Trie-join: hatékony trie-fa alapú módszerek string hasonlósági összekapcsolásra

Tanulmány Jianhua Feng, Jiannan Wang, Guoliang Li: Trie-join: a trie-based method for efficient string similarity joins cikke alapján.
Adatbázisrendszerek elméleti alapjai – ELTE-IK, 2012

Fazekas Róbert (FAROAAI.ELTE)
Maczika Száva (MASRAAI.ELTE)
Rákos Rudolf (RARRABI.ELTE)

Absztrakt

Ennek a tanulmánynak a célja, hogy segítse a Feng, Wang és Li által írt angol nyelvű tanulmányt [1] amely a Trie-fa alapú string összehasonlításokkal foglalkozik.
Egy string hasonlósági összekapcsolás megtalálja a hasonló párokat két stringekből álló halmazban. Ez egy nagyon sokszor alkalmazott módszer – gondoljunk csak adatbázisokban a szelekcióra – melynek sajnos nincsen igazán hatékony megoldása.
Az eddigi szültetett megoldások, mind az úgynevezett *„szűr és pontosít”* megoldást használják, amellyel a következő problémák vannak:

- nem hatékonyak rövid string halmazok összekapcsolására (ahol az átlagos hossz <30)
- Nagyméretű indexek szükségesek
- Csak nagy költségek árán képesek az eredeti halmazok megváltozásának dinamikus követésére.

Ezeknek a hibáknak a kijavítása érdekében az eredeti tanulmány javasol néhány Trie-fa alapú megoldást, amely képes kis átlaghosszúságú halmazon is hatékonyan működni. A szerzők felhasználják a trie adatstruktúrát indexek készítésére illetve a hasonló stringek hatékony megtalálására, javasolnak algoritmusokat a hatékony összekapcsolásra illetve rész-fa vágásokra is. A cikkben tárgyalt algoritmusok könnyen módosíthatóak, hogy az eredeti halmazok megváltozásához dinamikusan alkalmazkodjanak, a szerzők számos kísérletet futtattak melyek megmutatták, hogy az általuk javasolt módszerek nagyságrendekkel jobban teljesítenek a legújabb algoritmusoknál is, rövid stringek esetén.

Bevezetés

A hasonlósági összekapcsolást nagyon sok helyen vagyunk kénytelenek használni, például adat integritás illetve tisztítás esetén, hasonló elemek megtalálásra illetve eltávolítására, szűrésnél.
Az cikk a szerkesztési távolság alapú hasonlósági összekapcsolásokkal foglalkozik, amely képes megtalálni két halmazból azokat a stringeket, melyeknek a szerkesztési távolsága egy bizonyos határszám alatt van. Ez hasznos lehet, például hasonló lekérdezések megtalálásában, egy lekérdezési naplóban, illetve két adathalmaz egyesítésekor (pl.: nevek, helyek, címek)
Az eddigi tanulmányok mind a már említett *„szűr és pontosít”* típusú megoldásokkal foglalkoztak. Az ilyen algoritmusok a *szűr* fázisban először generálnak egy szignatúrát a

stringekből és ezek alapján keresik meg a hasonlóakat. Ez után a *pontosító* fázisban leellenőrzik, hogy ezek a párok helyesek voltak-e és kiírják az eredményt. Sajnos a szignatúrán alapuló módszereknek vannak hátrányai, például, hogy rövid stringekre nem lehet, jó szignatúrát gyártani, illetve a halmazok dinamikus megváltozását nem képesek kezelni (pl.: IDF (reciprok dokumentumbéli gyakoriság) felhasználása miatt a nagyméretű indexeket újra kell építeni).

Ezen problémák megoldására kínál az eredeti cikk trie-fa alapú megoldásokat, a trie-fa tulajdonságainál fogva sokkal kisebb tárhely igényű, mint a fent említett algoritmusok. A leg egyszerűbb algoritmus a két halmazból épít két fát és az összes stringre megnézi, hogy van e hozzá hasonló a másik fában, fontos megjegyezni, hogy amennyiben egy string nem elég hasonló egy csúcshoz, akkor a csúcs gyerekeihez sem lesz hasonló. Erre alapozva a szerzők adnak egy rész-trie vágási módszert, amivel gyorsítható az algoritmus. További teljesítmény javítás érdekében kifejlesztettek egy kétszeres rész-trie vágási technikát is, amely azon a feltételezésen alapul, hogy amennyiben két csúcs nem elég hasonló akkor ezen csúcsok gyerekei sem lesznek hasonlóak.

Annak a hatékony meghatározására, hogy két csúcs eléggé hasonló-e az eredeti cikk 3. szekciójában több algoritmust is adnak a szerzők, ezen algoritmusoknak elég csak egyszer hozzáférni a csúcshoz, így is javítva a hatékonyságot. A csúcsok többszöri vizsgálatának elkerülése érdekében a cikkben megtalálhatóak, olyan, algoritmusok melyek a fa építése közben keresik a hasonló párokat, illetve ezekhez hatékony vágási technikák is találhatóak.

A cikk leírja, hogy hogyan támogatható az eredeti halmazok változásainak dinamikus kezelése. Az eredeti halmazbeli elemeket megjelöli *régi*ként az újakat pedig *új*ként (az eredeti cikkben *visited* illetve *unvisited*) és csak az *új* csúcsokban keresik a hasonló párokat.

A magasabb szerkesztési küszöbbel rendelkező párok hatékony megtalálása érdekében ad a cikk egy két fát építő algoritmust, a string bal- illetve jobb-feléből, melyre bizonyítható, hogy a két fában alacsonyabb távolsággal (eredeti fele) keresett párokból előállítható a keresett halmaz.

A cikk végén kísérletekben bizonyítják, hogy az új algoritmus nagyságrendekkel jobb az eddigieknél, igazi adatok esetében.

Trie-alapú megoldás

Ez a rész arra szolgál, hogy formalizálja a hasonlósági összekapcsolást, a szerkesztési távolságot illetve a trie-alapú megoldásokat.

