Pharo Graphs

Sebastian Jordan Montaño and Stéphane Ducasse

December 1, 2021

Layout and typography based on the `sbabook` LaTeX class by Damien Pollet.

Contents

Illustrations

CHAPTER 1

Introduction

Graphs are everywhere and there is well-known algorithms to get the best out of them. In this booklet the most common graph algorithms will be explained along with some case studies where these algorithms can be applied. These algorithms are available in the *Pharo AI graph-algorithms* library available at: https://github.com/pharo-ai/graph-algorithms

This booklet will describe and implement this algorithms.

1.1 Paris metro as graph example

A typical example is a metro map as the one of Paris as shown in Figure 1-1. What we see is that metro lines are connected by some stations for example (Gare du Nord connects line 5 and 4) and some stations are real hub where multiple lines meet such as Chatelet.

When you visit Paris, you are often asking yourself:

- what are the different possibilities to go to that place?
- are they straight connections?
- finding the fastest one?
- finding the one with the least stops (because RER is a fast kind of metro but stopping only in certain places)
- finding the one with the least changes?
- what are all the places that I can reach in 5 stops?

All such questions can be answered by modeling the metro of Paris as a graph and applying algorithms to it.

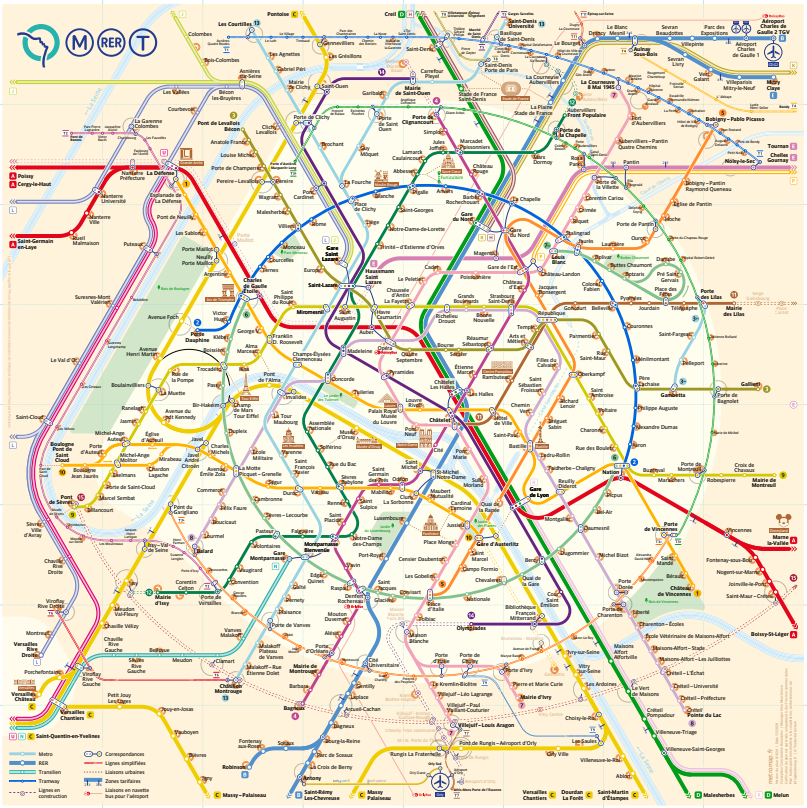

Figure 1-1 Newly design paris metro maps.

Several graphs can be represented:

- we can have graph with only the shared stations and eliminate the station in between (this would not be really useful for users because you do not want to only used shared stations).
- we can have a graph with the time between two stations.

1.2 About tests

To ensure that the algorithms are working properly, we have implemented several tests for this library. We have a graph fixture in which we have implemented different types of graphs. Each graph is implemented on a class side method and the method has a link to a picture to see the graph visually.

Then, in each of the tests for each of the algorithms, we construct a graph and check if the result after running the algorithm is the expected one.

Outline of the document

In this little book we will describe some algorithms that can be applied to graphs. We will start with some basic definitions, then in subsequent chapters we will describe the following algorithms: topological sort, shortest path, Kruskal, Tarjan, HITS, ...

All the algorithms of the library have the same API to set the nodes and the edges. The edges can be both weighted or unweighted. A detailed explanation of how to use the API will be given on Chapter 3.

1.3 How to install

You can install the library executing the following code snippet:

```
Metacello new
  repository: 'github://pharo-ai/graph-algorithms/src';
  baseline: 'AIGraphAlgorithms';
  load
```

CHAPTER 2

Basic definitions

There is a great set of mathematical problems that can be solved using graph models. Graphs are vastly used in all kind of computer science problems. Graphs are discrete mathematical structures that consist of a set of vertices (also called nodes) and a set of edges that connect those vertices. In computer science is more commonly used the term *node* instead of *vertex*. In this booklet the term node is going to be the one used.

There are multiple types of graphs according if the edges are directed or undirected, if the edges are weighted or unweighted and so on. In this chapter, basic graph concepts and graphs types are going to be explained to ease the following of the algorithms.

2.1 Type of Graphs

Directed Graph

A directed graph is a type of graph in which every edges has a direction: an ingoing node and an outgoing node. The most commonly way of representing a graph is to draw it. A directed graph can de drawn like in Figure 2-1:

Undirected Graph

An undirected graph is a type of graph in which the edges does not have a direction. They can be drawn without a line that does not have any arrow heads. It is understood that the graph has no direction if the direction is not specified explicitly as shown in Figure 2-2.

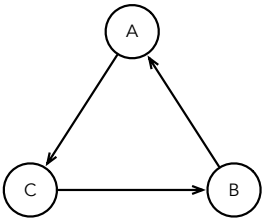

Figure 2-1 A directed graph.

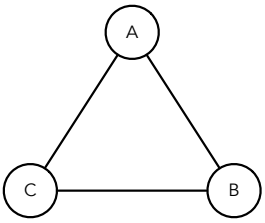

Figure 2-2 An undirected graph.

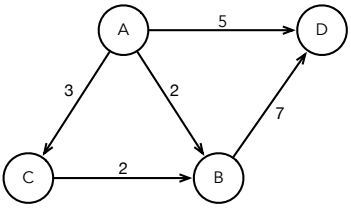

Figure 2-3 A weighted graph: edges have a weight.

Weighted Graph

A weighted graph is a graph that each of its edges has an associated weight (see Figure 2-3). In real life examples, the weights can represent several things. For example, a graph can be a map in which the nodes represent cities and the edges represent the distance between those cities.

Connected Graph

A connected graph is an undirected graph in which exists a path for every pair of nodes. For example, Figure 2-4 represents a connected graph because from any node you can get to any node.

But, Figure 2-5 is a disconnected graph because the nodes F and G are isolated from the rest.

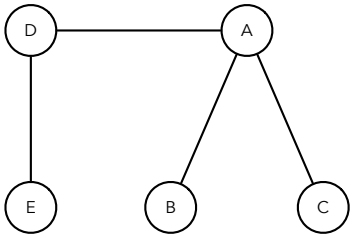

Figure 2-4 A connected graph: all the nodes are reachable from any others.

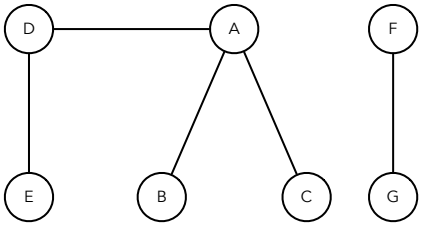

Figure 2-5 A disconnected graph: some nodes are not reachable from others.

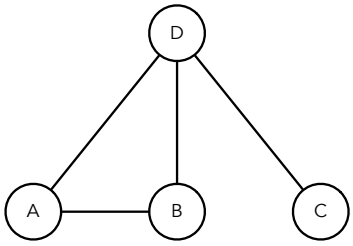

Figure 2-6 A Cycle in a graph.

2.2 Graph Cycle

A graph cycle is a sequence of adjacent nodes in which all nodes are different except of the first and the last one. That means, a graph cycle is a path that ends and starts in the same node without repeating any other node and it has a size grater than 3.

For example, in Figure 2-6 there is a cycle between nodes *A, B, D*.

But, in Figure 2-7 there is no cycle between from node A to C because the node D has to be traveled twice. Nevertheless, there is a cycle between node D, B and C.

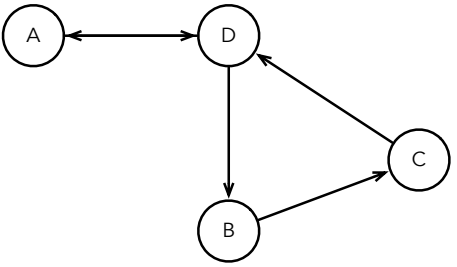

Figure 2-7 In a directed graph, direction is impacting cycle presence.

