ECS122A Lecture Notes on Algorithm Design and Analysis

Winter 2016

Professor Zhaojun Bai

Overview

Introduction and Getting Started

Introduction

- Algorithm is a tool for solving a well-specified computational problem
- Algorithms as a technology
- Basic questions about an algorithm
  1. Does it halt?
  2. Is it correct?
  3. Is it fast?
  4. How much memory does it use?
  5. How does data communicate?

Getting started: case study 1

- Problem: computing the nth Fibonacci number F_n
- Definition:

$$
\begin{aligned}
&F_0 = 0, \\
&F_1 = 1, \\
&F_n = F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2
\end{aligned}
$$

- Algorithms:
  1. Recursive
  2. Memorize the recursive
  3. Divide-and-conquer
  4. Approximate

Getting started: case study 2

- Problem: sorting
- Definition:

  Input: a sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  Output: a permutation (reordering) $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the a-sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$

- Algorithms:
  1. Insert sort
  2. Merge sort

Getting started: case study 2

Insert sort algorithm

- Idea: incremental approach
- Pseudocode (*expressing algorithm*)

```
InsertionSort(A)
n = length(A)
for j = 2 to n
    key = A[j]
    // insert ‘‘key’’ into sorted array A[1...j-1]
    i = j-1
    while i > 0 and A[i] > key do
       A[i+1] = A[i]
       i = i-1
    end while
    A[i+1] = key
end for
return A
```

Getting started: case study 2

Insert sort algorithm – *Remarks*:

- Correctness: argued by "loop-invariant" (a kind of induction)
- Complexity analysis:
  best-case, worst-case, average-case
- Insert sort is a "sort-in-place", no extra memory necessary
- Importance of writing a good pseudocode;
  "*expressing algorithm to human*"
- There is a recurisve version of insert sort, see Homework 1.

Getting started: case study 2

Merge sort algorithm

- Idea: divide-and-conquer approach
- Pseudocode

```
MergeSort(A,p,r)             // Merge-sort of array A[p...r]
if p < r then                // check for base case
   q = flooring( (p+r)/2 )   // divide
   MergeSort(A,p,q)          // conquer
   MergeSort(A,q+1,r)        // conquer
   Merge(A,p,q,r)            // combine
end if
```

Getting started: case study 2

- Pseudocode, cont'd

```
Merge(A,p,q,r)
n1 = q-p+1;  n2 = r-q
for i = 1 to n1         // create arrays L[1...n1+1] and R[1...n2+1]
    L[i] = A[p+i-1]
end for
for j = 1 to n2
    R[j] = A[q+j]
end for
L[n1+1] = infty; R[n2+1] = infty // mark the end of arrays L and R
i = 1; j = 1
for k = p to r          // Merge arrays L and R to A
    if L[i] <= R[j] then
       A[k] = L[i]
       i = i+1
    else
       A[k] = R[j]
       j = j+1
    end if
end for
```

Getting started: case study 2

Merge sort algorithm – *Remarks:*

- Merge sort is a divide-and-conquer algorithm consisting of three steps: divide, conquer and combine
- To sort the entire sequence A[1...n], we make the initial call

  MergeSort(A,1,n)

  where n = length(A).
- Complexity analysis:

$$T(n) = 2T\left(\frac{n}{2}\right) + n - 1 = O(n\lg(n))$$

Growth of Functions and Asymptotic Notations

Overview

- Study a way to describe behavior of functions in the limit ... *asymptotic efficiency*
- Describe growth of functions
- Focus on what's important by abstracting lower-order terms and constant factors
- How we indicate running times of algorithms
- A way to compare "sizes" of functions

  $$O \approx \leq$$
  $$\Omega \approx \geq$$
  $$\Theta \approx =$$

  In addition, $o \approx <$ and $\omega \approx >$

O-notation

- $g(n)$ is an asymptotic upper bound for $f(n)$:

$$f(n) = O(g(n))$$

  if there exists constants c and n_0 such that

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{for } n \geq n_0$$

- Example:
  - $2n + 10 = O(n^2)$, pick $c = 1$ and $n_0 = 5$

More on O-notation

- $O(g(n))$ is a set of functions

  $$O(g(n)) = \{f(n) : \exists\, c, n_0 \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for } n \geq n_0\}$$

- Examples of functions in $O(n^2)$:
  $n^2 + n$
  $n^2 + 1000n$
  $1000n^2 + 1000n$
  $n/1000$
  $n^2/\lg n$

Ω-notation

- $g(n)$ is an asymptotic lower bound for $f(n)$.

$$f(n) = \Omega(g(n))$$

  if there exists constants c and n_0 such that

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{for } n \geq n_0$$

- Example:
  - $\sqrt{n} = \Omega(\lg n)$, pick $c = 1$ and $n_0 = 16$

More on Ω-notation

- $\Omega(g(n))$ is a set of functions

  $\Omega(g(n)) = \{f(n) :\ \exists\, c, n_0 \text{ such that } 0 \le c \cdot g(n) \le f(n) \text{ for } n \ge n_0\}$

- Examples of functions in $\Omega(n^2)$:
  n^2
  $n^2 + n$
  $n^2 - n$
  $1000n^2 + 1000n$
  $1000n^2 - 1000n$
  $n^{2.00001}$
  $n^2 \lg n$
  n^3

Θ-notation

- $g(n)$ is an asymptotic tight bound for $f(n)$.

$$f(n) = \Theta(g(n))$$

  if there exists constants c_1, c_2 and n_0 such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 g(n) \quad \text{for } n \geq n_0$$

- Example:
  - $\frac{1}{2}n^2 - 2n = \Theta(n^2)$, pick $c_1 = \frac{1}{4}$ $c_2 = \frac{1}{2}$ and $n_0 = 8$.
  - If $p(n) = \sum_{i=1}^{d} a_i n^i$ and $a_d > 0$, then $p(n) = \Theta(n^d)$

More on Θ-notation

- $\Theta(g(n))$ is a set of functions

  $\Omega(g(n)) =$
  $\{f(n) : \ \exists\ c_1, c_2, n_0 \ \text{ such that } \ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 g(n) \ \text{ for } n \geq n_0\}$

- Examples of functions in $\Theta(n^2)$:
  n^2
  $n^2 + n$
  $n^2 - n$
  $1000n^2 + 1000n$
  $1000n^2 - 1000n$

Theorem

Theorem. O and Ω iff Θ.

Using limits for comparing orders of growth

In order to determine the relationship between $f(n)$ and $g(n)$, it is often usefuly to examine

$$\lim_{n\to\infty}\frac{f(n)}{g(n)} = L$$

The possible outcomes:

1. $L = 0$: $f(n) = O(g(n))$
2. $L = \infty$: $f(n) = \Omega(g(n))$
3. $L \neq 0$ is finite: $f(n) = \Theta(g(n))$
4. There is no limit: this technique cannot be used to determine the asymptotic relationship between $f(n)$ and $g(n)$.

Examples

1. $f(n) = n^2$ and $g(n) = n \lg n$
$$n^2 = \Omega(n \lg n)$$
2. $f(n) = n^{100}$ and $g(n) = 2^n$
$$n^{100} = O(2^n)$$
3. $f(n) = 10n(n+1)$ and $g(n) = n^2$
$$10n(n+1) = \Theta(n^2)$$

Divide-and-Conquer recurrences and Master Theorem

Divide-and-Conquer recurrences

- Divide-and-Conquer (DC) recurrence

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

  where constants $a \geq 1$ and $b > 1$, function $f(n)$ is nonnegative.

- Example: the cost function of Merge Sort

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

  where
  - $a = 2$ (the number of subproblems),
  - $b = 2$ ($n/2$ is the size of subproblems),
  - $f(n) = n$ is the cost to divide and combine.

The master theorem/method to solve DC recurrences

Case 1: *If* $n^{\log_b a}$ *is polynomially larger than* $f(n)$, *i.e,*

$$\frac{n^{\log_b a}}{f(n)} = \Omega(n^\epsilon) \quad \textit{for some constant } \epsilon > 0$$

Then

$$T(n) = \Theta(n^{\log_b a}).$$

Example: $T(n) = 7 \cdot T(\frac{n}{2}) + \Theta(n^2)$

The master theorem/method to solve DC recurrences

Case 2: *If* $n^{\log_b a}$ *and* $f(n)$ *are on the same order, i.e.,*

$$f(n) = \Theta(n^{\log_b a})$$

Then

$$T(n) = \Theta(n^{\log_b a} \lg n)$$

Example: $T(n) = 2 \cdot T(\frac{n}{2}) + \Theta(n)$

The master theorem/method to solve DC recurrences

Case 3: *If $f(n)$ is polynomially greater than $n^{\log_b a}$, i.e.,*

$$\frac{f(n)}{n^{\log_b a}} = \Omega(n^{\epsilon}) \quad \textit{for some constant } \epsilon > 0$$

*and $f(n)$ satisfies the regularity condition (*see next slide*).*
Then

$$T(n) = \Theta(f(n))$$

Example: $T(n) = 4 \cdot T(\frac{n}{2}) + n^3$

The master theorem – Remarks

1. $f(n)$ satisfies the *regularity condition* if

$$af\left(\frac{n}{b}\right) \leq cf(n)$$

   for some constant $c < 1$ and for all sufficient large n.

2. The proof of the master theorem is involved, shown in section 4.6, which we can safely skip for now.
3. The master method cannot solve every divide and conquer recurrences.

Divide and Conquer Algorithms

The Maxiumum Subarray Problem

Problem:

Input: *an array* $A[1...n]$ *of (positive/negative) numbers.*

Output: *Indices* i *and* j *such that* $A[i...j]$ *has the greatest sum of any nonempty, contiguous subarray of* A*, along with the sum of the values in* $A[i...j]$.

Note: Maximum subarray might not be unique, though its value is, so we speak of a maximum subarray, rather than the maximum subarray.

The Maxiumum Subarray Problem

Example 1:

Day	0	1	2	3	4
Price	10	11	7	10	6
Change $A[.]$		1	-4	3	-4

Maximum subarray: $A[3]$ $(i = j = 3)$, $sum = 3$

Example 2:

Day	0	1	2	3	4	5	6
Price	10	11	7	10	14	12	18
Change $A[.]$		1	-4	3	4	-2	6

Maximum subarray: $A[3...6]$ $(i = 3, j = 6)$, $sum = 11$.

The Maxiumum Subarray Problem

- Subproblem: Find a maximum subarray of $A[low...high]$
- Initial call: $low = 1$ and $high = n$
- Divide-and-Conquer algorithm
  1. Divide: the (sub)array into two subarrays of as equal size as possible by finding the midpoint mid
  2. Conquer: finding maximum subarrays of $A[low...mid]$ and $A[mid + 1...high]$
  3. Combine:
     - finding a max-subarray that crosses the midpoint
     - returning the best of the three
- This strategy works because any subarray must either lie entirely in one side of midpoint or cross the midpoint.

