Visualização de software baseada no modelo *Little House*

Trabalho Técnico

Gabriel Lage Calegari[1], **Kecia Aline Marques Ferreira**[1]

[1]Departamento de Computação
Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG)
Belo Horizonte – MG – Brasil

gabrielcalegari@gabrielcalegari.com, kecia@decom.cefetmg.br

Resumo. *Desenvolver e manter softwares com qualidade constituem-se tarefas difíceis. A baixa compreensão das estruturas de software e a complexidade de analisar os impactos de alterações são obstáculos para que um software evolua sem grandes deteriorações. Visualizar graficamente o sistema e suas características permitiria identificar mais facilmente possíveis necessidades de reestruturação do software e, com isso, amenizar seu processo de deterioração. Buscando contribuir com um recurso desta natureza, em um trabalho anterior de um dos autores do presente artigo, foi desenvolvido um modelo, denominado Little House, que consiste em uma figura macroscópica genérica das estruturas de software orientado a objetos, baseado no modelo bow-tie que descreve a web. Little House modela o software como uma rede que pode ser particionada em 6 componentes. O objetivo deste trabalho é desenvolver algoritmos de particionamento para o modelo Little House e criar uma ferramenta de visualização de software que exiba as redes de software para esse modelo.*

Abstract. *Developing and keeping the software systems with high quality are difficult tasks. The low understanding of the software structures and the complexity of analysing the impacts of changes in a software system are important impediments to a software evolve without substantial deteriorations. The graphical visualization of the system and its properties – modules, relationship between modules, metrics of modules, metrics of relationships between modules – may allow to better identify possible needs for refactoring, therefore, attenuate the process of deterioration. Aiming to contribute to overcome this issue, a previous work of one of the authors of this paper has defined a model, called Little House, which consists of a macroscopic picture of the object-oriented software structures, based on the bow-tie model that describes the Web. Little House represents a software system as a network which can be partitioned in 6 components. The goal of this work is to develop algorithms to this model and create a software visualization tool that shows the software networks based in Little House.*

1. Introdução

O conceito de software evoluiu de maneira surpreendente durante as últimas décadas de desenvolvimento dos sistemas computacionais, acompanhando, sobretudo, o desenvolvimento do hardware. Desenvolver software de qualidade é uma tarefa difícil, porque além

de todo o conjunto de técnicas complexas que é preciso dominar, o software acompanha as mudanças do mundo real, fazendo-se necessárias sucessivas inclusões e alterações no produto, que geram sua deterioração ao longo de sua evolução. As Leis de Lehman, que tratam da evolução de um software, postulam que a mudança contínua é inerente ao ciclo de vida dos sistemas de software [Lehman 1980].

No entanto, é largamente conhecida a problemática da manutenção de um software. Estima-se que 70% do custo total de um software é devido à sua manutenção, e que esses custos estão relacionados muito mais a melhorias do que a correções [Canfora and Cimitile 1995]. Muitos fatores tornam a manutenção uma tarefa complexa, entre os mais conhecidos estão: compreensão do software, análise de impactos das alterações e testes de regressão. Algumas pesquisas apontaram que compreender a estrutura, funcionalidade e comportamento de um software é responsável de 50 a 90% do tempo gasto em manutenção de software [Canfora and Cimitile 1995]. Uma falha na compreensão do software traz consequências para todo o processo de manutenção, sobretudo em análises de impactos.

Tendo em vista essas dificuldades, muitos trabalhos dedicaram-se a estudar como softwares evoluem, principalmente no que se refere a tamanho e complexidade. Contudo, levando em consideração que a maior parte dos softwares comerciais produzidos atualmente são softwares orientados a objetos, surge a necessidade de investigar especificamente como esse tipo de software é constituído na prática e como ele evolui. Desse modo, nos últimos anos foram publicados alguns trabalhos [Wen et al. 2009; Yao et al. 2009] que modelam um software como uma Rede Complexa, sendo as classes, vértices, e as conexões entre as classes, arestas. Um trabalho clássico [Newman 2003] dentro desse paradigma relata que as redes complexas sofrem do efeito *small-world*, caracterizado pela distância curta entre pares de vértices em uma rede, o que propicia a disseminação de informação na rede, isto é, uma modificação em uma classe de um software pode ser propagada para outras classes, gerando uma futura degradação do software.

Com o objetivo de compreender o efeito *small-world* e as dificuldades intrínsicas à manutenção de software, Ferreira et al. [2011] identificaram uma topologia para redes de software denominada *Little House*. De acordo com *Little House*, as classes de um software podem ser agrupadas em 6 componentes denominados In, Out, LSCC, Tubes, Tendrils e Disconnected. Esse modelo foi definido de modo a prover uma visão macroscópica dos sistemas de software. Visualizar graficamente o sistema e suas características, por exemplo, módulos, relacionamentos entre os módulos, métricas dos módulos, métricas dos relacionamentos entre os módulos permitiria identificar mais facilmente possíveis necessidades de reestruturação do software e, com isso, amenizar seu processo de deterioração. Outrossim, a visualização gráfica dos impactos de uma manutenção forneceria um recurso facilitador da sua avaliação. Todavia, ainda não há uma ferramenta que possibilite visualizar a estrutura de um software com base no modelo *Little House*. O objetivo deste trabalho é desenvolver algoritmos de particionamento para o modelo *Little House* e criar uma ferramenta de visualização de software que exiba as redes de software para esse modelo. Atualmente, encontra-se em desenvolvimento pelo grupo de pesquisa no qual o presente trabalho se insere, uma plataforma para avaliação quantitativa e visualização de software orientado a objetos. A definição de um algortimo apropriado para o particionamento de grafos conforme o modelo *Little House*, bem como

a implementação da ferramenta de visualização de software baseada neste modelo são essenciais para a completude dessa plataforma.

