Graphs

A graph is a type of data structure capable of handling networks. Graphs are widely used across various domains such as the following:

Transportation : To find the shortest routes to travel between two places Communication-signaling networks : To optimize the network of inter- connected computers and systems

Understanding relationships : To build relationship trees across families or organizations

Hydrology : To perform flow regime simulation analysis of various fluids

This current chapter will build fundamentals for graphs and the following topics will be covered:

Terminology and representations Graph implementations Graph traversals Depth-first search Breadth-first search Topological sort

Shortest-paths problems

Single-source shortest paths Minimum-cost spanning trees

Prim's algorithm

Kruskal's algorithm

Page 205

﻿

Graphs

Terminology and representations

A graph (G) is a network of vertices (V) interconnected using a set of edges (E). Let |V| represent the count of vertices and |E| represent the count of edges. The value of |E| lies in the range of 0 to |V| 2 – |V| . Based on the directional edges, the graphs are classified as directed or undirected. In directed graphs , the edges are directed from one vertex towards the other, whereas in undirected graphs, each vertex has an equal probability of being directionally connected with the others. An undirected graph is said to be connected if all the vertices are connected with at least one edge. If the vertices are indexed, then it is said to be a labeled graph, and if the edges are associated with some value (cost or weights), then it is said to be a weighted graph. Adjacent vertices (P and Q) connected by an edge are termed as neighbors (P, Q), and the connecting edge is termed as an incident. Figure 8.1 represents undirected, directed, and labeled (with weights) graphs.

Figure 8.1: Representation of an undirected graph, directed graph, and labeled (directed) graph (from left to right)

Consider a graph with n vertices. A sequence of interconnected vertices (v 1 , v 2 , v 3 … v n ) is termed as a path, and the path is said to be simple if all the vertices of the path are unique. The length of the path is the number of edges, which is one less than the number of vertices ( n-1 ). In case the vertices of a given path are not unique and the length of the path is greater than two, then the path becomes a cycle. A cycle is simple if all the intermediate vertices are unique and only the first and last vertices are same. An undirected graph with no cycles is called an acyclic graph, and a directed graph with no cycles is called a directed acyclic graph ( DAG ).

A graph can be further split into multiple subgraphs (S) as shown in Figure 8.2 .

Figure 8.2: A graph split into three sub-graphs. Even a single vertex forms a graph

[ 194 ]

Page 206

﻿

Chapter 8

A free tree is a form of a connected undirected graph with no cycles in simple form. It has |V|-1 number of edges.

Consider a graph (G) with n number of vertices, which can be represented in two forms. These forms can be directly used to perform mathematical computations: Adjacency matrix : The adjacency matrix is an n×n array, with rows representing the from vertices and columns representing to vertices. The numbers in the matrix can either denote the presence of a directed connection between two vertices, or can indicate the weight (or distance) associated with the edge

connecting the two vertices. As each position in the adjacency matrix can take in a numeric value, it requires one bit of memory. Thus, the asymptote of total memory requirement is θ(|V| 2 ) .

Adjacency list : As the name suggests, the adjacency list is an array of linked lists of length n . Each position in the array stores the pointers connecting linked lists of its adjacent connected vertices, and each linked list stores the value of its connecting edge. Unlike an adjacency matrix, the memory requirement of adjacency lists depends on both, the number of vertices ( |V| ) and the number of edges ( |E| ). The array takes into account vertex memory requirement, and the list takes in the account of edge memory requirement. Thus, the asymptote of total memory requirement is θ(|V| + |E|) .

The aforementioned two forms of representation can be used to transform and store both directed and undirected graphs. Also, based on the number of interconnecting edges, the graphs can be termed as sparse or dense. A graph with a relatively less number of edges is termed as sparse, whereas a graph with a relatively large number of edges is termed as dense. Also, in case of all interconnected vertices, the graph is termed as complete (special case of a dense graph).

Assume two vertices, P and Q. If P and Q belong to a directed graph with P pointed towards Q, then in case of an adjacency matrix, only the position of P (in row) and Q (in column) would be filled with the edge value, and the position of Q (in row) and P (in column) would be left blank. Similarly, in case of adjacency lists, the array will have both the vertices P and Q, but the edge value will be assigned only to the linked list of P, which stores a pointer towards Q. If P and Q belong to an undirected graph, then both the positions of P and Q will be filled with the edge value in case of an adjacency matrix, and both the edge values will be assigned to both the linked lists of P and Q in case of adjacency lists. Figure 8.3 elucidates both directed and undirected graphs along with their respective adjacency matrices and adjacency lists.

[ 195 ]

Page 207

﻿

Graphs

In an adjacency matrix, one denotes the presence of a directed connection (from row to column) and zero represents no connection.

Figure 8.3: Representation of graphs using an adjacency matrix and adjacency lists. Part 1 shows a directed graph and Part 2 shows an undirected graph. Part 3 and 5 shows an

adjacency matrix and adjacency list for directed graph. Parts 4 and 6 shows an adjacency matrix and adjacency list for an undirected graph.

[ 196 ]

Page 208

﻿

Chapter 8

Now let us analyze the memory efficiency of an adjacency matrix and adjacency list. The first differentiating factor is the number of edges in the graph. The matrix formulation of the adjacency matrix requires memory for each possible edge irrespective of its existence, whereas the linked list formulation of adjacency lists stores only those edges that are present in the graph. On the contrary, adjacency lists require additional memory for storing pointers, which can sometimes be relatively costlier. The cost primarily depends on the value of the edge (costlier in case of a binary flag indicating the mere existence of a connection). Thus, an adjacency matrix is relatively more efficient in case of denser graphs, as it has more number of edges (thereby more edge values), and an adjacency list is relatively more efficient in case of sparse graphs, as it has less number of edges (thereby less pointers to be stored).

The second differentiating factor is the number of computations required to complete a single iteration. Asymptotically, an adjacency matrix is relatively costlier than adjacency lists. In case of an adjacency matrix, all the positions within the matrix need to be scanned prior to the completion of the iteration, whereas in case of adjacency lists, only the linked connections (using pointers) are scanned, thereby reducing unnecessary lookups. Thus, the system runtime of an adjacency matrix is θ(|V| 2 ) and that of adjacency lists is θ(|V| + |E|) . Thus, the performance of an adjacency matrix is relatively poorer in case of sparse graphs, and is almost comparable to adjacency lists in case of dense graphs.

