Soluções e Comentários

# 4ª Maratona de Programação

cc.uffs.edu.br

4 de julho de 2015


Organização:

Prof. Leandro Zatesko (UFFS)

C.A. de Ciência da Computação (CACC)

Prof. Neilor Tonin (URI)

Equipe URI Online Judge


Esta foi a primeira Maratona de Programação da UFFS realizada no Portal URI Online Judge, e fiquei muito contente com o sucesso que foi a competição. Expresso mais uma vez meus mais sinceros agradecimentos à equipe do Prof. Neilor Tonin pela parceria. Foram 251 times inscritos no Portal, dos quais 138 de fato participaram (fizeram ao menos uma submissão), e 32 inscritos para participarem da competição presencialmente na UFFS, dos quais 20 compareceram.

A prova não foi livre de ocorrências. Acho justo fazer menção às três mais sérias:

1. Havia uma entrada do Problema B em que a primeira linha do arquivo não constava. Felizmente, a falha foi percebida ainda no início da competição, as submissões foram rejulgadas, e nada de mais grave aconteceu por conta disso.

2. Os casos de teste do Problema H enviados para a equipe do URI Online Judge estavam numa versão desatualizada. Infelizmente, essa falha só foi percebida quando a competição entrou no modo *blind*. O que fizemos foi retirar a competição do modo *blind* temporariamente, rejulgar todas as submissões para o Problema H e pôr a competição no modo *blind* novamente. Felizmente, esses rejulgamentos não alteraram em nada o placar. Todas as submissões que haviam sido rejeitadas continuaram sendo rejeitadas.

3. A última ocorrência a que faço menção foi sim séria, e por ela peço as mais sinceras desculpas aos competidores que foram afetados. Numa versão inicial do Problema E, o limite superior para os primos $p$ era de $10^6$, não de $10^7$. Quando fiz os primeiros testes, percebi que poderia tranquilamente subir esse limite para $10^7$, e assim incluí casos de teste com primos maiores que $10^6$. Porém, esqueci de alterar a descrição do problema. Alguns times fizeram submissões que passariam se o limite de fato fosse $10^6$ mas que acabaram não passando por a entrada conter números maiores. Infelizmente, não havia nem como rejulgar essas submissões. A falha foi percebida mais ou menos na metade da prova, a descrição foi corrigida, e os competidores avisados. Os times conseguiram então passar seus códigos, mas infelizmente não houve como anular a penalidade das submissões anteriores.

Ajustar o teor de uma prova nunca é uma tarefa fácil. O ideal é acertar a dificuldade dos problemas e a quantidade de problemas fáceis, médios e difíceis de modo não haja muito empate em número de balões nas primeiras colocações e de modo a que a distribuição dos times entre os diferentes números de balões seja a mais suave possível. Confesso que esta é a primeira competição que eu elaboro em que fico satisfeito com o resultado. Dos 134 times que de fato competiram, 1 fez 8 balões, 3 fizeram 7 balões, 7 fizeram 6, 3 fizeram 5, 9 fizeram 4, 14 fizeram 3, 12 fizeram 2, e 25 times fizeram 1 balão.

Sobre o motivo de ter dado aos problemas os títulos dos livros da série *Harry Potter*, a ideia me ocorreu no ano passado, quando o Prof. Pablo Heiber deu aos 3 problemas do Aquecimento da Regional Latinoamericana do ICPC (nossa Final Nacional da Maratona de Programação no Brasil) os títulos *The Fellowship of the Ring*, *The Two Towers* e *The Return of the King*, apesar de as histórias dos problemas não conterem nada sobre o universo tolkieniano. Procurei, desse modo, por uma série que tivesse títulos suficientes para compor uma competição. *Harry Potter* tinha 7 títulos, então, adicionei um Prefácio e um *Efílogo*, ficando com 9 problemas. Tentei não colocar nada sobre *Harry Potter* nas descrições dos problemas, mas não resisti à tentação e acabei pondo algumas referências nos problemas D e E.

Para finalizar esta introdução e começar de fato a discutir os problemas, considero que a ordem da prova era:

A I D B C F E G H

E na prática esta ordem se confirmou. A propósito, todas as minhas soluções para os problemas, bem como as entradas e saídas usadas na competição, estão no Dropbox do Clube de Programação.

