Contents

I AVL Trees, Splay Trees, and Amortized Analysis

1 AVL Trees

Target Speed up searching (with insertion and deletion).

Tool Binary search trees.

Problem Although $T_p = O(height)$, but the height can be as bad as $O(N)$.

Definition I.1 *An empty binary tree is height balanced. If T is a nonempty binary tree with T_L and T_R as its left and right subtrees, then T is height balanced iff*

1) T_L and T_R are height balanced, and

2) $|h_L - h_R| \leq 1$ where h_L and h_R are the height of T_L and T_R, respectively.

The empty tree height is -1.

Definition I.2 *The balance factor $BF(node) = h_L - h_R$. In an AVL tree, $BF(node) = -1, 0, 1$.*

1.1 Tree rotation

Tree rotation is an operation on a binary tree that changes the structure without interfering with the order of the elements.

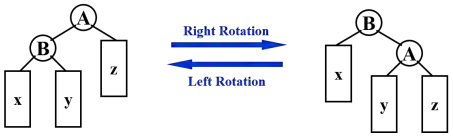


Figure 1: Tree rotation

After a rotation, the side of the rotation increases its height by 1 whilst the side opposite the rotation decreases its height similarly.

Algorithm 1 Right Rotation

```
tmp=B.rs
B.rs=A
A.ls=tmp
```

Time complexity: $O(1)$.

(旋转的是平衡因子异常的结点)

1) RR: the right subtrees's right subtree.

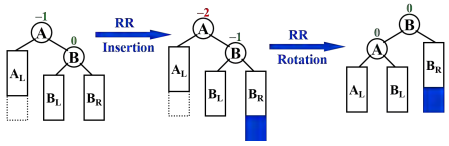


Figure 2: RR

2) LL

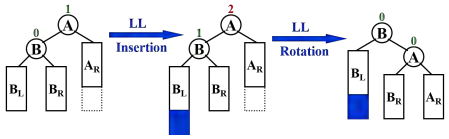


Figure 3: LL

3) LR

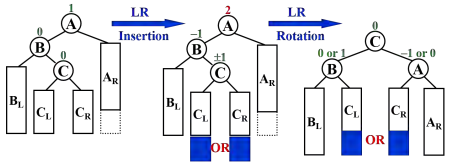


Figure 4: LR

4) RL

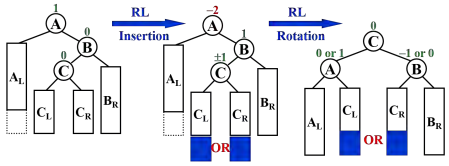


Figure 5: RL

1.2 Time Complexity of AVL

Let n_h be the minimum number of nodes in a height balanced tree of height h. Then the tree must look like

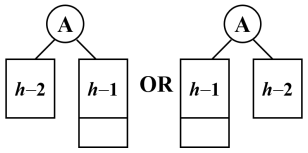


Figure 6: minimum AVL

$$n_h = n_{h-1} + n_{h-2} + 1$$

$$\Rightarrow n_h = F_{h+2} - 1$$

for $h \geq 0$.

$$
\begin{aligned}
F_i =& \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^i \\
\therefore n_h \sim& a^{h+2} \\
\therefore h =& O(\log n)
\end{aligned}
$$

2 Splay Trees

Target Any M consecutive tree operations starting from an empty tree take at most $O(M \log N)$ time.

It means taht the amortized time is $O(\log N)$.

2.1 *Insertion*

For any nonroot node X, denote its parent by P and grandparent by G:

1) P is the root: Rotate X and P.
2) P is not the root:

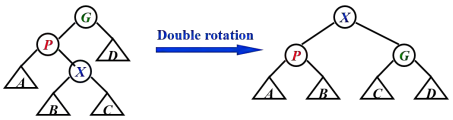


Figure 7: zig-zag

Figure 8: zig-zig

2.2 *Deletions*

1) Find X.
2) Remove X.
3) Find Max (T_L) (X 的前序结点).
4) Make T_R the right child of the root of T_L.

3 Amortized Analysis

- worst-case bound

$\geq$ amortized bound (Probability is not involved)

$\geq$ average-case bound

3.1 *Aggregate Analysis (聚合法)*

Show that for all n, a sequence of n operations takes worst-case time $T(n)$ in total. In the worst case, the average cost, or amortized cost, per operation is therefore $\frac{T(n)}{n}$.

3.2 *Accounting method (合算法)*

When an operation's amortized cost $\hat{c}_i$, exceeds its actual cost c_i, we assign the difference to specific objects in the data structure as credit. Credit can help pay for later operations whose amortized cost is less than their actual cost.

Note: For all sequences of n operations, we must have

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

3.3 *Potential method (势能法)*

The credit

$$\hat{c}_i - c_i = Credit_i = \Phi(D_i) - \Phi(D_{i-1})$$

$\Phi(D)$ is called potential function. D represents the data structure.

$$
\begin{aligned}
\sum_{i=1}^{n} \hat{c}_i =& \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\
=& \left(\sum_{i=1}^{n} c_i \right) + \Phi(D_n) - \Phi(D_0)
\end{aligned}
$$

$\Phi(D_n) - \Phi(D_0) \geq 0$.

In general, a good potential function should always assume its minimum at the start of the sequence.

II Red-Black Tree and B+ Tree

1 Red-Black Trees

Target Balanced binary search tree.

Definition II.1 *A red-black tree is a binary search tree that satisifies the following red-black properties:*

1) Every node is either red or ***black****.*

2) The root is ***black****.*

3) Every leaf (NIL, all NULL are connect the NIL) is ***black****.*

4) If a node is red, then both its children are ***black****.*

5) For each node, all simple paths from the node to descendant leaves contain the same number of ***black*** *nodes.*

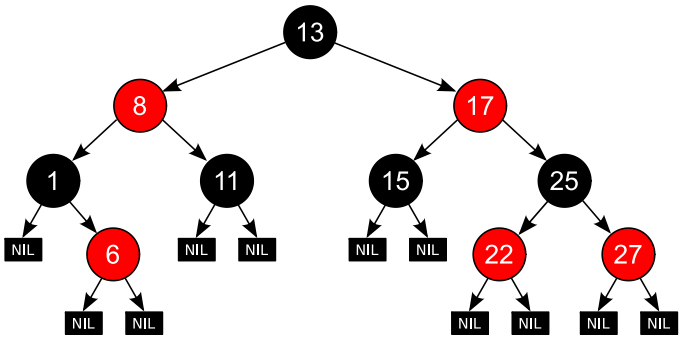


Figure 9: red-black tree

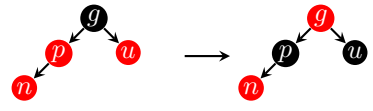
Definition II.2 *The black-height of any node x, denoted by bh(x), is the number of black nodes on any simple path from x (x not included) down to a leaf. bh(Tree)=bh(root).*

Lemma 1.1 *A red-black tree with N internal nodes has height at most* $2\ln(N+1)$.

1.1 Insert

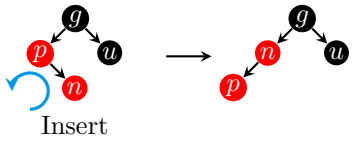
Insert node as n is red, there exist n's parent p, n's uncle u, n's grandpa g. Symmetric is same.

1) Case 1: The tree is empty, insert the first node.
2) Case 2: p is root and black, do nothing.
3) Case 3: p is root and red, dye p black.
4) Case 4
   a. Dye p and u nodes black and g node red.
   b. Recursively maintain g node.



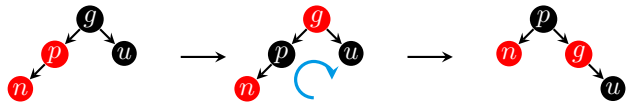
Insert

5) Case 5: Rotate p and then in Case 4.



Insert

6) Case 6
   a. Dye p black and g red.
   b. Rotate g.



Insert

1.2 Delete

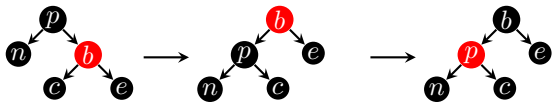
Delete node as n is red, there exist n's parent p, n's uncle u, n's grandpa g, n's son s, n's brother b, n's nephew c and e.