Graph implementations

Let us create an ADT ( Graph\_ADT ) for the implementation of functions on a given graph. The key features of ADT for a given graph analysis are the following:

Fixed number of vertices

Provision for addition and deletion of edges

Provision to support a mark array, which can assist algorithms in traversing along the graph

The vertices are denoted using non-zero integer values, and can additionally store vertex names or some kind of application-based predetermined values.

[ 197 ]

Page 209

﻿

Graphs

The following are some ADT functions that are widely used for implementing graph functions:

num\_vert : This function returns the number of vertices for a given graph. num\_edge : This function returns the number of edges for a given graph. weightEdge : This function returns the weight of an edge connecting two adjacent vertices. Its input is a pair of two connected vertices and its output is a numeric value indicating its weight.

assignEdge : This function is used to assign weight to a given edge of a graph. The input is a pair of vertices. It can take in only a non-zero positive value, as a zero value means no connection (thereby no assignment required) and a negative value can skew the computational results.

deleteEdge : This function is used to delete the weight of a given edge. The input is a pair of vertices, which has a connected edge.

firstVertex : This function returns the index of the first edge vertex based on a sorted list of vertices, which are connected to a given vertex. The input is a vertex for a given graph.

nextVertex : This function returns the subsequent index of vertices for a given pair of connected vertices such that the returned vertex will have an edge connecting to the first vertex. Assume that V1 is connected with V2 and V3 such that the index values of V2 are less than V3. Then, the firstVertex function will return the edge vertex of V1 as V2 (as the index value of V2 is less than V3), and the nextVertex function will return V3, as it is a subsequent connected vertex index of V1 for a given V2.

isEdge : This function returns a Boolean number, where 1 represents the presence of an edge, and 0 represents the absence of an edge.

getMark : This function returns the mark of a given vertex from an array mark . initMark : This function marks the unmarked vertex in an array mark .

Each graph algorithm needs to traverse every vertex before it can culminate its execution. The functions firstEdge and nextVertex facilitate such kind of traversing across the graph. It is generally implemented using loops, wherein each vertex searches for all its linked vertices and then obtains their corresponding edge weights.

The following R code implements graph ADT. It takes a number of vertices n as input:

Graph\_ADT <- setRefClass(Class = "adjacency\_Matrix",

fields = list(n = "integer"), methods = list(

## Initialise a graph of n vertices Initialize = function(n){},

[ 198 ]

Page 210

﻿

Chapter 8

## Return number of vertices and edges num\_vert = function(){}, num\_edges = function(){},

## Return weight of an edge for a pair of vertices v1 and v2 weightEdge = function(v1,v2){},

## Assign weight(wt) of an edge for a pair of vertices v1 and v2

assignEdge = function(v1,v2,wt){},

## Delete weight of an edge for a pair of vertices v1 and v2 deleteEdge = function(v1,v2){},

## Return first connecting vertex for a given vertex v firstVertex = function(v){},

# Return next vertex for a given v and its neighbor w nextVertex = function(v,w){},

## Check for presence of an edge for a pair of vertices v1 and v2

isEdge = function(v1,v2){}

))

The GraphADT function can be implemented using either adjacency matrix or adjacency list representations. In this chapter, a sample implementation of graph ADT is shown using both an adjacency list and adjacency matrix; however, the creation of a graph object has not been addressed. In lieu of the graph object, the assignEdge function can be used to build graphs based on edges.

In the adjacency matrix implementation, a list, mark , stores the output of the setMark function, and the getMark function can be used to extract the mark of a given vertex. The edge matrix mat is an n×n -dimensional array of integers, which store the weights of edges. The rows represent the from vertices, and the columns represent the to vertices. In case of no connection between two vertices, their edge weight is stored as zero:

adjacencyMatrix <-

setRefClass( Class = "adjacencyMatrix",

fields = list(n = "integer"), methods = list(

## Initialise the graph of n vertices Initialize <- function(n){

numVertices <<- as.integer(n) ## with n vertices numEdges <<- 0L ## with no connected edges mark <<- list() ## initialize mark list

## initialize the mark of all vertices to 0 (unvisited)

[ 199 ]

Page 211

﻿

Graphs

for(i in 1:numVertices) mark[[i]] <<- 0L

## generate a new nxn matrix with initial weights as 0 mat <- matrix()

for(i in 1:numVertices) for(j in 1:numVertices) mat[i,j] <<- 0L

},

## get number of vertices

num\_vert <- function() return(numVertices),

## get number of edges

num\_edges <- function() return(numEdges),

## return the first adjacent neighbor of vertex index v firstVertex <- function(v){ },

## return next adjacent vertices of index v after ## getting index w using firstVertex nextVertex <- function(v,w){ },

## Assign weight to each connected edge of indices v1 and v2 assignEdge <- function(v1,v2,wt){ },

## Delete a connected edge between indices v1 and v2 deleteEdge <- function(v1,v2){ },

## Check whether an edge exists between indices v1 and v2 isEdge <- function(v1,v2){ return(mat[v1,v2] != 0) },

## Get weight of the connected edge between indices v1 and v2 weightEdge <- function(v1,v2){

return(mat[v1,v2]) },

## Get the mark of a vertex of index v1 getMark <- function(v1){

return(mark[[v1]]) },

## initialise the mark of a vertex of index v1 with 1 initMark <- function(v1,val){

mark[[v]] <<- val}

))

[ 200 ]

Page 212

﻿

Chapter 8

For a given vertex V, the firstVertex function scans through the row V of the matrix mat to locate the first edge and its corresponding to vertex. If the function fails to find the first vertex, it returns the value n+1 . The nextVertex function is used to find the subsequent connected edge for the vertex V. If the edge is found, the nextVertex function will return the index value of the connected vertex, or else it will return the value n+1 . The following R snippet can be used to get firstVertex and nextVertex :

## return the first adjacent neighbor of vertex index v firstVertex <- function(v){

for(i in 1:numVertices) if(mat[v,i] != 0) return(i)

return(numVertices+1)

},

## return next adjacent vertices of index v after ## getting index w using firstVertex nextVertex <- function(v,w){ for(i in (w+1):numVertices) if(mat[v,i] != 0) return(i)

return(numVertices+1)

