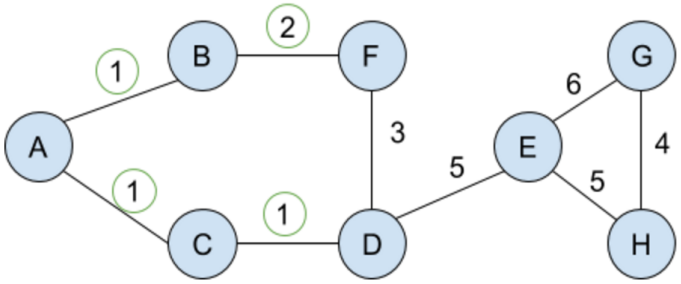
CS61B Spring 2016 Guerrilla Section 5 Worksheet

23 April 2016

Directions: In groups of 4-5, work on the following exercises. Do not proceed to the next exercise until everyone in your group has the answer and *understands why the answer is what it is*. Of course, a topic appearing on this worksheet does not imply that the topic will appear on the midterm, nor does a topic not appearing on this worksheet imply that the topic will not appear on the midterm.

1 MSTs

(1)



Consider the undirected graph above. We are trying to find the minimum spanning tree (MST) of the graph. The edges with their weight labels circled have already been added to our MST.

(a) What is the next edge to be added to our MST if we are using Kruskal's Algorithm?
(G, H). Kruskal?s algorithm adds the lightest edge that does not cause a cycle. The lightest edge is (D, F), but this causes a cycle. The next lightest edge is (G, H). This does not cause a cycle so we add it to our MST.

(b) What is the next edge to be added to our MST if we are using Prim's Algorithm?
Note: part (a) and part (b) are not related. Don't consider the edge that you added in part (a).
(D, E). Prim?s algorithm adds the lightest edge that is connected to the current tree, but does not cause a cycle. The lightest edge connected to the current tree is (D, F), but this causes a cycle. The next lightest edge connected to the current tree is (D, E) which does not cause a cycle so we add it to our MST.

(c) What is the weight of the complete MST?
19. Added edges: (G, H), (D, E), (E, H). The total weight is $1 + 1 + 1 + 2 + 4 + 5 + 5 = 19$. Note that both Prim?s Algorithm and Kruskal?s Algorithm always MST?s of the same total weight (the minimum total weight).

(2) Consider a graph with negative edges.

(a) How would we modify Kruskal?s Algorithm to find a MST on this graph?
No modifications necessary. The negative edges have no effect on Kruskal?s Algorithm.

(b) We now want to find a minimum spanning graph (it no longer needs to be a tree) for this graph. How would we modify Kruskal?s Algorithm to find the minimum spanning graph for this graph?
We could run Kruskal?s Algorithm, and then add all remaining negative edges. This would ensure that we have included all of the lightest edges and all of the negative edges, since the negative edges decrease the weight of the minimum spanning graph.

2 To Cycle Or Not To Cycle

(1) Given an undirected graph G = (V, E) and an edge e = (s, t) in G, create an O(V + E) time algorithm to determine whether G has a cycle containing e. No code needed. Just describe.
Remove e from the graph. Then DFS or BFS starting at s to find t.

(2) (Extra for Experts. Skip this and come back if you have time)
Given a connected, undirected, weighted, graph, describe an algorithm to construct a set with as few edges as possible such that if those edges were removed, there would be no cycles in the remaining graph. Additionally, choose edges such that the sum of the weights of the edges you remove is minimized. This algorithm must be as fast as possible.
Negate all edges.
Form an MST via Kruskals/Prim.
Return the set of all edges not in the MST (undo negation).

3 Start To Finish

You're given an undirected, positively weighted graph G = (V, E), a list of start vertices S, and a list of end vertices T. Describe an efficient algorithm that returns the shortest path, such that the path starts at one vertex from S and and ends at one vertex from T.

Hint: Consider adding dummy nodes to the graph to reduce this problem into something simpler.
Add a node S, connected to all start nodes and a node E connected to all end nodes with weight C (any non-negative constant). Find the shortest path from S to E, remove S and E from the final problem. This can be done with Dijkstra?s algorithm.