Delete:

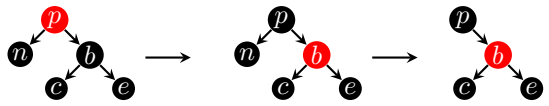
1) Case 0: n is root, delete it.
2) Case 1: n has degree 2, replace n by the largest one in its left subtree (predecessor), then delete the replacing node from the subtree, after that need to be re-maintained.
3) Case 2: n has degree 1, replace n by s. If n is black, should dye s black. If s is already black, need to be re-maintained.
4) Case 3: n is a leaf. If n is red, delete it. Else n is black, delete and then need to be re-maintained.

Re-maintained:

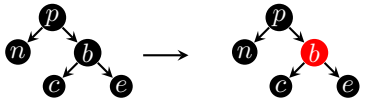
1) Case 1:
   a. Rotate p.
   b. Dye b black and p red.
   c. Maintain subtree rooted at p.



2) Case 2: Dye b red and p black.

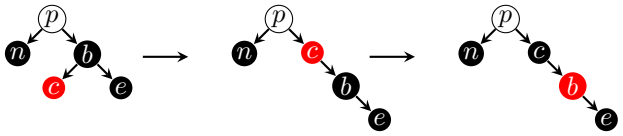


3) Case 3: Dye b black, recursively maintain p.



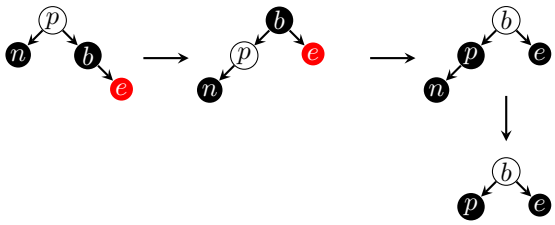
4) Case 4:

a. Rotate b.
b. Dye c red, b black.
c. Then in Case 5.



5) Case 5:

a. Rotate p.
b. Swap the color of p and b.
c. Dye distant nephew e black.



1.3 *Relationship between red-black tree and 4-order B-tree (2-3-4 tree)*

We can even say that the red-black tree and the 4-order B-tree (2-3-4 tree) are equivalent in structure.

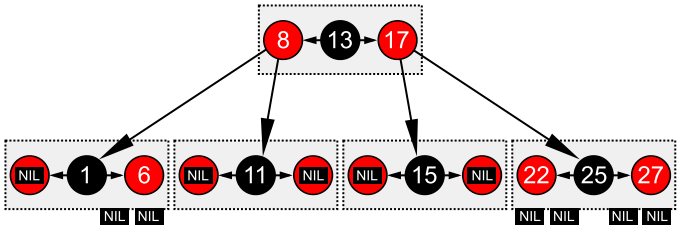


Figure 10: rbtree-btree-analogy

2 B+ Trees

Definition II.3 *A B+ tree of order (阶) M is a tree with the following structural properties:*

1) The root is either a leaf or has between 2 and M children.

2) All nonleaf nodes (except the root) have between $\lceil \frac{M}{2} \rceil$ and M children.

3) All leaves are at the same depth.

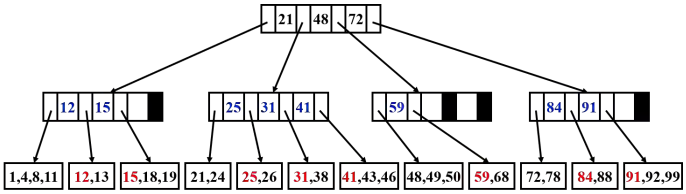


Figure 11: A B+ tree of order 4 (2-3-4 tree)

1) find
2) insert
3) delete

$$Depth(M, N) = O(\left\lceil \log_{\lceil \frac{M}{2} \rceil} N \right\rceil)$$

III Inverted Files Index

Solution:

1) scan string
2) Term-Document Incidence Matrix. e.g. Document sets

1 Compact Version - Inverted File Index

Definition III.1 *Index is a mechanism for locating a given term in a text.*

Definition III.2 *Inverted file contains a list of pointers (e.g. the nuber of the page) to all occurrences of that term in the text.*

Assume: bag of words (词顺序没那么重要)

1.1 Index Genertor

Algorithm 2 Index Genertor

```
for all document D do
    for all term T in D do
        if T doesn't exist in D then
            insert T in D
        end if
        Get T's posting list.
        Insert a node to T's posting list.
    end for
end for
```

1.2 While reading a term

1) **Word Stemming:** Process a word so that only its stem or root form is left.
2) **Stop Words:** Some words are so common that almost every document contains them, such as "a" "the" "it". It is useless to index them. They are called stop words. We can eliminate them from the original documents.

1.3 While accessing a term

Solution:

1) Search trees
2) Hashing

1.4 While not having enough memory

Memory in blocks (分块存储), and merge them.

Distributed indexing — Each node contains index of a subset of collection.

Solution:

1) Term-partitioned index
2) Document-partitioned indedx (robust)

1.5 Dynamic indexing

- Docs come in over time
  - postings update for terms already in dictionary.
  - new terms added to dictionary.
- Docs get deleted

Using main index and auxiliary index.

1.6 Compression

差分压缩 index

1.7 Thresholding

Document: only retrieve the top x documents where the documents are ranked by weight.

Problems:

- Not feasible for Boolean queries.
- Can miss some relevant documents due to truncation.

Query: Sort the query terms by their frequency in ascending order; search according to only some percentage of the original query terms.

2 Measures for a search engine

- How fast does it index
- How fast does it search
- Expressiveness of query language

2.1 Relevance measurement

1) A benchmark document collection
2) A benchmark suite of queries
3) A binary assessment of either Relevant or Irrelevant for each query-doc pair

	Relevant	Irrelevant
Retrieved	R_R	I_R
Not Retrieved	R_N	I_N

$$\text{Precision } P = \frac{R_R}{R_R + I_R}$$
$$\text{Recall } R = \frac{R_R}{R_R + R_N}$$

IV Leftist Heaps and Skew Heaps

1 Leftist Heaps

Target: Speed up merging in $O(N)$.
Heap: Structure Property + Order Property

Leftist Heap:

- Order Property — the same
- Structure Property — binary tree, but unbalanced

Definition IV.1 *The null path length, NPl(X) (距离), of any node X is the length of the shortest path from X to a node without two children. Define Npl(NULL)=-1.*

Note: Npl(X)=min{Npl(C)+1 for all C as children of X}

Definition IV.2 *The leftist heap property is that for every node X in the heap, the null path length of the left child is at least as large as that of the right child.*

Theorem IV.1 *A leftist tree with r nodes on the right path must have at least $2^r - 1$ nodes.*

Note: The leftist tree of N nodes has a right path containing at most $\lfloor \log(N+1) \rfloor$ nodes.

Declaration

```
struct TreeNode{
    ElementType Element;
    PriorityQueue Left;
    PriorityQueue Right;
    int Npl;
};
```

1.1 Merge (recursive version)

Have two leftist tree H_1 and H_2.

1) Merge(H_1->Right, H_2), the top of H_1 is less than H_2
2) Attach(H_2, H_1->Right), attach H_2 to H_1->Right
3) Swap(H_1->Right, H_1->Left)
if Nlp(Left) < Nlp(Right)

1.2 Merge (iteration version)

Have two leftist tree H_1 and H_2.

1) Sort the right paths without changing their left children.
2) Swap children if Nlp(Left) < Nlp(Right)

$T_p = O(\log N)$

1.3 Delete Min

1) Delete teh root
2) Merge the two subtrees.

$T_p = O(\log N)$

2 Skew Heaps

Target: Any M consecutive operations take at most $O(M \log N)$ time.
Merge: Always swap the left and right children except the largest of all the nodes on the right paths doesn't have its children swapped. No Npl.

Skew heaps have the advantage that no extra space is required to maintain path lengths and no tests are required to determine when to swap children.

It is an open problem to determine precisely the expected right path length of both leftist and skew heaps.

