Log Compression Algorithm

David Barrett-Lennard
15 July 2005

Abstract
This paper describes a log compression algorithm that is very efficient and results in maximal compression. An operation is stored as an ordered set of deletion intervals plus an ordered set of insertion intervals, allowing it to concisely represent any state difference. This means operations are closed under "addition" (i.e. merging). The merging algorithm involves a left to right scan through the intervals in one operation as well as the intervals in the other operation, choosing the next interval to process in a manner reminiscent of merge sort. The result is that merging is achieved in time linear in the number of string-wise intervals.

The merging process is effectively doing IT and ET but avoiding the ERV puzzles [1]. Of course that is simply because it doesn't exercise the problematic cases. I.e. it never ET's an insert backwards past a delete, and it never IT's an insert forwards past an insert [2].

When transforming lists L_1, L_2 of string-wise operations against each other it is conventionally necessary to transform each $L_1[i]$ against each $L_2[j]$, resulting in an $O(|L_1| |L_2|)$ algorithm. This can be prohibitive when $|L_1|$ and $|L_2|$ are large. The merging algorithm suggests that a general implementation of IT/ET may be possible that is practical for complex operations with many stringwise deletion and insertion intervals. The difficultly is in respecting the effects relation [1]. This is an area for future research.

[3] has also described a log compression algorithm that results in maximal compression. However, they express their solution directly in terms of IT and ET, and this raises questions such as

- What happens when $ET(O_a,O_b)$ when $O_b \rightarrow O_a$?
- What about the effects relation and the ERV puzzles?

Introduction
As in [1], by definition characters in text documents have identity. A character is originally inserted by a particular user at a particular site. The character keeps its unique identity even though its index position in the document needs to be adjusted according to continued editing operations. Note that a character doesn't simply relate to its appearance (eg its ASCII code) - for example each appearance of the letter 'A' in a document represents a different character.

We take the convention that strings use zero based index positions. Given string s, let s[i] be the i^{th} character. s[i,j) denotes the substring corresponding to the half open interval [i,j). chars(s) denotes the set of characters in s.

For string s, the following mutative operations are defined

- delete(s, p,n) - delete the range of characters [p,p+n) from s
- insert(s, p,t) - insert string t into s at position p

Definition 1: Let S represent a particular state of a given text document. A *range* r in S is a tuple (p,s) where p is an index position of a non-empty substring s that appears within S. Given range r, let p(r) denote the position, s(r) the substring and n(r) the length of the substring. Note therefore that

$$S[p(r), p(r) + n(r)) = s(r)$$

using the notation S[i,j) to denote a substring in S.

```
// C++
typedef std::string String;
struct Range
{
    Range() : p(0), n(0) {}
    Range(const String& _s, int _p) : p(_p), n(_s.size()), s(_s) {}

    int p;
    int n;
    String s;
};
```

Definition 2: Let S represent a particular state of a given text document. A *range list* L is a list of ranges (L_0, L_1, ... , L_{n-1}) in S satisfying $\forall i<j$, $p(L_i) + n(L_i) < p(L_j)$. ie the half open intervals do not overlap or touch, and are ordered by position from left to right.

We write state(L) to denote the text document state S on which L is defined. It is assumed that the strings in L will be found as sub-strings at precisely the specified positions within S. ie $\forall i$, $S[p(L_i), p(L_i) + n(L_i)) = s(L_i)$.

Given range list L, let $chars(L) = \cup_i \ chars(s(L_i))$.

```
// C++
typedef std::list<Range> Ranges;
```

Lemma 1: $state(L_1) = state(L_2) \ \& \ chars(L_1) = chars(L_2) \Rightarrow L_1 = L_2$
ie if L_1 and L_2 specify the same set of characters from the same associated document state, then they have an identical representation as a range list. Note that our definition disallows empty ranges, or ranges that overlap or touch. These restrictions lead to a uniqueness of representation.

Definition 3: An *extraction list* X is a range list interpreted as a set of extractions to be performed on an associated document state S. We write S' = S - X for the document state S' obtained by applying the extractions in X to S. For each X_i, the string $s(X_i)$ is to be extracted at position $p(X_i)$ from S. The positions are in "pre-extraction coordinates" - ie they relate to the locations of the extracted characters as seen in S before any actual extractions begin. ie state(X) = S.