Divide-and-Conquer algorithm, pseudocode

```
MaxSubarray(A,low,high)
if high == low                  // base case: only one element
   return (low, high, A[low])
else                            // divide, conquer and combine
   mid = floor( (low + high)/2 )
   (leftlow,lefthigh,leftsum) = MaxSubarray(A,low,mid)
   (rightlow,righthigh,rightsum) = MaxSubarray(A,mid+1,high)
   (xlow,xhigh,xsum) = MaxXingSubarray(A,low,mid,high)
   // combine
   if leftsum >= rightsum and leftsum >= xsum
      return (leftlow,lefthigh,leftsum)
   else if rightsum >= leftsum and rightsum >= xsum
      return (rightlow,righthigh,rightsum)
   else
      return (xlow,xhigh,xsum)
   end if
end if
```

Divide-and-Conquer algorithm, pseudocode, cont'd

```
MaxXingSubarray(A,low,mid,high)
leftsum = -infty; sum = 0   // Find a max-subarray of A[i..mid]
for i = mid downto low
    sum = sum + A[i]
    if sum > leftsum
       leftsum = sum
       maxleft = i
    end if
end for
rightsum = -infty; sum = 0  // Find a max-subarray of A[mid+1..j]
for j = mid+1 to high
    sum = sum + A[j]
    if sum > rightsum
       rightsum = sum
       maxright = j
    end if
end for
// Return the indices i and j and the sum of the two subarrays
return (maxleft,maxright,leftsum + rightsum)
```

Divide-and-Conquer algorithm

Remarks:

1. Initial call: MaxSubarray(A,1,n)
2. Base case is when the subarray has only 1 element.
3. **Divide** by computing mid.
   Conquer by the two recursive alls to MaxSubarray.
   Combine by calling MaxXingSubarray and then determining which of the three results gives the maximum sum.
4. What does MaxSubarray returns when all elements of A are negative?

Matrix-matrix multiplication

- Problem:
  Given $n \times n$ matrices A and B, compute the product $C = AB$.
- Traditional method: triple-loop
- Strassen's method: divide-and-conquer

Matrix-matrix multiplication

Strassen's method

Step 1: Divide

$$A = \begin{array}{cc} & \begin{array}{cc} \frac{n}{2} & \frac{n}{2} \end{array} \\ \begin{array}{c} \frac{n}{2} \\ \frac{n}{2} \end{array} & \left[\begin{array}{cc} A_{11} & A_{12} \\ A_{21} & A_{22} \end{array} \right] \end{array} \quad \text{and} \quad B = \begin{array}{cc} & \begin{array}{cc} \frac{n}{2} & \frac{n}{2} \end{array} \\ \begin{array}{c} \frac{n}{2} \\ \frac{n}{2} \end{array} & \left[\begin{array}{cc} B_{11} & B_{12} \\ B_{21} & B_{22} \end{array} \right] \end{array}$$

Matrix-matrix multiplication

Strassen's method

Step 2: Compute 10 matrices by $\pm$ only:

$$
\begin{aligned}
S_1 &= B_{12} - B_{22} \\
S_2 &= A_{11} + A_{12} \\
S_3 &= A_{21} + A_{22} \\
S_4 &= B_{21} - B_{11} \\
S_5 &= A_{11} + A_{22} \\
S_6 &= B_{11} + B_{22} \\
S_7 &= A_{12} - A_{22} \\
S_8 &= B_{21} + B_{22} \\
S_9 &= A_{11} - A_{21} \\
S_{10} &= B_{11} + B_{12}
\end{aligned}
$$

Matrix-matrix multiplication

Strassen's method

Step 3: Compute 7 matrices by multiplication:

$$
\begin{aligned}
P_1 &= A_{11} \cdot S_1 \\
P_2 &= S_2 \cdot B_{22} \\
P_3 &= S_3 \cdot B_{11} \\
P_4 &= A_{22} \cdot S_4 \\
P_5 &= S_5 \cdot S_6 \\
P_6 &= S_7 \cdot S_8 \\
P_7 &= S_9 \cdot S_{10}
\end{aligned}
$$

Matrix-matrix multiplication

Strassen's method

Step 4: Add and subtract the P_i to construct submatrices C_{ij}

$$\begin{aligned} C_{11} &= P_5 + P_4 - P_2 + P_6 \\ C_{12} &= P_1 + P_2 \\ C_{21} &= P_3 + P_4 \\ C_{22} &= P_5 + P_1 - P_3 - P_7 \end{aligned}$$

The product

$$C = \begin{array}{c c} & \begin{array}{cc} \frac{n}{2} & \frac{n}{2} \end{array} \\ \begin{array}{c} \frac{n}{2} \\ \frac{n}{2} \end{array} & \left[\begin{array}{cc} C_{11} & C_{12} \\ C_{21} & C_{22} \end{array} \right] \end{array}$$

Finding the closest pair of points in one-dimension, 1

Problem:

> *Given a set S of n points on a line (unsorted), find two points whose distance is smallest.*

Bruce-force:

- Pick two of n points and compute the distance
- Cost: $\Theta(n^2)$

Finding the closest pair of points in one-dimension, 2

Algorithm 1

1. Sort the points
2. Perform a linear scan

Cost: $\Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$

Remark: Unfortunately, Algorithm 1 cannot be extended to the 2-d case.

Finding the closest pair of points in one-dimension, 3

Algorithm 2 (Divide-and-Conquer):

1. Divide the set S by some point mid (say, median) into two sets S_1 and S_2, with the property:

$$p < q \text{ for all } p \in S_1 \text{ and } q \in S_2$$

2. Conquer: finds the closest pair *recursively* on S_1 and S_2, separately, gives us two pairs of points

$$\{p_1, p_2\} \quad \text{and} \quad \{q_1, q_2\},$$

   the closest pair in S_1 and S_2, respectively. .

3. Combine: the closest pair in the set S is

$$d = \min\{|p_1 - p_2|, |q_1 - q_2|\}$$

   or

$$d' = |p_3 - q_3|,$$

   where $p_3 \in S_1$ and $q_3 \in S_2$.

Finding the closest pair of points in one-dimension, 4

Observations:

- both p_3 and q_3 must be within distance d of mid if $\{p_3, q_3\}$ is to have a distance smaller than d.
- How many points of S_1 can lie in $(mid - d, mid]$?
  Answer: at most one
- How many points of S_2 can lie in $[mid, mid + d)$?
  Answer: at most one
- Therefore, the number of pairwise comparisons that must be made between points in different subsets is thus at most one.
- We can certainly find the points in the intervals $(mid - d, mid]$ and $[mid, mid + d)$ in linear time $O(n)$.

Finding the closest pair of points in one-dimension, 5

```
ClosestPair(S)
if |S| = 2, then
   d = |S[2] - S[1]|
else
   if |S| = 1
      d = infty
   else
      m = median(S)
      construct S1 and S2
      d1 = ClosestPair(S1)
      d2 = ClosestPair(S2)
      p = max(S1)
      q = min(S2)
      d = min(d1, d2, q-p)
   end if
end if
return d
```

Algorithm 2 can be extended to the 2-d case, see section 33.4 of CLRS.

Greedy Algorithms

Greedy algorithms

- Algorithms for solving (optimization) problems typically go through a sequence of steps, with a set of choices at each step.

- A greedy algorithm always makes the choice that looks best at the moment, without regard for future consequence
  "*take what you can get now*" strategy

- Greedy algorithms do not always yield optimal solutions, but for many problems they do.

An activity-selection problem

Problem:

Input: *Set* $S = \{1, 2, \ldots, n\}$ *of* n *activities*
$s_i =$ *start time of activity* i
$f_i =$ *finish time of activity* i

Output: *Maximum size subset* $A \subseteq S$ *of* *compatible* *activities*

Notes:

- activities i and j are compatible if the interval $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.
- Assume (without loss of generality):

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

An Activity-selection problem

Greedy algorithm:

- *pick the compatible activity with the earliest finish time.*

Why?

- Intuitively, this choice leaves as much opportunity as possible for the remaining activities to be scheduled
- That is, the greedy choice is the one that maximizes the amount of unscheduled time remaining.

An Activity-selection problem

Pseudocode:

```
GreedyActivitySelector(s,f)
n = length(s)
A = {1}
j = 1
for i = 2 to n
    if s[i] >= f[j]
        A = A U {i}
        j = i
    end if
end for
return A
```

Time complexity: after the array f[1...n] already sorted, the algorithm costs $O(n)$.

An Activity-selection problem

Why Greedy-Activity-Selector works?
The proof of the greedy algorithm producing the solution of maximum size of compatible activities is based on the following two key properties:

- **The greedy-choice property**
  a globally optimal solution can be arrived at by making a locally optimal (greedy) choice.

- **The optimal substructure property**
  an optimal solution to the problem contains within it optimal solution to subprograms.

Specifically, for the Greedy-Activity-Selectior, ...

An Activity-selection problem

The greedy-choice property:

There exists an optimal solution A such that the greedy choice "1" in A.

The proof goes as follows:

- let's order the activities in A by finish time such that the first activity in A is "k_1".
- If $k_1 = 1$, then A begins with a greedy choice
- If $k_1 \neq 1$, then let $A' = (A - \{k_1\}) \cup \{1\}$.
  Then
  1. the sets $A - \{k_1\}$ and $\{1\}$ are disjoint
  2. the activities in A' are compatible
  3. A' is also optimal, since $|A'| = |A|$
- Therefore, we conclude that there always exists an optimal solution that begins with a greedy choice.

An Activity-selection problem

The optimal substructure property:

If A is an optimal solution, then $A' = A - \{1\}$ is an optimal solution to $S' = \{i \in S, s[i] \geq f[1]\}$.

Proof:

By contradiction. If there exists B' to S' such that $|B'| > |A'|$, then let

$$B = B' \cup \{1\},$$

we have

$$|B| > |A|,$$

which is contradicting to the optimality of A.

An Activity-selection problem

In summary, *the greedy activity selector works!*

- After each greedy choice is made, we are left with an optimization problem of the same form as the original.
- *By induction* on the number of choices made, making the greedy choice at every step procedeuces an optimal solution.

Huffman coding

Huffman codes:

- Data compression, typically saving 20%-90%
- Basic idea:
  represent often encountered characters by shorter (binary) codes

Huffman coding

Example:

- Suppose we have the following data file with total 100K characters:

Char.	a	b	c	d	e	f
Freq.	45K	13K	12K	16K	9K	5K
3-bit fixed length code	000	001	010	011	100	101
variable length code	0	101	100	111	1101	1100

- Total number of bits required to encode the file:
  - Fixed-length code:
$$100\text{K} \times 3 = 300\text{K}$$
  - Var.-length code:
$$1{\cdot}45\text{K} + 3{\cdot}13\text{K} + 3{\cdot}12\text{K} + 3{\cdot}16\text{K} + 4{\cdot}9\text{K} + 4{\cdot}5\text{K} = 225\text{K}$$
- Var.-length code saves 25%.