A probléma formalizálása

Két adott string halmaz esetében a hasonlósági összekapcsolás megtalálja a hasonló elemeket, a cikk a szerkesztési távolságot használja a hasonlóság mérőszámaként. Két string szerkesztési távolsága alatt az egy karakternyi szerkesztéseknek (megváltoztatás, törlés, beszúrás) azt a minimális számát értjük, amellyel a két string átalakítható egymásba. Két string hasonló, ha a szerkesztési távolságuk egy küszöbszám alatt van.

Definíció:

Legyen két string halmaz $\mathcal{R}$ ***és*** $\mathcal{S}$ ***illetve egy megadott szerkesztési távolsági küszöb*** τ.

A hasonlósági összekapcsolás $\langle r,s\rangle \in \mathcal{R} \times \mathcal{S}$ ***, hogy***

$SZT(r,s) < \tau$ *azaz* $\{\langle r,s\rangle \mid SZT(r,s) \leq \tau, \quad r \in \mathcal{R}, \quad s \in \mathcal{S}\}$

Prefix vágás

A feladat legegyszerűbb megoldása, hogy minden string párra kiszámoljuk a szerkesztési távolságukat, nagyon költségigényes. A legtöbb esetben annak a kiderítésére, hogy két string hasonló-e nincs szükség a teljes stringek összehasonlítására.

Legyen két string $r = r_1r_2\ldots r_n$ és $s=s_1s_2\ldots s_m$ illetve legyen D egy $n+1 \times m+1$ mátrix, ahol D(i,j) jelölje a szerkesztési távolságot $r_1r_2\ldots r_i$ és $s=s_1s_2\ldots s_j$ között. Ekkor felhasználva a dinamikus programozási tételt:

$D(0,j) = j$ $(j \in (0..n))$

és $D(i,j) = \min (D(i-1,j) + 1), D(i,j-1)+1, D(i-1,j-1)+\theta)$

ahol $\theta = 0$ ha $r_i = s_j$; egyébként $\theta = 1$. D(i,j) aktív bejegyzés, ha $D(i,j) \leq \tau$. A mátrix számítása közben, a nem aktív bejegyzésekből könnyen ki lehet látni, ha egy prefixre már nem lesz aktív bejegyzés a későbbiekben a dinamikus programozási tétel alapján. Ilyen esetben hamar befejeződhet az algoritmus. Ezt a módszert hívják prefix vágásnak. A következő képen egy példa mátrix látható.

	j	0	1	2	3	4
i			e	b	a	y
0		0	1	2	3	4
1	k	1	1	2	3	4
2	o	2	**2**	**2**	**3**	4
3	b	3	3	2	3	4
4	y	4	4	3	3	3

Fig. 1 Prefix pruning. Matrix for computing edit distance of two strings "ebay" and "koby". *Shaded cells* denote *active entries* for $\tau = 1$

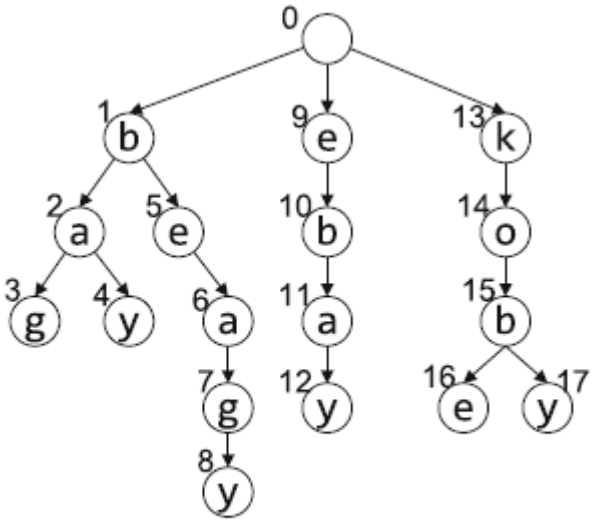
Megfigyelések

Résztrie-vágás

Legyen $\mathcal{R}$ egy string halmaz, illetve legyen $|\Sigma|$ a különböző karakterek száma $\mathcal{R}$-ben. Ekkor nem több mint $|\Sigma|^i$ különböző i hosszúságú prefix lehet $\mathcal{R}$-ben. Sok string esetében, soknak lesz közös prefixe, például legyen $|\mathcal{R}| = 10^6$ és $|\Sigma| = 26$, ekkor átlagosan $|\mathcal{R}| / |\Sigma| = 38462$ darab stringnek van közös egy karakter hosszú prefixe. Ez a legrosszabb eset, a cikk 6.2-es

szekciójában valós adatokon alapuló megfigyelések vannak. Ez alapján a prefix-vágást kiterjeszthetjük string halmazokra is. A trie adatstruktúrát használjuk arra, hogy indexeljük a stringeket. A trie egy fa struktúra, ahol egy út a gyökértől a levélig reprezentál egy stringet, minden csúcs az úton a string egy karakterével van címkézve.

Mivel a trie struktúrában számos stringnek, lehet közös őse (ezáltal közös prefixe) így a prefix vágás segítségével stringek csoportjait vághatjuk le a fáról. A következő kép egy Trie index struktúrát mutat be.



A sample data set

SID	String
s_1	bag
s_2	ebay
s_3	bay
s_4	kobe
s_5	koby
s_6	beagy

Fig. 2 Trie index of a sample data set

Adott trie és string s illetve csúcs n esetén n, az s aktív csúcsa, amennyiben SZT(s,n) ≤ τ. Ha n nem aktív csúcs, s minden prefixére akkor minden n alatt lévő string nem hasonló s-sel. Ezt felhasználva egy új vágási technika a *rész-trie vágás*: Adott trie és string s-re számítsuk ki az összes s-hez hasonló stringet, úgy, hogy minden n csúcsra, ha n nem aktív csúcs s bármely prefixére akkor az n gyökerű részfát nem járjuk be. A következő lemma megmutatja a vágási módszer helyességét.

***Lemma*:**

Legyen T egy trie és s egy string illetve egy adott τ szerkesztési küszöb.

