Roteiro de estudos para maratona de programação

Autores:
André Luís Tibola
Guilherme Fickel

Versão 0.3
17 de maio de 2010

Sumário

Capítulo 1

Introdução

Este livro tem como principal finalidade ser o ponto de partida para estudos objetivando a maratona de programação. Realizamos análise extensiva dos conceitos mais fundamentais para alunos nos anos iniciais da graduação utilizarem como material de estudo, e procuramos caracterizar os conteúdos presentes na maratona de programação para atender alunos os quais já possuem esses conhecimentos.

1.1 Estrutura do livro

O presente divide-se em três unidades com fins distintos, na parte inicial descrevemos o funcionamento da maratona de programação e traçamos alguns conhecimentos de C++ necessários ao entendimento dos codigos presentes no livro. Na segunda fase apresentamos as grandes áreas da maratona de programação, expomos alguns conceitos e propomos a resolução de problemas para cada uma das áreas apresentadas. A terceira parte consiste em material de apoio para resolução dos problemas propostos, nela apresentamos casos teste adicionais, explicação do problema, erros comuns cometidos nesses problemas e em alguns casos uma solução para ele; também é apresentada uma lista de problemas classificados por área de conhecimento, para estudo posterior.

Eis o objetivo de cada capítulo:

Preparando-se para a Maratona de Programação Apresenta a dinâmica da maratona, formas de estudo e recursos para apoio nos estudos.

C++ Para Programadores C São apresentados conceitos de C++ úteis na maratona de programação, presume-se que o leitor saiba programar em linguagem C.

Basics Neste capítulo é apresentado uma grande classe de problemas conhecida como Ad-Hoc, por não envolverem conceitos adicionais são o ponto de partida para a maratona de programação.

Estruturas de dados e Ordenação Bons conhecimentos sobre estruturas de dados e

ordenação são muito importantes na maratona, neste capítulos apresentamos os principais deles.

Problemas Matemáticos Este capítulo apresenta problemas que necessitam conhecimentos matemáticos específicos para serem solucionados.

Combinatórios São apresentados neste capítulo problemas que relatam a variabilidade combinatória na busca da solução e técnicas para abordar esses problemas: algoritmos gulosos, memorização e programação dinâmica.

Grafos Apresentamos conceitos fundamentais da teoria dos grafos e alguns dos principais algoritmos dessa área.

Geometria Estudamos problemas que envolvem geometria computacional.

1.2 Roteiro de estudos: Como utilizar este material

Todos os testes apresentados no livro possuem uma extensiva análise no anexo **??** caso encontre algum tipo de dificuldade. Recomendamos fortemente a resolução de todos eles, principalmente por serem na maioria problemas que já fizeram parte da maratona.

Traçamos inicialmente três perfis para os leitores deste, sendo a forma de utilizar o livro a seguinte nesses casos:

1.2.1 Alunos de anos iniciais

Independentemente de já ter participado da maratona de programação, recomendamos que os alunos de anos iniciais leiam todo o livro pois permitirá o melhor entendimento da dinâmica da maratona, e possibilitará um maior proveito dos estudos realizados posteriormente.

É muito importante que esses alunos participem dos simulados realizados ao longo do ano, para que entendam melhor a dinâmica da maratona e principalmente para que possam aprender a gerenciar o tempo (e o tempo de uso do computador) durante a prova.

1.2.2 Alunos de anos finais que nunca participaram

Recomendamos a esses alunos que façam uma leitura rápida do capítulo de preparação, também é recomendado a leitura do capítulo sobre C++ pois apresenta algumas construções comuns em problemas da maratona.

Em cada um dos tópicos, o aluno deve observar como é realizada a entrada e saída nos problemas, para isso recomendamos que façam a leitura dos exemplos apresentados, e posteriormente resolvam os problemas sugeridos.

Deve-se participar de todas as simulações que for possível, para ambientar-se com o sistema utilizado e aprender a gerir o tempo.

1.2.3 Alunos de anos finais que já participam

Recomendamos aos alunos que já participam da maratona, que leiam a seção 2.6 para que possam organizar os estudos. Esses alunos podem partir diretamente para os problemas de teste. Posteriormente, pode-se utilizar o anexo **??** para selecionar questões baseado nos tipos de problemas.

É importante para estes alunos, particionar o tempo entre estudos e simulados.

Capítulo 2

Preparando-se para a Maratona de Programação

Neste capítulo apresentaremos questões gerais sobre o funcionamento da maratona de programação tal como forma de correção da prova, respostas dos juízes e placar. Recomenda-se a leitura, especialmente aqueles que nunca participaram da maratona de programação.

2.1 Regras da Maratona de Programação

A maratona de programação é realizada em equipes de três pessoas que podem utilizar um computador. A prova consiste em diversos problemas e tem duração de 5 horas, os programas devem ser resolvidos utilizando-se C, C++ ou Java. Durante este tempo é vetado aos participantes comunicar-se com membros de outros times. Toda a comunicação com a equipe de apoio e os juízes deve ser feita através do sistema do auto-judge via navegador ou através dos "melancias"(membros da organização de camisa verde).

É permitido a utilização de material impresso para consulta, entretanto é vetado o uso de QUALQUER tipo de equipamento eletrônico (tão pouco acesso a internet). O consumo de alimentos e bebidas próprios é critério de cada sede, entretanto geralmente as sedes disponibilizam bebidas e alimentos para os participantes.

2.2 Dinâmica da Maratona

Como dito anteriormente, a prova tem duração de 5 horas e consiste de um número variável de problemas (geralmente entre 9 e 11). Os problemas pode ser resolvidos utilizando-se qualquer uma das linguagens de programação citadas.

Quando um time julga que a solução desenvolvida resolve o problema proposto, esta deve ser submetida aos juízes através da interface web disponível. A solução será submetida a uma bateria de testes (desconhecida dos participantes) e será considerada correta (e pontuará) se resolver corretamente todos os casos teste.

Durante a competição, sempre deve ser chamado um melancia para realizar qualquer ação que resulte no deslocamento da mesa em que sua equipe está (como ir ao banheiro ou comer). É possível imprimir qualquer código/texto desenvolvido durante a maratona através da interface web. Também é possível esclarecer dúvidas através da mesma.

A cada questão correta, é disposto na mesa da equipe um balão de cor correspondente ao problema. A prova segue dessa forma até os 25 minutos finais, quando o placar é congelado (mas os balões continuam chegando). Nos 5 minutos finais, ainda é possível submeter mas não é mais possível saber se o problema está correto ou não.

Cada uma das provas possui duas fases:

WarnUP Antes de cada "maratona"(dependendo o caso, no dia anterior) é realizada uma prova com geralmente duas questões com a finalidade de testar o sistema, e para os maratonistas ambientarem com o equipamento.

Prova A prova propriamente dita.

A maratona de programação pertence a um evento maior chamado ACM-ICPC International Collegiate Programming Contest que é dividido em 3 fases, sendo umas classificatórias para as outras:

Regional Brasileira Primeira fase da maratona, realizada em diversas sedes no brasil, todas as sedes do Brasil fazem a mesma prova.

Regional Sulamericana Conhecida também como final brasileira, é uma prova que acontece em diversas sedes na américa do sul (apenas uma sede em cada país), onde todos as sedes da américa realizam a mesma prova, todas da ásia outra, e assim por diante.

ACM-ICPC Conhecida por final mundial, é o contest propriamente dito. É realizado em apenas uma sede no mundo (definida a cada ano) e define quem é o ganhador mundial.

2.3 Formato das questões

De forma geral as questões contém as seguintes seções:

Background Pequeno texto sobre o problema, geralmente com descrição e caracterizaçao do mesmo.

Problem Em alguns casos é apresentado uma seção que explica o problema proposto de forma mais resumida.

Input Apresenta a forma que a entrada estará disposta e os limites de valores para a entrada.

Output Formato da saída que deve ser produzida (deve ser seguido estritamente).

Sample Input Exemplo de entradas possíveis.

Sample Output Saída esperada para as entradas de exemplo.

É importante notar que a saída deve seguir o formato descrito de forma exata.

2.4 Correção da Prova

As questões da maratona são corrigidas automáticamente, a entrada de dados para o programa é sempre feita através da entrada padrão ("teclado") e a saída dos resultados sempre é para a saída padrão ("tela"). Isto quer dizer que se o problema pedir para ler dois inteiros, somá-los e mostrar o resultado, deve-se: Ler dois inteiros do "teclado"e mostrar a soma na "tela".

Durante a correção do problema (e nos juizes online), pode-se ter diferentes tipos de resposta do juiz, dependendo da situação:

Compilation Error O programa não compilou:Deve-se verificar se a linguagem de programação foi escolhida corretamente e verificar possíveis erros.

Run time error O programa compilou corretamente, entretanto teve um erro em tempo de execução: Deve-se verificar a solução na busca de possíveis erros de programação que originem acessos inválidos de memória, estouros de pilha, falhas de segmentação...

Wrong Aswer O programa compilou e executou sem erros, entretanto não forneceu a saída correta para todos os casos: Deve-se verificar se algoritmo proposto funciona para todos os casos, revisar o problema na busca de possíveis condições não detectadas anteriormente.

Presentation Error O programa compilou e executou sem erros, além disso forneceu a resposta correta, entretanto possui algum erro de formatação da saída: Deve-se verificar se a saída está no formato especificado pela questão.

Time Limit Exceeded O programa demorou mais para terminar doque o permitido pelos juizes: Deve-se verificar a possibilidade do programa não estar parando em determinados casos. Deve-se verificar se o algoritmo proposto é rápido o suficiente para o problema proposto.

Accepted O programa compilou e executou sem erros, além disso forneceu a resposta correta para todos os casos teste: O programa está correto, um balão está a caminho...

2.5 Pontuação e classificação

A classificação na maratona é feita primariamente pelo número de problemas resolvidos. O critério para desempate é o *score* da equipe, que nada mais é que a soma dos instantes

em que a equipe submeteu cada um dos problemas corretos, mais as penalidades. Fica na frente a equipe com menor score.

Caso uma equipe resolva três problemas, que foram submetidos aos 30 minutos, 150 minutos e 250 minutos, o score total é 430.