Huffman coding

Prefix codes:

- Prefix codes: no codeword is also a prefix of some other code.
- Representation of prefix code:
  - full binary tree (every nonleaf node has two children)
  - All legal codes are at the leaves, since no prefix is shared
- Encoding and decoding with a prefix code:
  Codewords:

Char.	a	b	c	d	e	f
Code	0	101	100	111	1101	1100

Encode:

- beef $\longrightarrow$ 101110111011100
- face $\longrightarrow$ 110001001101

Decode:

- 101110111011100 $\longrightarrow$ beef
- 110001001101 $\longrightarrow$ face

Huffman coding

Priority queue – *review*

- A priority queue is a data structure for maintaining a set S of elements, each with an associated key.
- A min-priority queue supports the following operations:
  - Insert(S,x): inserts the element x into the set S, i.e., $S = S \cup \{x\}$.
  - Minimum(S): returns the element of S with the smallest "key".
  - ExtractMin(S): removes and returns the element of S with the smallest "key".
  - DecreaseKey(S,x,k): decreases the value of element x's key to the new value k, which is assumed to be at least as small as x's current key value.
- A max-priority queue supports the operations: Insert, Maximum, ExtractMax, IncreaseKey.

Note: use a heap to implement a prority queue is described in section 6.5 of CLRS 3rd ed.

Huffman coding

Constructing a Huffman code:
Let C = alphabet (set of characters)

Basic idea of Huffman coding:

1. Builds a full binary tree T in a *bottom-up* manner
2. Begins with $|C|$ leaves, performs a sequence of $|C| - 1$ "merging" operations to create T
3. "Merging" operation is *greedy:* the two with lowest frequencies are merged.

Pseudocode

```
Huffmancode(C)
//
// Produces a prefix code for alphabet C
//
n = |C|
Q = C    // min-priority queue, keyed by freq attribute
for i = 1 to n-1
    allocate a new node z
    z_left = x = ExtractMin(Q)
    z_right = y = ExtractMin(Q)
    freq[z] = freq[x] + freq[y]
    Insert(Q,z)
endfor
return ExtractMin(Q)  // the root of the tree
```

Huffman coding

Optimality

- Given a binary tree $T =$ code, for each $c \in C$, define

$$\begin{array}{rcl} f(c) & = & \text{frequency of } c \text{ in the file} \\ d_T(c) & = & \text{depth of } c\text{' leave in the tree } T \\ & = & \text{length of the code for } c \\ & = & \text{number of bits} \end{array}$$

  Then the number of bits ("cost of the tree T") required to encode the file

$$B(T) = \sum_{c \in C} f(c) d_T(c),$$

- A codework is optimal if $B(T)$ is minimal.

Huffman coding

Optimality, cont'd

To prove the greedy algorithm Huffmancode producing an optimal prefix code, we show that it exhibits the following two ingradients:

1. **The greedy-choice property**

   If $x, y \in C$ and $f(x) = f(y)$, then there exists an optimal code T such that

   - $d_T(x) = d_T(y)$
   - the codes for x and y differ only in the last bit

2. **The optimal substructure property**

   If $x, y \in C$ having the lowest frequencies, and let z be their parent. Then the tree

   $$T' = T - \{x, y\}$$

   represents an optimal prefix code for the alphabet

   $$C' = (C - \{x, y\}) \cup \{z\}.$$

The proofs are on pages 433-435 of CLRS 3rd Ed.

Huffman coding

Optimality, cont'd
By the above two properties, after each greedy choice is made, we are left with an optimization problem of the same form as the original. By induction, we have

Theorem. Huffman code is an optimal prefix code.

0-1 Knapsack problem[1]

- Given n items $\{1, 2, \ldots, n\}$
- Item i is worth v_i, and weight w_i
- Find a most valuable subset of items with total weight $\leq W$

0-1 knapsack problem can be expressed as

Find a subset $\mathcal{S} \subseteq \{1, 2, \ldots, n\}$ such that

$$\begin{array}{rl} \textit{maximize} & \displaystyle\sum_{i \in \mathcal{S}} v_i \\ \textit{subject to} & \displaystyle\sum_{i \in \mathcal{S}} w_i \leq W \end{array}$$

[1] Have to either take an item or not take it – cann't take part of it.

0-1 Knapsack problem

Three greedy solution strategies:

1. Greedy by highest value v_i
2. Greedy by least weight w_i
3. Greedy by largest value density $\frac{v_i}{w_i}$

All three appraches generate feasible solutions. However, we cannot guarantee that any of them will always generate an optimal solution!

0-1 Knapsack problem

Example

i	v_i	w_i	v_i/w_i
1	6	1	6
2	10	2	5
3	12	3	4

Total weight $W = 5$

Greedy by value density v_i/w_i:

- take items 1 and 2.
- value $= 16$, weight $= 3$
- Leftover capacity $= 2$

Optimal solution

- take items 2 and 3.
- value $= 22$, weight $= 5$
- no leftover capacity

Question: how about greedy by highest value? by least weight?

Dynamic Programming

Dynamic Programming

- Not a specific algorithm, but a technique (like Divide-and-Conquer and Greedy algorithms)
- Developed back in the day when "*programming*" meant "*tabular method*" (like linear programming)
- Used for optimization problems
  - Find a solution with the optimal value
  - Minimization or maximization
- Dynamic Programming is four-step (two-phase) method:
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution in a bottom-up fashion
  4. Construct an optimal solution from computed information

Rod cutting problem

- Problem statement: How to cut a rod into pieces in order to maximize the revenue you can get?
- Input: 1). A rod of length n
  2). an array of prices p_i for a rod of length i, $i = 1, \dots, n$
- Output: The maximum revenue r_n obtainable for rods whose length sum to n

Rod cutting problem

Example

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30
r_i	1	5	8	10	13	17	18	22	25	30

- r_i: maximum revenue of a rod of length i

Rod cutting problem

Brute force:

can cut up a rod of length n in 2^{n-1} different ways

Cost: $\Theta(2^{n-1})$

Rod cutting problem

Dynamic Programming – Phase I:

- Determine the optimal revenue r_n:

$$r_n = \max\{p_n,\ r_1 + r_{n-1},\ r_2 + r_{n-2},\ \ldots,\ r_{n-1} + r_1\}$$

- Alternatively, a simpler way is to observe that every optimal solution r_n has a leftmost cut:

$$\begin{aligned} r_n &= \max\{p_1 + r_{n-1},\ p_2 + r_{n-2},\ \ldots,\ p_{n-1} + r_1,\ p_n\} \\ &= \max_{1 \le i \le n} \{p_i + r_{n-i}\} && (1) \\ &= p_{i_*} + r_{n-i_*} && (2) \end{aligned}$$

Rod cutting problem

Dynamic Programming – Phase II:

How to compute r_n by the expression (1) ?

1. Recursive solution: top-down, no memoization
   Cost: $T(n) = 1 + \sum_{j=0}^{n-1} T(j) = \Theta(2^n)$ for $n > 1$

2. Iterative solution, bottom-up, memoization
   Pseudocode – see next page
   Cost: $T(n) = \Theta(n^2)$

Pseudocode

```
cut-rod(p,n)
// an iterative (bottom-up) procedure for finding ``r'' and
// the optimal size of the first piece to cut off ``s''
Let r[0...n] and s[0...n] be new arrays
r[0] = 0
for j = 1 to n
    // find q = max{p[i]+r[j-i]} for 1 <= i <= j
    q = -infty
    for i = 1 to j
        if q < p[i] + r[j-i]
           q = p[i] + r[j-i]
           s[j] = i
        end if
    end for
    r[j] = q
end for
return r and s
```

Rod cutting problem

Example

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30
r_i	1	5	8	10	13	17	18	22	25	30
s_i	1	2	3	2	2	6	1	2	3	10

- r_i: maximum revenue of a rod of length i
- s_i: optimal size of the first piece to cut
  Note: $s_i = i_*$ in expression (2).

Matrix-chain multiplication

Problem:

Input: *A sequence (chain) of* $(A_1, A_2, \ldots, A_n)$ *of matrices, where* A_i *is of order* $p_{i-1} \times p_i$.

Output: *full parenthesization (ordering) for the product* $A_1 \times A_2 \times \cdots \times A_n$ *that minimizes the number of (scalar) multiplications.*

Matrix-chain multiplication

- Counting the total number of orderings
  1. Define
     $P(n)$ = the number of orderings for a chain of n matrices
  2. Then for $n \geq 2$,
     $$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$
     and $P(1) = 1$
  3. It can be shown that $P(n) = \Omega(2^n)$
- Brute-force solution:

  exhaustive search for determining the optimal ordering is infeasible!

Matrix-chain multiplication

DP – Step 1: *characterize the structure of an optimal ordering*

- An optimal ordering of the product $A_1 A_2 \cdots A_n$ splits the product between A_k and A_{k+1} for some k, and we first compute $A_1 \cdots A_k$ and $A_{k+1} \cdots A_n$, and then multiply them together.
- Key observation: the ordering of $A_1 \cdots A_k$ within this ("global") optimal ordering must be an optimal ordering of (sub-product) $A_1 \cdots A_k$.
  Why? simply argue by contradiction:

  > *If there were a less costly way to order the product $A_1 \cdots A_k$, substituting that ordering within this (global) optimal ordering would produce another ordering of $A_1 A_2 \cdots A_n$, whose cost would be less than the optimum, a contradiction!*

- Similar observation holds for $A_{k+1} \cdots A_n$
- Thus, an optimal ("global") solution to the matrix-chain product contains within it the optimal ("local") solutions to subproblems.
  $=$ **the optimal substructure property**

Matrix-chain multiplication

DP – Step 2: *recursively define the value of an optimal solution*

- Define

$$m[i,j] = \text{min. number of multip. needed to compute } A_i \cdots A_j.$$

- By the definition,
  $m[1,n] =$ the cheapest way for the product $A_1 A_2 \cdots A_n$.
- $m[i,j]$ can be defined recursively:
  - if $i = j$, $m[i,i] = 0$.
  - if $i < j$, $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$ for some k

Matrix-chain multiplication

DP – Step 2: *recursively define the value of an optimal solution*

- Thus, for $1 \le i < j \le n$,

$$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{if } i < j \end{cases}$$

- In addition, to construct an optimal ordering, we keep track

$$s[i,j] = k_* = \text{the value s.t. } m[i,j] \text{ attains the minimum}$$

Matrix-chain multiplication

DP – Step 3: *compute the value of an optimal solution in a bottom-up approach*

- Compute the optimal costs $m[i, j]$ and orderings $s[i, j]$ in a bottom-up approach.
  pseudocode

- Cost: $T(n) = \Theta(n^3)$ since
  1. compute roughly $n^2/2$ entries of m-table
  2. for each entry of m-table, it finds the minimum of fewer than n expressions.

