Análise dos Problemas do Treino de 25/08/17

Maratona de Programação Unioeste

5 - Trem ou Caminhão?

- **Problema**

  Calcular qual o modo de transporte com melhor custo-benefício.

- **Solução**

  Apenas implementar o que é pedido.

7 - Restaurante

- **Problema**

  Dados os minutos de entrada e saída de cada cliente dizer o máximo de clientes que estiveram no restaurante ao mesmo tempo.

- **Solução**

  Processar os eventos em ordem crescente de tempo.

  Cliente chegou incrementa contador; cliente saiu decrementa.

  Após cada alteração, guardar a maior resposta.

  Uma implementação simples no C++ pode ser obtida utilizando um vetor de pair<int,int> (minuto / evento) e a função sort.

3 - Imagens de Satélite

- **Problema**

  Dado um grid, contar o número de regiões escuras conectadas por células adjacentes.

- **Solução**

  Apenas implementar um Flood Fill (BFS no grid).

  Ver discussões de outros treinos (problema Duende Perdido na discussão da 2ª seletiva).

2 – Projeto Genoma

- **Problema**

  Dadas duas strings *p* e *t*, encontrar todas as ocorrências de *p* e de complemento(*p*) dentro de *t*.

- **Solução**

  O problema pede para resolver um dos problemas fundamentais de strings: string matching. Existem vários algoritmos para resolver este problema. Se precisarmos apenas dizer se uma string aparece em outra, strstr é a opção mais simples.

  Neste problema precisamos encontrar todas as ocorrências. Embora isso possa ser implementado com strstr é uma opção menos eficiente.

2 – Projeto Genoma

- O algoritmo ‘naive’ para resolver o string matching é simples:

```
for(i = 0; i+m <= n; i++)
{
    for(j = 0; j < m; j++)
        if(p[j] != t[i+j])
            break;

    if(j == m)
        printf(“Achou em %d\n”, i);
}
```

- Em média, este algoritmo tem complexidade O(n) em string naturais ou aleatórias. Porém no pior caso tem complexidade O(nm), por exemplo se T = “AAAAAAAAAB” e P = “AAAAAB”. Nesse caso o algoritmo testaria até o último caracter de P em todos os caracteres de T, mas só encontraria um match na última tentativa.
- Na programação competitiva os juízes quase certamente incluirão casos assim. Com n,m <= 15.000 essa complexidade resultará em TLE.

2 – Projeto Genoma

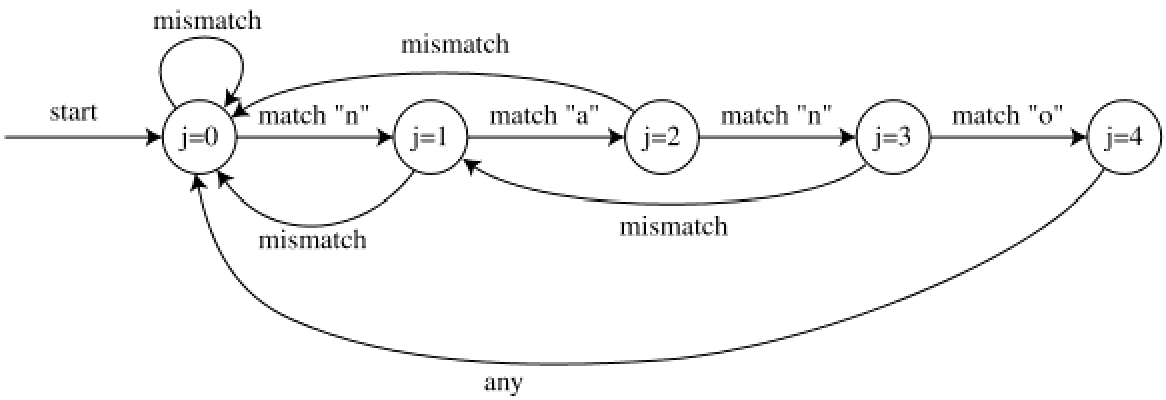
- Um algoritmo que melhora essa complexidade é o KMP.
- O KMP realiza um pré-processamento na string a ser procurada e constrói uma ‘tabela de falha’.
- Essa tabela indica qual a próxima posição que pode ser o início de um matching, permitindo pular todos os testes até aquela posição.
- Por exemplo se T = “AAAAAAAAAB” e P = “AAAAAB”.

```
AAAAAAAAAB          AAAAAAAAAB
| | | | |                    |
AAAAAB                AAAAAB
```

- Ao encontrar a diferença na posição 5, o KMP sabe que se começar a testar na posição 2, os caracteres das posições 1, 2, 3 e 4 darão match, pulando para comparar o caractere 5.