Amennyiben n nem aktív csúcs s bármely prefixére akkor n gyermekei nem lesznek

$s -$ *hez hasonlóak*

A lemma bizonyítása megtalálható az eredeti cikkben.

Kétszeres résztrie-vágás

A résztrie-vágás csak az $\mathcal{R}$-beli elemeket indexeli egy trie-ban, de persze az $\mathcal{S}$-beli elemekre is végre lehet hajtani a vágást. Ennek érdekében egy közös triet építünk $\mathcal{R}$ és $\mathcal{S}$-nek amire végrehajtjuk a résztrie-vágást. Bevezethetünk egy új technikát a kétszeres résztrie-vágást: Adott trie és bármely u,v csúcsok, amelyikben u nem aktív csúcs v minden szülőjére, illetve v nem aktív csúcs u minden szülőére, akkor az u és v alatt lévő csúcsokat levághatjuk a fáról. A következő lemma megmutatja a helyességet:

Lemma: Legyen u, v trie csúcsok illetve adott τ szerkesztési távolsági küszöb.

Amennyiben u nem aktív csúcs v minden ősére, illetve v nem aktív csúcs u minden ősére

a v és u alatti stringek nem lehetnek hasonlóak egymáshoz.

A bizonyítás az eredeti cikkben olvasható.

Trie alapú algoritmusok

Első körben bemutatásra kerül egy egyszerű trie-keresésen alapuló algoritmus, amely a rész-trie vágást használja. Adott $\mathcal{R}$ és $\mathcal{S}$ halmazokra a Trie-keresés először elkészíti a $\mathcal{R}$-beli elemekre a trie struktúrát, aztán minden $\mathcal{S}$-beli elemnek kiszámolja az aktívcsúcs-halmazát ($\mathcal{A}_s$), ekkor minden $r \in \mathcal{A}_s$ amennyiben r levél akkor $\langle r,s \rangle$ egy hasonló pár.

Az aktívcsúcs-halmazok elkészítésére Choudri and Kaushik, Ji et al. algoritmusát használjuk [2]. Minden $s=s_1s_2\ldots s_m$ string esetén az algoritmus inicializálja a halmazt, az üres string aktív csúcs halmazaként, ebbe minden τ-nál nem mélyebben lévő csúcs beletartozik, ezután inkrementálisan kiszámolja minden $s_1..s_i$ halmazra is $s_1..s_{i-1}$ felhasználásával.

A Trie-bejárás algoritmus

Sajnos a Trie-keresés nem képes kétszeres résztrie-vágást alkalmazni mivel a $\mathcal{S}$-beli elemekkel nem foglalkozik. Ezért a következő Trie-bejárás algoritmust javasolja a cikk.

A Trie-bejárás algoritmusa elkészíti a fát mind $\mathcal{R}$, mind $\mathcal{S}$ halmazokhoz és minden csúcsnak kiszámítja egyszer az aktív csúcs halmazát, még akkor is, ha ez a csúcs más csúcsoknak a prefixe.

A Trie-dinamikus algoritmus

A Trie-bejárás algoritmusa minden csúcsra ki kell, hogy számolja az aktív csúcsok halmazát, ez az alábbi tétel felhasználásával elkerülhető:

Amennyiben u a v csúcs aktív csúcsa akkor v is az u aktív csúcsa.
Ezen megfigyelés alapján a cikk a Trie-dinamikus algoritmust adja, amely elkerüli a felesleges kiszámolását a halmazoknak. A az algoritmus inkrementálisan építi fel a fát és minden új csúcs beillesztésekor kiszámolja annak az aktív csúcs halmazát, és a szimmetria tulajdonság alapján frissíti a már a fában lévő csúcsok halmazait is.

Trie-ÚtVerem algoritmus

A Trie-dinamikus algoritmus, amikor egy új stringet illeszt a fába akkor bármelyik már létező csúcshoz adhat új csúcsokat, ezért bármelyik már bent lévő csúcs aktív csúcs halmazát felhasználhatja az új csúcs halmazának kiszámításához. Ennek eléréséhez a halmazokat folyamatosan a memóriában kell tartani. Amíg a Trie-bejárás algoritmusa nem tartott a trie-n kívül mást a memóriában, sok felesleges számítást végzett addig Trie-dinamikus nem számol feleslegesen, de a memóriát intenzíven használja.
Ennek megoldásaként ajánlja a cikk a Trie-ÚtVerem algoritmust, amely kevés memóriát igényel mégis képes nagy hatékonyságot elérni. A csúcsok bejárása alatt fenntartunk egy „virtuális nem teljes" részfát (VNT) a már látogatott csúcsoknak. Minden még meg nem látogatott csúcsot megjelölünk meglátogatottként aztán a VNT-ben kiszámoljuk az aktív csúcs halmazát. Minden későbbi meg nem látogatott csúcsra csak a már meglátogatott csúcsokban számoljuk ki az aktív csúcsokat. A memória felhasználás mérséklése érdekében preorder járjuk be a fát és egy vermet használunk a gyökértől a csúcsig tartó út nyilvántartására (az addigi aktív csúcs halmazzal együtt). Minden egyes megvizsgált n csúcsra tudjuk a szülőjének aktív csúcs halmazát felhasználva kiszámítani az aktív csúcs halmazát, de elég a verem első τ elemének a halmazát frissíteni, mivel az ennél távolabb lévő elemek már biztosan nem aktív csúcsai n-nek. Kísérleti eredmények megmutatják, hogy ez az algoritmus nagyon sok felesleges frissítéstől kíméli meg magát.
A kód illetve annak részletes magyarázata megtalálható a cikkben.

Vágási technikák

Az algoritmusok hatékonyságának további javítása érdekében háromféle vágási technikát javasolnak a szerzők.
Hosszúsági vágás: Legyen r,s stringek amennyiben $| l(r) - l(s) | > \tau$ akkor a szerkesztési távolságuk biztosan nem kisebb, mint τ. Ennek a tulajdonságnak a felhasználásával bevezethetünk egy új vágási technikát. Minden csúcsnak eltároljuk az $[l_s,l_l]$ számokat, ahol l_s

a legrövidebb illetve l_l a leghosszabb string hossza a részfában. Legyen például u [2,3] illetve v [5,7] és mivel a hosszúságok különbsége nagyobb, mint τ (=1) így v-t kiejthetjük $\mathcal{A}_u$-ból.

