IPS Instituto Politécnico de Setúbal

Escola Superior de Tecnologia de Setúbal

Programação Avançada 2021-22

[3c] Grafos | Algoritmo Dijkstra

Bruno Silva, Patrícia Macedo

Sumário 📝

- Implementação dos Algoritmos no TAD Graph;
- Exercícios.

Relembrar o algoritmo do caminho de menor custo

O cálculo do caminho de menor custo entre dois pontos de um grafo, faz-se em duas etapas:

1. Aplicar o algoritmo de Dijkstra a partir do ponto de origem.
2. Construir o caminho de manor custo a partir da tabela construida pelo algoritmo de Dijsktra.
   Estes dois passos são traduzidos em dois algoritmos cujo o pseudocódigo se revê

Algoritmo do caminho de menor custo |Pseudocódigo.

```
Algorithm: Minumum_Cost_Path
Input - graph, vOri- start vertex ,vDst - destination vertex
Output – paths[] - array_with the sequence of_vertex from vOrig _to vDst
          cost - cost associate _to_ the path found
BEGIN
  Disjktra(graph,vOri, costs[], predecessors[]) //algoritm
  path<-[]
  v <- vDst
  WHILE v ≠ vOri DO
          add(path, 0,v)
          v<-predecessors[v]
  END WHILE
  add(path, 0,v)
  cost<-costs[vDst]
END Minumum_Cost_Path
```

Algoritmo Disjktra | Pseudocódigo

```
Algorithm: Dijkstra
Input - (graph,origin)
Output – costs[] e predecessors[]
BEGIN
    FOR EACH vertex v in graph
        costs[v] <- Infinit
        predecessor[v] <- null
    END FOR
    costs[origin] <- 0
    s <- {all vértices of graph}
    WHILE (s IS NOT EMPTY)
        u <- findLowerVertex(graph, costs[],s) //vertice não visitado de menor custo
        IF (costs[u] = Infinit) THEN  RETURN
        removeLowVertex(s,u) // remove vertex u from set s
        FOR EACH v adjacent_of_vertex(u)
            cost <- costs[u] + cost_between(u,v)
            IF (cost < costs[v]) THEN
                costs[v] <- cost;
                predecessor[v] <- u;
            END IF
       END FOR
    END WHILE
END Dijkstra
```

Considerações sobre a implementação dos algoritmo usando o TAD Graph

A implementação do pseudocódigo em JAVA usando o TAD Graph apresentado nas aulas anteriores prossupõe algumas considerações e adaptações. Existem 3 aspetos relavantes a ter em consideração:

1. Como determinar os vértices adjacentes a um determinado vértice.
2. Como obter o custo associado a uma aresta
3. Que tipo de dados selecionar para implementar as variáveis costs[] e predecessors[] ?

(1) Determinar vértices adjacentes

- Se analisarmos o pseudocódigo com alguma atenção verificamos que a implementação do algoritmo utilizando o TAD Graph estudado nas últimas aulas, é quase direto. - A única operação requerida é a determinação dos vertices adjacentes a um determinado vertice : `FOR EACH v adjacent_of_vertex(u)`
No TAD Graph não existe esta operação, no entanto os vertices adjacentes podem ser obtidos por conjução de duas operações

- `incidentEdges(u)` - devolve a lista das arestas incidentes a um vertice
- `opposite(u,edge)` - devolve o vertice oposto dado a aresta incidente e o vértice

(2) Custo associado a uma aresta

- O calculo do caminho de menor custo prossupõe que o grafo é valorado, ou seja que é possivel determinar o custo associado a uma aresta.
- `cost_between(u,v)` refere-se ao custo associado à aresta que liga o vertex u a v.
- Este custo vai estar associado ao elemento instanciado na aresta e é obtido por:
  `cost = edge.element().getCost();`

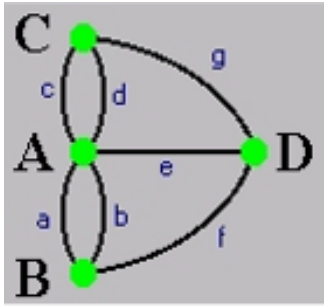
(3) Tipo de dados para costs e predecessors

- Outros dos aspetos a ter em conta é como se vai implementar as variáveis costs[] e predecessors[] ?
- Por uma questão de facilidade, opta-se por usar colecções do tipo `Map` para implementar esses dois arrays
  - `Map<Vertex, Integer> costs` ;
  - `Map<Vertex, Vertex> predecessors` ;

Exercícios 2.2

Implementação do algoritmo Dijkstra | Considerações iniciais

- O algoritmo irá ser implementado sobre uma instância concreta do TAD Graph, tendo como objetivo de calcular o caminho de menor custo entre dois pontos do mapa de *Konisberg*.



- Os vértices são instanciados pela classe `Local`;
- As arestas são instanciadas pela classe `Bridge` (representa as pontes). Cada instância de Bridge terá um custo associado.

Exercícios 2.2 (cont)

Continuando com o projeto iniciado nas aulas anteriores sobre implementação do ADT Grafo, vamos implementar o algoritmo para cálculo do caminho de menor custo entre dois vertices do mapa.

1. Implemente na classe `TadGraphMain` o método:

```
public int minimumCostPath(Vertex<Local> orig, Vertex<Local> dst, List<Vertex<Local>> localsPath)
```

que, a partir do vertice Origem e do vertice de Destino calcula o caminho de menor custo. Para tal, deve implementar o **método auxiliar**:

```
/** * Performs the Dijkstra algorithm starting from 'orig'
    * @param orig the initial vertex
    * @param costs minimum cost from 'orig' to all the other vertex
    * @param predecessors predecessors along the paths
    */
   private void dijkstra(Vertex<Local> orig,
                         Map<Vertex<Local>, Double> costs,
                         Map<Vertex<Local>, Vertex<Local>> predecessors)`
```

Exercícios 2.2 (cont)

2. Retifique o método `main`, de forma a criar o mapa de *Konisberg* com os seguintes valores associados às pontes:
    - (a - 2) (b -1) (c - 3) (d - 1) (g - 4) (f - 4) (e - 15)
3. Mostre a utilização do algoritmo para determinar o caminho de menor custo entre:

```
* O local B e o local C
* O local A e o local D
```

Exercícios 2.2 (cont)

4. Crie outro variante do método `MinimumCostPath` de forma a calcular a sequência de pontes que inclui o caminho de menor custo.

```
public int minimumCostPath(Vertex<Local> orig, Vertex<Local> dst,
                           List<Vertex<Local>> localsPath,
                           List<Edge<Bridge,Local>> bridgePath)
```

note que a sequência de pontes que compõem o caminho será "devolvido" através do parametro de entrada `List<Edge<Bridge,Local>> bridgePath`
Nota : terá que implementar uma nova implementação do método

```
private void dijkstra(Vertex<Local> orig,
                      Map<Vertex<Local>, Double> costs,
                      Map<Vertex<Local>, Vertex<Local>> predecessors
                      Map<Vertex<Local>,Edge<Bridge,Local>> edges)`
```

onde o parametro `Map<Vertex<Local>,Edge<Bridge,Local>> edges` irá conter a informação sobre qual a aresta a percorrer (value), dado um vertice (key).