Relatório - EP1

Tiago K C Shibata
Escola Politécnica
Universidade de São Paulo
tiago.shibata@usp.br

I. Introdução

Esse relatório acompanha o primeiro exercício programa (EP1) da disciplina PCS3556 - Lógica Computacional.

11 de Fevereiro de 2018

II. Tarefa

A tarefa consiste em implementar o algoritmo de fecho reflexivo e transitivo de uma relação binária $R \subseteq AXA$ sobre um conjunto finito A que é descrita por meio de um grafo direcionado, usando recursão em uma linguagem funcional (Elixir). A entrada e saída são representações do grafo como lista de tuplas (lista de arestas representadas por pares indicando origem e destino).

III. Estruturas de dados

No código, a lista de tuplas foi convertida para mapa de adjacências. Cada chave do mapa indica um vértice de origem. O valor da chave é um conjunto ($MapSet$). A escolha das estruturas de dados foi feita tendo em mente performance e facilidade: o uso do mapa evita que varramos toda a lista toda vez que formos buscar adjacências, e o conjunto permite fácil e rápida união de conjuntos quando desejamos adicionar arestas.

IV. Algoritmo

Para buscar todos os vértices alcançáveis a partir de uma origem (fecho transitivo), foi usada uma busca por profundidade em cima do mapa de adjacências. O algoritmo é bastante conhecido e dispensa apresentações.

V. Código e testes

As funções para conversão de lista de tuplas para mapas de adjacências foram implementadas e testadas:

```
def add_edge(graph, edge) do
  {source, destination} = edge
  Map.get_and_update(graph, source, fn(edges) ->
    {edges, edges && MapSet.put(edges, destination) || MapSet.new([destination])}
  end) |> elem(1) |> Map.put_new(destination, MapSet.new)
end

def edge_list_to_adjacency_map(edge_list) do
  Enum.reduce(edge_list, %{}, &(add_edge(&2, &1)))
end
```

Os testes foram desenvolvidos junto com o código:

```
test "adds edges to adjacency map" do
  assert ReflexiveTransitiveClosure.add_edge(%{}, {1, 2}) == %{1 => MapSet.new([2])
         , 2 => MapSet.new}
  assert ReflexiveTransitiveClosure.add_edge(%{2 => MapSet.new([3])}, {2, 3}) ==
         %{2 => MapSet.new([3]), 3 => MapSet.new}
  assert ReflexiveTransitiveClosure.add_edge(%{4 => MapSet.new([6])}, {4, 5}) ==
         %{4 => MapSet.new([5, 6]), 5 => MapSet.new}
end

test "converts edge list to adjacency map" do
  assert ReflexiveTransitiveClosure.edge_list_to_adjacency_map([{0, 1}, {1, 2}, {2,
          1}]) ==
      %{0 => MapSet.new([1]), 1 => MapSet.new([2]), 2 => MapSet.new([1])}
  assert ReflexiveTransitiveClosure.edge_list_to_adjacency_map([{0, 1}, {1, 2}, {1,
         3}]) ==
      %{0 => MapSet.new([1]), 1 => MapSet.new([2, 3]), 2 => MapSet.new, 3 => MapSet
          .new}
end
```

Uma função de busca por profundidade a partir de um vértice foi implementada, usando recursão:

```
def dfs_from_vertex(adjacency_map, visited, source) do
  visited = visited |> MapSet.put(source)
  reachable = MapSet.difference(adjacency_map[source], visited)
  case Enum.fetch(reachable, 0) do
    {:ok, neighbor} ->
      visited = dfs_from_vertex(adjacency_map, visited, neighbor)
      dfs_from_vertex(adjacency_map, visited, source)
    :error ->
      visited
  end
end
```

E uma função para chamar dfs_from_vertex para todos os vértices e retornar um mapa de conjuntos de vértices alcançáveis:

```
def dfs(adjacency_map, already_visited \\ MapSet.new, solution \\ %{}) do
  vertices = MapSet.new(Map.keys(adjacency_map))
  not_visited = MapSet.difference(vertices, already_visited)
  case Enum.fetch(not_visited, 0) do
    {:ok, source} ->
      solution = Map.put(solution, source, dfs_from_vertex(adjacency_map, MapSet.
          new, source))
      dfs(adjacency_map, MapSet.put(already_visited, source), solution)
    :error ->
      solution
  end
end
```

