Projeto de algoritmos

Vamos aprender várias técnicas de projeto de algoritmos que são fundamentais na vida de um cientista da computação. Os objetivos ao estudar este assunto são:

- Aprender os principais paradigmas de projeto de algoritmos;
- Identificar a estrutura ou propriedades de um problema e a partir disso definir princípios de projeto que podem ser usados para resolvê-lo;
- Ilustrar o impacto de certas escolhas de projeto na eficiência de uma solução.

É importante deixar claro que não existe uma bala de prata (silver bullet) no projeto de algoritmos, ou seja, não existe uma técnica universal que pode resolver qualquer problema computacional que você irá encontrar. Mas há vários paradigmas de projeto de algoritmos que podem ajudar você a resolver problemas de muitos domínios de aplicação diferentes.

A maioria dos problemas que consideraremos nesta parte são problemas de otimização. Em geral, um problema desses possui um conjunto de restrições que define o que é uma solução viável e uma função objetivo que determina o valor de cada solução. O objetivo é encontrar uma solução ótima, que é uma solução viável com melhor valor de função objetivo (maximização ou minimização).

Nas próximas seções, vamos estudar os seguintes paradigmas de projeto: Força Bruta, Backtracking ou tentativa e erro, Divisão e Conquista, Algoritmos Gulosos e Programação Dinâmica. Usaremos os termos "problema" e "subproblema" para nos referir igualmente a "uma instância do problema" e "uma instância menor do problema", respectivamente.

Força bruta

Força bruta (ou busca exaustiva) é um tipo de algoritmo de uso geral que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema. Esse tipo de algoritmo geralmente possui uma implementação simples e sempre encontrará uma solução se ela existir. Entretanto, seu custo computacional é proporcional ao número de candidatos à solução que, em problemas reais, tende a crescer exponencialmente.

A força bruta é tipicamente usada em problemas cujo tamanho é limitado ou quando não se conhece um algoritmo mais eficiente. Também pode ser usada quando a simplicidade da implementação é mais importante do que a velocidade de execução, como nos casos de aplicações críticas em que os erros de algoritmo possuem sérias consequências.

Exemplo 1

Considere um conjunto P de n pessoas e uma matriz M de tamanho n × n, tal que M[i, j] = M[j, i] = 1, se as pessoas i e j se conhecem e M[i, j] = M[j, i] = 0, caso contrário.

- Problema (similar ao problema do Clique em grafos): existe um subconjunto C, de r pessoas escolhidas de P, tal que qualquer par de pessoas de C se conhecem?
- Solução de força bruta: verificar, para todas as combinações simples (sem repetições) C de r pessoas escolhidas entre as n pessoas do conjunto P, se todos os pares de pessoas de C se conhecem.

Considere um conjunto P de 8 pessoas representado pela matriz abaixo (de tamanho 8 × 8):

x	1	2	3	4	5	6	7	8
1	1	0	1	1	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	1
4	1	0	1	1	1	1	1	1
5	1	1	0	1	1	0	0	0
6	1	0	1	1	0	1	1	1
7	1	0	1	1	0	1	1	0
8	0	1	1	1	0	1	0	1

• Existe um conjunto C de 5 pessoas escolhidas de P tal que qualquer par de pessoas de C se conhecem?
• Existem 56 combinações simples de 5 elementos escolhidos dentre um conjunto de 8 elementos:

1 2 3 4 5	1 2 4 6 8	1 3 5 7 8	2 3 5 6 8
1 2 3 4 6	1 2 4 7 8	1 3 6 7 8	2 3 5 7 8
1 2 3 4 7	1 2 5 6 7	1 4 5 6 7	2 3 6 7 8
1 2 3 4 8	1 2 5 6 8	1 4 5 6 8	2 4 5 6 7
1 2 3 5 6	1 2 5 7 8	1 4 5 7 8	2 4 5 6 8
1 2 3 5 7	1 2 6 7 8	1 4 6 7 8	2 4 5 7 8
1 2 3 5 8	1 3 4 5 6	1 5 6 7 8	2 4 6 7 8
1 2 3 6 7	1 3 4 5 7	2 3 4 5 6	2 5 6 7 8
1 2 3 6 8	1 3 4 5 8	2 3 4 5 7	3 4 5 6 7
1 2 3 7 8	**1 3 4 6 7**	2 3 4 5 8	3 4 5 6 8
1 2 4 5 6	1 3 4 6 8	2 3 4 6 7	3 4 5 7 8
1 2 4 5 7	1 3 4 7 8	2 3 4 6 8	3 4 6 7 8
1 2 4 5 8	1 3 5 6 7	2 3 4 7 8	3 5 6 7 8
1 2 4 6 7	1 3 5 6 8	2 3 5 6 7	4 5 6 7 8

Para resolver esse problema, precisamos enumerar todas as combinações simples de r elementos de um conjunto de tamanho n. O código abaixo faz isso:

```c
void combinacao_simples(int n, int r, int x[], int next, int k){
    int i;
    if (k == r){
        for (i = 0; i < r; i++)
            printf("%d ", x[i] + 1);
        printf("\n");
    }
    else{
        for (i = next; i < n; i++){
            x[k] = i;
            combinacao_simples(n, r, x, i + 1, k + 1);
        }
    }
}

int main(){
    int n, r, x[1000];
    printf("Entre com o valor de n: ");
    scanf("%d", &n);
    printf("Entre com o valor de r: ");
    scanf("%d", &r);
    combinacao_simples(n, r, x, 0, 0);
    return 0;
}
```

Exemplo 2

Considere um conjunto de n cidades e uma matriz M de tamanho n × n tal que M[i, j] = 1, se existe um caminho direto entre as cidades i e j, e M[i, j] = 0, caso contrário.

