Optimal Trie Compaction
is NP-Complete

Jens M. Dill
87-814

March 1987

Department of Computer Science
Cornell University
Ithaca, New York 14853-7501
Optimal Trie Compaction is NP-Complete

Jens M. Dill
Department of Computer Science
Cornell University
Ithaca, New York 14853

1. Introduction
A serious disadvantage of the trie as a data structure for information retrieval is that it wastes too much space. In a typical trie, a high percentage of memory cells contain nil values that represent the absence of data in a particular segment of the trie. One method of trie compaction regains space by overlapping the nodes of the trie in memory so that the space used for nil values in one node may hold a non-nil value from some other node of the trie. Al-Suwaiyel and Horowitz [AH84] have analyzed several algorithms for optimal and suboptimal trie compaction by this method and have conjectured that the optimal trie compaction problem is NP-complete.

The following proof of the NP-completeness of trie compaction was inspired by an investigation of a class of tries that are characterized by such a high percentage of nil values in each node that some workable compaction scheme is essential. Section 2 defines the trie compaction problem. Section 3 describes the proof in terms of a related problem, and section 4 fills in the details of the central construction. Section 5 discusses how the proof can be adapted to the special requirements of trie compaction.

2. Definitions
A trie [DIB59, Fre60] is a data structure that is designed to allow efficient retrieval of the information stored in it. Information is stored in units called records, each of which is identified by a unique key value. For the purpose of accessing a record in the trie, the key value is represented as a finite sequence of symbols drawn from a fixed alphabet, \( \Sigma \). The trie is structured as an \( m \)-ary tree, where \( m \) is the number of symbols in \( \Sigma \). Given a key value, the corresponding record is retrieved by traversing the tree from the root, using successive symbols from the key value to select the path from one node to the next.

A trie node, then, is represented as a collection of \( m \) cells, arranged so that they can be indexed by a symbol of \( \Sigma \) in constant time. Each cell can hold one of three kinds of values: an interior pointer to a subordinate node of the trie, a leaf pointer to a stored record,\(^*\) or a nil value indicating that the cell is not currently in use. There is a fairly subtle ambiguity implicit in this structure: if one key value happens to be a prefix of another key value, the cell indexed by the last character of the shorter key

\(^*\) or the record itself if the record is small enough to fit in a cell.
will need to store both a leaf pointer (for the shorter key) and an interior pointer (for the longer one). Most implementations deal with this problem by augmenting the key alphabet $\Sigma$ with one or more final symbols and declaring that the last symbol in a key value (and only the last) must be a final symbol.

A trie in which leaf pointers may be stored only in cells indexed by a final symbol is called a full trie; the alternative (in which the leaf pointer is stored in the cell indexed by the last character of the shortest prefix that is sufficient to distinguish the given key from all others in the trie) is called an abbreviated trie. Abbreviated tries are the general rule, because the space saved by the abbreviation process is usually substantial. The distinction is nevertheless important, since the existence of a reasonable trie compaction method would make full tries more appealing in some cases.

The trie compaction problem is to find a way of mapping the nodes of the trie into storage so that cells containing the nil value are not stored.* The trick is to do so without sacrificing the constant-time indexing properties of each node. Usually this means that the position of a cell within a node cannot be changed; the only permissible operation is to overlap two or more nodes so that the non-nil cells of each node are stored in locations that would hold nil-valued cells in all of the other nodes. We will focus on this particular aspect of the problem, since other compaction schemes would have to rely on more esoteric properties of a particular node-indexing scheme.

Formally speaking, trie compaction may be stated as an optimization problem: Given a trie consisting of $k$ interior nodes, each of which consists of a contiguous collection of $m$ cells, and a memory large enough to hold $M$ cells, can the nodes be mapped to contiguous regions of the memory in such a way that the relative positions of cells within the nodes are not altered and no two non-nil cells are assigned to the same memory location?

3. Proof Outline

The first step of the proof is to recognize that the structure of the trie and the contents of the cells are not essential to the problem. By abstracting away from these details, we get the more general problem of binary string collision avoidance (BSCA): Given a set of $k$ strings of binary digits, and an $M$-bit memory, can the strings be mapped into possibly overlapping regions of memory such that at most a single 1 is stored at each location in the memory?

