Nome:
Sandro Rudiero Saibro Viégas

Universidade Federal do Rio Grande do Sul
INF01203 - Estrutura de Dados

1 Introdução

O objetivo do trabalho é apresentar um programa que calcule, de forma eficiente, o coeficiente de Jaccard entre dois textos de tamanhos arbitrários, usando estruturas de dados vistas em aula. Para isso, foi implementada a árvore binária de pesquisa AVL, que armazena a lista de stopwords e as palavras que aparecem nos textos.

2 Árvores AVL

As árvores AVL são árvores binárias de pesquisa balanceada, nas quais a altura da sub-árvore esquerda de cada nó difere da altura da sub-árvore direita em no máximo 1. Essa estrutura foi utilizada devido à alta eficiência em suas operações de inserção e busca, que possuem complexidade $O(\log n)$ no pior caso, fornecendo um bom desempenho até para documentos muito grandes.

2.1 Operações implementadas

2.1.1 Operações básicas

1. **Inserção de palavra na árvore:** a função `AVL_insert` insere uma palavra na árvore e realiza a rotação adequada para mantê-la balanceada. O tempo de execução da inserção de um elemento na AVL é da classe $O(\log n)$, ja que uma árvore completamente balanceada com n nodo possui no máximo altura $\log n$. A operação começa por inserir recursivamente o novo nodo na árvore como em uma ABP tradicional:

```
Se raiz não existe:
    Cria nodo
    Retorna ponteiro para nodo
Senão:
    Se novo nodo < raíz:
        Insere novo nodo na subárvore esquerda
    Se novo nodo > raíz:
        Insere novo nodo na subárvore direita
    Se novo nodo = raíz:
        Termina o algoritmo
```

   Após inserir o nodo na árvore, a função calcula a altura (maior entre as alturas dos dois filhos) e o fator de balanceamento da raíz e realiza rotações, caso necessário, de acordo com a posição na qual o nodo foi inserido:

```
Se novo nodo inserido à esquerda do filho esquerdo da raíz
(nodo < f.esquerdo e FB > 1):
    Rotação à direita na raíz
Se novo nodo inserido à direita do filho direito da raíz
(nodo > f.direito e FB < -1):
    Rotação à esquerda na raíz
Se novo nodo inserido à direita do filho esquerdo da raíz
(nodo > f.esquerdo e FB > 1):
    Rotação à esquerda no filho esquerdo da raíz
    Rotação à direita na raíz
Se novo novo inserido à esquerda do filho direito da raíz:
(nodo < f.direito e FB < -1)
    Rotação à direita no filho direito da raíz
    Rotação à esquerda na raíz
```

2. **Busca de palavra na árvore:** o procedimento `AVL_search` realiza uma busca recursiva na árvore, comparando o elemento procurado com os nodos da árvore, de forma semelhante ao algoritmo de inserção. Caso a palavra do nodo atual seja igual à palavra procurada, a função retorna `true`.

2.1.2 Operações internas

O balanceamento de uma árvore AVL é feita através de operações internas de rotação em nós antecessores ao que foi inserido. As rotações duplamente à esquerda e duplamente à direita são compostas por combinações das duas rotações simples abaixo, e portanto não precisam ser implementadas explicitamente. A rotação duplamente à direita é uma rotação à esquerda no filho esquerdo e uma rotação à direita no nó raíz da subárvore. A rotação duplamente à esquerda é uma rotação à direita no filho direito e uma à esquerda no nó raíz.

1. **Rotação à direita:** essa rotação é aplicada em um nó raíz p com o seguinte algoritmo:

```
z ← filho direito de p
T2 ← filho esquerdo de z
filho direito de p ← T2
filho esquerdo de z ← p
Atualiza altura de p
Atualiza altura de z
Retorna ponteiro para z
```

2. **Rotação à esquerda:** essa rotação é aplicada em um nó raíz p com o algoritmo:

```
u ← filho esquerdo de p
T2 ← filho direito de u
filho esquerdo de p ← T2
filho direito de u ← p
Atualiza altura de p
Atualiza altura de u
Retorna ponteiro para u
```

2.1.3 Operações específicas

1. **Criação de AVL a partir de um documento de texto:** a função `AVL_from_file` percorre um documento de texto passado como ponteiro para `FILE` e cria uma AVL com palavras únicas que não estão presentes em uma dada árvore de stopwords, salvando o comprimento da AVL criada na variável `len`.

