Karl Brown

Algorithms 420

Project Write-up

1. I did not understand the project when I had to approach it for the programming competition, but now in a better state of mind the question is significantly clearer to me. The 3 test cases are easily separated. The graph is connected as that way you can reach the end point and access every part of the world, and the graph should never be complete or else game conditions would be violated. The graph should end up acyclic, as the conditions defined for the game are in order to complete the game. The graph is definitely directed, as one relationship fulfills a requirement and not the other way around. The weights of the edges are not relevant as you can either traverse or not traverse in the game, there is no difficulty between paths. Now that I really understood how the graph functioned, I had to choose a graph algorithm that would fit my needs then implement it. After looking at the Big O cost of adjacency operations, I see that it will less costly to use a List than a Matrix to represent my Graph. Given the fact that there are only speed improvement’s for Incidence Lists over Adjacency Lists during removal by O(|V|), and knowing I will not be performing any removals, I decide on an Adjacency List. I need to modify my implementation of my Adjacency List to check each time I add an item to see if there is already an edge with a different vertex, to classify the gameplay as Nonlinear. Furthermore, I need to check to make sure that the exact opposite edge/vertex combination does not already exist, as if it does the gameplay is Infeasible. I will be assuming that the entry data does not contain improperly formed test cases, only improper data inside the tests cases, for sanity’s sake. After each set of input, I will output either “Nonlinear gameplay possible”, “Linear gameplay”, or “Infeasible game”. Adding a Vertex is O(1), Adding an edge is O(|V|\*Hash Table Access) , checking an edge is O(Hash Table Access), and the Hash Table complexity is O(1) average O(N) worst case complexity. I have coded all the functions in to perform what should be done. I integrated the test cases, and the code functions accurately.

**import** java.util.HashMap;  
**import** java.util.LinkedList;  
**import** java.util.List;  
**import** java.util.Map;  
  
*/\*\*  
 \* Created by Karl on 5/6/2015.  
 \*/***public class** GraphMUD {  
  
 **public static void** main (String []args){  
 */\*\*  
 1. Take input, list by list  
 2. For each list, create a new GraphMUD with max\_vertices equal to n  
 3. Add edges for each line of input data  
 4. Run testCase() and obtain an answer based on creation of Graph   
 \*\*/* }  
  
 **private** Map<Integer,List<Integer>> **Adjacency\_List**;  
 **boolean isInfeasible** = **false**;  
 **boolean canBeNonlinear** = **false**;  
  
 */\*\*  
 \* Constructor to populate Adjacency List  
 \** ***@param max\_vertices*** *Maximum number of vertices currently in use. Determined by n  
 \*/* **public** GraphMUD(**int** max\_vertices){  
 **Adjacency\_List**= **new** HashMap<Integer,List<Integer>>();  
 **for**(**int** i=1;i<=max\_vertices;i++){  
 **Adjacency\_List**.put(i,**new** LinkedList<Integer>());  
 }  
 }  
  
 */\*\*  
 \* Edge creation modified to check for MUD conditions  
 \** ***@param source*** *Source Vertex  
 \** ***@param destination*** *Destination Vertex  
 \*/* **public void** setEdge(**int** source, **int** destination){  
  
 *//If there is another source vertex, it will be non-linear* **for**(**int** i=0;i<**Adjacency\_List**.size();i++)  
 **if**(getEdge(i,destination)){  
 **canBeNonlinear**=**true**;  
 }  
 *//Having a path both directions results in infeasible* **if**(getEdge(destination,source)){  
 **isInfeasible**=**true**;  
 }  
 **Adjacency\_List**.get(source).add(destination);  
 }  
  
 */\*\*  
 \* Check if an edge exists yet  
 \** ***@param source*** *Source Vertex  
 \** ***@param destination*** *Destination vertex  
 \** ***@return*** *Boolean to indicate edge presence  
 \*/* **public boolean** getEdge(**int** source, **int** destination){  
 **if**(source>**Adjacency\_List**.size()) {  
 **isInfeasible**=**true**;  
 **return false**;  
 }  
 **return Adjacency\_List**.get(source).contains(destination);  
 }  
  
 */\*\*  
 \* Gives data about each MUD test case  
 \** ***@return*** *String based on the type of Gameplay experienced  
 \*/* **public** String testCase(){  
 **if**(**isInfeasible**){  
 **return "Infeasible game"**;  
 }  
 **if** (**canBeNonlinear**){  
 **return "Nonlinear gameplay possible"**;  
 }  
 **return "Linear gameplay"**;  
 }  
  
}

1. It becomes clear to me after reading the problem a few times that I will be needed to input data into a graph and then process it. I broke down that the test case was an individual exercise, not multiple potential outcomes. I am not going to try so hard to anticipate malformed test cases once again, and assume we are working with quality inputs. The graph is connected as Munich is under analysis and you can reach every point of Munich from another part of Munich, but the graph cannot be complete as the autobahn and regular roads are separate entities and cannot be treated identically (amongst other reasons). The graph is acyclic, as the graph cannot be cyclic or you would drive in circles. The graph is definitely not directed, as that way one can move either way from point A to point B on a road. The weights of the Edges are absolutely relevant as a fun/distance factor. I have begun to understand how this problem is going to work out, and I see that I will likely be using a tree to handle this problem. I ended up selecting Dijkstra’s algorithm, as I wanted to use a Greedy approach and utilize a shortest path algorithm. A typical Edge style was used, incorporating Start Vertex/End Vertex/Weight characteristics, but an additional field to accommodate our Autobahn constraint. I used a Vertex class implementing comparable, in order to hold all of the Edges and sort them. Then I used a Graph constructor to create a Graph from the Edges supplied by the code. Following such I implemented Dijkstra utilizing a binary tree. The Dijistrka algorithm must be additionally modified to accommodate the fact that the number of Autobahn miles must be kept track of, and that entries with more Autobahn miles must be prioritized. In order to do so, the core algorithm is not modified, but the Edges are modified as described and the final calculation of shortest path is (taking into account ties over-ruled by Autoban distance). Utilizing a strong tiebreaker model in the actual Dijkstra method, in combination with a recursive method inside the Vertex class that calculates the Autobahn method (currently pending 100% implementation for third test case), I alter how the algorithm decides to keep edge weights minimal. I comment the code completely. I implement the three test cases, and the code functions properly except for a dynamic programming implementation of autobahn distance (a function I added to the Dijkstras algorithm) being off by 1 in the third test case (the first 2 are perfect).