Suporte para geração de testes a partir de máquinas de estados

28 de Abril de 2013

Grupo E:

Henrique Ferrolho - 120509079 - henriqueferrolho@gmail.com

Maria Marques - 120509104 - mariajoao.rmarques@gmail.com

Rui Gonçalves - 120509185 - ei12185@fe.up.pt

Índice

Introdução

No âmbito da unidade curricular Conceção e Análise de Algoritmos do curso Mestrado Integrado em Engenharia Informática e Computação foi-nos proposto a realização de um problema onde procederíamos à análise de uma máquina de estados. Para isso recorremos à criação de ficheiros texto que representam máquinas de estado, que por sua vez, vão ser representadas por grafos. O primeiro objetivo é verificarmos se as máquinas de estado são válidas, o segundo é determinar o caminho mais curto para atingir um determinado estado, o terceiro é encontrar os caminhos que passem por todos os estados com custo mínimo e o quarto (opcional) é determinar se a máquina de estados é equivalente a uma outra fornecida.

Neste relatório iremos explicar o contexto do problema, contemplando a sua formalização e respetivos esquemas de forma a facilitar a compreensão do mesmo.

Perguntas e respostas

P: Como vão representar as máquinas de estados?

R: Recorrendo a grafos dirigidos.

P: Quais irão ser os elementos dos grafos?

R: Os grafos vão ter:

- Vértices (vertexes) que representam estados;
- Arestas (edges) que representam transições entre estados, causadas por eventos;
- Um único estado inicial;
- Pelo menos um estado final;
- Todos os vértices devem ser atingíveis;
- Não pode haver transições com o mesmo evento, mesmo estado de origem e diferentes estados de destino.

P: O que é um caminho ponta-a-ponta e o que é que representa?

R: É um caminho que começa no estado inicial e termina num estado final. Representa uma execução possível da máquina de estados.

P: Como vão ser representados os caminhos ponta-a-ponta?

R: Vão ser representados por uma sequência alternada de nomes de vértices (estados) e arestas (eventos).

P: Como é dado o comprimento de um caminho?

R: É igual ao número de arestas.

Formalização do problema

- **Dados:**

  Construção de um grafo dirigido, G = (V, E), em que:

  - Vértices (V): corresponde aos estados da máquina de estados
  - Arestas (E): corresponde às transições entre estados causadas por eventos

- **Inputs:**

  G = <T, E>

  T é o conjunto de estados

  Ei,j é o evento da tarefa Ti para a tarefa Tj

- **Outputs:**

Tarefa 1:

Uma notificação sobre a validade da máquina de estados. Caso não seja válida é também mostrada uma lista das razões pela qual a máquina é inválida.

Tarefa 2:

S = (Ti, <Ti, Eij>, <Tf, _>)

Ti - é o estado inicial

Tf – é o estado final

Função objetivo – Min|S|

É mostrada uma lista de estados e de eventos, representando o caminho mais curto até ao estado especificado.

Tarefa 4:

Uma notificação sobre a equivalência de máquinas de estados. Caso as máquinas não sejam equivalentes é também mostrada uma lista das razões pelas quais não o são.

- **Restrições:**

Tarefa 1:

Um só estado inicial;

Pelo menos um estado final;

Todos os estados têm de ser atingíveis;

Não pode haver transições com o mesmo evento, mesmo estado de origem e diferentes estados de destino.

Tarefa 2:

O estado de destino tem de existir e tem de ser atingível.

Tarefa 4:

As máquinas têm de ter os mesmos estados iniciais, finais, o mesmo número de estados, o mesmo número de transições em cada estado e as mesmas transições para os mesmo estados.

Principais algoritmos implementados

1ª tarefa:

Para testarmos a validade das máquinas de estados, temos que verificar que existe um e um só estado inicial, pelo menos, um estado final, que todos os estados são atingíveis e que não pode haver transições com o mesmo evento, mesmo estado de origem e diferentes estados de destino.

Para verificarmos todos estes requisitos fizemos o seguinte:

- No construtor do grafo ao carregar os vértices, sempre que um vértice é um estado inicial ou final é incrementada uma variável associada ao grafo. No final a variável correspondente ao número de estados iniciais tem de ser igual a 1 e a de estados finais tem de ser maior ou igual a 1;

- Para verificar se todos os estados são atingíveis usámos o método getSources() que retorna um vetor com os estados que não têm arestas incidentes. Para o grafo ser válido, o vetor retornado tem que ser vazio ou conter apenas o estado inicial.

- Para fazer a última verificação é mais difícil: percorremos todos os estados do grafo e asseguramos que para o mesmo grafo não há transições com o mesmo evento para estados diferentes, ou seja, asseguramos que o grafo não é um NFA.

