Força Bruta / Backtracking

André Gustavo dos Santos

Departamento de Informática
Universidade Federal de Viçosa

INF 492 - 2012/1

Introdução

- Computadores atuais são tão rápidos que alguns problemas podem ser resolvidos na força bruta (busca exaustiva, testar todas as alternativas)
- Contar os elementos de um conjunto, por exemplo. Em alguns casos é mais fácil gerar todos eles do que encontrar uma fórmula por análise combinatória
- É claro, isso só será possível se o número de itens for suficientemente pequeno para ter tempo de gerar e contar todos eles

Introdução

Exemplo: UVa 725 - Division

Encontrar e escrever todos os pares de inteiros de 5 dígitos que usam todos os dígitos de 0 a 9 (exatamente uma vez cada), de tal forma que o primeiro dividido pelo segundo seja igual a N, ($2 \leq N \leq 79$).

Isto é, encontrar todos `abcde / fghij` = N, sendo cada letra um dígito diferente (podem começar com 0).

Ex.: Para $N = 62$, temos 79546 / 01283 = 62 e 94736 / 01528 = 62.

Solução 1

- Verificar todas as permutações dos dígitos em `abcdefghij`
- Quantas verificações? $10! = 3.628.800$

Introdução

Exemplo: UVa 725 - Division

Encontrar e escrever todos os pares de inteiros de 5 dígitos que usam todos os dígitos de 0 a 9 (exatamente uma vez cada), de tal forma que o primeiro dividido pelo segundo seja igual a N, ($2 \leq N \leq 79$).

Isto é, encontrar todos `abcde / fghij` = N, sendo cada letra um dígito diferente (podem começar com 0).

Ex.: Para $N = 62$, temos 79546 / 01283 = 62 e 94736 / 01528 = 62.

Solução 2

- Testar `fghij` = 01234 até 98765
- Para cada um, multiplicar por N e verificar se os dígitos são diferentes
- Quantas verificações? < 100.000

Introdução

- Computadores atuais tem clock na faixa de Gigahertz
- Então eles podem realizar bilhões de instruções por segundo
- Sabendo que qualquer cálculo interessante gasta 100 ou mais instruções, será possível enumerar ou contar um conjunto contendo milhões de elementos

- O que significa 1 milhão de elementos?
  - todas as possíveis permutações de 10 itens (são 3.628.800)
  - todas os possíveis subconjuntos de 20 itens (são 1.048.576)

- Para problemas maiores que esses, é necessário um esquema mais inteligente, que verifique somente os elementos que realmente interessam

Backtracking

É um método para gerar/enumerar/percorrer sistematicamente todas as alternativas em um espaço de soluções.

É uma técnica geral que deve ser adaptada para cada aplicação.

- Aplicado quando for possível trabalhar com uma *solução parcial*
- O método constrói soluções candidatas incrementalmente e abandona uma solução parcial quando identifica que tal solução parcial não levará a uma solução do problema
- Em cada passo do processo de enumeração são tentadas todas as possíveis alternativas recursivamente
- É um método de tentativa e erro: tenta um caminho; se não der certo, volta e tenta outro, marcando os caminhos por onde passa

Backtracking

Algoritmo geral

- Dada uma solução parcial C com elementos candidatos $c_1, c_2, \ldots, c_k$
- Se é uma solução
  - contar, imprimir, comparar, etc
- Se não é, verificar se pode adicionar mais um elemento
  - Se puder, adicionar c_{k+1} em C e continuar recursivamente
  - Apagar o último c_{k+1} adicionado e tentar outro iterativamente

- Método correto: verifica todas as alternativas
- Método eficiente: não testa a mesma alternativa mais de uma vez
- mas... só deve ser usado se realmente é necessário avaliar todas as alternativas

Backtracking - exemplos de aplicações

- Quais são todos os subconjuntos de $\{2, 5, 8, 9, 13\}$?
- Em quantos destes a soma dos elementos é ≤ 17 ?
- Quais são todas as permutações de $[2, 5, 8, 9, 13]$?
- Em quantas destas os itens 8 e 9 não aparecem juntos ?
- Dado um labirinto, ache um caminho para sair dele
- Colocar 8 rainhas num tabuleiro de xadrez sem que nenhuma ataque outra
- Resolver um quebra-cabeças com várias configurações e apenas uma (ou algumas) correta
- ...