Todas as penalidades são iguais, sendo apenas consideradas caso o problema que originou a penalidade seja aceito. Qualquer submisão com resultado diferente de *Accepted* originará uma penalidade de 20 (minutos) caso o problema seja aceito posteriormente, as penalidades são cumulativas. Por exemplo, consideremos as seguintes submisões de uma equipe:

Problema E - 45 minutos Accepted

Problema D - 120 Wrong Aswer

Problema D - 130 Wrong Aswer

Problema D - 150 Accepted

Problema A - 200 Time Limit Exceeded

Problema A - 210 Time Limit Exceeded

Problema A - 220 Time Limit Exceeded

Problema G - 240 Accepted

A equipe resolveu 3 problemas (primeiro critério de classificação). O score será determinado pelo momento que os problemas aceitos foram submetidos, isto é: $45+150+240 = 435$ mais as penalidades: 40 do problema D. Como o problema A não conseguiu accepted, as penalidades dele não são consideradas. Assim, o score total da equipe é 475.

2.6 Tipos de questões

Identifica-se 3 grandes áreas de problemas na maratona de programação:

- Problemas Ad-Hoc
- Problemas matemáticos
- Problemas computacionais

2.6.1 Problemas Ad-Hoc

Os problemas Ad-Hoc são assim chamados por não se utilizam de algum conceito especial, de forma que as soluções desenvolvidas para esses problemas são de fato para este fim específico, não tendo utilidade em outras situações.

Dentro dos problemas Ad-Hoc, podemos reconhecer dois grupos mais específicos de problemas:

- Problemas de autômatos
- Problemas de simulação

Problemas de autômatos são situações onde a entrada deve ser processada através de uma computação semelhante a autômatos - sem retrocessos, com operações ou requisitos bem definidos e, principalmente, de forma determinística. Já os problemas de simulação envolvem a reprodução de passos descritos pelo programa, mas nem sempre é possivel determinar exatamente como se dará a execução da série de "passos"descritos pelo problema. Enquanto problemas de autômatos geralmente são fáceis, problemas de simulação podem ser muito complicados de serem resolvidos - e da mesma forma muitas vezes são fáceis; a determinação da dificuldade de um problema de simulação é estudada na seção **??**

Existem outras categorias de problemas Ad-Hoc, podemos por exemplo ter situações onde é necessário "inventar" uma fórmula ou fazer pequenos conjuntos de testes. Embora a simplicidade de muitos desses problemas, o reconhecimento dos mesmos é muito importante para o melhor aproveitamento do tempo disponível durante a prova.

2.6.2 Problemas matemáticos

Nos problemas matemáticos, podemos reconhecer 3 grandes grupos de questões:

- Problemas com fórmulas clássicas
- Problemas com fórmulas menos conhecidas
- Problemas envolvendo números de tamanho arbitrário
- Problemas envolvendo geometria
- Problemas de geometria computacional

Problemas com fórmulas clássicas são extremamente comuns, entretanto frequentemente podem estar disfarçados em problemas que aparentam ser ad-hoc. Exemplos comuns são a sequência de Fibonacci, fatorial e números primos.

Alguns problemas que parecem ad-hoc, na realidade utilizam fórmulas já divulgadas que não são amplamente conhecidas, como por exemplo a função piso que calcula o número de zeros no final do fatorial de X, baseado em divisões de X por potências de 5.

Outro tipo comum de problema é onde necessitamos realizar operações sobre inteiros muito grandes, digamos 100 casas decimais. Esses problemas geralmente são resolvidos tendo o auxílio de código já pronto que implementa essas operações.

São também bastante comuns na maratona, problemas envolvendo figuras geometricas simples, como triangulos, triangulos circunscritos, e assim por diante. Por se tratarem de figuras conhecidas a bastante tempo, geralmente utilizam fórmulas já presentes em livros.

Além da geometria com formas básicas, estão presentes problemas que envolvem poligonos, cálculo da área da interecção de figuras geometricas e afins. Alguns problemas com poligonos possuem solução conhecida, entretanto a maior parte deles é necessário esboçar os desenhos no papel e verificar possíveis formas de resolver a situação proposta.

2.6.3 Problemas computacionais

Os problemas ditos computacionais envolvem conhecimentos específicos de análise de algoritmos, das quais podemos destacar quatro assuntos:

- Problemas de ordenação
- Problemas de estruturas de dados
- Problemas combinatórios
- Problemas de Grafos

A ordenação é um campo muito importante na computação, e naturalmente é um assunto sempre presente na maratona. Além de problemas onde a complexidade fica no entorno de funções de ordenação, sempre é necessário utiliza-lá em variados problemas como forma de simplificação.

Outro fundamento importante são as estruturas de dados, na maratona é comum aparecerem problemas onde o conhecimento de uma estrutura de dados específica pode facilitar o processamento ou mesmo resolver o problema.

Os problemas combinatórios são relacionados a variabilidade de combinações que acontecem quando tenta-se encontrar uma solução para o problema proposto. Alguns problemas podem utilizar uma busca exaustiva pela solução e não terão nenhum problema, entretanto alguns outros necessitam de adequação para que seja possível realizar o proposto pelo problema. São comuns problemas de encontrar a melhor combinação, maximar ou minimizar alguma variável e assim por diante.

Uma grande área na computação é a teoria dos grafos, esta área também está constantemente presente na maratona de programação. É necessário bom conhecimento dos algoritmos clássicos de grafos para que na competição tenha-se capacidade de modificá-los ao sabor dos desafios propostos.

2.7 Material de apoio

Além deste livro, sugerimos alguns sites que funcionam como canalizadores para conteúdos de estudo para a maratona de programação. São eles:

- http://maratona.wiki.br/index.php/P%C3%A1gina_principal
- http://www.algorithmist.com/index.php/Main_Page
- http://algorithmscafe.inf.ufes.br/index.php/Livros
- http://olimpiada.ic.unicamp.br/info_geral/programacao/programacao/Apostila
- http://lampiao.ic.unicamp.br/maratona/

2.8 Como testar os problemas resolvidos durante estudos

É possível testar as soluções para os problemas propostos como estudo no livro, para isso deve-se utilizar os juízes disponíveis online. Esses juízes utilizam o mesmo padrão de resposta que os utilizados durante a competição.

Além dos problemas propostos, esses sites com juízes online possuem muitos outros problemas para teste. São utilizados principalmente três juízes:

1. Spoj - http://br.spoj.pl/
2. UVa Online Judge - http://uva.onlinejudge.org/
3. ACM Live Archive - http://acmicpclive-archive.uva.es/nuevoportal/

Todos os sites é necessário cadastro para poder submeter os problemas. Ao longo do livro, sempre será indicado qual a fonte do problema em questão para que possa ser submetido na interface web. Destacamos que o Spoj é em português, enquanto os demais são em inglês.

Os conteúdos do Spoj são focados nas regionais brasileiras, regionais sulamericanas, seletivas realizadas ao redor do país, e competições da Olimpiada Brasileira de Informática (OBI). É muito recomendado especialmente para aqueles que estão iniciando na maratona de programação, pois transparece com fidelidade as questões encontradas na maratona de programação do Brasil.

O ACM Live Archive contém os arquivos de todas as competições realizadas no ACM International Collegiate Programming Contest (ICPC) - evento o qual compreende a maratona de programação. É possível encontrar as provas de todas as regionais continentais, isto é, regionais sulamericanas, regionais asiáticas... É indicado especialmente para aqueles que desejam conhecer as questões das regionais.

No UVa Online Judge encontramos um apanhado geral dos problemas de todas as maratonas de programação ao redor do mundo e mais alguns outros. É indicado para estudos direcionados, onde se procura assuntos específicos.

Capítulo 3

C++ Para Programadores C

Neste capitulo abordaremos alguns aspectos de C++ necessários para o entendimento dos códigos fonte utilizados no livro. Assume-se que o leitor tenha familiaridade com programação utilizando C. De forma geral, todas as funções utilizadas em C podem ser utilizadas em C++ sem problemas, dessa forma apresentaremos aqui apenas "incrementos"presentes no C++.

Em caso de dúvidas, recomenda-se consultar o site www.cppreference.com ou www.cplusplus.com que contém os principais aspectos de C++ de forma suscinta.

3.1 Compilando um programa C++

A primeira diferença é que nomearemos nossos arquivos com extensão ".cpp" e utilizaremos um comando diferente para compilação, o "g++". Adicionalmente, sempre utilizaremos os parâmetros de compilação que são utilizados pelo juiz da prova:

-lm Para linkar com a biblioteca "math", necessária para funções como seno, cosseno...

-02 Para otimizar o programa

Dessa forma, nosso comando para compilação será:

```
g++ -O2 -lm arquivofonte.cpp
```

3.2 Passagem por referência

Outra diferença do C++ é que podemos utilizar passagem por referência sem utilizar ponteiros, adicionando o símbolo "&" na declaração da variável. No exemplo abaixo, a variável menor é passada como referência. Ao final da chamada da funćão, "menor" tem valor 9;

Listing 3.1: Passagem por referência em C++

```
1  void calcula(int &m)
2  {
3       m = m - 1;
4  }
5
6  int main()
7  {
8       int menor = 10;
9       calcula(menor);
10      return 0;
11 }
```

3.3 Entrada e Saida

Em C++ é possível fazer entrada e saída tal como em C, isto é, através de "scanf" e "printf". Além disso, é possível fazer entrada e saída utilizando streams que é um método mais prático, e permite entrada e saida de strings (próxima seção).

Observe o exemplo abaixo, nele lemos 3 inteiros e mostramos sua soma:

Listing 3.2: I/O Utilizando stream em C++

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6       int a, b, c;
7       cin >> a >> b >> c;
8       a = a + b + c;
9       cout << "Resultado:" << a << "\n";
10      return 0;
11 }
```

Para utilização de I/O com streams devemos declarar o cabeçalho "<iostream>" (linha 1), e utilizar o namespace "std" (linha 2).

A entrada padrão é acessada através do objeto "cin" (linha 7) e a saída padrão através de "cout" (linha 9). É necessario ainda utilizar o operador especial ">>" para a entrada de dados da stream e "<<" para a saída (linhas 7 e 9).

Numa mesma linha utilizando entrada a partir de "cin" pode-se ler variáveis de tipos diferentes. Também é possível, fazer saída de tipos diferentes. Na linha 9 (nove) é feita saída de constantes literais e também de um inteiro (isso é possível pelo explicado no parágrafo seguinte).

Outra questão importante é que uma operação do tipo "cin >> x" retorna como resultado "cin", entretanto quando não é mais possível ler da entrada padrão (fim de arquivo) a operação descrita retorna "false". Isso permite processar toda a entrada facilmente, a técnica utilizada no exemplo a seguir será muito utilizada ao longo de todo livro.

Listing 3.3: Utilizando stream para processar toda a entrada

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int a;
7
8      while(cin >> a)
9        cout << a*a << endl;
10
11     return 0;
12 }
```

Neste exemplo lemos quantos números forem digitados, e mostramos o quadrado do número (para encerrar a digitação pressione Ctrl-d). A linha 8 deve ser entendida como: "enquanto conseguimos ler algum valor para "a". O "endl" na linha 9 é equivalente a "\n", entretanto é mais utilizado por ser de fácil digitação e evitar alguns erros.