O restante deste artigo está organizado da seguinte forma: Seção 2 discute trabalhos relacionados; Seção 3 descreve o modelo *Little House*; Seção 4 apresenta a proposta de decomposição para esse modelo; Seção 5 apresenta e discute os testes e resultados do algoritmo; Seção 6 apresenta a ferramenta de visualização de software construída; Seção 7 traz as conclusões e indicações de trabalhos futuros.

2. Trabalhos Relacionados

A visualização de software tem sido explorada por muitos pesquisadores nos últimos anos. Uma das razões para isso é que a visualização de software pode auxiliar na compreensão do software, o que é fundamental para realizar qualquer atividade sobre ele. O modelo *Blueprint* [Ducasse and Lanza 2005] é uma proposta de visualização de software em 2D, que promete representar semanticamente as classes, especialmente para um primeiro contato do engenheiro de software com o projeto. Anos antes, os mesmos autores propuseram um modelo de visualização polimétrica, que consiste em uma técnica de visualização de software leve enriquecida com informações métricas, com o objetivo de auxiliar nas fases iniciais da engenharia reversa [Lanza and Ducasse 2003]. A maioria das propostas de visualização são em 2D [Finnigan et al. 1997; Wilhelm and Diehl 2005; Voinea et al. 2005]. Wettel and Lanza [2007] propõem uma visualização 3D para representar o software como uma cidade, em que os prédios são classes e os pacotes são os quarteirões.

Entretanto, a abordagem utilizando-se grafos direcionados têm ganhado destaque [Sharafi 2011]. Nessa abordagem as classes são consideradas vértices, e as conexões entre as classes, arestas. Considerando-se que a classe A usa um serviço da classe B, há uma aresta partindo de A para B. Essa abordagem requer um desenho do grafo que permita que vértices e arestas possam ser exibidos de tal forma que a estrutura do software possa ser visualizada com clareza.

Nesse sentido, alguns conceitos de Redes Complexas têm sido aplicadas sobre essas estruturas de software, tendo-se confirmado que elas se comportam como redes livres de escala (*scale-free networks*) [Wheeldon and Counsell 2003; Baxter et al. 2006; Louridas et al. 2008], nas quais prevalece o fato de que um pequeno grupo de vértices possui grau elevado, enquanto que a maioria dos vértices possui grau pequeno. O grau de um vértice corresponde ao número de arestas conectadas a ele. Wen et al. [2009] identificaram características de redes livres de escala em oito programas Java. Yao et al. [2009] observaram que à medida que o software cresce essas propriedades tornam-se mais visíveis.

Ferreira et al. [2011] desenvolveram um estudo sobre as redes de software, baseando-se no modelo *bow-tie* [Broder et al. 2000], usado para representar a topologia macroscópica das páginas da Web. O estudo levou à identificação de uma topologia mais simples, denominada *Little House*. Ferreira et al. [2012a] realizaram um estudo de caso para caracterizar o modelo sobre um conjunto de softwares. Ferreira et al. [2012c] avaliaram os componentes do modelo sob um ponto de vista quantitativo. Por último, Ferreira et al. [2012b] mostraram como *Little House* pode ser utilizado em estudos de evolução de software. Nesses estudos, foi observado que os componentes LSCC e Out

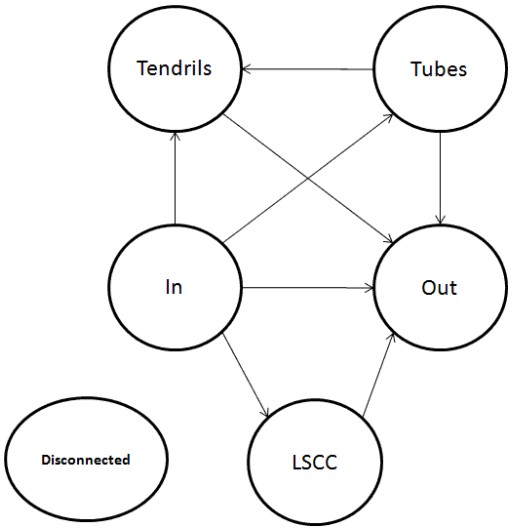


Figura 1. *Little House* - a estrutura macroscópica das redes de software

concentram as classes com mais problemas estruturais, bem como suas classes são as que mais sofrem degradação ao longo da evolução do software.

