Greedy Methods

Manoj Kumar
DTU, Delhi

SAMSUNG

DTU
Delhi Technological
UNIVERSITY

Greedy algorithms

- A greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.

Optimization problems

- An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
- A "greedy algorithm" sometimes works well for optimization problems.
- A greedy algorithm works in phases. At each phase:
  - You take the best you can get right now, without regard for future consequences
  - You hope that by choosing a *local* optimum at each step, you will end up at a *global* optimum.

Greedy algorithms have five pillars

- A candidate set, from which a solution is created.
- A selection function, which chooses the best candidate to be added to the solution.
- A feasibility function, that is used to determine if a candidate can be used to contribute to a solution.
- An objective function, which assigns a value to a solution, or a partial solution, and
- A solution function, which will indicate when we have discovered a complete solution.

Example: Making Changes

- Suppose you want to make changes of a certain amount of money, using the fewest possible notes and coins
- A greedy algorithm would do this would be:
  At each step, take the largest possible notes or coin that does not overshoot
  - Example: To make 758, you can choose:
    - a 500 rupees note
    - two 100 rupees notes,
    - a 50 rupees note,
    - a 5 rupees coin,
    - a 2 rupee coin
    - a 1 rupee coin
- For money, the greedy algorithm always gives the optimum solution

SAMSUNG

DTU
Delhi Technological University

The Knapsack Problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

The Knapsack Problem

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among n items, where the ith item worth b_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Note: assume b_i, w_i, and W are all integers
    - each item must be taken or left in entirety.
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

Fractional Knapsack problem

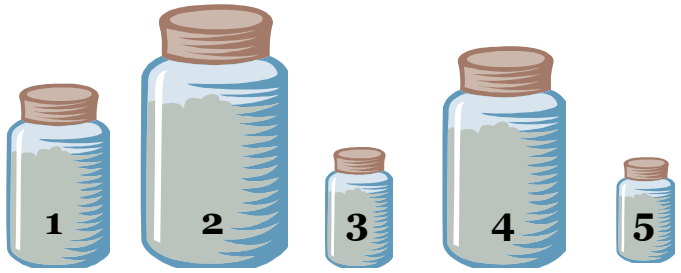
- Given: A set S of *n* items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most *W*.
- If we are allowed to take fractional amounts, then this is the **fractional knapsack problem**.
  - In this case, we let x_i denote the amount we take of item i
  - Objective: maximize $\sum_{i \in S} b_i (x_i / w_i)$
  - Constraint: $\sum_{i \in S} x_i \leq W$

SAMSUNG

DTU Delhi Technological University

Fractional Knapsack: Example

- Given: A set S of n items, with each item i having
  - b_i - a positive benefit
  - w_i - a positive weight
- Goal: Choose items with maximum total benefit but with weight at most W.



Items:	1	2	3	4	5
Weight:	4 ml	8 ml	2 ml	6 ml	1 ml
Benefit:	$12	$32	$40	$30	$50
Value: ($ per ml)	3	4	20	5	50

10 ml

"knapsack"

Solution:
- 1 ml of 5
- 2 ml of 3
- 6 ml of 4
- 1 ml of 2

SAMSUNG

DTU Delhi Technological University

Fractional Knapsack problem

- Greedy choice: Keep taking item with highest **value** (benefit to weight ratio)
  - Use a heap-based priority queue to store the items, then the time complexity is O(n log n).
- Correctness: Suppose there is a better solution
  - there is an item *i* with higher value than a chosen item *j* (i.e., $v_j < v_i$) , if we replace some j with i, we get a better solution
  - Thus, there is no better solution than the greedy one

