Mini SISU

Luiz A. D. Berto

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

luizberto@dcc.ufmg.br

Abstract. *This article has the objective of solving problem of stable matching between courses of an university and students, in which each course has a quantity of vacancies, and each student has a grade and presents two courses of his first and second option, respectively. At the end, the list of courses is printed, each one with its passing score, approved list and waiting list.*

Resumo. *Esse trabalho tem como objetivo resolver o problema de emparelhamento estável entre cursos de uma universidade e alunos, no qual cada curso apresenta uma quantidade de vagas, e cada aluno apresenta uma nota e dois cursos de primeira e segunda opção, respectivamente. Ao final, são impressos a lista de cursos, cada qual com sua nota de corte, lista de aprovados e lista de espera.*

1. Introdução

Nesse trabalho, foi abordado o seguinte problema: dada uma lista de m cursos de uma universidade, na qual cada curso c_i possui um nome e q_i vagas, e uma lista de n alunos, na qual cada aluno possui um nome e escolhe dois dos cursos da lista c, de forma que um curso c_j seja sua primeira opção, e um curso c_k seja sua segunda opção, o programa deve imprimir, para cada curso a nota de corte (nota do último aluno aprovado, caso forem aprovados q_i alunos, caso contrário 0), a lista de alunos aprovados e a lista de espera, ambas em ordem de nota, opção, e chegada.

Para tal, o programa utiliza uma adaptação do algoritmo de *Gale-Shapley*. Originalmente, o algoritmo de *Gale-Shapley* resolve o problema de emparelhamento estável, de forma que dois conjuntos s_1 e s_2, cada um com n elementos, de maneira que cada elemento do conjunto s_1 ordena os elementos do conjunto s_2 com uma lista de preferências, e vice-versa.

A adaptação funciona da seguinte maneira: as preferências de cada aluno são expressadas pelos *ids* dos cursos de primeira e segunda opção escolhidos. A preferência de cada curso é expressada da seguinte maneira (em ordem de importância):

- um curso prefere um aluno que tenha uma nota maior;
- um curso prefere um aluno que tenha o escolhido como primeira opção;
- um curso prefere um aluno que tenha se cadastrado no sistema mais cedo.

Com isso, temos os dois conjuntos (representados aqui como c e a), e as preferências, tal como descritas acima. A adaptação utilizada leva em consideração que nem sempre todos os alunos vão ser emparelhados com um curso, e nem sempre um curso preencherá todas as suas vagas.

2. Implementacao

A implementação, conforme requisitado, utiliza listas encadeadas para armazenar os cursos e alunos. Foram utilizadas as classes *Node*, *Lista*, *Aluno* e *Curso*, além de uma exceção definida, *ListaException*.

A classe *Node* define um nó genérico de uma lista duplamente encadeada, tendo os campos *valor*, *proximo* e *anterior*.

A classe *Lista* define uma lista duplamente encadeada genérica, com as seguintes operações:

- *adicionarNoFinal*, que adiciona um elemento na última posição;
- *adicionarAntes* que adiciona antes de um *Node* especificado;
- *remover*, que remove da lista um *Node* especificado. Caso ele exista na lista, lança *ListaException*;
- *vazia*, que é uma função booleana que retorna *true* caso a lista esteja vazia.

A classe *Aluno* define um aluno com as seguintes informações:

- *id* (ordem de entrada);
- *nome*;
- *nota*;
- *codigoPrimeiraOpcao*, que é a ordem de entrada do seu curso de primeira opção;
- *codigoSegundaOpcao*, que é a ordem de entrada do seu curso de segunda opção;
- *aprovadoPrimeiraOpcao*, que é um booleano que define se o aluno foi aprovado no seu curso de primeira opção;
- *aprovadoSegundaOpcao*, que é um booleano que define se o aluno foi aprovado no seu curso de segunda opção;
- *aplicouSegundaOpcao*, que é um booleano que define se o aluno chegou a se candidatar no seu curso de segunda opção;
- *possuiEsperancaAprovacao*, que é uma função booleana que retorna *true* caso o aluno não tenha sido aprovado em sua primeira opção e não tenha candidatado na segunda opção.

Já a classe *Curso* define um curso com os seguintes dados e funções:

- *id* (ordem de entrada);
- *nome*;
- *quantidadeVagas*, tal como especificada na entrada;
- *notaCorte*, que é a nota do último aluno da lista de aprovados, caso ela tenha *quantidadeVagas* alunos, caso contrário é 0;
- *listaIntermediaria*, que é a lista na qual os alunos são adicionados a cada iteração do algoritmo;
- *aprovados*, que é a lista final de aprovados do curso;
- *listaEspera*, que é a lista de espera do curso;
- *adicionarAlunoListaIntermediaria*, que adiciona um aluno na lista intermediaria;
- *adicionarAlunoListaEspera*, que adiciona um aluno na lista de espera;
- *processarListaIntermediaria*, que marca os *quantidadeVagas* primeiros alunos da *listaIntermediaria* como aprovados em primeira ou segunda opção (a depender de qual opção o curso é para aquele aluno), e remove os demais, os marcando como não aprovados em primeira ou segunda opção (novamente, a depender de qual opção o curso é para aquele aluno);

- *consolidarListaAprovados*, que transforma a lista intermediária na lista de aprovados, e atualiza a nota de corte caso necessário.

