

Segment-Based Dynamic Programming for Optimal Binary Search Trees: A Sub-Cubic Algorithm with Hierarchical Weight Partitioning

Supplementary Materials

Code: Cost of Optimal Binary Search Tree (HWP Algorithm)

```
1 def optimal_two_way_search_cost(keys, 38
   key_weights, gap_weights): 39
2     """ 40
3     Compute the optimal cost of a two-way- 41
   comparison search tree using HWP DP. 42
4     - keys: sorted list of keys (length n). 43
5     - key_weights: list of key weights beta 44
   [1..n]. 45
6     - gap_weights: list of gap weights alpha 46
   [0..n]. 47
7     Returns the minimal search cost (float). 48
8     """ 49
9     n = len(keys) 50
10    keys = [None] + list(keys) 51
11    beta = [None] + list(key_weights) 52
12    alpha = list(gap_weights) 53
13 54
14    prefixBeta = [0]*(n+1) 55
15    for i in range(1, n+1): 56
16        prefixBeta[i] = prefixBeta[i-1] + 57
   beta[i] 58
17 59
18    prefixAlpha = [0]*(n+2) 60
19    for i in range(1, n+2): 61
20        prefixAlpha[i] = prefixAlpha[i-1] + ( 62
   alpha[i-1] if i-1 <= n else 0) 63
21 64
22    order = sorted(range(1, n+1), key=lambda 65
   i: (-beta[i], -i)) 66
23    pi = [0]*(n+1) 67
24    pos = [0]*(n+1) 68
25    for rank, key_idx in enumerate(order, 69
   start=1): 70
26        pi[rank] = key_idx 71
27        pos[key_idx] = rank 72
28 73
29    dp = [[None]*(n+2) for _ in range(n+2)] 74
30    for i in range(0, n+1): 75
31        dp[i][i+1] = [0] 76
32 77
33    for L in range(2, n+2): 78
34        for i in range(0, n+2 - L): 79
35            j = i + L 80
36            w0 = (prefixBeta[j-1] - 81
   prefixBeta[i]) + (prefixAlpha[j] - 82
   prefixAlpha[i]) 83
37            if L == 2: 84
38                cost0 = w0 85
39                else: 86
40                    best_split = float('inf') 87
41                    for b in range(i+1, j): 88
42                        cost_split = dp[i][b][0] 89
43                        + dp[b][j][0] 90
44                        if cost_split < 91
   best_split: 92
45                            best_split = 93
   cost_split 94
46                            cost0 = w0 + best_split 95
47                            dp_list = [cost0] 96
48                            remaining_weight = w0 97
49                            for h in range(1, n+1): 98
50                                key_idx = pi[h] 99
51                                if not (i < key_idx < j): 100
52                                    dp_list.append(dp_list 101
   [-1]) 102
53                                continue 103
54                                remaining_weight -= beta[ 104
   key_idx] 105
55                                cost_eq = dp_list[-1] 106
56                                if L >= 3: 107
57                                    best_split = float('inf') 108
58                                    for b in range(i+1, j): 109
59                                        left_list = dp[i][b] 110
60                                        right_list = dp[b][j] 111
61                                        left_cost = left_list 112
   [h] if h < len(left_list) else left_list 113
   [-1] 114
62                                        right_cost = 115
   right_list[h] if h < len(right_list) else 116
   right_list[-1] 117
63                                        cost_split = 118
   left_cost + right_cost 119
64                                        if cost_split < 120
   best_split: 121
65                                            best_split = 122
   cost_split 123
66                            else: 124
67                                best_split = float('inf') 125
68                                dp_list.append( 126
   remaining_weight + min(cost_eq, 127
   best_split)) 128
69                                dp[i][j] = dp_list 129
70 130
71    final_list = dp[0][n+1] 131
72    return final_list[n] if n < len( 132
   final_list) else final_list[-1] 133
```

Listing 1: Python implementation of HWP-based optimal two-way search cost.