2. **Criação de AVL a partir de uma lista de stopwords:** a função `AVL_from_file_stopwords` funciona de maneira semelhante à função descrita acima, mas sem fazer comparações adicionas e armazenar o comprimento da árvore.

3. **Cálculo da intersecção entre duas árvores:** a função `intersection` percorre a AVL `A` recursivamente, com um caminhamento central à esquerda, incrementando o valor de `accumulator` quando o valor do nó de `A` estiver presente na AVL `B`.

4. **Cálculo do coeficiente de Jaccard:** a função `jaccard` utiliza a função de cálculo da intersecção para adquirir os valores de $|A \cap B|$ e $|A \cup B| = |A| + |B| - |A \cap B|$ e calcular o coeficiente de Jaccard

$$J = \frac{|A \cap B|}{|A \cup B|}.$$

2.2 Vantagens da estrutura

Algumas vantagens das árvores AVL, em relação a outras estruturas são:

- **Tempo de busca e inserção:** as operações de inserção e busca de árvores AVL possuem complexidade logarítmica no pior caso, deixando essa estrutura muito mais eficiente que árvores binárias de pesquisa não autobalanceáveis e listas encadeadas. Essa eficiência é resultado do algoritmo de balanceamento.

- **Balanceamento automático:** as operações internas de rotação garantem que a árvore se mantenha sempre equilibrada e evitam completamente o problema do desbalanceameto progressivo. Além disso, as operações de rotação da estrutura, utilizadas no algoritmo de balanceamento, possuem complexidade $O(1)$ e não apresentam impactos significativos no tempo de execução de cada inserção.

- **Menor espaço em memória:** em comparação a outras árvores balanceadas, como as rubro-negras, as árvores AVL são levemente mais eficientes em gerenciamento de memória, assim que precisam armazenar apenas as informações básicas de cada nodo (filhos, dados e altura).

2.3 Desvantagens da estrutura

Apesar das múltiplas vantagens de se utilizar uma árvore AVL, ela também possui algumas desvantagens, como:

- **Dificuldade de implementação:** a função de inserção da AVL, em conjunto às suas operações de rotação, pode ser mais complexa em comparação a outras estruturas de dados, como listas encadeadas. Essa complexidade pode gerar resultados inesperados, como a árvore não ser balanceada corretamente se não houver cuidado suficiente durante a implementação. Portanto, a implementação incorreta da árvore AVL pode ter um impacto significativo no tempo de execução do programa e deve ser evitada com um planejamento cauteloso.

- **Custo em conjuntos pequenos:** como as operações internas da estrutura possuem tempo de execução constante, elas não representam um problema considerável para programas que utilizam com frequência um número grande de elementos, como é o caso do programa que calcula o coeficiente de Jaccard, que pode receber documentos com milhões de palavras. Entretanto, para programas que lidam com conjuntos de dados pequenos, o tempo utilizado nas rotações pode se acumular e gerar um impacto negativo na execução total do algoritmo.

2.4 Comparação com outras estruturas vistas

Para avaliar mais aprofundadamente as vantagens e desvantagens das árvores balanceadas AVL, é importante compará-las com outras estruturas que foram estudadas na disciplina. As estruturas que serão comparadas são: arrays, listas encadeadas, árvores binárias de pesquisa desbalanceadas, árvores splay e árvores rubro-negras.