# A Prefácio

Fiquei um pouco decepcionado ao ver como alguns times, mesmo times experientes, tiveram dificuldades com este problema tão simples. Honestamente não consigo encontrar explicação. Considero este problema facílimo. Acredito que qualquer lógica que fosse implementada e que explorasse estes 5 casos passaria:

1. $a > 0$ e $b > 0$;

2. $a < 0$ e $b > 0$;

3. $a > 0$ e $b < 0$;
4. $a < 0$ e $b < 0$;
5. $a = 0$.

Observe que, destes 5 casos, 3 já eram explorados nos exemplos, restando aos times que pensassem um pouco apenas no comportamento do Teorema para os outros 2 casos.

Uma maneira muito simples e imediata de cobrir todos os 5 casos de uma só vez é primeiro calcular o resto $r$ e depois substitui-lo na equação dada na descrição do problema para isolar e calcular o quociente $q = (a - r)/b$. Obtendo o resto $r$ com a operação de resto de divisão disponível na linguagem de programação escolhida, podemos corrigi-lo (fazer com que fique no intervalo $[0..|b|]$) apenas somando-o com $|b|$ e aplicando novamente a operação de resto de divisão. Em C/C++:

```
#include<stdio.h>
#include<stdlib.h> // para usar a função abs

int main(void) {
  int a, b, r;
  scanf("%d %d", &a, &b);
  r = (a % abs(b) + abs(b)) % abs(b);
  printf("%d %d\n", (a - r) / b, r);
  return 0;
}
```

# B A Pedra Filosofal

No Problema de Seleção de Atividades (bem abordado no livro do Cormen, por exemplo), o qual classicamente atacamos com um algoritmo guloso simples, o objetivo é maximizar o número de requisições. Na variante cobrada aqui, o que deve ser maximizado é o tempo de uso do recurso disputado, não necessariamente o número de requisições. Nesta variante, uma mera estratégia gulosa não funciona. Mas basta que entendamos a estrutura do problema que a solução sai de modo quase imediato.

Primeiramente, ordenemos as requisições, utilizando como critério de ordenação primeiro o valor $i$ e depois o valor $j$. Vou chamar as requisições, já ordenadas, de $R_0, \ldots, R_{N-1}$, bem como os valores $i$ e $j$ de cada requisição $R_k$ de $R_k^i$ e $R_k^j$. Seja $S_{t,k}$ ($1 \leq t \leq 3600$, $0 \leq k \leq N-1$) o estado que representa o máximo de tempo de uso da pedra filosofal que é possível atingir considerando apenas as requisições $R_k, \ldots, R_{N-1}$ e apenas o intervalo de tempo $[t..3600]$. É óbvio que esse estado depende de no máximo 2 outros estados, porque temos no máximo 2 escolhas: escolher atender a requisição $R_k$ ou escolher desprezá-la. Se escolhemos atendê-la (e se $k < N-1$), vamos para o estado $S_{R_k^j,k+1}$. Se escolhemos não a atender (com $k < N-1$) o estado para o qual vamos é $S_{t,k+1}$. Assim, temos a recorrência que define as relações entre os estados que modelam a estrutura ótima do problema:

$$S_{t,k} = \begin{cases} R_k^j - R_k^i, & \text{se } k = N-1 \text{ e } R_k^i \geq t; \\ 0, & \text{se } k = N-1 \text{ e } R_k^i < t; \\ S_{t,k+1}, & \text{se } k < N-1 \text{ e } R_k^i < t; \\ \max\left\{S_{t,k+1}, R_k^j - R_k^i + S_{R_k^j,k+1}\right\}, & \text{se } k < N-1 \text{ e } R_k^i \geq t. \end{cases}$$

O estado cujo valor estamos interessados em computar é, claro, o estado $S_{1,0}$. Até que poderíamos simplesmente traduzir a recorrência acima para um código recursivo na nossa linguagem de programação favorita. Mas isso com certeza causaria uma recomputação exponencial de estados, e nossa solução tomaria *Time Limit Exceeded*. Para evitar recomputação, memorizamos os estados cujos valores já conhecemos. Isso é o que chamamos com o nome estiloso de *Programação Dinâmica* (PD). Se você ainda não sabe o que é PD, encorajo-o fortemente a estudar os capítulos relacionados nos livros do Cormen, do Skiena e dos irmãos Halim. Este problema é dos mais simples de PD. Se você já conhecia PD e teve alguma dificuldade com este problema, isso só mostra que você ainda não domina muito o assunto e precisa estudar mais.

No código no Dropbox, eu implemento uma PD *top-down*, mas uma PD *bottom-up* também seria aceita, com certeza.

