Implementacja odległości Levenshteina w języku JavaScript

Jakub Piśkiewicz

25 listopada 2021

1 Terminologia

W tym tekście użyto pewnych terminów, które mogą wymagać krótkiego objaśnienia.

- słowo - ciąg zera lub więcej znaków. Nie musi mieć żadnego sensu.
- string - w języku angielskim oznacza ciąg znaków. Nazwa bardzo często używana w językach programistycznych.

2 Różne typy miar odległości między słowami

Odległość między słowami to najmniejsza możliwa liczba operacji wymaganych aby przejść z jednego słowa w drugie. Odległość Levenshteina jest jednym czterech typów pomiarów odległości między słowami. Typy pomiarów odległości różnią się między sobą operacjami na słowach ze sobą porównywanych. [2] Dla odległości Levenshteina operacje proste na słowach to: **wstawienie** nowego znaku do słowa, **usunięcie** znaku ze słowa lub **zamiana** znaku w słowie na inny. [4]

3 Opis miary odległości Levenshteina

3.1 Operacje proste na słowach i ich przykłady

Tabela 1 przedstawia wszystkie operacje proste według miary odległości Levenshteina i przykład ich działania na słowie "helikopter".

nazwa operacji	przykład działania
wstawianie	helikopter → helikoptera
usuwanie	helikopter → helikopte
zamiana	helikopter → helikoptur

Tabela 1 - operacje proste i ich przykłady.

Działanie operacji prostych można także przedstawić w poniższy sposób na przykładzie słowa $A = uv$. Znak ε oznacza pusty string. [2]

- wstawianie - $A = uxw,\ \varepsilon \to x$
- usuwanie - $A = uv,\ x \to \varepsilon$
- zamiana - $A = vv,\ u \to v$

3.2 Koszty operacji na słowach

W wielu odmianach miar odległości między słowami każda z operacji elementarnych ma swój koszt. W przypadku odległości Levenshteina koszty operacji wstawiania i usuwania jednego znaku wynoszą 1. Koszt zamiany znaku na taki sam wynosi 0, natomiast koszt zamiany na znak inny wynosi 1.

3.3 Przykład

Odległość Levenshteina między słowami $A =$ ”silnik” i $B =$ ”wirnik” wynosi 2. W celu porównania obu słów należy wykonać następujące transformacje:

1. **s**ilnik → **w**ilnik - koszt operacji zamiany wynosi 1 (s → w).
2. s**i**lnik → w**i**rnik - koszt zamiany i na i jest zerowy.
3. wi**l**nik → wi**r**nik - koszt zamiany ($l \to r$) jest równy 1, wobec czego odległość wzrasta do 2.
4. wir**n**ik → wir**n**ik - koszt zamiany n na n jest zerowy, więc odległość nie zmienia się.
5. wirn**i**k → wirn**i**k - koszt operacji = 0. Odległość bez zmian.
6. wirni**k** → wirni**k** - odległość między słowami = 2.

Ostatecznie odległość między tymi słowami wyniosła 2, ponieważ jedynymi operacjami prostymi, których koszt > 0 wymaganymi do przejścia ze słowa A w słowo B były dwie operacje zamiany: s → w i l → r.

Ciekawostka - odległość Hamminga jest możliwa do obliczenia dla tych dwóch słów, ponieważ są one identycznej długości. Wyniosła by ona tyle samo co odległość Levenshteina, ponieważ jedynym typem operacji potrzebnym do przejścia z jednego słowa w drugie jest zamiana. Zamiana jest jedynym typem operacji dozwolonym w odległości Hamminga. W przypadku gdy słowa są takiej samej długości górną granicą ich odległości w metryce Levenshteina jest ich wzajemna odległość według metryki Hamminga.

