CTC-34 - Projeto: Autocomplete

- Grupos de 3.

Proposta do projeto:

- Implementar um FST para autocomplete de inglês, usando o dicionário do linux
    - O nome formal do algoritmo é "Minimal Acyclic Subsequential Transducers"
        - A entrada é uma lista ordenada de palavras
        - A saída é um FST quasi-***mínimo*** para indexar as palavras.
        - O algoritmo deve ser executado on-line, gerando o FST rapidamente a partir do dicionário de palavras, sem depender de arquivos gravados off-line.
    - Estou aberto a outras idéias sobre datasets e aplicações
- Implementar o autocomplete on-line
    - não precisa de interface “fancy”, apenas usável p/ verificar funcionalidade.
    - On-line: A cada letra digitada, deve-se carregar as primeiras palavras do dicionário que são prefixos da string de entrada (**DEMO 1**).
    - O algoritmo de criação do índice FST a partir da lista de palavras deve ser implementado por vocês.
    - Vale usar libs de grafos, DFA, ou FST, como base. E.g código p/ nós, arestas, BFS e algoritmos básicos de grafos (na dúvida pergunte), estatísticas, drawing e printouts.
- Comparar a implementação com uma outra alternativa (cada grupo terá uma)
        - Trie
        - Busca binária
        - Árvore balanceada com busca inteligente O(logn + k) (procurar uma boa implementação de CES12, ou usar lib)
        - HashTable (pelo menos p/ buscar palavras exatas. Mas HashTables podem ter listas ligadas que permitam acessar os próximos elementos)
    - Comparar
        - tempo de execução do autocomplete
        - Tempo de criação do índice a partir do dicionário como lista de palavras
        - Memória - esta é a vantagem fundamental da FST em relação à Tries ou outras árvores, e **não** o tempo em si.
            - **Contar: quantidade de nós e gasto total de memória**.
        - **Comparar usando testes apropriados, medindo o tempo e fazendo estatísticas suficientes**.
- Implementar Autômato de Levenshtein distância 1 p/ uma palavra de entrada
    - Algoritmo de construção:
        - Entrada: uma palavra-alvo e um parâmetro d.

        - Saída: DFA que aceita qualquer palavra cuja distância até a palavra-alvo seja ≤ d. Onde distância significa o número mínimo de trocas, inclusões ou deleções de caracteres para transformar a palavra dada na palavra-alvo ou vice-versa.
    - Objetivo neste projeto:
        - A palavra-alvo é a palavra digitada até o momento
        - Construir o DFA-Levenshtein para a palavra alvo e distância 1
        - Executar o DFA-Levenshtein EM PARALELO com o FST, aceitando apenas palavras aceitas por ambos DFA-Levenshtein E FST.
        - Ou seja, dada a palavra digitada pelo usuário, o usuário verá uma lista de palavras que completam a palavra digitada, E estão a distância 1 da palavra digitada.
        - Implementar demo on-line. (**DEMO 2**)
    - Já há implementação pronta em python de Levenshtein. Pode ser adaptada. Outra implementação do algoritmo de construção pode ser encontrada e utilizada. Podem também implementar do zero.
    - A intersecção dos autômatos do dicionário e Levenshtein pode ser simulada executando-se os 2 autômatos EM PARALELO, caractere por caractere, e verificando se a entrada é aceita por ambos.

Teoria e Algoritmo "Minimal Acyclic Subsequential Transducers" (MAST)

FST (Finite State Transducer) Mínimo
FST é um AF que pode imprimir strings a cada aresta.
Existe um algoritmo para construir um FST mínimo dada uma lista de palavras ordenadas lexicalmente. Além do link acima, baixei o pdf:
https://drive.google.com/file/d/1PpBKM9LiXSI2TzFOBe7okyuWpnNvWtPJ/view?usp=sharing