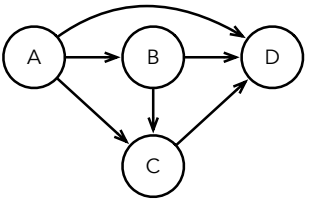

Figure 2-8 A directed Acyclic Graph.

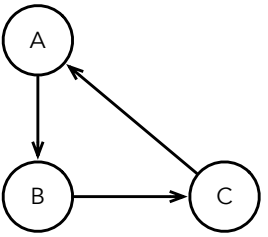

Figure 2-9 A strongly connected graph.

Directed Acyclic Graph (DAG)

Like the name suggests, a directed acyclic graph is a directed graph that does not have any cycles (as shown in Figure 2-8).

Strongly Connected Graph

Unlike the Connected Graph, a Strongly Connect is a **directed** graph in which there is a path for every pair of nodes. Figure 2-9 is a strongly connected graph.

Figure 2-10 is not strongly connected because it is not possible to reach node D from node A. However, if the directions of the graph are deleted, the graph becomes a **undirected** connected graph. For that reason, Figure 2-10 is called

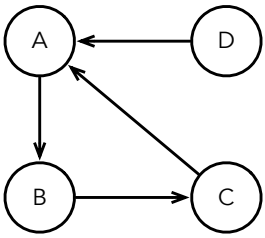

Figure 2-10 A weakly connected graph.

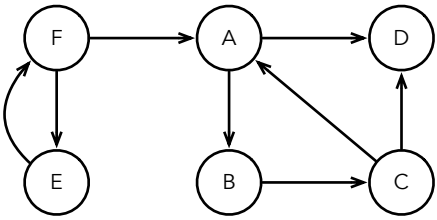

Figure 2-11 Graph with three strongly connected components

a weakly connected graph.

Strongly Connected Component

A strongly connected component of a directed graph is the maximal subgraph that is strongly connected. In the figure there are three strongly connected components in the graph: $\{A, B, C\}$, $\{F, E\}$, $\{D\}$.

2.3 Tree

A tree is a connected graph without cycles. That means that there is only one path between every pair of vertices. But, in the computer science context, normally a tree is represented as a **directed** graph. In that case, the definition will be that a **directed** tree is a directed acyclic graph in which every node has only one incoming (parent) node (See Figure 2-13). If you remove the direction of the directed acyclic graph the reaming graph has to be an **undirected** tree (See Figure 2-12).

2.4 Conclusion

These definitions set the stage for the algorithms that we will now describe. Identifying clearly the kind of graphs an algorithm is applied on is key because the working hypotheses are really important.

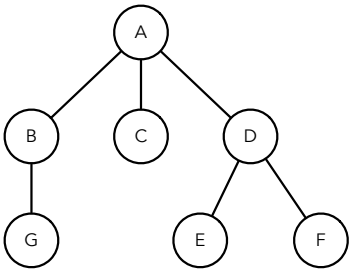

Figure 2-12 A tree: a connected graph without cycles.

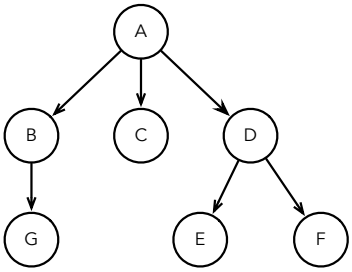

Figure 2-13 A directed tree.

CHAPTER 3

Graph Representation

So far the graphs were represented as drawings. But, to program them we need a data structure. The most used data structures to represent graphs are:

- An adjacency matrix where connection between nodes is basically a cross within a matrix whose entries are nodes. The cross can contain information when we want to manipulate weighted graphs.
- An adjacency list where connection between nodes are given as a list of pairs of nodes or triples in case of weighted edges.

Note that you can use an adjancency list to *define* a graph and use internally a matrix to perform the computation. Basically the internal representation should be seen more as a design choice and it should not impact the way we express algorithms. Providing a good API to manipulate graph will make the algorithm independent from the internal representation and let the developer implement optimizations when needed.

3.1 Graph description

In this library we use the adjacency list data structure to specify a graph. For example, the following graph is created using the messages `nodes:` and `edges:from:to:`. It also defines that the edges are coming from the first element to the second one.

```
| graph |
graph := AIGraphAlgorithm new.
graph nodes: #( $A $B $C ).
graph edges: #(
        #( $A $B )
```

```
        #( $A $C )
        #( $B $C ) )
  from: [:each | each first]
  to: [:each | each second].
graph run.
```

The previous snippet is a template in the sense that:

- First, we instantiate the graph algorithm (in this case an abstract one).
- Then, we instantiate the nodes. And finally, we set the edges. Note that for the edges we need to pass a list. The elements inside a list can be any kind of object. In the above example the objects are also a list.
- And then, we need to specific a block that is needed to obtain the `from:` and `to:` relationships. In the example, the *in node* is the first element of the list and the second one is the *out node*. So, we need only to send the messages `first` and `second`.

We will often define our graphs this way.

Blocks and symbols.

Since we are a bit lazy to type, we pass directly the method names as symbol that we want to apply on the edge to extract information.

```
| graph |
graph := AIGraphAlgorithm new.
graph nodes: #( $A $B $C ).
graph edges: #(
        #( $A $B )
        #( $A $C )
        #( $B $C ) )
  from: #first
  to: #second.
graph run.
```

Weigthed graphs

To represent weighted graphs we use the `edges:from:to:weight:` method.

```
| graph |
graph := AIGraphAlgorithm new.
graph nodes: #( $A $B $C ).
graph edges: #(
  #( $A $B 2 )
  #( $A $C 4 )
  #( $B $C 7 ) )
  from: [:each | each first]
  to: [:each | each second]
  weight: [:each | each third].
```

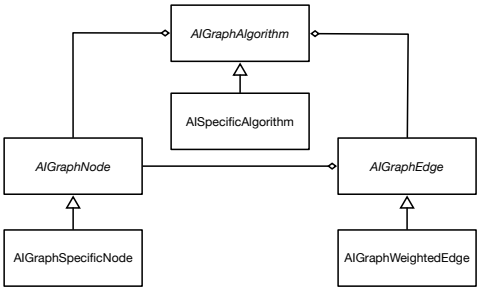

Figure 3-1 Basic graph object-oriented representation: two collections of elements.

```
graph run.
```

So in this case the library is going to take the third element as the weight for each of the edges.

3.2 Basic graph elements

As shown by Fig. 3-1, a graph is basically a collection of edges and a collection of nodes. The nodes are entities that can contain some specific value from the domain but also for the algorithm execution. This is why we get a rich hierarchy of nodes as shown below.

3.3 About nodes

The graph algorithms of this library use different nodes. All the nodes inherit from the same abstract class called `AIGraphNode` and show below where indentation represents inheritance (Fig. 3-2).

```
AIGraphNode
  AIBFSNode
  AIDisjointSetNode
  AINodeWithPrevious
    AiHitsNode
      AIWeightedHitsNode
  AIPathDistanceNode
  AIReducedGraphNode
  AITarjanNode
```

We have different subclasses because a specific algorithm may need some store some special state. But all the nodes share a common API.

The most important methods of the API are:

- `node from:`

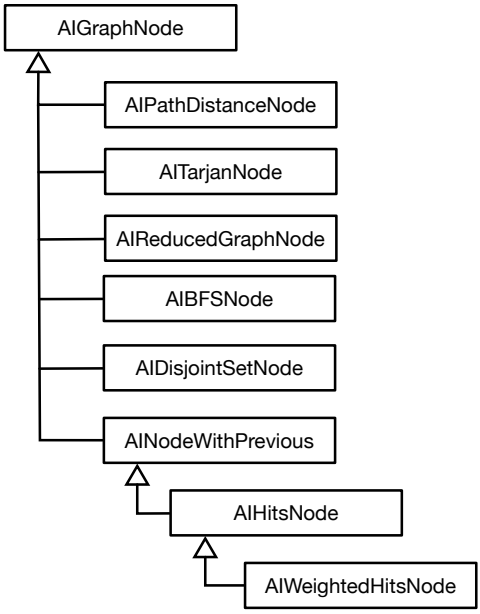

Figure 3-2 Node hierarchy.

