Voracious and Heuristic Algorithms: A focus on the Minimum Path Problem

Algoritmos Voraces y Heurísticas: Un enfoque en el problema de la Ruta mínima

DOI: http://doi.org/10.17981/ingecuc.16.2.2020.05

Scientific Research Article. Date Received: 08/12/2020. Acceptance Date: 07/05/2020.

Luis Adrián Lasso-Cardona
Universidad del Valle. Buga (Colombia)
luis.lasso@correounivalle.edu.co

Diego Fernando Franco-Ocampo
Universidad del Valle. Buga (Colombia)
diego.franco@correounivalle.edu.co

Alexander Agudelo-Acevedo
Universidad del Valle. Buga (Colombia)
alexander.agudelo@correounivalle.edu.co

Para citar este artículo:
L. Lasso-Cardona, D. Franco-Ocampo & A. Agudelo-Acevedo, "Voracious and Heuristic Algorithms: A focus on the Minimum Path Problem", *INGE CUC*, vol. 16, no. 2, pp. 67–85, 2020. DOI: http://doi.org/10.17981/ingecuc.16.2.2020.05

Abstract

Introduction— The problem of the shortest route or minimum cost route, has been one of the topics most studied by areas of knowledge such as Operations Research, Computer Science and Decision, Telecommunications, Plant Distribution, Planning of Projects, among others, searching, for example: optimize and reduce the costs that represent the distribution of goods, obtain the minimum amount of time necessary to complete a project, or calculate the shortest possible route between computers connected to a network.

Objective— We will study the behavior of three voracious algorithms that allow us to calculate the minimum cost route between two points (initial state and objective state) in a weighted graph and with heuristics.

Methodoly— Was implemented in Java, and the Greedy, A* and Dijkstra algorithms were adjusted to the problem in question. Subsequently, two instance cases were designed, one negative and one positive.

Results— In the negative instance results the heuristic of the node was modified to allow the selected algorithm to escape from local optima and thus obtain a complete result, that is to say reach the objective state, which, in some cases, will not necessarily be the most optimal result.

Conclusions— By comparing the three algorithms, it was determined that the Dijkstra algorithm always yields complete and optimal results. For its part, Greedy and A*, need heuristics to reach a complete result, but not optimal.

Keywords— Weighted graph; cost matrix; adjacency matrix; optimal route; voracious algorithms; greedy search; heuristics; Greedy; A-star; Dijkstra

Resumen

Introducción— El problema de la ruta más corta o ruta de mínimo costo, ha sido uno de los temas más estudiados por áreas del conocimiento como la Investigación de Operaciones, la Ciencias de la Computación y la Decisión, las Telecomunicaciones, la Distribución en Planta, la Planeación de Proyectos, entre otras, buscando, por ejemplo: optimizar y reducir los costos que representan la distribución de mercancías, obtener la mínima cantidad de tiempo necesaria para finalizar un proyecto, o calcular la ruta más corta posible entre ordenadores conectados a una red.

Objetivo— Estudiar el comportamiento de tres algoritmos voraces que permiten calcular la ruta de mínimo costo entre dos puntos (estado inicial y estado objetivo) en un grafo ponderado y con heurísticas.

Metodología— Se implementó una aplicación en Java, y se ajustaron los algoritmos Greedy, A* y Dijkstra al problema en cuestión. Posteriormente se diseñaron dos casos de instancia, una negativa y otra positiva.

Resultados— En los resultados de instancia negativa se modificó la heurística del nodo para permitir al algoritmo seleccionado escapar de óptimos locales y así, obtener un resultado completo, es decir llegar al estado objetivo, que, en algunas ocasiones, no necesariamente será el resultado más óptimo.

Conclusiones— Mediante la comparación entre los tres algoritmos se pudo determinar que el algoritmo de Dijkstra siempre arroja resultados completos y óptimos. Por su parte, Greedy y A*, necesitan de heurísticas para llegar a un resultado completo, pero no óptimo.

Palabras clave— Grafo ponderado; matriz de costos; matriz de adyacencia; ruta óptima; algoritmos voraces; búsqueda codiciosa; heurística; Greedy; A-star; Dijkstra

I. Introduction

The problem of the shortest route (optima), has been one of the topics most studied by areas of knowledge such as Operations Research, Telecommunications, Plant Distribution, Project Planning, among others. For example, one of the most common problems faced by logistics is to optimize and reduce the costs that represent the distribution of goods and vehicle maintenance. For business and industry, it is essential that the products or services are delivered on time at the best price using the shortest route or minimum cost available [1]. In general terms, network problems can be classified essentially into five areas: shorter route, maximum flow, minimum expansion tree, minimum cost flow, and project planning and control. In this grouping, the problem of the shortest route is considered by researchers as a central problem within the network area, due to the variety of practical applications, the existence of efficient solution methods and the application of subroutines in the search for a good solution in complex problems [2]. In graph theory there are several techniques and algorithms such as the Ford-Fulkerson to solve the problem of maximum flow, the Prim algorithm for the problem of the minimum cost generating tree, or the Dijkstra algorithm to find the shortest path between two points, given an origin and a destination [3].

The present investigation studied the behavior of three voracious algorithms that allow the calculation of the minimum cost path between two points (initial state and objective state) in a weighted and heuristic graph of 100 nodes and 160 edges that represent the paths between the nodes. To achieve this goal, a Java application was implemented, and the Greedy, A* and Dijkstra algorithms were modified in such a way that they allowed to include additional variables such as the heuristic value of each node. Furthermore, Dijkstra's algorithm, which generally calculates the minimum path from an initial node to all the nodes of the graph, was improved by making the route to be calculated be between an initial node and a target node.

The research began with a review of the literature that allowed knowing concepts associated with the study problem, such as graph theory, and related characteristics. Likewise, the most common algorithms to find the least cost path between two nodes were identified and described. Subsequently, the algorithms were implemented with the proposed improvements, in such a way that they allowed the inclusion of additional variables that expanded their functionality and improved their behavior. To check the correct operation of the algorithms, two test cases were carried out. The first of negative instance, that is, when it is not possible to reach a target node, and the second of positive instance, that is, when it is possible to reach a target node, but not exactly the least cost in some scenarios. In addition to the NI and PI cases, two additional test cases were carried out, in order to measure and average certain characteristics in the behavior of the implemented algorithms. Finally, the conclusions and opportunities for future research were established.

II. Methodology

The project basically developed in four phases: a) Data collection: an information search was carried out in primary sources, such as bibliographic databases and scientific articles under the terms "graph theory", "voracious algorithms", "greedy algorithm", "A-star algorithm", "Dijkstra algorithm", "minimum path problem" and "optimal route; b) Construction of the theoretical framework: the concepts related to the study problem were defined that laid the foundations for the implementation and improvement of the algorithms used; c) Implementation: an application was developed in the Java programming language, and the algorithms were implemented with the improvements proposed according to the design built in the previous phase; and d) Commissioning and testing: two test cases were carried out. The first of negative instance, that is, when it is not possible to reach a target node, and the second of positive instance, that is, when it is possible to reach a target node, but not exactly the least cost in some scenarios. Likewise, two additional test cases were carried out, with the purpose of measuring and averaging certain characteristics in the behavior of the implemented algorithms.

III. Theoretical framework

A. *Definition of the Graph Concept*

In 1736, Euler Örst introduced the notion of graphs, by solving the problem of the Konigsberg bridge. Graphics theory is a useful tool for solving problems in different areas such as geometry, algebra, number theory, operations research, optimization and computer science [4].

A graph $G = (V, E)$ is an ordered pair of a finite set of vertices $V = \{v_1, v_2, ..., v_n\}$ and a set of edges $E = \{e_1, e_2, ..., e_n\}$, such that every edge is an one- or two-element subset of the vertex set. We call an edge $e \in E$ a loop if it is an one-element subset of V [5].

B. *Undirected Graph*

A undirected graph is a graph $G = (V, E)$, where V is the set of vertices (nodes) and E is the set of edges (arcs). In other words, for a graph G, $V(G)$ and $E(G)$ respectively denote its vertex-set and the edge-set [6]. A graph is said to be undirected if $(v_i, v_j) \in E \Longleftrightarrow (v_j, v_i) \in E$.

When G is unweighted or undirected, we define its adjacency matrix as the $n \times n$ matrix A with $A_{ij} = 1$ if $v_j \sim v_i$, and 0 otherwise. For undirected graphs A is symmetric. For weighted graphs, $A_{ij} = w_{ji}$ [7].

C. *Directed Graph*

A digraph or directed graph D consists of a non-empty set $V(G)$ of nodes, a set $E(G)$ of directed edges, and an incidence function φD that joins each arc of D with an ordered pair of (not necessary distinct) nodes of D. Usually the directed graph is denoted by $D = (V, E, \varphi D)$. If e is an arc such that $(e) = (u,)$, then u is called the initial vertex of e and v is called the terminal vertex of also e is an arc from u to v. If $(e) = (u)$, then the arc e is called a loop or self-loop [8].

D. *Weighted graph*

A weighted graph is a triple $G = (V, E, w)$ is a graph in which every edge e is assigned a non negative number $w(e)$, called the weight of e. A path in a weighted graph G (weighted path) is a sequence of vertices and edges with a weight assigned to each edge [9]. In which $w: E \to R_{>0}$ is a weight function, where $xy \in E$, and $w_{xy} = w_{yx} = 0$ [10].

The set of all the neighbors of a node v in G is denoted by $N_G(v)$ or simply $N(v)$, and its cardinality by $d_G(v)$ or $d(v)$ [11].