Backtracking

- Algoritmo geral

```
// int c[] - solução parcial, de c[0] até c[k-1]
// int k - posição para inserir o próximo passo/candidato
// int n - tamanho máximo da solução

void backtrack(int c[], int k, int n)
{
    if ( ## terminou ##) {          // se chegou numa possível solução
        ## processa solucao ##;
        return;
    }

    for ( ## toda alternativa ## )          // para toda alternativa
        if ( ## alternativa viável ## ) { // se pode incluí-la
            c[k] = ## alternativa ##;     // inclui na solução parcial
            backtrack(c, k+1, n);         // gera o restante
        }
}
```

Backtracking - gerar todos os subconjuntos

```
void gerasub(bool c[], int k, int n)
{
    if (k == n) {          // se terminou de gerar um subconjunto
        imprimesub(c, n);
        return;
    }

    c[k] = true;           // subconjuntos com o elemento k
    gerasub(c, k+1, n);
    c[k] = false;          // subconjuntos sem o elemento k
    gerasub(c, k+1, n);
}

int main()
{
    bool c[1000];          // subconjunto parcial
    int n;                 // tamanho do conjunto

    cin >> n;
    gerasub(c, 0, n);    // subconjuntos de {0, 1, 2, 3, ..., n-1}
}
```

Backtracking - gerar todas as permutações

```
void geraperm(int c[], int k, int n)
{
    if (k == n) {           // se terminou de gerar a permutação
        imprimeperm(c, n);
        return;
    }

    for(int i=0;i<n;i++)            // para todo elemento i
        if (!pertence(c,k,i)) {     // se i não está na permutação parcial
            c[k] = i;               // inclui o elemento na permutação
            geraperm(c, k+1, n);    // gera o restante da permutação
        }
}

int main()
{
    int c[1000];            // permutação parcial
    int n;                  // tamanho do conjunto

    cin >> n;
    geraperm(c, 0, n);      // permutações de {0, 1, 2, 3, ..., n-1}
}
```

Backtracking - cortes

- É importante **cortar** a busca tão logo se descubra que a solução parcial não levará a uma solução aceitável do problema

```
void backtrack(int c[], int k, int n)
{
    if ( ## não há como continuar ##)      // corte
        return;

    if ( ## terminou ##) {          // se chegou numa possível solução
        ## processa solucao ##;
        return;
    }

    for ( ## toda alternativa ## )          // para toda alternativa
        if ( ## alternativa viável ## ) { // se pode incluí-la
            c[k] = ## alternativa ##;      // inclui na solução parcial
            backtrack(c, k+1, n);          // gera o restante
        }
}
```

Backtracking

- Conceitualmente, é como uma árvore de busca
- As soluções candidatas são nós da árvore
- Cada nó é pai das soluções parciais com um passo a mais
- As folhas são soluções que não podem mais crescer

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 1

- Testar todas as configurações por força bruta
- Quantas verificações? $\binom{64}{8} = 4.426.165.368$

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 2

- Cada rainha deve estar em uma linha diferente
- Então basta escolher uma coluna diferente para cada uma
- Permutações de $[1, 2, 3, 4, 5, 6, 7, 8]$
- Quantas verificações? $8! = 40.320$

Backtracking - exemplo

Exemplo: UVa 750 - 8 Queens Chess Problem

No tabuleiro de xadrez (8x8) é possível colocar 8 rainhas sem que nenhuma ataque outra.

Determine *todas* as possíveis configurações, dada a posição de uma das rainhas (ou seja, certa coordenada (a, b) deve conter uma rainha).