- `node from:edge:`
- `node adjacentNodes`
- `node model`
- `node model:`
- `node to:`
- `node to:edge:`

3.4 Graph algorithm inheritance tree

All of the algorithms are subclasses of `AIGraphAlgorithm` as shown below where indentation represents inheritance and shown in Fig. 3-3.

```
AIGraphAlgorithm
  AIBFS
  AIBellmanFord
  AIDijkstra
  AIGraphReducer
  AIHits
    AIWeightedHits
  AIKruskal
  AIShortestPathInDAG
```

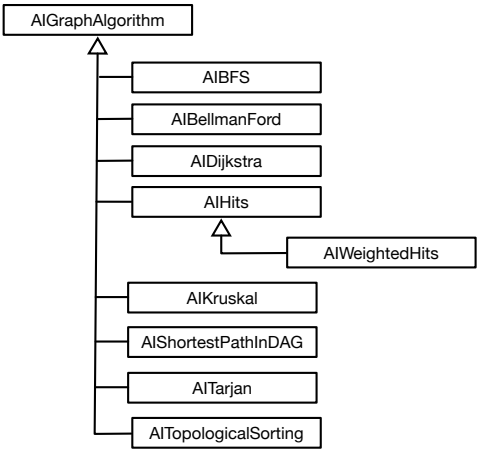

Figure 3-3 Algorithm hierarchy.

```
AITarjan
AITopologicalSorting
```

As the nodes, all the graph algorithms of this library share a common API also. The class `AIGraphAlgorithm` provides the common API to add nodes, edges, searching the nodes, etc.

Some of the methods of the API are:

- `algorithm nodes:`
- `algorithm nodes`
- `algorithm edges`
- `algorithm edges:from:to:`
- `algorithm edges:from:to:weight:`
- `algorithm findNode:`
- `algorithm run`

3.5 Conclusion

After this basic introduction we are ready to present and implement the algorithms.

CHAPTER 4

Topological sorting

Let us start with our first algorithm: a topological sort. A topological sort makes that a node is treated before the nodes that depend on it are treated. If you consider tasks, it means that you want to do first a task before doing the ones that depend on this one.

Topological sorting is a way of ordering a **directed acyclic graph** such that for every directed edge (U, V) from node U to node V, U becomes first in the resulting ordering.

We present here one algorithm named Khan's algorithm.

4.1 Example

The topological sorting can be applied to a graph in which its nodes represents software dependencies. For example to install a library, there are some modules that need to be installed before others. So, in this case, a topological sort is useful to know which modules to install first (the one that has no dependencies) and so on.

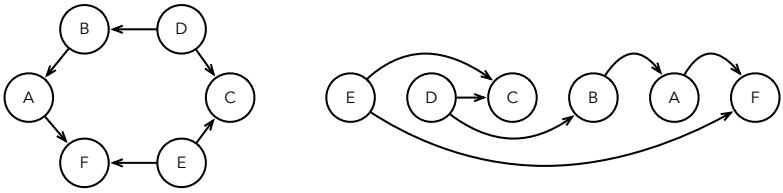

Figure 4-1 A graph and one of its topological sorts.

Now for a same graph multiple topological sorts are possible just because since we treat first the nodes having no dependencies their traversal order can be different and in addition the traversal order of their dependent can be different too. Figure 4-2 shows one graph and one of the possible topological sort.

4.2 Kahn's algorithm

To apply a topological sort to a graph, the graph must be a directed acyclic graph (DAG). There is at least one topological possible order for a DAG.

The algorithm that is used in this library is the Kahn's algorithm. It has a time complexity of $O(V+E)$. The pseudocode taken from https://en.wikipedia.org/wiki/Topological_sorting is the following one:

```
L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edge
while S is not empty do
    remove a node n from S
    add n to L
    for each node m with an edge e from n to m do
        remove edge e from the graph
        if m has no other incoming edges then
            insert m into S
if graph has edges then
    return error (graph is not a DAG)
else
    return L (a topologically sorted order)
```

This algorithm is implemented in the class `AITopologicalSorting` subclass of `AIGraphAlgorithm`. The parent class (`AIGraphAlgorithm`) provides all the mechanisms to handle the implementation of the graph data structure. `AITopologicalSorting` has the only responsibility: to implement the logic of the algorithm.

The following proposes a first implementation

```
AITopologicalSorting >> run

  topologicalSortedElements := OrderedCollection empty.
  nodesWithNoIncomingEdges := LinkedList empty.

  "Obtain all the nodes without incoming nodes"
  nodesWithNoIncomingEdges addAll:
    (nodes select: [ :node | node incomingNodes isEmpty ]).

  [ nodesWithNoIncomingEdges isNotEmpty ] whileTrue: [
    | node |
    node := nodesWithNoIncomingEdges removeFirst.
```

```
    topologicalSortedElements addLast: node model.

    "Remove all the edges of node from the graph"
    node adjacentNodes do: [ :adjacentNode |
      adjacentNode incomingNodes remove: node.
      adjacentNode incomingNodes ifEmpty: [
        nodesWithNoIncomingEdges add: adjacentNode ] ].
    node adjacentNodes: #(  ) ].

  "If the graph still has edges"
  (nodes anySatisfy: [ :node | node adjacentNodes isNotEmpty ])
    ifTrue: [ Error signal: 'Not a DAG (Directed Acyclic Graph)' ].

  "Return the topological order the first element being the node
    without any dependencies"
  ^ topologicalSortedElements
```

4.3 Improving the implementation

The method `run` is a bit too long to our taste. The fact that we have to add comments to separate its code logic is a call to define separate methods.

First we define two methods `initializeElements` and `gatherNoIncomingNodes`. And we express the algorithm by redefining the method `run` as follows:

```
AITopologicalSorting >> initializeElements

  topologicalSortedElements := OrderedCollection new.
  nodesWithNoIncomingEdges := LinkedList new
```

Note that `nodesWithNoIncomingEdge` uses a linked list because it has a better time complexity for removing the first element.

```
AITopologicalSorting >> gatherNoIncomingNodes
  "Obtain all the nodes without incoming nodes"

  nodesWithNoIncomingEdges addAll:
      (nodes select: [ :node | node isLeaf ]).
```

```
AITopologicalSorting >> run

  self initializeElements.
  self gatherNoIncomingNodes.
  [ nodesWithNoIncomingEdges isNotEmpty ] whileTrue: [
    | node |
    node := nodesWithNoIncomingEdges removeFirst.
    ...
```

Then we continue by extracting the handling of node dependencies and extract the validation.

```
AITopologicalSorting >> removeEdgesOf: node

  node adjacentNodes do: [ :adjacentNode |
    adjacentNode incomingNodes remove: node.
    adjacentNode incomingNodes ifEmpty: [
      nodesWithNoIncomingEdges add: adjacentNode ] ].
  node adjacentNodes: #( )].
```

```
AITopologicalSorting >> graphHasEdges

  ^ nodes anySatisfy: [ :node | node adjacentNodes isNotEmpty ].
```

```
AITopologicalSorting >> run

  self initializeElements.
  self gatherNoIncomingNodes.
  [ nodesWithNoIncomingEdges isNotEmpty ] whileTrue: [
    | node |
    node := nodesWithNoIncomingEdges removeFirst.
    topologicalSortedElements addLast: node model.
    self removeEdgesOf: node ].

  self graphHasEdges ifTrue: [
    Error signal: 'Not a DAG (Directed Acyclic Graph)' ].
  ^ topologicalSortedElements
```

Now the logic of the algorithm is clearer and at a nice level of abstraction. In addition we can focus on the structure of the algorithm, treated nodes are added one by one to the `topologicalSortedElements` collection while dependents are added to the working list `nodesWithNoIncomingEdges`. And the algorithm iterates until the working list gets empty.

4.4 Case study

Image that the graph shown in Figure 4-2 represents software dependencies. You want to install the module *G*. But, to install that module you must install all the other ones before in a topological order. You need to install module *C* and *A* before installing module *D*. So in this case the topological sorting is the algorithm that we need to solve the problem.

To solve this problem programatically we only need to declare the nodes, the edges and the run the algorithm.

```
"First define the nodes and the edges"
nodes := #( $A $B $C $D $E $F $G ).
edges := #( #( $A $B ) #( $A $C ) #( $B $E ) #( $C $E ) #( $C $D )
            #( $D $E ) #( $D $F ) #( $E $G ) #( $F $G ) ).
```

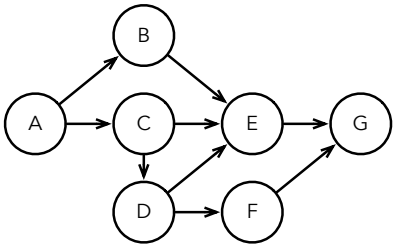

Figure 4-2 Software modules.

```
"Instantiate the graph algorithm"
topSortingAlgo := AITopologicalSorting new.
"Set the nodes and edges"
topSortingAlgo
  nodes: nodes;
  edges: edges from: #first to: #second.
"Run to obtain the result"
topologicalSortedElements := topSortingAlgo run.
```

Note that a *DAG* may have several topological orders which all of them are correct. If we look at the result we get the order in which the software dependencies need to be installed.

```
#( $A $B $C $D $E $F $G )
```

4.5 Conclusion

Topological sorting is simple algorithm working with a working list, the list where we add the next nodes to be treated. It shows that the algorithm works until the list gets empty. This is the typical structure of iterating algorithms based on working list.

