Procesarea șirurilor de caractere

Analiza algoritmilor

14 decembrie 2017

Valentin-Gabriel Radu
valentingabrielradu@gmail.com
323CD, Facultatea de Automatică și Calculatoare
Universitatea Politehnica din București

Cuprins

Introducere

Am ales să tratez subiectul ***Procesarea șirurilor de caractere***. Această temă încearcă să identifice care dintre algoritmii prezentați sunt mai potriviți în ce situații și de ce. Algoritmii pe care am decis să îi analizez în primă fază sunt:

- Knuth-Morris-Pratt (KMP) – rezolvă găsirea unui subșir într-o secvență dată
- Rabin-Karp - rezolvă găsirea unui subșir într-o secvență dată
- Aho-Corasick - găsește mai multe subșiruri într-o secvență dată, simultan

Voi analiza, spre exemplu, problema găsirii unui subșir într-o secvență dată, acești algoritmi putând fi folosiți pentru a determina acest lucru. Mai mult, voi analiza cum se comportă aceștia legat de problema găsirii repetate a unui subșir în mai multe secvențe date, și care anume excelează și de ce. În limita timpului disponibil și a finalizării unei analize complete a algoritmilor menționați mai sus, voi încerca să analizez și alți algoritmi pentru aceste tipuri de probleme (de exemplu, algoritmul Trie).

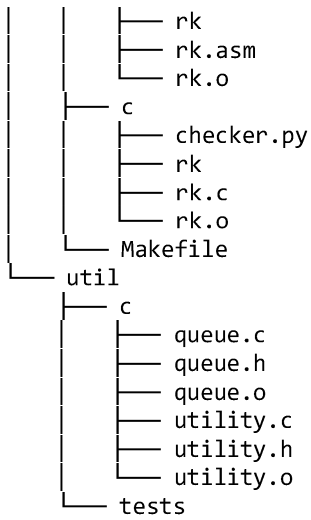
Pentru a analiza algoritmii, îi voi implementa atât într-un limbaj de programare de nivel înalt, și anume limbajul C, cât și în limbaj de asamblare, respectiv nasm pentru x86, pentru a încerca să evidențiez și diferențele între prelucrările făcute de compilator asupra codului și alte aspecte.

Ca problemă practică, voi încerca reimplementarea cât mai eficientă a utilitarului UNIX *grep*, folosind algoritmii studiați (grep identifică un șir de caractere primit ca argument în ceea ce citește de la standard input).

Conținutul arhivei

Arhiva conține implementările în C ale celor trei algoritmi, precum și implementări în limbaj de asamblare pentru doi dintre aceștia. Fiecare algoritm are câte un executabil asociat, ce se poate genera folosind un Makefile asociat fiecăruia. Structura proiectului este următoarea:

```
.
├── AC
│   ├── c
│   │   ├── ac
│   │   ├── ac.c
│   │   ├── ac.o
│   │   └── checker.py
│   └── Makefile
├── KMP
│   ├── asm
│   │   ├── kmp
│   │   ├── kmp.asm
│   │   └── kmp.o
│   ├── c
│   │   ├── checker.py
│   │   ├── kmp
│   │   ├── kmp.c
│   │   └── kmp.o
│   └── Makefile
├── RK
│   ├── asm
```



Mai jos sunt câteva comenzi utile pe care le puteți executa:

- Din rădăcina proiectului, tastați ”make” pentru a genera executabilele algoritmilor. Pentru a rula benchmark-ul, care va furniza ca input o serie de teste și va evalua corectitudinea, precum și timpul de execuție a algoritmilor, executați ”make check”.
- Din rădăcina fiecărui algoritm, tastați ”make” pentru a compila respectivul algoritm. Output-ul se ave anumele algoritmului (pentru KMp, de ex.: ”kmp”) și se va afla in fiecare din folderele denumite după limbajele în care este implementat respectivul algoritm (c și/sau asm). Tot în respectivele foldere se află și sursele pentru algoritmi în sine. Partea comună a algoritmilor (citirea, scrierea pe ecran , implementarea unei cozi pentru Aho-Corasick etc.) se află în folderul util din rădăcina proiectului.
- Tot în rădăcina fiecărui algoritm, puteți curăța codul executabil și unitățile obiect (Makefile-ul generează obiecte pentru fiecare fișier sursă, astfel nefiind necesară recompilarea cozii de exemplu, dacă doar s-a modificat codul unui algoritm) prin executarea ”make clean”.
- Pentru a executa benchmark-ul doar pentru unul din algoritmi, la fel, tastați ”make check”.
- Algoritmii sunt compatibili cu problema Aho-Corasick de pe Infoarena (http://www.infoarena.ro/problema/ahocorasick). Pentru a putea testa unul din algoritmi pe Infoarena, tastați ”make preprocess” în directorul unuia dintre algoritmi pentru a genera un fișier sursă ”preprocess.c” ce conține tot ceea ce este nevoie pentru a fi încărcată sursa pe site (site-ul acceptă o singură sursă); de asemenea, pentru KMP si RK am inclus două surse C bazate pe sursele deja existente, care pot fi folosite pentru a verifica corectitudinea algoritmilor respectivi. Testele folosite de problema Aho Corasick de pe site-ul Infoarena sunt cele folosite și de benchmark-ul inclus în arhivă.
- Aplicațiile pot fi executate folosind ”make runc”. În acest mod nu sunt foarte utileș mai degrabă, puteți face ca ele să genereze un output stil grep, în care cuvintele identificate în șirul furnizat sunt evidențiate în culori folosind o comandă de genul: echo "Kindness is a language we deaf can hear and the blind can read. - Mark Twain" | make runc ARGS="can deaf". Scrieți ”make runc verbose” în loc de ”make runc” în comanda de mai sus pentru a afișa exact pozițiile

identificate; scrieți ”make runc samecolour” (exclusivă cu verbose) pentru a colora output-ul în stil grep folosind o singură culoare.

- În principiu, makefile-urile se așteaptă ca make să fie rulat din directorul în care se află makefile-ul respectiv.

Algoritmul trivial

Algoritmul trivial presupune compararea manuală a m litere din șir cu cele m litere ale cuvântului. După ce se compară [i..i+m] litere din șir, se incrementează i și repetă din nou comparația până când ceea ce a rămas de comparat din șir este mai mic decât lungimea cuvântului căutat (adică n-m comparații în total).

Complexitate

Deoarece cuvântul căutat, care are m caractere, este parcurs de n – m ori, complexitatea algoritmului este O(n * m). Aceasta este slabă, durata de căutare crescând polinomial cu creșterea dimensiunii textului și a cuvântului de găsit. Acest lucru este adresat de următorii trei algoritmi, care reușesc căutarea în timp liniar.

Algoritmul Knuth-Morris-Pratt

KMP este un algoritm simplu conceput pentru a identifica o secvență S în cadrul unui șir V. Față de algorimtul trivial, acesta pleacă de la idea că atunci când există o diferență între șirul căutat și cel în care se caută, șirul căutat conține suficientă informație pentru a indica unde ar să fi început următoarea potrivire, evitând reparcurgerea inutilă a șirului.

De exemplu, pentru un test de genul: ”abxaba**b**abayabaya”, cu ”ababa” de găsit, este evident că la întâlnirea lui ”x”, care este primul caracter din șir care nu se găsește în cuvânt, vom începe din nou compararea șirurilor de la poziția lui x, acest comportament fiind similar algoritmului trivial. Inovația algoritmului apare în faptul că, mergând în continuare pe șirul prezentat, cu a doua nepotrivire apărând la ”b” aldin. Spre deosebire de algoritmul trivial, KMP știe că a întâlnit o secvență similară după începutul comparației a doua, imediat înainte de a da de nepotrivire – adică, primele două litere din cuvânt, ”ab”, nu mai trebuie comparate, adică să ne întoarcem din nou în șir, deoarece acestea au fost deja comparate și sunt ok. Aceste ”cunoștințe” despre șir sunt aflate de către KMP prin construirea unui vector cu informație ajutătoare legate de fiecare caracter din șirul căutat.

Complexitate

Astfel, algoritmul KMP poate realiza compararea cuvântului cu șirul în timp O(n), unde n reprezintă numărul de caracter din șir (șirul este parcurs o singură dată), cu condiția ca vectorul de informații ajutătoare să fie generat. Generarea vectorului ajutător durează O(m), unde m este lungimea cuvântului (acesta este parcurs o singură dată). Astfel, complexitatea totală temporală a algoritmului este O(n + m).

