UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA
ESTRUTURA DE DADOS 2

MATHEUS DE ABREU BOZZI

3º TRABALHO COMPUTACIONAL DE ESTRUTURA DE DADOS 2, BIN PACKING - EMPACOTAMENTO

VITÓRIA - ES
MARÇO / 2022

SUMÁRIO

1 INTRODUÇÃO

Segundo a especificação do trabalho, o problema de bin packing é um problema fundamental para minimizar o consumo de um recurso escasso, geralmente espaço. Aplicações incluem: empacotar os dados para tráfego na Internet, otimizar a alocação de arquivos, alocar a memória do computador para programas, etc. Indústrias de tecido, papel e outras utilizam a versão 2D desse problema para otimizar o corte de peças em placas de tamanho fixo. Empresas de logística utilizam a versão 3D para otimizar o empacotamento de caixas ou caminhões.

Suponha uma empresa que realiza um backup diário de todos os seus arquivos em discos com capacidade de 1 GB. Para diminuir os custos, idealmente a quantidade de discos utilizada a cada backup deve ser a mínima necessária para armazenar todos os arquivos. Em outras palavras, a tarefa é associar os arquivos a discos, usando a menor quantidade possível de discos. Infelizmente, esse problema é NP-hard, o que quer dizer que é muito pouco provável que exista um algoritmo eficiente (polinomial) para encontrar o empacotamento ótimo. Assim, o objetivo deste trabalho é projetar e implementar heurísticas que executam rapidamente e produzem boas soluções (próximas do ótimo).

Podemos formular o problema de bin packing da seguinte forma: dado um conjunto de N arquivos, todos com tamanhos entre 0 e 1,000,000 KB (1 GB), devemos descobrir uma forma de associá-los a um número mínimo de discos, cada um com capacidade 1 GB. Duas heurísticas bastante intuitivas surgem imediatamente.

Worst-fit e best-fit. A heurística worst-fit considera os arquivos na ordem que eles são apresentados: se o arquivo não cabe em nenhum dos discos atualmente em uso, crie um novo disco; caso contrário, armazene o arquivo no disco que tem o maior espaço restante. Por exemplo, este algoritmo colocaria os cinco arquivos de tamanho 700,000, 800,000, 200,000, 150,000 e 150,000 em três discos: {700,000, 200,000}, {800,000, 150,000} e {150,000}. A heurística best-fit é idêntica à exceção de que o arquivo é armazenado no disco que tem o menor espaço restante dentre todos os discos com espaço suficiente para armazenar o arquivo. Essa heurística colocaria a sequência anterior de arquivos em dois discos: {800,000, 200,000} e {700,000, 150,000, 150,000}.

Worst-fit e best-fit decrescentes. Empacotadores experientes sabem que se os itens menores são empacotados por último eles podem ser usados para preencher as lacunas em pacotes quase cheios. Essa ideia motiva uma estratégia mais esperta: processar os arquivos do maior para o menor. A heurística worst-fit decrescente é igual a worst-fit mas antes os arquivos são ordenados por tamanho em ordem decrescente. A heurística de best-fit decrescente é definida de forma análoga.

1.1 Objetivos

O objetivo é implementar quatro heurísticas para resolver um problema de empacotamento (bin packing). Este problema é uma abstração que serve para representar várias situações práticas. Para concretizar a situação, vamos considerar um cenário onde queremos armazenar arquivos de diferentes tamanhos em discos de tamanho fixo.

2 - DESENVOLVIMENTO

Para o desenvolvimento deste trabalho, foram seguidos os passos dados na especificação. Foram criados os arquivos **best_fit.c, best_fit.h, worst_fit.c e worst_fit.h** para que fosse desenvolvido o corpo das funções e as estruturas necessárias, e a **main.c**.

