

Приблизително търсене в текст

Андрей Дренски, 2018г.

Approximate string matching

Андрей Дренски, 2018г.

...какво значи „приблизително“?

- има няколко типа най-често срещани „грешки“ :
 - изтриване на един символ (deletion)
 - вмъкване на един символ (insertion)
 - размяна на два съседни символа (transposition)
 - заместване на символ с друг (substitution)
- това са по-скоро „технически“ грешки
- смисловите са в съвсем различна категория
 - но все пак можем да използваме същата идея

...какво значи „приблизително търсене“?

- Нека са дадени:
 - текст, в който да търсим (*text*) с дължина n
 - текст, който търсим (*pattern*) с дължина m (най-често $m \ll n$)
 - число k - допустим брой грешки (най-често $k \ll n$)
- Проблем:
 - да се намерят всички позиции в *text*, където се среща стринга *pattern*, но с най-много k грешки
- Пример: *pattern*="abc", *text*="хyzacsfjdklbd", $k = 2$

...какво значи „текст“?

- съществуват много приложения на този клас алгоритми:
 - търсене за обикновени текстове с правописни грешки
 - обработка на сигнали с евентуални загуби
 - търсене на ДНК последователности
 - търсене в речници
 - търсене от по-високо ниво, където цели думи са елементи от азбуката
- всяка от сферите на приложение има собствен модел за грешки и „близост“ между стрингове

...какво значи „близост на стрингове“?

Разстояние на Левенщайн (Levenshtein distance)

- нека имаме стринговете p и t с дължини m и n
- ще изчислим разстоянието като брой поправки, необходими за достигането на единия стринг от другия
- тези поправки ще съответстват на елементарните грешки: вмъкване, изтриване, субституция и транспозиция
- ще мемоизираме разстоянието между всеки префикс на p и всеки префикс на t

...какво значи „близост на стрингове“?

Разстояние на Левенщайн (Levenshtein distance)

- $C(i, j)$:= разстоянието между префикса на p с дължина i и префикса на t с дължина j
- $C(0, j) = j$ за всяко $j = \{0..n\}$
- $C(i, 0) = i$ за всяко $i = \{0..m\}$
- $C(i, j) = \min \{ C(i - 1, j) + 1, C(i, j - 1) + 1, C(i - 1, j - 1) + \delta(p_i, t_j) \}$
 - изтриване от p
 - вмъкване в p
 - субституция

...какво значи „близост на стрингове“?

Разстояние на Левенщайн (Levenshtein distance)

- $d(p_i, t_j) = 0$, ако $p_i = t_j$ и 1 иначе
- за транспозиции можем да включим и $C(i - 2, j - 2) + 1$
 - но само когато $p_{i-1} = t_j$ и $p_i = t_{j-1}$
- цените на операциите най-често са единични (може и да не са)
- можем да добавяме и премахваме правила
 - например за цели подстрингове
- реално това е алгоритъм за най-къс път в DAG

демо

...какво значи „близост на стрингове“?

Разстояние на Левенщайн (Levenshtein distance)

- цените на различните операции могат да са различни
 - например, изтриване на 'a' може да е по-скъпо от изтриване на 'b'
- повечето свойства обаче разчитат на единични цени на всички
- разстоянието на Левенщайн е метрика:
 - $d(x, y) \geq 0$ за $\forall x, y$
 - $d(x, y) = 0 \Leftrightarrow x = y$
 - $d(x, y) = d(y, x)$
 - $d(x, y) + d(y, z) \geq d(x, z)$ - неравенство на триъгълника

...какво значи „близост на стрингове“?

Разстояние на Левенщайн (Levenshtein distance)

- $d(x, y) \geq \text{abs}(|x| - |y|)$
 - разстоянието е поне разликата в дължините на двете думи
- $d(x, y) \leq \max\{|x|, |y|\}$
 - разстоянието е най-много колкото дължината на по-дългата дума

За транспозициите

Какво е разстоянието между “ca” и “abc”?

- наивно: “ca” \rightarrow “ac” \rightarrow “abc” с 2 операции
- гореописаният алгоритъм връща отговор 3:
 - “ca” \rightarrow “a” \rightarrow “ab” \rightarrow “abc”
 - “ca” \rightarrow “cc” \rightarrow “bc” \rightarrow “abc”
 - и други варианти
- Откъде идва разликата? Нима алгоритъмът не е коректен?

За транспозициите

Какво е разстоянието между “ca” и “abc”?

- има два начина за изчисляване на разстояние между стрингове, когато допускаме транспозиции:
 - Optimal String Alignment (OSA): не позволява даден подстринг да бъде редактиран повече от веднъж
 - Damerau-Levenshtein distance (DL): няма това ограничение
- $OSA("ca", "abc") = 3$
- $DL("ca", "abc") = 2$
- алгоритъмът по-горе изчислява OSA
- DL е малко по-трудна за изчисляване

За транспозициите

Какво е разстоянието между “ca” и “abc”?