Din punct de vedere al spațiului ocupat, algoritmul ocupă n pentru șir și m pentru vectorul auxiliar, deci O(n + m), din nou. Strict spațiu suplimentar este O(m), unde m este lungimea cuvântului. Pentru p cuvinte, ar ocupa O(m * p), dar se poate refolosi vectorul precedent la noua iterație, deci O(m).

Pentru verificarea a p cuvinte, complexitatea temporală ar fi O(p * m), deoarece vectorii auxiliari calculați nu pot fi refolosiți, aceștia depinzând de conținutul cuvântului. Complexitatea spațială rămâne aceeași dacă refolosim (realocăm) vectorul auxiliar deja folosit la cuvântul precedent.

Generarea vectorului auxiliar

Vectorul auxiliar indică de unde să continuăm comparația cuvântului cu șirul pe care facem căutarea, în caz că a eșuat comparația curentă. Astfel, pentru un șir unde o potrivire începe la S[m], iar nepotrivirea se întâmplă la S[m + i] pentru șirul W[i], atunci următoarea potrivire posibilă va începe la indexul m + i – T[i] în S (astfel, T[i] indică cât de mult să revenim după ce avem o nepotrivire).

```
void precompute(string text, size_t strlens, size_t* out)
{
	size_t j, i;

	j = 0, i = 1;
	while (i < strlens) {
		if (text[j] != text[i])
		{
			if (j > 0)
			{
				j = out[j - 1];
				continue;
			}
			else j--;
		}
		j++;
		out[i] = j;
		i++;
	}
}
```

Algoritmul

Algoritmul va parcurge șirul în care caută și va compara fiecare literă din șir cu fiecare din cuvânt. Când s-au comparat m litere, unde m este numărul de litere din cuvânt, avem o potrivire. Când eșuăm o comparație, folosim vectorul auxiliar pentru a seta poziția literei din cuvânt care se va compara cu litera curentă din șir la următoarea iterație.

```
void kmp() {
	/* ... */
	j = 0, match = 0;
	for (i = 0; i <= strlens; i++) {
		if (text[i] == pattern[j]) {
			j++;
			if (j == strlenp) {
				printf("Word %s appears from %d to %d.\n",
					pattern, (int)(i - strlenp + 1), (int)(i + 1));
				match++;
				j = out[j - 1];
			}
		} else {
			if (j != 0) {
				j = out[j - 1];
```

```
                        i--;
                    }
                }
            }
            /* ... */
}
```

Mai multe detalii, precum și o implementare funcțională a algoritmului se pot găsi în sursele algoritmului, pe care le-am atașat.

Algoritmul Rabin-Karp

Rabin-Karp este un algoritm ce servește același scop ca și algoritmul KMP, respectiv identificarea unei secvențe date într-un șir. Acesta pleacă de la idea că dacă aplicăm o funcție f asupra secvenței de găsit și a câte m litere din șir (unde m este lungimea cuvântului), aceasta va returna aceeași valoare în cazul în care cele două sunt egale.

Apoi, în funcție de cum este construită această funcție f, denumită funcție de hash, putem optimiza apelul acesteia, și anume să nu o recalculăm pentru toată noua subsecvență, ci să eliminăm din vechea valoare ceea ce corespunde literei de care am trecut și să adăugăm ceea ce corespunde literei care s-a adăugat subsecvenței de comparat, odată ce am înaintat în șir. O funcție f cu această proprietate este denumită și rolling hash function. Pentru algoritmul implementat de mine, am folosit o funcție simplă, însă cu ajutorul unor funcții optimizate se pot obține rezultate mai bune, așadar reușita acestui algoritm depinde de cât de puternică matematic este funcția respectivă de hash.

Funcția de hash

Am folosit o funcție de hash în care am ridicat BASE la puterea valoarea ASCII a literei curente. Evident, pentru a controla magnitudinea rezultatului, am păstrat mereu restul împărțirii acestui rezultat la PRIME. Astfel, când executăm, dacă știm H valoarea funcției de hash pentru subșirul anterior și dorim să calculăm hash-ul corespunzător șirului curent, putem afla acest lucru folosind:

```
H = (((((H - text[i - 1] * max_power) % PRIME) + PRIME)
            * BASE) + text[i + strlenp - 1]) % PRIME;
```

În cadrul expresiei, max_power este BASE la puterea lungimea cuvântului, iar strlenp este lungimea cuvântului.

