Pattern-based Enhancements to Unicode Bidirectional Algorithm

Murhaf Hossari, Arthur W S Cater
UCD School of Computer Science and Informatics,
UCD Dublin,
Belfield,
Dublin 4, Ireland
murhaf.hossari@gmail.com, arthur.cater@ucd.ie

Abstract

In this paper, we present an improvement upon the Unicode Bidirectional Algorithm, eliminating the need for manually added directional codes in many cases. The modified algorithm recognizes cases conforming to four general patterns, and provides the correct directionality to their constituent characters without the need to use directional codes. Experiments performed on 593 paragraphs used in Apple software localised for Arabic showed that this approach succeeds in 86.3% of our defined cases correctly (the recognized four general patterns), which constitutes 54.3% of the total paragraphs in our test set. We also present other wrongly displayed cases requiring future work.

Keywords: *Unicode Bidirectional Algorithm, Right-to-Left Text Layout, Internationalisation, and Localisation*

1. Introduction

Internationalisation and Localisation are the processes that make it possible for users around the world to use software in their own languages. Often, different character sets (different scripts) are needed to write text in different languages, and some languages are read and written right-to-left rather than left-to-right. Unicode is now widely used in software when there is a need to deal with multiple scripts. Unicode provides standard unique codes for characters belonging to different scripts. It is also responsible for character representation in computer software. The Unicode Bidirectional Algorithm is responsible for representing scripts that have different horizontal directions (Gross 2006, Khaddam and Vanderdonckt 2011, Unicode 2012b).

Most scripts have left to right directionality. This means that the order by which the characters are stored in memory, also called logical order, corresponds to the order of by which the characters are displayed in a left-to-right sequence. This left-to-right order is called the visual order. However, for some scripts such as those used to write Arabic, Hebrew, Farsi, Urdu and others, the characters are written and read from right to left. The logical order of characters in memory is the reverse of their visual order. Users cannot be expected to accept software that needs them to type strings backwards (Atkin and Stansifer 2004, Unicode 2012c).

The issue gets more complicated when text contains scripts that have different directionality properties e.g. Arabic text with embedded English fragments, or vice versa. This is very common in localised software and this is exactly when the bidirectionality issue arises.

The Unicode Bidirectional Algorithm ("Bidi Algorithm") takes logical order strings as its input and reorders the characters to produce the visual order. It provides this functionality based partly on the nature of individual characters - whether they are right to left or left to right characters - and partly on clear specifications and rules dealing with digits, punctuation and other symbols. Frequently it succeeds in producing the correct visual order (IBM Corporation 2006, Abdelhadi et al 2011).

For example, the sentence "Doubt is a pain too lonely to know that faith is his brother" can be displayed according to the following different layouts:

- Left-to-right layout: *Doubt is a pain too lonely to know that faith is his brother.*
- Right-to-left layout: *.rehtorb sih si htiaf taht wonk ot ylenol oot niap a si tbuoD*
- Bidirectional layout: *Doubt is a pain too lonely to know rehtorb sih si htiaf taht.*

However, the Bidi Algorithm does sometimes fail to display the bidirectional text as it was intended by the user. Unicode provides various directional formatting codes to overcome these cases, where the ordering of characters can be manually adjusted in order to override the default behaviour of the Bidi Algorithm and thus show the text in the correct way (Unicode 2012c).

In this paper we describe our approach to improve the behaviour of the Bidi algorithm, by first finding and classifying the situations where the Bidi algorithm needs the addition of directional formatting codes to show the text correctly, and then providing solutions that can help the algorithm produce the correct character order without the need for directional codes.

We first analyze a large number of cases that previously needed directional formatting codes in a data set taken from Apple software localised for Arabic. Next, we classify these cases into different groups based on their format. Finally, we modify the Bidi algorithm to handle cases in each group so that the correct bidirectional display layout is shown. This modification does not change the overall behaviour of the algorithm in the vast majority of the correctly displayed cases.

The rest of this paper is organized as follows: in Section 2 we review the Unicode Bidirectional Algorithm and the problematic cases that it faces. In Section 3 we explain four patterns of these problematic cases which we handled in our approach. In Section 4 we list the evaluation methods we followed. Section 5 presents possible use of our modifications to the Unicode Bidirectional Algorithm. Section 6 presents our future work and Section 7 provides the conclusions.

