Programming EXPO SCALA

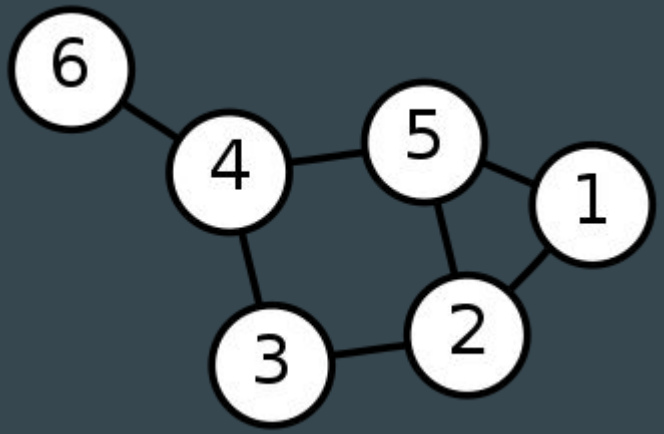
•••

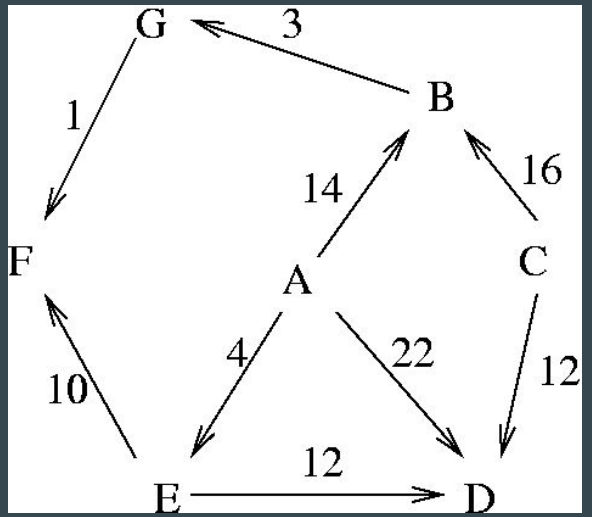
Djikstra Algorithm

Martin Mas - 03/01/18
martin.mas@edu.esiee.fr

Recall - Graph theory

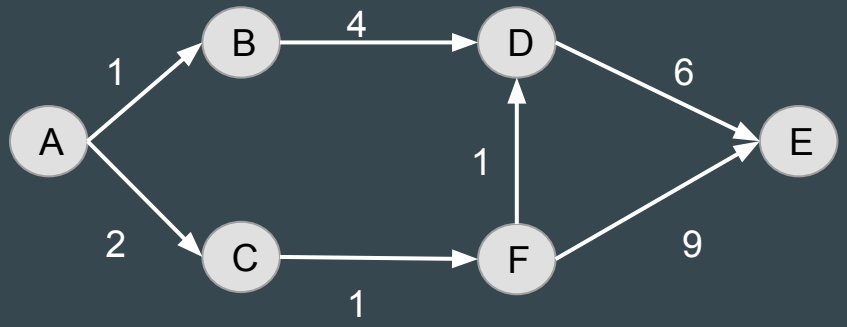
Connected objects (nodes) by links sometimes oriented or weighted.