Pseudocode

```
matrix-chain-order(p)
n = length(p) - 1
create arrays m[1...n,1...n] and s[1...n,1...n]
for i = 1 to n
  m[i,i] = 0
endfor
for d = 2 to n
  for i = 1 to n-d+1
     j = i + d - 1
     m[i,j] = +infty          //compute m[i,j] = min_k{...}
     for k = i to j-1
        q  = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j]
        if q < m[i,j]
           m[i,j] = q
           s[i,j] = k  // track k such that min. is attained.
        endif
     endfor
  endfor
endfor
return m and s
```

Matrix-chain multiplication

DP – Step 4: *construct an optimal solution from computed information*

Example: Let `p = [30 35 15 5 10 20 25]`

`matrix-chain-order(p)` generates the m-array and s-array:

```
m = [ 0 15750 7875 9375 11875 15125 ]   s = [ 0 1 1 3 3 3 ]
    [ 0     0 2625 4375  7125 10500 ]       [ 0 0 2 3 3 3 ]
    [ 0     0    0  750  2500  5375 ]       [ 0 0 0 3 3 3 ]
    [ 0     0    0    0  1000  3500 ]       [ 0 0 0 0 4 5 ]
    [ 0     0    0    0     0  5000 ]       [ 0 0 0 0 0 5 ]
    [ 0     0    0    0     0     0 ]       [ 0 0 0 0 0 0 ]
```

By s-array, an optimal parenthesization/ordering is given by

$$(\, A_1 \,(\, A_2 \, A_3 \,)\,)\,(\,(\, A_4 \, A_5 \,)\, A_6 \,)$$

Longest Common Subsequence

Problem statement:

Input: *Sequences*

$$\begin{aligned} X_m &= \langle x_1, x_2, \ldots, x_m \rangle \\ Y_n &= \langle y_1, \ldots, y_n \rangle \end{aligned}$$

Output: *longest common subsequence (LCS)* *of* X_m *and* Y_n

Longest Common Subsequence

Brute-force solution:

- For every subsequence of X_m, check if it is a subsequence of Y_n.
- Running time: $\Theta(n \cdot 2^m)$
- Intractable!

Longest Common Subsequence

DP-Step 1: *characterize the structure of an optimal solution*

Let $Z_k = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of

$$X_m = \langle x_1, x_2, \ldots, x_m \rangle \quad \text{and} \quad Y_n = \langle y_1, \ldots, y_n \rangle$$

Then

1. If $x_m = y_n$, then
   (a) $z_k = x_m = y_n$
   (b) $Z_{k-1} = \text{LCS}(X_{m-1}, Y_{n-1})$
2. If $x_m \neq y_n$, then
   (a) $z_k \neq x_m \Longrightarrow Z_k = \text{LCS}(X_{m-1}, Y_n)$
   (b) $z_k \neq y_n \Longrightarrow Z_k = \text{LCS}(X_m, Y_{n-1})$

In words, the optimal solution to the (whole) problem contains within it the otpimal solutions to subproblems = **the optimal substructure property**

Sketch of the proof: by contradiction!

Longest Common Subsequence

DP-Step 2: *recursively define the value of an optimal solution*

- Define
$$c[i,j] = \text{length of LCS}(X_i, Y_j)$$

- By the optimal structure property
$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1]+1 & \text{if } x[i] = y[j] \\ \max\{c[i,j-1], c[i-1,j]\} & \text{otherwise} \end{cases}$$

- $c[m,n] = \text{length of LCS}(X_m, Y_n)$

Longest Common Subsequence

DP-Step 3: *compute $c[i,j]$ in a bottom-up approach*

- Compute $c[i,j]$ in a bottom-up approach.
- Create $b[i,j]$ to record the optimal subproblem solution chosen when computing $c[i,j]$
- $c[i,j]$ is the length of LCS(X_i, Y_j)

  $b[i,j]$ shows how to construct the corresponding LCS(X_i, Y_j)
- pseudocode
- Cost:
  Running time: $\Theta(mn)$
  Space: $\Theta(mn)$

Pseudocode

```
LCS-length(X,Y)
set c[i,0] = 0 and c[0,j] = 0
for i = 1 to m // Row-major order to compute c and b arrays
    for j = 1 to n
        if X(i) = Y(j)
           c[i,j] = c[i-1,j-1] + 1
           b[i,j] = 'Diag'      // go to up diagonal
        elseif c[i-1,j] >= c[i,j-1]
           c[i,j] = c[i-1,j]
           b[i,j] = 'Up'      // go up
        else
           c[i,j] = c[i,j-1]
           b[i,j] = 'Left'      // go left
        endif
    endfor
endfor
return c and b
```

Longest Common Subsequence

DP–Step 4: *construct an optimal solution from computed information*

Example: $X_6 = \langle A, B, C, B, D, A, B \rangle$ and $Y_6 = \langle B, D, C, A, B, A \rangle$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

(1) Length of LCS $=$ `c[7,6]` $= 4$
(2) By the b-table ("$\uparrow, \leftarrow, \nwarrow$"), the LCS is $B\,C\,B\,A$

Dynamic Programming – Review/Summary

- Not a specific algorithm, but a technique (like Divide-and-Conquer and Greedy algorithms)
- Four-step (two-phase) method:
  1. Characterize the structure of an optimal solution
  2. Recursively define the value of an optimal solution
  3. Compute the value of an optimal solution in a bottom-up fashion
  4. Construct an optimal solution from computed information

Dynamic Programming – Review/Summary

Elements of DP:

1. **Optimal substructure**
   the optimal solution to the problem contains optimal solutions to subprograms
   Examples: Rod cutting, Matrix-chain product, LCS

2. **Overlapping subproblems**
   There are few subproblems in total, and many recurring instances of each
   (unlike divide-and-conquer, where subproblems are independent)
   Example: for LCS, only mn distinct subproblems

3. **Memoization**
   after computing solutions to subproblems, store in table, subsequent calls do table lookup .
   Example: for LCS, running time $\Theta(mn)$

0-1 Knapsack problem revisit

Problem:

Input: n *items* $\{1, 2, \ldots, n\}$
Item i *is worth* v_i *and weight* w_i
Total weight W

Output: *a subset* $S \subseteq \{1, 2, \ldots, n\}$ *such that*

$$\sum_{i \in S} w_i \leq W \quad \textit{and} \quad \sum_{i \in S} v_i \quad \textit{is maximized}$$

0-1 Knapsack problem revisit

Greedy solution strategy: three possible greedy approaches:

1. Greedy by highest value v_i
2. Greedy by least weight w_i
3. Greedy by largest value density $\frac{v_i}{w_i}$

All three appraches generate feasible solutions. However, cannot guarantee to always generate an optimal solution!

0-1 Knapsack problem revisit

Example:

i	v_i	w_i	v_i/w_i
1	6	1	6
2	10	2	5
3	12	3	4

Total weight $W = 5$

Greedy by value density v_i/w_i:

- take items 1 and 2.
- value $= 16$, weight $= 3$

Optimal solution – *by inspection*

- take items 2 and 3.
- value $= 22$, weight $= 5$

Next: Use the Dyanmic Programming technique to find the optimal solution!

0-1 Knapsack problem revisit

The knapsack problem exhibits the optimal substructure property:

Let i_k be the highest-numberd item in an optimal solution $S = \{i_1, \ldots, i_k\}$, Then

1. $S' = S - \{i_k\}$ *is an optimal solution for*

   weight $W - w_{i_k}$ *and items* $\{i_1, \ldots, i_{k-1}\}$,

2. *the value of the solution* S *is*

   v_{i_k} + *the value of the subproblem solution* S'.

0-1 Knapsack problem revisit

- Define
  $c[i, w]$ = the value of an optimal solution for items $\{1, \ldots, i\}$
  and maximum weight w.

- Then

$$c[i, w] = \begin{cases} 0 & \text{if } i = 0 \text{ or } w = 0 \\ c[i-1, w] & \text{if } i > 0 \text{ and } w_i > w \\ \max\left(v_i + c[i-1, w - w_i], c[i-1, w]\right) & \text{if } i > 0 \text{ and } w_i \leq w \end{cases}$$

- That says when $i > 0$ and $w_i \leq w$, we have two choices:
  - either includes item i, 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 item i, in which case it is a subproblem solution of $i - 1$ items and the same weight.

  The better of these two choices should be made.

0-1 Knapsack problem revisit

- The set of items to take can be deduced from the c-table by starting at $c[n, W]$ and tracing where the optimal values came from.
  - If $c[i, w] = c[i-1, w]$, item i is not part of the solution, and we continue tracing with $c[i-1, w]$.
  - Otherwise item i is part of the solution, and we continue tracing with $c[i-1, w-w_i]$.
- Running time: $\Theta(nW)$:
  - $\Theta(nW)$ to fill in the c table
    $(n+1)(W+1)$ entries each requiring $\Theta(1)$ time
  - $O(n)$ time to trace the solution
    starts in row n and moves up 1 row at each step.

0-1 Knapsack problem revisit

Example:

i	v_i	w_i
1	6	1
2	10	2
3	12	3

Total weight $W = 5$

By dynamic programming,

- we generate the following c-table:

$i\backslash w$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	6	6	6	6	6
2	0	6	10	16	16	16
3	0	6	10	16	18	22

- The items to take: $S = \{3, 2\}$

Graph Algorithms

Notion of graphs

- Graph $G = (V, E)$
  $V = \{v_i\}$: set of vertices
  E = set of edges = a subset of $V \times V = \{(v_i, v_j)\}$
- $|E| = O(|V|^2)$
  dense graph: $|E| \approx |V|^2$
  sparse graph: $|E| \approx |V|$,
- If G is connected, then $|E| \geq |V| - 1$.
- Some variants
  - undirected: edge $(u, v) = (v, u)$
  - directed: (u, v) is edge from u to v.
  - weighted: weight on either edge or vertex
  - multigraph: multiple edges between vertices
- Further reading: Appendix B.4, pp.1168-1172.

Notion of graphs

Representing graph by Adjacency Matrix

- $A = (a_{ij})$ is a $|V| \times |V|$ matrix, where

$$a_{ij} = \begin{cases} 1, & \text{if } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

- If G is undirected, A is symmetric.
- $\Theta(|V|^2)$ storage, too much storage for large graphs, can be very efficient for small graphs.