# C A Câmara Secreta

Se uma região do *grid* possui $n$ quadrados sendo $k$ deles parede, é óbvio que o número de possibilidades requerido é $\binom{n}{k} - 1$. Vamos ver, então, como calcular tanto o número de paredes $k$ numa região quanto o número binomial $\binom{n}{k}$.

A descrição do problema não diz quantas consultas serão feitas, então, só podemos assumir que serão muitas. Percorrer todos os quadrados de uma região para contar quantos são parede toma tempo $O(NM)$, o que representa um número da ordem de $10^3$ operações sendo realizadas *por consulta*. Na verdade, não há caso de teste com mais de $10^3$ consultas, então, acredito que mesmo essa abordagem ineficiente passaria no *time limit* ($10^3 \times 10^3 = 10^6$). Mas com uma abordagem um pouco mais inteligente, e tão simples de *codar* quanto, poderíamos descobrir quantas paredes há em determinada região em tempo $O(1)$. Para isso, precomputamos no momento da leitura do *grid*, para cada posição $(i, j)$, o *acumulado* de $(i, j)$, ou seja, o número de paredes que há na região definida por $(1, 1)$ e $(i, j)$. Isso pode ser feito em tempo constante para cada $(i, j)$ com uma lógica de PD. Vou chamar o acumulado de $(i, j)$ de $ac(i, j)$. Para facilitar, vou definir $ac(i, j)$ mesmo para o caso de $i = 0$ ou $j = 0$, ainda que as posições válidas do *grid* comecem a contar em 1. Temos, então, que:

$$ac(i,j) = \begin{cases} 0, & \text{se } i = 0 \text{ ou } j = 0; \\ ac(i-1,j) + ac(i,j-1) - ac(i-1,j-1) + \begin{cases} 1 & \text{se } grid(i,j) = \# \\ 0 & \text{se } grid(i,j) \neq \# \end{cases} & \text{se } 1 \leq i \leq N \text{ e } 1 \leq j \leq M. \end{cases}$$

Uma vez calculados os acumulados, podemos obter o número $k$ de paredes numa região definida por $(x_A, y_A)$ e $(x_B, y_B)$ em tempo $O(1)$ com:

$$k = ac(x_B, y_B) - ac(x_B, y_A - 1) - ac(x_A - 1, y_B) + ac(x_A - 1, y_A - 1).$$

Se você teve dificuldade para entender as equações acima, desenhe ;)

Agora, sabendo o $n$ e o $k$ da região, precisamos calcular $\binom{n}{k}$ ($0 \leq n \leq NM = 2500$, $0 \leq k \leq n$) módulo $10^9 + 7$. Há duas abordagens para essa tarefa.

Na primeira, precalculamos todos os coeficientes binomiais através do Triângulo de Pascal:

$$\binom{n}{k} = \begin{cases} 1, & \text{se } k = 0 \text{ ou } k = n; \\ \binom{n-1}{k} + \binom{n-1}{k-1}, & \text{c.c.} \end{cases}$$

Note que essas adições precisariam ser feitas módulo $10^9 + 7$ e que, para guardarmos todos esses coeficientes, precisaríamos de uma matriz de dimensões $2500 \times 2500$, ou seja, uma matriz com tamanho da ordem de 1 M, o que não é um problema. Em outros problemas, em que a matriz seria muito grande, precisamos usar a segunda abordagem, até mesmo por uma questão de tempo — se o número de coeficientes binomiais que temos para calcular é tão grande que não cabe em memória, provavelmente não teríamos tempo para calculá-los todos também.

Na segunda abordagem, ao invés de precalcularmos os coeficientes binomiais, precalculamos todos os fatoriais desde 0! até 2500! (tudo módulo $10^9 + 7$), pois:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Mas note que a divisão aqui não é a divisão usual, pois estamos trabalhando com resíduos módulo $10^9 + 7$. O que precisamos fazer é multiplicar $n!$ pelo inverso multiplicativo de $k!(n-k)!$, o qual podemos encontrar através do Algoritmo de Euclides Estendido ou do Pequeno Teorema de Fermat, já que $10^9 + 7$ é primo. Se você não faz ideia do que eu estou falando, vale a pena ir atrás desses conteúdos. O código disponibilizado no Dropbox usa a segunda abordagem com o Pequeno Teorema de Fermat.