},

The assignEdge function is used to append the edges of a graph in the array, and deleteEdge is used to delete the edge from the array. The weightEdge function is used to return the edge value of the given from and to vertices. The following R code implements the adjacency matrix representation of graphs. It takes in a number of vertices n as an input. The R script for assignEdge and deleteEdge is shown as follows:

## Assign weight (wt) to each connected edge of indices v1 and v2 assignEdge <- function(v1,v2,wt){

if(wt<0) stop(""Weight should be positive"")

## increase the count of edges as the weights are assigned if(mat[v1,v2] == 0) numEdges <<- numEdges + 1L ## replace 0 with the wt mat[v1,v2] <<- wt

},

## Delete a connected edge between indices v1 and v2 deleteEdge <- function(v1,v2){

if(mat[v1,v2] != 0) numEdges <<- numEdges - 1L mat[v1,v2] <<- 0

}

[ 201 ]

Page 213

﻿

Graphs

In case of adjacency lists, the data structure is not as simple as in the case of an adjacency matrix. Here, a list vertex of length n is initialized, and each element in the list is assigned to its edges using the linked lists form of data structure. These lists store the index value of connected vertices along with their edge weights. It takes in a number of vertices n as input:

adjacencyList <-

setRefClass( Class = "adjacencyList",

fields = list(n = "integer"), methods = list(

## Initialise the graph of n vertices Initialize <- function(n){

numVertices <<- n # with n vertices

numEdges <<- 0L # with no connected edges mark <<- list() # initialise mark list

## initialize the mark of all vertices to 0 (unvisited) for(i in 1:numVertices) mark[[i]] <<- 0L

## generate a list of edges each for ## each vertex in the list vertex <- list()

for(i in 1:numVertices)

vertex[[i]] <<- llistofEdges()

},

## get number of vertices

num\_vert <- function() return(numVertices),

## get number of edges

num\_edges <- function() return(numEdges),

## return the first adjacent neighbout of vertex index v firstVertex <- function(v){ },

## return next adjacent vertices of index v after ## getting index w using firstVertex nextVertex <- function(v,w){ },

## Assign weight to each connected edge of indices v1 and v2 assignEdge <- function(v1,v2,wt){ },

## Delete a connected edge between indices v1 and v2 deleteEdge <- function(v1,v2){ },

## Check whether an edge exists between indices v1 and v2 isEdge <- function(v1,v2){

pos <- currentPos(vertex[[v1]], firstAdjVert(vertex[[v1]])) while(pos < length(vertex[[v1]])){

adjVert <- nextAdjVertex(vertex[[v1]],vertex[[v1]][pos]) if(adjVert == v2){

[ 202 ]

Page 214

﻿

Chapter 8

return(TRUE)} else {pos = pos+1 }

}

},

## Get weight of the connected edge between indices v1 and v2

weightEdge <- function(v1,v2){

if(isEdge(v1,v2)){

adjEdge <- getValue(vertex[[v1]],v2) return(adjEdge)

} else {return (0)}

},

## Get the mark of a vertex of index v1 getMark <- function(v1){

return(mark[[v1]])

},

## initialise the mark of a vertex of index v1 with 1 initMark <- function(v1,val){

mark[[v]] <<- val

} ))

The functions firstVertex and nextVertex scan through the list to determine adjacent vertices using the R function as follows:

## return the first adjacent neighbour of vertex index v firstVertex <- function(v){ if(length(vertex[[v]]) == 0)

## indicates no adjacent neighbour return(numVertices+1)

## Move to the first adjacent vertex adjVert <<- firstAdjVert(vertex[[v]]) ## get the current position of AdjVert pos <<- currentPos(vertex[[v]],adjVert) ## get value of connecting edge

adjEdge <<- getValue(vertex[[v]],adjVert) return(adjVert)

},

## return next adjacent vertices of index v after ## getting index w using firstVertex nextVertex <- function(v,w){

if(isEdge(v,w)){

if(pos+1 > length(vertex[[v]])){

## move the next adjacent vertex of w adjVert <<- nextAdjVertex(vertex[[v]],w)

## get the current position of adjcent vertex

[ 203 ]

Page 215

﻿

Graphs

pos <<- currentPos(vertex[[v]],adjVert) ## get value of connecting edge

adjEdge <<- getValue(vertex[[v]],adjVert) return(adjVert)

}

## no connecting edge

} else return(numVertices+1)

},

The functions assignEdge and deleteEdge traverse across the linked lists of a given vertex. The following R code implements the adjacency list representation of graphs:

## Assign weight (wt) to each connected edge of indices v1 and v2 assignEdge <- function(v1,v2,wt){

if(wt<0) stop("Weight should be positive") ##check whether edge exists between v1 and v2 if(isEdge(v1,v2)){

## insert vertex v2 along with edge weight wt insertVertex(vertex[[v1]],v2,wt)

}

},

## Delete a connected edge between indices v1 and v2 deleteEdge <- function(v1,v2){

if(isEdge(v1,v2)){ removeEdge(v1,v2)

numEdges <<- numEdges - 1L

}

},

Graph traversals

The concept of traversing across various nodes (or vertices) in a graph along connected edges is termed as graph traversal. The traversing across nodes is typically organized, but can sometimes be random. However, in both the scenarios, traversing begins from a specified start node and ends at a specified final node. Usually, the start and final nodes are not directly connected. In order to establish an indirect connection, a selective organized search is instantiated across the various connected paths. Graph traversal algorithms are generally designed to begin at a given start node and then search for subsequent connected nodes before terminating at the given final node.

[ 204 ]

Page 216

﻿

Chapter 8

Additionally, these traversing algorithms also need to factor in some plausible issues such as the following:

Infinite looping : Traversing can end in an infinite loop provided the graph contain cycles

Disconnected graph : Traversing (from a specified start node) sometimes can culminate without reaching all the nodes, as they are not connected with the traversed path

These issues are typically addressed by keeping track of the nodes encountered along the paths traversed. The nodes once marked are generally not visited again, unless otherwise specified. Thus, infinite looping can be prevented from happening. In addition, if all the nodes are not visited in a traverse, then a new traversal is initiated from an unmarked node, which continues until all the nodes are visited at least once.

