Final Written HW

CIS 121—Fall 2017

Due: Wednesday, December 6, 2017

This is a closed book exam. No computers or internet-connected devices are allowed during the exam. You are permitted to use one 8.5”x11” page with handwritten notes (these can be on both sides of the paper). Please write your name on your page of notes and turn it in at the end of the exam. If you need scratch paper, please get some from the front of the classroom. There are 10 questions on this exam, plus 4 extra credit questions, for a total of 14 questions.

Your name:

Your PennKey (i.e. ccb):
1. **CIS 121 made you a better programmer**
   45 points

Analyze each of the snippets of Java code below. For each of the snippets, please answer the 3 questions about it:

1. What is the order of the growth in terms of Big O (with respect to the length of the list or the Strings) for the code as it is formulated? In addition to giving the Big O, please provide a short justification saying how you arrived at your answer (1 or 2 sentences).

2. How could it be improved using a data structure or algorithm from class? Say which data structure or algorithm you would use to improve it. Also give a short description of how you would use it to solve the problem. Just give your answer in plain English; do not write Java code or pseudo code.

3. Say how your change would improve the runtime. In addition to giving the Big O, please provide a short justification saying how you arrived at that analysis (1 or 2 sentences).

**Java code snippet 1:** The goal of this is to check whether an item is in a list. Here, the list of items is already sorted when passed into the function. In your analysis say whether `assert(isSorted(items));` should count towards the run time or not. If so, how much time does it take?

```java
public <V extends Comparable<V>> boolean containsItem(V item, V[] items) {
    assert(isSorted(items));
    int N = items.length;
    for (int i = 0; i < N; i++) {
        if (items[i].equals(item)) {
            return true;
        }
    }
    return false;
}
```

Your answers to the 3 questions:

1. 

2. 

3. 
Java code snippet 2: The goal of this is to reverse a string, so that “final exam” would become “maxe lanif”.

```java
public String reverseString(String str) {
    String rev = "";
    for (int i = str.length() - 1; i >= 0; i--) {
        rev += str.charAt(i);
    }
    return rev;
}
```

Your answers to the 3 questions:
1. 
2. 
3. 

Java code snippet 3: The goal of this is take a list of all the children in Philadelphia and print out the ones that are on Santa’s nice list.

```java
public void intersectLists(Kid[] phillyKids, Kid[] santasNiceList) {
    for (int i = 0; i < phillyKids.length; i++) {
        Kid phillyKid = phillyKids[i];
        for (int j = 0; j < santasNiceList.length; j++) {
            Kid goodKid = santasNiceList[j];
            if (phillyKid.equals(goodKid)) {
                System.out.println(phillyKid);
            }
        }
    }
}
```

Your answers to the 3 questions:
1. 
2. 
3. 
Java code snippet 4: Return the top k scores for on the final exam.

```java
public class HighestFirst implements Comparator<Exam>{
    // details omitted
}

public Exam[] getTopScores(Exam[] exams, int k) {
    java.util.Arrays.sort(exams, new HighestFirst());
    Exam[] topScoringExams = new Exam[k];
    for(int i = 0; i < k; i++) {
        topScoringExams[i] = exams[i];
    }
    return topScoringExams;
}
```

Your answers to the 3 questions:

1.

2.

3.
Java code snippet 5: Auto-complete. Google shows the auto-complete of the most common queries after users have typed in a few words or letters that are a prefix match of a common query. The `startsWith` is a built-in method in Java’s String class. You should state your assumptions about its running time.

```java
public String[] autocomplete(String prefix, String[] commonQueries, int maxMatches) {
    String[] matches = new String[maxMatches];
    int counter = 0;
    int N = commonQueries.length;
    int k = prefix.length;
    for (int i = 0; i < N; i++) {
        String query = commonQueries[i];
        if (query.startsWith(prefix)) {
            matches[counter++] = query;
            if (counter == maxMatches) {
                return matches;
            }
        }
    }
    return matches;
}
```

Your answers to the 3 questions:

1.

