

Package problem

Problem Statement: Optimal Parcel Delivery Route

Description: A courier company needs to find the optimal route to deliver packages to different destinations in a city. The city is represented by a set of intersections and streets connecting these intersections. Each intersection is a node in a network, and the streets are the edges connecting the nodes. Each edge has a weight representing the distance between intersections.

The objective is to find the shortest route to deliver all the packages, minimizing the total distance traveled by the courier. The courier can start at any intersection and must pass through all intersections exactly once before returning to the starting point.

Formulate the problem using graphs and describe the algorithm you would use to find the optimal parcel delivery route.

ENGINEERING METHOD

CLIENT	Message enterprice
USER	Delivery courier
FUNCTIONAL REQUIREMENT	<p>R1: determinate better rute</p> <p>R2: add place represented by vertex.</p> <p>R3: add route represented by edges.</p> <p>R4: add weight to the Edges.</p> <p>R5: minimize the distance of the Delivery Courier for the deliveries.</p>
NO FUNCTIONAL REQUIREMENT	<p>RN1: Accuracy: Although an approximate approach is used, the algorithm should provide solutions as close as possible to the optimal route. The goal is to minimize the total distance traveled by the courier to deliver all packages.</p>

	<p>RN2: Scalability: The algorithm must be able to efficiently handle large sets of intersections and streets. It must be scalable to adapt to cities with a large number of destinations.</p> <p>RN3: Flexibility: The system must allow the addition or removal of destinations in real time without affecting the overall performance. This implies that the algorithm must be able to update the optimal route efficiently when changes are made to the set of destinations.</p>
--	--

Name or identifier	RF1: determinate better rute		
Summary	Determine the optimal route to deliver packages to different destinations in a city.		
Inputs	input name	Datatype	Selection or repetition condition
	destination	T value	
General activities necessary to obtain the results	1. type the packet destination, 2. the program calculate the optimal route of the deliver. 3. the program show the designed route in console.		
Result or postcondition	Optimal route made an showed.		
Outputs	output name	Datatype	Selection or repetition condition
	toString	String	

Name or identifier	RF2: add place represented by vertex.		
Summary	Add place represented by a vertex in a graph.		
Inputs	input name	Datatype	Selection or repetition condition
	vertex	int	
General activities necessary to obtain the results	1. type the adress of the place 2. the program save the vertex in the list. 3. confirmation message.		
Result or postcondition	New place saved.		
Outputs	output name	Datatype	Selection or repetition condition
	msj	String	

Name or identifier	RF3: add route represented by edges.		
Summary	Add route represented by a edge in a graph.		
Inputs	input name	Datatype	Selection or repetition condition
	intialAdress	int	
	finalAdress	int	
General activities necessary to obtain the results	1. type the initial and the final destination. 2. the system assing the route 3. confirmation message		

Result or postcondition	New route saved		
Outputs	output name	Datatype	Selection or repetition condition
	msj	String	

Name or identifier	RF4: add weight to the Edges.		
Summary	Assign weights to edges representing the distance between intersections.		
Inputs	input name	Datatype	Selection or repetition condition
	weight	int	
	inicialValue	int	
	finalValue	int	
General activities necessary to obtain the results	<p>1. type the inicialValue and the finalValue to find the Edge.</p> <p>2. the program search the Edge.</p> <p>3.1 if the program not find the Edge, the program dont print a error message.</p> <p>3.2 if the program find the Edge, the user Will type the weight to asing to the respetive Edge.</p> <p>4 the program show a confirmation message.</p>		
Result or postcondition	Weight assigned.		
Outputs	output name	Datatype	Selection or repetition condition
	toString	String	

Name or identifier	RF5: minimize the distance of the Delivery Courier for the deliveries.		
Summary	Find the shortest route to deliver all packages.		
Inputs	input name	Datatype	Selection or repetition condition
	inicialValue	int	
	finalValue	int	
General activities necessary to obtain the results	1. type the inicialValue and the finalValue to find the initial and final point to design the route. 2. the program Will calculate the optimal route. 3 the program show the order of the vertex in the console.		
Result or postcondition	Optimal designed route		
Outputs	output name	Datatype	Selection or repetition condition
	toString	String	

SEARCH FOR CREATIVE SOLUTIONS:

To tackle the problem of finding the optimal parcel delivery route in a city represented by a network, you can use the "traveling salesman" algorithm (also known as the TSP, Traveling Salesman Problem). Here are some solution ideas:

Nearest Neighbor Algorithm: This algorithm starts at an initial node and, at each step, selects the nearest unvisited node as the next destination. It continues this process until all nodes have been visited and then returns to the starting point. This approach is simple and fast, but does not guarantee the optimal path.