- OSA дори не е метрика (!)
- $OSA(\text{“ca”}, \text{“ac”}) + OSA(\text{“ac”}, \text{“abc”}) < OSA(\text{“ca”}, \text{“abc”})$
 - не изпълнява правилото на триъгълника
- DL, от друга страна, е метрика

... толкова за разстоянията между стрингове

Приблизително търсене в текст

ТОЗИ ПЪТ НАИСТИНА

Класически алгоритъм [DP]

- подобен на алгоритъма за Levenshtein Distance
- за всеки индекс i в **pattern** и всеки индекс j в **text** ще мемоизираме минималния брой поправки, необходим за достигане от префикса $p_1..p_i$ до някой подстринг $t_{\text{start}}t_j$
- тънката разлика:
 - Levenshtein търси **pattern** в целия стринг **text**
 - тук търсим **pattern** *някъде* в **text** (няма смисъл да ги сравняваме изцяло)
- търсим за кои индекси j в **text** има подстринг на **text**, завършващ на този индекс, който може да се достигне от **pattern** с минимален брой поправки

Класически алгоритъм [DP]

- $C(0, j) = 0$ за всяко $j = \{0..n\}$ - можем да започнем търсенето от всяко j
- $C(i, 0) = i$ за всяко $i = \{0..m\}$
- $C(i, j) = \min \{ C(i - 1, j) + 1, \quad - \text{изтриване от pattern}$
 $C(i, j - 1) + 1, \quad - \text{вмъкване в pattern}$
 $C(i - 1, j - 1) + d(p_i, t_j) \} \quad - \text{субституция}$
- резултатът от търсенето са тези индекси j , за които $C(m, j)$ е минимално
- от всеки такъв индекс j можем да възстановим процеса на редактиране
- както разстоянието на Levenshtein, може да има огромен брой „пътища“
- транспозиции и различни цени се добавят аналогично на Levenshtein

Класически алгоритъм [DP]

- параметрите на задачата са m, n и k
 - k най-често е константа, но понякога участва в записа на сложностите
- времето за работа на класическия алгоритъм е $T(m, n) = O(mn)$
- можем да пазим само последователни редове или колони на таблицата
 - следователно нужната памет е само $O(\min\{m, n\})$
- ако искаме да възстановим процеса на редакция, трябва да запазим в паметта цялата таблица $\rightarrow M(m, n) = O(mn)$
 - не! понякога възстановяването се извършва отделно от търсенето, за $O(rm(k+m))$, където r е броя намерени съвпадения

Класически алгоритъм [DP]

- този алгоритъм е най-прост, но и най-гъвкав
 - лесно се добавят по-сложни операции
 - допуска различни цена на операциите
 - дава възможност за възстановяване на операциите
- ... голяма част от алгоритмите за този проблем се базират на него

демо

Ukkonen's Cut-off heuristic [CTF]

- идея: ако клетка в таблицата достигне $k+1$, то резултатът не зависи от нея
 - това става бързо (заради $k \ll m$ в общия случай)
 - ще казваме, че тя се превръща в неактивна
- ще попълваме всяка колона i до последната ѝ активна клетка a_i
- за всяка колона имаме, че стойността в $a_i \leq a_{i-1} + 1$
 - (не се увеличава с повече от 1)
- възможно е някоя активна да стане неактивна
 - тогава трябва да намерим последната активна \rightarrow това търсене става амортизирано.

Ukkonen's Cut-off heuristic [CTF]

- тогава времето за алгоритъма е $T(m,n) = O(kn)$ *средно*
 - доказателството е дошло доста след публикуването на алгоритъма
- отново сложността по памет е $O(\min\{m,n\})$
- възстановяването на операциите е не по-лесно от класическия [DP]
- промяната на цените на елементарните операции е в най-добрия случай трудна...

Diagonal Transitions [DTR]

- нека под D-диагонал d разбираме тези клетки в таблицата, за които $j-i=d$ (ред i , стълб j)
- наблюдение: в рамките на един D-диагонал стойностите не намаляват
 - а когато се увеличават, е с точно 1
- идея: за всеки диагонал пазим само индексите на тези колони, където стойността в клетката се е увеличила спрямо предишната
- ще мемоизираме $D(e,d) = \text{макс. колона } j \text{ такава, че } C(j-d,j)=e$
 - тук индексите e ще са между 0 и k , вкл.
 - d са всички възможни диагонали (заедно с няколко „служебни“) $\rightarrow n-m+k-2$ на брой

Diagonal Transitions [DTR]

- времето за работа вече със сигурност е $T(n,m)=O(kn)$ в най-лошия случай
- необходимата памет е даже $M(m,n) = O(k)$
 - тази таблица D също може да се попълва стълб по стълб
- недостатък: алгоритъмът разчита на единичните цени на операциите
 - модификации са в най-добрия случай (???)

Bit-parallelized matrix [BPM]

- наблюдение: в таблицата, генерирана от класическия алгоритъм, стойностите във всеки две съседни клетки се различават с най-много 1
- идея: нека закодираме тези разлики с битови вектори
 - можем да използваме само 7 битови вектора с дължина m
- успява да моделира прехода от една колона на матрицата към друга с <20 операции над тези битови вектори
 - съотв. на логическите конюнкция, дизюнкция, отрицание и изключващо или
 - също bitshift и $+$ \rightarrow за това малко по-късно
- този път зависим и от големината на азбуката b

Bit-parallelized matrix [BPM]

- за всеки символ от азбуката пазим по един битов вектор с дължина m , който показва на кои позиции в `pattern` се среща този символ
- след това симулираме работата на класическия алгоритъм:
 - на всяка итерация (т.е. за всяка колона) вместо да правим m на брой операции над стойностите в съответната колона, правим константен брой операции за тези битови вектори
- разчитаме, че операциите над битови вектори са константни
 - те най-често са!

Bit-parallelized matrix [BPM]

- сложност по време: $T(m,n) = O(bm+n)$
- сложност по памет: $M(m,n) = O(bm)$
- на практика b е константа, но алгоритъмът разчита на бързи операции с битови вектори \rightarrow това ограничава m до 64
- според параметрите получаваме различни сложности:
 - в най-добрия случай $T(m,n) = O(n)$ и $M(m,n) = O(1)$
 - в най-лошия имаме $T(m,n) = O(bm+nm)$ и $M(m,n) = O(bm)$
- освен гореспоменатите недостатъци: невъзможност за каквато и да е модификация над алгоритъма

Алгоритми с автомати

презентацията свършва тук