4. backtracking Algorithms pruning

main idea: exhaustive search + elimination (pruning)

suppose now we have a partial solution $(x_1, \cdots, x_i)$

First we add $x_{i+1} \in S_{i+1}$ and check: if $(x_1, \cdots x_i, x_{i+1})$ satisfies the constrains. If the answer is "yes" we continue to add the next x_i, else we delete x_i and backtracking to the previous partial solution $(x_1, \cdots x_{i-1})$

typical problem:

I. eight queens

an abstract concept. The game tree is just

Step 1: construct a game tree. (No tree is actually constructed.

Step 2: perform a depth-first search (post-order traversal) to examine the paths

II. turnpike reconstruction problem

```
bool backtracking (int i)
{   Found = false;
    if(i>n)
        return true;
    for(each x_i ∈ S_i) {
        Ok = Check((x_1,...x_i), R);
        if(Ok) {
            Count x_i in;
            Found = Backtracking(i+1);
            if(!Found)
                Undo(i);
        }
        if(Found) break;
    }
    return Found;
}
```

III. game tree

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

α pruning β pruning

Number of passes = 1+3

$= 1 + \lceil \log_2(N/M) \rceil$

N total M memory

If the number of runs is a Fibonacci number F_N, then the best way to distribute them is to split them into F_{N-1} and F_{N-2}

In general, for a k-way merge we need 2k input buffers and 2 output buffers for parallel operation

Replacement selection —— a longer run

Parallel: Maximum Finding: compare to $\sqrt{n}$, choose $\log\log n$

Parallel Ranking: can only reduce $W(n)$ from $n\log\log n$ to n

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

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

$\sqrt{n}$: $T(n) = O(\log\log n)$ $W(n) = O(n\log\log n)$

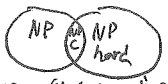
$\log\log n$: $T(n) = O(\log\log n)$ $W(n) = O(n)$

Random Sampling $T(n) = O(1)$ $W(n) = O(n)$ Bin Pack NP hard

K-way merge: require 2k tapes

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

$\rho(n)$ approximation

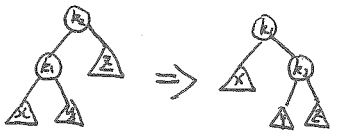


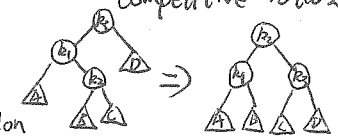
NP: 能在多项式时间内马全证一个解是否满足的一类问题

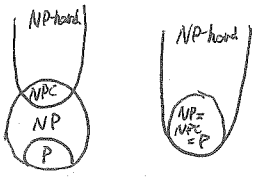
NP-hard: 不能在多项式时间内马全证

general-TSP 多项式时间可解需要 P = NP

Competitive ratio 是 online 相对于 offline 的表现







NP ≠ P

skip List:

search, insert O(logn)

HW1	DBD
HW2	BC ADC
HW3	TFFBC
HW4	TTDDBA
HW5	FTBDDC
HW7	ACAA
HW8	D
HW9	TF
HW10	FTT1C
HW11	TTCC
HW12	FFT?
HW13	FFC?
HW14	TTTTC
HW15	TTBD
pj1	AVL, Splay
pj2	mini Engine
pj3	shortest with heap
pj4	Best peak shape
pj5	Huffman code
pj6	texture packing
pj7	skip list
pj8	map reduce

```
AVL
static Position
SingleRotateWithLeft (Position K2)
{
    Position K1;
    K1 = K2->Left;
    K2->Left = K1->Right;
    K1->Right = K2;
    K2->Height = Max(Height(K2->Left),
                     Height(K2->Right)) + 1;
    K1->Height = Max(Height(K1->Left), K2->Height) + 1;
    return K1;
}
```

```
static Position
DoubleRotateWithLeft(Position K3)
{
    K3->Left = SingleRotateWithRight(K3->Left);
    return SingleRotateWithLeft(K3);
}
```

```
BinQueue Merge(BinQueue H1, BinQueue H2)
{   BinTree T1, T2, Carry = NULL;
    int i, j;
    if(H1->CurrentSize + H2->CurrentSize > Capacity) ErrorMessage();
    H1->CurrentSize += H2->CurrentSize;
    for(i=0, j=1; j<=H1->CurrentSize; i++, j*=2) {
        T1 = H1->TheTrees[i]; T2 = H2->TheTrees[i];
        switch(4*!!Carry + 2*!!T2 + !!T1) {
        000  case 0:
        001  case 1: break;
        010  case 2: H1->TheTrees[i] = T2; H2->TheTrees[i] = NULL; break;
        100  case 4: H1->TheTrees[i] = Carry; Carry = NULL; break;
        011  case 3: Carry = CombineTrees(T1, T2);
                     H1->TheTrees[i] = H2->TheTrees[i] = NULL; break;
        101  case 5: Carry = CombineTrees(T1, Carry);
                     H1->TheTrees[i] = NULL; break;
        110  case 6: Carry = CombineTrees(T2, Carry);
                     H2->TheTrees[i] = NULL; break;
        111  case 7: H1->TheTrees[i] = Carry;
                     Carry = CombineTrees(T1, T2);
                     H2->TheTrees[i] = NULL; break;
        }
    }
    return H1;
}
```

```
PriorityQueue Merge(PriorityQueue H1, PriorityQueue H2)
{
    if(H1==NULL) return H2;
    if(H2==NULL) return H1;
    if(H1->Element < H2->Element) return Merge(H1, H2);
    else return Merge1(H2, H1);
}
static PriorityQueue
Merge1(PriorityQueue H1, PriorityQueue H2)
{
    if(H1->Left == NULL)
        H1->Left = H2;
    else {
        H1->Right = Merge(H1->Right, H2);
        if(H1->Left->Npl < H1->Right->Npl)
            SwapChildren(H1);
        H1->Npl = H1->Right->Npl + 1;
    }
    return H1;
}
```