**EcoDriving for Electric Vehicles**

Diogo Reis e Tiago Magalhães

Faculdade de Engenharia da Universidade do Porto

Email: {up201505472, up201607931}@fe.up.pt

**KEYWORDS**

Electric Cars, Environment friendly,

String matching, exact string matching, approximate string matching, Knuth-Morris-Pratt Algorithm, Dynamic programming matrix.

**ABSTRACT**

Our work seeks to create a trip planning system, that considers an electric car needs this being the existence of points to recharge the cars battery. In this project, we string matching algorithms, for both exact matching and approximate matching with the intent of filtering through the recharge points of a given network.

**INTRODUCTION**

This project attempts to address the issues of string matching as part of user queries for points in a directed graph network that have an electric car recharge station, considering that the points in the graph with recharge stations are generated randomly. The queries can be of 3 different types, listing all the recharge stations in the network, searching for a given work in the names of the stations or approximately searching for a word in the names of the recharge stations.

**STUDY’S GOAL**

The objective of this project is to create a system through which we learn to with exact string matching and approximate string matching.

**MAIN DEFINITIONS**

* **String Matching**
* **Approximate String Matching**
* **Exact String Matching**

**Implementation**

1. **Input –** The input to this system the type of string matching we want to do; listing all recharge points, approximate string matching or exact string matching.
2. **Output –** The output will be the list of recharge points whose names fit the pattern provided in the input and whose edit distance to the pattern is under 3 in the case of approximate string matching.

**String Matching**

In this section, we will define the two main types of string matching we will explore in this project and the concept of string matching in the context of this project and in general.

**Concept:** String matching is the action in which we take a string of text to be analyzed and a string of text that makes up our pattern. Next, we will look for the pattern in our given string of text, now there are many ways to do this, but there are only 2 types of string matching that we are considering in this project are exact matching and partial or approximate matching.

**Exact Matching**: In exact string matching we want to find the pattern in its entirety in the text string that we are given, it doesn’t matter if it is in the middle of words of between words as long as we find the whole pattern somewhere in the string, so in this method we merely say whether we found the pattern or not it’s a binary decision. For this type of string matching we will be using the Knuth–Morris–Pratt algorithm.

**Approximate Matching:** In approximate string matching we want to find as much as possible of the pattern in the string, so for this method to work we need to search for the pattern in only words and not the whole string as the pattern might change mid word. We will also have to decompose the pattern into its constituent words and search for each of those words in the words of the text string.

**Knuth–Morris–Pratt Algorithm**

This algorithm is directed at exact string matching and partial string matching here we explain the exact string matching procedure.

The first thing we need to do is pre-process the pattern in order to increase the efficiency of this algorithm, for this we will give a order number to each character of the pattern and we create an array with the same number of elements as the number of characters in the string, next w iterate through the pattern with two variables w starting at position 0 and j at position 1, every loop we will compare element w with element j and increment j.

If element w and j are equal, element j in the new array will be w +1, next we increment w. if element w and j in the pattern are not equal and w is not 0 we equal w to its value previous element in the new array of integers, and compare with the same rules. This procedure is repeated until all element of the new array have been filled out. This Procedure has a time and space complexity of O(n).

We will now use the preprocessed pattern to identify the occurrent of the pattern.

Now we analyses the string of text with the pattern, first preprocessing the pattern to obtain the prefix array. Then we start comparing each element in the text with the pattern starting at the first point, every time a comparison fails we check the characters in which we failed in the pattern in the prefix array and start the comparison in the value of that element of the prefix array, say we failed in a c in the pattern and the corresponding prefix array value was a 2 , that means we start comparing the text with the pattern from the point we failed with the second character of the pattern. If we manage to make out way through the whole pattern string we know that the pattern is present in the text string.

This section of the algorithm has O(m) time and space complexity so the algorithm in a whole has O(m+n) time complexity and O(n) space complexity where m is the length of the text string and n the length of the pattern string.

Pseudo-Code for this algorithm:

**algorithm** *kmp\_table*:

**input**:

an array of characters, W (the word to be analyzed)

an array of integers, T (the table to be filled)

**output**:

nothing (but during operation, it populates the table)

**define variables**:

an integer, pos ← 1 (the current position we are computing in T)

an integer, cnd ← 0 (the zero-based index in W of the next character of the current candidate substring)

**let** T[0] ← -1

**while** pos < length(W) do

**if** W[pos] = W[cnd] **then**

**let** T[pos] ← T[cnd], pos ← pos + 1, cnd ← cnd + 1

**else**

**let** T[pos] ← cnd

**let** cnd ← T[cnd] (to increase performance)

**while** cnd >= 0 and W[pos] <> W[cnd] do

**let** cnd ← T[cnd]

**let** pos ← pos + 1, cnd ← cnd + 1

**let** T[pos] ← cnd (only need when all word occurrences searched)

**Approximate String Matching**

For approximate string matching we must first define the concept of edit distance.

The edit distance between two strings is the number of substitution, deletions or insertions that we need to execute on one string for it to be equal to the other.

To calculate the edit distance between two strings we first set up a matrix with m columns and n columns, where m is the length of the first string +1, and n the length of the second string +1.

Where element 0,0 has the labels of empty string(Epsilon), and all other elements are labeled by their corresponding element in the strings.

Then for each member of the matrix its edit distance value will the minimum between the edit distance element above and to the left plus 1 of the labels are different, the element to the left +1, and the element on top +1.

After completing the matrix the edit distance between the two strings will be located in the last element in the matrix, element [m,n].

This implementation presents a time complexity and a space complexity of O(m\*n). However this implementation cannot be directly applied as it will compare the two string too literally and will give as a much larger edit distance than required , so what we do is we split the pattern and the text string into its composing words and compare those obtaining the minimum edit distance for each of the words that make up the pattern returning the sum of the minimum edit distances.

This guarantees that if a string exists in its entirety in a text string the edit distance will be 0.

The time complexity for this algorithm is O((m\*n)^2) the space complexity is still O(m\*n).

**Program Logic Flow**

This version of the software maintains the same overall logic flow but now calculates a map containing only the Location which are recharge points and introduces a menu to choose between route calculation and queries about recharge points.

The query menu can list all recharge points ; it simply goes over the map with all the recharge points; when we select approximate search is asks us for the pattern and the applies the approximate matching algorithm described above and prints recharge points with edit distance lower than 3; lastly exact string matching asks for the pattern and applies the KMP algorithm printing recharge points that match the pattern.

**Conclusion and Future Work**

After this work, we were finally able to an obvious benefit of dynamic programming, and we understood a bit better the mechanisms behind things like CNTRL+F and search algorithms.

Some possible further improvements to the software is modify it to be compatible with UTF-8 encoding so as to be able to use special characters like Ç in the search parameters, something we avoid at all costs here as we are using single byte encoding for our characters, another improvement is having the software convert everything to lower case when comparing it, avoiding lower and upper case conflicts leading to not identifying strings that should be identified.

**References**

<https://en.wikipedia.org/wiki/Edit_distance>

<https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm>

All the slides available in moodle.

**Contribution**

Diogo Reis – 50%

Tiago Magalhães – 50%