Solução 2b

- Além disso não podem estar na mesma diagonal
- Cortar a busca se a nova rainha está na mesma diagonal das anteriores
  - (i, j) e (k, l) estão na mesma diagonal se $abs(i - k) == abs(j - l)$

Backtracking - dicas

- A grande aposta em usar Força Bruta para resolver um problema é se passa no Time Limit
- Se TL = 1 min e seu programa executa em 1 min 5 s, você pode tentar melhorar a parte crítica do código, em vez de começar do zero um algoritmo mais rápido porém não trivial

Backtracking - dica 1

Gerar x Filtrar

- Programas que geram muitas soluções candidatas e então escolhe as corretas (ou remove as incorretas) são chamados “filtros”
- Programas que geram exatamente as respostas corretas sem começos falsos são chamados “geradores”
- Geralmente programas “filtros” são mais fáceis de codificar, porém mais lentos
- Faça os cálculos para ver se um programa “filtro” dá conta ou se precisa escrever um “gerador”

Backtracking - dica 1

Gerar x Filtrar

- Programas que geram muitas soluções candidatas e então escolhe as corretas (ou remove as incorretas) são chamados “filtros”
- Programas que geram exatamente as respostas corretas sem começos falsos são chamados “geradores”
- Geralmente programas “filtros” são mais fáceis de codificar, porém mais lentos
- Faça os cálculos para ver se um programa “filtro” dá conta ou se precisa escrever um “gerador”

Backtracking - dica 1

Gerar x Filtrar

- Programas que geram muitas soluções candidatas e então escolhe as corretas (ou remove as incorretas) são chamados “filtros”
- Programas que geram exatamente as respostas corretas sem começos falsos são chamados “geradores”
- Geralmente programas “filtros” são mais fáceis de codificar, porém mais lentos
- Faça os cálculos para ver se um programa “filtro” dá conta ou se precisa escrever um “gerador”

Backtracking - dica 1

Gerar x Filtrar

- Programas que geram muitas soluções candidatas e então escolhe as corretas (ou remove as incorretas) são chamados “filtros”
- Programas que geram exatamente as respostas corretas sem começos falsos são chamados “geradores”
- Geralmente programas “filtros” são mais fáceis de codificar, porém mais lentos
- Faça os cálculos para ver se um programa “filtro” dá conta ou se precisa escrever um “gerador”

Backtracking - dica 2

Cortar espaço de busca inviável mais cedo

- Verificar se uma solução parcial pode levar a uma solução completa
- Se não puder, cortar (retornar e tentar outra)

Backtracking - dica 3

Usar simetrias

- Alguns problemas possuem simetrias
- Usar esse fato para reduzir o tempo de execução

Backtracking - dica 4

Pré-processamento

- Pode ser útil gerar tabelas ou outras estruturas de dados que permitam buscar o resultado de forma mais rápida
- Antes de iniciar propriamente o programa, gerar todos (ou boa parte) desses dados
- Uso de memória/espaço em vez de tempo

Backtracking - dica 5

Tentar resolver de “trás pra frente”

- Curiosamente, alguns problemas são melhor resolvidos pensando ao contrário, ou começando do fim para o início
- Pode ser útil processar os dados em outra ordem que não a óbvia
- Ex.: UVa 10360 - Rat Attack

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 6

Otimizar o código

- Entender como os cálculos são feitos em hardware pode ajudar a programar melhor
- Isso inclui entrada/saída, comportamento de cache, entre outros

1. Usar printf/scanf em vez de cin/cout
2. Usar quicksort em vez de mergesort
3. Acessar matriz linha por linha em vez de coluna por coluna
4. Manipulação de bits é mais rápida que usar array de booleanos
5. Usar <bitset> em vez de vector<bool> para o Crivo de Eratostenes
6. Declarar variáveis compostas gigantes uma única vez, com escopo global
7. Alocar memória uma única vez (com os limites fornecidos no enunciado) em vez de realocar dinamicamente par cada caso de teste

Backtracking - dica 7

Usar estruturas de dados e algoritmos melhores!

- Sério, isso pode ser melhor que todas as outras 6 dicas

Backtracking - dica 8

:-(

- Se tudo isso falhar, abandonar Foça Bruta...