Seletiva para Maratona de Programação de 2013

Instituto de Matemática e Estatística

Universidade de São Paulo

Comentários sobre os Problemas

Departamento de Ciência da Computação IME-USP

Problema A: Popularidade no Facebook

Autor do problema: Marcio T. I. Oshiro
Análise: Gabriel R. C. Peixoto e Marcio T. I. Oshiro

Vamos tratar de duas idéias diferentes para resolver esse problema. A primeira resulta em uma solução linear enquanto que a segunda tem complexidade $O(N \lg N)$, os testes dos juízes foram formulados pensando em permitir ambas as soluções.

Solução com Teorema de Erdős-Gallai

O teorema de Erdős-Gallai [1] diz que se $a_1 \geq \ldots \geq a_N \geq 0$ são números inteiros não negativos então existe um grafo simples que tem esses números como os graus de seus vértices se e somente se:

$$\sum_{i=1}^{N} a_i \text{ é um número par} \tag{1}$$

$$\sum_{i=1}^{k} a_i \leq k(k-1) + \sum_{i=k+1}^{N} \min(k, a_i) \qquad \text{para } k = 1, \ldots, N. \tag{2}$$

A intuição por trás de (2) é a seguinte. Do lado esquerdo temos a soma dos k maiores graus. Do lado direito temos um limitante superior dessa soma para que os k maiores graus sejam "servidos" por arestas. Nesse limitante, consideramos que os k vértices de maior grau estão todos conectados através de $k(k-1)/2$ arestas e em cada um dos vértices restantes podem haver no máximo k arestas incidentes vindas dos k vértices de maior grau.

Para resolver o problema, comece verificando se todos os números fornecidos estão entre 0 e $N-1$, se algum não estiver então a resposta é "impossível". A condição (1) também pode ser facilmente verificada em tempo linear.

Agora queremos ordenar a lista de números $a_1, \ldots, a_N$ de maneira decrescente. Como todos são inteiros entre 0 e $N-1$ então isso pode ser feito em tempo linear com um algoritmo de contagem. Isso não é necessário e soluções que fizeram essa passagem com um algoritmo de ordenação $O(N \lg N)$ devem passar no tempo.

A maneira mais ingênua de verificar (2) leva tempo $O(N^2)$ e deve estourar o limite de tempo, isso acontece porque calcular $s_k := \sum_{i=k+1}^{N} \min(k, a_i)$ toma tempo linear.

Podemos acelerar esse processo pre-calculando algumas quantidades. Considere i_k o maior índice tal que $a_{i_k} \geq k$ (por convenção considere $a_0 = +\infty$). Esses índices podem ser pre-calculados e armazenados em uma passagem só no vetor ordenado. Outra opção seria fazer uma busca binária por eles, o que resultaria em uma solução $O(N \lg N)$.

Armazene também em um vetor as somas parciais $r_k := \sum_{j=k+1}^{N} a_j$. Dessa maneira podemos calcular s_k em tempo constante através de quantidades pré-calculadas:

$$\begin{aligned}
s_k &= \sum_{i=k+1}^{N} \min(k, a_i) \\
&= \begin{cases} \displaystyle\sum_{i=k+1}^{i_k} k + \sum_{i=i_k+1}^{N} a_i & \text{se } i_k > k \\ \displaystyle\sum_{i=k+1}^{N} a_i & \text{caso contrário.} \end{cases} \\
&= \begin{cases} k(i_k - k) + r_{i_k} & \text{se } i_k > k \\ r_k & \text{caso contrário.} \end{cases}
\end{aligned}$$

[1] https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93Gallai_theorem

Solução gulosa (Teorema de Havel-Hakimi[2])

Considere o seguinte pseudo-código:

1: **function** EXISTE-GRAFO(N, $a[1 \dots N]$)
2:     **while** $N > 0$ **do**
3:         ORDENA($a[1 \dots N]$) # $a[1] \leq a[2] \leq ... \leq a[N]$
4:         **if** $a[N] >= N$ **or** $a[N - a[N]] == 0$ **then**
5:             **return** False
6:         **else if** $a[N] == 0$ **then**
7:             **return** True
8:         **end if**
9:         **for** $i \leftarrow 1, \dots, a[N]$ **do**
10:             $a[N - i] \leftarrow a[N - i] - 1$
11:         **end for**
12:         $a[N] \leftarrow 0$
13:         $N \leftarrow N - 1$
14:     **end while**
15: **end function**

Se todos os números fornecidos como entrada para esse algoritmo forem inteiros não negativos, ele vai devolver a resposta correta. Deixamos a prova da corretude desse algoritmo como exercício ao leitor.