To perform the extraction it is easiest to iterate through the ranges from right to left (ie in reverse order) so that each extraction isn't upset by previously performed extractions - because they are on the right. The in-place (ie mutative) algorithm is:-

```
// C++
// Let s = s - X
void ApplyExtractions(String& s, const Ranges& X)
{
    for (Ranges::const_reverse_iterator x = X.rbegin() ; x != X.rend() ; ++x)
    {
        s.erase(s.begin() + x->p, s.begin() + x->p + x->n);
    }
}
```

Definition 4: An *insertion list* I is a range list interpreted as a set of insertions to be performed on an associated document state S. We write S' = S + I for the document state S' obtained by applying the insertions in I to S. For each I_i, the string $s(I_i)$ is to be inserted at position $p(I_i)$ in S. The positions are in "post-insertion coordinates" - ie they relate to the final locations of the inserted characters as seen in S' after all insertions have been performed. ie state(I) = S'

To perform the insertion it is easiest to iterate through the ranges from left to right (ie in forwards order) so that for each insertion, all insertions to the left have already been performed - in keeping with the use of post-insertion coordinates. The in-place algorithm is:-

```
// C++
// Let s = s + I
void ApplyInsertions(String& s, const Ranges& I)
{
    for (Ranges::const_iterator i = I.begin() ; i != I.end() ; ++i)
    {
        s.insert(s.begin() + i->p, i->s.begin(), i->s.end());
    }
}
```

Comment on this convention
Note that there is a nice symmetry between extractions and insertions. If we reverse the direction of time then extractions look like insertions and vice versa. Note how pre-extraction coordinates become post-insertion coordinates when we reverse time. Also note how we iterate through the ranges in the opposite order - in keeping with the analogy of reversing the direction of time.

Lemma 2: Any sequence of deletions and insertions on a string can be modeled as a set of extractions followed by a set of insertions.

Proof: All characters in the original string that were not deleted must appear in the final string, in the same relative order. Therefore the overall effect of any sequence of deletions and insertions can be determined by what characters have been deleted and what characters have been inserted. Characters that were inserted then deleted again are not relevant.

Lemma 3: $(S - X_1) + I_1 = (S - X_2) + I_2 \Rightarrow X_1 = X_2 \ \& \ I_1 = I_2$
In other words, there is a uniqueness of representation when operations on a text document are modeled as a set of extractions followed by a set of insertions. This follows from lemma 1, noting that state(X_1) = state(X_2) = S, and chars(X_1) = chars(X_2) or else couldn't have equality of resultant state. A similar line of reasoning shows that $I_1 = I_2$.

Definition 5: Suppose $(S_1 - X) + I = S_2$. We write this as $S_1 + \Delta = S_2$ where $\Delta = [-X +I]$ is the *state difference* between S_1 and S_2. By the previous two lemmas, any changes to a text document can be uniquely represented as a state difference in this manner.

Note that chars(S_1) \ chars(S_2) = chars(X) and chars(S_2) \ chars(S_1) = chars(I).

Definition 6: Let $(S_1 + \Delta_1) + \Delta_2 = S_2$. By lemma 2 we can also write $S_1 + \Delta = S_2$ and Δ is uniquely defined. This gives as a well-defined way of merging state differences. We write this as $\Delta = \Delta_1 + \Delta_2$.

Commentary:
This is interesting because it shows that building a state difference can be done unambiguously without any concern for the effects relation or the need for tie breaking using site identifiers [1]. As it turns out, an efficient algorithm is straightforward and will be described in the remainder of this paper.

An operation is represented using a set of extractions and a set of insertions. This allows a single operation to concisely represent arbitrary differences between two document states. For example, a single operation could efficiently represent weeks of work on a text document. Allowing for operations to be closed under "addition" (ie merging) is a useful property.

```
// C++
// Partial implementation of an operation
struct Operation
{
    Ranges X;
    Ranges I;
};
```

Merging extractions: Let $(S_1 - X_1) - X_2 = S_2$. We want X satisfying $S_1 - X = S_2$.

X_1 and X_2 are contextually serialised. There can't be a character deleted by X_1 that is again deleted by X_2. Merging the extractions is conceptually very simple - it is just a matter of translating positions correctly.

