INTREPID CLIMBER

DIEGO FERNANDO HIDALGO LÓPEZ
LAURA DANIELA MARTÍNEZ ORTIZ

14 NOVEMBER 2021

ALGORITHMS AND DATA STRUCTURES

ICESI UNIVERSITY

Intrepid Climber

You climbed the highest mountain of your city. You are so excited about it that you need to tell it to all your friends, and you decided to start with those that are trying to be exactly where you are at this precise moment.
The mountain has N landmarks, and one of them is the top of the mountain, where you are now. Each of your friends that are climbing the mountain is in some other landmark, and you want to **visit all of them**. There are tracks that connect pairs of landmarks in such a way that there exists exactly one route (that is, a sequence of consecutive tracks) that goes down from the top of the mountain to each other landmark. To visit two friends in two different landmarks, you may have to go down some tracks, climb others, and go down others again. Going down the mountain is "easy", so you do not consume energy when you go down through the tracks. But **each time you climb a track, you consume a certain amount of energy**. After visiting all your friends, you can just sit and rest.

The first line contains two integers, representing respectively the number of landmarks and the number of your friends that are climbing the mountain. Landmarks are identified with distinct integers, being 1 the top of the mountain, where you initially are. Each of the next lines describes a different track with three integers A, B and C, indicating that there is a track from A to B that goes down and requires an amount C of energy to be climbed. The next line contains different integers representing the landmarks where your friends are. You may assume that the tracks between landmarks are such that there exists exactly one route that goes down from the top of the mountain to each other landmark.

Given the tracks between the landmarks, the energy required to climb them, and the landmarks where your friends are, compute the minimum total amount of energy required to visit all your friends from the top of the mountain and the path to achieve it.

Problem adapted from: 1751 - Intrepid Climber

Engineering method

Problem context

Someone climbed the highest mountain in their city. This person is so excited about it that they need to tell it to all their friends but wants to know the path to go in which the total amount of energy required is minimum.

Solution Development

In order to solve the problem previously presented, it will be used the steps indicated in the Method of the Engineering to develop a solution following a systematic approach and taking into account the proposed problematic situation.
Based on the description of the Engineering Method in the book "Introduction to Engineering" by Paul Wright, the steps to be followed in the development of the solution are presented as follows:

1. Problem identification
2. Compilation of necessary information
3. Search of creative solutions
4. Transition from the formulation of ideas to preliminary designs
5. Evaluation and selection of the best solution
6. Preparation of reports and specifications
7. Design implementation

1. Problem identification

At this stage, the needs of the problematic situation are recognized, as well as its symptoms and conditions under which it must be resolved.

<u>Identification of needs and conditions</u>

- It is necessary that the solution implements graphs and at least 2 of the algorithms learned.
- The climber wants to know the path to go in which the total amount of energy required is minimum.
- The climber wants to know the total amount of energy required to visit all their friends from the top of the mountain.
- The solution of the problem must allow the climber to enter:
    - Two integers representing respectively the number of landmarks and the number of their friends that are climbing the mountain.
    - Different tracks with three integers A, B and C, indicating that there is a track from A to B that goes down and requires an amount C of energy to be climbed.
    - In a line, different integers representing the landmarks where their friends are.

<u>Problem definition</u>

The climber requires the development of a program that lets them know the path to go in which they can tell all their friends that they were able to climb to the top of the mountain, but the path must require the minimum amount of energy.

2. Compilation of necessary information

In order to have complete clarity in the concepts involved, a search is made for the definitions of the terms most closely related to the problem presented.

Graph.

Are discrete structures consisting of vertices and edges that connect these vertices. There are different kinds of graphs, depending on whether edges have directions, whether multiple edges can connect the same pair of vertices, and whether loops are allowed.

