Computer Science 1D

Project 2

Take Me to the Baseball Game

**Team**:

Throw – Catch



**Team Members**:

Ethan Slattery – Product Owner

Sarah Singletary – Team Member 1

Osvaldo Moreno Ornelas – Scrum Master

Connor Peace – Team Member 2

Contents:

I) Big-O Analysis:

* Prim Minimum Spanning Tree.
* Djikstra Algorithm
* QuickSort

II) Data Structures:

* Map
* Heap
* Priority Queue
* Skip List
* STL List
* STL Vector

**Big-O Notation:**

* **Minimum Spanning Tree:**

In order to calculate the shortest distance to visit all the stadiums on each of the categories, the team made use of Prim’s Minimum Spanning Tree Algorithm.

The algorithm runs in O(n2) worse case Scenario. The reason is that the algorithm uses a while loop (n) to iterate through the edges in the graph, and an inner for loop to iterate through the near vertex (n). Best case, the graph has one vertex making it loop only in the while loop (n). Worse case for vertex number > 1, the algorithm iterates n vertex inside n positions, . The following loops does not alter the algorithm, since they are not dependent from each other (is never n\*m\*l\*p). See excerpt 1.1.

Code Excerpt 1.1

template <typename E>

typename Graph<E>::EdgeList Graph<E>::PrimJarnek() {

unvisitAll(); // ensure all edges and vertices are unvisited

EdgeList usedEdges; // List of edges to use in MST, for return

EdgePQueue edgePQ; // Priority Queue of unused edges

unsigned int VertexCount = 0; // Count of vertecies visited

// add first vertex to the list and set it to visited

vertices\_.begin()->visit();

VertexCount++;

// add incident edges to PQueue

EdgeItrList incident = vertices\_.begin()->incidentEdges();

for(EdgeItrItr i = incident.begin(); i != incident.end(); i++){

edgePQ.push(\*\*i);

}

// loop while vertecies in the done cloud is less than total

while( VertexCount < vertices\_.size() ) {

// Get Rid of any edges that are incident to two visited vertex

while(edgePQ.top().start().visited() && edgePQ.top().end().visited()){

edgePQ.pop();

}

// add shortest edge to the list and visit, then pop.

usedEdges.push\_back(edgePQ.top());

edgePQ.top().start().visit();

edgePQ.top().end().visit();

// Get incident lists from the end vertices

EdgeItrList endIncident = edgePQ.top().end().incidentEdges();

EdgeItrList startIncident = edgePQ.top().start().incidentEdges();

// pop the smallest and increment vertexcount

edgePQ.pop();

VertexCount++;

// add unvisted edges from start to PQueue

for(EdgeItrItr i = endIncident.begin(); i != endIncident.end(); i++){

if( !(\*\*i).start().visited() || !(\*\*i).end().visited() ){

edgePQ.push(\*\*i);

}

}

// add unvisted edges from end to PQueue

for(EdgeItrItr i = startIncident.begin(); i != startIncident.end(); i++){

if( !(\*\*i).start().visited() || !(\*\*i).end().visited() ){

edgePQ.push(\*\*i);

}

}

} // END OF MST WHILE LOOP

return usedEdges;

}

* **Dijkstra Shortest-Path Algorithm.**

In order to find the shortest path from point A to point B, we implemented the Dijikstra Algorithm.

The algorithm runs in O(n2) worse case. The reason for this is that the algorithm uses a priority queue (O(nlog(n))) plus a n2 because of the need to loop until n items are removed from the queue, then looping through the vertices of the last removed location (O(n)), plus relaxation only takes O(1) (constant) to be executed. See excerpt 1.2.

N2 + nlog(n) + 3 <= n2  c > 1

Code Excerpt 1.2

template<typename E>

void Graph<E>::Dijkstra(const E &e)

{

// Comparitor class for vertexItr, pushes smallest weights to the bottom

struct distanceCompare {

bool operator()(const VertexItr &lhs, const VertexItr &rhs) {

return (lhs->getDistance() > rhs->getDistance());

}

} compareDist;

// Reset the weight and parent of all vertex

resetDijkstra();

// find the vertex to use as a starting point

VertexItr eItr = findVertex(e);

// set source distance to 0

eItr->setDistance(0);

// Load iterators to vertices into a psudo PQueue

VertexItrVector unusedVertex;

for(VertexItr i = vertices\_.begin(); i != vertices\_.end(); i++){

unusedVertex.push\_back(i);

}

// actual algorithm

while(!unusedVertex.empty()){

// re-sort every time since weights have changed

std::sort(unusedVertex.begin(), unusedVertex.end(), compareDist);

// grab the next vertex with the smallest distance, and then pop it

VertexItr currentVertex = unusedVertex.back();

unusedVertex.pop\_back();