2.

3.
Solution

Java code snippet 1:
1. This code runs in linear time. It is $O(N)$ since it potentially has to check each item on the list if there is no match.
2. Since the list of items is already sorted, this function can be improved by using binary search instead of iterating over everything in the list.
3. Using binary search will reduce the runtime to logarithmic. An answer can be found in $O(\log N)$.

Java code snippet 2:
1. This code appears to run in linear time with respect to the length of the input string ($N$), but it’s actually worse than that, since Java’s concatenation operation for Strings creates a new string each time. The runtime is quadratic $O(N^2)$.
2. The runtime can be fixed by changing `rev` to be a `StringBuilder` and using its `append` method instead of the `+=` String operator.
3. Using `StringBuilder` reduces the runtime to be linear.

Java code snippet 3:
1. This code runs in $O(N \times M)$ where $N$ is the length of Santa’s nice list and $M$ is the number of children in Philadelphia.
2. The runtime can be improved by making a HashSet out one of the two lists. Then iterate of the other list, and check whether that child is in the has set.
3. The runtime of the has table method is $O(N + M)$ to construct the has table and to iterate through the other list.

Java code snippet 4:
1. Runtime would be linearithmic with respect to the number of exams ($N$), since the implementation of java.util.Arrays.sort uses a stable, adaptive, iterative mergesort. It would require $O(N \log(N))$ to do the sort.
2. Since we are only interested in the top $k$ exam scores, this code can be improved by using a MaxHeap instead of fully sorting the list. Create a MaxHeap from the list of exams, and then call `delMax` $k$ times.
3. The construction of the Heap is $O(N)$, and doing $k$ calls to `delMax` takes $O(k \times \log N)$.

Java code snippet 5:
1. It’s also OK to analyze the method as $O(N \times k)$ or $O(N)$, where $k$ is the length of the prefix and $N$ is the number of common queries being tested. $O(N \times k)$ assumes that Java’s `startsWith` method is $O(k)$. For longer prefixes, we can use some domain knowledge to assume that most common queries will not start with that prefix and assume that the `startsWith` returns false after checking one letter, hence $O(N)$.
2. Instead of representing the common queries as a list, they should be represented as a trie which efficiently support prefix-based searching.
3. The runtime of prefix match in a R-way trie is $O(k)$ to find the prefix, plus the runtime to find all keys that are rooted at that prefix by doing an in-order traversal of that sub-trie.
2 Asymptotic analysis of algorithms
16 points

For each of the following statements, say whether it is true or false. Give a one sentence explanation of why that it so.

1. To establish a worst case analysis for an algorithm, we should analyze its performance on a single, randomly drawn input to get a representative example of the possible inputs.

2. Implementing a specific algorithm is a common way of establishing a lower bounds for a problem.

3. If a problem that is $\Omega(N^3)$ has an algorithm that is $O(N^3)$, then the algorithm is optimal.

4. The $O$, $\Omega$ and $\Theta$ notations all ignore lower order terms but keeps constants.

5. If an algorithm is $O(N^2)$ that means its average running time is quadratic with respect to the input.

6. Given an algorithm implementation, one can always derive a big-Theta bound for its runtime.

7. Amortized runtime analysis is the same as average runtime analysis

8. A $O(N^2)$ algorithm could run faster/perform fewer operations than a $O(N)$ algorithm for same sized inputs.

Solution

1. False. A worst case analysis provides an upper bound on the cost of an algorithm (in terms of running time or memory). We should examining its performance on the input that will cause it to run the longest.

2. False. A specific algorithm is a common way of establishing an upper bound for a problem, but for a lower bound we need a proof that no algorithm can do better.

3. True. Big O indicates the upper bound on a problem, and Big Omega indicates the lower bound. We describe a algorithm as optimal when the bound equals upper bound to within a constant factor.

