CSE 373/L-22/12.05.2024/

Chapter - 15

Knapsack - Problem (0,1)

Brute Force Algorithm and dynamic

* Brute Force Method solve the sub-problem over and over again. But dynamic programming finds the solutions to sub problem and stores them in memory for later use.

* Dynamic programming algorithm is more effecient than brute force algorithm.

* Knapsack - Problem:

  - Items are indivisible.

⇒ Brute Force approach:

  - list all possible set within the weight limit
  - Choose the best one.
  - Ran time $T(n) = O(2^n)$

    → there are 2^n possible combination.

⇒

$\{I_0\ I_1\ I_2\} \xrightarrow{20} \{I_0 I_1, I_2\}$

$\{I_0, I_1, I_2 I_3\}^{8} \xrightarrow{20} \{I_0 I_2 I_3\}$

$\{I_0 I_1, I_2\} \xrightarrow{12} \{I_0 I_2\}$

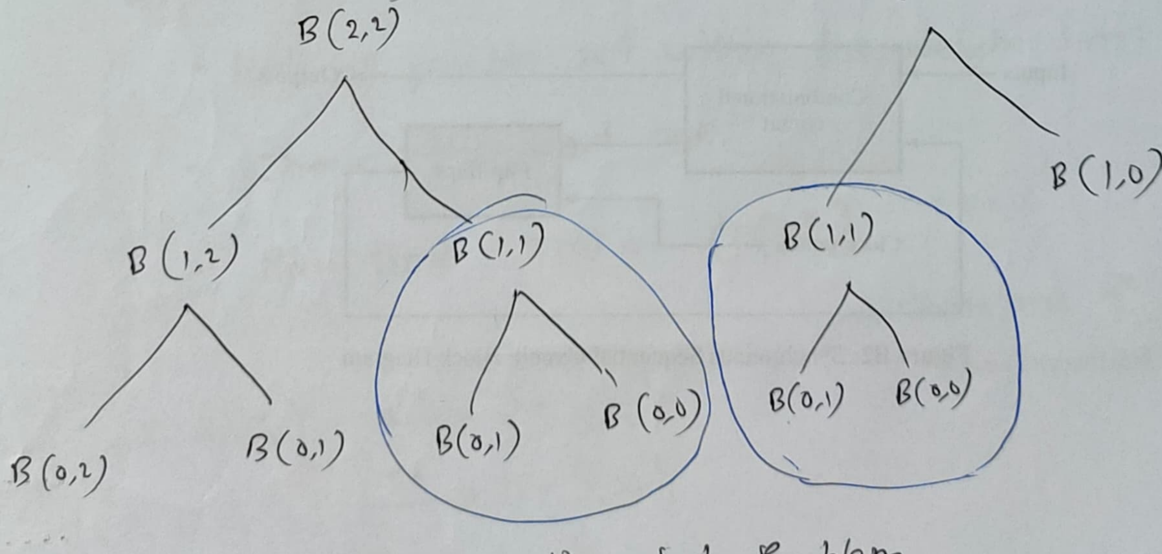
Sub-Problem:

$$B(k,w) = \begin{cases} B(k-1, w) & ; w_k > w \\ \max\{B(k-1,w), B(k-1, w-w_k) + b_k\} & ; \text{else} \end{cases}$$

0-1 Knapsack Algorithm - Recursive:

```
KS-0-1(k,w)
  if k=0 or w=0
     return 0
  if wk > w
     return KS-0-1(k-1,w)
  else
     return max(KS-0-1(k-1,w), KS-0-1(k-1, w-wk) + bk)
```

⇒



= Repeating Sub. Problem

⊛ 0-1 knapsack - Algorithm - Recursive Memoized

```
KS-0-1 (k, w)
    if B[k][w] != -1
        return B[k][w]

    if Wk > w
        B[k][W] = KS-0-1(k-1, w)

    else
        B[k][W] = max(KS-0-1(k-1, w), KS-0-1(k-1, w - Wk) + bk)

    return B[k][W]
```

⊛ 0-1 knapsack Algorithm - Bottom up - iterative

```
KNAPSACK-0-1 (W, n, b)
    let B[1..n, 1..W] be new table
    for w = 0 to W
        B[0, w] = 0
    for i = 1 to n
        B[i, 0] = 0
    for i = 1 to n
        for w = 1 to W
            if wi <= w
                if bi + B[i-1, w-wi] > B[i-1, w]
                    B[i, w] = bi + B[i-1, w-wi]
```

else $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$

return B.

⊛ Running Time $= O(W) + O(n) + O(n * W)$

$= O(n * W)$

$\boxed{\text{Slide- 18 - 35}}$ → Example.

⊛ If previous value of the column is same, then not selected.

⊛ Algorithm to retrieve selected item List (

$i = n$, $k = W$

while $i, k > 0$

if $B[i,k] \neq B[i-1,k]$

mark the i^{th} item as the knapsack

$i = i-1$, $k = k - w_i$

else $i = i-1$

$\boxed{\text{Slide - 38 - 44}}$ → Example

* We can reduce the space by using the immediate previous row by replacing them.

For that we can't keep track, which item is selected or not.

* Implement it?