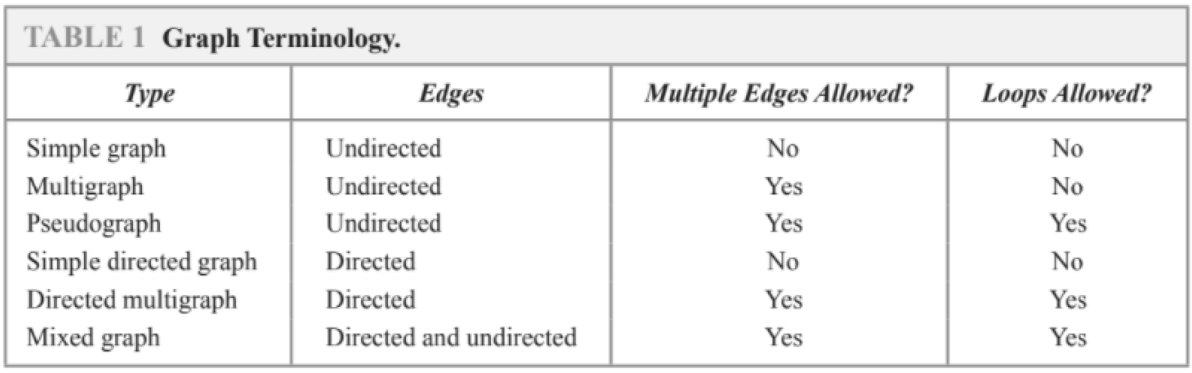


TABLE 1 **Graph Terminology.**

Type	*Edges*	*Multiple Edges Allowed?*	*Loops Allowed?*
Simple graph	Undirected	No	No
Multigraph	Undirected	Yes	No
Pseudograph	Undirected	Yes	Yes
Simple directed graph	Directed	No	No
Directed multigraph	Directed	Yes	Yes
Mixed graph	Directed and undirected	Yes	Yes

A graph G = (V, E) consists of V, a nonempty set of vertices (or nodes) and E, a set of edges. Each edge has either one or two vertices associated with it, called its endpoints. An edge is said to connect its endpoints.

A vertex (or node) of a graph is one of the objects that are connected together. The connections between the vertices are called edges or links.

A graph can be represented as:

- Adjacency lists.
- Adjacency matrix.
- Incidence matrices.

Source: Rosen, K. H. (2012). Discrete Mathematics and Its Applications (7.ª ed.). Recovered from:https://drive.google.com/drive/folders/1c-WHBgd4cRElaj9DFNrzo-xy1OGjVhok?zx=fj16nto5k23h

Graph search.

- *Breadth First Search (BFS)*

  BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes. As the name BFS suggests, you are required to traverse the graph breadthwise as follows: First move horizontally and visit all the nodes of the current layer then move to the next layer.

  A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing the same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

  To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue follows the First In First Out (FIFO) queuing method, and therefore, the neighbors of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

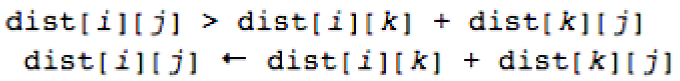
Source: Breadth First Search Tutorials & Notes | Algorithms. (2016). HackerEarth. Recovered from: https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/

- *Depth First Search (DFS)*
  The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.
  This recursive nature of DFS can be implemented using stacks. The basic idea is as follows: Pick a starting node and push all its adjacent nodes into a stack. Pop a node from the stack to select the next node to visit and push all its adjacent nodes into a stack. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.
  Source: Depth First Search Tutorials & Notes | Algorithms. (2016). HackerEarth. Recovered from: https://www.hackerearth.com/practice/algorithms/graphs/depth-first-search/tutorial/

Paths of minimum length.

- *Dijkstra algorithm*
  For a given source node in the graph, the algorithm finds the shortest path between that node and every other.
  We maintain two sets, one set contains vertices included in the shortest-path, another set includes vertices not yet included in the shortest-path. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.
  Source: GeeksforGeeks. (2021). Dijsktra's algorithm. Recovered from: https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/

- *Floyd-Warshall Algorithm*
  The Floyd Warshall Algorithm is for solving the All Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph.
  We initialize the solution matrix same as the input graph matrix as a first step. Then we update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and update all shortest paths which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number k as an intermediate vertex, we already have considered vertices {0, 1, 2, …, k-1} as intermediate vertices. For every pair (i, j) of the source and destination vertices respectively, there are two possible cases.
    1. k is not an intermediate vertex in the shortest path from i to j. We keep the value of dist[i][j] as it is.
    2. k is an intermediate vertex in shortest path from i to j. We update the value if:



```
dist[i][j] > dist[i][k] + dist[k][j]
 dist[i][j] ← dist[i][k] + dist[k][j]
```

Source: GeeksforGeeks. (2021c). Floyd Warshall Algorithm | DP-16. Recovered from: https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/

Kruskal algorithm.