4. False. All of these algorithms ignore both lower order terms and the constants. Tilde notation keeps constants and drops lower order terms.

5. False. Its worst case running time is quadratic, not its average running time.

6. False, e.g. differing $O$ and Omega bounds

7. False, amortized runtime looks at many operations in aggregate, average/best/worst case runtime looks at a single operation

8. True. all the big O notations only show asymptotic values when N becomes large, so when N is small it might run faster.
3  Huffman coding  
10 points

Draw a trie representing the Huffman code for the letters with the following frequencies, and give the encoding for each letter.

<table>
<thead>
<tr>
<th>Letter</th>
<th>Freq</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Solution

CCB

4  Compression  
24 points

Give a short answer to each of these questions, along with a 1 sentence explanation of your answer.

1. Does a balanced binary tree correspond to an optimal prefix-free code?
   No. We rely on an unbalanced tree to get shorter for more frequent characters and longer codes for less frequent ones.

2. How do we measure the effectiveness of a compression algorithm?
   Huffman provides an optimal prefix free code.

3. What key part of Huffman allows it to produce an encoding that is shorter than the input?
   The compression ratio measures of the effectiveness of a compression algorithm. It is calculated as the size of the compressed file over the size of the uncompressed file.

4. What key part of LZW allows it to produce an encoding that is shorter than the input?
   Letter frequency. Huffman assigns the shortest code to the most frequent letter. More frequent letters get shorter codes than less frequent letters.

5. Do all compression methods require that we have to transmit a table containing its codewords like we do with Huffman coding?
   LZW assigns a fixed length code to multiple letters at once. Substrings that repeat many times in a file can be represented with a single codeword. The longer the substring the greater the compression.

Solution

1. No. We rely on an unbalanced tree to get shorter for more frequent characters and longer codes for less frequent ones.

2. None. Huffman provides an optimal prefix free code.

3. The compression ratio measures of the effectiveness of a compression algorithm. It is calculated as the size of the compressed file over the size of the uncompressed file.

4. Letter frequency. Huffman assigns the shortest code to the most frequent letter. More frequent letters get shorter codes than less frequent letters.

5. LZW assigns a fixed length code to multiple letters at once. Substrings that repeat many times in a file can be represented with a single codeword. The longer the substring the greater the compression.
6. No. LZW doesn’t do so. It constructs its codeword table adaptively as it reads in the text. The method is the same when we are compressing and when we are decompressing so the same sequences get assigned the same codewords.

7. Variable length codewords can cause ambiguity. For example Morris code . . . - - - - - - could be SOS or V7 or several other patterns.

8. Fixes include: Using fixed length codes instead of variable length codes. Ensuring that no codeword is a prefix of another. Appending a special stop character to the end of each codeword.

5 Dijkstra’s algorithm for shortest paths
22 points

[FIGURE OMITTED]

Apply Dijkstra’s algorithm to the graph above, starting at vertex 0.

1. Describe how Dijkstra’s algorithm works.

2. What is edge relaxation?

3. Fill out a table that shows the distance to each vertex, and the edge that was taken to get there.

4. Draw the shortest paths tree and label the distance from 0 to each vertex.

5. Which vertex had the most updates to its distance through edge relaxation?

6. When do we know we’ve found the shortest path to vertex x?

Solution

1. Dijkstra’s algorithm works by
   - Considering vertices in increasing order of distance from s (non-tree vertex with the lowest distTo[] value).
   - Adding that vertex to tree and relaxing all edges adjacent from that vertex.

2. We decide whether to relax an edge $e = v \rightarrow w$, by checking if $e$ gives a shorter path to $w$. We compare the length of the shortest know path from $s$ to $w$ (distTo[w]) to the length of the shortest know path from $s$ to $v$ (distTo[v]) plus the cost of $e$. If the cost is lower then we update distTo[w] as distTo[v] plus the cost of $e$. We also record that $e$ is the edge to $w$.

