Robust Dictionary Attack of Short Simple Substitution Ciphers

Edwin Olson
MIT Computer Science and Artificial Intelligence Laboratory
Computer Engineering
Cambridge, MA 02140
Email: eolson@mit.edu

***Abstract*— Simple substitution ciphers are a class of puzzles often found in newspapers, in which each plaintext letter is mapped to a fixed ciphertext letter and spaces are preserved. In this paper, we describe a system for automatically solving them, even when the ciphertext is too short for statistical analysis, and when the puzzle contains non-dictionary words. Our approach is a based around a dictionary attack; we describe several important performance optimizations, as well as effective techniques for dealing with non-dictionary words. We present quantitative performance results for several variations of our approach as well as two other implementations.**

I. Introduction

Approaches to automatically solving simple substitution ciphers fall into two basic groups. The first family relies on statistical analysis of ciphertext frequencies (including digrams and N-grams) and permutes the mapping from ciphertext to plaintext such that the plaintext is highly probable. This family includes methods based on relaxation [1] and genetic algorithms [2]; they are typically iterative in nature. These approaches rely on having relatively long ciphertext passages (hundreds of letters) in order for their measured statistics to be meaningful. Notably, some methods can recover the plaintext even when spaces are permuted along with the other 26 letters [3].

The second family of approaches is based on dictionary attacks [4], [5], [6]. These approaches can be exceptionally fast and can operate on very short ciphertexts where the statistics are virtually meaningless. This performance comes at a cost: they typically require that spaces be preserved in the ciphertext and require a large dictionary that contains most (if not all) of the plaintext. Unfortunately, large dictionaries tend to produce spurious (non-sensical) solutions. The method by Hart [5] differs from other methods in this family, in that it purposefully uses a very small dictionary (135 words) and explicitly considers hypotheses that ciphertext words are not found in the dictionary.

In this paper we present our approach, a dictionary-based attack with several refinements:

- At each level of the search tree, we perform a set-intersection based optimization to reduce the search space of child nodes.
- We allow the substitution of a single letter at a node in the search tree, rather than requiring a whole word to be substituted.
- We dynamically select which ciphertext word or ciphertext letter to expand at each node. Our selection heuristic was selected after comparing several static and dynamic planning strategies.
- We employ three separate mechanisms for dealing with non-dictionary words; the resulting algorithm can robustly handle ciphertexts with multiple non-dictionary words, albeit with additional search costs.
- We compute posterior probabilities of candidate solutions using *a priori* trigram probabilities; this allows multiple solutions to be meaningfully ranked. This allows us to use larger dictionaries: even if many solutions exist (many non-sensical), the most likely solutions are automatically identified and brought to the user's attention.

Our approach is effective and fast on virtually all cryptograms, including those with very short ciphertexts (under 40 letters), even when non-dictionary words are present (including proper names, and word-play). In addition, our software is available under an open source license.

II. Approach Overview

Our algorithm begins by parsing the ciphertext into a number of ciphertext words. There are a number of challenges even here. Hyphenated words, for example, could be treated at least three different ways: the dictionary could contain a list of acceptable hyphenated words (including the hyphen), hyphens could be stripped (producing compound words), or hyphens could be replaced with spaces (producing two separate words). Casual English is wildly inconsistent with regard to hyphenated words, and the first two methods are likely to produce non-dictionary words: we cannot reasonably compile a list of all words that could be hyphenated. We opt for the conservative approach of splitting hyphenated words into separate words since it is more likely to result in dictionary words.

Apostrophes present a similar dilemma: while there are relatively few contractions, there are countless possessives. For simplicity, we strip apostrophes, with the result that possessive constructions masquerade as plurals.

For each ciphertext word, we look-up a list of candidate words with identical letter patterns. For example, both "HELLO" and "SILLY" are suitable replacements for the ciphertext "RXQQL", all of which have the canonical letter

pattern "ABCCD". We accelerate the dictionary look-up by storing our dictionary on disk sorted by letter pattern, rather than alphabetically.

