

A study of search algorithms

Coordonator
Lect. Univ. Dr. Sabo Cosmin

Student
Câmpan Ștefan Ioan

Abstract—When searching for something in a data structure it is important to know what algorithm is best suited for each individual search. There is no way to satisfy every possible search with one generic method. The study aims to present a comprehensive analysis of different search algorithms, some of the main usages of those algorithms as well as some security issues that may arise with some of them and possible solutions.

I. INTRODUCERE

A. Scopul proiectului

Scopul acestei lucrări este analiza modului de funcționare a diversi algoritmii de căutare, începând de la cei mai simplii și ajungând la cei mai complecși.

B. Obiective

Principalul obiectiv al proiectului este de a analiza functionarea algoritmilor prezentăți.

Ca și obiective secundare, se mai evidențiază unele dintre aplicațiile principale ale algoritmilor, precum și unele probleme de securitate ce pot surveni în cazul unora dintre acești algoritmi, totodată prezintând posibile soluții la acestea.

II. FUNDAMENTARE TEORETICĂ

Prezenta lucrare se bazează pe studiul a mai multor articole și cărți, una dintre cele mai importante fiind [1].

În primul rând este important să știm sau să putem defini ceea ce căutăm, pe urmă să definim sau să cunoaștem spațiul în care căutăm. Algoritmii de căutare necesită în general o structură sau o colecție de date sau elemente ordonate. Astfel, se poate răspunde la întrebarea cum căutăm?

Nu este indicat a se încerca căutarea unui număr într-un tablou de caractere. De asemenea, trebuie stabilit și mediul prin care se efectuează căutarea.

În acest scop, implementarea și analiza algoritmilor în prezenta lucrare este realizată în limbajul Python, versiunea 3.8.1.

III. CĂUTAREA SECVENTIALĂ



Fig. 1.

CĂUTAREA SECVENTIALĂ este cea mai simplă formă de căutare. Presupune parcurgerea unei colecții C , element cu element, până se găsește elementul t căutat, iar atunci se

întoarce poziția la care se află acesta, sau până când se ajunge la finalul colecției, caz în care se întoarce valoarea -1 pentru a semnifica aceasta.

În figura 1, pentru a găsi elementul cu valoarea "24", este nevoie de 10 iterări.

Timpul de execuție în acest caz, folosind notația Big O, este $O(n)$, unde n reprezintă numărul de elemente.

Acest tip de căutare este de obicei folosit pentru colecții de dimensiuni mici sau atunci când eficiența nu este necesară.

IV. CĂUTAREA BINARĂ

CĂUTAREA BINARĂ [1], se bazează pe principul înjumătățirii intervalului în care se efectuează căutarea.

De exemplu, în figura 1, dacă se caută poziția la care se găsește valoarea "17", intervalul inițial de căutare este 0 - 9. Valoarea care se verifică este cea de la mijlocul intervalului, în cazul acesta "9". Dacă valoarea căutată este mai mare decât "9", intervalul de căutare devine 5 - 9, iar dacă ar fi fost mai mică, intervalul de căutare ar fi devenit 0 - 3.

Acum se verifică din nou valoarea de la mijlocul noului interval, "17", fiind valoarea căutată, poziția la care se află fiind 7, iar algoritmul se sfărșește.

Dacă valoarea t căutată nu se află în colecția C , atunci se întoarce valoarea -1.

Timpul de execuție pentru o căutare binară este $O(\log n)$. O îmbunătățire a acestui algoritm o reprezintă căutarea prin interpolată. În [2], Yehoshua Perl et al. demonstrează că timpul de execuție pentru o astfel de căutare este $O(\log \log n)$.

V. CĂUTAREA BAZATĂ PE HASH

	<hash	cheie	valoare >
0			
1			...
.			...
i			...
.			...
n			...

CĂUTAREA BAZATĂ PE HASH [1], [3], sau căutarea într-un dicționar, este folositoare atunci când elementul pe care îl căutăm face parte dintr-o colecție C de dimensiuni foarte mari, cum ar fi o bază de date, iar elementele acestei colecții nu sunt neapărat ordonate. În capitolul 5.4 din [1], Heineman et al. folosesc o implementare a unui dicționar bazat pe liste.