2 – Projeto Genoma

- Um jeito de entender o KMP é como um autômato no padrão sendo procurado.
- Para o padrão “nano”:



- Assim o KMP realiza matching com O(m) pré-processamento e O(n) busca.
- Para implementação, ver cap.6 da bíblia.

9 - Sequências

- **Problema**

  Determinar se uma string é uma sequência-H.

  0 = sequência-H

  1 + [seq-H] + [seq-H] = sequência-H

- **Solução**

  Dependendo dos limites, poderíamos implementar uma solução com programação dinâmica. Mas como o enunciado diz que "não há restrição no comprimento da sequência" (ou seja, prepare-se pra receber sequências bem grandes), isso *indica* que deve haver uma solução mais simples.

9 - Sequências

- Pela definição, sequências-H são da forma:

  0

  1HH
- Ou seja, toda vez que lemos um 0 encontramos um sequência-H; toda vez que lemos um 1 devemos encontrar 2 sequências-H dali pra frente.
- Leu 1? Continuamos a procurar uma sequência-H, mas marcamos que, após a sequência-H atual (se houver), ainda precisamos encontrar outra sequência-H (cont++).
- Leu 0? Encontramos uma sequência-H (cont--).
- No final, deve ter restado 1 sequência-H sem ser causada por nenhum ‘1’ (ou seja, cont final deve ser -1).

9 - Sequências

- Outro jeito de ver é como balanceamento de parênteses: cada 1 é um ‘(‘, cada 0 é um ‘)’. Devemos ter uma sequência válida de parênteses seguida por exatamente um ‘)’ sobrando.
- Prestar atenção ao caso em que o contador ficar negativo (exceto no último caractere) -> nesse caso a sequência fica inválida (mesmo que no final o contador termine em -1). Ex.:
- 100 é válido, mas 010 não

  ())   )()
- Este é um problema ad-hoc, não há um caminho específico para enxergar a solução. Talvez seja mais intuitiva para alguns e menos intuitiva para outros.

4 – Palavras Cruzadas

- **Problema**

  Dado uma palavra-cruzada resolvida, imprimir a lista das palavras que compõe o jogo.

- **Solução**

  Parte do problema (determinar os quadrados com números) era a mesma do problema Palavras Cruzadas da 1ª seletiva.

  Iteramos linha por linha; em cada linha, iteramos do seu começo ao fim (palavras na horizontal) anotando todas as sequências de elementos do mesmo tipo (letras ou quadrados) encontradas. Ao encontrar uma sequência de letras > 1, marcamos em uma matriz de booleanos que esta casa tem número.

4 – Palavras Cruzadas

- Agora fazemos o mesmo coluna a coluna (palavras na vertical), e também marcamos as casas que têm número (pode ser na mesma ou em outra matriz).
- Percorrendo a matriz do canto superior esquerdo ao canto inferior direito (ou seja, modo usual de iterar por uma matriz), toda vez que encontrarmos uma casa marcada como número na matriz1 ou na matriz2, fazemos numero[i][j] = num++;
- As células marcadas na matriz1 são começos de palavras da lista de palavras horizontal, as células marcadas na matriz2 são começos de palavras na lista vertical.
- Não esquecer de ordenar as listas pelo número de cada palavra.

8 - Jogo de Búzios

- **Problema**

  O enunciado define as regras de um jogo e você deve calcular o vencedor e o número de rodadas.

- **Solução**

  O jogo é muito parecido com o problema de Josephus (URI 1030, 1031, 1032).

  Uma solução eficiente para o problema de Josephus usa uma fila em que repetidamente removemos a cabeça da fila e o inserimos de volta no fim.

  Também podemos simular o jogo de búzios usando uma fila, apenas precisamos adicionar um vetor indicando quantos búzios cada jogador tem.

