CS1114: Study Guide 1

In the first part of this course, we have covered the topics in this document. Please refer to the class slides for more details.

1 Finding the center of things: Bounding boxes, centroids, trimmed means, and medians

Given a collection of n 2D data points:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$$

we discussed several ways to find the center of the data:

Bounding box. One simple way is to compute the bounding box of the data. The bounding box is the smallest axis-aligned rectangle that completely encloses the data, and is defined by a top row (`top`), a bottom row (`bottom`), a left column (`left`), and a right column (`right`). To find the center of the data, we can find the midpoint of the bounding box:

$$\texttt{bounding box midpoint} = \left(\frac{\texttt{left} + \texttt{right}}{2}, \frac{\texttt{top} + \texttt{bottom}}{2} \right).$$

This definition of center is very easy to compute. Unfortunately, it is very easy to skew with a single bad outlier.

Centroid. A more robust measure of the center of the data is the *centroid*. The centroid is found by taking the average x-coordinate and the average y-coordinate:

$$\texttt{centroid} = \left(\frac{1}{n} \sum_{i=1}^{n} x_i, \frac{1}{n} \sum_{i=1}^{n} y_i \right).$$

The centroid is less sensitive to outliers than the bounding box midpoint. Unfortunately, one bad outlier can still skew the results (remember the example in class of the kindergarten class containing Arnold Schwarzenegger).

Trimmed mean. This led us to the definition of the trimmed mean. The trimmed mean is computed by removing, say, the top 5% of the data and the bottom 5% of the data, then computing the mean of the result. This measure is robust to outliers (at least when less than 10% of the data are outliers). To compute the trimmed mean of 2D data, we would compute the trimmed mean of the x-coordinates, and the trimmed mean of the y-coordinates.

Median. Another robust measure of the center is the *median*. For a set of numbers $x_1, x_2, x_3, \ldots, x_n$, the median is the number smaller than half of the other numbers, and larger than half of the other numbers (if the list were sorted, it would be the middle element of the list). It is not clear how to define the median for 2D data, but in class we have been using the *median vector*, whose x-ccordinate is the median of the x-coordinates, and whose y-coordinate is the median of the y-coordinates.

Next, we will look at several algorithms for finding the median—or, in fact, the element of any *rank* in an array. Recall that we defined an element to be rank k if it is the kth-largest element in the array. Thus, the median in an array of n elements has rank $\frac{n}{2}$.

2 Algorithms and running time

An algorithm is a set of steps for solving a problem (such as finding the median of an array). There may be many different algorithms for solving a particular problem—all correct algorithms will solve the problem, but different algorithms might vary in terms of ease of implementation, amount of memory used, and running time. In this class, we have been particularly interested in the last of these: running time.

2.1 Big-O notation.

To characterize the running time of a function, we have been using big-O notation, which expresses running time independent of what kind of hardware we run the algorithm on. Using this definition, the running time of an algorithm is, roughly speaking, proportional to the amount of work we have to do for an input of size n (as n gets very large).[1] This leads us to several classes of algorithms:

1. $O(1)$ (constant time). Algorithms which take a constant amount of time, no matter how big the input is (again, as n gets very large).

   Example: computing the third element in an array.

2. $O(n)$ (linear). Algorithms which do work proportional to the size of the input.

   Example: computing the largest element in an array.

3. $O(n \log n)$ (loglinear, or quasi-linear). Algorithms which do work proportional to the size of the input times the log of the size of the input.

   Example: quicksort (expected run time).

4. $O(n^2)$ (quadratic). Algorithms which do work in proportion to the size of the input squared.

   Example: computing the median using repeated find biggest.

In class, we've looked at several algorithms for finding the median in an array:

[1] Later CS courses cover the exact definition of big-O.

2.2 Repeated find biggest

This algorithm finds the biggest element in the array, removes it, then finds the second biggest, removes it, and repeats, until it has removed the biggest $\frac{n}{2}$ elements. The last element removed is the median.