Un dicționar funcționează pe principiul cheie - valoare. Având o colecție C , fiecare element $e \in C$ este asociat cu o cheie unică, k , astfel încât $k_i \neq k_j$. Asupra cheii se aplică o funcție de hash, $\text{hash}(k)$, iar valoarea obținută este folosită pentru a indexa elementul e_i și cheia k_i într-un dicționar D .

index = hash(k) % "lungime dicționar"

În mod ușor, un dicționar oferă o eficiență de $O(1)$. Însă și cum se arată în [4], [5], în implementarea prezentată mai sus eficiența unui dicționar în cazul inserării de elemente poate fi redusă la $O(n^2)$, prin exploatarea coliziunilor ce pot să apară la folosirea funcției hash(), ceea ce poate fi foarte periculos pentru aplicațiile și serverele web.

Această problema este evitată în Python¹ prin conceptul de adresare deschisă și a modului în care este structurat și implementat dicționarul.

VI. ARBORE BINAR DE CĂUTARE

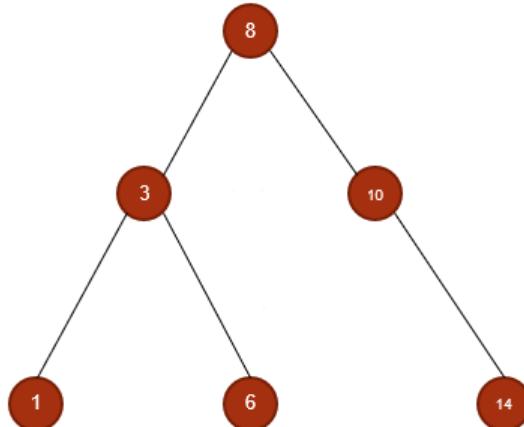


Fig. 2.

În [1], Heineman et al. evidențiază faptul că pentru colecții de elemente în care numărul acestora se poate modifica oricând, iar operațiile de inserare și de ștergere sunt frecvente, căutarea binară nu este eficientă.

În locul acesteia este mult mai dezirabil să se folosească un arbore binar de căutare.

Pentru ca un arbore să fie considerat un arbore binar de căutare, acesta trebuie să îndeplinească simultan următoarele condiții, pornind de la un nod oarecare:

- subarborele stâng al nodului să conțină doar valori mai mici decât ale acestuia
- subarborele drept al nodului să conțină doar valori mai mari decât ale acestuia

Pe lângă acestea, este de asemenea dezirabil ca arborele să fie echilibrat, adică, pornind de la un nod oarecare, diferența dintre lungimea oricărui subarbore nu poate fi mai mare de 1. Astfel, arborele binar devine un arbore AVL, propus de Adel'son-Vel'skii și Landis în anul 1962.

Arborii binari de căutare pot fi folosiți în contextul prevenirii surgerilor de memorie, având o eficiență de $O(\log n)$, atât pentru căutare cât și pentru inserarea sau ștergerea elementelor.

¹Implementarea dicționarului în Python, <https://github.com/python/cpython/blob/master/Objects/dictobject.c>

VII. CĂUTAREA ÎN ADÂNCIME

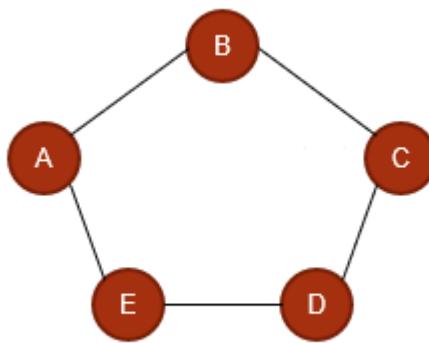


Fig. 3.

Grafurile sunt o structură de date utilizată des în diverse domenii precum logistica, rețelele de calculatoare, jocuri etc. Un graf $G = (N, M)$ este definit de un set de noduri N și un set de muchii M între perechi de noduri, care pot fi orientate sau nu sau pot să indice anumite proprietăți, cum ar fi de exemplu în figura 3.

În practică se folosesc mai multe opțiuni de a reprezenta un graf în memorie, cum ar fi matricea de adiacență, lista de adiacență sau dicționalele. Alegera depinde de problema și cazul specific în care se folosește graful.

Algoritmul de căutare în adâncime a unui drum între două noduri, A și C din figura 3 de exemplu, pornind de la nodul A, se bazează pe utilizarea principiului ultimul venit, primul servit, în spătă o stivă.

Trebuie avut în vedere faptul că acest algoritm nu găsește întotdeauna drumul cel mai scurt. În cazul de față, drumul găsit de algoritm între nodul A și nodul C este $A \rightarrow E \rightarrow D \rightarrow C$, deși există un drum mai scurt și anume $A \rightarrow B \rightarrow C$.

Eficiența acestui algoritm este $O(N + M)$, unde N este numărul de noduri, iar M este numărul de muchii.