3.4 Strings

Em C++ não utilizaremos vetores de char quando desejarmos manipular strings de caracteres. C++ fornece uma classe própria para strings, contendo as principais funções para manipulação de literais. As strings em C++ são o assunto dessa seção.

Uma primeira consideração muito importante é que strings são um tipo[1]. As principais funcionalidades que utilizaremos estão implementadas como métodos desta classe, e os principais operadores já estão sobrepostos [2] . Além disso, o operador de endereçamento de vetores ("[ ]") pode ser utilizado para acessar os caracteres da string individualmente. Os principais métodos da string podem ser consultados em http://www.cppreference.com/wiki/string/start , sendo os principais:

== Comparação entre strings (possui todos operadores de comparação)

+ Concatenação de strings

= Atribuição de uma string para outra (não utiliza-se strcpy)

"[ ]" Acessar elementos da string

size() Tamanho da string

A entrada e saída de strings deve ser feita utilizando-se cin e cout (como mostrado anteriormente). O exemplo abaixo lê uma string e conta o número de letras 'a'. Para utilizar strings, devemos declarar o cabeçalho "string"e utilizar o namespace std.

[1] Caso não saiba o que são tipos, uma breve descrição é dada em http://pt.wikipedia.org/wiki/Orienta%C3%A7%C3%A3o_a_objetos

[2] http://pt.wikibooks.org/wiki/Programar_em_C%2B%2B/Sobrecarga_de_operadores

Listing 3.4: Strings em C++

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7    string str;
8    int i, count;
9    cin >> str;
10   count = 0;
11   for(i = 0; i < str.size(); i++)
12     if(str[i] == 'a')
13       count++;
14   cout << count << "\n";
15   return 0;
16 }
```

Declaramos a variável do tipo string na linha 7 e fazemos leitura utilizando cin na linha 9. Na linha 11, utilizamos "str.size()" como limite do for, isto é, o valor de i irá variar de 0 até o número de caracteres da string lida (menos 1). Na linha 12, utilizamos "str[i]" para acessar cada um dos caracteres da string.

Assim como scanf, a leitura de strings com cin é delimitada pelos espaços em branco. Se desejamos ler **toda** a linha, utilizaremos a função "getline" da "iostream". Essa função é mostrada no código a seguir.

Listing 3.5: Utilização do getline

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7    string str;
8    int i, count;
9    getline(cin, str);
10   count = 0;
11   for(i = 0; i < str.size(); i++)
12     if(str[i] == 'a')
13       count++;
14   cout << count << "\n";
15   return 0;
16 }
```

A única modificação é na linha 9, para notar a diferença entre os dois modos deve-se escrever frases com espaços, como "maratona da acm"ou "apostila para maratona".

3.5 Escopo de variáveis

Outro recurso muito utilizado de C++ é o escopo de variáveis. A variável é apenas acessível no escopo que é definida, dessa forma uma variável declarada no inicio do "main" é acessível por todo o main, enquanto uma variável definida em um for é apenas válida dentro daquele for.

O exemplo abaixo mostra a utilização da variável "tmp" dentro do if. A variável tmp não é acessível nem mesmo no for, apenas até o "}" do if.

Listing 3.6: Escopo de variáveis

```
1  int main()
2  {
3    int i, vet[10];
4
5    for(i = 0; i < 10; i++) {
6      if(vet[i] > vet[10 - i]) {
7        int tmp;
8
9        tmp = vet[i];
10       vet[i] = vet[10 - i];
11       vet[10 - i] = tmp;
12     }
13   }
14   return 0;
15 }
```

O escopo também é muito utilizado para a declaração de variáveis na declaração de um for, poupando assim algum código. O exemplo abaixo utiliza esse recurso, ao longo do livro esse tipo de declaração será utilizada.

Listing 3.7: Escopo de variáveis

```
1  int main()
2  {
3    int vet[10];
4
5    for(int i = 0; i < 10; i++) {
6      if(vet[i] > vet[10 - i]) {
7        int tmp;
8
9        tmp = vet[i];
10       vet[i] = vet[10 - i];
11       vet[10 - i] = tmp;
12     }
13   }
14   return 0;
15 }
```

A grande diferença é a declaração da linha 5, devido a comodidade dessa abordagem ela é amplamente utilizada.

Outra coisa que deve ser notada quando tratamos de escopo é que todas as variáves são desalocadas quando saem de escopo. Isso facilitará a escrita de algoritmos utilizando C++.

3.6 Standart Template Library (STL)

Em C++ é possível ainda, escrever algoritmos e classes que são independentes do tipo de dados utilizada, para isso é necessário utilizar templates (Mais detalhes sobre os templates podem ser obtidos em http://www.cplusplus.com/doc/tutorial/templates/). Na maratona estaremos interessados em utilizar algumas dessas classes que já estão previamente disponíveis. Essas classes e algoritmos já disponíveis formam a chamada Standart Template Library (ou simplesmente STL), para consulta dos principais componentes da STL pode ser utilizado esta página http://www.cppreference.com/wiki/stl/start como ponto de partida.

3.6.1 Classes

Discutiremos agora as principais classes da STL que utilizaremos nos códigos da maratona de programação.

Pair

O template mais simples (e um dos mais utilizados) é o pair. Este template possibilita a criação de pares de valores (uma estrutura de dados par que representa pares) que podem ser acessados como "first" e "second". O principal par utilizado será de dois inteiros, e para tal utilizaremos um *typedef* para simplificar a **criação** de estruturas do tipo par de inteiros.

Observe o exemplo:

Listing 3.8: Utilização de um Par de Inteiros

```
1  #include <iostream>
2  #include <utility>
3  using namespace std;
4
5  typedef pair<int, int> ii;
6
7  int main()
8  {
9    ii a;
10
11   cin >> a.first >> a.second;
12   ii b(0,0);
13   a = b;
14
15   return 0;
16 }
```

Na linha 5 criamos a definição de "ii" como um par de inteiros (a notação <int, int> é a forma de explicitar os tipos quando utiliza-se templates, mais informações podem ser obtidas na referência mostrada anteriormente). Note-se que podemos criar um par sem valores como feito na linha 9 ou com valores como feito na linha 12. Os elementos do par são acessados com "first" e "second" como na linha 12[3]

Mais sobre pair pode ser encontrado em http://www.cplusplus.com/reference/std/utility/pair/

Vector

É possível em C++ utilizar vetores da mesma forma que C, entretanto durante a maratona deve-se evitar alocação dinâmica ao estilo C pois pode ocasionar erros e assim tomar muito tempo da prova com debug. Em C++ utilizaremos a classe vector para essa tarefa pois ajuda a evitar transtornos na hora da prova, mas mantém os benefícios a alocação dinâmica (e ainda alguma ajuda extra).

O vector é outro template muito utilizado, ele permite a criação de vetores para qualquer tipo de dado, tem as principais funções necessárias para gerenciamento e pode ser endereçado diretamente com "[ ]".

Os principais métodos da vector são:

"[ ]" Permite acessar os elementos do vector

clear() Remove todos os elemetos do vetor.

push_back(X) Adiciona um elemento após o final do vetor (cria uma nova posição com o valor X)

rezise(N) Redimenciona o vetor para ter N posições.

size() Retorna o tamanho do vetor

O exemplo abaixo lê um número arbritrário de inteiros para o vector e após calcula a soma. A saída do programa é o número de elementos lidos e a soma deles. A classe vector pertence ao cabeçalho "vector"

Listing 3.9: Utilização da vector

```
1 #include <vector>
2 #include <iostream>
3 using namespace std;
4
5 int main()
6 {
7   vector<int> vet;
8   int x, soma;
9
```

[3] não é utilizado () ao acessar os elementos do par pois os elementos são campos do par e não métodos, mais informações devem ser consultadas na referência de Orientação a Objetos

```
10   while(cin >> x)
11     vet.push_back(x);
12
13   soma = 0;
14   for(int i = 0; i < vet.size(); i++)
15     soma = soma + vet[i];
16
17   cout << "Numero de elementos:" << vet.size() << "\n";
18   cout << "Soma:" << soma << "\n";
19 }
```

Inicialmente o vetor não possui nenhum elemento. A linha 10 garante que toda entrada será lida (para encerar a entrada de dados basta digitar Ctrl + D), enquanto a linha 11 faz tanto o trabalho de alocar mais espaço quanto salvar os valores no vetor.

Na linha 15 acessamos o vetor da mesma forma que um vetor em C, e da mesma forma, ocorerrá falha de segmentação caso sejam acessados indices inválidos. Para evitar isso, utilizamos "vet.size()" (linha 14) para saber o tamanho do vector.

Como já dito, podemos utilizar qualquer tipo de dados com as classes da STL, o exemplo a seguir demonstra a utilização de um vector de pair. Neste exemplo lemos um número arbitrário de pares e calculamos a soma das diferenças.

Listing 3.10: vector de pair

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4
5  typedef pair<int, int> ii;
6
7  int main()
8  {
9    vector<ii> vet;
10   ii x;
11   int soma;
12
13   while(cin >> x.first >> x.second)
14     vet.push_back(x);
15
16   soma = 0;
17   for(int i = 0; i < vet.size(); i++)
18     soma += vet[i].second - vet[i].first;
19
20   cout << "Numero de elementos:" << vet.size() << "\n";
21   cout << "Soma das diferencas:" << soma << "\n";
22 }
```

Mais exemplos com vector podem ser encontrados em http://www.cppreference.com/wiki/stl/vector/

Queue

A classe queue implementa uma fila (http://pt.wikipedia.org/wiki/FIFO), provendo as principais funcionalidades:

empty() Verdadeiro quando a fila não possui nenhum elemento

front() Referência para o primeiro elemento da fila

pop() Remove o primeiro elemento da fila

push(X) Adiciona o elemento X na fila (no final)

A baixo um exemplo da utilização das queue, nele lemos um número qualquer de elementos e posteriormente calculamos sua soma.

Listing 3.11: Filas em C++

```
#include <queue>
#include <iostream>
using namespace std;

int main()
{
  queue<int> q;
  int i, soma;

  while(cin >> i)
    q.push(i);

  soma = 0;
  while(!q.empty()) {
    soma += q.front();
    q.pop();
  }

  cout << soma << "\n";
  return 0;
}
```

É importante notar que a fila não possui acesso aleatório, apenas sequencial. Mais informações sobre queue em http://www.cppreference.com/wiki/stl/queue/start.

Stack

A classe stack implementa uma pilha (http://pt.wikipedia.org/wiki/LIFO), tendo as principais operações:

empty() Verdadeiro quando a pilha não possui nenhum elemento

top() Referência para o topo da pilha

pop() Remove o elemento do topo da fila

push(X) Adiciona o elemento X na pilha (no topo)

O exemplo abaixo mostra os números lidos na ordem inversa.

Listing 3.12: Pilha em C++

