Announcement

We'll have a quiz on Tuesday, April. 12th on Huffman Coding

Announcement

Homework #6 has been posted in Blackboard and course website

Due: 10:05 am EST, Thursday, April 21

Dijkstra's Algorithm on Undirected Graph

Similar to Prim's MST algorithm, with the following difference:

- Start with tree consisting of one vertex – **source**
- "grow" tree one vertex/edge, which has minimum length of path, at a time to produce spanning tree
  - Construct a series of expanding subtrees T_1, T_2, …
- Keep track of shortest path from source to each of the vertices in T_i
- at each stage construct T_{i+1} from T_i: add ~~minimum weight edge~~ connecting a vertex in tree (T_i) to one not yet in tree
  - choose from "fringe" nodes
  - (this is the "greedy" step!)

edge (v,w) with lowest d(s,v) + d(v,w)

source *destination*

- algorithm stops when all vertices are included

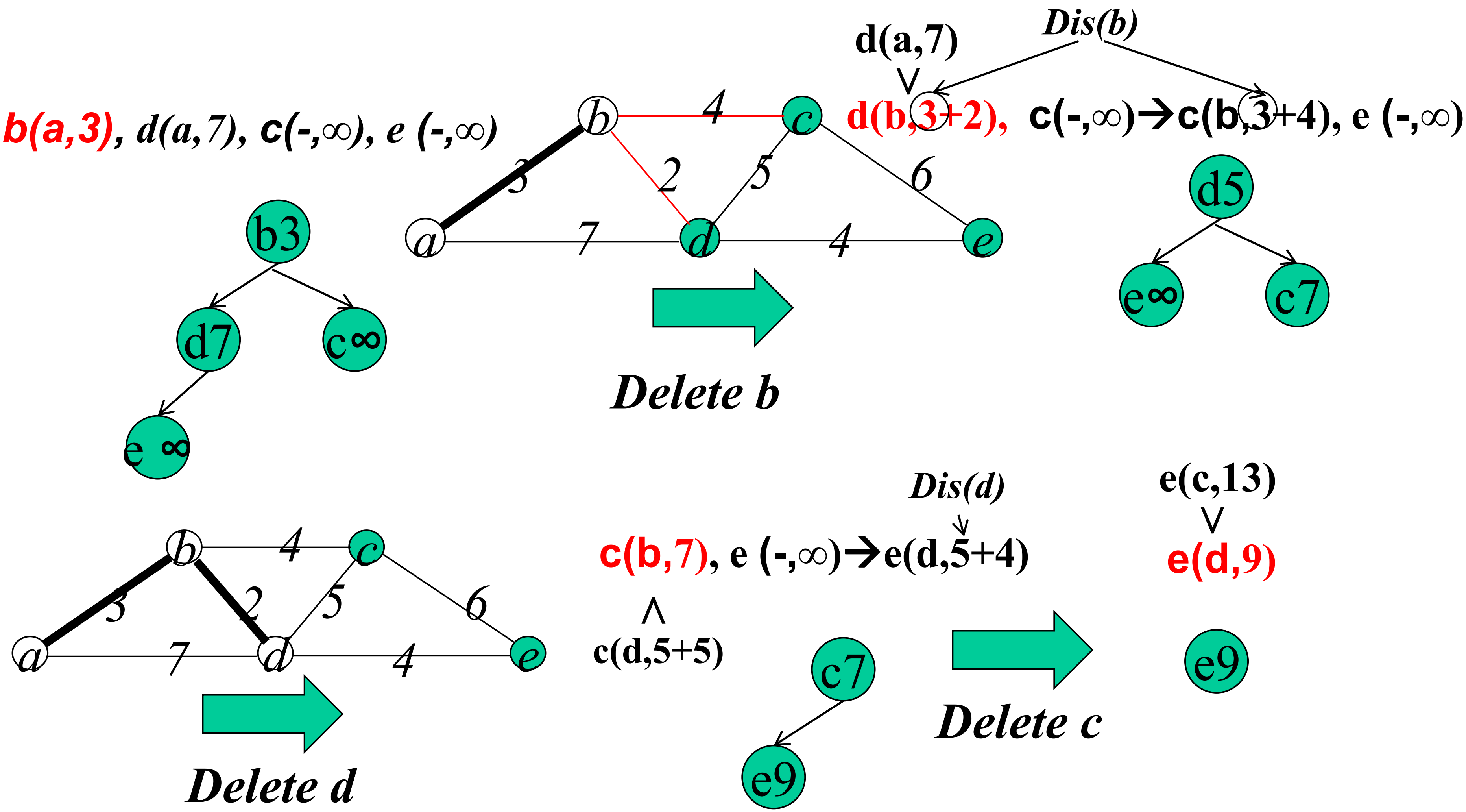
Pseudo Code

ALGORITHM *Dijkstra*(G, s)

//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph $G = \langle V, E\rangle$ with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V

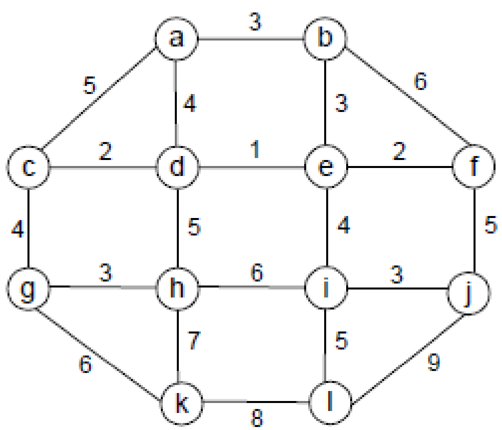
```
Initialize(Q)    //initialize vertex priority queue to empty
for every vertex v in V do
    d_v ← ∞;  p_v ← null
    Insert(Q, v, d_v)    //initialize vertex priority in the priority queue
