---

1. **Quantas câmeras são necessárias para vigiar o Louvre? e mais, onde devemos colocá-las?**

   Estas questões são importantes, ainda mais que roubos de obras de arte realmente já aconteceram no Louvre. Basta recordar o desaparecimento da *Monalisa*, roubada em 1911 pelo italiano Vincenzo Peruggia (o quadro foi reencontrado dois anos depois). Você poderá encontrar mais detalhes deste acontecimento em `http://pt.wikipedia.org/wiki/Mona_Lisa`.

   O nosso problema consistirá em encontrar um intruso no museu. Teremos como entrada o mapa do museu (tal como o Louvre), tendo um certo número de salas, em vários andares, que são ligadas entre elas por portas, escadas e elevadores. Temos a possibilidade de instalar um certo número de câmeras (ou outros dispositivos de detecção) nas salas, tendo como objetivo detectar a presença e o caminho seguido por eventuais intrusos. Infelizmente, o orçamento disponível é limitado, e é necessário portanto reduzir ao mínimo o número de câmeras a serem compradas. Em particular, não podemos colocar por exemplo uma câmera em cada sala. É claro que temos que nos assegurar que o conjunto de câmeras permite detectar todo e qualquer intruso, de maneira que ele não possa escapar.

   Objetivo: encontrar o menor número de salas onde colocar câmeras, de maneira que possamos detectar o movimento de intrusos, logo que eles se deslocam de uma sala para outra (vizinha).

   Observação: a fim de simplificar este problema, iremos supor que cada câmera pode vigiar a totalidade da sala onde ela está, assim como suas entradas/saídas (portas, salas vizinhas, escadas).

   Teremos como entrada um grafo $G = (V, E)$ cujos nós correspondem às salas do museu, e cujas arestas correspondem aos pares de salas vizinhas: duas salas são vizinhas se existe uma porta, uma escada ou um elevador, permitindo passar de uma a outra. Nosso objetivo será o de encontrar o menor subconjunto de nós $S \subseteq V$ tal que todas as arestas do grafo G sejam cobertas (uma aresta é coberta se pelo menos uma de suas extremidades pertence a S).

2. **Introdução**

   O objetivo desta prática é o de programar heurísticas (de complexidade polinomial) para resolução de um problema difícil, conhecido sob o nome de *cobertura de vértices*.

   Alguns arquivos devem ser baixados para a realização desta prática. Estes arquivos encontram-se no SIGAA:

   - `Graph` e `Edge`: representam um grafo por meio de listas de adjacência
   - `Point_2` e `Fenetre` para visualização de grafos
   - `VertexCover`: a ser completado

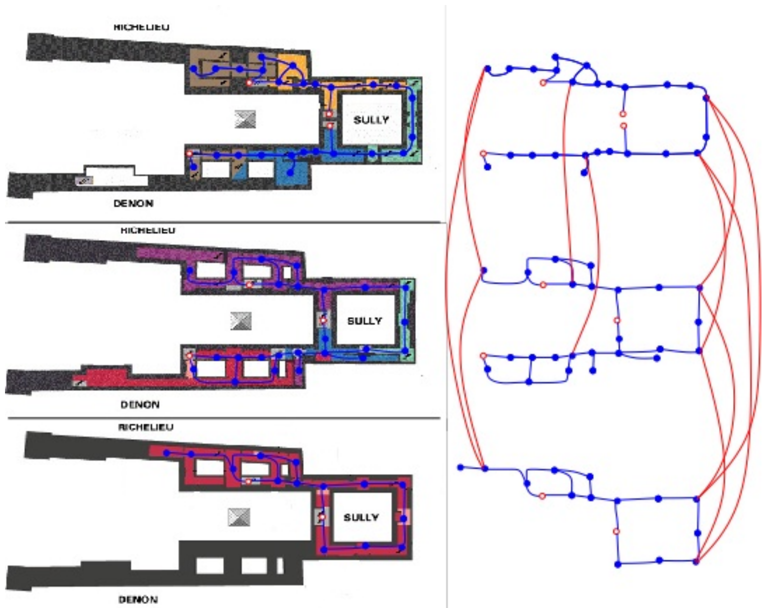


Figure 1: Mapa do Louvre, andares 0, 1 e 2 (à esquerda). Grafo correspondente à direita. Os arcos vermelhos correspondem a escadas/elevadores ligando duas salas em andares diferentes. Os nós vermelhos correspondem a antisalas vigiadas por câmeras

