String Algoritma: Suffix Tree Uygulaması

Muhammet Cüneyd Kurtbaş
Kocaeli Üniversitesi
Bilgisayar Mühendisliği (I.Öğretim)
200201132
cuneydkurtbas@gmail.com

***Abstract*—Suffix tree veri yapısı bir dizgi model (pattern) eşleştirme algoritmasıdır. Örneğin elinizde uzun bir dizgi olsun ve siz bu dizgi içinde alt dizgiler aramak ve hatta bu dizgilerden kaç adet bulunduğunu öğrenmek istiyorsunuz. İşte bu veri yapısı bu işlemleri kolaylaştırmak ile birlikte gayet hızlı işlem yapmamıza olanak sağlıyor.**

Bu tip bir yöntem kullanmadan ilkel olarak çözüm olarak şunu yapabilirdik. Ana dizgi içerisinde lineer olarak sırayla karşılaştırma yaparak devam eder ve sonuca ulaşabilirdik. Bu bir yöntemdir fakat dizgi boyutları arttıkça bu işlem çok maliyetli olmaktadır.

***Index Terms*—suffix, tree, algorithm, string, searching, bioinformatic**

I. GİRİŞ

Bu uygulamanın amacı sonek ağaçlarını ve sonek dizililerini kullanarak katarlar üzerinde bazı arama işlemleri yapmaktır. Bir katar için sonek, katarın herhangi bir karakterinden başlayarak sonuna kadar olan kısımdır. Bir kelimenin öneki ise kelimenin ilk karakterinden başlayarak herhangi bir karakterine kadar olan kısımdır.

II. YÖNTEM

Bir katarın (p) başka bir katar (s) içinde bulunması aslında p'nin s'in herhangi bir sonekinin öneki olmasına bağlıdır. Örnek olarak **al** katarı **galatasaray** katarının içinde bulunup bulunmadığı bulmak için **galatasaray** katarının tüm sonekleri oluşturulur ve **al** katarının bu soneklerin herhangi birinin öneki olup olmadığına bakılır. Aşağıda listelenen soneklere bakıldığında **al** katarı ikinci sonekin önekidir.

1. galatasaray, **2. alatasaray**, 3. latasaray, 4. atasaray, 5.tasaray, 6. asaray, 7. saray, 8. aray, 9. ray, 10. ay, 11. y

Görüldüğü gibi n uzunluğunda bir dizgi n adet suffix sahibidir. Suffix tree oluşturmak için ilk olarak suffixleri bulmamız gerekmektedir. Yukarıdaki örnekteki gibi suffixler elde edilerek, bunlar uzunluklarına göre sıralanmalıdır. Sonra sırası ile 1'den başlayarak tüm suffixler numaralandırılarak ağaç oluşturulur. Başlangıç noktasından dallanmalar yapılmaktadır. Önce 1 numara dallanır daha sonra gelen 2 numaraya ait dizgi ile ortak başlangıç noktaları varsa bu noktadan itibaren dallanma yapılır eğer yoksa yeni bir dallanma yapılır.

III. UYGULAMA

Bu uygulamada **mehmet** katarını ele alalım. Sonekler: 1.mehmet, 2.ehmet, 3.hmet, 4.met, 5.et, 6.t olacaktır.

İlk dallanmamızı 1 numaralı suffix ile yapıyoruz.

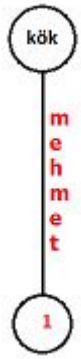


Fig. 1.

2 numaralı suffix olan "ehmet"' için inceleme yapıldığında şu ana kadar başlangıç noktasından itibaren "e" ile başlayan bir dallanma olmadığından yeni bir dallanma yapılır.

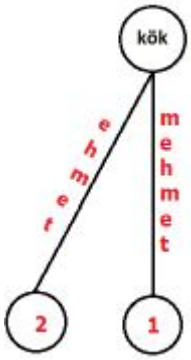


Fig. 2.

3 numaralı suffix "hmet"' için inceleme yapıldığında "h" ile başlayan bir dallanma olmadığından tekrar başlangıç noktamızdan yeni bir dallanma yapılır.

5 numaralı suffix "et" için inceleme yapıldığında tekrar kök noktasından taramaya başlıyoruz ve ilk olarak "e" başlangıcı arıyoruz. Görüldüğü gibi en iyi kesişim 2 numaralı suffixtedir.

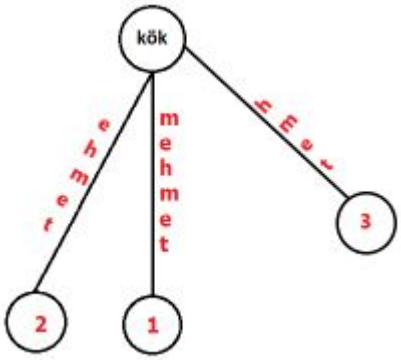


Fig. 3.