d_s ← 0;  Decrease(Q, s, d_s)    //update priority of s with d_s      source
V_T ← ∅
for i ← 0 to |V| − 1 do
    u* ← DeleteMin(Q)    //delete the minimum priority element
    V_T ← V_T ∪ {u*}
    for every vertex u in V − V_T that is [adjacent to u*] do
        if d_u* + w(u*, u) < d_u
            d_u ← d_u* + w(u*, u);  p_u ← u*
            Decrease(Q, u, d_u)
```

Example



b(a,3), d(a,7), c(-,∞), e (-,∞)
d(a,7)
Dis(b)
d(b,3+2), c(-,∞)→c(b,3+4), e (-,∞)
Delete b
b3
d7
c∞
e ∞
d5
e∞
c7
Dis(d)
e(c,13)
c(b,7), e (-,∞)→e(d,5+4)
e(d,9)
c(d,5+5)
c7
e9
Delete c
e9
Delete d

An Example

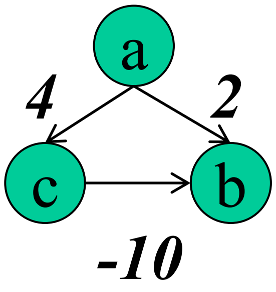


Tree vertices	Priority queue for the fringe vertices and unseen vertices
a(-,0)	b(a,3), d(a,4), c(a,5), e(-,∞), f(-,∞), g(-,∞), h(-,∞), i(-,∞), j(-,∞), k(-,∞), l(-,∞)
b(a,3)	d(a,4), c(a,5), e(b,3+3), f(b,3+6), g(-,∞), h(-,∞), i(-,∞), j(-,∞), k(-,∞), l(-,∞)
d(a,4)	c(a,5), e(d,4+1), f(b,9), h(d,4+5), g(-,∞), i(-,∞), j(-,∞), k(-,∞), l(-,∞)
c(a,5)	e(d,5), f(b,9), h(d,4+5), g(c,5+4), i(-,∞), j(-,∞), k(-,∞), l(-,∞)
e(d,5)	f(e,5+2), h(d,9), g(c,9), i(e,5+4), j(-,∞), k(-,∞), l(-,∞)
f(e,7)	h(d,9), g(c,9), i(e,9), j(f,7+5), k(-,∞), l(-,∞)
h(d,9)	g(c,9), i(e,9), j(f,12), k(h,9+7), l(-,∞)
g(c,9)	i(e,9), j(f,12), k(g,9+6), l(-,∞)
i(e,9)	j(f,12) , l(i,9+5), k(g,9+6)
j(f,12)	l(i,14), k(g,15)
l(i,14)	k(g,15)
k(g,15)	