Note que a linha 10 será executada $\sum a[i]$ vezes, número esse que pode valer até $N(N-1)$. Dessa forma, mesmo eliminando a ordenação da linha 2, esse algoritmo ainda seria quadrático, e portanto muito lento para esse problema. Porém vamos usar a ideia desse algoritmo para construir um outro que será executado em tempo $O(N \lg N)$. Faremos isso usando uma *Binary Indexed Tree*[3] (BIT). Outras estruturas de dados similares podem ser usadas.

Primeiro ordene os números $a[1..N]$ fornecidos em ordem crescente e verifique se todos estão entre 0 e $N - 1$. Agora armazene as diferenças entre eles na bit, de forma que *read(i)* devolva $a[i]$. A partir de agora todas as operações descritas irão consultar e alterar apenas a BIT, mas para simplificar a notação ainda iremos nos referir aos números $a[i]$, que devem ser entendidos como uma leitura de *read(i)*.

Subtrair 1 numa posição i da bit é equivalente à subtrair 1 dos números $a[i], a[i+1], \dots, a[N]$. Assim podemos fazer a operação das linhas 8 à 10 em tempo logarítmico. Resta o problema de manter os números ordenados (ou de maneira equivalente as entradas da bit não negativas).

Os únicos índices que podem estar fora de ordem depois das operações das linhas 8-10 são aqueles que tem o mesmo valor que $k := a[N - a[N]]$. Podemos nos valer desse fato para fazer as subtrações e a reordenação de uma só vez.

Usando duas buscas binárias, encontre os primeiros índices i e j tais que a soma acumulada na BIT Seja pelo menos k e $k + 1$ respectivamente. Pela definição de k sabemos que $k = a[i] = a[i+1] = \dots = a[j-1] < a[j]$ e que $N - i \geq a[N]$. Dessa forma queremos subtrair 1 do vetor nos índices $j, j+1, \dots, N-1$ e também nos primeiros $a[N] - (N - j)$ índices contando a partir de i. Ou seja, subtraímos 1 na BIT nos índices i e j e somamos 1 no índice $i + a[N] - (N - j)$.

Se implementarmos as busca binárias da maneira mais simples, isso é, usando uma busca binária comum e usando o método *read* como sub-rotina, teremos uma solução $O(N \lg^2 N)$ que não deve passar no limite de tempo. Mas podemos nos aproveitar da estrutura da BIT na busca binária e obter uma solução $O(N \lg N)$.

[2] http://en.wikipedia.org/wiki/Degree_(graph_theory)#Degree_sequence
[3] https://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees

Problema B: Duelo de espiões

Autor do problema: Gabriel R. C. Peixoto
Análise: Gabriel R. C. Peixoto

Se rolarmos D dados honestos, com $L_1, L_2, \ldots, L_D$ faces, qual é a distribuição de probabilidade da soma de suas faces? Isso pode ser calculado através de uma PD. Chamemos de $f(i,j)$ a probabilidade de que a soma dos i primeiros dados seja j. Condicionando no resultado do último lançamento podemos montar a recorrência:

$$f(i,j) = \begin{cases} \sum_{k=1}^{L_i} \frac{1}{L_i} f(i-1, j-k) & \text{se } i > 0, j > 0 \\ 1 & \text{se } i = 0, j = 0 \\ 0 & \text{se } i = 0, j \neq 0 \\ 0 & \text{se } j < 0. \end{cases}$$

Com essa recorrência podemos pré-calcular a distribuição de probabilidade de cada um dos ataques. Chamemos de $at_A(i,j)$ e $at_B(i,j)$ as probabilidades de que os i-ésimos ataques do primeiro e segundo jogador respectivamente causem j pontos de dano.

