Shortest Path Computation

Programming Assignment 12:

Design Document

EECS 293

By Kris Zhao

Code Design / Approach:

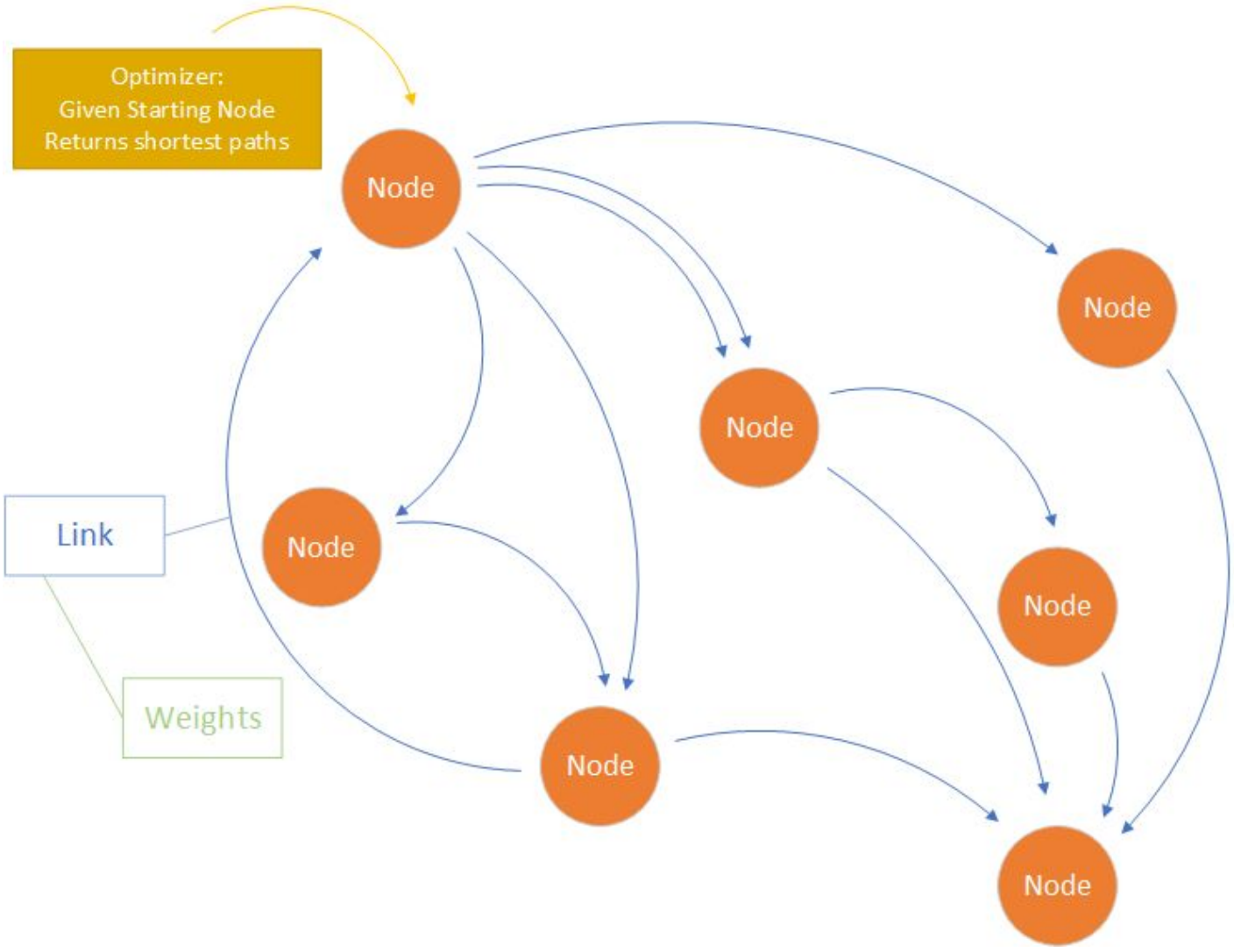
Main Objective:

The primary objective for the program is to create an easily implementable structure for the use of Dijkstra's algorithm. This will consist heavily of structuring the required methods and functionality required.

In order for the program to be widely applicable, there should be no other inputs to Dijkstra's algorithm other than simply a starting node. As we will deal with the necessity of having a time as a weight, we may also require the weight to be considered when picking the next possible nodes.

As Dijstra's algorithm finds the shortest path to every single node in the graph, our program should return a list or set of nodes with the relevant previous nodes set.

Hierarchical Structure:



To match the hierarchical structure, we are going to require various interfaces which will structure the graph. This will include creating the vertices (Nodes), and edges (Links) with weights (Weight). We will then include a static class which will allow us to run Dijkstra's algorithm (Optimizer).