Funcția de hash folosită este următoarea:

```
size_t hash(string text, int pos, int length)
{
      size_t S = 0, i;
      for (i = pos; i < length + pos; ++i) {
            S = (S * BASE + text[i]) % PRIME;
      }
      return S;
}
```

Complexitate

Algoritmul implică parcurgerea șirului, deci complexitatea este O(n). La aceasta se adaugă verificarea egalității celor două funcții de hash. Funcțiile de hash trebuie calculate o dată pentru cuvânt și o dată pentru primul subșir de lungimea cuvântului din text, de unde rezultă O(2), adică O(1). Apoi, pentru calcularea hash-ului subșirului curent, fac o scădere și o adunare, ceea ce reprezintă un număr constant de operații, independent de lungimea cuvântului, deci din nou O(1). Astfel, complexitatea rămâne O(n).

Totuși, aceasta nu este complexitatea finală, deoarece chiar dacă hash-urile obținute sunt egale, asta se poate întâmpla și pentru două cuvinte care nu sunt egale. De aceea, este necesar să verificăm manual, adică să comparăm caracterele subșirului și cuvântului, pentru a vedea dacă cele două sunt egale. Acest lucru are complexitatea O(m), unde m este lungimea cuvântului. Cum se poate întâmpla ca hash-urile tuturor subșirurilor să fie egale cu hash-ul cuvântului, rezultă o complexitate în cazul cel mai defavorabil O(n * m), ca în cazul algoritmului trivial. Totuși, în practică, acest lucru se întâmplă foarte rar, deci o complexitate pe cazul mediu potrivită este O(n + m) (la fel ca în cazul cel mai favorabil).

Din punct de vedere spațial, algoritmul ocupă spațiu pentru șir, plus spațiu pentru hash-ul subșirului curent și hash-ul cuvântului, deci O(n + 2), adică O(n). Pentru p șiruri, s-ar ocupa O(2 * p), adică O(p) spațiu pentru stocarea hash-urilor, dar la fel ca la KMP, putem refolosi spațiul unde am calculat hash-ul la precedenta executare, dacă rulăm algoritmul secvențial, deci rămâne O(n).

Pentru a verifica p cuvinte, complexitatea este O(p * (n + m)) – nu putem folosi hash-urile calculate deja, cu excepția cazului în care cuvintelele de găsit au aceeași lungime, caz în care nu mai este necesar să recalculăm hash-urile subsecvnțelor din șir, deoarece vor fi egale (acestea depind de lungimea cuvântului de găsit), iar astfel obținem aceeași complexitate, O(n + m). Din punct de vedere al spațiului însă, am avea nevoie de un alt vector de dimensiunea șirului în care să stocăm hash-urile, deci complexitatea ar fi O(2 * n). Astfel, acest algoritm este o îmbunătățțire din punct de vedere temporal față de KMP, însă pierde la capitolul comportament în cazul cel mai defavorabil, precum și la aspectul spațial.

Algoritmul

Un pseudocod pentru funcționalitatea algorimului ar putea fi următorul (include și recalcularea rolling hash-ului):

```
void RK() {
	/* ... */
	H = (((((H - text[i - 1] * max_power) % PRIME) + PRIME)
		* BASE) + text[i + strlenp - 1]) % PRIME;
	if (H == hashp) {
		int was = TRUE;
		for (j = 0; j < strlenp; j++)
		{
			if (text[i + j] != pattern[j]) {
				was = FALSE;
				break;
			}
		}
		if (was)
```

```
                {
                        if (y < 1000) {
                                output[y] = i;
                                y++;
                        }
                        match++;
                }
                was = FALSE;
        }
        /* ... */
}
```

Algoritmul Aho-Corasick

AC este un algoritm care servește același scop precum al celorlalți doi prezentați – acesta vine ca un algoritm care le face pe toate, urmărind să îmbunătățească atât comportamentul pe găsirea unui cuvânt în sine, cât și performanța în identificarea a p cuvinte.

Similar lui KMP, acesta face o preprocesare a cuvintelor de identificat, generând un automat cu stări finite pe baza celor p cuvinte. Acesta este un arbore de cuvinte-cheie, îmbogățit cu informații care să arate în ce stări se poate ajunge din starea curentă. După ce este construit trie-ul (arborele de cuvinte-cheie), șirul trebuie parcurs o singură dată și va identifica deodată toate cuvintele pe care le avem de găsit.

