Engineering Method
Economic Travel

Alejandro Coronel Sánchez / A00365049

Julián Andrés Riascos / A00365548

Kevin Alejandro Mera / A00364415

Andrés Alberto Aristizábal, Ph.D.

2021-1

ICESI University

Santiago de Cali

Valle del Cauca

Colombia

Problematic Context

The city of Santiago de Cali is the capital of the department of Valle del Cauca, and the most populated city in southwestern Colombia, with approximately 2,227,642 inhabitants according to the 2018 census. The city extends over 560.3 km2 (216 , 3 square miles) with 120.9 km2 (46.7 square miles) of urban area, making Cali the second largest city in the country by area and the third most populated. As the only major Colombian city with access to the Pacific coast, Cali is the main urban and economic center in the south of the country and has one of the fastest growing economies in Colombia. Traveling to Cali is a pleasure, its hotel and gastronomic infrastructure and its many places of interest are the reason why it is one of the cities with the highest tourist demand in Colombia.

Among the outstanding restaurants in Cali, we can find La Cocina, Roset, Platillos Voladores and El Bochinche, which are perfect for tasting the best pork dishes in the area. On the other hand, you can take time to awaken your senses in the Plaza de Caicedo, admire indigenous Colombian species in the Cali Zoo, feel the beauty of miracles in the Ermita church, explore the Cali Cathedral, look back at the La Merced Religious Art Museum, go up to the statue of Christ the Redeemer, walk through the famous and rumbera sixth avenue and the Granada neighborhood, among many more. Finally, after visiting all these places during the day, you can relax in the best hotels, among which the Marriot Hotel, the Spiwak Hotel, the El Peñón Hotel and the Cosmos Hotel stand out.

In Cali there is a great diversity of means of transport that can be obtained to move around the city: there is public transport, called Mio, which is the cheapest we can find with a price of 2,200, but it is also the one that takes the longest demand since they have established routes, and to go from one point of the city to another it may be necessary to make a transfer, which considerably increases the waiting time; We can find taxis, which increase the cost according to the distance that needs to be traveled, having as a minimum cost a value of 5,000 if it does not exceed 2 blocks around, when using a taxi the time is much less compared to the Mio; Finally, we find the services offered by digital platforms, which have the same minimum charge as taxis, but whose price progression according to distance is 15% lower compared to taxis, the time is also equal to that of taxis.

A group of young people from the Icesi University have decided to make an application that makes it easier for tourists to decide about which transport to use to travel to any of the mentioned tourist spots in the city, considering the lowest price or the shortest time. used for said tour.

Solution Development

To solve the problem of the quick basket properly, we will use the engineering method.

Step 1. Identification of the problem

- The program must be simple and easy to use.
- the most touristic places in Cali should be considered in the program.
- The program must show the best decision according to the needs of the tourist.
- The main factors that must be considered is time and money for the trip.
- It must have a graphical interface for user interaction.

Step 2. Information Compilation

Linked List

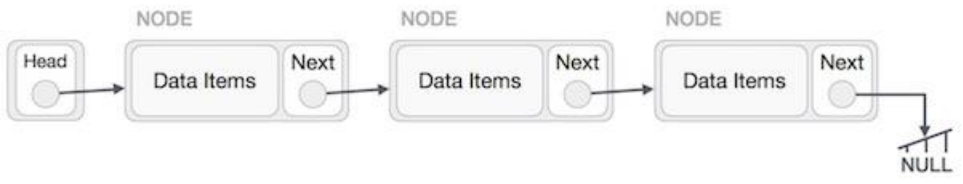
A linked list is a sequence of data structures, which are connected together via links.

Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** − Each link of a linked list can store a data called an element.
- **Next** − Each link of a linked list contains a link to the next link called Next.
- **LinkedList** − A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node point to the next node.



As per the above illustration, following are the important points to be considered:

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.

- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.
- **Circular Linked List** − Last item contains link of the first element as next and the first element has a link to the last element as previous.

Advantages:

1. LinkedList is that insertions and deletion can be done very quickly.
2. If you just want to insert an element right to the beginning of the LinkedList, that can be done in constant time O(1).
3. If you want to delete an element at the beginning of a LinkedList, again constant time O(1).

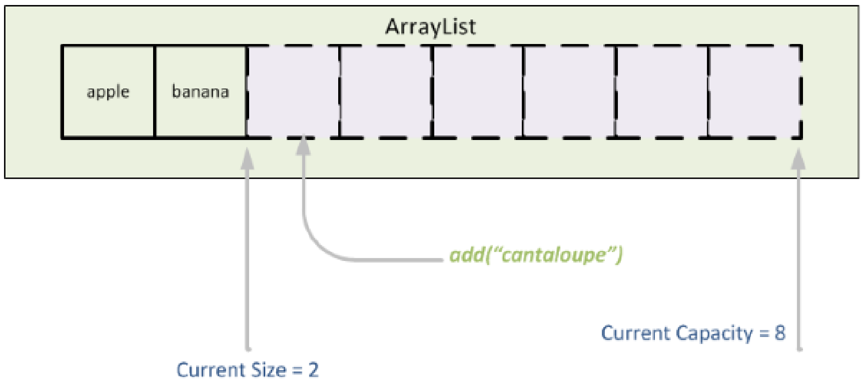
Disadvantages:

- If you want to append an item at the end of the list, that might require going through the whole LinkedList, until you reach the very last element, and then inserting the element, this will take Linear time O(n).

ArrayList

An ArrayList, or dynamically resizing array, allows you to have the benefits of an array while offering flexibility in size. You won’t run out of space in the ArrayList since it’s capacity will grow as you insert elements.

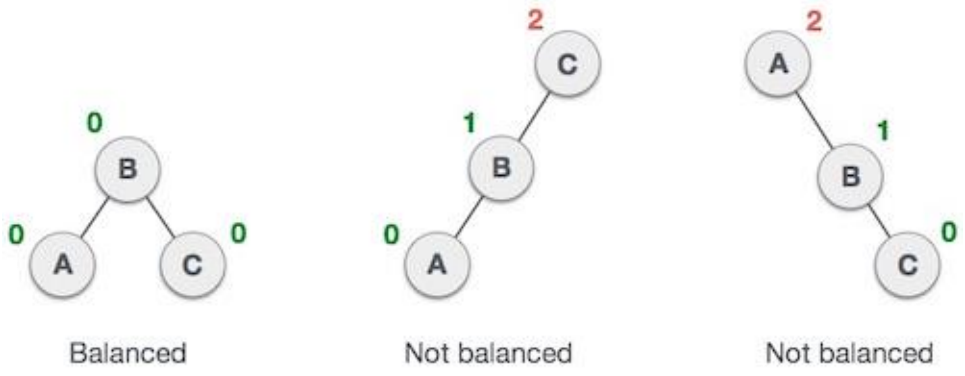
Array can contain both primitive data types as well as objects of a class depending on the definition of the array. However, ArrayList only supports object entries, not the primitive data types.



AVL Trees

AVL trees are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

Here we see that the first tree is balanced, and the next two trees are not balanced



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) − height(right-sutree)

AVL Rotations

To balance itself, an AVL tree may perform the following four kinds of rotations:

- Left rotation

- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations, and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2.

What is a Hash Function?

A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in the hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as the index in the hash table.

Choosing a good hashing function, **h(k)**, is essential for hash-table based searching. **h** should distribute the elements of our collection as uniformly as possible to the "slots" of the hash table. The key criterion is that there should be a minimum number of collisions.

If the probability that a key, **k**, occurs in our collection is **P(k)**, then if there are **m** slots in our hash table, a *uniform hashing function*, **h(k)**, would ensure:

$$\sum_{k|h(k)=0} P(k) = \sum_{k|h(k)=1} P(k) = \ldots = \sum_{k|h(k)=m-1} P(k) = \frac{1}{m}$$

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in (0,**r**], then,

$$\mathbf{h(k) = floor((mk)/r)}$$

will provide uniform hashing.

Most hashing functions will first map the keys to some set of natural numbers.

Having mapped the keys to a set of natural numbers, we then have several possibilities.

Use a **mod** function:

h(k) = k mod m.

When using this method, we usually avoid certain values of **m**. Powers of 2 are usually avoided, for **k mod 2^b** simply selects the **b** low order bits of **k**.

Use the multiplication method:

- Multiply the key by a constant **A**, 0 < **A** < 1,
- Extract the fractional part of the product,
- Multiply this value by **m**.

Thus, the hash function is:

h(k) = floor(m * (kA - floor(kA)))

In this case, the value of **m** is not critical

Use universal hashing:

A malicious adversary can always choose the keys so that they all hash to the same slot, leading to an average **O(n)** retrieval time. Universal hashing seeks to avoid this by choosing the hashing function randomly from a collection of hash functions.

Functional requirements

FR1: Display the map of Cali with places.

FR2: Quote a path travel by the lowest time or cost.