Referência mais geral sobre FST
https://drive.google.com/file/d/1Mg_cwntAHs7exV3bfoL45U-hukKHTamE/view?usp=sharing
A teoria mais completa envolve caminhos probabilísticos (as arestas estão associadas com um peso, além de uma string para a saída). É importante, por exemplo, em correção automática ou tradução fonema/escrita automática, onde caminhos no FST correspondem a possíveis strings de entrada parecidas, permitindo modelar possibilidades para cada transição no caminho. Isso é útil quando há um modelo considerando a probabilidade de trocar um caractere por outro. Exemplo: em reconhecimento de fala, depois que um classificador separa e classifica fonemas, há modelos que estimam a probabilidade de errar a classificação por um fonema parecido é maior em relação a outro fonema mais distante. E.g. trocar /v/ por /b/ é mais comum que trocar /v/ por /k/. Poderia ser aplicado ao nosso problema, construindo um modelo onde trocar uma tecla por outra vizinha no teclado é mais provável do que por outra distante. Pode ser um aprofundamento opcional do projeto, mas não há tempo de cobrirmos essa teoria.

Mas, para aplicação de autocomplete em vista, não há probabilidades e o FST é uma máquina de Mealy com alguns detalhes a mais (e.g. some o output ao longo do caminho para encontrar o índice da palavra no dicionário)

Dados: lista de palavras

Arquivo de dicionário do linux com 102K palavras e 971K caracteres:

```
$ wc /usr/share/dict/american-english
102305 102305 971578 american-english
```

Levenshtein Automata (inclui boa parte do código python)

A leitura é interessante também pelos muitos detalhes de implementação.
https://julesjacobs.com/2015/06/17/disqus-levenshtein-simple-and-fast.html
Cópia em PDF do blog acima:
https://drive.google.com/file/d/17TuRSC20HKUzrxIOvu7ps2s7eGQXUe2J/view?usp=sharing
O codigo dele esta no github
https://github.com/julesjacobs/levenshtein

Links Uteis

https://blog.burntsushi.net/transducers/
Cópia em PDF do blog acima:
https://drive.google.com/file/d/1nfaWYVImAuOuLvYWONmxX8oSHbt-w8vK/view?usp=sharing

- Discute sobre o algoritmo de criação do FST mínimo, com desenhos passo a passo
- Discute sobre aplicação de FST p/ índices, usando uma lib pronta.
- Mostra o tamanho dos índices para diferentes bases de dados, comparando com outras estruturas. Interessante como padrão de comparação.
- Se acharem o dicionário muito pequeno, ele tem sugestões de datasets maiores.
- Mostra código usando a lib pronta, que pode ser inspirador para a implementação de vocês.
- Discute limitações das FST → não resolve todos os problemas!

Veja uma comparação de espaço ocupado para indexar o DOI dataset, contra um zip:
The DOI data set contains 49,118,091 URLs and occupies 2800 MB on disk.

Format	Time	Max memory	Space on disk
plain	-	-	2800 MB
fst	27.40s	17.6 MB	113 MB (4.0%)
gzip	40.39s	1.8 MB	176 MB (6.3%)
xz	716.60s	97.6 MB	66 MB (2.6%)

https://www.openfst.org/twiki/bin/view/FST/WebHome
OpenFST lib

- ALGORITMO DE CRIAÇÃO DO ÍNDICE FST JÁ ESTÁ IMPLEMENTADO, PODEM USAR P/ COMPARAR OU APRENDER, MAS É PRECISO UMA IMPLEMENTAÇÃO INDEPENDENTE PRÓPRIA.
- Podem usar a lib como base, ou outra lib de grafos, DFA, ou FST se preferirem.
- Nunca usei. O básico parece fácil, mas não sei se compila, se é atual, etc.
- Ao invés de comparar com outra estrutura de dados, um grupo poderia comparar com a implementação disponível.

https://github.com/yhirose/cpp-fstlib
Exemplo de implementação USANDO a fstlib. Inclui chamadas p/ o algoritmo de criação do índice, mas não a implementação do mesmo.

https://www.openfst.org/twiki/bin/view/FST/FstHltTutorial
Tutorial da OpenFST (mais voltado a aplicações mais complexas usando probabilidades)

Apresentação da FSTLIB - ideia interessante de implementação da serialização.
https://drive.google.com/file/d/1u2vokEaHtONprKo4BcgQp5HMb9aBALQ7/view?usp=share_link

Dicas

Faça testes com listas de palavras pequenas

Não execute apenas com o dicionário completo. Crie uma FST para uma lista de poucas palavras, e DESENHE o grafo para conferir. Libs de grafos normalmente permitem output em .dot, ou alguma outra forma de desenhar o grafo. Considero um bom argumento para usar uma lib de grafos.