2. Unicode Bidirectional Algorithm

The Bidi Algorithm reorders characters in a bidirectional text in order to produce the correct visual representation of the text. Text has a direction that is the general flow - letters and words in sentences - where English language for example has left-to-right flow, Arabic and Hebrew have right-to-left flow. Mixed text contains left-to-right characters and right-to-left characters and it flows in both right and left directions according to characters' type when it is displayed. That is why reordering of characters is needed (Unicode 2012c).

To achieve the reordering, The Bidi Algorithm defines a set of rules to extract the directionality property of each existing character and to deal with the logical order of characters in order to reach the visual order. It proceeds in multiple phases as will be briefly explained:

In the first phase it separates the text into multiple paragraphs. It then runs the following phases on each paragraph. Splitting the paragraphs is done by detecting a paragraph separator at the end of the paragraph. In the second phase, each character of the paragraph is annotated with its directional type. Directional types can be categorized in three main groups:

- Strong types, which include left-to-right characters (e.g. Roman characters), right-to-left characters (e.g. Arabic characters).
- Weak types: mainly numbers, number separators and unit symbols.
- Neutral types: most punctuation marks and white space character.

In the third phase, the directional types are transformed into the corresponding embedding levels by following an ordered sequence of rules. Each rule resolves a certain directional type (weak, neutral, white space,...etc) into either a right-to-left or left-to-right embedding level. At the end of this phase, all weak and neutral types will have either right or left direction. In the fourth phase, the characters are reordered according to the resolved embedding levels from the previous stage. It can be then displayed correctly (Ishida 2003, Unicode 2012c). An example to illustrate the different stages of Unicode Bidirectional Algorithm is given below.

The term Embedding level denotes a simple way to refer to the directionality of the characters where even and odd levels refer to left-to-right and right-to-left directions respectively. It also indicates the nesting depth of the text, the level increases when text nesting goes deeper. Deeper levels are explicitly added by using directional codes (override and embedding format codes). Levels start from 0, which is left-to-right. Maximum level is 61.

The term Paragraph embedding level (Base level) denotes the paragraph direction, usually determined by the first strong type character of the paragraph but possibly explicitly set. The code R indicates a strong right-to-left character; L indicates a strong left-to-right character; N indicates a neutral character, and

WS indicates a white space. Spaces between directional types and embedding levels are just for clarification

For expository purposes, upper case is used to refer to right-to-left characters in the input and output text of the example which now follows.

Logical (memory storage) Order: *CANNOT CONNECT TO SERVER* "*mail server name*"

Paragraph segmentation will result in recognizing the text as one paragraph. Base level is set to right-to-left because the first character is right-to-left.

Firstly, the Bidi algorithm transforms the text into directional types:

RRRRRR WS RRRRRRR WS RR WS RRRRRR WS N LLLL WS LLLLLL WS LLLL N

Secondly, the algorithm uses the following rules to resolve embedding levels:

- Neutrals between two left-to-right types get left-to-right direction, and similarly for right-to-left.
- Neutrals between left-to-right and right-to-left types get the embedding direction, which is usually the direction of the paragraph (first strong character direction) unless it is forced otherwise.

RRRRRR R RRRRRRR R RR R RRRRRR R R LLLL L LLLLLL L LLLL R

Thirdly, the algorithm sets the embedding levels to the corresponding values:

111111 1 1111111 1 11 1 111111 1 1 2222 2 222222 2 2222 1

Finally, the algorithm reorders the embedding levels accordingly and displays the correct visual order:

"mail server name" REVRES OT TCENNOC TONNAC

2.1 Problematic Cases

The Unicode Bidirectional Algorithm displays the correct layout for a text in most of the cases. There are various reasons why it sometimes does not, such as (1) assuming that the paragraph direction is the direction of the first strong character, (2) complicated nesting of strings of different types that cause neutral/weak characters to be misinterpreted, (3) Strings with a special nature such as part numbers. Rarely, the string is ambiguous even for human eye (Ishida 2003, Unicode 2012c).

To solve these problematic cases, explicit directional codes are manually added to the problematic text to enforce the correct layout. Those codes provide the help that Unicode Bidirectional Algorithm needs by defining a separate embedding level, adding an invisible strong-type character, or overriding directionality (Ishida 2003, W3C 2007, Unicode 2012c).

