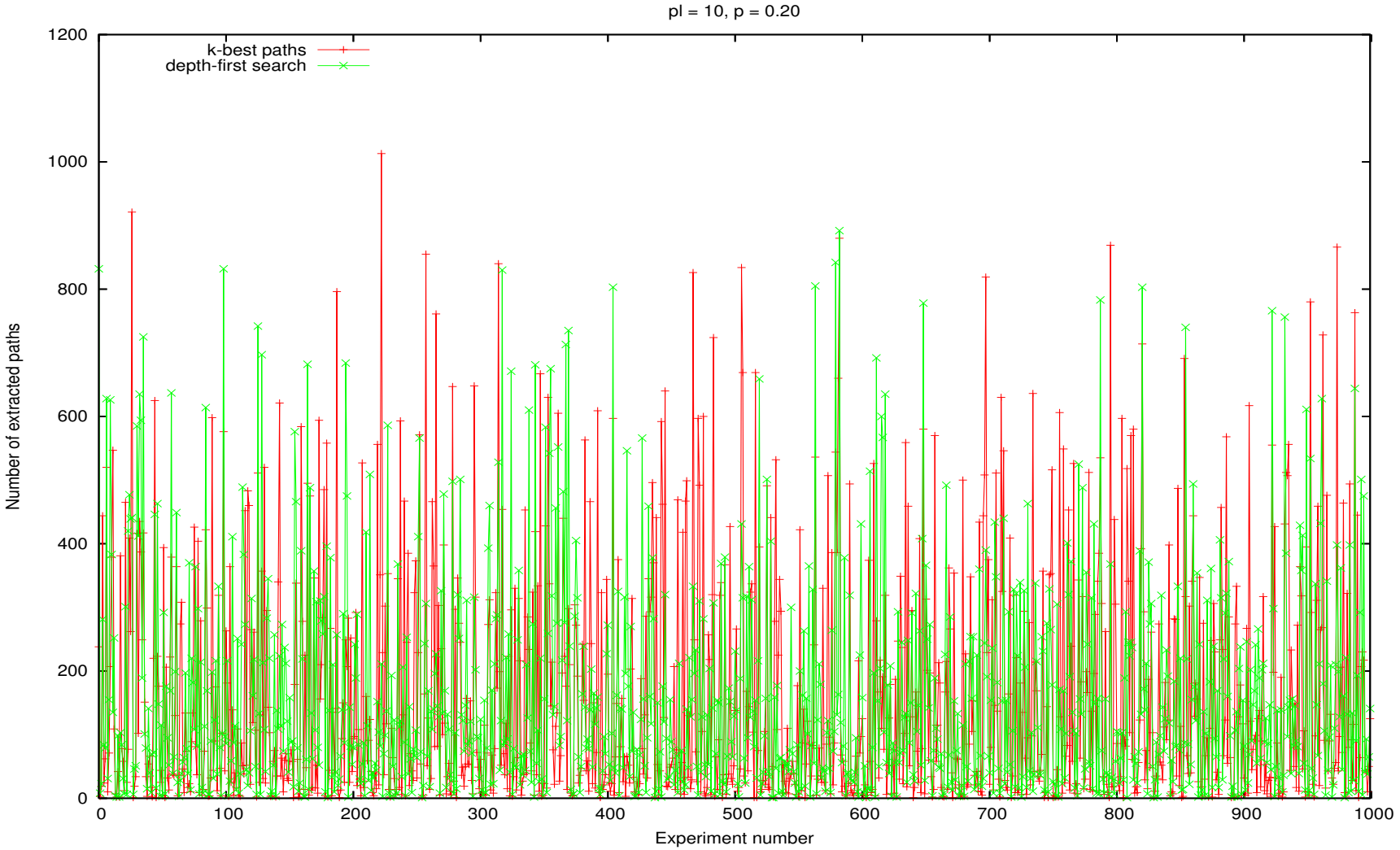
This appendix contains graphs that show the actual number of extracted paths for all the performed experiments, both for the k-best paths algorithm and the depth-first search.

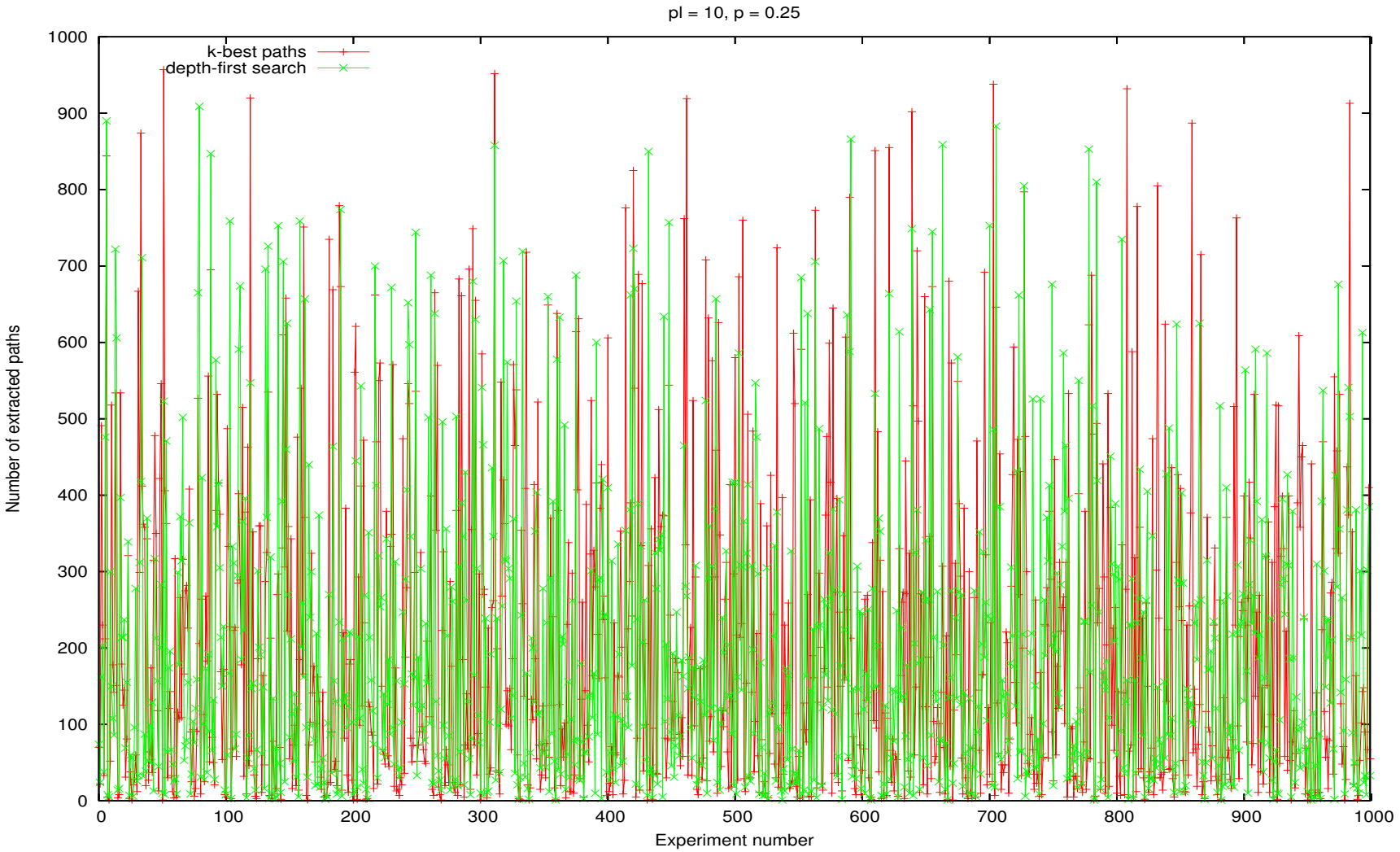

pl = 10, p = 0.00
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600
700
800
900
1000

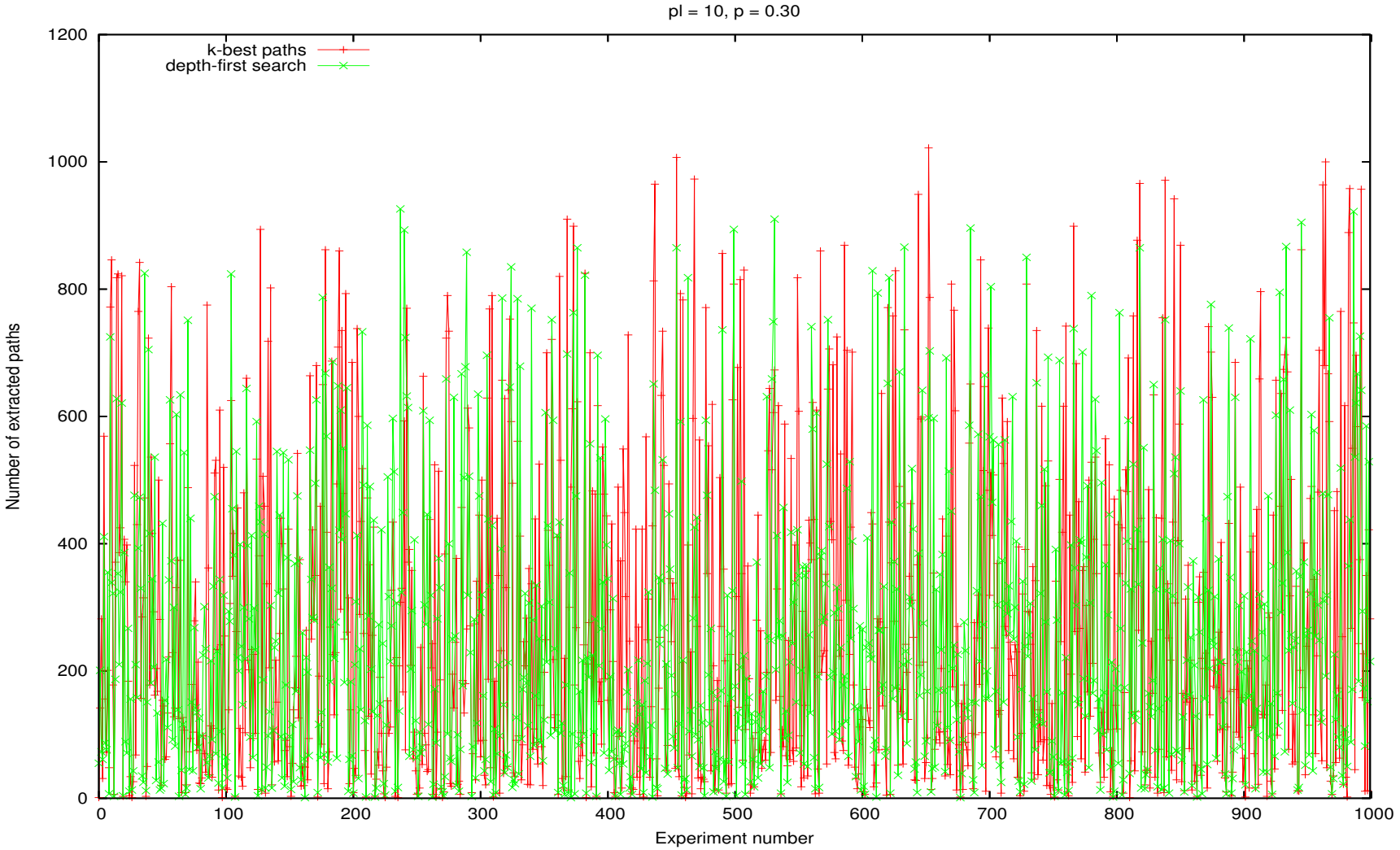

pl = 10, p = 0.05
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100
200
300
400
500
600
700
800
900
1000