2.1 Passo 1

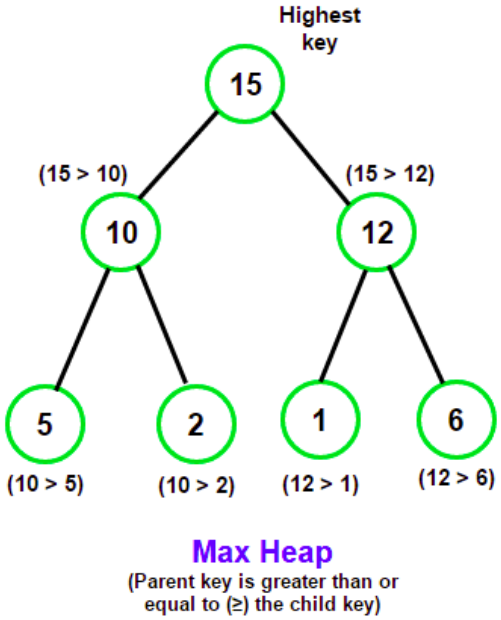
Para o primeiro passo, foi desenvolvido na main um código capaz de ler os arquivos de entrada e armazenar em um vetor de inteiros, chamado de array.

```
int main(int argc, char const* argv[]) {
    if (argc != 2) {
        printf("Execução: ./trab3 caminho_arquivo\n");
        return 0;
    }
    FILE* arqv_de_entrada = fopen(argv[1], "r");

    if (arqv_de_entrada == NULL) {
        printf("Erro ao abrir o arquivo de entrada\n");
        return 0;
    }

    char inputFile[50];
    fgets(inputFile, sizeof(inputFile), arqv_de_entrada);
    int N = atoi(inputFile);

    int* array = malloc(N * sizeof(int));
    int i = 0;
    while ((fgets(inputFile, sizeof(inputFile), arqv_de_entrada)) != NULL) {
        array[i] = atoi(inputFile);
        i++;
    }
```

2.2 Passo 2

Para o segundo passo, foi criado o worst_fit.c e worst_fit.h e de acordo com a especificação era indicado a utilização de uma fila de prioridade como a max-heap

que funciona da seguinte forma: “o elemento pai é sempre maior ou igual ao filho”.



Portanto, para desenvolver essa heurística foi criado uma estrutura Max_Heap

```
typedef struct max_heap {
        int* elmt;
        int n;
} Max_Heap;
```

Foi implementado à função init_max_heap que inicializa e aloca a estrutura de acordo com o 1GB que é o tamanho definido no trabalho.

```
Max_Heap* init_max_heap(int N) {
   Max_Heap* max_heap = malloc(sizeof(*max_heap));
   max_heap->n = 0;
   max_heap->elmt = malloc(N * sizeof(int*));
   return max_heap;
}
```

Também foi criada a função de inserir um elemento no heap. Na função de insert é verificado o caso em que o filho seja maior que o pai, nesse caso no while verificamos e é feito a troca dos elementos até que a ordem seja recuperada.

```
void insert_max_heap(Max_Heap* mh, int size) {
   mh->n++;
   mh->elmt[mh->n] = size;
   int i = mh->n;
   while ((mh->elmt[i / 2] < mh->elmt[i]) && i > 1) {
        int aux = mh->elmt[i];
        mh->elmt[i] = mh->elmt[i / 2];
        mh->elmt[i / 2] = aux;
        i = i / 2;
   }
}
```

A outra função implementada foi a de retirar o maior elemento do max-heap e após isso ordenar o heap verificando caso o nó pai possua um valor menor do que um dos filhos, então são feitas trocas até que a ordem dos elementos esteja de acordo.

```
int delete_max_heap(Max_Heap* mh) {
    int h = 1;
    int oldMax = mh->elmt[1];
    int aux = mh->elmt[1];
    mh->elmt[1] = mh->elmt[mh->n];
    mh->elmt[mh->n] = aux;
    mh->n = mh->n - 1;          // retira o maior elemento
    while (2 * h <= mh->n) {   // reordena o heap
        int j = 2 * h;
        if ((mh->elmt[j] < mh->elmt[j + 1]) && (j < mh->n)) {
            j++;
        }
        if (mh->elmt[h] > mh->elmt[j]) {
            break;
        }
        int aux2 = mh->elmt[h];
        mh->elmt[h] = mh->elmt[j];
        mh->elmt[j] = aux2;
        h = j;
    }
    return oldMax;
}
```

Foi implementada a função de desalocar a memória das estruturas também.

```
void free_max_heap(Max_Heap* max_heap) {
    free(max_heap->elmt);
    free(max_heap);
}
```

2.3 Passo 3

Para o terceiro passo, foi implementada a heurística worst fit utilizando o Max_Heap, que dentro do *for* vai verificando se o elemento do array é maior que o maior elemento do heap, se for, então utiliza a função insert, se não, deleta o maior elemento e diminui do valor do vetor de elementos da entrada e aí insere. Após isso é feito todo o processo de empacotamento e o número de discos utilizados é retornado.

```
int worstFit(int* array, int N, int disk_max) {
    Max_Heap* mh = init_max_heap(N);
    insert_max_heap(mh, disk_max);
```

```
    for (int i = 0; i <= N - 1; i++) {
      if (array[i] > mh->elmt[1]) {// elemento maior que o maior valor da heap
            insert_max_heap(mh, disk_max - array[i]);
        } else {
            int oldMax = delete_max_heap(mh);
            int max = oldMax - array[i];
            insert_max_heap(mh, max);
        }
    }
    int n = mh->n;
    free_max_heap(mh);

    return n;
}
```

