Department of Computing and Information Systems

COMP10002 Foundations of Algorithms
Semester 2, 2015
Assignment 2

Learning Outcomes

In this project you will demonstrate your understanding of dynamic memory and linked data structures. You will also extend your skills in terms of program design, testing, and debugging.

Compression

Data is stored in files as sequences of bytes. One way of representing data is via printable ASCII characters (see the Table on page 60 of the textbook) in a text file that can be manipulated by a text editor. Data can also be represented using binary codes. Both of these forms may contain various forms of *redundancy*. For example, there are only 95 printable ASCII characters, and so a text file could be represented using 7 bits per character rather than 8, allowing $1/8$ of the bits to be saved. Repeated substrings and other such patterns are also forms of redundancy that can be identified and exploited by a *text compression program*. As an extreme example, imagine a file that contains a million `'a'`s – does it really need a million bytes to store it? Or could it be represented in, say, $4{,}000$ bytes, or $1{,}000$ bytes, or even 50 bytes?

The *Lempel-Ziv 1978* (LZ78) compression mechanism is one such compression technique. In this project you will implement the encoder for a simple LZ78-style compression mechanism. To avoid unnecessary complexity, your program will generate text output rather than bit-by-bit output, and the overall compression rate will be calculated based on an inferred representation rather than by counting the bits and bytes that are generated; nevertheless, it is clear that the computed output sizes could achieved using simple binary codes.

LZ78 Compression

In the LZ78 scheme, a *dictionary* D of phrases is constructed from the input file, starting with an initial dictionary that contains just one phrase, the zero-length *empty* string, in $D[0]$. The rules for forming subsequent phrases from an input text T are as follows:

- Find the index in D of the longest phrase in the dictionary that matches the yet-to-be-processed prefix of T. Let the index of that matching phrase be k. (Such a longest match must exist, because the empty string in $D[0]$ is a prefix of every string.)
- Let c be the first character in T that didn't get included in the match with $D[k]$.
- Write the pair (c, k) to the compressed representation as a description of a *factor* parsed from the input file.
- Form a new phrase p by adding c to the end of $D[k]$.
- Add p to D as a new phrase, and increase the size of D by one.
- Remove p from T, since it has now been processed.
- Continue until T is empty.

For example, suppose T = "`ab:ab:ab:ab:`", a total of 12 characters. If it is processed using this mechanism, a total of seven phrases get added to the dictionary, and T gets represented as a sequence of seven factors:

Factor	(c, k)	p	T remaining
0		$D[0] =$ `""`	`"ab:ab:ab:ab:"`
1	`('a',0)`	$D[1] =$ `"a"`	`"b:ab:ab:ab:"`
2	`('b',0)`	$D[2] =$ `"b"`	`":ab:ab:ab:"`
3	`(':',0)`	$D[3] =$ `":"`	`"ab:ab:ab:"`
4	`('b',1)`	$D[4] =$ `"ab"`	`":ab:ab:"`
5	`('a',3)`	$D[5] =$ `":a"`	`"b:ab:"`
6	`(':',2)`	$D[6] =$ `"b:"`	`"ab:"`
7	`(':',4)`	$D[7] =$ `"ab:"`	`""`

That is, the string `"ab:ab:ab:ab:"` is completely represented by this text sequence containing seven factor descriptions:

```
a0 b0 :0 b1 a3 :2 :4
```

In this particular case (and for most other examples based on short input strings) the output is longer than the input; an effect exacerbated by the use of character output (including the newline or blank character at the end of each line). But if each output value c is coded into 7 bits (because that is enough to cover the 95 possibilities), and if there are n phrases in D at any given moment, the kth of them is coded using $\lceil \log_2 n \rceil$ bits (because ditto), and if the newline/blank is dropped (because if the lengths in bits of each of the two codewords is known, the newline isn't actually required); then reasonable savings will result on non-trivial files. On this simple example, only 63 bits, or 8 bytes, would be required by such an encoder, a saving of four bytes. *Note that you are not required to write a bit-by-bit encoder in this project, but may be interested to do so as a hobby once you have completed your final submission. The output from your program* **must** *be text in the form of the line shown above, but with newlines between the factors instead of blanks.*

Your Mission...

Copy the file `ass2d.c` from the LMS page. It provides a *decoder* for the data format shown in the line above, first reading a character c, and then, if that is successful, reading a phrase number k, with each input pair separated by a character that is read and discarded. The string $D[k]$ is then regenerated and written (enjoy the recursion); the character c is then written; and finally, the string $p \leftarrow D[k]c$ is added as a new phrase in the dictionary. Note that in the decoding program the dictionary is expected to be allowed to grow without limit; any controls on decoding memory space consumption are determined by the actions of the corresponding encoder.

To test the decoder, compile it, and run

```
ass2d < test0.txt | diff - test0-in.txt
ass2d < test1.txt | diff - test1-in.txt
ass2d < test2.txt | diff - test2-in.txt
```

to verify that you can decode the three example files and recreate the original inputs that they were generated from, stored in the three files `*-in.txt`. The standard tool `diff` is silent if the two streams it looks at are the same, with `-` meaning stdin; and is non-silent if it detects differences between them.

