15-210 Assignment 10

Karan Sikka
ksikka@cmu.edu
Section C
December 8, 2012

1: Task 2.1

Let W(T) be vertex weight of the root of T. Let Cldn(T) be the children of root of T. Let Grndcldn(T) be the grandchildren of the root of T. Let MIS(T) be the maximum weight sum for tree T.

2: Task 2.2

```
@memoized
MIS(T) =
  if length Cldn(T) == 0:
    return 1 // must be leaf; independent.
  else:
   // root of T may not be in the answer
    val children = Cldn(T)
    val child_results = map MIS children
    val case1 = max(child_results)

   // root of T may be in the answer
    val grandchildn = Grndcldn(T)
    val grncld_results = map MIS grandchildn
    val case2 = W(T) + max(grncld_results)

    return max(case1,case2)
```

3: Task 2.3

There are n distinct vertices in the tree, and each one is a tree. Then there are at most n subproblems.

The computation of MIS(T) with depth d, depends on the computation of MIS on all of it's children and all of it's subchildren. The computation ends when the depth of the subproblem is 0, or when you reach a leaf. Therefore, the longest chain of dependencies will be as long as the longest path in tree, which has depth d. Then d is an upper bound on the longest path in the DAG.

In the best case, you must compute the entire DAG because there may be some very high weighted leafs. Therefore the work of the algorithm is in $\Theta(n)$.

Since you have to compute the whole DAG, you must, in the best case, compute nodes in the longest path. Then the span is $\Theta(d)$.

4: Task 2.4

Change the subproblem to include this goal.

Then change the base case to just return the leaf.

Then make a new "max" function which takes in two sets of vertices, and returns the set with the larger vertex sum. This can be done in constant time like the normal max by keeping a "sum" counter which increases by the weight when adding a new vertex.

5: Task 3.1

Reason 1: If you move an element up to the hole, you create another hole where the element was. This means that some element after the new hole may become unfindable.

Reason 2: If you move up a (key,value) which was saved in position (h(key)) without having to probe linearly, then moving it up will make it unfindable...
(unless the entire sequence only has that one hole. Then the probing would wrap around and the moved value would be found.)

6: Task 3.2

Say you're filling a hole at position i. Then a (key,value) at position j may fill the hole, if h(key) ¡ i or k(key) ¿ j.

7: Task 3.3

Stop if the next value in the sequence (mod the length of the sequence) is a hole (accounting for the possibility of a wrap-around.

8: Task 3.4

```
/* Linearly search for a hole (stop searching),
 *  or a key,value which is safe to fill the hole.
 *  Note: since there is a hole at holeIndex, the algorithm
 *  is guaranteed to terminate. */
hashTable fillHole (HT,holeIndex,currPos) =
  // base case: don't move the next one here (stopping condition)
  if (nth HT currPos == HOLE) then HT
  else
    if h(key(nth HT currPos)) < holeIndex
      orelse h(key(nth HT currPos)) > currPos then
      // fill hole
      inject HT <(holeIndex, nth HT currPos), (currPos, HOLE)>
    else
      // try to fill w/ something else
```

```
      fillHole (HT,holeIndex,mod(currPos + 1,length HT))
```

9: Task 4.1