2.4 Passo 4

Para o quarto passo, foi utilizada a estrutura de *árvore binária de busca* (bst), como sugerida na especificação, para a heurística de Best Fit. Portanto, foi criado o arquivo best_fit.c e best_fit.h para a implementação dessa estrutura e a heurística best fit.

Foi criado a estrutura Bst da seguinte forma:

typedef struct bst {
int data;
struct bst r;*
struct bst l;*
} Bst;

Foi criado a função de inicializar e alocar dinamicamente o bst passando o tamanho de 1GB, inicialmente.

```
Bst* init_bst(int size) {
    Bst* bst = (Bst*)malloc(sizeof(*bst));
    bst->data = size;
    bst->l = NULL;
    bst->r = NULL;
    return bst;
}
```

Foi criada também a função de inserir um elemento na bst que para cada nó X, todos os nós na subárvore esquerda de X têm chave menor que X e todos os nós na subárvore direita de X têm chave maior ou igual que X. Então é feito de forma recursiva até inserir o elemento no lugar correto.

```
Bst* insert_bst(Bst* bst, int data) {
    if (bst == NULL) {
        return init_bst(data);
    }
    if (data < bst->data) {
```

```
        bst->l = insert_bst(bst->l, data);
    } else {
        bst->r = insert_bst(bst->r, data);
    }
    return bst;
}
```

Foi criada também a função de remover (delete_bst) um elemento X da bst de forma recursiva, para isso foram feitas algumas considerações para caso a bst tenha apenas o filho à direita ou caso tenha apenas filho à esquerda e assim criando um nó auxiliar para armazenar esses valores e percorrendo os nós da bst para deletar o dado correto e por fim retornando a bst sem o elemento.

```
Bst* delete_bst(Bst* bst, int data) {
    if (bst == NULL) {
        return bst;
    }
    if (data < bst->data) {
        bst->l = delete_bst(bst->l, data);

    } else if (data > bst->data) {
        bst->r = delete_bst(bst->r, data);
    } else {
        if (bst->l == NULL) {
            Bst* aux = bst->r;
            free(bst);
            return aux;
        } else if (bst->r == NULL) {
            Bst* aux = bst->l;
            free(bst);
            return aux;
        }
        Bst* aux = bst->r;
        while (aux->l != NULL) {
            aux = aux->l;
        }
        bst->data = aux->data;
        bst->r = delete_bst(bst->r, aux->data);
    }
    return bst;
}
```

Após isso foi implementado a função de busca na bst, que recebido um dado como parâmetro da função, procurava se o elemento da árvore é maior que o dado, isso significa que tem espaço disponível no disco, então é verificado à esquerda

recursivamente e retorna o dado buscado. Caso o dado não seja maior então é feito de forma recursiva ao lado direito, percorrendo assim toda a árvore e caso não encontre retorna 0.

```
int find_bst(Bst* bst, int data) {
    if (bst->data >= data) {
        if (bst->l == NULL) {
            return bst->data;
        } else {
            int elem = find_bst(bst->l, data);
            if (elem > 0) {
                return elem;
            } else {
                return bst->data;
            }
        }
    } else {
        if (bst->r == NULL) {
            return 0;
        } else {
            return find_bst(bst->r, data);
        }
    }
}
```

Por fim, foi criada a função para deslocar recursivamente os espaços de memória da bst.

```
void free_bst(Bst* bst) {
    if (bst == NULL) {
        return;
    }
    free_bst(bst->l);
    free_bst(bst->r);
    free(bst);
}
```