Like most dictionary attacks, our algorithm is fundamentally a depth-first search (see Alg. 1 and Alg. 2). At every node in the search tree, a ciphertext word is selected and its various plaintext alternatives are substituted. Each substitution creates a child node in which more of the plaintext is known than at the parent node. The state of the solver is a *map*, which records for each ciphertext letter the set of possible plaintext letters. Initially, each ciphertext letter could be any plaintext letter, i.e., has the set $\{A, B, C, ..., Z\}$.

When a node performs a substitution, it must make sure that the substitution is consistent with the substitutions made by its ancestor nodes:

- No ciphertext letter may be mapped to more than one plaintext letter. For example, (X=A, X=B) is forbidden.
- A plaintext letter can only be mapped to once. E.g., (X=A, Y=A) is forbidden.

We extend the classical dictionary attack by allowing a node in the tree to substitute a single letter, rather than a whole word. This is an important capability that improves performance, as we will show in Section VIII.

The particular choice of which ciphertext word (or letter) to substitute at each node has a profound impact on the search complexity. Some words have more candidate substitutions than others and will yield more child search nodes. Conversely, words that are longer and contain more frequently occurring letters will, once substituted, more greatly constrain future child nodes, resulting in fewer search nodes. The choice of which ciphertext word or letter to substitute at each node is the responsibility of a *planner*, which we will discuss in detail in Section V.

A significant optimization can performed on each node, reducing the set of all candidate lists significantly. This optimization is described in Section III. Its effect is to eliminate candidate words for ciphertext words that have not yet been substituted.

When the entire plaintext is known, we report a solution, but we do not terminate the search. Many ambiguous solutions are possible, and our algorithm enumerates them all. Additionally, whenever we reach a "dead-end" in the search tree (i.e., a node that has no children), we report a *partial solution*; these are useful when non-dictionary words are present, as described in Section VI.

Algorithm 1 Solve(Map)

```
1: for all X in {A, B, C, ..., Z} do
2:     if (UserProvidedClue(X) then
3:         Map(X) = { UserClue(X) }
4:     else
5:         Map(X) = {A, B, C, ..., Z}
6: SolveRecursive(Map)
```

Algorithm 2 SolveRecursive(Map)

```
1: if AllCipherTextKnown() then
2:     ReportFullSolution(Map)
3:     return
4: C = PlannerSelectUnknownLetterOrWord()
5: has_child = false
6: Map = SelfIntersection(Map)
7: for all P in Candidates(C) do
8:     if (IsConsistent(Map, C, P)) then
9:         NewMap = AddMappings(Map, C, P)
10:        SolveRecursive(NewMap)
11:        has_child = true
12: if (has_child = false) then
13:    ReportPartialSolution(Map)
14: return
```

III. Set-intersection candidate pruning

The set-intersection candidate pruning method considers constraints resulting from the simultaneous consideration of multiple candidate lists. It is a purely logic-inferential algorithm with no searching. It is an effective method: it can often solve cryptograms by itself, without any search.

Let us begin with an example. Suppose we have two ciphertext words and their candidate lists:

MCDMRCNSFX	MSCNPPRX
deadweight	aflutter
disdainful	bedrooms
gregarious	gorillas
perplexity	proceeds
	typhoons

Note that the candidate list for MCDMRCNSFX requires that $M \in \{d, g, p\}$, while the candidate list for MSCNPPRX requires that $M \in \{a, b, g, p, t\}$. Clearly, only $M \in \{g, p\}$ satisfies both constraints. We can thus eliminate all of the candidates where this joint constraint is not satisfied, yielding:

MCDMRCNSFX	MSCNPPRX
gregarious	gorillas
perplexity	proceeds

Now note that the candidate list for MCDMRCNSFX requires that $C \in \{r, e\}$ while the candidate list for MSCNPPRX requires that $C \in \{o, r\}$. Satisfying both constraints requires that we assign $C = r$, yielding a unique decryption: "gregarious gorillas".