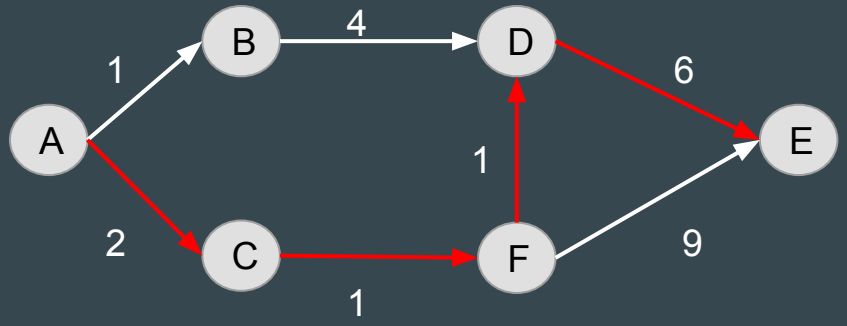


Dijkstra Algorithm

- find the minimum weighted path between two objects

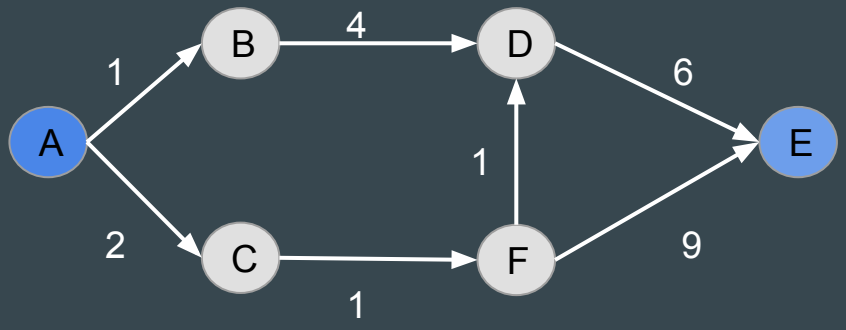


Minimum path from node A to E.



How to

- Inspect neighbors path cost
- Store costs in a priority list
- Select the minimum cost node within the priority list



Dijkstra from A to E

1) Select node : A
Neighbors : B and C
Priority Queue :
- B cost 1 (min)
- C cost 2

2) Select node : B
Neighbors : D
Priority Queue :
- C cost 2 (min)
- D cost 4 + 1 = 5

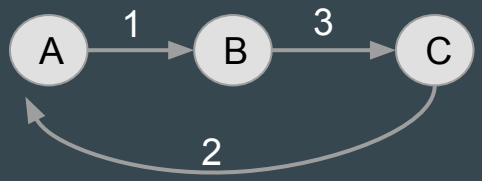
3) Select node : C
Neighbors : F
Priority Queue :
- D cost 5
- F cost 2 + 1 (min)

4) Select node : F
Neighbors : D and E
Priority Queue :
- D cost 5
- D cost 3 + 1
- E cost 9

In programming

2 ways of programming a graph :

- By a matrix where each row and column represent a node and where the values represent the cost edge between two nodes



≡

	A	B	C
A		1	0
B	0		3
C	2	0	

Pors : Easy to construct :

```
val adjMatrix = Array(
 Array(0,1,0,0),
 Array(0,0,3,5),
 Array(2,0,0,1),
 Array(1,0,0,0))
```

Cons : Algorithms needs a more complex structure to run. It adds many variable declaration in the code

Programming

- Object oriented
  - Nodes
    - A string id
    - A list of neighbor node stored as a map (id, cost)
  - Graph : list of nodes

Pors : More understandable and flexibility for other application (ie. drawing etc.)

Cons : More complex to declare and create appropriate functions (get set)

In Scala

Results :

- Object oriented :

```
Node E has 0 neighbor
State : false
 Dijkstra Cost : 10
 Dijkstra previous neighbor : D
A->C->F->D->E -- Path cost : 10
```

Scala Dijkstra code snippet

```
def dijkstra(src : String, dest : String) : List[Node] = {
  val node_src = findNode(src)
  var current_node = node_src
  val node_dest = findNode(dest)

  // Map node stack to memorize the accessible node
  // format : ((Node, Node),Int) -> (Accessible Node, the previous node to come to it), the total cost to come to it)
  var node_stack = collection.mutable.Map[(Node, Node),Int]()

  // While the current is not marked visited or the current node is not the final destination node then keep looping
  while (!current_node.getState() && current_node.getId() != node_dest.getId()) {
    // set current node to visited
    current_node.setState(true)

    // get node neighbors
    val neighbors = current_node.getNeighbors()

    // if neighbors is not empty
    if(!neighbors.isEmpty) {
      // for each neighbor not visited in neighbors list, add them into the node stack with : (Node, the previous node),
      // path cost from previous to node + current node cost
      for (node <- neighbors; if !node._1.getState()) node_stack += ((node._1, current_node) -> (node._2 + current_node.getCost()))

      // find the minimum node from the node stack regarding the memorized cost
      val min : ((Node, Node), Int)= findMin(node_stack)

      // Delete minimun element from the stack
      // and the other path with higher value going to this node
      for(n <- node_stack; if(n._1._1==min._1._1)) node_stack -= n._1
      // update minimum node Djikstra variable with its previous node and the total cost to come to it
      min._1._1.setDijkstra(min._1._2,min._2)
      // set the minimum node as the current node of the algorithm
      current_node = min._1._1
    }
  }
  println(node_dest)
  //printPath(node_src, node_dest)
  val path : List[Node] = djikstraPath(node_src, node_dest).toList
  return path
}
```

Thank you for your attention !
QUESTIONS ?