2ª tarefa:

Para encontrar o caminho mais curto desde o estado inicial até um determinado estado utilizamos um método getPath(), que por sua vez usa o método unweightedShortestPath() que usamos nas aulas práticas. Basicamente o algoritmo por trás deste método calcula o caminho mais curto desde um vértice de origem para todos os outros vértices. Depois basta verificar o caminho do vértice que nos interessa.

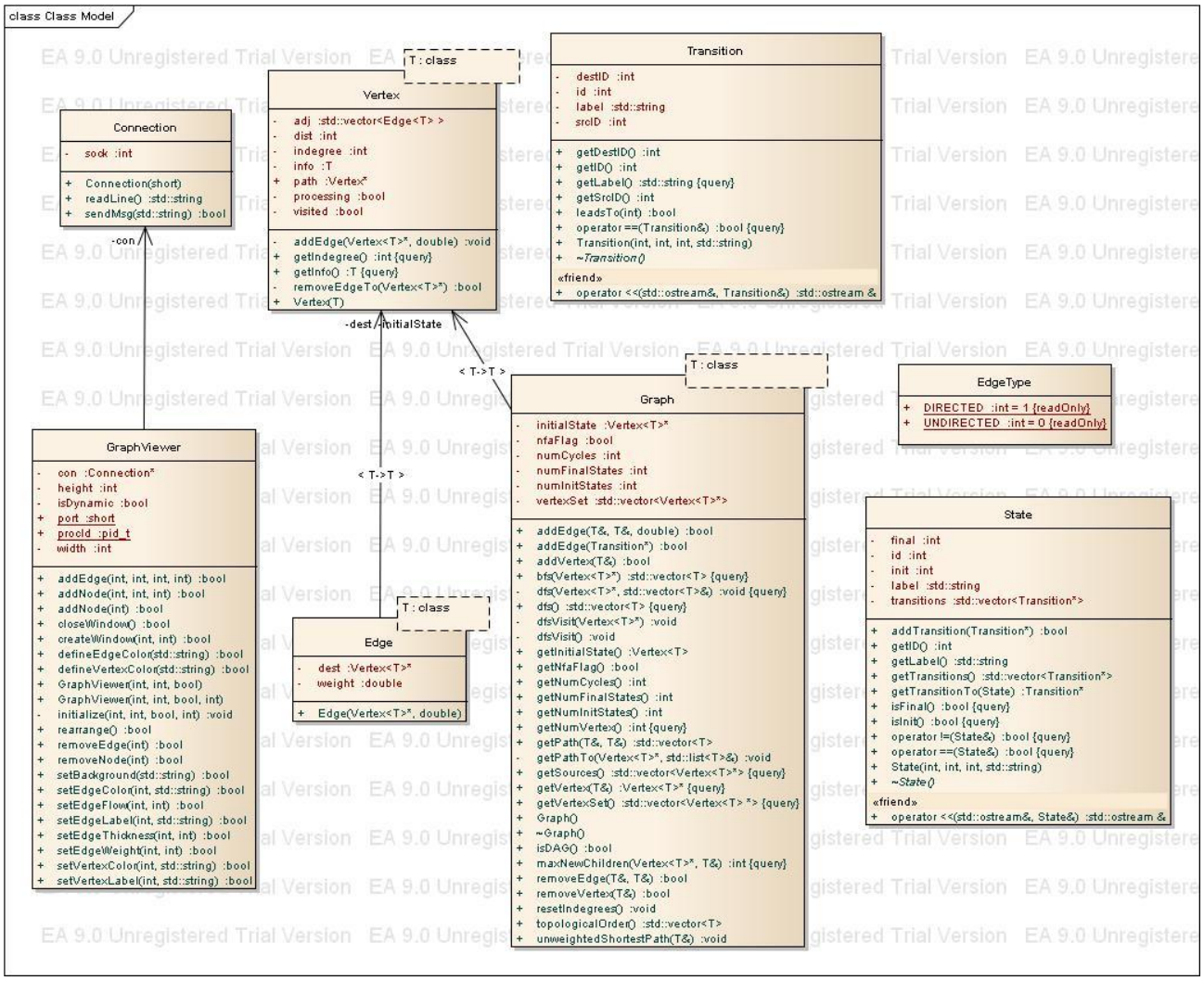
4ª tarefa:

Para determinar a equivalência entre duas máquinas de estados fazemos uma DFS nos dois grafos e analisamos estado a estado os seguintes parâmetros:

- Se um estado é inicial e o outro não;
- Se um estado é final e o outro não;
- Se o número de transições do estado a analisar é diferente do estado do outro grafo;
- Se algum evento de transição do estado a analisar não corresponde a nenhum evento de transição do outro estado.

Se algum destes parâmetros se verificar podemos concluir que os grafos não são equivalentes.