We now describe the algorithm in general, as listed in Alg. 3. We begin by selecting one ciphertext word and its corresponding list of plaintext candidates. Any candidates that are not consistent with the puzzle's current *map* are discarded. Assuming that the word's plaintext is one of the remaining candidate words, we compute the possible set of ciphertext-to-plaintext letter mappings. After considering all of the candidate words, we then compute the intersection of this set with the puzzle's current *map*. This intersection becomes the puzzle's new *map*.

We iterate over the ciphertext words, performing an intersection operation for each, until the puzzle's *map* has reached steady-state, i.e., until no more reductions are possible.

Algorithm 3 SelfIntersection(Map)

```
1: {Initialize Map to full sets}
2: repeat
3:    for all C in CiphertextWords do
4:       {Initialize NewMap to empty sets}
5:       for all X in {A, B, C, ..., Z} do
6:          NewMap(X) = {}
7:       for all P in Candidates(C) do
8:          if (IsConsistent(Map, C, P)) then
9:             NewMap =AddMappings(NewMap, C, P)
10:      Map = Intersect(Map, NewMap)
11: until no reductions performed
12: return Map
```

$SetIntersection$ is run at every node in the tree (including the root): it typically results in significant reductions to the size of the candidate lists, which reduces the branch factor of the depth-first search. Since run-time cost is exponential in the branching factor, this results in significant speed-ups, as quantified in Section VIII.

We note that this algorithm does not detect all possible inferences. For example, suppose that three separate candidate lists require that $X \in \{a, b\}$, $Y \in \{a, b\}$, and $Z \in \{a, b, c\}$. It can be deduced that $Z = c$, but this would not be detected by the algorithm described above. Our efforts to exploit these inferences have indicated that the search costs to identify them generally exceeds the benefit.

IV. Maintaining Candidate Lists

As we have described the algorithm thus far, we build a list of plaintext candidates for each ciphertext word and make extensive use of the function *IsConsistent* to skip over candidates that are not consistent with the puzzle's current *map*. If implemented in this manner, a great deal of CPU time would be spent in *IsConsistent* eliminating the same candidates over and over again.

When a word is eliminated at a node in the search tree, it can be eliminated from all of its children as well. Conceptually, when a candidate is ruled out, it is moved into a "disabled" list; only "enabled" words are searched at children nodes. Naturally, when a node has finished iterating over its candidates (and is about to return to the parent node), all of the candidates it disabled must be re-enabled. It is important that both of these operations–disabling individual words and re-enabling all the words that were disabled at one node in the tree– be very fast.

Our solution is to store each ciphertext word's candidate list in an array divided into two parts: the first part contains disabled words, and the second part contains enabled words (see Fig. 1). An index, $firstcand$, maintains the index of the first enabled word. This is initialized to zero at the root node, meaning that no candidates are disabled.

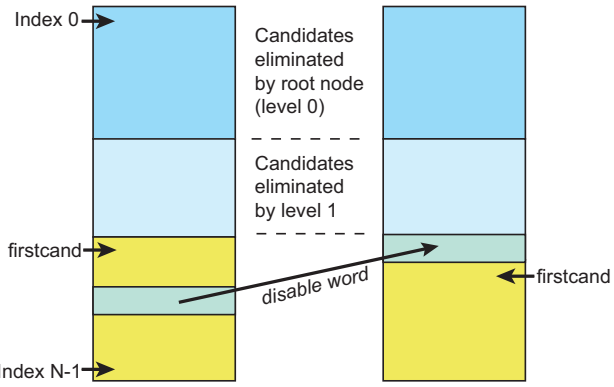

Fig. 1. Candidate list data structure. Each ciphertext word has an array of candidate words. As successive nodes in the search eliminate candidates, the eliminated candidates are shuffled to the end of the list of eliminated words and $firstcand$ is incremented. A node can re-enable the words it disabled by simply changing the pointer $firstcand$.