Derivation from project 2-5:

As an overview, the new code will utilize three separate classes as data structures with one main driver to execute Dijkstra's algorithm

Weight

The weight class captures the abstraction of a weight, priority, or time when calculating the shortest path. This is pulled as the flight times in assignments 2-5. As we will also have a reference to different characteristics, as well as times and bandwidth costs, I wanted to maintain a degree of flexibility while running Dijstra's algorithm. In the case of time dependent weights, utilize the arrival time to get to the next node as the weight.

Link

The Link class will capture an abstraction of a directed edge in the group. This will tie the nodes to each other while holding a weight to traverse the link. This will determine optimal routes from one node to another when computing Dijkstra's algorithm. This is akin to the flights that we have seen from previous assignments, which link one airport to another based on a time.

Node

The Node class will represent the vertices on the graph. This is similar to the airports and airport group from previous assignments. Here, we will know the next possible locations from the node's links. While running through Dijskstra's algorithm, we will jump from node to node and remember the weight / previous nodes.

Optimization

The optimization class represents the routefinder class from the previous assignments. Here, we will begin at a given node and utilize Dijkstra's algorithm to compute the shortest path to every single other node connected to the starting node.

Class design and breakdown:

Class: Optimizer

The optimizer class is the base point for any kind of shortest path calculations. We will require that all costs used by the optimizer for shortest path calculations must be non-negative. Let us assume for simplicity that each network has computations only computed once.

Method: public List<Node> shortestPathFrom(Node startingNode, boolean weightDependent)

The shortestPath method will be invoked and passed a node as the starting point. From this starting point, the method will run through all of the reachable nodes from it (given their links). There will also be two behaviors that the algorithm can exhibit, one that skips time based on weights, and one that calculates based on the sum. The shortestPath method should return the shortest paths from the given node to all of the other possible nodes.

For this method, we will require the starting node to be configured with the lowest cost by the user. This is due to the varying possibilities for a lowest cost (including non-numeric lowest costs).

PseudoCode:

```
NextNodes <- Priority queue of the next nodes that will be run through to
             determine the shortest route. The nodes are ordered based on
lowest
             cost to reach.
FinalizedNodes <- List of all completed (shortest route determined) nodes

Add starting node to NextNodes with the smallest weight possible (representing
0)

While there are nodes in NextNodes

      consideredNode <- Remove first off the ordered list NextNodes

      For each link from the consideredNode that is targeting a node not in the
      FinalizedNodes, and, given weightRestricted is true, is still takeable
      (time isn't past the departure time).

            targetNode <- the node at the end of the link.

            If weighRestricted is true
                  If link weight is less than the targetNode
                        Set the weight of the targetNode to the link weight

                        Set the previousNode of targetNode to consideredNode

            Else
                  If link weight + consideredNode weight is less than targetNode
                        Set the weight of the targetNode to the link weight +
                        consideredNode weight

                        Set the previousNode of targetNode to consideredNode

            Add targetNode to nextNodes

      Add consideredNode to FinalizedNodes

Return FinalizedNodes
```

Method: private static void configureNodeIfShorter(Node targetNode, Node previousNode, Weight linkCost, Weight previousCost)

This method is utilized to reduce the complexity and repeated code. Within it, we will compare two costs and configure the passed in node according to the result of the comparison.

Method: private static void verifyInputs(Node startingNode) throws UninitializedStartError

This is a private helper method which verifies the input to the algorithm. This will look to make sure that the nodes will not contain an uninitialized node.

Method: private static void clearCosts(Node startingNode)

This method is utilized to clear the costs of all previous nodes in the network. This will make sure that Dijkstra's algorithm has a clean slate to run each time given the startingNode.

Interface: Node extends Comparable:

The node interface will act as the publicly facing side for the Node in the network. It will be required to hold the following method signatures.

Each signature will be flushed out in depth for the AbstractNode which looks to provide basic implementation.

public boolean costKnown()

public Weight getLowestCost()

public void setLowestCost(Weight lowestCost)

public Node getPreviousNode()

public void setPreviousNode(Node previousNode)

public Set<Link> getLinks()

public void addLink(Link newLink)

Abstract Class: AbstractNode implements Node

The node class will represent a location / destination (vertex of the graph). Each node will then have the capabilities to find the previous location (nullable), as well as the cost to reach this node (sum of the previous weights). Most of these methods will be required by Dijsktra's algorithm but will not be used for any other reason.

Data Structure: private Weight lowestCost

This data structure is used to hold the lowest weight encountered at the node. This will be used for Dijkstra calculations.

Data Structure: private Node previousNode

This will store the previousNode when determining routes with Dijkstra's algorithm

Data Structure: private List<Link> links

This will hold all of the links from this nodes to other ones. These create the building blocks to the graph.

Method: Public boolean costKnown()

This will return whether the node has a cost. This will be utilized for Dijkstra initiation

Method: public Weight getLowestCost()

This will return the current cost computed by Dijkstra's algorithm on the previous iteration.

Method: public void setLowestCost(Weight lowestCost)

This method will set the new shortest cost by Dijkstra's algorithm.

Method: public Node getPreviousNode()

This method will return the previous node on the shortest path for Dijkstra's algorithm. The if there is no previous node, it will be null.

