3o trabalho de Estrutura de Dados II
Bin Packing - Empacotamento

Este trabalho trata de ***Busca e Ordenação em Memória Principal*** e deverá ser realizado em dupla. 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.

A avaliação ocorrerá em três etapas sequenciais:

1) Entrega do código fonte conforme a especificação, compilando e gerando o resultado esperado para os casos de teste fornecidos;
    a) O código deverá aplicar boas práticas de programação, como comentários, modularização e reuso de código. Por exemplo, o mesmo código poderia ser facilmente modificado para outros tipos de dados, ou outros métodos poderiam ser facilmente acrescentados.
2) Relatório comparando e explicando os resultados obtidos, em particular em relação ao desempenho dos métodos e estruturas utilizados.
3) Entrevista com os componentes do grupo sobre o trabalho executado.

A falha em uma etapa implica que as seguintes são automaticamente zeradas. Falha em responder as questões da entrevista, demonstrando que o entrevistado não compreende o próprio trabalho, implica em perda de pontos para a dupla. Portanto, se o seu companheiro do trabalho não estiver interessado na compreensão do mesmo, comunique ao professor para entregar o trabalho individualmente.

Atenção: plágio não será tolerado! Os grupos envolvidos receberão nota zero. Se algum colega pedir ajuda, não forneça o seu código como exemplo, mas trabalhem juntos sobre o código dele/dela.

Bônus: a (correta) inclusão/análise de outros métodos/estuturas além dos exigidos permitirá à dupla recuperar pontos (eventualmente) perdidos na avaliação (deixe claro no relatório se houver tais inclusões).

Entrega: o aluno deverá realizar via Google Classroom o upload de três arquivos: (i) um arquivo .zip contendo os arquivos fontes e makefile, (ii) um arquivo .pdf para o relatório e (iii) um arquivo .txt com os resultados conforme seção 4 deste documento.

1 Introdução

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 e 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.

2 Entrada e Saída

O seu programa deve ler um arquivo de entrada em formato de texto simples, onde cada linha contém um número inteiro positivo. A primeira linha é o valor *N* do número de arquivos. A seguir, as *N* linhas indicam os tamanhos dos arquivos, todos entre 0 e 1,000,000 KB.

Para um mesmo arquivo de entrada, o seu programa deve computar as quatro heurísticas: *worst-fit*, *best-fit*, *worst fit* decrescente e *best-fit* decrescente, exibindo (na TELA) o número de discos utilizados por cada heurística, um valor por linha, nessa ordem.

2.1 Execução do trabalho

O seu trabalho será executado da seguinte maneira:

./trab3 [problema.txt]

Isto é, o seu programa recebe como entrada um arquivo de uma instância de teste e gera como saída os quatro valores das heurísticas, como indicado acima. Um exemplo de execução para a instância com 5 arquivos discutida anteriormente:

```
./trab3 input_5.txt
3
2
3
2
```

Veja os arquivos de entrada para testes na pasta 'in' informada no ambiente classroom. Há instâncias de tamanho 5, 20, e 10^i, para *i* de 2 a 6.

MUITO IMPORTANTE: Neste trabalho não serão fornecidos arquivos com os resultados esperados. Você é responsável por implementar e testar o seu programa, certificando-se de que ele está gerando a resposta correta.

3 Detalhes de implementação

Para cada uma das duas heurísticas básicas (*worst-fit* e *best-fit*) você vai precisar desenvolver uma estrutura de dados eficiente que suporta todas as operações necessárias para se implementar a heurística.

Para *worst-fit*, você certamente vai precisar de algo como *insert* e *delete maximum*, então uma fila com prioridade (*max-heap*) pode ser um bom ponto de partida.

Para *best-fit*, você também vai precisar de *insert* e de um modo eficiente de encontrar o disco mais cheio dentre aqueles que possuem espaço para armazenar um arquivo. Implementar essa operação requer um projeto e implementação cuidadosos para garantir um bom desempenho, principalmente para a instância de tamanho 10^6. Dentre as estruturas que podem ser usadas estão *heaps* e árvores binárias de busca, por exemplo.

Dica para depuração da sua implementação. A soma de todos os tamanhos dos arquivos dividida por 1 milhão dá um limite inferior para o número de discos necessários. (Se sua implementação de alguma heurística der um valor menor, algo certamente está errado.)

Seu programa deve ser, obrigatoriamente, compilado com o utilitário make. Crie um arquivo Makefile que gera como executável para o seu programa um arquivo de nome trab3.

3.1 Passos para o desenvolvimento do trabalho

Você deve realizar todos os passos abaixo, na ordem indicada, para concluir o seu trabalho.

1. Leia atentamente todo esse documento de especificação. Certifique-se de que você entendeu tudo que está escrito aqui. Havendo dúvidas ou problemas, fale com o professor o quanto antes.
2. Implemente e teste a heurística *worst fit*.
3. Implemente e teste a heurística *worst fit* decrescente. Para isso, basta incluir no seu programa uma função de ordenação adequada. Você é livre para usar o algoritmo de ordenação que preferir (inclusive o qsort).
4. Implemente e teste um método força bruta para a heurística *best fit*.
5. Desenvolva uma estrutura de dados eficiente para *best fit* e utilize nas duas heurísticas de *best fit* e *best fit* decrescente.
6. Para as instâncias até 10^5, o seu trabalho deve calcular TODAS as quatro heurísticas em no máximo 1 minuto. Obter um desempenho similar para *best fit* na instância de 10^6 é mais difícil e fica como um desafio.

Algumas considerações finais:

- Ao longo do desenvolvimento do trabalho, certifique-se que o seu código não está vazando memória testando-o com o *valgrind*. Não espere terminar o código para usar o *valgrind*, incorpore-o no seu ciclo de desenvolvimento. Ele é uma ferramenta excelente para detectar erros sutis de acesso à memória que são muito comuns em C. Idealmente o seu programa deve sempre executar sem nenhum erro no *valgrind*.
- ATENÇÃO: Pense bem sobre as suas estruturas e algoritmos antes de implementá-los: quanto mais tempo projetando adequadamente, menos tempo depurando o código depois. Em resumo: se está complicado demais pare e pense no que você está fazendo porque não deveria ser assim, pois é possível implementar esse trabalho tranquilamente em menos de mil linhas de código C. Se você estiver usando muito mais do que isso, algo pode estar errado.

4 Resultados

Após terminar de implementar e testar o seu programa, você deve avaliar os resultados obtidos para os casos de teste e preencher uma tabela de resultados experimentais que também deve ser entregue junto com o trabalho. Veja o arquivo ED2_Trab3_resultados.txt com as informações que devem ser preenchidas.