3. Here is the table:

<table>
<thead>
<tr>
<th>Vertex</th>
<th>distance to</th>
<th>edge to</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>2.2</td>
<td>0→1</td>
</tr>
<tr>
<td>2</td>
<td>3.1</td>
<td>1→2</td>
</tr>
<tr>
<td>3</td>
<td>4.2</td>
<td>4→3</td>
</tr>
<tr>
<td>4</td>
<td>3.7</td>
<td>2→4</td>
</tr>
<tr>
<td>5</td>
<td>16.4</td>
<td>6→5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>0→6</td>
</tr>
<tr>
<td>7</td>
<td>8.2</td>
<td>8→7</td>
</tr>
<tr>
<td>8</td>
<td>5.2</td>
<td>3→8</td>
</tr>
</tbody>
</table>
5. The distance to vertex 3 is updated 4 times in total.

6. After that vertex has been processed, there will be no shorter paths to it.

6 Make money with an app  
20 points

An entrepreneurial team of Penn undergrads builds an app using Dijkstra’s algorithm that lets drivers find the shortest distance to their destination.

1. How is the system of roads represented in the graph? What are the vertices and edges? What type of graph is it?

The US congress has decided that in order to reduce the national deficit, all roads are now toll roads. Users of the app demand an update that will let them find the lowest cost route to their destination in terms of dollars instead of miles.

2. What do the Penn students need to do to update their graph?

President Trump decides to name the best, classiest road in each American city after himself. He decides that instead of charging drivers a toll, the government will pay drivers whenever they drive on any street named Trump Avenue.

3. Assuming that the graph automatically updates each time the tolls change, what happens to the directions provided by app? Will drivers still find the lowest cost ($$) route to their destination?

4. How could the Penn students determine if the app could be fixed, or if it is broken beyond all repair? If it is fixable, what is the solution?

5. If the app can no longer provide driving directions, how could the Penn students still use it to make money for themselves? What would the implications be for the US economy?

Solution

1. The system of roads can represented in the graph by making each intersection (including cul-de-sac) a vertex and each each span of road between two intersections an edge. The cost of each edge is its distance. The graph is a weighted directed graph.
2. The weights on edges would need to be updated to be the cost of driving on that span of road, instead of its distance.

3. By paying drivers to go on certain roads rather than charging them, Trump introduces negative weights. Dijkstra’s algorithm does not work with negative weights, so the app fails to give valid directions.

4. The students need to determine if the graph contains negative cycles or not. If there are no negative cycles, then they can replace Dijkstra’s algorithm with the Bellman-Ford algorithm to find the shortest paths for drivers. They can also use the Bellman-Ford algorithm to detect whether there are negative cycles in the graph.

5. If there are negative cycles, then they cannot use the Bellman-Ford algorithm to give drivers directions, but they can use it to find a negative cycle. By driving on that cycle, the students would make money. The payment they get from driving down Trump Avenue would be greater than the cost of the toll roads that complete the cycle. Assuming that they make a profit when the cost of fuel is factored in, then they have found a way of making money through arbitrage. This could be exploited to take as much money from the US government as they wanted, which would drive up the deficit.

7 Substring search with a DFA

25 points

Construct a deterministic finite-state automaton (DFA) for the pattern “GOOGOL”. Fill in the state transition table, draw the DFA (labeling each transition with its letter), and answer the questions below.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
<td>G</td>
<td>O</td>
<td>G</td>
<td>O</td>
<td>L</td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1. When the DFA is state \( j \) and it encounters a match character, what state does it move to?

2. When the DFA is in state \( j \) and it encounters a mismatch character, how do you determine what state it should move into?

3. Our table only shows the transitions for the 3 unique letters in our pattern. What state we should transition to if we encountered a different letter in the text, like “E”?