3.4 Właściwości

Każda miara odległości z kosztami operacji > 0 i możliwością cofnięcia każdej operacji za tym samym kosztem (na przykład koszt usunięcia znaku jest taki sam jak koszt wstawienia tego znaku) może być uznawana za metrykę pod warunkiem spełnienia następujących warunków:

- Odległość między tymi samymi słowami wynosi 0.
- Odległość między różnymi od siebie słowami jest wyższa od 0, ponieważ co najmniej jedna operacja o koszcie większym od 0 potrzebna jest do przyrównania tych słów do siebie.
- Dla trzech słów: A, B oraz C: $d(A, C) \leq d(A, B) + d(B, C)$. Gdzie $d(x, y)$ oznacza odległość międzdy słowami x i y. Czyli zachodzi nierówność trójkątna.

Taka metryka tworzy wraz ze słowami dla których określa ona odległość - przestrzeń metryczną. Przestrzeń metryczna to zbiór z zadaną na nim funkcją określającą odległość między jego elementami. [5]

Odległość Levenshteina spełnia te aksjomaty wobec czego jest to metryka.

3.5 Definicja

Aby stworzyć algorytm wykorzystujący metrykę odległości Levenshteina należy spojrzeć na jej definicję. [3]

$$lev(a,b) = \begin{cases} |a| & \text{jeżeli } |b| = 0, \\ |b| & \text{jeżeli } |a| = 0, \\ lev(tail(a), tail(b)) & \text{jeżeli } a[0] = b[0] \\ 1 + min \begin{cases} lev(tail(a), b) \\ lev(a, tail(b)) \\ lev(tail(a), tail(b)) \end{cases} & \text{w innym wypadku} \end{cases}$$

$|x|$ oznacza długość słowa x. $x[i]$ oznacza i-ty znak w słowie x.

W definicji znajduje się kilka kluczowych informacji o działaniu algorytmu:

- Jeżeli jedno ze słów zawiera zero znaków. Odległością Levenshteina jest długość drugiej sekwencji.
- W przypadku gdy pierwszy znak w obydwu słowach jest taki sam, funkcja $lev(a, b)$ jest wywoływana rekursywnie dla obydwu słów bez znaku pierwszego.
- Jeżeli długości słów są większe od 0 i pierwszy znak nie jest taki sam, wynik przyjmuje wartość najmniejszą spośród rekursywnych wywołań pomiaru odległości Levenshteina odpowiednio dla operacji wstawiania - $lev(tail(a), b)$, usuwania - $lev(a, tail(b))$ i zamiany - $lev(tail(a), tail(b))$.

4 Algorytm

4.1 Naiwna rekursywna implementacja

Na podstawie definicji odległości Levenshteina można stworzyć algorytm, który przy użyciu programowania rekursywnego znajduje odległość między dwoma ciągami znaków.

Oto implementacja tego algorytmu w języku JavaScript.

```
/* Funkcja porównująca długości dwóch słów metodą Levenshteina.
 * To jest implementacja rekursywna - bezpośrednie przełożenie
 * definicji odległości Leveshteina na algorytm. */
compareDistance(st1, st2) {
  // Odległośc między słowami (d - distance).
  let d = 0;

  /* Jeżeli ciąg pierwszy jest pusty,
   * to odległość między słowami wynosi
   * długość słowa drugiego. */
  if (st1.length === 0) {return st2.length}
  /* Jeżeli ciąg drugi jest pusty,
  * to odległość między słowami wynosi
  * długość słowa pierwszego. */
  if (st2.length === 0) {return st1.length}

  /* Jeżeli obydwa słowa zaczynają się od tej samej litery
   * funkcja jest wywoływana rekursywnie dla tych słów
   * bez pierwszego znaku. */
  if (st1[0] === st2[0]) {
    d = this.compareDistance(st1.substring(1), st2.substring(1));
  } else {
    d = 1 + Math.min(                              // Wybierany jest najniższy koszt spośród:
      this.compareDistance(st1.substring(1), st2), // Wstawianie
      this.compareDistance(st1, st2.substring(1)), // Usuwanie
      this.compareDistance(st1.substring(1), st2.substring(1)) // Zamiana
    );
  }

  return d;
}
```

Działanie tego algorytmu kończy się, gdy długość jednego lub obydwu słów spadnie do 0. [1]

Niestety ta implementacja nie jest zbyt wydajna. W związku z tym, że ten algorytm tworzy "drzewko decyzji" i oblicza wynik dla każdego ciągu znaków który znajduje się w tym drzewku, wykonuje on bardzo dużo niepotrzebnych operacji, co zwiększa jego złożoność czasową.