CHAPTER 5

Shortest path problem

The shortest path problem consists on finding a path between two pairs of nodes in which the sum of the weights is minimized. For a general graph this problem is NP-hard. For some kind of graphs this problem can be solved in linear time.

In this chapter we will present multiple algorithms:

- A version implementing breath first traversal,
- Dijkstra's algorithm for weighted nodes,
- an algorithm for Directed Acyclic Graphs and,
- Bellman-Ford algorithm for negative weighted graphs.

5.1 Examples

Let us take this graph as an example 5-1 As it is an unweighted graph, we can calculate the distance between 2 nodes using the BFS algorithm.

But, if we add weights to the graph, as in 5-2 we cannot no longer use BFS. But we can find the shortest distance using the Dijkstra algorithm.

The Dijkstra's algorithm does not work on graphs with negative weights. So, if we add negative weights to the graph, we must use the Bellman-Ford algorithm to solve the problem. Figure 5-3

If the graph has no cycles, a Directed Acyclic Graph (DAG) (like Figure 5-3), we can use an algorithm based on topological sort to find the shortest distance. This algorithm works for both negative and positive weights as long the graph has no cycles. This algorithm is better in terms of time complexity

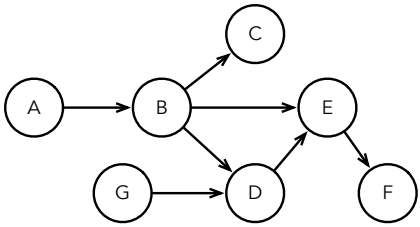

Figure 5-1 Short distance graph.

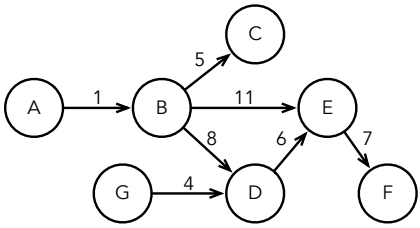

Figure 5-2 Short distance with weights.

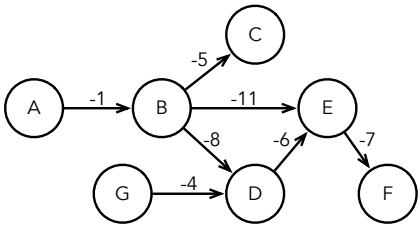

Figure 5-3 Short distance with negative weights.

but is more restricted as it only runs on DAG. On the other hand, Dijkstra's and Bellman-Ford both run in both cyclic and acyclic graphs.

5.2 Shortest path on unweighted graphs (BFS algorithm)

If the graph is unweighted or all edges have the same **non-negative** weight, the shortest path can be found in linear time $O(V + E)$ using Breadth First Search (BFS) algorithm.

BFS is an algorithm for traveling a graph in a traversal way. That means that the algorithm will travel the children of the starting node always in order ensuring that when the goal node, if exists, is founded the path will be the shortest one possible.

BFS is a single source shortest path algorithm. That means that before running the algorithm it is needed to specify a starting node. Then, the algorithm can tell us the shortest path between the starting node and all the other nodes.

The algorithm is the following one:

```
initialize a queue Q
mark start node as visited
Q.addLast(start)
while Q is not empty do:
  node := Q.removeFist()
  if node is the end then:
    return node
  for adjacent nodes of node do:
    if adjacentNode is not visited then:
      mark adjacentNode as visited
      Q.addLast(adjacentNode)
```

In the Pharo implementation we use a linked list as a queue. We use a `LinkedList` instead of an `OrderedCollection` because the `removeFirst` operation of `LinkedList` takes constant time and for an `OrderedCollection`, it takes linear time.

We define a new subclass of `AIGraphAlgorithm` named `AIBFS`. And we define a new method `run` as follows:

```
AIBFS >> run

  | node neighbours |
  queue := LinkedList with: start.
  start visited: true.

  [ queue isNotEmpty ] whileTrue: [
    node := queue removeFirst.
    neighbours := node adjacentNodes.

    neighbours do: [ :next |
      next visited ifFalse: [
        queue addLast: next.
        next visited: true.
        next previousNode: node ] ] ]
```

After running the algorithm, to reconstruct the shortest path between the start and the end node, we use the following method:

```
AIBFS >> reconstructPath

  | path previous |
  "If no path exists between the start and the end node"
  end previousNode ifNil: [ ^ #( ) ].
```

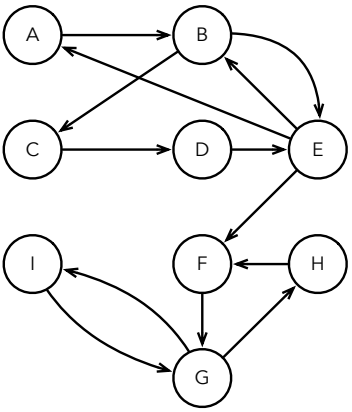

Figure 5-4 A graph for BFS.

```
  path := LinkedList empty.
  previous := end.
  path addFirst: end model.
  [ previous = start ] whileFalse: [
    previous := previous previousNode.
    path addFirst: previous model ].
  ^ path
```

5.3 Case study

For the BFS graph, the shortest path can be calculated using the class `AIBFS` like shown in the following example.

```
nodes := $a to: $i.
edges := #( #( $a $b ) #( $b $c ) #( $c $d ) #( $d $e ) #( $e $a )
            #( $b $e ) #( $e $b ) #( $e $f ) #( $f $g ) #( $g $h )
            #( $h $f ) #( $g $i ) #( $i $g ) ).
bfs := AIBFS new.
bfs
  nodes: nodes;
  edges: edges from: #first to: #second.

path := bfs runFrom: $a to: $g
```

The `path` variable contains all the nodes that are part of the path, if we inspect the variable we see:

```
#($a $b $e $f $g)
```

If we want to get the shortest path between the same starting node A and some other node, there is no need of re-running the algorithm. We only need

to change the end node and call the method reconstruct path.

```
bfs end: $d.
pathToD := bfs reconstructPath
```

5.4 Shortest path on weighted graphs (Dijkstra's algorithm)

The Dijkstra's algorithm is one of the most-know algorithms for calculating the shortest path in a weighted graph. As BFS, this algorithm is also a single source shortest path algorithm. In its naive implementation has a time complexity of $O(V^2)$. But, it can be optimized using a heap or a Fibonacci's heap as a data structure to a time complexity of $O((V + E) * logV)$. If a Fibonacci heap is used we get the best possible time complexity $O(E + V * logV)$ possible (at the moment). Dijkstra's algorithm can handle a graph with cycles. But, it cannot handle negative weights.

The algorithm idea is:

1. Mark all nodes as unvisited.
2. Assign to every node infinity as the distance value. Set it to zero for the initial. Set the initial node as current.
3. Consider all of unvisited neighbours of the current node and calculate their distances through the current node. Compare the newly calculated tentative distance to the current assigned value and assign the smaller one.
4. When all of the unvisited neighbours of the current node are checked, mark the current node as visited.
5. Select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3.

Depending on the data structure that is chosen, the time complexity will vary. Using an array is the most inefficient one because to get the most promising pair it is necessary to travel all the array $O(N)$. But with a heap, or a Fibonacci heap, the time complexity for getting the most promising pair is in the logarithmic order. Our implementation uses an array for now.