Directional codes defined as a global standard by Unicode are:

- U+202B RIGHT-TO-LEFT EMBEDDING (RLE)
- U+202A LEFT-TO-RIGHT EMBEDDING (LRE)
- U+202C POP DIRECTIONAL FORMATTING (PDF)
- U+200F RIGHT-TO-LEFT MARK (RLM)
- U+200E LEFT-TO-RIGHT MARK (LRM)

While using the directional codes can enforce the intended layout, there are some pitfalls for using them:

- They are manually added, and the fact that they are invisible can lead to mistakes.
- They are not trivial to use. They demand a good understanding of how the algorithm works which is not the case most of time.
- When localising software, translators will not always know how strings will be shown at runtime, strings can be dynamically composed from many substrings. Sometimes the only way to know there is a problem is to wait until it is reported. Then the translator fixes it with directional codes and then tries again.
- Being invisible, the Unicode characters in text are always in the danger of getting removed or changed whenever the text is modified or transferred among different environments.

In HTML, a few other ways were added to fix the problematic cases by creating new HTML tags that deal with bidirectional text. A property can be added to the HTML tag, that contains text, that will will be able to force the direction to either right-to-left or left-to-right rather than using the Bidi Algorithm implicit direction detection way. New embedding levels can be created to solve certain cases by adding special directionality HTML tags. Some new work is being done for HTML5 and new tags are being created for better support of bidirectional text (Lanin 2011). However, in our approach we propose a solution which is not related to a specific platform, moreover, it eliminates the need to add directional codes and HTML tags.

3. Patterns of Problematic Cases

Underlying the cases in which the Unicode Bidirectional Algorithm fails to produce the correct visual order, there are phenomena which are more common than others. We studied a large number of cases of bidirectional text, which are not displayed correctly in order to spot the most frequent cases. We used Apple software localised for Arabic, which has a significant amount of bidirectional strings and paragraphs. We first extracted the problematic cases by collecting the strings and paragraphs that contain directional codes. We assume that these strings have display problems as these directional codes would have been added manually by localisers to fix problems. Adding directional codes does not necessarily mean that the text is shown incorrectly by the Bidi Algorithm, due to the fact that strings might contain variables that are replaced with strings at runtime, so localisers sometimes add directional codes in order to guarantee that the layout would be correct regardless of what the variable was replaced with (Ishida 2003).

Studying these strings gave us closer look at the problematic cases that the Bidi Algorithm fails on. According to these cases we extracted the following frequently occurring causes:

1 Incorrect Paragraph Direction: this is the most common problem. It happens when a paragraph gets an incorrect direction (embedding level) due to its first character. As a result, the base level is set to a wrong direction. For example an Arabic paragraph that has the first word in English should have a right-to-left direction but because the first character is English, the paragraph is assigned a left-to-right direction by the Bidi algorithm.

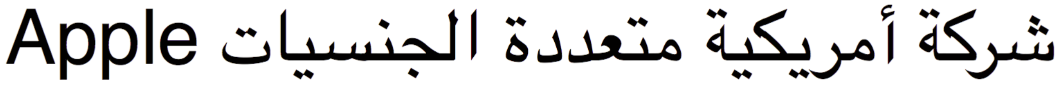
Apple شركة أمريكية متعددة الجنسيات

Figure 1. Layout without directional codes (incorrect layout)

Example:
A string belongs to this class will show incorrectly as in figure 1.

The correct layout is shown in figure 2.

For simplicity, will use lowercase for English and uppercase for Arabic Logical order:

apple IS AN AMERICAN MULTINATIONAL CORPORATION

Visual order by Unicode Bidirectional Algorithm:

apple NOITAROPROC LANOITANITLUM NACIREMA NA SI

The correct visual order:

NOITAROPROC LANOITANITLUM NACIREMA NA SI apple

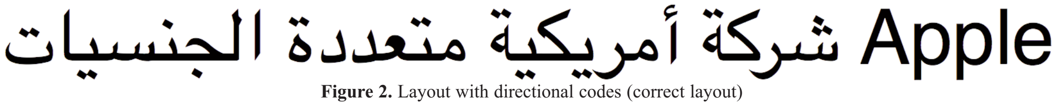
شركة أمريكية متعددة الجنسيات Apple

Figure 2. Layout with directional codes (correct layout)

This is usually fixed by adding a directional code “RLM (Right to left mark)” at the beginning of the paragraph. Because RLM has a right-to-left strong type, the paragraph will get a right-to-left direction and the problem is solved. However, other higher-level protocols are sometimes applied to the Unicode Bidirectional Algorithm where the paragraph direction is determined by the maximum number of characters of each directional type (Unicode 2012c).