4. What is the name of the algorithm that we studied to construct a DFA from a pattern, and what is its run-time?

Solution

Here is the state transition matrix:

<table>
<thead>
<tr>
<th>pattern.charAt(j)</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>G</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>L</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
1. When the DFA is state $j$ and it encounters a match, what state does it move to? The DFA advances to state $j + 1$ on match transitions.

2. When the DFA is in state $j$ and it encounters a mismatch character, how do you determine what state it should move into? When there is a mismatch, then the next transition will be determined by moving the DFA through the substring of the pattern from 1 to $j$, ending at state $x$. The transition for the mismatch character will be whatever it is for that character at state $x$. For example if there is a mismatch at state 5, then we move through the machine for pattern letters 1-5 "OOGO", which would put us in state $x = 2$. Taking a mismatch "G" at 5 would therefore transition us to state 1. Taking the mismatch "O" would take us to state 3.

3. What state should we transition to if we encountered a different letter in the text, like “E”? State 0.

4. What is the name of the algorithm that we studied to construct a DFA from a pattern, and what is its run-time? Knuth-Morris-Pratt (KMP) constructs the DFA in linear time.

Up to 17 points for the correct transition matrix and graphical representation. Minus 1 point for each incorrect transition (floor of 0). 2 points for a correct answer to each of the questions.

8 Preserving national security and privacy

10 points

The NSA and the FISA court which oversees it are struggling to come to an agreement on a way to preserve privacy rights while upholding national security. At issue is the fact NSA would like to monitor all messages sent by email and text anyone in the world in order to detect a key phrase like “sleeper cell attack” that might indicate that an attack is imminent (in which case they would raise the terror threat level).

1. You are the judge in charge advocating for the rights of citizens, and deciding what the NSA should be allowed to do. Assuming that the NSA has the capability of monitoring all transmissions as they stream by, make an argument the NSA should be allowed to search for “sleeper cell attack” using a DFA, and that the use of a DFA is sufficient to ensure the right of citizens to keep their communications private.

2. You are a defense contractor working for the NSA. A judge has granted the NSA the right to search the stream for any number of key phrases, and to raise the terror threat level, provided they use a finite state machine to do so. If the NSA has a list of high-risk cities, and it wants to detect a set of patterns like “attack [U.S. CITY NAME] [tomorrow or next week or next month]”, would a DFA be effective? If so, how? If not, what type of machine would you construct?

Solution

1. The DFA would be adequate to preserve privacy rights since we only need to preserve 1 character from the streaming text in memory while we are performing the pattern recognition with a DFA.

2. The DFA is only designed to recognize a single pattern, so it is not a good method for recognizing a set of patterns. Instead, we construct a non-deterministic finite state automaton (NFA) that recognizes regular expressions, like the one for the pattern that the NSA wants to recognize.
9 Matching parentheses
10 points

Design an algorithm to say whether a string contains has all of its parentheses properly balanced or not. Here are several example inputs, and the value that your algorithm should return:

- \( ()()() \) – true
- \( \{\text{(the [small] cat)}\} \) – true
- \( \text{(the cat} \) – false
- \( \{()\} \) – false

Solution

Students should propose using a stack. Whenever an open paren is encountered, it is pushed onto the stack. Whenever a close paren is encountered, an item is popped from the stack and checked to see if it is the matching type. Any non-paren characters are skipped. The algorithm returns false if it tries to pop when the stack is empty, or if it pops and the returned open paren isn’t the same type as the closed paren that initiated the pop. The algorithm returns true if the stack is empty when the input string is completely consumed.

10 Trivia
3 points

1. Who said “Object-oriented programming is an exceptionally bad idea that could only have originated in California”?

2. Which of these data structures were invented by one of the textbook authors? R-way tries, TSTs, hashing, 3-way quicksort.

3. Which of the following awards has your professor won? The MacArthur Award, the Sloan Foundation Fellowship, the Turing Award.