```
#include <stack>
#include <iostream>
using namespace std;

int main()
{
  stack<int> s;
  int i;

  while(cin >> i)
    s.push(i);

  while(!s.empty()) {
    cout << s.top() << "\n";
    s.pop();
  }

  return 0;
}
```

Map

O map implementa um array associativo de chave e valor, isto é, possibilita que um array guarde valores do tipo que desejarmos e seja indexado pelo tipo que desejarmos.

O map que mais utilizaremos é um associação de inteiros indexados por string. Este map é mostrado no exemplo a seguir.

Listing 3.13: Map em C++

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main()
{
  map<string, int> mp;
  string s;
  int id = 1, t;

  while(cin >> s) {
    //consultamos o map
    t = mp[s];
```

```
15
16      //ja existe?
17      if(t == 0) {
18        //nao: adicionamos no map
19        mp[s] = id;
20        id++;
21      } else {
22        //sim: mostramos o valor
23        cout << mp[s] << "\n";
24      }
25    }
26
27    return 0;
28  }
```

Na linha 8 criamos um map que utiliza uma chave do tipo "string", para indexar valores do tipo "int". Isso quer dizer que podemos fazer uma atribuição do tipo "mp["batatinha"] = 123;". Quando acessamos um indice que não existe (ele passa a existir) é retornado o valor 0, dessa forma devemos evitar utilizar o valor 0 quando trabalhamos com map.

O intuido deste exemplo é atribuir ids para cada uma das strings lidas. Digitamos quantas strings quisermos (linha 12), para cada string verificamos seu valor no map (linha 14), como dito, caso nenhum valor tenha sido atribuído a essa chave o valor retornado no acesso é 0, essa é a forma de saber que determinada string ainda não foi digitada. Caso o valor ainda não exista, atribuímos a ele um id (linha 19), caso a string já tenha recebido um id anteriormente, mostramos o id da string (linha 23).

3.6.2 Algoritmos

A STL também possui diversos algoritmos disponíveis, eles são providos pelo cabeçalho "algorithm". A lista completa pode ser obtida em http://www.cppreference.com/wiki/stl/algorithm/start, apresentaremos aqui apenas a ordenação por ser o mais utilizado na maratona.

sort

A rotina sort provê a ordenação, podendo ser através de critérios pré-estabelecidos ou utilizando um critério próprio de comparação.

As duas formas mais utilizadas para sort são:

```
sort(iterador_inicial, iterador_final);
sort(iterador_inicial, iterador_final, rotina_comparacao);
```

O exemplo abaixo demonstra a utilização da primeira forma de sort.

Listing 3.14: STL sort

```
1 #include <iostream>
2 #include <algorithm>
3 #include <vector>
```

```
4  using namespace std;
5
6  int main()
7  {
8    vector<int> v;
9    int x;
10
11   while(cin >> x)
12     v.push_back(x);
13
14   sort(v.begin(), v.end());
15
16   for(int i = 0; i < v.size(); i++)
17     cout << v[i] << "\n";
18
19   return 0;
20 }
```

Observe que na linha 14 (quando utilizamos sort), os iteradores são "begin()" e "end()" (ambos métodos de "v"). Pode-se pensar nos iteradores como uma analogia aos ponteiros, de qualquer forma, ao utilizar sort em um vector sempre serão estes os iteradores.

Quando tivermos um vector no qual os elementos são estruturas de dados, ou quando desejarmos ordenar seguindo outros critérios utilizaremos a segunda forma da função sort. Seu uso é mostrado no exemplo a seguir:

Listing 3.15: sort com critério próprio

```
1  #include <iostream>
2  #include <algorithm>
3  #include <vector>
4  using namespace std;
5
6  typedef pair<int, int> ii;
7
8  bool cmp(ii a, ii b )
9  {
10       int x,y;
11       x = a.first * a.second;
12       y = b.first * b.second;
13       return (x < y);
14 }
15
16 int main()
17 {
18   vector<ii> v;
19   ii x;
20
21   while(cin >> x.first >> x.second)
22     v.push_back(x);
23
24   sort(v.begin(), v.end(), cmp);
```

```
25
26    for(int i = 0; i < v.size(); i++)
27      cout << v[i].first << " " << v[i].second << "\n";
28
29    return 0;
30  }
```

Neste exemplo utilizamos a função “cmp” como critério de comparação, esta função retorna verdadeiro quando o produto para o elemento “a” é menor que do elemento “b”. Na prática, a função de comparação deve retornar verdadeiro caso o elemento “a” deva proceder o elemento “b” (após a ordenação).

stable_sort

Tal como a sort, esta rotina permite a ordenação, entretanto a diferença é que stable_sort mantém a ordem relativa entre dois termos caso eles sejam iguais (do ponto de vista da ordenação corrente).

Isso é especialmente útil quando a entrada já está ordenada segundo algum critério, ou quando realizamos mais de uma ordenação sobre o mesmo conjunto de dados.

A forma de uso do stable_sort é a mesma que sort.

Capítulo 4

Problemas Ad-Hoc

Este capítulo trata de dois tipos de problemas comuns na maratona: AdHoc e Simulação. Ambos não requerem nenhum conhecimento específico das áreas de estrutura de dados ou de algoritmos avançados. Por terem esta característica são os tópicos mais adequados para quem está começando na Maratona de Programação.

Ao final de cada capítulo estão disponíveis os enunciados dos problemas citados.

4.1 AdHoc

É uma classe de problemas que não se encaixa em nenhuma área específica da computação ou análise de algoritmos. Para resolvê-los geralmente são necessários apenas conhecimentos básicos de matemática e, talvez, de estrutura de dados, pois o enfoque reside em encontrar a solução e não em como resolver. Ou seja, geralmente a implementação de problemas AdHoc é trivial. Cada problema Ad-Hoc é único, logo não existe nenhuma fórmula ou técnica em específico para ajudá-lo a resolvê-los.

4.1.1 Exemplo: Odd or Even(Final 2006)

O problema Odd or Even (http://br.spoj.pl/problems/ODDOREVE/) é um bom exemplo de um problema Ad-Hoc: não é necessário nenhum conhecimento avançado de algoritmos nem estrutura de dados, apenas saber o que são números pares e ímpares. A solução exige apenas um conhecimento: saber que todo numero par dividido por 2 retorna como resto 0. Basta, então, somar o número de dedos em cada mão e utilizar o operador que retorna o resto (%) de uma divisão por 2 para averiguar se é par ou ímpar. Vale ressaltar que a solução proposta para este problema é específica, não há uma classe de problemas que possam se beneficiar deste algoritmo.

Segue, abaixo, a solução proposta.

Obs.: foi utilizado a função "abs()" da biblioteca **ctdlib** que retorna o valor absoluto do argumento.

Listing 4.1: Solução de Odd or Even

```
#include<iostream>
#include<cstdlib>

using namespace std;

int main() {
    int N, V;
    int joao, maria;

    while(cin >> N && N) {
        joao = maria = 0;
        for (int i=0; i<N; i++) {
            cin >> V;
            if (!(V%2))
                maria++;
        }
        for (int i=0; i<N; i++) {
            cin >> V;
            if (V%2)
                joao++;
        }
        cout << abs(joao - maria) << endl;
    }
    return 0;
}
```

4.1.2 Teste: Copa do Mundo(Regional 2008)

O problema Copa do Mundo (http://br.spoj.pl/problems/COPA/) é um ótimo exemplo de problema **real** da maratona.

Dicas

- A distribuição de pontos nos informa algo?
- É possível gerar uma equação para resolver o problema?

4.2 Problemas de Automatos/Ad-Hoc

Outro problema recorrente nas competiçõees é o processamento da entrada através de autômatos. Um autômato funciona como o reconhecedor de uma linguagem e pode para modelar máquinas simples. Basicamente um autômato possui um estado, e muda esse estado (para outro) dependendo da entrada que recebe. O autômato é uma máquina de estados finitos.

4.2.1 Exemplo de Automato Simples - Loop Musical(Regional 2008)

O problema Loop Musical (http://br.spoj.pl/problems/LOOPMUSI/) apresenta uma estrutura que é fácilmente atacada com a utilização de autômatos. Podemos distinguir dois estados possíveis para a curva:

- Crescente
- Decrescente

E podemos ter dois estimulos em cada estágio:

- Valor acima da curva
- Valor abaixo da curva

Temos assim 2 estados com duas possíveis mudanças de estados cada. Ao analisarmos o problema, podemos perceber fácilmente que o problema quer que contemos o número de vezes que estamos no estado <crescente> e recebemos um estímulo <Valor abaixo da curva>, isto é, contar o número de curvas crescentes.

O código abaixo mostra uma possível solução para esse problema, nesta solução lemos toda a entrada para um vetor, mas é possível fazer o processamento sem armazenar valores intermediários (apenas o valor inicial, e 3 valores denotando o ponto que estamos analisando).