- Problema (problema do Ciclo Hamiltoniano em Grafos): existe uma forma de, saindo de uma cidade qualquer, visitar todas as demais cidades, sem passar duas vezes por nenhuma cidade, e ao final retornar para a cidade inicial?
- Note que, se existe uma forma de sair de uma cidade X qualquer, visitar todas as demais cidades (sem repetir nenhuma) e depois retornar para X, então existe uma forma de fazer o mesmo para qualquer outra cidade do conjunto, já que existe um Ciclo Hamiltoniano (uma forma circular de visitar todas as cidades) e qualquer cidade do ciclo pode ser usada como ponto de partida. Como vimos, qualquer cidade pode ser escolhida como cidade inicial. Sendo assim, vamos escolher, arbitrariamente a cidade n como ponto de partida.
- Solução de força bruta: testar todas as permutações das n − 1 primeiras cidades, verificando se existe um caminho direto entre a cidade n e a primeira da permutação, assim como um caminho entre todas as cidades consecutivas da permutação e, por fim, um caminho direto entre a última cidade da permutação e a cidade n.

Considere um conjunto de 8 cidades representado pela matriz abaixo (de tamanho 8 × 8):

x	1	2	3	4	5	6	7	8
1	0	0	1	0	1	1	1	0
2	0	1	0	0	1	0	0	1
3	1	0	1	1	0	1	1	0
4	0	0	1	1	0	0	1	0
5	1	1	0	1	1	0	0	0
6	0	0	1	1	0	0	1	1
7	1	0	0	1	0	1	1	1
8	0	1	1	1	0	1	0	1

Existe uma forma de, a partir da cidade 8, visitar todas as demais cidades, sem repetir nenhuma, e ao final retornar para a cidade 8? Exitem 5040 permutações das 7 primeiras cidades da lista original. Como enumerar todas as permutações de n valores distintos? O código abaixo faz essa enumeração.

```
void permutacao(int n, int x[], int used[], int k){
    int i;
    if (k == n){
        for (i = 0; i < n; i++)
            printf("%d ", x[i] + 1);
        printf("\n");
    }
    else{
```

```
        for (i = 0; i < n; i++){
            if (!used[i]){
                used[i] = 1;
                x[k] = i;
                permutacao(n, x, used, k + 1);
                used[i] = 0;
            }
        }
    }
}

int main(){
    int i, n, x[100], used[100];
    printf("Entre com o valor de n: ");
    scanf("%d", &n);
    /* se um elemento i estiver em uso, entao used[i] = 1,
    caso contrario, used[i] = 0. */
    for (i = 0; i < n; i++)
        used[i] = 0;
    permutacao(n, x, used, 0);
    return 0;
}
```

Backtracking (Tentativa e erro)

Backtracking refere-se a um tipo de algoritmo para encontrar todas (ou algumas) soluções de um problema computacional, que incrementalmente constrói soluções candidatas e abandona uma candidata parcialmente construída tão logo quanto for possível determinar que ela não pode gerar uma solução válida.

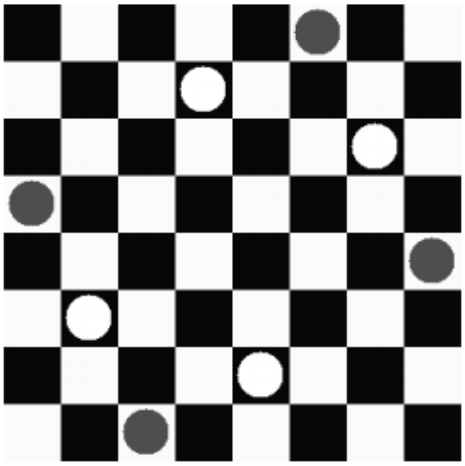
Backtracking pode ser aplicado para problemas que admitem o conceito de "solução candidata parcial" e que exista um teste relativamente rápido para verificar se uma candidata parcial pode ser completada como uma solução válida. Quando aplicável, backtracking é frequentemente muito mais rápido que algoritmos de enumeração total (força bruta), já que ele pode eliminar um grande número de soluções inválidas com um único teste.

Enquanto algoritmos de força bruta geram todas as possíveis soluções e só depois verificam se elas são válidas, backtracking só gera soluções válidas.

Alguns exemplos famosos de uso de backtracking:
- Problema das Oito Rainhas;
- Passeio do Cavalo;
- Labirinto.

