CS1.201
Data Structures and Algorithms

Project Report

Tree Search Library

Team 5

Atharva S. Gogate - 2020112001

Agrim Rawat - 2020102037

Akash C.R. - 2020111004

Yash Agrawal - 2020114005

Keshav Agarwal - 2020102048

Data Structures Used

Priority Queue (PQ)

- A **priority queue** is a data structure similar to a heap in which every element has a priority corresponding to it.
- An element with "higher" priority is served/dequeued before that with "lower" priority.
- In our priority queue, the following operations have been used:
    a. `init_pq`: Initializes a PQ and its parameters.
    b. `input_pq`: Takes an input array and adds to priority queue.
    c. `sift_up`: Sifts up a node to correct place w.r.t. priority
    d. `sift_down`: Sifts down a node to correct place w.r.t. priority
    e. `peek`: Returns the highest priority node in the priority queue
    f. `insert_node`: Inserts a node into the priority queue.
    g. `pop`: Deletes the highest priority node
    h. `destroy_pq`: Frees all pointers.

Tree

- A **tree** is a data structure that works like a hierarchical tree structure. It is a **connected acyclic graph**.
- In other words, a connected graph with no cycles is called a tree.
- The branches of the tree are known as **edges**.
- Elements of the tree are called **nodes**.
- A node with no children is called a **leaf node**.
- In our tree data structure, we store the `number`, `parent` index, `value` and the children in a `children` array.
- The following operations have been used:
    a. `init_tree`: Initializes a tree and its parameters.
    b. `create_tree`: Creates a tree from a given input array.
    c. `print_tree`: Prints the tree
    d. `destroy_tree`: Frees all pointers.

Algorithms and Complexity

Priority Queue

Sift Up

- **Working:**
  Keep swapping the node with its parent till we either reach the root node or we reach a parent with higher priority
- **Time Complexity:** $O(\log n)$.
  In worst case, we need to swap a leaf node till we reach the root node at the height of the priority queue which is $\log n$

Sift Down

- **Working:**
  a. Keep swapping the node with its children till we either reach the last node or we reach a child with lower priority.
  b. Select the child with higher priority.
  c. If node already has higher priority than either child then return.
  d. Else swap with higher priority child.
- **Time Complexity:** $O(\log n)$.
  In worst case, we need to swap the root node till we reach the maximum depth/height of the priority queue which is $\log n$.

Insertion

- **Working:**
  a. Insert the node at the end of priority queue in $O(1)$
  b. Sift up the node to correct position in $O(\log n)$
  c. Update the size of priority queue
- **Time Complexity:** $O(\log n)$.
  In worst case, we take the last node to the root at height of priority queue.

Deletion

- **Working:**
    a. If the priority queue is empty we return NULL
    b. Swap first and last element in pqueue in $O(1)$
    c. Decreasing the PQ size by 1 makes the last node deleted
    d. Sift down the new top element to regain PQ property in $O(\log n)$
- **Time Complexity:** $O(\log n)$.
  In worst case, we take the root node to the last node at max depth of the priority queue.

Peek

- **Working:**
    a. If pqueue empty, we return NULL
    b. Else return the top-most node
- **Time Complexity:** $O(1)$

Tree

Create Tree

- **Working:** This function just takes the input node's state number and puts it at that number in the tree. It also puts the pointer to the node in the `children` arrray of its parent node.
- **Time Complexity:** $O(N)$, where N is the size of the input.

Depth First Traversal

- **Working:**
    a. Here, first we go to the depth of one node as much as we can, and then we do the same for its sibling nodes.
    b. In this process, the node with the max seen time is chosen to be traversed first.
    c. So, the comparator tells whether the first node has a greater seen time or not and based on that we decide its position in the priority queue.
- **Time Complexity:** $O(N)$, where N is the size of the input.

Breadth First Traversal

- **Working:**
    a. Here, first we go to all the children of one node, and then we do the same for their children.
    b. In this process, the node with the least seen time is chosen to be traversed first.
    c. So, the comparator tells whether the first node has a lesser seen time or not and based on that we decide its position in the priority queue.
- **Time Complexity:** $O(N)$, where N is the size of the input.

Greedy Traversal

- **Working:**
    a. In this we select the node with the maximum values.
    b. So, the comparator returns whether the first node has greater value stored than the second node.
- **Time Complexity:** $O(N)$, where N is the size of the input.

Procedure and Working

In order to use the tree search library so as to traverse the tree using any desired algorithm we have to follow the below mentioned procedure:

1. Take input into input array:

- We will create an array nodes.
- We take user input and store all the necessary information in these nodes which are stored in an input array.

2. Create a tree with the input array:

- We call the `create_tree` function, which puts the input node at its index in the tree. It also puts the pointer to the node in the `children` array of its parent node.

3. Traverse the tree:

We have to traverse the created tree using any of the user mentioned tree search algorithms. This can be implemented using Priority Queues as follows.

1. Push the root node in PQ.
2. Pop the topmost node:
   The highest priority node is deleted. Priority of the node is decided based on the traversal algorithm.
3. Push its children into the while also maintaining PQ property, which will keep the highest priority element at the top
4. Repeat steps 2 and 3, till the priority queue is completely empty, i.e., all the nodes are popped out based on their priority.

4. Statistics

Allocate the memory to global arrays `max_depth`, `avg_depth`, `visited`, `max_child` and initialize them with `0` using `calloc()`
We first allocate the value of `avg_depth[0] = output[0].depth = 0` and `max_depth[0] = output[0].depth = 0`.

1. `max_depth`:
   - For `max_depth[i]`, we compare the depth of the `output[i]` with the `max_depth[i-1]` where $i \in [1, \text{input size}]$.
   - We set `max_depth[i]` to the maximum out of `max_depth[i-1]` and `output[i]`.
   - Then we repeat the process with `i+1`.

   $$\text{max\_depth}[i] = \max(\text{max\_depth}[i-1], \text{output}[i])$$
2. `avg_depth`:
   - For `avg_depth[i]`, we multiply `avg_depth[i-1]` with `i` and add the depth of `output[i]` and finally we divide it with `i+1`.
   - We do this because the `avg_depth` array stores the values of the average depth of all the nodes that have been traversed till `i`, where $i \in [1, \text{input size}]$.

   $$\text{avg\_depth}[i] = \frac{\text{avg\_depth}[i-1] * i + \text{output}[i]}{i+1}$$
3. `max_child`:
   - At the end of every iteration, the `max_child[i]` = maximum number of children of any node till the i^{th} iteration.

- We take a frequency array `visited`.
- Value of `visited[parent]` increments if one of its child is visited.
- On i^{th} iteration, we update the `visited[parent]` of the child visited.
- The max children at that point is `visited[parent]` if `visited[parent]` is greater than `max_child[i-1]`, else it is `max_child[i-1]`.

$$\text{max\_child}[i] = \max(\text{max\_child}[i-1], \text{visited}[\text{parent}])$$

Division of Work

1. Atharva S. Gogate
    - node structure
    - comparator function
2. Agrim Rawat
    - Priority Queue
    - Tree
3. Akash C.R.
    - Tree traversal and printing
    - Utility files
4. Yash Agrawal
    - Statistics functions
    - Plotting the stats
5. Keshav Agarwal
    - Main function
    - Tree

Written By - Atharva S. Gogate, Agrim Rawat, Akash C.R., Yash Agrawal, Keshav Agarwal