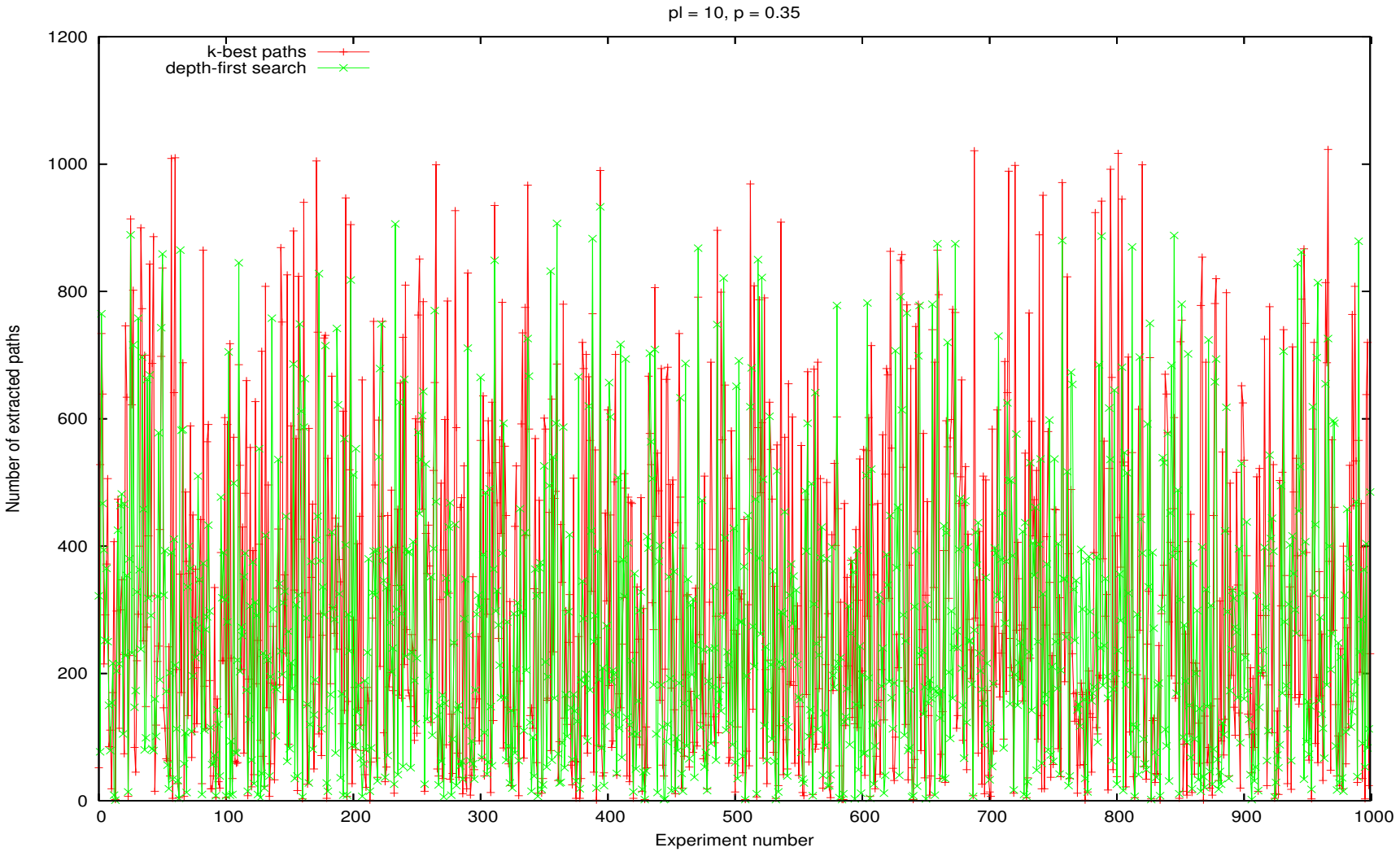

pl = 10, p = 0.10
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100
200
300
400
500
600
700
800
900
1000

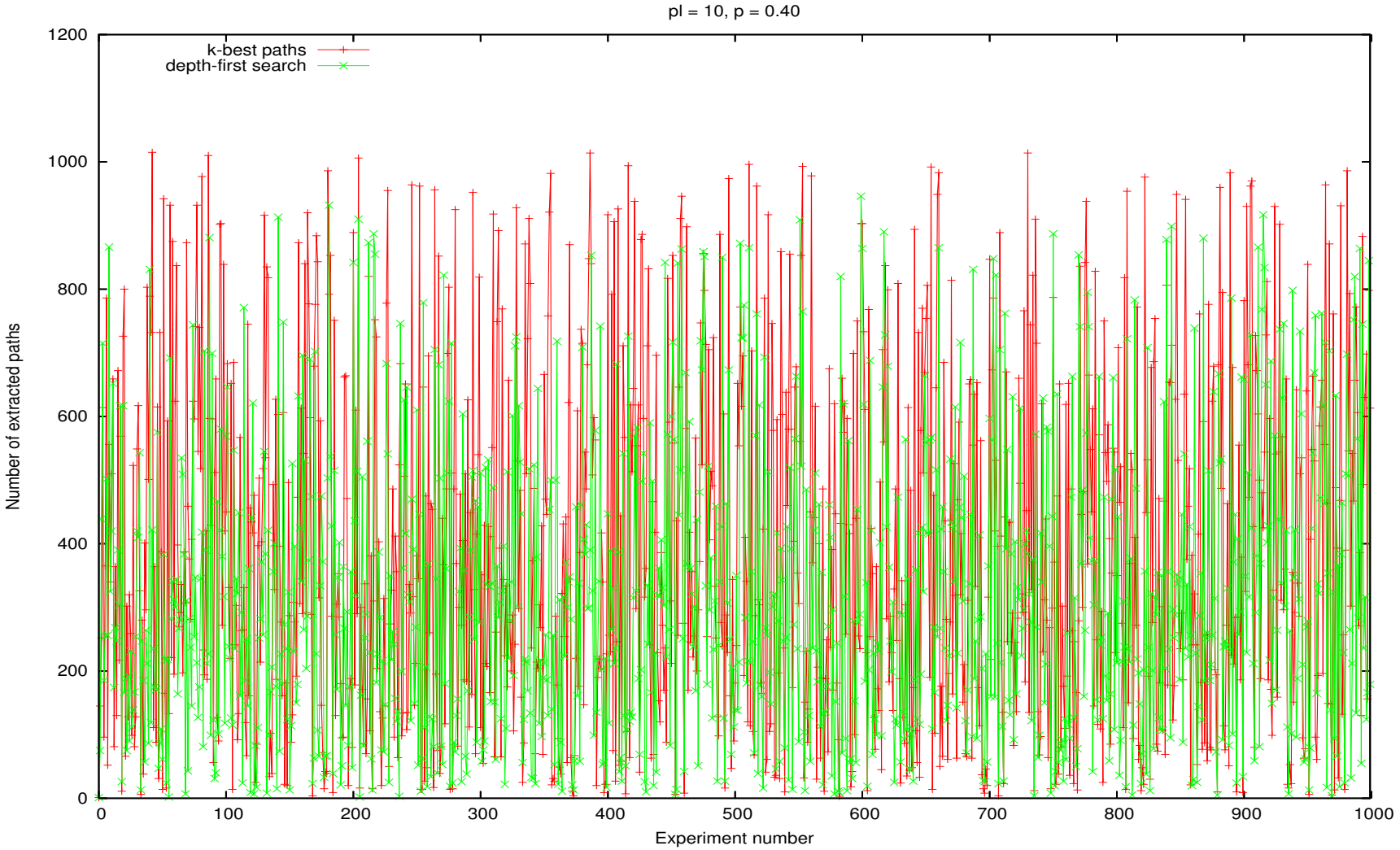

pl = 10, p = 0.15
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100
200
300
400
500
600
700
800
900
1000

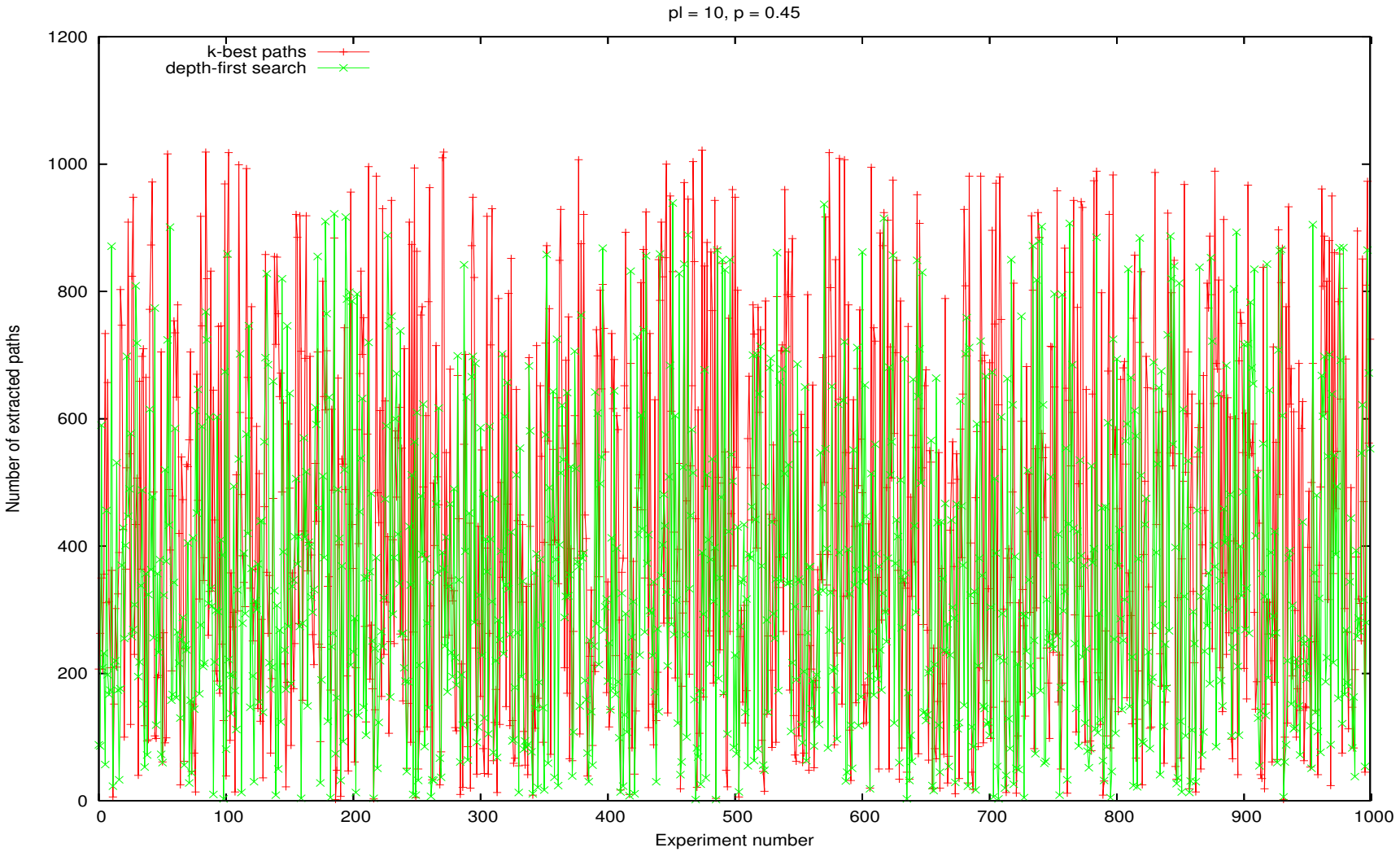

pl = 10, p = 0.20
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
800
900
1000

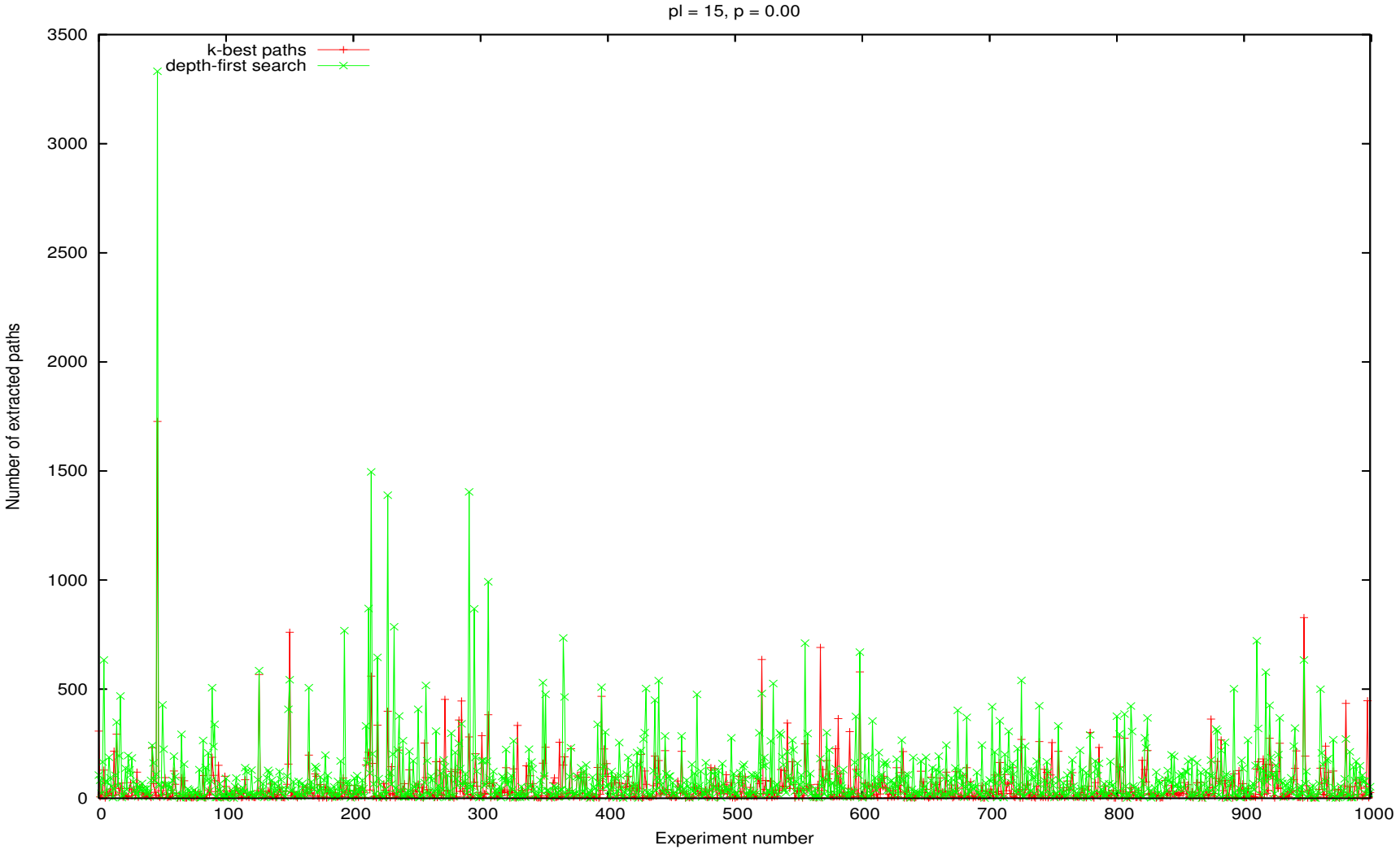

pl = 10, p = 0.25
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100
200
300
400
500
600
700
800
900
1000

