SNS COLLEGE OF ENGINEERING

Coimbatore-107

COURSE NAME: ANALYSIS OF ALGORITHM

II YEAR/ IV SEMESTER

UNIT – II

BRUTE FORCE METHOD

Topic

Brute Force Method: Traveling Salesman Problem – Knapsack Problem – Extra Topic

(Algorithm & Analysis of Matrix Multiplication)

Brute Force → Travelling Sales man Problem

UNIT - 2

TSP is classic optimization Problem.
⇒ consider every possible route (Permu.) & select one route with minimum total distance.

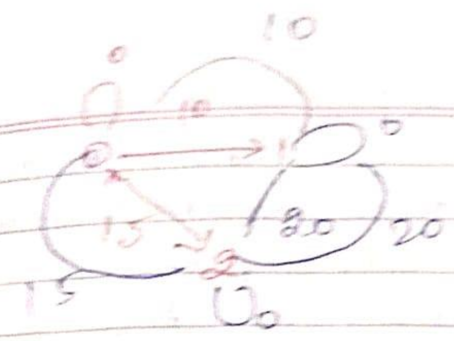
No. of city = 3. Distance ↓

city	0	1	2
0	0	10	15
1	10	0	20
2	15	20	0

Steps: possible routes

* Generate all permutations of cities:
  (i) 0 → 1 → 2 → 0
  (ii) 0 → 2 → 1 → 0
  (iii) 1 → 0 → 2 → 1
  (iv) 1 → 2 → 0 → 1
  (v) 2 → 0 → 1 → 2
  (vi) 2 → 1 → 0 → 2
* Calculate the total distance for each permutation.
  → Add distances of all routes

Route 1: (0 → 1 → 2 → 0)
Distance : 0 → 1 : 10
Distance : 1 → 2 :



Route 1:

0 → 1 → 2 → 0

0 → 1 : 10

1 → 2 : 20

2 → 0 : 15

10 + 20 + 15 = 45

Route 2:

0 → 2 → 1 → 0

0 → 2 : 15

2 → 1 : 20

1 → 0 : 10

45

Route 3:

1 → 0 → 2 → 1

1 → 0 : 10

0 → 2 : 15

2 → 1 : 20

45

Routes:

Syllabus: Unit - II

Brute Force:

(i) Selection sort

(ii) Bubble sort

(iii) Seq. Search

(iv) Closest pair & Convex Hull

(v) Travelling salesman

(vi) Knapsack problem

(vii) Assignment problem

Divide & Conquer:

(i) Merge sort

(ii) Quicksort

(iii) Binary search

③. Finding Shorest Route :
⇒ Compare the total distance Calculated for each route.
⇒ All are 45.

④. All are same. So Any route can be followed.

Analysis :
⇒ ~~Expensive for More cities~~
⇒ ~~Because permutation grows~~ factorially.

⇒ For 'n' cities, (n−1)! routes explored.
⇒ eg: n=3 ; 2! routes explored.
~~n=9 ; 9! = 362,880, routes~~

$\boxed{\text{Time complexity}} = \boxed{O(n! * n)}$

n → calculate distance for each route

n! → Generating all permutations

Space complexity = $\boxed{O(n!)}$

Stores all permutations ← each permutation generated & Stored.

UNIT. 2

0/1 knapsack problem using Brute force method

Knapsack problem:

=> It is a classic problem in optimization. 0/1 variant include:

* Items : A set of items, each with weight and a value.

* knapsack Capacity : Maximum weight that knapsack can carry

* Goal : Find combination of items that have the maximum value which should not exceed knapsack capacity.

Example:

Given:

(i). Items :-

Item1 : weight = 2, Value = 3

Item2 : Weight = 3, Value = 4

Item3 : Weight = 4, Value = 5.

(ii) knapsack capacity : 5

Step1:

Generate all possible combinations. For 'K' items, we can have 2^k subsets.

∴ For '3' items, we have $2^3 = 8$ subsets.

From eg, we evaluate all possible subsets & keep track of all maximum value that doesn't exceed knapsack capacity.