Exemplo - Problema das oito rainhas

O problema das 8 rainhas é um problema famoso, cujos primeiros relatos datam do início do século XIX, estudado por Gauss e bem conhecido dos livros que tratam de combinatória. O problema consiste em dispor 8 rainhas sobre um tabuleiro de xadrez de tal modo que elas não se ataquem. No jogo de xadrez, as rainhas podem se atacar (ou se movimentar) na horizontal, na vertical e também ao longo das diagonais. Sendo assim, o objetivo é dispor as 8 rainhas de modo que elas não compartilhem linhas, colunas e nem diagonais. A figura abaixo representa uma das 92 soluções possíveis para este problema:



Espaço de busca

Testar todas possíveis posições das rainhas: Um tabuleiro 8x8 tem 64 posições. Para a primeira rainha temos 64 possibilidades, para a segunda 63, para a terceira 62, … Logo temos C64,8, ou seja, o número de combinações de 64 elementos tomados 8 a 8. C64,8 = 64!/(8!*(64-8)!) = 64!/(8!*56!) = 64*63*62*61*60*59*58*57/8! = 4426165368. Espaço de busca grande demais!
Vamos reduzir o espaço de busca sabendo que as rainhas ocupam diferentes colunas (numeradas de zero a sete).
A rainha R1 tem 8 possibilidades de linhas na coluna 0,
A rainha R2 tem 8 possibilidades de linhas na coluna 1,
…
A rainha R8 tem 8 possibilidades de linhas na coluna 7.
No total temos 8^8 = 16777216.
Vamos reduzir o espaço de busca ainda mais sabendo que as rainhas não podem ocupar a mesma linha.
A rainha R1 tem 8 possibilidades de linhas na coluna 0,
A rainha R2 tem 7 possibilidades de linhas na coluna 1,
A rainha R3 tem 6 possibilidades de linhas na coluna 2,
…

No total temos 8! = 40320, ou seja, o número de permutações de 8 elementos. Esse espaço de busca só é viável de ser testado para problemas de tamanho pequeno.

Solução

O problema das N rainhas é uma generalização do problema das 8 rainhas e consiste em dispor n rainhas (onde n > 4) sobre um tabuleiro de dimensões n × n, de tal modo que elas não se ataquem. O número de soluções existentes para cada valor de n cresce exponencialmente. Por conta disso, é necessário utilizar um algoritmo eficiente para otimizar o processo de busca pelas soluções. A técnica empregada para solucionar este tipo de problema é conhecida por Backtracking e consiste em um refinamento da busca por força bruta, em que múltiplas soluções podem ser eliminadas sem ser explicitamente examinadas.

Um procedimento sistemático de colocar as n rainhas no tabuleiro é começar pela primeira coluna e considerar as n posições disponíveis. Depois, para cada uma das posições da primeira coluna, considerar as posições da segunda coluna que são válidas (ou seja, não levam a situações de ataque). Repete-se esse processo até preencher as n colunas do tabuleiro. Se chegarmos à última coluna e for possível colocar as n rainhas, então temos uma solução para o problema. Se chegarmos a alguma coluna onde não haja nenhuma posição válida para colocar uma rainha, retornamos à coluna anterior procurando pela próxima posição válida. Aplicando esta ideia repetidamente às colunas restantes, no final obtemos o conjunto de todas as soluções possíveis.

Uma implementação utilizando a técnica de backtracking para o problema das n rainhas é dado abaixo.

```c
//Autor: Marcos Castro
#include <stdio.h>
#include <stdbool.h>

int N = 8; //número de rainhas
// conta a quantidade de soluções
int sol = 0;

// função para mostrar o tabuleiro
void mostrarTabuleiro(int tab[N][N], int N){
    for(int i = 0; i < N; i++){
        for(int j = 0; j < N; j++){
            if(tab[i][j] == 1)
                printf("R\t");
            else
                printf("-\t");
        }
        printf("\n\n");
    }
```

```
    printf("\n");
}

// verifica se é seguro colocar a rainha numa determinada coluna
bool seguro(int tab[N][N], int N, int lin, int col){
    int i, j;
    // verifica se ocorre ataque na linha
    for(i = 0; i < N; i++) {
        if(tab[lin][i] == 1)
            return false;
    }
    //verifica se ocorre ataque na coluna
    for(i = 0; i < N; i++) {
        if(tab[i][col] == 1)
            return false;
    }
    // verifica se ocorre ataque na diagonal principal
    // acima e abaixo da posição (lin,col)
    for(i = lin, j = col; i >= 0 && j >= 0; i--, j--) {
        if(tab[i][j] == 1)
            return false;
    }
    for(i = lin, j = col; i < N && j < N; i++, j++) {
        if(tab[i][j] == 1)
            return false;
    }
    // verifica se ocorre ataque na diagonal secundária
    // acima e abaixo
    for(i = lin, j = col; i >= 0 && j < N; i--, j++){
        if(tab[i][j] == 1)
            return false;
    }
    for(i = lin, j = col; i < N && j >= 0; i++, j--){
        if(tab[i][j] == 1)
            return false;
    }
    // se chegou aqui, então está seguro (retorna true)
    return true;
}

void executar(int tab[N][N], int N, int col){
    if(col == N){
        printf("Solucao %d:\n\n", sol + 1);
        mostrarTabuleiro(tab, N);
        sol++;
        return;
    }
    for(int i = 0; i < N; i++){
        // Testa todas as linhas com o objetivo de encontrar uma linha segura
        // na coluna atual
        if(seguro(tab, N, i, col)){
            // insere a rainha (marca com 1)
            tab[i][col] = 1;
            // chamada recursiva
```

```
            executar(tab, N, col + 1);
            // remove a rainha (backtracking)
            tab[i][col] = 0;
        }
    }
}

int main(int argc, char *argv[]){
    // tabuleiro (matriz)
    int tab[N][N];

    for(int i = 0; i < N; i++)
        for(int j = 0; j < N; j++)
            tab[i][j] = 0;

    // imprime todas as soluções
    executar(tab, N, 0);

    // imprime a quantidade de soluções
    printf("\nEncontradas %d solucoes!\n", sol);
    return 0;
}
```