We apply a mix of rules in order to improve the way that the Unicode Bidirectional Algorithm determines the paragraph direction. These rules consist of our defined rules which depend on last character and word count to determine paragraph direction combined with the rules that are already defined by the Bidi algorithm (first character and character count). The rules work as follows:

- If the majority of words and the majority of characters have the same directional type then:
    - The paragraph direction is the same as the direction of the majority of words.
- If the majority of words has different direction than the majority of characters then:
    - If the first and last strong character have the same direction then the paragraph direction is set to their direction.
    - If the first and last strong character have different directions then the paragraph direction is the direction of the majority of words.

However, these rules do not solve all the cases but we find that it improves the Algorithm's sense of the language of the paragraph. A possible alternative solution is to use more complicated language detection techniques but that is not suitable for use in the Bidi Algorithm as performance is vital when rendering text. Another possible solution, which is specific to localised software, deduces the language of the paragraph based on the language package containing it. For example, if the paragraph is taken from the Arabic localised content package, then treat it as a right-to-left paragraph.

2 Embedded contra-flowing bracketed text: this second cause is that the paragraph contains embedded text with the opposite direction which is surrounded by quotation marks, parentheses, brackets, or the like, and this embedded text also contains punctuation and/or more deeply embedded bidirectional text.

Example1:

Logical order: the application is *"SOME ARABIC NAME!"*

Visual order by the Bidi Algorithm: the application is *"EMAN CIBARA EMOS!"*

The correct visual order: the application is *"!EMAN CIBARA EMOS"*

The exclamation mark belongs to the right-to-left string and should show to its left. Translators might fix this either by adding an RLM directional code right after the exclamation mark, or by surrounding the right-to-left string (including the exclamation mark) with a pair of right-to-left embedding codes RLE and PDF.

Example 2:

Logical order: the application is *(NAME, co)*

Visual order by Unicode Bidirectional Algorithm: the application is *(EMAN, co)*

The correct visual order: the application is *(co ,EMAN)*

Similar solution as used in example 1 can be used to solve example 2.

The solution for this issue is based on the observation that material surrounded by parentheses, quotation marks, brackets or the like is usually a single coherent piece of text that is not tightly related to its surroundings. For this, we consider the surrounding characters to have some sort of balancing features where it usually has an opening character and closing one in a proper text. In case this scenario was detected, we treat balanced characters as having stronger combinatory feature than other punctuations within the text they surround. This means that these balanced characters define a separate segment that contains the balanced characters and the text within them. In our approach, we process this segment separately by the original Bidi algorithm. We then combine the result of this segment with the result of the surrounding text. This would produce the correct layout in most cases. Applying this to the examples above:

Example 1:

Logical order: *the application is "SOME ARABIC NAME!"*

Our algorithm will parse the string first and when the balanced characters are found it will be segmented into two strings:

- *the application is*
- *"SOME ARABIC NAME!"*

Processing these two strings by Unicode Bidirectional Algorithm rules will produce the following visual order:

- *the application is*
- *"!EMAN CIBARA EMOS"* (On its own this is a right-to-left string so the exclamation mark is placed correctly)

Combining those two segments will lead to the intended display of the whole string:

the application is *"!EMAN CIBARA EMOS"*

Example 2:

algorithm is "Bidi Parenthesis Algorithm" (Unicode 2012a). The proposed enhancement will be either applied to the original bidi algorithm or will be added as higher-level protocol that can be used when needed.

3 Embedded contra-flowing partly-bracketed text: The third cause is somewhat similar to the previous in that it also contains balanced characters, but here the problem occurs with the display of the balanced characters themselves. It occurs when the paragraph contains a substring with an opposite direction to the paragraph direction (base direction), the substring is composed of some characters surrounded by balanced characters and some others not surrounded by them and they are either following or preceding them.

Possible directional representations of the case:

RRR LLL (LLLLLL) RRRRR

RRRRR RRR "LLLL" LL RRR

LLL LLL RRR {RRRRR RRR RR} LLL LL

LLLLLLLLL LL [RR] RRRRRRRR LLL LLLLL

VPN (PPTP) قطع الاتصال

Figure 3. Correct Visual Order.