- Exemplos de grafos planares: `k4, cube, dodecahedron, delaunay`
- Grafo do Louvre (0, 1 e 2 andares): `Louvre.txt`
- Outros grafos: `outerplanar.txt`

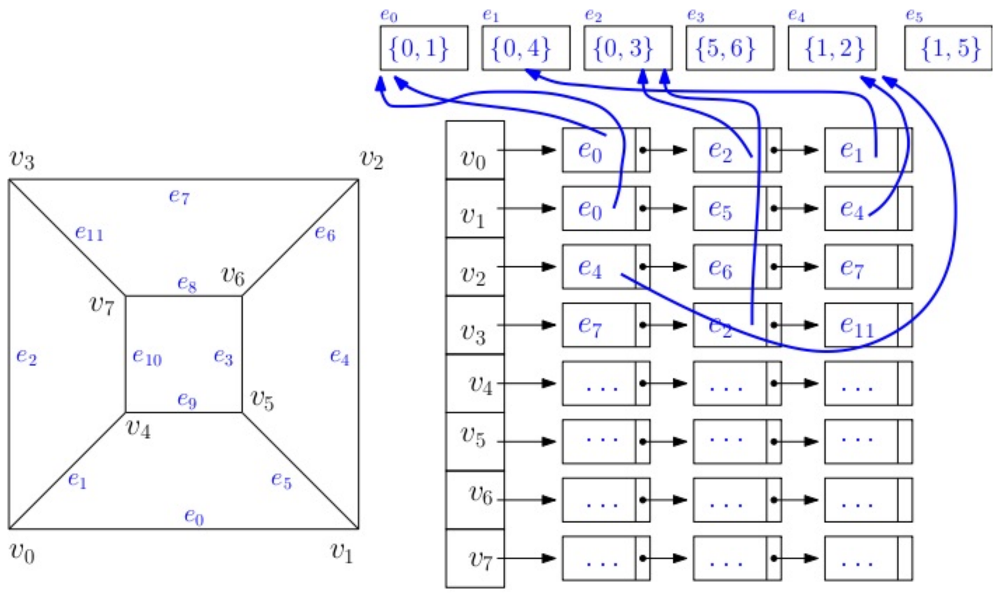
3. **Aquecimento: familiarização com grafos**

   Para a manipulação de grafos em Java utilizaremos a classe `Graph` que fornece uma implementação a base de listas de adjacência (ver imagem abaixo):

   - os nós são (implicitamente) representados por inteiros, de 0 a $n-1$.
   - cada aresta (u, v) (classe `Edge`) contem dois campos, os índices de suas duas extremidades, u e v.
   - para cada nó, seus vizinhos são armazenados em uma lista encadeada (`LinkedList<Edge>`)

   (a) Em sequência à descrição fornecida acima, vamos utilizar uma classe `Graph` munida de:

   - um vetor de listas encadeadas `LinkedList[] vertices` que permite armazenar para cada nó, a lista de arestas adjacentes.
   - um vetor `Point_2 points` que contem as coordenadas geométricas dos pontos associados aos nós (útil para visualização).