4 One Path to Traverse them All, and Topological Sort Them

Given a directed acyclic graph G, write an algorithm that determines if G contains a path that goes through every vertex exactly once. Briefly justify why the algorithm is correct, and state the runtime.

Assume that the graph is implemented with the following API, where nodes are represented by integers.

```
public class Graph {
        // Returns true if this graph has an edge from u to v.
        public boolean hasEdge(int u, int v);

        // Returns a list of integers, in a topologically sorted order for this graph;
        // implemented in the way described in lecture.
        public List<Integer> topologicalOrder();
}

// Please implement your algorithm in this method:
public boolean onePath(Graph g) {
        /* The solution */
        List<Integer> topoSorted = g.topologicalOrder();
        for (int i = 0; i < topoSorted.length - 2; i++) {
                if (!g.hasEdge(topoSorted[i], topoSorted[i+1])
                        return false;
        }
        return true;
        /* End solution */
}
```

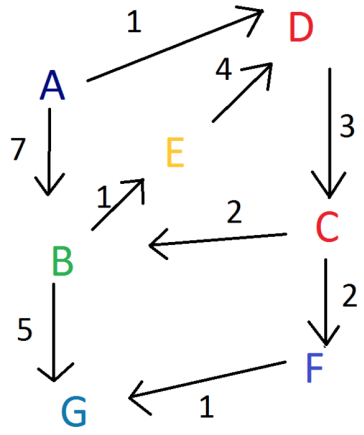
Justification: The main idea/justification is that if a DAG has a path that traverses every vertex, then its topological sorting must be unique, and additionally, every pair of nodes in the topological sorting must have an edge between them.
Runtime (in Θ notation): $\Theta(|V| + |E|)$, the runtime of topological sorting.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

5 The Amazing Race!

Directions: With the United States as your starting point, your goal is to travel to Greece as fast as you can. However, there is a twist such that you can only travel the paths shown on the graph toward Greece. Use A* search to find the optimal path to Greece, breaking ties alphabetically if necessary. Ready, set, go! (Note: This graph was not designed by a geographer.)



Letter in graph	Name of location	Heuristic to End
A	America (USA)	6
B	Brazil	4
C	China	7
D	Dominican Republic	7
E	Egypt	2
F	France	6
G	Greece	0

A: c = 6

A→B: c = 11
A→D: c = 8
A→D→C: c = 11
A→B→E: c = 10
A→B→G: c = 12
A→B→E→D: c = 19
A→D→C→B: c = 10
A→D→C→F: c = 12
A→D→C→B→G: c = 11
ADCBG is the solution based on A* search. This is not the optimal solution; the optimal solution is ADCFG with a cost of 7. A* search gave a suboptimal solution because F has an inadmissible heuristic. (The optimal path from F to G is 1, but the heuristic values given provided a heuristic value of 6 for F).

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

6 Which Sort to Use?

For each of the following scenarios, choose the best sort to use and explain your reasoning.

(a) The list you have to sort was created by taking a sorted list and swapping N pairs of adjacent elements.
Insertion sort, since a list created in such a manner will have at most N inversions. (Recall that insertion sort runs in $\Theta(N+K)$ time, where K is the number of inversions.) Bubble sort is also correct since that would also take $\Theta(N)$ time on such a list.

(b) The list you have to sort is the list of everyone who took the US census and you want to sort based on last name.
MSD radix sort. LSD radix sort would work, but we would have to pad each last name with extra characters so that all names were the same length.

(c) You have to sort a list on a machine where swapping two elements is much more costly than comparing two elements (and you want to do the sort in place).
Selection sort, since in its most common implementation, selection sort performs N swaps in the worst case, whereas all other common sorts perform $\Omega(N \log(N))$swaps in at least some cases.

(d) Your list is so large that not all of the data will fit into RAM at once. As is, at any given time most of the list must be stored in external memory (on disk), where accessing it is extremely slow.
Merge sort is ideal here, since its divide-and-conquer strategy works well with the restriction on only being able to hold a partition of the list in RAM at any given time. Sorted runs of the list can be merged in RAM and flushed to disk one block at a time, minimizing disk reads and writes. Google ?external sorting? for more details.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

7 Empirical Analysis

Andrew has performed timing tests on several sorting algorithms: selection sort, insertion sort, merge sort, and tree sort (repeated insertions into a binary search tree with no attempt to balance, followed by a traversal of the tree). He timed each sorting algorithm on several datasets of 2000 values. Unfortunately, Andrew forgot to label each experiment with its sort! Help Andrew by figuring out which times go with which sorting method.