FR3: Determinate a path travel with limitations on the time or cost.

FR4: Determinate the lowest route time or cost to travel around all 42 places of Cali.

FR5: Show the travel route with the departure and arrival place.

FR6: Display the route way and notify the value calculated by the time or cost.

Step 3. Search of creative solutions

Since we already investigated and inquired about the operation of data structures such as hash tables or hash maps, graphs, linked list and arraylist. We can consider different possible solutions to our problem. These can be:

Option 1. Using Linked List

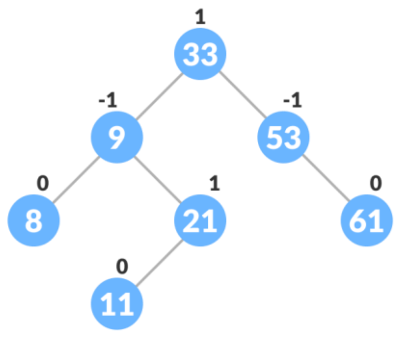
Where the nodes would be the tourist places, in such a way that after visiting each node when performing a recursive search, a value x would be increased (either price) and the most efficient would be the smallest of each route

Option 2. Using ArrayList

We can handle an arraylist of tourist places together with an arraylist of routes (edges), in such a way that by making a relationship between both, the most efficient route can be found depending on the type

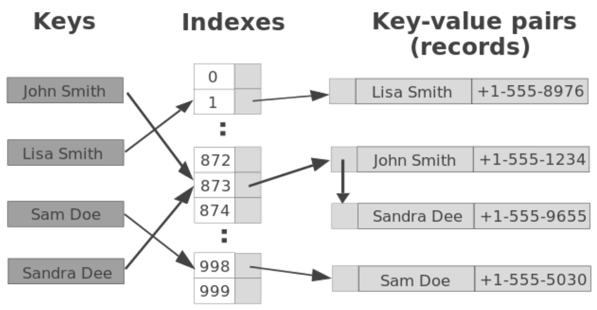
Option 3. Using AVL tree

Another option is using this kind of structure, and the way we should take to solve the problem with it must be use the number of nodes passed through to get higher the price of the time of a travel.



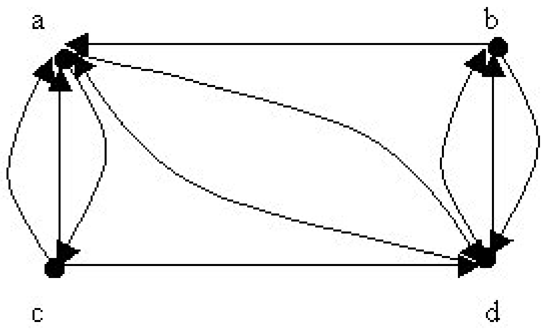
Option 4. Using Hash tables or hash maps

On the other hand, we could think of using hash tables/maps to, in this case, store the places and its times and cost in reaching these, accessing in an efficient way, and therefore having quick queries. Consider the following picture for places instead persons.



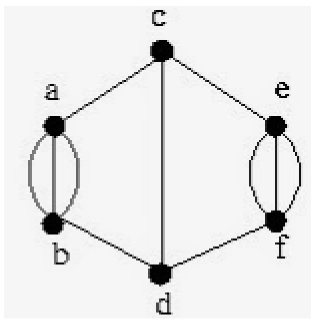
Option 5. Using guided multigraph

Use this type of graphs is a good option if we want to do a more realistic application, because the direction would mean the heading of the street. People cannot transit in against via, and this type of graph allows that option. And multigraph due to the means of transport that could be used.



Option 6. Using unguided multigraph and HashTables or hasMaps

Finally, we have unguided multigraph if we want to assume that we can go and return by the same route between two tourist places (which in many cases is like this). And multigraph due to the means of transport that could be used. The Hash tables/maps could be useful at the time of importing vertices and edges and later doing basic operations such as query, addition, and elimination. Then it can be passed to a matrix to implement algorithms required for graphs



Step 4. Ideas transitions to preliminary design

At this point, **we discard the Option 1. Using only LinkedList** since it would be difficult and very ambiguous to resort to this type of structure for the implementation of this problem, since each node, despite having its own value, would not give us a real value between different nodes, that is, the cost of the routes. with their respective means of transport.

We can see that **Option 2. Using only ArrayList** very similar to option 1, since here instead of using linked lists, only the java API array list will be used. Of course, with this structure recursive methods will not be used, which would facilitate the implementation of complex methods and that we already have a structure that provides a variety of methods for our solution, the problem here would be that it is not efficient enough since it would be cumbersome like option 1 in implementing this option to our solution. It does not leave any alternative to use other data structures that could be adapted in a better way to what you want to do in the program and thus fulfilling the requirements.

Now, we could consider the **Option 3. Using AVL tree** it would be a good idea to try to save the tourist places in the nodes and given the altitude of the tree the cost of transportation between each of these places. The problem would be that it is not known with certainty that the tree's altitude is proportional to the distance of the route depending on the means of transport used. On the other hand, it would not tell us a true relationship between the distance of each place. Well, if we have two nodes at the same height, we would be interpreting that they are at the same distance when it is not.

Careful review of the previously discarded alternatives leaves us with the following options to be analyzed:

Option 4. Using Hash tables/maps only

- The access times to the hash table or has map would be very fast.
- Would have a good organization when using hash tables or hash map.
- There is an approach to solving the problem.

Option 5. Using guided multigraph

- Appropriate data structures are used.
- Better organization according to the problem that arises.
- It can be represented more easily graphically.

Option 6. Using unguided multigraph and Hash tables/maps

- Appropriate data structures are used.
- Better organization according to the problem that arises.

- It can be represented more easily graphically.
- Edges do not need to be guided.
- The access times to the hash table/map would be very fast.

Step 5. Evaluation and best solution selection

The criteria we chose in this case are the ones we list below. Next to each one a numerical value has been established with the aim of establishing a value that indicates which of the possible values of each criterion have the most tendency to be a better candidate (i.e., they are more desirable).

<u>Criteria:</u>

1. Criterion A. Organization. The option is organized and understandable for future improvements and data structures provide a logic organization when executing the program:

   [2] The structure to be used perfectly models the solution to the problem.
   [1] The structure to be used partially models the solution to the problem.

2. Criterion B. Ease of implementation.
   [1] Easy
   [0] Hard

3. Criterion C. Expanse.
   [3] The solution allows for representing ways in any sense and does not limit the implementation.
   [2] The solution gives rise to represent one-way roads by limiting the representation.
   [1] The solution is not able to represent paths or edges between places.

4. Criterion D. Efficiency.
   [1] The solution it is efficient in the basic operations of adding, removing or querying existence.
   [0] The solution is not efficient when it comes to basic query, add, and delete operations.

Evaluation

Evaluating the criteria of the last three alternatives, we have the following table:

	Criterion A	Criterion B	Criterion C	Criterion D	Total
Option 4. Using Hash tables/maps only	[1] The structure to be used partially models the solution to the problem.	[0] Hard	[1] The solution is not able to represent paths or edges between places.	[1] The solution it is efficient in the basic operations of adding, removing or querying existence.	3
Option 5. Using guided multigraph	[2] The structure to be used perfectly models the solution to the problem.	[0] Hard	[2] The solution gives rise to represent one-way roads by limiting the representation.	[0] The solution is not efficient when it comes to basic query, add, and delete operations.	4
Option 6. Using unguided multigraph and Hash tables/maps	[2] The structure to be used perfectly models the solution to the problem.	[1] Easy	[3] The solution allows for representing ways in any sense and does not limit the implementation.	[1] The solution it is efficient in the basic operations of adding, removing or querying existence.	7

Selection

According to the previous analysis of criteria, it is concluded that option 6 should be chosen since it accumulates a higher score compared to option 4 or 5. We can see that option 5 could be a good option too, but for this program we do not need to consider that the roads are one-way.

Step 6. Preparation of Reports and Specifications

Problem specification (input / output)

Problem: an efficient system to know the fastest or cheapest route for a tourist in the city of Cali.

Inputs: The tourist's budget or the estimated time required for the tour

Outputs: An interface that shows you the best route based on your requirements.