Listing 4.2: Solução de Loop musical

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
    int n;
    vector<int> v;

    while(cin >> n) {
            if(n == 0)
                break;
            v.clear();
            int t;
            for(int i = 0; i < n; i++) {
                    cin >> t;
                    v.push_back(t);
            }
                        #define proximo(x) ((x + 1) % n)
                        #define anterior(x) ((x + n - 1) % n)
            int loop = 0;
            for(int i = 0; i < v.size(); i++)
```

```
22                                                  if(((v[anterior(i)] < v[i])
                                                       && (v[proximo(i)] < v[i])
                                                       ) ||
23                                                     ((v[anterior(i)] > v[i])
                                                          && (v[proximo(i)] > v[
                                                          i])))
24                                                              loop++;
25
26                  cout << loop << endl;
27      }
28      return 0;
29  }
```

4.2.2 Teste: Problema do Jingle Composing(Final 2009)

O problema Jingle Composing (http://acm.uva.es/archive/nuevoportal/region.php?r=sa\&year=2009) é estritamente um autômato, consiste em tratar a entrada e para cada valor possível tomar a ação correspondente. Caso tenha dificuldades, são apresentadas dicas no anexo **??**.

4.3 Problemas de Simulação/Ad-Hoc

Pode-se enxergar problemas de simulação como uma sub-classe dos Ad-Hoc cuja resposta não pode (ou a dificuldade é deveras alta) ser moldada numa fórmula ou equação. Em geral os problemas de simulação se caracterizam por uma configuração inicial, uma série de regras que ditam como o sistema descrito poderá se modificar a cada iteração, e uma condição de parada, sendo que esta pode ser pré-determinada ou não.

Uma das grandes dificuldades na Maratona de Programação é definir quando um problema deve ser resolvido por simulação ou não. Isto deve-se ao fato de que é comum um mesmo problema ter solução tanto por simulação quanto por outro método, porém a solução por simulação geralmente é computacionalmente mais onerosa e, em muitos casos, gera uma resposta com "time limit exceeded".

Para conseguir identificar se é um problema à ser solucionado por simulação requer um pouco de experiência, conhecimento sobre técnicas computacionais e, análise das possíveis configurações de entrada de dados e, em ambientes de competição, observação do comportamento dos times adversários quanto à este problema. Se ele foi resolvido rapidamente e sem respostas erradas, muito provavelmente será um problema de simulação.

4.3.1 Perigos da simulação não finita

Um problema de simulação sempre deve ter uma condição de parada bem definida para impedi-lo de executar indefinidamente. Esta condição pode ser o número de iterações previamente conhecido ou uma configuração final específica.

Há outros casos, porém, que o próprio problema pede que se identifiquem casos em que a simulação não tenha fim e retorne, então, uma saída apropriada. Geralmente nestes casos a simulação entra num loop, ou seja, tendo um ponto de partida o sistema seguirá um caminho, voltará ao ponto inicial e seguirá novamente pelo mesmo caminho. Para identificar tais loops é necessário que, a cada passo da simulação, guarde-se numa estrutura de dados o seu estado e, a cada nova iteração, verifique se o sistema já não assumiu nenhuma das configurações anteriores; caso positivo, ele entrou num loop.

4.3.2 Exemplo: 3n + 1

Foi o primeiro problema da maratona de programação para muitos. Modelar o problema é relativamente simples. Sua resolução pode ser tanto recursiva quanto iterativa, porém a última, sempre que possível, a última deve ser escolhida devido sua maior performance e, consequentemente, menor risco de levar um timeout. O problema pode ser visto em http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=3&page=show_problem&problem=36

Listing 4.3: Solução de 3n+1

```
#include <iostream>

using namespace std;

int main() {
        unsigned int i, j, maior, n, contador, aux;
        bool trocou = false;
        while(cin >> i >> j) {
                maior = trocou = 0;
                if(i<j) {
                        aux = j;
                        j = i;
                        i = j;
                        trocou = true;
                }
                for(int index=i; index<=j; index++) {
                                n = index; contador = 1;
                                while(n != 1) {
                                        if(n%2 == 0) n/=2;
                                        else n = 3*n+1;
                                        contador++;
                                }
                                if(contador > maior) maior = contador;
                }
                if(trocou)
                        cout << j << " " << i << " " << maior << endl;
                else
                        cout << i << " " << j << " " << maior << endl;
        }

```

```
31          return 1;
32  }
```

Este problema possui duas características que a maioria dos iniciantes erram: assumir que o valor de i sempre é menor que o de j e não utilizar **unsigned int** ou **long int**.

Esta solução funciona, mas é um bom ponto de partida para aprimorarmos sua performance. Como podemos ver, para todo número M existe apenas uma solução. Tendo resolvido ela, qualquer outro problema que possua M como sub-problema poderá aproveitar sua solução previamente calculada, evitando uma série de operações computacionalmente exaustivas e desnecessárias.

Então, para evitarmos inúmeras operações repetidas e desnecessárias, emos que guardar as soluções parciais numa estrutura de dados para, posteriormente, utilizá-las quando necessário. Existem várias soluções possíveis para tal. Uma solução prática e eficiente é armazenar os valores calculados num vetor. Esta estrutura possui um tempo de atualização e consulta constantes, O(1).

Esta técnica de armazenar valores calculados para serem reaproveitados se chama Memorization e será abordada no Capítulo X.

4.3.3 Teste: Brothers (Final 2009)

Este exemplo (http://acm.uva.es/archive/nuevoportal/region.php?r=sa\&year=2009) mostra uma questão real de simulação na maratona

Dicas

- É conhecido o ponto de parada da simulação?
- Quantos laços são necessários para atualizar o estado do tabuleiro a cada iteração?

Capítulo 5

Estruturas de dados e Ordenação

O uso de estruturas de dados permite modelar uma grande classe de problemas mais facilmente. Seu correto uso permite uma abstração maior dos detalhes técnicos do problema e da solução, podendo o programador se focar mais na **solução** do que no **como**.

Já as técnicas de ordenação também são de grande utilidade numa vasta gama de problemas computacionais. Muitas soluções algoritmicas são modeladas em cima de dados ordenadas.

Este capítulo aborda brevemente, então, estes dois assuntos com um enfoque do seu uso na Maratona de Programaçao.

5.1 Pilha

A pilha, **stack** na STL do C++, consiste numa estrutura de dados do tipo LIFO (Last Int, First Out), ou seja, "Último a Entrar, Primeiro a Sair". Por exemplo, podemos fazer uma analogia com uma pilha de livros: o livro mais recentemente adicionado (**push**) vai para o topo da pilha. Quando eu quiser acessá-lo novamente eu tenho que retirar um a um (**top**) todos os livros adicionados após ele, até encontrá-lo.

Esta estrutura é muito corriqueira no dia-a-dia e também o é na computação. Varias soluções podem ser modeladas com o uso de pilhas, como calculadoras com notação pós-fixada (ex.: 5 2 +), buscas em profundidade conforme será visto no capítulo 8 (**Grafos**), etc.

5.1.1 Exemplo: Calculadora Pós-Fixada

Uma calculadora pós-fixada caracteriza-se por tratar expressões que possuem a seguinte forma: **operando operando operador**. A calculadora deve, então, ler os parâmetros de entrada e os adicionar na pilha **P**. Sempre que encontrar um operador ela deverá desempilhar de **P** dois argumentos e efetuara operação. Em seguida deverá empilhar este resultado e continuar a ler a equação. No final, haverá apenas um único valor na pilha correspondente à resposta.

Obs.: é utilizado a função "atoi()" da biblioteca **cstdlib** que recebe como argumento uma C string (vetor de char's) e retorna o número correspondente.

Listing 5.1: Exemplo: Calculadora Pós-Fixada

```
#include <stack>
#include <string>
#include <iostream>
#include <cstdlib>

using namespace std;

stack<int> P;

int calcula() {
        string s;
        int arg1, arg2, resultado;
        while(cin >> s) {
                if(s[0] != '+' && s[0] != '-' && s[0] != '*' && s[0] != '/')
                        P.push(atoi(s.c_str()));
                else {
                        arg1 = P.top(); P.pop();
                        arg2 = P.top(); P.pop();
                        if(s[0] == '+') resultado = arg1+arg2;
                        else if(s[0] == '-') resultado = arg1-arg2;
                        else if(s[0] == '*') resultado = arg1*arg2;
                        else if(s[0] == '/') resultado = arg1/arg2;
                        P.push(resultado);
                }
        }
        return P.top();
}

int main() {
        cout << calcula();
        return 1;
}
```

Existe algum erro neste código? Mesmo supondo que todas as expressões de entrada estejam formatadas corretamente, ele funciona para todos os casos?

5.2 Fila

Fila, conhecida como **queue** na STL do C++, é uma estrutura de dados do tipo FIFO (First In, First Out), "Primeiro a Entrar, Primeiro a Sair". Para compreender mais facilmente pode-se fazer uma analogia com uma fila de super-mercado: o primeiro à entrar na fila será o primeiro atendido. Se eu for a terceira pessoa a entrar na fila, independentemente de quantos outros também entrarem depois de mim, eu serei o terceiro cliente a ser atendido.

Em computação a estrutura FIFO é utilizado para armazenar buffers, lidar com comunicações, e, mais específico na maratona, auxiliar na criação de buscas em largura (conforme será visto no capítulo 8), para guardar um caminho numa busca, entre outros.

5.2.1 Exemplo: Guardar Caminho numa Busca

Por ser uma estrutura de dados do tipo FIFO, é possível recuperar um caminho percorrido numa busca ou qualquer outro procedimento através de uma fila. O pseudo-código a seguir exemplifica sua utilização:

Listing 5.2: Exemplo: Calculadora Pós-Fixada

```
#include <queue>
#include <iostream>

using namespace std;

queue<int> fila;

int funcao_de_busca(int inicio, int final) {
        while( ... ) {
                if(ACHEI_PROXIMO_NODO)
                        fila.push(proximo_nodo);
        }
}

int main() {
        funcao_de_busca(inicio, final);
        for(i=0; i<fila.size(); i++) { // imprime caminho do inicio ao final
                cout << fila.front() << " -> ";
                fila.pop();
        }
        return 1;
}
```

5.3 Algoritmos de ordenação

Muitas vezes, ter os dados ordenados facilita a construção de uma solução. Em outras, diminui em muito o seu custo computacional, podendo sua complexidade variar de $O(N!)$ para $O(N^2)$.

Sua utilidade será clara no problema a seguir.

5.3.1 Exemplo: Dinner Hall (final 2009)

Listing 5.3: Exemplo: Calculadora Pós-Fixada

```
#include <vector>
```

```
2  #include <iostream>
3  #include <algorithm>
4  #include <cstdlib>
5
6  using namespace std;
7
8  typedef struct cartao {
9          int h, m, s;
10         char estado;
11 } TpCartao;
12
13 bool cmp(TpCartao a, TpCartao b) {
14         if(a.h < b.h) return true;
15         if(a.h != b.h) return false;
16
17         if(a.m < b.m) return true;
18         if(a.m != b.m) return false;
19
20         if(a.s < b.s) return true;
21         return false;
22 }
23
24 int main() {
25         int n, contador, max, numeroE, numeroX, maxIn, cartoesLidosU;
26         vector<TpCartao> cartoes;
27         TpCartao caux;
28         cin >> n;
29         while(n!=0) {
30                 cartoes.clear();
31                 numeroE = numeroX = cartoesLidosU = max = contador = 0;
32                 for(int i=0; i<n; i++) {
33                         scanf("%d:%d:%d %c", &caux.h, &caux.m, &caux.s, &
                               caux.estado);
34                         cartoes.push_back(caux);
35                         if(caux.estado == 'X') numeroX++;
36                         else if(caux.estado == 'E') numeroE++;
37                 }
38                 sort(cartoes.begin(), cartoes.end(), cmp);
39                 if(numeroE > numeroX) maxIn = (cartoes.size()-(numeroE+
                       numeroX))/2;
40                 else maxIn = (cartoes.size()-(numeroE+numeroX))/2 + (numeroX
                       - numeroE);
41
42                 for(int i=0; i<cartoes.size(); i++) {
43                         if(cartoes[i].estado == 'E') {  // cartao de entrada
44                                 contador++;
45                         } else if(cartoes[i].estado == '?'){
46                                 if(cartoesLidosU < maxIn) // ainda pode-se
                                       considerar como entrada
47                                         contador++;
48                                 else
```