Time to sort 2000 random values	Time to sort 2000 values already in increasing order	Time to sort 2000 values already in decreasing order	Sorting method (Selection, Insertion, Merge, or Tree)
1098	29	1685	Insertion
183	1624	1570	Tree
191	207	195	Merge
1698	1776	1734	Selection

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

8 What's that Sort!?

Which sorting algorithms do the following illustrate? Your options are merge sort, insertion sort, selection sort, heap sort, quick sort. Algorithms illustrated may not conform exactly to those presented in discussion and in lecture. Please note that each of these are snapshots as the algorithm runs, not all iterations of its running.

(a) 5103 9914 0608 3715 6035 2261 9797 7188 1163 4411
0608 1163 5103 3715 6035 2261 9797 7188 9914 4411
0608 1163 2261 3715 6035 5103 9797 7188 9914 4411

Selection Sort

(b) 5103 9797 0608 3715 6035 2261 9914 7188 1163 4411
0608 3715 2261 1163 4411 5103 9797 6035 9914 7188
0608 3715 2261 1163 4411 5103 6035 7188 9797 9914

Quicksort

(c) dze ccf hwy pjk bkw xce aux qtr
ccf dze hwy pjk bkw xce aux qtr
ccf dze hwy pjk aux bkw qtr xce
aux bkw ccf dze hwy pjk qtr xce

Merge Sort

(d) dze ccf bkw hwy pjk xce aux qtr xpa atm
dze ccf bkw hwy pjk xce aux qtr atm xpa
dze ccf bkw hwy pjk xce aux atm qtr xpa
dze ccf bkw hwy pjk xce atm aux qtr xpa
dze ccf bkw hwy pjk atm aux qtr xce xpa
dze ccf bkw hwy atm aux pjk qtr xce xpa
dze ccf bkw atm aux hwy pjk qtr xce xpa
dze ccf atm aux bkw hwy pjk qtr xce xpa
dze atm aux bkw ccf hwy pjk qtr xce xpa
atm aux bkw ccf dze hwy pjk qtr xce xpa

Insertion Sort (from the right)

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!

9 Trieing to Find Partial Matches

Given a list of N input words all of length at most k and M query words, we would like to find, for each query word, the number of input words that match the first k/2 letters of the query word. Describe an algorithm that accomplishes this and give its running time as a function of N,M, and k.

Solution 1: Use a trie where every node of the trie tracks how many of its descendants are complete words. Insert all N input words into such a trie. This takes $O(Nk)$ time (assuming a constant alphabet size). Then for each query word, find the node in the trie corresponding to the word's first k/2 letters and output the number stored in that node (and output 0 if there is no such node). This takes $O(k)$ time for each query word. So the entire algorithm takes $O((N+M)k)$ time (note that in the best case, all of the query words begin with some letter that is not in the trie at all, in which case this solution will run in $O(Nk+M)$ time).

Solution 2: Use a HashMap where the keys are strings of length k/2 and the values are integers representing how many of the input words begin with those k/2 letters. For each input word x, look up the length k/2 prefix of x in the HashMap. If it is present, increment the corresponding value. If it is not present, add it as a key with corresponding value of 1. Then for each query word, check if its length k/2 prefix is present in the HashMap. If so, output the corresponding value. If not, output 0. In the worst case, inserting the input words into the HashMap takes $\Theta(N2k)$ (because in the worst case all input words hash to the same bucket and comparing the inserted word to all previously inserted words takes up to O(Nk) time because we have to compare the first k/2 letters of each word) and similarly, looking up all the query words takes $\Theta(MNk)$ time. So in the worst case this solution will take $\Theta((N+M)Nk)$ time. If we assume however that the keys are distributed uniformly among the buckets then this solution is also $\Theta((N+M)k)$ time.

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section!