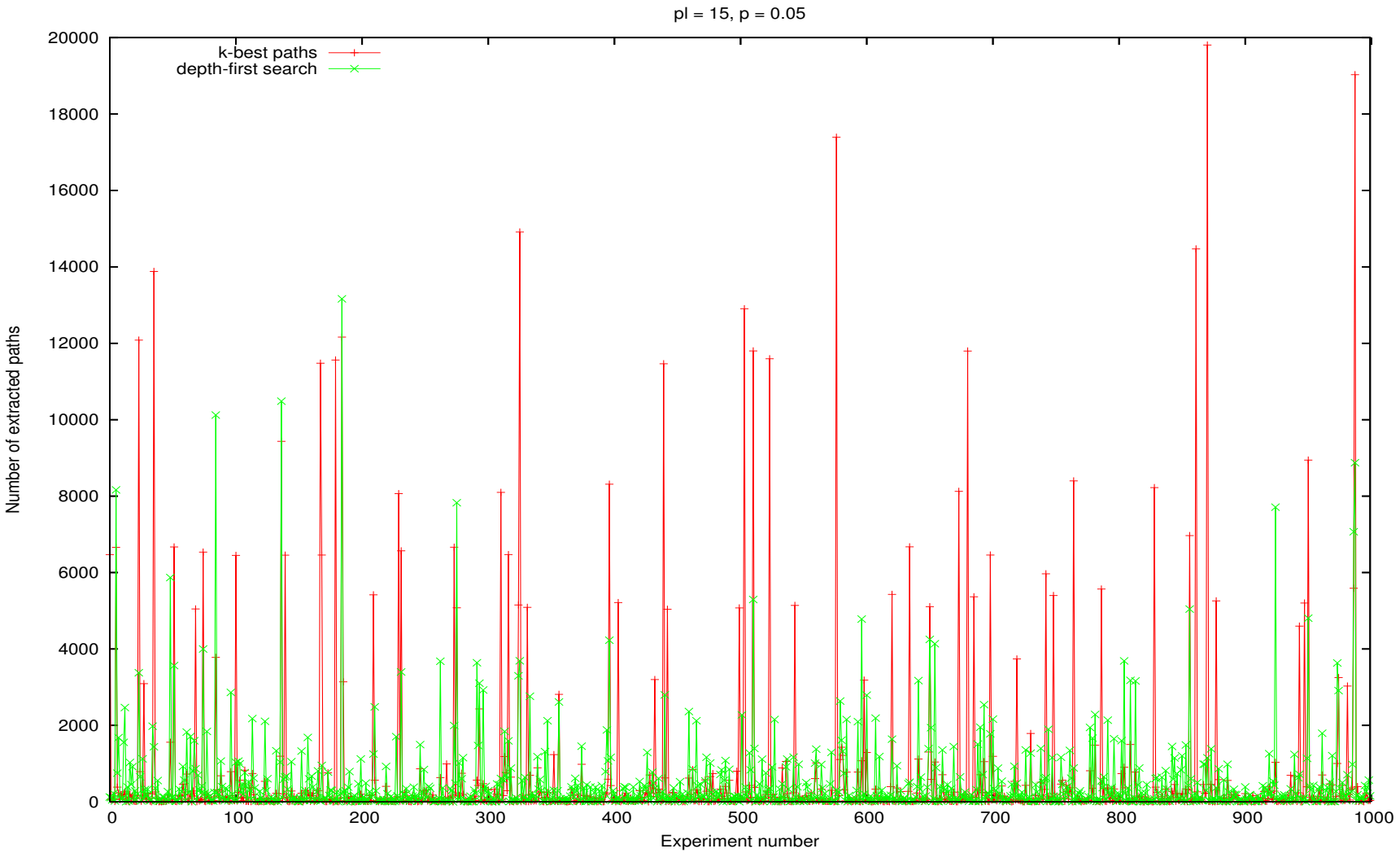

pl = 10, p = 0.30
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
800
900
1000

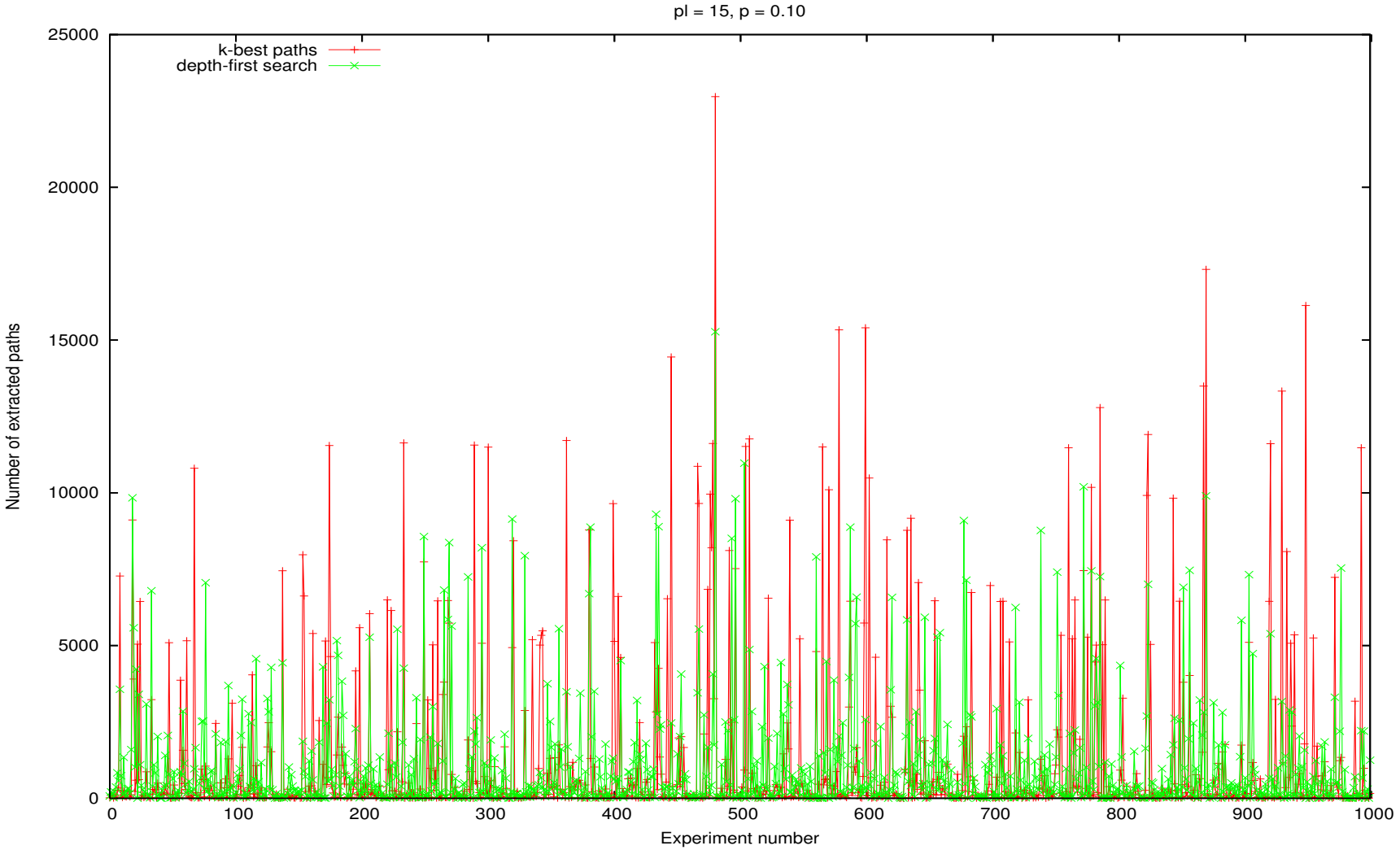

pl = 10, p = 0.35
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
800
900
1000

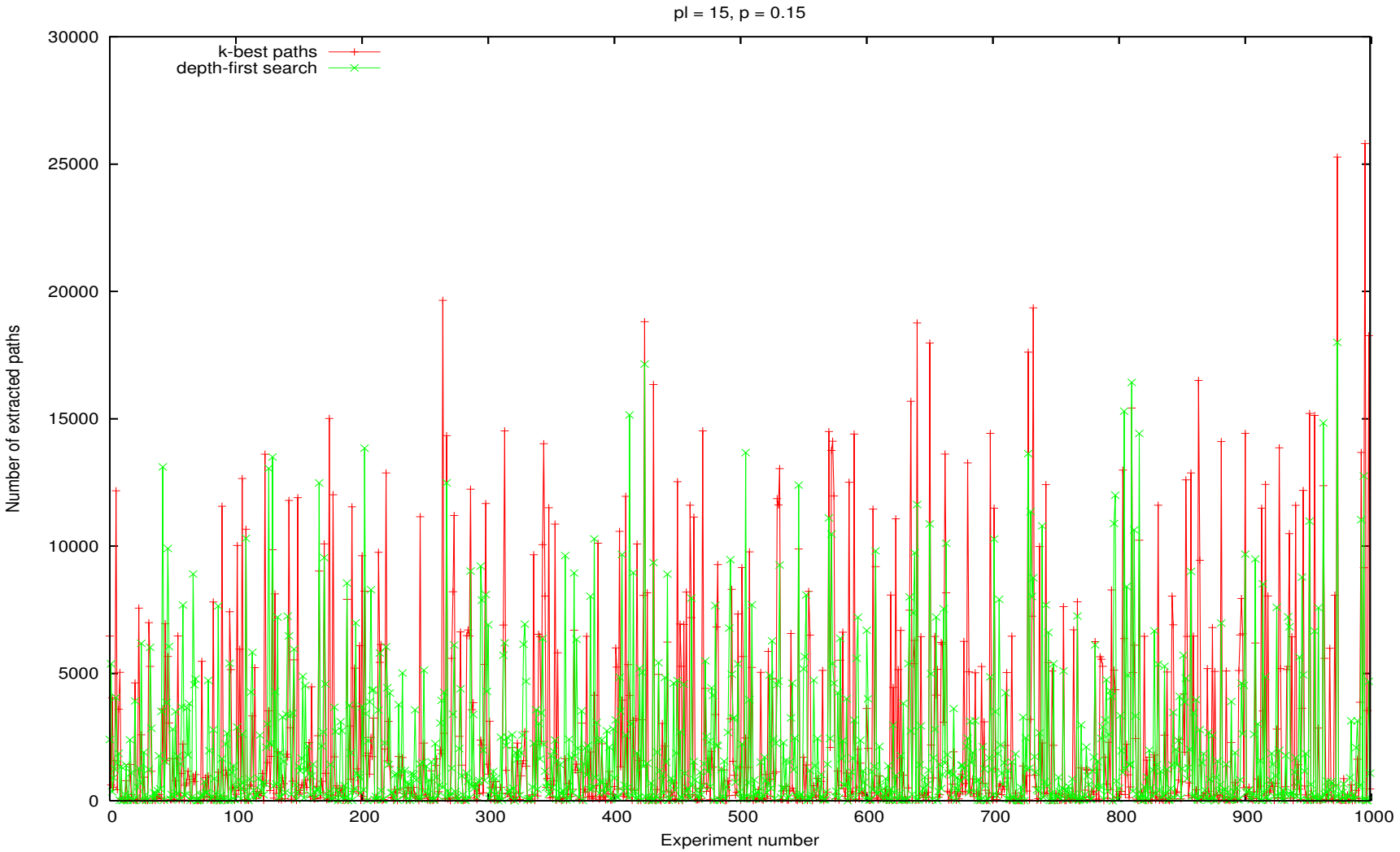

pl = 10, p = 0.40
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
800
900
1000