Running time: $O(n^2)$. Explanation: every time we remove the biggest, we scan the entire array. The first time, we scan n elements, the second time, $n-1$ elements, and so on, until the last time we scan $\frac{n}{2}$ elements. The total amount of work is thus:

$$n + (n-1) + (n-2) + \ldots + \frac{n}{2}$$

There are $\frac{n}{2}$ terms in this sum, and each is $\geq \frac{n}{2}$. Thus, we do at least $\left(\frac{n}{2}\right)\left(\frac{n}{2}\right) = \frac{n^2}{4}$ amount of work—this is proportional to n^2, so repeated find biggest is $O(n^2)$. (Note that it is possible to compute the sum above exactly—the exact sum is still proportional to n^2).

2.3 Quicksort

The next approach we covered was to sort the array first—it is then very simple to find the median of a sorted array. So how do we sort an array? One approach is to use repeated find biggest for the entire array, storing the biggest elements in another array in order as we find them. However, this is not very helpful for finding the median, because it is doing strictly *more* work than the repeated find biggest approach above. Thus, we looked at another sorting algorithm called *quicksort*. Quicksort has three steps:

1. Choose an element to be the pivot (this could just be the first element, `A(1)`, but a better choice is a random array element).

2. Partition the array into three subarrays `A1`, `A1`, and `A3`: one for elements smaller than the pivot (`A1`), elements equal to the pivot (`A2`), and elements larger than the pivot (`A3`).

3. Recurse: quicksort arrays `A1` and `A3` separately, then join them together (sandwiching `A2`).

Running time: $O(n \log n)$ (expected), $O(n^2)$ (worst case). Explanation: The worst case running time of quicksort is $O(n^2)$: if we choose the first element as the pivot each time, and the array is already sorted, then we break the array into one empty partition (`A1`) and one large partition with only one element removed (`A3`) each time we partition, resulting in having to partition $n-1$ times (scanning all of the remaining elements in the large partition each time).

However, if we get lucky and partition the array in half each time, we divide the array in half $\log_2(n)$ times, and each time we scan approximately n elements in total. This leads to a running time of $O(n \log n)$ in the best case.[2] For random inputs, we get "lucky enough," and quicksort has an expected running time of $O(n \log n)$.

[2] A fact about logarithms is that we can convert between bases using the formula $\log_a n = \log_b n / \log_b a$—thus, all logarithmic functions $\log_a$ are proportional to each other, so it doesn't matter what base logarithm we use inside the big-O.

2.4 Quickselect

However, for finding the median, it turns out that we can do better. The intuition is that we don't care about sorting the entire array—we just care about what the median element is. Thus, we can skip some of the work in quicksort; the resulting algorithm is called *quickselect*. Quickselect can be used to find any element of rank k, and is very similar to quicksort—in fact, the first two steps are identical to quicksort:

1. Choose an element to be the pivot (this could just be the first element, `A(1)`).
2. Partition the array into three subarrays: one for elements smaller than the pivot (`A1`), elements equal to the pivot (`A2`), and elements larger than the pivot (`A3`).
3. If `k <= length(A3)`, then recursively find the k^{th} largest element in `A3`
4. If `k > length(A2) + length(A3)`, then let `j = k - (length(A2) + length(A3))`, and recursively find the j^{th} largest element in `A1`
5. Otherwise, return `x`.

Unlike quicksort, quickselect "throws away" one of the partitions `A1` or `A3` each time, because it knows which partition the kth-largest element must belong to (quicksort, in comparison, recurses on both partitions).

Running time: $O(n)$ (expected), $O(n^2)$ (worst case). Explanation: The same worst case of quicksort applies to quickselect. On the other hand, we can get lucky in several ways. The first is if `length(A3)` is exactly $k-1$—we then know that the pivot is the kth-largest element, and can immediately return it. The other way we could get lucky is by partitioning the array into two equal-sized parts each time (as with quicksort). Again, if we get so lucky, we will split in half $\log_2 n$ times, but each time we split we will do *half* as much work as the previous time, because we throw away half of the array. Thus, the amount of work we do is approximately:

$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \ldots + 1$$

This sum is approximately $2n$. Thus, we do work in proportion to n, so in this case quickselect is $O(n)$. Again, for random inputs we get "lucky enough," and quickselect has an expected running time of $O(n)$.