Logical order: *the application is (NAME, co)*

Segmentation of the string in a similar way:

- *the application is*
- *(NAME, co)*

Processing each of the string and combining the results yields:

the application is (co ,EMAN)

Unicode is gathering feedback for a proposed enhancement for the bidi Algorithm that will help solving this case. The proposed

Example
For simplicity, we will again use lowercase for English and uppercase for Arabic

Logical order: *CANNOT FIND SERVER pop "mail.me.com"*

Visual order by the Bidi Algorithm: *"pop "mail.me.com REVRES DNIF TONNAC*

The correct visual order: *pop "mail.me.com" REVRES DNIF TONNAC*

Directional codes may be used by translators similarly as before, either by adding an LRM after the last quotation mark or by surrounding the fragment (pop "mail.me.com") with a left-to-right embedding code pair (LRE and PDF).

The solution we propose is to first diagnose this case by character-level parsing of the paragraph, segmenting it accordingly, performing the standard Bidi Algorithm on each of the segments, and finally combining them to get the expected result.

Applying this to the example:

Logical order: *CANNOT FIND SERVER pop "mail.me.com"*

After parsing the string and detecting that it belongs to this class, segmentation will produce two strings:

- CANNOT FIND SERVER
- *pop "mail.me.com"*

By applying the standard Bidi Algorithm on each of the strings:

- *REVRES DNIF TONNAC*
- *pop "mail.me.com"*

Combining the results will produce*: pop "mail.me.com" REVRES DNIF TONNAC*

4 URLs ending in '/' embedded in a right-to-left paragraph: This fourth cause occurs when website URLs containing '/' as a last character are embedded in a right-to-left paragraph and not followed by a left-to-right character.

Example:

Logical order: *CANNOT FIND WEBSITE http://www.me.com/mail/*

Visual order produced by the Bidi Algorithm: */http://www.me.com/mail ETISBEW DNIF TONNAC}*

The correct visual order: *http://www.me.com/mail/ ETISBEW DNIF TONNAC*

Translators usually fix this by adding an LRM directional code after the URL in order to enforce a left-to-right direction on the '/'.

The solution we propose is to first detect URLs that meet the conditions of this class, then to segment the paragraph in a way that the URL creates a separate segment, then perform the Bidi Algorithm on each of the segments. Back to the previous example:

Logical order: *CANNOT FIND WEBSITE http://www.me.com/mail/*

After parsing the string and detecting that it belongs to this case, segmentation will produce two strings:

- CANNOT FIND WEBSITE

- http://www.me.com/mail/

By applying the standard Unicode Bidirectional Algorithm on each of the strings:

- ETISBEW DNIF TONNAC

- http://www.me.com/mail/

Combining the strings back together will produce: *http://www.me.com/mail/ ETISBEW DNIF TONNAC*

4. Evaluation

We applied the proposed solutions on strings used in Apple software localised for Arabic. Extracting the text that contains Unicode directional codes helped us in analyzing the cases where the Unicode Bidirectional Algorithm needs help to provide the correct layout. However, the data we used is property of Apple and is not currently available for public use.

We extracted 593 paragraphs that need Unicode directional codes to produce the correct visual order. Many of these cases contain variables that are replaced with real values at runtime due to the fact that the data is used in localised software. Variables can stand for usernames, computer names, server names, number of pictures, minutes, hours, etc.

Having those variables in the paragraphs as they are when evaluating may lead to ambiguity or misinterpretation of the results. Variables usually consist of one or few neutral characters which means that applying the Unicode Bidirectional Algorithm to the paragraph with the variables might hide some cases that cause problems for the layout at runtime, when they are replaced by strong type characters. In addition, some variables are surrounded by Unicode directional codes even though, while looking at the variable itself, it is not apparent to us why it needs directional codes. For these reasons, we replaced the variables with representative data and tried to cover the possibilities for the values that a variable might be replaced with.

Possible replacements included randomly created words in left-to-right characters, right-to-left characters and numbers with neutrals. Also the replacements can have various sizes according to specified ranges. As we are dealing with Arabic localised content, it is most probable that adding left-to-right characters would create more problems.

Case	**Number of Occurrences**
Case 1	254
Case 2	57
Case 3	49
Case 4	13
Total	373

Table 1. Number of paragraphs in each recognized case in the evaluation set.