2.5 Passo 5

Para o quinto passo, foi implementada a heurística best-fit utilizando a estrutura de árvore binária de busca (bst). Para essa heurística foi feito um *for* que percorre todos os elementos do array de entrada. É então procurado esse elemento na árvore, se não achares (res ==0) então é inserido na bst passando como dado o valor de 1GB menos o valor do elemento do array e incrementado o número de discos utilizados. Para o caso que encontre o elemento, é deletado e depois inserido o novo valor passando o valor atualizado para o disco. Por fim, é liberada à bst e retornado à quantidade de discos gastos pela heurística.

```
int bestFit(int* array, int N, int disk_max) {
    int n = 0;
```

```
    Bst* bst = init_bst(disk_max);
    for (int i = 0; i <= N - 1; i++) {
        int res = find_bst(bst, array[i]);
        if (res == 0) {
            bst = insert_bst(bst, disk_max - array[i]);
            n = n + 1;
        } else {
            bst = delete_bst(bst, res);
            bst = insert_bst(bst, res - array[i]);
        }
    }
    free_bst(bst);
    return ++n;
}
```

2.6 Passo 6

Para o sexto passo, foi implementada a função qsort de forma decrescente para depois aplicar as heurísticas novamente.

```
    qsort(array, N, sizeof(int), cmpfunc);
```

```
int cmpfunc(const void* a, const void* b) {
    return (*(int*)b - *(int*)a);  // desc
}
```

Por fim, foram chamadas às funções e com o retorno obtive o número de discos utilizados por cada heurística e foi desalocado o array do arquivo de entrada.

```
    int worst_fit = worstFit(array, N, DISK_SIZE);
    printf("%d\n", worst_fit);
    int best_fit = bestFit(array, N, DISK_SIZE);
    printf("%d\n", best_fit);

    qsort(array, N, sizeof(int), cmpfunc);

    int worst_fit_desc = worstFit(array, N, DISK_SIZE);
    printf("%d\n", worst_fit_desc);
    int best_fit_desc = bestFit(array, N, DISK_SIZE);
    printf("%d\n", best_fit_desc);

    free(array);
    fclose(arqv_de_entrada);
```

2.7 Passo 7

Para o sétimo passo, foram feitos os testes e verificado o tempo para cada para os arquivos de entrada. (A tabela preenchida será enviada em anexo). As figuras abaixo mostram a execução dos testes de entrada. OBS: Para marcar o tempo de execução foi utilizado o parâmetro time no terminal do ubuntu.