3 Graphs

The above algorithms gave us a fast way of finding the median. However, we can do better in detecting the lightstick by first finding the largest *blob* of pixels (this is presumably the lightstick), and finding the median of only that blob. Defining what a blob is led us to the concept of a *graph*.

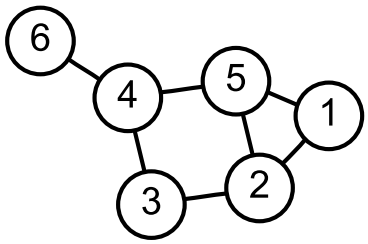

Figure 1: A graph with six nodes and seven edges.

3.1 Definition

A graph consists of a set of objects (called vertices) and links between the objects (called edges). More formally, a graph G consists of a vertex set $V = \{v_1, v_2, \ldots, v_n\}$ and an edge set E, where E contains pairs of vertices (v_i, v_j). An example of a graph is shown in Figure 1. Graphs can represent many different things, including airports and direct connections, people and friendship relationships, Internet hosts and wired connections, and so on.

3.2 Paths and cycles

When dealing with graphs, we often want to talk about sequences of vertices linked by edges. A *path* is just such a sequence, i.e., a sequence of vertices $v_{i_1}, v_{i_2}, \ldots, v_{i_k}$, where each consecutive pair of vertices is linked by an edge. For instance, $1, 5, 4, 6$ is a path in the graph in Figure 1. A path that starts and ends at the same vertex is called a *cycle*. An example of a cycle in the graph above is $2, 3, 4, 5, 2$.

3.3 Types of graphs

There are many special types of graphs that are useful to know about. These include:

Connected graphs. Two vertices are *connected* in a graph if there is a path between them. A graph itself is said to be connected if all pairs of vertices are connected (for instance, the graph in Figure 1 is connected).

Trees. A *tree* is a connected graph with no cycles. An example of a tree is shown in Figure 2.

Planar graphs. A *planar graph* is a graph that can be drawn with no edge crossings (remember that when drawing a graph, it is acceptable to move vertices around, and to draw curved edges, as long as the connectivity doesn't change). The graphs in Figures 1 and 2 are both planar. However, the graph in Figure 3 is not planar (though this is difficult to prove).

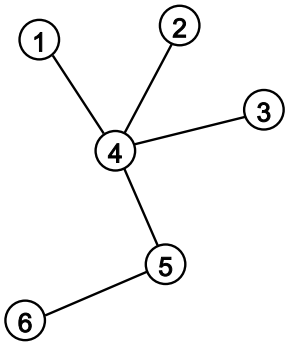

Figure 2: An example of a tree.

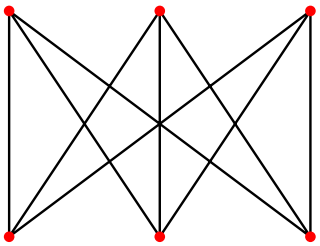

Figure 3: An example of a non-planar graph.

3.4 Graph problems

Many problems in the real-world can be posed as graph problems. Here, we review some common graph problems.

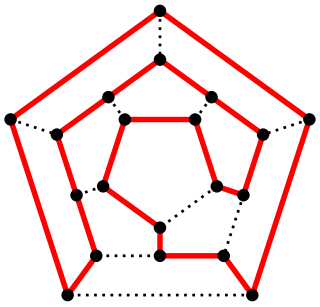

Figure 4: A graph with a Hamiltonian cycle (shown in red).

Hamiltonian cycle A graph has a Hamiltonian cycle if it contains a cycle that visits each vertex exactly once (except for the first vertex, which will also be the last vertex in the cycle; i.e., this vertex is visited twice). An example is shown in Figure 4.

Eulerian cycle A graph has a Eulerian cycle if it contains a cycle that visits each *edge* exactly once (but can visit each vertex as many times as needed).

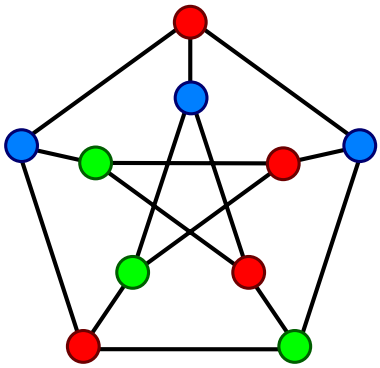

Figure 5: A valid coloring of a graph.