Algorithm ***fractionalKnapsack*****(*S, W*)**

Input: set S of items w/ benefit b_i and weight w_i; max. weight W

Output: amount x_i of each item i to maximize benefit with weight at most W

for ***each item i in S***

$x_i \leftarrow 0$

$v_i \leftarrow b_i / w_i$ {value}

$w \leftarrow 0$ {current total weight}

while $w < W$

remove item i with highest v_i

$x_i \leftarrow \min\{w_i, W - w\}$

$w \leftarrow w + \min\{w_i, W - w\}$

SAMSUNG

DTU Delhi Technological University

The Knapsack Problem

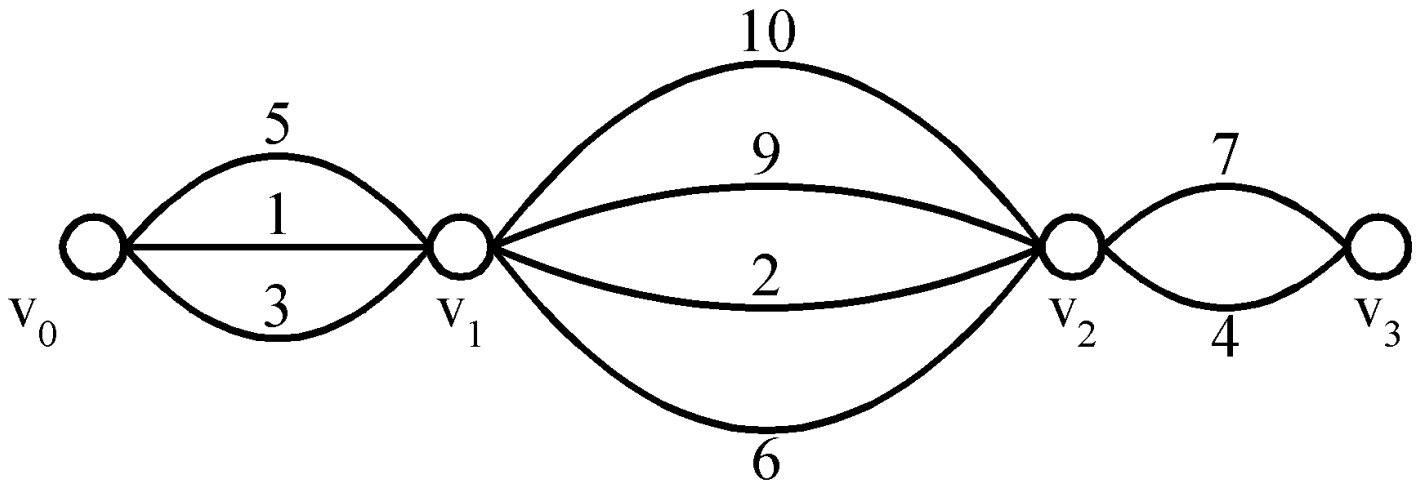
- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm.
- The optimal solution to the 0-1 Knapsack problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 1 is worth $ 75, item 2 is worth $100 and item 3 is worth $200.*
    - *According to greedy algorithm, items 1 and 2 are selected with total value = $175*
    - *But best selection could be item 2 and 3 with total value = $300.*