// IF the smallest vertex has weight infintiy then it is unconnected, skip

if( currentVertex->getDistance() != INF ){

// get iterators to all the neightbors of current vertex

VertexItrVector neighbors = currentVertex->adjacentVertex();

for(typename VertexItrVector::iterator j = neighbors.begin(); j != neighbors.end(); j++){

// relaxation algorithm

int sumDistance = currentVertex->getDistance() + currentVertex->distanceTo(\*j);

if( sumDistance < (\*\*j).getDistance() ){

(\*\*j).setDistance(sumDistance);

(\*\*j).setParent(currentVertex);

}

}//END FOR LOOP

}//END INF IF STATEMENT

}//END WHILE

}//END DIJKSTRA

* **Generic Quicksort to sort all the stadiums.**

In order to organize the list of stadiums by their different properties, we used a quicksort that would allow us to do so in the most effective way. The problem of an overflow or an array out of bounds was solved by using a vector instead of an array for storing data.

The sort runs in nLogn best case (O(n2) worst case). When used with the right pivot (the middle) it can be highly efficient (O(nlog(n)) and avoid a stack Overflow. See excerpt 1.3.

Code Excerpt 1.3

// Performs a Quick Sort using the comparator passed in

template< typename Iterator, typename Compare >

void QuickSort( Iterator first, Iterator last, Compare compare ) {

if(first != last) {

Iterator left = first;

Iterator right = last;

Iterator pivot = left++;

while(left != right) {

if( compare(\*left, \*pivot) ) {

++left;

} else {

while( (left != right) && compare( \*pivot, \*right ) )

--right;

std::iter\_swap( left, right );

}

}// End while(left != right)

--left;

std::iter\_swap( pivot, left );

QuickSort( first, left, compare );

QuickSort( right, last, compare );

}// End if(first != last)

}

// If a comparator isn't passed in, this sorts it from least to greatest

template< typename Iterator >

inline void QuickSort( Iterator first, Iterator last ) {

QuickSort( first, last, std::less\_equal< typename std::iterator\_traits< Iterator >::value\_type >() );

}

Data Structures

**Map:**

The Map data structure is the main structure for the stadiums. The stadiums are stored in a map as a vertex with different properties, and edges that represent the distances between each of the stadiums. The map is used to implement the trips around the stadiums and to give the costumer the distance that will be traveled in every trip.

The map used in the application is an adjacency list map. The list uses 2 list that are related to each other, but not dependent, meaning that it is not a matrix (O(n2)). The map is a list of vertex that connect to a list of edges by the use of iterators. This implementation was chosen to increase the efficiency of the operations.

Operations such as initialization take O(n) worse case. Insertions takes O(1), instead of O(n2) as in a array matrix implementation. Comparisons also take (O(1)).

**Heap:**

The heap data structure is used inside the Dijikstra algorithm to store the vertices in the graph in order of cost. The queue is vector based, in order to avoid crashes, memory leaks, and make debugging easier.

The heap runs in O(log(n)) when bubbling up or down. Insertion occur in O(1) plus the bubble up (nlogn + 1). Deletions runs with the same efficiency.

**Priority Queue:**

The priority queue is used as a wrapper class for the heap. The priority queue is used inside the Djikstra algorithm to store all the edges and pop them in order to navigate in the shortest distance. The priority queue executes in the same notation as the Heap O(log n).

**Skip List:**

The skip list was used to store the list of stadiums in the application. It is the best option to store the stadiums in alphabetical order and access them in a smaller number of comparisons. The skip list drops the stadiums into the graph in order to perform any operations for a trip.

The skip list gives the best performance in deletion (O(log(n)) < O(n)) and addition (O(log(n)) < O(n)). Search is also more efficient (O(log(n)) because all other data structures use O(n) to search (excluding trees).

**STL List:**

The STL List was used in the program for the graph. The graph uses the STL list as a form to store all of the vertices and edges in a list. The STL List is also used to store items in the moment of display. Makes GUI implementation simpler and easier to maintain.

Using a list is more efficient since Deletions and insertion run in O(1). And there is no worry about having empty spots in the list or going out of bounds or memory allocation (STL takes care of that).

**STL Vector:**

The STL Vector is used as the inner list for the Heap. It is used because it ensures more maintainability and also ensures that the data inside the heap will be easier to track and see. It is not used to store any other data inside the program.

The Vector helps us make insertion and deletions in the heap in O(1), and bubble up and down O(logn) without the need to check for NULL pointers or leaks that might make the program more inefficient or more costly to maintain.

**Sources:**

1) Goodrich, Michael T., Roberto Tamassia, and David M. Mount. *Data Structures and Algorithms in C*. Hoboken, NJ: Wiley, 2011. Print.