The following R code shows the structure of a graph traverse algorithm. It takes in Graph\_ADT , number of vertices n , and a vector of vertices as inputs:

graph\_Traverse <- function(Graph\_ADT,n,vertices) {

## Initialise marks to zero verticesMarks <- list() for( i in 1:n)

verticesMarks[[i]] <- Graph\_ADT$initMark(vertices[i],0) ## 0 means not visited

## Initiate traversing upon checking for unmarked nodes for(i in 1:n)

if(Graph\_ADT$getMark(vertices[i])==0) initTraverse(verticesMarks,vertices[i])

}

The following are some approaches to implement graph traversal algorithms (the initTraverse function): Depth-first search (DFS) Breadth-first search (BFS) Topological sort

[ 205 ]

Page 217

﻿

Graphs

Depth-first search

DFS is a recursive implementation of the graph traversal algorithm, applicable to both directed and undirected graphs. At each step during the traverse, DFS recursively checks and visits all the unvisited nodes, which are directly connected with the node under consideration. Simultaneously, all the visited nodes along the path are pushed into a stack in the order of the traverse. During the traverse, if nodes with no unvisited nodes directly connected to them are encountered, then such nodes are popped out of the stack leaving behind the ones with directly connected unvisited nodes. This keeps track of those nodes that determine the forward traverse path such that all the nodes are visited prior to culmination. The following R code implements the DFS algorithm with three inputs. The inputs are Graph\_ADT , n (number of nodes in the graph), and v (node under consideration to perform DFS):

DepthFirstSearch <- function(Graph\_ADT, n, v) {

## Ensure all nodes are visited and processed prior node v preVisit(v)

## mark the node v under consideration as 1 (i.e. visited) VerticesMarks <- list()

VerticesMarks[[v]] <- Graph\_ADT$initMark(v,1)

## Recursively visit all connected nodes of v till all are marked as 1

## get the first vertex

node <- Graph\_ADT$firstVertex(v)

## check node belongs to neighboring nodes using conVert function while(node %in% conVert(v)){

## check if the node is unvisited

if(Graph\_ADT$getMark(VerticesMarks[[node]] == 0)) ## recursively run DFS

DepthFirstSearch(Graph\_ADT,n, node) ## assign next neighbouring vertex node <- Graph\_ADT$nextVertex(v,node)

}

## Run post processing remaining un-visited nodes postVisit(v)

}

[ 206 ]

Page 218

﻿

Chapter 8

An example of a DFS algorithm on an undirected graph is shown in Figure 8.4.

Figure 8.4: Represents an undirected graph (left) along with its ﬁnal search path (right) using a DFS algorithm

Figure 8.4 illustrates an initial undirected graph and its corresponding final search path obtained using the DFS algorithm. Figure 8.5 illustrates the working of DFS in detail using stacks. The sub-function nextVertex selects the node with the lowest index value among all the directly connected unvisited nodes. As the graph is undirected, the DFS algorithm can move in either of the directions as against the directed graph, where the DFS algorithm can move only in a single direction.

[ 207 ]

Page 219

﻿

Graphs

The asymptote of the DFS algorithm is θ(|E|+|V|) . |E| represents a visit to each node (traversing all edges only once) and |V| represents visiting each node (only once).

Fig 8.5: Illustration of recursive processing using the DFS algorithm and using A as a start vertex

[ 208 ]

Page 220

﻿

Chapter 8

Breadth-first search

BFS works on a principle similar to the DFS algorithm except the following:

BFS is not a recursive implementation unlike DFS

To keep track of the marked nodes, BFS uses queues as against the stacks used in DFS

Prior to moving to the next node, BFS ensures visits to all of its directly unmarked connected nodes unlike DFS, wherein only one of the unmarked connected nodes is visited at each iteration

The following R code implements the BFS algorithm using four inputs. The inputs are Graph\_ADT , startVertex (starting nodes of the graph to begin the traverse), queue (an empty queue to keep track of connected nodes in the order of visit), and n (total number of nodes in the graph):

BreadthFirstSearch <- function(Graph\_ADT,startVertex, queue, n) {

## initialise an empty queue with a start vertex queue <- initQueue(startVertex)

## Initialise first vertex by marking it as 1 (visited) VerticesMarks <- list()

VerticesMarks[[v]] <- Graph\_ADT$initMark(v,1)

## Subsequently start processing in queues while(length(queue) != 0){

## extract first element in the queue v <- extQueue(queue)

## Pre-Process all directly connected nodes of v preVisit(v)

## Mark visited nodes with 1 and accordingly queue the nodes node <- firstVertex(v)

while(node %in% conVert(v)){ if(getMark(graph[node] == 0)){

graph <- Graph\_ADT$initMark(node,1) queue <- initQueue(node)

}

node <- Graph\_ADT$nextVertex(startVertex,node)

} } }

[ 209 ]

Page 221

﻿

Graphs

Figure 8.6 illustrates an initial undirected graph and its corresponding final search path obtained using the BFS algorithm.

Figure 8.6 : Represents an undirected graph (left) along with its ﬁnal search path (right) using the BFS algorithm

Application of the BFS algorithm on queues is illustrated in Figure 8.7.

Figure 8.7 : Illustration of processing using the BFS algorithm and using 1 as a start vertex

[ 210 ]

Page 222

﻿

Chapter 8

Topological sort

The topological sort algorithm is primarily used in scenarios where nodes are conditionally dependent on previous nodes. In other words, the graph traversal can only happen if all its predecessor-connected nodes are visited (or processed). It is generally used in jobs where each stage is scheduled one after the other. For example, during construction of a tower, the columns cannot be raised till the foundation is complete, and roofing cannot be done till the columns are erected. Here, laying the foundation is followed by the erection of columns, which is followed by laying of the roof.

DAG form the basis of topological sort algorithms. In DAG, all the nodes are directionally connected, which takes care of order, and none of the nodes form a cycle, which ensures no conflict with any predecessor nodes (already visited and marked). Thus, DAG safeguards the linearity order among interconnected nodes, thereby being suitable for implementation of the topological sort algorithm. Figure 8.8 illustrates an example of a DAG acceptable for implementing the topological sort algorithm. The topological sort of this graph is 1, 2, 3, 4, 5, 6, 7, 8.

Figure 8.8: An example graph to perform topological sort

The topological sort algorithm is performed using both the DFS and BFS algorithm on DAGs.