# D O Prisioneiro de Azkaban

Este é um problema *ad hoc* de simulação. Tudo o que se tem de fazer é simular as regras do jogo para contabilizar o número de rodadas feitas por cada jogador. O código que pus no Dropbox fala por si só. Para facilitar um pouco a implementação, usei o `map` do C++ para mapear as cartas a valores inteiros ($4 \mapsto 0$, $5 \mapsto 1$, ..., $3 \mapsto 9$), mas esse mapeamento poderia também ser feito de outras formas.

As manilhas recebem na minha implementação o valor 10, e o desempate também se dá através de um mapeamento dos 4 naipes aos inteiros do intervalo [0..3]. Considero que qualquer um com um curso básico de Algoritmos e Programação deveria ser capaz de resolver este problema, embora a implementação seja um pouco trabalhosa.

# E O Cálice de Fogo

Quantos divisores tem o número 630? Ora, $630 = 2^1 \cdot 3^2 \cdot 5 \cdot 7$, e todo divisor de 630 precisa ser da forma $2^a \cdot 3^b \cdot 5^c \cdot 7^d$, com $0 \le a \le 1$, $0 \le b \le 2$, $0 \le c \le 1$ e $0 \le d \le 1$. Como há 2 possibilidades para a escolha da multiplicidade do fator primo 2 ($a = 0$ ou $a = 1$), 3 possibilidades para o fator primo 3 ($b = 0$, $b = 1$ ou $b = 2$), 2 possibilidades para o fator primo 5 ($c = 0$ ou $c = 1$) e 2 para o fator primo 7 ($d = 0$ ou $d = 1$), temos que o número de divisores é $2 \times 3 \times 2 \times 2 = 24$. Generalizando, sendo $p_1^{m_1} \cdots p_k^{m_k}$ a decomposição em fatores primos de $N$, o número de divisores de $N$ pode ser dado por:

$$(m_1 + 1)(m_2 + 1) \cdots (m_k + 1)$$

Sabendo disso, podemos discutir a solução para este problema.

A estrutura básica da nossa solução consiste em:

1. Gerar todos os primos no intervalo $[2..10^7]$ através do *Crivo de Eratóstenes*.

2. Fatorar $N$ para descobrir a multiplicidade $\text{fat}(p)$ de cada primo $p$ gerado na decomposição do inteiro $N$. Note que, se $p$ não aparece na decomposição de $N$, então, $\text{fat}(p) = 0$. Um algoritmo simples de fatoração por divisão e conquista, como explicado no livro dos irmãos Halim, é suficiente. Observe que $\sqrt{N} \le 10^6$, o que significa que os primos que geramos são suficientes para a decomposição de $N$, pois, se esgotarmos todos os primos gerados pelo Crivo e ainda $N > 1$, então, o que tiver sobrado de $N$ é um primo novo, o qual devemos inserir no nosso vetor de primos.

3. Para cada *consulta* definida por um primo $p$ dito por Dolores, precisamos incrementar $\text{fat}(p)$ e calcular o produto $\prod_{q<p}(\text{fat}(q) + 1)$, módulo $10^9 + 7$.

A implementação dos passos 1 e 2 da estrutura acima apresentada é imediata. Vamos discutir o passo 3, contudo. Calcular o produto acumulado de todos os $(\text{fat}(q) + 1)$ para todos os primos $q < p$ a cada consulta custa tempo $O(\pi(p))$ *por consulta*, sendo $\pi(p)$ o número de números primos menores que $p$. Do Teorema dos Números primos, $\pi(n) \sim n/(\ln n)$, o que nos traz um número de multiplicações da ordem de $10^6$ *por consulta*. Não sabemos o número de consultas, mas só podemos presumir que são muitas. Na verdade, há casos de teste com cerca de $10^5$ consultas. Como $10^5 \times 10^6 = 10^{11}$, é óbvio que esta estratégia ingênua receberia *TIME LIMIT EXCEEDED*.

Precisamos, então, de uma estrutura de dados que nos permita obter de modo eficiente o produto acumulado sem precisar fazer um número linear em $\pi(p)$ de multiplicações. Essa estrutura ainda precisa ser dinâmica e permitir a atualização eficiente da multiplicidade de $p$ cada vez que ela for incrementada. Você pode escolher entre uma árvore de segmentos multiplicativa ou uma *BIT* (*Binary Indexed Tree*, ou árvore de Fenwick) multiplicativa para o serviço. As implementações que eu apresento usam *BIT* (se você não conhece *BIT*, vale a pena estudar o assunto, já que se trata de uma estrutura de dados muito simples e muito útil). Como se trata de uma *BIT* multiplicativa, todas as posições da *BIT* são inicializadas com 1. As duas operações da *BIT* são:

- `long long getbit(int i);`
  Calcula em tempo $O(\log \pi(p_i)) = O(\log \pi(\sqrt{N}))$ o produto acumulado $\prod_{q<p_i}(\text{fat}(q) + 1)$.