2.1 Amortized Analysis for Skew Heaps

Insert & Delete are just Merge

- $T_{amortized} = O(\log N)$?
- D_i = the root of the resulting tree
- $\Phi(D_i)$ = number of heavy nodes

Definition IV.3 *A node p is heavy if the number of descendants of p's right subtree is at least half of the number descendants of p, and light otherwise. The number of descendants of a node includes the node itself.*

The only nodes whose heavy/light status can change are nodes that are initially on the right path.

$$H_i = l_i + h_i \ (i = 1, 2), \ T_{worst} = l_1 + h_1 + l_2 + h_2$$

Before merge: $\Phi_i = h_1 + h_2 + h$
After merge: $\Phi_{i+1} \le l_1 + l_2 + h$

$$T_{amortized} = T_{worst} + \Phi_{i+1} - \Phi_i \le 2(l_1 + l_2)$$

$\because l = O(\log N) \ \therefore T_{amortized} = O(\log N)$

V Binomial Queue

A binomial queue is not a heap-ordered tree, but rather a collection of heap-ordered trees, known as a forest. Each heap-ordered tree is a binomial tree.

A binomial tree of height 0 is a one-node tree. A binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B'_{k-1}.

B_k consists of a root with k children, which are B_0, ..., B_{k-1}. B_k has exactly 2^k nodes. The number of nodes at depth d is $\binom{k}{d}$ (Binomial coefficient).

A priority queue of any size can be uniquely represented by a collection of binomial trees. (二进制)

1 Operations

1.1 FindMin

The minimun key is in one of the roots. There are at most $\lceil \log N \rceil$ roots, hence $T_p = O(\log N)$. Or just remember the minimun, it's $O(1)$.

1.2 Merge

二进制加法. 合并是任意的. $T_p = O(\log N)$. Must keep the trees in the binomial queue sorted by height.

1.3 Insert

If the smallest nonexistent binomial tree is B_i , then $T_p = Const(i + 1)$. Performing N Inserts on an initially empty binomial queue will take $O(N)$ worst-case time. Hence the average time is constant.

1.4 DeleteMin

1) FindMin in B_k
2) Remove B_k from H gets H'
3) Remove root from B_k gets H''
4) Merge (H',H'')

2 Implementation

Binomial Queue

```
typedef struct BinNode *Position;
typedef struct Collection *BinQueue;
typedef struct BinNode *BinTree;

struct BinNode{
    ElementType Element;
    Position LeftChild;
    Position NextSibling;
};

struct Collection{
    int CurrentSize;/* total number of nodes
        */
    BinTree TheTrees[MaxTrees];
};
```

MergeTrees

```
BinTree MergeTrees(BinTree T1, BinTree T2){
    if(T1->Element>T2->Element) return
        MergeTrees(T2, T1);
    T2->NextSibling=T1->LeftChild;
    T1->LeftChild=T2;
    return T1;
}
```

Merge

```
BinQueue Merge(BinQueue H1, BinQueue H2){

}
```

DeleteMin

```
BinQueue DeleteMin(BinQueue H){

}
```

3 Analysis

Claim 3.1 *A binomial queue of N elements can be built by N successive insertions in $O(N)$ time.*

VI Backtracking

1 Rationale of the Backtracking Algorithms

The basic idea is that suppose we have a partial solution $(x_1, \dots, x_i)$ where each $x_k \in S_k$ for $1 \le k \le i < n$. First we add $x_{i+1} \in S_{i+1}$ and check if $(x_1, \dots, x_i, x_{i+1})$ satisfies the constrains. If the answer is "yes" we continue to add the next x, else we delete x_i and backtrack to the previous partial solution $(x_1, \dots, x_{i-1})$.

先决策点少的状态.

2 The Turnpike Reconstruction Problem

Given N points on the x-axis with coordinates $x_1 < x_2 < \cdots < x_N$. Assume that $x_1 = 0$. There are $N(N–1)/2$ distances between every pair of points. Given $N(N–1)/2$ distances. Reconstruct a point set from the distances.

手玩使用性质搜索

3 Tic-tac-toe: Minimax Strategy

The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game.

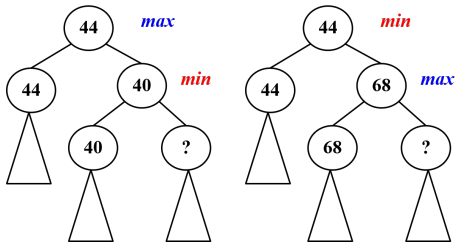
Use an evaluation function to quantify the "goodness" of a position. For example:

$$f(P) = W_{Computer} - W_{Human}$$

where W is the number of potential wins (在此种局面下让一方一直下可以得到的不同的胜利局数) at position P.

The human is trying to minimize the value of the position P, while the computer is trying to maximize it.

3.1 α-β pruning



(a) α pruning (b) β pruning

Figure 12: α-β pruning

When both techniques are combined. In practice, it limits the searching to only $O(\sqrt{N})$ nodes, where N is the size of the full game tree.

VII Divide and Conquer

Recursively:

1) **Divide** the problem into a number of sub problems
2) **Conquer** the sub problems by solving them recursively
3) **Combine** the solutions to the sub problems into the solution for the original problem

General recurrence:

$$T(N) = aT(N/b) + f(N)$$

1 Closest Points Problem

Given N points in a plane. Find the closest pair of points. (If two points have the same position, then that pair is the closest with distance 0.)

Let $a = b = 2, f(N) = cN, k = \log_2 N$

$$\begin{aligned} T(N) &= 2T(N/2) + cN \\ &= 2^k T(N/2^k) + kcN \\ &= N + cN \log N = O(N \log N) \end{aligned}$$

2 Methods for solving recurrences

2.1 Substitution method

Guess, then prove by induction. Always assume $T(n) = \Theta(1)$ for small n.

Example: $T(N) = 2T(\lfloor N/2 \rfloor) + N$

Guess: $T(N) = O(N \log N)$

Proof: Assume it's true for all $n < N$, in particular for $m = \lfloor N/2 \rfloor$.

Then there exists a constant $c > 0$ so that

$$T(\lfloor N/2 \rfloor) \le c\lfloor N/2 \rfloor \log \lfloor N/2 \rfloor$$

Substituting into the recurrence:

$$\begin{aligned} T(N) &= 2T(\lfloor N/2 \rfloor) + N \\ &\le 2c\lfloor N/2 \rfloor \log \lfloor N/2 \rfloor + N \\ &\le cN(\log N - \log 2) + N \\ &\le cN \log N \text{ for } c \ge 1 \end{aligned}$$

Must prove the exact form. (数学归纳的结果需要和假设一致)

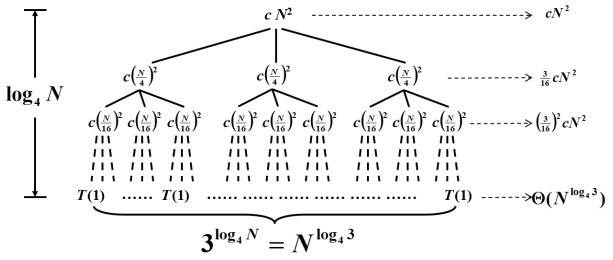


Figure 13: Recursion-tree method

2.2 Recursion-tree method

Example: $T(N) = 3T(N/4) + \Theta(N^2)$

$$\begin{aligned} T(N) &= \sum_{i=0}^{\log_4 N-1} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3}) \\ &< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cN^2 + \Theta(N^{\log_4 3}) \\ &= \frac{cN^2}{1-3/16} + \Theta(N^{\log_4 3}) \\ &= O(N^2) \end{aligned}$$

仅猜测, 需要用法一进行证明.

2.3 Master method

Theorem VII.1 (Master method I)

Let $a \geq 1$ and $b > 1$ be constants, let $f(N)$ be a function, and let $T(N)$ be defined on the nonnegative integers the recurrence $T(N) = aT(N/b) + f(N)$. Then

1) If $\exists\ \epsilon > 0,\ f(N) = O(N^{\log_b a - \epsilon})$, then $T(N) = \Theta(N^{\log_b a})$.
2) If $f(N) = \Theta(N^{\log_b a})$, then $T(N) = \Theta(N^{\log_b a} \log N)$.
3) If $\exists\ \epsilon > 0,\ f(N) = \Omega(N^{\log_b a + \epsilon})$, and if $\exists\ n > 0, N > n$, $\exists\ c < 1,\ af(N/b) < cf(N)$ (regularity condition), then $T(N) = \Theta(f(N))$.

Theorem VII.2 (Master method II)

The recurrence $T(N) = aT(N/b) + f(N)$ can be solved as follows:

1) If $\exists\ \kappa < 1,\ af(N/b) = \kappa f(N)$, then $T(N) = \Theta(f(N))$.
2) If $\exists\ K > 1,\ af(N/b) = Kf(N)$, then $T(N) = \Theta(N^{\log_b a})$.
3) If $af(N/b) = f(N)$, then $T(N) = \Theta(f(N) \log_b N)$.