No entanto, ainda não há uma aplicação que automatize a geração da figura *Little House*. A decomposição *bow-tie* é realizada por meio da aplicação *Pajek* [Batagelj and Mrvar 2010], amplamente utilizada em estudos de Redes Complexas, sendo que após isso, é necessário modificar manualmente a imagem gerada por *Pajek* para se obter o *layout* de *Little House*. *Pajek* é uma ferramenta de uso livre, mas não tem seu código aberto. A forma como *Pajek* realiza o particionamento de um grafo no modelo *bow-tie* não está especificada em um manual ou artigo referente à ferramenta. Isso inviabiliza que *Pajek* seja estendido. Após extensa pesquisa na literatura [Han et al. 2005; Donato et al. 2008; Broder et al. 2000; Yang et al. 2011] sobre o modelo *bow-tie*, encontrou-se somente uma proposta de decomposição para esse modelo descrita por Yang et al. [2011]. Contudo, essa proposta considera a existência de outros componentes não pertencentes à definição original, totalizando 10 componentes. O presente trabalho visa construir uma proposta de decomposição para o modelo *Little House*, que respeite a definição original, além de uma aplicação que automatize a geração da figura *Little House*.

3. O Modelo *Little House*

O modelo *Little House* é constituído de 6 componentes com a mesma denominação do modelo *bow-tie* [Broder et al. 2000]. A Figura 1 mostra o modelo *Little House*. O grafo é direcionado, sendo que uma aresta direcionada de A para B indica que A depende de B. Dentro de cada componente as classes se conectam livremente umas às outras. São eles:

- **LSCC**: *Large Strong Connected Component*, ou Componente Gigante Fortemente Conectado. Neste componente, a partir de uma classe é possivel alcançar todas as outras dentro desse mesmo componente. Ou seja, todas as classes desse componente são dependentes direta ou indiretamente de outra classe desse componente.
- **In**: classes deste componente usam serviços de quaisquer classes do software, mas só podem ser usadas por elementos pertencentes a este mesmo componente.

- **Out**: classes deste componente podem ser usadas por quaisquer classes do software, mas só usam classes pertencentes a este mesmo componente.
- **Tendril**: classes deste componente usam classes do próprio componente ou de *Out*. Além disso, uma classe de *Tendril* pode ser usada somente por classes do próprio componente ou que pertençam a *Tubes* ou a *In*.
- **Tubes**: classes desse componente usam classes do próprio componente, de *Out* ou de *Tendril*. Além disso, uma classe de *Tubes* pode ser usada somente por classes do próprio componente ou que pertençam a *In*.
- **Disconnected**: as classes desse componente não se conectam a classes de outros componentes do software.

4. Algoritmos para o modelo Little House

O modelo *Little House* é uma adaptação do modelo *bow-tie*. Os trabalhos prévios [Han et al. 2005; Donato et al. 2008; Broder et al. 2000; Yang et al. 2011] sobre o modelo *bow-tie* sugerem que o particionamento do grafo seja feito por componentes. Seguindo essa característica, a estratégia para o projeto dos algoritmos de particionamento propostos neste trabalho é a seguinte: em primeiro lugar encontram-se as classes que formam o Componente Gigante Fortemente Conectado (*Largest Strong Connected Component* - LSCC), uma vez que é o núcleo do modelo, e com quem os vértices só de entrada (IN) e só de saída (OUT) se comunicam; portanto, tendo-se encontrado LSCC pode-se encontrar os vértices que compõem IN e os vértices que compõem OUT. A partir de IN e OUT, pode-se encontrar os vértices que formam os componentes restantes. A seguir cada algoritmo é descrito em alto nível. A descrição em alto nível do componente LSCC segue o algoritmo de Tarjan [1972], todas as outras descrições são de nossa autoria.

4.1. LSCC - *Largest Strong Connected Component*

Há alguns algoritmos publicados que permitem encontrar componentes fortemente conectados (SCC) de um grafo. Os mais conhecidos são o algoritmo de Tarjan [1972] e o algoritmo de Kosaraju [Sharir 1981]. De maneira geral, os dois algoritmos tem a mesma complexidade O(V + E), porém, na prática o algoritmo de Tarjan se revela mais eficiente, uma vez que se o grafo estiver representado como uma matriz de adjacências, o algoritmo de Kosaraju requer um tempo de O(V^2). Portanto, elegeu-se o algoritmo de Tarjan para determinar o SCC do grafo.

O algoritmo de Tarjan recebe um grafo direcionado como entrada e produz partições do grafo. Essas partições são listas de vértices fortemente conectados, ou seja, listas cujos vértices formam um caminho para qualquer outro vértice dessas mesmas listas. Cada vértice do grafo aparece em um único componente fortemente conectado.

Esse algoritmo realiza uma busca em profundidade no grafo, iniciando-se de um vértice arbitrário. O algoritmo não visita vértices que já tenham sido explorados. Os vértices são colocados em uma pilha, à medida que são visitados, formando sub-árvores da árvore de busca. A cada sub-árvore retornada, os vértices são retirados da pilha e, então, identifica-se se cada vértice é raiz de um componente fortemente conectado, através de uma propriedade-chave desse algoritmo: a propriedade raiz. Nesse algoritmo, raiz é considerado o primeiro vértice encontrado durante a busca em profundidade. Assim,

uma vez identificado um vértice como raiz, e a recursão tenha sido aplicada para seus vértices sucessores, todos os vértices na pilha a partir das raízes formam um componente fortemente conectado.

