

Universidade de São Paulo – Escola de Artes, Ciências e Humanidades

Bacharelado em Sistemas de Informação

Introdução à Ciência da Computação II

Professores Luciano Digiampietri e Fábio Nakano

Data de entrega: 17/11/2012

EXERCÍCIO PROGRAMA 3

Problema da Coloração de Mapas

Neste trabalho você deverá implementar/completar uma classe que encontre a **solução ótima** para o problema da Coloração de Mapas utilizando *backtracking* e além disso deverá implementar também a solução **gulosa** para o mesmo problema.

O problema da Coloração de Mapas consiste em colorir todas as cidades de um mapa usando o menor número de cores de forma que duas cidades adjacentes sempre tenham cores diferentes.

A solução ótima para este problema pode ser encontrada utilizando a estratégia de *backtracking* e tem complexidade exponencial no número de cidades. É possível evitar uma implementação com complexidade exponencial utilizando a estratégia gulosa para a solução deste problema, infelizmente esta solução nem sempre será ótima. Por isso, neste EP serão utilizadas as duas estratégias de forma a comparar a obtenção ótima, porém com grande custo computacional, com a solução rápida (porém nem sempre ótima).

Para a estratégia usando tentativa e erro (*backtracking*), sempre que houver mais de uma solução com coloração mínima, você poderá escolher entre qualquer uma dessas soluções ótimas. Para a solução gulosa, você deverá utilizar a estratégia definida adiante neste enunciado.

Os métodos para identificar a solução não terão retorno (serão *void*) e deverão mudar o atributo cor de todas as cidades de um dado mapa.


A **estratégia gulosa** deverá colorir as cidades da seguinte maneira: dado o arranjo de cidades do mapa atribuir uma cor para cada cidade na ordem que as cidades aparecem nesse arranjo (começando pela cidade localizada na primeira posição do arranjo). Para cada cidade do arranjo, a cor atribuída a ela deverá ser a menor cor que não tenha sido usada em suas vizinhas. Isto é, a primeira cidade do arranjo irá receber a cor 1, a segunda cidade, se for vizinha da primeira, deverá receber a cor 2, caso contrário, receberá a cor 1, e assim por diante. Notem que duas cidades em sequência no arranjo de cidades não são necessariamente vizinhas (cada cidade possui um arranjo próprio para indicar quais são suas vizinhas).

É sabido que em um mapa planar qualquer (por exemplo, na versão planificada do Mapa Mundi) é possível colorir todos os países utilizando-se apenas 4 cores. Isto pode ser demonstrado analisando-se as propriedades de vizinhança possíveis neste tipo de mapa. Porém, neste EP serão utilizados mapas não planares (hiperdimensionais), assim, alguns mapas poderão necessitar de dezenas de cores para serem coloridos.


A seguir são apresentados alguns exemplos (a numeração corresponde aos mapas que já estão disponíveis no código do EP3).

Para este EP, colorir uma cidade significa mudar o valor do atributo numérico cor, presente em cada cidade. As cores válidas são: 1, 2, 3, 4, 5, 6, ... (inteiros positivos).

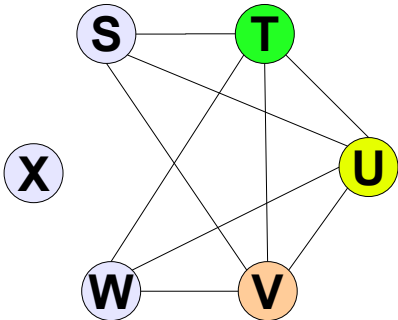
Exemplo 0:

	<pre>##### Exemplo: mapa composto por 2 cidades. ##### Solucao BackTracking: Imprimindo coloracao (2 cidades): Cidade: Y, cor: 1. Cidade: X, cor: 1. Cores usadas: 1 ##### Solucao Gulosa: Imprimindo coloracao (2 cidades): Cidade: Y, cor: 1. Cidade: X, cor: 1. Cores usadas: 1 Tempo da solucao usando BackTracking: 0 ms [cores: 1]; tempo da solucao gulosa: 0 ms [cores: 1].</pre>
---	---