Any instance of a trie compaction problem can easily be reduced to an instance of BSCA by the straightforward substitution of strings for nodes, 0s for nil-valued cells, and 1s for cells containing pointer values. The inverse reduction is harder to define, since the mapping from trie compaction to BSCA is neither one-to-one nor onto.† We will therefore concentrate first on showing that BSCA is

* By "not stored," we mean that no space is reserved for the nil-valued cells, and that the number of unused cells sprinkled through the region used to store the tree is kept to an acceptable minimum. To simply not assign storage to the nil-valued cells and not use the unassigned storage for other purposes would be an unacceptably trivial solution.

† We will explain this in more detail in section 5.
Optimal Trie Compaction is NP-Complete

NP-complete; and then on showing that the proof is still valid when restricted to those instances of BSCA that correspond to trie compaction problems.

Showing that BSCA is in NP is straightforward. Given an instance of BSCA and a proposed solution (in the form of a mapping from strings to memory locations), all we have to do is check all of the 1s in all of the strings to make sure that no two are assigned to the same memory location. This can easily be done in polynomial time.

We will show that BSCA is NP-complete by reduction from 3-CNF satisfiability (3SAT). An instance of 3SAT is a boolean expression in the form of a conjunction of clauses, each of which is a disjunction of three terms. Each term is either a variable or the negation of a variable. The problem is to find an assignment of truth values to the variables so that at least one term in each clause is true. We will let \( w \) denote the number of clauses and \( v \) denote the number of variables. Note that \( v \leq 3w \), so the size of an instance of 3SAT is \( O(w) \).

From a given instance of 3SAT, we will construct an instance of BSCA that consists of \( k = v + 7w + 1 \) strings, classified as follows:

- \( v \) truth-assignment strings, each of which can be placed in one of two places in memory to represent the assignment of a true or false value to one of the variables.

- \( 7w \) clause-satisfiability strings, each of which can be placed in one of seven positions in memory to select the pattern of terms that is expected to be true in a given clause under a satisfactory assignment.

- one substrate string, which acts to restrict the placement of the other strings.

We set the size of the memory equal to the length of the substrate string. This is what makes it possible for the substrate to control the placement of the other strings: once the substrate is placed, the other strings must match up with the gaps in the substrate.

There are seven ways in which truth values can be assigned to three terms in order to insure that at least one of the terms is true. For each clause, we construct a separate string to represent each of these possibilities. The substrate is arranged so that it contains \( 7w \) slots arranged in seven groups so that each group can accept one string associated with each clause. The first group of slots is distinguished from the others because it also has extra blank spaces to match the truth-assignment strings. A solution to 3SAT may be guessed, therefore, by placing one string for each clause in the first group of slots and then trying to fit all \( v \) variable-assignment strings into the remaining space. The assignment of the remaining \( 6w \) strings to the remaining slots has no effect on the solution. The extra \( 6w \) slots, therefore, simply serve to dispose of the unneeded clause-satisfiability strings without creating new opportunities for placement of the critical strings.

The variable-assignment strings operate on the following principle. Given a BSCA subproblem with a memory of size \( 3v \) and with \( v \) strings of the form \( 10^{v-1}1 \), any solution must place one of the 1s from each of the \( v \) strings into the middle third of the memory. Since the middle third has precisely \( v \)
cells available, we can use the pigeonhole principle to show that each of the cells in the middle third is associated with a distinct string, namely the one that placed the 1 in it. We associate each 3SAT variable with one of the cells of the middle third, and assign it a value of true if the other end of the associated string falls in the right-hand third of memory, and false if the other end falls in the left-hand third. It is easy to show that any solution to this BSCA subproblem represents a truth assignment to a set of \( v \) boolean variables, and any such truth assignment is representable as a solution to the BSCA subproblem.