Divisão e conquista

Divisão e conquista é um paradigma para o desenvolvimento de algoritmos que faz uso da recursividade. Para resolver um problema utilizando esse paradigma, seguimos três passos:
1. O problema é dividido em pelo menos dois subproblemas menores;
2. Os subproblemas menores são resolvidos recursivamente: cada um desses subproblemas menores é dividido em subproblemas ainda menores, a menos que sejam tão pequenos a ponto de ser simples resolvê-los diretamente;
3. Soluções dos subproblemas menores são combinadas para formar uma solução do problema inicial.

Os algoritmos Mergesort e Quicksort fazem uso desse paradigma.

Exemplo - Máximo de um vetor

Vamos ilustrar uma primeira aplicação do paradigma divisão e conquista, calculando o maior elemento do vetor utilizando esse paradigma.

A ideia é ir dividindo o vetor ao meio até chegar em um subvetor com um único elemento que é o caso base. O maior elemento desse subvetor é o próprio elemento. Depois voltamos das recursões comparando o maior do subproblema da esquerda com o maior do subproblema da direita e retornando o maior entre eles até retornarmos o maior para o vetor de entrada. Esse

algoritmo é implementado no código em C dado abaixo. A ideia geral utilizada nesse exemplo pode ser estendida para vários outros problemas conforme veremos no exemplo da subsequência de soma máxima.

```c
float maior(int *v, int left, int right){
    if (left == right){ //Caso base
        return v[left];
    }
    else { //Divide o vetor ao meio e compara na volta
        int meio = (left + right)/2;
        float maiorleft = maior(v,left,meio);
        float maiorright = maior(v,meio+1,right);
        if (maiorleft > maiorright) {
            printf("O maior entre %d e %d é %f\n", left, right, maiorleft);
            return maiorleft;
        }
        else {
            printf("O maior entre %d e %d é %f\n", left, right, maiorright);
            return maiorright;
        }
    }
}
```

Exemplo - Problema da subsequência de soma máxima

O problema da subsequência de soma máxima consiste em, dada uma sequência de valores inteiros, encontrar uma subsequência cuja soma é máxima.
Formulação do problema:

Entrada

A = (a1, a2, … , aN) um vetor de inteiros

Saída

Indices i e j tais que a soma ai + ai+1 + … + aj é máxima.

Obs: Quando todos os valores de A são negativos, então a subsequencia vazia é escolhida e o valor máximo da soma é considerado 0.

Exemplo: Entrada: A = (-2,11,-4,13,-5,-2)
Saída: **i = 1 e j = 3**, a1 + a2 + a3 = 11 + -4 + 13 = 20.

Para resolver esse problema de otimização, poderíamos utilizar o paradigma força bruta. A seguir é dada a implementação da força bruta em C.

```
void subSeqSomaMax_FB(int A[], int n, int *i, int *j){
    int soma_max = 0, i = 0, j = 0;
    for (int primeiro = 0; primeiro < n; primeiro++){
        for (int ultimo = primeiro; ultimo < n; ultimo++){
            soma = 0;
            for (int k = primeiro; k <= ultimo; k++)
                soma += A[k];
            if (soma > soma_max){
                soma_max = soma;
                *i = primeiro;
                *j = ultimo;
            }
        }
    }
}
```

Podemos melhorar um pouco o algoritmo anterior, obtendo o seguinte código:

```
void subSeqSomaMax_FB(int A[], int n, int *i, int *j){
    int soma_max = 0, i = 0, j = 0;
    for (int primeiro = 0; primeiro < n; primeiro++){
        soma = 0;
        for (int ultimo = primeiro; ultimo < n; ultimo++){
            soma += A[ultimo]; //Podemos aproveitar os valores já calculados
                               //para subsequência anteriores
            if (soma > soma_max){
                soma_max = soma;
                *i = primeiro;
                *j = ultimo;
            }
        }
    }
}
```

Mas mesmo com essa nova versão, o algoritmo tem ordem de complexidade $O(n^2)$. Será que é possível melhorar a eficiência desse algoritmo? Sim, vamos mostrar isso com uma solução que utiliza o paradigma da divisão e conquista.