Subsets:

i=0

(i) Subset 1: No items selected → weight = 0

value = 0. ; max value = 0

i=1

(ii) Subset 2: only item 1 selected → weight = 2

Value = 3 ; max value = 3.

i=2

(iii) subset 3: only item 2 Selected → weight = 3

value = 4. ; max value = 4

i=3

(iv) Subset 4: only item 3 selected → weight = 4

value = 5. max value = 7

i=4

(v) Subset 5: Item 1 + Item 2 → weight = 2+3

max value = 7

weight = 5, value = 3+4 = 7

i=5

(vi) Subset 6: Item 1 + Item 3 → weight = 2+4 = 6

Value : 3+5 = 8 max value = 8

exceed capacity by weight. so invalid

i=6

(vii) Subset 7: Item 2 + Item 3 → weight = 3+4 = 7

Value : 4+5 = 9 max value = 9

i=7

(viii) subset 8: Item 1 + Item 2 + Item 3

Weight = 2+3+4 = 9 max value = 12

Value = 3+4+5 = 12

$\boxed{\therefore \text{Subset 5} = (\text{Item 1} + \text{Item 2})}$ is best valid combination, with weight = 5, value = 7 ; maximum value that fits within the knapsack's capacity.

Algorithm KnapsackBruteForce (weight [], values [], n, capacity)

```
{                    // Loops over 2^n possible subsets
maxValue = 0;  for (i=0; i<(1<<n); i++
{  totweight = 0 → current weight    // (1<<n) is equivalent to
   totValue = 0 → current value
   // check each 'i' to see if the item is included
   for (j=0; j<n; j++)
   {
     if (i & (1<<j)%2==1)
     {
     totweight = totalweight + weight[j];
     totvalue = totalvalue + value[j];
     }
   }  // If totalweight is within capacity check if value=max
   if (totweight <= capacity && totvalue>M
   {
   maxvalue = totalvalue;
   }
}
return maxvalue;
}
```

SNS COLLEGE OF ENGINEERING

Coimbatore-107

Explanation:

*. outer loops (i loop) iterates through all possible subsets → $2^3 = 8$ subsets.

* $(1<<n) = 2^n$.

* i ranges from '0' to $2^n - 1$ represent all possible combination

*. Inner loops (j loop) checks items present in current subset.

$\boxed{i \cancel{\&} (1<<j) \% 2} == 1$ jth

keypart that determines items ~~that~~ are included in the subset, represented by 'i'

i/(1<<j) shifts 'i' right by 'j' positions

% 2 == 1; checks if result is odd

i.e. j^{th} bit = 1; item included

Analysis:

⇒ Outer loop runs for 2n times.

⇒ Inner loops runs 'n' times for each iteration of outer loop. to check ~~index~~ whether each item is included in current sub set.

Time complexity = $O(2^n \times n)$ → No. of possible subsets. → No. of items

Space Complexity:

Space used by arrays weights & values for 'n' items

O(n)

maxvalue, totalvalue & totalvalue (Used)

constant Space $\Rightarrow O(1)$

$\boxed{S(n) = O(n)}$

~~No addit~~

Matrix Multiplication & Analysis

A → m×n Matrix

B → n×p matrix

C → m×p matrix.

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

C_{ij} = element at row i, column j in matrix C

A_{ik} = element at row i, column k in Matrix A

B_{kj} = element at row k, Column j in matrix B

n ⇒ No. of columns in A or Rows in 'B'

Algorithm: matrixmult [A[ ], B[ ], m, n, p]

```
{
 for (i = 0; i<m; i++)
 for (j=0; j<p; j++)
   C[i][j]=0;
```

for(k=0; k<n; k++)
{
C[i][j] += A[i][k] * B[k][j]
}
}
}

Analysis:

$\boxed{O(m \times n \times p) = O(n^3)}$ — Time complexity

$O(m \times n + m \times p)$

→ Large matrices – slow
Best use case → Small to medium sized matrices

Coin Change Problem

dp[i] → min no. of coins to make amount i. Coins = [1, 2, 5] Amt = 1

Goal:
Find minimum no. of coins to make the amount.

Base Case:
(i) dp[i] → Array checking min. no. of coins to make amount i
(ii) Initialize all values to '∞'

Recurrence Relation:
dp[i] = min(dp[i], dp[i-coin] + 1