Closest insertion algorithm: This algorithm starts with an initial path containing two nodes and, at each step, selects the closest unvisited node and inserts it at the position that minimizes the total path distance. It repeats this process until all nodes have been visited and then returns to the starting point. This approach can produce better solutions than the nearest neighbor algorithm, but still does not guarantee the optimal path.

Branching and pruning algorithm: This algorithm uses an exhaustive search strategy to explore all possible routes. It starts at an initial node and, at each step, generates all possible extensions of the current path and evaluates them. Only those extensions that have the potential to lead to an optimal solution are further explored. A pruning technique is used to avoid exploring branches that clearly will not lead to the optimal solution. This approach guarantees finding the optimal path, but can be computationally expensive for large instances of the problem.

Dynamic programming algorithm: This approach is useful when the number of nodes is relatively small. A table is used to store and compute the minimum distances between all combinations of nodes. The table is iteratively filled and used to construct the optimal route at the end of the process. This algorithm is also guaranteed to find the optimal path, but can require a significant amount of time and memory, especially for larger problems.

Genetic algorithm: This approach is based on principles of biological evolution and uses heuristic search techniques. It starts by generating an initial population of possible routes (chromosomes) and applies genetic operators such as selection, crossover and mutation to evolve the population in each generation. The quality of each pathway is evaluated by a fitness function that considers the total distance traveled. Over time, the population tends to converge to more optimal solutions. This approach can be effective in finding good, but not necessarily optimal, solutions and is useful when there are time or computational resource constraints.

Ant colony optimization algorithm: This algorithm is inspired by the behavior of ants as they search for the shortest route between their colony and a food source. Ants deposit pheromones on the ground, and other ants follow these clues to find efficient paths. In the context of the packet delivery problem, the nodes of the graph would be the intersections and the pheromones would be represented as additional information on the edges. The ants would perform exploratory traverses and, based on the number of pheromones encountered, make decisions to construct the optimal route. This approach is especially useful when there are

multiple packages and additional constraints, such as time windows for delivery, must be considered.

TRANSITION FROM IDEA FORMULATION TO PRELIMINARY DESIGNS

for the selection of possible solutions should be based on the skills and knowledge seen in class for the implementations and structure that give fulfillment to the assignment of the statement therefore the solution of:

Develop a simple computer system using a programming language such as Python or Java, which allows the loading and registration of passenger information, as well as the prioritization of the order of entry and exit of the aircraft according to the established criteria.

Since we could make use of all the algorithms structures seen in the following sections, which allow us to make the indicated ordering of passengers by priority, we could develop a simple computer system using a programming language such as Python or Java.

VALUATION AND SELECTION OF THE BEST SOLUTION

The option of developing a computer system using a programming language such as Python or Java, for loading and recording passenger information and prioritizing the order of entry and exit of the aircraft, is valid because it meets the following criteria:

Application of knowledge acquired in university classes: the development of a computer system using programming languages such as Python or Java is a practical application of knowledge acquired in university classes on programming and algorithms.

Technical feasibility: Python and Java programming languages are widely used and have a large amount of resources and libraries available for the development of computer applications, which makes it technically and economically feasible to develop a computer system with these languages.

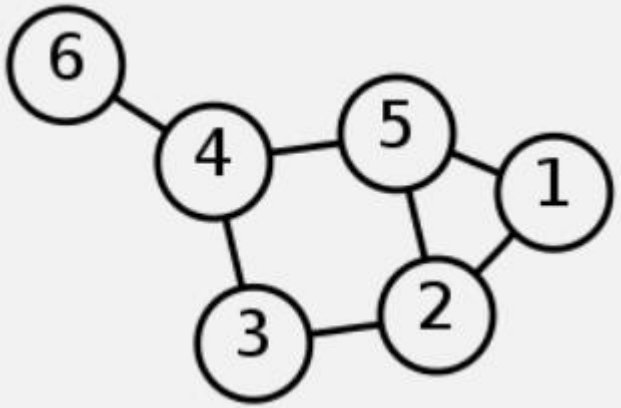
Efficiency in the process: The prioritization of the order of entry and exit of the aircraft according to the established criteria can be carried out efficiently using ordering algorithms and data structures that have been studied in university classes.

Ease of maintenance: The use of programming languages such as Python or Java allows easy maintainability of the system, since these languages are known for

their ease of reading and writing, which facilitates the modification and correction of errors in the code.

In summary, the development of a computer system using programming languages such as Python or Java, for the loading and registration of passenger information and the prioritization of the order of entry and exit of the aircraft, is a valid option because it meets the criteria of application of university knowledge, technical feasibility, efficiency in the process and ease of maintenance.