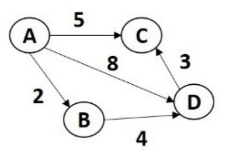


Fig. 1. Weighted graph.
Source: Author.

E. *Heuristic concept*

There are such complex problems that it is not possible to solve them to find an optimal solution, but where it is still important to find a good feasible solution that is reasonably close to being optimal. In general, a heuristic method or also called approximation algorithm, is a procedure that tries to discover a very "good" feasible solution, but not necessarily an optimal solution, for the problem. That is why such methods are designed to solve a particular type of problem, such as the problem of the shortest route [12], where the use of a heuristic or meta-heuristic provides mechanisms to explore regions and escape from the bad local optimum. quality that often differs considerably from the overall optimal value [13].

IV. Problem formulation

Given a set of nodes (cities) with values that represent some degree of importance (heuristics), and their respective edges (roads that connect the cities) with weight (length in kilometers), trace the route that you send to a vehicle throughout of the minimum length trajectory between two cities (origin and destination) obtaining a Positive Instance (*PI*) or a Negative Instance (*NI*), using a Greedy search, the algorithm of A* and the Dijkstra algorithm.

It is said that a *PI* is the one where the algorithm was able to trace a route (not necessarily the most optimal one) between the city of origin (Initial State - *IS*) and the city of destination (Objective State - *OS*). Otherwise, an *NI* would be obtained. The latter case is very recurrent when using the Greedy search or the A* algorithm.

Once the route between the *IS* and the *OS* is obtained, establish if this route was a *PI* or an *NI*, comparing the results of the three algorithms, and allowing to modify the heuristic of the nodes in those cases when the algorithm used is trapped in optimal local, so that it is possible to reach a solution, but not precisely the most optimal in some cases.

According to the results obtained in the theoretical framework, two types of search techniques will be used on informed graphs, that is, with a cost and heuristics associated with each node of the graph. The first technique focused on using the voracious algorithms A* and Greedy, which are guided by a heuristic function that do not always result in the lowest cost path, that is, they are not complete or optimal. In the second technique, the uniform cost search was used, taking into account only the costs associated with the edges (paths) between nodes, and that yielded a complete and optimal result. In this case, Dijkstra's algorithm was modified, which allows calculating the least-cost route from an initial node to the rest of the graph's nodes, representing a considerable computational and storage expense in large graphs. Modifying the base algorithm allowed the initial state node and the target state node as inputs. In addition, storage structures were used for the adjacent unselected nodes, the lowest node, the weights of such nodes, and the selected node, making the algorithm more selective.

A. *Greedy Search*

Voracious, avid, fast-forward or greedy algorithms are algorithms that are used primarily to solve optimization problems by making short-range decisions based on immediately available information, regardless of future consequences [14]. A Greedy algorithm is an algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum. In a Greedy algorithm, the optimal solution is built up one piece at a time. At each stage, the best feasible candidate is chosen as the next piece of the solution [15].

Greedy search (in general) proceeds at each step by evaluating each neighbor of the current state, and moving to the one with the highest score if doing so improves the score. The set of neighbors of each state in the search defines the search space [16]. In a weighted graph, a local search iteratively determines a better solution of the environment of the current solution.

An inherent limitation of greedy algorithms is that they lack the foresight to choose suboptimal solutions in the current iteration that will allow for better solutions later [17]. Moreover, if the number of such neighbor states grows very large, or if each neighbor state takes too long to evaluate, even the simple greedy algorithm may not terminate quickly enough [16].

A Greedy algorithm works if a problem exhibit the following two properties: 1) Greedy Choice Property: A globally optimal solution can be arrived at by making a locally optimal solution. In other words, an optimal solution can be obtained by making "greedy" choices. 2) Optimal Substructure: Optimal solutions contains optimal sub solutions [18].

Every implementation of a Greedy algorithm must have the following elements: 1) The set C of candidates or entries. 2) Selection function. Inform what is the most promising element to complete the solution. This element will not have been rejected or chosen previously (then it will belong to $C \rightarrow S$). 3) Feasibility function. Report if a solution can be reached from a set. It applies to the selected set united with the most promising element. 4) Objective Function. Returns the goodness of the solution found. Normally you want your value to be maximum or minimum. 5) Solution function. Check if the candidate subset forms a solution (it does not matter if it is optimal or not) [19]. In general, the Greedy search is not optimal and is not complete. It has a complexity in time of O(bm), where b = branch factor and m = solution depth. In space, it is O(bm) because it stores all the nodes in memory.

The basic scheme of a Greedy algorithm is the following:

Given a finite set of C entries, a Greedy algorithm returns a set S (selected) such that $S \in C$ and that, in addition, complies with the constraints of the initial problem. Each set C that satisfies the constraints is usually called promising, if it also achieves that the objective function is minimized or maximized (as appropriate), then, S is said to be an optimal solution.

This can be represented as [20]:

$$\sum_{i \leq k} w(x_i) = max\left\{\sum_{x \in I} w(x): |I| = k, I \in C(S)\right\} \qquad (1)$$

The scheme of a voracious algorithm can be [21]:

```
function Voracious $C$: set): set
 //$C$ set of candidates
 $S \leftarrow Ø$ // initially empty solution
 While $\neg$solucition($S$) $\wedge$ $C \neq Ø$ do
   $x \leftarrow$ select ($C$) //gets locally optimal candidate
   if feasible ($S \cup \{x\}$) then $S \leftarrow S \cup \{x\}$
   else C $\leftarrow$ C-{x} // delete candidate
End while
if solution ($S$) then
  $v \leftarrow$ value (objective_function (S))
else $S \leftarrow Ø$ // empty solution
return ($S$, $v$)
```

B. *A* Algorithm (A-star)*

It is a search algorithm developed in 1968, which finds the shortest route between two points. A* algorithm was initially designed for the graph transversal problems. Later, it was commonly used for path finding applications such as computer games, artificial intelligence and robotics [22]. A* algorithm is mainly designed to identify an optimal path from a given initial position to a given goal position and it combines uniform-cost search (Dijkstra) and greedy search algorithms [23]. Compared to other artificial intelligence algorithms it has many advantages, such as shorter running time, high efficiency, easy implementation. A* algorithm is a progressive global search algorithm, an algorithm from local start searching, through local speculation global search [24].

It is said that A* is an informed algorithm, since in each step it decides which branch to follow depending on a rule or heuristic. If the selected heuristic is optimal, the complexity of the algorithm is reduced to O(n). For this reason, it is widely used for minimal path searches [25]. A* algorithm is defined as best-first algorithm, because each node in the configuration space is evaluated [26].

The heuristic cost of Algorithm A* is expressed by the estimated function f(n):

$$f(n) = g(n) + h(n) \tag{2}$$

$$h(n) = \sqrt{(n_x - g_x)^2 + (n_y - g_y)^2} \tag{3}$$

$$g(n) = \sqrt{(n_x - s_x)^2 + (n_y - s_y)^2} \tag{4}$$

Where $g(n)$ is the minimum cost from the source node to the current node. $h(n)$ is the minimum cost from the current node to the destination node. n_x and n_y are the coordinates of the current node n. g_x and gy are the coordinates of the target node g, s_x, and s_y are the coordinates of the initial node s [27]. A* uses the heuristic function $h(n)$, where $h(n) \geq 0$ and calculates its value in each node of the work area to obtain the optimal solution, and choose the next best step that contains the lowest value of f(n) [28].

The basic scheme of A* algorithm is the following: 1) Set s as source. Do $f(s) = g(s) + h(s) = 0 + h(s)$ and add node s to the set T. 2) Select the node i of the set T that represents the lowest value of the function $f(i)$. If there is a tie between the minors $f(i)$, one is chosen arbitrarily, but always in favor of the node t. 3) If $i = t$ stop and save i in the set V. 4) If $i \neq t$ move i the T a V and add neighboring nodes j of i. For each link (i, j) with cost c_{ij} calculate: $f(j) = g(i) + c_{ij} + h(j)$. If $f(j)$ was not calculated add node j to T. If $j \in T$ then $f(j)$ was already calculated and its value was greater than the current one, update $f(j)$. If $j \in V$ and $f(j)$ current is less than when j was marked as closed, update $f(j)$ and move j the V a T. Return to step 2 [29].

The time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the search space is a tree, there is a single goal state, and the heuristic function h meets the following condition:

$$|h(n) - h^*(n)| = O(\log h^*(n)) \tag{5}$$

Where h^* is the optimal heuristic, the exact cost to get from n to the goal. In other words, the error of h will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from n to the goal [30].

In general, A* is optimal and is complete. It has a complexity in time of $O(b^m)$, where b = branch factor and m = solution depth. In space, it is $O(b^m)$.

C. *Dijkstra's Algorithm*

Also called the minimal path algorithm belongs to the group of voracious classical algorithms. It was developed by Edsger Dijkstra in 1959. Dijkstra's algorithm is a search algorithm that computes the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree. Dijkstra's algorithm employs the greedy approach to solve the single source shortest problem. It repeatedly chooses from the unselected vertices, vertex v nearest to source s and announces the distance to be the actual shortest distance from s to v [31]. It is important to be clear, that with some adjustments you can get the algorithm to indicate the lowest cost between an *SI* and an *OS*.

In general, Dijkstra is optimal and is complete. It has a complexity $O(n^2)$ where n is the number of vertices.