Kruskal's algorithm finds a minimum spanning forest of an undirected edge-weighted graph. If the graph is connected, it finds a minimum spanning tree. (A minimum spanning tree of a connected graph is a subset of the edges that forms a tree that includes every vertex, where the sum of the weights of all the edges in the tree is minimized. For a disconnected graph, a minimum spanning forest is composed of a minimum spanning tree for each connected component.) It is a greedy algorithm in graph theory as in each step it adds the next lowest-weight edge that will not form a cycle to the minimum spanning forest.

A single graph can have many different spanning trees. A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree. The weight of a spanning tree is the sum of weights given to each edge of the spanning tree.

Source: GeeksforGeeks. (2021). Kruskal’s Minimum Spanning Tree Algorithm | Greedy Algo-2. Recovered from: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/?ref=lbp

Prim algorithm.

Is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

It starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets, and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Source: GeeksforGeeks. (2021). Prim’s Minimum Spanning Tree (MST) | Greedy Algo-5. Recovered from: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

Generics

Generics mean parameterized types. The idea is to allow type (Integer, String, … etc, and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is called a generic entity.

Source: Geeks for Geeks. (2021). Sorting Algorithms. Geeks for Geeks. Recovered from: https://www.geeksforgeeks.org/generics-in-java/

Abstract Data Type (ADT)

The abstract data type is a special kind of data type, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations. But how those operations are working is totally hidden from the user. The ADT is made of primitive data types, but operation logics are hidden.
Source: Chakraborty, A. (2019). Abstract Data Type in Data Structures. Tutorials Point. Recovered from: https://www.tutorialspoint.com/abstract-data-type-in-data-structures

3. **Search of creative solution**
At the end of the brainstorming, the following solutions are proposed:
    - *Alternative 1:* Weighted, multiple, undirected graph with a Dijkstra algorithm traversed by BFS.
    - *Alternative 2:* N-ary trees.
    - *Alternative 3:* Weighted, multiple, undirected graph with a Floyd-Warshall algorithm.
    - *Alternative 4:* Unweighted, multiple, undirected graph traversed by BFS.
    - *Alternative 5:* Linked lists (in an arraylist the linked lists are stored and within each list is stored a node with its adjacents)
    - *Alternative 6:* Unweighted, multiple, undirected graph traversed by DFS.

4. **Transition from the formulation of ideas to preliminary designs**
First, ideas that are not feasible are discarded. We decided to discard alternative 2 because el recorrido sería más costoso, pues tiene que ir visitando cada árbol para pasar de un vértice x a un vértice y.
Also, we decided to discard alternative 5 because es más costoso tanto temporalmente como espacialmente.
And, we decided to discard alternative 6 because tendría que recorrer todos los landmarks y luego buscar en la lista de adyacencia los pares de landmarks necesarios.

The review of the other 3 alternatives leads us to the following:
    - *Alternative 1*
        - It will give as a result the minimum energy necessary to visit the landmarks where their friends are.
        - Allows the climber to take different paths. Shows the energy needed to take a path.
        - It shows the result with all the required information.
    - *Alternative 3*
        - It will give as a result the minimum energy necessary to visit the landmarks where their friends are.
        - Shows the energy needed to take a path.
        - Find out all the paths between any pair of landmarks, and then it must find the path between the required landmarks.
    - *Alternative 4*
        - It doesn’t show the energy needed to take a path.
        - It will give as a result the path that needs to visit the least amount of landmarks.
        - The output is not presented with the necessary information, since it only shows the landmarks on the path and does not show the energy.

5. **Evaluation and selection of the best solution**

As the engineering design process evolves, the engineer can evaluate alternative ways to solve the problem in question. Commonly, the engineer abandons the design possibilities that are not promising, thus obtaining a set progressively smaller of options. Feedback, modification and evaluation can occur repeatedly as the device or system evolves from concept to final design.

<u>Criteria</u>
The criteria we chose in this case are those listed below. Next to each criteria, a numerical value has been established with the aim of establishing a weight that indicates which of the possible values of each criteria have more weight and, therefore, are more desirable.