A raiz é identificada através de um índice de profundidade de pesquisa para cada vértice: `v.index`, que é o número do vértice na ordem em que ele é descoberto. Também é dado a cada vértice um valor `v.lowlink` que é igual ao menor `v.index` de algum vértice alcançável através de `v`, e é sempre menor ou igual a `v.index` se nenhum outro vértice é alcançável a partir de `v`. Assim, `v` é um vértice-raiz de um componente fortemente conectado se e somente se `v.lowlink == v.index`. Tendo-se encontrado vários componentes fortemente conectados, escolhe-se o maior, e tem-se então o LSCC. O pseudo-código para a identificação dos vértices pertencentes ao LSCC é mostrado a seguir:

```
algorithm tarjan is
  input: graph G = (V, E)
  output: set of strongly connected components (sets of vertices)

  index := 0
  S := empty
  for each v in V do
    if (v.index is undefined) then
      strongconnect(v)
    end if
  repeat

  function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)

    // Consider successors of v
    for each (v, w) in E do
      if (w.index is undefined) then
        // Successor w has not yet been visited; recurse on it
        strongconnect(w)
        v.lowlink := min(v.lowlink, w.lowlink)
      else if (w is in S) then
        // Successor w is in stack S and hence in the current SCC
        v.lowlink := min(v.lowlink, w.index)
      end if
    repeat

    // If v is a root node, pop the stack and generate an SCC
    if (v.lowlink = v.index) then
      start a new strongly connected component
      repeat
        w := S.pop()
        add w to current strongly connected component
      until (w = v)
      output the current strongly connected component
    end if
  end function
```

4.2. IN

Pela definição, os vértices que fazem parte de IN são aqueles que apontam para os vértices de outros componentes, não possuem arestas de entrada partindo de vértices de outros

componentes, e são os únicos que apontam para vértices que compõem LSCC. O algoritmo que permite encontrar IN, leva em consideração o componente LSCC.

O algoritmo baseia-se na seguinte ideia: todo vértice que aponta para um vértice de LSCC, e que não faz parte de LSCC, pertence a IN. Porém, pela definição, como os vértices que compõem IN também se comunicam com os vértices de TENDRILS, TUBES e OUT, e os elementos de IN se comunicam livremente, então, há outros vértices que fazem parte de IN. Eles podem ser encontrados através de uma busca recursiva em cada vértice que já se sabe compor IN. Isto é, cada vértice que aponta para elementos do conjunto IN já encontrado, e que ainda não está nele, é, então, incluído nele. Essa inclusão é feita recursivamente, até verificar todos os vértices. Assim, tem-se então, o componente IN. O pseudo-código para a identificação dos vértices pertencentes ao componente IN é mostrado a seguir:

```
algorithm IN is
  input: graph G = (V, E), LSCC (set of vertex)
  output: set of IN vertices

  IN := empty;

  // Find the vertices which point to LSCC and are not a LSCC vertex.
  for each v in V do
   for each e in E := adjacency(v)
    if (LSCC.contains(e.to) && !LSCC.contains(e.from)) then
      IN.add(v)
    end if
   repeat
  repeat

  // Find the vertices which point to the IN vertices
  for each v in IN do
   discoverIN(v, G)
  repeat

  function discoverIN(v, G)
   // Consider each edge of graph G
    for each e in E do
     if (e.to == v) then
      if (!IN.contains(e.from)) then
       // Add each vertex which points to v
       IN.add(e.from)
       // Find new vertices of IN recursively from the vertex just added
       discoverIN(e.from, G)
      end if
     end if
    repeat
   end function
```

4.3. OUT

O algoritmo que permite encontrar os vértices que fazem parte de OUT é similar ao algoritmo IN, uma vez que os vértices que compõem OUT são aqueles que apenas são apontados por vértices de outros componentes e são os únicos apontados por vértices de LSCC. Desse modo, o algoritmo que permite encontrar vértices de OUT também leva em conta o componente LSCC.

Cada vértice de LSCC que aponta para um vértice que não faz parte de LSCC é, então, um vértice pertencente a OUT. Tendo-se encontrado essa lista inicial de vértices

que compõem OUT, é preciso encontrar os outros vértices que são apontados pelos outros componentes. Eles são encontrados através de uma busca recursiva em cada vértice que já se sabe compor OUT. Cada vértice que é apontado por elementos do conjunto OUT já encontrado, e que ainda não foi classificado, é, então, incluído nele. Essa inclusão é feita recursivamente, até verificar todos os vértices. Ao fim, tem-se, então, o componente OUT. O pseudo-código do algoritmo é mostrado a seguir:

```
algorithm OUT is
  input: graph G = (V, E), LSCC (set of vertex)
  output: set of OUT vertices

  OUT := empty;

  // Find the vertices which are pointed by LSCC and are not a LSCC vertex.
  for each v in V do
   for each e in E := adjacency(v)
    if (LSCC.contains(e.from) && !LSCC.contains(e.to)) then
      OUT.add(v)
    end if
   repeat
  repeat

  // Find the vertices which are pointed by the OUT vertices
  for each v in OUT do
   discoverOUT(v, G)
  repeat

  function discoverOUT(v, G)
   // Consider each edge of graph G
    for each e in E do
     if (e.to == v) then
      if (!OUT.contains(e.to)) then
       // Add each vertex which is pointed by v
       OUT.add(e.to)
       // Find new vertices of OUT recursively from the vertex just added
       discoverOUT(e.to, G)
      end if
     end if
    repeat
   end function
```

