Các mô hình giải bài

LẬP TRÌNH THI ĐẤU

Đỗ Phan Thuận

Bộ môn Khoa Học Máy Tính, Viện CNTT & TT,
Trường Đại Học Bách Khoa Hà Nội.

Ngày 9 tháng 10 năm 2015

Nội dung

- Các mô hình giải bài
- Duyệt toàn bộ
- Quay lui
- Chia để trị

Bài toán ví dụ

- Problem C trong NWERC 2006: Pie

Các mô hình giải bài

- Mô hình giải bài là gì?
- Là một phương pháp xây dựng bài giải cho một loại bài toán riêng biệt
- Một ví dụ là "Duyệt trâu (Brute force)"
  - bài toán yêu cầu tìm một đối tượng có đặc tính riêng (*loại bài toán*)
  - áp dụng mô hình "Brute force": duyệt qua tất cả các đối tượng, với mỗi đối tượng, kiểm tra xem nó có đặc tính cần tìm không, nếu có, dừng lại, nếu không, tiếp tục tìm
- Bài giảng này sẽ nói về một số mô hình giải bài thông dụng
- Mỗi mô hình ứng dụng cho nhiều loại bài toán khác nhau

Duyệt toàn bộ

- Cho một tập hữu hạn các phần tử
- Yêu cầu tìm một phần tử trong tập thỏa mãn một số ràng buộc
  - hoặc tìm **tất cả** các phần tử trong tập thỏa mãn một số ràng buộc
- Đơn giản! Chỉ cần duyệt qua tất cả các phần tử trong tập, với mỗi phần tử thì kiểm tra xem nó có thỏa mãn các ràng buộc không
- Tất nhiên là cách này không hiệu quả...
- Nhưng nhớ là ta luôn tìm bài giải đơn giản nhất mà chạy trong giới hạn thời gian
- Duyệt toàn bộ luôn là mô hình giải bài đầu tiên bạn nên nghĩ đến khi giải một bài toán

Bài toán ví dụ: Vito's family

- http://uva.onlinejudge.org/external/100/10041.html

Duyệt toàn bộ

- Làm thế nào nếu không gian tìm kiếm của bài toán phức tạp hơn?
  - Tất cả hoán vị của n phần tử
  - Tất cả tập con của n phần tử
  - Tất cả các cách xếp n con hậu trên bàn cờ quốc tế $n \times n$ mà không có 2 con hậu nào ăn nhau
- Làm thế nào để duyệt qua không qian tìm kiếm của bài toán?
- Xem xét trực tiếp trên các ví dụ này

Duyệt kế tiếp qua các hoán vị

- Đã được cài đặt trên nhiều thư viện chuẩn:
  - next_permutation trong C++
  - itertools.permutations trong Python

```
int n = 5;
vector<int> perm(n);
for (int i = 0; i < n; i++) perm[i] = i + 1;

do {
    for (int i = 0; i < n; i++) {
        printf("%d ", perm[i]);
    }
    printf("\n");

} while (next_permutation(perm.begin(), perm.end()));
```

Duyệt kế tiếp qua các hoán vị

- Trên Python còn đơn giản hơn nữa...
- Lưu ý là có $n!$ hoán vị độ dài n, vì vậy chỉ có thể duyệt qua được tất cả các hoán vị nếu $n \leq 11$
  - Ngược lại thì phải cần phương pháp khác tốt hơn

Duyệt kế tiếp qua các tập con

- Còn nhớ cách biểu diễn bit các tập con?
- Mỗi số nguyên từ 0 đến $2^n - 1$ biểu diễn một tập con khác nhau của tập $\{1, 2, \dots, n\}$
- Chỉ cần duyệt kế tiếp qua các số nguyên

```
int n = 5;
for (int subset = 0; subset < (1 << n); subset++) {
    for (int i = 0; i < n; i++) {
        if ((subset & (1 << i)) != 0) {
            printf("%d ", i+1);
        }
    }
    printf("\n");
}
```


Duyệt kế tiếp qua các tập con

- Tương tự trong Python
- Nhớ là có 2^n tập con độ dài n, vì vậy thường là chỉ duyệt qua được tất cả tập con nếu $n \leq 25$
  - Ngược lại thì cần phải tìm cách giải tốt hơn

Quay lui

- Ta vừa xem 2 cách duyệt toàn bộ, tuy nhiên cả 2 cách đều khá chuyên biệt
- Sẽ rất hay khi có một mô hình chung "framework"
- Quay lui!