SAMSUNG

DTU Delhi Technological University

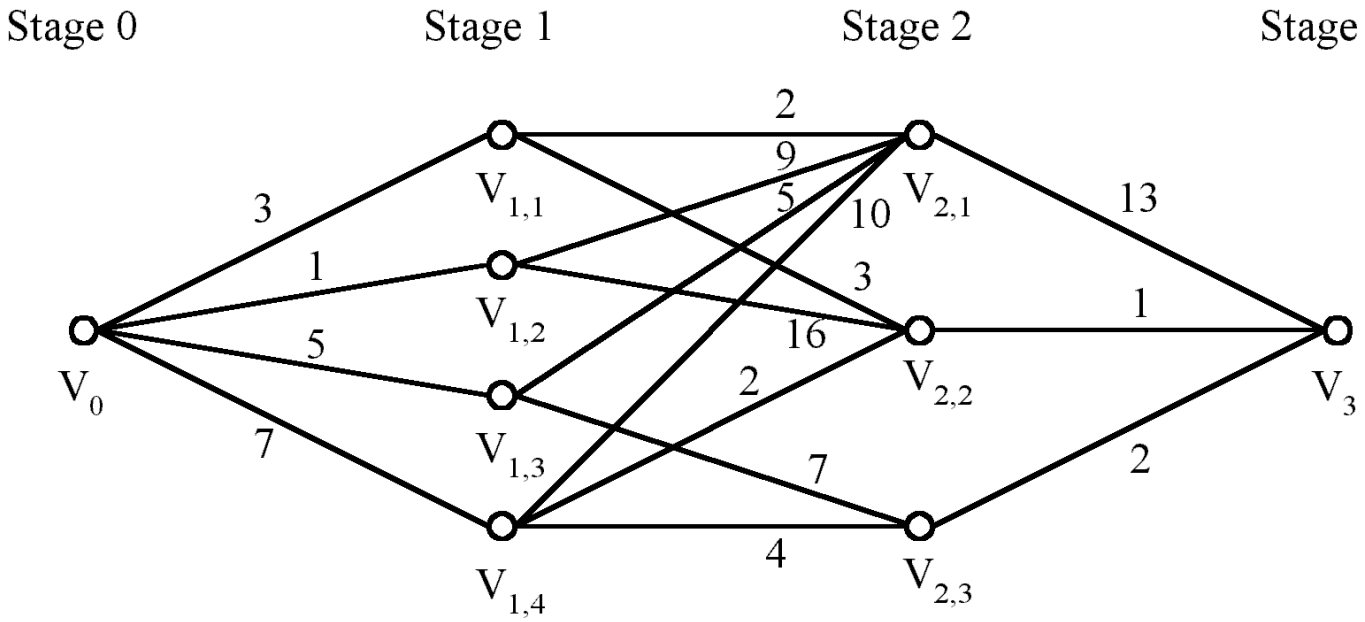
Shortest paths on a special graph

- Problem: Find a shortest path from v_0 to v_3.
- The greedy method can solve this problem.
- The shortest path: 1 + 2 + 4 = 7.



Shortest paths on a multi-stage graph

- Problem: Find a shortest path from v_0 to v_3 in the multi-stage graph.



- Greedy method: $v_0v_{1,2}v_{2,1}v_3 = 23$
- Optimal: $v_0v_{1,1}v_{2,2}v_3 = 7$
- The greedy method does not work.

SAMSUNG

DTU Delhi Technological University

Other Greedy Algorithms

- MST algorithms
  - Kruskal's and Prim's Algorithms
- Single Source Shortest Path Algorithm.
  - Dijkstra Algorithm
- Huffman Coding

SAMSUNG

DTU
Delhi Technological
UNIVERSITY

Dynamic Programming

Manoj Kumar
DTU, Delhi

SAMSUNG

DTU
Delhi Technological
UNIVERSITY

Dynamic Programming

- Dynamic Programming is an algorithm design technique for optimization problems: often minimizing or maximizing.
- Like divide and conquer, DP solves problems by combining solutions to subproblems.
- Unlike divide and conquer, subproblems are not independent.
  - Subproblems may share subsubproblems,
  - However, solution to one subproblem may not affect the solutions to other subproblems of the same problem. (More on this later.)

D.ynamic Programming..

- DP reduces computation by
  - Solving subproblems in a bottom-up fashion.
  - Storing solution to a subproblem the first time it is solved.
  - Looking up the solution when subproblem is encountered again.
- Key: determine structure of optimal solutions

SAMSUNG

DTU
Delhi Technological University

Steps in Dynamic Programming

1. Characterize structure of an optimal solution.
2. Define value of optimal solution recursively.
3. Compute optimal solution values either top-down with caching or bottom-up in a table.
4. Construct an optimal solution from computed values.

We'll study these with the help of examples.

0/1 Knapsack

- Problem statement:
  - A thief robbing a store and can carry a maximal weight of W into their knapsack. There are n items and i^{th} item weigh w_i and is worth v_i dollars. What items should thief take?
- Exhibit No greedy choice property.
  - No greedy algorithm exists.
- Exhibit optimal substructure property.
- Only dynamic programming algorithm exists.