```
49                                          contador--;
50                                  cartoesLidosU++;
51                              } else { // cartao de saida
52                                  contador--;
53                              }
54                              if(max < contador) max = contador;
55                      }
56                      cout << max << endl;
57                      cin >> n;
58              }
59              return 0;
60      }
```

A resolução do problema consiste em entender que a soma de pessoas que entrara com as que sairam deve ser igual no final. Assim, contando o número de cartões, é possível sabemos quantas pessoas entraram no total. Para resolvê-lo mais facilmente podemos ordenar os cartões pelos horários e então, lendo-os do primeiro ao ultimo, acrescentando 1 à um determinado contador quando lê-se um cartão de uma pessoa que está entrando e retirando 1 quando é um cartão que uma pessoa sai.

Quando o cartão é incerto nós podemos calcular que n são cartões de entrada e m são cartões de saída. Assumimos, então, que até lermos n cartões todos os cartões incertos é de entrada. Posteriormente desse valor, que é de saída.

Bolar uma solução que não involvesse a ordenação dos cartões seria muito mais difícil, e o custo computacional da solução certamente seria bem maior.

Capítulo 6

Grafos

Neste capítulo trataremos de grafos, eles são uma forma de representação de dados comum em diversos problemas de computação. Grafos são geralmente representados por pontos (vértices) ligados por linhas ou setas (arestas), podemos por exemplo representar cidades como sendo vértices de um grafo e as rodovias que ligam essas cidades serão as arestas dele. Após modelado o problema, podemos utilizar algoritmos tradicionais da teoria dos grafos para determinar, por exemplo, a menor distância entre duas cidades.

Dessa forma, devemos desenvolver principalmente duas habilidades para sermos bem sucessidos em problemas envolvendo grafos: ter capacidade de modelar os problemas na forma de grafos; e, conhecer os algoritmos clássicos que operam sobre grafos. Neste capítulo abordaremos alguns dos principais algoritmos utilizados em teoria dos grafos, e esperamos que o leitor desenvolva habilidade em modelar soluções envolvendo grafos através da resolução dos problemas propostos.

Grafos podem ser direcionados ou não, a figura 6.1 mostra um grafo não direcionado. Nos grafos não direcionados, as arestas "permitem"uma transição em qualquer um dos sentidos.

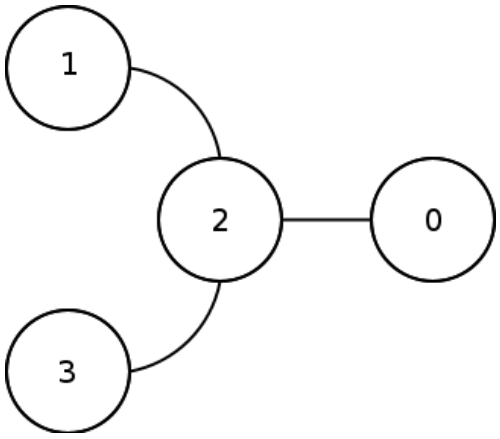

Figura 6.1: Grafo não Dirigido

Quando temos grafos direcionados (também conhecidos como digrafos), as arestas permitem apenas a transição no sentido indicado pela mesma. A figura 6.2 mostra um grafo dirigido.

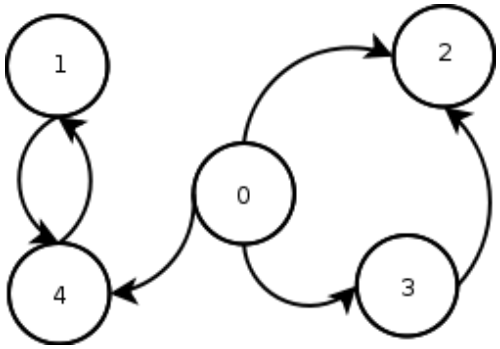

Figura 6.2: Grafo Dirigido (Digrafo)

Os grafos podem ainda ser ponderados (ou valorados), isto é, as arestas podem possuir pesos; nesse caso arestas com pesos maiores geralmente representam transições mais custosas. Na figura 6.3 é mostrado um grafo não dirigido ponderado, note-se que o caminho de 1-4, tem peso diferente de 4-1.

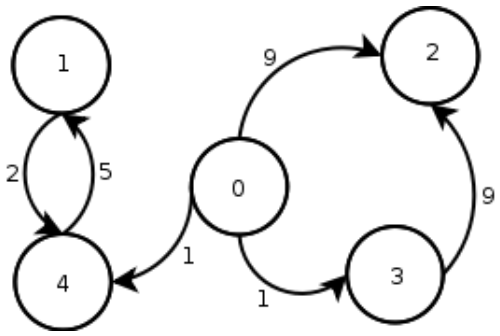

Figura 6.3: Grafo Dirigido Ponderado
s

Sempre iniciaremos determinando qual o grafo mais apropriado para o problemam, disso surge outra questão: “que estrutura de dados utilizaremos para representar grafos?” - este é o assunto da seção seguinte.

6.1 Representação de grafos

Predominantemtente são utilizadas duas abordagens para representar grafos: matrizes de adjacência e listas de adjacência. Aqui utilizaremos listas de adjacência por, em geral, apresentarem maior desempenho. Digamos que nosso grafo possui n elementos, algumas operações sobre matrizes de adjacência podem necessitar a análise de n^2 elementos (todas

as ligações) enquanto com listas de adjacência verificaremos apenas o número de vezes igual o número de arestas do grafo (que é no máximo n^2)[1].

Utilizaremos "vetores de vetores de inteiros" para representar os grafos, caso sejam grafos ponderados será utilizada pares de inteiros "pair<int, int>". Desejamos representar o grafo da figura 6.4.

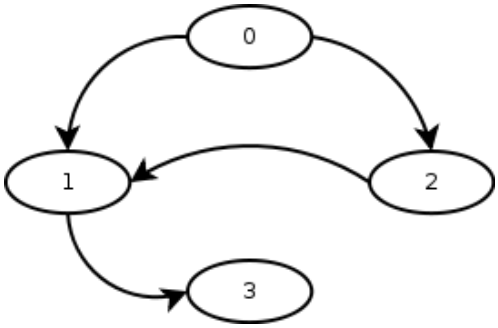

Figura 6.4: Exemplo de grafo

A declaração do grafo (não ponderado) será:

```
vector< vector<int> > grafo;
```

Cada posição do vetor representa um vértice, nela está contido um vetor - a lista de adjacências. Essa lista possui as arestas que partem desse vértice - no caso de um grafo não dirigido adicionamos ambas arestas.

Os dados contidos em cada uma das posições do vetor será:

```
grafo[0] = {1, 2}
grafo[1] = {3}
grafo[2] = {1}
grafo[3] = {}
```

Essa estrutura de dados responde (de forma eficiente) apenas à pergunta: "quais arestas partem do vértice X?". Se desejarmos também saber "quais arestas chegam no vértice X?" é necessário utilizar matrizes de adjacência ou criar um grafo auxiliar.

6.2 Exemplo de leitura de um grafo

No exemplo abaixo lemos um grafo e posteriormente mostramos as ligações de cada vértice.

A primeira linha da entrada possui dois inteiros N e X que representa o número de vértices no grafo e o número de arestas. É seguida por X linhas contendo inteiros A e B, representando uma aresta (direcionada) de A para B. O exemplo anterior teria o formato:

[1] seja n o número de vértices e m o número de arestas, a varedura na representação por listas toma tempo $O(n + m)$

```
4 4
0 1
0 2
1 3
2 1
```

O programa abaixo lê um grafo nesse formato e depois percorre todo o grafo mostrando as arestas.

Listing 6.1: Leitura de um grafo

```
1  #include <vector>
2  #include <iostream>
3  using namespace std;
4
5  int main()
6  {
7    vector< vector<int> > g;
8    int a, b, n, x;
9
10   cin >> n >> x;
11
12   g.resize(n);
13
14   for(int i = 0; i < x; i++) {
15     cin >> a >> b;
16     g[a].push_back(b);
17   }
18
19   for(int i = 0; i < g.size(); i++) {
20     cout << i << ":";
21     for(int j = 0; j < g[i].size(); j++)
22       cout << " " << g[i][j];
23     cout << "\n";
24   }
25
26   return 0;
27 }
```

Na linha 10 obtemos o número de vértices e de arestas, eis que falta apenas a definição destas arestas para que tenhamos o grafo completamente especificado.

Na linha 12 adequamos o tamanho do vetor ao número de vértices do grafo.

Utilizamos a repetição da linha 14 para fazer a leitura das arestas, adicionamos o valor de “b” na posição “a” com sentido de que existe uma aresta que parte de “a” e vai até “b”.

Na linha 19, verificamos cada um dos “g.size()” vértices, e para cada vértice “i”, verificamos suas “g[i].size()” arestas (observe-se que g[i][j] não representa uma aresta de “i” para “j”, mas a j-ésima aresta do vértice i)

6.3 Busca em largura

Uma interpretação muito importante acerca de grafos não ponderados, é de que na realidade as arestas representam todas o mesmo peso. Além de uma "ligação" ou um "caminho" entre os vértices, eles podem representar um "passo", uma unidade de distância, um estágio de transformação, etc. entre os vértices.

Esta seção trata da busca de um caminho em um grafo, em especial, uma busca onde garantimos que os vértices visitados primeiramente terão suas arestas examinadas anteriormente a outros. Eis que, partindo de um vértice "a" qualquer e garantindo essa restrição, na primeira vez que atingirmos o vértice "b" podemos **afirmar** que este caminho possui o número mínimo de passos até tal, ou seja, que este é o menor caminho entre "a" e "b". Isso funciona pois os vértices visitados mais cedo estão mais próximos do vértice inicial, e consequêntemente tem distância menor até a origem.

Eis que para garantir esta restrição, basta que utilizemos uma fila para armazenar os vértices visitados. O exemplo a seguir mostra uma rotina para busca em largura utilizando listas de adjacência. MAX representa o valor máximo possível de vértices, geralmente especificado no problema.

Listing 6.2: Busca em largura

