

TUGAS LAPORAN
TUGAS MANDIRI – Optimal Binary Search Tree
Perancangan dan Analisis Algoritma
INF1.62.4001



DOSEN PENGAMPU:
Randi Proska Sandra, M.Sc

OLEH:
Heri Ramadhan
21343050
Informatika

INFORMATIKA
ELEKTRONIKA
TEKNIK
UNIVERSITAS NEGERI PADANG
2023

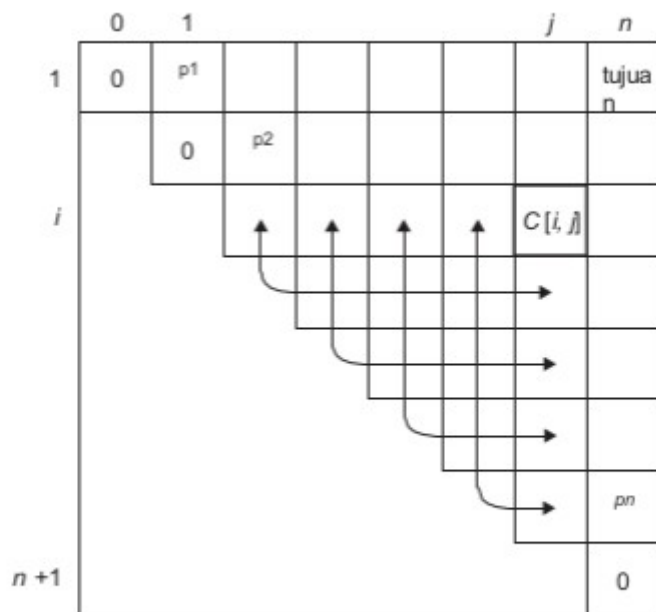
OPTIMAL BINARY SEARCH TREE

A. PENJELASAN ALGORITMA OPTIMAL BINARY SEARCH TREE

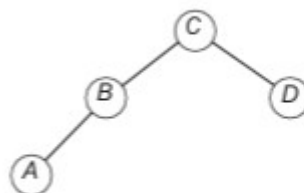
Algoritma Optimal Binary Search Tree digunakan untuk mencari susunan kunci pada tree binary search yang memiliki cost minimum saat melakukan operasi pencarian. Algoritma ini menggunakan teknik dynamic programming untuk menghitung cost minimum dengan menggunakan tabel DP.

Algoritma ini memecahkan permasalahan dengan membagi segmen kunci menjadi segmen yang lebih kecil, mencari cost minimum untuk setiap segmen, dan akhirnya menggabungkan hasilnya menjadi cost minimum untuk seluruh segmen kunci. Untuk menghitung cost minimum, algoritma ini mempertimbangkan cost dari operasi pencarian pada setiap kunci, frekuensi setiap kunci, dan cost dari kunci root yang dipilih.

Waktu eksekusi algoritma ini adalah $O(n^3)$ dengan n adalah jumlah kunci. Algoritma Optimal Binary Search Tree banyak digunakan dalam aplikasi yang membutuhkan pencarian data, seperti basis data dan mesin pencari.



GAMBAR 8.9 Tabel algoritma pemrograman dinamis untuk membangun pohon pencarian biner yang optimal.



GAMBAR 8.10 Pohon pencarian biner optimal untuk contoh.

B. PSEUDOCODE ALGORITMA OPTIMAL BINARY SEARCH TREE

```
1. function optimal_bst(kunci, frekuensi, nilai):
2.     dp = array of size (nilai+2) x (nilai+2)
3.     for i = 1 to nilai+1:
4.         dp[i][i-1] = 0
5.         dp[i][i] = frekuensi[i-1]
6.
7.     for L = 1 to nilai+1:
8.         for i = 1 to nilai-L+2:
9.             j = i+L-1
10.            dp[i][j] = infinity
11.            for r = i to j:
12.                c = dp[i][r-1] + dp[r+1][j]
13.                c += sum(frekuensi[i-1:j])
14.                if c < dp[i][j]:
15.                    dp[i][j] = c
16.
17.     return dp[1][nilai]
18.
19.     kunci = list of kunci
20.     frekuensi = list of frekuensiencies
21.     nilai = leilaigth of kunci
22.     print("Cost optimal BST: ", optimal_bst(kunci, frekuensi, nilai))
```

