ADJ Example: Updating a Priority Queue

This handout is an example of using the ADJ approach for an algorithmic problem.

Problem

Design an efficient algorithm to update the priority of an entry inside of a priority queue. Analyze the problem, design an algorithm for updating the priority queue, and justify the algorithm's optimality.

Analysis

To begin, we need to better define the problem. We will assume that the priority queue (PQ) is a maximum PQ, and that the specific implementation we need to support is that given in Algorithms by Sedgewick. These two assumptions are reasonable as they are associated with SER222, which uses the Sedgewick textbook and which covers only maximum PQs in lecture.[1] The algorithm will assume that the data structure it operates over is already a valid PQ, and we expect that it produces a valid PQ as well. The goal of this algorithm is to solve the problem where some element inside of a maximum PQ needs to be moved to a different location because an update to its priority violates the heap order rule. For the target element, we will need to (if necessary) adjust its position in the tree such that the tree after execution of the algorithm will be a valid PQ (functionally meaning that operations will work as intended).

The Goal

A potential design for this problem would look like the pseudocode for an algorithm that can update a priority queue (Definition 5). Based on the problem statement, a valid solution can be represented generically as the pseudocode outline in Algorithm 1. Here, the state of a priority queue is modeled as three global variables: N, keys, values. Per definition 5, the state of these variables will be a complete binary tree that is heap-ordered. The algorithm will take both the value of the entry that is changing, and its new priority.

Algorithm 1 Pseudocode for outline of potential solutions.

```
Integer N       //number of entries in PQ
Key[] keys      //contains N elements
Value[] values  //contains N elements

void update(Value val, Key new_pri):
    //Input: the value of the node is changing, and the new priority.
    //Output: updated contents in keys and values.
```

Metrics

From the prompt, there are two metrics that must be evaluated to justify a particular solution as optimal.

[1] Please note that this justification for the reasonableness of these assumptions is strongly tied to this particular problem. In homework, it unlikely you will be able to use this same reasoning.

- M1: The ability of the algorithm to process a heap sorted array with at most one node which violates the heap rule (Definition 4) and produce a heap sorted array with no violations of the maximum heap rule (Definition 4). The array must contain a complete tree (Definition 2) at all times.
- M2: The efficiency of the algorithm. For a cost metric, we use the number of comparisons and exchanges as a measure of computational time needed for a particular design.

A design is optimal if it meets M1, and for M2, it is at least as fast as other solutions that satisfy M1.

Design

Note that a priority queue cannot contain duplicate keys, since then the priority becomes ambiguous based on which element we see first (when running delMax), and then the priority is no longer deterministic. The proposed solution can be seen in Algorithm 2.

Algorithm 2 Pseudocode for update algorithm.

```
void update(Value val, Key new_pri):
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N-1:
      if values[i] = val:
        idx = i

    if idx = NULL:
      return

    //Step 2: if needed, adjust position of target key/value.
    keys[idx] = new_pri;
    if less(keys[idx/2], new_pri):
      swim(idx)
    else if less(new_pri, max(keys[2*idx], keys[2*idx]+1):
      sink(idx)
```

Justification

We need now need to show that the design meets M1 (correctness requirement), and M2 (efficiency goal).

M1

This will be shown by cases:

- **Case 1:** tree does not contain node with value.

  Since we are working with a priority queue, we know the tree will be complete before we run the algorithm. We need to show that after the algorithm executes, the array is heap-ordered (Definition 4) and the tree is complete (Definition 2). In Step 1 of the algorithm, we loop over element and check its contents. No changes are made to N, keys, or value, so after that step Definition 2 will hold, and Definition 4 will hold iff it held for the original tree. If there is no node to update, and the algorithm will terminate before step 2 (see if-conditional), the algorithm will be sound for this case.

- **Case 2:** tree contains node with value.

Since we are working with a priority queue, we know the tree will be complete before we run the algorithm. We need to show that after the algorithm executes, the array is heap-ordered (Definition 4) and the tree is complete (Definition 2). As discussed for Case 1, Step 1 is sound. For Step 2, we defer to the mechanisms for swim and skin which are already known to result in a heap-ordered array that stores a complete tree. Note that both functions will terminate if there is no work to do (thus we neglect the case of the node needing to move or not move). The if-statements evaluates the case of being out of order with parent or children, and follows the formula as defined.

M2

Per Definition 5, insert and delMax for a typical PQ take $O(logn)$ time. This means that the potential best efficiency we can get is $O(logn)$ time or we would be able to implement insert() faster by using our update(). This is a contradiction because we already have taken $O(logn)$ to be the best case for insert(). From the code above, we see that there is one linear loop, meaning $O(n)$ performance, and then either swim or sink runs for an additive $O(logn)$ factor. This means the algorithm is $O(n)$. We now need to show there is no $O(logn)$ performing algorithm (above it was implied a lower constant time solution is not possible). We propose that the linear time loop to find the target element is a bottleneck that cannot be avoided.

Initially we have an array to search for the index of the target value. Given that we are looking at one element, we have two children to consider. Both of these will be smaller than the parent, but have no relationship otherwise. There is no relationship between a value and the priority associated with it. In order to explore only a *log* number of nodes, we have to follow a single root to leaf path. Any branching factor has the potential to visit a linear number of nodes. If the value isn't at the root or its children, then we need to move downward. Moving downward means selecting a child, but since there is no way to tell which is likely to lead to the target , the choice must be arbitrary, and we probably need to backtrack to explore a different path. This means we cannot have a log time solution, and linear time is required. ■

Alternative Design

An alternative proposed solution[2] can be seen in Algorithm 3. Although it can be shown to fulfill M1, trying to support it's optimality for M2 results in an incomplete argument indicating that it is not correct.

Algorithm 3 Pseudocode for alternative update algorithm. (Code derived from Sedgewick.)

```
Integer N       //number of entries in PQ
Key[] keys      //contains N elements
Value[] values  //contains N elements

void update(Value val, Key new_pri):
    //Input: the value of the node is changing, and the new priority.
    //Output: Updated contents in keys and values.
    Integer idx = NULL

    //Step 1: find target key/value
    for i = 0 to N-1:
      if values[i] = val:
        idx = i

    if idx = NULL:
      return

    //Step 2: adjust position of target key/value.
   for int k = N/2 to 1 step -1
     sink(k);
```

[2]This was inspired by the priority queue implementation in Python's default library.