```
#define MAX 100

int visitados[MAX], sucessor[MAX], distancia[MAX];
vector< vector<int> > g;

void bfs(int inicio, int final)
{
    queue<int> f;

    for(int i = 0; i < MAX; i++)
      visitados[i] = false;

    distancia[inicio] = 0;
    f.push(inicio);
    while(!f.empty()) {
      int a = f.front();
      f.pop();

      for(int i = 0; i < g[a].size(); i++)
        if(!visitados[g[a][i]]) {
          visitados[g[a][i]] = true;
          sucessor[g[a][i]] = a;
          distancia[g[a][i]] = distancia[a] + 1;
          f.push(g[a][i]);
          if(g[a][i] == final)
            return;
        }
    }
}
```

Neste exemplo salvamos todas informações relevantes: tracking para evitar repetição (e ciclos), o caminho e as distâncias (linha 3). Em muitos casos precisaremos apenas saber a distância, ou mesmo se existe ou não um caminho. Na linha 10 inicializamos o vetor para evitar analizar o mesmo vértice duas vezes, embora MAX seja geralmente maior que "g.size()" não é um grande trabalho extra. Também adicionamos o vértice inicial na fila, e determinamos o peso inicial como 0 (linhas 13 e 14);

A busca ocorre na repetição da linha 15, dado o vértice "a" que é retirado da fila (que foi visitado a mais tempo), verificamos cada uma de suas arestas (não visitadas) e atualizamos a distância e o sucessor das mesmas, pois, "a" é o menor caminho entre o inicio e as arestas em questão. Caso encontremos o vértice final paramos a busca pois já atingimos o objetivo.

É possível que não exista um caminho entre "inicio" e "final", caso exista, ao final da busca o vértice final tem que ter sido visitado ("visitado[final] == true"). Se quisermos saber a distância total, basta obtermos o valor "distancia[final]", se desejamos saber o caminho podemos utilizar uma pilha para reconstruí-lo a partir das informações no vetor "sucessor", sendo "a" o vértice inicial e "b" o final, teremos algo como no exemplo a seguir.

Listing 6.3: Reconstruir caminho

```
1 stack<int> s;
2 int i = b;
3
4 while(i != a) {
5   s.push(i);
6   i = sucessor[i];
7 }
8 s.push(a);
```

Neste exemplo, buscamos os sucessores a partir do nó final, até atingir o nó inicial.

6.3.1 Exemplo: Duende

Como dito anteriormente, podemos com a busca em largura encontrar o menor caminho em um grafo. O problema "Duende Perdido" (http://br.spoj.pl/problems/DUENDE/) trata da busca do menor caminho em um espaço bidimensional, como nunca existe a necessidade do duente "voltar" podemos modelar esse problema como um grafo onde a posição inicial do duende é um vértice do qual apenas partem vértices.

Uma vez que vizualizemos a construção desse problema como um grafo, podemos entender que a busca em largura soluciona esse problema[2] e que na realidade não precisamos construir o grafo para resolver o problema, mas sim aplicar o algoritmo descrito neste cenário.

De posse dessas informações podemos adotar uma estratégia simples: partindo da posição inicial do duende, analisamos os vizinhos (que não tem parede de cristal) até

[2]Por tratar-se de uma entrada de pequena dimensão, esse problema pode ser resolvido de diversas formas, entretanto uma busca em largura é o mais adequado

encontrar a saida. Utilizaremos uma fila para guardar a ordem que devemos analisar as posições e isso também garante que a primeira vez que encontrarmos uma saída, esta é a mais próxima do duente.

O exemplo abaixo pode ser utilizado para solucionar este problema.

Listing 6.4: Duende Perdido

```
#include <queue>
#include <iostream>
using namespace std;

typedef pair<int, int> ii;
#define MAX 10
int g[MAX][MAX];
int distancia[MAX][MAX], visitados[MAX][MAX];

int main()
{
  int n,m;
  ii ci;

  cin >> n >> m;
  for(int i = 0; i < n; i++)
    for(int j = 0; j < m; j++) {
      cin >> g[i][j];
      if(g[i][j] == 3)
        ci = ii(i, j);
    }

  for(int i = 0; i < MAX; i++)
    for(int j = 0; j < MAX; j++)
      visitados[i][j] = false;

  queue<ii> q;
  distancia[ci.first][ci.second] = 0;
  visitados[ci.first][ci.second] = true;
  q.push(ci);
  int d;
  while(!q.empty()) {
    ii a = q.front(), nt[4];
    q.pop();

    if(g[a.first][a.second] == 0) {
      d = distancia[a.first][a.second];
      break;
    }

    nt[0] = ii(a.first + 1, a.second);
    nt[1] = ii(a.first, a.second + 1);
    nt[2] = ii(a.first - 1, a.second);
    nt[3] = ii(a.first, a.second - 1);

```

```
46      for(int i = 0; i < 4; i++)
47        if(((nt[i].first < n) && (nt[i].first >= 0)) &&
48          ((nt[i].second < m) && (nt[i].second >= 0))) {
49              if((!visitados[nt[i].first][nt[i].second]) &&
50                 (g[nt[i].first][nt[i].second] != 2)) {
51                    visitados[nt[i].first][nt[i].second] = true;
52                    distancia[nt[i].first][nt[i].second] = distancia[a.first][
                          a.second] + 1;
53                    q.push(ii(nt[i].first,nt[i].second));
54              }
55        }
56    }
57
58    cout << d << endl;
59    return 0;
60  }
```

Nas linhas 15-18 lemos a configuração dos salões, a linha 19 economiza o esforço de posteriormente realizar uma busca apenas para encontrar a posição inicial do duende. Na 23-25 setamos todas as posições como não visitadas.

A diferença da busca (linhas 32-56) é que armazenamos um par na fila, correspondendo a posição (i, j) do salão. Diferentemente de um grafo onde pode-se ter um número qualquer de arestas, nesse caso teremos no máximo 4 possibilidades de próxima posição, assim criamos as 4 possibilidades (41-44) e as validamos (47-48). Para ser uma possível próxima solução, além de não ter sido visitado (como nos casos anteriores) não pode ser um salão com paredes de cristal (valor 2), estes testes são feitos nas linhas 49 e 50. As linhas 51 a 53 fazem o papel de memorizar as distâncias e marcar as posições como visitadas, além de adicionar as mesmas na fila.

Podem existir diversas saidas, dessa forma, apenas testamos no inicio do laço se estamos em uma posição final e interrompemos em caso afirmativo. Como trata-se de uma busca em largura, a primeira posição final encontrata é a de menor distância.

Na linha 58 mostramos a distância entre o duende e a saída mais próxima, que é a saída esperada pelo problema.

6.3.2 Teste: Playing with Wheels

Sugerimos a resolução do problema Playing with Wheels (http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=37&page=show_problem&problem=1008) como exercício.

6.4 Busca em profundidade

Outra estratégia possível para realizar uma busca, é visitar as arestas do vértice mais recentemente visitado, esta estratégia é especialmente eficiênte quando desejamos apenas

verificar a existência de um caminho entre dois vértices (ou quando por definição existe apenas um). Esta busca é a busca em profundidade.

A busca em profundidade possui ainda diversas aplicações, das quais podemos citar busca de ciclos, bipartição de grafos. Embora sua simplicidade, o entendimento da busca em profundidade é necessário para compreensão de diversos algoritmos úteis.

Como dito anteriormente, verificamos primeiro os vértices atingidos mais recentemente, algoritmicamente falando, utilizamos uma pilha para guardar a ordem de verificação. A dinâmica é a mesma da busca em largura, no exemplo abaixo testamos apenas se existe um caminho entre um nó inicial e final.

Listing 6.5: Busca em Profundidade

```
#define MAX 100
int visitados[MAX];
vector< vector<int> > g;

bool dfs(int inicio, int final)
{
    stack<int> s;

    for(int i = 0; i < MAX; i++)
      visitados[i] = false;

    s.push(inicio);
    while(!s.empty()) {
      int a = s.top();
      s.pop();
      for(int i = 0; i < g[a].size(); i++)
        if(!visitados[g[a][i]]) {
          visitados[g[a][i]] = true;
          s.push(g[a][i]);
          if(g[a][i] == final)
            return true;
        }
    }

    return false;
}
```

6.4.1 Teste: Bicoloring

Para está seção sugerimos o problema Bicoloring (http://uva.onlinejudge.org/index.php?option=com_onlinejudge&Itemid=8&category=37&page=show_problem&problem=945).

6.5 Caminhos Mínimos em grafos ponderadoss

Esta seção trata do problema de encontrar o caminho mínimo entre dois pontos num grafo ponderado, isto é, dado um grafo com arestas valoradas descobrir qual o caminho de um vértice A para outro B que possui o menor custo total (valor somado).

6.5.1 Algoritmo de Dijkstra

Nesta seção apresentamos o algoritmo proposto por Dijkstra para encontrar o menor caminho em um grafo ponderado de **pesos positivos** (Se o grafo possui também pesos negativos é necessário utilizar o algoritmo de Bellman-Ford).

Este algoritmo é semelhante a busca em largura, entretanto, aqui sempre iremos expandir o nó que atualmente tem menor custo, isto é, tentaremos atingir o nó final sempre pelo caminho que possui menor custo total e para isso, sempre devemos escolher o menor para cada nó intermediário. A primeira vez que atingimos um vértice não é o menor caminho até este.

Para memorizar quais os caminhos de menor custo, utilizaremos uma fila de prioridade (http://www.cppreference.com/wiki/stl/priority_queue/start), será necessário ainda armazerar o menor custo para cada vértice (objetivo do algoritmo).

A seguir apresentamos uma implementação do algoritmo de Dijkstra.

Listing 6.6: Algoritmo de Dijkstra

```
1  vector< ii > adj[NMAX];
2  int N, D[NMAX], pi[NMAX];
3
4  void dijkstra(int s) {
5    for(int i = 0; i < N; i++) {
6      D[i] = INF;
7      pi[i] = -1;
8    }
9    priority_queue<ii,vector<ii>, greater<ii> > Q;
10   D[s] = 0;
11   Q.push(ii(0,s));
12
13   while(!Q.empty()) {
14     ii top = Q.top();
15     Q.pop();
16
17     int u = top.second, d = top.first;
18
19     if (d<=D[u])
20       for(int i = 0; i < adj[u].size(); i++) {
21         int v = adj[u][i].first, cost = adj[u][i].second;
22         if ( D[v] > D[u] + cost ) {
23           D[v] = D[u] + cost;
24           pi[v] = u;
25           Q.push( ii( D[v], v ) );
26         }
```