Generarea automatului cu stări finite

Pentru a genera acest automat, vom urma câțiva pași:

1. Pentru un dicționar de cuvinte, vom construi trie-ul asociat. Acesta este un arbore care pleacă de la o rădăcină R și are ca descendenți toate caracterele aflate pe prima poziție ale cuvintelor din dicționar, în mod unic. Apoi, acești descendenți au ca descendenți caracterele unice de pe poziția a doua din fiecare cuvânt din vector, iar fiecare caracter de pe această poziție 2 are ca părinte caracterul de pe poziția 1 din șirul din care a venit; ș.a.m.d. se repetă acești pași până când fiecare dintre cuvinte este epuizat, pe rând.
   Apoi, îmbogățim acest trie cu câteva informații, și anume, în loc în descendenți, adică în noduri, să stocăm caracterul în sine, stocăm starea, pentru fiecare nod aceasta fiind un număr unic. Stocăm alăturat fiecărui nod și litera corespunzătoare lui, precum și cuvântul care se termină în acest nod, dacă există (funcția output a automatului). Astfel, toate nodurile terminale ale arborelui vor avea un cuvânt asociat, însă și nodurile interne pot avea asociat așa ceva.
2. Trebuie construită și o funcție de eșec, care reprezintă următorul lucru: când nodul x trebuie să găsească litera următoare din șir ca descendent și nu o poate face (deci șirul și cuvântul curent sunt diferite), atunci va folosi valorile din această funcție pentru a reveni la un nod ”anterior”, și se va încerca potrivirea cu descendenții acelui nod, ș.a.m.d. până ce ajugem la rădăcină. Pentru nodurile imediat descendente ale rădăcinii, funcția de eșec va conduce către nodul rădăcină, iar pentru restul nodurilor vom folosi o coadă în care le vom împinge pe rând pe toate, vizitându-le într-o anumită ordine (inordine, postordine, preordine), ca în cazul unui arbore obișnuit. Când extragem un nod din coadă, vom căuta începând de la rădăcină*, pentru fiecare din copiii lui care*

nu are descendenți, cel mai apropiat descendent care are același caracter cu caracterul copilului respectiv. După ce îl găsim, copiem cuvântul asociat (sau partea de cuvânt asociată a acestuia) ca și cuvânt asociat respectivului copil și setăm funcția de eșec pentru respectivul copil către nodul (descendentul) identificat. Dacă nu găsim respectivul descendent, atunci funcția de eșec va conduce către nodul rădăcină.

Astfel, în final, avem trei componente pentru automatul cu stări finite (în codul sursă sunt vectori care vector[x] este echivalent cu f(x)):

1. Funcția goto (denumită trie în codul sursă) – merge din nod în nod pe conexiunile trie-ului; aceasta reprezintă, în esență, trie-ul.
2. Funcția failure – funcția de eșec descrisă mai sus
3. Funcția output – conține pentru fiecare nod, adică fiecare stare, cuvântul care se ”termină” în respectivul nod, inclusiv cu actualizările de la pasul 2 de mai sus.
   Această funcție poate fi implementată mai eficient folosind un șir de caractere în care caracterul p conține 1 dacă se termină un șir acolo, 0 altfel; de asemenea, pentru a evita traversarea acestui vector sau șir de caractere, putem implementa output folosind un hash table, adică în loc să marcăm un vector după litera respectivă și să fie necesar să recăutăm prin lista de cuvinte odată ce avem o potrivire, deoarece nu știm ce potrivire exact,

```
Trie* buildTrie(char** argv, int argc, int verbose, size_t* argvlen) {
    size_t i, j;
    Trie state = 1;
    Trie current = 0;
    Queue* queue;
    Trie stat, fail;
    int t;

    memset(trie, -1, sizeof(trie));

    for (i = 1 + verbose; i < argc; ++i) {
        current = 0;
        for (j = 0; j < argvlen[i]; j++) {

            t = ((unsigned char**)argv)[i][j] - OFFSET;

            if (trie[current][t] == (Trie)-1) {
                trie[current][t] = state;
                state++;
            }
            current = trie[current][t];
        }
        output[current][i - 1 - verbose] = 1;
    }

    for (i = SCHAR_MIN; i < SCHAR_MAX + 1; ++i) {
        if (trie[0][i] == (Trie)-1) {
            trie[0][i] = 0;
        }
    }
```

```
    queue = newQueue(sizeof(Trie));
    if (queue == NULL) {
        return NULL;
    }

    memset(failure, -1, sizeof(failure));
    for (i = SCHAR_MIN; i < SCHAR_MAX + 1; ++i) {
        if (trie[0][i] != 0) {
            failure[trie[0][i]] = 0;
            enqueue(queue, &trie[0][i]);
        }
    }

    while (queue->size) {
        stat = *(Trie*)(dequeue(queue));
        for (i = SCHAR_MIN; i < SCHAR_MAX + 1; ++i) {
            if (trie[stat][i] != (Trie)-1) {
                fail = failure[stat];
                while (trie[fail][i] == (Trie)-1) {
                    fail = failure[fail];
                }
                fail = trie[fail][i];
                failure[trie[stat][i]] = fail;
                for (j = 0; j < argc - verbose - 1; j++) {
                    if (output[fail][j] == 1) {
                        output[trie[stat][i]][j] = 1;
                    }
                }
                enqueue(queue, &trie[stat][i]);
            }
        }
    }

    destroy(queue);

    return *trie;
}
```