Agora chamemos de $g(A,B,p)$ a probabilidade de que jogador $p = 0,1$ ganhe sendo que é a vez dele jogar e que o primeiro jogador tenha A pontos de vida e o segundo jogador B pontos.

Suponha que o primeiro jogador faça um ataque que cause j pontos de dano, então a sua chance de ganhar será $1 - g(A, B-j, 1)$ já que no próxima vez do jogador B ele começa com $B - j$ pontos e é a vez dele. Dessa forma podemos montar a recorrência:

$$g(A,B,0) = \begin{cases} \max_{1 \le i \le N_A} \sum_j at_A(i,j)\,[1 - g(A, B-j, 1)] & \text{se } A > 0 \\ 0 & \text{se } A \le 0, \end{cases}$$

$$g(A,B,1) = \begin{cases} \max_{1 \le i \le N_B} \sum_j at_B(i,j)\,[1 - g(A-j, B, 0)] & \text{se } B > 0 \\ 0 & \text{se } B \le 0. \end{cases}$$

Assim podemos resolver o problema original novamente com programação dinâmica.

Problema C: As pirâmides de Ecaterimburgo

Autor do problema: Gabriel R. C. Peixoto
Análise: Gabriel R. C. Peixoto

Vamos verificar se o ponto X "enxerga" a face oposta ao vértice A num tetraedro (pirâmide de lados triangulares) de vértices A, B, C e D. Para isso considere o plano que passa pelos vértices B, C e D e olhe para a posição de X e A em relação à esse plano. Se X e A estiverem do "mesmo lado" desse plano, então X não vai enxergar a face oposta ao vértice A pois outras faces da pirâmide vão impedir. Caso eles estiverem em lados opostos do plano então não haverá nenhum obstáculo e X conseguirá enxergar essa face. Um caso de fronteira ocorre se X estiver no plano BCD. Nesse caso ele não vai enxergar a face.

Resta verificar se os pontos A e X estão ou não em lados opostos do plano BCD. Podemos fazer isso usando uma combinação de produtos escalares e vetoriais.

Produtos escalares e vetoriais

Essas são operações entre dois vetores. O produto escalar (ou produto interno) resulta em um número real enquanto que o produto vetorial resulta em um novo vetor.

Dados dois vetores $u = (u_x, u_y, u_z)$ e $v = (v_x, v_y, v_z)$, o produto escalar entre eles será denotado por $< u, v >$ e é definido como:

$$< u, v >= u_x v_x + u_y v_y + u_z v_z.$$

A propriedade que nos interessa do produto escalar é que $< u, v >= \|u\| \|v\| \cos\theta$, onde $\| \bullet \|$ indica a norma do vetor e θ é o ângulo entre eles. Dessa forma se dois vetores apontarem na mesma direção então o produto interno entre eles será positivo, enquanto que ele será negativo se eles apontarem em direções opostas. O produto interno será zero se os vetores forem perpendiculares.

Já o produto vetorial é definido como:

$$u \times v = (u_y v_z - u_z v_y, u_z v_x - u_x v_z, u_x v_y - u_y v_x) = \det \begin{pmatrix} \vec{i} & \vec{j} & \vec{k} \\ u_x & u_y & u_z \\ v_x & v_y & v_z \end{pmatrix}.$$

A segunda forma é mnemônica, nela $\vec{i}$, $\vec{j}$ e $\vec{k}$ representam os três vetores da base canônica. A propriedade que nos interessa dessa operação é que $u \times v$ é perpendicular à u e v.

Voltando ao problema original

Considere $u = C - B$ e $v = D - B$ e calcule o vetor $w = u \times v$, este vetor será perpendicular ao plano BCD. Calcule ainda os vetores $p = X - B$ e $q = A - B$.

Se os pontos A e X estiverem do mesmo lado do plano BCD, então ou os vetores p e q estarão ambos na mesma orientação que w ou ambos estarão em orientação contrária. Isso é, o sinal de $< p, w >$ deve ser igual ao sinal de $< q, w >$, se os sinais forem opostos isso quer dizer que eles estão em lados opostos do plano.

Um caso especial ocorre quando $< p, w >= 0$. Nesse caso o observador está no plano BCD e não consegue ver a face da pirâmide. Note que uma restrição do enunciado garante que $< q, w >\neq 0$.

