Luis Fernando Muñoz A00046396
Sebastian Arango A00130532
Santiago del Campo A00137608

Phase 1: Identifying the problem.

Symptoms and needs

- The company wants to prevent potentially clients to get bored with their ads.
- Yogures Aliyogu wants to reduce costs in online advertisement.
- The ads should be showed in the most visited sites.
- If the publicity results expensive in well ranked sites, they would rather invest in “nearby” websites.

Problem’s definition

The company Yogures Aliyogu wants to know better the relationships that exist between internet domains, with the goal of redirect better the investment in online advertisement. Specifically, they want to reduce the cost of online advertisement through the investment in potentially lucrative websites, avoiding very famous ones like facebook or twitter, but at the same time, “close enough to them”, making the company able to profit similarly had they invested in the famous ones. To avoid the customers boredom, and at the same time to invest intelligently, they try to not repeat the same ads in a “loop” of websites, in other words, starting from a given website, the user cannot encounter an ad more than once while following a sequence of websites in a loop. You have been hired for this task.

Requirements list

Functional requirements

Name	R1 – Find cycles that can be made with a given domain
Summary	To find cycles starting from the given domain
Inputs	

Domain
Outputs
Set of domains belonging to the same cycle

Name	R2 –Find the shortest path
Summary	Given two domains, the shortest path of links between them is found
Inputs	
Two domains	
Outputs	
Sequence of domains making up the shortest path between the two given domains	

Non-Functional requirements

-The graph must be visualized in a GUI

Phase 2: Data mining

Definitions

Graph: Object joint called vertex or nodes linked by edges that represent relations between their elements.

Link: is an element of an electronic document that references another resource. It has origin, destination and direction.

Edge: abstract object connecting two vertices or nodes. In some cases it can be directed or not.

Vertex: Is the unit of which the graph is formed.

Phase 3: Search of creative solutions

(1)

One way to solve this problem is to use data in a hash map where the key would be a domain and the value a vertex.

(2)

The domain structure is saved as adjacency lists.

(3)

Another form is to use data in a matrix, where the row is the origin vertex and the column is the destiny.

(4)

The domain structure can be modeled using a graph, for more flexibility, this structure can be implemented using either adjacency lists or a matrix representation.

Phase 4: Preliminary designs

(1)

-Very efficient when trying to find in constant time a vertex given a domain.

(2)

-Compared to a matrix representation, space efficiency is one of its perks.

(3)

-Helps to manage more easily the nodes, that is because they are saved as numbers.

(4)

-This combination offers more flexibility even though requires more work to implement it.

Phase 5: Assessment of solutions

Criteria

A. The solution allows to access to a node in a temporal complexity:

[2] <O(1)

[1] O(V)

B. The database model in the solution is:

[2] Intuitive and easy to understand

[1] Complex and hard to code for programmers

C. The learning level thanks to implement the solution is:

[3]High

[2]Medium

[1]Low

D. The solution uses space complexity:

[2] <O(n^2)

[1] O(n^2)

Soluciones	Criterio A	Criterio B	Criterio C	Criterio D	Sumatoria
(1)	1	2	2	1	7
(2)	1	2	2	2	8
(3)	1	2	3	2	10
(4)	2	2	3	2	11

Phase 6: Specifications and reports

TAD Definitions

GraphMatrix
Representation: Matrix n^2 where the rows are the start vertices and the columns represent the arrival points
Invariants: n is the number of vertices. Every Object in the matrix belong to the same class
Operations:

CreateGraph	boolean	----> Graph
addEdge	E, V, V	----> boolean
addVertex	V	----> void
getVertices	True	----> List<V>
getEdges	True	----> List<E,V,V>
getLabel	V, V	----> E
getNeighbors	V	----> List<V>
isThereEdge	V, V	-----> boolean
isUndirected	True	-----> boolean
getNumberOfVertices	True	-----> int
getValue	int	-----> V
getInteger	V	-----> int
getEdgesArray	True	-----> List<E>[][]

CreateGraph(boolean d)
Pre: true
Post: A new graph has been created, if d is true, is undirected, directed otherwise

addEdge(E e, V v1, V v2)
Pre: e, v1 y v2 son != null
Post: The matrix is filled in the positions in which the vertices are joined, if is directed, v1 point v2

addVertex(V v)
Pre: v != null
Post: New vertex added in the graph increasing the size of the matrix by rows and columns

getVertices()
Pre: True
Post: list of vertices returned

getEdges()
Pre: True
Post: list of edges returned

getLabel(V v1,V v2)
Pre: v1 and v2 are in the graph
Post: edge connecting the two vertices is returned, null otherwise

getNeighbors(V v)
Pre: v is in the graph
Post: List of neighbors of v is returned

isThereEdge(V v1,V v2)
Pre: True
Post: There are vertices v1 and v2 and there exist an edge connecting them