⇒

```
KNAPSACK-0-1(W, n, b)
    let B[0...1, 0...W) be new tables
    for w = 0 to W
        B[0,w] = 0
    for i = 0 to 1          // B[1,0] = 0
        B[i,0] = 0
    for i = 1 to n
        for w = 1 to W
            if wi ≤ w
                if bi + B[0, w-wi] > B[0,w]
                    B[1,w] = bi + B[0, w-wi]
                else B[1,w] = B[0,w]
            else B[1,w] = B[0,w]
        for index = 0 to W
            B[0,index] = B[1,index]
    return B
```

L-23/14.05.2024

Chapter-22

Elementary Graph

Graph, $G = (V, E)$

E → set of edges

V → set of vertices

→ Dense Graph:

$$|E| \approx |V|^2$$

→ Sparse Graph:

$$|E| \approx |V|$$

* On the basis of direction:

(i) Undirected Graph:

- edge, $E(u,v) = E(v,u)$
- no self loop exist

(ii) Directed Graph:

$$E(u,v) \Rightarrow u \rightarrow v$$

$$E(v,u) \Rightarrow v \rightarrow u$$

* Weighted Graph:

- each E has an associated weight.
- $w : E \rightarrow \mathbb{R}$

Representation:

$G = (V, E)$, $V \rightarrow \{1, 2, 3 \ldots |V|\}$

- Stored as a 2D array : $|V| \times |V|$ matrix.

$A = [V, V]$

$$A = a_{ij} = \begin{cases} 1 & \text{; if edge exist between } i \text{ \& } j \Rightarrow (i,j) \in E \\ 0 & \text{; else} \end{cases}$$

- As it is stored in 2D array,

memory required $= \Theta(V^2)$

$\Rightarrow$ Used when the graph is dense graph.

$\Rightarrow$ For this matrix or array method,

- we can quickly determine if there is any edge between two vertices. Just check the value of $A[i, j] = 0$ or 1.

- But there is no quick way to determine the vertices adjacent from another vertex.

$\boxed{\text{Slide - 4}}$

By using the ~~List~~ Linked List structure

- contains one array of vertex list with address pointer, that point the vertices

$Adj = [|V|]$

$Adj[u]$ = contains a linked list of all vertices of u.

Memory required = $\Theta(V+E)$

⇒ Used when the graph is sparse.

⇒ No quick way to determine edge between two vertices.

- But we can quickly find out all the vertices adjacent from a given vertex.

Slide-6

* Graph Searching

(i) BFS ⇒ Breadth-first Search.

- Used Queue to search.
- Basically BFS build a breadth first tree. From where we can find the ~~sm~~ sortest path or smallest number of edges to reach a vertex from the root vertex.
- works on both directed or undirected.
- tree may change depends on the starting ~~inde~~ vertex.

⊛ Color Codes:

White ⇒ Not discovered

Gray ⇒ Not fully explored.

- There are at least one white adjacent.

Black ⇒ Fully explored

- No white adjacent.

⊛ BFS Algorithm:

$\overset{\text{starting vertex/root of the tree}}{}$

BFS(G, s) → starting vertex/root of the tree

```
      for each vertex u ∈ G.V - {s}
O(V)      u.color = WHITE
          u.d = ∞
          u.π = NIL
      s.color = GRAY
      s.d = 0
      s.π = NIL
      Q = ∅
O(V) ← ENQUEUE(Q, s)
      while Q ≠ ∅
O(V) ←    u = DEQUEUE(Q)
          for each v ∈ G.Adj[u]
O(E)          if v.color == WHITE
                  v.color = GRAY
O(V+E)            v.d = u.d + 1
                  v.π = u
                  ENQUEUE(Q, v)
          u.color = BLACK
```

Slide-11, 12

⊛ BFS Running Time: $O(V+E)$

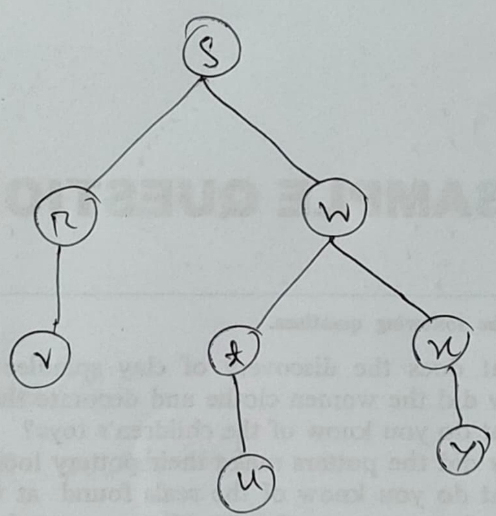
⊛ Properties:

- calculates the sortest path distance to the source node.

⇒ $\delta(s,v)$ = minimum number of edge from $s \rightarrow v$.

if not reachable then ∞.

⊛ Tree Generated from the example $\boxed{\text{Slide-12}}$



⇒ sortest distance and path to all vertices from S.

⊛ BFS first explore the adjacent vertex then go to the deep.

(ii) DFS ⇒ Depth-first search.

⊛ DFS try to reach as deep as possible then come back to explore.

⇒ use stack for search.

- works in both directed or undirected.

- It also produce a tree known as depth-first ~~tree~~ forest

→ consist of trees.

↓

all are different

⊛ DFS Algorithm

```
DFS (G)
  for each vertex u ∈ G.V
      u.color = WHITE
      u.π = NIL
  time = 0
  for each vertex u ∈ G.V
      if u.color == WHITE
          DFS-VISIT (G, u)
```

DFS-VISIT (G, u)

time = time + 1

u.d = time

u.colon = GRAY

for each v ∈ G.Adj[u]

if v.colon == WHITE

v.π = u

DFS-VISIT (G, v)

u.colon = BLACK

time = time + 1

u.f = time

$\boxed{\text{Slide - 19, 20}}$

Quize-3

Dynamic Programming

12.05.2024