Each clause-satisfiability string contains a section that is intended to serve as the memory for an instance of the subproblem mentioned above. This section is a string of length \( 3v \) that contains mostly 0s. For each variable that appears in the clause, a 1 appears in the appropriate place in either the left or the right third of the section, so as to force that particular variable to a particular truth value. Since there is a distinct string for each of the seven possible variable assignments that could satisfy that clause, we can always choose the one that works, if any. The clause-satisfaction strings do not place any 1s in the middle third of this section. Instead, the substrate is set up to fill the middle third with 1s in the last six of the seven groups of clause-satisfaction slots, forcing the variable-assignment strings to fall into the first group. The variable assignments are made consistent across all \( w \) clauses in the first group of slots by padding the strings of the subproblem with enough 0s to make them as long as a single slot, and then repeating the pattern \( w \) times.

Let us, for the moment, assume that the set of strings described above can be constructed in polynomial time from a given instance of 3SAT. All we need to do is show that a solution to the 3SAT problem exists if and only if the corresponding BSCA problem has a solution. Assume we have a solution to the 3SAT problem. Then we can solve the BSCA problem as follows. Place the substrate string into memory starting at the first location. Since the substrate fills all of memory, the other strings have to be placed into the appropriate slots in the substrate. For each clause, choose the clause satisfaction string that matches the variable assignment that solves the 3SAT problem. Place that string into the first slot for that clause within the substrate. Place each of the other six clause-satisfaction strings into any of the other six slots for that clause in the substrate. Repeat until all clause-satisfaction strings have been placed. Now, for each variable, choose one of the variable-assignment strings and place it so that it overlaps the first group of clause-satisfaction slots in such a way that within each clause, one 1 fills the middle-third cell assigned to that variable, and the other 1 is either in the left or the right third, whichever position does not conflict with the assignment prescribed by the clause-satisfaction string. Since all clause-satisfaction strings in the first group of slots agree on the same variable assignments, each variable-assignment string can be placed consistently. We have thus exhibited a solution to the BSCA problem.

Going the other way, from an arbitrary solution of the BSCA problem to a solution of the 3SAT problem, is just a matter of reading the results from the placement of the variable-assignment
strings. We are assured from the construction of the following facts. For each clause, there are seven satisfiability strings which cannot be placed anywhere except within the corresponding seven slots of the substrate. Once all of the satisfiability strings have been placed, there are no remaining slots to be filled. The variable assignment strings may be placed only within the first group of clausesatisfiability slots, and there are exactly $v$ places for $v$ variable assignments. The first group of slots must contain one satisfiability string for each clause, and the clause-satisfiability strings must all agree in their requirements for each variable, or else the corresponding variable-assignment string will have no place to go. Thus, any solution to the BSCA problem must be in the form of an image of a solution of the 3SAT problem.

The only remaining part of the proof is to exhibit the construction of an instance of BSCA that has all of the properties required above and to show that the construction can be done in polynomial time. This we do in the next section.

4. Details of the Proof

It will be convenient to construct the strings of the BSCA problem out of units of uniform size. The strategy we described above for variable assignment suggests that a length of $v$ bits is sufficient for the units in the variable-assignment strings. For controlling the placement of clause-satisfiability strings, however, we have to distinguish among $w$ clauses, so it would be convenient to have at least $w$ bits in each unit. Since we are not constrained by constant factors, we can take advantage of the fact that $v \leq 3w$, and work with a unit size of $3w$ bits.

Now we are ready to specify the units out of which the strings will be constructed. Each unit will be named by a boldface letter, subscripted if necessary. We start with the simplest units:

$$Z = 0^{3w}$$

$$B = 1^{3w}$$

$B$s will be used in the substrate string to provide long blocks of $1$s that help force the other strings into proper alignment. $Z$s are used in the clause-satisfiability strings to fit over the $B$s in the substrate, and in the variable-assignment strings to pad them to the length of the clause-satisfiability strings.