Theorem VII.3 (Master method III)

The solution to the equation

$$T(N) = aT(N/b) + \Theta(N^k \log^p N)$$

where $a \geq 1, b > 1, p \geq 0$ is

$$T(N) = \begin{cases} O(N^{\log_b a}) & \text{if } a > b^k \\ O(N^k \log^{p+1} N) & \text{if } a = b^k \\ O(N^k \log^p N) & \text{if } a < b^k \end{cases}$$

VIII Dynamic Programming

Solve sub-problems just once and save answers in a table. Use a table instead of recursion.

1 Fibonacci Numbers

$$F(N) = F(N-1) + F(N-2)$$

2 Ordering Matrix Multiplications

Problem: In which order can we compute the product of n matrices with minimal computing time?

Suppose we are to multiply n matrices $M_1 * \cdots * M_n$ where M_i is an $r_{i-1} \times r_i$ matrix. Let m_{ij} be the cost of the optimal way to compute $M_i * \cdots * M_j$. Then we have the recurrence equations:

$$m_{ij} = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le l < j}\{m_{il} + m_{l+1j} + r_{i-l} r_l r_j\} & \text{if } j > i \end{cases}$$

3 Optimal Binary Search Tree

The best for static searching (without insertion and deletion)

Given N words $w_1 < w_2 < \cdots < w_N$, and the probability of searching for each w_i is p_i . Arrange these words in a binary search tree in a way that minimize the expected total access time.

$$T(N) = \sum_{i=1}^{N} p_i(1 + d_i)$$

T_{ij} ::= (相当于)OBST for $w_i, \ldots, w_j (i < j)$

c_{ij} ::= cost of $T_{ij} (c_{ii} = 0)$

r_{ij} ::= root of T_{ij}

$$w_{ij} ::= \text{weight of } T_{ij} = \sum_{k=i}^{j} p_k (w_{ii} = p_i)$$

T_{ij} is optimal $\Rightarrow r_{ij} = k$ is such taht

$$c_{ij} = \min_{i < l \le j}\{w_{ij} + c_{i,l-1} + c_{l+1,j}\}$$

4 All-Pairs Shortest Path

For all pairs of v_i and $v_j (i \neq j)$, find the shortest path between.

Define $D_{ij}^k = \min\{$ length of path $i \to \{l \le k\} \to j\}$ and $D_{ij}^{-1} = Cost_{ij}$. Then the length of the shortest path from i to j is D_{ij}^N.

$$D_{ij}^k = \min\{D_{ij}^{k-1},\ D_{ik}^{k-1} + D_{kj}^{k-1}\},\ k \ge 0$$

IX Greedy Algorithms

别人恐惧我贪婪

1 Activity Selection Problem

Given a set of activities $S = \{a_1, a_2, \ldots, a_n\}$that wish to use a resource (e.g. a classroom). Each a_i takes place during a time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if $f_j \le s_i$ or $f_i \le s_j$ (i.e. their time intervals do not overlap).

Select a maximum-size subset of mutually compatible activities.

1.1 DP

$$c_{1,j} = \begin{cases} 1 & \text{if } j = 1 \\ \max\{c_{1,j-1}, c_{1,k(j)} + 1\} & \text{if } j > 1 \end{cases}$$

where $c_{1,j}$ is the optimal solution for a_1 to a_j , and $a_{k(j)}$ is the nearest compatible activity to a_j that is finished before a_j .

If each activity has a weight

$$c_{1,j} = \begin{cases} w_j & \text{if } j = 1 \\ \max\{c_{1,j-1}, c_{1,k(j)} + w_j\} & \text{if } j > 1 \end{cases}$$

Greedy solution isn't correct with weighted activities.

1.2 Greedy

Greedy Rule: Select the interval which ends first (but not overlapping the already chosen intervals).

Theorem IX.1 *Consider any nonempty subproblem S_k, and let a_m be an activity in S_k with the earliest finish time. Then a_m is included in some maximum-size subset of mutually compatible activities of S_k. (用贪心策略得到的解元素代替最优解中的元素, 得到的解不会更差)*

Implementation:

1) Select the first activity; Recursively solve for the rest.

2) Remove tail recursion by iterations.

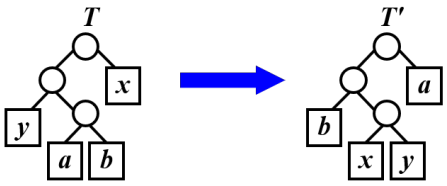
$O(N \log N)$

2 Huffman Codes

Representation of the original code in a binary tree. Find the full binary tree of minimum total cost where all characters are contained in the leaves.

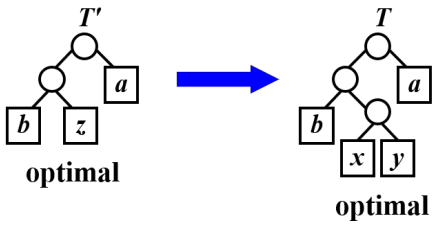
Lemma 2.1 *Let C be an alphabet in which each character $c \in C$ has frequency $c.freq$. Let x and y be two characters in*

C having the lowest frequencies. Then there exists an optimal prefix code for C in which the codewords for x and y have the same length and differ only in the last bit.



$$Cost(T) \geq Cost(T')$$

Lemma 2.2 *Let C be a given alphabet with frequency c.freq defined for each character $c \in C$. Let x and y be two characters in C with minimum frequency. Let C′ be the alphabet C with a new character z replacing x and y, and $z.freq = x.freq + y.freq$. Let T′ be any tree representing an optimal prefix code for the alphabet C′. Then the tree T, obtained from T′ by replacing the leaf node for z with an internal node having x and y as children, represents an optimal prefix code for the alphabet C.*



$$Cost(T') + x.freq + y.freq = Cost(T)$$

X NP-Completeness

1 Easy vs. Hard

- The easiest: $O(N)$ –since we have to read inputs at least once.
- The hardest: undecidable problems.

2 The Class NP

2.1 TURING MACHINE

2.1.1 Task To simulate any kind of **computation** which a mathematician can do by some **arithmetical method** (assuming that the mathematician has infinite time, energy, paper and pen, and is completely dedicated to the work).

2.1.2 Components Infinite Memory and Scanner

2.1.3 Operations

1) Change the finite control state.
2) Erase the symbol in the unit currently pointed by head, and write a new symbol in.
3) Head moves one unit to the left (L), or to the right (R), or stays at its current position (S).

A **Deterministic Turing Machine** executes one instruction at each point in time. Then depending on the instruction, it goes to the next unique instruction.

A **Nondeterministic Turing Machine** is free to choose its next step from a finite set. And if one of these steps leads to a solution, it will always choose the correct one.

2.2 NP: Nondeterministic polynomial-time

The problem is **NP** if we can prove any solution is true in polynomial time.

Note: Not all decidable problems are in NP.

$$P \subseteq NP$$

2.3 NP-Complete Problems

An **NP-complete problem** has the property that any problem in NP can be polynomially reduced (归约) to it.

Definition X.1 (Reduction)

Given $\forall$ instance $\alpha \in$ Problem A, if we can find a program $R(\alpha) \to \beta \in$ Problem B with $T_R(N) = O(N^{k_1})$, and another program $D(\beta)$ to get an answer in time $O(N^{k_2})$. And more if the answer for β is the same as the answer for α. Then

2.3.1 Example Suppose that we already know that the Hamiltonian cycle problem is NP-complete. Prove that the traveling salesman problem is NP-complete as well.

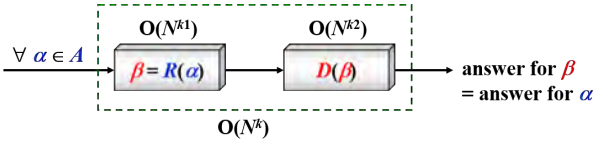


Figure 14: Reduction

- Hamiltonian cycle problem: Given a graph $G = (V, E)$, is there a simple cycle that visits all vertices?
- Traveling salesman problem: Given a complete graph $G = (V, E)$, with edge costs, and an integer K, is there a simple cycle that visits all vertices and has total cost $\leq K$?