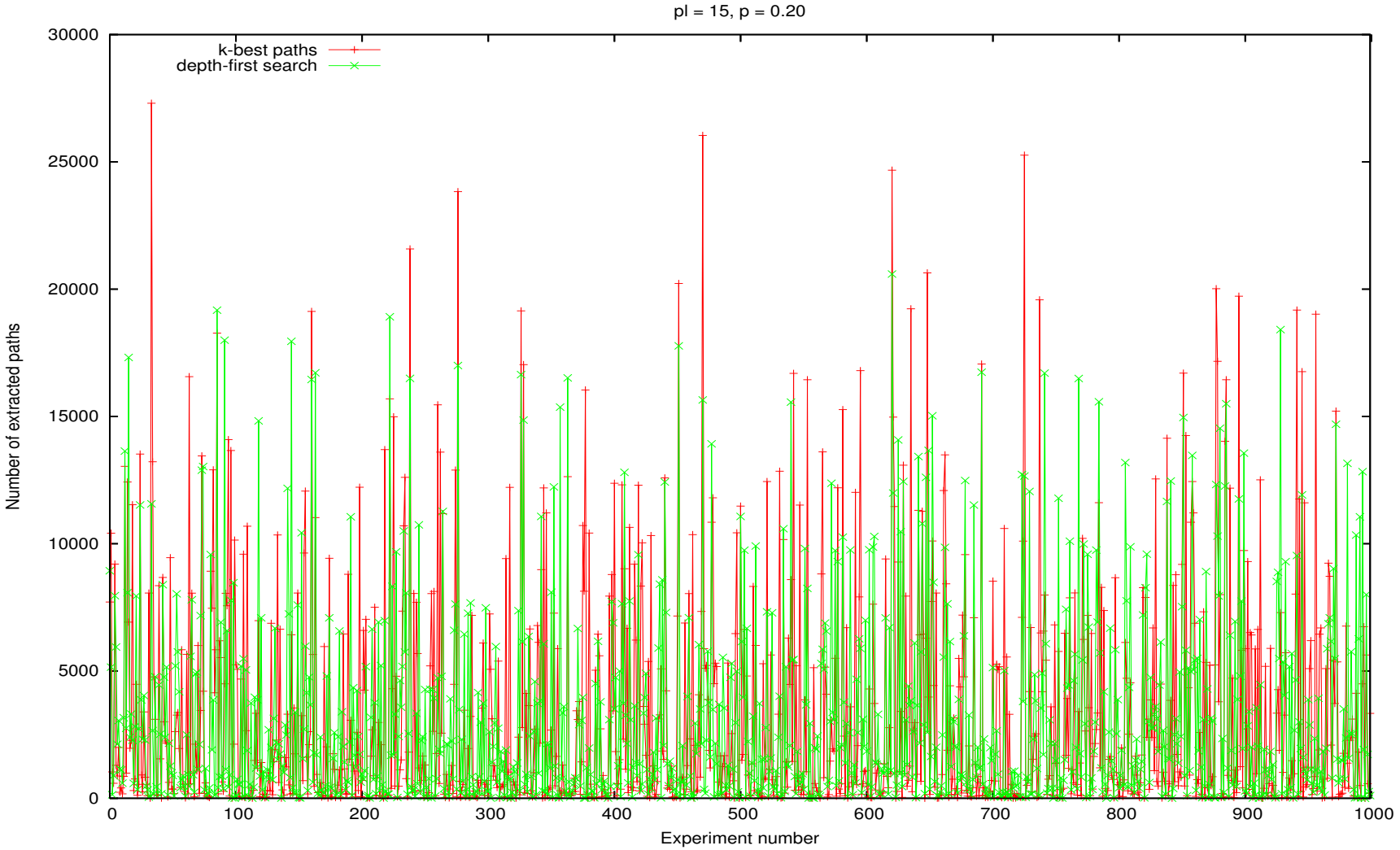

pl = 10, p = 0.45
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
200
400
600
800
1000
1200
0
100
200
300
400
500
600
700
800
900
1000

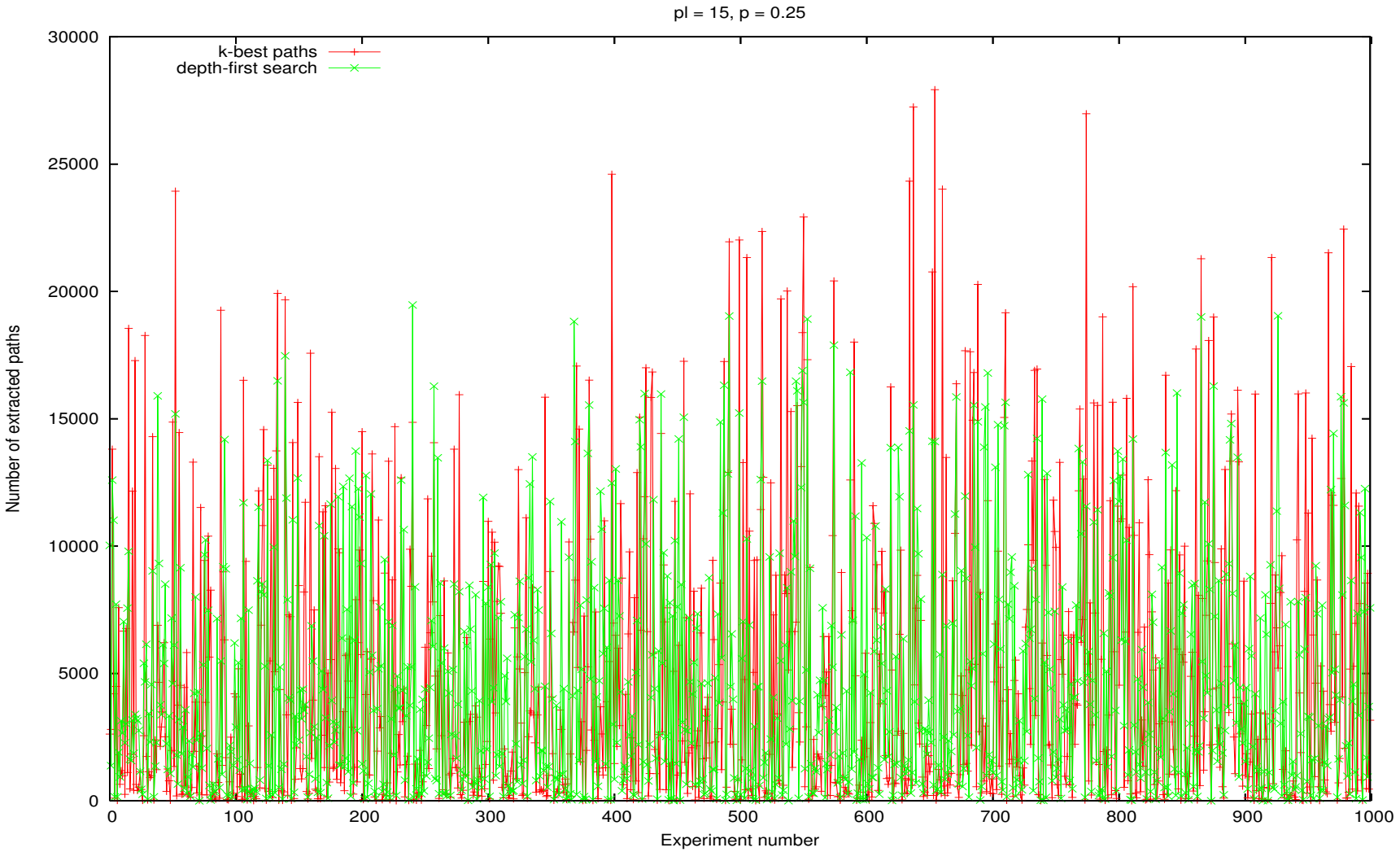

pl = 15, p = 0.00
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
500
1000
1500
2000
2500
3000
3500
0
100
200
300
400
500
600
700
800
900
1000

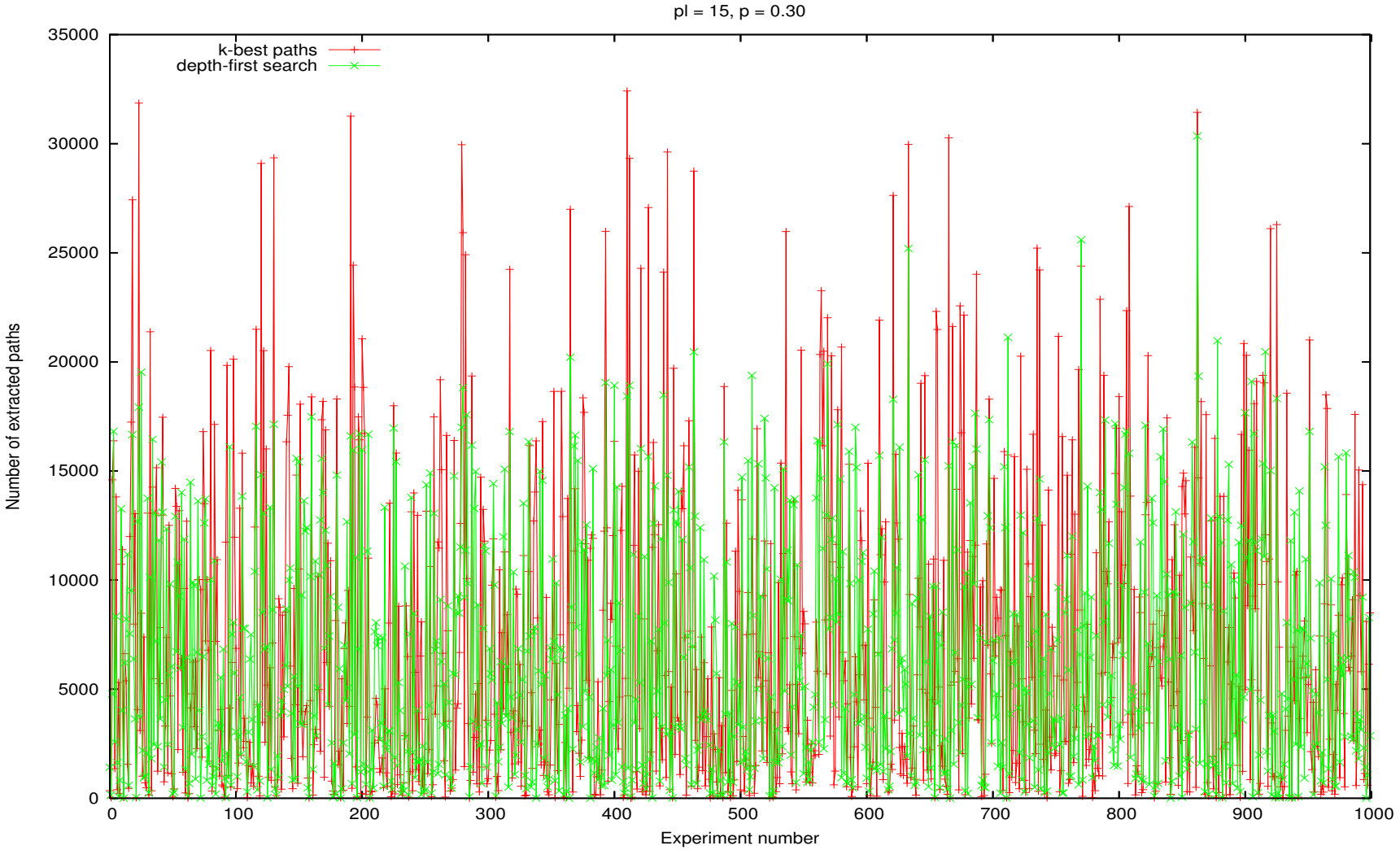

pl = 15, p = 0.05
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
2000
4000
6000
8000
10000
12000
14000
16000
18000
20000
0
100
200
300
400
500
600
700
800
900
1000

