Projeto parcial
Closest string problem

Lucas Emanuel Resck Domingues

22 de agosto de 2021

Tarefa 1 (1 ponto) Escolha um problema de otimização combinatória, apresente sua definição e apresente argumentos da sua NP-dificuldade (não é necessário fazer a redução ou a demonstração).

Vamos trabalhar o problema da ***closest string*** ("string mais próxima", em tradução livre). Dadas n strings de tamanho m, $s_1, \cdots, s_n$, buscamos uma nova string s de tamanho m tal que $\mathrm{d}(s_i, s) \leq d$, para todo $i \in \{1, \cdots, n\}$, porém com d sendo o menor valor possível. Em geral, chamaremos d de custo, pois é o valor que queremos minimizar. A função $\mathrm{d}(\cdot, \cdot)$ é a distância de Hamming, *i.e.*, o número de posições diferentes entre duas strings. Vamos considerar, neste problema, um alfabeto fixo Σ. De forma um pouco mais abstrata, estamos buscando no espaço métrico de Hamming (Σ^m, d) a menor bola fechada, com centro s e raio d, que contém todas as n strings $s_1, \cdots, s_n$.

Vejamos um exemplo. Suponhamos que queremos encontrar a string mais próxima a ambas as strings $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{T}}\boxed{\mathrm{O}}$, $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{M}}\boxed{\mathrm{E}}$ e $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{M}}\boxed{\mathrm{O}}$. Então é óbvio que a solução será $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{?}\boxed{?}$. Nossas opções para a terceira letra são T e M e para a quarta letra são O e E. Para cada combinação, temos o seguinte valor d mínimo:

- $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{T}}\boxed{\mathrm{O}} \Rightarrow 2$
- $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{T}}\boxed{\mathrm{E}} \Rightarrow 2$
- $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{M}}\boxed{\mathrm{O}} \Rightarrow 1$
- $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{M}}\boxed{\mathrm{E}} \Rightarrow 2$

Ou seja, nossa solução é $\boxed{\mathrm{G}}\boxed{\mathrm{A}}\boxed{\mathrm{M}}\boxed{\mathrm{O}}$.

O problema da *closest string* é NP-difícil. De forma intuitiva, vamos pensar em um problema mais simples, em que o alfabeto é $\Sigma = \{0, 1\}$. Então a string assume a forma $\boxed{0}\boxed{0}\boxed{1}\cdots\boxed{0}\boxed{1}$, por exemplo, que por acaso é a representação do problema da mochila 0-1. No problema da mochila, você é incentivado a colocar um item pois ele tem valor positivo, mas é desincentivado pois tem peso positivo e há restrição de peso. Você nunca sabe "qual vale mais a pena", se é deixar a mochila mais leve ou mais valiosa. Colocar um item, mesmo que leve e valioso, pode atrapalhar muito os outros itens, a ponto de deixar a mochila menos valiosa ao final. Isto é, não se sabe se é possível sacrificar um item valioso para ter um ganho melhor com outros itens, possivelmente menos valiosos. O dilema é parecido nesse caso: você é incentivado a utilizar o caractere 0 pois isso vai se parecer com algumas *strings*, porém é incentivado a utilizar o caractere 1 ao mesmo tempo, pois se parece com outras, e você não sabe qual vale mais a pena. Não se sabe se é possível sacrificar um "bom" caractere (aquele que diminuiria um erro local, por exemplo) em uma posição para se ter um ganho melhor com outros caracteres em outras posições, e vice-versa.

Claro, essa intuição não prova nada. Uma demonstração da NP-dificuldade do problema é dada por Frances e Litman [1]. A demonstração não é trivial e, acredito, não cabe neste trabalho.

Tarefa 2 (2,5 pontos) Proponha um algoritmo exato para a resolução deste problema. Calcule sua complexidade.