Além disso, dispomos dos métodos a seguir (já implementados):

- o construtor `Graph(int n)` que instancia um grafo de tamanho n (com n nós),
- o construtor `Graph(Point_2[] points)` que instancia um grafo e inicializa o vetor com as coordenadas geométricas,
- o construtor `Graph(String filename)`, que cria um grafo (com coordenadas) a partir de um arquivo dado,
- o método `int sizeVertices()` que retorna o número de nós,
- o método `LinkedList<Edge> getEdges(int u)` que retorna a lista de arestas adjacentes ao nó `u` (contidas, por definição, em `vertices[u]`),
- o método `drawGraph()` que imprime o grafo em uma janela Java.

(b) A primeira etapa consiste a munir a nossa representação de um método permitindo adicionar arestas entre nós (que servirá para a construção do grafo em seguida). Complete o método `public void addEdge(int u, int v)` que adiciona uma aresta entre os nós `u` e `v`. Será necessário criar uma nova aresta (classe `Edge`), e adicioná-la às listas de adjacência dos dois nós `u` e `v`.

Atenção: Vamos supor que os nossos grafos são simples (sem arestas de um nó para ele mesmo nem arestas múltiplas). Você deve lembrar que:

- não se pode inserir uma aresta se os índices de suas extremidades não são válidos (índices somente entre 0 e $n-1$);
- não se pode inserir a aresta se $u = v$;
- não se pode adicionar a aresta se os nós u e v já são adjacentes. Somente um objeto do tipo `Edge` é criado em memória para armazenar uma aresta.

(c) Às vezes, para melhor compreender o seu código, você precisará de métodos permitindo imprimir o conteúdo de um grafo (veja exemplo acima). Complete então o método `String toString()` (da classe `Graph`) que:

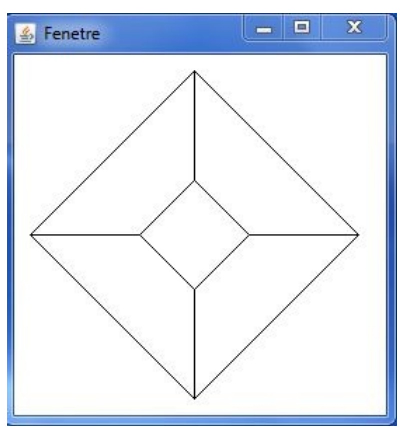
- imprime cada nó em uma linha: seu índice e sua lista de arestas adjacentes (utilize o método `toString()` da classe `LinkedList`);
- imprime as coordenadas geométrica dos pontos associados aos nós do grafo (utilize o método `toString()` da classe `Point_2`).

Para testar o seu código, utilize a função `main` a seguir (da classe `Graph`):

```
public static void main(String[] args) {
  System.out.println("Testing class Graph");
  Graph g=graphFromFile("cube.txt");
  System.out.println(g.toString());
  g.drawGraph();
}
```

Você deve obter o resultado como a seguir para o arquivo `cube.txt`:

```
0:  [ (0,4) (0,3) (0,1) ]
1:  [ (1,5) (1,2) (0,1) ]
2:  [ (2,6) (2,3) (1,2) ]
3:  [ (3,7) (2,3) (0,3) ]
4:  [ (4,7) (4,5) (0,4) ]
5:  [ (5,6) (4,5) (1,5) ]
6:  [ (6,7) (5,6) (2,6) ]
7:  [ (6,7) (4,7) (3,7) ]
point 0:  (5.0,0.0)
point 1:  (0.0,5.0)
point 2:  (-5.0,0.0)
point 3:  (0.0,-5.0)
point 4:  (1.66,0.0)
point 5:  (0.0,1.66)
point 6:  (-1.66,0.0)
point 7:  (0.0,-1.66)
```