The implementation in Pharo is the following:

```
AIDijkstra >> run

  | pq |
  pq := self newPriorityQueue.
  pq add: start -> 0.

  [ pq isNotEmpty ] whileTrue: [
    | assoc node minWeight |
    assoc := self removeMostPromisingPair: pq.
```

```
    node := assoc key.
    minWeight := assoc value.
    node visited: true.

    "Skip if the path weight is less than the one obtained from the
    pq.
    This is an optimization for not processing unnecessary nodes."
    node pathDistance < minWeight ifFalse: [
      node outgoingEdges do: [ :edge |
        edge to visited ifFalse: [
          | newDistance |
          newDistance := node pathDistance + edge weight.

          newDistance < edge to pathDistance ifTrue: [
            self updateDistance: newDistance of: edge to
    previousNode: node.
            pq add: edge to -> newDistance ] ] ] ] ]
```

```
AIDijkstra >> updateDistance: newDistance of: aNode previousNode:
    previousNode

  aNode previousNode: previousNode.
  aNode pathDistance: newDistance
```

In this implementation we will use an Oredered Collection as a data structure for the priority queue.

```
AIDijkstra >> newPriorityQueue
  "This is the naive implementation of the data structure."

  ^ OrderedCollection new
```

```
AIDijkstra >> removeMostPromisingPair: aPriorityQueue
  "This is the naive implementation of the data structure."

  | minValue |
  minValue := aPriorityQueue detectMin: [ :assoc | assoc value ].
  ^ aPriorityQueue remove: minValue
```

5.5 Case study

In the graph shown in Figure 5-5, the shortest path between node A and B is `#( $A $C $B )`. The shortest path between node A and node F is `#( $A $C $B $D $E $F )`

```
nodes := $A to: $F.
edges := #( #( $A $B 5 ) #( $A $C 1 ) #( $B $C 2 ) #( $B $E 20 )
            #( $B $D 3 ) #( $C $B 3 ) #( $C $E 12 ) #( $D $C 3 )
            #( $D $E 2 ) #( $D $F 6 ) #( $E $F 1 ) ).
```

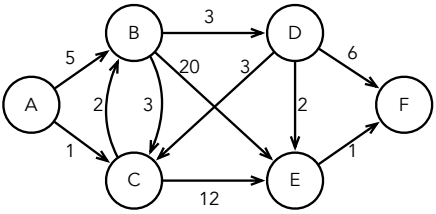

Figure 5-5 Dijkstra graph.

```
dijkstra := AIDijkstra new.
dijkstra nodes: nodes.
dijkstra
  edges: edges
  from: #first
  to: #second
  weight: #third.

shortestPathAToB := dijkstra runFrom: $A to: $B.
pathDistanceAToB := (dijkstra findNode: $B) pathDistance.

dijkstra end: $F.
shortestPathAToF := dijkstra reconstructPath.
pathDistanceAToF := (dijkstra findNode: $F) pathDistance.

dijkstra reset.
shortestPathBToE := dijkstra runFrom: $B to: $E.
```

5.6 Shortest path on Directed Acyclic Graphs (DAG)

If the graph is a directed acyclic weighted graph (DAG), we can calculate the shortest path using an algorithm based on topological sort. Using this algorithm we have a time complexity of $O(V + E)$.

This algorithm is also single source shortest path. The idea of the algorithm is to order the nodes in a topological order, using the topological sort algorithm. Then, keep track of the path weight from the start node to the other nodes. Then, start popping the nodes in order and store in a collection the ones that have the lowest path weight.

As this algorithm runs in graphs that has no cycles, it *can* handle negative weights. The pseudocode is:

1. Initialize the initial distance to every node to be infinity and the distance of the start node to be 0.
2. Create a topological order of all nodes.

3. For every node u in topological order:
   - Do following for every adjacent node v of u
   - IF (v pathWeight > u pathWeight + weight(u, v)) THEN v pathWeight: u pathWeight + weight(u, v)

5.7 DAG shortest path implementation

The Pharo implementation is as follows.

```
AIShortestPathInDAG >> initializePathWeights

  nodes do: [ :node | node pathWeight: Float infinity ].
  start pathWeight: 0
```

```
AIShortestPathInDAG >> run

  | topSorter stack sortedNode |
  self initializePathWeights.
  topSorter := AITopologicalSorting new
    addNodesFromDifferentGraph: nodes;
    yourself.
  topSorter run.
  stack := topSorter topologicalSortedElements.

  [ stack isNotEmpty ] whileTrue: [
    sortedNode := self findNode: stack removeFirst.
    sortedNode outgoingEdges do: [ :nextEdge |
      nextEdge to pathWeight >
          (sortedNode pathWeight + nextEdge weight)
        ifTrue: [
          nextEdge to pathWeight: sortedNode pathWeight +
            nextEdge weight.
          nextEdge to previousNode: sortedNode ] ] ]
```

5.8 DAG shortest path refactored

Now we are ready to refactor our code.

```
AIShortestPathInDAG >> run

  | stack sortedNode |
  self initializePathWeights.
  stack := self topologicalSortedNodes.

  [ stack isNotEmpty ] whileTrue: [
    sortedNode := self findNode: stack removeFirst.
```

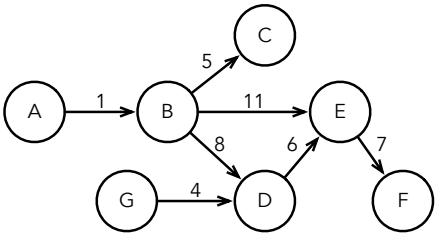

Figure 5-6 DAG with weigthed paths.

```
    sortedNode outgoingEdges do: [ :nextEdge |
      nextEdge to pathDistance >
      (sortedNode pathDistance + nextEdge weight)
        ifTrue: [
          self updatePathDistance: nextEdge previousNode: sortedNode
    ] ] ]
```

```
AIShortestPathInDAG >> topologicalSortedNodes

  | topSorter |
  topSorter := AITopologicalSorting new
    addNodesFromDifferentGraph: nodes;
    yourself.
   topSorter run.
   ^ topSorter topologicalSortedElements.
```

```
AIShortestPathInDAG >> updatePathDistance: edge previousNode:
    previousNode

  edge to pathDistance: previousNode pathDistance + edge weight.
  edge to previousNode: previousNode
```

Case study

On this weighted DAG (See Figure 5-6), the following snippet calculates the shortest path between node A and node F.

```
nodes := $A to: $G.
edges := #( #( $A $B 1 ) #( $B $C 5 ) #( $B $E 11 ) #( $B $D 8 )
            #( $D $E 6 ) #( $E $F 7 ) #( $G $D 4 ) ).
shortestPathInDAG nodes: nodes.
shortestPathInDAG
  edges: edges
  from: #first
  to: #second
  weight: #third

pathAtoF := shortestPathInDAG runFrom: $A to: $F.
```

5.9 Shortest path on weighted graphs with negative weights (Bellman-Ford algorithm)

In the Dijkstra's algorithm when a node is marked as visited, the algorithm already found the best distance to it, because adding any positive numbers will only increase the path distance. When we are dealing with negative numbers the assumption is not true.

The Bellman-Ford algorithm can handle negative weighted graphs. It runs in $O(V * E)$ time. As the other algorithms of this chapter, this is also a single source shortest path algorithm. The logic behind the algorithm is to perform at worst $V - 1$ times an edge relaxation. Relaxing an edge means to update the value of the distance from the starting node to the node to which to edge goes. Then, run the algorithm one more time, if an edge can reduce its distance (be relaxed) means that the node to which the edge goes is part of a negative cycle.

The algorithm works as follows:

1. Set the distance to every node to be infinity
2. Set the distance to the starting node to be 0
3. Perform V-1 times the edge relaxation
4. Run another V-1 times the edge relaxation, if an edge can be still relaxed means that is part of a negative cycle.

5.10 Pharo implementation

The Pharo implementation is:

```
AIBellmanFord >> run

  start pathDistance: 0.
  self relaxEdges.
  "Run the algorithm one more time to detect if there is any
    negative cycles.
  The variation is if we can relax one more time an edge,
  means that the edge is part of a negative cycle.
  So, we put negative infinity as the path distance"
  self relaxEdgesToNegativeInfinity
```

```
AIBellmanFord >> relaxEdges

  | anEdgeHasBeenRelaxed |
  "Relax the edges V-1 times at worst case"
  nodes size - 1 timesRepeat: [
    anEdgeHasBeenRelaxed := false.
```

```
    edges do: [ :edge |
      edge from pathDistance + edge weight < edge to pathDistance
    ifTrue: [
        edge to pathDistance: edge from pathDistance + edge weight.
        edge to previousNode: edge from.
        anEdgeHasBeenRelaxed := true ] ].

    "If no edge has been relaxed means that we can stop the
    iteration before V-1 times"
    anEdgeHasBeenRelaxed ifFalse: [ ^ self ] ]
```

```
AIBellmanFord >> relaxEdgesToNegativeInfinity
  "This method is called after a first relaxation has occurred
    already.
  The algorithm is the same as the previous one but with the only
    difference that now if an edge can be relaxed we set the path
    distance
  as negative infinity because means that the edge is part of a
    negative cycle."

  | anEdgeHasBeenRelaxed |
  "Relax the edges V-1 times at worst case"
  nodes size - 1 timesRepeat: [
    anEdgeHasBeenRelaxed := false.

    edges do: [ :edge |
      edge from pathDistance + edge weight < edge to pathDistance
    ifTrue: [
        edge to pathDistance: Float negativeInfinity.
        anEdgeHasBeenRelaxed := true ] ].

    "If no edge has been relaxed means that we can stop the
    iteration before V-1 times"
    anEdgeHasBeenRelaxed ifFalse: [ ^ self ] ]
```

5.11 Longest path problem

To calculate the longest path of a graph we can simply multiply all the nodes weights by -1 and then calculate the shortest path. If the graph is a DAG, then we can use the topological sort based algorithm. If not, we can use Bellman-Ford.