4 numaralı suffix ”met”’ için inceleme yapıldığında bu noktada durum değişmektedir. Ağaca baktığımızda başlangıç noktasından itibaren bir ”m” ile başlangıç vardır. Hatta bir adım daha arama yaptığımızda sadece ”m” ile değil ”me” ile bir başlangıç olan dallanma vardır (1 numaralı suffix). Dolayısı ile kesişme noktasından bir dallanma yapılır.

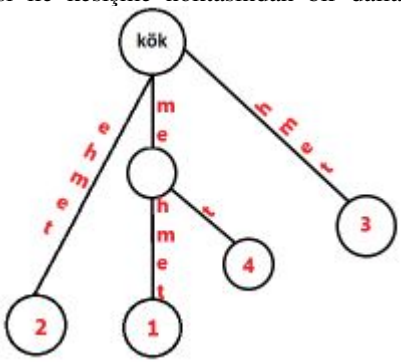


Fig. 4.

Dolayısı ile bir kesişim noktası yapıp yeni bir dallanma yaratıyoruz. (Fig 5.)

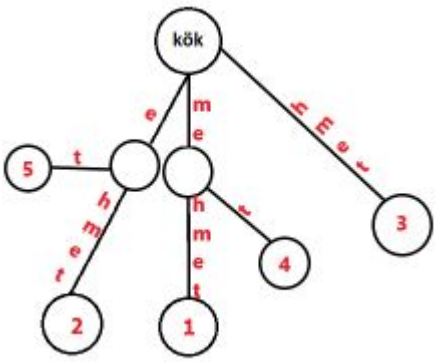


Fig. 5.

Son olarak 6 numaralı suffix olan ”t” için incelemede; tek başına bir dallanma yapması gerektiği görülüyor. (Fig 6.)

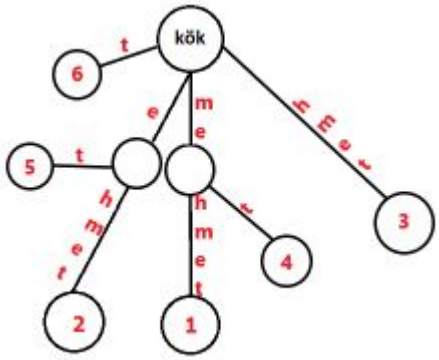


Fig. 6.

C KODLARI

```
struct SuffixTreeNode {
    struct SuffixTreeNode *children[MAX_CHAR];
    struct SuffixTreeNode *suffixLink;
    int start;
    int *end;

    int suffixIndex;
};
Node *newNode(int start, int *end)
{
    count++;
    Node *node =(Node*) malloc(sizeof(Node));
    int i;
    for (i = 0; i < MAX_CHAR; i++)
        node->children[i] = NULL;

    node->suffixLink = root;
    node->start = start;
    node->end = end;
    node->suffixIndex = -1;
    return node;
}
```

```
int edgeLength(Node *n) {
    return *(n->end) - (n->start) + 1;
}

int walkDown(Node *currNode)
{
    if (activeLength >= edgeLength(currNode))
    {
        activeEdge =
        (int)text[activeEdge+edgeLength(currNode
        activeLength -= edgeLength(currNode);
        activeNode = currNode;
        return 1;
    }
    return 0;
}
```

```
void buildSuffixTree()
{
    size = strlen(text);
    int i;
    rootEnd = (int*) malloc(sizeof(int))
    *rootEnd = - 1;

    root = newNode(-1, rootEnd);

    activeNode = root;
    for (i=0; i<size; i++)
        extendSuffixTree(i);
    int labelHeight = 0;
    setSuffixIndexByDFS(root, labelHeigh

    freeSuffixTreeByPostOrder(root);
}
```

```
void freeSuffixTreeByPostOrder(Node *n)
{
    if (n == NULL)
        return;
    int i;
    for (i = 0; i < MAX_CHAR; i++)
    {
        if (n->children[i] != NULL)
        {
            freeSuffixTreeByPostOrder(n->children[i]);
        }
    }
    if (n->suffixIndex == -1)
        free(n->end);
    free(n);
}
```

```
int sonekYarat()
{
    int x=300, y=100, j, i;
    int index=0,n=0,m=0;
    int gez=0;
    char arr[12][12];
    for(i=0;i<parcalananDiziSayisi;i++)
    {
        m=0;
        if(siraliYaprakDizisi[i]==-1)
        {
            strcpy(arr[gez],yapraktakiVeri[i+1]);
            gez++;
            strcpy(arr[gez],yapraktakiVeri[i+2]);
            gez++;
            if(i+2<=6)
            {
                for(j=i+3;j<i+5;j++)
                {
                    for(int p=0;p<2;p++)
                    {
                        if(strcmp(arr[p],yapraktakiVeri[j])==0)
                        {
                            yaprak[index]=gez ;
                            m=1 ;
                        }
                        else
                        {
                            n=1 ;
                        }

                    }
                    if(m==0)
                    {
                        gez++ ;
                        yaprak[index]=gez ;
                    }
                }
                index++;
                gez=0;
```