2.4.1 Arrays

Os *arrays*, ou vetores, são estruturas de armazenamento de dados pré-definidas na linguagem C, e portanto necessitam de poucas definições no desenvolvimento do programa. Entretanto, ao contrário das árvores de pesquisa, os vetores não são versáteis em relação ao tamanho de seu conjunto de dados. Dessa forma, para um projeto que manipula conjuntos de tamanhos arbitrários e/ou mutáveis, os vetores são uma péssima escolha.

2.4.2 Listas simplesmente encadeadas

Diferente dos *arrays*, as listas encadeadas possuem tamanho mutável e baixa complexidade de implementação, configurando uma boa possibilidade em projetos que envolvem o armazenamento de documentos de tamanhos arbitrários. Além disso, para um conjunto de dados que será percorrido linearmente durante a execução do programa, como o primeiro documento no cálculo do coeficiente de Jaccard, a estrutura pode ser mais eficiente que uma AVL. Isso acontece porque é possível armazenar um ponteiro para o último elemento da lista e inserir novos elementos a partir desse ponteiro em tempo $O(1)$. Como temos, entretato, a restrição de que a lista não pode armazenar palavras repetidas, faz com que essa prática seja ineficaz, já que precisaríamos buscar cada elemento na lista antes da inserção.

No caso de precisarmos buscar um elemento na lista antes de inserí-lo, a operação de busca teria, no caso médio, complexidade $O(n)$, que é mais ineficiente que a inserção em uma árvore binária de pesquisa balanceada, como a AVL.

2.4.3 Árvores binária de pesquisa não-autobalanceáveis

A diferença marcante entre uma árvore AVL e uma ABP tradicional, é que a primeira está sempre balanceada, ou seja, todas as operações são executadas no melhor caso. Isso implica que as árvores binárias de pesquisa não-autobalanceáveis podem se aproximar de uma lista simplesmente encadeada, dependendo da sequência de inserções feitas na árvore. Dessa forma, uma ABP pode acabar realizando operações com complexidade $O(n)$, enquanto as AVL sempre executarão inserções e buscas com tempo de execução $O(\log n)$.
Para casos menores, a implementação de uma ABP pode ser mais apropriada por conta de sua simplicidade. Entretanto, para esse projeto, que envolve a criação e análise de estruturas de dados que podem armazenar milhões de elementos, a AVL é mais indicada.

2.4.4 Árvores Splay

As árvores Splay são, assim como as AVL, árvores binárias de pesquisa autobalanceáveis. Embora árvores Splay também sejam autobalanceáveis, elas são menos rígidas que as AVL, e fazem modificações dinâmicas na árvore que favorecem buscas e inserções de elementos semelhantes. Dessa forma, para algoritmos que fazem bom proveito de alguma ordem específica ou que manipulam conjuntos de dados com muitas repetições, a Splay é mais eficiente que a AVL.
Como os textos fornecidos não possuem nenhuma ordem lexicográfica clara, as operações de inserção e busca poderiam acabar sendo menos efetivas que as da AVL. Portanto, para esse caso específico, as árvores Splay não trazem nenhum benefício considerável em relação á árvores AVL ou árvores rubro-negras.

2.4.5 Árvores rubro-negras

As árvores rubro-negras são outra classe de árvores ABP autobalanceáveis, onde cada nó armazena uma informação a mais que outras estruturas: sua cor. Essas árvores possuem um critério de balanceamento menos restrito que as árvores AVL, e por isso podem ser mais eficientes em programas específicos que fazem mais operações de inserção e exclusão do que operações de busca. Entretanto, como é necessário armazenar uma informação adicional em cada nó, sua eficiência de memória é pior que a de árvores AVL. Em um programa que armazena textos com até milhões de palavras, essa propriedade pode se revelar problemática e o uso de uma AVL traz mais segurança de que não haverá ocupação desnecessária de memória.

3 Conclusão

Em geral, a árvore AVL mostrou-se a escolha mais adequada para armazenar as palavras distintas de três documentos com tamanhos arbitrários. O planejamento resultou, assim, em um programa capaz de calcular eficientemente a intersecção e união entre dois textos e consequentemente a similaridade textual entre eles.