0/1 Knapsack Problem: Formal description

Formal description: Given two n-tuples of positive numbers

$$\langle v_1, v_2, \ldots, v_n \rangle \quad \text{and} \quad \langle w_1, w_2 \ldots, w_n \rangle,$$

and $W > 0$, we wish to determine the subset $T \subseteq \{1, 2, \ldots, n\}$ (of items to take) that

$$\text{maximizes} \quad \sum_{i \in T} v_i,$$

$$\text{subject to} \quad \sum_{i \in T} w_i \leq W.$$

SAMSUNG

DTU Delhi Technological University

0/1 Knapsack Problem

Remark: This is an optimization problem.

Brute force: Try all 2^n possible subsets T.

Question: Any solution better than the brute-force?

SAMSUNG

DTU Delhi Technological University

0/1 Knapsack: Solution

- Let i be the highest-numbered item in an optimal solution S for W pounds. Then $S' = S - \{i\}$ is an optimal solution for $\boldsymbol{W - w_i}$ pounds and the value to the solution S is V_i plus the value of the subproblem.
- We can express this fact in the following formula: define $c[i, w]$ to be the solution for items $1,2, \ldots, i$ and maximum weight w. Then

$$c[i,w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i\text{-}1, w] & \text{if } w < w_i \\ \max [v_i + c[i\text{-}1, w\text{-}w_i], c[i\text{-}1, w]\} & \text{if } i > 0 \text{ and } w \geq w_i \end{cases}$$

SAMSUNG

DTU Delhi Technological University

0/1 Knapsack: Solution

- This says that the value of the solution to i items either include i^{th} item,
  - in which case it is v_i plus a subproblem solution for (i - 1) items and the weight excluding w_i, or
  - does not include i^{th} item, in which case it is a subproblem's solution for (i - 1) items and the same weight.
- That is, if the thief picks item i, thief takes v_i value, and thief can choose from items W - w_i, and get $c[i - 1, W - w_i]$ additional value.
- On other hand, if thief decides not to take item i, thief can choose from item 1,2, . . . , i- 1 upto the weight limit w, and get $c[i - 1, w]$ value. The better of these two choices should be made.

SAMSUNG

DTU Delhi Technological University

0/1 Knapsack: Solution

- The algorithm takes as input the
  - maximum weight W,
  - the number of items n,
  - and the two sequences $v = <v_1, v_2, \ldots, v_n>$ and $w = <w_1, w_2, \ldots, w_n>$.
- It stores the $c[i, j]$ values in the table, that is, a two dimensional array, $c[0 \ldots n, 0 \ldots w]$ whose entries are computed in a row-major order.
- That is, the first row of c is filled in from left to right, then the second row, and so on.
- At the end of the computation, $c[n, w]$ contains the maximum value that can be picked into the knapsack.
-

SAMSUNG

DTU Delhi Technological University

0/1 Knapsack: Algorithm

```
Dynamic-0-1-knapsack (v, w, n, W)
for w ← 0 to W
   do  c[0, w] ← 0
for i ←1 to n
   do  c[i, 0] ← 0
       for  w ←1 to W
        do if  w_i ≤ w
             then if  v_i + c[i-1, w-w_i] > c[i-1, w]
                    then c[i, w] ← v_i + c[i-1, w-w_i]
                    else c[i, w] ← c[i-1, w]
             else  c[i, w] = c[i-1, w]
```

- The set of items to take can be deduced from the table, starting at $c[n.\ w]$ and tracing backwards where the optimal values came from. If $c[i,\ w] = c[i\text{-}1,\ w]$ item i is not part of the solution, and we are continue tracing with $c[i\text{-}1,\ w]$. Otherwise item i is part of the solution, and we continue tracing with $c[i\text{-}1,\ w\text{-}W]$.