Egyszeres-ág vágás: Amennyiben u a v egy leszámozottja és a részfáikban ugyanazok a levelek találhatóak meg, akkor v-t ki lehet hagyni $\mathcal{A}_u$-ból, mivel a v már semmiképp nem adna új levelet, ami párja lehet egy u-beli levélnek.

Számolásos vágás: Legyen két csúcs u,v amennyiben pontosan egy string van amelynek u és v is prefixe akkor v kihagyható $\mathcal{A}_u$-ból mivel a részfájukban nem lesz két darab string, amelyek párban lehetnének.

Ezek a vágási technikák könnyen beilleszthetők bármelyik az előzőekben említett algoritmus aktív csúcs halmaz kiszámolási részébe a Trie-dinamikus kivételével.

A halmazok dinamikus megváltozásának és különböző halmazok összekapcsolásának támogatása

Inkrementális hasonlósági összekapcsolás

Ez a rész megmutatja, hogy hogyan támogatható hatékonyan az eredeti halmazok megváltozásának dinamikus követése. Feltesszük, hogy a $\mathcal{S}$ halmazt önmagával kapcsoltuk össze, és egy string hozzá adásával $\Delta\mathcal{S}$ halmazt képezzük belőle.

Formálisan:

Definíció: Adott $\mathcal{S}$ string zalmaz illetve $\Delta\mathcal{S}$ megváltozott halmaz,

τ szerkesztési távolsági küszöb ekkor a inkrementális összekapcsolás megadja az

összes párt, hogy $SZT(r,s) \leq \tau$ ($r \in \Delta\mathcal{S}$, $s \in \mathcal{S} \cup \Delta\mathcal{S}$)

A Trie-bejárás illetve a Trie-ÚtVerem algoritmust ki tudjuk bővíteni, hogy támogassa az inkrementális összekapcsolásokat, de a Trie-Dinamikus sajnos nem képes erre lévén, hogy minden megváltozáskor újra kellene építenie a fát.

A Trie-ÚtVerem algoritmus felhasználásával a kiterjesztés ötlete a következő. Legyen a trie index $\mathcal{T}$ amit a $\mathcal{S}$ halmazból készítettünk. Legyen az új halmaz $\Delta\mathcal{S}$, ekkor frissítjük a $\mathcal{T}$-t $\mathcal{T}$'-re méghozzá, hogy beillesztjük a $\Delta\mathcal{S}$-beli elemeket. A $\mathcal{T}$'-ben legyen $\Delta\mathcal{T}$ a $\Delta\mathcal{S}$ elemeinek részfája. Ekkor kiegészítjük a Trie-ÚtVerem algoritmust, hogy találja meg a hasonló párokat a $\Delta\mathcal{T}$-ben is. Ezt úgy tesszük, hogy amikor az algoritmus eléri az n csúcsot az aktív csúcs

halmazt a $\mathcal{T}$' segítségével számolja ki. Pszeudó kód illetve annak részletes magyarázata az eredeti cikkben olvasható.

Különböző halmazok összekapcsolása

Az alábbi rész kitér arra, hogy miként egészítsük ki az algoritmust, hogy képes legyen két különböző string halmaz összekapcsolásának támogatására. Ismét a Trie-ÚtVerem algoritmust használjuk az ötlet elmagyarázására. A következő definíció felhasználásával egyszerűsítjük a magyarázatot.

Definíció: Legyen $\mathcal{T}$ egy trie, n egy csúcs illetve $\mathcal{S}$ egy halmaz. Az n megjelenik $\mathcal{S}$-ben amennyiben van olyan s elem amelynek prefixe az n.

A Trie-ÚtVerem algoritmusból elkészítjük a Trie-ÚtVerem$^+$ algoritmust a következő módosításokkal. Az új algoritmus elkészíti a Trie indexet a $\mathcal{S} \cup \mathcal{R}$ halmazra, és minden $\mathcal{R}$-beli elemnek elkészíti az aktív csúcs halmazát, de csak az $\mathcal{S}$-beli elemek felhasználásával. $\mathcal{A}r :=$ { n | minden olyan n csúcs, hogy $|n| \leq \tau$ és $n \in \mathcal{S}$ }. Illetve bevezetjük, hogy csak $u \in \mathcal{R}$ elemek kerülhetnek a verembe.

A Trie-ÚtVerem javítása nagy szerkesztési távolság küszöbök esetén.

Mivel a Trie-ÚtVerem algoritmusának minden csúcsra ki kell számítania az aktív csúcs halmazt, ezért ez nagy τ esetén nagyon erőforrás-igényes lesz. Ennek javítására bevezetjük a Bináris-Trie-ÚtVerem (B-TUV) algoritmust.

Minden $r=r_1..r_n$ stringre bevezetjük a közvetkező jelöléseket. $L(r)=r_1 \ldots r_{(|r|/2)}$ illetve $R(r)=r_{((|r|+1)/2)}...r_{|r|}$ a r bal illetve jobb felére. Megállapíthatjuk, hogy r hasonlít s-hez τ küszöb mellett amennyiben a következők közül valamelyik igaz:

1. L(r) hasonló, mint s valamelyik prefixe (τ / 2) küszöbbel
2. R(r) hasonló, mint s valamelyik suffixe (τ / 2) küszöbbel

Ezt a feltételezést felhasználva javasoljuk a B-TUV algoritmust. Adott $\mathcal{S}$ string halmaz illetve τ küszöb. Előre elkészítjük az L($\mathcal{S}$) halmazt, ami a stringek bal felét tartalmazza. Ekkor lefuttatjuk a Trie-ÚtVerem$^+$ algoritmusát az L($\mathcal{S}$) és $\mathcal{S}$ halmazokon a τ / 2 küszöbbel. Ekkor minden $L(r) \in L(\mathcal{S})$-beli elemre, hogy megtaláljuk, azokat az s elemeket, amiknek a prefixe hasonló L(r)-hez bejárjuk a L(r) aktív csúcs halmazát ($\mathcal{S}$-beli elemek) és megkeressük a leveleket. Ekkor megkaptuk azokat a S leveleket, amelyeknek a prefixe hasonló L(r)-hez. Ugyanezt végigfuttatva R($\mathcal{S}$)-re megkapjuk azokat a s-eket, amelyeknek a suffixei vannak R(r)-től τ/2 távolságra. Ezután leellenőrizzük a párokat és kiírjuk őket. Az alábbiakban optimalizálási technikákat javasolunk a B-TUV algoritmus javítására.

