Erasmo Evangelista de Oliveira

Estudo de um Algoritmo para Montagem Automática de Avaliações Acadêmicas

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtenção do título de Bacharel em Ciência da Computação.

Orientador
Prof. Joaquim Quinteiro Uchôa

Lavras
Minas Gerais - Brasil
2001

Erasmo Evangelista de Oliveira

Estudo de um Algoritmo para Montagem Automática de Avaliações Acadêmicas

Monografia de Graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências da disciplina Projeto Orientado para obtenção do título de Bacharel em Ciência da Computação.

Aprovada em *29 de Julho de 2001*

Prof. Bruno Oliveira Schneider

Prof[a]. Renata Couto Moreira

Prof. Joaquim Quinteiro Uchôa
(Orientador)

Lavras
Minas Gerais - Brasil

Dedico a Deus,
aos meus pais e irmãos,
e aos amigos e colegas de classe.

Agradecimentos

Agradeço aos colegas e professores que me auxiliaram na confecção dessa monagrafia, principalmente a Awdrey e Joaquim pelo apoio irrestrito na formatação desse documento. Agradeço também aos meus pais, que apesar de todas as dificuldades, propiciaram minha formação acadêmica.

Resumo

Nosso projeto propõe um algoritmo que implementa uma solução para montagem automática de avaliações acadêmicas. Com base nesse algoritmo é implementado um aplicativo que dispõe de uma base de dados com questões de diversos pesos que serão sorteadas e incluídas no arquivo final que conterá a avaliação.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Esforços têm sido despendidos por pesquisadores, acadêmicos e professores engajados no estudo da teoria de complexidade computacional, visando obterem implementações razoáveis que solucionem problemas NP-Completos. Dentre os problemas NP-Completos clássicos abordados na literatura especializada encontra-se o problema *Bin Packing*, no qual buscamos a primeira inspiração para a implementação que realizaremos.

O termo *bin packing* tem por significado "empacotamento de caixas". O problema *Bin Packing* consiste em encontrar uma solução para empacotar objetos de diferentes tamanhos em caixas de tamanho fixos (*bins*), utilizando o menor número de caixas.

De maneira mais formal o problema pode ser assim definido:

> "Dados uma seqüência de números a_1 , a_2 ,..., a_n e dois números b e k como entrada. O problema é determinar se o conjunto pode ser particionado em k subconjuntos tal que a soma de números em cada subconjunto é menor ou igual a b"[Manber] p. 357.

A implementação de algoritmos que resolvam o problema *Bin Packing* pode ser realizada por heurísticas clássicas, tais como:

- *First Fit* - o objeto é empacotado na primeira caixa que o caiba;
- *Best Fit* - o objeto é empacotado na caixa que deixa o menor espaço vazio;
- *Decreasing First Fit* - os objetos são ordenados em ordem decrescente e em seguida é utilizado o *First Fit*;

- *Decreasing Best Fit* - os objetos são ordenados em ordem decrescente e em seguida é utilizado o *Best Fit*.

O primeiro foco do nosso estudo será tentar adaptar alguma dessas heurísticas na tentativa de se implementar um aplicativo para montagem automática de avaliações acadêmicas. Para realizar essa tarefa o aplicativo disporá de uma base de dados com questões de diversos pesos que serão sorteadas e incluídas no arquivo final que conterá a avaliação. A totalização dos pesos das questões não deve estar aquém ou além do valor total da avaliação, ou seja a soma dos pesos das questões deve totalizar exatamente 100%.

O segundo foco de nosso estudo é verificar se esse problema de inserir questões de diferentes valores numa avaliação acadêmica pode ser considerado uma variação do problema *Bin Packing* e se essa variação apresenta um maior ou menor grau de complexidade que o *Bin Packing* .

O trabalho está organizado da seguinte forma: no Capítulo 2 é apresentado a descrição do problema, alguns algoritmos e aplicações correlatas, bem como são descritos alguns pontos críticos encontrados na formulação do problema; no Capítulo 3 propõe-se uma implementação para solucionar o problema, bem como apresenta-se a descrição do algoritmo (em pseudo-código) dessa implementação, sendo que o Apêndice A apresenta as classes (em C++) da implementação proposta.