Quero implementar eficientemente e as libs de grafos disponíveis são ruins ou tem uma interface inconveniente. Quero minha própria implementação de grafos.

Não precisam competir em eficiência, mas se preferem otimizar, o que eu faria:

- Implementaria, primeiro, usando uma lib de grafos.
- Anotaria todas as funções e algoritmos utilizados, sem contar debugs e desenhos.
- Depois, implementaria uma lib própria com a mesma interface. Bastaria então trocar o #include (ou equivalente) da lib de prateleira pela sua, e temos a implementação eficiente.

Vantagens:

- lib de prateleira é usada para debug e desenhos. É possível reverter rapidamente da sua para a de prateleira se precisar voltar ao debug fácil.
- Se não gosta da interface da lib de prateleira, ou se ela é muito complexa, implemente uma camada de adaptação com uma interface mais conveniente. Desenvolva uma implementação desta camada que chama a lib de prateleira, e outra que chame a sua implementação de grafos. Alternar entre uma e outra continuaria trivial.
- Note o passo intermediário “Anote todas…”. Não precisarão de muitos algoritmos de grafos diferentes, talvez apenas BFS/DFS além do básico de vértices e arestas. Mas, se você sabe bem o que precisa ser implementado, é mais fácil **decidir** se vale a pena implementar a sua própria lib de grafos.

Levenshtein:

- Não deve ser criados um DFA-Levenshtein para cada palavra do dicionário - seria muito lento. Não se deve criar uma lista de DFA-Levenshteins e se aplicar a palavra digitada a cada DFA da lista.
- O propósito é criar um DFA-Levenshtein para a palavra digitada pelo usuário, e verificar se este DFA criado aceita as próximas palavras do dicionário.
- À medida que o usuário digita, o DFA-Levenshtein criado até então pode ser adaptado para incluir o novo caractere digitado.
- a matriz é um passo intermediário. O objetivo é do algoritmo de criação do DFA-Levenshtein, a partir de uma palavra, criar um automata. A matriz p/ o dicionário inteiro fica muito grande, fica lento.

Problemas e correções

Implementação minimamente eficiente: estrutura de dados para buscar nós equivalentes.

O passo mais demorado em uma implementação ineficiente é buscar na FST se existe um nó equivalente, o maior sufixo já congelado para cada novo nó a ser inserido. Uma busca linear representa complexidade quadrática desnecessária. Veja a função `FindMinimized`.

- Com busca linear, a criação do dicionário terá ordem de grandeza de centenas de segundos.
- Esta demora dificulta o debug e desenvolvimento de vocês mesmos.
- Não serão aceitas implementações assim.

Solução: estrutura de dados (outro dicionário) armazena nós já congelados para procura.

- Calcular se dois nós são equivalentes é mais complexo do que usar o operador == em um campo. Para usar uma estrutura de dados pronta de biblioteca, é preciso normalmente implementar uma função de comparação, que responda se dois nós são equivalentes. Já precisaram fazer o mesmo no lab de TSP em CTC12
- A estrutura de dados não deve armazenar uma segunda cópia do nó, apenas um ponteiro para o nó na FST (caso contrário o gasto de memória será duplicado, e usar menos memória é justamente a vantagem principal da FST).

Dicionário do linux: caracteres especiais

O dicionário do linux EM INGLÊS inclui palavras com acentos como ètude ou Å ngstrom. Todas as letras acentuadas aparecem DEPOIS da letra z. Como a maioria das linguagens de programação não segue esta convenção, isso atrapalha a criação da FST pois a ordem total[1] da comparação entre chars não é a mesma ordem que gerou a ordenação lexicográfica do dicionário.

A ordem precisa ser consistente, ou seja existe uma ordem total entre cada caractere, inclusive variações acentuadas dos caracteres, e esta ordem se reflete na ordem lexicográfica das palavras.

