JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Greedy Algorithms

Foundations of Algorithms
Guven

JOHNS HOPKINS WHITING SCHOOL *of* ENGINEERING

Reading Assignment

- Cormen, Chapter 16

Greedy Algorithms

- Attempts to solve optimization problems
- Dynamic programming can be overkill
  - Greedy algorithms tend to be easier to grasp and code
- Problems exhibit optimal substructure
- Problems exhibit the **greedy-choice** property
  - Pick the best move at each iteration
    - Towards a solution
  - Make a locally optimal selection, expecting a globally optimal solution
    - Not guaranteed

Greedy Strategy

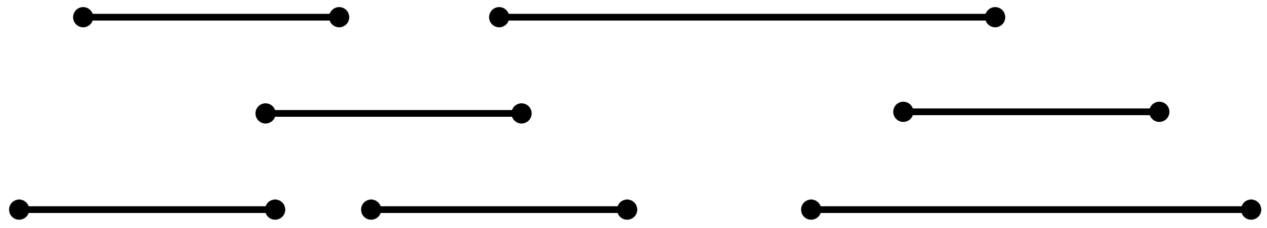
- Pick the choice that seems best at the moment
    - Prove that when there is a choice to make, one of the optimal choices is the greedy choice. Therefore, it's always safe to make the greedy choice
- Show that all subproblems resulting from the greedy choice are empty except for one of the subproblem
    - It is the locally optimal pick

Activity Selection Problem

- Problem: Maximize money's worth out of a park
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Goal: Ride as many rides as possible
  - Another, alternative goal: Maximize time spent on rides
- This problem is called activity selection problem

Activity Selection Problem

- Input: Set S of *n* activities, $a_1, a_2, \ldots, a_n$
  - s_i = start time of activity i
  - f_i = finish time of activity i
- Output: A ⊆ S max number of compatible activities
  - Two activities are compatible if their intervals don't overlap
- Examples



Optimal Substructure

- Assume activities are sorted by finishing times
  - $f_1 \le f_2 \le \ldots \le f_n$
- Suppose an optimal solution includes activity a_k
  - This generates two subproblems
  - Selecting from $a_1, \ldots, a_{k-1}$ activities compatible with one another, and finish before a_k starts (compatible with a_k)
  - Selecting from $a_{k+1}, \ldots, a_n$, activities compatible with one another, and that start after a_k finishes
  - The solutions to the two subproblems must be optimal
    - Prove using the cut-and-paste approach.

Recursive Solution

- Let S_{ij} = subset of activities in S that start after a_i finishes and finish before a_j starts
- Subproblems: Selecting maximum number of mutually compatible activities from S_{ij}
- Let c[i, j] = size of maximum-size subset of mutually compatible activities in S_{ij}
- Recursive Solution $c[i,j] =$

$$\begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max\limits_{i<k<j} (c[i,k] + c[k,j] + 1) & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Repeated Subproblems

- Consider a recursive algorithm that tries all possible compatible subsets to find a maximal set, and notice repeated subproblems
- S: 1∈A?
    - yes: S': 2∈A?
        - yes: S''
        - no: S'-{2}
    - no: S-{1}: 2∈A?
        - yes: S''
        - no: S-{1,2}

Greedy-choice Property

- Dynamic programming - memoize - $O(n^2)$
- Activity selection $\longrightarrow$ greedy-choice property
    - Locally optimal choice $\Longrightarrow$ globally optimal solution
    - **Theorem:**
      if S is an activity selection problem sorted by finish time, then $\exists$ optimal solution $A \subseteq S$ such that $\{1\} \in A$

        - Sketch of proof: if $\exists$ optimal solution B that does not contain {1}, can always replace first activity in B with {1}. Same number of activities, thus optimal.