Solution

1. Edsger Dijkstra

2. TSTs and 3-way quicksort were invited by Sedgwick.

3. The Sloan Fellowship

11 Seam Carving
15 points

Computer science researchers came up with a very cool algorithm that they published at SIGGRAPH that allows them to dynamically resize an image, so that it can display on cell phones and web browsers. The
method tries to resize an image with minimal distortion by removing one pixel from each row in the image (to narrow its width). Instead of simply selecting one row to cut, the algorithm can pick any pixel to remove. The researchers were able to automatically assign a value to each pixel that represents how perceptible it would be if removed, called its “energy function”. To narrow an image, they find a seam. The seam is a path of pixels, each connected to one of its eight neighbors, that goes from the top of the image to the bottom of it. The pixels in the seam are removed to narrow an image.

Given the energy function for every pixel in the image, how would you find the best seam to carve out of the image? Please include the following information in your answer:

1. How you would represent the image as a graph? What are the vertices and how are they connected? You can draw a picture if you'd like.

2. What kind of graph it is? For example, directed/undirected, weighted/unweighted, cyclic/acyclic, etc.

3. What graph processing algorithm would you apply to find the seam?

4. Would you need any special vertices that weren’t in the original image?

5. If you wanted to reduce the height of the image, instead of narrowing its width, how would your graph change?

Solution

1. There is a vertex for each pixel. There are 3 edges going out from each pixel to its 3 downward neighbors.

2. Weighted DAG.

3. Somehow, find the shortest path from the dummy source vertex to the dummy sink vertex:
   - 3 Points: Topologically sort the DAG and relax the edges in order for a runtime of $O(WH)$
   - 2 Points: Run Dijkstra from the dummy source for a runtime of $O(WH \log(WH))$
   - 1 Points: Run Bellman Ford from the dummy source for a runtime of $O((WH)^2)$

4. You could add a single source vertex (that has an edge to each vertex in the top row) and a sink vertex (that has an edge from each vertex in the bottom row).

5. Same as above, but edges go from left to right instead of top to bottom. Source connects to every pixel in the leftmost column. Sink combines every pixel in the rightmost column.

12 Photoshop Flood Fill

15 points

Photoshop has a magic wand tool that allows you to click on a pixel in an image in order to select all of the pixels that share the same color that are adjacent to one another. This lets you turn a gray sky blue with a flood fill. Describe a graph processing algorithm that would let you implement the ability to select all adjacent pixels that are colored gray. In your explanation say how you would represent the image as a graph. What are the vertices? When do they have an edge? Is it a directed or undirected graph? Would it be preferable to use DFS or BFS for a very large image?
Solution
To solve this problem, we can create an undirected graph where vertices are pixels, and edges connect adjacent pixels. The graph can be undirected because the adjacency relationship is symmetric (i.e. if i is adjacent to j, then j is adjacent to i).

To find all pixels of the same color of the clicked one, we run a slightly modified version of BFS. When iterating over adjacent nodes in the graph, we mark only nodes with the same color as the origin (e.g. gray) as selected, and continue the search from these nodes only. The algorithm completes in $O(|V| + |E|)$, the same as BFS, because the extra color check takes constant time. The algorithm is guaranteed to find only pixels of the same color because we ignore nodes that are a different color. And it is guaranteed to find all adjacent pixels of the same color because BFS exhaustively searches the graph.

If the images is large, it may be preferable to use BFS to DFS because the same-color area may be very big, and the recursion stack required by DFS might result in stack overflow.

13 Finding the rank of an entry
10 points

Assume you have a BST and a min-heap and you are given an entry that exists in both data structures. You want to find the rank of the entry. An item is of rank k if it is the kth smallest value in its data set.

1. In what runtime could you find the rank of the entry in the BST? Why? (Note: you do not need to provide a full algorithm).

2. Assume that you changed the BST so that each node contained the total count of all of its descendants. How would that change the runtime?

3. Can you find the rank of this element in the same runtime in your min-heap? Why or why not?

Solution
Trevin provided this question. Thanks, Trevin!

