Trabalho Prático 2

Sugestões Doodle

1. Introdução

O trabalho tem por objetivo solucionar o problema de erros de digitação dos usuários do Doodle. As informações fornecidas para a solução do problema são: a palavra digitada pelo usuário, as palavras de um dicionário e um número que representa no máximo o quão “diferente” uma sugestão pode ser da palavra digitada pelo usuário. Através destas informações, devemos dar sugestões de palavras do dicionário que são parecidas com a palavra digitada.

A solução implementada utiliza o conceito de distância mínima de edição. A distância mínima de edição é a quantidade de operações mínima para transformar uma palavra A em outra palavra B. As operações disponíveis são: inserção, substituição e remoção de um caractere, sendo que cada operação contribui em 1 na distância. Para cada palavra recebida do dicionário tal distância é calculada, e no final são listadas as palavras que têm a distância menor ou igual ao limite estabelecido.

2. Modelagem do problema

O armazenamento do dicionário pode ser feito em uma estrutura que guarda a palavra digitada pelo usuário, todas as palavras e suas respectivas distâncias. A construção desta estrutura pode ser feita à medida que as entradas vão sendo coletadas. O maior problema em questão é: como calcular a distância mínima de edição?

Força bruta

Uma possível solução para o problema é calcular com força bruta. Neste paradigma de resolução, precisamos verificar todas as possíveis distâncias de edição da palavra original para a palavra de destino, sendo que cada verificação precisa acontecer somente uma vez.

Precisamos alcançar a palavra de destino de todas as formas possíveis, adotaremos a seguinte estratégia: operamos sobre a última letra da palavra de origem e caminhamos para a esquerda, até chegarmos à primeira letra. Para que este algoritmo funcione, vale pensar um pouco sobre como fazer operações que não nos distanciam da solução e nem nos façam repetir verificações:

- *Remoção*: Remover um caractere à direita de onde estamos nunca fará sentido em nossa estratégia. Isso significaria que estaríamos alterando as distâncias que já calculamos previamente, e assim precisaríamos de calculá-las novamente. Remover um caractere à esquerda significaria deixar de testar outras operações como substituição e inserção naquela posição, e assim não estaríamos testando todas as

possibilidades. Assim, em nossa estratégia **só faz sentido removermos o caractere atual da palavra de origem.**

- *Inserção*: Como estamos caminhando da direita para a esquerda, esperamos que alguma hora iremos chegar ao início da palavra. Para garantir que isso sempre aconteça, não vamos inserir nenhuma letra à esquerda. **Sempre vamos inserir letras à direita**, e mais: vamos **inserir sempre última letra da palavra de destino**. A inserção pode ser interpretada como sendo a remoção da última letra da palavra de destino, porque estaremos eliminando-a de ser inserida novamente nas execuções seguintes.
- *Substituição*: Consideramos que **a substituição de uma letra sempre é feita pela letra última letra da palavra de origem**. Caso contrário, estaríamos substituindo por uma letra que não contribui em nada para deixar a palavra de origem mais perto da palavra de destino, e assim teríamos que operar sobre esta letra mais vezes para caminharmos para uma solução. A operação de substituição pode ser considerada como a remoção das últimas letras das duas palavras, porque estaremos eliminando a possibilidade da mesma letra da palavra de destino ser utilizada novamente nas execuções seguintes e não precisamos mais operar sobre a letra da palavra de origem. **A substituição terá custo 0 caso as últimas letras sejam iguais.**

Agora temos operações que contribuem para o nosso algoritmo. Com estas operações pertinentes, temos a garantia de que nosso algoritmo sempre para quando chegarmos ao início da palavra de origem ou ao início da palavra de destino, que consideramos como uma palavra vazia. Quando chegarmos neste ponto caso haja letras restantes em uma das palavras, significa que precisamos fazer a mesma quantidade de operações que a quantidade de letras restantes. Vamos analisar o que acontece com a execução deste algoritmo com a palavra de origem sendo AB, e a palavra de destino sendo CD:

- A**B** - C**D:** Início
  - **A** - C**D:** Remoção (+1)
    - **(vazio)** - C**D:** Remoção (+1) e Fim (+2)**. (D = 4)**
    - **A** - **C:** Inserção (+1)
      - **(vazio)** - **C:** Remoção (+1) e Fim (+1)**. (D = 4)**
      - **A** - **(vazio):** Inserção (+1) e Fim (+1)**. (D = 4)**
      - **(vazio)** - **(vazio):** Substituição (+1) e Fim (+0)**. (D = 3)**
    - **(vazio)** - **C:** Substituição (+1) e Fim (+1)**. (D = 3)**
  - A**B** - **C:** Inserção (+1)
    - **A** - **C:** Remoção (+1)
      - **(vazio)** - **C:** Remoção (+1) e Fim (+1)**. (D = 4)**
      - **A** - **(vazio):** Inserção (+1) e Fim (+1)**. (D = 4)**
      - **(vazio)** - **(vazio):** Substituição (+1) e Fim (+0)**. (D = 3)**
    - A**B** - **(vazio):** Inserção (+1) e Fim (+2)**. (D = 4)**
    - **A** - **(vazio):** Substituição (+1) e Fim (+1)**. (D = 3)**
  - **A** - **C:** Substituição (+1)
    - **(vazio)** - **C:** Remoção (+1) e Fim (+1)**. (D = 3)**
    - **A** - **(vazio):** Inserção (+1) e Fim (+1)**. (D = 3)**
    - **(vazio)** - **(vazio):** Substituição (+1) e Fim (+0)**. (D = 2)**

Encontramos 2 como sendo a distância mínima de edição de AB para CD, destacada em vermelho. Destacado em amarelo estão **situações que compartilham o mesmo estado das duas palavras** ao longo da execução e em diferentes lugares. Esta é uma característica que nos dá a oportunidade de resolver o problema de uma forma mais esperta **utilizando programação dinâmica**, e tal característica tem o nome de sobreposição de problemas.

Programação dinâmica

Identificada a sobreposição de problemas, podemos adotar uma estratégia que evite o cálculo da solução para o mesmo problema em partes diferentes da execução do algoritmo.

Caracterização dos subproblemas

Casos base: Podemos ver a menor unidade dos subproblemas como sendo alguma das palavras de origem ou destino vazias. Neste caso, a solução é trivial e é dada pelo número de caracteres restantes na outra palavra.

Hipótese de indução: Assumimos então que estamos com problemas com as palavras de origem e destino com tamanhos (N' ; $M'+1$), ($N'+1$; M') e (N' ; M'), sendo N' e M' menores que N e M que são os tamanhos originais das palavras de origem e destino.

Passo indutivo: Queremos resolver um problema maior de tamanho ($N'+1$; $M'+1$). A solução deste problema se dá pelo cálculo das distâncias caso fizermos as 3 operações nas últimas letras. Como todas as 3 operações diminuem nosso problema de formas diferentes, encontraremos subproblemas menores com palavras de tamanhos (N' ; $M'+1$), ($N'+1$; M') e (N' ; M'). Como problemas com palavras deste tamanho por hipótese nós conseguimos resolver, a solução é válida.

Equação de recorrência

A partir da prova acima, podemos extrair a seguinte equação de recorrência abaixo. $OPT(i, j)$ nos dá a solução ótima para o problema de tamanho $(i;j)$.

```
OPT(i, j) = {
        j, se i igual a 0;
        i, se j igual a 0;
        min(OPT(i-1, j), OPT(i, j-1), OPT(i-1, j-1)), se i > 0 e j > 0.
}
```

Para prova

Algoritmo

Estamos variando o tamanho de duas palavras durante o nosso algoritmo. Para todas estas variações precisamos calcular a distância de edição de uma palavra para a outra. Uma

estrutura de armazenamento que faz sentido para recuperarmos tais cálculos é uma matriz $N+1$ por $M+1$, sendo que a matriz nos índices i e j nos dará a solução do subproblema de tamanho (i;j). Já sabemos preencher a linha 0 e a coluna 0 desta matriz, por que são nossos menores subproblemas. O preenchimento das demais se dá pela aplicação das operações. A seguinte equação de recorrência representa o algoritmo descrito:

```
Entrada: Palavra A de tamanho n, Palavra B de tamanho m.
Início:
    Seja D uma matriz de n linhas por m colunas
    Para i de 0 até n
        Para j de 0 até m
            Se i igual a 0
                D[i][j] = j
            Senão, se j igual a 0
                D[i][j] = i
            Senão, se A[i-1] igual a B[j-1]
                D[i][j] = D[i-1][j-1]
            Senão
                D[i][j] = 1 + min(D[i][j-1], D[i-1][], D[i-1][j-1])
    Retorna D[n][m]
```

3. Análise Teórica do Custo Assintótico

3.1. Análise Teórica do Custo Assintótico de Tempo

O programa desenvolvido realiza uma poda para evitar o cálculo da distância mínima de edição para palavras do dicionário em que a diferença do tamanho para a palavra de referência seja maior do que a distância máxima já especificada. Seja N o número de caracteres da palavra de referência e d a distância máxima das sugestões, isso faz com que linhas que tem o tamanho superior à $N+D$ e inferior à $N-D$ sejam descartadas como soluções válidas.

