

Greedy Algorithms

CSE 780

Reading: Sections 16.1, 16.2, 16.3, Chapter 23.

1 Introduction

Optimization Problem:

Construct a sequence or a set of elements $\{x_1, \dots, x_k\}$ that satisfies given constraints and optimizes a given objective function.

The Greedy Method

```
for  $i \leftarrow 1$  to  $k$  do
    select an element for  $x_i$  that looks best at the moment
```

Remarks

- The greedy method does not necessarily yield an optimum solution.
- Once you design a greedy algorithm, you typically need to do one of the following:
 1. Prove that your algorithm always generates optimal solutions (if that is the case).
 2. Prove that your algorithm always generates near-optimal solutions (especially if the problem is NP-hard).
 3. Show by simulation that your algorithm generates good solutions.
- A partial solution is said to be *feasible* if it is contained in an optimum solution. (An optimum solution is of course feasible.)
- A choice x_i is said to be *correct* if the resulting (partial) solution $\{x_1, \dots, x_i\}$ is feasible.
- If every choice made by the greedy algorithm is correct, then the final solution will be optimum.

2 Activity Selection Problem

Problem: Given n intervals (s_i, f_i) , where $1 \leq i \leq n$, select a maximum number of mutually disjoint intervals.

Greedy Algorithm:

Greedy-Activity-Selector

Sort the intervals such that $f_1 \leq f_2 \leq \dots \leq f_n$

$A \leftarrow \emptyset$

$f \leftarrow -\infty$

for $i \leftarrow 1$ **to** n

if $f \leq s_i$ **then**

 include i in A

$f \leftarrow f_i$

return A

Proof of Optimality

Theorem 1 *The solution generated by Greedy-Activity-Selector is optimum.*

Proof. Let $A = (x_1, \dots, x_k)$ be the solution generated by the greedy algorithm, where $x_1 < x_2 < \dots < x_k$. It suffices to show the following two claims.

(1) A is feasible.

(2) No more interval can be added to A without violating the “mutually disjoint” property.

Claim (2) is obvious, and we will prove claim (1) by showing that for any i , $0 \leq i \leq k$, the (partial) solution $A_i = (x_1, \dots, x_i)$ is feasible. (A_i is feasible if it is the prefix of an optimum solution.)

Induction Base: $A_0 = \emptyset$ is obviously feasible.

Induction Hypothesis: Assume A_i is feasible, where $0 \leq i < k$.

Induction Step: We need to show that A_{i+1} is feasible. By the induction hypothesis, A_i is a prefix of some optimum solution, say $B = (x_1, \dots, x_i, y_{i+1}, \dots, y_m)$.

- If $x_{i+1} = y_{i+1}$, then A_{i+1} is a prefix of B , and so feasible.
- If $x_{i+1} \neq y_{i+1}$, then $f_{x_{i+1}} \leq f_{y_{i+1}}$, i.e.,

finish time of interval $x_{i+1} \leq$ finish time of interval y_{i+1} .

Substituting x_{i+1} for y_{i+1} in B yields an optimum solution that contains A_{i+1} . So, A_{i+1} is feasible.

Q.E.D.

3 Huffman Codes

Problem: Given a set of n characters, C , with each character $c \in C$ associated with a frequency $f(c)$, we want to find a binary code, $code(c)$, for each character $c \in C$, such that

1. no code is a prefix of some other code, and
2. $\sum_{c \in C} f(c)|code(c)|$ is minimum, where $|code(c)|$ denotes the length of $code(c)$.

(That is, given n nodes with each node associated with a frequency, use these n nodes as leaves and construct a binary tree T such that $\sum f(x)depth(x)$ is minimum, where x ranges over all leaves of T and $depth(x)$ means the depth of x in T . Note that such a tree must be *full*, every non-leaf node having two children.)

Greedy Algorithm:

Regard C as a forest with $|C|$ single-node trees

repeat

 merge two trees with least frequencies

until it becomes a single tree

Implementation:Huffman(C) $n \leftarrow |C|$ initialize a priority queue, Q , to contain the n elements in C **for** $i \leftarrow 1$ **to** $n - 1$ **do** $z \leftarrow \text{Get-A-New-Node}()$ $left[z] \leftarrow x \leftarrow \text{Delete-Min}(Q)$ $right[z] \leftarrow y \leftarrow \text{Delete-Min}(Q)$ $f[z] \leftarrow f[x] + f[y]$ insert z to Q **return** Q **Time Complexity:** $O(n \log n)$.