Proof 2.1 *TSP is obviously in NP, as its answer can be verified polynomially.*

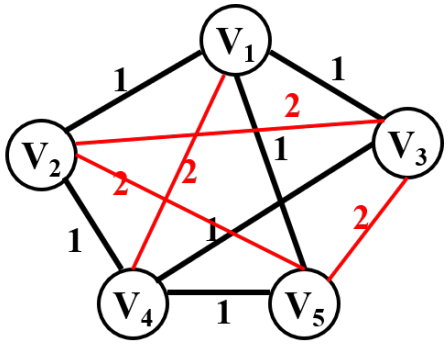


Figure 15: G and G'

G has a Hamilton cycle iff G' has a traveling salesman tour of total weight $|V|$.

$$K = |V|$$

3 A Formal-language Framework

3.1 Abstract Problem

An abstract problem Q is a binary relation on a set I of problem instances and a set S of problem solutions.

3.1.1 Example For SHORTEST-PATH problem

$I = \{\langle G, u, v\rangle : G = (V, E)$ is an undirected graph; $u, v \in V\}$

$S = \{\langle u, w_1, w_2, \ldots, w_k, v\rangle : \langle u, w_1\rangle, \ldots, \langle w_k, v\rangle \in E\}$.

For every $i \in I$, SHORTEST-PATH$(i) = s \in S$.

For decision problem PATH:

$I = \{\langle G, u, v, k\rangle : G = (V, E)$ is an undirected graph;
$u, v \in V; k \geq 0$ is an integer$\}$

$S = \{0, 1\}$.

For every $i \in I$, PATH$(i) = 0$ or 1.

3.2 Encodings

Map I into a binary string $\{0, 1\}^* \to Q$ is a **concrete problem**.

3.3 Formal-language Theory

for decision problem Q

1) An **alphabet** Σ is a finite set of symbols
2) A **language** L over Σ is any set of strings made up of symbols from Σ
3) Denote **empty** string by ε
4) Denote **empty** language by $\emptyset$
5) Language of all strings over Σ is denoted by Σ^*
6) The **complement** of L is denoted by $\Sigma^* - L$
7) The **concatenation** of two languages L_1 and L_2 is the language

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

8) The closure or **Kleene star** of a language L is the language

$$L^* = \{\varepsilon\} \cup L \cup L_2 \cup L_3 \cup \cdots$$

where L_k is the language obtained by concatenating L to itself k times
9) Algorithm A **accepts** a string $x \in \{0, 1\}^*$ if $A(x) = 1$
10) Algorithm A **rejects** a string x if $A(x) = 0$
11) A language L is **decided** by an algorithm A if every binary string in L is **accepted** by A and every binary string not in L is **rejected** by A
12) To **accept** a language, an algorithm need only worry about strings in L, but to **decide** a language, it must correctly accept or reject every string in 0, 1*

$$P = \{L \subseteq \{0, 1\}^* : D\}$$

where D means there exists an algorithm A that **decides** L in polynomial time.
13) A **verification algorithm** is a two-argument algorithm A, where one argument is an ordinary input string x and the other is a binary string y called a **certificate**.
14) A two-argument algorithm A **verifies** an input string x if there exists a certificate y such that $A(x, y) = 1$.
15) The **language** verified by a verification algorithm A is

$$L = \{x \in \{0, 1\}^* : D\}$$

where D means there exists $y \in \{0, 1\}^*$ such that $A(x, y) = 1$.

A language L belongs to **NP** iff there exist a two-input polynomial-time algorithm A and a constant c such that

$$L = \{x \in \{0, 1\}^* : D\}$$

where D means there exists a certificate y with $|y| = O(|x|^c)$ such that $A(x, y) = 1$. We say that algorithm A verifies language L in polynomial time.

complexity class co-NP = the set of languages L such that

$$\bar{L} \in NP$$

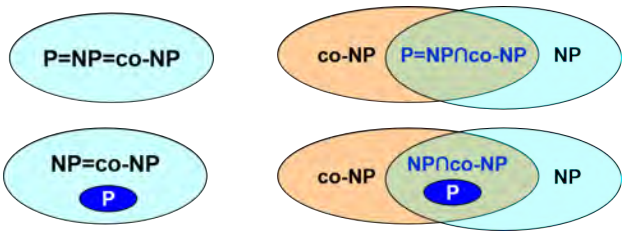


Figure 16: Four possibilities

16) A language L_1 is **polynomial-time reducible** to a language L_2 ($L_1 \leq_P L_2$, L_1 no harder than L_2) if there exists a polynomial-time computable function

$$f : \{0,1\}^* \to \{0,1\}^*$$

such that for all $x\{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$. We call the function f the **reduction function**, and a polynomial-time algorithm F that computes f is called a **reduction algorithm**.

A language $L \subseteq \{0,1\}^*$ is **NP-complete** if

a. $L \in$ NP, and
b. $\forall L' \in$ NP, $L' \leq_P L$.

XI Approximation

Find near-optimal solutions in polynomial time

1 Approximation Ratio

Definition XI.1 *An algorithm has an approximation ratio of $\rho(n)$ if $\forall$input of size n, the cost C of the solution produced by the algorithm is within a factor of $\rho(n)$ of the cost C^* of an optimal solution:*

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq \rho(n)$$

If an algorithm achieves an approximation ratio of $\rho(n)$, we call it a $\rho(n)$-approximation algorithm.

Definition XI.2 *An approximation scheme for an optimization problem is an approximation algorithm that takes as input not only an instance of the problem, but also a value $\varepsilon > 0$ such that $\forall$fixed ε, the scheme is a $(1+\varepsilon)$-approximation algorithm.*

We say that an approximation scheme is a polynomial-time approximation scheme (PTAS) if $\forall$ fixed $\varepsilon > 0$, the scheme runs in time polynomial in the size n of its input instance.

e.g.

- PTAS: $O(n^{2/\varepsilon})$,
- Fully polynomial-time approximation scheme (FPTAS): $O((1/\varepsilon)^2 n^3)$(关于 $(1/\varepsilon)$ 和 n 都是多项式级的)

2 Approximate Bin Packing

Given N items of sizes $S_1, S_2, \ldots, S_N$, such that $0 < S_i \leq 1$ $\forall$ $1 \leq i \leq N$. Pack these items in the fewest number of bins, each of which has unit capacity.

2.1 Next Fit

Theorem XI.1 *Let M be the optimal number of bins required to pack a list I of items. Then next fit never uses more than $2M$–1 bins. There exist sequences such that next fit uses $2M$–1 bins.*

2.2 First FIt

Theorem XI.2 *Let M be the optimal number of bins required to pack a list I of items. Then first fit never uses more than $17M/10$ bins. There exist sequences such that first fit uses $17(M$–$1)/10$ bins.*

2.3 Best Fit

Place a new item in the tightest spot among all bins.
$T = O(NlogN)$ and bin no.$\leq 1.7M$

2.4 On-line Algorithms

Place an item before processing the next one, and cann't change decision.

Theorem XI.3 *There are inputs that force any on-line bin-packing algorithm to use at least* $\frac{5}{3}$ *the optimal number of bins.*

2.5 Off-line Algorithms

View the entire item list before producing an answer.

Theorem XI.4 *Let* M *be the optimal number of bins required to pack a list* I *of items. Then first fit decreasing never uses more than* $11M/9+6/9$ *bins. There exist sequences such that first fit decreasing uses* $11M/9+6/9$ *bins.*

3 The Knapsack Problem

3.1 fractional version

A knapsack with a capacity M is to be packed. Given N items. Each item i has a weight w_i and a profit p_i . If x_i is the percentage of the item i being packed, then the packed profit will be $p_i x_i$.

An optimal packing is a feasible one with maximum profit (maximum profit density $\frac{p_i}{w_i}$). That is, we are supposed to find the values of x_i such that $\sum_{i=1}^{n} p_i x_i$ obtains its maximum under the constrains

$$\sum_{i=1}^{n} w_i x_i \leq M$$

$x_i \in [0,1]$ for $1 \leq i \leq n$.

3.2 0-1 version

x_i is either 1 or 0.

The approximation ratio is 2 if we are greedy on taking the maximum profit or profit density.