Capítulo 2

O Problema de Montagem de Avaliações Acadêmicas

2.1 Comentários Iniciais

O aplicativo para montagem automática de avaliações acadêmicas terá como entrada uma base de dados com questões de um determinado assunto de determinada disciplina. A saída final será um arquivo texto contendo, na maioria das vezes (senão em todas), uma avaliação diferente a cada execução do programa.

As avaliações serão salvas num arquivo formato texto, onde cada prova deve ser montada com questões sorteadas aleatoriamente da base de dados. A soma dos pesos das questões deverão ser suficientes para cobrir a pontuação exata atribuída a avaliação.

A adaptação do problema do *bin packing* para o aplicativo de montagem automática de avaliações acadêmicas pode ser estabelecida através das seguintes metáforas:

- a avaliação a ser montada corresponde aos *bins* ou caixas do problema *bin packing*.
- as questões correspondem aos objetos a serem empacotados.

Por conseguinte o que deve ser feito é encontrar uma solução para "empacotar" as questões de diferentes pesos em uma única avaliação de tamanho fixo (*bin*). Na adaptação do problema do *bin packing* para o aplicativo de montagem automática de avaliações assumir-se-á que:

- número de caixas disponíveis é de apenas uma "caixa" que corresponde a avaliação a ser montada, onde serão inseridas as questões;
- o tamanho da caixa (avaliação) é de 100;
- os pesos terão intervalos discretos de 5%, ou seja, teremos questões com pesos iguais a 5%, 10%, ... 95%;
- não haverá questões com pesos 0 e 100%;

Dadas as considerações estabelecidas acima, fica evidente a necessidade da reformulação do problema, que agora pode ser assim definido: Seja p_1,p_2,..., p_n um conjunto de números inteiros múltiplos de cinco onde cada um dos números pertença ao intervalo [5,95] e representem os pesos das questões que poderão fazer parte de uma avaliação; deseja-se extrair um subconjunto em que a soma de seus elementos seja exatamente 100.

2.2 Algoritmos para o Problema *Bin Packing*

Atualmente várias aplicações práticas tem sido desenvolvidas baseadas em algoritmos para o problema *Bin Packing* ou em variações do problema como o *cutting stock problem*, uma variação bi-dimensional do *Bin Packing* que consiste em encontrar o melhor arranjo de formas retangulares a serem recortadas de um retângulo, de maneira que os desperdícios sejam evitados, minimizando o número de pedaços não aproveitados e o número de retângulos. É um problema de grande valia na otimização de processos de produção industrial. São exemplos dessas aplicações a automatização do corte de panos em malharias para produção de roupas e a extração de peças de chapas metálicas [German].

Existe também uma inundação de aplicações comerciais que utilizam algoritmos bin packing para escalonamento de processadores, planejamento de programação para emissoras de televisão e estações de rádio [German].

Outro exemplo de aplicação do algoritmo do *bin packing* é em problemas de gerenciamento de memória em que há requisições de diferentes tamanhos de blocos de memória, e os blocos necessitam ser alocados em várias partes da memória [Manber] pág. 364.

As implementações para essas aplicações são construídas das mais variadas formas, tanto em *hardware* como em *software*, porém seguindo os paradigmas

clássicos de algoritmos seqüenciais ou paralelos. Recentemente é bastante comum encontrar projetos que utilizam-se de algoritmos genéticos como por exemplo [Falkenauer] .

2.3 Pontos Críticos

Existem alguns pontos críticos no problema em questão que deverão ser analisados antes de se pensar em qualquer implementação:

- exigência dos pesos das questões totalizarem o exato valor da prova;
- aleatoriedade no sorteio das questões;
- definição de como armazenar e gerenciar a base de dados.

2.3.1 Exigência dos Pesos das Questões Totalizarem o Exato Valor da Prova

Pode-se dizer que a satisfação da exigência dos pesos das questões totalizarem o valor exato da prova é o ponto chave para resolução do problema tratado nesse estudo.