Problema E: Poker

Autor do problema: Gabriel R. C. Peixoto
Análise: Gabriel R. C. Peixoto

Para resolver esse problema basta implementar cuidadosamente as regras descritas no enunciado. Abaixo estão algumas dicas que podem facilitar a implementação desse problema:

- Ordene as cartas pelo seu valor. Isso facilita bastante para verificar os vários tipos de mão, como também para aplicar as regras de desempate.
- Você pode associar cada mão de cinco cartas à dois inteiros que determinem o quão forte é essa mão. O primeiro simplesmente é um numeração dos tipos de mão, com *Carta mais alta* valendo 1 e *Straight Flush* valendo 9.

  O segundo inteiro é um que vai ser calculado diferentemente para cada tipo de mão e serve para desempatar mãos de mesmo tipo. Por exemplo, para dois pares podemos calcular o segundo inteiro como $13 * 13 * A + 13 * B + C$, onde A é o valor do par mais valioso, B é o valor do segundo par e C é o valor da carta restante. Aqui estou numerando os valores das cartas de 0 a 12 da menos valiosa (dois) para a mais valiosa (ás).

  Dessa forma se quisermos comparar duas mãos que são dois pares, basta comparar o segundo inteiro, quanto maior mais valioso.

  Contas análogas podem ser feitas para todos os tipos de regras de desempate.
- Se você fizer uma rotina que implemente essa idéia de dois inteiros do item anterior, você pode avaliar uma mão de 7 cartas simplesmente calculando os dois inteiros para cada uma das $\binom{7}{5} = 21$ mãos de 5 cartas possíveis e tomar o par de inteiros mais valioso como a representação dessa mão de 7 cartas. Depois basta comparar o par de inteiros de cada jogador para saber quem ganha.

Problema F: Votação em Ecaterimburgo

Autor do problema: Carlos E. Ferreira
Análise: Marcio T. I. Oshiro

Este é um problema puramente de ordenação. Bastava ordenar os candidatos em ordem não-crescente de votos, seguindo o critério de desempate, e imprimir os V primeiros candidato mais aqueles que empatarem com o V-ésimo candidato mesmo após aplicar o critério de desempate.

A dificuldade desse problema era tomar cuidado para não errar na implementação. A solução mais simples é manter um vetor de votos para cada candidato como $V + 1$ posições. Na primeira é armazenado o número total de votos para o candidato e nas demais o número de votos na opção correspondente. Assim, na ordenação, comparamos os valores na primeira posição desse vetor (total de votos). Se empatar, comparamos a posição seguinte (votos como primeira opção). Empatando novamente, passamos para a próxima posição (votos como segunda opção), e assim por diante.

O limite de tempo para este problema estava bem folgado, sendo aceito inclusive algoritmos de ordenação quadráticos, como o *bubble sort*.

Problema G: Banho de sol no jardim

Autor do problema: André H. Pereira
Análise: André H. Pereira

Para resolver esse problema é necessário apenas conhecimento básico de trigonometria. A solução esperada era $O(N)$.

O problema equivalente ao que é pedido no enunciado é o de descobrir qual o ângulo formado entre o topo de cada prédio e o jardim. Os prédios podem ainda ser divididos em 2 grupos, os à esquerda e os à direita do jardim. Observe que não basta considerar apenas o prédio mais alto da direita e o mais alto da esquerda.

Uma vez que os ângulos já foram calculados basta obter o ângulo formado pela diferença entre o maior ângulo dos prédios à esquerda e o maior ângulo dos prédios à direita.

Feito isso uma proporção é necessária para converter o ângulo para minutos.

Problema H: Escalonamento de salas de aula

Autor do problema: Carlos E. Ferreira
Análise: Marcio T. I. Oshiro e Gabriel R. C. Peixoto

Considere cada professor como um vetor indexado pelas turmas, onde cada posição tem valor 1, se o professor dá aula para a turma correspondente, ou valor 0, caso contrário. Agora, o problema consiste em permutar, se possível, as colunas (turmas) de forma que, para cada professor, as posições com valor 1 apareçam consecutivamente. Isso é conhecido como a propriedade dos uns consecutivos (*consecutive ones property*).