O problema é NP-difícil, então não precisamos ter medo de algoritmos com complexidade exponencial. Vamos propor aqui um algoritmo de força bruta, que testa todas as soluções possíveis, porém apenas dentre aquelas que fazem sentido serem testadas.

Sejam $s_1, \cdots, s_n$ as n strings de tamanho m. Então, para cada posição i, a string s pode assumir algum dos valores $S_i = \{s_1[i], \cdots, s_n[i]\}$. Nosso algoritmo será aquele que testa cada uma das combinações possíveis com um valor de S_1 para a posição 1 de s, um valor de S_2 para a posição 2 de s, $\cdots$, um valor de S_m para a posição m de s, e no final escolhe a melhor das soluções. Na verdade, é mais prático manter uma variável indicando a melhor das soluções durante a iteração e atualizando-a quando necessário.

Para montar os conjuntos S_i, existe um custo $O(mn)$ (são n strings de tamanho m). Para uma determinada posição i, no pior caso, $S_i = \{s_1[i], \cdots, s_n[i]\}$ é igual a todo o alfabeto Σ. Logo, como são m posições, o número de combinações possíveis para s é $O(|\Sigma|^m)$.

Para cada combinação de s, são necessários n cálculos de distância de Hamming com as outras strings s_i, que, cada um, toma tempo $O(m)$. Depois de calcular todos os $\mathrm{d}(s, s_i)$, precisamos encontrar o maior desses valores, o que pode ser feito em tempo linear $O(n)$. Ou seja, esse algoritmo tem complexidade $O(mn + (mn + n)|\Sigma|^m) = O(mn|\Sigma|^m)$.

Algorithm 1 Algoritmo exato para o problema da *closest string*.

Require: Strings $s_1, \cdots, s_n$ de tamanho m
  Cria os conjuntos $S_i = \{s_1[i], \cdots, s_n[i]\}$, para cada i
  Inicializa uma solução qualquer s_0
  Calcula o valor d_0 da solução s_0
  for cada combinação possível de s, com $s[i] \in S_i$ **do**
    for s_i **do**
      $d_i \leftarrow \mathrm{d}(s, s_i)$
    end for
    $d \leftarrow \max d_i$
    if $d < d_0$ **then**
      $s_0 \leftarrow s$
      $d_0 \leftarrow d$
    end if
  end for
  return s_0, d_0

Tarefa 3 (3 pontos) Proponha e implemente um algoritmo baseado em árvores ou por aproximação. Apresente argumentos para sua corretude e/ou aproximação e calcule sua complexidade.

No que diz respeito a algoritmos de aproximação, Li, Ma e Wang propuseram, e demonstraram, uma aproximação em tempo polinomial com razão de aproximação $1+\epsilon$ para o problema da *closest string* [2], como descrito na Tarefa 1. A implementação do algoritmo não é trivial e acredito que esteja um pouco além do que é requerido por este trabalho. Além disso, não existe uma implementação desse algoritmo acessível, então decidi por não me estender muito nesse resultado.

Vamor propor e implementar, nesta tarefa, um algoritmo no estilo *branch and bound*, que nos permita explorar todas as soluções, porém ignorar aquelas em que temos certeza que não são boas o suficiente. Imagine que o Algoritmo 1 funcione a partir de ramificações. Ou seja, o primeiro nó da árvore é vazio e possui como filhos todas as possibilidades para a primeira letra; escolhido um filho, seus filhos possuem todas as possibilidades para a segunda letra; e assim por diante, como na Figura 1. Esse é o *branch*.