Case study

We multiply by -1 the weight of the edges of the previous graph used as an example for the DAG algorithm and then we calculate the shortest path between two nodes using the Bellman-Ford algorithm. Doing that, actually we

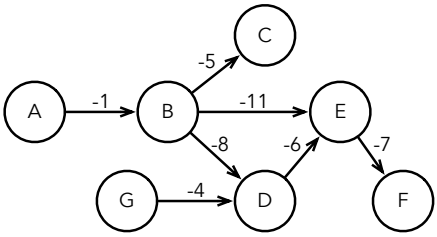

Figure 5-7 A negative weighted graph used for experimenting with Bellman-Ford algorithm.

are calculating the longest path for the original graph.

```
bellmanFord := AIBellmanFord new.
nodes := $A to: $F.
edges := #( #($A $B -1) #($B $C -5) #($B $E -11)
            #($B $D -8) #($E $F -7) #($D $E -6)
            #($G $D -4) ).
bellmanFord nodes: nodes.
bellmanFord
  edges: edges
  from: #first
  to: #second
  weight: #third.

pathFromAtoF := bellmanFord runFrom: $A to: $F.
pathDistanceFromAToF := (bellmanFord findNode: $F) pathDistance
```

If we look at the path between `A` and `F`, we see `#( $A $B $D $E $F )` which is actually the longest path of the original graph.

5.12 Conclusion

In this chapter we saw some of the most-know algorithms for calculating the shortest (and longest) distance on graphs. Calculate the shortest path, or distance, in a graph is not a trivial problem and it is a NP-Hard problem. For some types of graphs, like trees or DAG (Directed Acyclic Graphs), we can solve the problem in linear time.

CHAPTER 6

Minimum spanning trees

The minimum spanning tree is a subset of the edges of a undirected weighted graph that connects all the nodes of the graph without any cycles and with the total sum of the weights minimized. There are several algorithms for obtaining the minimum spanning tree of a weighted graph, the most famous is the Kruskal's algorithm. In the case of an disconnected graph, the algorithm returns a minimum spanning forest (i.e., it will return a list a minimum spanning trees)

We will show you how to implement Kruskal but before that we have to introduce a little data-structure called *Disjoint-Set*.

6.1 Motivating scenario

Imagine that you have a telecommunication company and you want to build a connection between different neighbourhoods. Some of the connections are more expensive than others. For example, one connection has to pass under the ground or above some mountains. So, you have a graph in which the nodes represent the different neighbourhoods and the edges represent all the possible cables that can be built to make the connections between the neighbourhoods. The weights represent the cost of actually building the connection.

Imagine that we have the graph shown in Figure 6-2, we would like to get the tree that allows us to get all the nodes of the graph without circle as shown in Figure 6-2.

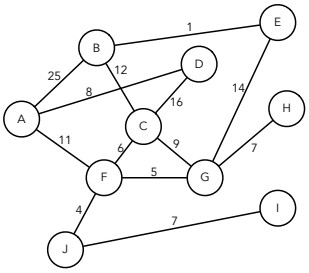

Figure 6-1 Connections costs between neighbourhoods.

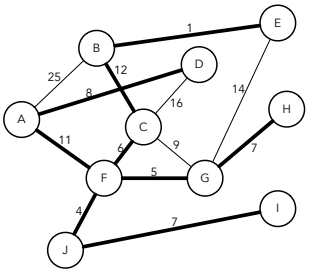

Figure 6-2 Minimum spanning tree with C as root.

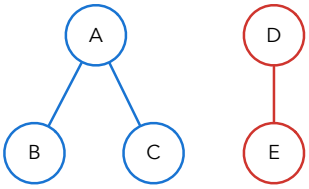

Figure 6-3 Two Union-Find sets.

6.2 Disjoint-Set data structure

A disjoint-set, also called union-find data structure, is a data structure that stores disjoint sets. It provides two operations:

- *unite* that groups two disjoint sets into one and
- *find* that returns two elements belong to the same disjoint set.

For example, in Figure 6-3 we have two sets of elements $\{A, B, C\}$ and $\{D, E\}$. If we call the operation find with A and D nodes, as they do not belong to the same set, the operation will return false. With A and B nodes, the operation find will return true.

But when we invoke the operation unite with A and D, it will join the two sets

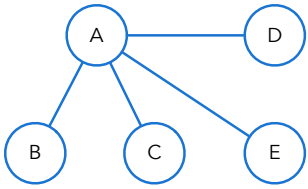

Figure 6-4 Union-Find set: result of the unite operation between A and D.

to have only one set with all the elements, as in Figure 6-4.

This data structure is used in Kruskal's algorithm to detect if adding a new edge creates a cycle in the minimum spanning tree that is being built.

The time complexity of both of the operations is $O(a(n))$, where a is the amortized time complexity. Each time that the `find:` method is invoked an operation called *path compression*. This is due to the path compression operation that this data structure has an amortized linear time complexity.

In Pharo, this data structure represents a node in the Kruskal's graph algorithm.

```
AIDisjointSetNode >> union: aDSNode

  | root1 root2 |
  root1 := aDSNode find.
  root2 := self find.

  root1 = root2 ifTrue: [
    "The nodes already belong to the same component"
    ^ self ].

  root1 parent: root2
```

```
AIDisjointSetNode >> find
  "Return the root of the component but modifying the parent/child
    structure during the process of finding a root."

  | root next node |
  node := self.
  root := node.
  [ root = root parent ] whileFalse: [ root := root parent ].

  "Compress the path leading back to the root.
  This is the path compression operation that gives the linear
    amortized time complexity"
  [ node = root ] whileFalse: [
    next := node parent.
    node parent: root.
    node := next ].
```

```
  ^ root
```

6.3 Kruskal's algorithm

As said above, the Kruskal's algorithm calculates the minimum spanning tree (or forest) of an undirected weighted graph. The algorithm has a time complexity of $O(V * log(E)) = O(E * log(E))$. This time complexity is achieved thanks to the Disjoint-Set data structure. This algorithm uses the Disjoint-Set data structure to check if adding an edge to the spanning tree creates a cycle.

The pseudocode is:

```
1. Sort edges in ascending weight.
2. Pick the smallest edge.
    Check if its two nodes are already unified.
      If they are not, unified them and include the edge to the
    spanning tree.
      Else, discard it.
3. Repeat step 2 until there are all nodes are connected.
```

This is the implementation of the algorithm in Pharo:

```
AIKruskal >> run

  | treeEdges sortedEdges |
  sortBlock := [ :e1 :e2 | e1 weight < e2 weight ].
  treeEdges := OrderedCollection new.
  nodes do: [ :node | node makeSet].
  sortedEdges := edges asSortedCollection: sortBlock.
  sortedEdges
    reject: [ :edge |
      "Only join the two nodes if they don't belong to the same
    component"
      edge from find = edge to find ]
    thenDo: [ :edge |
      edge from union: edge to.
      treeEdges add: edge ].
  ^ treeEdges
```

6.4 Kruskal's algorithm for maximum spanning tree

As contrary to the minimum spanning tree, the maximum spanning tree of a graph is a subset of edges of a graph that connects all nodes with the **maximum** possible distance.

This is exactly the same algorithm except that we have to order the edges in descending weight instead of ascending.

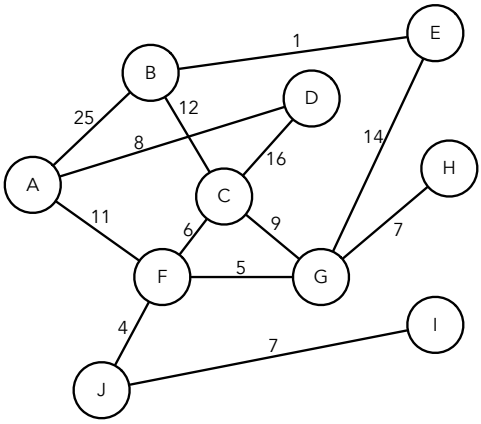

Figure 6-5 Connections costs between neighbourhoods.

```
1. Sort edges in descending weight.
2. Pick the biggest edge...
```

In the implementation we only need to change one line:

```
sortBlock := [ :e1 :e2 | e1 weight > e2 weight ].
```

6.5 Case study

We can now apply our algorithm to the graph shown at the beginning of this chapter in Figure 6-5.

