Finding exact/approximate tandem repeats using heuristic string compression

Saad Mneimneh
Computer Science
Hunter College of CUNY

General description

We have seen in class how the Z algorithm can be used to determine all periodic prefixes of a string s. As a result, exact tandem repeats can be identified if they occur at the beginning of the string. For instance, consider the string $s = abaababaabx$. The Z algorithm will identify the following periodic prefixes: $abaaba$ and $abaababaab$ with periods of 3 and 5 respectively, which implies the existence of tandem repeats aba and $abaaba$. This also implies that the string can be compressed as $(abaab)2x$, by choosing the largest period prefix. In this project, we seek to achieve two objectives:

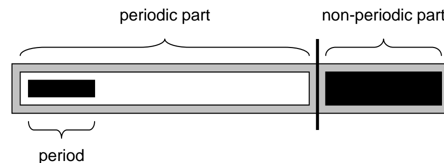
- find as many tandem repeats as we can, but allow for approximate repeats
- compress the string using the knowledge of these tandem repeats

Since the Z algorithm identifies tandem repeats that occur at the beginning of the string, we need an additional mechanism to find other tandem repeats that occur at arbitrary positions in the string. Moreover, the Z algorithm only identifies exact repeats. Both of the objectives listed above will be achieved by simple heuristics. The general requirements of the program are as follows:

- The program may receive its input from a file, a redirection, or a pipe. A parameter for the exactness of the repeats should also be provided (discussed later). For these cases, the program should output all the repeats it finds, and the compressed string, for each line of input it receives.
- The program may also receive input from the terminal. In this case, the program will output the compressed string in real time, by updating the compression while the user is typing (this will require to use non-canonical mode and suppress echoing on the terminal for the program to display the appropriate output). Upon pressing return, the program displays the tandem repeats and waits for new input.
- We will refrain from using the Z algorithm for the following two reasons:
  - we are interested in approximate repeats
  - the Z values must be updated from every new character that the user inputs, and this requires a processing time proportional to the length of the string for every key press

The first heuristic, compression

The compression will be based on a divide and conquer paradigm. It will first identify a periodic part. Based on that, it will then divide the string into two substrings, and recursively compress the two substrings in the same way. The two substrings will be the period and the non-periodic part, as illustrated below. If no periodic part was identified, the two substrings will be the two halves of the string.



The identification of the periodic part is done using the following heuristic. That heuristic, without the recursion, is used for the real-time update of the compression in case input is received from the terminal.

Let $s_1 = a$ be the first character of the string. We wait until another a is received, say $s_j = a$, (which matches the first a). As additional characters are received, we match those characters to the string $s_1 \dots s_{j-1}$, to obtain as many matches as we can, i.e. $s_1 \dots s_{j-1}$ is assumed to be the period. If a mismatch occurs, we record the longest periodic part reached so far, and we start looking for another a, say $s_k = a$. We then match $s_1 \dots s_{k-1}$ in the same way, and so on. In this case, every key press requires only a constant time processing. Here's an example on how the algorithm evolves when input is provided from the terminal.

The user presses 'a'. That's the first character. We now look for another 'a'. The user presses 'b', nothing happens. The user presses 'a', we start matching 'ab', a period of size 2. The user presses 'a', the match fails, but that's another 'a', so we start matching 'aba', a period of size 3. The user presses 'b', the matching continues. The user presses 'a', we have matched the period 'aba'. We continue to match 'aba'. The user presses 'b', the matching fails. The longest periodic part so far is 'abaaba' of period 2. The user presses 'a', that's another a, so we start matching 'abaabab', a period of size 7. The user presses 'a', the matching fails, but that's another 'a', so we start matching 'abaababa', a period of size 8. The user presses 'b' the matching fails. The user stops. So far, the longest periodic part is 'abaaba' of period 2. Therefore, we have:

$$(aba)2baab$$

Subsequently, the recursive process will divide the string into 'aba', and 'baab'. Note that the entire string received is 'abaababaab', which can be expressed as (abaab)2. The heuristic is therefore not guaranteed to find the largest periodic part of the string.

The second heuristic, approximate repeats

To allow for approximate repeats, the matching process may "cheat". For instance, when matching s_j to s_i, the matching process may match s_j to any of $s_{i-r} \dots s_{i+r}$. Furthermore, if s_j does not match any of these, a count is incremented. As long as the count is less or equal to a fraction α of the current

period size, the match continues; otherwise, a mismatch is declared. Both r and α are parameters. When both are zero, we have exact repeats.

Further detail

Further detail on the project and implementation can be obtained through discussion between the student and the professor.