- `void setbit(int i, long long x);`
  Multiplica o fator $(\text{fat}(p_i)+1)$ por $x$ (o que reflete não só na posição $i$ da *BIT*, mas em $O(\log \pi(\sqrt{N}))$ posições).

Assim, toda vez que estamos para incrementar $\text{fat}(p_i)$, antes de fazermos o incremento precisamos chamar a função `setbit` com $x = (\text{fat}(p_i)+2)/(\text{fat}(p_i)+1)$ — isso terá o efeito de retirarmos o antigo fator $(\text{fat}(p_i)+1)$ da *BIT* e colocarmos o novo fator $(\text{fat}(p_i)+2)$. Note que, como as operações são todas módulo $10^9+7$, a divisão aqui não é a divisão usual, mas a multiplicação por inverso multiplicativo, o qual pode ser encontrado com o Algoritmo de Euclides Estendido ou através do Pequeno Teorema de Fermat.

# F A Ordem da Fênix

O problema em questão é simplesmente o clássico *Longest Repeated Substring* (LRS), com a pequena modificação de que a resposta só deve ser impressa se possuir comprimento no mínimo 3. Como o tamanho $n$ da *string* pode ser $10^5$, é muito improvável que soluções ingênuas (quadráticas ou piores que isso) passem. Precisamos de algum algoritmo mais eficiente — e que seja rápido de *codar* em tempo de competição. Utilizando um *array* de sufixos, é possível fazer o serviço em tempo $O(n \log n)$. Embora a construção do *array* de sufixos seja complicada de entender mas curta de *codar*, uma vez construído o *array* a solução para o LRS sai quase imediatamente. O livro dos irmãos Halim explica muito bem o que é um *array* de sufixos e como construí-lo, e também discute como resolver o problema LRS (e outros problemas) com o *array* construído. Se você não domina muito bem esse tópico, estude. E também não é porque o algoritmo de construção é um pouco complicado de entender que você deva simplesmente copiá-lo sem saber exatamente como ele funciona. Assim procedendo, você pode não conseguir resolver problemas mais elaborados envolvendo *strings* no futuro.

# G O Enigma do Príncipe

Decidir se uma configuração do jogo *Flood It!* pode ser resolvida em $k$ cliques é um problema $\mathcal{NP}$-completo. A versão apresentada na descrição do problema é um pouco diferente da versão clássica. Na versão clássica, se temos por exemplo o *grid*

```
1 4
0202
```

e clicamos na casa $(1, 4)$, ficamos com:

```
1 4
2202
```

Porém, na nossa versão, esse clique faz com que fiquemos com:

```
1 4
2222
```

Todavia, essa ligeira modificação que fizemos no problema não faz com que ele deixe de ser $\mathcal{NP}$-completo.

Como as dimensões do *grid* são pequenas e o número de cores também, podemos atacar o problema com força-bruta (*backtracking*), tendo alguns cuidados para podar o *backtracking* e evitar um *Time Limit Exceeded*. As estratégias de poda listadas a seguir são suficientes para fazer o código passar:

- Sendo $\min_k$ o menor número de cliques descoberto até o momento que torna a configuração inicial do jogo monocromática, se o número $k$ de cliques já feitos não é menor que $\min_k$, podemos podar o ramo corrente do *backtracking*.
- Quando a escolha de uma cor para recolorir a componente em que se encontra a casa $(0, 0)$ não aumenta o número de casas da componente, podemos podar o ramo corrente do *backtracking* para explorar outras cores (as operações de recoloração podem ser feitas através de uma DFS simples).
- Quando uma cor não ocorre na vizinhança da componente em que se encontra a casa $(0, 0)$, não precisamos explorá-la agora, pois ela certamente não aumentará o número de casasa da componente.

O código no Dropbox é suficientemente claro quanto aos demais detalhes de implementação.

# H As Relíquias da Morte

Não é difícil perceber que o que é pedido neste problema é que encontremos uma árvore geradora mínima (*minimum spanning tree*, *MST*) num grafo completo, em que os vértices do grafo são os segmentos de reta e os pesos das arestas são definidos pelas distâncias entre os segmentos. Vou dividir os comentários da minha solução para este problema em duas partes: na primeira vou abordar o problema de

encontrar uma *MST* num grafo completo, e na segunda vou abordar o problema de calcular a distância entre dois segmentos de reta quaisquer.