Uma técnica muito utilizada no projeto de algoritmos é a redução de um problema cuja solução é desconhecida, num problema em que sabe-se resolver. Partindo dessa premissa, lançou-se uma primeira tentativa de reduzir o problema em questão ao *problema da mochila* ou *knapsack problem*. Segundo [Manber] pág. 108 , o problema da mochila pode ser definido da seguinte forma:

> Dado um inteiro K e n itens de diferentes tamanhos tal que o i-ésimo item tem tamanho inteiro k_i , encontrar um subconjunto de itens cuja a soma seja exatamente K, ou determinar que tal subconjunto não existe.

Existem diversas instâncias para o problema da mochila, porém uma em especial auxiliará na satisfação da exigência de que a soma dos pesos das questões seja o exato valor da avaliação. Essa instância é denominada problema da partição de um inteiro ou *integer partition* que consiste em gerar partições inteiras de n , tal que os items dessas partições são inteiros diferentes de zero, que somados resultam exatamente no valor de n.

Então, constatou-se que nosso problema poderia ser reduzido ao problema da partição inteira para satisfazer a exigência de que a soma dos pesos das questões seja o exato valor da avaliação.

Todavia, partindo do pressuposto que não faria sentido ter questões com peso menores que cinco, verificou-se que uma boa estratégia seria gerar todas as partições de números inteiros múltiplos de cinco[1] do número 100. Com isso conquistou-se a garantia de que a soma dos pesos das questões totalizaria exatamente 100% .

Para exemplificar o que foi feito, tome como exemplo o número 20. Se quisésemos obter todas as suas partições, em múltiplos de cinco, teríamos o seguinte:

5+5+5+5=20;

10+5+5=20;

10+10=20;

15+5=20;

Vale lembrar que, segundo [Manber] pág.357, ambos os problemas: *bin packing* e *knapsack* são NP-completos e ambos são reduções do problema da partição.

Porém o problema da partição pode ser eficientemente resolvido por alguma variação do algoritmo do problema da mochila se os tamanhos da entrada (número inteiro a ser particionado) são inteiros pequenos. No entanto se o tamanho da entrada é o máximo valor que pode ser representado por um tipo inteiro, tal algoritmo recebe a denominação de *pseudopolynomial algorithm* [Manber] pág.357 e é exponencial em relação ao tamanho da entrada.

Mas essa não é uma preocupação grave para o problema aqui apresentado, pois o número 100 possui 627 partições possíveis, logo não é tarefa difícil obter todas as 627 possibilidades de distribuição de pontos para as questões da avaliação num tempo relativamente curto.

2.3.2 Aleatoriedade

Implementações que requerem seleção aleatória de algum número utilizam, na verdade, a geração de números pseudo-aleatórios; pois esses números são gerados através de algum procedimento determinístico de acordo com algum esquema fixo, dependendo dos passos do procedimento. Se o esquema é completamente determinístico, então os números gerados não podem ser aleatórios no sentido real da palavra [Manber] pág. 160.

[1] Os elementos da partição devem ser múltiplos de cinco porque os pesos das questões são múltiplos de cinco.

Encontrar métodos eficientes para geração de números pseudo-aleatórios de forma a se aproximar ao máximo da aleatoriedade real é um problema que se encontra em aberto e plausível de discussão.

Fugiria do escopo desse estudo qualquer tentativa de implementação de algoritmos de geração números pseudo-aleatórios; logo a aleatoriedade no sorteio das questões será implementada utilizando-se algoritmos de geração de números pseudo-aleatórios existentes.

2.3.3 Armazenamento e Gerenciamento da Base de Dados

Inicialmente estudou-se duas propostas de implementação da base dados. Na primeira proposta a base de dados seria armazenada numa tabela com informações sobre as questões: número da questão, classificação (questão teórica ou prática), peso, assunto e conteúdo. As alterações, inserções e consultas a essa tabela seriam gerenciadas por um SGBD (sistema gerenciador de banco de dados).

