Homework3

Parker Gabel

<2019-03-25 Mon>

1 11.3

1.1 (a)

0	10	0	20	0	2
10	0	3	5	0	0
0	3	0	0	15	0
20	5	0	0	11	10
0	0	15	11	0	3
2	0	0	10	3	0

1.2 (b)

Vertex -> (Vertex #, Weight)

1	(2, 10)	(4, 20)	(6, 2)	
2	(2, 10)	(3, 3)	(4, 5)	
3	(2, 3)	(5, 15)		
4	(1, 20)	(2, 5)	(5, 11)	(6, 10)
5	(3, 15)	(4, 11)	(4, 11)	(6, 3)
6	(1, 2)	(4, 10)	(5,3)	

1.3 (c)

The adjacency matrix has no overhead for pointers so
$\Theta(\text{Matrix}) = 2|V^2| = 2(6^2) = 2(36) = 72$
The list does have that overhead so
$\Theta(\text{List}) = 4|V| + 4|E| = 4(6) + 4|9| = 60$
So the list uses less space

1.4 (d)

$\Theta(\text{Matrix}) = 2|V^2| = 72$
$\Theta(\text{List}) = 4|V| + 2|E|$

2 11.4

Tree represented by an adjaceny list.

1	(6, 2)	(2, 10)	
6	(5, 3)	(1, 2)	
5	(5, 15)	(2, 10)	
2	(1, 10)	(3, 3)	(4, 5)
4	(2, 5)	(5, 11)	
3	(2, 3)		

3 11.14

We produce a DFS tree and check if any of the nodes in the tree point to any of
its ancestors. We assume an appropriate graph and vertex class has been implemented.

```
class IsCyclic {
static private boolean _isCyclic(Vertex node, HashSet<String, Boolean> visited,
HashSet<String, Boolean> stack) {
    if (stack.get(node.name())) {
return true;
    }
    if (visited.get(node.name())) {
return false;
    }
    visited.get(node.name()) = true;
    stack.get(node.name()) = true;
    for (Vertex child : graph.getChildren(node)) {
if (_isCyclic(child, visited, stack)){
    return true;
}
```

```
    }
    stack.get(node.name()) = false;
    return false;
    }

    public static boolean isCyclic(Graph graph) {
// This function returns a HashSet with the name of each vertex as a key and
// The bolean is set to false
HashSet<String, Boolean> visited = newGraphSet(graph);
HashSet<String, Boolean> stack = newGraphSet(graph);
for(Vertex vertex : graph.getVertices()) {
    if (_isCyclic(vertex, visited, stack)) {
return true;
    }
}
return false;
    }
}
```

Notice this has the same complexity as a DFS so it is $\Theta(|V| + |E|)$.

4 11.17

3 -> 2 -> 4 -> 1 -> 6 -> 5
3 -> (2, 3)
2 -> (4, 5)
4 -> (6, 2)

5 11.20

If the graph has two or more edges with the same weight it is possible to produce different trees. Each algorithm has a point at which they must choose some edge with the minimum weight and if two or edges have the same weight this choice is ambiguous. Depending on the implementation of each respective algorithm there may be a different choice of edge. This would result in a different MST. If the graph had all unique weights then the MST would be necessarily unique and the

two algorithms would always produce the same MST.

6 11.22

Yes both work fine with negative weights. In Prim's, the least weight edged connected to a vertex not in the tree is added. If that edge happens to have a negative weight that is not a problem. The algorithm just continues. Likewise, In Kruskal's, the least weight edge that connects two components is added. If that is negative then that isn't a problem.

7 11.23

Dijkstra's algorithm produces a tree that contains the shortest path from a source vertex to every other vertex that is reachable in the graph. If the graph is connected then this is a tree that contains every vertex so it is a spanning tree. It is not necessarily a MST. Dijkstra's algorithm is concerned with minimizing the path between the source vertex and every other vertex. A MST is a tree with the minimum total weight. The minimum weight from a source to another vertex may not result in a minimum weight in the MST. Consider the following graph.
1 -> (2, 5) -> (3, 5) -> (4, 5)
2 -> (1, 5) -> (4, 1)
3 -> (1, 5)
4 -> (1, 5) -> (2, 1)
Let the source vertex be 1. Dijkstra's algorithm would find the shortest path from 1 to 4 is just the edge connecting 1 and 4 but the MST would not have this edge. The MST would have the edges 1 to 3, 1 to 2 and 2 to 4. So Dijkstra's algorithm does not necessarily produce an MST.

8 7.4

It would reduce the number of comparisons made as the binary search on the sorted part would find the position to place the element in at

most $\log_2(n)$ with n equal to the length of the sorted part of the array. It would not change the asymptotic running time of the algorithm however because the swaps still have the same running time so it is still $O(n^2)$.

9 7.6

9.1 Insertion Sort

Insertion Sort is stable because if an element that appears before the element to be inserted is equal to the element to be inserted, the comparison in the algorithm will dictate that it is never be inserted before that element.

9.2 Bubble Sort

Bubble sort is stable because the algorithm progressively compares elements in the array and only swaps them if there is an inversion. So this algorithm will always maintain the relative ordering of elements that are equal.

9.3 Selection Sort

Selection sort is not stable because the algorithm swaps the minimum element in
the unsorted part of the array with the first position in the array and this may
invert the order of two equal elements. The algorithm can be stable if instead of swapping, it inserts the minimum element at the first position in the unsorted array and pushes the following elements back one position. This would
guarantee that the relative ordering of equal elements would remain.

9.4 Shell sort

Shell sort is not stable because the algorithm relies on swapping non-adjacent elements. This may result in two equal elements inverting their position if they
are in separate sub lists. It can't be made stable because the non-adjacent

swapping is the fundamental reason why the algorithm works.

9.5 Merge sort

Merge sort is stable because when merging lists, two elements are swapped only
if the left value is strictly greater than the right value. This would leave the equal values in their original order.

9.6 Quick sort

Quick sort is not stable because the partition algorithm swaps non-adjacent elements and most partition schemes would swap a value that is equal to the pivot with an element that is less than the pivot if those values were in positions that could be swapped by the partition algorithm. The algorithm could
be made stable by adding some logic in the partition algorithm that checks if
the index of the equal element was less than the resulting pivot position. If its index is less then it is placed in the lesser partition otherwise it is place in the greater partition.

9.7 Binsort

Binsort is stable because everytime an element of equal value to an element that appeared earlier in the list occurs it is appended to the bin. So relative ordering of the equal elements is always preserved.

9.8 Radix sort

Radix sort is not stable because it places values into the bin from the bottom of the bin up then replaces into the array from the top down. This does not preserve order. I cannot be made stable.

10 7.9

The worst case for quick sort occurs when a maximum or minimum element is chosen as a pivot. The findPivot method implemented in 7.5 choses the median index as its pivot so any permutation with 0 or 7 at position 3(zero-origin indexing) will cause a worst case to occur.

[0 1 2 7 3 4 5 6]

11 7.19

Notice that for n calls to merge sort, you first call mergesort on one list, then two list, then 4 lists and so on

$1 + 2 + 4 + 8 + \ldots = \sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

When mergsort is called on the final set of lists of size 2^n in this series, insertion sort is run instead. So the amount of insertion sorts run is

$2^n = (n + 1)/2$