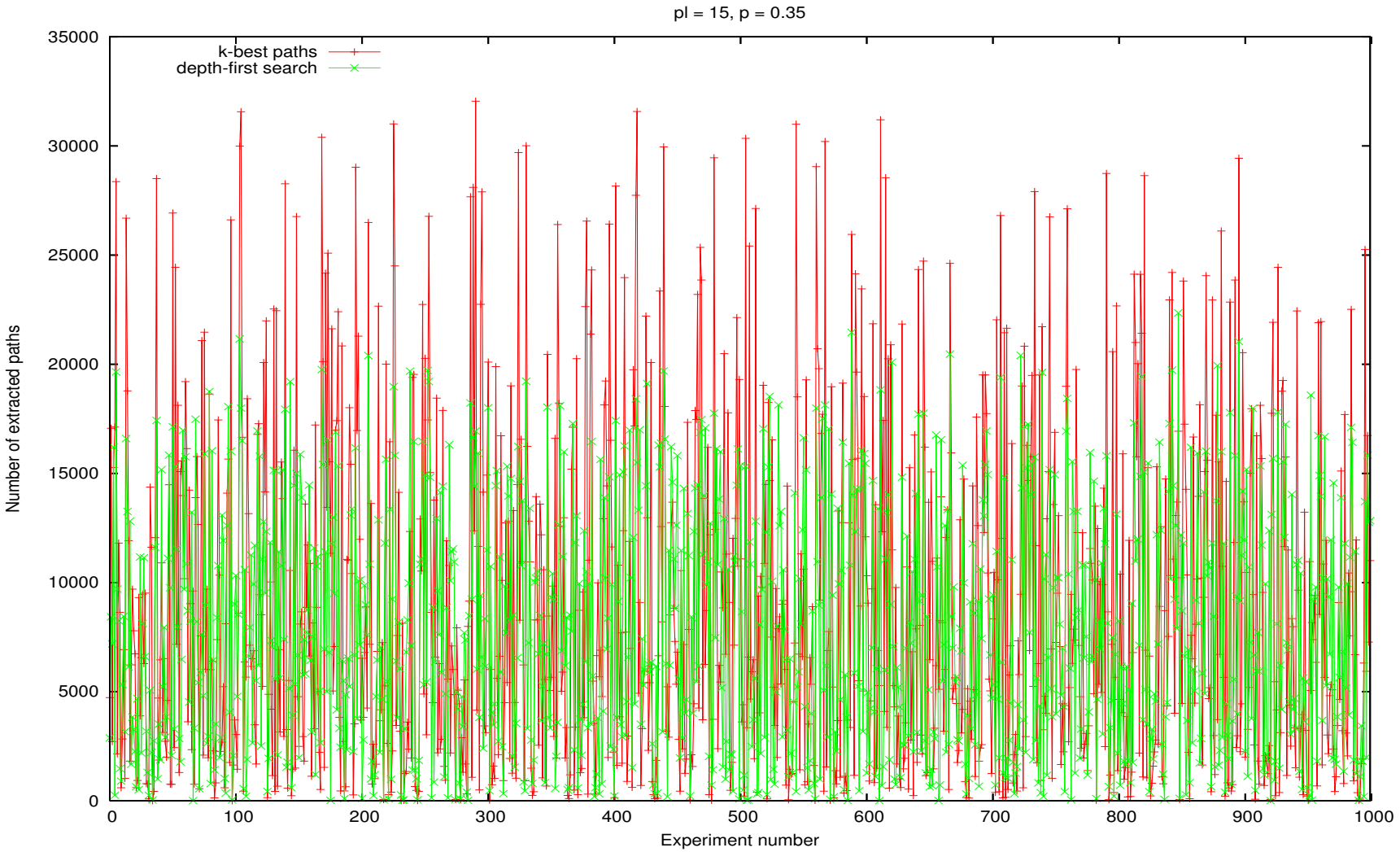

pl = 15, p = 0.10
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
0
100
200
300
400
500
600
700
800
900
1000

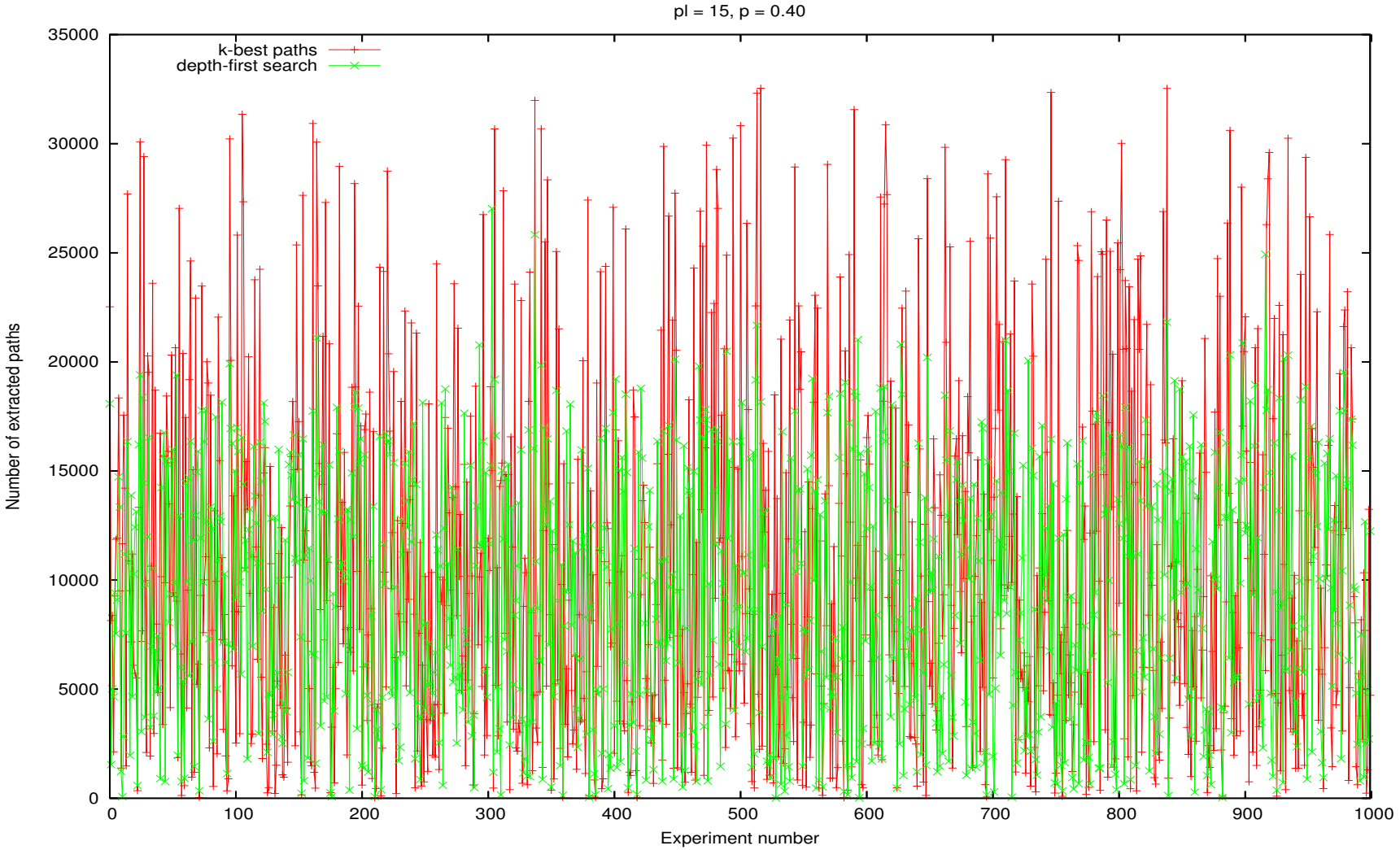

pl = 15, p = 0.15
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
0
100
200
300
400
500
600
700
800
900
1000

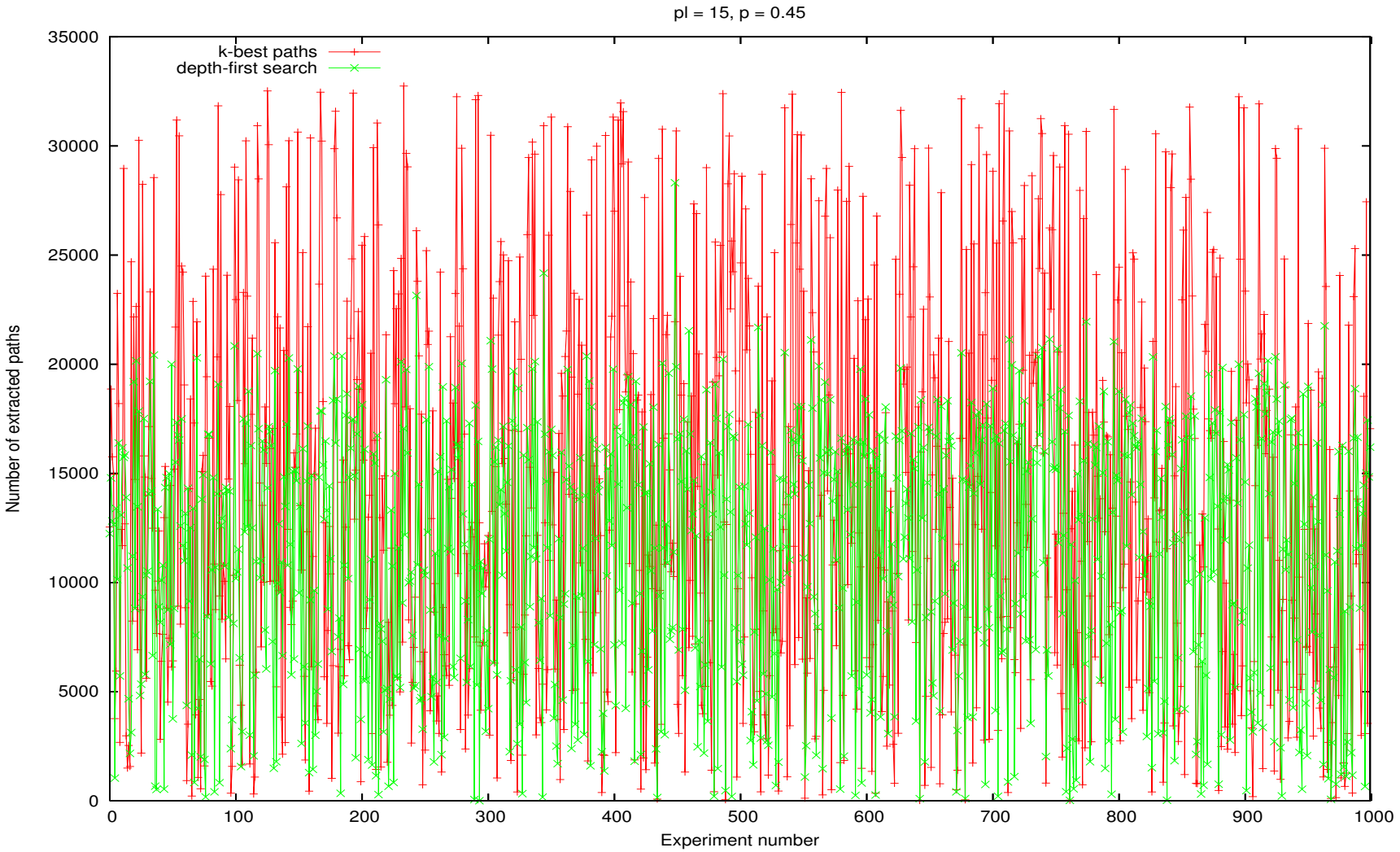

pl = 15, p = 0.20
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
0
100
200
300
400
500
600
700
800
900
1000