Afterwards, we classified the paragraphs, each class representing one of the types of case that we are attempting to fix. Recognizing each case and grouping them permits counting how frequent each type of case is; it also helped us detect the cases that did not fall under any of the types we defined and which would require future work. Of the 593 paragraphs, we found 254, 57, 49 and 13 paragraphs of the first, second, third and fourth types respectively: a total of 373, or 62.9%. Table 1 shows the number of paragraphs in each recognized case.

We used the Java reference code provided by Unicode (Unicode 2009) for implementing the original Bidi Algorithm. It processes one paragraph at one run. We ran our tests on the original Unicode Bidi Algorithm using strings containing directional codes that translators had added, and on the modified algorithm using strings with those directional codes removed. If for some string the modified algorithm provides the same embedding levels and ordering in its output, without the help of directional codes, as the original algorithm provides with that help (discarding the directional codes when comparing), that is a success for the modified algorithm.

The results we found were that 254 paragraphs fell under the first case, 57 paragraphs fell under the second case, 49 paragraphs fell under the third case and 13 paragraphs fell under the forth case. Which made a sum of 373 recognized cases. The tests revealed that the modified algorithm succeeded in 322 cases, that is, 86.3% of the 373 recognised as instances of general classes, and 54.3% of the 593 strings.

5. Possible Use of the Modifications to the Unicode Bidirectional Algorithm

The Bidi Algorithm is an established global standard that is widely used for the purpose of displaying bidirectional text. For this reason, changing the standard is a difficult decision to be made. An alternative is to apply the modified version we propose in order to detect the cases of the types listed and add the correct directional codes to the text before processing by the standard Bidi Algorithm. In this way adding directional codes can be done automatically for these cases and reduce the danger of errors caused by manually adding them.

6. Future Work

During our analysis of the data that needs directional codes to provide the correct visual order of characters, we encountered four other productive patterns that possible solutions could readily be provided for. Those cases include:

1 File names written in right-to-left characters: When naming a file using right-to-left characters it will cause problems in layout because file extensions are using latin characters with a period which can be misplaced in layout. Also when displaying file paths problems exist because paths might contain left-to-right folder names or disk names (Microsoft MSDN 2012).

Example

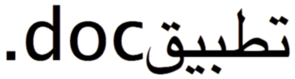
.docتطبيق

Figure 4. Visual order by Unicode Bidirectional Algorithm

Logical order: *NAME.doc*

Visual order by Unicode Bidirectional Algorithm: *.docEMAN*

Correct visual order: doc.EMAN

2 Unit symbols and mathematical signs in right-to-left text: Numbers in right-to-left scripts have a left-to-right flow, however, mathematical signs and other neutral unit symbols should be displayed in a certain layout in those scripts and this usually causes problems.

Example:

Logical order for the string (*-2%*) in a right-to-left script: *-2%*

The visual order by Unicode Bidirectional Algorithm is the same: *-2%*

The correct layout should be: *%2-*

3 Phone numbers in right-to-left scripts: As previously mentioned, numbers in right-to-left scripts have a left-to-right flow, however, the overall layout of the script would be right-to-left which creates problems when formatting phone numbers (or any similar concept of numbers such as part numbers,...etc)

Example:

Logical order: *PHONE NUMBER IS +987 (65) 432 1098*

Visual order by Unicode Bidirectional Algorithm: *1098 432 (65) 987+ SI REBMUN ENOHP*

Correct visual order: *+987 (65) 432 1098 SI REBMUN ENOHP*

4 A list of bidirectional items separated by a neutral: When there is a sequence of bidirectional named items (items named with a mix of left-to-right and right-to-left characters) separated by punctuation or any neutral character, the Bidi Algorithm tends to place same-direction characters together in an embedding level, punctuation and all.

Example:

If there was a list of application names separated by commas where some applications have Arabic names and others have English names.

Logical Order: *CONTACTS, TOOL airport, finder, PREVIEW, safari*

Visual order produced by Bidi Algorithm: *safari ,WEIVERP ,airport, finderLOOT ,STCATNOC*

Correct visual order: *safari ,WEIVERP ,finder ,airport LOOT ,STCATNOC*

This problem is harder to detect, because establishing that a bidirectional text makes a list of items separated by a punctuation mark cannot be done simply. It needs more complicated tests such as parsing the words based on dictionaries or large corpus to get an idea about the text and whether it looks like separate items/names separated by punctuation marks or it is just a regular cohesive text with punctuation marks. However, if separators were a set of punctuation marks it would be easier to detect those lists, because normal text should not usually contain two commas or two dashes adjacent to each other.

