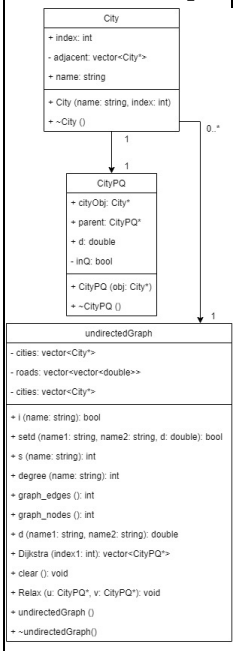
ECE250 – Project 5
Single Source Shortest Path - Dijsktra's Algorithm Design Document

Qinying Wu, q227wu

Apr 16th, 2020

1. Overview of Classes

Class Name	Description	Member Variables and Functions (legend: *public*, *private*)
City	The city object in the undirected graph	*index: int* the index of the city in the cities vector for accessing in Relax() *name: string* the name of the city *adjacent: vector<City*>* the cities that have a road connected to the current city
CityPQ	The city object in the priority queue	*cityObj: City** the city object in the undirected graph *d: double* the weight of the shortest path from the start city to the current city *parent: CityPQ** the predecessor of the current city in a path *inQ: bool* flag to indicate the state whether the city is waiting to be visited or not
undirectedGraph 	The undirected graph	*cities: vector<City*>* storing the list of cities in the graph (nodes) *roads: vector<vector<double>>* storing the road connection between cities *road_count: int* store the number of roads in the graph *i (name: string): bool* inserts a city into the graph. Returns true if the city is inserted successfully, returns false if the city already exists in the graph *setd (name1: string, name2: string, d: double): bool* assigns or updates a distance d to the edge between two nodes name1 and name2. Returns true if assigned or updated successfully. Returns false if one or both cities do not exist, or invalid d (<=0) *s (name: string): int* search for a city with the specified name, parameter name is the name of the city. Returns the index of the city stored in the cities vector if found, FAILUREINT if not found *degree (name: string): int* returns the degree of the given city, parameter name is the city name *graph_nodes (): int* returns the number of cities in the graph *graph_edges (): int* returns the number of roads in the graph *d (name1: string, name2: string): double* returns the weight of the edge between the two nodes name1 and name2 if both cities are found *Dijkstra (index1: int): vector<CityPQ*>* returns the vector of CityPQ objects after performing the Dijkstra's algorithm containing the shortest path from city1 indicated by index1 to city2 indicated by city2. Parameters index1 is the corresponding index of the city in the cities vector of the undirectedGraph object *clear (): void* deletes all the cities and roads from the graph *Relax (u: CityPQ*, v: CityPQ*):* void determines the minimum path weight from a city to its adjacent city. Parameter u is the originating city, v is u's adjacent city (there exist a path between city u and v)
Constant FAILURESTR: string = "failure" for indicating failure state for string functions FAILUREINT: int = -1 for indicating failure state for integer functions SUCCESS: string = "success" for indicating success states INFINITY: double = std::numeric_limits<double>::infinity()		

2. Constructors and Destructor Decisions

Class Name	Constructor	Destructor
City	Takes in string parameter name, integer parameter index and assign it to the respective member variables.	Clears the adjacent vector
CityPQ	Takes in the pointer-type City object obj and assign it to the cityObj. Parameter inQ is assigned false, d is assigned to INFINITY, and parent is assigned to nullptr	cityobj and parent are dereferenced to nullptr
undirectedGraph	Member variable road_count is initialized to 0	Clears the road and cities vector