Solução sugerida: reordenar o dicionário

Algumas linguagens trabalham (inclusive por default) com caracteres Unicode, e portanto não tem dificuldade em representar tais caracteres especiais. Portanto, bastaria usar um algoritmo de ordenação para ordenar as strings do dicionário e gerar (off-line) um outro arquivo

[1] Ordem total: relação de ordem onde, p/ qualquer par de elementos (x,y), $x \neq y \Leftrightarrow x < y$ ou $x > y$

representando o dicionário ordenado consistentemente com o operador de comparação da linguagem da sua implementação. Estou aberto a discutir outras soluções.
Tenham consciência de qual foi a ordem gerada para conseguir distinguir respostas corretas e erradas rapidamente. ‘étude’ vem depois de ‘etiquette’ (preferível), ou ‘étude’ vem depois do z?
Se escolherem uma linguagem onde representar caracteres Unicode apresenta maior dificuldade, posso aceitar uma simplificação como eliminar palavras acentuadas **- a combinar**.

PS: para alguns pode parecer um detalhe irrelevante. Mas o número de palavras acentuadas é grande o suficiente para interferir nas respostas para várias palavras comuns. *Sem esta correção, será difícil debugar o algoritmo*.

FAQ

A interface com o usuário p/ o auto-complete precisa ser web?

Pode ser, mas não precisa e talvez não valha a pena perder tempo com isso. Pode ser uma aplicação / executável / script no seu computador. A rigor nem precisa janelas, mas preciso conseguir usar e ver o auto-complete rápido e on-line, mostrando a lista de palavras do auto-complete a cada letra digitada, e janelas e caixas de texto simples podem ser mais fáceis de implementar do que interface textual.

A lib de grafos gera um arquivo que representa o grafo. Posso usar?

Ao usar uma lib de grafos, é comum existir API p/ salvar o grafo em arquivo. Assim:

1. pode-se usar o algoritmo p/ montar a FST, e salvar a FST como um grafo, off-line.
2. Ao inicializar o auto-complete, poderia-se carregar o arquivo, com a FST pronta, o que é pode ser mais rápido que criá-la de novo a partir do arquivo de palavras ordenadas.

Verdade, mas **exijo criação on-line da FST a partir apenas do dicionário**. Em segundos, não centenas de segundos.

No entanto, podem implementar as operações de salvar/ler FST de arquivo se ajudar no debug, desenvolvimento, ou comparação de resultados. Mas desabilitem antes de apresentar.

E lembrem-se:

- Gravar a FST em arquivo é válido apenas porque a FST visa dados estáticos, onde o índice também é estático e não precisa ser modificado com inclusões / exclusões. Se mudar o dicionário com inserções e deleções de palavras, a FST será invalidada.
- O tempo de criação do índice é o tempo de execução do algoritmo MAST, NÃO o tempo de carregar o arquivo que representa o grafo, e NÂO o tempo de execução de um auto-complete depois que a FST é criada.
- Representações de grafo em arquivo podem ser mais ou menos eficientes em espaço (i.e., o tamanho do arquivo). O gasto de memória a ser comparado é o espaço de

memória efetivamente ocupado pelo índice em runtime (durante a execução do auto-complete), e NÃO o tamanho do arquivo.
- A própria biblioteca de grafo pode ser ineficiente em memória se os objetos nó, aresta, etc, forem mais complexos que o necessário, com campos e funcionalidades não usadas.

Professor, no arquivo de dicionário do Linux, as palavras não têm seus respectivos valores inteiros, nesse caso eu posso adicionar na mão mesmo?

Qual é o propósito de um índice? É fazer buscas. E para que fazemos buscas? Para encontrar algo e fazer algo com a coisa encontrada. O que queremos encontrar e fazer? Depende da aplicação. Poderia ser, e.g., buscamos por 'abacate', e além da confirmação que está no dicionário, recebemos um número-índice que permite buscar a descrição de abacate rapidamente em uma grande Database de descrições e imagens - e.g., indicando o disco, arquivo, tabela, setor, byte, etc (whatever is needed) - onde está a descrição de "abacate".