- We can find the item in $O(N)$ (where N is the number of elements in the tree). To find the rank of the element, we can do an in-order traversal of the BST until we find it, simply maintaining a count of the nodes visited prior. Since the rank of the element can be at most N, the runtime of this search is $O(N)$ in the worst case.

- If each BST contains the count of its descendants, then we do not need to do the complete in-order traversal. Instead we can search for the element, and update a rank counter, r, using the following rules. We begin the search at the root, and initialize $r = 0$. Every time we take a right branch from some node $t_i$ along the search, we increase $r$ by $1 + \text{cnt(leftSubtree)}$, where $\text{cnt(leftSubtree)}$ is the number of descendants for the node pointed to by the left branch from $t_i$. When we find the element finally at node $t_j$, we set the rank $k = r + 1 + \text{cnt(leftSubtree)}$, where in this case $\text{cnt(leftSubtree)}$ gives the number of descendants for the node pointed to by the left branch from $t_j$.

  Assuming the tree is balanced, we are guaranteed to complete the search in $O(lg N)$ time.

- No, we cannot get it in the same runtime. The rank of an element in a min heap can be determined by removing min until we find the element. The delete min takes $O(lg N)$ time, and we need $k$ calls to it, so we get the rank in $O(k lg N)$, which is equivalent (in the worst case) to $O(N lg N)$.
14 Dijkstra’s Mod

10 points

Consider a graph with integer edge weights bounded by some positive integer k. Describe a modification (to the priority queue) of Dijkstra’s algorithm to improve the complexity to $O(V + E)$.

Solution

The runtime of Dijkstra’s algorithm for finding shortest paths in a directed graph with a PQ implementation is $O(E \log V)$ because inserting vertices into the PQ takes $O(\log V)$ time in the worst case. Therefore in order to improve the runtime to $O(E + V)$, we need to implement a data structure that can execute inserts and removals in linear time. For this we can use an array of arrays.

Because each edge weight is an integer bounded by $k$, we know that the distance from every node to the origin is also an integer, and that the maximum distance is $k(V – 1)$ (since there are at most $V – 1$ edges between two nodes). If we replace the priority queue with a distance-indexed array, $\text{distArray}$, of length $kV$, then each node $w$ can be inserted into $\text{distArray}$ at index $\text{distTo}[w]$ when it is first discovered. Dijkstra’s algorithm processes nodes in order of increasing distance from the origin. Therefore we simply need to initialize $\text{distArray}$ by putting the origin node in $\text{distArray}[0]$, and then walk through $\text{distArray}$ one index at a time. The algorithm terminates when we reach $\text{distArray}[kV-1]$.

Some additional consideration needs to be made about the specific structure of the storage arrays at each index of $\text{distArray}$. They need to support (a) constant-time retrieval (for when we need to change the location of a node that is already in $\text{distArray}$), and (b) iterating through contents in minimal time, so that we can quickly scan for nodes at each index of $\text{distArray}$ as we work from its start to end. One option might be to maintain both a linked list and a length-$V$ array at each index of $\text{distArray}$; when we insert a node $w$ into $\text{distArray}[w]$, we both put it into the linked list, and into the array at index $[w]$. The array supports constant-time retrieval, and the linked list supports minimal time iteration over all nodes in the array at a given distance.

The entire algorithm is as follows. Initialize $\text{distArray}$ by inserting the origin node, $s$, at location $\text{distArray}[0]$. For each graph edge $(s, w)$, set $\text{distTo}[w] = \text{len}(s, w)$, $\text{edgeTo}[w]=s$, and place node $w$ in $\text{distArray}[w]$. The rest of the algorithm is the same as the original, except instead of deleting the minimum entry from PQ at each iteration, we simply iterate over indices in $\text{distArray}$ from 1 to $kV – 1$. At each index, if there is one or more nodes in that location, we process the nodes as usual. The algorithm terminates when we reach $\text{distArray}[kV-1]$.