4. **Cálculo de uma cobertura de vértices: algoritmo guloso (de Gavril)**

Vamos agora descrever o método que permite calcular uma cobertura de vértices para um grafo dado. O algoritmo a seguir é baseado no cálculo de um matching maximal. Trata-se de uma heurística, com um fator de aproximação C igual a 2: o tamanho da solução encontrada será no máximo 2 vezes o tamanho de uma solução ótima. Além disso, esta heurística é simples de se analisar e fácil de ser implementada: tempo $O(|E|)$. Veja o pseudocódigo abaixo:

```
Algoritmo ApproxVertexCover(G)
  S = {}; \\a cobertura de vértices S é inicialmente vazia
  Seja E o conjunto de todas as arestas do grafo
  Enquanto todas as arestas em E não tiverem sido cobertas faça
    escolha uma aresta e=(u,v) em E
    Se e não está coberta
      adicione os nós u e v no conjunto S
      marque como cobertas todas as arestas adjacentes aos nós u e v
  retorne S
```

Nós vamos utilizar (e completar) uma classe `VertexCover` munida de:

- um campo `Graph g`, que é o grafo para o qual queremos calcular a cobertura de vértices,
- um vetor `boolean[] vertices` que representa a cobertura de vértices S (o nó u pertence a S se e somente se `vertices[u]` for `true`).

Além disso, dispomos dos métodos a seguir (já implementados):

- o construtor `VertexCover(Graph g)` que inicializa os campos da classe,
- o método `void addVertexToCover(int u)` que adiciona o nó u à cobertura de vértices S
- o método `drawVertexCover()` que imprime o grafo assim como os nós da cobertura de vértices.

(a) Vamos agora codificar um procedimento que permite testar se um conjunto S de nós é uma cobertura de vértices. Para isto, será necessário testar se todas as arestas são cobertas por algum nó de S.

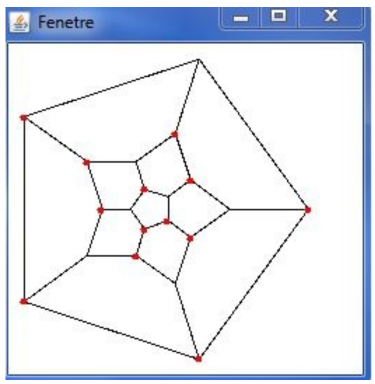
Complete o método `public boolean checkValidity()` da classe `VertexCover` que testa se o conjunto S (representado pelo campo `vertices`) é uma cobertura de vértices.

Teste com a função `testValidity()` a ser chamada na função `main` (com o arquivo `dodecahedron.txt`):

```
public static void main(String[] args) {
  System.out.println("Testing class VertexCover");
  Graph g=Graph.graphFromFile("dodecahedron.txt");
  g.drawGraph();
  testValidity(g);
}
```

O resultado deve ser:

```
Creating an Adjacency List Representation of a graph from a file
Opening OFF file... dodecahedron.txt
Reading vertices...done 20 vertices
Reading edges...done 30 edges
Graph representation created
Testing vertex cover validity... ok
```



(b) Agora vamos codificar o algoritmo guloso (denominado `ApproxVertexCover`) descrito acima.

Complete o método `public void gavrilCover()` da classe `VertexCover` que realiza o algoritmo de Gavril.

Sugestão: utilize um campo `boolean covered` na classe `Edge` para marcar as arestas cobertas durante o desenrolar do algoritmo.

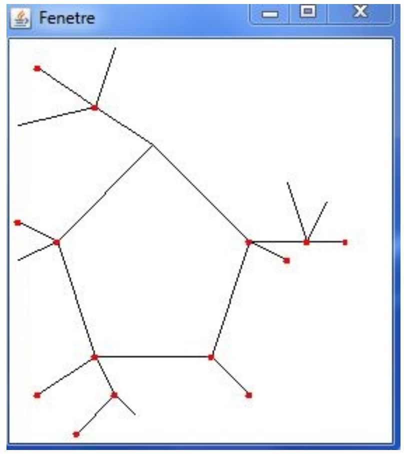
Para testar o seu código, utilize a função `main` a seguir (classe `VertexCover`):

```
public static void main(String[] args) {
  System.out.println("Testing class VertexCover");
  Graph g=Graph.graphFromFile("outerplanar.txt");
  g.drawGraph();
  testGavril(g);
}
```