Step 7. Design implementation

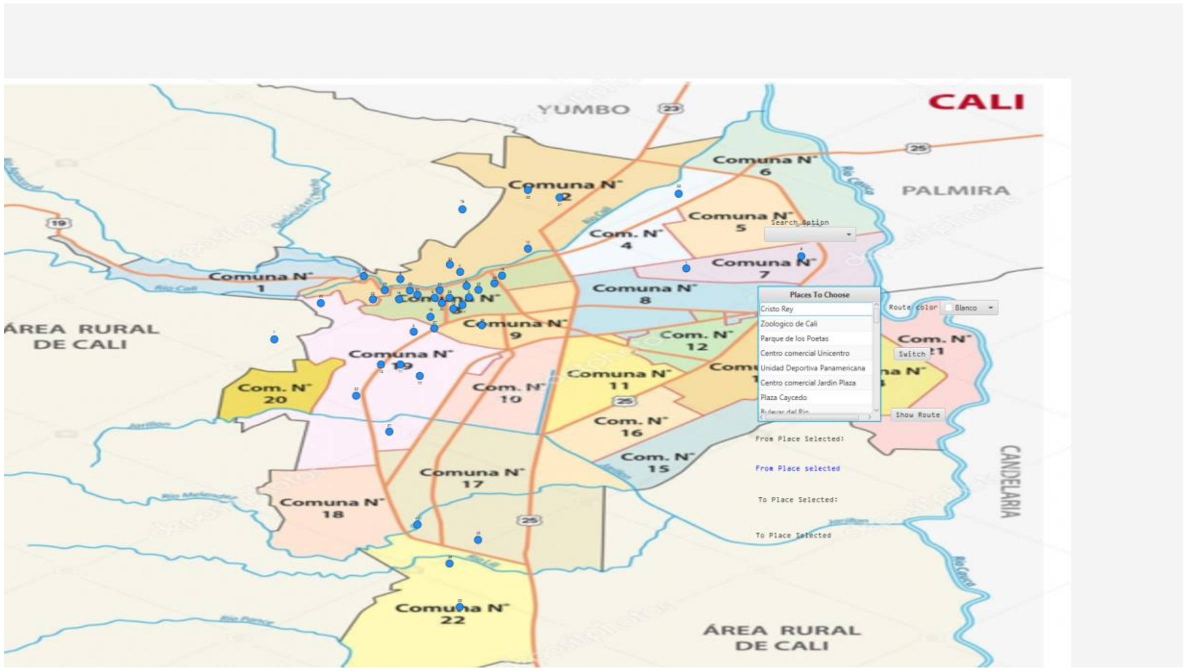
Implementation on Java programming language.

List of things to implement:

a) Representation of the most touristic places in Cali using graphs.
b) Import the data to later be stored in a hash table/map.
c) Go from a hash table/map to an array to implement algorithms that directly solve the problem.
d) Selection of the means of transport to use on the tour.
e) Receive the tourist's budget or the maximum time available for the tour.
f) Show the best route for the tourist.

GUI Mockups

When you load the program, you will see exactly as the image below. A map of Cali is showed with 42 enumerated touristic places. At the right you have a table with the list of these places. You can use the scroll to zoom it. The other controls are explained in the next pic.



Places To Choose
- Cristo Rey
- Zoologico de Cali
- Parque de los Poetas
- Centro comercial Unicentro
- Unidad Deportiva Panamericana
- Centro comercial Jardin Plaza
- Plaza Caycedo
- Bulevar del Rio

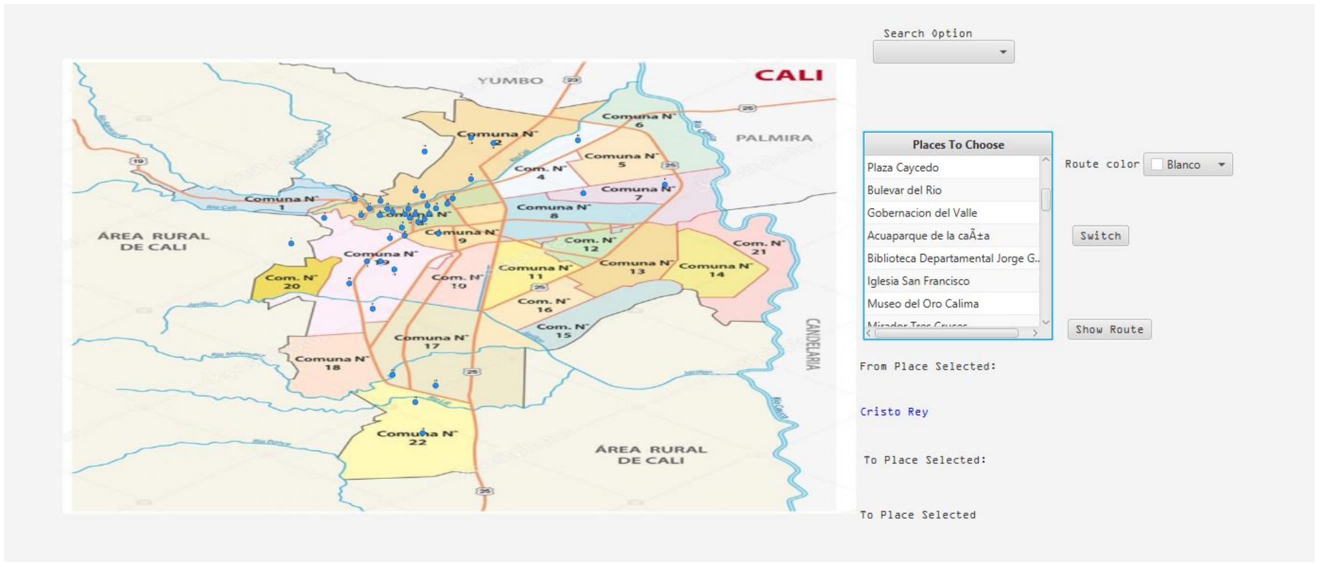
From Place Selected:

From Place selected

To Place Selected:

To Place Selected

When you click a place on the list for the first time, the program understand that is the place of departure. For select the arrival place please click the button "Switch" and then select another place on the list. (The letter color blue makes what selection you have activated).



Places To Choose
- Plaza Caycedo
- Bulevar del Rio
- Gobernacion del Valle
- Acuaparque de la caÃ±a
- Biblioteca Departamental Jorge G..
- Iglesia San Francisco
- Museo del Oro Calima
- Mirador Tres Cruces

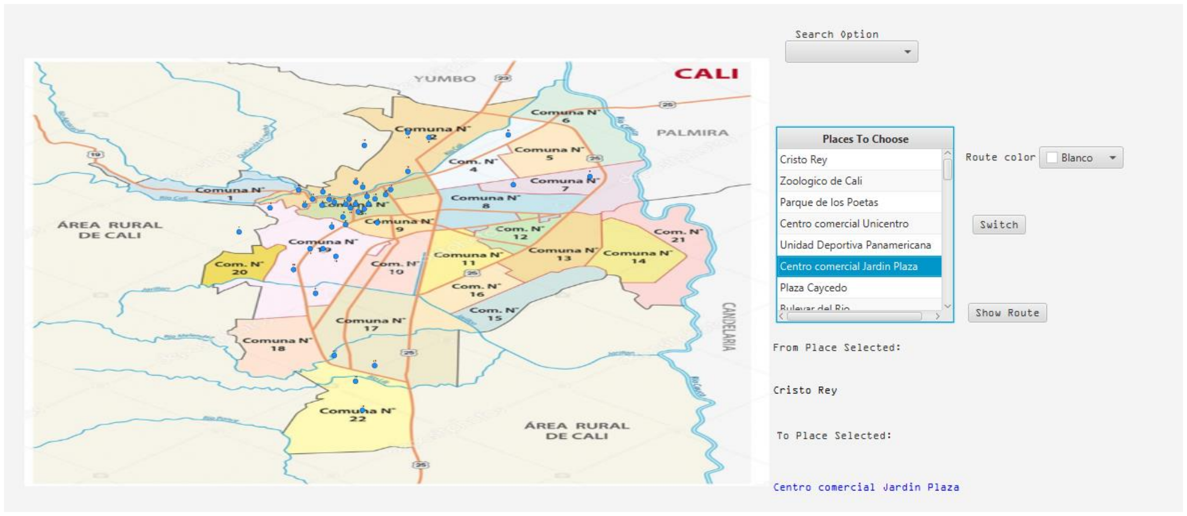
From Place Selected:

Cristo Rey

To Place Selected:

To Place Selected

As you can see, the arrival place for this example is “Centro commercial Jardín Plaza”