No arquivo *main.cpp* estão definidas as principais estruturas lógicas do programa. A função *lerEntrada* lê as informações necessárias para a execução do programa de *stdin*. Os cursos e alunos são colocados no final da respectiva lista encadeada.

Após a leitura, é chamada a função processar, que de fato implementa a adaptação do algoritmo de *Gale-Shapley*. Ela funciona da seguinte maneira:

1. Para cada aluno, o coloca em uma lista intermediária de seu curso de primeira opção. A inserção nessa lista é feita em ordem. Essa decisão foi tomada pois para cada curso o critério de desempate é diferente, porque depende da comparação da primeira e segunda opção do aluno com o *id* do curso, o que não poderia ser feito dentro da classe *Aluno*, e por consequência, numa lista genérica;
2. Para cada curso, a lista intermediária é processada. Os q_i primeiros alunos são mantidos na lista intermediária, e a variável *aprovadoPrimeiraOpcao* é setada como *true*, e os demais são removidos. Para os alunos removidos, a variável *aprovadoPrimeiraOpcao* é setada como false, e a variável rejeitado é setado como *true*;
3. Após isso, são filtrados apenas os alunos que não foram aprovados em primeira opção para que eles sejam colocados em ordem na lista intermediária de seu curso de segunda opção;
4. Novamente, para cada curso, a lista intermediária é processada. De maneira análoga à iteração anterior, os q_i primeiros alunos são mantidos na lista intermediária, e a variável *aprovadoSegundaOpcao* é setada como *true*, e os demais são removidos. Para os removidos, a variável *aprovadoPrimeiraOpcao* ou *aprovadoSegundaOpcao*, dependendo de qual for o caso, é setada como *false*, e a variável *rejeitado* é setada como *true*;
5. Dessa vez, são filtrados os alunos que foram rejeitados, para que novamente eles sejam colocados em ordem na lista intermediária de seu curso de segunda opção. Esse passo é necessário pois, um aluno que foi aprovado em primeira opção e foi removido da lista intermediária durante o passo 4 não teve a oportunidade de entrar na lista intermediária do seu curso de segunda opção;
6. O passo 4 é repetido, e com isso finaliza-se o emparelhamento estável.

Após isso, a função *atribuirListasDeEspera* é chamada, e para cada aluno que não foi aprovado em primeira opção, ela o coloca na lista de espera de seu curso de primeira opção, e caso ele também não tenha sido aprovado em segunda opção, ele é colocado na lista de espera de seu curso de segunda opção. Ao final, a função *consolidarAprovados* é chamada, na qual, para cada curso, a lista de aprovados é consolidada, apontando para a lista intermediária (note que, nesse ponto, a lista intermediária do curso c_i tem no máximo q_i alunos. Caso a lista de aprovados tenha q_i alunos, a nota de corte é atribuída como a nota do último aluno; caso contrário, ela se mantém em 0. Com isso, a lista de aprovados e de espera de todos os cursos está consolidada, e a saída é imprimida de acordo com a especificação.

3. Análise de Complexidade

Seja m o número de alunos, e n o número de cursos. O laço while que verifica se ainda existem alunos com esperança de aprovação, no pior caso, é executado no máximo m

vezes.

Para cada iteração, o método *popularListasIntermediarias* itera sobre a lista de alunos ($O(m)$). A cada iteração, o custo para encontrar o curso de primeira (ou segunda) opção do aluno é de $O(n)$, e após encontrado, o aluno será inserido em ordem na lista intermediária. Para todas as iterações, esse custo será de $O(m^2n)$.

O processamento da lista intermediária é $O(m)$, pois nele a lista é percorrida, e os q_i primeiros alunos são mantidos, enquanto o restante é marcado como não aprovado.

O filtro da lista de alunos que ainda possuem esperança de aprovação é $O(n)$, pois a lista de alunos é percorrida por completo uma única vez.

Portanto, como a função que domina o custo é *popularListasIntermediarias*, que é $O(m^2)$, e ela está no interior do while que é $O(m)$, o custo total do programa é $O(m^3n)$

4. Conclusão

Conclui-se que a dependência de um contexto de curso para ordenar as listas de alunos explicita que listas encadeadas não são a estrutura de dados adequada para esse problema, uma vez que a inserção em ordem de n elementos custa $O(n^2)$, e em uma árvore binária, por exemplo, custaria $O(n \times log(n))$. Conclui-se também que, a complexidade e sofisticação do algoritmo de emparelhamento estável é muito alta, o que ofuscou um dos objetivos do trabalho, que era avaliar o conhecimento de listas encadeadas (que se mostraram ineficientes para a resolução do problema), e que, principalmente, fez com que a relação entre o tempo investido no trabalho e os 10 pontos oferecidos fosse desbalanceada.

References

Gale, David, and Lloyd S. Shapley. ”College admissions and the stability of marriage.” The American Mathematical Monthly 69.1 (1962): 9-15.