No caso, queremos encontrar as próximas palavras no dicionário. Bastaria associar a cada palavra a posição em bytes onde está a palavra no dicionário. Se o dicionário está em arquivo, pode-se abrir o arquivo já na posição certa. Se o dicionário está na memória, pode-se acessar diretamente o byte correspondente ao início da palavra com um ponteiro - e para encontrar as próximas palavras basta acessar os próximos bytes do dicionário sequencialmente. Afinal, o índice existe para evitar ler o arquivo inteiro ou percorrer todo o dicionário em O(n) para achar a palavra em questão, assim, permitindo encontrar a palavra em O(1). **Se não for assim não faz sentido existir um índice**!
Podem adaptar a ideia. O número que deve ser associado a cada palavra é aquele que facilita o que se quer fazer ao encontrar a palavra.

A Lib tem overhead, e o consumo de memória é menor no FST mas não tão menor. Mas pelo número de arestas temos compressão de mais de 10x em relação ao número de caracteres. Posso comparar a memória utilizando o número de nós e arestas?

Pode, mas são duas métricas diferentes - é preciso entender e mostrar que são coerentes. Usar a lib, é atalho de implementação - mais fácil implementar sobre lib pronta. Deve haver um overhead na representação genérica de grafos, vértices e arestas, com vários campos desnecessários na estrutura de dados.

- Verifique como a lib representa o grafo e verifique, aproximadamente, se os números batem. E.g.: a estrutura de dados usada pela lib gasta tantos bytes para as arestas e tantos para os vértices. Dado o número de arestas e vértices, uma multiplicação deveria aproximar a memória ocupada pela FST.

Use alguma ferramenta de introspecção da linguagem ou do SO para aproximar a memoria ocupada pela FST em si, ou pelo processo.

O autocomplete deve ser independente do dicionário? Digo, o FST me gera o índice da palavra no Dicionário, mas também se a palavra não é completa o FST gera o índice da primeira palavra que tenha aquele prefixo, a ideia seria pegar esse índice e gerar o "complete" a partir dele printando 3/4 palavras que tenham índices subsequentes, mas para isso é necessário utilizar o dicionário também. A minha pergunta é se posso utilizar o dicionário para completar a palavra ou só o FST?

A princípio, se você tem um dicionário, a FST deveria ser um índice correto para achar a posição correta da palavra (ou prefixo), em O(1), então use o dicionário. Tá tudo na mão pra fazer rápido, sem busca sequencial.
Imagine que além das próximas palavras, quiséssemos mostrar mais dados armazenados no dicionário junto das palavras - imagem, classe gramatical, pronúncia, definição, etc. Seríamos obrigados a usar o dicionário original p/ o autocomplete.

Mas, neste projeto aceito uma busca na própria FST para achar as palavras que completariam a digitação - apenas cuidado com convenções diferentes que produzam resultados diferentes. Confiram alguns resultados olhando no dicionário diretamente.

Inclusive porque apareceram idéias sobre outras aplicações e variações que utilizaram buscas na própria FST de forma interessante.

Sobre verificar quanta memória um processo ocupa

https://linuxhint.com/check_memory_usage_process_linux/
https://stackoverflow.com/questions/3853655/in-linux-how-to-tell-how-much-memory-processes-are-using
https://askubuntu.com/questions/176001/what-do-virt-res-and-shr-mean-in-the-top-command

Exemplos comandos simples:

```
$ top =p<PID>
$ pmap <PID>
```

Alguns alunos sugeriram esta lib em python. Talvez permita estimar a memória da FST, não do processo inteiro.
https://docs.python.org/3/library/tracemalloc.html

Posso escolher outro projeto, ou variação deste?

Pode, inclusive, sobre outras partes da disciplina inteira. Mas:

- A dificuldade deve ser comparável. E.g. parsers muito simples não valem.
- Não recomendo implementar um compilador. As fases finais da compilação (geração de código) são as mais difíceis de implementar por quem não tem experiência. Mas pode haver um contexto em que faça sentido.
- É preciso combinar **LOGO**, e fechar o escopo **LOGO**. Não chegue nas últimas semanas com o projeto em aberto.