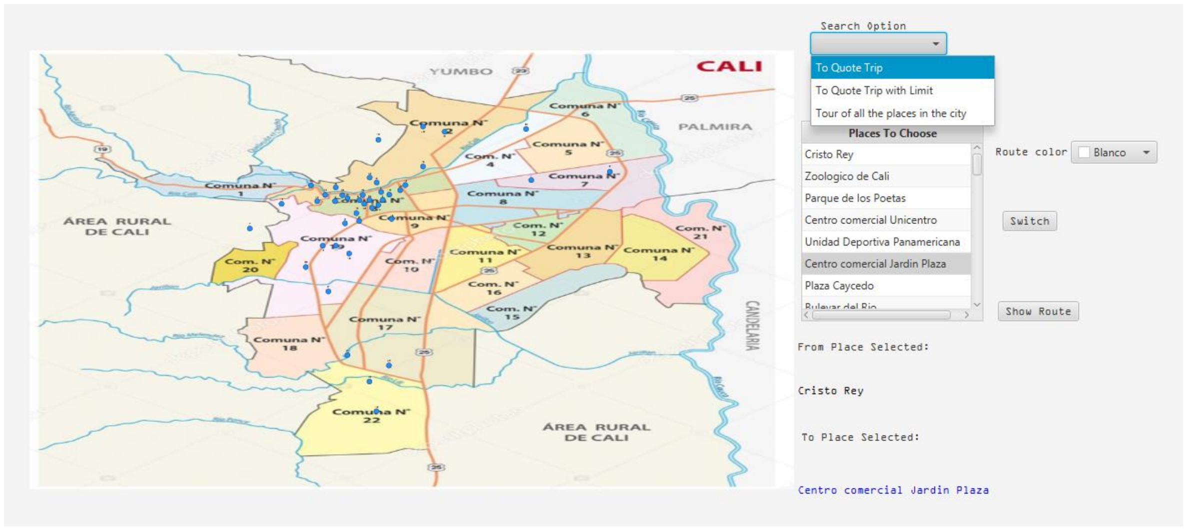
From Place Selected:

Cristo Rey

To Place Selected:

Centro comercial Jardin Plaza

Now, click the top bar called “search option.” You have 3 options, the first one let you know how many time or money you will spend for this trip. The second one is similar but now the program requires a time limitation or cost limit. The program will notify you if you can do that trip with that budget of time or money. Finally, the last one will give the best route taking into consideration the time or cost to travel around the 42 places.



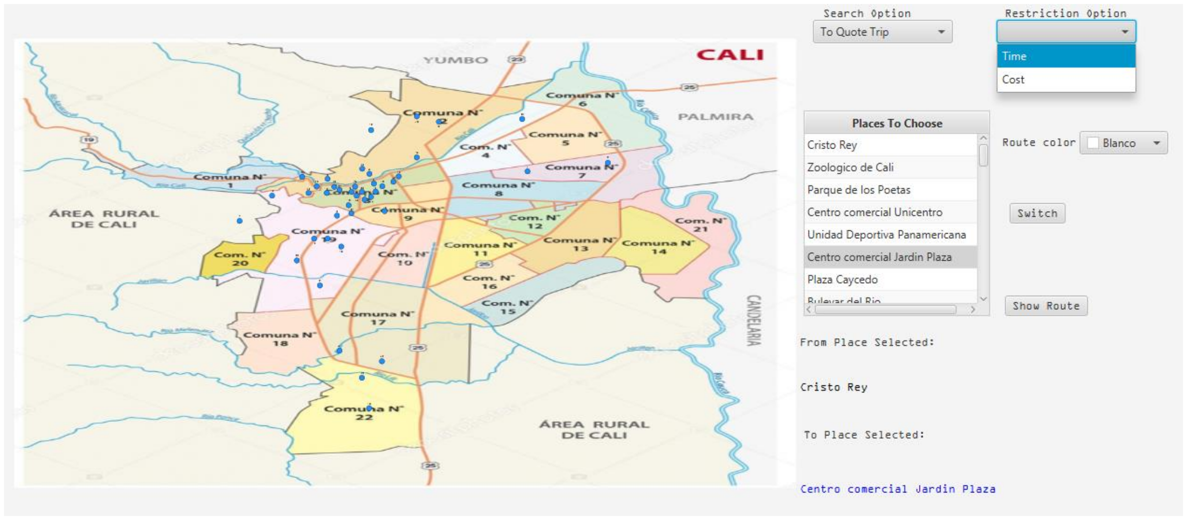
From Place Selected:

Cristo Rey

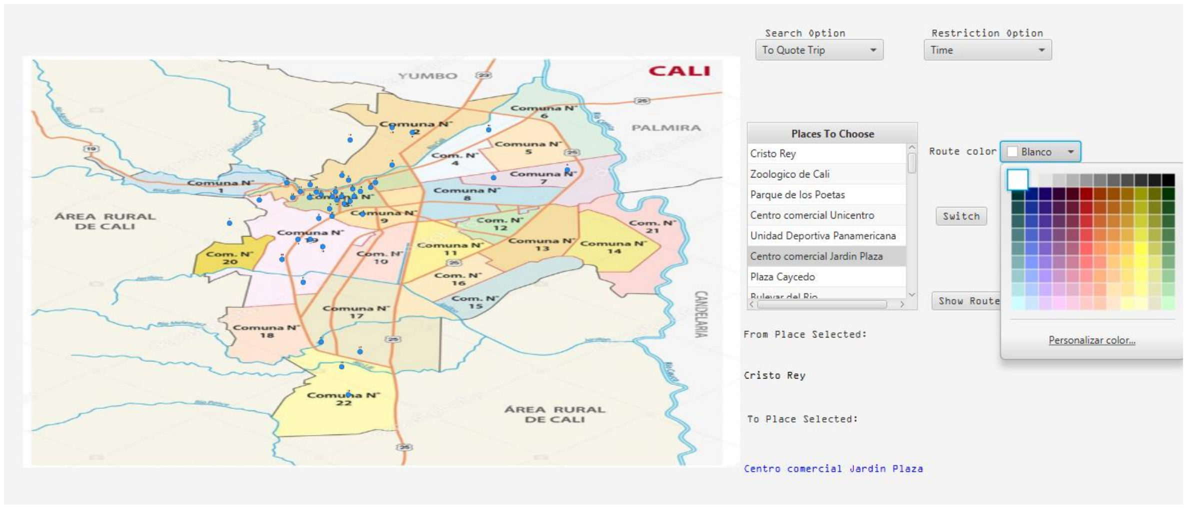
To Place Selected:

Centro comercial Jardin Plaza

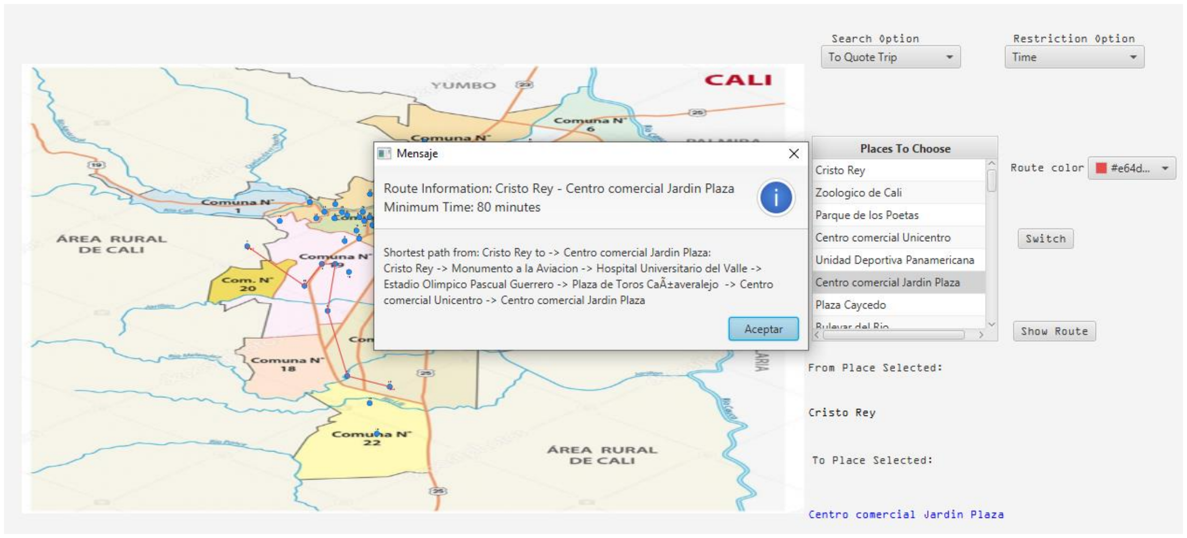
Another top bar will be displayed, and you can select between quote by time or cost



For best kindness to the user, you can select the color of the route that you want to do



Click on “show route” button and a message will appear, this notify about route information with the departure and arrival place, a minimum time for the route is showed too. In the bottom you will se the route that you must follow for this route is the fastest.



Search Option: To Quote Trip
Restriction Option: Time

Route Information: Cristo Rey - Centro comercial Jardin Plaza
Minimum Time: 80 minutes

Shortest path from: Cristo Rey to -> Centro comercial Jardin Plaza:
Cristo Rey -> Monumento a la Aviacion -> Hospital Universitario del Valle -> Estadio Olimpico Pascual Guerrero -> Plaza de Toros CaÃ±averalejo -> Centro comercial Unicentro -> Centro comercial Jardin Plaza

Places To Choose:
Cristo Rey
Zoologico de Cali
Parque de los Poetas
Centro comercial Unicentro
Unidad Deportiva Panamericana
Centro comercial Jardin Plaza
Plaza Caycedo
Bulevar del Rio

Route color: #e64d...