Greedy-choice Property

- **Theorem** (cont.): There is an optimal solution to the subproblem S_{ij}, that includes the activity with the smallest finish time in set S_{ij}
- Hence, there is an optimal solution to S that includes a_1
- Therefore, make this greedy choice without solving subproblems first and evaluating them
- Solve the subproblem that ensues as a result of making this greedy choice
- Combine the greedy choice and the solution to the subproblem

Recursive Algorithm

```
def actpick_r(s,f,i,j):
  m=i+1
  while m<j and s[m] < f[i]:
    m=m+1
  if m<j:
    return {a[m]} union actpick_r(s,f,m,j)
  else:
    return Ø
```

- Initial call: `actpick_r(s,f,0,n+1)`
- Complexity Θ(n)

Typical Steps

- Cast the optimization problem as one in which we make a choice and are left with one subproblem to solve
- Prove there's always an optimal solution that makes the greedy choice, so that the greedy choice is always safe
- Show that greedy choice and optimal solution to subproblem $\Longrightarrow$ optimal solution to the problem
- Make the greedy choice and solve top-down
- May have to preprocess input to put it into greedy order
    - Example: Sorting activities by finish time

Greedy Activity Selection

- Sort the activities by finish time
- Schedule the first activity
- Then schedule the next activity in sorted list which starts after previous activity finishes
- Repeat until no more activities
- Intuition: Always pick the shortest ride available at the time

Elements of Greedy Algorithms

- Greedy-choice property
    - A globally optimal solution can be arrived at by making a locally optimal (greedy) choice
- Optimal substructure
- Can be solved by Dynamic Programming

Knapsack Problem

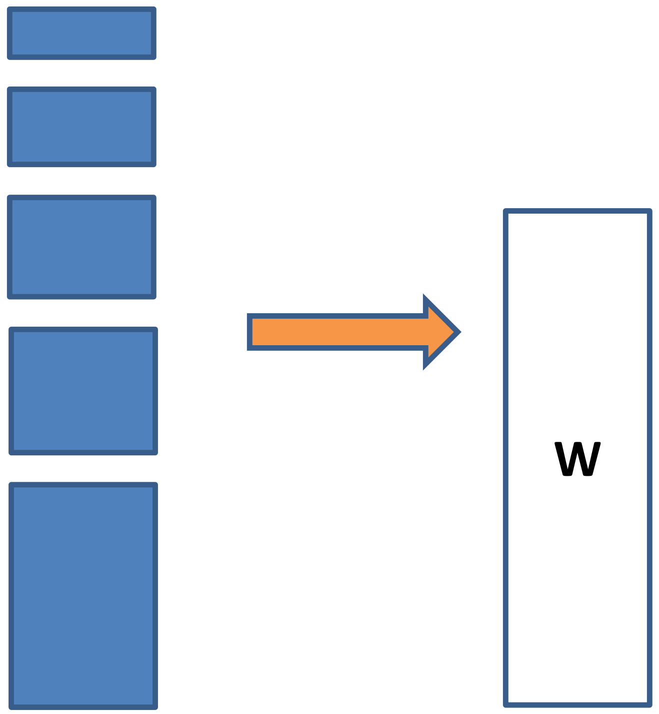
- The famous knapsack problem:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector's market. But, the thief 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 heist?

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (w_i , b_i and $W \in \mathbb{Z}$ for all i)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

Example

- w=[2,3,4,5,9], b=[3,4,5,8,10], W=20



The Knapsack Problem

- More formally, the 0-1 knapsack problem
  - The thief must choose among n items, where the i^{th} item worth v_i dollars and weighs w_i pounds
  - Carrying at most W pounds, maximize value
    - Assume v_i, w_i, and W are all integers
    - “0-1” - 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

0-1 Knapsack Problem

- Problem, in other words, is to find

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

Brute Force

- Enumerate all items
  - Since there are n items, there are 2^n possible combinations of items
  - We go through all combinations and find the one with the most total value and with total weight ≤ W
  - Complexity $O(2^n)$

Define Subproblem

- If items are labeled with i=1,...,n, then a subproblem - find an optimal solution for S_k labeled with some index set $\{j\} \subseteq \{i\}$
- Valid subproblem definition
- Can we describe the final solution S_n in terms of subproblems S_k?