The basic scheme of the Dijkstra algorithm using a priority queue as an auxiliary data structure is as follows [3]:

```
Dijkstra (graph G, source_node s)
for u ∈ V[G] do
  distance [u] = INFINITY
  father_node [u] = NULL
  seen [u] = false
  distance [s] = 0
  insert (queue, (s, distance [s]))
  while queue ≠ Ø do
  u = extract_min (queue)
  seen [u] = true
  for all v ∈ adjacency[u] do
      if not seen [v] and distance [v] >
            distance [u] + length (u, v) do
        distance [v] = distance [u] +
                          length (u, v)
        parent_ node [v] = u
        insert (queue, (v, distance [v]))
```

In the algorithm, a route will be determined by the sum of all the metrics of all the links through which it passes. At the end of exploring all the states, the algorithm calculates among all the possible routes generated during the exploration the one with the lowest metric, this being the shortest or least cost route.

D. *Implementation*

To achieve the research purpose, an application was implemented in the Java programming language version 1.8 with a Graphical User Interface (GUI), which shows a map (graph) with 100 cities (nodes) and their respective routes (160 edges) that connect the cities, which gave the user the option to select the city of origin (*IS*) and the destination (*OS*), and the algorithm that will be used to calculate the minimum cost route.

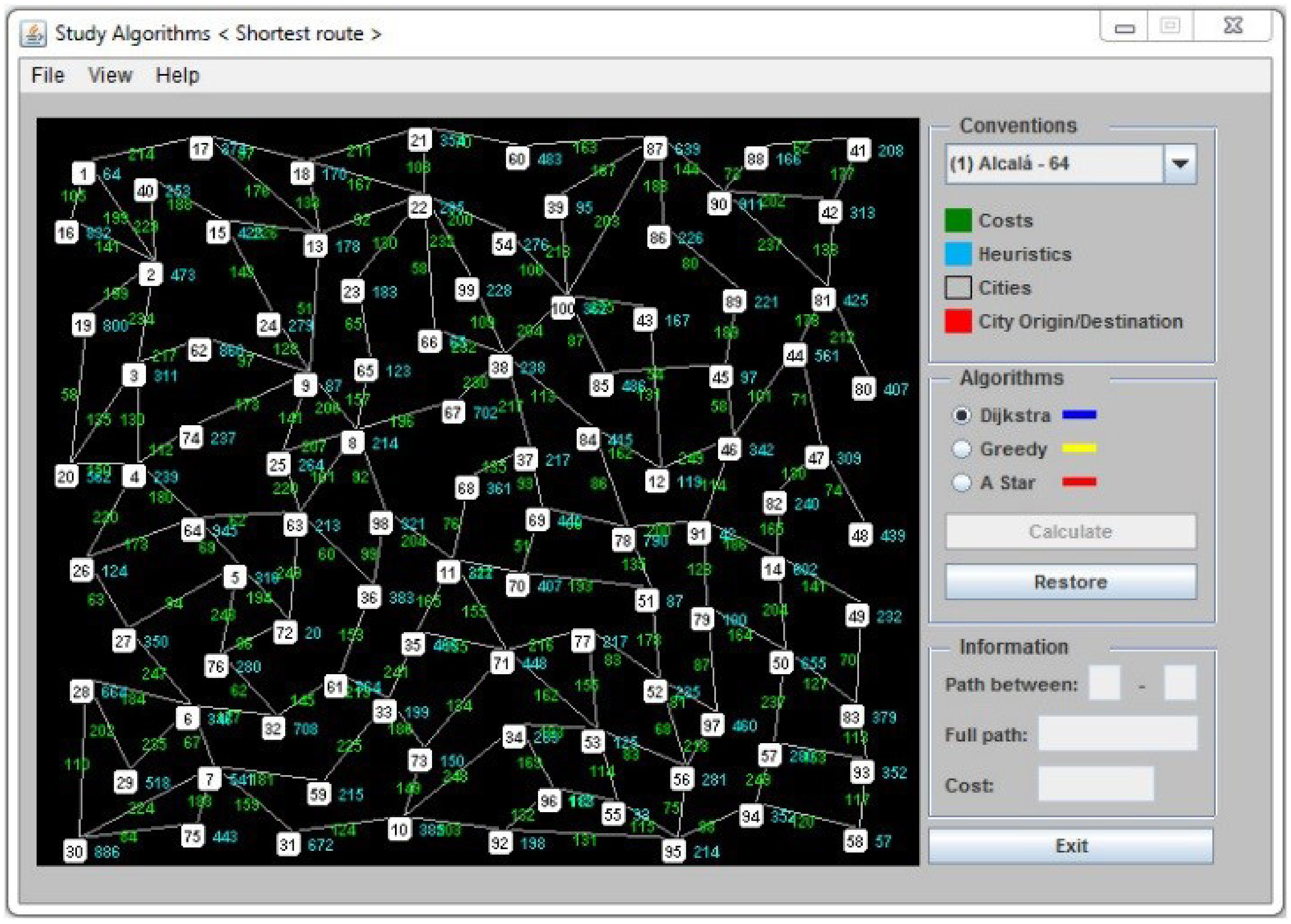


Fig. 2. Main window of the application.
Source: Authors.

In the graph, each node is assigned a default value (heuristic) that represents some degree of importance, and each edge has a default value (weight) that represents the distance in kilometers between each node. In both cases, these values can be modified by the user, as necessary. This feature was very useful in cases in which the Greedy search or the A* algorithm resulted in an *NI*, that is, they could not reach a solution, due to the fact that it was not possible to continue expanding some of the nodes involved in the route between *IS* and *OS*.

Once a solution is given to the minimum cost route between *IS* and *OS*, either before an *PI* or an *NI*, the route layout, the arrangement of nodes involved and the total cost (kilometers traveled) are shown on the screen. Additionally, the user has the possibility to consult the step by step followed by the Greedy search and the A* algorithm.

As mentioned previously, due to the nature of the algorithms, it was necessary to modify their characteristics to achieve the goal of obtaining the minimum cost route between two nodes. Next, the pseudo-code of the algorithms implemented in the application.

1) *Greedy Algorithm*

```
Let IS = Initial State and OS = Target State
Let string neighbors = “”
Let the arrays expanded [ ] and visited [ ]
visited [ ] ∪ IS
While (true) {
  expanded [ ] ∪ first position visited [ ]
  if (OS Є expanded [ ]) break
  remove positions from visited [ ]
  neighbors = search neighbors
                  lastElement(expanded[ ])
  visited [ ] = neighbors ∉ expanded [ ]
  sort visited [ ]
}
```

2) *A* Algorithm*

```
Let IS = Initial State and OS = Target State
Let string neighbors = “”
Let the arrays expanded [ ], visited [ ],
costRuta[ ] and functionVisited[ ]
visited [ ] ∪ IS
While (true) {
  expanded [ ] ∪ first position visited [ ]
  if (OS Є expanded [ ]) break
  remove positions from visited [ ]
  neighbors = search neighbors
                    lastElement(expanded[ ])
  visited [ ] = neighbors ∉ expanded [ ]
  costRuta[ ] = getCostRuta(visited [ ],
      lastElement(expanded[ ])) +
      costRuta[lastElement(expanded[ ])]
  functionVisited[ ] =
      heuristic of each element of visited [ ] +
      costRuta[ ] the visited [ ]
  sort visited [ ] by functionVisited[ ]
  remove positions from functionVisited[ ]
}
```

3) *Dijkstra’s algorithm*

```
function Dijkstra (matrizCosts [ ] [ ], origin, destination) {
  string adyacentNoSelect = “”; set conjuntoS[ ]
  fill (pathNodes[ ], “”)
  fill (vectorWeight [ ], Integer.MAX_VALUE)
  fill (nodeSelect, false)
  vectorWeight [origin] = 0
  while(conjuntoS ¬contains(destination)){
     nodeMinWeight = getNodeMinWeight ()
     nodeSelect [nodeMinWeight] = true
     adyacentNoSelect = getAdyacentNoSelect(conjuntoS, getAdyacent(nodeMinWeight))
     indexVW = nodeMinWeight
     for index < length(adyacentNoSelect){
       colum = getAdyacentNoSelect (index)
       if(vectorWeight [indexVW] + matrizCosts [indexVW][colum] < vectorWeight [colum]){
             vectorWeight [colum] = vectorWeight [indexVW] + matrizCosts [indexVW][colum]
            pathNodes [colum] = nodeMinWeight
          }
       }
     }
 return getPath(origin, destination)
}
function getPath(origin, destination){
  string path = destination; string originAux = origin
  while(originAux ≠ destination){
    origin = pathNodes [destination]
    path ∪ origin
    destination = origin
  }
  return reverse(path)
}
```

V. Results

The obtaining of results is divided into two procedures: the first is focused on test cases with results of *NI*, and the second with results of *PI*, both with the same input values. With the help of these two examples, it is possible to demonstrate that by modifying the information (*heuristic*) provided to an algorithm, it manages to reach an *PI*, but not necessarily the most optimal route.

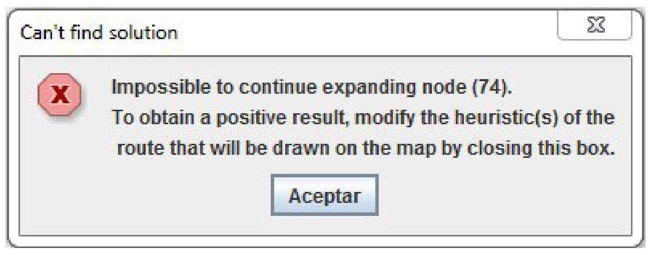
Inputs: *IS* = node 1, *OS* = node 79

A. *Negative Instance (NI)*

For the *NI* test case, it was shown that the Greedy and A* algorithms do not arrive at a positive response (arriving at *OS*), being stuck in local optima.

1) *Solution by Greedy*

Output:



Can't find solution

Impossible to continue expanding node (74).
To obtain a positive result, modify the heuristic(s) of the route that will be drawn on the map by closing this box.