Method: public void setPreviousNode (Node previousNode)

This sets the previous node for the current one.

Method: public Set<Link> getLinks()

This will give you all the links stored in the data structure above. The passed on links can then be used to progress and discover the next new nodes to be reached.

Method: public void addLink(Link newLink)

This method is used for modification to a node. Imagine creating a new flight or new schedule. These will all go to build up the graph connecting the various nodes. We will check to make sure that the link has correct originator.

Interface: Weight<T> extends comparable

The interface provides the necessary methods for any class of type weight. It will require the following method signatures:

public T getWeight();

public Weight weightSumWith(Weight weight);

Each signature is built upon in the AbstractWeight class below.

Abstract Class: AbstractWeight<T> implements Weight

A weight will be used to structure various different weighting systems, be it an arrival time, or monetary costs. With various different types of costs, the only information we will require is how the two weights would be added together (x hp, y stamina for game decisions), and how those values will be compared (so that we know which one costs less or more).

Data Structure: protected T weight;

The weight which is stored. Generics are utilized so that the weight could be times, custom user created classes, or numbers.

Method: public T getWeight()

This method is utilized to return the weight associated with the object. This can be used for comparisons, and general access.

Interface: Link

The interface for link requires the following method signatures:

public Weight getCost()

public Node getTargetNode()

public Node getSourceNode()

public boolean isTakeable(Weight cutoff)

Each is built off in the abstract class which provides basic implementation

Abstract Class: AbstractLink implements Link

The link interface represents any component which will link a node to a different one. The link is crucial for determining the shortest path as the weight associated with it will determine how routes are chosen.

Data Structure: private final Weight cost;

This will represent the cost of traversing the link. This will be any kind of cost as defined by the above class description. These will be used to determine which link should be chosen as the shortest path.

Data Structure: private final Node targetNode;

The targetNode is used to find the termination point for this link. The origin will hold the link, and the destination is the targetNode

Data Structure: private final Node sourceNode;

The sourceNode is to ground the link. This will be verified whenever links are associated to make sure there were no errors.

Method: protected AbstractLink(Weight cost, Node sourceNode, Node targetNode)

To create a link, we simply need to know the cost to traverse the link as well as the node the link is pointing to This will give us the corresponding link object.

Method: public Weight getCost()

The cost for a link will reflect the metric the link is using. This could be based off times or values associated with traversing the link (as described above).

Method: public Node getTargetNode()

The target for the link will be the node this link points to. This effectively connects the two nodes together. We simply return the target node defined above.

Method: public Node getSourceNode()

The source for the link will be the node this link originates from. This effectively connects the two nodes together. We simply return the source node defined above.

Method: public boolean isTakeable(Weight cutoff)

There is a method which determines whether the link is traversable in the case of time determined links. This will simply return whether or not the Weight given cutoff value is after the link's weight.

Error Handling Design:

Standard approach:

Within our interface, we will throw exceptions to the user whenever there is incorrect construction of our defined components, such as incorrect inputs. This will notify the user that their graph is incorrectly constructed so that they know when they have inserted something erroneously. This must be done outside of constructors such as in build methods. As the components will be built up by the user across multiple instances or operations, each error will also be thrown and handled in a decentralized manner by the user.

Otherwise, I do not anticipate many other locations for errors as long as the user follows the structures laid out for them. If there are other errors that arise, we will look to handle them locally rather than globally.

Error avoidance:

Throughout, we want to emphasise error avoidance. This means checking requisite values before employing them in our code. This will hopefully reduce the number of stray errors during and after compiling.

At the same time, most errors should be thrown to the user during construction of objects, etc, as informing the user of the failed graph construction is crucial.

Custom Errors:

In the case that a starting node has not been properly initialized for use within Dijsktra's algorithm, we should throw an UninitializedStartError. This will let the user know that they made a mistake with the creation and utilization of the algorithm.

Testing Approach

In order to verify the operation of our program, we will utilize JUnit testing paired with .

JUnit Testing:

All methods which have been fully defined should be tested using JUnit testing, We will look to provide this testing after the code is complete in order to verify the operations and functionality of our software.

In addition to this, as there is a lot of frameworks and interfaces laid down, we will require testing of basic functionality of extended classes. We note here that there is a substantial number of getter / setter methods. These will not require explicit testing.

Functional Testing:

After unit testing has been completed, we will test our defined structure and functions using large overall tests. This will verify the structure as well as the algorithm working overall.

Stress testing:

In order to stress the system, we will look to put the algorithm under heavy loads. These can find very exceptional errors as well as verify typical functions.

Long Chain Testing:

We will try and run Dijskstra's algorithm through our code on a chain which is extremely long in length. Then, to verify the correctness of the count, we will look at the weight of the final node to make sure it equals the number of links we had created.

Repetitive testing:

In order to verify the operation over multiple repetitions of the algorithm (and to make sure that previous runs does not affect current runs), we will run the algorithm multiple times over an increasing network.