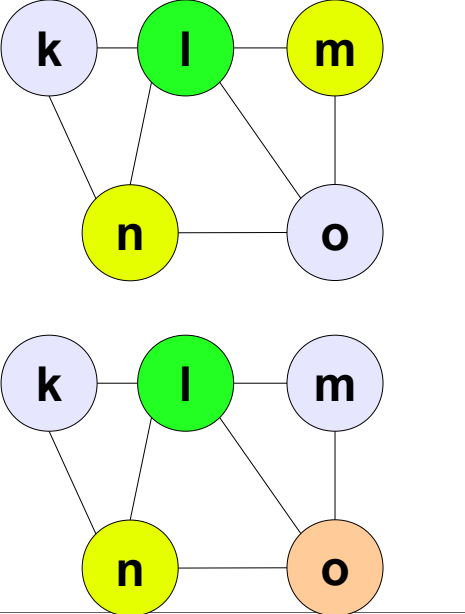
Exemplo 1:

	<pre>##### Exemplo: mapa composto por 2 cidades. ##### Solucao BackTracking: Imprimindo coloracao (2 cidades): Cidade: Y, cor: 1. Cidade: X, cor: 2. Cores usadas: 2 ##### Solucao Gulosa: Imprimindo coloracao (2 cidades): Cidade: Y, cor: 1. Cidade: X, cor: 2. Cores usadas: 2 Tempo da solucao usando BackTracking: 0 ms [cores: 2]; tempo da solucao gulosa: 0 ms [cores: 2].</pre>
---	---

Exemplo 7:

	<pre>##### Exemplo: mapa composto por 6 cidades. ##### Solucao BackTracking: Imprimindo coloracao (6 cidades): Cidade: S, cor: 1. Cidade: T, cor: 2. Cidade: U, cor: 3. Cidade: V, cor: 4. Cidade: W, cor: 1. Cidade: X, cor: 1. Cores usadas: 4 ##### Solucao Gulosa: Imprimindo coloracao (6 cidades): Cidade: S, cor: 1. Cidade: T, cor: 2. Cidade: U, cor: 3. Cidade: V, cor: 4. Cidade: W, cor: 1. Cidade: X, cor: 1. Cores usadas: 4 Tempo da solucao usando BackTracking: 0 ms [cores: 4]; tempo da solucao gulosa: 0 ms [cores: 4].</pre>
--	---

Exemplo 8:



```
##### Exemplo: mapa composto por 5 cidades.
##### Solucao BackTracking:
Imprimindo coloracao (5 cidades):
Cidade: k, cor: 1.
Cidade: l, cor: 2.
Cidade: m, cor: 3.
Cidade: n, cor: 3.
Cidade: o, cor: 1.
Cores usadas: 3
##### Solucao Gulosa:
Imprimindo coloracao (5 cidades):
Cidade: k, cor: 1.
Cidade: l, cor: 2.
Cidade: m, cor: 1.
Cidade: n, cor: 3.
Cidade: o, cor: 4.
Cores usadas: 4
Tempo da solucao usando BackTracking: 0 ms [cores: 3]; tempo
da solucao gulosa: 0 ms [cores: 4].
```

O sistema será composto pelas seguintes classes (muitas delas já implementadas, disponibilizadas juntamente com o código do EP). A classe hachurada é aquela que vocês precisarão implementar/completar. A solução de cada aluno deve compilar e executar corretamente considerando a versão original das demais classes (isto é, as demais classes não devem ser modificadas).

Nome	Tipo	Resumo
Colorir	classe abstrata	Classe que contém os métodos para a solução ótima e solução gulosa do problema da coloração de mapas
Cidade	classe	Classe que descreve uma cidade: nome, cor e arranjo de vizinhas.
ExecutaTestes	classe	Classe “executável” (que possui método <i>main</i>) que pode ser usada para testar as demais classes.
Mapa	classe	Classe que contém o arranjo das cidades de um mapa e alguns métodos auxiliares.

A seguir cada uma das classes é detalhada.