In the case of a DFS approach, when a node is visited, no pre-processing is performed (using the preVisit function), whereas during recursive implementation, if the same node is revisited, then that node is returned as an output (using the postVisit function). Thus, the order of output-returned nodes is a reverse sort. The output for the preceding example DAG using the DFS algorithm is 8, 7, 6, 4, 5, 2, 3, 1. Thus, the topological sort is reverse of the output, that is – 1, 3, 2, 5, 4, 6, 7, 8. This is also called In-order search (pre-order and post- order). Also, in-order search leads to a sorted output.

[ 211 ]

Page 223

﻿

Graphs

The following R code implements a topological sort using the recursive DFS algorithm. The inputs are Graph\_ADT , n (total number of nodes in the graph), and vertices (a vector of vertices of the graph):

## Main function to perform topological sort

Topological\_DFS\_sort <- function(Graph\_ADT, n, vertices) {

## initialise all nodes with 0 (unvisited) verticesMarks <- list() for(i in 1:n)

verticesMarks[[i]] <<- Graph\_ADT$initMark(vertices[i],0)

## Process all nodes by recursive traversing for(i in 1:n)

if(Graph\_ADT$getMark(vertices[i]) == 0)

topological\_secondary(Graph\_ADT,i)

}

## recursive secondary function to help main function topological\_secondary <- function(Graph\_ADT,i) {

## Mark the node as 1 (visited)

verticesMarks[[i]] <<- Graph\_ADT$initMark(vertices[i], 1)

## Perform traversing across connected nodes v <- Graph\_ADT$firstVertex(vertices[i]) while(v %in% conVert(vertices[i])){

if(Graph\_ADT$getMark(vertices[i] == 0)) topological\_secondary(vertices,v)

v <- Graph\_ADT$nextVertex(vertices[i],v)

}

return(v)

}

In the case of the BFS approach, the topological sort algorithm is implemented using queuing logic. Here, the nodes are inserted into the queue, not only purely based on their index value (as described in the previous section), but also taking the account of each node's pre-requisites. One of the widely used pre-requisites is the count of inward edges for each node. These counts determine the constraints for each node. Once each node is assigned its respective counts, the nodes with zero count are considered as starting nodes, and are placed in the queue in a predefined order (for example, based on their index value). Then the queuing process begins, where each node is pushed out of the queue, and all its relevant connected nodes are pushed into the queue. Once a node is pushed out, the counts of its directly connected nodes are decreased by a value of one, and the nodes, which have their current count reduced to zero, are pushed into the queue. The order in which the nodes are pushed out of the queue determines the output of the topological sort. Sometimes, the queue becomes empty and not all the nodes are visited.

[ 212 ]

Page 224

﻿

Chapter 8

These situations arise because of some cyclicity present in the graph or on violation of any of the node's prerequisites. The output of the preceding example using the BFS algorithm in topological sort is 1, 2, 3, 4, 5, 6, 7, 8, 9.

The following R code shows an implementation of BFS in topological sort. The inputs are Graph\_ADT , x (total number of nodes in the graph), vertices (a vector of vertices of the graph), and queue (an empty queue to keep track of connected nodes in the order of visit):

Topological\_BFS\_sort <- function(Graph\_ADT, queue, n, vertices) {

## Initialise a list to track count of inwards edges for each node countEdge <- list()

## initialise count of each node to 0 for (i in vertices) countEdge[[i]] <- 0

## Assign count (inward nodes) prerequisite to each node for(i in vertices){

v <- Graph\_ADT$firstVertex(vertices[i]) while(v %in% conVert(vertices[i])){ countEdge[[v]] <- countEdge[[v]] + 1

v <- Graph\_ADT$nextVertex(vertices[[i]],v)

} }

## Initialize queue with nodes which have zero count of inward edges

for(i in vertices)

if(countEdge[[i]] == 0)

queue <- Graph\_ADT$initQueue(i)

## Process the nodes which are in the queue while(length(queue) != 0){

v <- extQueue(queue) print(v)

w <- Graph\_ADT$firstVertex(v)

while(w %in% conVert(vertices[v])){

## Decrease the count prerequisite by 1 countEdge[[w]] <- countEdge[[w]] - 1

if(countEdge[[w]] == 0) ## no prerequisites queue <- initQueue(w)

w <- Graph\_ADT$nextVertex(vertices[v],w)

} } }

[ 213 ]

Page 225

﻿

Graphs

Shortest path problems

Consider a city with a large number of roads interconnecting all its core areas, and you need to drive from an area P to an area Q. As the network of road is dense, you can have multiple options to reach area Q; nonetheless, you would desire to take the shortest route. However, the shortest route can have higher traffic; hence, you may now desire to take a new route, which minimizes travel time with a trade-off with distance. Adding further constraints, not all the routes allow bi-directional traffic. In other words, the shortest route needs to satisfy multiple constraints, and then, suggest the best possible route. Analogously, in a graph, each node corresponds to a city, and the edges correspond to the interconnecting roads. The weights of the edges can either be compared to distance or to travel time (based on traffic). These graphs can also be directed or un-directed based on whether a lane allows traffic in a single direction or both. Hence, it is not trivial to deduce the shortest path satisfying all the constraints. The graph in Figure 8.9 illustrates a network of roads with respective distances and directions.

Assume that you need to travel from node A to node F. Then, there are five possible routes connecting node A to node F. Each path comprises of a set of intermediate nodes (except one direct connection), and each edge connecting these nodes contributes for the calculation of distance from node A to node F. The distance of travel from A to E to F is 33, whereas the distance from A to C to F is only 19. In addition, the distance from A to B to F is 26, from A to B to D to F is 18, and A directly to F is 25. Thus, the shortest distance from A to F is 18. Now, this brings out some interesting nuances such as the following:

Not all direct connections have a minimal cost. Here, the shortest distance from A to F is not a direct connection.

Paths with less intermediate nodes need not have a lower cost. Here, the shortest path has the maximum number of intermediate nodes (that is – 2).

Unconnected nodes assume an infinite distance between them, such as the distance of the edge directly connecting C and D.

All the distances (or costs / weights) assume to take in positive values. Negative values imply a reverse direction in case of directed graphs and zero value (disconnect) in case of undirected graphs.