IsUndirected()
Pre: True
Post: returns true if the graph is undirected, false otherwise

getNumberOfVertices()
Pre: True
Post: return number of vertices of the graph

getValue(int i)
Pre: True
Post: return the value associated to the given integer, null if there is no such integer

getInteger(V v)
Pre: v != null
Post: return the integer associated to the given value, null if there is no such integer

getEdgesArray()
Pre: True
Post: return the matrix containing the edges

TAD GraphList
Representation: **List of V elements in the graph, which point to its neighbors by edges E.**
Invariants: **Every Object in the matrix belong to the same class.** **V is the number of vertices.**
Operations:

Operation	Input	Output
CreateGraph	**boolean**	**----> Graph**
addEdge	**E, V, V**	**----> boolean**
addVertex	**V**	**----> void**
getVertices	**True**	**----> List<V>**
getEdges	**True**	**----> List<E,V,V>**
getLabel	**V, V**	**----> E**
getNeighbors	**V**	**----> List<V>**
isThereEdge	**V, V**	**-----> boolean**
isDirected	**True**	**-----> boolean**
getNumberOfVertices	**True**	**-----> int**
getVertex	**V**	**-----> Vertex**

CreateGraph(boolean d)
Pre: true
Post: A new graph has been created, if d is true, is undirected, directed otherwise

addEdge(E e, V v1, V v2)
Pre: e, v1 y v2 son != null
Post: The matrix is filled in the positions in which the vertices are joined, if is directed, v1 point v2

addVertex(V v)
Pre: v != null
Post: New vertex added in the graph increasing the size of the matrix by rows and columns

getVertices()
Pre: True
Post: list of vertices returned

getEdges()
Pre: True
Post: list of edges returned

getLabel(V v1,V v2)
Pre: v1 and v2 are in the graph
Post: edge connecting the two vertices is returned, null otherwise

getNeighbors(V v)
Pre: v is in the graph
Post: List of neighbors of v is returned

isThereEdge(V v1,V v2)

Pre: True
Post: There are vertices v1 and v2 and there exist an edge connecting them

IsUndirected()
Pre: True
Post: returns true if the graph is undirected, false otherwise

getNumberOfVertices()
Pre: True
Post: return number of vertices of the graph

getVertex(V v)
Pre: True
Post: return vertex associated to the given value, null if there is no such vertex

Test design

GraphList

Class	Method	Scene	Param	Results
GraphList	addEdge()	Scenario1()	(2,”santi”,”jose”)	Edge added
GraphList	getVertex()	Scenario1()	(“pepe”)	Vertex successfully obtained
GraphList	getValues()	Scenario1()	()	Values successfully obtained
GraphList	addVertex()	Scenario1()	(“ja”)	Vertex successfully

				added
GraphList	getLabel()	Scenario1()	(“juan”,”pepe”)	Label successfully obtained
GraphList	isThereEdge()	Scenario1()	(“pepe”,”juan”)	Returns true
GraphList	getNeighbours()	Scenario1()	(“pepe”)	Neighbours successfully obtained

GraphMatrix

GraphMatrix	addEdge()	Scenario1()	(2,”santi”,”jose”)	Edge added
GraphMatrix	getVertex()	Scenario1()	(“pepe”)	Vertex successfully obtained
GraphMatrix	expand()	Scenario1()	(“elemento”)	Matrix successfully expanded
GraphMatrix	getValue()	Scenario1()	(0)	Returns “pepe”
GraphMatrix	getValues()	Scenario1()	()	Values successfully obtained
GraphMatrix	addVertex()	Scenario1()	(“j”)	Vertex successfully added
GraphMatrix	getLabel()	Scenario1()	(”pepe”,“juan”)	Label successfully obtained
GraphMatrix	isThereEdge()	Scenario1()	(“pepe”,”juan”)	Returns true
GraphMatrix	getNeighbours()	Scenario1()	(“pepe”)	Neighbours successfully obtained
GraphMatrix	getEdgesArray()	Scenario1()	()	EdgesArray Obtained

				succesfully

GraphAlgorithm

Class	Method	Scene	Param	Results
GraphAlgorithm	bfs()	Scenario1()	(grafo,2)	Ancestors successfully formed
GraphAlgorithm	dfs()	Scenario1()	(grafo)	Ancestors successfully formed
GraphAlgorithm	kruskal()	Scenario1()	(grafo)	Minimum Spanning tree successfully formed
GraphAlgorithm	dijkstra()	Scenario1()	(grafo,1)	Minimum Distances successfully determined
GraphAlgorithm	prim()	Scenario1()	(grafo)	Minimum Spanning Tree successfully formed
GraphAlgorithm	floydWarshall()	Scenario1()	(grafo,"B")	All shortest distances determined

Web

Class	Method	Scene	Param	Results
web	findShortestPath()	Scenario1()	(d1,d2)	Graph containing sequence of domains in order to get from d1 to d2

web	findCycles()	Scenario1()	(d5)	Graph containing the cycles formed by the Domain given