Notes on Dijkstra's algorithm

Applicable to both undirected and directed graphs

Doesn't work with negative weights



b(a,2)*, *c(a,4)

c(a,4)

a->c->b = -6 $\longrightarrow$ ***Shorter than the existing path!***

Efficiency:

- O(|*E*| log |*V*|) when the graph is represented by adjacency linked list and priority queue is implemented by a min-heap.
- Reason: the whole process is almost the same as Prim's algorithm. Following the same analysis in Prim's algorithm, we can see that it has the same complexity as Prim's algorithm.

Huffman Trees – Coding Problem

Text consists of characters from some *n*-character alphabet

In communication, we need to code each character by a sequence of bits – codeword

Fixed-length encoding: code for each character has *m* bits

- Standard seven-bit ASCII code does this

Variable-length encoding: using shorter codeword for more frequent characters and longer codeword for less frequent ones

- For example, in Morse telegraph code, a(.-), e(.), q(--.-), z(--..)
- How many bits represent the characters?
- Two important questions:
  - What is the lower bound of average number of bits?
  - How to avoid confusion in code decoding?

Basic Concept of Information and Coding

You have *m* message, 1, 2, …,*m* to transfer, with probability p_1, p_2, … p_m, using digital communication, how many bits is needed for coding?

Example: you have two messages; you only need 1 bit to code: *0* represents message "1" and *1* represents message "2".

<u>Shannon Theorem</u>: let $$H = -\sum_i p_i \log_2 p_i$$

If *B* is the average # bits per message for the best code, then

$$H \le B \le H + 1$$

Huffman Tree – From coding to a binary tree

To avoid confusion, here we consider prefix codes, where no codeword is a prefix of a codeword of another character

This way, we can scan the bit string constructed from a text from left to right until get the first group of bits that is a valid codeword for some character. For example, “1” for “A” and “01” for “B”.

Solution:

Associate the characters with leaves of a binary tree in which all the left edges are labeled by 0 and all the right edges are labeled by 1 (or vice versa)

The codeword of a character (leaf) is a sequence of labels along the path from the root to this leaf

Huffman tree can reduce the total bit string by assigning shorter codeword (higher level in the tree) to frequent character and longer codeword (lower level) to less frequent ones.

Huffman Coding Algorithm

Step 1: Initialize *n* one-node trees and label them with the characters in a dictionary. Record the frequency of each character in its tree's root to indicate the tree's weight