Uma forma de resolver esse problema é através de uma estrutura de dados chamada PQ(R)-*tree*[4]. Aqui iremos descrever outras maneiras de resolver.

Solução 1

Seja $T(i)$ o conjunto de turmas para quais o professor i dá aula. Dizemos que os professores i_1 e i_2 se cruzam se $T(i_1) \cap T(i_2) \neq \emptyset$, $T(i_1) \setminus T(i_2) \neq \emptyset$ e $T(i_2) \setminus T(i_1) \neq \emptyset$. Isto é, i_1 e i_2 se cruzam se dão aulas para turmas em comum, mas um não dá aulas para todas as turmas do outro e vice-versa.

Considere o grafo G onde os vértices são os professores e existe uma aresta ligando dois professores se eles se cruzam. Primeiro vamos considerar o caso no qual G tem apenas uma componente conexa. Vamos considerar os professores em uma sequência i_1, i_2, ..., i_n tal que $\{i_1, i_2, \ldots, i_a\}$ induz uma componente conexa em G para $a = 1, \ldots, n$. Isso pode ser facilmente obtido por uma busca em profundidade (*DFS*).

A ideia do algoritmo é manter conjuntos de turmas S_1, S_2, ..., S_k que devem ser considerados nessa ordem, isto é, na permutação final as turmas em S_p precedem as de S_q se $p < q$. As turmas dentro de um mesmo conjunto S_p podem permutar arbitrariamente entre si.

Considere que temos pelo menos dois professores no grafo, caso contrário é trivial. Comece com $S_1 = T(i_1) \setminus T(i_2)$, $S_2 = T(i_1) \cap T(i_2)$ e $S_3 = T(i_2) \setminus T(i_1)$. Os três conjuntos são não vazios, pois i_1 cruza com i_2. Para cada um dos professores restantes fazemos o seguinte. Marcamos todas as turmas para as quais o professor i dá aula. Se todas as turmas de algum S_q estão marcadas, dizemos que S_q está completo. Sejam S_p o primeiro conjunto da esquerda para direita que tem intersecção não vazia com $T(i)$ e S_u o último. Todo S_q, com $p < q < u$, deve ser completo, caso contrário é impossível obter a permutação desejada. Agora temos dois casos:

(*a*) $T(i) \subseteq \{S_1, S_2, \ldots, S_k\}$.

Se S_p não é completo, então o separamos em dois conjuntos $S_{p_1} = S_p \setminus T(i)$ e $S_{p_2} = S_p \cap T(i)$. Substituimos S_p por S_{p_2} e posicionamos S_{p_1} imediatamente à esquerda de S_{p_2}. Um tratamento análogo é feito com S_u, caso ele não seja completo.

(*b*) $T(i) \not\subseteq \{S_1, S_2, \ldots, S_k\}$.

Nesse caso, $p = 1$ ou $u = k$ e não ocorre de S_1 e S_k serem ambos completos, pois i cruza com algum professor anterior. Então temos 3 opções: $S_p = S_1$ é completo, ou $S_u = S_k$ é completo, ou $S_p = S_u$ não é completo. Caso, $S_p = S_1$ é completo, então separe S_u, se necessário, como no caso (a) e adicione um novo conjunto $S_q = T(i) \setminus \{S_1, S_2, \ldots, S_k\}$ à esquerda de S_1. Os outros dois casos são análogos, sendo que no terceiro, a posição do conjunto novo dependerá se $p = 1$ ou $p = k$. Se não ocorrer nenhuma das 3 opções, então é impossível obter uma permutação como desejada.

Isso encerra o algoritmo para o caso de G ter apenas uma componente conexa. O algoritmo completo repetirá esse procedimento para cada componente conexa de G em uma certa ordem. Sejam G_1, G_2, ..., G_c as componentes de G. Seja $T(G_a) = \cup_{i \in V(G_a)} T(i)$, para todo $a = 1, 2, \ldots, c$. Dizemos que G_a precede G_b, ou $G_a \prec G_b$, se, para todo $i \in V(G_a)$ tal que $T(i) \cap T(G_b) \neq \emptyset$, temos que $T(G_b) \subseteq T(i)$. Não é difícil provar que se $T(G_a) \cap T(G_b) \neq \emptyset$ então $G_a \prec G_b$ ou $G_b \prec G_a$.