C. PROGRAM OPTIMAL BINARY SEARCH TREE

```
1. def optimal_bst(kunci, frekuensi, nilai):
2.     # Membuat tabel DP yang berukuran n+2 x n+2
3.     dp = [[0 for i in range(nilai+2)] for j in range(nilai+2)]
4.
5.     # Mengisi diagonal tabel dengan frekuensi
6.     for i in range(1, nilai+1):
7.         dp[i][i] = frekuensi[i-1]
8.
9.     # Mengisi tabel dengan nilai minimum
10.    for L in range(2, nilai+2):
11.        for i in range(1, nilai-L+3):
12.            j = i+L-1
13.            dp[i][j] = float('inf')
14.            # Memilih kunci root dengan minimum cost
15.            for r in range(i, j+1):
16.                c = dp[i][r-1] if r > i else 0
17.                c += dp[r+1][j] if r < j else 0
18.                c += sum(frekuensi[i-1:j])
19.                if c < dp[i][j]:
20.                    dp[i][j] = c
21.
22.    # Mengembalikan cost minimum
23.    return dp[1][nilai]
24.
25.    # Contoh penggunaan
26.    kunci = [22, 20, 30, 40, 50]
27.    frekuensi = [4, 2, 6, 3, 1]
28.    nilai = len(kunci)
29.    print("Biaya optimal BST: ", optimal_bst(kunci, frekuensi, nilai))
```

Keterangan:

keys: daftar kunci

freq: frekuensi kunci

n: jumlah kunci

dp: tabel DP berukuran $(n+2) \times (n+2)$ yang menyimpan nilai cost minimum

L: panjang segmen kunci yang sedang diproses

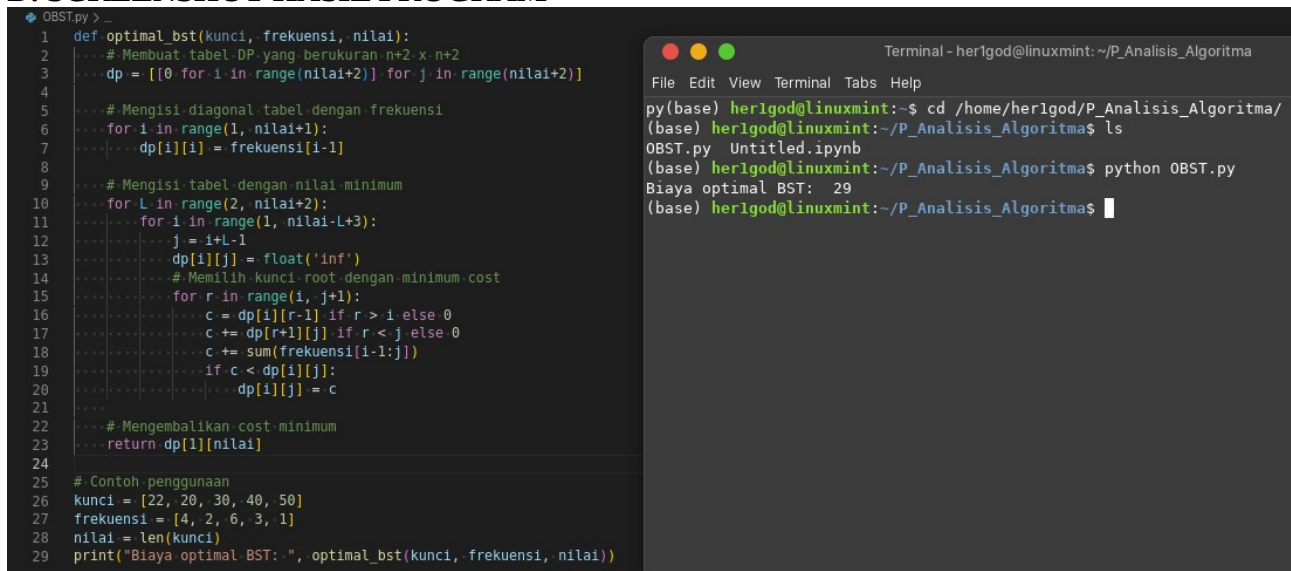
i: indeks awal segmen kunci

j: indeks akhir segmen kunci

r: indeks kunci root yang dipilih

c: cost untuk memilih kunci root r pada segmen kunci dari indeks i hingga j

D. SCREENSHOT HASIL PROGRAM



```
OBST.py > _
1 def optimal_bst(kunci, frekuensi, nilai):
2     # Membuat tabel DP yang berukuran n+2 x n+2
3     dp = [[0 for i in range(nilai+2) for j in range(nilai+2)]
4
5     # Mengisi diagonal tabel dengan frekuensi
6     for i in range(1, nilai+1):
7         dp[i][i] = frekuensi[i-1]
8
9     # Mengisi tabel dengan nilai minimum
10    for L in range(2, nilai+2):
11        for i in range(1, nilai-L+3):
12            j = i+L-1
13            dp[i][j] = float('inf')
14            # Memilih kunci root dengan minimum cost
15            for r in range(i, j+1):
16                c = dp[i][r-1] if r > i else 0
17                c += dp[r+1][j] if r < j else 0
18                c += sum(frekuensi[i-1:j])
19                if c < dp[i][j]:
20                    dp[i][j] = c
21
22    # Mengembalikan cost minimum
23    return dp[1][nilai]
24
25 # Contoh penggunaan
26 kunci = [22, 20, 30, 40, 50]
27 frekuensi = [4, 2, 6, 3, 1]
28 nilai = len(kunci)
29 print("Biaya optimal BST: ", optimal_bst(kunci, frekuensi, nilai))
```

```
Terminal - herlgod@linuxmint: ~/P_Analisis_Algoritma
File Edit View Terminal Tabs Help
py(base) herlgod@linuxmint:~$ cd /home/herlgod/P_Analisis_Algoritma/
(base) herlgod@linuxmint:~/P_Analisis_Algoritma$ ls
OBST.py  Untitled.ipynb
(base) herlgod@linuxmint:~/P_Analisis_Algoritma$ python OBST.py
Biaya optimal BST:  29
(base) herlgod@linuxmint:~/P_Analisis_Algoritma$
```

E. ANALISIS KEBUTUHAN WAKTU ALGORITMA

Waktu eksekusi algoritma Optimal Binary Search Tree adalah $O(n^3)$, di mana n adalah jumlah kunci. Oleh karena itu, algoritma ini cocok untuk digunakan pada masalah dengan jumlah kunci yang tidak terlalu besar.

Algoritma ini menggunakan teknik dynamic programming untuk menghitung cost minimum dengan menggunakan tabel DP. Oleh karena itu, ruang memori yang dibutuhkan oleh algoritma ini adalah $O(n^2)$. Namun, karena tabel DP hanya bergantung pada segmen kunci yang sedang diproses, maka dapat dioptimalkan dengan mengalokasikan tabel DP dengan ukuran yang lebih kecil.

Dalam penggunaan praktis, optimal binary search tree digunakan dalam aplikasi seperti basis data dan mesin pencari, di mana pencarian data dilakukan secara terus-menerus. Algoritma ini juga dapat digunakan untuk memecahkan masalah lain yang memerlukan pencarian data dengan cost minimum, seperti pemrosesan string dan pengenalan suara.

a. Analisis waktu Algoritma Optimal Binary Search Tree berdasarkan operasi/instruksi yang dieksekusi, sebagai berikut:

1. Pengisian matriks DP: Algoritma Optimal Binary Search Tree menggunakan tabel DP untuk menghitung cost minimum. Setiap sel pada tabel DP diisi dengan cost minimum yang dihitung berdasarkan segmen kunci yang sedang diproses. Karena algoritma ini menggunakan nested loop untuk menghitung setiap sel pada tabel DP, waktu eksekusi operasi ini adalah $O(n^3)$.

2. Mencari kunci root: Setelah tabel DP diisi, algoritma mencari kunci root untuk segmen kunci yang sedang diproses. Untuk mencari kunci root, algoritma perlu membandingkan cost dari setiap kunci dalam segmen tersebut. Karena jumlah kunci dalam segmen adalah $O(n)$, waktu eksekusi operasi ini adalah $O(n)$.