Levél optimalizáció

Minden L($\mathcal{S}$)-beli elemre a B-TUV algoritmus kiszámolja az aktív csúcs halmazt, aminek érdekében be kell járnia a leszámozottakat a fában, hogy megtalálja a leveleket. Némely levelet le lehet vágni. Mivel L(r) r string bal fele ezért |r| = 2|L(r)| vagy 2|L(r)|+1. Minden s-re, ahol SZT(r,s) ≤ τ igaz, hogy a ||s|-|r|| ≤ τ, tehát 2|L(r)|-τ ≤ |s| ≤ 2|L(r)|+1+τ , tehát minden nem ilyen csúcsot levághatjuk a részfájával együtt.

Trie-méret optimalizáció

Az $\mathcal{S}$-beli párok megtalálására, az algoritmus két Trie fát készít el egyet L($\mathcal{S}$)-hez, egyet pedig R($\mathcal{S}$)-hez. Ezen fák méretének csökkentése nem csak a memória felhasználáson javít, de a Trie-ÚtVerem$^+$ futásának igényén is. Figyelembe véve a B-TUV algoritmus alap feltevését és r,s, stringeket, hogy SZT(r,s) ≤ τ. Ekkor észrevehetjük, hogy ahhoz az első feltételt igazoljuk, nem kell az egész stringet leellenőrizni. Elég, ha találunk olyan P(s) prefixet, hogy SZT(L(r),P(s)) ≤ τ/2. A második feltételhez, ugyanez alkalmazható a suffixekre. Ezt úgy érhetjük el, hogy a keresett prefixek hosszát (|s| + τ) / 2 illetve a suffixek hosszát (|s| + τ +1) / 2-ben maximáljuk. Amennyiben van ilyen prefix/suffix akkor a pár hasonló volt.

A B-TUV algoritmust úgy módosítjuk, hogy minden s ∈ $\mathcal{S}$ elemre levágjuk a prefixeket (|s| + τ) / 2 hosszra és elkészítjük a $P_{max}(\mathcal{S},\tau)$ halmazt. Illetve elkészítjük, a $S_{max}(\mathcal{S},\tau)$ halmazt a suffixeknek a (|s| + τ +1) / 2 hosszal. Ekkor használhatjuk a $S_{max}(\mathcal{S},\tau)$, $P_{max}(\mathcal{S},\tau)$ halmazokat mint Trie-indexeket.

A következő kép szemlélteti a vágási technikákat.

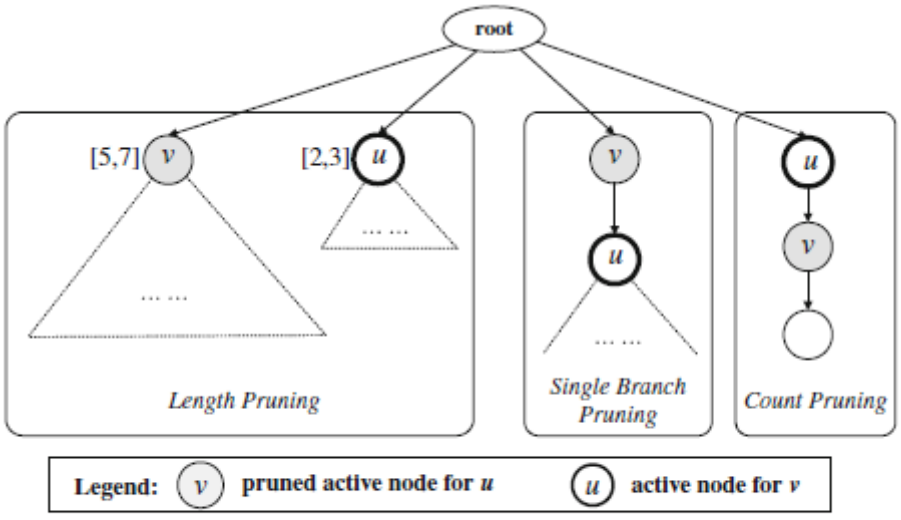


Fig. 10 Three pruning techniques ($\tau = 1$)

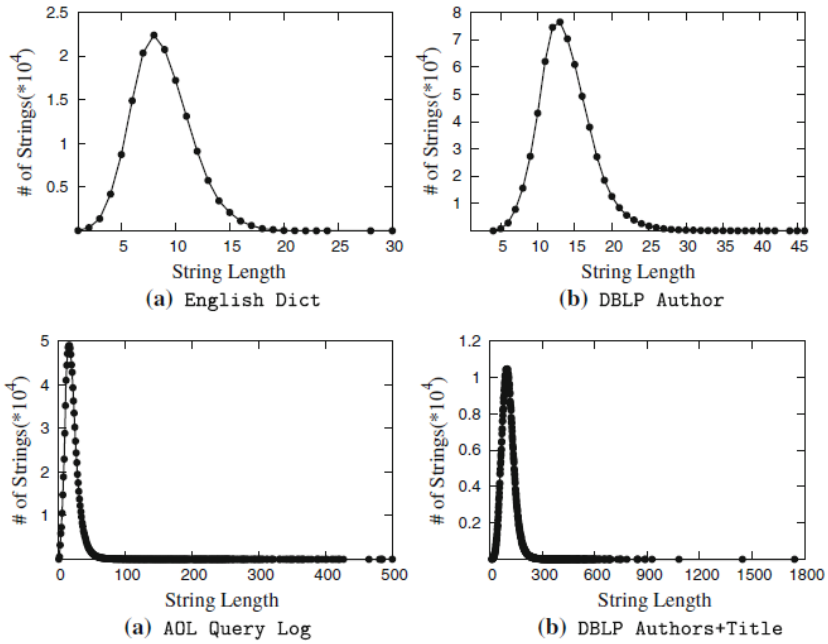
Kísérletek

Az eredeti cikk kimerítő kísérlet sorozatot végzett annak kiderítésére, hogy milyen gyorsak az általuk megalkotott algoritmusok. Adathalmaznak a következőket használták :