[4] http://www.ic.unicamp.br/~meidanis/research/pqr/

Logo, processando as componentes por uma ordem topológica dada por $\prec$, obtemos uma permutação desejada, quando existe. Essa ordem é importante, pois se $G_a \prec G_b$ e processamos G_a primeiro, então sabemos que os professores de G_a satisfazem a propriedade dos uns consecutivos. Ao permutar as turmas de $T(G_b)$, G_a continua satisfazendo a propriedade dos uns consecutivos, pois $T(G_b) \subseteq T(G_a)$.

Solução 2

A segunda solução usa de uma estratégia gulosa. Ela também vai funcionar montanto uma lista de conjuntos de salas, indicando que cada conjunto de salas dessa lista deve necessariamente ficar "junto" e que conjuntos consecutivos na lista devem vir um imediatamente após o outro. Mas dentro de cada conjunto qualquer ordem é aceitável.

Comece escolhendo um professor que dê aula para um número máximo de salas (em caso de empate você pode escolher qualquer um). Sua lista de conjuntos de salas vai começar com um único elemento, que são as salas que esse professor dá aula.

Chamemos os conjuntos da lista $A_1, A_2, \ldots, A_k$, chamemos de B o conjunto das salas que ainda não estão nessa lista.

Percorra a lista dos demais professores, chame de C o conjunto das salas que um determinado professor da aula e considere diversos casos (não necessariamente exclusivos) análogos aos considereados na Solução 1:

1. Se C tiver interseção não vazia com elementos não consecutivos da sua lista, então não há maneira de organizar as salas. Nesse caso pare a execução e imprima *impossivel*.

2. Se C fizer intersecção com elementos consecutivos da lista, isso é, se existem $1 \leq a < b < k$ tais que $A_i \cap C \neq \emptyset$ se e somente se $a \leq i \leq b$. Então necessariamente precisa acontecer que $A_i \cap C = A_i$ para $a < i < b$, caso contrário a resposta é *impossivel*. Apenas os elementos das pontas podem ter interseção não cheia. Nesse caso separe esses conjuntos em 2.

3. Se $C \cap B \neq \emptyset$ então teremos que colocar esse conjunto no começo ou final da lista. A escolha do professor com um número maximal de salas faz com que nunca seja possível adicionar esse conjunto tanto no começo quanto no final, a não ser quando a lista consistir de um único elemento, nesse caso você pode adicionar em qualquer lugar.

   Supondo que queremos inserir $C \cap B$ no início, então $C \cap A_1 \neq \emptyset$ e se $C \cap A_2 \neq \emptyset$ então deve acontecer $C \cap A_1 = A_1$, caso contrário não vamos conseguir satisfazer esse professor. Analogamente para inserção no final.

   Se você não conseguir adicionar nem no começo nem no final então a resposta é *impossivel*.

4. Se existir $1 \leq a \leq b \leq k$ tal que $C = \bigcup_{i=a}^{b} A_i$, então todas as restrições desse professor já estão encodadas na lista e podemos marca-lo como processado.

Processe a lista de professores e atualize a lista adequadamente até que não encontre mais nenhum que se enquadre nesses casos. Isso vai acontecer porque todos os professores estão contidos em apenas um dos conjuntos da lista ou ao conjunto das salas sobrando. Nesse caso chame essa rotina inteira recursivamente para as salas que você precisa organizar e os professores que dão aula para essas salas.

Ao final você vai obter uma lista de conjuntos de salas. Imprima essa lista, podendo imprimir cada conjunto na ordem que você preferir. Você também pode escolher se prefere colocar as salas que "sobraram" no início ou final da lista.

Problema I: Entendendo o sorobov

Autor do problema: Marcio T. I. Oshiro
Análise: Marcio T. I. Oshiro

Para cada número N dado, o problema pedia para imprimir uma matriz 8×9 representando uma configuração do sorobov. Cada coluna dessa matriz corresponde a um dígito de N (considerando zeros à esquerda se necessário) e são definidas independentemente das outras.