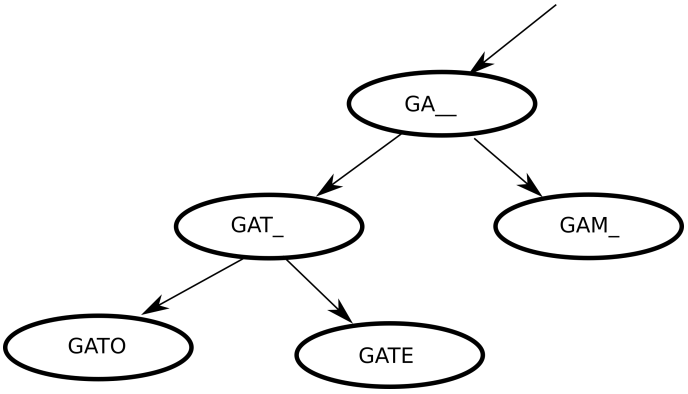

Figura 1: Exemplo de árvore.

Para saber se é válido explorar uma subárvore, isto é, quando realizar o *bound*, vamos utilizar um *lower bound* para o menor custo d em uma subárvore. Note que, em um nó com as i primeiras letras definidas, temos o custo dessas i primeiras letras já calculado; precisamos nos preocupar apenas com as $m-i$ letras restantes, isto é, $s[i+1:m]$. Note que, se quisermos minimizar a soma das distâncias de Hamming de s às strings $s_1, \cdots, s_n$, sob a condição das i primeiras letras fixas, basta que tomemos, em cada posição de $i+1$ a m, aquela letra mais comum para aquela posição. Seja essa a string s^*, isto é, s^* toma as i primeiras letras definidas pelo nó e o restante é definido pela letra mais comum àquela posição. Ora, se d_{OPT} é custo mínimo ótimo para as strings $s_1, \cdots, s_n$, com centro s^{**}, então vale que $\mathrm{d}(s^{**}, s_i) \leq d_{OPT}$ para todo j, de forma que

$$\sum_j \mathrm{d}(s^{**}, s_j) \leq n d_{OPT},$$

e, finalmente,

$$\frac{\sum_j \mathrm{d}(s^*, s_j)}{n} \leq \frac{\sum_j \mathrm{d}(s^{**}, s_j)}{n} \leq d_{OPT},$$

sendo a primeira desigualdade justificada pelo fato de que s^* minimiza a soma das distâncias de Hamming. Ou seja, a média das distâncias de Hamming a s^* é um lower bound para o custo ótimo de uma subárvore. Sendo assim, não precisamos explorá-la caso já tenhamos uma solução melhor. O Algoritmo 2 descreve um pseudocódigo para essa solução.

Algorithm 2 Algoritmo *branch and bound* para o problema da *closest string*.

Require: Strings $s_1, \cdots, s_n$ de tamanho m
  Cria os conjuntos $S_i = \{s_1[i], \cdots, s_n[i]\}$, para cada i
  Inicializa uma solução qualquer s_0
  Calcula o valor d_0 da solução s_0
  $s \leftarrow$ empty string
  BRANCH(s)
  return s_0, d_0

  procedure BRANCH(s)
    $i = \text{length}(s)$
    if $i = m$ **then**
      $d \leftarrow$ custo de s
      if $d < d_0$ **then**
        $s_0 \leftarrow s$
        $d_0 \leftarrow d$
      end if
      return
    end if
    $l \leftarrow$ *lower bound* para o custo mínimo de s ▷ Aqui usamos o que foi descrito na solução.
    if $l \geq d_0$ **then**
      return
    end if
    for char $\in S_i$ **do**
      BRANCH($s +$ char) ▷ Estamos adicionando char à string s.
    end for
  end procedure