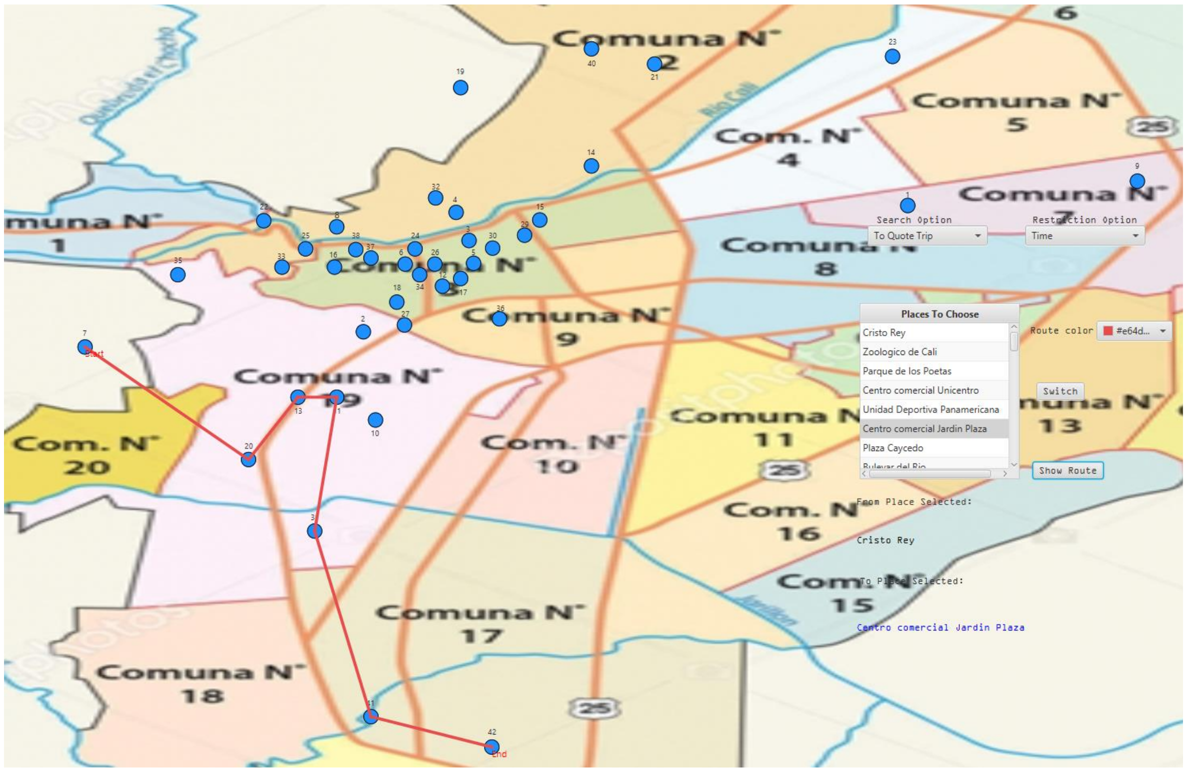
From Place Selected:

Cristo Rey

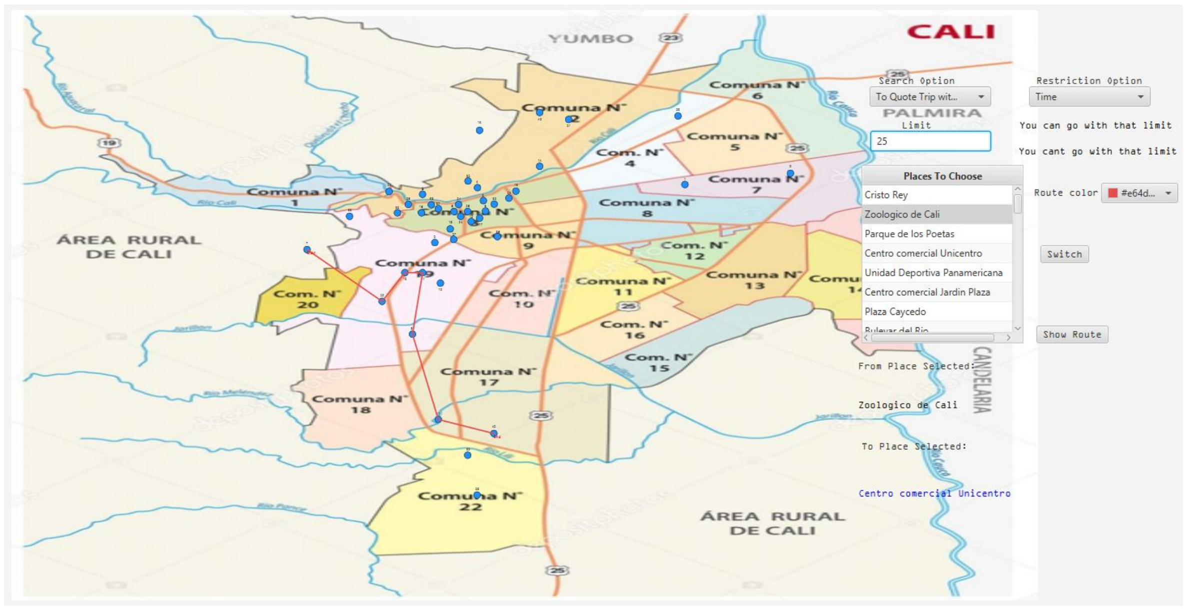
To Place Selected:

Centro comercial Jardin Plaza

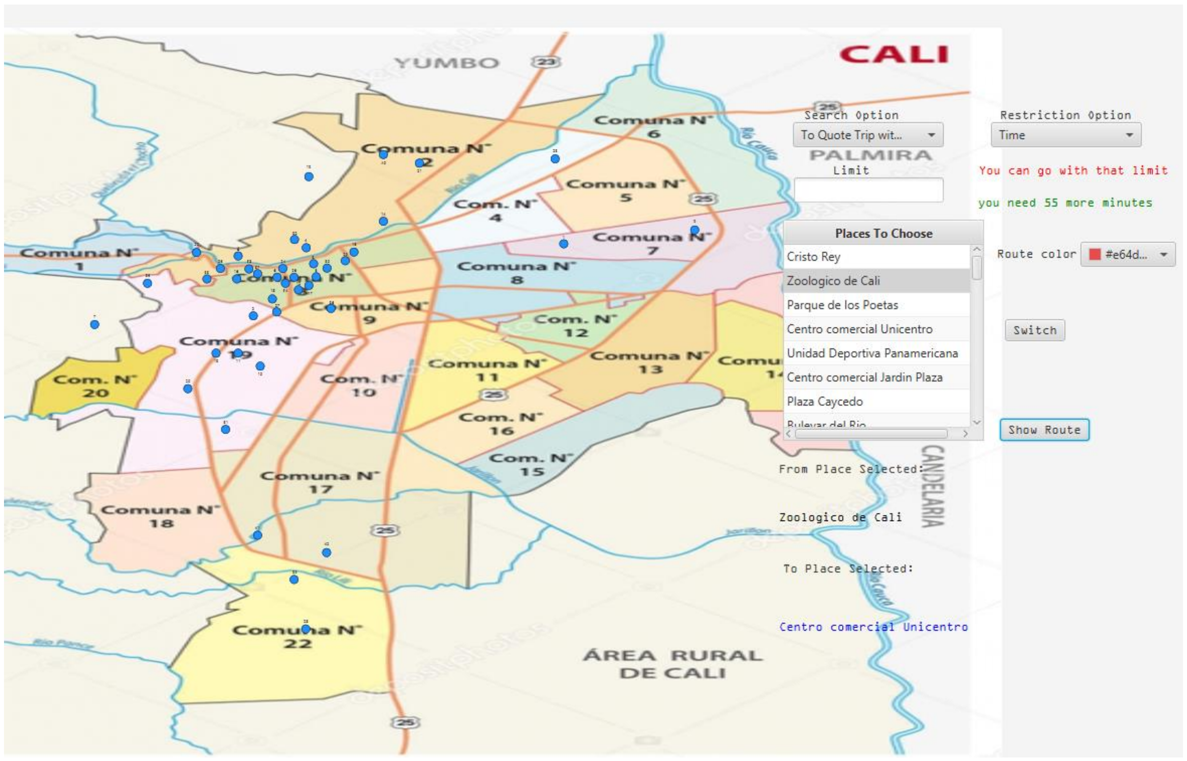
After click accept button, you will see on the map the route, the departure, and the arrival place too.