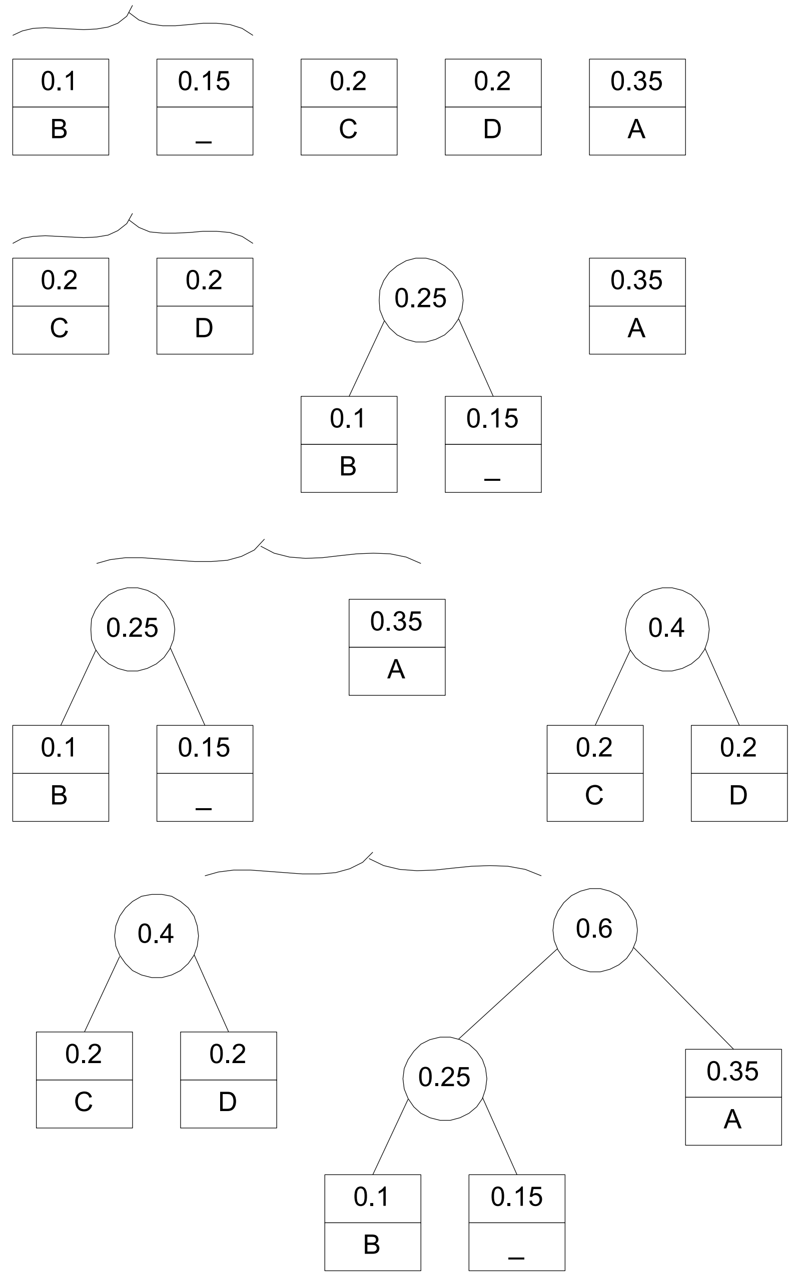
Step 2: Repeat the following operation until a single tree is obtained.

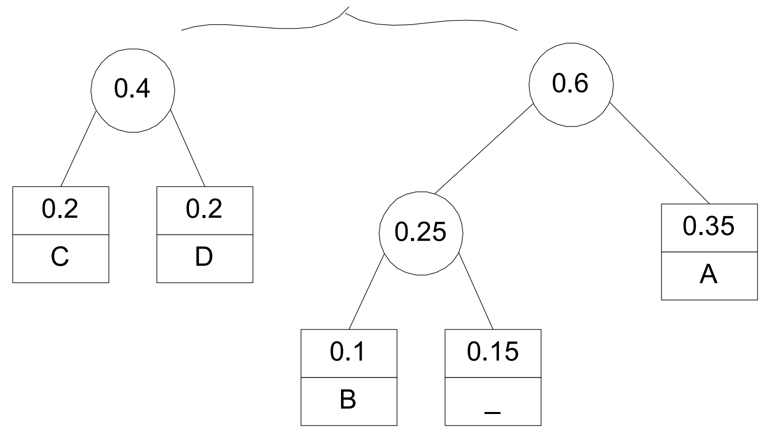
- Find two trees with the smallest weights.
- Make them the left and right subtrees of a new tree
- record the sum of their weights in the root of the new tree as its weight