At each node in the search tree, we push the value of $firstcand$, onto a stack. Disabling a word is an $O(1)$ operation: the candidate to be disabled is swapped with the candidate at $firstcand$, and $firstcand$ is incremented. Re-enabling all of the words is done by popping the value $firstcand$ off the stack; no shuffling of words is required, resulting in an $O(1)$ operation.

Note that this scheme does not preserve the order of the candidates, but the order is not important for correctness.

(In fact, the candidate lists are presorted according to the frequency of the occurrence of each candidate. This does not change the total run-time of the algorithm, but it does cause the search to explore the most likely solutions first. In other words, the most likely solution tend to be found earlier in the search. The permutation of the candidate order due to the enabling/disabling logic described above might seem to defeat this optimization. However, the root node is totally unaffected by the permutation, and the root node's search order has the greatest effect by far.)

V. Planning

At each node in the search tree, the planner selects an unknown ciphertext letter or word for substitution. The choice is critical for good performance, since each option involves a different amount of work for a different amount of plaintext recovery. Planners fall into one of two basic categories:

- Static: The ciphertext words are sorted according to some metric when the puzzle is initialized, and the ith word is always selected at depth i in the search tree.
- Dynamic: At every level of the search tree, the ciphertext word or letter to substitute is chosen, independently of what other nodes at the same level of the tree have done. Some metric is computed for each ciphertext word, which could include the amount of plaintext that would be recovered if that ciphertext word was selected, and/or the number of candidate words enabled for that ciphertext word. The best ciphertext word or letter is then selected.

Static planners are attractive since they do not require computation at each level of the search tree. However, they are unable to take exploit opportunities that arise in the middle of a search: the size of candidate lists is sensitive to the particular *plaintext* substituted made at earlier nodes in the tree, not just which ciphertext word was substituted. For example, substituting "RXQQL=hello" may greatly reduce the candidate list for one ciphertext word while having little effect on a second; substituting "RXQQL=silly" could have the opposite effect. Dynamic planners, capable of exploiting these situations, almost invariably outperform static planners. We consider only dynamic planners for the remainder of this paper.

In either static or dynamic planners, there are a number of plausible cost metrics that could be used to determine the best word or letter to substitute. Suppose, for example, that a particular ciphertext word C has N plaintext candidates. If substituting C gives us L mappings (i.e., we learn the correct translation of L ciphertext letters), we could define the cost as:

$$cost(N, L) = N^{1/L} \tag{1}$$

This cost metric, which we call the *effective branching factor (EBF)* metric, computes the effective branching factor B that would occur in the hypothetical situation that only one mapping is discovered per search node. Note that the total work of this hypothetical situation is $N = B^L$, which matches the actual amount of work N at this node. It is an appealing metric because it captures the fact that the cost of a depth-first search is the product of the branch factors at every depth of the tree. It provides a principled way of computing a one-dimensional ranking from two different statistics of a ciphertext word: the number of candidates and the amount of plaintext that would be recovered.

A simpler metric, the *linear* metric, is the number of candidates per amount of plaintext recovered:

$$cost(N, L) = N/L \tag{2}$$

Eqn. 2 tends to more heavily penalize words with many candidates and, to our initial surprise, generally out-performs the previous metric. We attribute this to the effects of the $SetIntersection$ algorithm: at each node in the tree, the number of candidates is reduced substantially. It is thus advantageous to make a substitution as quickly as possible (i.e., pick a word with few candidates), and let the $SetIntersection$ optimization reduce the branching factor of child nodes. In other words, a ciphertext word with many candidates is bad, even if the effective branching factor is small, because *all* of those candidates must be considered without the benefits of $SetIntersection$ along the way.