GRAPHS TAD

TAD < Graph>

Graph = {Vertex, Edges}
{inv: {Vertex(has least one), Edges(i fis direct only a to b, if isnt a to b and b to a)
Operaciones primitivas: addVertex(Value)=Value->Vertex searchVertex(Value)=Value->Verte or notFoundMessage deleteVertex(Value)=Value->deleteVertex, deleteEdges addEdge(Value1, Value2) = Value1, Value2->Edge deleteEdge(Value1, Value2) = Value1, Value2->deleteEdges


```

dfs()=dfs->dfsvisit->vertexObjective
bfs()=bfs->vertexObjective
floydWarshall() = floydWarshall->orderedPairVertex
Dijkstra(Origin) = Dijkstra->shortestWayOrg
Prim(Value) = Value->minimumSpanningTree

```

-addVertex(Value): Modifying Operation

Adds a Vertex to the Graph

PRE{Graph!=null}

Pos{Vertex}

- searchVertex (Value): Analyzing Operation

search a Vertex to the Graph

PRE{Graph!=null}

Pos{Vertex or notFoundMessage}

-deleteVertex(Value): Modifying Operation

Remove a Vertex and its related Edges from the Graphs

PRE{Value==VertexValue}

POS{Vertex==null and Edges==null}

addEdge(Value1, Value2): Modifying Operation

add an Edge in base of the existing vertex (vertex1 is the initial and vertex2 is the final of the Edge)

PRE{Value1 == VertexValueInitial && Value2 == VertexValueFinal}

POS{Edge}

deleteEdge(Value1, Value2): Modifying Operation

delete an Edge from a pair of existing vertex(vertex1 is the initial and vertex2 is the final of the Edge)

PRE{Value1 == VertexValueInitial && Value2 == VertexValueFinal}

POS{Edge == null}

DFS: Analyzing Operation

DFS (Depth-First Search) is an algorithm used to traverse or search a network. Its name is due to the fact that it explores first in depth before backtracking.

PRE{graph j= null}

The dfs() method performs the DFS traversal on the network. First, it sets the color of all vertices to white and the predecessor to null. Then, it iterates over each vertex in the vertex list. If the vertex has not yet been visited (it is white), it calls the dfsVisit() method to visit the vertex and its neighbors.

The dfsVisit() method is the core of the DFS algorithm. It takes a vertex u and a time u as parameters. It increments the time by one and sets the distance from the vertex u as the current time. Then, it changes the color of u to gray to mark it as visited.

Next, it iterates over the neighboring vertices of u. If a neighbor v has not been visited (it is colored white), it sets its predecessor as u and recursively calls dfsVisit() to visit v.

After visiting all neighbors of u, u is marked as black to indicate that it has been fully explored. Finally, the time is incremented by one.

In summary, this code implements the DFS algorithm to traverse all vertices.

POS{vertexObjetive or notFoundVertex}

BFS: Analyzing Operation

BFS, which stands for "Breadth-First Search", is a search algorithm used to traverse or search for elements in a graph or tree in a systematic way. Its name "breadth-first" comes from the fact that it explores nodes in layers, first visiting the nodes closest to the origin node before moving on to those farther away.

PRE{graph j= null}

In summary, this code implements the BFS algorithm to traverse a graph starting from a given source vertex. The algorithm uses a queue to store the vertices being visited and performs a breadth-first traversal, first visiting all vertices at a distance of 1 from the origin vertex, then vertices at distance 2, and so on.

POS{vertexObjetive or notFoundVertex}

floydWarshall():Analyzing Operation

The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted graph, even in the presence of edges with negative weights.

Unlike Dijkstra's algorithm, which finds the shortest paths from a single origin to all other vertices, the Floyd-Warshall algorithm finds the shortest paths between all pairs of vertices in the graph.

PRE{graphs j= null and ArrayVertex!=null}

a method called floydWarshall is declared, which does not return any value (void).

The size of the network stored in the vertexes variable (which appears to be a list of vertices) is obtained and stored in the size variable.

Two two-dimensional arrays distances and next, both of size x size, are created to store the minimum distances and the next vertices in the shortest path.

The matrices distances and next are initialized. The distances array is filled with the maximum possible value (Integer.MAX_VALUE) in all positions except the main diagonal, where it is set to zero. The next array is filled with -1 in all positions.

Next, each vertex of the network is iterated over and the distances and next matrices are updated with the distances and next vertices corresponding to the adjacent vertices. This is done using a nested loop: for each vertex u, a list of its adjacent vertices is obtained and the distances and next matrices are updated with the distance (v.getWeight()) and the index of the adjacent vertex in the vertex list.