Essa primeira proposta foi descartada e adotou-se uma segunda, na qual as informações sobre as questões: total de questões, número da questão, classificação (questão teórica ou prática) , peso e assunto seriam armazenadas em linhas de um arquivo texto e o conteúdo armazenado num arquivo isolado que teria por nome o número da questão.

A segunda proposta foi adotada pois isentaria o sistema da necessidade se instalar um SGDB para gerenciar a base dados. O preço da adoção dessa proposta foi aumento nas linhas de códigos para tratar as operações de atualização e recuperação de informações contidas na base de dados e a dependência da edição manual dos arquivos textos pelo usuário para adicionar, remover ou modificar informações relativas as suas questões.

Capítulo 3

Implementação Proposta

3.1 Comentários Iniciais

A solução encontrada para o problema foi inspirada nos algoritmos *Decreasing First Fit* e *Decreasing Best Fit* nos quais o vetor de objetos é ordenado em ordem decrescente e em seguida inseridos nas caixas conforme as heurísticas *First Fit* ou *Best Fit*.

O vetor de objetos no nosso caso é o vetor de entrada que contém todas as questões da base de dados. Porém esse vetor é ordenado em ordem crescente de pesos e não em ordem decrescente como nos algoritmos *Decreasing First Fit* e *Decreasing Best Fit*. Porém não se tem a preocupação de definir se a questão deve ser inserida na caixa (avaliação) que deixa o menor espaço(algoritmo *Best Fit* ou inserida na primeira caixa (algoritmo *First Fit*), pois dispomos de apenas uma caixa (avaliação).

Depois de ordenado o vetor de pesos é gerado um vetor das partições possíveis do número 100, em seguida essas partições são permutadas dentro do vetor de partições. Após se realizar a aleatorização do vetor de partições procura-se então por uma partição que esteja contida no vetor de pesos das questões.

3.2 Pseudo-Código

Uma vez definida a estratégia de ataque ao problema, necessitou-se elaborar um pseudo-código que definisse em linhas gerais como seria a implementação do aplicativo de montagem automática de avaliações acadêmicas. Pode-se resumir o algoritmo de montagem da prova em cinco passos:

- obter o vetor de questões;
- ordenar as questões;
- obter o vetor de partições;
- aleatorizar o vetor de partições;
- procurar uma partição possível no vetor de questões.

Para um melhor entendimento do algoritmo observe a representação gráfica do pseudo-código para o aplicativo de montagem automática de avaliações na Figura 3.1.

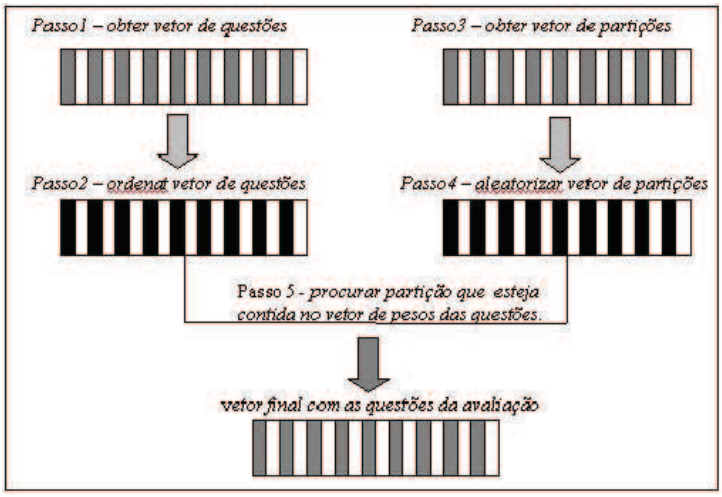

Figura 3.1: Representação gráfica do algoritmo de montagem automática de avaliações

3.3 Refinamentos

O pseudo-código representado acima, foi o pontapé inicial para se definir como seria realizada a implementação da solução para o problema em questão.