Notion of graphs

Representing graph by Incidence Matrix

- $B = (b_{ij})$ is a $|V| \times |E|$ matrix, where

$$b_{ij} = \begin{cases} 1, & \text{if edge } j \text{ enters vertex } i \\ -1, & \text{if edge } j \text{ leaves vertex } i \\ 0, & \text{otherwise} \end{cases}$$

Notion of graphs

Representing graph by Adjacency List

- For each vertex v,

$$\text{Adj}[v] = \{ \text{ vertices adjacent to } v \}$$

- Variation: could also keep second list of edges coming into vertex.
- How much storage is needed?
  Answer: $\Theta(|V| + |E|)$ ("sparse representation")
  - Degree of a vertex of a undirected graph $=$ the number of incident edges
  - For a digraph: Out-degree and In-degree
  - For undirected graph:
    $\#$ of items in the adj. list $= \sum_{v \in V} \text{degree}(V) = 2|E|$
  - For digraph:
    $\#$ of items in the adj. list $= \sum_{v \in V} \text{out-degree}(V) = |E|$

Breadth-First Search (BFS)

- An archetype for many important graph algorithms
- Input: Given $G = (V, E)$ and a source vertex s,

  Output: $d[v] =$ distance from s to v for all $v \in V$.
- distance $=$ fewest number of edges $=$ shortest path
- BFS basic idea: discovers all vertices at distance k from the source vertex before discovering any vertices at distance $k + 1$
  or expanding frontier – "greedy" – propagate a wave 1 edge-distance at a time.

Review: queue and stack data structure

- Queues and stacks are dynamic sets in which the elements removed from the set by the delete operation is prescribed.

- The queue implements a First-In-First-Out (FIFO) policy.
  The stack implements a Last-In-First-Out (LIFO) policy.

- Queue supports the following operations:
  Enqueue(Q,v): insert element v into the queue Q
  Dequeue(Q,v): delete element v from the queue Q

- There are several way efficient ways to implement queues and stacks
  In section 10.1 of the textbook, it describes a way how to use a simple array to implement each.

BFS

```
BFS(G,s)
// Breadth-First Search
for each vertex u in V-{s}
   d[u] = +infty
endfor
d[s] = 0
Q = empty  // create FIFO queue
Enqueue(Q, s)
while Q not empty
   u = Dequeue(Q)
   for each v in Adj[u]
       if d[v] = +infty,
          d[v] = d[u] + 1
          Enqueue(Q, v)
       endif
   endfor
endwhile
return d
```

BFS

- Running time: $O(|V| + |E|)$

  $O(|V|)$: because every vertex enqueued at most once

  $O(|E|)$: because every vertex dequeued at most once and we examine (u, v) only when u is dequeued at most once if directed, at most twice if undirected.

  Note: not $\Theta(|V| + |E|)$!

- Correctness of BFS
  shortest path proof – see pp.597-600 (more later).
  similar with weighted edges – Dijkstra's algorithm

Depth-First Search (DFS)

- another archetype for many important graph algorithms
- methodically explore *every* vertex and *every* edge
- Input: Given $G = (V, E)$

  Output: Two timestamps for every $v \in V$

  $d[v] =$ when v is first discovered.

  $f[v] =$ when v is finished.

  and classification of edges

DFS

- DFS idea: *go as far as possible, then "back up"* :
  - edges are explored out of the most recently discovered vertex v that still have unexplored edges leaving
  - when all of v's edges have been explored, the search "backtracks" to explore edges leaving the vertex from which v was discoverd.
- Three-color code for search status of vertices
  - **White** $=$ a vertex is undiscovered
  - **Gray** $=$ a vertex is discovered, but its processing is incomplete
  - **Black** $=$ a vertex is discovered, and its processing is complete

Review: Queue and Stack

- Queues and stacks are dynamic sets in which the elements removed from the set by the delete operation is prescribed.
- The queue implements a First-In-First-Out (FIFO) policy.
  The stack implements a Last-In-First-Out (LIFO) policy.
- Queue supports the following operations:
  Enqueue(Q,v): insert element v into the queue Q
  Dequeue(Q,v): delete element v from the queue Q
- There are several way efficient ways to implement queues and stacks.
  Section 10.1 describes an implementation by using arrays.

DFS

```
DFS(G)    // main routine
for each vertex u in V
     color[u] = ``white''
endfor
time = 0
for each vertex u in V
    if color[u] = ``white''
       DFS-Visit(u)
    endif
endfor
// end of main routine
```

```
DFS-Visit(u)    // subroutine
color[u] = ``gray''
time = time + 1
d[u] = time
for each v in Adj[u]
    if color[v] = ``white''
       DFS-visit(v)
    endif
end for
color[u] = ``black''
time = time + 1
f[u] = time
// end of subroutine
```

DFS

- Vertices, from which exploration is incomplete, are processed in a LIFO stack.
- Running time: $\Theta(|V| + |E|)$

  not big-O since guaranteed to examine every vertex and edge.
- More properties of DFS, see pp.606-608

DFS

Classification of edges

- **T** = Tree edge = encounter new vertex (gray to white)
- **B** = Back edge = from descendant to ancestor (gray to gray)
- **F** = Forward edge = from ancestor to descendant (gray to black)
- **C** = Cross edge = any other edges (between trees and subtrees): (gray to black)

Note: In an undirected graph, there may be some ambiguity since edge (u,v) and (v,u) are the same edge. Classify by the first type that matches.

DFS vs. BFS

1. DFS: vertices from which the exploring is incomplete are processed in a LIFO order (stack)

   BFS: vertices to be explored are organized in a FIFO order (queue)

2. DFS contains two processing opportunities for each vertex v, when it is "discovered" and when it is "finished"

   BFS contains only one processing opportunity for each vertex v, and then it is dequeued

Applications

1. For a undirected graph,
   (a) a DFS produces only Tree and Back edges
   (b) acyclic (tree) iff a DFS yeilds no back edges

2. A directed graph is acyclic iff a DFS yields no back edges

3. Topological sort of a dag (= directed acyclic graph)

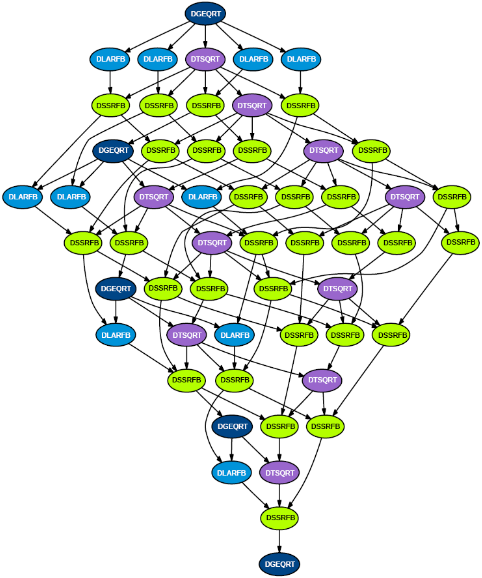
4. Strongly connected components, see Sec.22.5

Topological sort

- A topological sort (TS) of a dag $G = (V, E)$ is a linear ordering of all its vertices such that if $(u, v) \in E$, then u appears before v.
- A TS is not possible if G has a cycle.
- The ordering is not necessarily unique.

Topological sort

An application:



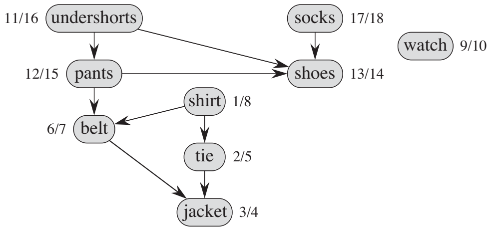
Topological sort

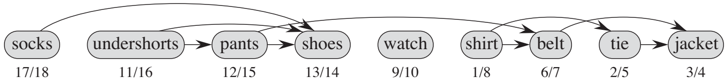
- TS Algorithm
  1. run DFS(G) to compute finishing times $f[v]$ for all $v \in V$
  2. output vertices in order of decreasing times

- Running time: $\Theta(|V| + |E|)$

Topological sort

Example: Getting dressed





Topological sort

Theorem (correctness of the algorithm):
TS(G) produces a toplogical sort of a dag G.

Proof: *Just need to show that if* $(u, v) \in E$*, then* $f[v] < f[u]$.
When we explore edge (u, v), u is gray, what's the color of v?

- Is v gray too?
  no, because then v would be ancestor of u, edge (u, v) is a back edge, a contradiction of a dag.
- Is v white?
  yes, then v is descendant of u, by DFS, $d[u] < d[v] < f[v] < f[u]$
- Is v black?
  yes, then v is already finished. Since we're exploring (u, v), we have not yet finished u, therefore $f[v] < f[u]$

Minimum Spanning Tree (MST)

- Undirected connected graph $G = (V, E)$
- Weight function $w : \quad E \longrightarrow \mathbf{R}$

- Spanning tree: a tree that connects all vertices
- Minimum Spanning Tree (MST) T:

$$w(T) = \sum_{(u,v) \in T} w(u, v) \quad \text{is minimized}$$

MST

Idea of "growing" a MST:

- construct the MST by successively select edges to include in the tree
- guarantee that after the inclusion of each new selected edge, it forms a subset of some MST.

One of the most famous greedy algorithms, along with Huffman coding

MST

Basic properties:

- Optimal substructure: optimal tree contains optimal subtrees.

  Let T be a MST of $G = (V, E)$. Removing (u, v) of T partitions T into two trees T_1 and T_2. Then T_1 is a MST of $G_1 = (V_1, E_1)$ and T_2 is a MST of $G_2 = (V_2, E_2)$.[2]

- Greedy-choice property:

  Let T be a MST of $G = (V, E)$, $A \subseteq T$ be a subtree of T, and (u, v) be min-weight edge in G connecting A and $V - A$. Then $(u, v) \in T$.[3]

Both properties can be proven by using simple contradiction arguments, see sec.23.1

---

[2] The subgraph G_1 is induced by vertices in T_1, i.e., $V_1 = \{\text{vertices in } T_1\}$ and $E_1 = \{(x, y) \in E; x, y \in V_1\}$. Similarly for G_2.

[3] It is an abuse of notation we will view A as being both edges and vertices.

MST

Prim's algorithm

- Basic idea:
  - builds one tree, so that A is always a tree
  - starts from a root r
  - at each step, find the next lightest edge crossing cut $(A, V - A)$ and add this edge to A (*greedy choice*)

- How to find the next lightest edge quickly?

  Answer: use a **priority queue**

Review: Priority queue

A priority queue maintains a set S of elements, each with an associated value called a "key", and supports the following operations:

- Search(S,k):
  returns x in S with key[x] = k
- Insert(S, x)/Delete(S, x):
  inserts/deletes the element x into the set S
- Maximum(S)/Minimum(S):
  returns x in S with largest/smallest key
- Extract-max(S)/Extract-min(S):
  removes and returns x in S with largest/smallest key
- Increase-key(S, x, k)/Decrease-key(S, x, k):
  increases/decreases the value of element x's key to the new value k

The priority queue has been used in Huffman coding.