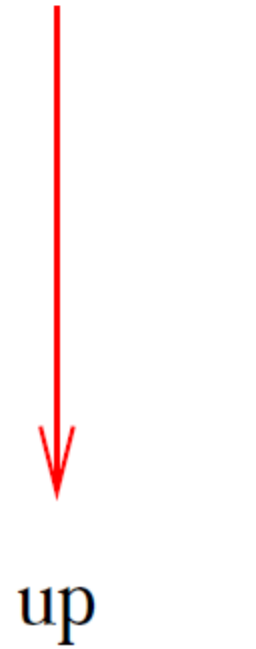
SAMSUNG

DTU Delhi Technological University

Analysis

- This dynamic-0-1-kanpsack algorithm takes $\theta(nw)$ times, broken up as follows: $\theta(nw)$ times to fill the c-table, which has $(n+1).(w+1)$ entries, each requiring $\theta(1)$ time to compute. $O(n)$ time to trace the solution, because the tracing process starts in row n of the table and moves up 1 row at each step.

c[i,w]	w=0	1	2	3	…	…	W
i= 0	0	0	0	0	…	…	0
1							
2							
:							
n							



Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

c[i,w]	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

- The algorithm computing *c[i,w]* does not keep record of which subset of items gives the optimal solution.
- To compute the actual subset, we can add an auxiliary boolean array **keep[i,w]** which is 1 if we decide to take the i^{th} item in c[i, w] and 0 otherwise.

SAMSUNG

DTU Delhi Technological University

```
Dynamic-0-1-knapsack (v, w, n, W)
for w ← 0 to W
   do  c[0, w] ← 0
for i ←1 to n
   do  c[i, 0] ← 0
       for  w ←1 to W
        do if  w_i ≤ w
             then if  v_i + c[i-1, w-w_i] > c[i-1, w]
                    then c[i, w] ← v_i + c[i-1, w-w_i]
                         keep[i,w] ← 1
                    else c[i, w] ← c[i-1, w]
                         keep[i,w]← 0
             else  c[i, w] = c[i-1, w]
                   keep[i,w] ← 0
```

SAMSUNG

DTU Delhi Technological University

Constructing the Optimal Solution

Question:

- How do we use values in keep[i, w] to determine the subset T of items having the maximum value?
- If keep[n,W] is 1, then $n \in T$, We can repeat this argument for keep[n-1,W-w_n].
- If keep[n,W] is 0, then $n \notin T$, We can repeat this argument for keep[n-1,W].

- Therefore te following part of the program will output the elements of T.

```
K←W
for i ← n downto 1
   do if (keep[i,K] ==1)
         then   print i
                K← K-w_i
```

Complete Algorithm

```
Dynamic-0-1-knapsack (v, w, n, W)
for w ← 0 to W
   do  c[0, w] ← 0
for i ←1 to n
   do  c[i, 0] ← 0
        for  w ←1 to W
         do if  w_i ≤ w
              then if  v_i + c[i-1, w-w_i] > c[i-1, w]
                       then c[i, w] ← v_i + c[i-1, w-w_i]
                            keep[i,w] ← 1
                       else c[i, w] ← c[i-1, w]
                            keep[i,w]← 0
              else  c[i, w] = c[i-1, w]
                    keep[i,w] ← 0

K←W
for i ← n downto 1
   do if (keep[i,K] ==1)
         then   print i
                K← K-w_i
```

Example

Let $W = 10$ and

i	1	2	3	4
v_i	10	40	30	50
w_i	5	4	6	3

c[i,w]	0	1	2	3	4	5	6	7	8	9	10
$i = 0$	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	10	10	10	10	10	10
2	0	0	0	0	40	40	40	40	40	50	50
3	0	0	0	0	40	40	40	40	40	50	70
4	0	0	0	50	50	50	50	90	90	90	90

keep[i,w]	0	1	2	3	4	5	6	7	8	9	10
i= 0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	**0**	0	1	1	1	1	1	1
2	0	0	0	0	1	1	1	**1**	1	1	1
3	0	0	0	0	0	0	0	**0**	0	0	1
4	0	0	0	1	1	1	1	1	1	1	**1**

Solution
T={2,4}