- English Dict : Az Cygwin Aspell helyesírás ellenőrző szó gyűjteménye
- DBLP Author : A DBLP könyvtár szerzői
- AOL Query Log: 1 millió véletlenszerű különböző lekérdezés
- DBLP Author+Title : Lsd. DBPL Author plusz a cikkek címe

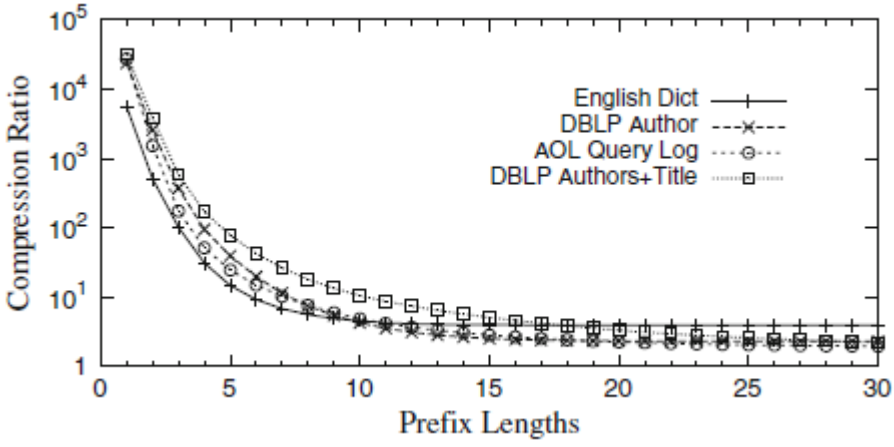
A stringek hosszának eloszlását szemlélteti a következő ábra.

Fig. 18 String length distribution



Közös prefixek

Adott $\mathcal{S}$ halmaz illetve l prefix hosszúságnál legyen P_l az $\mathcal{S}$-beli prefixek száma melyek nem hosszabbak, mint l, legyen P^d_l az összes ilyen különböző prefix. Látható, hogy a következő hányados (P_l / P^d_l) meghatározza, hogy a stringek mennyire osztoznak a prefixeken.



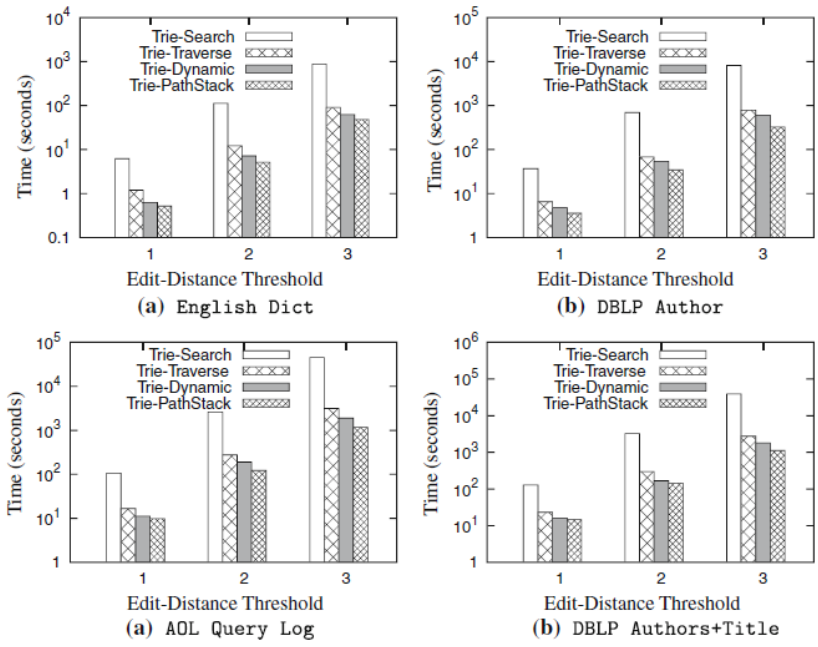
Kísérleti körülmények

Összehasonlítás képen a szerzők a következő algoritmusokat használták: All-Pairs-Ed[3], Ed-Join[4], Part-Enum[5], M-tree[6], melyeknek a leírása megtalálható az eredeti cikkben. A tesztelés egy Intel Core 2 Quad X5450 3.00Ghz processzorral és 4 GB memóriával felszerelt Ubuntut futtató gépen történt.

A Trie alapú algoritmusok összehasonlítása

A Trie alapú algoritmusaink összehasonlításához, alapnak használjuk a Trie-keresés algoritmust, amelyik csak a rész-trie vágást használja. Ahogy a következő ábrán is látszik a többi algoritmus akár 1-2 nagyságrenddel is jobban teljesít a Trie-keresésnél. Az eredmények megmutatják, hogy a kétszeres rész-trie vágás milyen jó technika. A Trie-dinamikus illetve a Trie-ÚtVerem algoritmus a szimmetria tulajdonság kihasználásának köszönhetően nagyjából kétszer gyorsabb, mint a Trie-bejárás. A Trie-ÚtVerem pedig a Trie-dinamikusnál is gyorsabb mivel a Trie-dinamikus minden beillesztés után $|\mathcal{A}_n|$ csúcsot frissít, míg a Trie-ÚtVerem csak τ ($\ll|\mathcal{A}_n|$) darabhoz.