8 - Jogo de Búzios

```
for(int i = 1; i <= n; i++)
    fila.push(i), cont[i] = 1;
cont[k] = 2;

int turno = 1;

while(fila.size() > 1)
{
    int atual = fila.front();
    fila.pop();
    int prox = fila.front();

    if(turno % 2 == 1) passar = 1;
    else               passar = 2;

    v[atual] -= passar;
    v[prox] += passar;

    if(v[atual] > 0)
        fila.push(atual);

    turno++;
}
```

10 - Carga Pesada

- **Problema**

  Encontrar a altura máxima na rota entre duas cidades no grafo.

- **Solução**

  Podemos resolver o problema com uma versão modificada do algoritmo de Dijkstra.

  Inicializamos a melhor altura para todos os nós como -1, e a melhor altura para o nó de origem como INF.

  Em cada iteração, pegamos o nó de melhor altura ainda não explorado e exploramos seus vizinhos. Por aquele caminho, chegaremos no vizinho com altura máxima

  min(melhor[atual], g[atual][vizinho])

10 - Carga Pesada

- min(melhor[atual], g[atual][vizinho]) significa que a melhor altura estará sempre limitada ou pela melhor altura no nodo atual ou pela altura máxima da aresta atual em consideração.
- Se esta altura for maior que a melhor altura vista até agora para aquele vizinho, atualizamos melhor[vizinho].
- Intuitivamente, podemos perceber que o algoritmo está correto observando que ao selecionarmos um nó para ser explorado (nó de maior altura ainda não processado) não há como encontrarmos uma nova rota para este nó com altura melhor do que a altura atual.
- Dijkstra pode ser provado por indução, a prova pode ser adaptada para este algoritmo.
- No entanto, existe uma solução bem mais simples...

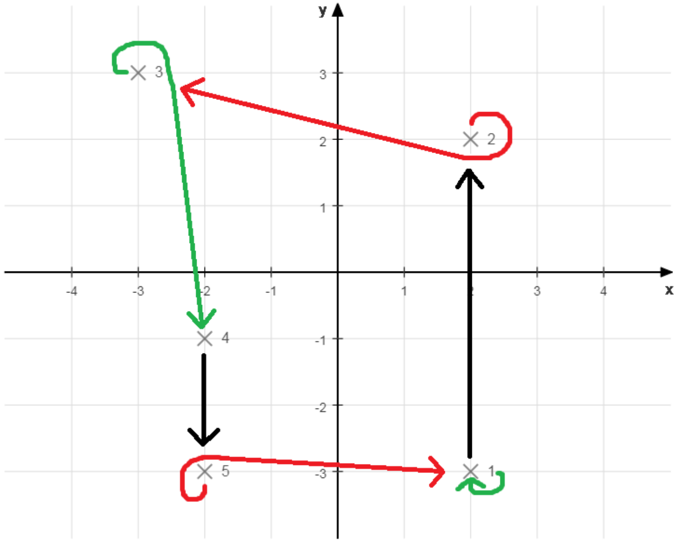
10 - Carga Pesada

- Como a altura de cada aresta C é <= 50, podemos simplesmente testar para cada altura *h* entre 1 e 50, montar o grafo adicionando apenas arestas com *altura* >= *h* (significando: se o caminhão tem altura *h*, ele passa por esta aresta?) e testar se X alcança Y com BFS/DFS. O maior *h* em que X alcança Y é a maior altura possível.
- Ainda, podemos fazer busca binária pelo valor de *h* (se X alcança Y com um caminhão de altura *h*, também alcança com qualquer caminhão menor; se com um caminhão de altura *h* X já não alcança Y, então qualquer caminhão maior também não alcançará).
- Então mesmo que o limite de C fosse alto, poderíamos fazer busca binária para resolver em O(log C * (V+E)).

6 - RoboCoffee

- **Problema**

  Dados os pontos que o robô deve percorrer, e sabendo que ele só consegue girar em sentido horário, calcular quantas voltas completas sobre seu eixo o robô fez.



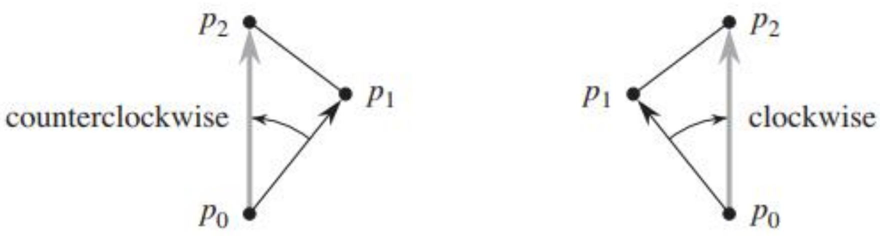
- **Solução**

  Podemos calcular a soma dos ângulos de rotação do robô em cada ponto. Ao final, o valor dessa soma dividido por 2π será o número de voltas completas do robô.