[ 214 ]

Page 226

﻿

Chapter 8

Figure 8.9 : Example of a road network connecting six nodes (A to F)

Single-source shortest paths

This section deals with the analysis of all possible shortest paths for a given single source (that is – a start vertex V) in a graph G. The shortest paths are determined between the start vertex V and all other vertices in the graph. In other words, the computation of the shortest distance between a start vertex V and an end vertex W would involve finding all possible shortest paths from start vertex V to all other intermediate vertices (worst-case scenario). It is widely used in computer routing networks, which involve transfer of data from one start source to multiple sources across the network. The time taken to transfer data or the edge network connectivity governs the cost parameter of the graph network.

As studied earlier, graphs can be broadly classified based on direction and edge weights. In case of undirected and unweighted (or equal weighted) graphs, the BFS algorithm is widely used to estimate single-source shortest paths. Once the edges are assigned with different weights, Dijkstra's algorithm is widely used to estimate single-source shortest paths regardless of their directions. The key features of Dijkstra's algorithm are as follows: It maintains a track of the shortest possible distance between the source vertex and all other vertices of the graph

It also keeps a track of the path that outlines the shortest possible route from the source vertex to all other vertices of the graph

[ 215 ]

Page 227

﻿

Graphs

Initially, an infinite value is assigned to each vertex of the graph. The value here represents the distance from the given source vertex. To begin with, the value of the source vertex is decreased to zero, and all its adjacent neighbors are processed with the updated distance values. Then, the vertex with the least value is extracted, and the values of all its adjacent unmarked vertices are updated accordingly. This process continues until all the vertices are extracted and all the values processed. In the end, the algorithm returns two key outputs. One output displays the shortest possible distances to each vertex from the source vertex V, while the second output shows the linkage of each vertex with its parent vertex. The second output is used to deduce the shortest path to any vertex from the given source vertex V. In the current implementation of Dijkstra's algorithm, all the vertices and their distance values are initially stored in a priority queue. The priority queue is used for insertion ( push ) and extraction ( extractMinVertex ) of key-value pairs. The Push function is used in insertion of a new key-value pair to the priority queue, and the extractMinVertex function is used in extraction of a key-value pair with the least value. The value represents the distance of the key (vertex) from the source vertex. The vertices are subsequently extracted, processed, and stored in two different hash maps. One hash map stores the shortest distances from the source vertex, and the other stores parent vertices to keep track of the shortest path from the source vertex.

The current implementation of the priority queue function uses R5 classes. The R code is as follows:

PriorityQueueInit <-

setRefClass("PriorityQueueInit",

fields = list(keys = "integer", values = "integer"), methods = list(

push = function(key,value) { keys <<- append(keys, key)

values <<- append(values, value)

},

extractMinVertex = function() {

minPos <- which(values==min(values)) key <- keys[[minPos]]

value <- values[[minPos]]

return(list(key=key,value=value))

}

))

[ 216 ]

Page 228

﻿

Chapter 8

Using the preceding priority function and two hash maps (from the cran package hashmap), the following R code implements Dijkstra's algorithm. The four inputs to this function are Graph\_ADT , a sourceVertex , a vector of all the vertices of the graph, and the number of vertices, n :

DijkstraShortestPath <- function(graph,sourceVertex,vertices,n) {

library(hashmap) ## To create new hashmap instance ## Initiate a new priority queue

priorityQueue <- PriorityQueueInit$new()

# Initiate a hashmap to store shortest distance from # source vertex to every vertex

distanceMap <- hashmap(keys=vertices, values = rep(0,n))

# Initiate another hashmap to store the parent vertex to keep a # track of shortest path from source vertex to every vertex parentMap <- hashmap(keys = sourceVertex, values = "NULL")

# initialize priority queue with value of all vertices to infinity for( i in vertices) priorityQueue$push(vertices[i],Inf) ## Set the distance of sourceVertex as zero

priorityQueue$values[which(priorityQueue$keys==sourceVertex)] <- 0 ## Begin iteration till the all the vertices from ## priorityQueue becomes empty

while(length(priorityQueue$keys) != 0){

## Extract vertex with minimum value from priority queue

headVertex <- priorityQueue$extractMinVertex()

## Assign the key of the head vertex as current vertex currentVertex <- headVertex$key

## Append distancemap with current key and its value distanceMap[[currentVertex]] <- headVertex$value

## Check for all directly connected vertices for current vertex for(conVert in getConVertex(graph,currentvertex)){

## get all the corresponding edge value

edgeValue <- getEdgeValue(graph,currentvertex,conVert) ## Check priority queue contains the adjacent connected ## vertex (conVert) or not

if(!priorityQueue$keys %in% conVert){ next }

## Now evaluate the distance of the adjacent vertex (conVert) updDistance <- distanceMap[[currentVertex]] + edgeValue

## Updated parentmap using value of the adjacent vertex ## in priorityQueue

if(priorityQueue$values[which(priorityQueue$keys==conVert)] > updDistance){

priorityQueue$values[which(priorityQueue$keys==conVert)] <- updDistance

parentmap[[conVert]] <- currentVertex

} }

[ 217 ]

Page 229

﻿

Graphs

} }

The time complexity of the current implementation is θ(|E| log(V)) , as during the worst case scenario, the size of the priority queue will be |V| , and the number of push and extract operations will be |E| . However, memory complexity of the current implementation is θ(|E| + |V|) , because during the worst case scenario, the size of the priority queue and distance map will be |V| , and the size of the parent map will be |E| .

Let us understand the working of Dijkstra's algorithm based on the graph given in Figure 8.9 . Initialize vertex A as the source vertex with value zero and rest of the vertices with value infinity. Then extract A, as it has the minimum value, and check for all its adjacent connected vertices. Update vertices B, C, E, and F with the respective distance values of edges (from source vertex A). Then, extract vertex C, as it has the least value among the remaining lot of vertices. Now, search for its connected vertices, which is F. The current value of F is 25, based on edge (A, F); however, based on the edge connection from C, the distance of F from A is the sum of edge distances (A, C) and (C, F), which comes out to be 19 (lesser than 25). Hence, update the value of F with 19, and assign C as a parent of F. Now, based on the updated vertex values, select the unmarked/unvisited vertex with the least distance, and continue updating the adjacent vertices. Table 8.10 shows the updated vertex values at the end of every extraction.