A tarefa, basicamente, consistia em determinar onde apareceria o 0 acima e abaixo da linha 3 (os hífens) para cada coluna. A posição do 0 acima dependia se seu valor era maior ou igual a 5. A posição do 0 abaixo dependia de seu valor, se fosse menor que 5, ou seu valor subtraido de 5, se fosse maior ou igual a 5. Determinando a posição dos zeros, bastava imprimir a saída corretamente conforme pedido.

Problema J: Turismo em Ecaterimburgo

Autor do problema: Jesus A. P. Mesias
Análise: Marcio T. I. Oshiro

O problema basicamente pedia para determinar, quando possível, quantas cópias de cada arco de um grafo dirigido deveriam ser feitas para que fosse possível percorrer, a partir de um vértice, todos os arcos exatamente uma vez, terminando no vértice de partida. A quantidade de cópias deveria ser determinada de forma a minimizar a dificuldade total da rota.

Para todo vértice v do grafo, denotaremos por $d_e(v)$ a quantidade de arcos que entram em v e por $d_s(v)$ a quantidade de arcos que saem de v. Um grafo com a propriedade pedida é chamado de grafo euleriano. Um grafo dirigido é euleriano se e somente se ele é fortemente conexo e $d_e(v) = d_s(v)$ para todo vértice v. Se o grafo G do problema já for euleriano, não é preciso fazer cópias de arcos. Então, suponha que G não seja.

Para ser possível transformar G em euleriano, através de cópias de arcos, G precisa ser fortemente conexo. Um grafo dirigido é fortemente conexo se de cada vértice é possível alcançar qualquer outro vértice. Essa verificação pode ser feita em tempo linear com algoritmos como Tarjan[5] ou Kosaraju[6]. Mas, como $N \leq 50$, você poderia implementar ideias mais simples, como N buscas em profundidade/largura, ou mesmo usar o algoritmo de Floyd-Warshall[7].

Suponha que G é fortemente conexo, caso contrário é impossível. O próximo passo é determinar quais arcos devem ser copiados e quantas vezes para que $d_e(v) = d_s(v)$ para todo vértice v. Sejam $V^+ = \{v \in V(G) \mid d_e(v) - d_s(v) > 0\}$ e $V^- = \{v \in \ V(G) \mid d_e(v) - d_s(v) < 0\}$. Note que $\sum_{v \in V^+} d_e(v) = \sum_{v \in V(G)^-} d_s(v)$. A ideia será encontrar caminhos partindo de vértices em V^+ e terminando em vértice de V^-. Criando uma cópia de cada arco de um desses caminhos, que começa em algum $v^+ \in V^+$ e termina em algum $v^- \in V^-$, temos que $d_e(v^+) - d_s(v^+)$ diminui de 1, $d_e(v^-) - d_s(v^-)$ aumenta de 1 e para os demais vértices essa diferença continua igual. Logo, no final teremos $V^+ = V^- = \emptyset$. A questão agora é como encontrar os caminhos de forma a minimizar a dificuldade total da rota.

Para encontrar os caminhos, utilizaremos um algoritmo de fluxo máximo de custo mínimo[8][9] no grafo $G' = (V(G) \cup \{s,t\},\ A(G) \cup A')$, onde $A' = \{sv \mid v \in V^+\} \cup \{vt \mid v \in V^-\}$. Considere a seguinte função capacidade para os arcos

$$u(vw) = \begin{cases} d_e(w) - d_s(w), & \text{se } v = s \\ d_s(v) - d_e(v), & \text{se } w = t \\ \infty, & \text{se } vw \in A(G) \end{cases} .$$

O custo de um arco em $A(G)$ é sua dificuldade e o custo de um arco em A' é 0. Um fluxo máximo deve ter valor $\sum_{v \in V^+} d_e(v)$, sendo que cada unidade de fluxo de s a t corresponde a um caminho de um vértice em V^+ para um vértice em V^-. O custo do fluxo máximo mais a soma das dificuldades dos arcos em $A(G)$ é o valor pedido pelo problema.