4.4. TUBES

Pela definição do modelo *bow-tie* [Broder et al. 2000], os vértices que fazem parte de TUBES são aqueles vértices que são alcançáveis a partir de IN e que podem alcançar OUT. A definição não deixa claro se as conexões entre um elemento de TUBES e os elementos de OUT e IN tem de ser diretas ou não. Como experimentalmente notou-se que são poucos os vértices que realizam essa conexão direta entre IN e OUT, e que a ferramenta *Pajek* [Batagelj and Mrvar 2010], utilizada em estudos de Redes Complexas e que já particiona grafos de acordo com esse modelo, inclui outros vértices além desses, optou-se então por considerar vértices de TUBES aqueles que realizam um caminho de no máximo dois vértices entre IN e OUT. Dessa forma, TUBES possui vértices que constituem um caminho curto entre um vértice de IN e um de OUT.

O algoritmo inicia-se incluindo ao conjunto TUBES todos aqueles vértices que ainda não foram classificados e para os quais há uma aresta vinda de IN e de onde sai uma aresta para OUT. Em seguida, são adicionados também a TUBES os vértices que são

apontados por algum desses vértices recentemente adicionados a TUBES e que apontam para OUT. Além disso, se há algum vértice ainda não classificado cuja aresta parte de IN e que se liga a um outro vértice que também não foi classificado, e que aponta para OUT, então ambos os vértices são incluídos ao conjunto TUBES.

Por último, também fazem parte de TUBES todos os vértices que apontam para os vértices já presentes no conjunto, e também os que apontam para esses, e assim, recursivamente. O pseudo-código do algoritmo proposto para identificar os vértices de TUBES é descrito a seguir:

```
algorithm TUBES is
  input: graph G = (V, E), LSCC, IN, OUT (set of vertex)
  output: set of TUBES vertices

  TUBES := empty;

  // Subtract of the graph the vertices of LSCC, IN and OUT
  S := V - LSCC - IN - OUT

  // Find the vertices which connect directly IN to OUT
  for each v in S do
   if (v is reachable from IN && OUT is reachable from v) then
    TUBES.add(v)
   end if
  repeat

  // Update S
  S := V - LSCC - IN - OUT - TUBES

  // Find the vertices which connect IN to OUT through a vertex of TUBES
  for each v in TUBES do
   EDGES := G.getOutEdgesOf(v)
   for each e in EDGES do
    if (S.contains(e.to)) then
     if (OUT is reachable from e.to) then
      TUBES.add(e.to)
     end if
    end if
   repeat
  repeat

  // Update S
  S := V - LSCC - IN - OUT - TUBES

  // Find the vertices which connect IN to OUT through a path of size 2
  for each v in S do
   if (v is reachable from IN) then
    EDGES := G.getOutEdgesOf(v)
    for each e in EDGES do
     if (OUT is reachable from e.to || TUBES.contains(e.to)) then
      TUBES.add(v)
      if (!TUBES.contains(e.to)) then
       TUBES.add(e.to)
      end if
     end if
    repeat
   end if
  repeat

  // Update S
  S := V - LSCC - IN - OUT - TUBES

  // Find the vertices which point to vertices of TUBES
  for each v in TUBES do
   EDGES := G.getInEdgesOf(v)
   for each e in EDGES do
```

```
    if (S.contains(e.from)) then
     TUBES.add(e.from)
     discoverInVertexForTubes(e.from, G, S)
     S.remove(e.from)
    end if
   repeat
  repeat

  // Find recursively the vertices which point to vertices of TUBES
  function discoverInVertexForTubes(v, G, S)
   EDGES := G.getInEdgesOf(v)
   for each e in EDGES do
    if (S.contains(e.from)) then
     if (!TUBES.contains(e.from)) then
      TUBES.add(e.from)
      S.remove(e.from)
      discoverInVertexForTubes(e.from, G, S)
     end if
    end if
   repeat
  end function
```

4.5. TENDRILS

Para o componente TENDRILS, é necessário verificar os vértices que são alcançáveis de IN ou de TUBES, ou aqueles que podem alcançar OUT. Esses vértices são incluídos ao conjunto TENDRILS. Além disso, faz-se uma verificação recursiva para adicionar a TENDRILS cada um dos vértices para os quais ainda não se identificou seu respectivo componente e que apontam para um vértice recém adicionado a TENDRILS. Esse componente é identificado conforme o algoritmo descrito a seguir:

```
algorithm TENDRILS is
  input: graph G = (V, E), LSCC, IN, OUT, TUBES (set of vertex)
  output: set of TENDRILS vertices

  TENDRILS := empty

  // Subtract of the graph the vertices of LSCC, IN, OUT and TUBES
  S := V - LSCC - IN - OUT - TUBES

  // Find the vertices which are reachable from IN or TUBES, or which reaches OUT
  for each v in S do
   if (v is reachable from IN || v is reachable from TUBES || OUT is reachable from v)
       then
    TENDRILS.add(v)
    discoverVertexForTendrils(v, G, S)
   end if
  repeat

  // Find recursively the vertices which connect to vertices of TENDRILS
  function discoverVertexForTendrils(v, G, S)
   VERTICES := G.getNeighbors(v)
   for each vertex in VERTICES
    if (S.contains(vertex))
     TENDRILS.add(vertex)
     discoverVertexForTendrils(vertex, G, S)
    end if
   repeat
  end function
```