Counter Example

- S_4: w=[2,4,5,3], b=[3,5,8,4] w_{tot}=14; b_{tot}=20
- S_5: w=[2,4,5,9], b=[3,5,8,10] w_{tot}=20; b_{tot}=26
- Solution for S_4 is not part of S_5
- Need to include parameter w which represents total weight of the subset of items
- Subproblem: B[k,w]

Recurrent Subproblem

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

- Best subset of S_k has total weight w either:
  - The best subset of S_k-1 has total weight w, or
  - The best subset of S_k-1 has total weight w-w_k plus the item k
- First case, w_k>w - total weight would exceed w
- Second case, w_k≤w - the item k can be in the solution

Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
- The optimal solution to the 0-1 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 2 is worth $100. Assign values to the other items so that the greedy strategy will fail

Fractional Knapsack

- max $\sum_i x_i b_i$ such that $\sum_i x_i \leq W$
- Sort items with value/weight order and apply the same greedy approach as before

0-1 Knapsack Algorithm

```
for w in weights:
  B[0,w]=0
for i in range(0,n+1):
  B[i,0]=0
  for w in weights:
    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]
```

Example

- n=4, W=5; Elements (weight,benefit): [(2,3), (3,4), (4,5), (5,6)]

w, i	0	1	2	3	4
0	0				
1	0				
2	0				
3	0				
4	0				
5	0				

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	**0**			
2	0				
3	0				
4	0				
5	0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0				
4	0				
5	0				

Items:
(w,b)
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0			
2	0	**3**			
3	0	**3**			
4	0	**3**			
5	0	**3**			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	**0**		
2	0	3	**3**		
3	0	3			
4	0	3			
5	0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0		
2	0	3	3		
3	0	3	**4**		
4	0	3	**4**		
5	0	3	**7**		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	**0**	
2	0	3	3	**3**	
3	0	3	4	**4**	
4	0	3	4		
5	0	3	7		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	**5**	
5	0	3	7		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

JOHNS HOPKINS UNIVERSITY | Engineering for Professionals

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	
2	0	3	3	3	
3	0	3	4	4	
4	0	3	4	5	
5	0	3	7	7	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

w, i	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	**0**
2	0	3	3	3	**3**
3	0	3	4	4	**4**
4	0	3	4	5	**5**
5	0	3	7	7	**7**

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

Example (cont.)

- Backtrack has to be added to the algorithm, similar to the LCS
    - An auxiliary array to mark if i^{th} item added or not
- Algorithm complexity O(nW)

Huffman Coding

- Variable length coding
- Most character code systems (ASCII, unicode) use fixed length encoding
- If frequency data is available and there is a wide variety of frequencies, variable length encoding can save 20% - 90% space
- Which characters should we assign shorter codes; which characters will have longer codes?