Graph coloring A *graph coloring* is the assignment of colors to each vertex such that no two vertices connected by edges are the same color. For instance, the graph in Figure 5 is shown with a valid coloring using three colors. Colorings are useful for solving problems where we need a assign one of a list of possible labels to each vertex, but where there are conflicts preventing certain pairs of vertices from having the same label (these conflicts can be represented by edges. An example was given in class of using graph colorings to solve the problem of assigning frequencies to radio stations, where two radio stations were connected if they had an overlapping broadcast region (and thus had to be assigned different frequencies to avoid interference). (See the slides for Lecture 7 for more details on this problem.)

A useful fact is that planar graphs can always be colored with four colors. Other graphs may require many more than four colors for a valid coloring.

4 Connected components

Even if a graph is not connected, it will consist of parts that *are* connected. Each part is called a connected component. To find a connected component, we need a way to traverse a graph from some starting point, marking down all of the vertices we visit. This leads to the problem of *graph traversal*.

4.1 Graph traversal

The basic strategy in graph traversal is to start with a vertex, visit its neighbors, then visit its neighbors' neighbors, and so on, until we are done. To make this into an algorithm, we need a way to keep track of vertices which we still need to visit (a "todo list"). Two ways of implementing such a todo list are *stacks* and *queues*.

4.2 Stacks (DFS) and queues (BFS)

A stack is a way of storing a todo list where, when we need to take something from the list, we always take the last thing that was added (as with a stack of cafeteria trays). Thus, a stack is a LIFO (last in, first out) data structure. A queue is a todo list where we always take the *first* thing added off the list (as with a queue of people in line for a movie). A queue is thus a FIFO (first in, first out) data structure.

When we implement graph traversal using a stack, the resulting algorithm is called *depth-first search* (DFS). When we use a queue, the resulting algorithm is called *breadth-first search* (BFS). Please see the slides for more details on these algorithms.

Figure 6: Conceptual diagram of a singly-linked list with elements 42, 18, and 10.

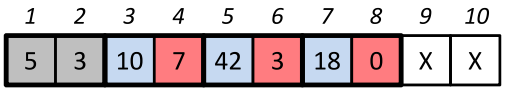

Figure 7: Possible layout of a singly-linked list in memory.

5 Linked lists

In order to implement a todo list, a useful data structure is a *linked list*. A linked list is an alternative to an array, and consists of cells, where each cell contains a *value* (as with an array), and a *pointer* to the next cell (the last cell contains a pointer to 0, called the *null pointer*). In addition to cells, a linked list contains a header with two pieces of information: the location of the first cell, and the size of the list. An example of a (conceptual) linked list is shown in Figure 6, and a possible representation in memory is shown in Figure 7.

Figure 8: Conceptual diagram of a doubly-linked list with elements 42, 18, and 10.

Linked lists allow for efficient insertion and deletion from the start of a list, unlike with an array, where we need to shift elements around to insert or delete from the front. Unfortunately, to efficiently implement queues, we also need to be able to quickly insert or delete from the *end* of a linked list. To enable efficient operations at the end, we described in class a more complex *doubly-linked list* (as opposed to the singly-linked lists described above). Each cell in a doubly-linked list also has a pointer to the previous element, and the header of the

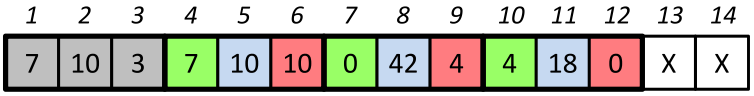

Figure 9: Possible layout of a doubly-linked list in memory.

list contains a pointer to the last element. An example of a doubly-linked list is shown in Figure 8, and a possible layout in memory is shown in Figure 9.

Doubly-linked lists allow for efficient ($O(1)$) insertion and deletion from the start and end of a list, unlike arrays, where these operations are $O(n)$ in general (where n is the length of the array). On the other hand, finding a given element of a linked list may take longer than with an array. To access the middle element of a linked list, we must traverse the list from the beginning (or end), until we reach the middle.