4.6. DISCONNECTED

Os vértices que compõem DISCONNECTED são os vértices restantes, que não foram classificados por nenhum dos algoritmos anteriores. No entanto, há uma outra maneira de verificar os vértices essencialmente desconectados: são aqueles que não foram alcançados pela busca em profundidade do algoritmo de Tarjan, e que possui um `v.index = -1`. A seguir, é mostrado o pseudo-código do algoritmo proposto para se identificar o componente DISCONNECTED.

```
algorithm DISCONNECTED is
  input: graph G = (V, E), LSCC, IN, OUT, TUBES, TENDRILS (set of vertex)
  output: set of DISCONNECTED vertices

  // Subtract of the graph the vertices of LSCC, IN, OUT, TUBES and TENDRILS
  DISCONNECTED := V - LSCC - IN - OUT - TUBES - TENDRILS
```

5. Resultados

Os algoritmos implementados foram testados em 6 softwares diferentes. Foram selecionados softwares Java que possuem uma quantidade não muito grande de classes, entre 100 e 500 classes, para viabilizar a verificação da correção dos algoritmos. Posteriormente, um teste foi realizado também com um software de maior porte, que possui mais de 1000 classes. São eles:

- **JHotDraw:** possui 600 classes. É um *framework* customizável com interface gráfica de usuário que permite o desenvolvimento de aplicações de desenho gráfico. É um software que utiliza amplamente padrões de projeto.
- **JUnit:** possui 252 classes. É um *framework* de código-aberto com suporte à criação de testes automatizados na linguagem de programação Java. Por ser um software intensamente utilizado na indústria, pressupõe-se que sua qualidade estrutural tenha uma qualidade tal que permita seu uso com grande sucesso.
- **DBUnit:** composto por 385 classes, é um software-extensão de JUnit, amplamente difundido para automatizar manipulação de banco de dados durante as operações de teste.
- **Connecta:** é uma ferramenta acadêmica que realiza medições em software Java. A ferramenta coleta métricas de software, além de gerar um arquivo em formato texto contendo informações sobre o grafo que representa o software analisado. A ferramenta possui 384 classes.
- **VISOFT:** é a ferramenta de visualização de software produzida neste trabalho.
- **Sweet Home 3D:** composto por 1453 classes, é uma aplicação de código-aberto de design de interiores amplamente utilizado por profissionais da área de Arquitetura e traduzido para diversos idiomas.

Os grafos dos softwares foram gerados pela ferramenta *Connecta*, que exporta os dados do grafo para um arquivo de texto. Em todos os softwares avaliados nos testes, todos os componentes particionados pelos algoritmos implementados foram comparados aos componentes identificados pelo algoritmo do programa *Pajek* .

Os softwares DBUnit, Connecta, JUnit e VISOFT produziram resultados iguais aos apresentados pelo *Pajek* . Já os softwares JHotDraw e Sweet Home 3D apresentaram

divergências nos componentes TUBES e TENDRILS. *Pajek* acrescenta vértices a TUBES que nosso algoritmo considera como sendo de TENDRILS, sendo que o contrário também se verifica. Essa diferença se explica pela forma como definimos a constituição de TENDRILS e TUBES em nossos algoritmos de particionamento. Nosso algoritmo considera como sendo vértices de TUBES todos aqueles vértices que fazem um caminho de até no máximo 2 vértices entre os componentes IN e OUT, e os que formam maiores caminhos como sendo de TENDRILS. O algoritmo implementado por *Pajek* aplica uma estratégia diferente dessa, a qual não conhecemos, uma vez que não se encontra publicada em um manual ou artigo.

Para seis softwares diferentes, de diferentes números de classes, os algoritmos implementados para o particionamento segundo o modelo *Little House* geram resultados corretos, pois obedecem as definições do modelo.

6. VISOFT - uma ferramenta de visualização de software

A funcionalidade de desenhar um grafo disposto segundo o modelo *Little House* foi implementada em uma ferramenta denominada VISOFT, cuja construção foi iniciada com este trabalho. Para isso, utilizou-se uma biblioteca gráfica para visualização de grafos, denominada JUNG [2010]. Essa biblioteca simplifica o processo de visualização gráfica ao prover uma série de estruturas de dados e oferecer um *framework* para inúmeras outras funcionalidades. A seguir encontra-se uma descrição geral de como funciona o núcleo de visualização de VISOFT.

Como entrada, a aplicação recebe um arquivo em formato de texto compatível com aquele utilizado por *Pajek* e de extensão .net, gerado pela ferramenta *Connecta* [Ferreira 2011], que analisa *bytecodes* Java para gerar métricas de software, além de realizar análise de impactos de modificações em software.

Os algoritmos de particionamento para o modelo *Little House* são aplicados sobre o grafo e cada um dos componentes são dispostos na tela, sob o *layout* de *Little House*. A aplicação foi definida para apresentar duas visões: macroscópica, na qual cada grupo de classes é visto como um único vértice que corresponde a um componente de *Little House*, e circular, em que os vértices que formam cada componente são dispostos formando um círculo. Nesse segundo modo, é possível observar com mais detalhes quais classes formam cada componente e com que classes elas se relacionam.