Next, the Floyd-Warshall algorithm itself is performed. There are three nested loops that iterate over all vertex pairs (i, j) and a third intermediate vertex k. At each iteration, a check is made to see if the distance from i to j through k is less than the current distance stored in the distances array. If so, the distance is updated and the next vertex is updated to the shortest path stored in the next array.

After the Floyd-Warshall algorithm is completed, the weights and predecessors of the vertices in the original network are updated. We iterate over each vertex u and its adjacent vertices v, and update the weight of v with the corresponding distance in the distances matrix and set the predecessor of v to the corresponding vertex in the vertex list.

In summary, this code implements the Floyd-Warshall algorithm to find the shortest paths in a weighted network and updates the weights and predecessors of the vertices in the original network based on the results of the algorithm.

POS{ orderedPairVertex }

Dijkstra: Analyzing Operation

Dijkstra's algorithm for finding the shortest path from a given source vertex to all other vertices in a weighted graph.

PRE{graphs j= null and ArrayVertex!=null}

The dijkstra method takes an origin parameter, which is the origin vertex from which the shortest paths will be calculated.

The originPosition variable is initialized to 0 and will be used to store the position of the origin vertex in the vertex list.

A priority queue s is created using PriorityQueue and passed a custom comparator ComparatorDistance which will sort the vertices in descending order of their distances.

A for loop is performed to iterate over all vertices except the source vertex. Within the loop, the vertex distances are set to Integer.MAX_VALUE (a maximum value) to indicate that initially no path has been found from the origin to them. The predecessors are also set to null and the vertices are added to the priority queue s.

In the case of the origin vertex, its position is stored in originPosition instead of adding it to the priority queue.

The distance from the origin vertex is set to 0, since there is no distance from the origin to itself.

A while loop is started and executed until the priority queue s is empty.

At each iteration, the vertex with the smallest distance from the priority queue s is extracted using the poll() method.

Iterate over all vertices adjacent to the extracted vertex (u) using a for loop.

For each adjacent vertex, an alternative distance (alt) is calculated by summing the current distance of the extracted vertex with the weight of the arc connecting them.

If the alternative distance (alt) is less than the current distance of the adjacent vertex, an update is performed in the following steps:

The adjacent vertex is removed from the priority queue s.

The distance of the adjacent vertex is updated with the alternative distance (alt).

The weight of the adjacent vertex is set to 0 to indicate that it will no longer be used in future calculations (this is done to ensure that longer paths through this vertex are not considered).

The predecessor of the adjacent vertex is set as the extracted vertex (u).

The updated adjacent vertex is added to the priority queue s.

The while loop continues to extract vertices from the priority queue and perform the above steps until the queue is empty, meaning that the shortest paths from the source vertex to all other vertices in the network have been found.

POS{ shortestWayOrg }

Prim: Analyzing Operation

Prim's algorithm for finding the minimum spanning tree in an undirected weighted graph.

PRE{graphs j= null and ArrayVertex!=null}

A method called "prim" is defined that takes a parameter of type T (probably some generic data type) called "value". The purpose of this method is to find the minimum spanning tree in a network.

A priority queue named "q" is created using Java's PriorityQueue class. Collections.reverseOrder is used to reverse the order of the elements in the priority queue, which causes the elements with lower weight to have a higher priority.

We iterate over all the vertices of the network stored in the "vertexes" list. For each vertex "u":

The weight of vertex "u" is set to the maximum possible integer value using Integer.MAX_VALUE.

The color of vertex "u" is set to white (Color.WHITE).

If the value of vertex "u" is equal to the value provided as parameter, the weight of vertex "u" is set to 0 and its predecessor is set to null.

The vertex "u" is added to the priority queue "q".

An iterator named "it" is created to traverse the priority queue "q".

A while loop is started and executed as long as the priority queue "q" is not empty:

The vertex with the highest priority (lowest weight) is extracted from the priority queue "q" using the poll() method and stored in the variable "u".

Iterate over all vertices adjacent to vertex "u". For each adjacent vertex "v":

If the color of vertex "v" is white, which means that it has not yet been visited:

A for-each loop is performed on the priority queue "q" to search for vertex "s" with the same value as vertex "v".

If a vertex "s" with the same value is found and the weight of "v" is less than the weight of "s", the following is performed:

The vertex "s" is removed from the priority queue "q".

The weight of "s" is updated with the weight of "v".

The predecessor of "s" is set to "u".

The updated vertex "s" is added to the priority queue "q".

The color of vertex "u" is set to black (Color.BLACK).

POS{ minimumSpanningTree }