3.3 Dynamic Programming Solution

$W_{i,p}$ is the minimum weight of a collection from $\{1,\dots,i\}$ with total profit being exactly p.

$$W_{i,p} = \begin{cases} \infty & i = 0 \\ W_{i-1,p} & p_i > p \\ \min\{W_{i-1,p}, w_i + W_{i-1,p-p_i}\} & \text{otherwise} \end{cases}$$

$i = 1,\dots,n$, $p = 1,\dots,np_{max}$. $O(n^2 p_{max})$

If p_{max} is large, can round all profit values up to lie in smaller range.

$$\forall \text{ feasible solution } P, (1+\varepsilon)P_{alg} \leq P$$

ε is precision parameter.

4 The K-center Problem

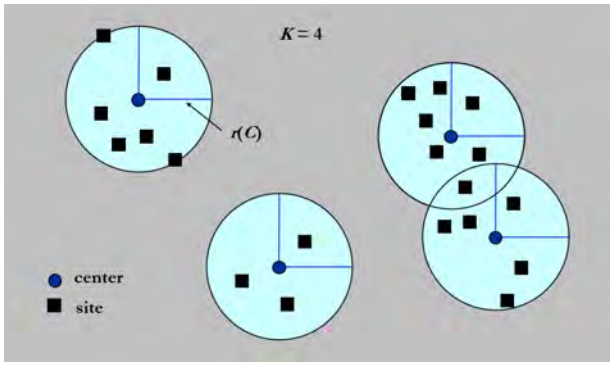


Figure 17: K-center Problem

- Input: Set of n sites $s_1,\dots,s_n$
- Center selection problem: Select K centers C so that the maximum distance from a site to the nearest center is minimized.

4.1 Distance

1) $dist(x,x) = 0$ (identity)
2) $dist(x,y) = dist(y,x)$ (symmetry)
3) $dist(x,y) \leq dist(x,z) + dist(z,y)$ (triangle inequality)

$$\begin{aligned} dist(s_i, C) &= \min_{c \in C} dist(s_i, c) \\ &= \text{distance from } s_i \text{ to the closest center} \\ r(C) &= \max_i dist(s_i, C) \\ &= \text{smallest covering radius} \end{aligned}$$

Target: Find a set of centers C that minimizes $r(C)$, subject to $|C| = K$.

4.2 A Greedy Solution

If we know that $r(C^*) \leq r$ where C^* is the optimal solution set, and choose center at the site, we can verify whether r meets the requirements.

```
1 Centers Greedy-2r(Sites S[], int n, int K,
    double r){
2     Sites  S_[]=S[]; /* S_ is the set of the
          remaining sites */
3     Centers C[]={};
4     while(!S_.empty){
5         Select any s from S_ and add it to C;
6         Delete all s_ from S_ that are at
              dist(s_, s) ≤ 2r;
```

```
    } /* end-while */
    if (|C|<=K) return C;
    else ERROR(No set of K centers with
        covering radius at most r);
}
```

Theorem XI.5 *Suppose the algorithm selects more than K centers. Then for any set C^* of size at most K, the covering radius is $r(C^*) > r$.*

Binary search for r to know $r(C^*)$. Solution radius is $2r_1$ — 2-approximation.

Or

```
Centers Greedy-Kcenter(Sites S[], int n, int
    K){
    Centers  C[] = {};
    Select any s from S and add it to C;
    while (|C|<K) {
        Select s from S with maximum dist(s, C);
        Add s it to C;
    } /* end-while */
    return C;
}
```

Theorem XI.6 *The algorithm returns a set C of K centers such that $r(C) \leq 2r(C^*)$ where C^* is an optimal set of K centers.(2-approximation)*

Theorem XI.7 *Unless $P = NP$, there is no ρ-approximation for center-selection problem for any $\rho < 2$.*

5 Three aspects to be considered

- A: Optimality(quality of a solution)
- B: Efficiency(cost of computations)
- C: All instances

Researchers are working on

- A+C: Exact algorithms for all instances
- A+B: Exact and fast algorithms for special cases
- B+C: Approximation algorithms

Even if P=NP, still we cannot guarantee A+B+C .

XII Local Search

Solve problems approximately — aims at a local optimum.

Framework of Local Search:

- Loacl

  Define neighborhoods in the feasible set.

  A local optimum is a best solution in a neighborhood.
- Search

  Start with a feasible solution and search a better one within the neighborhood.

  A local optimum is achieved if no improvement is possible.

Neighbor Relation:

- $S \sim S'$: S' is a neighboring solution of S — S' can be obtained by a small modification of S.
- $N(S)$: neighborhood of S — the set $\{S' : S \sim S'\}$.

```
SolutionType Gradient_descent(){
    Start from a feasible solution S in FS;
    MinCost=cost(S);
    while(1){
        S_=Search(N(S)); /* find the best S_
            in N(S) */
        CurrentCost=cost(S_);
        if(CurrentCost<MinCost){
            MinCost=CurrentCost;
            S=S_;
        }
        else  break;
    }
    return S;
}
```

1 The Vertex Cover Problem

Given an undirected graph $G = (V, E)$. Find a minimum subset S of V such that for each edge (u, v) in E, either u or v is in S.

- Feasible solution set FS : all the vertex covers.
- $cost(S) = |S|$
- $S \sim S'$: Each vertex cover S has at most $|V|$ neighbors
- Search: Start from $S = V$; delete a node and check if S' is a vertex cover with a smaller cost.

2 Simulated Annealing

The material is cooled very gradually from a high temperature, allowing it enough time to reach equilibrium at a succession of intermediate lower temperatures.

3 Hopfield Neural Networks

Graph $G = (V, E)$ with integer edge weights w (positive or negative).

- If $w_e < 0$, where $e = (u, v)$, then u and v want to have the same state(± 1).
- if $w_e > 0$ then u and v want different states.

The absolute value $|w_e|$ indicates the strength of this requirement.

Find a sufficiently good configuration S of the network — an assignment of the state s_u to each node u.

Definition XII.1 *In a configuration S, edge $e = (u, v)$ is good if $w_e s_u s_v < 0$ ($w_e < 0$ iff $s_u = s_v$); otherwise, it is bad.*

Definition XII.2 *In a configuration S, a node u is satisfied if the weight of incident good edges $\geq$ weight of incident bad edges.*

$$\sum_{u,v:e=(u,v)\in E} w_e s_u s_v \leq 0$$

Definition XII.3 *A configuration is stable if all nodes are satisfied.*

3.1 State-flipping Algorithm

```
ConfigType State_flipping(){
    Start from an arbitrary configuration S;
    while(!IsStable(S)){
        u = GetUnsatisfied(S);
        su = -su;
    }
    return S;
}
```

Claim 3.1 *The state-flipping algorithm terminates at a stable configuration after at most $W = \sum_e |w_e|$ iterations.*

Related to Local Search:

- Problem: To maximize Φ.
- Feasible solution set FS: configurations
- $S \sim S'$: S' can be obtained from S by flipping a single state

Claim 3.2 *Any local maximum in the state-flipping algorithm to maximize Φ is a stable configuration.*

4 The Maximum Cut Problem

Given an undirected graph $G = (V, E)$ with positive integer edge weights w_e, find a node partition (A, B) such that the total weight of edges crossing the cut is maximized.

$$w(A, B) := \sum_{u\in A, v\in B} w_{uv}$$

Related to Local Search:

- Problem: To maximize $\Phi(S) = \sum_{e \text{ is good}} |w_e|$.
- Feasible solution set FS: any partition (A, B)
- $S \sim S'$: S' can be obtained from S by moving one node from A to B, or one from B to A.

A special case of Hopfield Neural Network with w_e all being positive.

Claim 4.1 *Let (A, B) be a local optimal partition and let (A^*, B^*) be a global optimal partition. Then $w(A, B) \geq \frac{1}{2} w(A^*, B^*)$.*

Unless P = NP, no 17/16(1.0625) approximation algorithm for MAX-CUT.

4.1 Big-improvement-flip

Only choose a node which, when flipped, increases the cut value by at least

$$\frac{2\varepsilon}{|V|} w(A, B)$$

Claim 4.2 *Upon termination, the big-improvement-flip algorithm returns a cut (A, B) so that*

$$(2 + \varepsilon) w(A, B) \geq w(A^*, B^*)$$

Since $\forall x \geq 1$, we have $\left(1 + \frac{1}{x}\right)^x \geq 2$, so the objective function doubles at least every $\frac{n}{\varepsilon}$ flips.

Claim 4.3 *The big-improvement-flip algorithm terminates after at most $O\left(\frac{n}{\varepsilon} \log W\right)$ flips.*

4.2 K-L heuristic

A better local, k-flip.