Note that state(X) = S_1 = state(X_1), so there is no need to translate positions in X_1. However positions in X_2 are in the context of having already performed X_1, so we need to exclude the effect of X_1 by shifting positions in X_2 to the right by the relevant number of characters that have been extracted by X_1.

An efficient implementation can scan through both the ranges in X_1 and X_2 from left to right - stepping whichever range comes next. A variable can keep track of the accumulated number of characters deleted by X_1. This is used to adjust the positions of X_2.

Ranges to be added to X that touch need to be coalesced.

This algorithm is $O(|X_1| + |X_2|)$

```
// Given [-X1 -X2], merge X2 into X1.
void MergeExtractions(Ranges& X1, const Ranges& X2)
{
    int shift = 0;
    Ranges::iterator x1 = X1.begin();
    Ranges::const_iterator x2 = X2.begin();
    while(x2 != X2.end())
    {
        //      [11...)
        //        [22..)
        //
        // Skip over all [11..) that are strictly to the left of the next [22..) in its shifted
        // position.  These [11..) don't require adjustment.  However we need to accumulate the
        // shift.
        // These intervals don't actually overlap because the effect of [11..) is to shift the
        // next [22..) further to the right, past the end of the [11..)
        while(x1 != X1.end() && x1->p < shift + x2->p)
        {
            shift += x1->n;
            ++x1;
        }

        /*
        Now process all [11..) that coalesce with the next [22..).  The effect is to split the next
        [22..) into pieces that are shifted by different amounts.

        Initially we position the [22..) according to the initial shift.  In the following
        example it aligns on the left with the next [11..).

        As we process each [11..) we have to split the next [22..) into a left portion that is
        appended to r using the current shift,  and a right portion that is shifted further to the
        right and requires further processing.

                   2222222222222222222
                   111     111111       1111

            --->      2222222222222222222
                   111     111111       1111

            --->      22222      222222222222222
                   111     111111       1111

            --->      22222      2222222    22222222
                   111     111111       1111


        So     r = 11122222111111222222211112222222 is the coalesced string to be extracted

        [t1,t2) represents the next substring from [22..) to be appended to r
        */
```

```
        Range r;
        r.p = shift + x2->p;
        String::const_iterator t1 = x2->s.begin();
        while(x1 != X1.end() && x1->p <= shift + x2->p + x2->n)
        {
            // Append next substring from [22..) to end of r
            String::const_iterator t2 = x2->s.begin() + x1->p - (shift + x2->p);
            r.s.append(t1,t2);
            t1 = t2;

            // Append next [11..) to end of r
            r.s += x1->s;

            // Accumulate shift from [11..), and erase [11..) because its effect is built into r
            shift += x1->n;
            x1 = X1.erase(x1);
        }
        r.s.append( t1, x2->s.end() );  // Append remainder of [22..) to end of r
        r.n = r.s.size();
        X1.insert(x1, r);

        ++x2;
    }
}
```

Merging insertions: Let $(S_1 + I_1) + I_2 = S_2$. We want I satisfying $S_1 + I = S_2$.

I_1 and I_2 are contextually serialised. A character inserted by I_1 can't also be inserted by I_2. Merging the insertions is conceptually very simple - it is just a matter of translating positions correctly.

Note that state(I) = S_2 = state(I_2), so there is no need to translate positions in I_2. However positions in I_1 relate to the intermediate state $S' = S_1 + I_1$, and need to be translated to corresponding positions in S_2. So positions in I_1 must be shifted to the right according to the relevant number of characters that have been inserted by I_2.

An efficient implementation can scan through both the ranges in I_1 and I_2 from left to right - stepping whichever range comes next. A variable can keep track of the accumulated number of characters inserted by I_2. This is used to adjust the positions of I_1.

Ranges to be added to I that touch need to be coalesced.