Quay lui

- Định nghĩa các trạng thái
  - Có một trạng thái ban đầu "rỗng"
  - Một số trạng thái bộ phận
  - Một số trạng thái là hoàn toàn
- Định nghĩa cách chuyển từ một trạng thái sang trạng thái kế tiếp
- Tư tưởng cơ bản:
  1. Bắt đầu với trạng thái rỗng
  2. Sử dụng vòng lặp để duyệt qua tất cả các trạng thái nhờ các luật chuyển
  3. Nếu trạng thái hiện tại không thỏa mãn thì dừng nhánh tìm kiếm hiện tại
  4. Thực hiện toàn bộ tất cả các trạng thái cần tìm

Quay lui

- Sơ đồ chung:

```
state S;
void generate() {
    if (!is_valid(S))
        return;
    if (is_complete(S))
        print(S);
    foreach (possible next move P) {
        apply move P;
        generate();
        undo move P;
    }
}
S = empty state;
generate();
```


Sinh tất cả các tập con

- Sử dụng quay lui cũng rất đơn giản:

```
const int n = 5;
bool pick[n];

void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            if (pick[i]) {
                printf("%d ", i+1);
            }
        }
        printf("\n");
    } else {
        // either pick element no. at
        pick[at] = true;
        generate(at + 1);
        // or don't pick element no. at
        pick[at] = false;
        generate(at + 1);
    }
}

generate(0);
```


Sinh tất cả các hoán vị

- Sử dụng quay lui cũng rất đơn giản:

```
const int n = 5;
int perm[n];
bool used[n];
void generate(int at) {
    if (at == n) {
        for (int i = 0; i < n; i++) {
            printf("%d ", perm[i]+1);
        }
        printf("\n");
    } else { // decide what the at-th element should be
        for (int i = 0; i < n; i++) {
            if (!used[i]) {
                used[i] = true;  perm[at] = i;
                generate(at + 1);
                // remember to undo the move:
                used[i] = false;
            }
        }
    }
}
memset(used, 0, n);
generate(0);
```

n hậu

- Cho n con hậu và bàn cờ quốc tế $n \times n$, hãy tìm tất cả các cách để xếp n con hậu lên bàn cờ sao cho không có 2 con hậu nào ăn nhau
- Đây là một loại tập chuyên biệt để duyệt qua, vì vậy không tìm được trong thư viện chuẩn
- Cũng có thể sử dụng mẹo dùng bit để duyệt qua tất cả các tập con kích thước n của $n \times n$ ô, nhưng cách này rất chậm
- Hãy sử dụng quay lui

n hậu

- Duyệt qua các ô theo thứ tự tăng dần
- Bước chuyển là chọn hoặc không chọn đặt con hậu lên ô
- Không đặt hậu nếu nó ăn quân hậu khác đã có trên bàn cờ

```
const int n = 8;
bool has_queen[n][n];
int queens_left = n;

// generate function

memset(has_queen, 0, sizeof(has_queen));
generate(0, 0);
```


n hậu

```
void generate(int x, int y) {
    if (y == n) {
        generate(x+1, 0);
    } else if (x == n) {
        if (queens_left == 0) {
            for (int i = 0; i < n; i++) {
                for (int j = 0; j < n; j++) {
                    printf("%c", has_queen[i][j] ? 'Q' : '.');
                }
                printf("\n");
            }
        }
    } else {
        if (queens_left > 0 and no queen can attack cell (x,y)) {
            // try putting a queen on this cell
            has_queen[x][y] = true;  queens_left--;
            generate(x, y+1);
            // undo the move
            has_queen[x][y] = false; queens_left++;
        }
        // try leaving this cell empty
        generate(x, y+1);
    }
}
```