Now write an LZ78 encoder `ass2e.c` that takes an input text file (any text file at all, but in the first instance you should develop your program using the three `*-in.txt` files, without editing them in any way), and writes a corresponding "compressed" output file in the format expected by `ass2d`. You will also find these three commands very useful while debugging your encoder:

```
ass2e < test0-in.txt | diff - test0.txt
ass2e < test1-in.txt | diff - test1.txt
ass2e < test2-in.txt | diff - test2.txt
```

to check that you are generating *exactly* the same intermediate form as my reference implementation does. As you build confidence, you can check your encoder on any file `fyle.txt` using this chain:

```
ass2e < fyle.txt | ass2d | diff - fyle.txt
```

including `pg11.txt` and etc. And a reminder: you must *not* change `ass2d.c` in order to make your encoder work; when your programs gets tested, I'll be using *my* copy of `ass2d.c`.

Option 1: An OK Solution (12/15 marks)

Write a first version of the encoder that:

- reads the entire input into a `realloc()`'ed string of (eventually) approximately the right size before commencing any further processing;
- uses an array of string pointers (as is used in `realloc.c`) to store the factors as they are formed;
- assumes that there is no memory limit in terms of data structures and that "it will be ok" to request memory in the encoder (but as always, follows all `malloc()` and `realloc()` calls with an `assert()`), again following the style of `realloc.c`;
- performs linear search through that array looking for the longest matching prefix at each step.

Despite using linear search, your program should be able to encode `pg11.txt` within just a few seconds of CPU time. The "encoded" output file for `pg11.txt` has $32{,}272$ factors in it; and `ass2d` says that those factors could be represented in $84{,}653$ bytes, or 4.135 bits per character.

A couple of hints: you'll need to think very carefully about how you handle the last factor of the input so that the reconstructed text generated by `ass2d` is *exactly* the same as the original input file to `ass2e`; and you should think carefully about which way your linear search should run, and when it can terminate.

Option 2: A Better Solution (15/15 marks)

Replace the search structure with something better. Possibilities that you might want to consider include a binary search tree for the strings (you may embed the `treeops.c` code in your program if you wish); or a hash table that, for example, hashes the first two characters of each string to help reduce the cost of the search, and chains into a list all phrases that start with the same first two characters. In this second case you'll need to be sure that you handle one-character matches properly as well; and decide how to manage the set of lists that will emerge.

Either of these approaches should greatly accelerate the encoding process, and give rise to substantial reductions in running time on files like `pg11.txt` and `pg2600.txt`. But note that the worst case cost for encoding might still be bad – you can imagine what happens when a long string of n identical characters is input. Decoding speed will be unchanged between Option 1 and Option 2, and in any case is fast, since it doesn't require searching for factors.

Option 3: Going Above and Beyond (15/15 marks)

If you genuinely want a challenge, but no extra marks, find out what a *trie* is, and implement the dictionary as a trie. Probably best to not submit these programs, and instead lock in the marks with a truly careful Option 2 program.

Option 4: Holiday Fun

Write a back-end encoding program that takes the output from your `ass2e` program and writes a stream of bits packed into 32-bit words. You will need to use C's masking and shifting operations to pack bits into `unsigned int`'s (see Chapter 13), and use `fwrite()` to write them to `stdout`. Then write the corresponding back-end decoder program that converts bits from `unsigned int`'s read from `stdin` back in to the intermediate text form assumed in this project. If `bit2e` is the back-end encoder and `bit2d` is the back-end decoder, then the sequence

```
ass2e < fyle.txt | bit2e | bit2d | ass2d | diff - fyle.txt
```

should not report any differences

With care, you should be able to attain the compression rates predicted by `ass2d`, and the volume of data passed through the central pipe between `bit2e` and `bit2d` will be around half of the original data size. With additional thought and insight, you will be able to *beat* the `ass2d` predicted bit rate, by using other coding methods that are more sensitive to the biased distribution of the values being coded (the phrase numbers in particular) than is a uniform binary code. *Please do not submit any of these programs.*

The boring stuff...

This project is worth 15% of your final mark. A rubric explaining the expectations against which you will be marked will be provided via the LMS.

You need to submit your program for assessment; detailed instructions on how to do that will be posted on the LMS once submissions are opened. Submission will *not* be done via the LMS; instead you will need to log in to a Unix server and submit your files to a software system known as `submit`. You can (and should) use `submit` **both early and often** – to get used to the way it works, and also to check that your program compiles correctly on our test system, which has some different characteristics to the lab machines. Only the last submission that you make before the deadline will be marked.

You may discuss your work during your workshop, and with others in the class, but what gets typed into your program must be individual work, not copied from anyone else. So, do **not** give hard copy or soft copy of your work to anyone; do **not** "lend" your "Uni backup" memory stick to others for any reason at all; and do **not** ask others to give you their programs "just so that I can take a look and get some ideas, I won't copy, honest". The best way to help your friends in this regard is to say a very firm "**no**" when they ask for a copy of, or to see, your program, pointing out that your "**no**", and their acceptance of that decision, is the only thing that will preserve your friendship. *A sophisticated program that undertakes deep structural analysis of C code identifying regions of similarity will be run over all submissions in "compare every pair" mode. Students whose programs are so identified will be referred to the Student Center.* See `https://academichonesty.unimelb.edu.au` for more information.

Deadline: Programs not submitted by **10:00am on Monday 19 October** will lose penalty marks at the rate of two marks per day or part day late. Students seeking extensions for medical or other "outside my control" reasons should email Alistair, `ammoffat@unimelb.edu.au` as soon as possible after those circumstances arise. Marks and a sample solution will be available on the LMS by Monday 5 October.

And remember, algorithms are fun!