Data Compression

- Suppose a file only has 6 letters {a,b,c,d,e,f} with frequencies

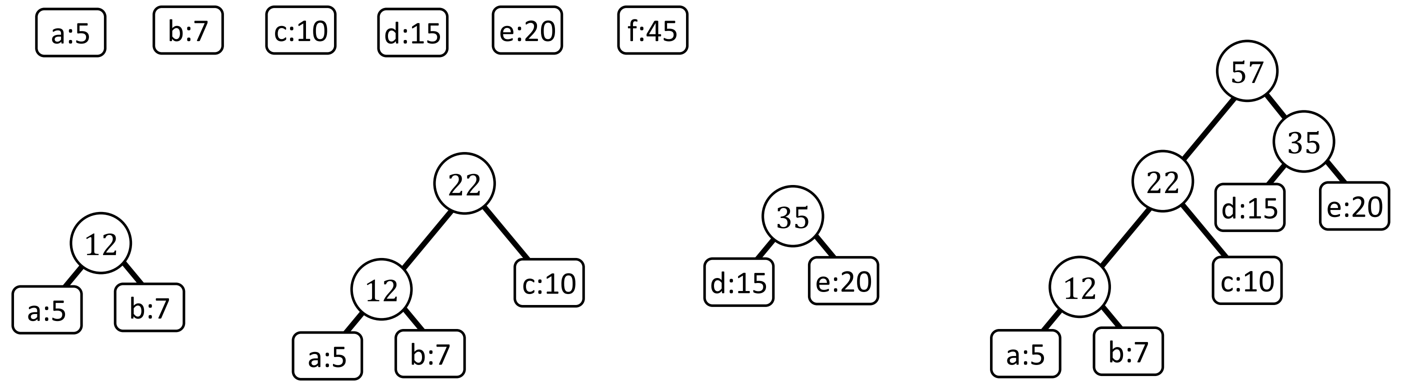
a	b	c	d	e	f
.45	.13	.12	.16	.09	.05
000	001	010	011	100	101
0	101	100	111	1101	1100

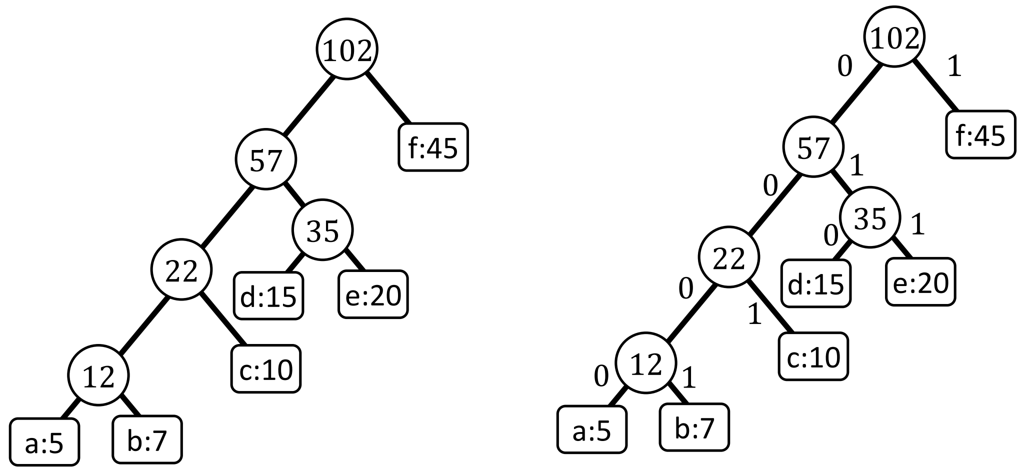
- Fixed length: 3G
- Variable length:
  .45*1+.13*3+.12*3+.16*3+.09*4+.05*4=2.24G

Encoding Scheme

- Input: Given a set of n letters [c_1,…, c_n] with frequencies [f_1,…, f_n]
- Construct a full binary tree T to define a prefix code that minimizes the average code length
    - Average(T) = $\sum_{i=1}^{n} f_i \text{length}_T(c_i)$
- Huffman coding can be generated by a greedy strategy

Example





a	b	c	d	e	f
5	7	10	15	20	45
0000	0001	001	010	011	1

Huffman Coding Algorithm

- Use of priority queue
  - Heap implementation
  - Each heap operation O(log n)
  - main loop is n-1
- To build the Huffman coding tree, time complexity: O(nlogn)

Huffman Coding Algorithm

```
def huffman():
  n=|C|
  Q=C
  for i in range(1,n):
    z = Node()
    z.left = extract_min(Q)
    z.right = extract_min(Q)
    z.freq = z.left.freq+z.right.freq
    insert(Q,z)
  return extract_min(Q)
```

Shortest Path Problem

- Find the shortest (minimum weight) path between a particular pair of vertices
    - Weighted directed graph
    - Non-negative edge weights
- Dijkstra's algorithm
    - Solves the “one vertex, shortest path” problem
- Rough solution: Create a table of information about the currently known best way to reach each vertex (distance, previous vertex) and improve it until it reaches the best solution