Solução com divisão e conquista

O funcionamento do paradigma de divisão e conquista se baseia no princípio da otimalidade:

O sufixo direito de um subcadeia central ótima também é prefixo ótimo para a metade direita e

O prefixo esquerdo de um subcadeia central ótima também é sufixo ótimo para a metade esquerda.

Abaixo é dado o código da solução utilizando divisão e conquista.

```
int divisaoEConquista(int *a, int left, int right, int *soma_max, int *I,
                      int *J){
    int Ileft, Iright, Jleft, Jright, i, j;
    if (left == right){
        if (a[left] > 0) {
            *soma_max = a[left];
            *I = left;
            *J = left;
        }
        else {
            *soma_max = 0;
            *I = 0;
            *J = 0;
        }
        return *soma_max;
    }
    else{
        int meio = (left + right)/2;
        int soma_max_bordEsq = 0, soma = 0, k;
        //Soma a parte esquerda da cadeia central
        for (k = meio; k >= left; k--){
            soma = soma + a[k];
            if (soma > soma_max_bordEsq)
                soma_max_bordEsq = soma;
        }
        int soma_max_bordDir = 0;
        soma = 0;
        //Soma a parte direita da cadeia central
        for (k = meio + 1; k <= right; k++){
            soma = soma + a[k];
            if (soma > soma_max_bordDir)
                soma_max_bordDir = soma;
        }

        int valorEsq = divisaoEConquista(a, left, meio, soma_max,
            &Ileft, &Jleft);
        int valorDir = divisaoEConquista(a, meio+1, right, soma_max,
            &Iright, &Jright);
        if (valorEsq >= valorDir){
            i = Ileft;
            j = Jleft;
            *soma_max = valorEsq;
        }
        else {
            i = Iright;
            j = Jright;
            *soma_max = valorDir;
        }
        if (*soma_max >= soma_max_bordEsq+soma_max_bordDir){
            *I = i;
```

```
            *J = j;
        }
        else {
            *soma_max = soma_max_bordEsq+soma_max_bordDir;
            *I = left;
            *J = right;
        }
        return *soma_max;
    }
}
```

Algoritmos gulosos

Podemos definir algoritmos gulosos da seguinte forma:
Constrói uma solução iterativamente através de uma sequência de decisões míopes (chamadas escolhas gulosas) esperando que dê tudo certo no final.

A definição acima pode parecer bem informal, mas resume bem as características de um algoritmo guloso que veremos através de exemplos.

Suponha que tenhamos disponíveis moedas com valores de 100, 50, 25, 10, 5 e 1. O problema do troco é um problema que consiste em obter um determinado valor com o menor número de moedas possível. Por exemplo, se quisermos dar um troco de 2.89, a melhor solução, isto é, o menor número de moedas possível para obter o valor, consiste em 9 moedas: 2 de valor 100, 1 de valor 50, 1 de valor 25, 1 de valor 10 e 4 de valor 1. De forma geral, ao realizar esse cálculo agimos como um algoritmo guloso: em cada passo adicionamos a moeda de maior valor possível, de forma a não passar da quantia necessária. Nesse caso, a nossa escolha gulosa foi a moeda de maior valor que somada à quantia atual não ultrapassa a quantia desejada.

Embora seja uma afirmação difícil de provar, é verdade que com os valores dados das moedas, e tendo-se disponível uma quantidade adequada de cada uma, o algoritmo sempre irá fornecer uma solução ótima para o problema. Entretanto, ressalta-se que para diferentes valores de moedas, ou então quando se tem uma quantidade limitada de cada uma, o algoritmo guloso pode vir a não chegar em uma solução ótima, ou até mesmo não chegar a solução nenhuma (mesmo esta existindo). O algoritmo para resolver o problema do troco é apresentado a seguir.

```
void Troco(int troco, int moedas[]){
    int soma = 0, cont = 0, num_moedas = 0;
    while (soma < troco){
        if (soma + moedas[cont] <= troco){
            soma = soma + moedas[cont];
            num_moedas++;
```

```
        }
        else
            cont++;
    }
    if (soma == troco)
        printf("O número mínimo de moedas é: %d\n",num_moedas);
}

void Troco2(int valor, int i, int moedas[], int num_moedas){
    if (valor == 0){
        printf("O número mínimo de moedas é: %d\n",num_moedas);
        exit(0);
    }
    else if (valor >= moedas[i]){
        valor = valor - moedas[i];
        num_moedas++;
        Troco2(valor, i, moedas, num_moedas);
    }
    else
        Troco2(valor, i+1, moedas, num_moedas);
}

int main(){
    int moedas[6] = {100,50,25,10,5,1};
    int troco;
    printf("Digite o valor do troco\n");
    scanf("%d", &troco);
    Troco(troco, moedas);
    Troco2(troco, 0, moedas, 0);

    return 0;
}
```

O algoritmo apresentado é caracterizado como guloso porque a cada passo ele escolhe o maior valor possível, sem refazer suas decisões, isto é, uma vez que um determinado valor de moeda foi escolhido, não se retira mais este valor do conjunto solução (ele não volta à trás como o backtracking).