So, like in the other graph algorithms we only need to declare the nodes and the edges an then call the method `run` to obtain the result.

```
nodes := $A to: $J.
edges := #( #( $A $B 25 ) #( $A $D 8 ) #( $A $F 11 ) #( $B $A 25 )
            #( $B $E 1 ) #( $B $C 12 ) #( $C $B 12 ) #( $C $D 16 )
            #( $C $F 6 ) #( $C $G 9 ) #( $D $A 8 ) #( $D $C 16 )
            #( $E $B 1 ) #( $E $G 14 ) #( $F $A 11 ) #( $F $C 6 )
            #( $F $G 5 ) #( $F $J 4 ) #( $G $F 5 ) #( $G $C 9 )
            #( $G $E 14 ) #( $G $H 7 ) #( $H $G 7 ) #( $I $J 7 )
            #( $J $F 4 ) #( $J $I 7 ) ).
kruskal := AIKruskal new.
kruskal nodes: nodes.
kruskal
  edges: edges
  from: #first
  to: #second
  weight: #third.
minimumSpanningTree := kruskal run
```

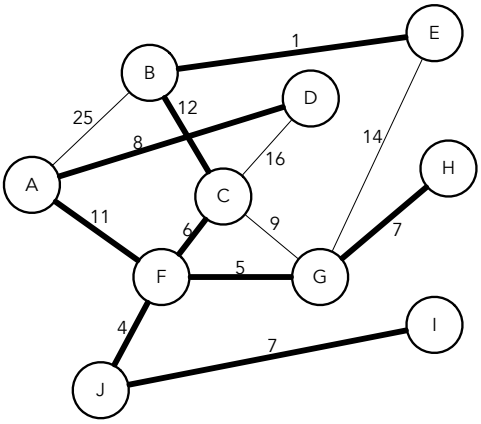

Figure 6-6 Minimum spanning tree.

If we inspect the `minimumSpanningTree` variable, we get a collection the edges of the minimum spanning tree. *DSN* means *DisjointSetNode*.

```
DSN $B -> DSN $E weight: 1
DSN $J -> DSN $F weight: 4
DSN $F -> DSN $G weight: 5
DSN $F -> DSN $C weight: 6
DSN $I -> DSN $J weight: 7
DSN $H -> DSN $G weight: 7
DSN $A -> DSN $D weight: 8
DSN $A -> DSN $F weight: 11
DSN $C -> DSN $B weight: 12
```

If we want to obtain the maximum spanning tree, we only need to call the `maxSpanningTree` method when creating the graph algorithm.

```
kruskal := AIKruskal new maxSpanningTree.
```

6.6 Conclusion

Data structures play a powerful role when it comes to algorithms. In this specific case thank to the Disjoint-Set data structure we can detect cycles in amortized linear time complexity with few lines of code. Also, the Kruskal algorithm has many real life applications and it is an important algorithm in the context of graph theory.

CHAPTER 7

Strongly Connected Components in a Graph

A graph is strongly connected if every of its nodes are reachable from every other node. That means, that from all nodes, you can reach any node of the graph. The strongly connected component of a graph if the **maximum** subset which itself forms also a strongly connected graph.

The most knew algorithms for finding the strongly connected components of a graph are: Tarjan's and Kosaraju's algorithms. Both algorithms have a time complexity of $O(V + E)$ and are based on DFS (depth search first). But Tarjan's algorithm is faster on practice. Because Kosaraju's algorithm does two passes of DFS and Tarjan's only one.

7.1 Motivating example

Finding strongly connected components in a graph has several real life applications. For example it is used on social media to find groups of friends to suggest commonly liked pages (see Figure 7-1).

In Figure 7-1 we have 5 different strongly connected components that are highlighted with colors.

7.2 Tarjan's algorithm

To find the strongly connected components of a graph, Tarjan's algorithm assigns a low-link value and an ID to each node. At the beginning, the low-link value is the same as the node ID. Then, as the algorithm is running it

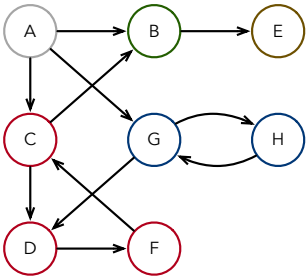

Figure 7-1 A social network: each node is a person and an edge a connection.

updates the low-link value to be the smallest index of any node known that is reachable. It does a DFS pass to all the node to update low-link value. If at the end of the DFS call the low link value of a node is the same as its ID, means that that node is the beginning of a strongly connect component. The pseudocode is:

```
1. Mark the as unvisited and without an ID nor a low-link value.
2. Start DFS. When visiting a node assign it an ID and a low-link
    value same as the ID.
3. Mark current node as visited and add it to the stack.
4. On DFS callback,
    First, min the current node's low-link value with the low-link
    value of the adjacent node.
    Then, if the adjacent node is on the stack
              then min the current node's low-link with the adjacent
    node's ID.
5. After visiting all adjacent nodes, if the current node has its ID
    value as the same of its low-link value
    Then it means that there is a strongly connected component.
    So, pop all nodes from the stack until current node is reached.
```

The DFS callback is when we are going back from the recursion.

7.3 Tarjan's implementation

In the Pharo implementation a `traverse:` method is used. This is the DFS call. All the magic happens in that method.

```
AITarjan >> run
  "Initialize an empty array for the strongly connected components"

  sccs := OrderedCollection new.
  stack := Stack new.
  runningIndex := 0.
```

```
  nodes do: [ :node |
    node isTarjanUndefined ifTrue: [
      "If the node has no low-link value set make a dfs call"
      self traverse: node ] ].
  ^ self stronglyConnectedComponents
```

```
AITarjan >> traverse: aTarjanNode

  aTarjanNode tarjanIndex: runningIndex.
  aTarjanNode tarjanLowlink: runningIndex.
  runningIndex := runningIndex + 1.

  self putOnStack: aTarjanNode.

  aTarjanNode adjacentNodes do: [ :adjacentNode |
    adjacentNode isTarjanUndefined
      ifTrue: [
        "If the adjacent node doesn't have a low link"
        self traverse: adjacentNode.
        aTarjanNode tarjanLowlink:
          (aTarjanNode tarjanLowlink
            min: adjacentNode tarjanLowlink) ]
      ifFalse: [
        "If the adjacent node had already a low link value"
        adjacentNode inStack ifTrue: [
          aTarjanNode tarjanLowlink:
            (aTarjanNode tarjanLowlink
              min: adjacentNode tarjanIndex) ] ] ].

  "If the node is the beginning of a strongly connected component"
  (self isRootNode: aTarjanNode) ifTrue: [
    self addNewSccForNode:: aTarjanNode ]
```

```
AITarjan >> putOnStack: aTarjanNode

  stack push: aTarjanNode.
  aTarjanNode inStack: true
```

```
AITarjan >> addNewSccForNode: aTarjanNode

  | currentNode stronglyConnectedComponent |
  stronglyConnectedComponent := OrderedCollection empty.

  [ currentNode := stack pop.
  currentNode inStack: false.
  stronglyConnectedComponent add: currentNode ]
    doWhileFalse: [ currentNode = aTarjanNode ].

  sccs add: stronglyConnectedComponent.
  stronglyConnectedComponent do: [ :each |
    each cycleNodes: stronglyConnectedComponent ]
```

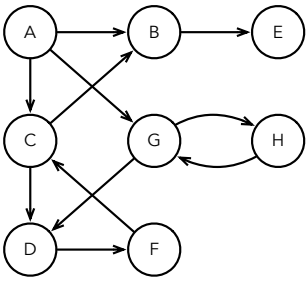

Figure 7-2 A social network: each node is a person and an edge a connection.

The method `stronglyConnectedComponents` returns a list of group of elements. Each group represents a strongly connected component.

```
AITarjan >> stronglyConnectedComponents

  sccs ifNil: [ self run ].
  ^ sccs collect: [ :component |
    component collect: [ :each | each model ] ]
```

7.4 Case study

The graph in Figure 7-2 represents a set of connected people on a social media. An edge represents a follow. One person can follow and can be followed. We want to know which and how many strongly connected components the social network has.