- *Criteria A.* Waiting time.
  - [3] Little time.
  - [2] Normal time.
  - [1] Long time.
- *Criteria B.* Results.
  - [3] Satisfied.
  - [2] Partially satisfied.
  - [1] Not satisfied.
- *Criteria C.* Incluye toda la información de las entradas
  - [3] Satisfied.
  - [2] Partially satisfied.
  - [1] Not satisfied.

<u>Evaluation</u>
Evaluating the above criteria in the alternatives that are maintained, we obtain the following table:

	Criteria A	**Criteria B**	**Criteria C**	**Total**
<u>Alternative 1</u>	2	3	3	8
<u>Alternative 3</u>	1	3	3	7
<u>Alternative 4</u>	2	2	2	6

<u>Selection</u>
According to the previous evaluation, the Alternative 1 must be selected because it obtained the highest score according to the defined criteria.

6. **Preparation of reports and specifications**
   Review the functional and non-functional requirements section and the class diagram.

7. **Design implementation**
   Review the implemented project.

Requirement specification

Functional requirements

In order to correctly meet with the needs and functionalities required for this project, the system to be developed must be able to:

FR1: Manage the information of the landmarks.

- **FR1.1 -** Register a new landmark
- **FR1.2 -** Add the connections to the other landmarks.
- **FR1.3 -** Add the energy needed to climb the other landmarks.
- **FR1.4 -** Add a friend to a landmark.
- **FR1.5 -** Change the information of a landmark.

FR2: Show the information of the landmarks and the connections (the identifier, the friends located on the landmarks and the energy needed to climb).

FR3: Show the minimum energy required to get to the landmarks where the friends are located.

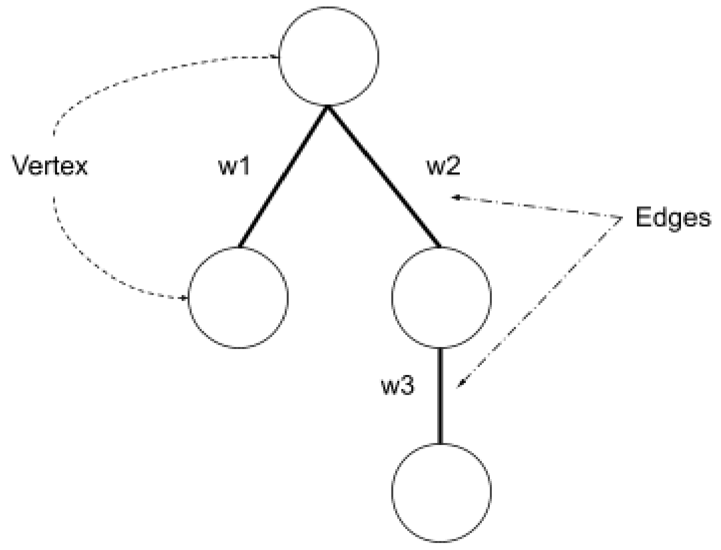
FR4: Register and show the information from a file with the landmarks, the location of the friends and the energy required to climb them.

Non-functional requirements

In order to guarantee the correct operation of the system and assure the quality of the software, the system must have the next validations:

NFR1: Tests. Implement tests to make sure that the methods work correctly.
NFR2: Generics.

Abstract Data Type (ADT) Design

Graph ADT
 G = (V, E) V is a set of vertex, V = $\{v_1, v_2, \ldots, v_n\}$ E is a collection of edges, E = $\{e_1, e_2, \ldots, e_m\}$ W is the weight.
{inv: $e_i\ (v_k, v_j)$ V can't be empty $w_r > 0$}
Primitive operations:

createGraph:		→Graph
addVertex:	Graph x Vertex	→Graph
addEdge:	Vertex x Vertex x Weight	→ Graph
dijkstra:	intSource x Graph	→ Graph
bfs:	intSource x Graph	→ Tree
getVertex:	Graph	→ List
getEdge:	Graph	→ List

CreateGraph() → Constructor
"Creates a new empty graph"
{pre: TRUE}
{post: The graph has been initialized}