Classe Colorir

Classe abstrata que deve ser completada pelo aluno. Há dois métodos que deverão ser completados, além disso cada aluno poderá adicionar atributos e/ou métodos auxiliares. Cada aluno não deverá mudar a assinatura dos métodos `encontrarColoracaoBackTracking` e `encontrarColoracaoGuloso`. Estes métodos deverão modificar o atributo `cor` de cada uma das cidades do mapa, respeitando-se a regra que duas cidades vizinhas não podem ter a mesma cor.

```
public abstract class Colorir {
    /* Este metodo deve mudar o atributo cor de cada cidade do mapa
     * de forma que cidades vizinhas tenham cores diferentes e
     * usando o menor numero de cores possivel (comecando com cor=1).
     */
    public static void encontrarColoracaoBackTracking(Mapa mapa) {
        //TODO COMPLETAR
    }

    /* Este metodo deve mudar o atributo cor de cada cidade do mapa
     * de forma que cidades vizinhas tenham cores diferentes e
     * usando a seguinte estrategia gulosa:
     *     dado o arranjo de cidades do mapa, atribuir uma cor para cada
     *     cidade na ordem que as cidades aparecem nesse arranjo
     *     (comecando pela cidade localizada na primeira posicao do arranjo);
     *     para cada cidade do arranjo, a cor atribuida a ela devera ser a
     *     menor cor que nao tenha sido usada em suas vizinhas.
     */
    public static void encontrarColoracaoGuloso(Mapa mapa) {
        //TODO COMPLETAR
    }
}
```

Classe Cidade

Classe que representa uma cidade. Uma cidade é composta por 4 atributos, um construtor e dois métodos.

Atributos:

```
public String nome; // Nome da cidade
public boolean cidadeVisitada = false; // atributo auxiliar que pode ser utilizado para
marcar se uma cidade foi visitada durante o backtracking
public int cor = 0; // atributo numerico para indicar a cor que a cidade deve ser pintada,
este atributo devera receber um valor maior que zero
public Cidade[] vizinhas; // arranjo que contem as distancias das cidades vizinhas (cada
objeto do tipo Distancia contem uma cidade e uma distancia)

/* Construtor da classe Cidade */
Cidade(String nomeDaCidade) {
    nome = nomeDaCidade;
}

/* metodo para complementar o construtor, adicionando o arranjo de cidades vizinhas */
void adicionarArranjoDeCidadesVizinhas(Cidade[] vizinh) {
    vizinhas = vizinh;
}

/* metodo para verificar se uma dada cidade eh vizinha da cidade atual */
boolean ehVizinha(Cidade cidade1) {
    for (int i=0; i<vizinhas.length; i++) {
        if (vizinhas[i] == cidade1) return true;
    }
    return false;
}
```

Classe ExecutaTestes

Classe “executável” (possuidora do método main) e que chama três conjuntos de teste para o EP2. Possui um atributo estático para indicar qual conjunto de testes será executado: CONJUNTO_DE_TESTE.

O primeiro conjunto de dados é composto por 9 mapas diferentes. Para o segundo e terceiro conjuntos de dados, os mapas são gerados aleatoriamente e por isso os resultados irão variar.

Para o primeiro conjunto de dados, as saídas esperadas estão no arquivo saida1_ep3.txt. O tempo de execução é impresso apenas de maneira informativa, mas é importante tomar cuidado para não desenvolver uma solução muito ineficiente (isto é, que fique testando diversas colorações impossíveis, por exemplo). **Durante os testes, todo teste que demorar para executar mais de 100 vezes o tempo da solução gabarito será considerado incorreto.**