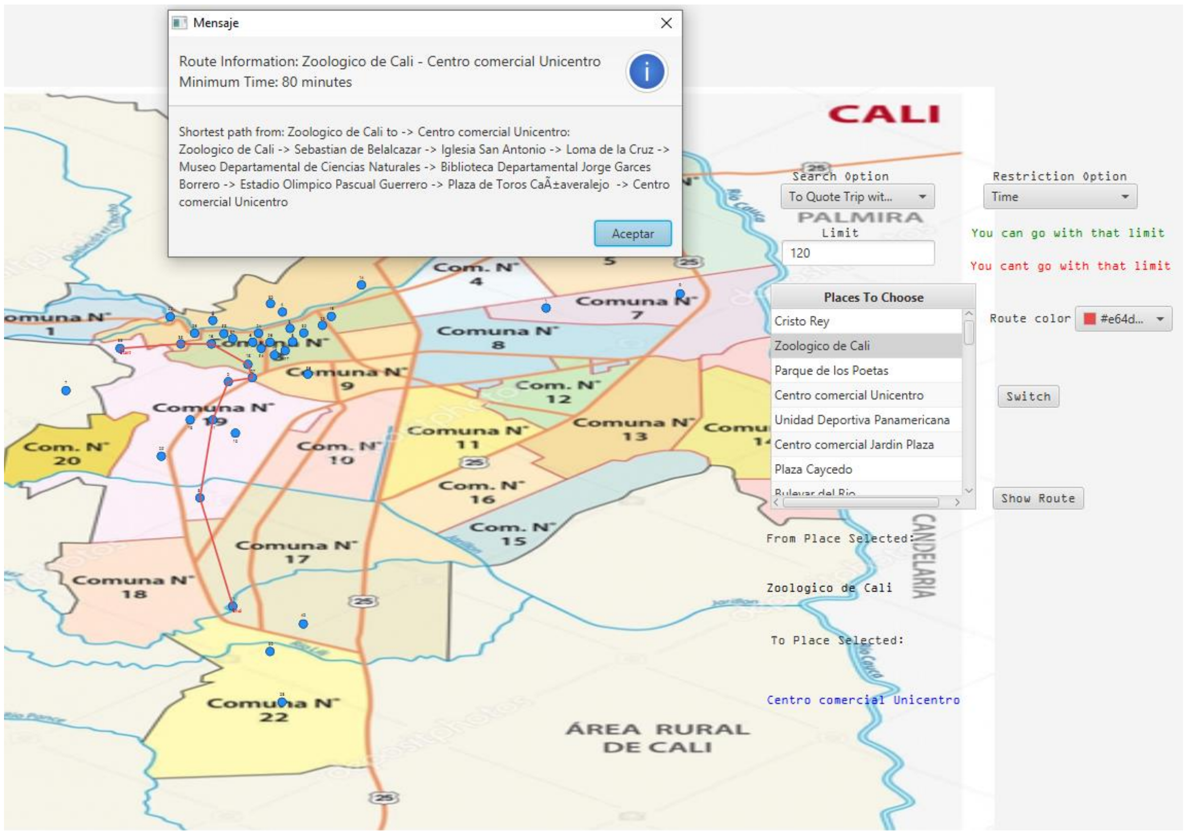
Select the second option and type the limit time that you want to spend in that trip, then click on show route button.


Search Option: To Quote Trip wit...
Restriction Option: Time
Limit: 25
You can go with that limit
You cant go with that limit
Places To Choose
Cristo Rey
Zoologico de Cali
Parque de los Poetas
Centro comercial Unicentro
Unidad Deportiva Panamericana
Centro comercial Jardin Plaza
Plaza Caycedo
Route color: #e64d...
From Place Selected:
Zoologico de Cali
To Place Selected:
Centro comercial Unicentro

The program will show you that is impossible travel in that time and shows how many times more you need.


Search Option: To Quote Trip wit...
Restriction Option: Time
Limit:
You can go with that limit
you need 55 more minutes
Places To Choose
Cristo Rey
Zoologico de Cali
Parque de los Poetas
Centro comercial Unicentro
Unidad Deportiva Panamericana
Centro comercial Jardin Plaza
Plaza Caycedo
Route color: #e64d...
From Place Selected:
Zoologico de Cali
To Place Selected:
Centro comercial Unicentro

After type a time that overspend the minimum time of this journey you will see on the right a green letter that you can go with that limit and another message with the route information from the “Zoologico de Cali” to “Centro commercial Unicentro”, then the route line is displayed on the map. (The same process for limited cost).


Mensaje

Route Information: Zoologico de Cali - Centro comercial Unicentro
Minimum Time: 80 minutes

Shortest path from: Zoologico de Cali to -> Centro comercial Unicentro:
Zoologico de Cali -> Sebastian de Belalcazar -> Iglesia San Antonio -> Loma de la Cruz -> Museo Departamental de Ciencias Naturales -> Biblioteca Departamental Jorge Garces Borrero -> Estadio Olimpico Pascual Guerrero -> Plaza de Toros CaÃ±averalejo -> Centro comercial Unicentro

Aceptar

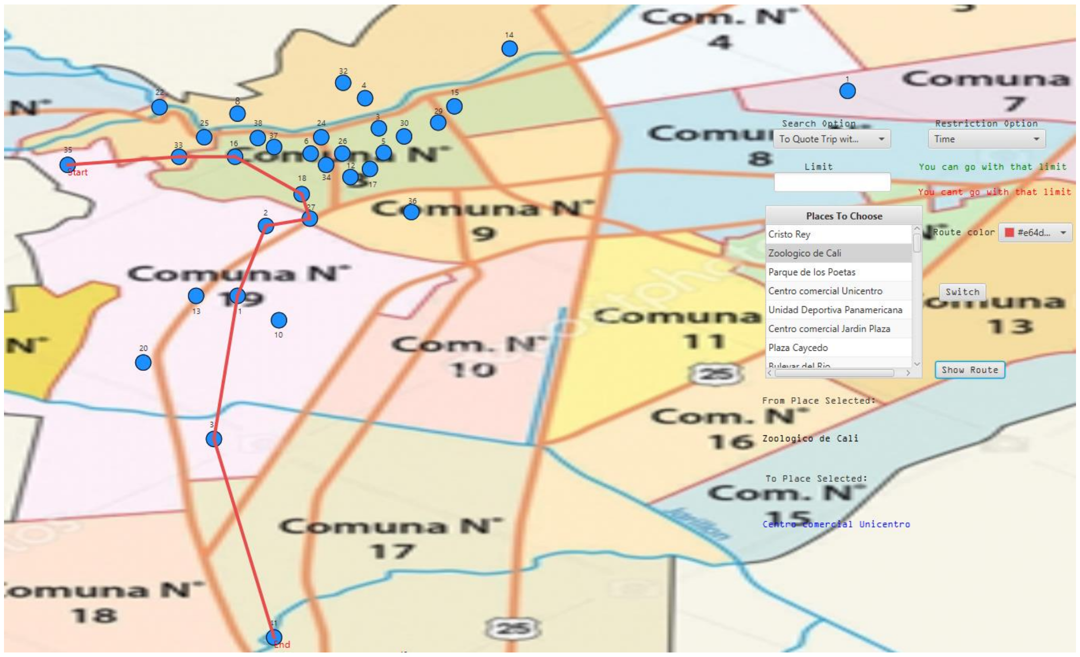
Search Option: To Quote Trip wit...
Restriction Option: Time
Limit: 120
You can go with that limit
You cant go with that limit

Places To Choose
- Cristo Rey
- Zoologico de Cali
- Parque de los Poetas
- Centro comercial Unicentro
- Unidad Deportiva Panamericana
- Centro comercial Jardin Plaza
- Plaza Caycedo
- Bulevar del Rio