The code for solving the problem is similar to the other algorithms. We instantiate the nodes, the edges and call the method `run`.

```
nodes := $a to: $h.
edges := #( #( $a $b ) #( $a $c ) #( $a $g ) #( $b $e ) #( $c $b )
            #( $c $d ) #( $d $f ) #( $f $c ) #( $g $h ) #( $g $d )
            #( $h $g ) ).
tarjan := AITarjan new.
tarjan
  nodes: nodes;
  edges: edges from: #first to: #second.
stronglyConnectedComponents := tarjan run
```

If we inspect the `stronglyConnectedComponents` variable we see that that is a collection that contains 5 elements. Each element is a list that contains the nodes corresponding to the strongly connected component.

```
an OrderedCollection($e)
an OrderedCollection($b)
an OrderedCollection($c $f $d)
an OrderedCollection($h $g)
an OrderedCollection($a)
```

So, our graph has 5 strongly connected components, which are: $\{A\}$, $\{B\}$, $\{E\}$, $\{G, H\}$, $\{C, D, F\}$

7.5 Reducing a Graph

If we want to collapse all strongly connected components of a graph into a single one, we can use the Tarjan's algorithm to help in the task. Note that this algorithm for reducing a graph does not work on weighted graphs.

This is useful when we want to treat all the strongly connected components as one node. For example in a telecommunication network it can be useful for simplifying the analysis of costs.

To do that:

1. Find circuits using Tarjan's algorithm (strongly connected components which size is > 1).
2. Merge all nodes in circuit into one collapsed node.
3. Remove the nodes that were merged.
4. Add the new collapsed nodes.
5. Replace the old references to the merged nodes to reference the new collapsed nodes.

7.6 Case study

We want to reduce the same graph collapsing all strongly connected components into one node.

```
nodes := $a to: $h.
edges := #( #( $a $b ) #( $a $c ) #( $a $g ) #( $b $e ) #( $c $b )
            #( $c $d ) #( $d $f ) #( $f $c ) #( $g $h ) #( $g $d )
            #( $h $g ) ).
graphReducer := AIGraphReducer new.
graphReducer
  nodes: nodes;
  edges: edges from: #first to: #second.
reducedGraph := graphReducer run
```

`reducedGraph` is a collection that contains all the new nodes of the graph. We can see that now there is only 5 nodes (because the graph contained 5

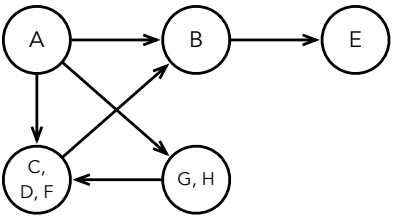

Figure 7-3 Reduced graph: a collection containing all the new nodes.

strongly connected components). Also, you can inspect the collapsed nodes and you will see the new adjacencies.

```
Merged nodes: $a
Merged nodes: $b
Merged nodes: $e
Merged nodes: $h, $g
Merged nodes: $c, $f, $d
```

7.7 Conclusion

Although Tarjan's algorithm can be a bit complicated to understand a first sight, once we understand the logic behind the updating the low-link values it gets clearer. The strongly connected components can represent several things in real life and the Tarjan algorithm is very useful because it runs on linear time.

CHAPTER 8

Link analysis

Link analysis is a technique used to evaluate relationships between nodes. Link analysis is used on several fields, such as search engines, fraud detection, among others. There is several algorithms of different kinds to perform link analysis. Here we are only going to focus on the Hyperlink-Induced Topic Search (HITS) algorithm.

This algorithm was originally developed to rate web pages. But, nowadays modern search engines do not use this algorithm since there is more advanced techniques. HITS has been also used to identify the important classes that should be commented in a large software system or the classes that a developer should read to get an insight of the key classes.

8.1 Hyperlink-Induced Topic Search (HITS) algorithm

Hyperlink-Induced Topic Search (HITS) algorithm, also knows as Hubs and Authorities, is an algorithm that rates every the nodes of a graph. Every node has a hub and a authority score. A hub is a node that may not be relevant but references relevant nodes. An authority is a node that contains relevant information.

The algorithm does the following:

1. Assign to each node a hub and an authority score equal to 1.
2. Run the authority update rule for each node.
3. Run the hub update rule for each.
4. Normalize the values by dividing each Hub score by the square root of the sum of the squares of all Hub scores, and dividing each Author-

ity score by the square root of the sum of the squares of all Authority scores.

5. Repeat from the second step as necessary.

The update rules are simple:

Authority update rule Update each node's authority score to be equal to the sum of the hub scores of each node that points to it.

Hub update rule Update each node's hub score to be equal to the sum of the authority scores of each node that it points to.

8.2 HITS implementation

The Pharo implementation is as follows. The `k` number is the number of times that the scores are going to be updated. The default value is `20` but it can also be set manually.

```
AIHits >> run

  self initializeNodes.
  k timesRepeat: [
    nodes do: [ :node | self computeAuthoritiesFor: node ].
    nodes do: [ :node | self computeHubsFor: node ].
    self normalizeScores ].
  ^ nodes
```

```
AIHits >> initializeNodes

  "Here we are using float instead of int because of the
    normalization."
  nodes do: [ :n |
    n auth: 1.0.
    n hub: 1.0 ]
```

```
AIHits >> computeAuthoritiesFor: aNode

  aNode auth:
    (aNode incomingNodes
      inject: 0
      into: [ :sum :node | sum + node hub ])
```

```
AIHits >> computeHubsFor: aNode

  aNode hub:
    (aNode adjacentNodes
      inject: 0
      into: [ :sum :node | sum + node auth ])
```

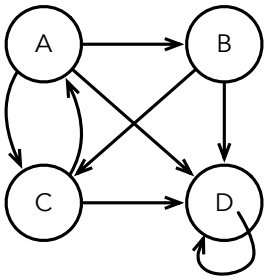

Figure 8-1 A graph to play with the HITS algorithm.

```
AIHits >> normalizeScores

  | authNorm hubNorm |
  authNorm := 0.
  hubNorm := 0.

  nodes do: [ :node |
    authNorm := authNorm + node auth squared.
    hubNorm := hubNorm + node hub squared ].

  authNorm := authNorm sqrt.
  hubNorm := hubNorm sqrt.

  "To avoid dividing by 0"
  authNorm = 0 ifTrue: [ authNorm := 1.0 ].
  hubNorm = 0 ifTrue: [ hubNorm := 1.0 ].

  nodes do: [ :n |
    n auth: n auth / authNorm.
    n hub: n hub / hubNorm ]
```

8.3 Case study

Here we calculate the hubs and authorities scores for all the nodes of the graph shown in Figure 8-1 with 3 iterations.

```
nodes := #( 'A' 'B' 'C' 'D' ).
edges := #( #( 'A' 'B' ) #( 'A' 'C' ) #( 'A' 'D' ) #( 'B' 'C' )
            #( 'B' 'D' ) #( 'C' 'A' ) #( 'C' 'D' ) #( 'D' 'D' ) ).
hits := AIHits new.
hits
  nodes: nodes;
  edges: edges from: #first to: #second;
  k: 3.
nodes := hits run
```

If we inspect the nodes, these are the scores calculated after 3 iterations.

```
('A' auth: 0.17 hub: 0.65)
('B' auth: 0.27 hub: 0.54)
('C' auth: 0.49 hub: 0.41)
('D' auth: 0.81 hub: 0.34)
```

8.4 Weighted HITS

There are cases where the Hits algorithm does not behave as expected and sometimes the Hits algorithm puts 0 as values for the hubs and authorities. Using weights in a graph helps in obtaining better results. Establishing the weights is a responsibility of the user.

For more information, you can read these papers:

- *Modifications of Kleinberg's HITS Algorithm Using Matrix Exponentiation and Web Log Records* by Miller et al.
- *An Improved Weighted HITS Algorithm Based on Similarity andPopularity* by Zhang et al.

In terms of implementation, it is only necessary to multiply the weights with the scores in each iteration. That means changing `computeAuthorities-For:` and `computeHubsFor:` methods. This is done in `AIWeightedHits` class.

```
AIWeightedHits >> computeAuthoritiesFor: aNode

  aNode auth: (aNode incomingEdges
       inject: 0
       into: [ :sum :edge | sum + (edge weight * edge from hub) ])
```

```
AIWeightedHits >> computeHubsFor: aNode

  aNode hub: (aNode outgoingEdges
       inject: 0
       into: [ :sum :edge | sum + (edge weight * edge to auth) ])
```

8.5 Conclusion

Even if the HITS algorithm is not used anymore in the modern search engines, it is a very good algorithm for having a first look on how to classify links according to their relevance in the network.