MST

Prim's algorithm – pseudocode

```
MST-Prim(G, w, r)
Q = empty
for each vertex u in V
    key[u] = infty
    pi[u] = Nil
    Insert(Q, u)
endfor
Decrease-key(Q,r,0)
while Q not empty
    u = Extract-Min(Q)
    for each v in Adj[u]
        if v in Q and w(u,v) < key[v]
            Decrease-key(Q, v, w(u,v))
            pi[v] = u   // parent of v
       endif
    endfor
endwhile
return A = { (v, pi[v]): v in V-{r} }   // MST
```

MST

Prim's algorithm – running time:

- depends on how the priorty queue is implemented
- Suppose Q is a binary heap (see Section 6.1)
  - Initialize Q and the first for loop: $O(|V| \lg |V|)$
  - Decrease key of root r: $O(\lg |V|)$
  - While-loop:
    a) $|V|$ Extract-Min calls: $O(|V| \lg |V|)$
    b) $\leq |E|$ Decrease-Key calls: $O(|E| \lg |V|)$
- Total: $O(|E| \lg |V|)$

Note: G is connected, $\lg |E| = \Theta(\lg |V|)$ *(why?)*

MST

Kruskal's algorithm

- Basic idea:
  - scan edges in increasing of weight
  - put edge in if no loop created
- Why does this result in MST?
  Answer: min-weight edge is always in MST (the greedy-choice property).
- Implementation data structure: disjoint-set

Review: Disjoint-Set data structure

Disjoint-Set maintains a collection of $S = \{S_1, S_2, ...S_k\}$ of disjoint dynamic sets. Each set is identified by a representative, which is some member of the set.

A disjoint-set data structure supports the following operations:

- Make-set(x):
  creates a new set whose only member (and thus representative) is x.
- Union(x, y):
  unites the sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets: $S_x \cup S_y$. The representative is any member of $S_x \cup S_y$.
- Find-set(x):
  returns (a pointer to) the representative of the (unique) set containing x.

To learn more about the disjoint-set data structure, see Chapter 21.

MST

Kruskal's algorithm – pseudocode:

```
MST-Kruskal(G, w)
A = emtpy
for each vertex v in V
    Make-set(v)
endfor
Sort the edges E in nondecreasing order by w
for each edge (u,v) in E
    if Find-set(u) \= Find-set(v)
        A = A U {(u,v)}
        Union(u,v)
    endif
endfor
return A
```

MST

Kruskal's algorithm – running time:

- depends on the implementation of the disjoint-set
- Sort: $\Theta(|E| \lg |E|)$
- $|V|$ Make-Set ops
- $2|E|$ Find-Set ops
- $|V| - 1$ Union ops
- Total: $O(|E| \lg |V|)$

Note: G is connected, $\lg |E| = \Theta(\lg |V|)$

Shortest-path problems

- Generalization of BFS to handle weighted graphs
- Directed graph $G = (V, E)$,
- Weight function $w \ : \ E \longrightarrow \mathbf{R}$
- Weight of path $p = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Shortest-path weight $u \rightsquigarrow v$

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightsquigarrow v\} & \text{if there exists a path } u \rightsquigarrow v \\ \infty & \text{otherwise} \end{cases}$$

- Shortest-path $u \rightsquigarrow v$
  any path p such that $w(p) = \delta(u, v)$

Shortest path problems

Variants:

- Single-source: find shortest-paths from a given source vertex $s \in V$ to every vertex $v \in V$.
- Single-destination: find shortest-paths to a given destination vertex *(reverse the direction of each edge to become the single-source problem.)*
- Single-pair: find shortest-path from u to v. *(no way know that's better in worst case than solving single-source)*
- All-pairs: find shortest-paths from u to v for all $u, v \in V$. *(skip, if interested, see algorithms in Chapter 25).*

Shortest-path problems

Negative-weight edges and well-definedness

- Negative-weight edges are OK, as long as no negative-weight cycles reachable from the source.
  ... otherwise, can always get a shorter path by going around the cycle again.
- The shortest path problem is ill-posed in graph with negative-weight cycle
- Bellman-Ford algorithm can detect and report the existence of negative-weight cycle

Shortest-path problems

- Optimal substructure property:
  subpaths of shortest-paths are shortest-paths.

  Thus, will see greedy and dynamical programming algorithms.

Single-Source Shortest Path (SSSP) algorithms

- Notation: $d[v]$: shortest-path estimate
  $\pi[v]$: predecessor of v
- Output of SSSP algorithms:
  $d[v] = \delta(s, v) =$ shortest-path weight $s \rightsquigarrow v$
  $\pi[v] =$ predecessor of v on a shortest path from s.
- Two key components of shortest-path algorithms
  - Initialization

```
for every vertex v in V
    d[v] = infty
    pi[v] = nil
endfor
d[s] = 0      // s = source vertex
```

  - Relaxing an edge (u, v): can we improve the shortest-path estimate $d[v]$ by going through u and taking the edge (u, v)?

```
if d[v] > d[u] + w(u,v)
    d[v] = d[u] + w(u,v)
    pi[v] = u
endif
```

Shortest-paths properties

1. Triangular inequality
   for all $(u, v) \in E$, $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$
2. Upper-bound property
   Always have $d[v] \geq \delta(s, v)$ for all v.
   Once $d[v] = \delta(s, v)$, it never changes
3. No-path property
   If $\delta(s, v) = \infty$, then $d[v] = \infty$ always
4. Convergence property
   If $s \rightsquigarrow u \rightarrow v$ is a shortest-path, and $d[u] = \delta(s, u)$. Then after "Relax $u \rightarrow v$", $d[v] = \delta(s, v)$
5. Path relaxation property
   Let $p = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_k$ be a shortest-path. If we relax in order, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(v_0, v_k)$

The Bellman-Ford algorithm

- Most basic shortest-paths algorithm for the shortest-path problem
- Allow negative-weight edges
- Compute $d[v]$ and $\pi[v]$ for all $v \in V$
  - $d[v] = \delta(s, v)$: the shortest-path weight from the source s to v.
  - $\pi[v]$: the parent (predecessor) of v.
- Return TRUE if no negative-weight cycles reachable from source s, FALSE otherwise.

The Bellman-Ford algorithm – *pseudocode*

```
Bellman-Ford(G, w, s)
for each vertex v in V              // initialization
    d[v] = infty
    pi[v] = nil
endfor
d[s] = 0
for i = 1 to |V|-1                  // |V|-1 passes
    for each edge (u,v) in E        // in a prescribed order
        if d[v] > d[u] + w(u,v)     // relax if necessary
            d[v] = d[u] + w(u,v)
            pi[v] = u
        endif
    endfor
endfor
for each edge (u,v) in E            // final check pass
    if d[v] > d[u] + w(u,v)
       return FALSE
    endif
endfor
return TRUE, d, pi
```

The Bellman-Ford algorithm

- Running time: $\Theta(|V| \cdot |E|)$.
- Values you get on each pass and how quickly it converges depends on order of relaxation (processing edges). But guaranteed to converge after $|V| - 1$ passes, assuming no negative-weight cycles.

Dijkstra's algorithm

- No negative weight edges
- Like BFS. If all weights $= 1$, use BFS.
- Use $Q =$ priority queue keyed by $d[v]$
  (BFS uses FIFO queue)
- Have two sets of vertices:
  - $S =$ vertices whose final shortest-path weights are determined
  - $Q =$ priority queue $= V - S$

Dijkstra's algorithm – *pseudocode*

```
Dijkstra(G, w, s)
for each vertex v in V              // Initialization
    d[v] = infty
    pi[v] = nil
endfor
d[s] = 0
S = empty
Q = V                               // priority queue keyed by d[v]
while Q is not empty
    u = Extract-Min(Q)
    S = S U {u}
    for each vertex v in Adj[u]
        if d[v] > d[u] + w(u,v)     // Relax if necessary
            d[v] = d[u] + w(u,v)
            pi[v] = u
        endif
    endfor
endwhile
return d, pi
```

Dijkstra's algorithm

- Running time: $O(|E| \lg |V|)$ (binary heap)
- Similar to the BFS and MST-algorithms, Dijkstra's algorithm is a greedy algorithm. It always chooses the "lightest" or "closest" vertex in $V - S$ to insert into S

The SSSP in DAG

- DAG: can have negative-weight edges, but no negative-weight cycle.
- How fast can do it?

  Answer: $O(|V|+|E|)$, instead of $\Theta(|V|\cdot|E|)$ by Bellman-Ford

The SSSP in DAG – *pseudocode*

```
DAG-Shortest-Path(G, w, s)
Topological sort of the vertices of G
for each vertex v in V
    d[v] = infty
    pi[v] = nil
endfor
d[s] = 0
for each vertex u taken in topologically sorted order
   for each vertex v in Adj[u]
       if d[v] > d[u] + w(u,v)
            d[v] = d[u] + w(u,v)
            pi[v] = u
       endif
   endfor
endfor
return d, pi
```

Shortest-path problems – Proofs

- Weight of path $p = v_0 \to v_1 \to \cdots \to v_k$:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

- Shortest-path weight $u \leadsto v$

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\leadsto} v\} & \text{if there exists a path } u \leadsto v \\ \infty & \text{otherwise} \end{cases}$$

- Shortest-path $u \leadsto v$
  any path p such that $w(p) = \delta(u,v)$

Shortest-path problems – Proofs

Triangular inequality property:
for all $(u, v) \in E$, $\delta(u, v) \leq \delta(u, x) + \delta(x, v)$.

Proof: Note that

$$\underline{\text{Weight of shortest path } s \rightsquigarrow v} \leq \underline{\text{weight of any path } s \rightsquigarrow v}$$

The path $s \rightsquigarrow u \rightarrow v$ is a path $s \rightsquigarrow v$, and if we use a shortest path $s \rightsquigarrow u$, its weight is $\delta(u, x) + \delta(x, v)$.

Shortest-path problems – Proofs

Upper-bound property:

Always have $d[v] \geq \delta(s,v)$ for all v.
Once $d[v] = \delta(s,v)$, it never changes.

Proof. Initally true.
Suppose there exists a vertex such that $d[v] < \delta(s,v)$. *Without loss of generality,* v *is first vertex for which this happens. Let* u *be the vertex that causes* $d[v]$ *change. Then* $d[v] = d[u] + w(u,v)$. *So*

$$\begin{aligned} d[v] &< \delta(s,v) \\ &\leq \delta(s,u) + w(u,v) \\ &\leq d[u] + w(u,v) \end{aligned}$$

which implies $d[v] < d[u] + w(u,v)$,
Contradicts $d[v] = d[u] + w(u,v)$.
Once $d[v]$ *reaches* $\delta(s,v)$, *it never goes lower. It never goes up, since relaxations only lower shortest-path weights.*

Shortest-path problems – Proofs

No-path property:

If $\delta(s, v) = \infty$, then $d[v] = \infty$ always.