Abaixo o resultado:

```
Testing class VertexCover
Creating an Adjacency List Representation of a graph from a file
Opening OFF file... outerplanar.txt
Reading vertices...done 21 vertices
```

```
Reading edges...done 21 edges
Graph representation created
Testing greedy algorithm ... running Gavril algorithm...done
Testing vertex cover validity... ok
```



5. **Percurso em profundidade e árvores de cobertura**

   (a) Implemente um método permitindo listar os nós (seus índices) segundo um percurso em profundidade em um grafo.

   Para isto, sugerimos que você siga a estratégia a seguir. Escreva uma função que (de maneira recursiva) explora os vizinhos de um nó dado v: logo que um vizinho u não foi ainda visitado, adicionamos o seu índice à sequência resultado, e continuamos o percurso recursivamente a partir de u.

   Complete o método `public String dfsOrderRecursive(int v, boolean[] visited)` que retorna uma cadeia de caracteres representando as etiquetas dos nós (segundo um percurso em profundidade). O método recebe como entrada:

   - um nó `v` que representa o nó de onde parte o percurso em uma dada etapa,
   - um vetor `booleean[] visited` que descreve os nós que já foram visitados durante o percurso.

   Para testar o seu código, utilize a função `main` a seguir (classe `VertexCover`):

```
public static void main(String[] args) {
  System.out.println("Testing class VertexCover");
  Graph g=Graph.graphFromFile("outerplanar.txt");
  g.drawGraph();
  testDFSTraversal(g);
}
```

   Veja aqui um resultado possível: o resultado pode diferir segundo a ordem na qual se visita os vizinhos de um nó dado.

```
Testing class VertexCover
Creating an Adjacency List Representation of a graph from a file
Opening OFF file... outerplanar.txt
Reading vertices...done 21 vertices
Reading edges...done 21 edges
Graph representation created
Testing dfs traversal ... done
vertex labels (dfs order: recursive traversal):
 0 4 15 16 3 17 18 20 19 2 12 14 11 13 10 1 9 8 5 7 6
```

(b) Inspirando-se do método codificado no item anterior, calcule uma árvore de cobertura do grafo G. Complete o método `public void dfsSpanningTree(Graph tree, int ancestor, int v, boolean[] visited)` que recebe como entrada:

- uma árvore=grafo (já inicializada) `tree` com n nós, e que inicialmente não tem arestas (as arestas são adicionadas à medida que o percurso em profundidade é realizado),
- um nó `v` que representa o nó de onde parte o percurso em uma dada etapa,
- um nó `ancestor` que é o nó visitado na etapa anterior (e que é o ancestral do nó `v` na árvore de cobertura),
- um vetor `boolean[] visited` que descreve os nós que já foram visitados durante o percurso em profundidade.

Na saída do método, a variável `tree` deve conter a referência da árvore T calculada (árvore de cobertura DFS de G).

Observações: (i) supomos que o primeiro nó visitado é aquele de índice 0: na primeira chamada, o índice `ancestor` será também igual a 0, o nó ancestral é indefinido; (ii) tenha atenção para criar as novas arestas durante o percurso: o grafo G e a árvore T não devem compartilhar referências (por exemplo do tipo `Edge`).