Utilizou-se, primeiramente, essa técnica de codificar o algoritmo num pseudo-código mais próximo da linguagem natural e com funções mais globais . Num

segundo momento realizou-se alguns refinamentos *top-down*[1], no intuito de desenvolver um programa mais bem estruturado.

Os refinamentos permitiram uma inserção gradual quanto ao nível de detalhes do algoritmo, facilitando o desenvolvimento de código para os esqueletos das classes, ou seja, realizando a definição de atributos e métodos de encapsulamento privados e desdobramento de código dos métodos das classes.

No refinamento top-down dos passos descritos pelo pseudo-código apresentado no início do capítulo, pôde-se obter algo mais próximo de C++, linguagem na qual o código do aplicativo foi implementado. Nas seções seguintes desse capítulo apresentar-se-á o refinamento de cada passo deste pseudo-código.

3.3.1 Obtenção do Vetor de Questões

Os pesos das questões, bem como os demais atributos: total de questões , número da questão, classificação e assunto são lidos a partir de um arquivo de configuração e armazenados num vetor de objetos da classe **questao**[2].

O pseudocódigo que realiza a leitura dos atributos supra-citados pode ser observado na Figura 3.2

3.3.2 Ordenação do Vetor de Questões

A finalidade de se ordenar o vetor de questões é facilitar a busca de uma questão de determinado peso. A ordenação do vetor de objetos da classe **"questao"** é realizada em ordem crescente de pesos através do algoritmo *quicksort* que tem como complexidade **O(n log n)** no caso médio. Implementou-se uma classe denominada *quicksort* , na qual o principal método é representado pela seguinte assinatura **ordena(int low, int high)**. Os parâmetros *low* e *high* representam os limites inferiores e superiores do vetor a ser ordenado. No arquivo principal do programa foram realizados os seguintes passos:

- chamada do construtor da classe *quicksort* que recebe como parâmetro um vetor de objetos da classe questao;
- chamada do método ordena para o objeto da classe *quicksort*, passando-se como limite inferior 0 e limite superior o total de questões menos 1;

[1] O termo *top-down* é empregado aqui como sendo uma estratégia de se partir de um pseudo-código que trate o problema sob um prisma abrangente e superficial e atinja um algoritmo cujo o pseudo-código possua uma riqueza maior de detalhes na descrição dos procedimentos, se aproximando da linguagem na qual a solução para o problema foi implementada.

[2] vide interface da classe questao na Seção A.1

```
Procedure monta_vet_questoes(arquivo)
   {Input:arquivo={atributos das questões}
   Output:vetor_questoes}
   begin
      arquivo("config");
        //associa ao arquivo uma string ``config'' que definirá seu nome

      if (nao_existe_arquivo)
      begin
         write(mensagem de erro);
         exit;//aborta execução do programa
      end;

      reposiciona_ponteiro(inicio)
        //reposiciona o ponteiro de posição do arquivo no inicio
      read( total_questoes);
        //lê o primeiro token do arquivo e
        //armazena seu valor em total_questoes

       for i:=0 to total_questoes
         //armazena os atributos de cada questão
       begin
          read(arquivo,questao.num_questao);
            //lê do arquivo e armazena atributo num_questao
          read(arquivo,questao.classificacao);
            //lê e armazena classificação
          read(arquivo,questao.peso);
            // lê e armazena peso
          read(arquivo,questao.assunto);
            // lê e armazena assunto
          vetor_questoes[i]=questao;
            // armazena questão
       end;
end;
```

Figura 3.2: Pseudo-código do procedimento de leitura do vetor de questões

3.3.3 Obtenção do Vetor de Partições

A obtenção do vetor de partições é realizada através de um algoritmo recursivo que gera todas as partições possíveis de um inteiro. Conforme citado na Seção 2.3.1, esse algoritmo garante que as distribuições dos pesos das questões totalizem exatamente os 100% de pontos da avaliação. Na função principal (*main*) de nosso aplicativo, primeiramente criou-se um objeto do tipo **particao**, denominado **part**. Em seguida foi executado o seguinte método **part.geraParticao(200,100,0)** para gerar o vetor das partições possíveis do número 100.