Is it advantageous to run $SetIntersection$ at the earliest opportunity, without regard to how much plaintext would be recovered? The *lazy* cost function does exactly that:

$$cost(N, L) = N \tag{3}$$

This cost function is, in aggregate, the best performing of the cost functions we evaluated. Like the *linear* metric, it appears to be successful due to the effects of the $SetIntersection$ algorithm. Performing a substitution– even a single letter– reduces the candidate lists so dramatically that it is worth doing it, even though the amount of plaintext recovered at that node is rather small.

While the *effective branching factor* is principled, it is out-performed by heuristics that attempt to model the benefits of the $SetIntersection$ algorithm. However, the $SetIntersection$ algorithm's performance varies from puzzle to puzzle. When $SetIntersection$ does not reduce the candidate lists substantially, the other methods can out-perform it. It is possible that a more complicated metric could be devised that would better estimate the efficacy of $SetIntersection$; this would lead to an even faster algorithm.

VI. Non-Dictionary Words

Non-dictionary words generally cause dictionary attacks to fail since substituting the incorrect plaintext word usually leads to a contradiction.

Our algorithm includes three strategies for dealing with non-dictionary words. The first strategy is to report partial solutions at any terminal node in the search tree. By the time a contradiction is encountered, many letter mappings have been made; often, many of them are *correct*. Thus, we report a partial solution; a human user can often trivially determine the correct answer from such a partial solution.

The second strategy for dealing with partial solutions is to ignore a subset of the ciphertext. If we are unable to solve the puzzle with all words enabled, we try solving the puzzle with one word disabled at a time. If we are still unsuccessful, we try solving the puzzle with all combinations of two words disabled at a time, then three. This strategy is sufficient to solve all puzzles with up to three non-dictionary words. Note that the $SelfIntersection$ algorithm operates as usual, though it *also* ignores the disabled words.

Puzzles with more than three non-dictionary words pose an additional challenge. At this point, trying to blindly guess which words are non-dictionary words becomes unreasonable. Instead, we do two things:

- We disable the $SelfIntersection$ algorithm. The $SelfIntersection$ algorithm considers all the ciphertext words at every node; if a non-dictionary word is considered, candidate lists for all other words are immediately affected, even at the root node. The result is that candidates for in-dictionary words can be erroneously pruned out, leading to failure.
- We switch to a Random planner. Any time that the planner picks a ciphertext word whose plaintext *is* in the dictionary, a significant amount of plaintext is recovered. If the planner picks a few dictionary words, a human can usually discern the entire correct plaintext.

The quick brown fox jumped over the lazy dogs.
The quick frown box jumped over the lazy dogs.
The jacky frown box palmed over the quiz dogs.
The quick franz wax jumped aver the glob days.
Bye lamps grown fox jacked over bye quiz doth.
She fatly grown pox jambed over she quiz dock.

TABLE I

A SUBSET OF THE 1750 POSSIBLE SOLUTIONS TO "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOGS."

The combination of disabling the $SelfIntersection$ algorithm and the Random planner ensure that the entire search space will eventually be searched. The success of this strategy is dependent upon the Random planner selecting ciphertext words that *are* in the dictionary. (In contrast, note that our second strategy relies on guessing the set of words that are *not* in the dictionary.) In practice, the Random planner must only identify a few dictionary words correctly before an adequate partial solution is found. are In addition, we provide a method for users to indicate those words that are likely to be non-dictionary words. Many newspaper cryptograms consist of famous quotes, with the authors attributed at the end; it is easy to pick out the proper names in this case.

VII. RANKING SOLUTIONS

Particularly with short cryptograms, many puzzles have multiple solutions. If the puzzle contains non-dictionary words, the problem is compounded since our mechanisms for dealing with non-dictionary words involve discarding constraints that would otherwise eliminate many possible solutions.

For example, the puzzle "The quick brown fox jumped over the lazy dogs" has 1750 different solutions (see Table I), even when using our smallest library with 110,916 words. (Keep in mind that the dictionary contains common abbreviations, foreign words, and proper names.) With 1750 possibilities, some computer assistance is needed to bring the most likely candidates to the operator's attention.

