CO360-Advanced Data Structures

Assignment -2

Prim's Algorithms using different heap data structures

Date : 08-April-2018

TEAM
Arvind Ramachandran 15CO111
Aswanth P P 15CO112
Joe Antony 15CO220

The source code for prims algorithm is mainly divided into three parts that is one main file used to read input from file and call MST generate function and second one graph.h used to run Prim's algorithm and the third one respective heap file where functions related to each heap is written.The file structure is explained below

File structure

1. Main File (main.cpp)

Purpose :

1. Read the graph properties stored in a file to program variables
2. Call generateMST() function using three different heap
3. Display time elapsed for PRims algorithm execution using each heap

2. Prims Algorithm (Graph{ .cpp, .h})

Purpose:

1. Create graph of vertex V
2. addEdge for the graph created
3. generateMST for the graph created using different heap specified as argument

Functions and complexity :

1. Graph(int V)
   This parameterised constructor that create an adjacency list of v nodes
   adj = new list<pair<int, int> > [V]
   Graph is stored as list of pair : list< pair<int, int> > *adj
   Time complexity : O(1)

2. void addEdge(int u, int v, int w)
   If u and v represent an edge with cost w then this data can be stored on list head of u with pair value {v,w}

That is each node in the list contains a pair of value which indicate destination and cost from the respective list head.
There is only two push back operations are needed to add each edge
Time complexity : O(1)

3. void generateMST(int heap_type)
   Generate minimum spanning tree for the given graph using heap specified as argument by Prims algorithm

3. Binary Heap (BinaryHeap{ .cpp, .h})

Complexity Analysis :

1. BinaryHeap()
   Constructor used to initialize number of operations variables to zero
   Time complexity : O(1)

2. pair<int, int> top()
   It is used to return the top node from the list:vector<pair<int,int>>nodes
   If the list is empty returns the pair (-1,-1)
   Time complexity : O(1)

3. pair<int, int> pop()
   It is used to pop out the minimum node from the heap using a call to extractMin() function .If node list is empty return a pair {-1,-1}
   Time complexity : O(1)

4. void push(pair<int, int>)
   It is used to push the value provided as argument in to the heap using insertValue() function

5. int getOperations()
   It is used to return the value of variable operations
   Time complexity : O(1)

6. pair<int, int> extractMin()

It is used to extract the minimum value in the heap which is located in the very first location and erased from the heap once it is extracted.Also calls minHeapfy() inorder to maintain the heap property after the removal of a node
Time complexity : O(log n)

7. void insertValue(pair<int, int>)
   It is used to insert value pair provided as argument.It appends the value provided to the end of node list. Later swap accordingly to build the heap. Since each call to insertValue() takes o(log n) times to construct a heap of size of n need to make n calls to inserValue()
   Time complexity : O(nlog n)

8. void minHeapify(int)
   It is used to maintain min heap property after extract min operation . This is done by dividing the heap into two sections and selecting one among them and proceed further using recursive call . hence time complexity will be o(log n)
   Time complexity : O(logn)

   The overall complexity using binary heaps will be n*O(logn)+m*O(logn)= **O((n+m)logn) .**

3. Binomial Heap (BinomialHeap{.cpp, .h})

Complexity Analysis :

1. Creation of a Binomial Heap :
   Creating and returning a new heap.
   Time Complexity : O(1)

2. Finding the minimum key :
   Returns a pointer to the node in heap H whose key is minimum.
   Time Complexity : Because there are at most lg n + 1 roots to check, the running time of this function is O(lg n).

3. Union of two binomial heaps :
   Creates and returns a new heap that contains all the nodes of heaps H 1 and H 2 .
   Time Complexity : O(lg n)

4. Insert Node :
   Inserts node x, whose key field has already been filled in, into heap H .
   Time Complexity : O(1)

5. Extract Min :
   Deletes the node from heap H whose key is minimum, returning a pointer to the node.
   Time Complexity : O(lg n)

6. Decrease Key :
   Assigns to node x within heap H the new key value k, which is assumed to be no greater than its current key value.
   Time Complexity : O(lg n)

The overall complexity using binomial heaps will be n*O(logn)+m*O(logn)= **O((n+m)logn) .**

4. Fibonacci Heap (FibonacciHeap{ .cpp, .h })

Functions and complexity :

1. void insertATreeInHeap(Node *)
   It is used to insert a tree to the main heap
   Time complexity: O(1)

2. list<Node*> removeMinFromTreeReturnFHeap(Node *, Node*&)
   It is to extract and remove the min value from the heap
   Time complexity: O(log n)

3. Node* getMin()
   Return the minimum data pair ,return (-1,-1) if it is empty
   Time complexity: O(1)

4. pair<int, int> extractMin()
   Deletes the node from heap H whose key is minimum, returning a pointer to the node.
   Time complexity: O(log n)

5. Node* mergeFibonacciTrees(Node *, Node *)
   It is used to merge two fibonacci trees provided as argument
   Time complexity: O(1)

The overall complexity using Fibonnaci heaps will be n*O(logn)+m*O(1)= **O(nlogn+m).**