Funções de busca por profundidade foram testadas:

```
test "does DFS" do
  assert ReflexiveTransitiveClosure.dfs(
    ReflexiveTransitiveClosure.edge_list_to_adjacency_map([{0, 1}, {1, 0}])) ==
    ReflexiveTransitiveClosure.edge_list_to_adjacency_map([{0, 0}, {0, 1}, {1, 0},
        {1, 1}])
  assert ReflexiveTransitiveClosure.dfs(
    ReflexiveTransitiveClosure.edge_list_to_adjacency_map([{0, 1}, {1, 2}, {2, 0}])
        ) ==
    ReflexiveTransitiveClosure.edge_list_to_adjacency_map([{0, 0}, {0, 1}, {0, 2},
        {1, 0}, {1, 1}, {1, 2}, {2, 0}, {2, 1}, {2, 2}])
  # 3--->2--->5--->0
  # |     ^
  # |     |
  # |  /----->6<-\
  # \->1<-----/  |
  #    \-->7-----/
  input = [{1, 2}, {1, 6}, {1, 7}, {2, 5}, {3, 1}, {3, 2}, {5, 0}, {6, 1}, {7, 6}]
  output = ReflexiveTransitiveClosure.edge_list_to_adjacency_map([
    {0, 0},
    {1, 0}, {1, 1}, {1, 2}, {1, 5}, {1, 6}, {1, 7},
    {2, 0}, {2, 2}, {2, 5},
    {3, 0}, {3, 1}, {3, 2}, {3, 3}, {3, 5}, {3, 6}, {3, 7},
    {5, 0}, {5, 5},
    {6, 0}, {6, 1}, {6, 2}, {6, 5}, {6, 6}, {6, 7},
    {7, 0}, {7, 1}, {7, 2}, {7, 5}, {7, 6}, {7, 7}])
  assert ReflexiveTransitiveClosure.dfs(ReflexiveTransitiveClosure.
         edge_list_to_adjacency_map(input)) == output
  # Test in random order
  assert ReflexiveTransitiveClosure.dfs(ReflexiveTransitiveClosure.
         edge_list_to_adjacency_map(Enum.shuffle(input))) == output
end
```

Por fim, foi feita e testada a conversão final, de mapa de adjacências para lista de pares:

```
def adjacency_map_to_edge_list(adjacency_map) do
  Enum.reduce(adjacency_map, [], fn({vertex, destinations}, acc) ->
    Enum.map(destinations, &({vertex, &1})) ++ acc
  end)
end

def reflexive_transitive_closure(edge_list) do
  adjacency_map_to_edge_list(dfs(edge_list_to_adjacency_map(edge_list)))
end
```

Testes:

```
def assert_lists_have_same_elements(a, b) do
  assert Enum.sort(a) == Enum.sort(b)
end

def to_edge_list(map) do
  ReflexiveTransitiveClosure.adjacency_map_to_edge_list(map)
end

test "converts adjacency map to edge list" do
  assert_lists_have_same_elements(
    to_edge_list(%{0 => MapSet.new([1]), 1 => MapSet.new([2]), 2 => MapSet.new([1])
          }),
    [{0, 1}, {1, 2}, {2, 1}])
  assert_lists_have_same_elements(
    to_edge_list(%{0 => MapSet.new([1]), 1 => MapSet.new([2, 3]), 2 => MapSet.new,
          3 => MapSet.new}),
    [{0, 1}, {1, 2}, {1, 3}])
end

test "generates reflexive transitive closure" do
  assert assert_lists_have_same_elements(ReflexiveTransitiveClosure.
        reflexive_transitive_closure(
    [{1, 2}, {1, 6}, {1, 7}, {2, 5}, {3, 1}, {3, 2}, {5, 0}, {6, 1}, {7, 6}]),
    [
      {0, 0},
      {1, 0}, {1, 1}, {1, 2}, {1, 5}, {1, 6}, {1, 7},
      {2, 0}, {2, 2}, {2, 5},
      {3, 0}, {3, 1}, {3, 2}, {3, 3}, {3, 5}, {3, 6}, {3, 7},
      {5, 0}, {5, 5},
      {6, 0}, {6, 1}, {6, 2}, {6, 5}, {6, 6}, {6, 7},
      {7, 0}, {7, 1}, {7, 2}, {7, 5}, {7, 6}, {7, 7}])
end
```

VI. Créditos

Agradeço ao professor Ricardo Luis de Azevedo da Rocha pelo conhecimento transmitido.

References

[1] Friedel Ziegelmayer. *Elixir ExDoc*. https://hexdocs.pm/elixir/, acessado em 11/02/2018