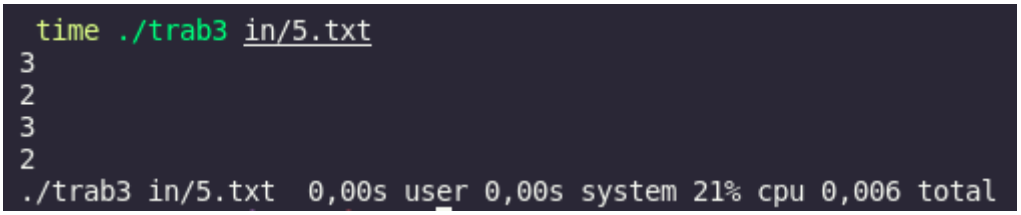


Figura 1 - Execução do trabalho para entrada 5.txt

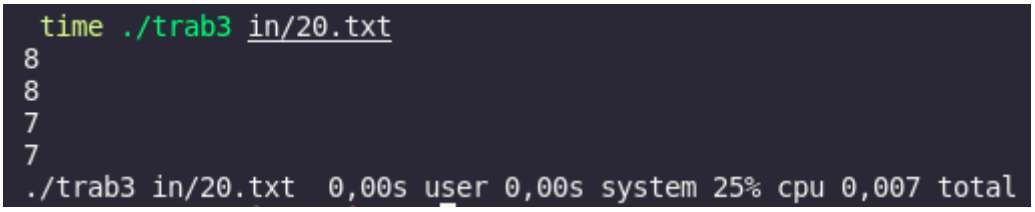


Figura 2 - Execução do trabalho para entrada 20.txt

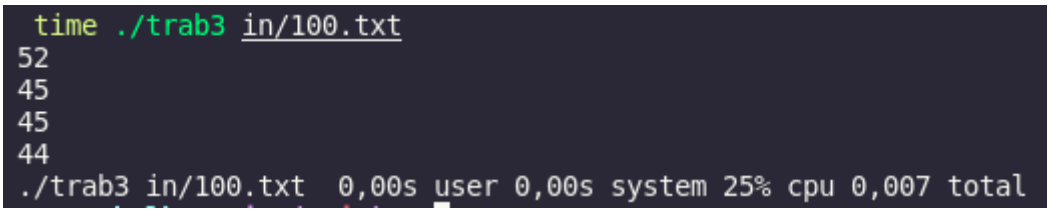


Figura 3 - Execução do trabalho para entrada 100.txt

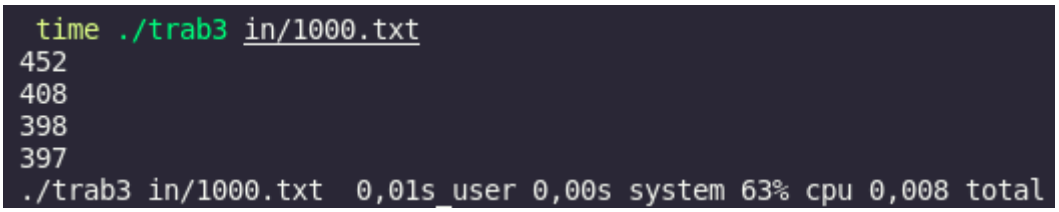


Figura 4 - Execução do trabalho para entrada 1000.txt

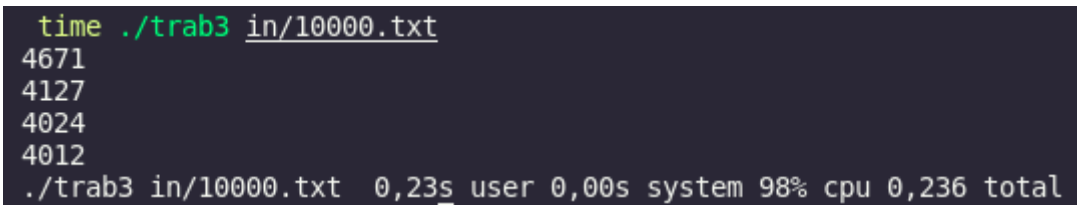


Figura 5 - Execução do trabalho para entrada 10000.txt

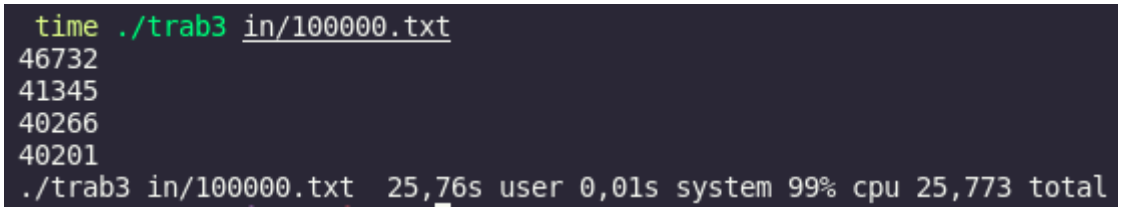


Figura 6 - Execução do trabalho para entrada 100000.txt

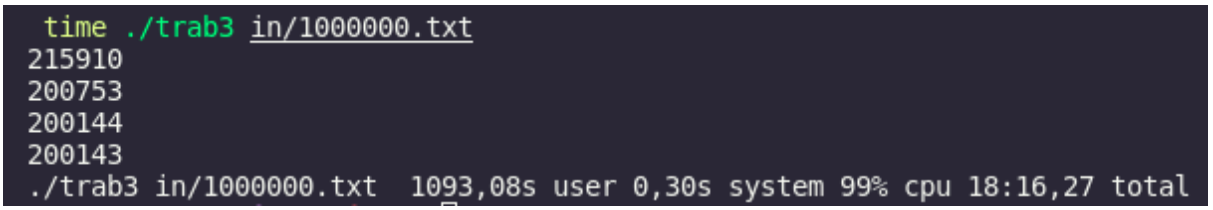


Figura 7 - Execução do trabalho para entrada 1000000.txt

Para as instâncias até 10^5, os tempos de todas as quatro heurísticas foram de no máximo 1 minuto (25 segundos). Para a entrada de 10^6 o desempenho para o best-fit decrescente foi o que mais consumiu tempo de processamento, e por fim as quatro heurísticas foram finalizadas em 18 minutos e 16 segundos já que é mais difícil e ficou como desafio.

3 - CONCLUSÃO

Neste trabalho, chegou-se aos resultados esperados nas heurísticas com as estruturas max-heap e bst que foram utilizadas, na ordenação foi utilizada a função de comparação qsort. Ao longo do desenvolvimento do trabalho, foram verificadas as alocações de memória. A parte principal foi avaliar quais estruturas seriam adequadas para o problema dado e implementá-las, e por fim, os resultados foram satisfatórios. O computador utilizado possui memória RAM de 4GB e é antigo (quase 8 anos), então o resultado pode ser melhor em computadores melhores.