We do not attempt to enforce rules of grammar; instead, we compute the posterior probability of each solution using a table of trigram probabilities. Our probability tables include the occurrence of spaces, but not punctuation. We found that trigram tables produced a noticeable improvement in solution ranking over digrams.

Recall that our algorithm reports partial solutions, in addition to full solutions. In full solutions, all ciphertext letters are mapped to plaintext letters; in partial solutions, some letters are not mapped. This presents a small difficulty in producing a posterior probability, since we do not know the entire outcome. We opt to treat unknown letters by selecting the trigram entry that matches and has the lowest probability. This creates a desirable bias towards more-complete solutions over less-complete ones.

Our trigram ranking system is very effective; the intended answer usually appears within the first handful of solutions.

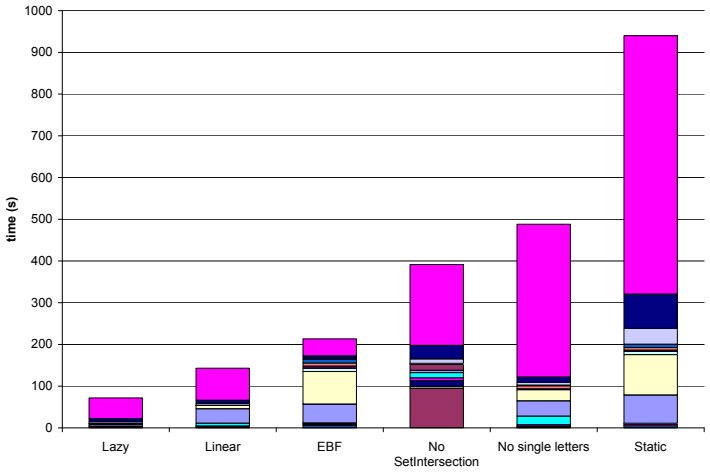

Fig. 2. Performance results. In each experiment, only a single parameter is altered; otherwise, the lazy planner with all optimizations is used. The *lazy* planner is typically faster than the *EBF*, *linear*, or *static* planners. Set-intersection and the use of single-letters show dramatic performance improvements.

In the case of "the quick brown fox...", the desired solution was ranked first, out of 1750.

In our implementation, we compute chi^2 values rather than probabilities. The best solutions are kept in a Min-Heap [7] with a fixed maximum capacity (nominally 500). When the solution set grows too large, we can quickly extract the worst solution using the standard heap-remove method.

VIII. RESULTS

We have performed quantitative measurements of the performance impacts of our improvements over naive-dictionary based attacks (see Fig. 2). We compare three different dynamic planners, a static planner, the impact of the set-intersection optimization, and the impact of allowing a planner to select a single-letter substitution (rather than requiring a node to substitute an entire ciphertext word). In each case, we have modified only a single experimental variable: when not otherwise specified, the lazy planner is used with all optimizations.

Each column in the plot represents the total run time for the algorithm on a set of 21 cryptograms; colors within a column indicate the run time on a particular puzzle. We have opted for a cumulative time metric, rather than arithmetic or geometric mean, since it best models the experience of a user trying a variety of puzzles. Note that many puzzles are easily solved (in milliseconds) by all implementations, and their contribution to the total run time is not readily discerned from the figure. In all cases, either the correct plaintext or a readily interpretable plaintext (with only minor errors due to non-dictionary words) was recovered.

The performance of the "lazy" planner with our other improvements is significantly better than other permutations. The choice of dynamic planner (lazy, EBF, or linear) impacts performance within about a factor of three. The impact of the set intersection algorithm is even more substantial: it reduces run time by a factor of 5.45. The ability to substitute a single letter improves performance by a factor of 6.79. Dynamic

Algorithm	Puzzles solved	Time (solved puzzles)	Our time
Hart	24%	0.143 s	1.045 s
Dunn	38%	18.664 s	0.567 s

Fig. 3. Comparison to other algorithms. Cumulative run times for puzzles successfully solved are given, versus the cumulative run time for our approach on the same set of puzzles. Hart and Dunn's methods solved only a fraction of the puzzles in our benchmark set. Hart's method is typically very fast, and in small number of cases that it is successful, is noticeably faster than our approach. Dunn's method is both less successful and slower than ours. Note that our method successfully solved all the puzzles in the benchmark set.

planning is the biggest win of all, improving performance by a factor of over 13. We do not claim that all of these improvements are cumulative, but the point remains that dramatic speedups have been obtained.