O pseudo-código para efetuar a geração de partições é descrito na Figura 3.3

```
Procedure geraParticao
Input:n,k,t
Output:vetor_particoes
begin
     particao[t]=k;
     if (n=k)armazenaParticao(particao);
        //armazena a partição num vetor de partições
     j=min(k,n-k);
        //retorna o menor dentre os dois valores passados como parâmetro
     while (j > 1)
     begin
       geraParticao(n-k,j,t+1);
              //chamada recursiva do procedimento geraParticao
       j:=j-5;
     end;
end;
```

Figura 3.3: Pseudo-código do algoritmo recursivo de geração de partições

3.3.4 Aleatorização do Vetor de Partições

A aleatorização do vetor de partições foi uma estratégia de ataque ao problema que não garantiu a solução ótima, uma vez que o vetor de todas as partições possíveis assume uma configuração fortuita, depois de ser randomizado pela função randomizaParticoes.

A medida que uma partição possível[3] se distanciar das primeiras posições do vetor de partições os tempos de execução do programa aumentaram. Porém a aleatorização é imprescindível para a implementação, por dois motivos: obter configurações de provas diferentes e contar com uma técnica que de alguma maneira permita ao algoritmo encontrar uma partição possível nas posições iniciais do vetor de partições.

A implementação da função de aleatorização está intimamente relacionada ao "grau de aleatoriedade" que lhe é atribuído e de como é implementado o algoritmo de geração de números aleatórios da função *rand()* da biblioteca padrão[4] da linguagem C++ .

O grau de aleatoriedade é obtido através da soma de uma constante (que pertence a entrada do programa) com o relógio do sistema . Esse grau é a entrada de uma outra função da biblioteca padrão do C++ , *srand(n)*, onde n é um valor

[3] Partição possível é aquela que contém pesos de questões que pertença a base de dados.

[4] A função rand gera um inteiro entre 0 e uma constante simbólica definida no arquivo de cabeçalho cstdlib. Esse valor deve ser pelo menos 32767, que é o valor máximo de um inteiro de 2bytes.

inteiro sem sinal que funcionará como semente para aleatorização dos números através da função *rand()*.

A implementação o algoritmo de geração de números aleatórios da função *rand()* gera números pseudo-aleatórios. O pseudo-código para aleatorização do vetor de partições é apresentado na Figura 3.4.

```
Procedure randomizaParticoes
Input:grau_aleat, particoes;
Output:vetor de partições aleatorizado
begin
    srand(time(0)+grau_aleat);//pega o tempo como semente
    i:=0;
    for i to 2*particoes.size();i++)
    begin
      pos1:=(rand()mod particoes.size());
      pos2:=(rand()modparticoes.size());
      tmp:=particoes[pos2];
      particoes[pos2]:=particoes[pos1];
      particoes[pos1]:=tmp;
      i:=i+1;
    end;

end;
```

Figura 3.4: Pseudo-código para aleatorizar vetor de partições

3.3.5 Encontrar Partição Possível no Vetor de Questões

Para encontrar uma partição possível que esteja contida nos pesos de questões disponíveis, elaborou-se um algoritmo (apresentado nas Figuras 3.5 e 3.6) que varre cada elemento de cada partição do vetor de partições e verifica se para tal elemento existe uma questão no vetor de questões com o peso correspondente. Se existir a questão ela é armazenada num vetor de questões que constituirá a avaliação. Caso o elemento da posição seguinte de uma partição não possua elemento correspondente esse vetor é esvaziado e começa-se o processo novamente.