Características gerais dos algoritmos gulosos

De forma geral, os algoritmos gulosos e os problemas por eles resolvidos são caracterizados pelos itens abordados a seguir. Para facilitar o entendimento, cada um dos itens será relacionado ao exemplo exposto anteriormente (problema do troco):

- Há um problema a ser resolvido de maneira ótima, e para construir a solução existe um conjunto de candidatos. No caso do problema do troco, os candidatos são o conjunto de moedas (que possuem valor 100, 50, 25, 10, 5 e 1), com quantidade de moedas suficiente de cada valor;

- Durante a "execução" do algoritmo são criados dois conjuntos: um contém os elementos que foram avaliados e rejeitados e outro os elementos que foram analisados e escolhidos;
- Há uma função que verifica se um conjunto de candidatos produz uma solução para o problema. Neste momento, questões de otimalidade não são levadas em consideração. No caso do exemplo, esta função verificaria se o valor das moedas já escolhidas é exatamente igual ao valor desejado.
- Uma segunda função é responsável por verificar a viabilidade do conjunto de candidatos, ou seja, se é ou não possível adicionar mais candidatos a este conjunto de tal forma que pelo menos uma solução seja obtida. Assim como no item anterior, não há preocupação com otimalidade. No caso do problema do troco, um conjunto de moedas é viável se seu valor total não excede o valor desejado;
- Uma terceira função, denominada função de seleção, busca identificar qual dos candidatos restantes (isto é, que ainda não foram analisados) é o melhor (o conceito de melhor dependerá do contexto do problema). No exemplo, a função de seleção seria responsável por escolher a moeda com maior valor entre as restantes;
- Por fim, existe a função objetivo, a qual retorna o valor da solução encontrada. No exemplo, esta função seria responsável por contar o número de moedas usadas na solução.

Algoritmo guloso genérico

```
1:  função ALGORITMOGULOSO(C: conjunto)                    ▷ C é o conjunto de candidatos
2:      S ← ∅                                              ▷ S é o conjunto que irá conter a solução
3:      enquanto C ≠ ∅ e não solução(S) faça
4:          x ← seleciona C
5:          C ← C − {x}   // REMOVE x DO CONJUNTO CANDIDATO
6:          se é viável S ∪ {x} então
7:              S ← S ∪ {x}
8:          fim se
9:      fim enquanto
10:     se solução(S) então
11:         retorne S
12:     senão
13:         retorne "Não existe solução!"
14:     fim se
15: fim função
```

Algoritmo guloso para o problema da subsequência de soma máxima

Podemos resolver o problema da subsequência de soma máxima de uma forma mais eficiente do que usando a técnica de divisão e conquista. O algoritmo guloso abaixo resolve o problema em tempo O(n). A escolha gulosa que o algoritmo faz é descartar todos os prefixos

negativos, atualizando o índice primeiro para o próximo elemento após o prefixo negativo que acabou de ser identificado.

```
void guloso(int *a, int *I, int *J, int* soma_max, int n){
    int primeiro = 0, ultimo, soma = *soma_max = *I = *J = 0;
    for (ultimo = 0; ultimo < n; ultimo++){
        soma = soma + a[ultimo];
        if(soma > *soma_max){
            *soma_max = soma;
            *I = primeiro;
            *J = ultimo;
        }
        else if (soma < 0){
            primeiro = ultimo + 1;
            soma = 0;
        }
    }
}
```

Outro exemplo - Problema de seleção de atividades

Este problema consiste em programar o uso de um recurso entre diversas atividades concorrentes, mais especificamente, selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis. Seja S = a1, a2, … , an um conjunto de n atividades que desejam utilizar um mesmo recurso, o qual pode ser utilizado por apenas uma atividade por vez. Cada atividade ai terá associado um tempo de início (si) e um tempo de término (fi), sendo que 0 ≤ si < fi < ∞ (isto é, o tempo de início deve ser menor que o tempo de fim, e este por sua vez, deve ser finito). Caso uma atividade ai seja selecionada, ela irá ocorrer no intervalo de tempo [si, fi). Diz-se que duas atividades são compatíveis se o intervalo de tempo no qual uma delas ocorre não se sobrepõe ao intervalo de tempo da outra (considerando a restrição de que o recurso pode ser utilizado por apenas uma atividade por vez). Sendo assim, as atividades ai e aj são compatíveis se os intervalos [si, fi) e [sj, fj) não se sobrepõem, ou seja, se si ≥ fj (a atividade ai inicia depois que aj termina) ou então sj ≥ fi (a atividade aj inicia depois que ai termina). Como exemplo, considere-se o conjunto S de atividades a seguir, o qual está ordenado de forma monotonicamente crescente de tempo de término (veremos qual é a vantagem de fazer essa ordenação):

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

No caso dessas atividades listadas na tabela acima, temos como exemplos de conjuntos de atividades mutuamente compatíveis, ou seja, que atendem à restrição explicitada anteriormente, os conjuntos {a2, a6, a11} e {a3, a9, a11}. Estes porém, não são um subconjunto máximo, pois pode-se obter dois conjuntos de atividades compatíveis com quatro elementos (subconjuntos máximos), sendo eles: {a1, a4, a8, a11} e {a2, a4, a9, a11}.