De modo a facilitar a visualização, cada um dos componentes possui uma cor diferente, e o desenho pode ser ampliado ou reduzido com o auxílio do *mouse*. Além disso, é permitido rotacionar a figura e selecionar vértices específicos de cada componente. Cada vértice é identificado por meio de rótulos, o qual pode ser alterado para um número de identificação interna, para o nome da classe que ele representa ou para o número do componente do qual ele faz parte. Esta última é a opção padrão da aplicação. Além disso, nessa primeira versão da ferramenta, a figura pode ser exportada para o formato *PNG*. A Figura 2 mostra a visão inicial da ferramenta VISOFT. A Figura 3 mostra os resultados gerados pela ferramenta para os programas JUnit e Sweet Home 3D.

7. Conclusão

A visualização de software constitui-se tarefa relevante para a compreensão do software, e para as consequentes tomadas de decisão, uma vez que o uso de recursos visuais na

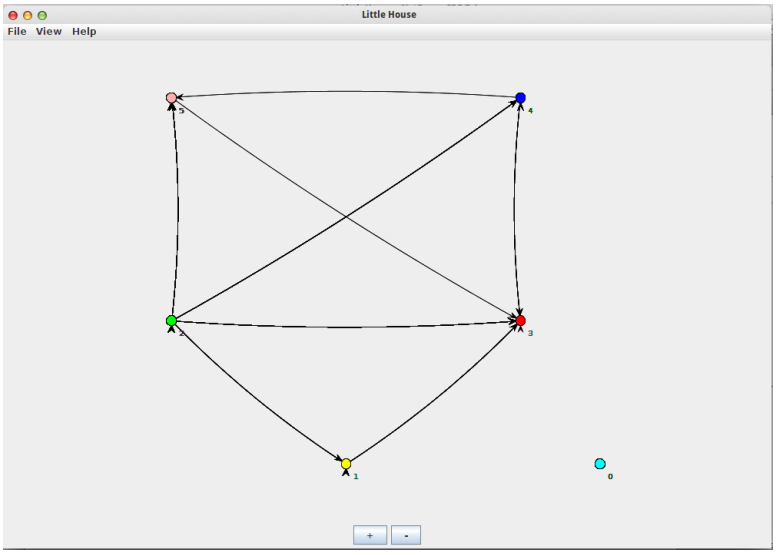
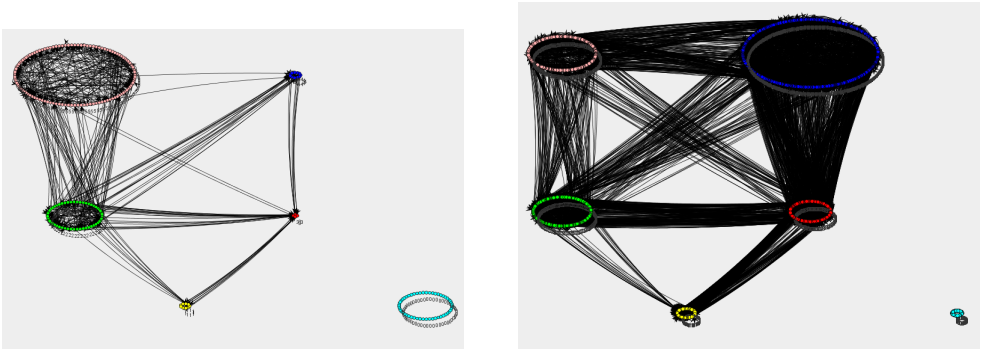


Figura 2. VISOFT - visão inicial da ferramenta



(a) (b)

Figura 3. Modelo *Little House* gerado por VISOFT para os softwares (a) JUnit e (b) SweetHome 3D.

representação de propriedades do software podem ajudar na descoberta de fatos e relacionamentos que de outra forma poderiam permanecer ocultos ou complexos.

Este trabalho teve como objetivo a construção de uma ferramenta de visualização de software baseada no modelo denominado *Little House*, que considera o software como uma rede complexa e representa as conexões entre as classes por meio de seis agrupamentos de classes que possuem determinadas características comuns em termos de rede. Durante a construção da ferramenta foi necessário o projeto e desenvolvimento de algoritmos para o modelo *bow-tie*, usado para definir a estrutura macroscópica da Web, e que serve de base para o modelo *Little House*.

A aplicação produzida, denominada VISOFT, tem como tarefa básica particionar uma rede de software segundo *Little House* e dispor essa rede sob o *layout* que identifica o modelo. Outras funcionalidades da aplicação incluem: interação com o mouse, uso de rótulos para identificar vértices, alteração do modo de visão e exportação da imagem

gerada.

A ferramenta desenvolvida neste trabalho é parte essencial de uma plataforma que vem sendo desenvolvida e que tem por objetivo prover análise quantitativa e visualização gráficas de softwares orientados por objetos. O propósito dessa ferramenta é proporcionar recursos apropriados para a avaliação da qualidade estrutural de softwares por meio de métricas, bem como recursos para aprimorar a compreensão da estruturas de software. Esse tipo de recurso pode ser utilizado tanto para fins de refatoração quanto para manutenção de software. Busca-se, com isso, contribuir para a melhoria da qualidade dos softwares desenvolvidos.