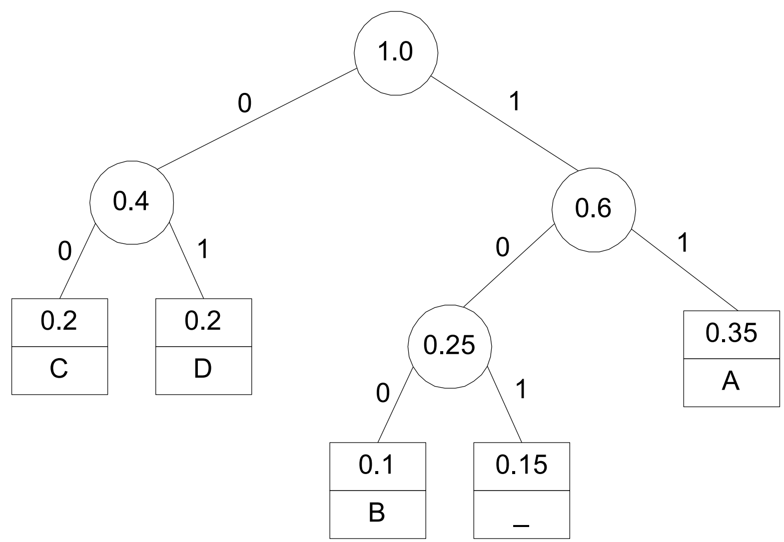
Example: alphabet {A, B, C, D, _} with frequency

character	A	B	C	D	_
probability	0.35	0.1	0.2	0.2	0.15

Repeat:
Step1: Order the nodes based on frequency
Step2: Merge the two nodes with smallest probabilities
Until one node left




0.4
0.2
C
0.2
D
0.6
0.25
0.1
B
0.15
_
0.35
A


1.0
0
1
0.4
0
1
0.2
C
0.2
D
0.6
0
1
0.25
0
1
0.1
B
0.15
_
0.35
A

The Huffman Code is

Character	A	B	C	D	_
Probability	0.35	0.1	0.2	0.2	0.15
Codeword	11	100	00	01	101
Code length	2	3	2	2	3

Average # of bits per character (average code length)=

$$\sum_{i=1}^{5} p_i * l_i = 0.35 * 2 + 0.1 * 3 + 0.2 * 2 + 0.2 * 2 + 0.15 * 3 = 2.25$$

Probability of a character Code length of a character

Therefore, BAD is encoded as 1001101 and 100110110111010 is decoded as BAD_AD

Notes on Huffman Tree (Coding)