Bài toán ví dụ: The Hamming Distance Problem

- http://uva.onlinejudge.org/external/7/729.html

Chia để trị

- Cho một bài toán, tư tưởng cơ bản là
  1. chia bài toán thành một hoặc nhiều bài toán con
  2. giải đệ quy mỗi bài toán con
  3. kết hợp lời giải các bài toán con thành lời giải bài toán ban đầu
- Một số thuật toán chia để trị chuẩn:
  - Quicksort
  - Mergesort
  - Thuật toán Karatsuba
  - Thuật toán Strassen
  - Rất nhiều thuật toán trong tính toán hình học
    - ★ Bao lồi (Convex hull)
    - ★ Cặp điểm gần nhất (Closest pair of points)

Chia để trị: Độ phức tạp thuật toán

```
void solve(int n) {
    if (n == 0)
        return;

    solve(n/2);
    solve(n/2);

    for (int i = 0; i < n; i++) {
        // some constant time operations
    }
}
```

- Độ phức tạp tính toán của thuật toán chia để trị?
- Thường mô tả độ phức tạp tính toán bằng công thức đệ quy:
  - $T(n) = 2T(n/2) + n$

Chia để trị: Độ phức tạp thuật toán

- Nhưng làm thế nào để giải được công thức đệ quy này?
- Thường đơn giản nhất là sử dụng định lý thợ để giải
  - Định lý thợ cho phép đưa ra lời giải cho công thức đệ quy dạng $T(n) = aT(n/b) + f(n)$ theo ký pháp hàm tiệm cận
  - Đa phần các thuật toán chia để trị thông dụng có công thức đệ quy theo form này
- Định lý thợ cho biết $T(n) = 2T(n/2) + n$ có thời gian tính $O(n \log n)$
- Nên thuộc định lý thợ

Giảm để trị (Decrease and conquer)

- Đôi khi không cần chia bài toàn thành nhiều bài toán con, mà chỉ giảm về một bài toán con kích thước nhỏ hơn
- Thường gọi là Giảm để trị
- Ví dụ thông dụng nhất là tìm kiếm nhị phân

Tìm kiếm nhị phân

- Cho một mảng các phần tử **đã được sắp xếp**, hãy kiểm tra xem mảng có chứa phần tử x không
- Thuật toán:
  1. Trường hợp biên: mảng rỗng, trả lời KHÔNG
  2. So sánh x với phần tử ở vị trí giữa mảng
  3. Nếu bằng, tìm thấy x và trả lời CÓ
  4. Nếu nhỏ hơn, x chắc chắn nằm bên nửa trái mảng
     1. Tìm kiếm nhị phân (đệ quy) tiếp nửa trái mảng
  5. Nếu lớn hơn, x chắc chắn nằm bên nửa phải mảng
     1. Tìm kiếm nhị phân (đệ quy) tiếp nửa phải mảng

Tìm kiếm nhị phân

```
bool binary_search(const vector<int> &arr, int lo, int hi, int x) {
    if (lo > hi) {
        return false;
    }

    int m = (lo + hi) / 2;
    if (arr[m] == x) {
        return true;
    } else if (x < arr[m]) {
        return binary_search(arr, lo, m - 1, x);
    } else if (x > arr[m]) {
        return binary_search(arr, m + 1, hi, x);
    }
}

binary_search(arr, 0, arr.size() - 1, x);
```

- $T(n) = T(n/2) + 1$
- $O(\log n)$

Tìm kiếm nhị phân trên các số nguyên

- Đây có lẽ là ứng dụng phổ biến nhất của tìm kiếm nhị phân
- Cụ thể, cho hàm $p : \{0, \ldots, n-1\} \rightarrow \{T, F\}$ thỏa mãn nếu $p(i) = T$, thì $p(j) = T$ với mọi $j > i$
- Yêu cầu tìm chỉ số j nhỏ nhất sao cho $p(j) = T$ càng nhanh càng tốt

i	0	1	$\cdots$	$j-1$	j	$j+1$	$\cdots$	$n-2$	$n-1$
$p(i)$	F	F	$\cdots$	F	T	T	$\cdots$	T	T

- Có thể thực hiện trong $O(\log(n) \times f)$, với f là giá của việc đánh giá hàm p

Tìm kiếm nhị phân trên các số nguyên

```
int lo = 0,
    hi = n - 1;

while (lo < hi) {
    int m = (lo + hi) / 2;

    if (p(m)) {
        hi = m;
    } else {
        lo = m + 1;
    }
}

if (lo == hi && p(lo)) {
    printf("lowest index is %d\n", lo);
} else {
    printf("no such index\n");
}
```