Proof of Correctness:

The algorithm can be rewritten as:

```
Huffman( $C$ )
  if  $|C|=1$  then return a single-node tree;
  let  $x$  and  $y$  be the two characters in  $C$  with least frequencies;
  let  $C' = C \cup \{z\} - \{x, y\}$ , where  $z \notin C$  and  $f(z) = f(x) + f(y)$ ;
   $T' \leftarrow$  Huffman( $C'$ );
   $T \leftarrow T'$  with two children  $x, y$  added to  $z$ ;
  return( $T$ ).
```

Lemma 1 *If T' is optimal for C' , then T is optimal for C .*

Proof. Assume T' is optimal for C' . First observe that

$$\text{Cost}(T) = \text{Cost}(T') + f(x) + f(y).$$

To show T optimal, we let α be any optimal binary tree for C , and show $\text{Cost}(T) \leq \text{Cost}(\alpha)$.

Claim: We can modify α so that x and y are children of the same node, without increasing its cost.

Let β be the resulting tree, which has the same cost as α . Let z denote the common parent of x and y . Let β' be the tree that is obtained from β by removing x and y from the tree. β' is a binary tree for C' . We have the relation

$$\text{Cost}(\beta) = \text{Cost}(\beta') + f(x) + f(y).$$

Since T' is optimal for C' ,

$$\text{Cost}(T') \leq \text{Cost}(\beta')$$

which implies

$$\text{Cost}(T) \leq \text{Cost}(\beta) = \text{Cost}(\alpha).$$

Q.E.D.

Theorem 2 *The Huffman algorithm produces an optimal prefix code.*

Proof. By induction on $|C|$.

I.B.: If $|C| = 1$, it is trivial.

I.H.: Suppose that the Huffman code is optimal whenever $|C| \leq n - 1$.

I.S.: Now suppose that $|C| = n$. Let x and y be the two characters with least frequencies in C . Let C' be the alphabet that is obtained from C by replacing x and y with a new character z , with $f(z) = f(x) + f(y)$. $|C'| = n - 1$. By the induction hypothesis, the Huffman algorithm produces an optimal prefix code for C' . Let T' be the binary tree representing the Huffman code for C' . The binary tree representing the Huffman code for C is simply the tree T' with two nodes x and y added to it as children of z . By Lemma 1, the Huffman code is optimal. **Q.E.D.**

4 Minimum Spanning Trees

Problem: Given a connected weighted graph $G = (V, E)$, find a spanning tree of minimum cost.

Assume $V = \{1, 2, \dots, n\}$.

4.1 Prim's Algorithm

function $Prim(G = (V, E))$

$E' \leftarrow \emptyset$

$V' \leftarrow \{1\}$

for $i \leftarrow 1$ **to** $n - 1$ **do**

 find an edge (u, v) of minimum cost such that $u \in V'$ and $v \notin V'$

$E' \leftarrow E' \cup \{(u, v)\}$

$V' \leftarrow V' \cup \{v\}$

return (T)

Implementation:

- The given graph is represented by a two-dimensional array $cost[1..n, 1..n]$.
- To represent V' , we use an array called $nearest[1..n]$, defined as below:

$$nearest[i] = \begin{cases} 0 & \text{if } i \in V' \\ \text{the node in } V' \text{ that is "nearest" to } i, & \text{if } i \notin V' \end{cases}$$

- Initialization of $nearest$:
 $nearest(1) = 0$;
 $nearest(i) = 1$ for $i \neq 1$.

- To implement “find an edge (u, v) of minimum cost such that $u \in V'$ and $v \notin V'$ ”:

```

min ← ∞
for i ← 1 to n do
  if nearest(i) ≠ 0 and cost(i, nearest(i)) < min then
    min ← cost(i, nearest(i))
    v ← i
    u ← nearest(i)

```

- To implement “ $V' \leftarrow V' \cup \{v\}$ ”, we update *nearest* as follows:

```

nearest(v) ← 0
for i ← 1 to n do
  if nearest(i) ≠ 0 and cost(i, v) < cost(i, nearest(i)) then
    nearest(i) ← v

```

Complexity: $O(n^2)$

Correctness Proof:

A set of edges is said to be *promising* if it can be expanded to a minimum cost spanning tree. (The notion of “promising” is the same as that of “feasible”.)