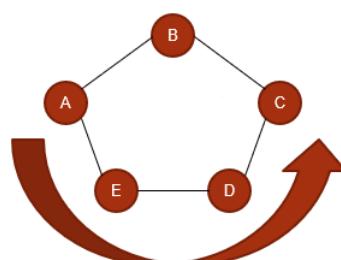


Fig. 4.

VIII. CĂUTAREA ÎN LĂTIME

Folosind exemplul de la secțiunea precedentă, algoritmul de căutare în lătime va găsi întotdeauna cel mai scurt drum între nodurile A și C, anume $A \rightarrow B \rightarrow C$.

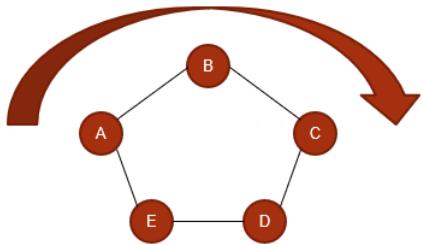


Fig. 5.

Spre deosebire de căutarea în adâncime, acest algoritm utilizează principiul primul venit, primul servit, în spate o coadă.

În mod similar cu algoritmul din setiunea VII, algoritmul de căutare în lătime are o eficiență de $O(N + M)$.

IX. CONCLUZII

Algoritmii prezentati sunt fundamentele multor alte metode, programe și algorimi mult mai complecsi, inclusiv inteligențele artificiale. Studiul lor oferă posibilitatea unei mai bune cunoașteri a problemelor ce pot să apară în domeniul IT, precum și o înțelegere a metodelor prin care acestea sunt rezolvate.

X. ANEXĂ

Mai jos se găste codul sursă al algoritmilor prezentati.

```
def cautare_sequentiala(tab, val):
    for i in range(len(tab)):
        if tab[i] == val:
            return i
    return -1
```

Fig. 6.

```
def cautare_binara(tab, val):
    st = 0
    dr = len(tab) - 1
    while st <= dr:
        mij = st + (dr - st)//2

        if tab[mij] == val:
            return mij
        elif val > tab[mij]:
            st = mij + 1
        else:
            dr = mij - 1

    return -1
```

Fig. 7.

```
def search(node, val):
    if node is None or node.key == val:
        return node
    if node.key > val:
        return search(node.left, val)
    return search(node.right, val)
```

Fig. 8.

```
def dfs(self, nod_s, nod_f):
    stack = [nod_s]
    nod_s.culoare = "gri"
    parinte = {}
    while stack:
        curent = stack.pop()
        if curent is not nod_f:
            for vecin in curent.vecini:
                if self.noduri[vecin].culoare != "gri":
                    self.noduri[vecin].culoare = "gri"
                    stack.append(self.noduri[vecin])
                    parinte[self.noduri[vecin]] = curent
        else:
            return self.get_drum(nod_s, curent, parinte)
    return None

def get_drum(self, nod_s, nod_f, parinte):
    drum = [nod_f.numar]
    while nod_f != nod_s:
        nod_f = parinte[nod_f]
        drum.insert(0, nod_f.numar)
    return drum
```

Fig. 9.

```
def bfs_path(self, nod_s, nod_f):
    coada = [nod_s]
    nod_s.culoare = "gri"
    parinte = {}
    while coada:
        curent = coada.pop(0)
        if curent is not nod_f:
            for vecin in curent.vecini:
                if self.noduri[vecin].culoare != "gri":
                    self.noduri[vecin].culoare = "gri"
                    coada.append(self.noduri[vecin])
                    parinte[self.noduri[vecin]] = curent
        else:
            return self.get_drum(nod_s, nod_f, parinte)
    return None

@staticmethod
def get_drum(nod_s, nod_f, parinte):
    drum = [nod_f.numar]
    while nod_f != nod_s:
        nod_f = parinte[nod_f]
        drum.insert(0, nod_f.numar)
    return drum
```

Fig. 10.

REFERENCES

- [1] G. Pollice, S. Selkow, and G. T. Heineman, *Algorithms in a Nutshell*. O'Reilly Media, 2008.
- [2] Y. Perl, A. Itai, and H. Avni, “Interpolation search—a log log n search,” *Communications of the ACM*, vol. 21, no. 7, pp. 550–553, 1978.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.
- [4] A. Klink and J. Walde, “Efficient denial of service attacks on web application platforms,” in *28th Chaos Communication Congress*, 2011.
- [5] S. A. Crosby and D. S. Wallach, “Denial of service via algorithmic complexity attacks.,” in *USENIX Security Symposium*, pp. 29–44, 2003.