We also compared our algorithm's performance to two other implementations whose source code was readily available: Hart's method [5] which is a dictionary attack with an intentionally minimal dictionary, and a more conventional dictionary attack by K. Dunn [6] using a 133k word dictionary and employing a simple static planner. Neither method does well on our benchmark set (see Fig. 3): they produce gibberish on the majority of the test cases. Hart's algorithm fails on 76% of our tests, though when it does work, it is generally faster than our algorithm. Dunn's algorithm fails on 62%, and is typically slower than our algorithm.

We note that the average difficulty of our benchmark puzzles is quite high: many puzzles contain non-dictionary words, and some of the puzzles have only very common letter patterns.

IX. Availability

Our software, including source code, is available under the GNU Public License (GPL). The project's website is at http://www.blisstonia.com/software/Decrypto. An on-line version is available at http://www.blisstonia.com/software/WebDecrypto.

X. Conclusion

We have presented a robust dictionary-based attack on simple substitution ciphers. Our approach employs dynamic planning, set-intersection optimizations, and several mechanisms for dealing with non-dictionary words. Multiple solutions are ranked according to their posterior probability using a table of trigram probabilities. We also employ an effective yet simple data structure that allows candidate lists to be efficiently maintained.

Our algorithm can be applied to non-English languages with appropriate modification of the parser (the handling of diacritical marks is more complex in other languages), and an appropriate dictionary. The solver itself makes no assumptions about the number of letters in the language, for example.

In the future, we are considering more elaborate statistical analysis of proposed plaintext solutions. These improvements could not only improve the posterior rankings of candidates, but could direct the search towards more probable candidates in the first place. On the other hand, our existing algorithm only struggles on short puzzles, and short puzzles often contain statistically unusual puzzles; the added complexity may not be worth the effort. This is an area for future experimentation.

Acknowledgements

We are indebted to Pete Wiedman, an avid solver and creator of cryptograms, for his extensive testing of our software. His reports of performance anomalies led to some of the improvements described in this paper. He also contributed extensively to the dictionary.

References

[1] S. Peleg and A. Rosenfeld, "Breaking substitution ciphers using a relaxation algorithm," *Commun. ACM*, vol. 22, no. 11, pp. 598–605, 1979.

[2] R. Spillman, M. Janssen, B. Nelson, and M. Kepner, "Use of a genetic algorithm in the cryptanalysis of simple substitution ciphers," *Cryptologia*, vol. XVII, no. 1, pp. 31–44, 1993.

[3] T. Jakobsen, "A fast method for cryptanalysis of substitution ciphers," *Cryptologia*, vol. 19, no. 3, pp. 265–274, 1995. [Online]. Available: citeseer.ist.psu.edu/jakobsen95fast.html

[4] M. Lucks, "A constraint satisfaction algorithm for the automated decryption of simple substitution ciphers," in *CRYPTO*, 1988, pp. 132–144. [Online]. Available: citeseer.ist.psu.edu/lucks88constraint.html

[5] G. W. Hart, "To decode short cryptograms," *Commun. ACM*, vol. 37, no. 9, pp. 102–108, 1994.

[6] K. Dunn, "Ciphergram solution assistant," 1997. [Online]. Available: "http://www.gtoal.com/wordgames/kdunn/CSA_unix/csa.htm"

[7] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, 1st ed., ser. The MIT Electrical Engineering and Computer Science Series. The MIT Press, 1990.