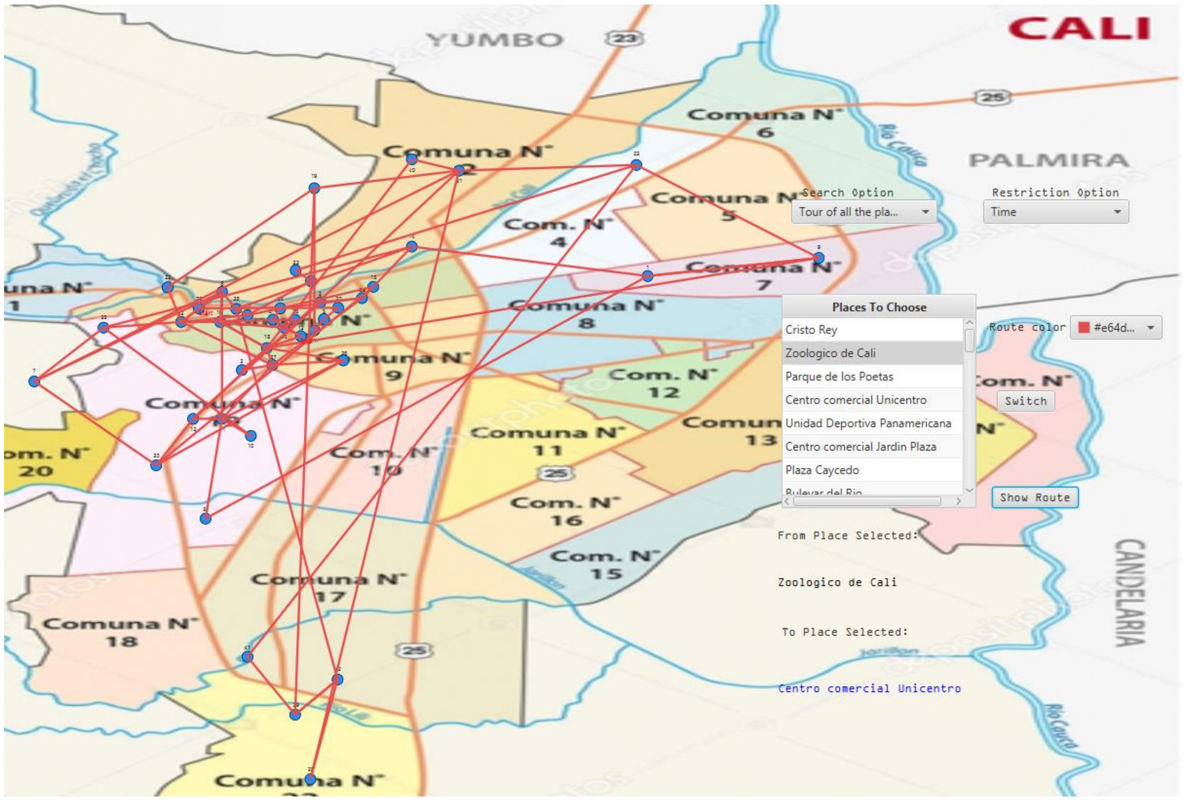
Route color: 🟥 #e64d...

From Place Selected:
Zoologico de Cali

To Place Selected:
Centro comercial Unicentro



Finally, the last option will show the total time or cost required to travel by all 42 places in Cali.



From Place Selected:

Zoologico de Cali

To Place Selected:

Centro comercial Unicentro

Class Diagram Design

For better reading convenience of the diagram, the link of the file that is housed in the docs folder is attached. Class diagram design link

QueueTest

Sceneries configuration

Name	**Class**	**Stage**
setupScenary1	QueueTest	A queue with 4 elements
setupScenary2	QueueTest	A queue of integer by a defined integer Array input = {1, 2, 3, 4}

Test cases design

Test objective: check if queue is peeking well				
Class	**Method**	**Setup**	**Input**	**Result**
Queue	peek	setupScenary1	None	See 1 out the queue since first integer added to queue

Test objective: check if queue is peeking well by creating queue due an array

Class	**Method**	**Setup**	**Input**	**Result**
Queue	peek	setupScenary2	None	See 1 out the queue since first integer added to queue

Test objective: check if queue is dequeuing well

Class	**Method**	**Setup**	**Input**	**Result**
Queue	dequeue & peek	setupScenary1	None	After dequeue see 2 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 3 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 4 when peeking.

Test objective: check if queue is dequeuing well by creating queue due an array

Class	**Method**	**Setup**	**Input**	**Result**

Queue	dequeue & peek	setupScenary1	None	After dequeue see 2 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 3 when peeking.
Queue	dequeue & peek	setupScenary1	None	After dequeue see 4 when peeking.

Test objective: check if queue is not empty

Class	Method	Setup	Input	Result
Queue	size	setupScenary1	None	True, queue is not empty and have size = 4

GraphTest

Sceneries configuration

Name	Class	Stage
setupScenary1	GraphTest	A Graph with 5 vertex

Test cases design

Test objective: check if vertices are storing well in a matrix from a hashMap

Class	Method	Setup	Input	Result
Graph	VertexToMatrixTime	setupScenary1	An expected int matrix verify = {{0, 18, 0, 0, 0}, {18, 0, 6, 0, 0}, {0, 6, 0, 21, 14}, {0, 0, 21, 0, 0}, {0, 0, 14, 0, 0}};	The vertex time stored in position [1][0] is the same of verify matrix
Graph	VertexToMatrixTime	setupScenary1	An expected int matrix verify = {{0, 18, 0, 0, 0}, {18, 0, 6, 0, 0}, {0, 6, 0, 21, 14}, {0, 0, 21, 0, 0}, {0, 0, 14, 0, 0}};	The vertex time stored in position [4][2] is the same of verify matrix

Test objective: check if vertices are storing well in a matrix from a hashMap

Class	Method	Setup	Input	Result
Graph	VertexToMatrixCost	setupScenary1	An expected int matrix verify = {{0, 8056, 0, 0, 0}, {8056, 0, 8188, 0, 0}, {0, 8188, 0, 5747, 8139}, {0, 0, 5747, 0, 0}, {0, 0, 8139, 0, 0}};	The vertex cost stored in position [1][0] is the same of verify matrix
Graph	VertexToMatrixCost	setupScenary1	An expected int matrix verify = {{0, 8056, 0, 0, 0}, {8056,	The vertex cost stored in position [4][2] is the same of verify matrix

			0, 8188, 0, 0}, {0, 8188, 0, 5747, 8139}, {0, 0, 5747, 0, 0}, {0, 0, 8139, 0, 0}};	

Test objective: check if edges are storing well in a matrix from a hashMap				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	edgesToMatrix	setupScenary1	An expected int matrix verify = {{0, 18, 0, 0, 0}, {18, 0, 6, 0, 0}, {0, 6, 0, 21, 14}, {0, 0, 21, 0, 0}, {0, 0, 14, 0, 0}};	The edge time stored in position [1][0] is the same of verify matrix
Graph	edgesToMatrix	setupScenary1	An expected int matrix verify = {{0, 18, 0, 0, 0}, {18, 0, 6, 0, 0}, {0, 6, 0, 21, 14}, {0, 0, 21, 0, 0}, {0, 0, 14, 0, 0}};	The edge time stored in position [4][2] is the same of verify matrix

Test objective: check if Floyd Warshall is doing fine for the time				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	floydWarshall	setupScenary1	An expected int matrix verify = {{0, 18, 24, 45, 38}, {18, 0, 6, 27, 20}, {24, 6, 0, 21, 14}, {45, 27, 21, 0, 35}, {38, 20, 14, 35, 0}};	The edge time stored in position [1][0] is the same of verify matrix
Graph	floydWarshall	setupScenary1	An expected int matrix verify = {{0, 18, 24, 45, 38}, {18, 0, 6, 27, 20}, {24, 6, 0, 21, 14}, {45, 27, 21, 0, 35}, {38, 20, 14, 35, 0}};	The edge time stored in position [4][2] is the same of verify matrix

Test objective: check if Floyd Warshall is doing fine for edges with time and cost				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	floydWarshallEdges	setupScenary1	An expected int matrix verify = {{0, 18, 24, 45, 38}, {18, 0, 6, 27, 20}, {24, 6, 0, 21, 14}, {45, 27, 21, 0, 35}, {38, 20, 14, 35, 0}};	The edge time stored in position [1][0] is the same of verify matrix

Graph	floydWarsh allEdges	setup Scena ry1	An expected int matrix verify = {{0, 18, 24, 45, 38}, {18, 0, 6, 27, 20}, {24, 6, 0, 21, 14}, {45, 27, 21, 0, 35}, {38, 20, 14, 35, 0}};	The edge time stored in position [4][2] is the same of verify matrix
Graph	floydWarsh allEdges	setup Scena ry1	An expected Edge matrix verify2 = {{null, 8056,16244, 21991,24383},{8056, null, 8188, 13935, 16327},{16244,8188 8, null, 5747, 8139}, {21991, 13935,5747, null, 13886}, {24383, 16327, 8139, 13886, null}};	The edge cost stored in position [1][0] is the same of verify2 matrix
Graph	floydWarsh allEdges	setup Scena ry1	An expected Edge matrix verify2 = {{null, 8056,16244, 21991,24383}, {8056, null, 8188, 13935, 16327}, {16244,81888, null, 5747, 8139}, {21991, 13935,5747, null, 13886}, {24383, 16327, 8139, 13886, null}};	The edge cost stored in position [4][2] is the same of verify2 matrix

Test objective: check if the pay function is working good

Class	Method	Setup	Input	Result
Graph	priceToPay	setup Scenary1	Path between “El Bochinche” and “Torre de Cali”	True, the cost is returned and is $8056
Graph	priceToPay	setup Scenary1	Path between “El Bochinche” and “La 14”	True, the cost is returned and is $13803

Test objective: test the calculation of the travel with a limited time				
Class	Method	Setup	Input	Result
Graph	travelWithTimeLimit	setup Scenary1	Path = “El Bochinche” to “La 14” with time limit 5 minutes	False, is impossible to travel this path in 5 minutes
Graph	travelWithTimeLimit	setup Scenary1	Path = “El Bochinche” to “Torre de Cali” with time limit 5 minutes	True, is possible travel this path in 20 minutes