The significant portion of the variable-assignment string is a pair of consecutive units of the form

$$V = 10^{3w-1}.$$ 

The idea is to take $v$ instances of the string $VV$ and overlap them with a section of substrate that has the form $ZZZ$. If $v$ happens to equal $3w$, everything works as we expect, but if $v$ is smaller than $3w$, we have too many degrees of freedom. We get around this difficulty by filling in the extra space in the middle unit of the substrate with extra $1$s:
Summary of Notation

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma$</td>
<td>Alphabet of key symbols.</td>
</tr>
<tr>
<td>$m$</td>
<td>The number of symbols in $\Sigma$; also the number of cells in a node.</td>
</tr>
<tr>
<td>$k$</td>
<td>The number of nodes in the trie, or the number of strings in an instance of BSCA.</td>
</tr>
<tr>
<td>$M$</td>
<td>The number of cells that can be stored in the memory.</td>
</tr>
<tr>
<td>$v$</td>
<td>The number of variables in a given instance of 3SAT.</td>
</tr>
<tr>
<td>$w$</td>
<td>The number of clauses in a given instance of 3SAT.</td>
</tr>
<tr>
<td>$p$</td>
<td>Indexes an enumeration of the seven possible ways of assigning truth values to three terms so that their disjunction is also true.</td>
</tr>
<tr>
<td>$c$</td>
<td>Padding factor. $M$ is set equal to $c$ times the length of the substrate.</td>
</tr>
<tr>
<td>$Z$</td>
<td>A unit of $3w$ successive 0s.</td>
</tr>
<tr>
<td>$B$</td>
<td>A unit of $3w$ successive 1s.</td>
</tr>
<tr>
<td>$V$</td>
<td>The unit that is used to construct the truth-assignment strings.</td>
</tr>
<tr>
<td>$A$</td>
<td>The unit that helps control the position the truth-assignment strings.</td>
</tr>
<tr>
<td>$K_j$</td>
<td>The &quot;key&quot; unit that identifies the clause-satisfiability string to which it belongs.</td>
</tr>
<tr>
<td>$L_j$</td>
<td>The &quot;lock&quot; unit in the substrate that matches only the key unit of the corresponding satisfiability string.</td>
</tr>
<tr>
<td>$T_{pj}$</td>
<td>A &quot;true-forcing&quot; unit that identifies a set of variables that need to be true to make clause $j$ true.</td>
</tr>
<tr>
<td>$F_{pj}$</td>
<td>A &quot;false-forcing&quot; unit that identifies a set of variables that need to be false to make clause $j$ true.</td>
</tr>
</tbody>
</table>

\[
A = 0^v 1^{3w - v}
\]

Thus the section of substrate where the $V$s are allowed to overlap looks like $ZAZ$. We still need to exclude the $V$s from the places where they are not allowed to overlap. This is easy. Note that the pattern $VV$ contains exactly two 1s, positioned so that they must overlap adjacent units of the substrate. Therefore, we can protect a section of the substrate where we don't want $VV$s by making the first, last, and every second unit of the section a $B$ (all 1s).

We use a lock-and-key mechanism to fit the clause-satisfiability strings into their proper positions over the substrate. For each value of $j$, from 1 to $w$, we define

\[
K_j = 1^{j-1} 0 1^{3w - 2j} - 2 1^{j-1}
\]

\[
L_j = 0^{j-1} 10^{3w - 2j} - 2 1^{j-1}
\]

For each clause, there are seven clause-satisfiability strings, each of which can fit interchangeably into one of seven slots in the substrate. The first slot for each clause has the form $BBL_j BAZB j BB$, while the remaining six slots have the form $BBL_j B B j B j B j$. Thus, the only place where the $VV$s can overlap is in the center three units of the first slot for each clause. The clause-satisfiability strings that fit into these slots begin and end with $K_j$ (to match the $L_j$ in the substrate), and have the property that they do not anywhere contain a run of $12w$ consecutive 0s that could span the four

* Note that in the case where $w = 1$, the quantity $3w - 2j - 2$ is negative, so we must make a special-case definition: $K_1 = 101$, and $L_1 = 010$. 
consecutive $B$s that serve to separate one slot from the next in the substrate. This is more than sufficient to guarantee that the clause-satisfiability strings will fit only where we want them.