Classe Mapa

Esta classe contém um mapa (isto é, um arranjo de Cidades) e alguns métodos auxiliares.

```
Cidade[] cidadesDoMapa; // arranjo com todas as cidades do mapa
```

```
/* construtor da classe Mapa */
```

```
Mapa (Cidade[] cids){  
    cidadesDoMapa = cids;  
}
```

```
/* metodo para retornar o numero de cidades do mapa atual */
```

```
public int numeroDeCidades(){  
    return cidadesDoMapa.length;  
}
```

```
/* dado um mapa, verifica se existem problemas com a coloracao */
```

```
static boolean verificarSolucao(Mapa mapa){  
    Cidade[] cidades = mapa.cidadesDoMapa;  
    for (int i=0;i<cidades.length;i++){  
        if (cidades[i].cor<=0){  
            System.err.println("A cidade "+cidades[i].nome+" esta colorida com uma cor  
invalida: " + cidades[i].cor + ".");  
            return false;  
        }else{  
            Cidade[] vizinhas = cidades[i].vizinhas;  
            for (int c=0;c<vizinhas.length;c++){  
                if (cidades[i].cor==vizinhas[c].cor){  
                    System.err.println("Duas cidades vizinhas estao com a mesma  
cor: " + cidades[i] + " e " + vizinhas[c] + "; cor destas cidades: " + cidades[i].cor +  
".");  
                    return false;  
                }  
            }  
        }  
    }  
    return true;  
}
```

```
/* verifica se duas cidades sao vizinhas
```

```
* Dica: eh possivel (e recomendavel) implementar uma boa solucao sem utilizar este metodo  
*/
```

```
static boolean duasCidadesSaoVizinhas(Cidade cidade1, Cidade cidade2){  
    Cidade temp;  
    for (int i=0;i<cidade1.vizinhas.length;i++){  
        temp = cidade1.vizinhas[i];  
        if (temp == cidade2) return true;  
    }  
    return false;  
}
```

```

/* retorno o valor da maior cor usada no mapa */
static int maiorCorDoArranjo(Mapa mapa) {
    int max = 0;
    Cidade[] cidades = mapa.cidadesDoMapa;
    for (int i=0;i<cidades.length;i++){
        if (cidades[i].cor>max) max = cidades[i].cor;
    }
    return max;
}

/* metodo que imprime as cidades de um mapa e sua coloracao */
static void imprimirColoracao(Mapa mapa){
    Cidade[] cidades = mapa.cidadesDoMapa;
    System.out.println("Imprimindo coloracao (" +cidades.length+" cidades):");
    for (int i=0;i<cidades.length;i++){
        if (cidades[i].cor <= 0){
            System.err.println("Ha cores negativas ou nulas na solucao.");
        }else{
            System.out.println("Cidade: " + mapa.cidadesDoMapa[i].nome + ", cor: " +
cidades[i].cor+".");
        }
    }
}

/* coloca a cor 'zero' em todas as cidades do mapa */
static void zerarColoracaoMapa (Mapa mapa) {
    for (int i=0;i<mapa.cidadesDoMapa.length;i++){
        mapa.cidadesDoMapa[i].cor = 0;
    }
}

/* metodo gerador de mapas aleatorios */
static Mapa geradorDeMapa(int cidades, int ligacoes){
    ...
}

```

Regras de Elaboração e Submissão

- Este trabalho é individual, cada aluno deverá implementar e submeter via COL sua solução.
- A submissão será feita via um arquivo zip ou rar (o nome do arquivo deverá ser o número USP do aluno, por exemplo 1234567.zip). Este arquivo deverá conter APENAS o arquivo Colorir.java
- Além deste enunciado, você encontrará na página da disciplina um zip contendo todos os arquivos envolvidos neste trabalho (note que o arquivo Colorir.java a ser implementado também está disponível no site, você precisará apenas completá-lo).
- Qualquer tentativa de cola implicará em nota zero para todos os envolvidos.
- Guarde uma cópia do trabalho entregue e verifique, no Col, se o arquivo foi submetido corretamente.
- A data de entrega é 17 de novembro (sábado) até às 23:00h (com um hora de tolerância), não serão aceitos trabalhos entregues após esta data (não deixe para submeter na última hora).
- Se necessário, implemente na classe Colorir métodos auxiliares e crie atributos. Estes métodos e atributos deverão ser estáticos (*static*) (lembre-se de zerar/inicializar os atributos necessários em cada execução do método *encontrarColoracaoBackTracking* e *encontrarColoracaoGuloso*). Não crie nenhuma classe nova. Não utilize conceitos ou classes que não foram estudados em aula. Só complemente a implementação da classe solicitada (e, no máximo, implemente métodos auxiliares e/ou crie atributos na própria classe).
- **Caso o EP não compile a nota do trabalho será zero.** É importante que você teste seu trabalho executando a classe ExecutaTestes (note que ela não testa todas as funcionalidades do método implementado, por exemplo, ela não verifica se na sua solução foram usadas mais cores do que o necessário para todos os mapas possíveis).
- Preencha o cabeçalho existente no início do arquivo Colorir.java (há espaço para se colocar turma, nome do professor, nome do aluno e número USP do aluno).
- Todas as classes pertencem ao pacote ep3_2012
- Todos os mapas terão no mínimo uma cidade.