Tìm kiếm nhị phân trên các số nguyên

- Tìm vị trí của x trong mảng đã sắp xếp *arr*

```
bool p(int i) {
    return arr[i] >= x;
}
```

- Cách sử dụng khác ở phần sau

Tìm kiếm nhị phân trên các số thực

- Đây là phiên bản tổng quát hơn của tìm kiếm nhị phân
- Cho hàm $p : [lo, hi] \rightarrow \{T, F\}$ thỏa mãn nếu $p(i) = T$, thì $p(j) = T$ for all $j > i$
- Yêu cầu tìm số thực nhỏ nhất j sao cho $p(j) = T$ càng nhanh càng tốt
- Do làm việc với số thực, khoảng $[lo, hi]$ có thể bị chia vô hạn lần mà không dừng ở một số thực cụ thể
- Thay vào đó có thể tìm một số thực j' rất sát với lời giải đúng j, sai số trong khoảng $EPS = 2^{-30}$
- Có thể làm được trong thời gian $O(\log(\frac{hi-lo}{EPS}))$ tương tự cách làm tìm kiếm nhị phân trên mảng

Tìm kiếm nhị phân trên các số thực

```
double EPS = 1e-10,
       lo = -1000.0,
       hi = 1000.0;

while (hi - lo > EPS) {
    double mid = (lo + hi) / 2.0;

    if (p(mid)) {
        hi = mid;
    } else {
        lo = mid;
    }
}

printf("%0.10lf\n", lo);
```


Tìm kiếm nhị phân trên các số thực

- Có nhiều ứng dụng thú vị
- Tìm căn bậc hai của x

```
bool p(double j) {
    return j*j >= x;
}
```

- Tìm nghiệm của hàm $f(x)$

```
bool p(double x) {
    return f(x) >= 0.0;
}
```

- Đây cũng được gọi là phương pháp chia đôi trong phương pháp tính (Bisection method)

Bài toán ví dụ

- Problem C from NWERC 2006: Pie

Tìm kiếm nhị phân câu trả lời

- Có thể khó tìm ra lời giải tối ưu một cách trực tiếp, như thấy trong bài toán ví dụ trên
- Mặt khác, dễ dàng kiểm tra một số x nào đó có phải là lời giải không
- Có phương pháp sử dụng tìm kiếm nhị phân để tìm lời giải nhỏ nhất hoặc lớn nhất của một bài toán
- Chỉ áp dụng được khi bài toán có tính chất tìm kiếm nhị phân: nếu i là một lời giải, thì tất cả $j > i$ cũng là lời giải
- $p(i)$ kiểm tra nếu i là một lời giải, thì có thể áp dụng một cách đơn giản tìm kiếm nhị phân trên p để nhận được lời giải nhỏ nhất hoặc lớn nhất

Các loại chia để trị khác

- Tìm kiếm nhị phân rất hữu ích, có thể dùng để xây dựng các bài giải đơn giản và hiệu quả
- Tuy nhiên tìm kiếm nhị phân là chỉ là một ví dụ của chia để trị
- Hãy xem tiếp 2 ví dụ nữa

Nhị phân hàm mũ (Binary exponentiation)

- Yêu cầu tính x^n, với x, n là các số nguyên
- Giả thiết ta không có phương thức `pow`
- Phương pháp ngây thơ:

```
int pow(int x, int n) {
    int res = 1;
    for (int i = 0; i < n; i++) {
        res = res * x;
    }

    return res;
}
```

- Độ phức tạp $O(n)$, tuy nhiên với n lớn thì sao?

Nhị phân hàm mũ

- Hãy sử dụng chia để trị
- Để ý 3 đẳng thức sau:
  - $x^0 = 1$
  - $x^n = x \times x^{n-1}$
  - $x^n = x^{n/2} \times x^{n/2}$
- Hoặc theo ngôn ngữ hàm:
  - $pow(x, 0) = 1$
  - $pow(x, n) = x \times pow(x, n-1)$
  - $pow(x, n) = pow(x, n/2) \times pow(x, n/2)$
- $pow(x, n/2)$ được sử dụng 2 lần, nhưng ta chỉ cần tính 1 lần:
  - $pow(x, n) = pow(x, n/2)^2$

Nhị phân hàm mũ

- Hãy sử dụng các đẳng thức đó để tìm câu trả lời theo cách đệ quy

```
int pow(int x, int n) {
    if (n == 0) return 1;
    return x * pow(x, n - 1);
}
```

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$
  - $O(n)$
  - Vẫn chậm như trước...