Finally, adding these improvements to the Unicode Bidirectional Algorithm will introduce a new challenge. Algorithms that are responsible for text processing and rendering have to be very quick. Performance is a vital part when thinking about those algorithms. Improvements to the algorithm by applying sentence and word segmentation will add more rules and processing, which in turn will decrease the efficiency of the algorithm. We used regular expressions to detect certain patterns. Optimizing the way the regular expressions are used can improve the performance of our improved version of the algorithm. Working on improving the efficiency of the patterns detection techniques is an important avenue for future work.

7. Conclusions

When displaying bidirectional text that contains characters in both left-to-right and right-to-left scripts, the Unicode Bidirectional Algorithm is used as a global standard and a widely used tool to achieve this purpose. It reorders the characters based on the order they were typed in (Logical Order) using several rule-based phases of processing. It displays the characters after they have been reordered in the correct readable way (Visual Order) most of the time. Sometimes, however, it needs the help of invisible directional codes to explicitly force the direction of a character or a string of characters. Analysis of the cases where the Unicode Bidirectional Algorithm fails to show the correct visual layout allowed identification of a few patterns that fit many of these problem cases. Solutions are provided for four of those patterns, and a modified algorithm is found to show the correct visual layout in the majority of those cases. Using as evaluation data 593 strings extracted from Apple Arabic localised software, experiments showed that the modified algorithm corrected the layout of 86.3% percent of the cases that were fitted by patterns, 54.3% of all the cases that frustrated the original Unicode Bidirectional Algorithm.

We also presented problematic patterns that we did not address in our current work. Future work will aim to extend our approach to handle these cases, to find still more patterns and solutions in order to improve the overall behaviour of the Unicode Bidirectional Algorithm.

References

adMob (2010) AdMob Mobile Metrics [online], available: http://metrics.admob.com/ [accessed 15 Aug 2010]

Abdelhadi, A., Mouss, L. H. and Kadri, O. (2011) 'Efficient Algorithms for the Integration of Arabic Language in Mobile Phone', International Journal of Computer and Electrical Engineering Vol.3, No.3, June 2001, pp. 379-383.

Atkin, S. and Stansifer, R. (2004) 'Implementations of Bidirectional Reordering Algorithms', 18th International Unicode Conference, Hong Kong.

Gross, S. (2006) 'Internationalization and Localization of Software', Master Thesis, Eastern Michigan University.

IBM Corporation (2006) 'National Support Guide and Reference', AIX 5L Version 5.3 , Publication No. SC23-4902-03.

Ishida, R. (2003), 'What you need to know about the bidi algorithm and inline markup', ITSC, available: http://www.w3.org/International/articles/inline-bidi-markup/bidi.pdf

Khaddam, I. and Vanderdonckt, J. (2011) 'Flippable User Interfaces for Internationalization', EICS'11, Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems, New York, pp 223-228.

Lanin, A. (2011) 'Additional Requirements for Bidi in HTML', W3C, available: http://www.w3.org/International/docs/html-bidi-requirements/

Microsoft MSDN (2012) 'Formatting Control Characters', available: http://www.microsoft.com/middleeast/msdn/control.aspx\#Samples

Seng J. (2001) 'Internationalisation and Localisation of the Internet', Synthesis (Singapore), available: http://www.i-dns.net/pdf/internationalisation_localisation.pdf

Unicode (2009) 'Reference code implementing Unicode Bidirectional Algorithm' , available: http://www.unicode.org/Public/PROGRAMS/BidiReferenceJava/

Unicode (2012a) 'Bidi Parenthesis Algorithm', available : http://www.unicode.org/review/pri231/

Unicode (2012b) 'The Unicode Standard', available: http://www.unicode.org/versions/Unicode6.0.0/

Unicode (2012c) 'Unicode Bidirectional Algorithm', Unicode Standard Annex #9, Unicode 6.0.0, available : http://www.unicode.org/reports/tr9/

W3C (2007) 'Unicode control vs. markup for bidi Support', available: http://www.w3.org/International/questions/qa-bidi-controls.en.php