Step 1 make 1-flip as good as we can - $O(n) \Rightarrow (A_1, B_1)$ and v_1

Step k make 1-flip of an unmarked node as good as we can - $O(n - k + 1) \Rightarrow (A_k, B_k)$ and $v_1, \ldots, v_k$

Step n $(A_n, B_n) = (B, A)$

Neighborhood of $(A, B) = \{(A_1, B_1), \ldots, (A_{n-1}, B_{n-1})\}$ - $O(n^2)$

XIII Randomized Algorithms

1) What to Randomize?

Average-case Analysis: The world behaves randomly — randomly generated input solved by traditional algorithm

Randomized Algorithms: The algorithm behaves randomly — make random decisions as the algorithm processes the worst-case input

2) Why Randomize?

Efficient deterministic algorithms that always yield the correct answer are a special case of

- efficient randomized algorithms that only need to yield the correct answer with high probability.
- randomized algorithms that are always correct, and run efficiently in expectation.

1 A Quick Review

- $Pr[A]$:= the probability of the event A
- $\bar{A}$:= the complementary of the event A (A did not occur)

$$Pr[A] + Pr[\bar{A}] = 1$$

- $E[X]$:= the expectation (the "average value") of the random variable X

$$E[X] = \sum_{j=0}^{\infty} j \cdot Pr[X = j]$$

2 The Hiring Problem

- Hire an office assistant from headhunter
- Interview a different applicant per day for N days
- Interviewing Cost = $C_i \ll$ Hiring Cost = C_h
- Analyze interview & hiring cost

Assume M people are hired.

Total Cost: $O(NC_i + MC_h)$

2.1 Naïve Solution

```
int Hiring(EventType C[], int N){ /*
    candidate 0 is a least-qualified dummy
    candidate */
    int Best = 0;
    int BestQ = the quality of candidate 0;
    for(i=1;i<=N;i++){
        Qi = interview(i); /* Ci */
        if(Qi>BestQ){
            BestQ = Qi;
            Best = i;
            hire(i);  /* Ch */
        }
    }
    return Best;
}
```

Worst case: The candidates come in increasing quality order $O(NC_h)$

2.2 Assume candidates arrive in random order

Randomness assumption: any of first i candidates is equally likely to be best-qualified so far.

$$X = \text{number of hires}$$

$$E[X] = \sum_{j=1}^{N} j \cdot Pr[X = j]$$

$$X_i = \begin{cases} 1 & \text{if candidate } i \text{ is hired} \\ 0 & \text{if candidate } i \text{ isn't hired} \end{cases}$$

$$\Rightarrow X = \sum_{i=1}^{N} X_i$$

$$E[X_i] = Pr[\text{candidate } i \text{ is hired}] = \frac{1}{i}$$

$$\Rightarrow E[X] = E\left[\sum_{i=1}^{N} X_i\right] = \sum_{i=1}^{N} E[X_i]$$

$$\Rightarrow O(C_h \ln N + NC_i)$$

Radomized Algorithm: takes time to randomly permute the list of candidates;

2.2.1 Radomized Permutation Algorithm

Target: Permute array $A[]$.

Assign each element $A[i]$ a random priority $P[i]$, and sort.

```
void PermuteBySorting(ElemType A[], int N){
    for(i=1;i<=N;i++)
        A[i].P = 1 + rand()%(N^3);
        /* makes it more likely that all
            priorities are unique */
    Sort A; /* using P as the sort keys*/
}
```

Claim 2.1 *PermuteBySorting produces a uniform random permutation of the input, assuming all priorities are distinct.*

2.3 Online Hiring Algorithm -hire only once

```
int OnlineHiring(EventType C[], int N, int k
    ){
    int Best = N;
```

```
    int BestQ = -∞ ;
    for(i=1;i<=k;i++){
        Qi = interview(i);
        if(Qi>BestQ)BestQ = Qi;
    }
    for(i=k+1;i<=N;i++){
        Qi = interview(i);
        if(Qi>BestQ){
            Best = i;
            break;
        }
    }
    return Best;
}
```

$S_i :=$ the i-th applicant is the best.
S_i is true if

$$\{A := \text{the best one is at position } i\}$$
$$\cap\{B := \text{no one at positions } k+1 \sim i-1 \text{ are hired}\}$$

A and B are independent.

$$Pr[S_i] = Pr[A \cap B] = Pr[A] \cdot Pr[B]$$
$$= \frac{1}{N}\frac{k}{(i-1)} = \frac{k}{N(i-1)}$$
$$Pr[S] = \sum_{i=k+1}^{N} Pr[S_i] = \sum_{i=k+1}^{N} \frac{k}{N(i-1)} = \frac{k}{N}\sum_{i=k}^{N-1}\frac{1}{i}$$

The probability we hire the best qualified candidate for a given k:

$$\frac{k}{N}\ln\left(\frac{N}{k}\right) \le Pr[S] \le \frac{k}{N}\ln\left(\frac{N-1}{k-1}\right)$$

The best value of k to maximize the above probability:

$$f(k) = \frac{k}{N}\ln\left(\frac{N}{k}\right)$$
$$f'(k) = \frac{1}{N}\ln\left(\frac{N}{k}\right) - \frac{1}{N} = 0$$
$$k = \frac{N}{e}$$
$$f_{max}\left(\frac{N}{e}\right) = \frac{1}{e}$$

3 Quicksort

Deterministic Quicksort

- $\Theta(N^2)$ worst-case running time
- $\Theta(N \log N)$ average case running time, assuming every input permutation is equally likely

Central splitter := the pivot that divides the set so that each side contains at least $n/4$

Modified Quicksort := always select a central splitter before recursions

Claim 3.1 *The expected number of iterations needed until we find a central splitter is at most 2.*

$$Pr[\text{find a central splitter}] = \frac{1}{2}$$

Type j : the subproblem S is of type j if

$$N\left(\frac{3}{4}\right)^{j+1} \le |S| \le N\left(\frac{3}{4}\right)^{j}$$

Claim 3.2 *There are at most* $\left(\frac{4}{3}\right)^{j+1}$ *subproblems of type* j.

$$\left.\begin{array}{l} E[T_{type\ j}] = O\left(N\left(\frac{3}{4}\right)^{j}\right) \times \left(\frac{4}{3}\right)^{j+1} = O(N) \\ Number\ of\ different\ types\ = \log_{4/3} N = O(\log N) \end{array}\right\}$$
$$= O(N \log N)$$

XIV Parallel Algorithm

To describe a parallel algorithm

- Parallel Random Access Machine (PRAM)
- Work-Depth (WD)

1 PRAM

A PRAM employs p synchronous processors, all having unit time access to a shared memory. Each processor has a local memory. At each time unit, a processor can read/write/do some computation with respect to its local memory.

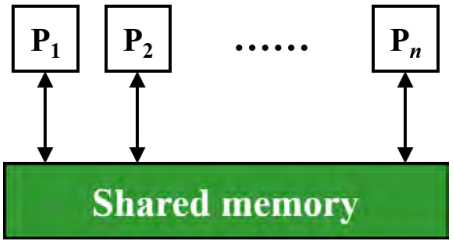


Figure 18: PRAM

e.g.

```
for P_i, 1 <=i<=n pardo
    A(i):=B(i)
```

1.1 To resolve access conflicts

- Exclusive-Read Exclusive-Write (EREW)
- Concurrent-Read Exclusive-Write (CREW)
- Concurrent-Read Concurrent-Write (CRCW)
  - Arbitrary rule
  - Priority rule (P with the smallest number)
  - Common rule (if all the processors are trying to write the same value)

2 The summation problem

Input: $A(1), A(2), \ldots, A(n)$
Output: $A(1) + A(2) + \cdots + A(n)$

2.1 PRAM model

```
for(P_i, 1<=i<=n)pardo{
    B(0, i):=A(i);
    for(h=1 to logn)do{
        if(i<=n/(2^h))
            B(h, i):=B(h-1, 2i-1)+B(h-1, 2i);
        else stay idle
    }
}
```