Fig. 3. Greedy error. *NI* case.
Source: Authors.

Using Greedy was not possible to get a positive response (reach the OS). In the route traced by the algorithm (Fig. 4), it is observed that in the impossibility of expanding node 74 (its neighbors had been expanded), which does not allow reaching the OS (Table 1).

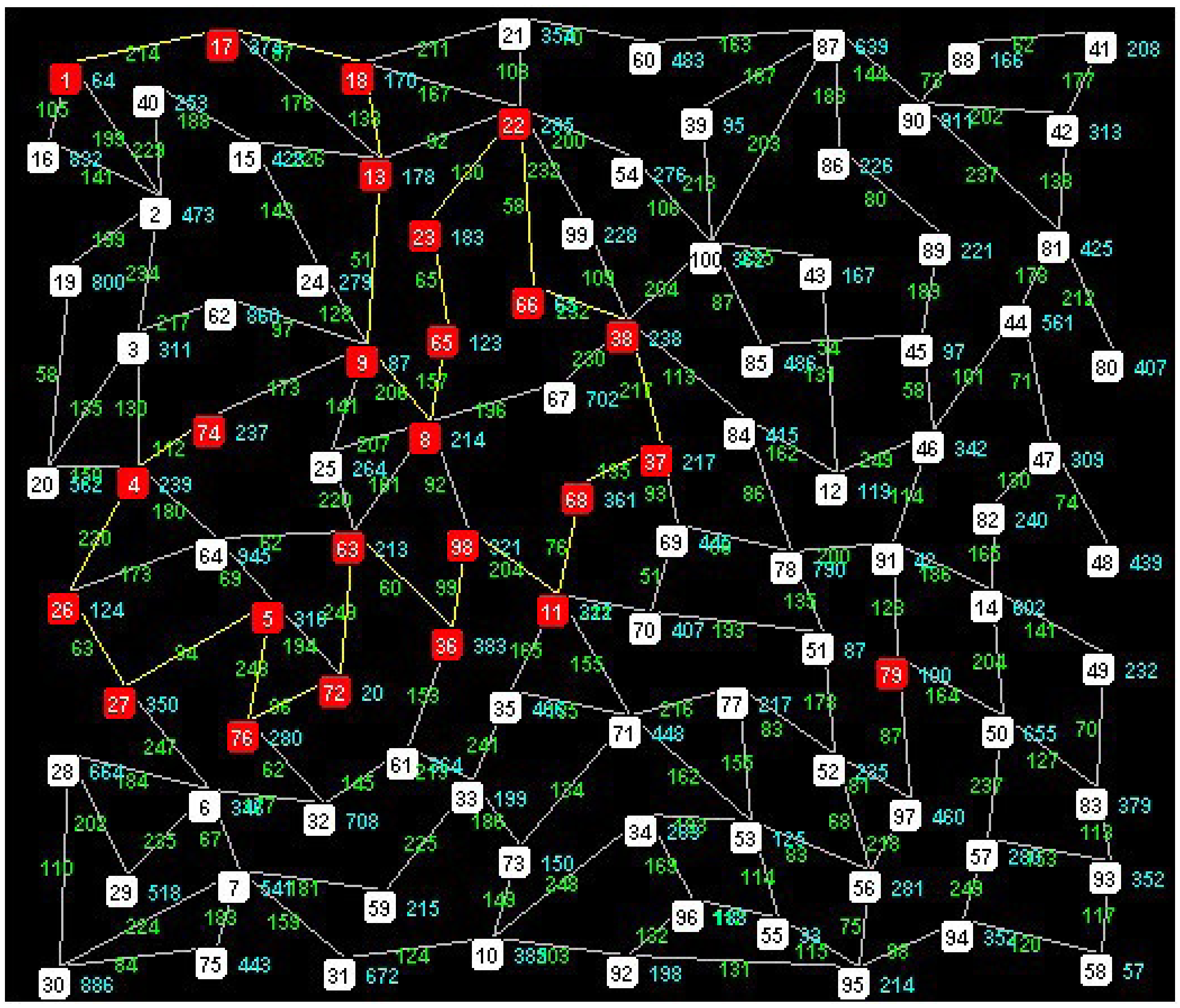


Fig. 4. Route traced by Greedy. NI case.
Source: Authors.

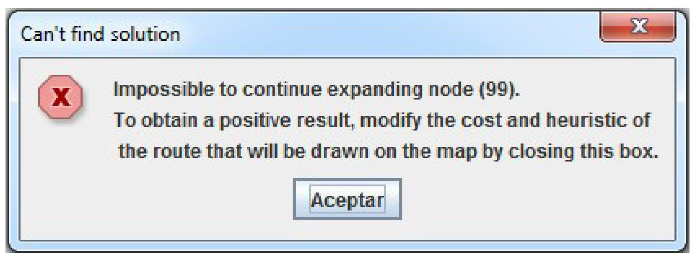
TABLE 1. STEP-BY-STEP GREEDY. NI CASE.

Expanded	Neighbors	Visited
1	2, 16, 17	17, 2, 16
17	1, 13, 18	18, 13
18	13, 17, 21, 22	13, 22, 21
13	9, 15, 17, 18, 22	9, 22, 15
9	8, 13, 24, 25, 62, 74	8, 74, 25, 24, 62
8	9, 25, 63, 65, 67, 98	65, 63, 25, 98, 67
65	8, 23	23
23	22, 65	22
22	13, 18, 21, 23, 54, 66, 99	66, 99, 54, 21
66	22, 38	38
38	37, 66, 67, 84, 99, 100	37, 99, 100, 84, 67
37	38, 68, 69	68, 69
68	11, 37	11
11	35, 68, 70, 71, 98	98, 70, 71, 35
98	8, 11, 36	36
36	61, 63, 98	63, 61
63	8, 25, 36, 64, 72	72, 25, 64
72	5, 63, 76	76, 5
76	5, 32, 72	5, 32
5	27, 64, 72, 76	27, 64
27	5, 6, 26	26, 6
26	4, 27, 64	4, 64
4	3, 20, 26, 64, 74	74, 3, 20, 64
74	4, 9	-
Can't find solution. Impossible to continue expanding node (74)		

Source: Authors.

2) *Solution by A**

Output:


Can't find solution

Impossible to continue expanding node (99).
To obtain a positive result, modify the cost and heuristic of the route that will be drawn on the map by closing this box.

Fig. 5. A* error. NI case.
Source: Authors.

Like Greedy, with A* it was not possible to get a positive response (reach the *OS*), in this case due to the impossibility of expanding at node 99 (its neighbors had been expanded) (Fig. 6).

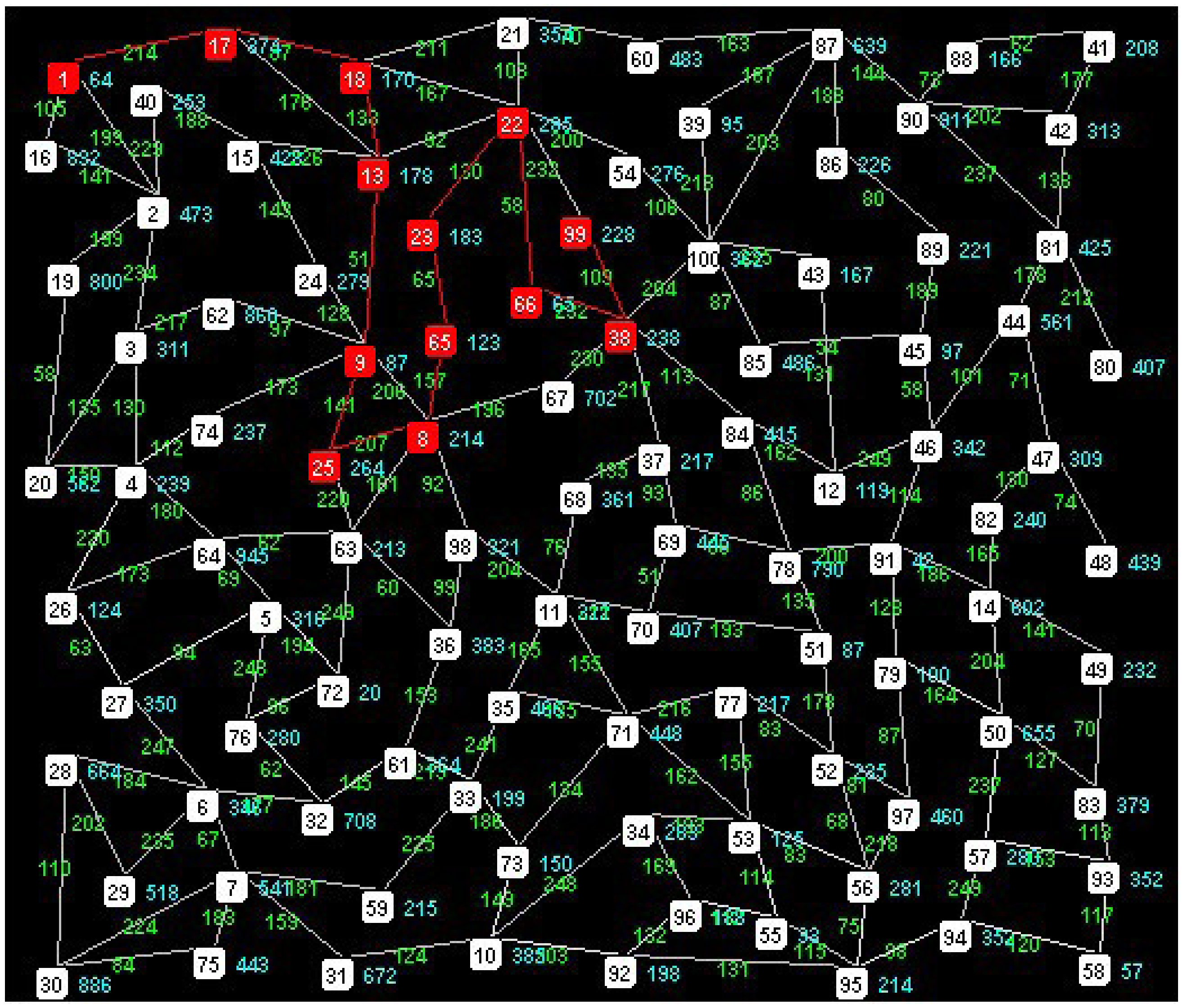


Fig. 6. Route traced by A*. NI case.
Source: Authors.