Druga implementacja pokazuje, że ten algorytm da sie zoptymalizować przy użyciu programowania dynamicznego.

4.2 Implementacja wykorzystująca programowanie dynamiczne

Programowanie dynamiczne polega na rozbiciu większego problemu na mniejsze "podproblemy" i stopniowe rozwiązywanie tych "podbroblemów" aż główny problem stanie się na tyle trywialny, że jego rozwiązanie będzie wystarczająco proste.

Poniżej znajduje się program wykorzystujący programowanie dynamiczne w celu znalezienia odległości Levenshteina.

```
// Funkcja, która oblicza długość między dwoma ciągami znaków.
compareDistance( ) {
  const st1 = this.state.string1;
  const st2 = this.state.string2;

  /* Jeżeli ciąg pierwszy jest pusty,
   * to odległość między słowami wynosi
   * długość słowa drugiego. */
  if (!st1.length) {
    this.setState.result({result: st2.length, resultTable: null})
  };
  /* Jeżeli ciąg drugi jest pusty,
   * to odległość między słowami wynosi
   * długość słowa pierwszego. */
  if (!st2.length) {
    this.setState.result({result: st1.length, resultTable: null})
  };

  /* Matryca reprezentująca odległości między coraz to większymi
   * ciągami znaków. */
  let minCostMatrix = new Array(st1.length);
  for (let i = 0; i < minCostMatrix.length; i++) {
    minCostMatrix[i] = new Array(st2.length).fill(0);
  }

  // Odległość początkowa - koszt zamiany ostatniego znaku w obu słowach.
  minCostMatrix[st1.length - 1][st2.length - 1] = this.findCost(st1, st2, st1.length - 1, st2.length - 1);

  for (let j = st2.length - 2; j >= 0; j--) {
    minCostMatrix[st1.length - 1][j] = 1 + minCostMatrix[st1.length - 1][j + 1];
  }

  for (let i = st1.length - 2; i >= 0; i--) {
    minCostMatrix[i][st2.length - 1] = 1 + minCostMatrix[i + 1][st2.length - 1];
  }

  for (let i = st1.length - 2; i >= 0; i--) {
    for (let j = st2.length - 2; j >= 0; j--) {
      /* Koszt zamiany znaków znajdujących się na skrzyżowaniu
       * indeksów i oraz j + koszt zamiany wszystkich poprzednich
       * znaków aż do tego momentu. */
      let replace = this.findCost(st1, st2, i, j) + minCostMatrix[i + 1][j + 1];
      // Koszt usunięcia jednego znaku.
      let del = 1 + minCostMatrix[i + 1][j];
      // Koszt wstawienia jednego znaku.
      let ins = 1 + minCostMatrix[i][j + 1];
      // Zapisanie kosztu minimalnego w matrycy.
      minCostMatrix[i][j] = Math.min(replace, del, ins);
    }
  }

  this.setState({
    result: minCostMatrix[0][0],
    resultTable: this.compileResultTable(minCostMatrix),
  });
}
```

Algorytm tworzy "matrycę" zawierającą najniższe koszty wybrane z pośród trzech operacji. Program, który stworzyłem przedstawia te koszty w formie tabeli.

Obydwa programy zamieściłem jako strony internetowe GitHub pages. Można dzięki nim porównać prędkość obliczania odległości Levenshteina w implementacji rekursywnej i programowania dynamicznego.

Literatura

[1] Jose Luis Ordiales Coscia. Easy to understand dynamic programming – edit distance, 2014.

[2] Wikipedia. Edit distance — Wikipedia, the free encyclopedia, 2021.

[3] Wikipedia. Levenshtein distance — Wikipedia, the free encyclopedia, 2021.

[4] Wikipedia. Odległość levenshteina — Wikipedia, wolna encyklopedia, 2021.

[5] Wikipedia. Przestrzeń metryczna — Wikipedia, wolna encyklopedia, 2021.