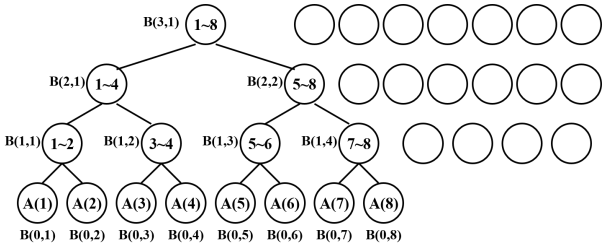


Figure 19: The summation problem $B(h,i) = B(h-1, 2i-1) + B(h-1, 2i)$

$T(n) = \log n + 2$

2.2 Work-Depth (WD) Presentation

```
for(P_i, 1<=i<=n)pardo
    B(0, i):=A(i);
for(h=1 to logn)
    for(P_i, 1<=i<=n/(2^h))pardo{
        B(h, i):=B(h-1, 2i-1)+B(h-1, 2i);
    }
```

3 Measuring the performance

- Work load –total number of operations: $W(n)$
- Worst-case running time: $T(n)$

All asymptotically equivalent:

1) $W(n)$ operations and $T(n)$ time
2) $P(n) = \frac{W(n)}{T(n)}$ processors and $T(n)$ time (on a PRAM)
3) $\frac{W(n)}{p}$ time using any number of $p \leq \frac{W(n)}{T(n)}$ processors (on a PRAM)
4) $\frac{W(n)}{p} + T(n)$ time using any number of p processors (on a PRAM)

4 Prefix-Sums

Input: $A(1), A(2), \ldots, A(n)$
Output: $\sum_{i=1}^{1} A(i), \sum_{i=1}^{2} A(i), \ldots, \sum_{i=1}^{n} A(i)$
Technique: Balanced Binary Trees

$$C(h,i) = \sum_{k=1}^{\alpha} A(k)$$

where $(0, \alpha)$ is the rightmost descendant leaf of node (h,i).

```
if(i==1)C(h,i):=B(h,i);
else if(i%2==0)C(h,i):=C(h+1,i/2);
else if(i%2==1)C(h,i):=C(h+1,(i-1)/2)+B(h,i);
```

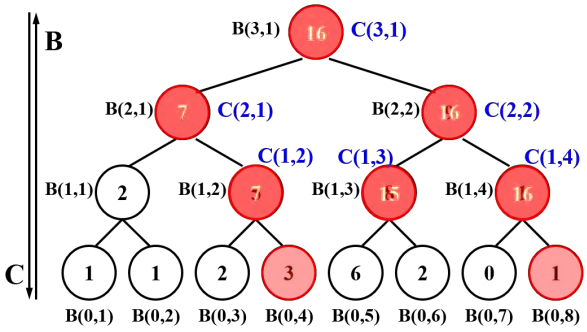


Figure 20: Prefix-Sums

5 Merging

Merge two non-decreasing arrays $A(1), A(2), \ldots, A(n)$ and $B(1), B(2), \ldots, B(m)$ into another non-decreasing array $C(1), C(2), \ldots, C(n+m)$.

Technique: Partitioning

To simplify, assume:

1) the elements of A and B are pairwise distinct
2) $n = m$
3) both $\log n$ and $n/\log n$ are integers

5.1 Parallel Ranking

The ranking problem, denoted RANK(A,B) is to compute:

1) $RANK(i, B)$ for every $1 \le i \le n$, and
2) $RANK(i, A)$ for every $1 \le i \le n$

$$\begin{aligned} RANK(j, A) &= i \quad \text{if } A(i) < B(j) < A(i+1), i \in [1, n) \\ RANK(j, A) &= 0 \quad \text{if } B(j) < A(1) \\ RANK(j, A) &= n \quad \text{if } B(j) > A(n) \end{aligned}$$

Claim 5.1 *Given a solution to the ranking problem, the merging problem can be solved in $O(1)$ time and $O(n+m)$ work.*

```
1 for(Pi, 1<=i<=n)pardo
2     C(i+RANK(i,B)):=A(i)
3 for(Pi, 1<=i<=n)pardo
4     C(i+RANK(i,A)):=B(i)
```

1) Partitioning: $p = \frac{n}{\log n}$

$$\begin{aligned} A_{Select}(i) &= A(1 + (i-1)\log n) \quad i \in [1, p] \\ B_{Select}(i) &= B(1 + (i-1)\log n) \quad i \in [1, p] \end{aligned}$$

Compute $RANK$ for each selected element.

$T = O(\log n)$, $W = O(p \log n) = O(n)$

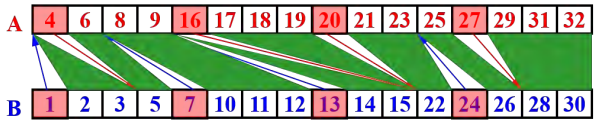


Figure 21: Partitioning

2) Actual Ranking: At most $2p$ smaller sized ($O(\log n)$) problems.

$T = O(\log n)$, $W = O(p \log n) = O(n)$

$T = O(\log n)$, $W = O(n)$

6 Maximum Finding

Replace "+" by "max" in the summation algorithm.

6.1 Compare all pairs

```
1 for(Pi, 1<=i<=n)pardo
2     B(i):=0;
3 for(i and j, 1<=i,j<=n)pardo
4     if((A(i)<A(j))||((A(i)=A(j))&&(i<j)))
5         B(i)=1;
6     else B(j)=1;
7 for(Pi, 1<=i<=n)pardo
8     if(B(i)==0)A(i) is a maximum in A;
```

Using common CRCW to resolve access conflicts.

$T(n) = O(1)$, $W(n) = O(n^2)$

6.2 A Doubly-logarithmic Paradigm

Assume that $h = \log\log n$ is an integer ($n = 2^{2^h}$).

Partition by $\sqrt{n}$:

$$\begin{aligned} A_1 &= A(1), \ldots, A(\sqrt{n}) && \Rightarrow M_1 \sim T(\sqrt{n}), W(\sqrt{n}) \\ A_2 &= A(\sqrt{n}+1), \ldots, A(2\sqrt{n}) && \Rightarrow M_2 \sim T(\sqrt{n}), W(\sqrt{n}) \\ & \ldots \\ A_{\sqrt{n}} &= A(n-\sqrt{n}+1), \ldots, A(n) && \Rightarrow M_{\sqrt{n}} \sim T(\sqrt{n}), W(\sqrt{n}) \end{aligned}$$

$$M_1, \ldots, M_{\sqrt{n}} \Rightarrow A_{\max} \sim \begin{array}{l} T = O(1) \\ W = O(\sqrt{n}^2) = O(n) \end{array}$$

$$\begin{cases} T(n) \le T(\sqrt{n}) + c_1 \\ W(n) \le \sqrt{n} W(\sqrt{n}) + c_2 n \end{cases} \Rightarrow \begin{array}{l} T(n) = O(\log\log n) \\ W(n) = O(n \log\log n) \end{array}$$

Partition by $h = \log\log n$:

$$\begin{aligned} A_1 &= A(1), \ldots, A(h) && \Rightarrow M_1 \sim O(h) \\ A_2 &= A(h+1), \ldots, A(2h) && \Rightarrow M_2 \sim O(h) \\ & \ldots \\ A_{n/h} &= A(n-h+1), \ldots, A(n) && \Rightarrow M_{n/h} \sim O(h) \end{aligned}$$

$$M_1, \ldots, M_{n/h} \Rightarrow A_{\max}$$

$T(n) = O(h + \log\log(n/h)) = O(\log\log n)$
$W(n) = O(h(n/h) + (n/h)\log\log(n/h)) = O(n)$

6.3 Random Sampling

with very high probability, on an Arbitrary CRCW PRAM

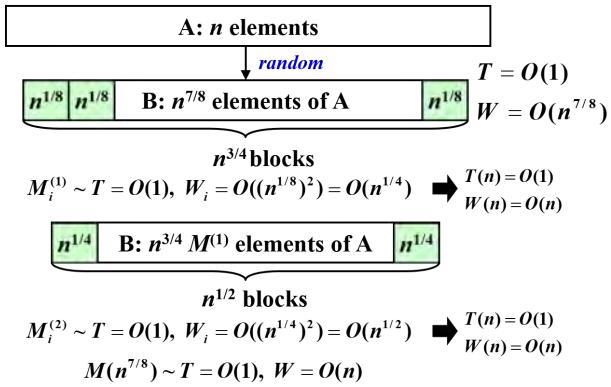


Figure 22: Random Sampling

Theorem XIV.1 *The algorithm finds the maximum among n elements. With very high probability it runs in $O(1)$ time and $O(n)$ work. The probability of not finishing within this time and work complexity is $O(1/n^c)$ for some positive constant c.*

XV External Sorting

摸了