Table 2. Step-by-step A*. NI case.

Expanded	Neighbors	Visited
1	2, 16, 17	17, 2, 16
17	1, 13, 18	18, 13
18	13, 17, 21, 22	13, 22, 21
13	9, 15, 17, 18, 22	9, 22, 15
9	8, 13, 24, 25, 62, 74	25, 24, 74, 8, 62
25	8, 9, 63	8, 63
8	9, 25, 63, 65, 67, 98	65, 63, 98, 67
65	8, 23	23
23	22, 65	22
22	13, 18, 21, 23, 54, 66, 99	66, 21, 99, 54
66	22, 38	38
38	37, 66, 67, 84, 99, 100	99, 37, 84, 100, 67
99	22, 38	-
Can't find solution. Impossible to continue expanding node (99)		

Source: Authors.

B. *Positive Instance (PI)*

1) *Solution by Dijkstra*

Output:

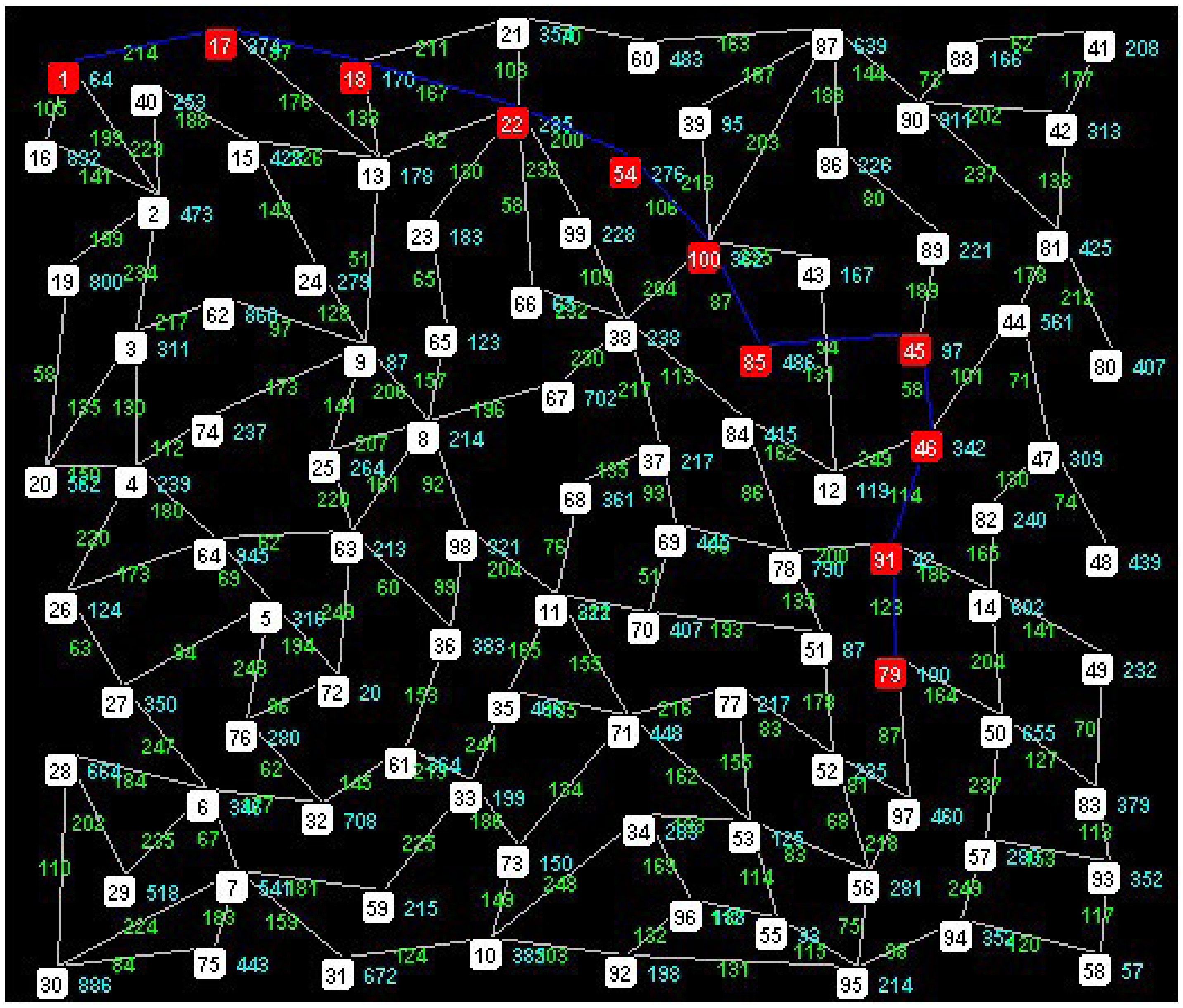


Fig. 7. Route traced by Dijkstra. PI case.
Source: Authors.

Full path: 1-17-18-22-54-100-85-45-46-91-79

Cost: 1215

The Dijkstra algorithm yielded a positive response that is complete, being also the most optimal, since the result obtained corresponds to the minimum cost route between *IS* and *OS* according to the costs specified as input.

2) *Solution by Greedy*

In order for Greedy's algorithm to achieve a positive instance, one of the possibilities is to modify the heuristic of node 74 (*h* = 237) to *h* = 100, so that it is less than the heuristic of node 8 (*h* = 214) and node 25 (*h* = 264) (Fig. 8), which allows modifying the previous path and reaching the *OS* (Fig. 9). Table 3 shows the step-by-step algorithm.

Output:

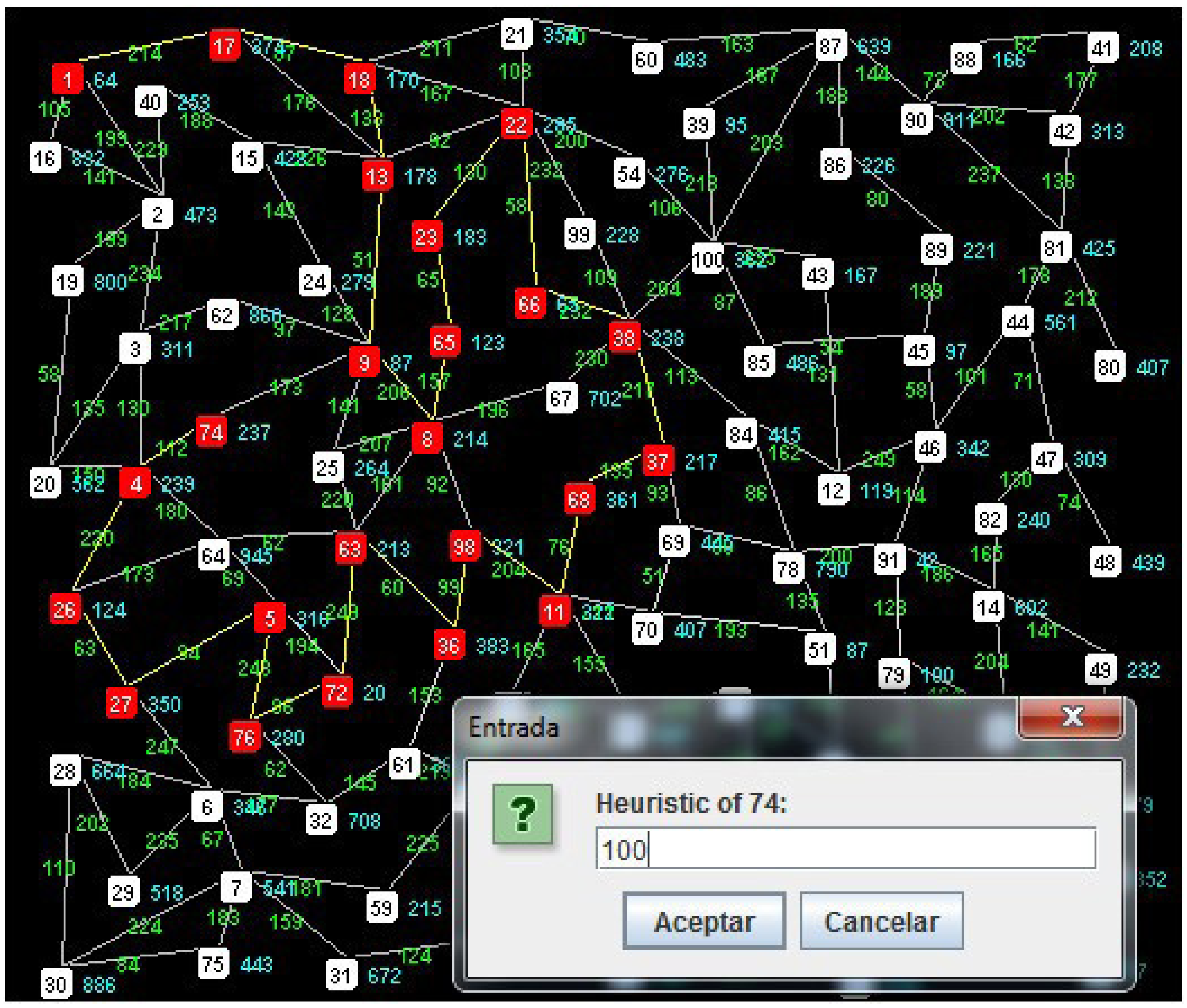


Fig. 8. Modification of heuristics. IP case.
Source: Authors.