Fig. 20 Comparison of the four algorithms



A következő ábra megmutatja, hogy adott τ értékek mellett hány darab aktív csúcsot kell az algoritmusoknak tárolniuk. Látszik, hogy a Trie-dinamikus nagyon sok csúcsot tárol, mivel minden csúcsnak az aktív csúcs halmazát tárolnia kell. A többi algoritmusban az aktív csúcs halmazok száma megegyezik a trie fa legnagyobb mélységével. A szimmetria tulajdonság felhasználása miatt a Trie-ÚtVerem algoritmus tárolja a legkevesebb csúcsot.

Table 3 Maximal #active nodes on AOL Query Log

τ	TRIE-SEARCH, TRIE-TRAVERSE	TRIE-DYNAMIC	TRIE-PATHSTACK
1	2444	42346799	**2172**
2	31374	230511829	**18477**
3	257896	2444928000	**201825**

Vágási technikák értékelése

A vágási technikák értékelését úgy oldották meg a szerzők, hogy a Trie-ÚtVerem algoritmusba építették be őket. A következő ábra megmutatja, hogy hogy változtattak a vágások az aktív csúcsok számán és így a teljesítményen is. Látszik, hogy minden vágási technika javít a teljesítményen.

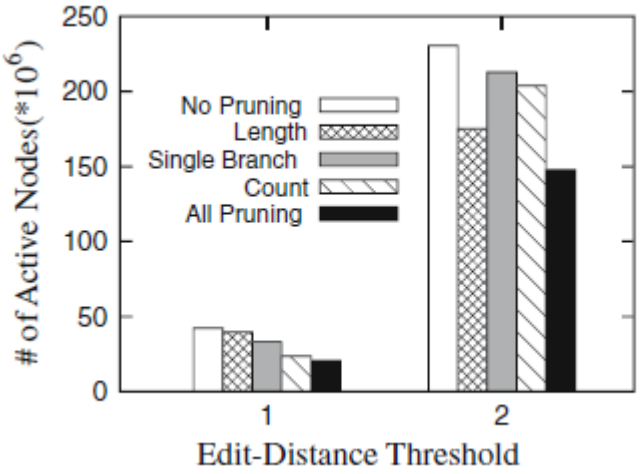


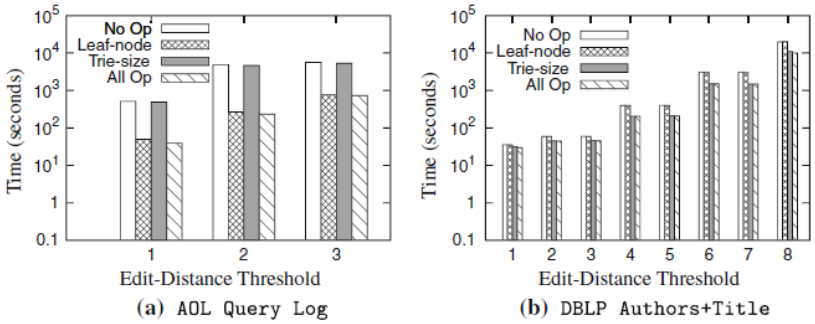
Table 4 Performance improvement of three pruning techniques on AOL Query Log

τ	No pruning (second)	Pruning (second)	Improvement (%)
1	9.76	7.35	24.7
2	122.48	104.48	14.7
3	1,174.94	1,066.12	9.3

A Bináris-Trie-ÚtVerem algoritmus értékelése

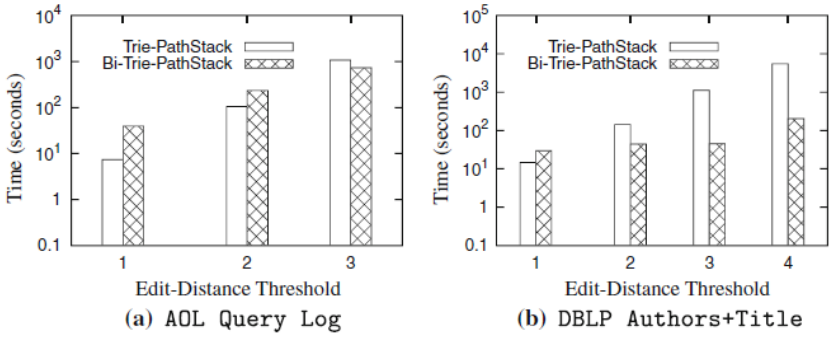
Első körben a B-TUV algoritmus optimalizálási technikáit elemzi a cikk, lefuttatják az algoritmust az optimalizálások nélkül illetve egyenként, az összessel is. Ennek az eredményét mutatja a következő ábra. Látszik, hogy két optimalizálási technika javít sokat a teljesítményen. Látható, hogy az AOL Query adathalmazon a Levél optimalizáció a leghatékonyabb, amíg a DBLP Authors+Title halmazon pedig a Trie-méret optimalizáció a legjobb. Ennek az oka, hogy az AOL Query Log több rövid stringet tartalmaz, aminek következtében sokkal több levél lesz a stringek jobb és bal feleinek leszámozottjaiban. Tehát a Levél optimalizáció a leghatékonyabb. Ezzel szemben a DBLP Authors+Title sok hosszú stringet tartalmaz. Hosszú stringek esetében a prefixek illetve suffixek szűrése sokat javít a hatékonyságon, így a Trie-méret a legjobb optimalizációs technika.

Fig. 22 Comparison of running time of BI-TRIE-PATHSTACK with different optimization techniques



Következőre össze hasonlítja a cikk a B-TUV illetve a Trie-ÚtVerem algoritmusokat. Mind a kettőt lefuttatják az AOL Query illetve a DBLP Authors+Title halmazon. a következő ábra megmutatja ezt. Megfigyelhető, hogy a Trie-ÚtVerem alacsony szerkesztési távolság esetén még gyorsabb, mint a B-TUV, de ahogy nő a küszöb úgy lesz egyre jobb a B-TUV algoritmus, amíg gyorsabb nem lesz, mint a Trie-ÚtVerem.

Fig. 23 Comparison of running time between TRIE-PATHSTACK and BI-TRIE-PATHSTACK



Létező algoritmusokkal való összehasonlítás

A következőkben az eddigi létező algoritmusokkal hasonlítja össze a cikk Trie-alapú algoritmusokat.