A solução gulosa para esse problema consiste em escolher a atividade mutuamente compatível (que ainda não foi escolhida) que tem o tempo de término menor. Essa será nossa escolha gulosa. A ordenação por tempo de término de forma crescente facilita a realização da escolha e consequentemente a implementação do algoritmo. A implementação do algoritmo em C é dada abaixo.

```c
//Referência: geeksforgeeks.org
// C program for activity selection problem.
// The following implementation assumes that the activities
// are already sorted according to their finish time
#include<stdio.h>
// Prints a maximum set of activities that can be done by a single
// person, one at a time.
//  n   -->  Total number of activities
//  s[] -->  An array that contains start time of all activities
//  f[] -->  An array that contains finish time of all activities
void printMaxActivities(int s[], int f[], int n){
    int i, j;
    printf ("Following activities are selected:\n");
    // The first activity always gets selected
    i = 0;
    printf("%d ", i);

    // Consider rest of the activities
    for (j = 1; j < n; j++){
        // If this activity has start time greater than or
        // equal to the finish time of previously selected
        // activity, then select it
        if (s[j] >= f[i]) {
            printf ("%d ", j);
            i = j;
        }
    }
}

// driver program to test above function
int main(){
    int s[] =  {1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12};
    int f[] =  {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    int n = sizeof(s)/sizeof(s[0]);
    printMaxActivities(s, f, n);
    return 0;
}
```

Observa-se no código que a escolha gulosa é feita percorrendo as atividades por ordem de término e escolhendo a próxima atividade que é compatível com as outras atividades já escolhidas.

Programação dinâmica

A programação dinâmica (PD) assim como o método de divisão e conquista resolve problemas combinando as soluções para subproblemas. O termo "programação" refere-se a um método tabular.

Os algoritmos divisão e conquista dividem um problema em subproblemas independentes, resolve-os recursivamente e combina as soluções. Ao contrário, a PD se aplica quando os subproblemas se sobrepõem, isto é, quando os subproblemas compartilham subsubproblemas. Um algoritmo de PD resolve cada subproblema somente uma vez e depois grava a sua resposta em uma "tabela" (na verdade, as respostas podem ser armazenadas de várias formas diferentes), evitando assim o trabalho de recalcular a resposta de cada subsubproblema. Em geral, a PD é aplicada a problemas de otimização.

O desenvolvimento de um algoritmo de PD envolve quatro etapas:

1. Caracterizar a estrutura de uma solução ótima;
2. Definir recursivamente o valor de uma solução ótima;
3. Calcular o valor de uma solução ótima (normalmente de baixo para cima);
4. Construir uma solução ótima com as informações calculadas.

Solução para o problema de seleção de atividades com PD

A primeira coisa a ser feita é definir uma subestrutura ótima, e então utilizá-la para construir uma solução ótima para o problema em questão a partir de soluções ótimas para subproblemas. Para tal, é necessária a definição de um espaço de subproblemas apropriado. Inicialmente, define-se conjuntos

Sij = {ak ∈ S: fi ≤ sk < fk ≤ sj} (1)

sendo que Sij é o conjunto de atividades que podem ser executadas entre o final da atividade ai e o início da atividade aj.

Para representar o problema todo, são adicionadas “atividades fictícias” a0 e an+1, e convenciona-se que f0 = 0 e sn+1 = ∞. A partir disso, S = s0,n+1 e os intervalos para i e j são dados por 0 ≤ i, j ≤ n + 1. Para que os intervalos de i e j sejam restringidos ainda mais, supõe-

se que as atividades estão dispostas em ordem monotonicamente crescente de tempo de término, isto é: $f0 \leq f1 \leq f2 \leq \ldots \leq fn < fn+1$. Com essa suposição, o espaço de subproblemas passa a ser selecionar um subconjunto máximo de atividades mutuamente compatíveis de Sij, para $0 \leq i < j \leq n + 1$, sendo que $Sij = \varnothing \ \forall \ i \geq j$. Para mostrar que tal afirmação é verdadeira, vamos supor que exista uma atividade $ak \in Sij$ para algum $i \geq j$, de tal forma que na sequência ordenada, aj é seguido por ai, ou seja, $fj \leq fi$. Entretanto, a partir da suposição que $i \geq j$ e de (1) tem-se que $fi \leq sk < fk \leq sj < fj$, que contradiz a hipótese que ai segue aj na sequência ordenada.

Agora, para "ver" a subestrutura do problema de seleção de atividades, considere um subproblema não vazio Sij, e suponha que uma solução para este subproblema envolva a atividade ak. No entanto, a utilização de ak irá gerar dois subproblemas:

1. Sik: conjunto de atividades que podem ser executadas entre o final da atividade ai e o início da atividade ak, isto é, que começam depois de ai terminar e terminam antes de ak começar.
2. Skj: conjunto de atividades que podem ser executadas entre o final da atividade ak e o início da atividade aj.