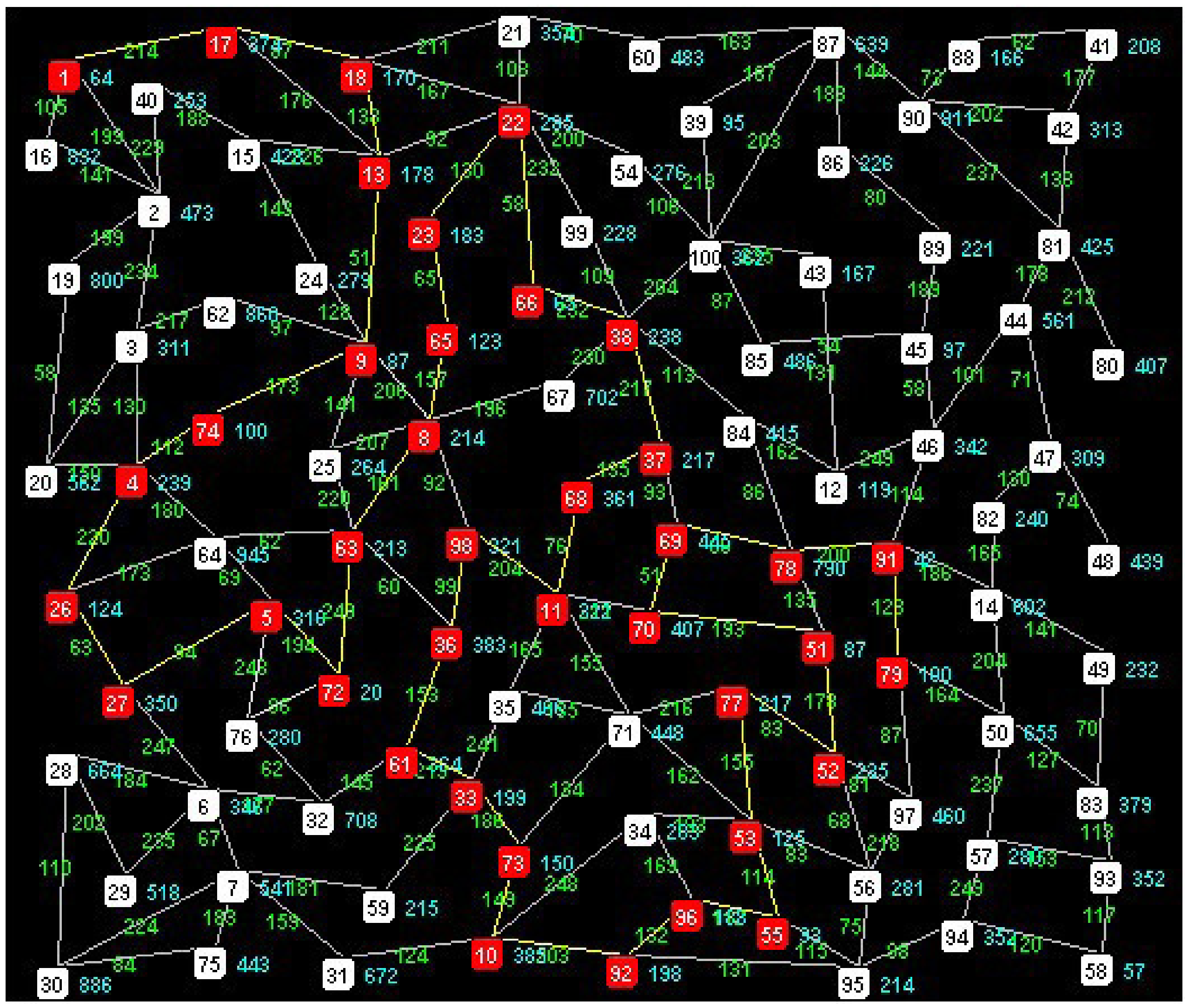


Fig. 9. Route traced by Greedy. PI case.
Source: Authors.

Table 3. Step-by-step Greedy. PI case.

Expanded	Neighbors	Visited
1	2, 16, 17	17, 2, 16
17	1, 13, 18	18, 13
18	13, 17, 21, 22	13, 22, 21
13	9, 15, 17, 18, 22	9, 22, 15
9	8, 13, 24, 25, 62, 74	74, 8, 25, 24, 62
74	4, 9	4
4	3, 20, 26, 64, 74	26, 3, 20, 64
26	4, 27, 64	27, 64
27	5, 6, 26	5, 6
5	27, 64, 72, 76	72, 76, 64
72	5, 63, 76	63, 76
63	8, 25, 36, 64, 72	8, 25, 36, 64
8	9, 25, 63, 65, 67, 98	65, 25, 98, 67
65	8, 23	23
23	22, 65	22
22	13, 18, 21, 23, 54, 66, 99	66, 99, 54, 21
66	22, 38	38
38	37, 66, 67, 84, 99, 100	37, 99, 100, 84, 67
37	38, 68, 69	68, 69
68	11, 37	11
11	35, 68, 70, 71, 98	98, 70, 71, 35
98	8, 11, 36	36
36	61, 63, 98	61
61	32, 33, 36	33, 32
33	35, 59, 61, 73	73, 59, 35
73	10, 33, 71	10, 71
10	31, 34, 73, 92	92, 34, 31
92	10, 95, 96	96, 95
96	34, 55, 92	55, 34
55	53, 95, 96	53, 95
53	34, 55, 56, 71, 77	77, 56, 34, 71
77	52, 53, 71	52, 71
52	51, 56, 77, 97	51, 56, 97
51	52, 70, 78	70, 78
70	11, 51, 69	69
69	37, 70, 78	78
78	51, 69, 84, 91	91, 84
91	14, 46, 78, 79	79, 46, 14
Full path: 1-17-18-13-9-74-4-26-27-5-72-63-8-65-23-22-66-38-37-68-11-98-36-61-33-73-10-92-96-55-53-77-52-51-70-69-78-91-79 Cost: 8812 (sum of the heuristics)		

Source: Authors.

Clearly it is observed that Dijkstra delivers a lower route cost, which is effectively the purpose.

3) *Solution by A**

In the case of A*, it was necessary to modify the heuristics of node 99 ($h = 228$) to $h = 350$, so that it is greater than the heuristics of node 66 ($h = 65$) and node 38 ($h = 238$), and again the heuristic of node 74 ($h = 237$) to $h = 100$, so that it is less than the heuristic of node 4 ($h = 239$), allowing modifying the previous route and reaching the *OS* (Fig. 10). Table 4 shows the step-by-step algorithm.

Output:

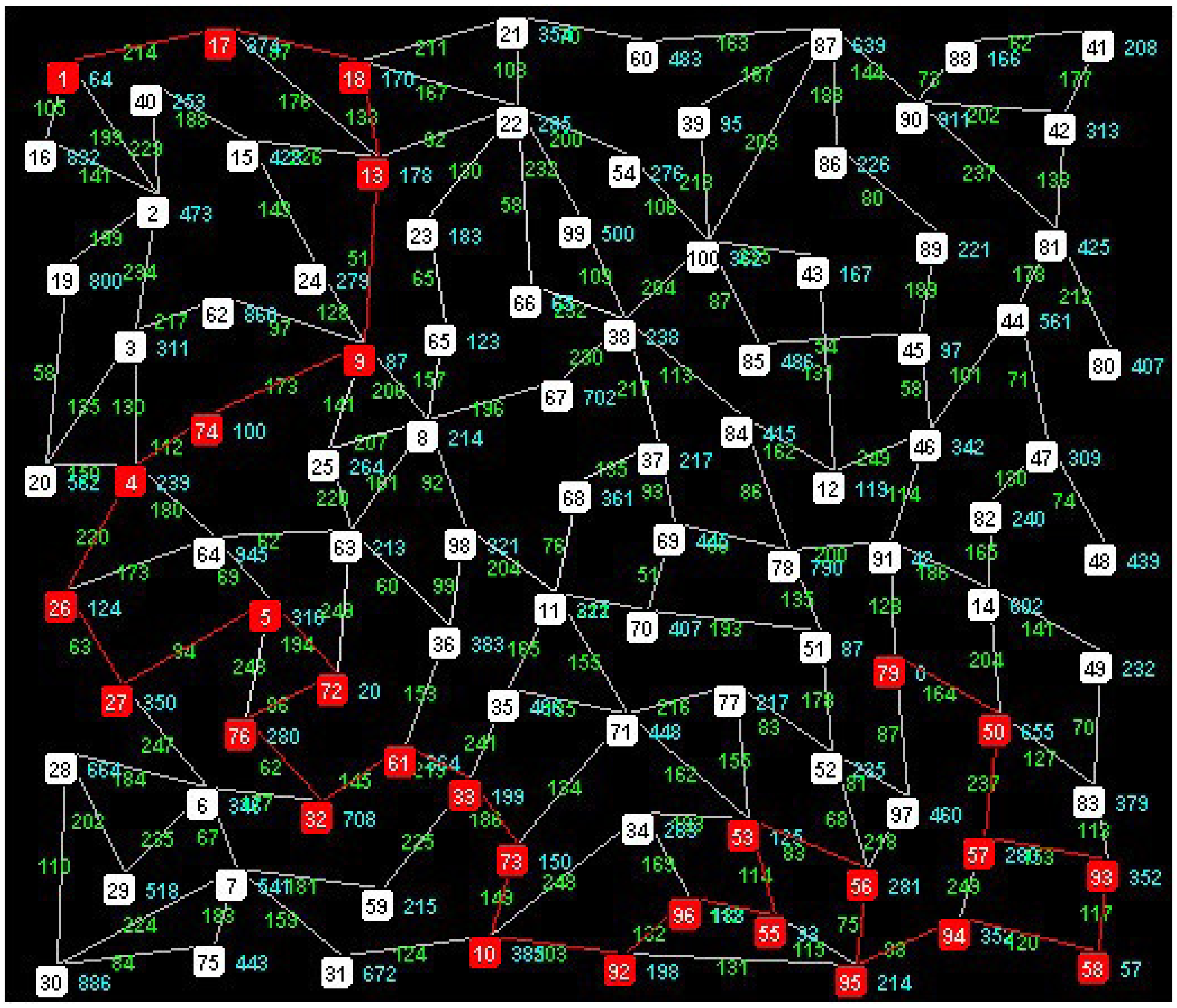


Fig. 10. Route traced by A*. PI case.
Source: Authors

TABLE 4. STEP-BY-STEP A*. PI CASE.