A implementação do Algoritmo 2 for realizada em Python, podendo ser conferida neste link. Por exemplo, se pedirmos ao programa qual a *closest string* para

$$\underbrace{\boxed{G}\boxed{A}\boxed{T}\boxed{O}\cdots\boxed{G}\boxed{A}\boxed{T}\boxed{O}}_{8 \text{ vezes}}$$

$$\underbrace{\boxed{G}\boxed{A}\boxed{M}\boxed{E}\cdots\boxed{G}\boxed{A}\boxed{M}\boxed{E}}_{8 \text{ vezes}}$$

$$\underbrace{\boxed{G}\boxed{A}\boxed{M}\boxed{O}\cdots\boxed{G}\boxed{A}\boxed{M}\boxed{O}}_{8 \text{ vezes}}$$

obtemos

$$\underbrace{\boxed{G}\boxed{A}\boxed{T}\boxed{O}\cdots\boxed{G}\boxed{A}\boxed{T}\boxed{O}}_{4 \text{ vezes}}\underbrace{\boxed{G}\boxed{A}\boxed{M}\boxed{E}\cdots\boxed{G}\boxed{A}\boxed{M}\boxed{E}}_{4 \text{ vezes}}$$

em 42343 iterações[1]. Um ponto importante é que, sem o *bound*, esse número se torna 174761, significantemente maior.

A corretude do algoritmo é direta, pois é herdada do algoritmo da Tarefa 2 (que é exato) juntamente com a corretude do *bound*, como discutido anteriormente. Porém, esse processo de *bound* não garante que um número significativo de casos será eliminado: ainda, no pior caso, poderíamos passar por todos os nós. Note que nosso Algoritmo 2 é essencialmente o Algoritmo 1, porém no formato de árvore (com recursões), além do fato de que são calculados os lower bounds. O cálculo do lower bound tem complexidade $O(mn+m)$[2]. Como a quantidade de nós é $O(|\Sigma|^m)$[3], isso faz com que a complexidade do algoritmo continue a mesma da Tarefa 2: $O(mn|\Sigma|^m)$.

[1] *Sets* em Python não são determinísticos, então é possível que resultados diferentes apareçam, inclusive com um número de iterações diferente.

[2] Para perceber isso, veja que a primeira parcela se refere ao custo atual até a posição i, enquanto a segunda parcela se refere ao lower bound a partir da posição $i+1$. Esse custo é linear, pois é possível ter os "custos de cada posição" previamente calculados. Para mais detalhes, por favor, conferir a implementação.

[3] Não queria me estender aqui, porém a quantidade de nós pode ser dada por, no pior caso, $0+|\Sigma|+\cdots+|\Sigma|^m = \frac{|\Sigma|^{m+1}-|\Sigma|}{|\Sigma|-1} \leq \frac{|\Sigma|^{m+1}}{|\Sigma|-1} = O(|\Sigma|^m)$, pois Σ é fixo.

Tarefa 4 (2 pontos) Proponha e implemente uma busca local para este problema. Calcule a complexidade, ela deve ser polinomial.

Suponhamos que temos a solução

$$s = \boxed{s_1}\boxed{s_2}\cdots\boxed{s_{m-1}}\boxed{s_m}.$$

Uma busca local bem simples, porém intuitiva e óbvia, é apagar aleatoriamente $r \leq m$ letras e preencher a solução da melhor forma possível. São, ao todo, $|\Sigma|^r$ possibilidades, no pior caso, sendo que para checar cada uma toma-se tempo $O(mn + n) = O(mn)$, como descrito na Tarefa 2: n cálculos da distância de Hamming, que tomam $O(m)$ cada uma, e o cálculo de um máximo ao final, que é feito em tempo linear $O(n)$. Logo, a complexidade dessa busca local é $O(mn|\Sigma|^r) = O(mn)$, dado que Σ é fixo por hipótese e r é um parâmetro fixo. A implementação dessa busca local será feita na Tarefa 5, Algoritmo 3, pois será utilizada na meta-heurística.

Tarefa 5 (3,5 pontos) Proponha e implemente uma meta-heurística para esse problema. Calcule a complexidade, ela deve ser polinomial.

Dada a busca local da Tarefa 4, é bastante conveniente aplicar aqui a meta-heurística GRASP. Não existe muito segredo: vamos construir uma heurística construtiva (aleatoriezada), aplicar a nossa busca local algumas vezes e repetir todo esse processo algumas boas vezes.