The expected average number of bits per character is 2.25

Fixed-length encoding needs at 3 bits for each character

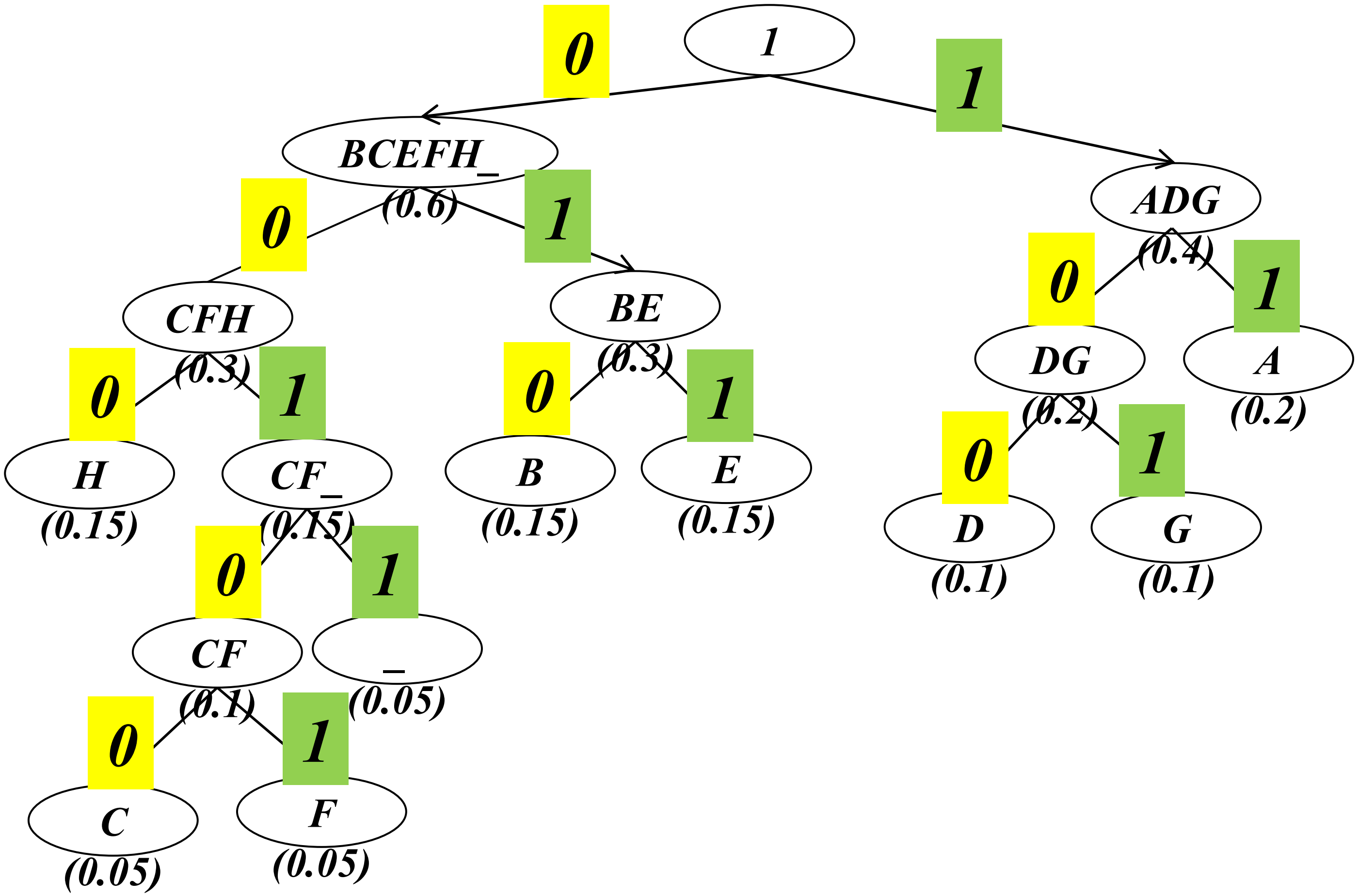
This is an important technique for file (data) compression

Huffman tree/coding has more general applications:

- Assign *n* positive numbers w_1, w_2, …, w_n to the *n* leaves of a binary tree
- We want to minimize weighted path length $\boxed{\sum_{i=1}^{n} l_i w_i}$ with l_i be the depth of the leaf *l*
- This has particular applications in making decisions – decision trees

Another example

Character	A	B	C	D	E	F	G	H	_
Probability	0.2	0.15	0.05	0.1	0.15	0.05	0.1	0.15	0.05


1
0
1
BCEFH_
(0.6)
ADG
(0.4)
0
1
0
1
CFH
(0.3)
BE
(0.3)
DG
(0.2)
A
(0.2)
0
1
0
1
0
1
H
(0.15)
CF_
(0.15)
B
(0.15)
E
(0.15)
D
(0.1)
G
(0.1)
0
1
CF
(0.1)
_
(0.05)
0
1
C
(0.05)
F
(0.05)

Character	A	B	C	D	E	F	G	H	_
Probability	0.2	0.15	0.05	0.1	0.15	0.05	0.1	0.15	0.05
Codeword	11	010	00100	100	011	00101	101	000	0011
Code length	2	3	5	3	3	5	3	3	4

Average number of bits per character (code length):

$$0.2 * 2 + 0.15 * 3 + 0.05 * 5 + 0.1 * 3 + 0.15 * 3 + 0.05 * 5 + 0.1 * 3 + 0.15 * 3 + 0.05 * 4 = 3.05$$

Reading Assignments

Read Chapter 10. Iterative Improvement

Chapter 11: Limitations of Algorithm Power

Basic Asymptotic Efficiency Classes (big O, big Θ, and big Ω)

		n=10	n=100
1	**constant**	**constant**	**constant**
log *n*	**logarithmic**	**1** (with base 10)	2 (with base 10)
n	**linear**	**10**	**100**
n* log *n	***n* log *n***	***10*** (with base 10)	***200*** (with base 10)
n^2	**quadratic**	**100**	**10,000**
n^3	**cubic**	**1000**	**1,000,000**
2^n	**exponential**	**1024**	$\sim 1.26*10^{30}$
n!	**factorial**	**3,628,800**	$\sim 9.33*10^{157}$

Polynomial-Time Complexity

Polynomial-time complexity: the complexity of an algorithm is

$$\boxed{a_b n^b + a_{b-1} n^{b-1} + \ldots + a_1 n^1 + a_0 \Rightarrow \Theta(n^b)}$$

with a fixed degree *b*>0. Usually b<=3

If a problem can be solved in polynomial time, it is usually considered to be theoretically tractable in current computers.