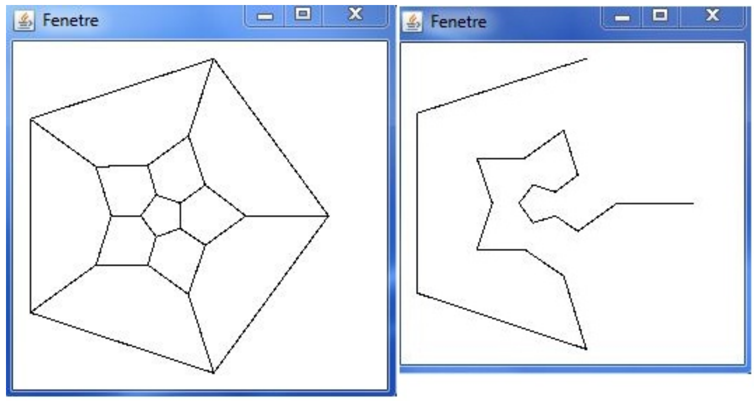
Teste com a função `testSpanningTree()`, a ser chamada a partir da função `main`.

```
public static void main(String[] args) {
  System.out.println("Testing class VertexCover");
  Graph g=Graph.graphFromFile("dodecahedron.txt");
  g.drawGraph();
  testSpanningTree(g);
}
```

Resultado obtido

6. **Cálculo de uma cobertura conexa de vértices (algoritmo de Savage)**

Temos agora todos os ingredientes para implementar uma heurística para o problema de cobertura conexa de vértices, sugerida por C. Savage (1982). Este algoritmo é bastante simples de implementar, com complexidade $O(|E|)$ e permite obter uma cobertura de

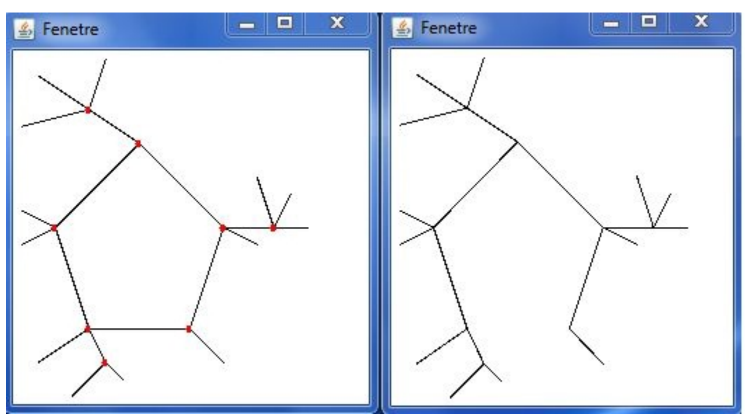


vértices aproximada que é *conexa* e com fator de aproximação 2. Implemente no seu algoritmo as etapas a seguir:

- calcule uma árvore de cobertura T, correspondente a um percurso DFS do grafo,
- adicione à cobertura de vértices todos os nós internos de T (os nós que tem grau $>$ 1),
- adicione à cobertura de vértices a raiz de T (o nó de índice 0, no nosso caso): observe que a raiz poderia ter grau 1.

Complete a função `public void spanningTreeCover()` da classe `VertexCover` que calcula uma cobertura de vértices conexa.

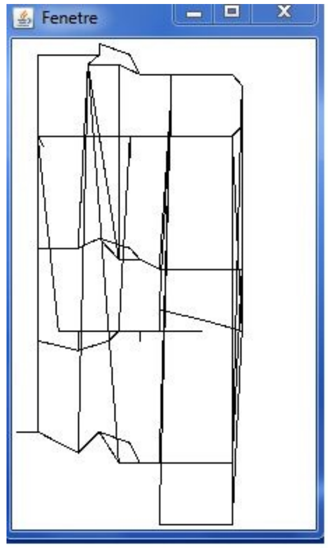
Teste a sua função com a função `testSpanningTreeCover(Graph g)`, chamada a partir da função `main` da classe `VertexCover`. Você deve obter o resultado a seguir com o arquivo `outerplanar.txt`:



Bom, agora basta a você responder a seguinte questão

De quantas câmeras você precisa para vigiar o museu do Louvre?

No SIGAA, você encontra o arquivo representando o mapa do Louvre.



Bom, você pode utilizar os métodos codificados nesta prática... mas você pode também implementar outros se você quiser!

*Este trabalho prático é de autoria de Luca Castelli Aleardi (Poly, France)