Quando usamos o Algoritmo de Kruskal (implementado com *make–union–find*) ou o Algoritmo de Prim (implementado com *heap binária*, no C++ disponível através do *container* `priority_queue`) para encontrarmos a *MST* de um grafo completo, ficamos com um algoritmo de tempo $O(|V|^2 \log |V|)$. Uma leve adaptação do Algoritmo de Prim, contudo, nos conduz a um algoritmo de complexidade $O(|V|^2)$ bastante simples, que dispensa o uso de estruturas de dados mais elaboradas (note que este algoritmo é apropriado somente porque o grafo é completo):

1. Marque todos os vértices do grafo $G$ como não visitados.
2. Escolha um vértice inicial $s$ de $G$, defina $\text{dist}[s] = 0$ e $\text{dist}[u] = \rho(s, u)$ para todo vértice $u \neq s$ de $G$, sendo $\rho(s, u)$ o peso da aresta $su$.
3. Enquanto houver vértices não visitados, visite um vértice $u$ ainda não visitado com $\text{dist}[u]$ mínimo e, para todo vértice $w$ ainda não visitado, atualize $\text{dist}[w]$ para $\rho(u, w)$ se $\rho(u, w) < \text{dist}[w]$.
4. O custo da *MST* é dado pela soma de todos os $\text{dist}[u]$ mínimos tomados.

Perceba que o uso de uma *heap* binária para a extração do mínimo não melhora a complexidade do algoritmo, já que cada vértice $u$ visitado causa a atualização do valor dist[] de um número linear em $|V(G)|$ de vértices $w$. Ou seja, ou pagamos $O(|V(G)|)$ para encontrarmos o $\text{dist}[u]$ mínimo e $O(|V(G)|)$ para atualizarmos todos os $\text{dist}[w]$, ou pagamos $O(\log |V(G)|)$ para encontrarmos o $\text{dist}[u]$ mínimo e $O(|V(G)| \log |V(G)|)$ para atualizarmos todos os $\text{dist}[w]$ — melhor ficarmos com a primeira opção, até porque a implementação é mais simples.

Um outro problema que trata de encontrar *MST* num grafo completo é o problema *Resgate em Queda Livre*, também de minha autoria, disponível no *URI Online Judge*. Nesse problema, mais fácil, os vértices são somente pontos, não segmentos de reta.

Discutamos agora sobre como podemos encontrar a distância entre dois segmentos de reta. Não parece uma tarefa fácil. Há algumas abordagens que podemos utilizar. Em algumas delas, nosso código se torna uma análise complicada de casos recheado de implementações de fórmulas matemáticas elaboradas. Mas eu prefiro uma abordagem mais computacional que parte da mera observação de que a distância entre um segmento $\overline{ab}$ e um segmento $\overline{cd}$ pode ser dada por

$$\min\{\text{dist}(p, \overline{cd}) : p \in \overline{ab}\},$$

e distância entre ponto e segmento de reta é fácil de calcular. O que queremos, então, é encontrar o escalar $\alpha^*$ no intervalo $[0, 1]$ que minimiza a função $f(\alpha) = \text{dist}(a + \alpha(\overrightarrow{ab}), \overline{cd})$ sendo $a + \alpha(\overrightarrow{ab})$ o ponto obtido a partir da translação do ponto $a$ pelo vetor $\alpha(\overrightarrow{ab})$. Note que a função $f$ é decrescente no intervalo $[0, \alpha^*]$ e crescente no intervalo $[\alpha^*, 1]$, o que significa que, ao invés de deduzirmos alguma fórmula mirabolante para $\alpha^*$, podemos aproveitar que temos um computador na nossa frente e realizar uma busca ternária no intervalo $[0, 1]$ para encontrarmos $\alpha^*$.

No código que apresento no Dropbox, itero a busca ternária até que a diferença entre os limites do intervalo da busca seja menor que $10^{-9}$ (não dá *Time Limit Exceeded* porque a busca ternária converge rápido e porque são só $10^3$ buscas que temos de fazer no máximo). Como esses erros menores que $10^{-9}$ podem se multiplicar por $10^3$ (são $10^3$ arestas que são tomadas para a construção da *MST*), retiro um erro de $10^{-6}$ na hora de chamar a função `ceil()` da `math.h`. Todas as saídas para este problema foram conferidas.