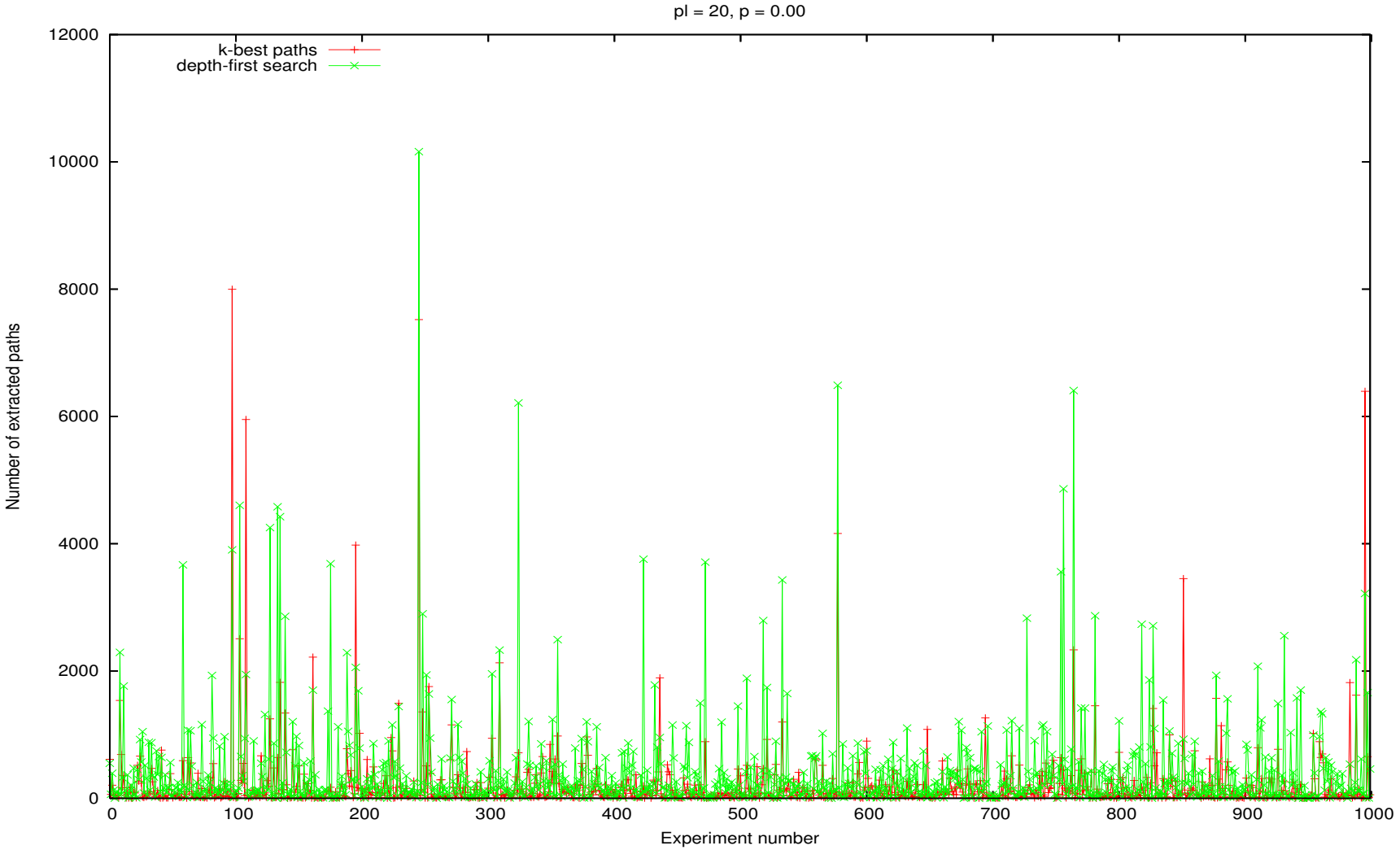

pl = 15, p = 0.25
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
0
100
200
300
400
500
600
700
800
900
1000

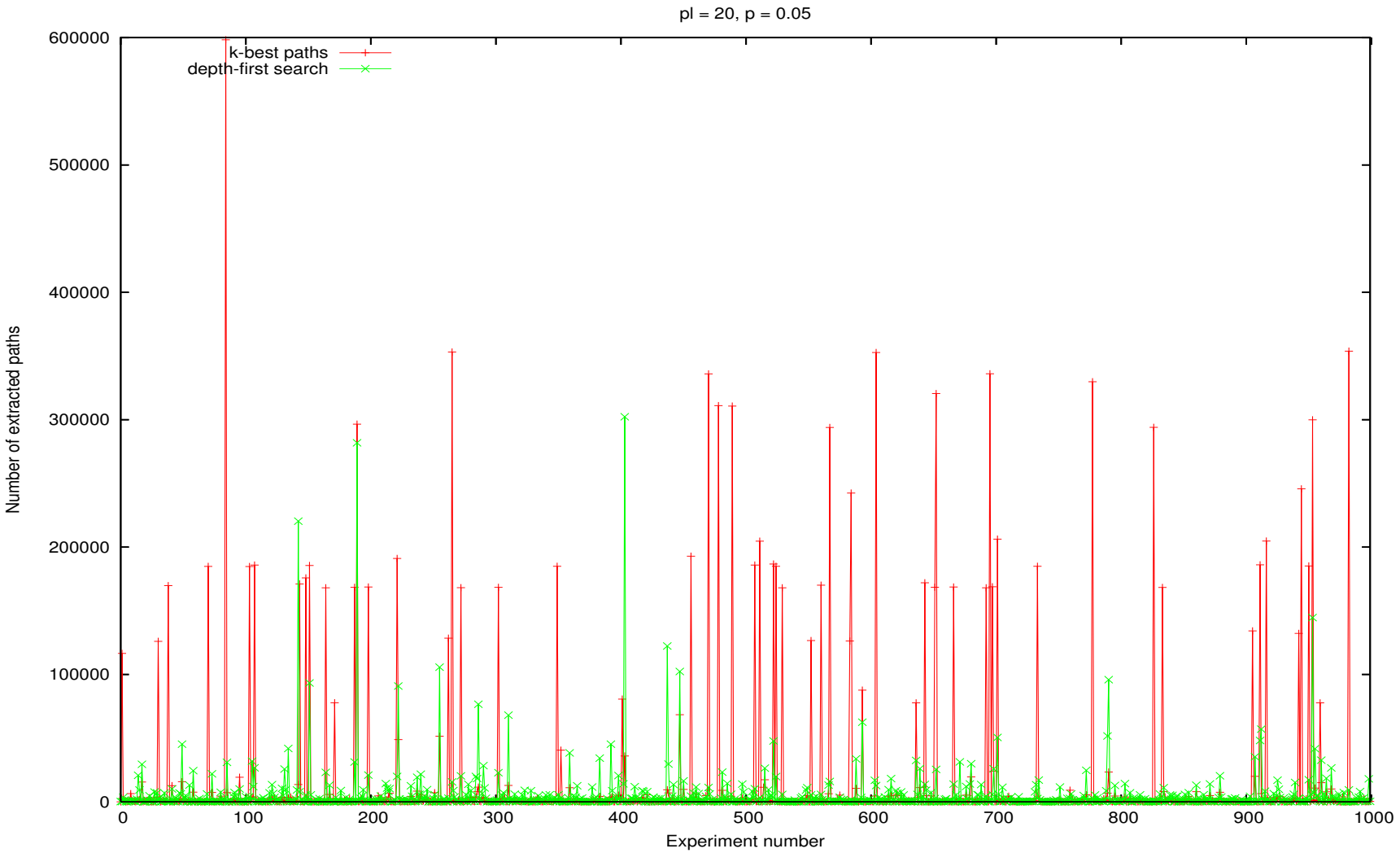

pl = 15, p = 0.30
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
35000
0
100
200
300
400
500
600
700
800
900
1000

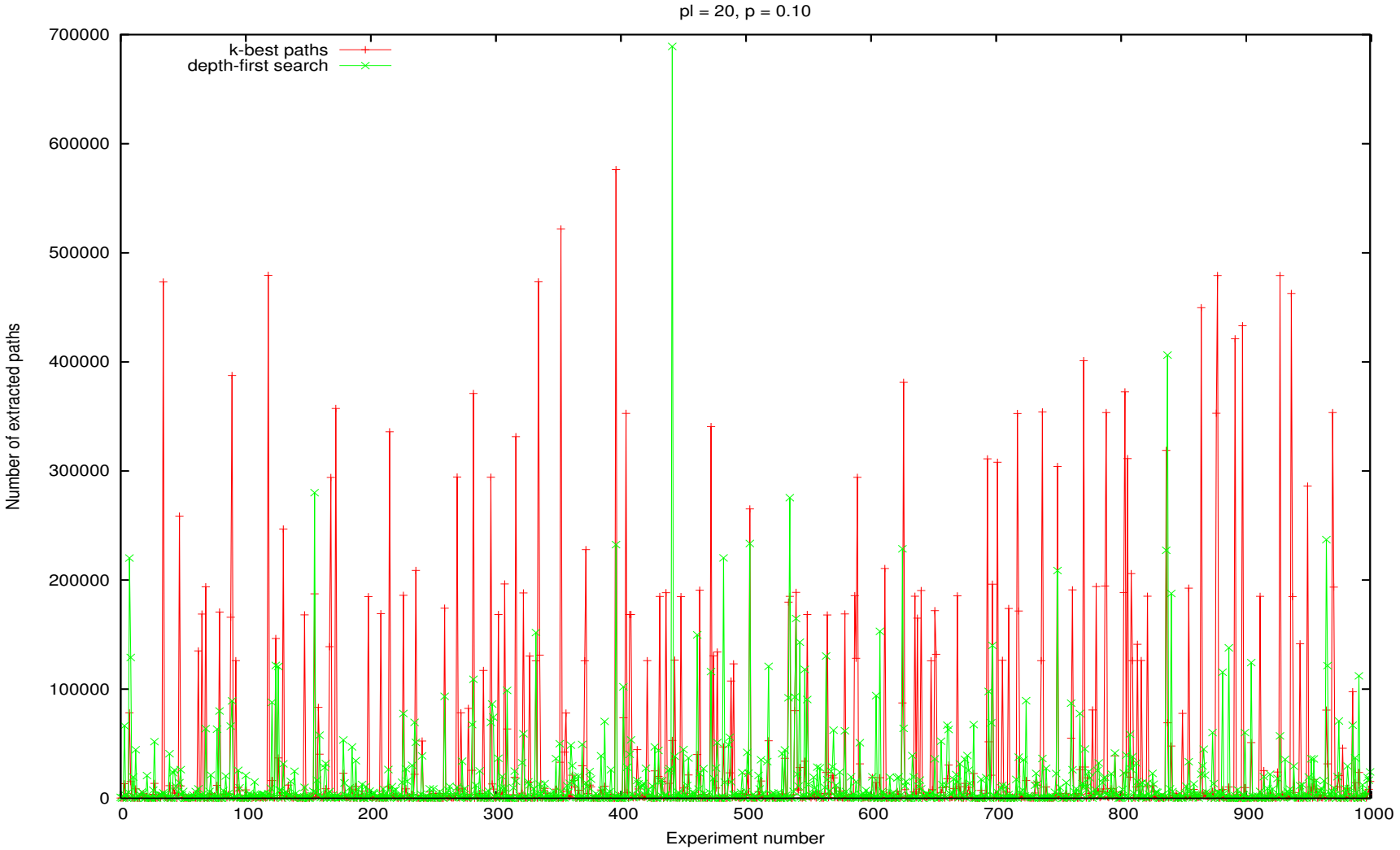

pl = 15, p = 0.35
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
35000
0
100
200
300
400
500
600
700
800
900
1000

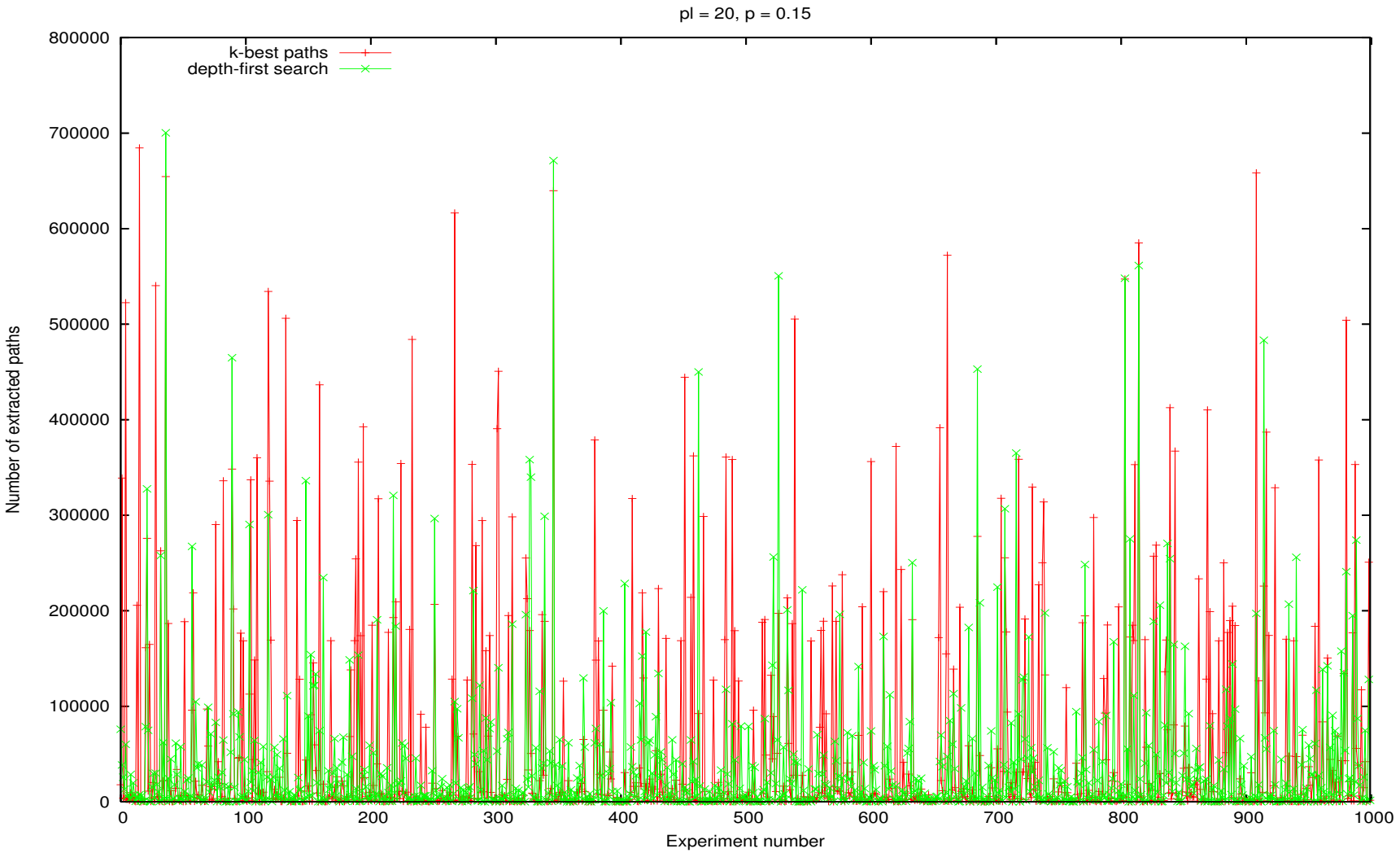

pl = 15, p = 0.40
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
35000
0
100
200
300
400
500
600
700
800
900
1000