Como trabalhos futuros, identificamos que a integração com a aplicação *Connecta* é importante para obter métricas por classe e por componente. Da mesma forma, estudos sobre histórico de evolução de software por meio de *Little House* poderão ser realizados mais facilmente com a aplicação da ferramenta.

Referências

Batagelj, V. and Mrvar, A. (2010). Pajek: Program for large network analysis. URL: `http://vlado.fmf.uni-lj.si/pub/networks/pajek/`.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. (2006). Understanding the shape of java software. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 397–412, New York, NY, USA. ACM.

Broder, A., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., Tomkins, A., and Wiener, J. (2000). Graph structure in the web. *Comput. Netw.*, 33(1-6):309–320.

Canfora, G. and Cimitile, A. (1995). Software maintenance. In *Proc. 7th Int. Conf. Software Engineering and Knowledge Engineering*, pages 478–486.

Donato, D., Leonardi, S., Millozzi, S., and Tsaparas, P. (2008). Mining the inner structure of the web graph. *Journal of Physics A: Mathematical and Theoretical*, 41(22):224017.

Ducasse, S. and Lanza, M. (2005). The class blueprint: visually supporting the understanding of glasses. *Software Engineering, IEEE Transactions on*, 31(1):75–90.

Ferreira, K. A. M. (2011). *Um Modelo de Predição de Amplitude da Propagação de Modificações Contratuais em Software Orientado por Objetos*. PhD thesis, Universidade Federal de Minas Gerais.

Ferreira, K. A. M., Bigonha, M., Bigonha, R. S., and Gomes, B. M. (2011). Software evolution characterization-a complex network approach. *X Brazilian Simposium on Software Quality-SBQS*, pages 41–55.

Ferreira, K. A. M., Moreira, R. C. N., and Bigonha, M. A. S. (2012a). Identificação de padrões de características estruturais em software orientado a objetos. In *XI SBQS Simpósio Brasileiro de Qualidade de Software*, pages 1–15.

Ferreira, K. A. M., Moreira, R. C. N., Bigonha, M. A. S., and Bigonha, R. S. (2012b). The evolving structures of software systems. In *Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on*, pages 28–34.

Ferreira, K. A. M., Moreira, R. C. N., Bigonha, M. A. S., and Bigonha, R. S. (2012c). A generic macroscopic topology of software networks - a quantitative evaluation. In *Software Engineering (SBES), 2012 26th Brazilian Symposium on*, pages 161–170.

Finnigan, P. J., Holt, R. C., Kalas, I., Kerr, S., Kontogiannis, K., Müller, H. A., Mylopoulos, J., Perelgut, S. G., Stanley, M., and Wong, K. (1997). The software bookshelf. *IBM Systems Journal*, 36(4):564–593.

Han, J., Yu, Y., Liu, G., and Xue, G. (2005). An algorithm for enumerating sccs in web graph. In *Web Technologies Research and Development - APWeb 2005*, volume 3399 of *Lecture Notes in Computer Science*, pages 655–667. Springer Berlin Heidelberg.

JUNG (2010). Java universal network / graph framework. URL: `http://jung.sourceforge.net`.

Lanza, M. and Ducasse, S. (2003). Polymetric views - a lightweight visual approach to reverse engineering. *Software Engineering, IEEE Transactions on*, 29(9):782–795.

Lehman, M. M. (1980). Lifecycles and the laws of software evolution. In *In Proceedings of the IEEE, Special Issue on Software Engineering*, pages 19:1060–1076.

Louridas, P., Spinellis, D., and Vlachos, V. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26.

Newman, M. E. J. (2003). The structure and function of complex networks. *SIAM REVIEW*, 45:167–256.

Sharafi, Z. (2011). A systematic analysis of software architecture visualization techniques. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on*, pages 254–257.

Sharir, M. (1981). A strong-connectivity algorithm and its applications in data flow analysis. *Computers & Mathematics with Applications*, 7(1):67 – 72.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160.

Voinea, L., Telea, A., and van Wijk, J. J. (2005). Cvsscan: Visualization of code evolution. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 47–56, New York, NY, USA. ACM.

Wen, L., Dromey, R., and Kirk, D. (2009). Software engineering and scale-free networks. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 39(4):845–854.

Wettel, R. and Lanza, M. (2007). Visualizing software systems as cities. In *Visualizing Software for Understanding and Analysis, 2007. VISSOFT 2007. 4th IEEE International Workshop on*, pages 92–99.

Wheeldon, R. and Counsell, S. (2003). Power law distributions in class relationships. In *Source Code Analysis and Manipulation, 2003. Proceedings. Third IEEE International Workshop on*, pages 45–54.

Wilhelm, M. and Diehl, S. (2005). Dependency viewer - a tool for visualizing package design quality metrics. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–2.

Yang, R., Zhumadar, L., and Nasraoui, O. (2011). Bow-tie decomposition in directed graphs. In *Information Fusion (FUSION), 2011 Proceedings of the 14th International Conference on*, pages 1–5.

Yao, Y., Huang, S., Ren, Z.-p., and Liu, X.-m. (2009). Scale-free property in large scale object-oriented software and its significance on software engineering. In *Proceedings of the 2009 Second International Conference on Information and Computing Science - Volume 03*, ICIC '09, pages 401–404, Washington, DC, USA. IEEE Computer Society.