Algoritmul

Pentru a găsi cuvinte în șir, începem traversarea trie-ului. Pentru început, căutăm să vedem dacă caracterul curent din șir are descendent în vreunul din nodurile pornind de la rădăcină. Dacă are, ne mutăm în acel nod, avansând totodată cu 1 și în text. Dacă starea curentă are output (funcția de output, apelată cu starea curentă și cuvântul curent cercetat returnează 1), atunci am găsit unul dintre cuvintele căutate și îl afișăm. Dacă nu avem descendent egal cu caracterul curent din șir, atunci folosim funcția de eșec pentru a afla unde anume trebuie să revenim în cadrul arborelui, ca stare.

```
Trie nextState(Trie currentState, Trie nextInput) {
    Trie stat = currentState;
    while (trie[stat][nextInput] == (Trie)-1) {
        stat = failure[stat];
    }
    return trie[stat][nextInput];
}
```

```
int ahocorasick(char** argv, int argc, string text,
    int verbose, int sameColour, int fromFile) {
    size_t i, j, t, match = 0, strlentext;
    size_t* argvlen;
    bool found;
    Trie currentState;

    buildTrie(argv, argc, verbose, argvlen);
    currentState = 0;
    strlentext = strlen(text);

    for (i = 0; i < strlentext; i++) {
        currentState = nextState(currentState, text[i] - OFFSET);
        found = FALSE;
        for (j = 0; j < argc - verbose - 1; j++) {
            if (output[currentState][j] == 1) {
                found = j + 1 + verbose;
                break;
            }
        }
        if (found == FALSE) {
            continue;
        }
        for (j = 1 + verbose; j < argc; j++)
        {
            if (output[currentState][j - 1 - verbose] == 1) {
                printf("Word %s appears from %d to %d.\n",
                      argv[j], (int)(i - argvlen[j] + 1), (int)i);
                match++;
            }
        }
    }
    free(argvlen);
    return match;
}
```

Complexitate

Deoarece șirul este parcurs o singură dată, avem o complexitate O(n). Apoi, pentru fiecare caracter din șir, încercăm parcurgerea arborelui, însă vom parcurge un cuvânt prin arbore, adică O(m), unde m e lungimea cuvintelor concatenate, doar atunci când l-am găsit. Folosind o structură de date eficientă în care să stocăm trie-ul, complexitatea algoritmului se poate reduce la O(n + m + z), unde z reprezintă numărul de apariții ale cuvintelor în text.

Pentru implementarea curentă, etapa de preprocesare implică construcția trie-ului, în care parcurgem fiecare cuvânt, deci O(m); de asemenea, construcția funcțiilor de eșec și output se face prin parcurgerea trie-ului, care conține, în esență, cuvintele, deci avem din nou O(m), adică complexitatea finală pentru construcție este O(m).

Din punct de vedere al spațiului ocupat, trebuie să ținem cont de spațiul ocupat de arborele trie, precum și de vectorii funcțiilor de output și failure, pe lângă vectorul care stochează textul.

