Christopher Stoll

Dr. Duan

435 Algorithms

November 28, 2011

Dynamic Programming Algorithm for Name Matching

The objective of this project was to create and implement an algorithm that was capable of providing dynamic name suggestions to users as they fill in fields on a web based form. The algorithm must be able to give reasonably accurate suggestions from a list of possible entries, and it must be able to do so in real time. A dynamic programming algorithm developed by Philip Top, Farid Dowla, and Jim Gansemer[1] was originally selected, but it was slower and more complex than required. Their algorithm was simplified for the desired application; the approach they described for creating the dynamic programming matrix and backtracking was used without any further processing.

The algorithm starts by creating a two dimensional matrix composed of the two names being compared. For each cell in the matrix the two characters, represented by the x and y axes, are systematically compared to determine a cell score. For those cells the algorithm takes the maximum of four different considerations. First, the algorithm considers the score of the diagonally to the upper left in addition to the current characters' matching score. If the two characters are exactly the same they are scored as 1, if they do not match then they are checked against a character matrix to determine their special matching score. If no match or partial match is made then a mismatch penalty is assessed. The second consideration is the value of the cell directly above the current

---

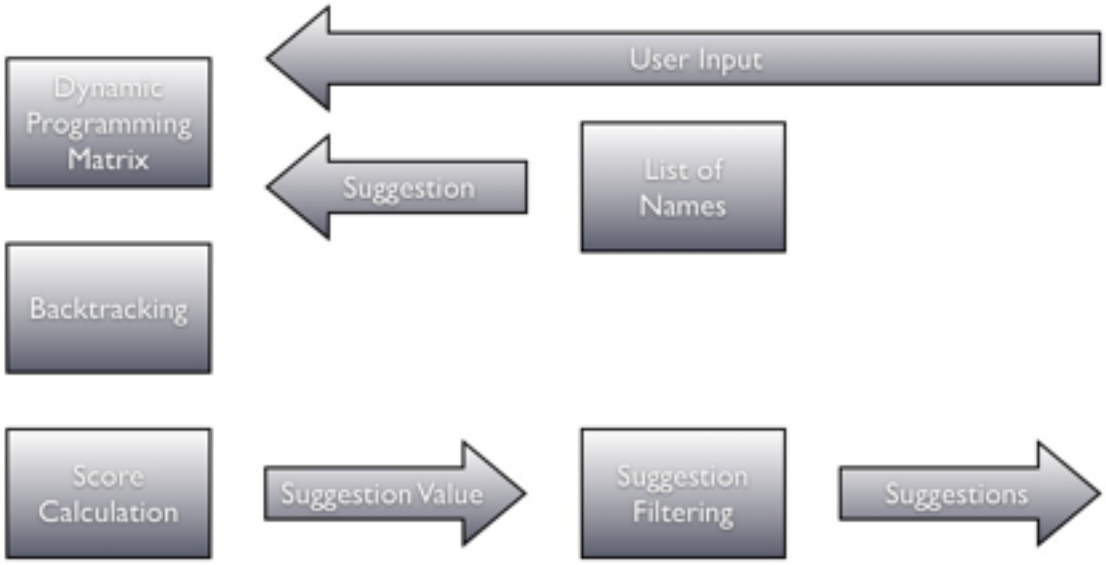
[1] P. Top, F. Dowla, and J. Gansemer, "A Dynamic Programming Algorithm for Name Matching," in *IEEE Symposium on Computational Intelligence and Data Mining, 2007. CIDM 2007*, 547-551.

cell added to the gap score calculation. If there is a gap in one of the names, as occurs between first and last names, then fewer points are reduced than when there is a regular mismatch. The third consideration is the same as the second except that the calculated value is added to the cell to the left rather than above. The final consideration is zero, the matrix should contain no values less than zero. The maximum of these is recorded as the cell's score. The algorithm also forgives users for entering transposed characters.