Table 8.10: Illustration of updated vertices values at the end of each extraction using Dijkstra's algorithm

[ 218 ]

Page 230

﻿

Chapter 8

Minimum-cost spanning tree

A Minimum Spanning Tree ( MST ) works on graphs with directed and weighted (non- negative costs) edges. Consider a graph G with n vertices. The spanning tree is a subgraph of graph G with all its n vertices connected to each other using n-1 edges. Thus, there is no possibility of a cycle with the subgraph. If the spanning tree does have a cycle, then it is advisable to remove any one edge, probably the one with the highest cost. The spanning tree with the least sum of edge weights is termed as a MST. It is widely used in applications such as laying of power cables across the city, connecting all houses using the least length of power cables. Here, the weight of each edge is the length of the cable, and the vertices are houses in the city. The most common algorithms to find the minimum cost spanning tree are Prim's algorithm and Kruskal's algorithm. Figure 8.11 shows the minimum cost spanning tree for an undirected-weighted graph.

Figure 8.11: Illustration of an undirected graph (left) and its minimum-cost spanning tree (right)

Prim's algorithm

Prim's algorithm works on lines similar to Dijkstra's algorithm to find the least cost edges connecting all the vertices in the graph. In case of Dijkstra's algorithm, the selection of least cost edges depends primarily on the source vertex, whereas in case of Prim's algorithm, the least cost edge does not depend on any source vertex. Now let us understand the working of Prim's algorithm in detail. Consider a graph G with n vertices, all edges weighted with non-negative costs. Initially, select any vertex V from the graph to begin the traverse. Then, look for all its connected edges, and select the one with the least cost. Let us assume the next selected vertex based on least cost is W. Now, again look for all connected edges of V and W, and select the edge with the least cost. Now, add this new vertex with the V and W. Continue searching for least cost edges until all the vertices are traversed across the graph. The selected edges will then form a MST of the graph such that one can traverse from any vertex to any other vertex. Prim's algorithm works greedily, because at each step, the algorithm tries to select the new (unmarked) vertex, which has the least edge cost compared to all the edges connected to a certain marked vertex.

[ 219 ]

Page 231

﻿

Graphs

The algorithm's key purpose is to select least cost edges, but it does not care whether the selected vertices form a minimum-cost spanning tree or not. This trickles down to a question of whether the output of Prim's algorithm is actually a minimum-cost spanning tree? The proof for the output being a MST is an exercise question for the readers. Now, let us understand the working of Prim's algorithm with an example graph shown in Figure 8.12 . Initialize the algorithm with vertex A, and scan for all its connected edges. It leads to vertex D, as the connecting edge (A, D) has the least cost, 8. Now, assign edge (A, D) to MST, and scan for all other edges connected to vertices A and D. It leads to vertex E, as the connected edge (D, E) has the least cost, 10. Now, assign edge (D, E) to MST, and scan for all other edges connected to vertices A, D, and E. This leads to vertex C, as the connecting edge (A, C) has the least cost, 12. Now, assign edge (A, C) to MST, and scan for all other edges connected to vertices A, C, D, and E. This leads to vertex B, as the connecting edge (C, B) has the least cost, 15. Now, assign edge (C, B) to MST. Thus, a minimum-cost spanning tree is obtained with all the vertices connected using least cost edges.

The key difference in the implementation of Prim's algorithm and Dijkstra's algorithm is in the way the value of a vertex is updated over an extraction. In Dijkstra's algorithm, the distance value of each vertex is updated based on the value of the current vertex and the connecting edge value with the current vertex. However, in case of Prim's algorithm, the distance value of each vertex depends only on the edge value of the current connecting vertex. In the former approach, each vertex seeks closeness towards the source vertex, whereas in the latter approach, each vertex can seek closeness towards any vertex in the graph.

The following R code implements Prim's algorithm. It uses the same priority queue as that of Dijkstra's algorithm:

primMST <- function(Graph\_ADT,vertices,n) {

library(hashmap) ## To create new hashmap instances ## Initiate a new priority queue

priorityQueue <- PriorityQueueInit$new()

## Initiate a hashmap to store shortest distance from source ## vertex to every vertex

distanceMap <- hashmap(keys=vertices, values = rep(0,n)) ## Initialise a list to store final MST result MSTResult <- list()

# initialize priority queue with value of all vertices to infinity for( i in vertices) priorityQueue$push(vertices[i],Inf) ## begin with a random vertex

startVertex <<- vertices[sample(1:n, 1)] ## Set the distance of startVertex as zero

priorityQueue$values[which(priorityQueue$keys==startVertex)] <- 0 ## Begin iteration till the all the vertices from priorityQueue

[ 220 ]

Page 232

﻿

Chapter 8

## becomes empty

while(length(priorityQueue$keys) != 0){

## Extract vertex with minimum value from priority queue along ## with its value

headVertex <- priorityQueue$extractMinVertex()

## Assign the key of the head vertex as current vertex currentVertex <- headVertex$key

## Append distancemap with current key and its value distanceMap[[currentVertex]] <- headVertex$value

# Check for all directly connected vertices for current vertex for(conVert in getConVertex(graph,currentvertex)){

## get all the corresponding edge value

edgeValue <- getEdgeValue(graph,currentvertex,conVert)

## Check priority queue contains the adjacent connected vertex ## (conVert) or not

if(!priorityQueue$keys %in% conVert){ next } ## Update the distance with the edge value updDistance <- edgeValue

## Check whether the value of the adjacent vertex

if(priorityQueue$values[which(priorityQueue$keys==conVert)]

> updDistance){

priorityQueue$values[which(priorityQueue$keys==conVert)]

<- updDistance

MSTResult[[currentVertex]] <- conVert

} } } }

Kruskal's algorithm

Similar to Prim's algorithm, Kruskal's algorithm is also a greedy algorithm in which the algorithm greedily selects edges based on edge value to generate a MST. Initially, partition all the vertices into an equivalent number of |V| sets, each with an individual vertex. Then, select an edge with the least cost and combine the equivalent sets of the from and to vertices into a single set. Also add the edge into the MST. Continue the process of selecting the minimum edge until all the vertices combine into a single set. In case of combining two inequivalent sets, first find the sets containing the from and to vertices, and accordingly merge those two sets. If both, from and to vertices for a particular edge, lie in the same set, then ignore the edge and proceed ahead.