Com a poda feita, o pior caso de execução da função de cálculo da distância mínima de edição se dá pela entrada da palavra de referência de tamanho n e uma possível sugestão de tamanho $N+D$. Como pode ser visto no pseudocódigo referente ao algoritmo em questão na modelagem do problema, temos 2 loops que iteram sobre a matriz com N linhas e $N+D$ colunas. Assim, a complexidade para o cálculo da distância mínima é de $O(N \times (N+D))$.

No pior caso, todas as palavras do dicionário passarão pela poda e irão ser avaliadas como possíveis sugestões. Seja S a quantidade de linhas na entrada, a complexidade assintótica de tempo final para o cálculo das sugestões é de $O(S \times N \times (N+D))$.

3.2. Análise Teórica do Custo Assintótico de Espaço

Caso uma palavra do dicionário não seja uma sugestão válida, o programa não a armazena. No pior caso o programa deve armazenar todas as S palavras do dicionário, e para cada palavra deve ser chamada a função para cálculo da distância mínima. Como pode-se ver pelo pseudocódigo, a função aloca uma matriz de tamanho máximo N (tamanho da palavra de referência) por $N + D$ (palavra a ser avaliada). Assim, a complexidade de espaço final do programa se dá por $O(S \times N \times (N + D))$.

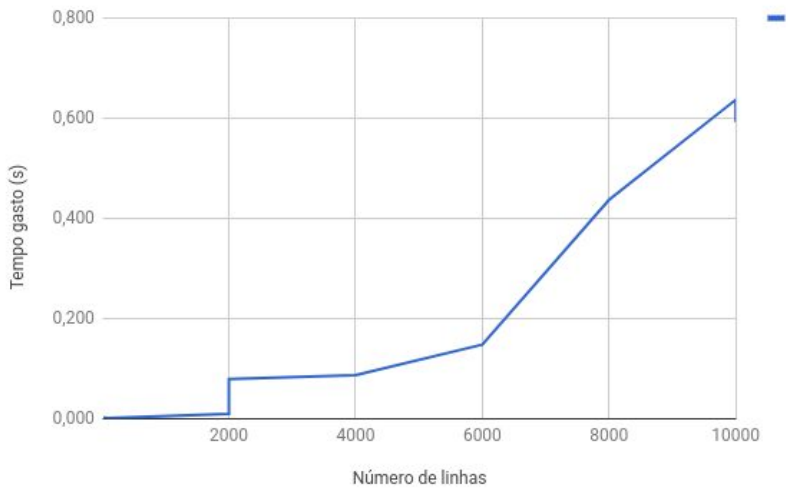
4. Análise do Experimento

O experimento feito utiliza os casos de teste fornecidos pelo monitor. Os dados de memória utilizada e tempo gasto para a execução foram coletados da saída do *script* "*run_tests.sh*". O experimento foi feito usando uma máquina com processador Intel Core™ i7-3630QM CPU 2.40GHz × 8 e memória RAM de 7,5 GiB.

As execuções foram feitas de tal forma que conseguiremos ver variações de S (número de linhas do arquivo de entrada) de 15 linhas até 10000 linhas, N (número de caracteres da palavra de referência) de 6 a 89 caracteres e D (distância máxima) de 2 a 8. Tais variações influenciam o tempo de execução do programa, diminuindo ou aumentando de acordo com seus valores.

Para evitar valores que não representem a fielmente o tempo gasto foram feitas 5 execuções para cada caso de teste, sendo tirada a média dos 5 tempos para se chegar ao tempo gasto em cada ponto. Pelos casos de teste fornecidos, a influência maior na complexidade de tempo do programa se dá pela quantidade de linhas do arquivo de entrada. A representatividade do número de caracteres da palavra de referência e do número de edições possíveis não pôde ser observada. Abaixo se encontram os tempos de execução de acordo com a quantidade de linhas:

Tempos de execução de acordo com quantidade de linhas do dicionário



Pela análise do gráfico podemos concluir que a quantidade de linhas do dicionário têm influência linear sobre o tempo de execução do programa. Esta observação confirma a análise do custo assintótico de tempo feita na seção anterior.

A falta de cenários propícios para o estudo da complexidade nos casos de teste não possibilitou a confirmação da influência dos parâmetros N tamanho da palavra de referência e D número máximo de edições, mas pode-se perceber que nos casos de teste a variação é baixa de ambos..

5. Conclusão

Programação dinâmica foi um paradigma pertinente para este problema. A repetição dos cálculos no algoritmo de força bruta faz com que o tempo de execução cresça exponencialmente, e para casos de teste relativamente pequenos este tempo já se torna impraticável. A poda de entradas evita também que cálculos de distância de edição sejam desperdiçados como já sabemos a quantidade máxima. Com esta solução os usuários do Doodle vão ter as sugestões para seus erros calculadas de forma eficiente, cumprindo assim o objetivo do trabalho.