3. Performance Analysis (V is the count of vertices in the graph, E is the count of edges)

Dijkstra (The PQ is a std::vector to represent a binary heap)	$T_{Total} = T_{initialize\ cityPQ\ objects} + T_{Relax} + T_{Extract\ min} + T_{tinsert\ cities\ into\ PQ}$ $T_{initialize\ cityPQ\ objects} = O(V)$ for all scenarios $T_{Relax} = E*O(1) = O(E)$ Relax is performed the number of times equivalent to the count of edges in the graph $T_{Extract\ min} = O(1)$ since it is always the first element in the PQ (PQ is a minimum binary heap) $T_{insert\ cities\ into\ PQ} = E*O(lg(V)) = O(ElgV)$ every time a city is inserted to the front of the queue, binary heap sort is performed to always make sure the vertex with the minimum distance is at the front of the queue $T_{total} = O(ElgV)$
s	O(V) iterate through the list of existing cities in the graph to determine if any of them match the parameter city
i	O(V)+O(E)+O(1) =O(V+E)

	O(V): run the s function to determine whether the city to be inserted is already in the graph O(E): increase the vector capacity that stores the road connections between cities in the graph after insertion O(1): insert the city into the graph by appending it to the cities vector of the undirectedGraph class
setd d	2*O(V)+O(1) =O(V) 2*O(V): run the s function twice (once for each city parameter) to determine whether they are existing in the graph O(1): setting (setd) or returning (d) the distance of the road between the two cities in the adjacency matrix
degree	O(V)+O(1) =O(V) O(V): runs the s function to see whether the city in the parameter exists in the graph O(1): determine the count of adjacent cities of the city given as the parameter
graph_nodes graph_edges	O(1) returns the count of the cities vector of the undirectedGraph object (graph_nodes) or the value of the road_count member variable (graph_edges)
clear	O(1) clears the cities and the roads vector in the undirectedGraph by calling the built-in vector clear() function
BinaryHeapSort	O(lgV) performs heapify starting at the given root (parameter)
Relax	O(1) assigns d and parent to each of the adjacent cities of the current-processing PQ city object

4. Test Cases

for functions with two cities as the parameter, also test it by swapping the order of the cities when passing as parameters a) i – inserting a city i) insert to an empty graph (no cities no roads) ii) insert a unique or duplicate city to a non-empty graph b) setd – insert a road, d – return the road value between two cities, i) to an empty graph (no cities no roads), before/after calling the clear function ii) before/after inserting a city iii) between two identical cities, two distinct existing, or two non-existing cities, or between one existing and one non-existing city c) s – search for a city in the graph i) search an existing or non-existing city ii) search in an empty graph or before/after calling the clear function iii) search before/after calling i d) degree – determine the count of adjacent cities to a city i) pass an existing or non-existing city in the graph as the parameter e) graph_nodes, graph_edges (also for degree) i) when the graph is empty or before/after calling the clear function ii) before/after inserting a city, before/after calling the setd function f) Dijkstra (print_path and shortest_d) i) pass in two identical cities, two distinct existing, or two non-existing cities, or between one existing and one non-existing city ii) pass in two existing cities that are not reachable from one another iii) call when graph is empty, before/after calling the clear function iv) call before calling i/setd g) clear i) when the graph is empty/non-empty ii) before/after calling i/setd h) Relax i) Passing two identical/distinct cities as the parameter i) BinaryHeapSort i) Set the root at different heights of the binary tree	Example INPUT: clear setd city1;city2;33 d city1;city2 shortest_d city1;city2 print_path city1;city2 graph_edges graph_nodes i city1 i city1 s city1 i city2 i city3 i city4 s city2 setd city1;city1;22 setd city1;city2;22 setd city1;city3;1 setd city2;city1;1 setd city1;city4;1 d city1;city1 d city2;city1 d city6;city3 shortest_d city1;city3 shortest_d city4;city3 print_path city1;city2 print_path city1;city1 degree city1 degree city2 degree city6 clear s city1 degree city1 s city2 setd city1;city2;33 d city1;city2 shortest_d city1;city2 print_path city1;city2 graph_edges graph_nodes	Example OUTPUT: success failure failure failure failure number of edges 0 number of nodes 0 success failure found city1 success success success found city2 failure success success success failure direct distance city2 to city1 1 failure shortest distance city1 to city3 1 shortest distance city1 to city3 2 city1 city2 failure degree of city1 3 degree of city2 1 failure success not found failure not found failure failure failure failure number of edges 0 number of nodes 0