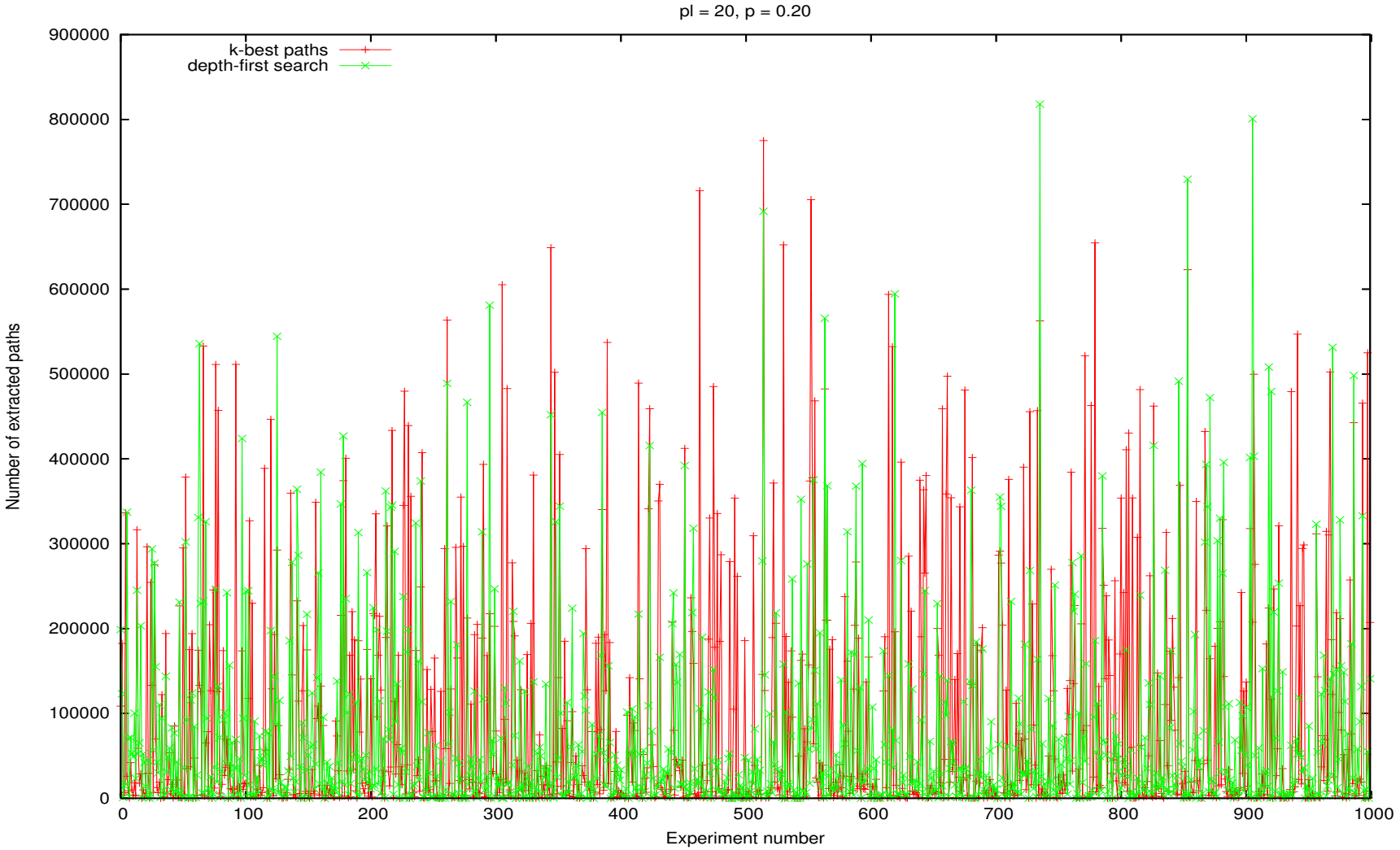

pl = 15, p = 0.45
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
5000
10000
15000
20000
25000
30000
35000
0
100
200
300
400
500
600
700
800
900
1000

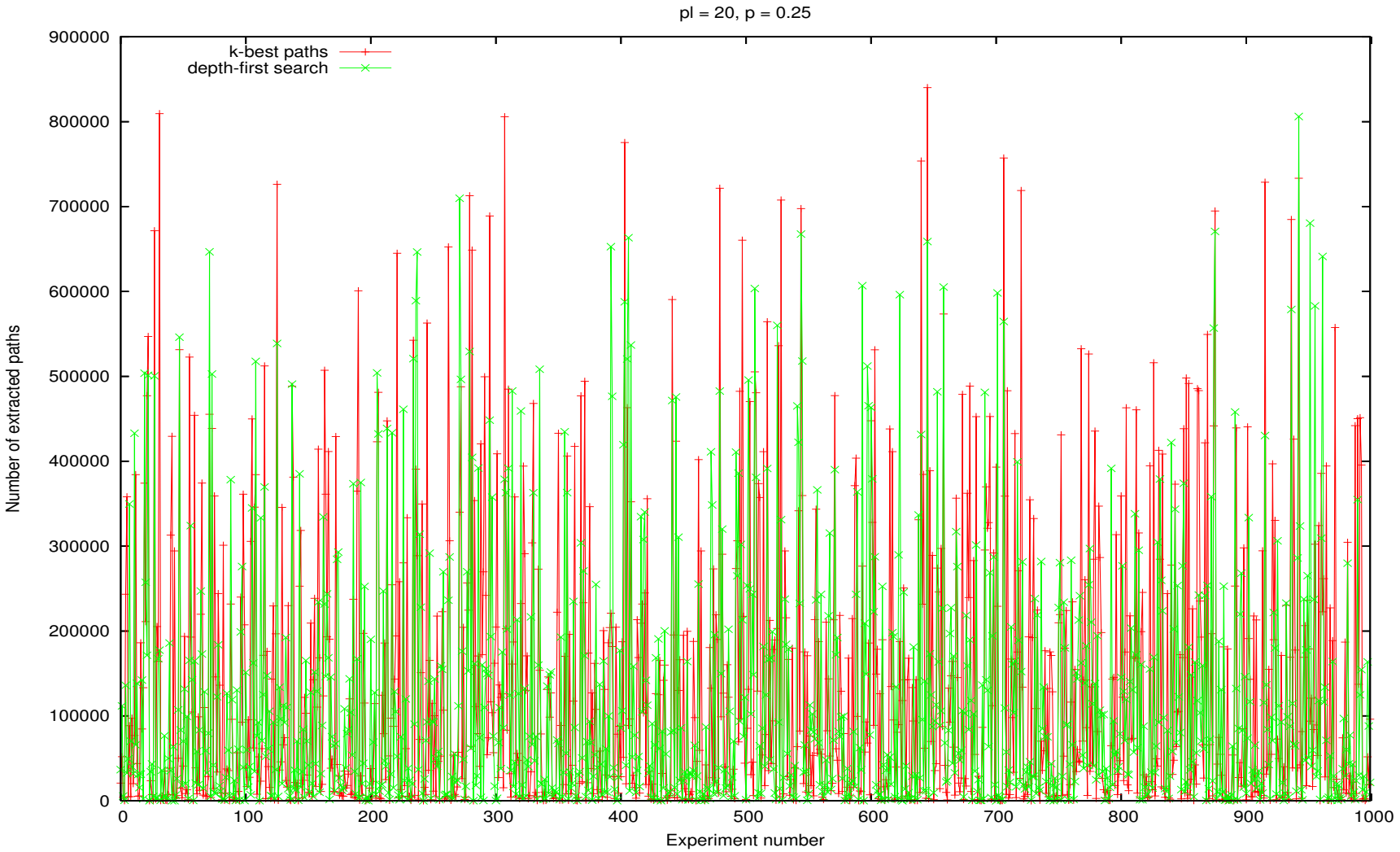

pl = 20, p = 0.00
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
2000
4000
6000
8000
10000
12000
0
100
200
300
400
500
600
700
800
900
1000

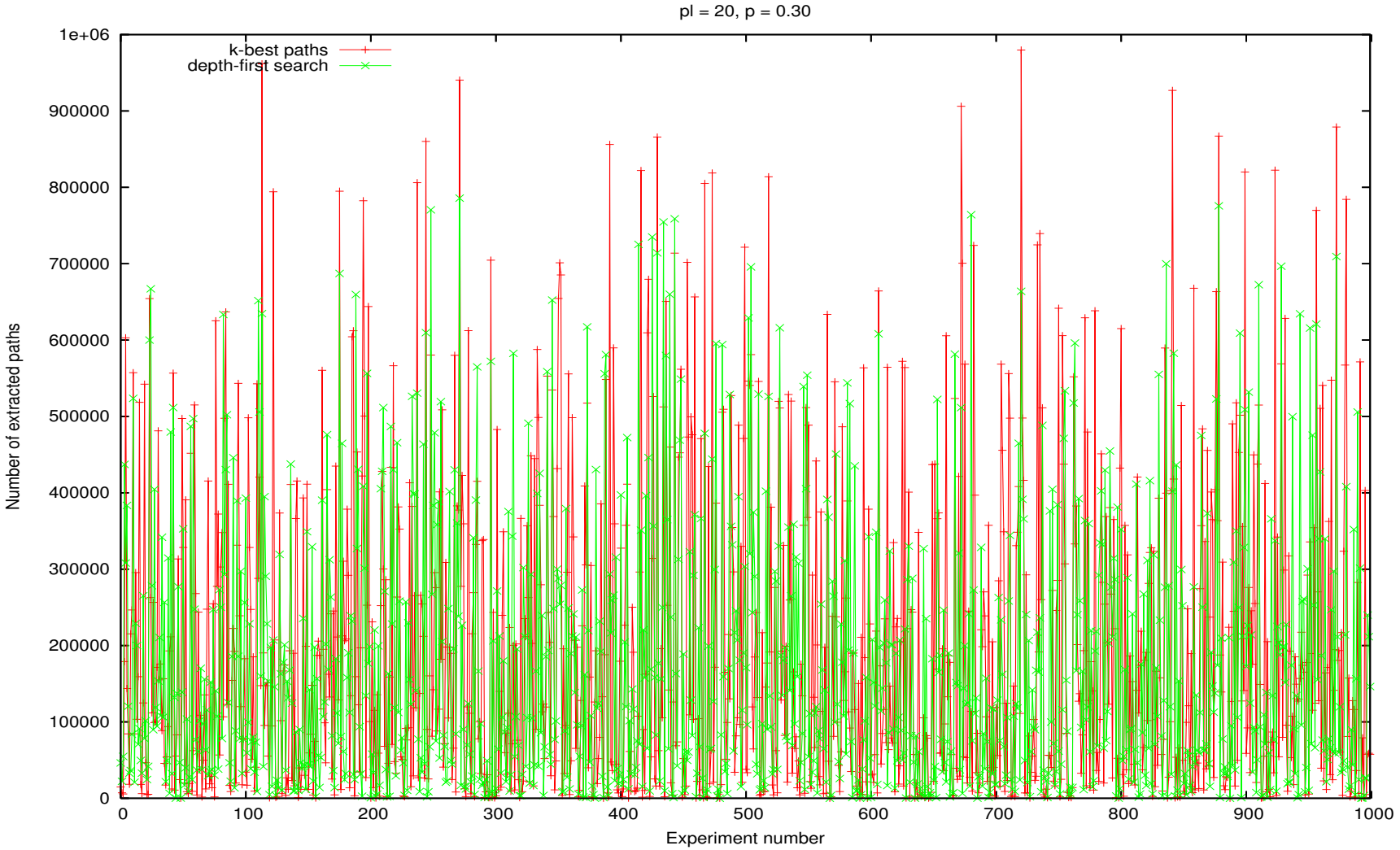

pl = 20, p = 0.05
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100000
200000
300000
400000
500000
600000
0
100
200
300
400
500
600
700
800
900
1000

