Documentação TP1

Thiago Martin Poppe

14/04/2019

1 Uma breve descrição

1.1 Problema proposto

Implementar um algoritmo para o novo sistema de vagas do Sisu (Sistema de Seleção Unificada) capaz de encontrar o melhor jeito de alocar os candidatos (com sua nota do Enem e a lista de preferência) nas universidades, dado a nota de corte e o número de vagas.

1.2 Requisitos

- O algoritmo não deve gerar pares instáveis, isto é:
  - Existir um candidato c_1 alocado na universidade u_1 e um candidato c_2 alocado na universidade u_2, pórem o candidato c_1 prefere u_2 e o candidato c_2 prefere u_1.
  - Existir um candidato c_1 alocado em u, porém um outro canditato não alocado possui nota maior do que c_1 e tem interesse em u.
  - Existir um canditato alocado em u_2, porém ele possui uma preferência maior sobre u_1 (que possui vagas) e ele atende a nota mínima.
  - Existir um candidato sem alocação, porém existe vagas em uma universidade de seu interesse e o mesmo atende a nota mínima.
- O algoritmo deve obter a alocação que é simultaneamente a melhor possível para todos os candidatos (com base nas restrições citadas).

2 Estruturas de Dados e Algoritmos

2.1 Estruturas de Dados utilizadas

Para armazenar as universidades, canditatos e as preferências de cada canditato, utilizei vetores de nomes $univs$, $cands$ e $pref$, respectivamente. Para armazenar os canditatos alocados em cada universidade, utilizei um outro vetor, de nome $vetor\_cand$, que armazena ponteiros para os respectivos candidatos. Quando a universidade estiver cheia, esse mesmo vetor virará um heap binário, mais especificamente um min-heap.

Tal TAD foi escolhido visto que, uma vez montado, o min-heap gastará um custo $log(n)$ para inserir um novo candidato com uma nota melhor, pois o mesmo sempre será uma árvore balanceada. Como o candidato com a nota mais baixa estará sempre na raiz do heap, será bem fácil remover o mesmo e inserir um novo candidato com uma nota maior. Essa atualização fará com que o 2º candidato com menor nota fique na raiz do heap e assim sucessivamente.

2.2 Algoritmos utilizados

Nessa seção, darei mais ênfase nos algoritmos voltados para a construção e atualização do nosso min-heap. Os algoritmos explicados serão o de construção de um min-heap (chamaremos de $buildMinHeap$) e o de transformar uma sub-árvore em um min-heap (chamaremos de $minHeapfy$). A seguir, teremos o pseudo-código de cada um deles, seguidos de uma explicação sobre os mesmos.

Algorithm 1 Transformar sub-árvore em um min-heap válido

```
1: procedure MINHEAPFY(heap, n, i)
2:     l ← 2 * i + 1
3:     r ← 2 * i + 2
4:     min ← i
5:     if l < n and heap[l] < heap[min] then
6:         min ← l
7:     if r < n and heap[r] < heap[min] then
8:         min ← r
9:     if min ≠ i then
10:        swap(heap, min, i)
11:        minHeapfy(heap, n, min)
```

Algorithm 2 Gerar um min-heap válido

```
1: procedure BUILDMINHEAP(heap, n)
2:     for i ← (n/2) − 1 downto 0 do
3:         minHeapfy(heap, n, i)
```

Primeiramente, devemos notar que o nosso heap está sendo tratado como se fosse um vetor de 1D. Sendo assim, temos a seguinte propriedade: o filho da esquerda de um nó i estará no índice $2 * i + 1$ e o da direita estará no índice $2 * i + 2$. Com isso, temos que os nós folhas (ou seja, aqueles que não possuem filhos) estarão da metade do nosso vetor em diante. Isso explica porque começamos com $i = (n/2) - 1$ e decrescemos o mesmo até $i >= 0$ no loop presente na função buildMinHeap.

Um min-heap válido segue a propriedade de que o nó pai sempre será menor do que seus filhos. Com isso, durante a execução do algoritmo minHeapfy, estamos sempre escolhendo o elemento que satisfaz essa propriedade e trocando o nó pai de lugar com o mesmo. Visto que um heap nada mais é do que uma árvore binária que sempre será balanceada, temos que

a complexidade desse algoritmo é da ordem de $log(n)$ operações, pois essa é a altura de uma árvore balanceada. Como chamamos esse algoritmo para n/2 elementos, temos que no total, nosso algoritmo é da ordem de $O(n * log(n))$.

3 Análise de Complexidade

3.1 Complexidade Temporal

Apesar de usar uma estrutura de dados com custo temporal $O(n * log(n))$, ainda assim teremos uma complexidade temporal total equivalente a $O(n * m)$, onde n representa o número de candidatos e m o número de universidades. Para um caso em particular onde $n = m$, teremos um custo $O(n^2)$, porém vale ressaltar que m pode ser maior que n, logo, é mais correto dizer que nosso algoritmo possui custo $O(n * m)$.