Cabe frisar que no vetor de questões pode haver mais de uma questão com o mesmo peso, logo ao se verificar que um elemento da partição possui uma questão com o peso correspondente, todos as questões com esse peso são armazenadas num vetor auxiliar; em seguida sorteia-se uma dessas questões para fazer parte do

vetor final de questões que constituirá a avaliação.

```
Procedure procuraParticao
Input:vet_ordernado,p  //vet_ordenado corresponde ao vetor de
                       //questões e p ao vetor de partições
                       //geradas
Output:existe_prova //valor booleano
begin
    existe_questao=true;//representa a pertinência de cada
                       //elemento
                       //da partição ao vetor de pesos
    existe_prova=false;//representa a existência de uma partição
                       //que distribuirá os pontos da prova
    i:=0;

    //loop que varre o vetor de partições
    //aleatorizado em busca de uma partição possível
    while(existe_prova=false && i<=626);
    begin
        ptr=vetor_pesos.begin();  //reposiciona o ponteiro
                                  //no início do vet. pesos
        indice_questoes.clear();  //esvazia o vetor de índices
        j=particoes.retornaParticao(i).size()-1;
            //indice recebe tamanho da particao corrente -1

        //loop que varre cada partição a procura de
        //pesos inexistentes
        while(existe_questao==true && j>=0)
        begin
            //procedimentos caso encontre um elemento j
            //qualquer que pertença a partição e ao vet.
            //de pesos
            if (binary_search(ptr,vetor_pesos.end(),
                p.retornaElemento(i,j)))
            begin
                    indices_aux.clear();//esvazia vetor
                                        //de índices auxiliares
```

Figura 3.5: Pseudo-código para encontrar partição possível no vetor de questões - Parte 1

```
                do
                begin
                    //reposiciona ponteiro no elemento encontrado
                    ptr=find(ptr,vetor_pesos.end(),
                                p.retornaElemento(i,j));
                    indices_aux.push_back(ptr
                                        - vetor_pesos.begin());
                        //armazena indice questao
                    ptr++;//muda o ponteiro de posição
                end
                while(find(ptr,vetor_pesos.end(),
                      p.retornaElemento(i,j))
                      <>find(ptr+1,vetor_pesos.end(),
                      p.retornaElemento(i,j)));
                srand(time(0)+j);//semente para aleatorização
                //sorteia um pesos encontrados para elemento
                //da partição e armazena
                indice_questoes.push_back(indices_aux[rand()
                                        mod indices_aux.size()]);
                existe_questao=true;
            end
            else existe_questao=false;//seta questão como
                                      //inexistente no v. de pesos
        j:=j+1;
        end; //fim da varredura de partição

        //caso todos os elementos de determinada partição
        //pertencerem ao vetor de pesos
        if (existe_questao==true)
        begin
            existe_prova=true;//reseta o flag de exist. de prova
            dist_pts=p.retornaParticao(i);//armaz/ part. encont.
            indice_part=i;//armazena o índice da part/ encontrada
        armazenaNum_Questao(indice_questoes, vet_ordenado);
        end
        else existe_questao=true;// reseta o flag de existência
                                 //de questão
        i:=i+1;//incrementa índice do vetor de partições
   end;//fim da varredura do vetor de partições
   return existe_prova;
end;
```

Figura 3.6: Pseudo-código para encontrar partição possível no vetor de questões - Continuação

Capítulo 4

Conclusão

Partimos da hipótese de que o problema de montar avaliações acadêmicas a partir de questões de pesos variados seria uma variação do problema do *bin packing* e verificamos que esse problema se assemelhava mais ao problema da partição de um inteiro (uma variação do problema da partição). Para visualizar essa semelhança procurou-se encarar o problema sob uma ótica diferente da seqüencia natural em que o problema é apresentado. Ao invés de inserir questões até atingir o valor exato da prova, gerou-se todas as distribuições de pontos possíveis e selecionou-se uma distribuição para montar a avaliação acadêmica. Partindo dessa constatação verificou-se que o problema era redutível ao problema do *bin packing* que por sua vez constitui um problema NP-Completo.

Fica evidente que ao aleatorizarmos o vetor de partições no intuito de obtermos avaliações diferentes a cada execução, faz com que os tempos de execução do algoritmo fiquem amarrados a probabilidade de se encontrar uma partição possível no início do vetor de partições.

Não procuramos encontrar uma solução ótima para o problema, apenas apresentamos uma das soluções possíveis.