Dijkstra's Algorithm

- Principle of Optimality Applies
- $A^{(k)}(i,j)$ = length of the shortest path from node i to j where the label of every intermediate node is ≤ k
- $A^{(k)}(i,j) = \min(A^{(k-1)}(i,j), A^{(k-1)}(i,k)+A^{(k-1)}(k,j))$

Dijkstra's Algorithm

```
def dijkstra(v1,v2):
  E = {all vertices}
  while E not empty:
    v = vertex with min distance
    pop v from E
    for each outgoing edge e of v:
      e is (v1,v2) edge
      dist = v.distance + e.weight
      if dist < v2.distance:
        v.distance = dist
        v2.prev = v1
```

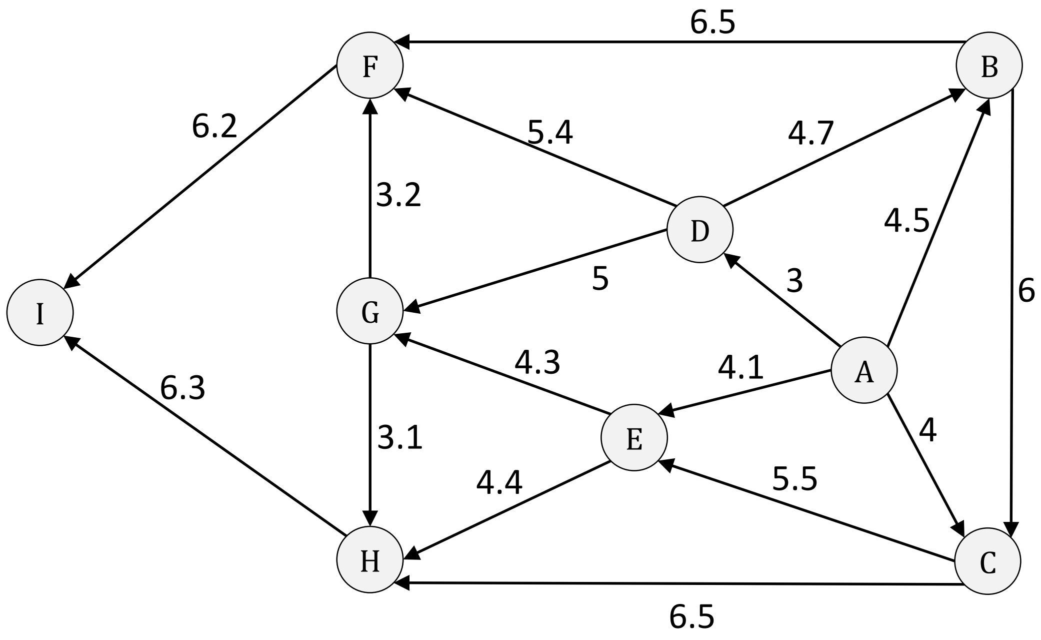
Correctness

- Dijkstra’s algorithm is a greedy algorithm
  - Choices are the best locally
  - Greedy algorithms do not guarantee a global optimum
    - Dijkstra can generate a global optimum for this convex problem
- Optimal substructure
  - For every known vertex, computed distance is the shortest distance to that vertex from the origin
  - For every unknown vertex v, its computed distance is the shortest path to v from the origin, considering only currently known vertices and v

Time Complexity

- Initialization: $O(|V|)$ using heap
- For all vertices: $O(|V|)$
- Find and remove min distance vertices $O(\log|V|)$
- Potentially $|E|$ updates
- Update weights $O(\log|V|)$
- Reconstruct path $O(|E|)$
- Total: $O(|V|\log|V|+|E|\log|V|)=O(|E|\log|V|)$

Example


6.5
F
B
6.2
5.4
4.7
3.2
D
4.5
5
3
6
I
G
4.3
4.1
A
6.3
3.1
E
4
4.4
5.5
H
C
6.5