UML


class Class Model
Connection
- sock :int
+ Connection(short)
+ readLine() :std::string
+ sendMsg(std::string) :bool
-con /
T : class
Vertex
- adj :std::vector<Edge<T> >
- dist :int
- indegree :int
- info :T
+ path :Vertex*
- processing :bool
- visited :bool
- addEdge(Vertex<T>*, double) :void
+ getIndegree() :int {query}
+ getInfo() :T {query}
- removeEdgeTo(Vertex<T>*) :bool
+ Vertex(T)
Transition
- destID :int
- id :int
- label :std::string
- srcID :int
+ getDestID() :int
+ getID() :int
+ getLabel() :std::string {query}
+ getSrcID() :int
+ leadsTo(int) :bool
+ operator==(Transition&) :bool {query}
+ Transition(int, int, int, std::string)
+ ~Transition()
«friend»
+ operator <<(std::ostream&, Transition&) :std::ostream &
-dest / initialState
< T->T >
T : class
Graph
- initialState :Vertex<T>*
- nfaFlag :bool
- numCycles :int
- numFinalStates :int
- numInitStates :int
- vertexSet :std::vector<Vertex<T>*>
+ addEdge(T&, T&, double) :bool
+ addEdge(Transition*) :bool
+ addVertex(T&) :bool
+ bfs(Vertex<T>*) :std::vector<T> {query}
- dfs(Vertex<T>*, std::vector<T>&) :void {query}
+ dfs() :std::vector<T> {query}
- dfsVisit(Vertex<T>*) :void
- dfsVisit() :void
+ getInitialState() :Vertex<T>
+ getNfaFlag() :bool
+ getNumCycles() :int
+ getNumFinalStates() :int
+ getNumInitStates() :int
+ getNumVertex() :int {query}
+ getPath(T&, T&) :std::vector<T>
- getPathTo(Vertex<T>*, std::list<T>&) :void
+ getSources() :std::vector<Vertex<T>*> {query}
+ getVertex(T&) :Vertex<T>* {query}
+ getVertexSet() :std::vector<Vertex<T>*> {query}
+ Graph()
+ ~Graph()
+ isDAG() :bool
+ maxNewChildren(Vertex<T>*, T&) :int {query}
+ removeEdge(T&, T&) :bool
+ removeVertex(T&) :bool
+ resetIndegrees() :void
+ topologicalOrder() :std::vector<T>
+ unweightedShortestPath(T&) :void
EdgeType
+ DIRECTED :int = 1 {readOnly}
+ UNDIRECTED :int = 0 {readOnly}
GraphViewer
- con :Connection*
- height :int
- isDynamic :bool
+ port :short
+ procId :pid_t
- width :int
+ addEdge(int, int, int, int) :bool
+ addNode(int, int, int) :bool
+ addNode(int) :bool
+ closeWindow() :bool
+ createWindow(int, int) :bool
+ defineEdgeColor(std::string) :bool
+ defineVertexColor(std::string) :bool
+ GraphViewer(int, int, bool)
+ GraphViewer(int, int, bool, int)
- initialize(int, int, bool, int) :void
+ rearrange() :bool
+ removeEdge(int) :bool
+ removeNode(int) :bool
+ setBackground(std::string) :bool
+ setEdgeColor(int, std::string) :bool
+ setEdgeFlow(int, int) :bool
+ setEdgeLabel(int, std::string) :bool
+ setEdgeThickness(int, int) :bool
+ setEdgeWeight(int, int) :bool
+ setVertexColor(int, std::string) :bool
+ setVertexLabel(int, std::string) :bool
T : class
Edge
- dest :Vertex<T>*
- weight :double
+ Edge(Vertex<T>*, double)
State
- final :int
- id :int
- init :int
- label :std::string
- transitions :std::vector<Transition*>
+ addTransition(Transition*) :bool
+ getID() :int
+ getLabel() :std::string
+ getTransitions() :std::vector<Transition*>
+ getTransitionTo(State) :Transition*
+ isFinal() :bool {query}
+ isInit() :bool {query}
+ operator !=(State&) :bool {query}
+ operator ==(State&) :bool {query}
+ State(int, int, int, std::string)
+ ~State()
«friend»
+ operator <<(std::ostream&, State&) :std::ostream &

Lista de casos de utilização

1- Validação de máquinas de estados;

2- Gerar setups de teste;

3- Verificar equivalência de comportamentos efetivos de sistemas numa abordagem de teste baseada em engenharia reversa.

Dificuldades

A principal dificuldade sentida foi na terceira tarefa. É-nos pedido encontrar um conjunto de caminhos ponta-a-ponta de comprimento total mínimo, cobrindo todas as transições. O enunciado não parece fazer sentido e após falar com a professora Ana Paula Rocha decidimos que não era necessário submeter esta tarefa e que iriamos falar sobre ela com mais detalhe brevemente.

Contribuição no projeto

O aluno Henrique Ferrolho contribuiu maioritariamente com o desenvolvimento do código, os outros alunos contribuíram mais na realização do relatório, mas no geral o grupo trabalhou bem, conseguido assim os alunos com maior facilidade em certas áreas transmitir esses conhecimentos aos restantes.