Proof. $d[v] \geq \delta(s, v) = \infty \Rightarrow d[v] = \infty$.

Shortest-path problems – Proofs

Convergence property:

If $s \rightsquigarrow u \to v$ is a shortest-path, and $d[u] = \delta(s, u)$. Then after "Relax $u \to v$", $d[v] = \delta(s, v)$.

Proof. After relaxation

$$\begin{aligned} d[v] &\le d[u] + w(u, v) \\ &= \delta(s, u) + w(u, v) \\ &= \delta(s, v) \end{aligned}$$

On the other hand, we have $d[v] \ge \delta(s, v)$. Therefore, it must have $d[v] = \delta(s, v)$.

Shortest-path problems – Proofs

Path relaxation property

Let $p = v_0 \to v_1 \to \cdots \to v_k$ be a shortest-path. If we relax in order, $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, even intermixed with other relaxations, then $d[v_k] = \delta(v_0, v_k)$.

Proof. Induction to show $d[v_i] = \delta(s, v_i)$ *after* (v_{i-1}, v_i) *is relaxed.*

- *Basis step:* $i = 0$. *Initially* $d[v_0] = \delta(s, v_0) = \delta(s, s)$
- *Inductive step: Assume* $d[v_{i-1}] = \delta(s, v_{i-1})$. *Relax* (v_{i-1}, v_i). *By convergence property,* $d[v_i] = \delta(s, v_i)$ *afterward and* $d[v_i]$ *never changes.*

Shortest-path problems – Proofs

Correctness of the Bellman-Ford algorithm: It is guaranteed to converge after $|V| - 1$ passes, assuming no negative-weight cycles.

Proof. *Use path-relaxation property.*
Let v *be reachable from* s*, and let* $p = v_0 \to v_1 \to \cdots \to v_k$ *be the shortest path from* s *to* v*, where* $v_0 = s$ *and* $v_k = v$*.*
Since p *is acyclic, it has* $\leq |V| - 1$ *edges, so that* $k \leq |V| - 1$ *edges.*
Each iteration of the **for** *loop realxes all edges:*

- *First iteration relaxes* (v_0, v_1)
- *Second iteration relaxes* (v_1, v_2)
- *...*
- k*th iteration relaxes* (v_{k-1}, v_k)

By the path-relaxation property, $d[v] = d[v_k] = \delta(s, v_k) = \delta(s, v)$*.*

Shortest-path problems – Proofs

Correctness of Dijkstra's algorithm: Show that $d[u] = \delta(s, u)$ when u is added to S in each iteration.

- We prove by contradiction. Suppose there exists u such that $d[u] \neq \delta(s, u)$. Without loss of generality, let u be the first vertex for which $d[u] \neq \delta(s, u)$ when u is added to S in each iteration.
- Observation:
  - $u \neq s$, since $d[s] = \delta(s, s) = 0$.
  - Therefore, $s \in S$ and $S \neq \emptyset$
  - There must have be some path $s \rightsquigarrow u$, since otherwise $d[u] = \delta(s, u) = \infty$ by no-path property.

  So, there is a path $s \rightsquigarrow u$. Then there is a shortest path $s \overset{p}{\rightsquigarrow} u$.
- Just before u is added to S, path p connects a vertex in S (i.e., s) to a vertex in $V - S$ (i.e., u). Let y be first vertex along p that's in $V - S$ and and let x be y's predecessor.
- Decompose p into

$$s \overset{p_1}{\rightsquigarrow} x \rightarrow y \overset{p_2}{\rightsquigarrow} u$$

  (could have $x = s$ or $y = u$, so that p_1 or p_2 may have no edges.)

Shortest-path problems – Proofs

Correctness of Dijkstra's algorithm, cont'd

- Claim:[4] $d[y] = \delta(s, y)$ when u is added to S.
- Now we can get a contradiction to $d[u] \neq \delta(s, u)$:

$$y \text{ is on shortest path } s \leadsto u, \text{ and all edge weights are nonnegative}$$
$$\Downarrow$$
$$\delta(s, y) \leq \delta(s, u)$$
$$\Downarrow$$
$$d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u] \text{ (upper bound property)}$$

  Also, both y and u were in Q when we chose u, so that

$$d[u] \leq d[y]$$

  Therefore, $d[y] = \delta(s, y) = \delta(s, u) = d[u]$. Contradicts assumption that $d[u] \neq \delta(s, u)$.

- Hence, Dijkstra's algorithm is correct.

---

[4] Proof. $x \in S$ and u is the first vertex such that $d[u] = \delta(s, u)$ when u is added to S $\Rightarrow d[x] = \delta(s, x)$ when x is added to S. Relaxed (x, y) at that time, so by the convergence property, $d[y] = \delta(s, y)$.

NP-completeness

NP-Completeness – Outline

I. Introduction

II. P and NP

III. NP-complete (NPC): formal definition

IV. How to prove a problem is NPC

V. How to solve a NPC problem: approximate algorithms

I. Introduction

Outline

1. Tractable and intractable problems
2. NP-complete problems – informal definition
3. P vs NP
4. Optimization problems and decision problems

I. Introduction

Tractable and intractable problems

- Problems that are solvable by polynomial-time algorithms are tractable

- Problems that require superpolynomial time are intractable.

Almost all the algorithms we have studied thus far have been polynomial-time algorithms on inputs of size n, their worst-case running time is $O(n^k)$ for some constant k.

I. Introduction

NP-complete (NPC) problems: informal definition

A class of very diverse problems share the following properties:

1. We *only know* how to solve those problems in time much larger than polynomial, namely exponential time.

2. If we could *solve one NPC porblem* in polynomial time, then there is a way to *solve every NPC problem* in polynomial time.

I. Introduction

Reasons to study NPC porblems: practical

- you can use a known algorithm for it, and accept that it will take a long long time to solve;
- you can settle for approximating the solution, e.g., finding a nearly best solution rather than the optimum; or
- you can change your problem formulation so that it is solvable in polynomial time.

I. Introduction

Reasons to study NPC porblems: theoretical

- We stated above that "*We only know*" how to solve those problems in time much larger than polynomial, *Not that we have proven* that these problems require exponential time.
- Indeed, this is one of the most famous problems in computer science:

$$\mathrm{P} = \mathrm{NP}\ ?$$

  or

  Whether NPC problems have polynomial solutions?

- First posed in 1971
  http://www.claymath.org/millennium-problems

I. Introduction

P-vs-NP Example 1.

- Shortest path:
  finding the shortest path from a single source in a directed graph.
- Longest path:
  finding the longest *simple* path between two vertices in a directed graph.

The first one is solvable in polynomial time, and the second is NPC, but the difference appears to be slight.

I. Introduction

P-vs-NP Example 2.

- Euler tour:
  given a connected, directed graph G, is there a cycle that visits each edge exactly once (although it is allowed to visit each vertex more than once)?
- Hamiltonian cycle:
  given a connected directed graph G, is there a simple cycle that visits each vertex exactly once?

The first one is solvable in polynomial time, and the second is NPC, but the difference appears to be slight

I. Introduction

P-vs-NP Example 3.

- Minimum spanning tree (MST):
  Given a weighted graph and an integer k, is there a spanning tree whose total weight is k or less?

- Traveling salesperson problem (TSP):
  given a weighted graph and an integer k, is there a cycle that visits all vertices exactly once whose total weight is k or less?

The first one is solvable in polynomial time, and the second is NPC, but the difference appears to be slight

I. Introduction

P-vs-NP Example 4.

- Circuit value:
  given a Boolean formula and its input, is the output True?
- Circuit satisfiability (SAT):
  given a Boolean formula, is there a way to set the inputs so that the output is True?

The first one is solvable in polynomial time, and the second is NPC, but the difference appears to be slight.

I. Introduction

Optimization problems and Decision problems

- Most of problems occur naturally as optimization problems,
- but they can also be formulated as decision problems, that is, problems for which the output is a simple *Yes* or *No* answer for each input.

Remarks:

- *To simplify discussion, we can consider only decision problems, rather than optimization problems.*
- The optimization problems are at least as hard to solve as the related decision problems, we have not lost anything essential by doing so.

I. Introduction

Optimization-vs-Decision Example 1.

Graph coloring: A coloring of a graph $G = (V, E)$ is a mapping

$$C : V \to S$$

where S is a finite set of "colors", such that

$$(u, v) \in E \Rightarrow C(u) \neq C(v)$$

- optimization problem: given G, determine the smallest number of colors needed.
- decision problem: given G and a positive integer k, is there a coloring of G using at most k colors?

I. Introduction

Optimization-vs-Decision Example 2.

Hamiltonian cycle: A Hamiltonian cycle is cycle that passes through every vertex exactly once.

- decision problem: Does a given graph have a Hamiltonian cycle?
- optimization problem: Give a list of vertices of a Hamiltonian cycle.

I. Introduction

Optimization-vs-Decision Example 3.

TSP (Traveling Salesperson Problem): given a weighted graph and an integer k, is there a cycle that visits all vertices exactly once (Hamiltonian cycle) whose total weight is k or less?

- optimization problem: given a weighted graph, find a minimum Hamiltonian cycle.
- decision problem: given a weighted graph and an integer k, is there a Hamiltonian cycle with total weight at most k?

I. Introduction – recap

1. Tractable and intractable problems
   polynomial-boundness: $O(n^k)$
2. NP-complete problems – informal definition
3. P vs NP
   difference may appear "*only slightly*"
4. Optimization problems and decision problems

II. P and NP

- An algorithm is said to *polynomial bounded* if its worst-case complexity $T(n)$ is bounded by a polynomial function of the input size n, i.e. $T(n) = O(n^k)$.

  Examples: Algorithms for Shortest path, MST, Euler tour, Circuit value.

- **P** is the class of decision problems that can be *solved* in polynomial time, i.e., they are polynomial bounded

- **NP** is the class of decision problems that are *verifiable* in polynomial time.[5]

  i.e., if we were given a "certificate" (= a solution), then we could verify that whether the certificate is correct in polynomial time.

  Examples: Certificates for Circuit-SAT, longest path, Hamiltonian cycle, graph coloring.

[5] The name "NP" stands for "Nondeterministic Polynomial time".