  Ao avançar do ponto *i* para o ponto *i+1*, precisamos determinar se o ponto *i+1* está em sentido horário ou anti-horário em relação à direção atual do robô (dado pelo vetor do ponto *i-1* ao ponto *i+1*).

6 - RoboCoffee

- Este é um dos problemas fundamentais na computação geométrica e pode ser resolvido utilizando produto vetorial.



(a) (b)

$$(p_1 - p_0) \times (p_2 - p_0) = (x_1 - x_0)(y_2 - y_0) - (x_2 - x_0)(y_1 - y_0) .$$

If this cross product is positive, then $\overrightarrow{p_0 p_1}$ is clockwise from $\overrightarrow{p_0 p_2}$; if negative, it is counterclockwise.

- Se o próximo ponto está em sentido horário, o ângulo de rotação será o ângulo entre os vetores p_0p_1 e p_1p_2. Já se estiver em sentido anti-horário, será o ângulo entre os vetores p_1p_0 e $p_1p_2 + \pi$ (que foi o que ele rotacionou para passar sua direção de p_0->p_1 para p_1->p_0).

6 - RoboCoffee

- Para calcular o ângulo entre dois vetores a e b, utilizamos o produto escalar:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \, \|\mathbf{b}\| \cos(\theta)$$

$$\theta = \operatorname{acos}\left(\frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\| \, \|\mathbf{b}\|}\right)$$

1 - Multisoma

- **Problema**

  Dado o tabuleiro do jogo, encontrar a configuração de maior valor.

-3	-1	-2		5	-1	
	-3	2	4		5	-2

-3	-1	-2			5	-1
-3			2	4	5	-2

- **Solução**

  Podemos resolver o problema com programação dinâmica.

  Definimos o estado de um subproblema como sendo: número de peças da 1ª linha que ainda temos que colocar, número de peças da 2ª linha que ainda temos que colocar, e número de quadrados do tabuleiro que ainda temos.

1 - Multisoma

-3	-1	-2		5	-1	
	-3	2	4		5	-2

- Por exemplo:
  - estado (1, 2, 2) = {-3}, {-3, 2}, [ ][ ] (melhor solução: 9)
  - estado (2, 0, 3) = {-3, -1}, {}, [ ][ ][ ] (melhor solução: 0)

- Em cada estado (pa, pb, k) temos 3 possibilidades:
  - 1. podemos parear a última peça de ambas linhas (somando sua multiplicação à resposta) → estado (pa-1, pb-1, k-1)
  - 2. podemos deixar a última peça de cima pareada com um espaço vazio → estado (pa-1, pb, k-1)
  - 3. podemos deixar a última peça de baixo pareado com um espaço vazio → estado(pa, pb-1, k-1)
  - Em tese poderíamos deixar ambas casas vazias e ir para o estado (pa, pb, k-1) mas isso não representa ganho nenhum.

- Os casos base são i == 0 ou j == 0 ou k == 0, nesses casos não há mais pareamento possível e retornamos 0.

1 - Multisoma

- Temos que tomar cuidado também para não cair em estados inválidos (pa > k ou pb > k, pois nestes casos não haveria espaços suficiente no tabuleiro para colocar todas as peças restantes).

```
int solve(int pa, int pb, int k)
{
    if(max(pa, pb) > k)                 return -INF;
    if(k == 0 || pa == 0 || pb == 0) return 0;
    if(calculado[pa][pb][k])         return dp[pa][pb][k];

    calculado[pa][pb][k] = 1;
    return dp[pa][pb][k] = max(solve(pa-1, pb-1, k-1) + a[pa-1] * b[pb-1],
                           max(solve(pa-1, pb, k-1),
                               solve(pa, pb-1, k-1)));
}
```

- Como pa, pb e k variam de 0 a N, temos $O(N^3)$ estados. Cada estado é calculado em $O(1)$, portanto a complexidade final é $O(N^3)$, sendo $N \le 400$ é suficiente para passar no tempo.