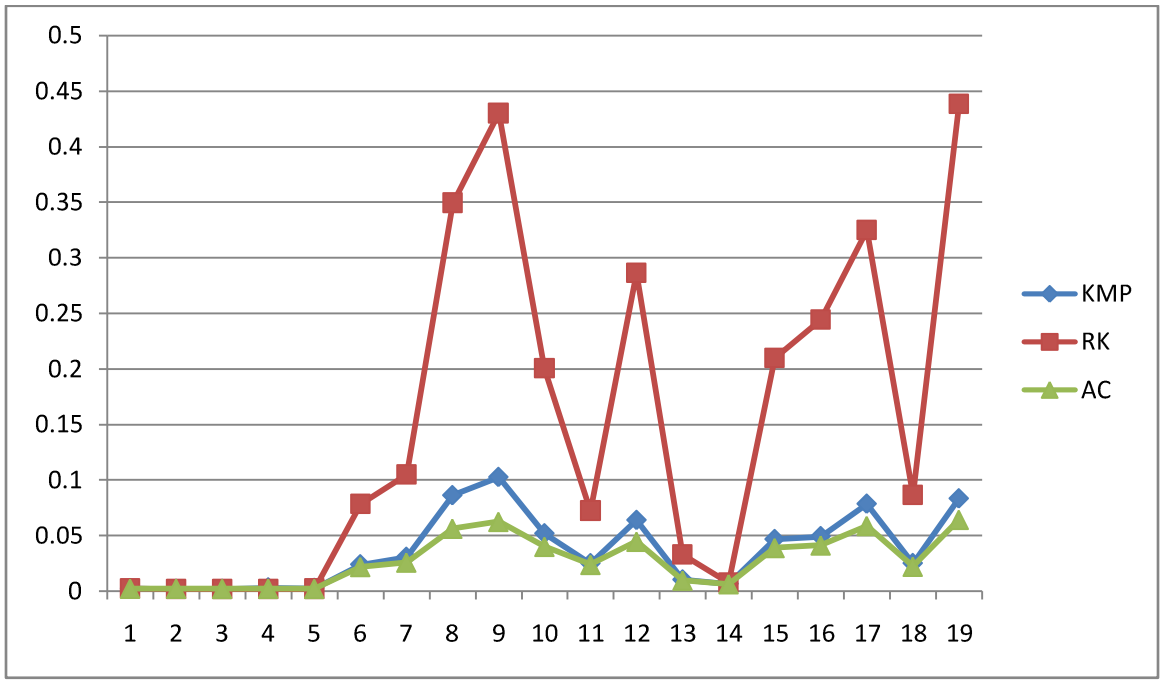
Teste sintetice

Problema Aho Corasick de pe Infoarena conține un set de 19 teste ce au fost oferite ca input pentru cei trei algoritmi. Testele au dificultăți acoperind tot spectrul uzual, adică sunt și ușoare, și medii, și dificile. Am obținut următoarele rezultate la executarea benchmark-ului.

```
====== KMP ======
Test 1.....passed (0.00265121459961)
Test 2.....passed (0.00206589698792)
Test 3.....passed (0.00205898284912)
Test 4.....passed (0.00337910652161)
Test 5.....passed (0.00288486480713)
Test 6.....passed (0.0241501331329)
Test 7.....passed (0.0309019088745)
Test 8.....passed (0.0864119529724)
Test 9.....passed (0.102813005447)
Test 10....passed (0.0520839691162)
Test 11....passed (0.0251700878143)
Test 12....passed (0.0641179084778)
Test 13....passed (0.0104959011078)
Test 14....passed (0.0067880153656)
Test 15....passed (0.0468978881836)
Test 16....passed (0.0495059490204)
Test 17....passed (0.0787560939789)
Test 18....passed (0.0252780914307)
Test 19....passed (0.0835509300232)
    Result: 19/19
====== RK ======
Test 1....passed (0.00232791900635)
Test 2....passed (0.00211501121521)
Test 3....passed (0.00202894210815)
Test 4....passed (0.00217080116272)
Test 5....passed (0.002277135849)
Test 6....passed (0.0785939693451)
Test 7....passed (0.104871988297)
Test 8....passed (0.349488019943)
Test 9....passed (0.430165052414)
Test 10...passed (0.200603961945)
Test 11...passed (0.0725238323212)
Test 12...passed (0.286458015442)
Test 13...passed (0.0332770347595)
Test 14...passed (0.00793194770813)
Test 15...passed (0.2099609375)
Test 16...passed (0.244245052338)
Test 17...passed (0.324943065643)
Test 18...passed (0.0866320133209)
Test 19...passed (0.438446998596)
   Result: 19/19
====== AC ======
Test 1....passed (0.00238299369812)
Test 2....passed (0.00235509872437)
Test 3....passed (0.0022029876709)
Test 4....passed (0.00230598449707)
Test 5....passed (0.0022189617157)
Test 6....passed (0.021947145462)
Test 7....passed (0.0259380340576)
```

```
Test 8....passed (0.0564517974854)
Test 9....passed (0.0626277923584)
Test 10...passed (0.0401501655579)
Test 11...passed (0.0238540172577)
Test 12...passed (0.0447068214417)
Test 13...passed (0.00966095924377)
Test 14...passed (0.00641798973083)
Test 15...passed (0.0390620231628)
Test 16...passed (0.0413830280304)
Test 17...passed (0.0589239597321)
Test 18...passed (0.0221920013428)
Test 19...passed (0.0644850730896)
   Result: 19/19
```