```
27          }
28      }
29  }
```

Na linha 1 declaramos nossa lista de adjacências, o primeiro elemento do pair representa a adjacência e o segundo o custo até esse vértice. Na linha 2 temos N - o número de vértices no caso atual, D - a menor distância até cada um dos vértices, e pi - o antecessores de cada vértice. NMAX representa o valor máximo possível de vértices. O objetivo deste código é preencher D e pi.

Na linha 5 inicializamos o vetor com as menores distâncias, atribuimos como "infinito"[3] para identificar que este vértice não foi visitado ainda, também inicializamos os sucessores.

A busca acontece na repetição da linha 13. Avaliamos o vértice que possui a menor distância até o vértice inicial, e verificamos cada uma das suas adjacências (linha 20). O teste mais característico do algoritmo é encontrado na linha 22 ("if (D[v] ¿ D[u] + cost)") e deve ser lido como: "Se o menor caminho até v (partindo de s), é o que passa por u, então atualize o menor caminho até v".

Eis que, se visitarmos todos os caminhos que chegam num dado vértice e memorizarmos o de menor custo, podemos afirmar que esse é o menor custo entre a origem e o vértice em questão. Esse algoritmo faz isso para todos vértices, partido de "s"; Uma vez executado a partir de "s", a menor distância de "s"até qualquer vértice "v", será aquela armazenada em "D[v]".

6.5.2 Floyd-Warshall

O algoritmo apresentado na seção anterior encontra o menor caminho entre dois vértices. Em alguns casos é necessário determinar o menor caminho entre todos os pares de vértices do grafo, nesse caso utiliza-se o algoritmo Floyd-Warshall. Mais informações deste algoritmo podem ser encontradas em http://pt.wikipedia.org/wiki/Algoritmo_de_Floyd-Warshall.

6.5.3 Teste: Quase menor caminho

O problema "Quase menor caminho"(http://br.spoj.pl/problems/QUASEMEN/) da final brasileira de 2008 apresenta um desafio interessante relacionado ao problema de encontrar o menor percurso. Sua resolução é importante para aqueles que desejam testar seus conhecimentos nessa área.

6.6 Minimun Spanning Tree (Árvore Geradora Mínima)

Esta seção trata de árvores geradoras em um grafo não dirigido. Árvore geradora é qualquer sub árvore ("v"vértices, "v - 1"arestas e conexo) que contenha todos os vértices do grafo.

[3] Geralmente utiliza-se o valor 0x3F3F3F3F para designar infinito, porém alguns problemas podem ter limites maiores

Quanto realizarmos uma busca em profundidade ou em largura, estamos criando uma árvore geradora.

Se considerarmos um grafo ponderado, o custo de uma sub árvore é a soma dos valores de suas arestas. Eis que o objetivo desta seção é determinar a árvore geradora com o menor custo total, nominada árvore geradora mínima (de agora em diante MST - Minimun Spanning Tree).

6.6.1 Algoritmo de Prim

O algoritmo de Prim encontra a árvore geradora mínima para um dado grafo, seu funcionamento é baseado nas condições de optimibilidade das árvores geradoras, mais especificamente na propriedade dos cortes:

> Se T é uma MST de um grafo, então cada uma das arestas t de T é uma aresta mínima dentre as que atravessam o corte determinado por T-t

Segue abaixo uma implementação do algoritmo de Prim, o retorno da função é o custo total da MST:

Listing 6.7: Algoritmo de Prim

```
1  vector< ii > adj[NMAX];
2  int N; int D[NMAX], pi[NMAX]; bool visited[NMAX];
3
4  int prim()
5  {
6    int ans = 0;
7    for(int i = 0; i < N; i++) {
8      D[i] = INF;
9      pi[i] = -1;
10     visited[i] = false;
11   }
12   priority_queue<ii,vector<ii>, greater<ii> > Q;
13   D[0] = 0;
14   Q.push(ii(0,0));
15
16   while(!Q.empty()) {
17     ii top = Q.top(); Q.pop();
18     int u = top.second, d = top.first;
19
20     if(!visited[u])
21     {
22       ans += d; visited[u] = true;
23       for(int i = 0, i < adj[u].size(); i++) {
24         int v = adj[u][i].first, cost = adj[u][i].second;
25         if ( !visited[v] && ( D[v] > cost ) ) {
26           D[v] = cost; pi[v] = u;
27           Q.push( ii( D[v], v ) );
28         }
29       }
```

```
30      }
31    }
32    return ans;
33  }
```

O algoritmo consiste em expandir a árvore pelas arestas de menor valor (linha 16), para tal é utilizado novamente uma fila de prioridade (linha 12). O teste da linha 25 é conhecido como relaxamento das arestas e esta presente em todas implementações do algoritmo de Prim.

6.7 Componentes fortemente conexos

Dizemos que um digrafo é fortemente conexo se é possível a partir de qualquer vértice chegar a qualquer outro.

Da mesma forma, podemos dizer que um subgrafo (um subconjunto dos vértices de um grafo) é fortemente conexo caso exista um caminho entre cada um dos seus componentes. Chamamos esses subgrafos de componentes fortes do grafo.

Muitos problemas podem ser simplificados através da identificação dos componentes fortes do grafo, por exemplo, podemos criar um novo grafo onde todos os elementos de um dado componente são representados por apenas um vértice. Este novo grafo não terá ciclos, permitindo assim uma ordenação topológica do grafo.

6.8 Pontes e articulações

Uma **aresta** é uma ponte se ela for a única a atravesar um corte do grafo (não direcionado), isto é, ao remover esta aresta aumentamos o número de componentes do grafo.

Similarmente, pontos de articulação são **vértices** os quais a remoção aumenta o número de componentes do grafo. Grafos que não possuem articulações (e são conexos) são chamados de biconexos, assim, é necessária a remoção de pelo menos dois vértices para que deixe de ser conexo.

Capítulo 7

Listagem de problemas por assunto

Os problemas apresentados aqui são parte do ICPC da ACM, disponíveis em:
http://uva.onlinejudge.org/

Matemáticos

Geral	113, 202, 256, 275, 276, 294326, 332, 347, 350, 356, 374, 377, 382, 386, 412, 465, 471, 474, 485, 498550, 557, 568, 594, 725, 727, 846, 10006, 10014, 10019, 10042, 10060, 10071, 10093, 10104, 10106, 10107, 10110, 10125, 10127, 10162, 10190, 10193, 10195, 10469
Números Primos	190, 191, 378, 406, 476, 477, 478, 516, 543, 583, 686, 10112, 10221, 10140, 10432, 10451, 10402, 10200, 10490, 10242, 10301, 10445, 10481, 10495, 10468, 10245, 10394, 10699, 10789, 10852, 10871, 10924, 160, 294, 406, 513, 543, 583
Geometria	190, 191, 378, 438, 476, 477, 478, 10112, 10221, 10242, 10245, 10301, 10432, 10451, 105, 10979, 270, 688, 10095, 109, 11096
Big Numbers	324, 424, 495, 623, 713, 748, 10013, 10035, 10106, 10220, 10334
Bases	343, 355, 389, 446, 575, 10183, 10551
Combinatórios	369, 530, 10213, 10288, 10497, 10648, 10790, 10844, 10910, 10918, 11401, 136, 147, 195
Formulas	106, 264, 486, 580
Fatorial	160, 324, 10323, 10338
Fibonacci	495, 10183, 10334, 10450
Sequencias	138, 10408
Modulo	10176, 10551
Permutações	10098, 10252, 136, 146

Programação dinâmica

Geral	108, 116, 136, 348, 495, 507, 585, 640, 836, 10003, 10036, 10074, 10130, 10404
Longest Inc, /Dec, Subsequence	111, 231, 497, 10051, 10131, 103, 10100, 10192, 10405, 10453
Longest Common Subsequence	531, 10066, 10100, 10192, 10405
Counting Change	147, 357, 674
Edit Change	164, 526
SubSet Sum	10120147

Grafos

Buscas (DFS e BFS)	459, 10102, 10000, 544, 334, 10594, 10178, 102, 10798, 208, 216, 291, 612
Menor Caminho	10389, 238, 567, 436, 104, 10171, 423, 10166, 10436, 10557, 10724, 10793, 10803, 10816, 10860, 10959, 10987, 114, 439, 523, 589, 627
All-Pairs Shortest Path	1043, , 436, 534, 544, 567, 10048, 10099, 10171, 112, 117, 122, 193, 336, 352, 383, 429, 469, 532, 536, 590, 614, 615, 657, 677, 679, 762, 785, 10000, 10004, 10009, 10010, 10116, 10543
Fluxo	820, 10092, 10249
Busca Bipartida Maxima	670, 753, 10080
Flood Fill	352, 572
Pontos de Articulação	315, 796
MST	10034, 10147, 10397, 10369, 10307, 10099, 10048, 544, 534, 10462, 10600
Busca de União	459, 793, 10507
Xadrez	167, 278, 439, 750

Diversos

Ad-Hoc	101, 102, 103, 105, 118, 119, 121, 128, 142, 145, 146, 154, 155, 187, 195220, 227, 232, 271, 272, 291, 297, 299, 300, 311, 325, 333, 335, 340, 344, 349, 353, 380, 384, 394, 401, 408, 409, 413, 414, 417, 4 34, 441, 442, 444, 447, 455, 457, 460, 468, 482, 483, 484, 489, 492, 494, 496, 499537, 541, 542, 551, 562, 573, 574 , 576, 579, 586, 587, 591602, 612, 616, 617, 620, 621, 642, 654, 656, 661, 668, 671, 673729, 755, 837, 10015, 100 17, 10018, 10019, 10025, 10038, 10041, 10045, 10050, 10055, 10070, 10079, 10098, 10102, 10126, 10161, 10182, 10189, 10281, 10293, 10487
Anagramas	153, 156, 195, 454, 630
Ordenação	120, 10152, 10194, 10258, 10008, 10020, 10026, 10062, 10131, 10152, 103, 10698, 10810, 10905, 11057, 120, 156, 299, 400, 612, 630, 688
Criptografia	458, 554, 740, 10008, 10062
Algoritmos Gulosos	10020, 10249, 10340, 10026, 10672, 10700, 10821, 10954, 10982, 11039, 11532, 120, 410, 714
Jogos de Carta	162, 462, 555
Parser BNF	464, 533
Simulação	130, 133, 144, 151, 305, 327, 339, 362, 379402, 440, 556, 637, 758, 10033, 10500, 100, 10050, 101, 10267, 10409, 10698, 10813, 10903, 10935, 10978, 11000, 11150, 114, 11459, 11530, 118, 119, 180, 327, 371, 694
Relacionados com saída	312, 320, 330, 337, 381, 391, 392400, 403, 445, 488, 706, 10082
Manipulação de array	466, 10324, 10360, 10443
Busca Binária	10282, 10295, 10474
BackTracking	216, 291422, 524, 529, 539, 571, 572, 574, 10067, 10276, 10285, 10301, 10344, 10400, 10422, 10452
3n + 1	100, 371, 694