IV. DENEYSEL SONUÇLAR

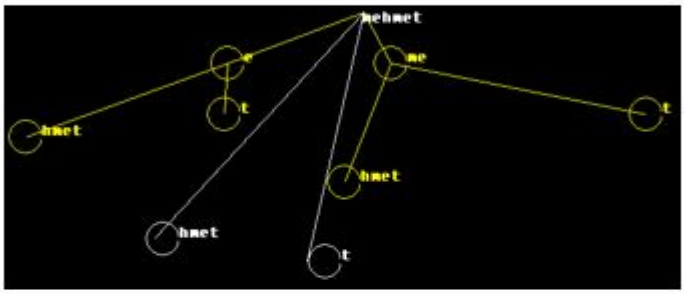
Katarlar ne kadar uzun olursa olsun sınırlı sayıda farklı karakterin birleşmesiyle oluşur. Soneklere dayalı bazı veri yapıları kullanılarak arama işlemi daha düşük maliyetle gerçekleştirilir.

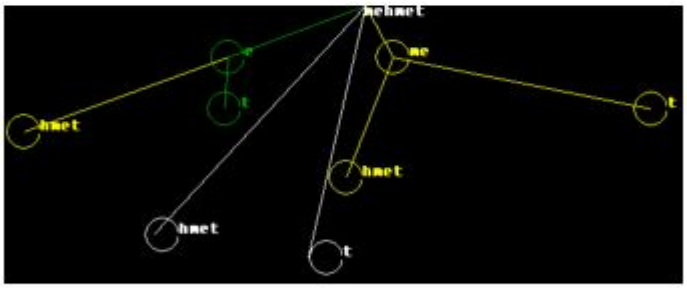
n uzunluklu s katarın sonek ağacı aşağıdaki özelliklere sahiptir:

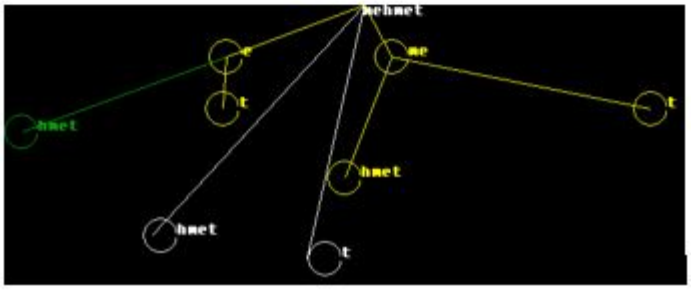
- Ağacın 1'den n'e kadar numaralandırılmış n adet yaprağı vardır.
- Kök dışında her düğümün en az iki çocuğu vardır.
- Her kenar s'in boş olmayan bir alt katarı ile etiketlenir.
- Aynı düğümden çıkan kenarların etiketleri farklı karakter ile başlamalıdır.
- Kökten başlayıp k. yaprağa giden yoldaki kenarların etiklerinin birleştirilmesi ile k. sonek elde edilir.

Uygulama, Windows işletim sistemi üzerinde Dev C++ IDE'si kullanılarak C dilinde gerçeklenmiştir. Uygulama sonuçlarına ait Ekran Görüntüleri;

```
.....................................................
              SUFFIX TREE ISLEM MENUSU
.....................................................
* S KATARI ICIN SONEK AGACI OLUSTURULABILIR MI? << 1 >>

* S KATARI ICINDE P KATARI VAR MI?              << 2 >>

* S KATARI ICINDE TEKRAR EDEN EN UZUN ALT KATAR?<< 3 >>

* S KATARI ICINDE EN COK TEKRAR EDEN AL KATAR?  << 4 >>

.....................................................
       ISLEM NUMARASINI GIRINIZ: 1
```







References

- Geeksforgeeks: Generalized Suffix Tree
- Stackoverflow: Ukkonen's suffix tree algorithm
- Wikipedia: suffix tree
- Youtube Playlist: Suffix Tree using Ukkonen's algorithm:
- Blog: koseburak.net/blog/suffix-tree/
- Stanford University
- cs.cmu.edu/ ckingsf/bioinfo-lectures/suffixtrees.pdf
- Overleaf: New to LaTeX?