Aceste rezultate sunt evidențiate în graficul următor:



Concluzii

Concluziile furnizate de către grafic sunt elocvente: așa cum ne așteptam, în ciuda overhead-ului cu preprocesarea mai mare decât în cazul celorlalți doi algoritmi, Aho Corasick este algoritmul superior, deoarece se scalează mai bine decât ceilalți doi pentru seturi de date mari. Rabin-Karp se comport[cel mai slab, deoarece trebuie să calzuleze hash-urile textului pentru fiecare din cuvintele specificate, în vreme ce Knutt-Morris-Pratt doar trece de atâtea ori prin text, construind în schimb vectori care consumă mult mai puțin timp. Astfel, pentru seturi de date mici, algorimtii se comportă similar, pentru cele medii, KMP și AC au un comportament apropiat, deoarece câștigurile lui AC sunt compensate de overhead-ul construirii structurilor adiționale. Cu adevărat performanța acestor algoritmi se relevă pe teste comprehensive, mari, unde AC reușește să confirme teoria.

Astfel, pentru aplicații care sunt cu precădere folosite pentru a căuta un singur cuvânt în cadrul unui text, alegerea firească ar trebui să fie KMP, datorită ușurinței de implementare, precum și a diferențelor neglijabile față de AC. KMP este potrivit pentru funcționalitate de genul Ctrl+F (găsire în cadrul unui document). Pentru căutări mai complexe, atunci când dorim să identificăm mai multe cuvinte în cadrul unui text deosebit de mare, eventual și cuvinte de dimensiuni mari, singura alegere disponibilă din cele trei este Aho Corasick, acesta oferind performanță bună. Folosind diverse optimizări, aceasta poate fi crescută și mai mult.

Alte mențiuni

Analiza complexității și punctelor critice din implementări am efectuat-o utilizând și aplicația valgrind, alături de modulul acesteia callgrind și modulul de interpretare a datelor gprof2dot. Implementarea trie-ului pentru AC constituie și baza implementării unei soluții care să acopere implementarea Trie.

Referințe

[1] Knuth–Morris–Pratt(KMP) Pattern Matching(Substring search) - https://www.youtube.com/watch?v=GTJr8OvyEVQ

[2] Aho-Corasick – implementation and animation - http://blog.ivank.net/aho-corasick-algorithm-in-as3.html

[3] Aho-Corasick - http://www.infoarena.ro/problema/ahocorasick

[4] Potrivirea sirurilor - http://www.infoarena.ro/problema/strmatch

[5] Aho-Corasick Algorithm for Pattern Searching - http://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/

[6] Trie | (Insert and Search) - http://www.geeksforgeeks.org/trie-insert-and-search/

[7] Aho-Corasick algorithm - https://en.wikipedia.org/wiki/Aho%E2%80%93Corasick_algorithm

[8] – Implementing a generic Queue in C - https://codereview.stackexchange.com/questions/141238/implementing-a-generic-queue-in-c

[9] Rabin-karp algorithm - https://en.wikipedia.org/wiki/Rabin%E2%80%93Karp_algorithm

[10] Knutt-Morris-Pratt algorithm - https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

[11] Aho Corasick Build Tree, Fail, Matching - https://www.youtube.com/watch?v=ePafMI_rSJg&t=1s, https://www.youtube.com/watch?v=qPyhPXPl3T4, https://www.youtube.com/watch?v=IcXimoT_YXA&t=8s