Space and Time Tradeoffs

Chapter07

Space and Time Tradeoffs

- Computing the value of the function at many points (precompute and store in Mathematical tables)
- **Preprocess the problem's input** in whole or part and **store** the additional information obtained in a **table** to accelerate problem solving. This approach is called **Input Enhancement. Eg.** Sorting by counting, string matching
- **Dynamic programming :** Recording solutions to overlaping subproblems
- Use extra space to facilitate **faster and more flexible access to data** later. This approach is called **Pre-structuring. Eg.** Hashing

Space and Time Tradeoffs

- Time and space do not have to compete with each other. Eg. Sparse graph processing (data structure used to store the graph decides time and space requirements)
- In data compression space reduction is the goal

Sorting by Counting:

- Count the number of elements smaller than a given element in the list and record in a table.
- These numbers indicate the position of the element in the sorted list.
- Copy the elements to new array based on count value

Sorting by Counting

```
Algorithm ComparisionCountingSort (A[0..n - 1])
//Sorts an array by Comparison counting.
//Input: see parameters
//Output: S[0..n - 1] in non-decreasing order
for i ← 0 to n – 1 do count[i] ← 0;
for i ← 0 to n - 2
      for j ← i + 1 to n - 1
            if A[i] < A[j]
                  count[j] ← count[j]+1;
            else count[i] ← count[i]+1;
For i ← 0 to n – 1 do S[count[i]] ← A[i]
return S;
```

Analysis:

- Input size: number of elements n.
- Basic operation: Comparison A[i] < A[j]

$$C(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1$$

$$= \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1]$$

$$= \sum_{i=0}^{n-2} (n-1-i)$$

$$= \frac{n(n-1)}{2}$$

Sorting by Distribution Counting

- Above algorithm takes linear amount of extra space
- But the above idea leads to **distribution counting a linear sorting algorithm**
- If the elements in the list is between some lower bound l and some upper bound u.(elements repeated several times), we can compute the frequency of each element and store in an array F[0 to u-l].
- Fill F[0] positions with l, F[1] positions with l+1, F[2] positions with l+2 and so on to get the sorted list
- It is convenient to process the input array from right to left
- The time complexity of this algorithm is **linear**

Sorting by Distribution Counting

Algorithm DistributionCounting(A[0..n - 1])
//Sorts an array of integers from a limited range by distribution counting.
//Input: Array A[0…n-1] of integers between l and u (l<=u)
//Output: Array S[0..n - 1] of A's elements sorted in non-decreasing order
for j ← 0 to u - l do D[j] ← 0 //initialize frequencies
for i ← 0 to n - 1 do D[A[i] – l]= D[A[i] – l]+1 //compute frequencies
for j ← 1 to u - l do D[j] ← D[j - 1] + D[j] //reuse for distribution
for i ← n - 1 down to 0 do
    j ← A[i] - l
    S[D[j] - 1] ← A[i]
    D[j]=D[j]-1
return S;

Input Enhancement in String Matching

- Brute Force Method
  - Worst case - Θ (mn)
  - Average case – Θ (n)
- New algorithms
  - Knuth-Morris-Pratt algorithm (left-to-right comparison)
  - Boyer-Moore algorithm (right-to-left comparison)
- Simplified version of Boyer-Moore is Horspool's algorithm

Horspool's Algorithm

- Do right to left match and decide about the size of shift based on the **character 'c' of the text that was aligned against the last character of the pattern.**
- Four cases:

Case 1: If there are no c's in the pattern

S_0 ……. S ……. S_{n-1}

|

B A R B E R

B A R B E R

Horspool's Algorithm

Case 2: If there are occurences of character c in the pattern, but not as last one

S_0 …….. B ……. S_{n-1}

|

B A R B E R

B A R B E R

shift should align the rightmost occurrence of c in the pattern with the c in the text

Horspool's Algorithm

Case 3: If c is the last character in the pattern, but there are no c's among the other m-1 characters

S_0 ……..M E R ……. S_{n-1}

|

L E A D E R

L E A D E R

shift by the length of the pattern

Horspool's Algorithm

Case 4: If c happens to be the last character in the pattern, but there are other c's among the first m-1 characters

S_0 …….. O R ……. S_{n-1}

|

R E O R D E R

R E O R D E R

shift should align the rightmost occurrence of c among the first m-1 characters in the pattern with the c in the text

Horspool's Algorithm

- t (c) = the pattern's length m, if c is not among the first m-1 characters of the pattern (case 1 and 3)

  otherwise

  the distance from the rightmost c among the first m-1 characters of the pattern to its last character (case 2 and 4)

Creating Shift Table (Input Enhancement Technique)

Algorithm ShiftTable(P[0..m − 1])

//Fills the shift table used by Horspools and Boyer Moore algorithms.

// Input: Pattern P and an alphabet of possible characters

//Output: Table[0..size − 1] indexed by the alphabet's characters and filled with shift sizes computed by formula.

initialize all the elements of Table with m.

for j ← 0 to m − 2 do

    Table[P[j]] ← m − 1 − j

return Table

Horspool's Algorithm

Step 1 : For a given pattern of length m and the alphabet used in both the pattern an the text, construct the shift table as described.

Step 2 : Align the pattern against the beginning of the text.

Step 3 : Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text.

Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either **all m character pairs are matched (then stop)** or a mismatching pair is encountered after k>=0 character pairs are matched successfully.

In the later case, retrieve the entry t(c) from the c's column of the shift table where c is the text's character currently aligned against the last character of the pattern and **shift the pattern by t(c) characters to the right along the text**.

Horspool String Matching

Algorithm Horspool (P[0..m − 1], T[0..n − 1])
//Implements Horspool's Algorithm
//Input: pattern P[0..m-1] and Text T[0..n-1]
//output Position of the substring or -1 if there is no match
ShiftTable(P[0..m − 1])
i ← m − 1
while i <= n − 1 do
    k ← 0
    while k <= m − 1 and P[m − 1 − k] = T[i − k] do
        k ← k+1
    if k = m
        return i − m + 1
    else
        i ← i + Table[T[i]]
return −1

Boyer Moore Algorithm

- 2 tables
  - Bad symbol shift table (same as shift table of Horspool)
  - Good- Suffix shift table

- Mismatched Character is ‘c’
- Number of character match is ‘ k’

Boyer Moore Algorithm

- Bad symbol shift – Guided by mismatched character
- Good-suffix shift – Guided by the successful match of the last k > 0 characters

$d_1 = \max(t_1(c) - k , 1)$

Boyer Moore Algorithm

Step 1 : For a given pattern and the alphabet used in both the pattern an dthe text, construct the bad symbol table as described.

Step 2 : Using the pattern, construct the good suffix shift table as described.

Step 3 : Align the pattern against the beginning of the text.

Step 4 : Repeat the following step until either a matching substring is found or the pattern reaches beyond the last character of the text. Starting with the last character in the pattern, compare the corresponding characters in the pattern and the text until either all m character pairs are matched (then stop) or a **mismatching** pair is encountered after k>=0 character pairs are matched successfully. In the later case, **retrieve the entry $t_1(c)$ from the c's column of the bad symbol table where c is the text's mismatched character. If k>0, also retrieve the corresponding d2 entry from the good suffix table. Shift the pattern to the right by the number of positions computed by the formula**

d = d1 if k=0

Max{d1,d2} if k>0

where d1 = max{ $t_1(c)$ - k , 1 }