AddVertex(Graph, Vertex) → Modifier
“Adds a new vertex to the graph”
{pre: Graph != null ^ v_1!= null}
{post: New vertex inserted in the graph}

AddEdge(Vertex,Vertex, Weight) → Modifier
“Adds a new edge, with the respective weight, to the graph connecting two existing vertices”
{pre: v_1!= null ^ v_2 != null ^ weight ≥0 ^ Graph != null}
{post: New edge inserted in the graph}

Dijkstra(intSource, Graph) → Analyzer
“Uses the Dijkstra algorithm to find the shortest paths between nodes in a graph”
{pre: Graph != null ^ V = {v_1, v_2,..., v_n} != null}
{post: Returns the shortest path between a given source node and every other}

BFS(intSource, Graph) → Analyzer
“Uses the Breadth-first search algorithm starting from a specific vertex”
{pre: Graph != null ^ V = {v_1, v_2,..., v_n} != null}
{post: Returns the parent links trace the shortest path back to root}

getVertex(Graph) → Analyzer
“Returns the vertices of the graph ”
{pre: Graph != null}
{post: List of the vertices of the graph}

getEdge(Graph) → Analyzer
“Returns the edges of the graph ”
{pre: Graph != null}
{post: List of the edges of the graph}

Vertex ADT
V is a set of vertex, V = {$v_1, v_2,..., v_n$} v_1= {value = <v>, dist = <d>}
{inv: value != null dist $\geq$ 0}
Primitive operations:

createVertex:	Value	→Vertex
getValue:	Vertex	→Integer
getDist:	Vertex	→ Integer
setValue:	Vertex x Value	→ Vertex
setDist:	Vertex x Dist	→ Vertex

CreateVertex(Value) → Constructor
"Creates a new vertex with a value"
{pre: value != null}
{post: The vertex has been initialized}

GetValue(Vertex) → Analyzer
"Returns the vertex's value"
{pre: vertex != null}
{post: Value of the specific vertex}

GetDist(Vertex) → Analyzer
"Returns the vertex's distance"
{pre: vertex != null}
{post: Distance of the specific vertex}

SetValue(Vertex, Value) → Modifier
"Modify the value of the specific vertex"
{pre: vertex != null ^ value != null}
{post: v_1= {value = <v>, dist = <d>}

SetDist(Vertex, Dist) → Modifier
“Modify the distance of the specific vertex”
{pre: vertex != null ^ dist != null}
{post: v_1= {value = <v>, dist = <d>}

Adjacency lists ADT
V is a set of vertex, V = {v_1, v_2,..., v_n}
{inv: V can't be empty}

Primitive operations:

createAdjList:		→AdjList
addVertex:	AdjList x Vertex	→AdjList
addEdge:	Vertex x Vertex x Weight	→ AdjList
dijkstra:	intSource x AdjList	→ AdjList
bfs:	intSource x AdjList	→ AdjList
getVertex:	AdjList	→ List

CreateAdjList() → Constructor
“Creates a new empty adjacency list”
{pre: TRUE}
{post: The adjacency list has been initialized}

AddVertex(AdjList, Vertex) → Modifier
“Adds a new vertex to the adjacency list”
{pre: AdjList != null ^ v_1!= null}
{post: New vertex inserted in the AdjList}

AddEdge(Vertex,Vertex, Weight) → Modifier
“Adds a new edge, with the respective weight, to the adjacency list connecting two existing vertices”
{pre: v_1!= null ^ v_2 != null ^ weight ≥0 ^ AdjList != null}
{post: New edge inserted in the adjacency list}

Dijkstra(intSource, AdjList) → Analyzer
“Uses the Dijkstra algorithm to find the shortest paths between nodes in a adjacency list”
{pre: AdjList != null ^ V = $\{v_1, v_2,..., v_n\}$!= null}
{post: Returns the shortest path between a given source node and every other}

BFS(intSource, AdjList) → Analyzer
“Uses the Breadth-first search algorithm starting from a specific vertex”
{pre: AdjList != null ^ V = $\{v_1, v_2,..., v_n\}$!= null}
{post: Returns the parent links trace the shortest path back to root}

getVertex(AdjList) → Analyzer
“Returns the vertices of the adjacency list”
{pre: AdjList != null}
{post: List of the vertices of the adjacency list}