This algorithm is $O(|I_1| + |I_2|)$

```
// Given [+I1 +I2], merge I2 into I1 (ie I1 += I2)
void MergeInsertions(Ranges& I1, const Ranges& I2)
{
    int shift = 0;
    Ranges::iterator i1 = I1.begin();
    Ranges::const_iterator i2 = I2.begin();
    while(i1 != I1.end())
    {
        while (i2 != I2.end() && i2->p < shift + i1->p)
        {
            I1.insert(i1, *i2);
            shift += i2->n;
            ++i2;
        }
        if (i2 == I2.end())
        {
            do
            {
                i1->p += shift;
                ++i1;
            } while(i1 != I1.end());
            return;
        }
        i1->p += shift;
        while (i2 != I2.end() && i2->p <= i1->p + i1->n)
```

```
        {
            i1->s.insert(i1->s.begin() + i2->p - i1->p,
                i2->s.begin(), i2->s.end());
            i1->n += i2->n;
            shift += i2->n;
            ++i2;
        }
        ++i1;
    }
    while(i2 != I2.end())
    {
        I1.insert(I1.end(), *i2);
        ++i2;
    }
}
```

Transposing insertions then extractions: Let $(S_1 + I) - X = S_3$. We want X', I' satisfying $(S_1 - X') + I' = S_3$.

X may delete a character inserted by I. In that case both X' and I' won't mention the character.

Let $S_2 = S_1 + I$. Now state(X') = S_1, whereas state(X) = S_2. Extractions in X need to be shifted (to the left) to exclude the effect of I_1. Also, state(I') = S_3, whereas state(I) = S_2, therefore insertions in I need to be shifted (to the left) to include the effect of X.

Now state(X) = state(I) = S'. Therefore positions in X and I can be directly compared. An efficient implementation can scan through both the ranges in X and I from left to right - stepping whichever range comes next. A variable can keep track of the accumulated number of characters inserted by I. This is used to shift the positions of X to the left. Similarly a variable can keep track of the accumulated number of characters extracted by X and this is used to shift the positions of I to the left. Overlapping sections are characters that are inserted then deleted so these characters are discarded from X' and I'. Note finally that the extraction of characters from X can cause ranges in I to coalesce. Also, excluding the insertion of characters from I can cause ranges in X to coalesce.

```
// C++

/*
Split away prefix t from range r for given p.

        [rrrrrrrrrrrrrrrrr)

 --->   [ttttttt)[rrrrrrrr)
                 |
                 |
                 p
*/
void SplitRange(Range& r, int p, Range* t)
{
    int n1 = p - r.p;
    int n2 = r.n - n1;

    if (t)
    {
        t->p = r.p;
        t->n = n1;
        t->s = String(r.s.begin(), r.s.begin() + n1);
    }

    r.p = p;
    r.s.erase(r.s.begin(), r.s.begin() + n1);
    r.n = n2;
}

// The given I,X are contextually serialised as  [+I -X].
// Transform both I,X so they are instead contextually serialised as  [-X +I].
void TranposeInsertionsThenExtractions(Ranges& I, Ranges& X)
{
    int si = 0;     // Accumulated shift from I
    int sx = 0;     // Accumulated shift from X
```

```
Ranges::iterator i = I.begin();
Ranges::iterator x = X.begin();

while(i != I.end() && x != X.end())
{
    if (i->p + i->n <= x->p)
    {
        i->p -= sx;
        si += i->n;
        ++i;
    }
    else if (x->p + x->n <= i->p)
    {
        x->p -= si;
        sx += x->n;
        ++x;
    }
    else
    {
        if (i->p < x->p)
        {
            Range t;
            SplitRange(*i, x->p, &t);
            t.p -= sx;
            si += t.n;
            I.insert(i, t);
        }
        else if (x->p < i->p)
        {
            Range t;
            SplitRange(*x, i->p, &t);
            t.p -= si;
            sx += t.n;
            X.insert(x, t);
        }
        else
        {
            if (x->n < i->n)
            {
                SplitRange(*i, x->p + x->n, NULL);
                sx += x->n;
                si += x->n;
                x = X.erase(x);
            }
            else if (i->n < x->n)
            {
                Range t;
                SplitRange(*x, i->p + i->n, NULL);
                si += i->n;
                sx += i->n;
                i = I.erase(i);
            }
            else
            {
                sx += x->n;
                si += i->n;
                i = I.erase(i);
                x = I.erase(x);
            }
        }
    }
}
if (si)
{
    while (x != X.end())
    {
        x->p -= si;
        ++x;
    }
}
if (sx)
{
    while (i != I.end())
    {
        i->p -= sx;
        ++i;
    }
}
```

```
    CoalesceRanges(I);
    CoalesceRanges(X);
}
```

Merging operations: We are now ready to state the algorithm to merge operations.