[5]https://en.wikipedia.org/wiki/Tarjan's_strongly_connected_components_algorithm
[6]https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm
[7]https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
[8]http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=minimumCostFlow1
[9]http://www.ime.usp.br/~pf/flows/

Problema K: O incidente de Sverdlovsk

Autor do problema: Jesus A. P. Mesias
Análise: Marcio T. I. Oshiro

Para este problema, uma algoritmo de força-bruta que, para cada setor, conta quantas áreas circulares o cobrem, é muito ineficiente. Uma forma mais eficiente é fazer a contagem dos setores por coordenada X, isto é, primeiro consideramos os setores com centros $(1, Y)$, depois $(2, Y)$ e assim por diante, para todo $Y \in [1, M]$.

Note que fazemos isso para a coordenada X porque uma restrição do enunciado nos garante que $N \leq 10^3$, enquanto que M pode valer até 10^5.

Seja $i \in [1, N]$. Primeiro, desconsidere os círculos que não cobrem nenhum setor da forma (i, Y), para $Y \in [1, M]$. Seja c um círculo, denotaremos por (i, f_c) o primeiro setor coberto por c e (i, l_c) o último. Note que $1 \leq f_c \leq l_c \leq M$. Vamos aplicar um algoritmo de varredura sobre um vetor que contém os valores f_c e l_c ordenados. Iniciamos um contador com 0 e, sempre que uma posição do vetor for um f_c, incrementamos o contador, se for um l_c decrementamos o contador. Suponha que em f_{c_1} o contador adquiriu valor K e em l_{c_2} é a primeira vez, a partir de f_{c_1}, que o contador fica com valor menor que K. Então, sabemos que os setores (i, f_{c_1}) até (i, l_{c_2}) são cobertos por pelo menos K círculos. Usamos isso para fazer a contagem para cada $i \in [1, N]$ com tempo $O(C \log C)$. Logo a complexidade final do algoritmo fica $O(NC \log C)$.

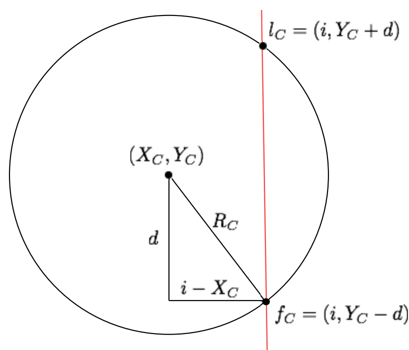

Figura 1: Esboço do cálculo de f_c e l_c.

Problema L: Produção em Ecaterimburgo

Autor do problema: Carlos E. Ferreira
Análise: Marcio T. I. Oshiro

Dada uma lista de N pedidos contendo o tempo no qual o pedido i estará disponível, d_i, e o tempo necessário para processá-lo, p_i, o problema pede para determinar o menor instante no qual todos os pedidos estariam finalizados. Este é um problema de escalonamento onde queremos minimizar o chamado *makespan*, denotado por $C_{\max}$.

Um ponto importante é que existe apenas uma máquina e isso facilita bastante a tarefa, pois basta encontrar a melhor ordem de processamento dos pedidos para determinar $C_{\max}$ em tempo linear.

Uma primeira ideia (gulosa) seria processar os pedidos na ordem em que eles ficam disponíveis, isto é, em ordem não-decrescente do d_i. Essa ideia funciona! Para verificar isso, considere uma ordenação qualquer dos pedidos no qual existe uma inversão. Uma inversão é um par de pedidos a e b tal que b é executado logo depois de a, mas $d_a > d_b$. Note que existe uma inversão se e somente se os pedidos não são processados em ordem não-decrescente do d_i. Como $d_b < d_a$, podemos inverter a ordem de processamento de a e b sem aumentar o $C_{\max}$ (e possivelmente o diminuindo). Logo, podemos remover uma inversão sem piorar a solução. Como a ordenação não-decrescente em d_i é a única que não possui inversões, ela é ótima. A intuição disso é que não vale a pena deixar a máquina ociosa se existem pedidos esperando para serem processados, pois isso atrasaria todos os pedidos ainda não processados.

Portanto, é possível determinar a ordem dos pedidos em tempo $O(N \lg N)$ e a partir dessa ordem, calcular $C_{\max}$ em tempo $O(N)$.