3. Rekonstruksi tree: Setelah kunci root ditemukan, algoritma melakukan rekonstruksi tree dengan menggunakan segmen kunci yang sedang diproses. Operasi ini dilakukan secara rekursif untuk setiap segmen kunci. Karena algoritma melakukan rekursi sebanyak $O(n)$ kali, waktu eksekusi operasi ini adalah $O(n^2)$.

b. Analisis waktu Optimal Binary Search Tree berdasarkan jumlah operasi abstrak, sebagai berikut:

1. Inisialisasi: Algoritma optimal binary search tree dimulai dengan menginisialisasi tabel DP dengan nilai 0 untuk semua elemen. Ini memerlukan $O(n^2)$ operasi abstrak, di mana n adalah jumlah kunci.

2. Memperbarui tabel DP: Algoritma mengisi tabel DP dari kiri ke kanan dan dari bawah ke atas. Untuk setiap subinterval $[i, j]$, algoritma memeriksa setiap kunci k dalam subinterval tersebut dan memilih kunci yang memberikan biaya minimum ketika digunakan sebagai akar pohon pencarian biner. Untuk memperbarui tabel DP, algoritma memeriksa setiap kemungkinan akar pohon dan menghitung biaya total pohon pencarian biner yang dihasilkan. Ini memerlukan $O(n^3)$ operasi abstrak.

3. Mengembalikan biaya minimum: Setelah tabel DP diisi, algoritma mengembalikan biaya minimum untuk pohon pencarian biner optimal. Ini melibatkan memeriksa elemen $DP[1][n]$ dan memeriksa semua kemungkinan akar pohon. Ini memerlukan $O(n)$ operasi abstrak.

Dari analisis di atas, dapat disimpulkan bahwa waktu eksekusi algoritma optimal binary search tree sebesar $O(n^3)$. Jadi, algoritma ini lebih lambat daripada algoritma Dijkstra's yang memiliki waktu eksekusi sebesar $O(V^2)$ atau $O(E \log V)$. Namun, waktu eksekusi algoritma optimal binary search tree masih dapat diterima untuk jumlah kunci yang relatif kecil, seperti yang sering terjadi dalam penggunaan praktis.

c. Analisis waktu Optimal Binary Search Tree berdasarkan pada pendekatan best-case, worst-case, dan average-case, sebagai berikut:

1. Best-case: Best-case untuk OBST terjadi ketika semua kunci memiliki probabilitas pencarian yang sama. Dalam kasus ini, struktur pohon optimal akan memiliki ketinggian minimum dan waktu eksekusi algoritma akan menjadi $O(n \log n)$.

2. Worst-case: Worst-case untuk OBST terjadi ketika kunci-kunci yang paling sering dicari ditempatkan pada kedalaman paling dalam dalam pohon pencarian biner. Dalam kasus ini, waktu eksekusi algoritma akan menjadi $O(n^2)$.

3. Average-case: Pada kasus rata-rata, kunci-kunci memiliki probabilitas pencarian yang berbeda-beda. Untuk menghitung waktu eksekusi rata-rata, diperlukan informasi tentang probabilitas pencarian masing-masing kunci. Dalam kasus ini, waktu eksekusi algoritma akan berkisar antara $O(n^2)$ hingga $O(n^3)$.

Dari analisis tersebut, dapat disimpulkan bahwa waktu eksekusi OBST sangat dipengaruhi oleh distribusi probabilitas pencarian kunci. Jika distribusinya tidak terlalu jauh dari probabilitas yang sama, maka waktu eksekusi akan relatif cepat. Namun, jika distribusinya sangat tidak seimbang, maka waktu eksekusi akan menjadi sangat lambat. Oleh karena itu, dalam penggunaan praktis, sangat penting untuk memperhitungkan distribusi probabilitas pencarian kunci untuk memilih algoritma yang tepat untuk membangun struktur data pencarian biner yang optimal.

F. REFERENSI

Anna Levitin. Introduction to the Design & Analysis of Algorithms 3rd Edition

<https://www.geeksforgeeks.org/optimal-binary-search-tree-dp-24/>