Expanded	Neighbors	Visited
1	2, 16, 17	17, 2, 16
17	1, 13, 18	18, 13
18	13, 17, 21, 22	13, 22, 21
13	9, 15, 17, 18, 22	9, 22, 15
9	8, 13, 24, 25, 62, 74	74, 25, 24, 8, 62
74	4, 9	4
4	3, 20, 26, 64, 74	26, 3, 20, 64
26	4, 27, 64	27, 64
27	5, 6, 26	5, 6

Expanded	Neighbors	Visited
5	27, 64, 72, 76	72, 76, 64
72	5, 63, 76	76, 63
76	5, 32, 72	32
32	6, 61, 76	61, 6
61	32, 33, 36	33, 36
33	35, 59, 61, 73	73, 59, 35
73	10, 33, 71	10, 71
10	31, 34, 73, 92	92, 34, 31
92	10, 95, 96	96, 95
96	34, 55, 92	55, 34
55	53, 95, 96	53, 95
53	34, 55, 56, 71, 77	56, 77, 34, 71
56	52, 53, 95, 97	95, 52, 97
95	55, 56, 92, 94	94
94	57, 58, 95	58, 57
58	93, 94	93
93	57, 58, 83	57, 83
57	50, 93, 94	50
50	14, 57, 79, 83	79, 83, 14
Full path: 1-17-18-13-9-74-4-26-27-5-72-76-32-61-33-73-10-92-96-55-53-56-95-94-58-93-57-50-79 Cost: 3805		

Source: Authors.

As can be seen in the results of Table 3 and Table 4, A* performs fewer node expansions, which reduces the number of operations when calculating the (adjacent) neighbors of the expanded node, and the visited nodes, making the space in memory be less. On the other hand, like Greedy, A* offers a positive response thanks to the modification of the heuristic. Furthermore, A* calculated a route with fewer nodes between the *IS* and the *OS*, being more optimal than Greedy, but not exceeding Dijkstra, that is, almost complete and optimal.

C. *Test cases*

In addition to the *NI* and *PI* cases, two additional test cases were carried out, in order to measure and average certain characteristics in the behavior of the implemented algorithms. These results allow us to identify that Greedy and A* behave computationally in a very similar way. The opposite occurs with Dijkstra, where the measured characteristics were much higher, reflecting that its property of being complete has a correlation with the consumption of resources both in memory, and in the number of instructions that the CPU must execute to achieve the best result according to the input parameters.

The quantified characteristics were:

a) *Number of assignments*: Refers to any operation that changes the value of a variable or position in a data structure.

b) *Number of comparisons*: Refers to the use of relational operators and similar functions of the programming language.

c) *Number of operations on data structures*: Any operation carried out on a data structure.

d) *Number of nodes in the final path*: This reflects the number of nodes required to reach the *SO*.

e) *Cost*: Value in distance and what represents the heuristics of each node in the case of Greedy.

The algorithm execution time was not taken into account, since it is a measure that is closely related to the hardware and software characteristics of the computer where the algorithms are executed.

Table 5. Test case results.

Algorit.	Num. Assign.	Num. Comp.	Num. Oper. data struc.	Num. Nodes end path	Cost
Test case 1: IS = 1 and OS = 79					
Greedy	8469	8203	711	39	8812
A*	6402	6224	1090	29	3805
Dijkstra	46926	50513	17792	11	1215
Test case 2: IS = 1 and OS = 83					
Greedy	9999	9699	823	46	10628
A*	5948	5782	1019	27	3364
Dijkstra	56645	63601	21281	13	1484
Test case 3: IS = 15 and OS = 94					
Greedy	8905	8631	742	41	9687
A*	4822	4685	853	22	2801
Dijkstra	57050	64136	21436	11	1346

Source: Authors.

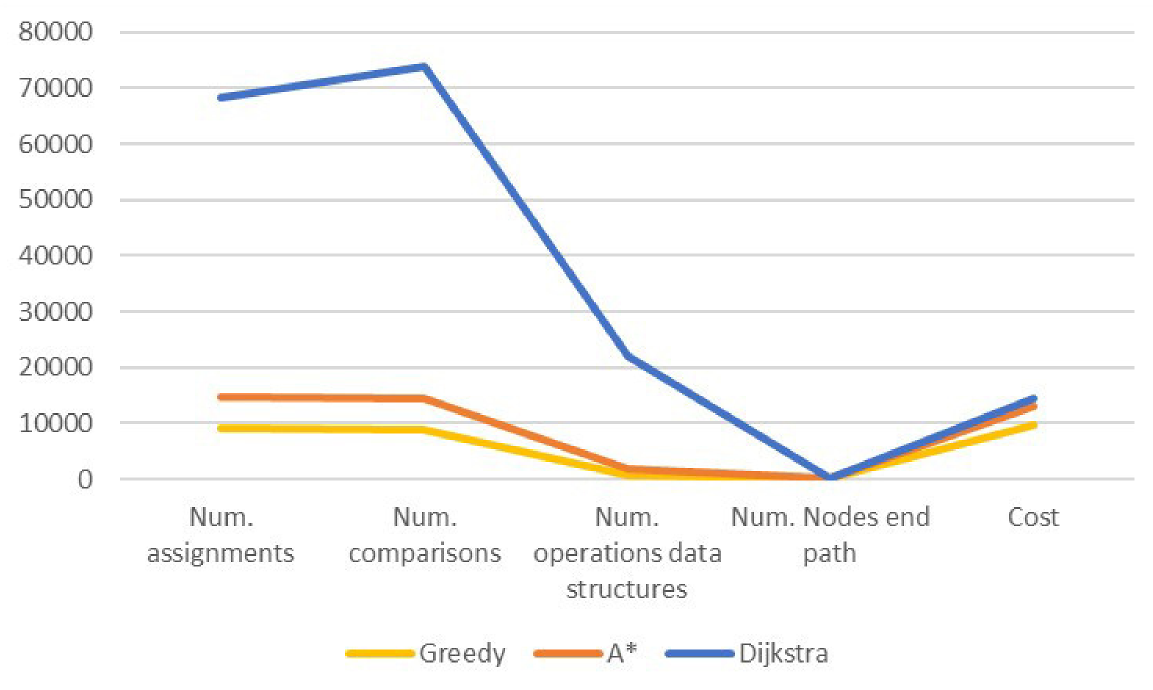


Fig. 11. Average test cases.
Source: Authors.

IV. Conclusions

The test cases allowed to show that, although it is true, the three algorithms used for the study have the same objective, their behavior differs depending on the value of the heuristic assigned to them, except for the Dijkstra algorithm. Obviously it is observed that the best algorithm to solve the problem of the shortest route is Dijkstra, which, being a complete algorithm, in addition to delivering a positive response, offers an optimal response with the lowest cost according to the weights associated with the edges, but showing that its computational cost is very high when the graph is large.

On the other hand, for some information inputs, both the Dijkstra and A* algorithms offer the same complete and optimal positive response, with the difference that A* needs the heuristic to achieve Dijkstra. In this aspect, if the input information for the Greedy algorithm is modified, we can achieve great improvements achieving a positive response, without necessarily becoming optimal. That is to say, Greedy is not totally incomplete, being dependent on the heuristic. That is why heuristics is a fundamental factor to be taken into account when using voracious algorithms. With variations in the heuristics, better results are obtained.

On the other hand, the research laid the theoretical and practical bases for the future study and improvement of other algorithms related to the problem in question, opening new opportunities for future research, in which more current techniques such as Artificial Intelligence, Machine Learning and Deep Learning, that allow to include more variables and restrictions, and achieve an optimal result computationally in larger cases.

Finally, it can be concluded that given the need to implement increasingly fast and efficient computer systems, it is essential that algorithms that solve highly complex optimization problems apply techniques that improve their performance making them more complete and optimal, and in this aspect metaheuristics provide better tactics to create heuristics that prevent them from being trapped in local optima that differ from the overall optimum value.

Financing

Scientific research article derived from the research project “Evaluation of minimum route methods in a weighted and heuristic graph”, prepared by the SIEL (Research and Development in Information Systems and Electronics) Study Seedbed of the Universidad del Valle (Buga, Colombia). Start year: 2018, end year: 2019.