Ao analisarmos o pior caso, entendemos porque ele será $O(n * m)$. Primeiramente, suponha um caso onde cada candidato c_i, para $i = 1, 2, ..., n$, esteja interessado em todas as universidades u_j, para $j = 1, 2, ..., m$, de tal forma que nenhum candidato possua a nota mínima para entrar em nenhuma universidade. O nosso algoritmo deve verificar para cada candidato a sua lista de preferência de tamanho m para no final concluir que esse candidato não estará alocado. Com isso, teremos um custo final igual a $O(n * m)$.

3.2 Complexidade Espacial

Para armazenarmos as universidades usamos um vetor de 1D, que terá tamanho máximo m, sendo m o número de universidades no total. O mesmo raciocínio será usado para armazenar os candidatos, sendo o tamanho desse vetor igual a n.

Cada universidade irá possuir um outro vetor de candidatos alocados, sendo esse de tamanho máximo α, onde α é o número de vagas dessa universidade. Visto que, apenas podemos alocar um candidato em uma universidade, teremos no total $\sum_{i=1}^{m} \alpha_i = n$, sendo esse o caso onde todos candidatos foram alocados.

Cada candidato irá possuir um vetor de preferências, com tamanho máximo $\beta = m * n$. Este tamanho corresponde ao caso onde todos os candidatos estejam interessados em todas as universidades.

$\therefore$ Temos que a complexidade espacial resultante será $O(m + 2n + \beta) = O(\beta) = O(m * n)$.

4 Avaliação Experimental

Todas análises foram feitas usando arquivos com 10 candidatos e 8 universidades.

4.1 Configuração inicial

Inicialmente, escolhemos números aleatórios para a ordem de preferência de cada candidato, número de vagas total de cada universidade e para as notas mínimas. Com esses valores iniciais obtemos os seguintes resultados:

Média da taxa de alocação das universidades: 0.67

Média da satisfação dos candidatos: 0.59

4.2 Aumento no tamanho da lista de preferências

Ao ampliarmos o tamanho da lista de preferências de cada candidato, obtemos os seguintes resultados:

Média da taxa de alocação das universidades: 0.67

Média da satisfação dos candidatos: 0.54

Podemos observar que nesse caso, houve uma diminuição na média da satisfação dos candidatos e não ocorreu nenhuma mudança na média da taxa de alocação das universidades. Isso se deve ao fato de que um candidato possuia uma preferência alta para uma universidade, porém, foi trocado por um outro candidato que possuia uma nota maior.

4.3 Aumento no tamanho do número de vagas

Ao ampliarmos o número de vagas de cada universidade, obtemos os seguintes resultados:

Média da taxa de alocação das universidades: 0.20

Média da satisfação dos candidatos: 0.83

Podemos observar nesse caso, que um aumento considerável do número de vagas aumenta a média da satisfação de cada candidato. Porém, vale notar que uma vez que todos os candidatos estão alocados, aumentar o número de vagas só irá resultar em uma taxa de alocação menor. Tal caso pode ser observado nesse resultado, onde obtemos uma taxa muito baixa se comparada à inicial.

4.4 Diminuição da nota mínima

Ao diminuirmos as notas mínimas de cada universidade, obtemos os seguintes resultados:

Média da taxa de alocação das universidades: 0.87

Média da satisfação dos candidatos: 0.68

Podemos observar nesse caso, que uma diminuição considerável das notas mínimas aumenta a média da satisfação de cada candidato e a média da taxa de alocação. Isso era de se esperar, visto que, ficará mais fácil para os candidatos entrarem nas universidades e as mesmas terão mais vagas ocupadas.

4.5 Aumento da nota mínima

Ao aumentarmos as notas mínimas de cada universidade, obtemos os seguintes resultados:

Média da taxa de alocação das universidades: 0.60

Média da satisfação dos candidatos: 0.45

Podemos observar nesse caso que, um aumento considerável das notas mínimas diminui a média da satisfação de cada candidato e também a média da taxa de alocação. O resultado era previsível, visto que, ficará mais difícil para os candidatos entrarem nas universidades e as mesmas terão menos vagas ocupadas.

4.6 Considerações finais

É importante ressaltar que tais resultados foram obtidos para uma configuração específica. Fazendo observações com outros arquivos, obtivemos resultados diferentes. Por exemplo, com um aumento do tamanho da lista de preferência dos candidatos, obtivemos também um aumento na satisfação e na taxa de alocação. Optei por mostrar apenas um dos resultados para não tornar essa parte muito massiva.

5 Como usar o Makefile

O código está dividido em algumas pastas para manter a organização e manutenção do mesmo. Fora dessas pastas, possuimos um Makefile que irá compilar o nosso programa atráves do comando *make compile*. Ele irá gerar um executável *tp*1 fora das pastas. Durante a compilação, o Makefile cria uma pasta auxiliar *build* que irá conter os arquivos .o para a compilação. Caso queira excluir essa pasta, use o comando *make clean*.

Obs.: Esse Makefile é uma adaptação de outro Makefile que utilizamos em PDS2 para compilar arquivos em C++.