The only remaining units that we need to define are the ones that go in the center section of the clause-satisfiability strings to force the variable assignments to satisfy the clause. Remember that each clause is a disjunction of three terms, each of which is a variable or the negation of a variable. Thus we will need to use two subscripts: let $j$ range over an enumeration of the clauses, from 1 to $w$, and let $p$ enumerate the seven different ways of assigning truth values to three terms in such a way that at least one term is true. Then we can define $T_{pj}$ (the true-forcing unit) and $F_{pj}$ (the false-forcing unit) iteratively as follows: Initially set both $T_{pj}$ and $F_{pj}$ to all 0s ($0^{3w}$). Then for each term $i$ in clause $j$, let $x_i$ denote the index of the variable in term $i$. If the $x_i$th variable needs to be assigned a true value to satisfy the $p$th term assignment, then set the $x_i$th bit of $T_{pj}$ to 1. Otherwise, set the $x_i$th bit of $F_{pj}$ to 1.

Now we can put together each of the clause-satisfiability strings. For the $j$th clause and the $p$th term assignment, the string looks like $K_jT_{pj}ZF_{pj}ZK_j$. Note that this matches either the $L_jBZAZBL_j$ of the first slot for this term in the substrate, or the $L_jBZBZBL_j$ of each subsequent slot for this term. Further note that a 1 in the $T_{pj}$ unit will force the corresponding variable-assignment string to the right, while a 1 in the $F_{pj}$ unit will force it to the left. It is easy to prove that no clause-satisfiability string contains a run of $12w$ consecutive 0s, since the $K_j$ units always begin and end with 1s, and that it is not possible for both $T_{pj}$ and $F_{pj}$ to contain all 0s.

We are now ready to lay out the entire construction in one piece. First the substrate string:

$$[BBL_jBZAZBL_jBB]_{j=1,w} \quad [(BBL_jBZBZBL_jBB]_{j=1,w})^6$$

It begins with one slot for each clause that allows the variable assignment strings to overlap, and is then followed by six repetitions of one slot for each clause with variable assignments excluded. The total length of the substrate is $77w$ units or $231w^2$ bits. Next, the $7w$ clause-satisfiability strings, for $j$ ranging from 1 to $w$, and $p$ ranging from 1 to 7:

$$K_jT_{pj}ZF_{pj}ZK_j$$

The length of a clause-satisfiability string is 7 units, or $21w$ bits. Finally, the $v$ variable-assignment strings, all identical:

$$[V^vV^v]_w$$

Each variable-assignment string is $11w$ units or $33w^2$ bits long, and presents a copy of the $V^vV^v$ pattern in exactly the same relative position in each of the first $w$ slots of the substrate, assuring consistent assignments for the same variable in different clauses.

From this construction, it is straightforward to verify that (a) from a given 3SAT problem, the corresponding BSCA problem can be constructed in polynomial time, (b) from each solution of the
3SAT problem, we can derive a solution of the corresponding BSCA problem,* (c) from each solution of the BSCA problem, we can derive a solution to the corresponding 3SAT problem. This completes the proof that BSCA is NP-complete.

5. Relation of String Collision Avoidance to Trie Compaction
There are a few subtle differences between BSCA and the trie compaction problem. The first is that BSCA implicitly allows strings of all 0s to be included in the set. Such a string would correspond to an interior trie node with no descendants, which cannot normally occur. The second point is that we have relaxed the restriction that all of the strings be the same length. This makes the problem slightly more general, but will require us eventually to show that our proof need not depend on the lengths of the strings.

The third difference is that an arbitrary instance of BSCA may not reduce directly to a valid instance of a trie compaction problem. The obvious mapping from an instance of BSCA to a trie compaction problem maps each string to a distinct interior node, taking 0s to nil-valued cells and 1s to non-nil cells. If there are \(k\) strings, and thus \(k\) nodes, the 1s will need to be arranged so that exactly \(k-1\) of them map to interior pointers and at least one is left over to be mapped to a leaf pointer. Some trie structures separate leaf pointers from interior pointers and assign them to different regions of a node. The mapping will fail to produce a valid trie unless the right number of 1s fall into each region. For example, any non-nil cell indexed by a final symbol may be occupied only by a leaf pointer. If an instance of BSCA contains fewer than \(k-1\) 1s, then the mapping fails to produce a valid trie.