When an algorithm's complexity is larger than polynomial, i.e., exponential, theoretically it is considered to be too expensive to be useful – intractable

Polynomial-Time Complexity

Polynomial time complexity

1	constant
log n	logarithmic
n	linear
n log n	n log n
n^2	quadratic
n^3	cubic
2^n	exponential
$n!$	factorial

List of Problems

Sorting $O(nlogn)$

Searching $O(n)$

All shortest paths in a graph $O(|V|^{\mathbf{3}})$

Minimum spanning tree $O(|E|\log|V|)$

Assignment problem $O(n!)\sim O(\boldsymbol{n^3})$

Towers of Hanoi $O(2^n)$

Knapsack problem $O(2^n)$

Traveling salesman problem $\boldsymbol{O}(n!)\sim O(\boldsymbol{n^2}2^n)$ **–** Current record 85,900 cities (Applegate et al. 2006)
http://en.wikipedia.org/wiki/Travelling_salesman_problem#Computational_complexity

…

Lower Bound

Problem A can be solved by algorithms $a_1, a_2, \ldots, a_p$

Problem B can be solved by algorithms $b_1, b_2, \ldots, b_q$

We may ask

- Which algorithm is more efficient? This makes more sense when the compared algorithms solve the same problem
  - It's not fair to compare selection sorting with Warshall's algorithm
- Which problem is more complex? We may compare the complexity of the best algorithm for **A** and the best algorithm for **B**

For each problem, we want to know the lower bound: the best possible algorithm's efficiency for a problem → $\Omega(.)$

Tight lower bound: we have found an algorithm in the this lower-bound efficiency class $\Theta(.)$. The constant factor makes the difference.

Trivial Lower Bound

Many problems need to 'read' all the necessary items and write the 'output'

→ Their sizes provide a trivial lower bound

Example:

1. Generate all permutations of n distinct items → $\Omega(n!)$
   Why?
   Is this a tight lower bound? ***Yes.***
2. Evaluate the polynomial at a given *x*

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

→ $\Omega(n)$, Is this tight? ***Yes.***

Notes on Trivial Lower Bound

Multiplying two *n*x*n* matrices → $\Omega(n^2)$

- because we need to process $2n^2$ elements and output n^2 elements
- We do not know whether this is tight – a lower bound of $\Omega(n^2 log n)$ has been proven Raz 2002

Many trivial lower bounds are too low to be useful

- TSP → $\Omega(n^2)$ because its input is $n(n-1)/2$ intercity distance and output is $n+1$ city in sequence
- There is no known polynomial-time algorithm to solve it

Trivial lower bound sometime have problems

- We do not need to process all the input elements
- For example: searching an element in a sorted array. What is its complexity?

Information-Theoretic Arguments

This approach seeks to establish a lower bound based on the amount of information, which has to produce – an information-theoretic lower bound

Recall the problem of guess the number from 1...n by asking ‘yes/no’ questions

Fundamentally, it is a coding problem. If the input number can be encoded into m bits, each ‘yes/no’ question just resolve one bits and therefore, the lower bound is m

We know that $m=\log_2 n$

Solution: a decision tree. We will apply the decision tree to find the lower bound for several problems

Find the Smallest from three numbers using comparison

Leaves in the decision tree is the possible output. The output size is at least *3* (maybe larger than *3*) here

Complexity for the worst case: the height of this decision tree

Given *L* leaves, the height of the binary tree is at least $\lceil \log_2 L \rceil$