		r	o	b	e	r	t		a	l	t	o	n	b	e	r	g
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
r	0	1	0.6	0.2	1.2	1	0.6	0.3	0	0	0	0	0	0	0	1	0.6
o	0	0.6	2	1.6	1.2	1	0.6	0.3	0	0	1	1	0.6	0.2	1.2	0.6	0.6
b	0	0.2	1.6	3	2.6	2.2	1.8	1.5	1.1	0.7	1.9	0.8	1.6	1.6	1.2	1	0.6
	0	0	1.3	2.7	2.6	2.3	2	2.8	2.5	2.2	1.9	1.7	1.4	1.4	1.2	0.9	0.6
a	0	0	0.9	2.3	2.3	2.2	1.9	2.5	3.8	3.4	3	2.6	2.2	1.8	1.4	1	0.6
l	0	0	0.5	1.9	1.9	1.9	2.9	3.4	3.4	4.8	4.4	4	3.6	3.2	2.8	2.4	2
t	0	0	1.5	1.5	1.5	1.5	2.9	2.7	3.2	4.4	5.8	5.4	5	4.6	4.2	3.8	3.4
o	0	0	1	1.3	1.1	1.1	2.5	2.5	2.8	4	5.4	6.8	6.4	6	5.6	5.2	4.8
n	0	0	0.6	0.9	0.9	0.7	2.1	2.2	2.4	3.6	5	6.4	7.8	7.4	7	6.6	6.2
s	0	0	0.2	0.5	0.5	0.5	1.7	1.9	2	3.2	4.6	6	7.4	7.4	7	6.6	6.2

An example matrix created by this algorithm.

Next is where the current algorithm diverges from that of Top, Dowla, and Gansemer. The algorithm looks at the bottom of the matrix along the x axis in order to find the maximum in that vector. If no maximum is found then all of the values are assumed to be zero and the target string is assumed to be too poor of a fit to consider as a suggested match. If, on the other hand, a maximum is found, then backtracking is performed. The backtracking is performed to determine the maximum value in the matrix for the backtrack between the two names. The maximum value is multiplied by two and then divided by the average length of the two names being compared, this is in turn multiplied by ten to give a final matching score.



The algorithm is not very useful unless it is iterating over a list of suggested names which need to be compared for suitability. In the present implementation the algorithm is implemented as a JavaScript class which is instantiated in a web page. The logic in the web page iterates over a name list and feeds those suggestions into the algorithm. The algorithm gives scores for each of the suggestions and the page sorts them, presenting the user with the top ten possibilities.

In order to create the dynamic matrix the algorithm must iterate over each character in each string. If one string has a length of n and the other m, then this takes (n * m) time to complete. The backtracking portion of the algorithm begins by finding the maximum value on the bottom row of the matrix, which takes (n) time. After finding the maximum the backtracking method moves back along the path of likely coincidence, which takes at most (n+m) time. The score is calculated in constant time, so the algorithm works in ((n*m) + n + (n+m)) time, which simplifies to (n * m). The algorithm must be run for each item in the name list, if there are i items in the list then the algorithm takes (i * (n * m)) time to run. For the sample data used in this project the average name

length is 13 (the minimum name length is 7 and the maximum is 29), so the algorithm would run in (i * (13 * 13) or (169 * i) time. The most important factor for this algorithm is the i, the number of items to check against.

Based upon the theoretical analysis the time to run the algorithm over a given data set should grow linearly with changes in the size of the data set or sizes of the average name length. Empirical analysis, as shown below, confirms this.

String tested (“n”)	Time to evaluate all 2,256 suggestions
abcde	122 ms
abcdefghij	229 ms
abcdefghijklmno	316 ms
abcdefghijklmnopqrst	396 ms
abcdefghijklmnopqrstuvwxy	474 ms

Effects of “n” on the running time. (average of 10 runs)

Suggestion Array Length	Time to evaluate “abcde fghij klmno pqrst uvwxy z”
141	36 ms
282	72 ms
564	147 ms
1,128	295 ms
2,256	588 ms

Effects of “i” on the running time. (average 10 runs)

For the empirical analysis a list of employee names from a local company was mixed with an equal sized list of notable people from Ohio. Fictitious names were mixed in to pad the list to a length of 2,256. Data from the local company was used to check the algorithm’s performance against known real-world misspellings as well as know, problem-

atic nickname to real name variation. The list of suggested names was cleaned of periods (such as after “Jr.”), but additional processing could be done to further improve performance (converting the strings to lower case).

This project demonstrates that it is practical to implement a dynamic programing algorithm for name matching within a web page where real time responses are required. As is evident in the supplied demonstration file, the performance is both fast and reliable. Use of this algorithm could increase the efficiency of many areas of an enterprise in which it was introduced. The list of employee names could be sent to the client and cached along with the algorithm class, this would reduce network usage as well as server usage on both the web and database servers.