We deal with these differences by showing that our construction can be manipulated so that any instance of 3SAT can be mapped to an instance of BSCA that is obviously equivalent to a trie compaction problem.

The first point is the easiest to dispose of; we need only verify that our construction never produces a string of all 0s. This is quite clearly the case.

The second point is almost as easy to handle. To make the strings all the same length, we need only pad the short ones with 0s to the length of the longest (the substrate). For the sake of argument, we will assume the padding is on the right. We are not quite done; we now face the problem that all of the strings are exactly as long as the memory, and must therefore all start at location 0. We can get around this problem in one of three ways. We can permute memory addresses to wrap around so that the excess 0s overlap harmlessly with the bottom of memory. We can make the rule that an attempt to retrieve a bit from outside of memory always returns a 0, so that it won't matter if the excess is not stored. Or we can enlarge the memory to, say \(c\) times the length of the substrate string, and construct

---

* Actually, we can derive many solutions, but one suffices.
Optimal Trie Compaction is NP-Complete

c - 1 additional strings of all 1s to force all the overlapping to occur within the bounds of the substrate string, with the excess 0s falling on the string to the right.

The third problem is hard to solve in general, since the exact nature of the problem depends on a specific detail of the trie implementation; we must know how leaf pointers and interior pointers are distinguished. If the method is some non-positional attribute like a special mark, we have no problem. We simply mark any $k - 1$ non-nil cells in the node corresponding to the substrate string\(^*\) as interior pointers, and the rest as leaf pointers.

If the two types of pointers are distinguished by their position within the node, we get into difficulty at once because the construction used to reduce 3SAT to BSCA relies on being able to place 1s in certain very precisely specified positions. Most of the strings are padded to the length of the substrate, and the existence of this padding gives us some freedom to shift the 1s in those strings to positions occupied by leaf pointers. This is not a guaranteed solution, and furthermore, the substrate string alone places a very large number of 1s in very specific positions, probably resulting in many more interior pointers than we can handle.

The obvious solution is to increase the number of nodes to match the number of interior pointers left over. We do this by breaking up the $c - 1$ additional strings that we introduced to extend the memory size. We need to pick a pattern that contains few 1s, all of them in leaf-pointer positions, that cannot be placed over the substrate string, and that forms an efficient space-filler when multiple copies are overlapped. An example of such a pattern is VVVV, which contains four 1s, spaced to fall at the same relative positions in four adjacent units. We presume that this pattern can be shifted so that all four 1s represent leaf pointers. Assume that we have $q$ leftover interior pointers. Note that $q$ is less than $231w^2$. Since each VVVV string fills in four 1s, we will need at most $924w^2$ additional memory cells to hold them. We can get that many empty cells by setting $c = 5$. We can fill in any extra space by shifting the VVVV string so that exactly one of the 1s is in an interior pointer position. This creates a space-filling\(\dagger\) pattern that can be arbitrarily repeated without changing the balance between interior pointers and nodes.

What this shows is that if the rules for forming tries are not made unusually complicated, then any instance of 3SAT can be reduced to an instance of BSCA that is the image of a valid trie compaction problem under the straightforward mapping.

\(^*\) Note that $k - 1 = w + 7w^2 + c - 1$, which is less than $10w + c$, which is considerably less than $144w^2$, which in turn is the number of 1s in that portion of the substrate represented by 0s, so there are enough non-nil pointers in the node representing the substrate to point to all of the other nodes in the trie.

\(^\dagger\) We are ignoring here the possibility that some of the 0s used to pad these VVVV strings may fall outside the available memory. This does not present much of a problem, since we can presumably arrange most of the padding to fall on the side that does not cause trouble, and then add sufficient memory cells to catch the rest. If we need to, we can add another node or two, specially constructed to contain all leaf pointers and to occupy the extra space.
References