Onde cada uma das atividades é um subconjunto das atividades de Sij. Sendo assim, o número de atividades da solução para Sij é a soma do tamanho das soluções de Sik e Skj, mais uma unidade, correspondente à atividade ak. Agora, falta (i) mostrar a subestrutura ótima e (ii) utilizá-la para mostrar que é possível construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas.
(i) Seja Aij uma solução ótima para o problema Sij, e suponha que tal solução envolva a atividade ak. Esta suposição implica que as soluções Aik e Akj (para os problemas Sik e Skj, respectivamente) também devem ser ótimas. Aqui aplica-se o argumento de "recortar e colar".
(ii) O subconjunto de tamanho máximo Aij de atividades mutuamente compatíveis é definido como:
$\mathbf{Aij = Aik \cup \{ak\} \cup Akj}$ (2)

Isso porque a partir das definições do item (i), pode-se afirmar que a solução geral Aij (conjuntos de tamanho máximo de atividades mutuamente compatíveis em Sij) pode ser obtida a partir da divisão do problema principal em dois subproblemas, e posterior busca dos conjuntos de tamanho máximo para cada um destes subproblemas (Aik e Akj).

No desenvolvimento de uma solução de programação dinâmica, o segundo passo consiste na definição de uma solução recursiva para o cálculo do valor de uma solução ótima. Seja c[i, j] o número de atividades no subconjunto Sij que contém o número máximo de atividades mutuamente compatíveis com i, j. Considerando um conjunto não vazio Sij (lembrando: $Sij = \varnothing \ \forall \ i \geq j$) e considerando que a utilização de uma atividade ak implicará em outros dois subconjuntos Sik e Skj, a partir de (2) temos que:

c[i, j] = c[i, k] + c[k, j] + 1

, ou seja, o número de atividades no subconjunto Sij é o número de atividades em Sik (denotado por c[i, k]) , o número de atividades em Skj (c[k, j]) mais a atividade ak. O problema com essa equação recursiva é que o valor de k não é conhecido, sendo que sabe-se apenas que existem j − i − 1 valores possíveis para k (k = i + 1, . . . , j − 1). Mas levando-se em consideração que o conjunto Sik deve usar um destes valores para k, toma-se o melhor deles. Sendo assim, a definição recursiva completa é:

$$c[ij] = \begin{cases} 0, & \text{se } S_{ij} = \emptyset; \\ \max\{\text{c[i,k]+ c[k,j] + 1}\}, & \text{se } S_{ij} \neq \emptyset. \\ i<k<j \end{cases}$$

A partir dessa equação, completamos a definição da solução por PD e já podemos implementá-la. Lembrando que na hora de implementar temos que incluir a lógica para verificar se um novo candidato acrescentado à solução mantém a solução viável e se essa nova solução é melhor do que a melhor solução atual.

Abaixo é dado o código em C que implementa a solução por PD.

```c
#include <stdlib.h>
#include <stdio.h>

//max{c[i,k]+ c[k,j] + 1}

typedef struct TAtividade {
    int c;
    int id;
} atividades;

void imprimeAtividades(atividades **a, int i, int j){
    int k;
    if (a[i][j].c > 0){
        k = a[i][j].id;
        printf(" %d ", k);
        imprimeAtividades(a,i,k);
        imprimeAtividades(a,k,j);
    }
}

int selecaoAtividadesPD(atividades **a, int s[], int f[], int i, int j){
    if (j - i < 2)
        return 0;
    if (ativ[i][j].c > 0)
        return ativ[i][j].c;
```

```
    for(int k=i+1; k < j; k++){
        a[i][k].c = selecaoAtividadesPD(a,s,f,i,k);
        a[k][j].c = selecaoAtividadesPD(a,s,f,k,j);
        if (f[i] <= s[k] && f[k] <= s[j] &&
                  a[i][k].c + a[k][j].c + 1 > a[i][j].c) {
            a[i][j].c = a[i][k].c + a[k][j].c + 1;
            a[i][j].id = k;
        }
    }
    return a[i][j].c;
}

int main(){
    int s[] =  {0, 1, 3, 0, 4, 3, 5, 6, 8, 8, 2, 12, 100};
    int f[] =  {0, 4, 5, 6, 6, 8, 9, 10, 11, 12, 13, 14, 200};
    int n = sizeof(s)/sizeof(s[0]);
    printf("%d\n", n);
    atividades **ativ = (atividades**)malloc((n)*sizeof(atividades*));
    for (int i =0; i < n; i++) {
        ativ[i] = (atividades*)malloc((n)*sizeof(atividades));
    }

    for (int i =0; i < n; i++)
        for (int j =0; j < n; j++)
            ativ[i][j].c = 0;

    int maxAct = selecaoAtividadesPD(ativ, s, f, 0, n-1);
    printf("Máximo de atividades compatíveis: %d\n", maxAct);
    printf("Atividades selecionadas:\n");
    imprimeAtividades(ativ,0,n-1);

    printf("\nc:\n");
    for (int i=0; i < n; i++) {
        for (int j=i+2; j < n; j++)
            printf("c[%d,%d] = %d; ", i, j, ativ[i][j].c);
        printf("\n");
    }

    for (int i =0; i < n; i++) {
        free(ativ[i]);
    }
    free(ativ);
    return 0;
}
```