SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 10

Alunos	**NUSP**
Felipe Camargo Cerri	15451119
Gabriel Campanelli Iamato	15452920

Exercício 10 – Máxima Subsequência Crescente

Item a

Comentário

Nesse trabalho, foram implementados dois algoritmos para a solução do problema de máxima subsequência crescente (MSC): um por tabela hash e outro por busca binária. Ambos apresentam complexidade de tempo relativamente baixa, sendo **O(n) e O(nlogn)** respectivamente como será verificado teoricamente e empiricamente a seguir.

Tabela Hash

A tabela hash é uma estrutura de dados que permite a busca por um elemento em O(1), ela funciona através de endereçamento direto ao utilizar alguma função definida na implementação que relaciona cada entrada com uma posição diferente em uma lista.

1) **Decisões de Implementação**

O grande desafio de uma tabela hash eficiente é minimizar colisões entre diferentes entradas sem grande desperdício de memória, para contornar o desafio escolhemos um **fator de carga de aproximadamente 0.75** de forma que é alocado mais espaço na memória que o necessário a fim de otimizar a eficiência do algoritmo de busca. Além disso, por simplicidade e eficiência escolhemos utilizar o

método de divisão, inicialmente encontramos o maior número primo mais próximo do fator de carga escolhido e o definimos como o módulo da operação, para otimizar a função. O **tratamento de colisões** foi feito através da utilização de uma **lista encadeada** em cada posição da tabela.
Dessa forma, tabela utilizada é **estática, aberta, com uso de lista encadeada e com função de divisão**

MSC por Tabela Hash

O algoritmo aproveita do acesso indexado da tabela hash para otimizar a eficiência temporal ao máximo, podemos compartimentar se funcionamento em quatro etapas:

1) Armazenamos todos os números da entrada na ordem fornecida em um vetor auxiliar e em uma tabela hash para acesso indexado a qualquer elemento
2) O próximo passo é percorrer o vetor auxiliar verificando a existência de um número diretamente anterior ao atual através da tabela hash, caso o número já exista não estamos no primeiro elemento de uma sequência e podemos passar para o próximo item.
3) Caso o número diretamente anterior não exista significa que estamos no primeiro elemento da sequência, dessa forma fazemos a verificação dos elementos seguintes através da tabela hash contabilizando a quantidade de números na sequência.
4) Ao final verificamos se a sequência atual é a máxima até o momento e a guardamos caso seja. Finalmente podemos passar para o próximo item do vetor.

Complexidade

Note que será necessário alocar espaço para o vetor auxiliar de tamanho **n** (seja **n** a quantidade de elementos) e para tabela hash, sendo que da forma que definimos ela ocupará um espaço de **n/fator de carga + k** (em que **k** é o número de colisões) o que leva a uma complexidade espacial de **O(n + n/0.75 + k) = O(n)**, visto que as colisões **k** serão mínimas. Além disso, como vamos percorrer o vetor auxiliar uma vez (**O(n)**) e realizar buscas em **O(1)**, a complexidade final de tempo será **O(n)**.

MSC por Ordenação

A resolução do MSC por ordenação é relativamente simples e apesar de ser muito otimizada, não supera a solução por tabela hash. A ideia do algoritmo é receber a entrada em um vetor e ordená-lo, uma vez ordenado a busca por sequências é trivial visto que podemos percorrer o array do inicio ao fim contabilizando o tamanho de cada sequência enquanto respeite o critério e reiniciar a contagem toda vez que um número não subsequente é encontrado, se atentando a armazenar a maior subsequência.

Complexidade

Apesar de ocupar menos memória na prática que o algoritmo anterior, ainda precisamos armazenar todos os elementos de forma que a **complexidade de espaço é O(n)**. O cálculo da complexidade temporal pode ser feita em duas etapas: para a ordenação vamos tomar como base algum algoritmo de ordenação otimizado, assim a complexidade será **O(nlogn)** (a exemplo nós utilizamos o **quicksort** que possui O(nlogn) para o caso médio), finalmente como vamos percorrer o vetor mais uma vez para achar as subsequências termos uma **complexidade temporal de O(n + nlogn) = O(nlogn).**

Código

As funções julgadas não essenciais (como o quicksort que já foi implementado anteriormente) não estão presentes no documento mas podem ser conferidas na submissão do RunCodes.