Test objective: test the calculation of the travel with a limited cost				
Class	Method	Setup	Input	Result
Graph	priceToPayWithLimit	setupScenary1	Path = “El Bochinche” to “La portada al mar” with limit cost 5000	False, is impossible to travel this path with $5000
Graph	priceToPayWithLimit	setupScenary1	Path = “El Bochinche” to “Torre de Cali” with limite cost 9000	True, is possible travel this path with $9000

Test objective: check if Prim it is doing fine for the time				
Class	Method	Setup	Input	Result
Graph	primForTime	setupScenary1	An expected int matrix verify = {{0, 18, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 6, 0, 21, 0}, {0, 0, 0, 0, 0}, {0, 0, 14, 0, 0}}	The int time stored in position [1][0] is the same of verify matrix
Graph	primForTime	setupScenary1	An expected int matrix verify = {{0, 18, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 6, 0, 21, 0},	The int time stored in position [4][2] is the same of verify matrix

			{0, 0, 0, 0, 0}, {0, 0, 14, 0, 0}}	

Test objective: check if Prim it is doing fine for the cost				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	primForCost	setup Scenary1	An expected int matrix verify = {{0, 8056, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 5747, 0}, {0, 0, 0, 0, 0}, {0, 0, 8139, 0, 0}}	The int cost stored in position [1][0] is the same of verify matrix
Graph	primForCost	setup Scenary1	An expected int matrix verify = {{0, 8056, 0, 0, 0}, {0, 0, 0, 0, 0}, {0, 0, 0, 5747, 0}, {0, 0, 0, 0, 0}, {0, 0, 8139, 0, 0}}	The int cost stored in position [4][2] is the same of verify matrix

Test objective: check the construction of the pathways times				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	constructPathTime	setup Scenary1	Vector path = {"El Bochinche", “Torre de Cali”}	True, the path between El Bochinche and Torre de Cali is a direct way

Graph	constructPathTime	setup Scenary1	Vector path = {"El Bochinche", “Torre de Cali”, “Zoológico”, “Portada al mar”}	True, the path between El Bochinche and Portada al mar has several ways like Torre de Cali and Zoologico

Test objective: Check the adding vertex function

Class	Method	Setup	Input	Result
Graph	addVertex	setup Scenary1	New vertex = “Universidad Icesi”	True, the vertex to add to the graph is new
Graph	addVertex	setup Scenary1	New vertex = “La 14”	False, “La 14” vertex is already on the graph

Test objective: check the construction of the pathways costs

Class	Method	Setup	Input	Result
Graph	constructPathCost	setup Scenary1	Vector path = {"El Bochinche", “Torre de Cali”}	True, the path between El Bochinche and Torre de Cali is a direct way and the most economic
Graph	constructPathCost	setup Scenary1	Vector path = {"El Bochinche", “Torre de Cali”, “Zoológico”}	True, the path between El Bochinche and Torre de Cali has one more way like Torre de Cali. This way is the most economic

Test objective: check if vertices from a hashMap “vertices” are added to another hashMap by indicator as the key to “verticesv2”

Class	Method	Setup	Input	Result
Graph	verticesToHasMap2	setup Scenary1		True, both hasMaps have the same size and verticesv2 is not null

Test objective: check the search of the name by the vertex indicator				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	searchDueIndicator	setupScenary1	Int indicator = 4	True, the vertex is found, and the name is “Portada al mar”
Graph	searchDueIndicator	setupScenary1	Int indicator = 0	True, the vertex is found, and the name is “El Bochinche”

Test objective: check the search of the vertex by the vertex name				
Class	**Method**	**Setup**	**Input**	**Result**
Graph	searchVertex	setupScenary 1	"El Bochinche"	True, the vertex is found, and the name is "El Bochinche"
Graph	searchVertex	setupScenary 1	"Universidad Icesi"	False, the vertex with name "Universidad Icesi" is not found

VertexTest

Sceneries configuration

Name	Class	Stage
setupScenary1	VertexTest	A Graph with 5 vertex

Test objective: check the searching of the time on an edge				
Class	**Method**	**Setup**	**Input**	**Result**
Vertex	searchEdge	setup Scena ry1	“El Bochinche” and “Torre de Cali”	The Edge between “El Bochinche” and “Torre de Cali” was found and has a time of 18

Test objective: check adding a neighbour to a vertex				
Class	**Method**	**Setup**	**Input**	**Result**
Vertex	addNeighbo ur	setup Scena ry1	Vertex v1 = “Torre de Cali”	False, the neighbour “Torre de Cali” is already

				a neighbour of “El Bochinche”
Vertex	addNeighbour	setupScenary1	Vertex v1 = “Torre de Cali”	True, the neighbour “Torre de Cali” is added to neighbour of “La 14”

Test objective: check adding connection between vertices				
Class	**Method**	**Setup**	**Input**	**Result**
Vertex	addConnection	setupScenary1	Vertex v1 = “El Bochinche” Vertex v2 = “La 14”	True, the connection between “El Bochinche” to “La 14” is linked

ADT DESIGNS

Queue ADT		
Queue = {{$element_1, element_2, element_3$ … $element_n$}, front, back}		
{inv: $n \geq 0 \wedge n = size(stack) \wedge front = element_1 \wedge back = element_n$}		
Primitive Operations:		
• CreateQueue • IsEmpty • Enqueue • Dequeue • Front	• Queue • Queue x Element • Queue • Queue	→ Queue → boolean → Queue → Element → Element

CreateQueue()
“Make an empty Queue q”
{Pre: True}
{post: Queue q_1 = ∅}
Operation type: constructor

IsEmpty()
“Indicates if queue is empty or no”
{Pre: Queue q_1}
{post: True if q_1 = ∅, false if $q_1 \neq$ ∅}
Operation type: Analyzer

Enqueue (newElement)

“Adds the newElement to queue q_1”
{Pre: (Queue $q_1 = \emptyset \lor q_1 \neq \emptyset$) ∧ newElement }
{post: ($q_1 = \{element_1, element_2, element_3 \ldots\ element_n\}$ Ù back = newElement) $\lor(q_1 = \{newElement\})$}
Operation type: Modifier

Dequeue ()
“Remove the front element on the queue q_1”
{Pre: (Queue $q_1 \neq \emptyset$) Ù ($q_1 = \{element_1, element_2, element_3 \ldots\ element_n\}$)}
{post: $q_1 = \{element_1, element_2, element_3 \ldots\ element_{n-1}\}$}
Operation type: Modifier

Front ()
“Gives the value stored at the front of the queue”
{Pre: (Queue $q_1 \neq \emptyset$) Ù ($q_1 = \{element_1, element_2, element_3 \ldots\ element_n\}$)}
{post: $element_1$}
Operation type: Analyzer

Graph ADT		
Graph G(V, E) = {V(G) = $\{vertex_1, vertex_2, vertex_3 \ldots\ vertex_n\}$, E(G) = $\{(vertex_a, vertex_b), (vertex_b, vertex_d), \ldots (vertex_x, vertex_y)$ }}		
{inv: $\forall E(G)/(vertex_a, vertex_b) = (vertex_b, vertex_a) \land (vertex_a \cup vertex_b)$ ® $(vertex_a \neq vertex_b)$}		
Primitive Operations:		
• CreateGraph • floydWarshall • prim • addEdge • addVertex	• G(V,E) • G(V,E) • Vertex1 x Vertex2 x K name x double • E data x int indicator	→ Graph → Integer → MST(G) → E(G) → V(G)

CreateGraph()
“Make an empty Graph G”
{Pre: True}
{post: Graph $g = G(V, E)$}
Operation type: constructor

floydWarshall()
“Founds the minimum path between two vertices”
{Pre: Graph g}
{post: Integer as minimum value path}
Operation type: Analyzer

prim()
“Founds the minimum path traveling all vertices”
{Pre: Graph g}
{post: $MST(G)$}
Operation type: Analyzer

addEdge ()
“Adds an edge to the graph”
{Pre: (Graph g) $\wedge$ $(vertex_a, vertex_b)$}
{post: $True, the\ edge\ is\ added\ to\ graph\ g, E(g)$
Operation type: Modifier

addVertex()
“Adds a vertex to the graph”
{Pre: (Graph g)}
{post: $True, vertex\ is\ added\ to\ graph\ g, V(g)$}
Operation type: Modifier

Post-mortem Report

For this last time, the work was quite time consuming, initially because we decided to use hash maps to import vertices and edges. Then go to a graph represented by a matrix of time and costs. We believe that the complexity of making this program is reflected in the quality of the work that was done, it is the best program that we have done, its functionality and utility that it offers as a solution to the problem we are considering is quite effective and interactive with the user. It should be noted that if we had had much more time, the graphical interface could have been improved by offering greater coverage of information for the user. Our teamwork stood out throughout the semester, and we believe that this last program was the reflection of applying all the knowledge we learned in this course in an effective and useful way that allowed us to solve the problem of a tourist looking for economy in time. and cost to visit the city of Cali.