Lemma 2 *If a tree T is promising and $e = (u, v)$ is an edge of minimum cost such that u is in T and v is not, then $T \cup \{(u, v)\}$ is promising.*

Proof. Let T_{\min} be a minimum spanning tree of G such that $T \subseteq T_{\min}$. If $e \in T_{\min}$, then there is nothing to prove. If $e \notin T_{\min}$, adding e to T_{\min} will create a cycle. The cycle contains an edge $e' = (u', v') \neq e$ such that u' is in T and v' is not. Since e has minimum cost, $\text{cost}(e) \leq \text{cost}(e')$. Substituting e for e' will result in a spanning tree T'_{\min} that contains $T \cup \{e\}$. Obviously, $\text{cost}(T'_{\min}) \leq \text{cost}(T_{\min})$. Therefore, T'_{\min} is a minimum spanning tree, and $T \cup \{e\}$ is promising. **Q.E.D.**

Theorem 3 *The tree generated by Prim’s algorithm has minimum cost.*

Proof. Let $T_0 = \emptyset$ and T_i ($1 \leq i \leq n-1$) be the tree as of the end of the i th iteration. T_0 is promising. By Lemma 1 and induction, T_1, \dots, T_{n-1} are all promising. So, T_{n-1} is a minimum cost spanning tree. **Q.E.D.**

4.2 Kruskal's Algorithm

Sort edges by increasing cost

$T \leftarrow \emptyset$

repeat

$(u, v) \leftarrow$ next edge

if adding (u, v) to T will not create a cycle **then**

$T \leftarrow T \cup \{(u, v)\}$

until T has $n - 1$ edges

Analysis: If we use an array $E[1..e]$ to represent the graph and use the union-find data structure to represent the forest T , then the time complexity of Kruskal Algorithm is $O(e \log n)$, where e is the number of edges in the graph.

4.3 The union-find data structure

There are N objects numbered $1, 2, \dots, N$.

Initial situation: $\{1\}, \{2\}, \dots, \{N\}$.

We expect to perform a sequence of *find* and *union* operations.

Data structure: use an integer array $A[1..N]$ to represent the sets.

```
procedure init( $A$ )
```

```
  for  $i \leftarrow 1$  to  $N$  do  $A[i] \leftarrow 0$ 
```

```
procedure find( $x$ )
```

```
   $i \leftarrow x$ 
```

```
  while  $A[i] > 0$  do  $i \leftarrow A[i]$ 
```

```
  return( $i$ )
```

```
procedure union( $a, b$ )
```

```
  case
```

```
     $A[a] < A[b]$ :  $A[b] \leftarrow a$ 
```

```
     $A[a] > A[b]$ :  $A[a] \leftarrow b$ 
```

```
     $A[a] = A[b]$ :  $A[a] \leftarrow b, A[b] \leftarrow A[b] - 1$ 
```

```
  end
```

Theorem 4 *After an arbitrary sequence of union operations starting from the the initial situation, a tree containing k nodes will have a height at most $\lfloor \log k \rfloor$.*

5 Single Source Shortest Path

- Problem: Given an undirected, connected, weighted graph $G(V, E)$ and a node $s \in V$, find a shortest path between s and x for each $x \in V$. (Assume positive weights.)
- Assume $V = \{1, 2, \dots, n\}$.
- Observation: a shortest path between s and v may only pass through nodes which are closer to s than v .
- That is, if $d(v_1) \leq d(v_2) \leq d(v_3) \leq \dots \leq d(v_n)$, where $d(x)$ denotes the shortest distance between s and x , then shortest-path(s, v_k) may only pass through nodes in $\{v_1, \dots, v_{k-1}\}$.
- This suggests that we compute shortest paths for nodes v_k in the order of $v_1, v_2, v_3, \dots, v_n$.
- The resulting paths form a spanning tree.
- We will construct such a tree using an algorithm similar to Prim's.

Dijkstra's Algorithm($G = (V, E), s$)

$D[s] \leftarrow 0$

$Parent[s] \leftarrow 0$

$V' \leftarrow \{s\}$

for $i \leftarrow 1$ **to** $n - 1$ **do**

 find an edge (u, v) such that $u \in V', v \notin V'$

 and $D[u] + length[u, v]$ is minimum;

$D[v] \leftarrow D[u] + length[u, v];$

$Parent[v] \leftarrow u;$

$V' \leftarrow V' \cup \{v\};$

endfor

Data Structures:

- The given graph: $length[1..n, 1..n]$.
- Shortest distances: $D[1..n]$, where $D[i]$ = the shortest distance between s and i . Initially, $D[s] = 0$.
- Shortest paths: $Parent[1..n]$. Initially, $Parent[s] = 0$.
- $nearest[1..n]$, where

$$nearest[i] = \begin{cases} 0 & \text{if } i \in V' \\ \text{the node } x \text{ in } V' \text{ that} \\ \quad \text{minimizes } D[x] + length[x, i], & \text{if } i \notin V' \end{cases}$$

Complexity: $O(n^2)$