SCC0220 - Laboratório Introdução à Ciência da Computação II

Relatório de execução da aula prática 5

Alunos	NUSP
Felipe Camargo Cerri	15451119
Gabriel Campanelli Iamato	15452920

Exercício 5 – Bubble Sort e Quick Sort

Item a

Comentário

Foram implementadas duas formas de ordenação de vetores, o Bubble Sort e o Quick Sort. O Quick Sort passou em todos os testes do RunCodes, enquanto o Bubble Sort teve tempo de execução excedido em 2 testes. Ambos os algoritmos de ordenação são baseados em troca de elementos, e serão apresentados a seguir.

Bubble Sort

O algoritmo do Bubble Sort funciona da seguinte forma: percorre o vetor inteiro, e a cada iteração, verifica cada elemento com o seu sucessor e os troca de lugar caso estejam na posição errada. A cada iteração i de 0 a n no vetor (seja n o tamanho do vetor), o algoritmo precisa percorrer n-i-1 vezes o vetor, o que representa dois laços de repetição aninhados, ocasionando no pior caso uma complexidade de $O(n^2)$.

Entretanto, foi construída uma versão otimizada do Bubble Sort, note que a variável booleana “troca” guarda se durante a iteração i que percorre o vetor houve alguma troca de elementos, se não houve , isso significa que o vetor já está ordenado, então o algoritmo é interrompido previamente. Ainda com essa

mudança, o pior caso permanece $O(n^2)$, mas o melhor caso passa a ser O(n), pois ele apenas vai percorrer o vetor de n elementos, concluir que já está ordenado, e encerrar o algoritmo.

Quick Sort

O algoritmo de Quick Sort usa a ideia de dividir e conquistar, ou seja, separar um vetor maior em dois subvetores em um processo recursivo. O algoritmo escolhe um elemento pivô, e os outros elementos serão rearranjados de forma que os elementos à direita do pivô sejam maiores que ele e os da esquerda menores. Para isso, percorre o vetor com i começando do início (com i++) e com j percorrendo do final (com j--). Quando encontra-se um elemento x tal que x>=v[i] e outro y tal que y<=v[j], troca-se os elementos v[i] e v[j]. Esse processo continua até que i seja igual a j, e os elementos à direita do pivô e à esquerda estão arranjados corretamente de acordo com ele, mas repete-se o processo para os 2 subvetores, escolhendo-se outro pivô, em um processo recursivo.

Os piores casos são quando o vetor já está ordenado e escolhe-se ou o último ou primeiro como pivô (em todas as iterações) ou quando escolhe-se o maior ou menor elemento em todas as iterações (caso geral que engloba o primeiro), fazendo com que a cada iteração , tira-se 1 elemento, tendo que realizar n chamadas , dado n o tamanho do vetor. Como a cada chamada percorre-se o vetor, a complexidade é O(n) * O(n), totalizando $O(n^2)$ para o pior caso.

Para evitar o pior caso, a abordagem utilizada foi escolher-se um pivô aleatório usando-se a função “rand()” da biblioteca “time.h”. A abordagem usa o conceito de *Randomized-Partition* , que quase impossibilita que o Quick Sort caia no pior caso diminuindo muito a probabilidade de que todos os elementos escolhidos sejam ou o menor ou o maior elemento, uma ideia simples da prova é pensar que a cada iteração de n elementos, a probabilidade de escolher o maior ou menor é 2/n. Essa abordagem é recomendada por Cormen, Leiserson, Rivest e Stein (2012, p. 161), que demonstram que para essa estratégia a complexidade esperada para a maioria dos casos é O(n*logn), considerando-se log na base 2.

💻 Código

Bubble Sort

```c
void bubble_sort_aprimorado(prato *vetor, int tamanho_vetor){
    int i, j;
    bool troca = true;
    // Se não for feita nenhuma troca, retorna
    for(i = 0; i<tamanho_vetor && troca; i++){
        troca = false;
        // Para em tamanho_vetor -i - 1. O -i refere-se aos elementos já ordenados
        for(j = 0; j<tamanho_vetor-i-1; j++){
            // realiza a troca conforme as especificações do projeto
            if(vetor[j].prioridade>vetor[j+1].prioridade || (vetor[j].prioridade==vetor[j+1].prioridade && vetor[j].tempo< vetor[j+1].tempo)){
                //a função "swap" troca os dois elementos
                swap(&vetor[j], &vetor[j+1]);
                troca  = true;
            }
        }
    }
}
```

Quick Sort

```c
void quick_sort(prato *vector, int ini, int fim){
    // Define-se pivo como elemento aleatório (abordagem que dificulta cair no pior caso do Quick Sort)
    int i =ini, j = fim;
    srand(time(NULL));
    int index_pivo = ini +rand()%(fim-ini+1);
    //armazena-se a prioridade e o tempo do pivô
    int pivo_prioridade = vector[index_pivo].prioridade;
    int pivo_tempo = vector[index_pivo].tempo;
   // Primeira etapa, percorre enquanto i<j
    do{
        // percorre i e j até que um elemento da esquerda pertença à direita
        while(vector[i].prioridade<pivo_prioridade || (vector[i].prioridade == pivo_prioridade && vector[i].tempo> pivo_tempo)) i++;
        while(vector[j].prioridade>pivo_prioridade || (vector[j].prioridade == pivo_prioridade && vector[j].tempo< pivo_tempo)) j--;
        // Não é estavel (realiza troca de elementos iguais)
        if(i<=j){
            swap(&vector[i], &vector[j]);
            i++;
            j--;
        }
    } while(i<j);
    // Continua iterações
    if(j>ini){
        quick_sort(vector, ini, j);
    }
    if(i<fim){
        quick_sort(vector, i, fim);
    }
}
```

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. Como os resultados foram testados no RunCodes, serão apresentados apenas os tempos de execução de cada algoritmo, com casos de teste do RunCodes:

```
Tempo de Execução Bubble Sort: 0.002029(base)
```

```
Tempo de Execução Quick Sort: 0.001404(base)
```

Tempo de execução mais curto (497 casos) : Quick Sort

```
Tempo de Execução Bubble Sort: 0.020447(base)
```

```
Tempo de Execução Quick Sort: 0.002826(base)
```

Tempo de execução mais curto (1193 casos): Quick Sort

```
Tempo de Execução Quick Sort: 0.053143(base)
```

```
Tempo de Execução Bubble Sort: 8.407246(base)
```

Tempo de execução mais curto (38286 casos): Quick Sort

A partir dos testes, percebe-se que Quick Sort é um algoritmo mais eficiente do que o Bubble Sort no que tange tempo de execução, e apresentou eficiência mais rápida nos casos testados. A diferença de processamento ficou mais clara ao passo que o tamanho dos casos teste aumentava.

Referências

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: teoria e prática. 3. ed. Tradução da 3. ed. americana por Arlete S. Marques. Rio de Janeiro: Elsevier, 2012.