Iniciamos pela construção da heurística construtiva. Uma forma muito intuitiva é inicializar uma solução que possui, em cada posição, aquela letra que mais se repete na mesma posição das outras strings. Então, no nosso clássico exemplo, teríamos as strings |G|A|T|O|, |G|A|M|E| e |G|A|M|O|, que resultariam em uma solução inicial |G|A|M|O|. Coincidentemente (ou não), essa é a solução ótima para esse exemplo. Para introduzirmos a aleatoriedade, podemos restringir uma lista por tamanho, sorteando entre os caracteres que mais se repetem, controlando esse sorteio por meio de um parâmetro α. O Algoritmo 3 descreve esse raciocínio com maiores detalhes.

Algorithm 3 Meta-heurística GRASP para o problema da *closest string*.

Require: Strings $s_1, \cdots, s_n$ de tamanho m
Require: Número de iterações
Require: Parâmetros α, r
  Cria os conjuntos $S_i = \{s_1[i], \cdots, s_n[i]\}$, para cada i
  Ordena os elementos de S_i pelo número de repetições
  Remove elementos repetidos de S_i
  Inicializa uma solução qualquer s_0
  for número de iterações **do**
    $s \leftarrow$ CONSTRUCAOALEATORIA()
    $s \leftarrow$ BUSCALOCAL(s)
    $s_0 \leftarrow$ ATUALIZASOLUCAO(s)
  end for
  $d_0 \leftarrow$ custo de s_0
  return s_0, d_0

  procedure CONSTRUCAOALEATORIA
    $s \leftarrow$ empty string
    for cada posição $1 \leq i \leq m$ **do**
      $s_i \leftarrow$ sorteia dentre os $\alpha|S_i|$ candidatos de S_i que mais se repetem
      $s \leftarrow s + s_i$
    end for
    return s
  end procedure

  procedure BUSCALOCAL(s)
    Apaga aleatoriamente r caracteres de s
    $s \leftarrow$ busca pela solução que melhor preenche s ▷ Potencialmente uma recursão.
    return s
  end procedure

A implementação do Algoritmo 3 for realizada em Python, podendo ser conferida neste link.

Construir os conjuntos S_i de candidatos tem complexidade $O(nm \ln m)$, por causa do ordenamento. A construção aleatória toma tempo linear em m, pois cada sorteio é constante. A busca local testa todas as possibilidades para os r elementos faltantes. No pior caso, são $|\Sigma|^r$ possibilidades, onde cada uma toma tempo $O(mn)$ para o cálculo do custo d. Atualizar a solução envolve calcular o custo em tempo $O(mn)$. Finalmente, se N = número de iterações, então temos como complexidade para nosso algoritmo

$$O(nm \ln m + N(m + mn|\Sigma|^r + mn)) = O(nm \ln m + Nmn|\Sigma|^r).$$

Assumimos, a princípio, que Σ é fixo (Tarefa 1); N, r são parâmetros fixos também. Logo, a complexidade final do algoritmo é $O(nm \ln m)$.

Vejamos um exemplo de aplicação da nossa meta-heurística. Suponhamos que possuímos as seguintes sequências de DNA:

- |A|T|C|T|A|T|A|G|A|A|G|T|

- A T C T A C A G T A A C
- A T C T A C A G A A G T
- A T C T A T A G A A G T

As posições em azul correspondem àquelas que contém letras diferentes entre as strings. Aplicando nossa meta-heurística GRASP, conseguimos o resultado A T C T A C A G A A G C, com custo de $d = 2$.

Referências

[1] Moti Frances e Ami Litman. "On covering problems of codes". Em: *Theory of Computing Systems* 30.2 (1997), pp. 113–119.

[2] Ming Li, Bin Ma e Lusheng Wang. "On the closest string and substring problems". Em: *Journal of the ACM (JACM)* 49.2 (2002), pp. 157–171.