II. P and NP

- $\mathsf{P} \subseteq \mathsf{NP}$, since if a problem is in P then we can solve it in polynomial time without even being given a certificate.
- Open question: Does $\mathsf{P} \subset \mathsf{NP}$ or $\mathsf{P} = \mathsf{NP}$

II. P and NP

- The size of the input can change the classification of P or NP.
- Examples:
  - Prime-testing problem:

$$O(n) \quad \xrightarrow{n=10^m} \quad O(10^m)$$

  - Knapsack problem

$$O(nW) \quad \xrightarrow{W=10^m} \quad O(n \cdot 10^m)$$

- Knowing the effect on complexity of the size of the input is important.
- Unfortunately, even with strong restrictions on the inputs, many NPC problems are still NPC.

  Example: 3-Conjuntive Normal Form Satisfiability problem

III. NP-complete

- NP-complete (NPC) is the term used to describe decision problems that are the hardest ones in **NP** in the following sense

  > *If there were a polynomial-bounded algorithm for an NPC problem, then there would be a polynomial-bounded time for each problem in NP.*

III. NP-complete

Formal definition:

- A decision problem A is **NP-complete (NPC)** if

  (1) $A \in$ NP and

  (2) every other problems B in NP is *polynomially reducible* to A, denoted as

$$B \leq_T A$$

III. NP-complete

Polynomial reduction $B \leq_T A$

- Let A and B be two decision problems, B is polynomially reducible to A, if there is a poly-time computable transformation T such that

$$\text{Yes-instance of } A \quad \overset{\text{iff}}{\Longleftrightarrow} \quad \text{Yes-instance of } B$$

III. NP-complete

- Cook's theorem (1971): Circuit-SAT is NPC.

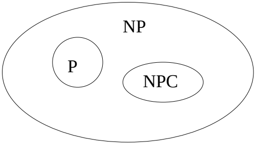
  First result deomonstrating that a specific problem is NPC.

- Known NPC problems:
  - Graph coloring
  - Hamiltonian cycle
  - TSP
  - Knapsack
  - Subset sum
  - ......
  - ......

III. NP-complete

P, NP and NPC:

- How most theoretical computer scientists **view** the relationships among P, NP and NPC:
  - Both P and NPC are wholely contained within NP
  - $\mathrm{P} \bigcap \mathrm{NPC} = \emptyset$



- If a problem satisfies the property (2), but not necessarily the property (1), we say the problem is NP-hard.
- "NP-hard" does not mean "in NP and hard". It means "at least as hard as any problem in NP". Thus a problem can be NP-hard and not be in NP.

I–III recap

1. Tractable and intractable problems
2. Optimization problems and decision problems
3. P and NP
4. NP-complete: formal definition
5. Polynomial reduction

IV. How to prove a problem is NP-complete

- The reducibility relation is *transitive*.
- To prove that a problem $A \in$ NP is NPC, it suffices to prove that some other NPC problem B is *polynomially reducible* to A:

  Step 1: choose some known NPC problem B
  Step 2: define a polynomial transformation T *from* B *to* A
  Step 3: show that $B \leq_T A$

- Why? the logic is as follows:

  Since B *is NPC, all problems in NP is reducible to* B*.*
  Show B *is reducible to* A*.*
  Then all problems in NP is reducible to A*.*
  Therefore, A *is NPC*

IV. How to prove a problem is NP-complete

Examples:

1. Directed HC $\leq_T$ Undirected HC (Next)
2. Subset-Sum $\leq_T$ Job Scheduling (Lecture Notes)
3. Graph 3-COLOR $\leq_T$ 4-COLOR (Homework #8)
4. Subset Sum $\leq_T$ Set Partition (Homework #8)

$$\left\{ \begin{array}{l} \text{Directed HC} \\ \text{Subset-Sum} \\ \text{Graph 3-COLOR} \\ \text{Subset Sum} \end{array} \right\} \text{are NPC} \Longrightarrow \left\{ \begin{array}{l} \text{Undirected HC} \\ \text{Job Scheduling} \\ \text{4-COLOR} \\ \text{Set Partition} \end{array} \right\} \text{are NPC.}$$

IV. How to prove a problem is NP-complete

Example:

- The directed HC is known to be NPC.
- Show that

$$\text{directed HC} \quad \leq_T \quad \text{undirected HC}$$

- Therefore we conclude that the undirected HC is also NPC.

IV. How to prove a problem is NP-complete

Example, cont'd:

- Define transformation:

  Let $G = (V, E)$ be a directed graph. Define G to the undirected graph $G' = (V', E')$ by the following transformation T:

  - $\underline{v \in V} \quad \longrightarrow \quad \underline{v^1, v^2, v^3 \in V' \text{ and } (v^1, v^2), (v^2, v^3) \in E'}$
  - $\underline{(u, v) \in E} \quad \longrightarrow \quad \underline{(u^3, v^1) \in E'}$

- T is polynomial-time computable.

- Show that

$$G \text{ has a HC} \iff G' \text{ has a HC}.$$

IV. How to prove a problem is NP-complete

Example, cont'd:

"$\Rightarrow$" Suppose that G has a directed HC: $v_1, v_2, \ldots, v_n, v_1$
Then

$$v_1^1, v_1^2, v_1^3, v_2^1, v_2^2, v_2^3, \ldots, v_n^1, v_n^2, v_n^3, v_1^1$$

is an undirected HC for G'.

"$\Leftarrow$"

1. Suppose that G' has an undirected HC, the three vertices v^1, v^2, v^3 that correspond to one vertex from G must be traversed **consecutively** in the order v^1, v^2, v^3 or v^3, v^2, v^1, since v^2 **cannot** be reached from any other vertex in G'.
2. Since the other edges in G' connect vertices with superscripts 1 or 3, if for any one triple the order of the superscripts is 1, 2, 3, then the order is 1, 2, 3 for all triples. Otherwise, it is 3, 2, 1 for all triples.
3. Therefore, we may assume that the undirected HC of G' is

$$\underline{v_{i_1}^1, v_{i_1}^2, v_{i_1}^3}, \underline{v_{i_2}^1, v_{i_2}^2, v_{i_2}^3}, \ldots, \underline{v_{i_n}^1, v_{i_n}^2, v_{i_n}^3}, \underline{v_{i_1}^1}.$$

Then

$v_{i_1}, v_{i_2}, \ldots, v_{i_n}, v_{i_1}$ is a directed HC for G.

IV. How to prove a NP-complete problem

Hint for showing

$$\text{Subset-Sum} \le_T \text{Set-Partition}$$

Let S be an instance of Subset-Sum with $w = \sum_{s \in S} s$ and the target c.

Define the set S' (i.e., the transformation T from S to S') as follows:

$$S' = S \cup \{2w - c,\, w + c\},$$

Show that

S is a Yes-instance of Subset-Sum $\Longleftrightarrow$ S' is a Yes-instance of Set-Partition

---

Subset sum decision problem: Given a positive integer c, and the set $S = \{s_1, s_2, \ldots, s_n\}$ of positive integers s_i for $i = 1, 2, \ldots, n$. Assume that $\sum_{i=1}^{n} s_i \ge c$. Is there a $J \subseteq \{1, 2, \ldots, n\}$ such that $\sum_{i \in J} s_i = c$

IV. How to prove a NP-complete problem

Hint, cont'd:

$\Longrightarrow$: Let $J \subseteq S$ and the elements in J sum to c. Therefore, $J \cup \{2w - c\}$ sum to $2w$. Note that the elements in $\overline{J}$ sum to $w - c$. Hence, $\overline{J} \cup \{w + c\}$ also sums to $2w$. Therefore, S' can be partioned into $J \cup \{2w - c\}$ and $\overline{J} \cup \{w + c\}$ where both partitions sum to $2w$. Therefore, a Yes-instance of Subset-Sum transforms to a Yes-instance of Set-Partition.

$\Longleftarrow$: Assume S' can be partitioned into two sets, A and $\overline{A} = S' - A$, such that

$$\sum_{x \in A} x = \sum_{x \in \overline{A}} x. \tag{3}$$

Since $w + (2w - c) + (w + c) = 4w$, the sum of the elements in both sets must be equal to $2w$. Therefore, $2w - c$ must be in one set and $w + c$ must be in the other because $(2w - c) + (w + c) = 3w$. Thus, by the Set-Partition (3), there must exist a subset of elements that sum to c, because $c + (2w - c) = 2w$. Therefore, a Yes-instance of Set-Partition transforms to a Yes-instance of Subset-Sum.

V. How to solve a NPC problem

Example 1: Bin Packing problem

Suppose we have an unlimited number of bins, each of capacity 1, and n objects with sizes $s_1, s_2, \ldots, s_n$, where $0 < s_i \leq 1$.

- Optimization problem: Determine the smallest number of bins into which objects can be packed and find an optimal packing.
- Decision problem: Do the objects fit in k bins?

Theorem. Bin Packing problem is NPC (reduced from the subset sum).

V: How to solve a NP-complete problem

Approximate algorithm for the Bin Packing

- *First-fit strategy (greedy):*

  places an object in the first bin into which it fits.

- Example: Objects $= \{0.8, 0.5, 0.4, 0.4, 0.3, 0.2, 0.2, 0.2\}$
- *First-fit strategy* solution:

B_1	B_2	B_3	B_4
		0.2	
0.2	0.4	0.3	
0.8	0.5	0.4	0.2

- Optimal packing:

B_1	B_2	B_3
	0.2	0.2
0.2	0.3	0.4
0.8	0.5	0.4

V. How to solve a NP-complete problem

Theorem. Let $S = \sum_{i=1}^{n} s_i$.

1. The optimal number of bins required is at least $\lceil S \rceil$
2. The number of bins used by the first-fit strategy is never more than $\lceil 2S \rceil$.

V. How to solve a NP-complete problem

The vertex-cover problem:

- A vertex-cover of an undirected graph $G = (V, E)$ is a subset set of $V' \subseteq V$ such that if $(u, v) \in E$, then $u \in V'$ (inclusive) or $v \in V'$.
- In other words, each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all edges in E.
- The size of a vertex cover is the number of vertices in it.
- Decision problem: determine whether a graph has a vertex cover of a given size k
- Optimization problem: find a vertex cover of minimum size.
- **Theorem.** The vertex-cover problem is NPC.

V. How to solve a NP-complete problem

The vertex-cover problem:

- An approximate algorithm

  $C = \emptyset$
  $E' = E$
  while $E' \neq \emptyset$
      let (u, v) be an arbitrary edge of E'
        $C = C \cup \{u, v\}$
        remove from E' every edge incident on either u or v.
  endwhile
  return C

- **Theorem.** The size of the vertex-cover is no more than twice the size of an optimal vertex cover.

I–V recap

1. Optimization problems and decision problems
2. Formal definitions of P, NP, NPC and NP-hard
   Polynomial reduction
3. How to prove a problem is NP-complete
   4 case studies
4. Approximate algorithms for solving NPC problems:
   2 case studies