Let $O_1 = [-X_1 +I_1]$, $O_2 = [-X_2 + I_2]$

Then

$[O_1\ O_2]$ = $[-X_1\ +I_1\ -X_2\ +I_2]$
~ $[-X_1\ -X_2'\ +I_1'\ +I_2]$ (by transposing $+I_1$, $-X_2$)
~ $[-X\ +I]$ (by merging extractions and insertions)

where ~ denotes equivalence in terms of state change.

```
// C++
// In-place algorithm
// Merge O2 into O1
void Merge(Operation& O1, const operation& O2)
{
    Ranges X2 = O2.X;
    TranposeInsertionsThenExtractions(O1.I, X2);
    MergeExtractions(O1.X,X2);
    MergeInsertions(O1.I,O2.I);
}
```

This algorithm is $O(|O_1.X| + |O_1.I| + |O_2.X| + |O_2.I|)$.

In-place algorithms
We have used in-place algorithms where possible to maximise performance by avoiding unnecessary copying. This can provide significant benefits when merging edits into a very large state difference.

Compressing the log: This algorithm should readily allow hours, weeks or even months of edits on a complex document to be compressed into a concise state difference. Note that the state difference is provably as concise as it can be. In practice the merging process results in a relatively small number of net extractions and insertions. Therefore merging many edits is very fast and probably closer to O(n) than the worst case which is $O(n^2)$, where n is the number of simple edits to be compressed. The worst case requires edits to be "disconnected" from each other so they don't coalesce, and this doesn't generally occur in practice.

Relationship to IT and ET
There is clearly a close relationship between the algorithm to merge state differences and IT/ET. For example transposing insertions and extractions essentially involves the ET of the extractions backwards past the insertions, and the IT of the insertions forwards past the (transformed) extractions.

However, it seems better to reserve IT/ET terminology for where we have to consider the effects relation and make use of site identifiers to break ties. There are a number of reasons for this approach.

- Our exposition of the state difference can be understood in isolation without any talk of ERV puzzles and using site ids to break ties. In other words we demonstrate very clearly how the calculation of a state difference avoids the problematic aspects of IT and ET.
- The approach taken provides very efficient algorithms. For example the algorithm to transpose insertions and deletions performs the equivalent of both an ET and an IT in a single scan of the two range lists.

- As it turns out, ET of insertions backward past insertions drops some vital information needed for merging insertions. Given $[+I_1 +I_2]$, let I_2 insert a string in the middle of a string inserted by I_1. Then if I_2 is ET'd backwards past I_1 the knowledge about the relative position of I_2 in I_1 is lost.

Performance advantage of left to right scans

When ITing lists L_1, L_2 against each other it is generally necessary to IT each $L_1[i]$ against each $L_2[j]$, resulting in an $O(|L_1| |L_2|)$ algorithm. This can be prohibitive when $|L_1|$ and $|L_2|$ are large.

We allow a single operation to store any number of extractions and insertions, efficiently representing an arbitrary state difference. Consider that operations o_1, o_2 were instead represented as a composite of string-wise extraction and insertion operations. Then $IT(o_1,o_2)$ would result in the prohibitive quadratic order algorithm. By contrast our approach uses a left to right scan that is an $O(|o_1| + |o_2|)$ algorithm, where $|o|$ is the total number of string-wise extractions and insertions in operation o. In essence we avoid comparing every string-wise interval in o_1 with every string-wise interval in o_2.

This should make the technique of operational transform appropriate for configuration management.

References

[1]	Du Li and Rui Li. Ensuring consistency in real-time group editors. *ACM Transactions on Computer-Human Interaction,* April 2004. Under review.
[2]	Du Li and Rui Li. An Operational Transformation Algorithm and Performance Evaluation. *Journal of CSCW,* July 2005. Under review.
[3]	Haifeng Shen and Chenzheng Sun. A Log Compression Algorithm for Operation-based Version Control Systems. In *Proceedings of IEEE 26th Annual International Computer Software and Application Conference*, pages 867-872. IEEE computer society, Aug 2002.