Funções utilizadas

```c
//funcoes da resolucao por hash
void insere_hashing(int valor, estruturaHash tabelaHash);
int busca_hashing(int valor, estruturaHash tabelaHash); //retorna o valor caso exista
int subsequencia_crescente(estruturaHash tabelaHash, int *vet, int n);
void libera_hashing(estruturaHash tabelaHash);
int proximo_primo(int n);

//funcoes da resolucao por ordenacao
void swap (int *a, int *b);
int achar_pivo(int *vector, int ini, int fim);
void quick_sort(int *v, int ini, int fim);
int msc_ordenacao(int *v, int tam_vetor);
```

Funções da Tabela Hash

```c
//estrutura de no utilizada para lista encadeada
typedef struct no_ {
    struct no *proximo;
    int valor;
} NO;

//estrutura de hash que contem a tabela e o modulo utilizado
typedef struct tabelahash_ {
    NO *tabela;
    int modulo;
} estruturaHash;
```

structs

```c
void insere_hashing(int valor, estruturaHash tabelaHash) {
    int B = tabelaHash.modulo;
    int indice = valor % B; //funcao hash

    NO *aux = &(tabelaHash.tabela[indice]); int count = 0;

    if (aux->valor != 0) //verifica se ja existe um valor no nó
        count++;
    while (aux->proximo != NULL) { //percorre a lista encadeada caso exista colisao
        aux = (NO*)aux->proximo;
        count++;
    }

    if (count != 0) { //declara um no para o proximo elemento caso tenha colisao
        aux->proximo = malloc(size: sizeof(NO));
        aux = (NO*)aux->proximo;
    }

    aux->valor = valor;
    aux->proximo = NULL;
}
```

inserção

```c
int busca_hashing(int valor, estruturaHash tabelaHash) {
    int indice = valor % tabelaHash.modulo; //funcao de hashing
    NO *aux = &(tabelaHash.tabela[indice]);

    //percorre a lista encadeada pra achar o valor caso a lista tenha mais de um elemento
    while (aux->proximo != NULL && aux->valor != valor) {
        aux = (NO*)aux->proximo;
    }

    if (aux->valor == valor)
        return valor;
    else
        return 0; //caso o valor nao tenha sido encontrado

}
```

busca

MSC por Hash

```c
int subsequencia_crescente(estruturaHash tabelaHash, int *vet, int n){
    int aux, maxSub = 0, tempMaxSub = 1;
    for (int i = 0; i<n; i++) {
        aux = vet[i];
        if (busca_hashing(valor: aux-1, tabelaHash) == 0) { //se for o primeiro elemento da subsequencia
        while (busca_hashing(valor: ++aux, tabelaHash) != 0) { //caso ache um elemento subsequente
            tempMaxSub++; //a sequencia atual é incrementada
        }
        maxSub = max(maxSub, tempMaxSub); //determina se a sequencia atual é a melhor global
        tempMaxSub = 1;
        }
    }

    return  maxSub;
}
```

MSC por Ordenação

```c
int msc_ordenacao(int *v, int tam_vetor){
    // Se o tamanho do vetor é 1, a msc é 1
    if(tam_vetor == 1){
        return 1;
    }
    int msc = 1;
    int seq_atual = 1;
    for(int i = 1; i<tam_vetor; i++){
        if(v[i] == v[i-1]+1){
            seq_atual++;
        }
        else{
            if(seq_atual>msc){
                msc = seq_atual;
            }
            seq_atual = 1;
        }
    }
    if(seq_atual>msc){
            msc = seq_atual;
    }

    return msc;

}
```

□ Saída

Para comprovar a eficiência dos dois métodos de ordenação, foi utilizada a biblioteca <time.h> da linguagem C para medir o tempo de execução dos algoritmos com casos de teste do RunCodes.

Tempos de Execução

Seja **n** a quantidade de números contida na entrada

	Caso 1: **n = 7** **Saída: 3**	**Caso 5:** **n = 1000** **Saída: 37**	**Caso 7:** **n = 50000** **Saída: 77**	**Caso 9:** **n = 600000** **Saída: 92**
Tabela Hash	0.000020	0.000274	0.013007	0.170993
Ordenação	0.000016	0.000424	0.021951	0.193566
Algoritmo mais rápido	Ordenação	Tabela Hash	Tabela Hash	Tabela Hash

A partir dos testes, percebe-se que a solução por tabela hash é moderadamente mais eficiente especialmente para entradas maiores do que o algoritmos que usam ordenação no que tange tempo de execução, de modo que foi

possível verificar a diferença teórica nas complexidades sendo **O(n)** no algoritmo por tabela hash e **O(nlogn)** no por ordenação**.** Vale ressaltar que apesar da superioridade em eficiência temporal do algoritmo por hash em relação ao segundo, o primeiro apresenta como leve desvantagem o uso de espaço extra na prática, dado que o algoritmo requer armazenamento adicional para realizar as buscas indexadas especialmente se utilizarmos alguma tabela hash eficiente e consequentemente com fator de carga menor a 1, mesmo que os dois tenham a mesma complexidade assintótica **O(n).**