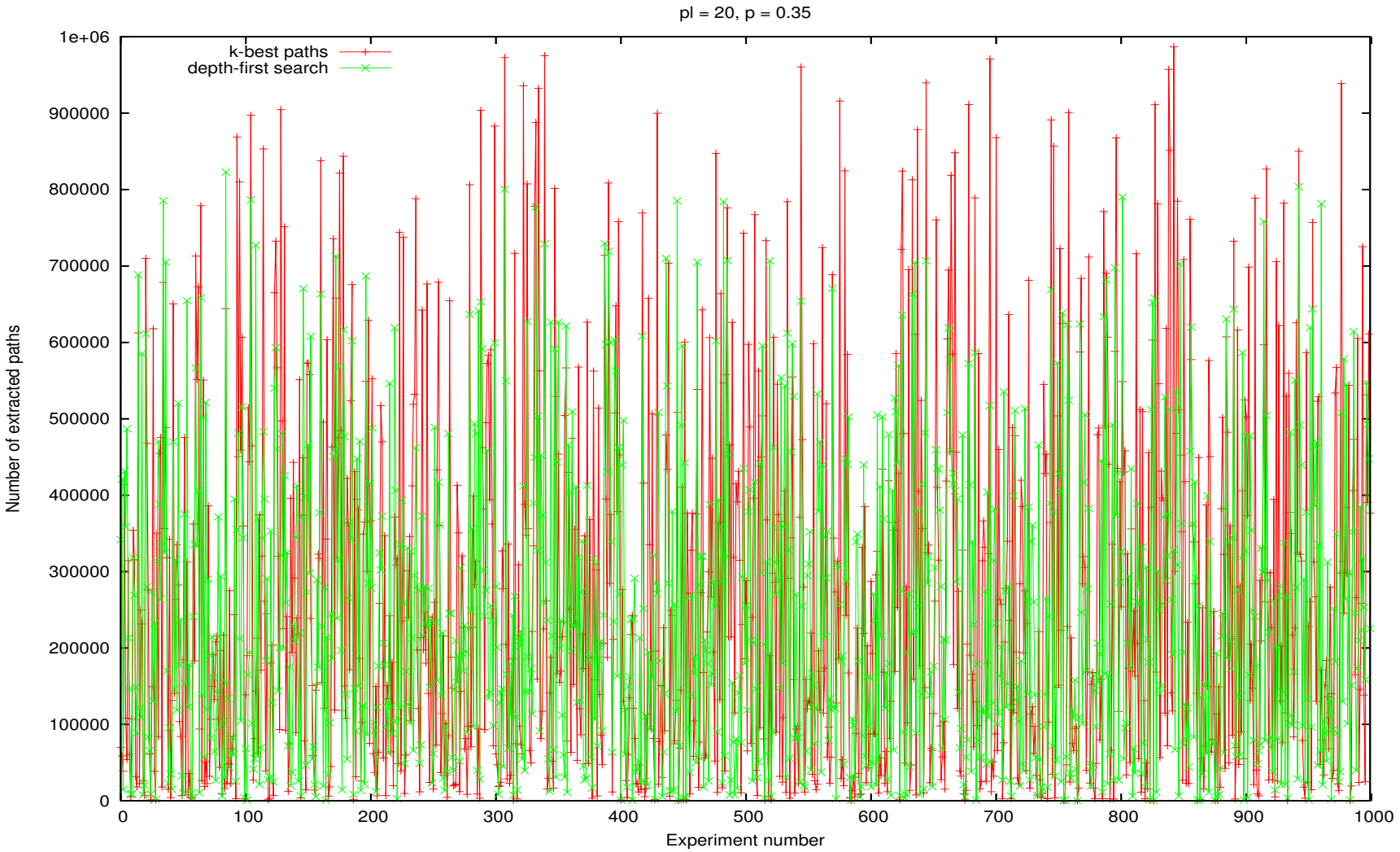

pl = 20, p = 0.10
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100000
200000
300000
400000
500000
600000
700000
0
100
200
300
400
500
600
700
800
900
1000

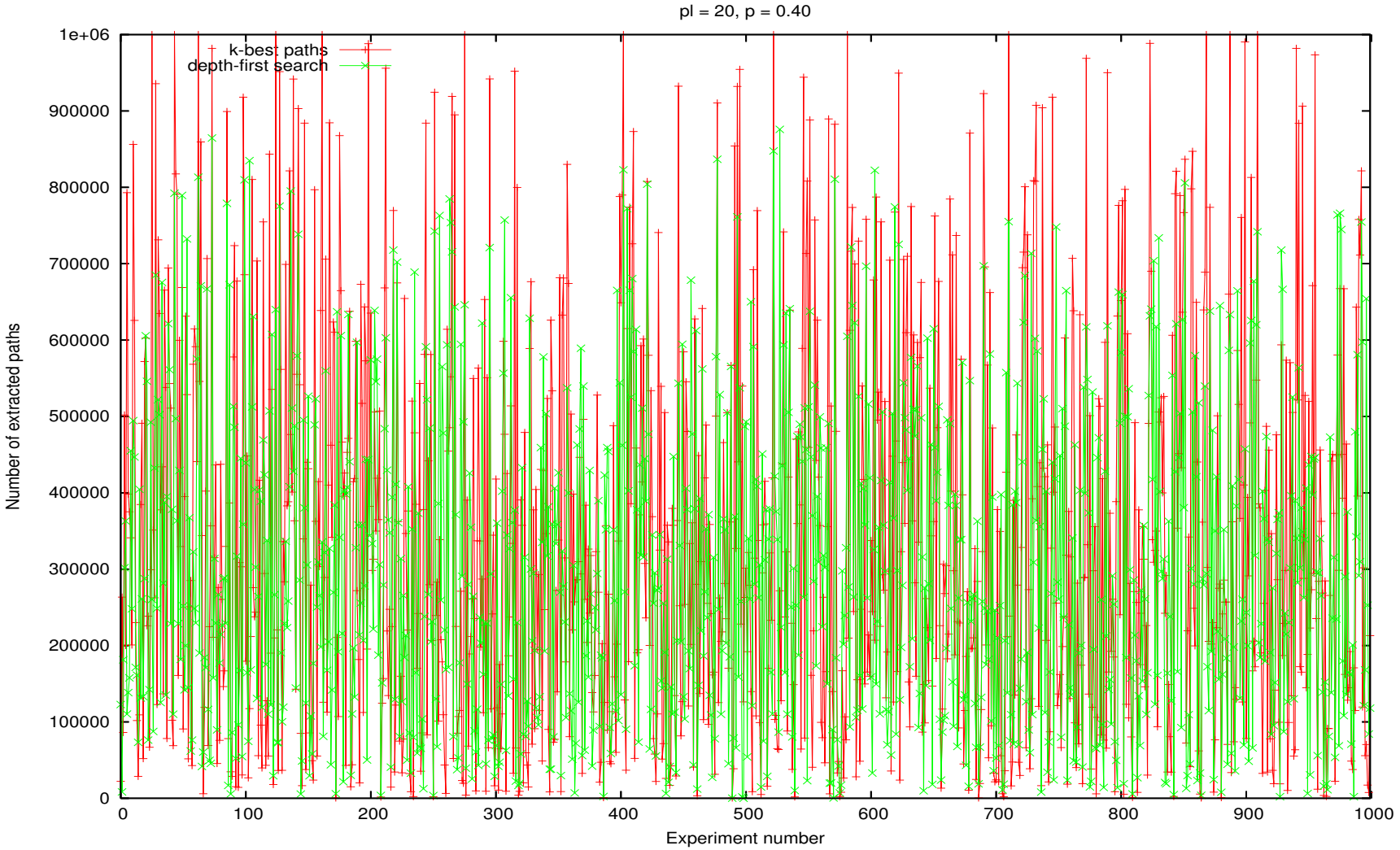

pl = 20, p = 0.15
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100000
200000
300000
400000
500000
600000
700000
800000
0
100
200
300
400
500
600
700
800
900
1000

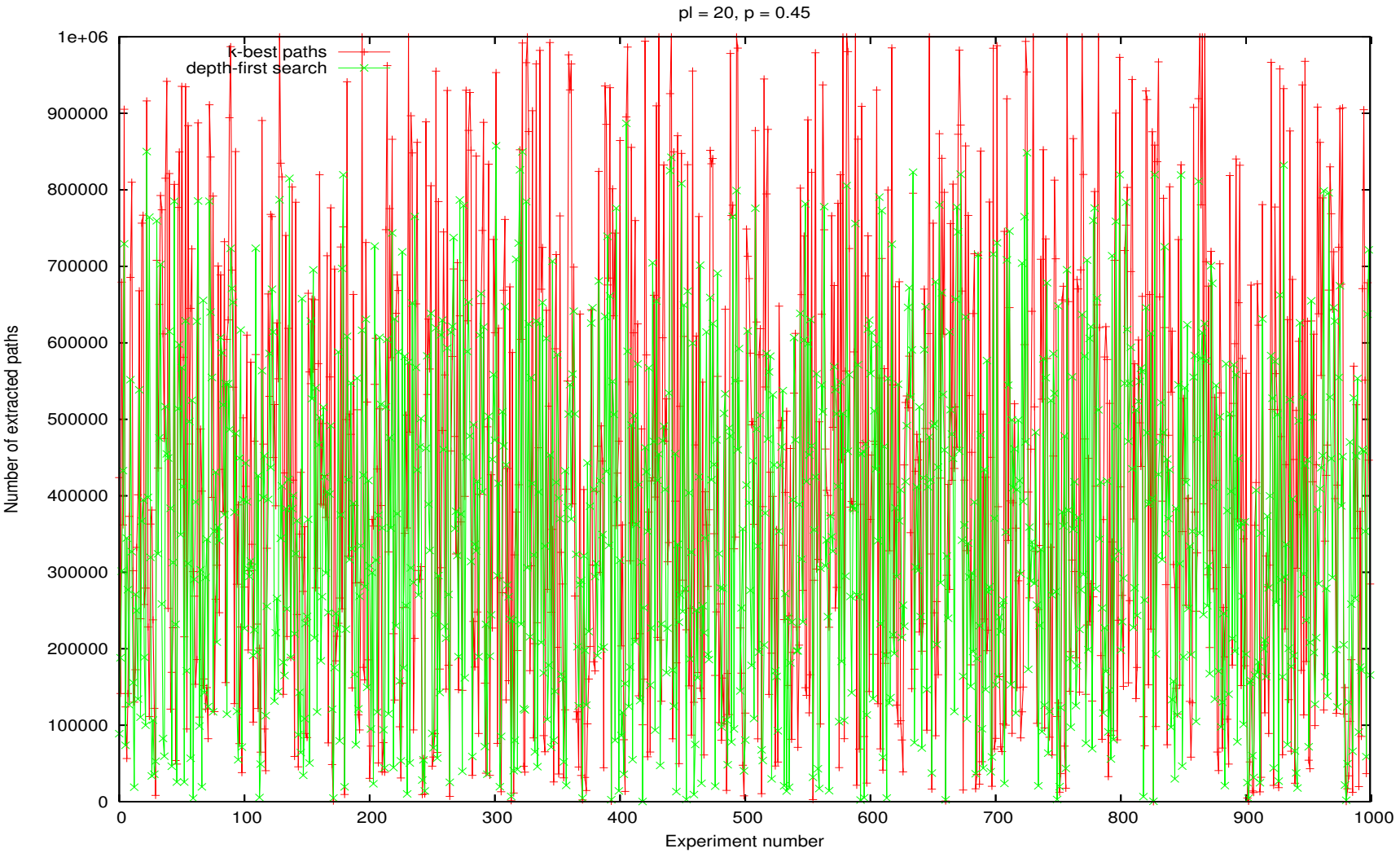

pl = 20, p = 0.20
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
0
100
200
300
400
500
600
700
800
900
1000


pl = 20, p = 0.25
k-best paths
depth-first search
Number of extracted paths
Experiment number
0
100000
200000
300000
400000
500000
600000
700000
800000
900000
0
100
200
300
400
500
600
700
800
900
1000


pl = 20, p = 0.30
k-best paths
depth-first search
Number of extracted paths
Experiment number
1e+06
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900
1000


pl = 20, p = 0.35
k-best paths
depth-first search
Number of extracted paths
Experiment number
1e+06
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900
1000


pl = 20, p = 0.40
k-best paths
depth-first search
Number of extracted paths
Experiment number
1e+06
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900
1000


pl = 20, p = 0.45
k-best paths
depth-first search
Number of extracted paths
Experiment number
1e+06
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
100
200
300
400
500
600
700
800
900
1000