References

[1] N. Ojekudo & N. Akpan, “Anapplication of Dijkstra’s Algorithm to shortest route problem”, *IOSR-JM*, vol. 13, no. 3, pp. 20–32, 2017. Available: http://iosrjournals.org/iosr-jm/pages/v13(3)Version-1.html

[2] B. Obregón, “Teoría de redes: El problema de la ruta más corta”, *tesis magistral*, prog. ing., UNAM, CDMX, MX. 2005. Available at: http://www.ptolomeo.unam.mx:8080/xmlui/bitstream/handle/132.248.52.100/539/obregonquintana.pdf?sequence=12

[3] J. Barelles, “Algoritmos para la resolución de problemas en redes”, *trabajo grado*, dpto Mat Comp, UJI, Cast, Esp, 2017. Available from http://repositori.uji.es/xmlui/bitstream/handle/10234/173687/TFG_2017_BarellesMenes_Jorge.pdf?sequence=1&isAllowed=y

[4] Y. Talebiy & H. Rashmanlouz, “Application Of Dominating Sets In Vague Graphs”, *Appl Math E-Notes*, vol. 17, pp. 251–267, 2017. Available from https://www.emis.de/journals/AMEN/2017/AMEN-170503.pdf

[5] M. Dod, “Domination in graphs with application to network reliability”, *Thesis Doctor*, Fac Math Comp Sci, FUMT, Fog, DE, 2015. Available from https://tubaf.qucosa.de/api/qucosa%3A23011/attachment/ATT-0/

[6] S. Guze, “An application of the selected graph theory domination concepts to transportation networks modelling”, *Zesz Nauk Uniw Rzesz*, vol. 52, no. 124, pp. 97–102, 2017. https://doi.org/10.17402/250

[7] S. Vishwanathan, N. Schraudolph, R. Kondor & K. Borgwardt, “Graph Kernels”, *JMLR*, vol. 11, pp. 1201–1242, 2010. Available from http://www.jmlr.org/papers/volume11/vishwanathan10a/vishwanathan10a.pdf

[8] K. Abdulkalek & A. Kilicman, “Topologies on the Edges Set of Directed Graphs”, *Int J Math Anal*, vol. 12, no. 2, pp. 71–84, 2018. https://doi.org/10.12988/ijma.2018.814

[9] N. Jicy & M. Sunil, “Some new connectivity parameters for weighted graphs”, *J Uncertain Math Sci*, pp. 1–9, 2014. Available: https://www.researchgate.net/publication/263773151_Some_new_connectivity_parameters_for_weighted_graphs_Some_new_connectivity_parameters_for_weighted_graphs

[10] A. Ban, “Decomposing Weighted Graphs”, *JGT*, vol. 86, no. 2, pp. 250–254, 2017. https://doi.org/10.1002/jgt.22124

[11] S. Mathew, “Partial trees in weighted graphs-I”, *Proyecciones J Math*, vol. 30, no. 2, pp. 163–174, 2011. http://dx.doi.org/10.4067/S0716-09172011000200003

[12] C. Salas, “Un Algoritmo de Dos Fases para la Optimización de Costos en el Traslado de Cargas con Exceso de Dimensiones”, *Tesis magistral*, UAEH, HGO, MX, 2014. Disponible en http://dgsa.uaeh.edu.mx:8080/bibliotecadigital/bitstream/handle/231104/1918/AT18381.pdf?sequence=3&isAllowed=y

[13] G. González & F. González, “Metaheurísticas aplicadas al ruteo de vehículos. Un caso de estudio. Parte 2: algoritmo genético, comparación con una solución heurística”, *Rev Ing Inv*, vol. 27, no. 1, pp. 149–157, 2007. Available from https://revistas.unal.edu.co/index.php/ingeinv/article/view/14795/15626

[14] L. Rodríguez & A. Varona. Técnicas de diseño de algoritmos. Algoritmos voraces. (2015). *Curso de informática*. PV, Esp: UPV. Recuperado de https://ocw.ehu.eus/pluginfile.php/46102/mod_resource/content/1/03_Algoritmos_Voraces/03_Algoritmos_Voraces.pdf

[15] A. Subhadra, “Greedy Algorithms: Analysis, Design & Applications”, *IJIFR*, vol. 3, no. 5, pp. 1749–1764, 2016. Available: https://www.academia.edu/21855750/Greedy_Algorithms_Analysis_Design_and_Applications

[16] D. Maxwell, “Optimal Structure Identification With Greedy Search”, *JMLR*, vol. 3, pp. 507–554, 2002. Available: https://www.jmlr.org/papers/v3/chickering02b.html

[17] B. Simmons, C. Hoeppke & W. Sutherland, “Sutherland. Beware greedy algorithms”, *J Anim Ecol*, vol. 88, no. 5, pp. 804–807, 2019. https://doi.org/10.1111/1365-2656.12963

[18] A. Malik, A. Sharma & V. Saroha, “Greedy Algorithm”, *IJSRP*, vol. 3, no. 8, pp. 1–5, 2013. Available: http://www.ijsrp.org/research-paper-0813.php?rp=P201564

[19] O. Debdi, "Aprendizaje Interactivo de Algoritmos Voraces: del Enfoque Individual al Colaborativo", *Tesis Doctoral*, URJC, MD, ES, 2014. Available: http://hdl.handle.net/10115/13242

[20] U. Faigle, "The greedy algorithm for partially ordered sets", *Discrete Math*, vol. 28, no. 2, pp. 153–159, 1979. https://doi.org/10.1016/0012-365X(79)90092-X

[21] M. Abad. Algoritmos voraces, Anàlisi i Disseny d'Algorismes. (2007-2008). *Curso de informática*. BCN, ES: UPC. Available from http://www.cs.upc.edu/~mabad/ADA/curso0708/GREEDY.pdf

[22] N. Shrikant & A. Selvakumar, "Implementation of A* Algorithm to Autonomous Robots-A. Simulation Study", *Eng Technol Open Acc*, vol. 1, no. 3, pp. 88–91, 2018. https://doi.org/10.19080/ETOAJ.2018.01.555564

[23] P. Sudhakara & V. Ganapathy, "Trajectory Planning of a Mobile Robot using Enhanced A-Star Algorithm", *Indian J Sci Technol*, vol. 9, no. 41, pp. 1–10, 2016. https://doi.org/10.17485/ijst/2016/v9i41/93816

[24] J. Peng, Y. Huang & G. Luo, "Robot Path Planning Based on Improved A* Algorithm", *Cybern Inf Technol*, vol. 15, no. 2, pp. 171–180, 2015. https://doi.org/10.1515/cait-2015-0036

[25] R. Rodríguez-Puente & M. Lazo-Cortés, "Búsquedas de caminos mínimos haciendo uso de grafos reducidos", *Ing Ind*, vol. 38, no. 1, pp. 32–42, 2017. Available at: https://rii.cujae.edu.cu/index.php/revistaind/article/view/497/759

[26] F. Duchon, A. Babineca, M. Kajana, P. Beño, M. Florek, T. Fico & L. Jurišica, "Path planning with modified A star algorithm for a mobile robot", *Procedia Manuf*, vol. 96, pp. 59–69, 2014. https://doi.org/10.1016/j.proeng.2014.12.098

[27] X. Dai, S. Long, Z. Zhang & D. Gong, "Mobile Robot Path Planning Based on Ant Colony Algorithm With A* Heuristic Method", *Front Neurorobot*, vol. 13, pp. 1–9, 2019. https://doi.org/10.3389/fnbot.2019.00015

[28] A. KumarGuruj, H. Agarwal & D. Parsediya, "Time-Efficient A* Algorithm for Robot Path Planning", *Procedia Technol* vol. 23, pp. 144–149, 2016. https://doi.org/10.1016/j.protcy.2016.03.010

[29] A. Beriain, "Matemáticas en un Navegador GPS: Algoritmos de Camino más Corto y Calculo de Posición", *trabajo grado*, UR, LR, Esp, 2016. Available from https://biblioteca.unirioja.es/tfe_e/TFE002201.pdf

[30] M. Nosrati, R. Karimi & H. Hasanvand, "Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches", *World Appl Program*, vol. 2, no. 4, pp. 251–256, 2012. Available at: https://pdfs.semanticscholar.org/831f/f239ba77b2a8eaed473ffbfa22d61b7f5d19.pdf

[31] N. Gupta, K. Mangla, A. Kumar & M. Umar. "Applying Dijkstra's Algorithm in Routing Process", *IJNTR*, vol. 2, no. 5, pp. 122–124, 2016. Available from https://www.ijntr.org/download_data/IJNTR02050040.pdf

Luis Adrián Lasso Cardona. Systems Engineer from Universidad del Valle (Colombia). M. Sc. in Educational Technology Management from Universidad de Santander (Colombia). Assistant Professor of the Faculty of Engineering and Faculty of Management Sciences of the Universidad del Valle (Buga, Colombia). Professor of the Faculty of Engineering of the Unidad Central del Valle del Cauca (Tulua, Colombia).. https://orcid.org/0000-0002-3354-1554

Diego Fernando Franco Ocampo. Electrical Engineer from Universidad del Valle (Colombia). M. Sc. Engineering with emphasis in automation. Universidad del Valle (Colombia). Coordinator of Electronics Technology and Information Systems Technology at Universidad del Valle (Buga, Colombia).. https://orcid.org/0000-0002-4797-8263

Alexander Agudelo Acevedo. Information Systems Technologist, Universidad del Valle (Colombia). Professor of the Faculty of Engineering Universidad del Valle (Buga, Colombia). Professor of the Faculty of Administrative Sciences and Faculty of Engineering Universidad del Valle (Buga, Colombia).. https://orcid.org/0000-0003-2200-349