Apêndice A

Interfaces das Classes

As interfaces das classes implementadas na linguagem C++ são apresentadas nas seções a seguir. Maiores detalhes sobre a implementação podem ser obtidos nos arquivos fontes da implementação.

A.1 Classe “questao”

A classe “questao” apresenta apenas atributos, definindo um molde[1] para cada instância das questões da avaliação. A interface da classe pode ser visualizada na Figura A.1.

```
class questao{
    public:
        int peso;
        char assunto, classificacao;
            string num_questao;
};
```

Figura A.1: Interface da classe “questao”

A.2 Classe “particao”

A classe particao implementa a geração de todas as partições possíveis de um número inteiro [2]. A interface dessa classe pode ser observada na Figura A.2.

[1] Parecido com um registro em Pascal ou uma *struct* em C++

[2] Em nossa implementação esse inteiro corresponde ao número 100

```
class particao
{
   private:
    int p[21];
    vector<int*> particoes;
    int cont;
    int min( int x, int y) ;
    void armazenaParticao(int* particao);
    void imprimeParticao(int*part,int t);
   public:
    Particao();
    void geraParticao (int n, int k, int t);
    void randomizaParticoes();
    ~Particao(){};
};
```

Figura A.2: Interface da classe "particao"

A.3 Classe "quicksort"

A classe quicksort implementa a ordenação de um vetor de objetos da classe questao apresentada na seção A.1. A interface dessa classe pode ser observada na figura A.2

```
class quicksort
{
   private:
    vector <questao> vetor_questoes;
   public:
    Quicksort (vector <questao> questoes);
    void ordena(int low, int high);
    void imprimeVetor (int n);
    ~Quicksort(){};
};
```

Figura A.3: Interface da classe "quicksort"

A.4 Classe "avaliacao"

A classe avaliação implementa a montagem do arquivo contendo a avaliação, procurando uma partição cujos elementos correspondam a pesos de questões existentes no vetor de questões. A interface para essa classe é apresentada na figura A.4.

```
class avaliacao
{
  private:
    int indice_part;
    vector<int> dist_pts;
        // distribuição dos pesos das questões da avaliação
    vector<string> vet_num_quest;
  public:
    bool procuraParticao(vector<questao> vet_ordenado,
                         vector<int>vetor_pesos, Particao p);
    void imprimeDistPontos(vector<int> pesos);
    void imprimeNumQuestoes(vector<string> questoes);
    vector<int> retornaDist_Pontos();
    string retornaNum_Questao(int count,
                         vector<questao> vetor_questoes);
    void armazenaNum_Questao(vector<int> indices,
                         vector<questao> vq);
    vector<string> retornaVet_Num_Questoes();
};
```

Figura A.4: Interface da classe “avaliacao”

Apêndice B

Sites Consultados

1. http://citeseer.nj.nec.com/falkenauer96hybrid.html
2. http://www-di.inf.puc-rio.br/c̃elso/bin-packing.ps
3. http://www.cs.gsu.edu/m̃atskp/Algorithms/NP/node11.html
4. http://secretary.erc.caltech.edu/updates99/billgr/detailpage/binpack.html
5. http://www.mfgconsultants.com/drusrey/research/bin_pack.html
6. http://www.mfgconsultants.com/drusrey/research/bin_pack.html
7. http://www.aridolan.com/ga/gaa/Binpack5_19.html
8. http://www.research.ibm.com/pdos/enabling/ebb/doc/cbppt11.html
9. http://prodlog.wiwi.uni-halle.de/sicup/non_pub/abstracts/waescher_97.html

Referências Bibliográficas

[Manber] Manber, Udi. Introduction to algorithms: A Creative Approach. Addison-Wesley Publishing Company, 1989.

[Falkenauer] Falkenauer E. & Delchambre A. - A Genetic Algorithm for Bin Packing and Line Balancing - Falkenauer, Delchambre (ResearchIndex url: http://citeseer.nj.nec.com/ falkenauer92genetic.html

[German] German, Brandon. Bin Packing Project - Problem, 1998 url: http://cs.ua.edu/reu/1998_stuff/German/problem.html