Az első összehasonlítási alap az indexek mérete (MB). Az összehasonlítás alapját az Ed-Join, All-Pairs-Ed, Part-Enum algoritmusok szolgálják, melyeket az összes adathalmazon le lettek futtatva illetve finom hangolva lettek, hogy a legjobb teljesítményükkel hasoníthassuk össze. A Trie algoritmusok közül a Trie-ÚtVerem illetve a B-TUV algoritmusokat mutatja be a cikk. Az eredmények láthatóak a következő ábrán. Látható, hogy az eddig létező algoritmusok sokkal nagyobb indexeket készítenek, mint a Trie alapúak, ennek oka, hogy amíg azok sok szignatúrát tárolnak a stringekre, a Trie esetében a prefixek tárolása közös, így a méret csökken.

Table 5 Comparison of index sizes (MB) on four data sets

Data sets	TRIE-PATHSTACK	BI-TRIE-PATHSTACK	Part-Enum	All-Pairs-Ed	Ed-Join
English Dict	2	4	16	30	10
DBLP Author	16	25	54	155	65
AOL Query Log	29	80	120	305	160
DBLP Authors +Title	96	127	142	830	751

A második összehasonlítás alapja a hatékonyság. Mivel a nem Trie alapú algoritmusok hatékonysága nagyban függ a paramétereik helyes beállításától, ezért a cikk a legjobb eredményeket hasonlítja össze a beállítások finomhangolása után. A következő ábra megmutatja, hogy milyen az algoritmusok teljesítménye egymáshoz képest.

Láthatóak, hogy az M-Tree algoritmus teljesít a leghosszabbul minden esetben. Az az ábrán látszik, hogy a Trie-ÚtVerem tizenötször jobban teljesít, mint a legjobb nem Trie alapú Ed-Join algoritmus az English Dict adathalmazon. A b ábrán látszik, hogy a Trie-ÚtVerem egy nagyságrenddel jobban teljesít az Ed-Joinnál a DBLP Author adathalmazon $\tau = 2$-re. A c ábrán a B-TUV algoritmus teljesít a legjobban az AOL Query Log halmazon. A d ábrán az Ed-Join viszont gyorsabb, mint a B-TUB a DBLP Authors+Title halmazon $\tau = 4$-gyel.

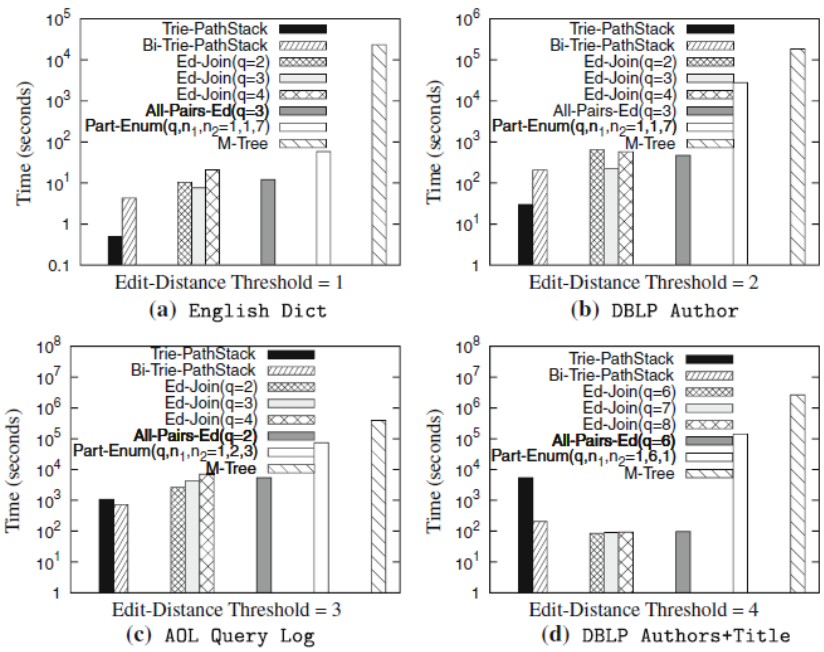


Fig. 24 Comparison of running time with state-of-the-art methods on four data sets

Következtetés

Látható, hogy azon halmazokon, ahol a stringek átlagos hossza kisebb, mint 30 a Trie alapú algoritmusok jobban teljesítenek, de a hosszabb halmazokon az Ed-Join jobbnak bizonyul, ez azért van mivel rövid stringek sokkal valószínűbben tartalmaznak sok közös prefixet, amire a Trie alapú algoritmusok nagyon gyorsak, de hosszú stringekre ez nem igaz, így sokszor kell kiszámolni az aktív csúcs halmazokat.

Összefoglalás

A cikk a Trie alapú algoritmusokat tanulmányozta a string hasonlósági összekapcsolások megoldására. Adott több megoldást is a feladatra, illetve adott több vágási technikát és optimalizációt amivel gyorsíthatóak az algoritmusok. A cikk végén kísérlet sorozat bemutatja, hogy a cikk által kínált algoritmusok milyen viszonyban álnak a létező legjobb algoritmusokkal.

Irodalom jegyzék

[1] Jianhua Feng, Jiannan Wang, Guoliang Li: Trie-join: a trie-based method for efficient string similarity joins; http://people.inf.elte.hu/kiss/12abea/Trie-join.pdf

[2] Ji, S., Li, G., Li, C., Feng, J.: Efficient interactive fuzzy keyword search. In WWW, pp. 433–439 (2009); Chaudhuri, S., Kaushik, R.: Extending autocompletion to tolerate errors. In: SIGMOD Conference, pp. 707–718 (2009)

[3] Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: WWW, pp. 131–140 (2007)

[4] Xiao, C., Wang, W., Lin, X.: Ed-join: an efficient algorithm for similarity joins with edit distance constraints. PVLDB **1**(1), 933–944 (2008)

[5] Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: VLDB, pp. 918–929 (2006)

[6] Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB, pp. 426–435 (1997)