[ 221 ]

Page 233

﻿

Graphs

Let us understand the implementation of Kruskal's algorithm for the graph in Figure 8.9 . First, split the five vertices into five different sets. Then, select edge (A, D), as it has the least weight among all the other edges. As A and D are in two different sets, combine them into a single set, and add the edge (A, D) into MST. Then, select edge (D, E), as it has the second least edge weight. As D and E are in two different sets, combine them into a single set, and add edge (D, E) into MST. Then, select edge (A, C), as it has the third least edge weight. As A and C are in two different sets, combine them into a single set, and add edge (A, C) into MST. Then, select edge (C, B), as it has the fourth least edge weight. As C and B are in two different sets, combine them into a single set, and add edge (C, B) into MST. Thus, all the vertices, A, B, C, D, and E, are present in a single set and the selected edges form a MST. Figure 8.12 illustrates the steps involved in Kruskal's algorithm for the graph in Figure 8.9 .

Figure 8.12: Illustration of Kruskal's algorithm on an example graph

The edges are processed in the order of edge weights using a priority queue reference class ( kruskalArray ). Here, pre-sorting of edge weights is not required, hence it reduces system runtime. In the kruskalArray reference class, the edges, along with their from and to vertices can be appended using the push function, and the edge with minimum weight can be extracted using the extractMinEdge function. Once the edge is extracted out, it is then removed from the array. The kruskalArray function is implemented using the R5 class, and it is as follows:

kruskalArray <- setRefClass("kruskalArray",

fields = list(fromVertex = "numeric", toVertex = "numeric", weight = "numeric"), methods = list(

## insert new from and to vertices along with edge push = function(f, t, w){

fromVertex <<- append(fromVertex,f) toVertex <<- append(toVertex,t) weight <<- append(weight,w)

},

[ 222 ]

Page 234

﻿

Chapter 8

## extract from and to vertices having minimum edge value ## also remove from, to and edge value from the array extractMinEdge = function() {

minPos <- which(weight==min(weight)) from <- fromVertex[[minPos]] to <- toVertex[[minPos]]

fromVertex <<- fromVertex[[-minPos]] toVertex <<- toVertex[[-minPos]] weight <<- weight[[-minPos]] return(list(from=from,to=to))

}

))

Using the disjoinSetPointer function, the operations of union , differ , and find are undertaken. Two different sets of vertices are combined using the union operation, and the differ operation is used to check whether two sets are disjoint or not. In case of a set with more than one vertex, the find operation is used to check whether a vertex belongs to that set or not. The disjoinSetPointer function is implemented using the R5 class, and it is given as follows:

disjoinSetPointer <- setRefClass("disjoinSetPointer",

fields = list(vertex = "vector", set1 = "vector", set2 = "vector",

currentVertex = "integer"), methods = list(

## merge two sets

union = function(set1,set2){

return(c(set1,set2))

},

## check whether set1 and set 2 are disjoint ## return TRUE if they are disjoint differ = function(set1,set2){ if(sum(set1 %in% set1) ==0){

return(TRUE)} else(return(FALSE))

},

## Find whether a vertex is in a set or not ## returns root of the currentVertex

## function ROOT returns root of the vector find = function(currentVertex){

return(ROOT(vertex[currentvertex]))

}

))

[ 223 ]

Page 235

﻿

Graphs

The following R code implements Kruskal's algorithm using the preceding two reference classes. The inputs are Graph\_ADT , the number of vertices, n , and the number of edges, e , in the graph:

kruskalMST <- function(Graph\_ADT,n,e) {

## initialize reference classes disjoinSetPointer and kruskalArray vertexArray <- disjoinSetPointer$new() edgeArray <- kruskalArray$new()

## Initialise a list to store final MST result MSTResult <- list()

## Put all the edges in the edgeArray for(i in 1:n){

j <- firstVertex(i) while(i <= n){

edgeArray$push(i,j, Graph\_ADT$weightEdge(i,j))

} }

## Initialise n equivalent sets numMST <- n

## Iteratively combine equivalent sets based on edge weights ## edges are extracted based on their value. Smallest edges are extracted first for(i in 1:e){

while(numMST >= 1){

# get the from and to vertices having minimum edge value temp <- edgeArray$extractMinEdge() fromVertex <- temp$from toVertex <- temp$to

## Check whether two vertices are in different sets if(vertexArray$differ(fromvertex,toVertex)){

## if yes, then combine from and to vertices into one set vertexArray$union(fromvertex,toVertex) ## add this edge to MST

MSTResult[[i]] <- c(fromVertex,toVertex) ## decrease the sets by 1 numMST <- numMST - 1

} } }

return(MSTResult)

}

[ 224 ]

Page 236

﻿

Chapter 8

The asymptote of Kruskal's algorithm, based on system runtime in a worst-case scenario is θ(|E| log|E|) , as all the edges need to be processed before the completion of generating a minimum-cost spanning tree. However, quite often, the number of minimum-value edge extractions is equivalent to the number of vertices in the graph (as shown in the preceding example). This makes the algorithm have a system runtime asymptote of θ(|V| log|E|) , generally observed in average-and best-case scenarios.

Exercises

1. Generate an adjacency matrix and an adjacency list for the following graph:

2. Generate a DFS and BFS tree for the preceding graph.

3. Prove the following hypothesis: An acyclic undirected graph with n nodes has no

more than n-1 edges.

4. Find the MST for the graph given in Figure 8.13 using Prim's and Kruskal's algorithm. Do they give the same MSTs? If no, in what type of situations do they give different MSTs.

5. Starting from vertex B, can you obtain single-source shortest paths using Dijkstra's algorithm? Do the edges obtained in question 4 of Exercises overlap with the edges obtained using Dijkstra's algorithm? If yes, explain the logic behind the overlap.

[ 225 ]

Page 237

﻿

Graphs

Summary

This current chapter covers the fundamentals of graphs and introduces terminology and representation. The later sections of this chapter covers searching techniques in graphs using DFS and BFS. This chapter also introduces in-order search in scenarios where nodes are conditionally dependent. The chapter also covers Dijkstra's algorithm widely used to estimate single-source shortest paths regardless of their directions. The concept of MST is introduced with algorithms such as Prim and Kruskal, which are covered to extract MST from a directed and weighted graph. The next chapter will extend of static algorithms to randomized algorithms, and it also introduces the fundamentals of programming.

[ 226 ]