Nhị phân hàm mũ

- Để ý đẳng thức thứ 3:
  - $n/2$ không là số nguyên khi n lẻ, vì vậy chỉ sử dụng nó khi n chẵn

```
int pow(int x, int n) {
    if (n == 0) return 1;
    if (n % 2 != 0) return x * pow(x, n - 1);
    int st = pow(x, n/2);
    return st * st;
}
```

- Độ phức tạp?
  - $T(n) = 1 + T(n-1)$ nếu n lẻ
  - $T(n) = 1 + T(n/2)$ nếu n chẵn
  - Do $n - 1$ chẵn khi n lẻ:
  - $T(n) = 1 + 1 + T((n-1)/2)$ nếu n lẻ
  - $O(\log n)$
  - Nhanh!

Nhị phân hàm mũ

- Để ý là x không nhất thiết là số nguyên và $\star$ không nhất thiết là phép nhân số nguyên...
- Cũng dùng được cho:
  - Tính x^n, với x là số thực và $\star$ là phép nhân số thưc
  - Tính A^n, với A là một ma trận và $\star$ là phép nhân ma trận
  - Tính $x^n \pmod{m}$, với x là một số nguyên và $\star$ là phép nhân số nguyên lấy mod m
  - Tính $x \star x \star \cdots \star x$, với x là bất kỳ loại phần tử gì và $\star$ là một toán tử phù hợp
- Tất cả có thể giải trong $O(\log(n) \times f)$, với f là giá để thực hiện một toán tử $\star$

Số Fibonacci

- Nhắc lại dãy Fibonacci được định nghĩa như sau:
  - $\mathrm{fib}_1 = 1$
  - $\mathrm{fib}_2 = 1$
  - $\mathrm{fib}_n = \mathrm{fib}_{n-2} + \mathrm{fib}_{n-1}$
- Ta có dãy $1, 1, 2, 3, 5, 8, 13, 21, \ldots$
- Có rất nhiều biến thể của dãy Fibonacci
- Một kiểu là cùng công thức nhưng bắt đầu bởi các số khác, ví dụ:
  - $f_1 = 5$
  - $f_2 = 4$
  - $f_n = f_{n-2} + f_{n-1}$
- Ta có dãy $5, 4, 9, 13, 22, 35, 57, \ldots$
- Với những cái khác số thì sao?

Số Fibonacci

- Thử với một cặp xâu, và đặt $+$ là phép toán ghép xâu:
  - $g_1 = A$
  - $g_2 = B$
  - $g_n = g_{n-2} + g_{n-1}$
- Ta thu được dãy các xâu:
  - A
  - B
  - AB
  - BAB
  - $ABBAB$
  - $BABABBAB$
  - $ABBABBABABBAB$
  - $BABABBABABBABBABABBAB$
  - $\ldots$

Số Fibonacci

- g_n dài bao nhiêu?
  - $\mathrm{len}(g_1) = 1$
  - $\mathrm{len}(g_2) = 1$
  - $\mathrm{len}(g_n) = \mathrm{len}(g_{n-2}) + \mathrm{len}(g_{n-1})$
- Trông quen thuộc?
- $\mathrm{len}(g_n) = \mathrm{fib}_n$
- Vì vậy các xâu trở nên rất lớn rất nhanh
  - $\mathrm{len}(g_{10}) = 55$
  - $\mathrm{len}(g_{100}) = 354224848179261915075$
  - $\mathrm{len}(g_{1000}) =$

    434665576869374564356885276750406258025646605173717
    804024817290895365554179490518904038798400792551692
    959225930803226347752096896232398733224711616429964
    409065331879382989696499285160037044761377951668492
    28875

Số Fibonacci

- Nhiệm vụ: Hãy tính ký tự thứ i trong g_n
- Dễ dàng thực hiện trong $O(\mathrm{len}(n))$, nhưng sẽ cực kỳ chậm với n lớn
- Có thể giải trong $O(n)$ sử dụng chia để trị