CS 5002 Final Exam - Fall 2018

SOLUTIONS

Please answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, you can continue on the back of the page.

Good Luck!

Question	Points	Score
Combinatorics and probability	25	
Data structures	15	
Algorithms and proofs	20	
Divide and Conquer Algorithms	12	
Graphs and Trees	25	
Total:	97	

Problem 1 ***(25 pts)***

(a) Let's assume that eight different airlines fly from Seattle to Boston, and seven different airlines fly from Boston to Zurich. How many different pairs of airlines can you choose on which to book your trip from Seattle to Zurich, when you have an option to pick an airline for the leg of the trip from Seattle to Boston, and an airline for the continuation flight to Zurich? ***(5 pts)***

Solution: To solve this problem, we rely on the fact that the companies flying from Seattle to Boston are different from the companies flying from Boston to Zurich. That allows us to make a simplifying assumption that we can independently choose airlines for both parts of the complete trip. We can now write:

- Number of ways to choose a flight from Seattle to Boston: $\binom{8}{1} = 8$
- Number of ways to choose a flight from Boston to Zurich: $\binom{7}{1} = 7$

The total number of different pairs of companies that we can choose can now be found using the product rule. Let's denote the number of pairs of companies as $Pairs$:

$$Pairs = \binom{8}{1}\binom{7}{1} = 56 \tag{1}$$

(b) Some software development company has 12 senior-level engineers, and 8 junior-level engineers. How many ways are there for the company to form a team of six members, if: ***(10 pts)***

- The number of senior and junior team members has to be equal?

Solution: If the number of senior and junior team members has to be equal, we immediately know that we have to have three junior and three senior members on the team. Since we do not care about the order of the members on the team, we can find the number of ways to form the team with three junior and three senior members to be equal to:

$$\#teams = \binom{12}{3}\binom{8}{3} \tag{2}$$

- The team member must have more junior-level members than senior-level members, but such that there is at least one senior-level member on the team?

Solution: If the number of junior team members has to be greater than the number of senior members, we have two possible options:

- There is one senior member and five junior members
- There are two senior member and four junior members

To count all possible options, we can now apply the summation rule, to get:

$$\#teams = \binom{12}{1}\binom{8}{5} + \binom{12}{2}\binom{8}{4} \tag{3}$$

(c) Let's assume that every child born is equally likely to be a boy or a girl, and let's consider a family with exactly four children. Let GGBB indicate that the first two children born are girls and the last two children born are boys. Similarly, let GBGB indicate that the oldest child is a girl, the second born child is a boy, and so on. ***(10 pts)***

- How many tuples of exactly four children are possible?

Solution: The number of tuples of exactly four children can be found by observing that for every child, we have two options - a child can be a boy and a girl. Using our original simplifying assumption that every child born is equally likely to be a boy or a girl, we get the number of possible tuples to be equal to:

$$\#tuples = 2^4 = 16 \tag{4}$$

- What is the probability that exactly one of the four children is a boy?

Solution: To find the probability that exactly one child is a boy, we need to consider all the possible tuples containing only one boy. There are four such tuples:

$$(BGGG), (GBGG), (GGBG), (GGGB) \tag{5}$$

Assumong all 16 tuples are equally likely to happen, the probability that exactly one child is a boy can be found as:

$$\mathbb{P}[1boy] = \frac{\#single\_boy\_tuples}{\#tuples} = \frac{4}{16} = \frac{1}{4} \tag{6}$$

- What is the probability that at least two children are girls?

Solution: To find the probability that some family has at least two girls among four children, we first observe that that means that the family may have two girls, three girls or four girls. We could easily compute that probability as a sum of individual probabilities, but in this problem, we can do something simpler.
We can observe that some family has at least two girls if and only if it doesn't have a family of all four boys, or a family with exactly one girl.
We know that that the probability that a family has no girls is equal to $\frac{1}{16}$, and we have already computed the probability that the family has one boy. By symmetry of the relationship, we can immediately see that the probability that a family has one girl is equal. Now, we can find the probaility that a family has at least two girls as follows:

$$\mathbb{P}[at-least-2-girls] = 1 - \mathbb{P}[no\_girls] - \mathbb{P}[exactly\_one\_girl] = 1 - \frac{1}{16} - \frac{1}{4} = \frac{11}{16} \tag{7}$$

Here is a blank page for you to use, if you need it.

Problem 2 ***(15 pts)***

(a) This question pertains to the following function. ***(5 pts)***

```
Input: String str

DoSomething(str){
  queue q;
  stack s;
  for(i = 0; i < str.length(); i++){
    q.add(str[i])
  }
  while(!isEmpty(q)){
    s.push(q.remove())
  }
  while(!isEmpty(s)){
    char = s.pop()
    if(char == 'f'){
      q.add('*')
    }
    else{
      q.add(char)
    }
  }
    print(q)
}
```

Assuming that method (`print(q)`) allows us to print the whole data structure q from its first to its last element, briefly describe in "natural language" what the algorithm is doing.

Solution: The given algorithm takes a string, and reverses it. In doing so, it potentially modifies the reversed string. If the original string contained lower-case letter 'f', that element of the string is replaced by a '*'. String reversal is implemented using a stack and a queue.

What is the runtime of this algorithm, considered using Big-O notation.

Solution: The runtime of the given function is $O(n)$, where n represents the lenght of the string. To see that, let's observe what is going on in the function:

- We first push the whole string into a queue - the runtime of this operation is $O(n)$
- We then dequeue the whole string from the queue and push it onto the stack - the runtime of this operation is also $O(n)$
- Lastly, we pop the whole string from the stack, examine the popped element, possibly modify it, and push into the queue - the runtime of this operation is also $O(n)$
- Lastly, we print the queue - the runtime of this operation is also $O(n)$

Each of these individual operations are sequential, therefore the runtime of the whole function is $O(n)$.

(b) Assume you are given a linked list that is sorted in an increasing order. Write a pseudocode for the method `sortedInsert(list, node)` that inserts the `node` into the correct position in the given linked list, such that the sorted order is preserved. If the given `node` already exists in the list, then the duplicate is added after the original node. ***(10 pts)***

As an example, let's assume that we have a correct implementation of method `sortedInstert(list, node)`, and we are given a linked list 1 – 3 – 5 – 9 –25. If we use method `sortedInsert` to add node 7 to the given list, the resulting list should be 1 – 3 – 5 –7 –9 –25. Similarly, if we try to add node 5 into the given list, the resulting list should be 1 – 3 – 5 – 5 – 7 – 9 –25.

Solution: The basic strategy we take to implement method `sortedInsert(list, node)` is to iterate down the list looking for the place to insert the new node. That could be the beginning of the list, end of the list, or at a point just before a node which is larger than the new node.

Pseudocode for one possible approach is listed below:

```
// Uses special case code for the end of the list
void sortedInsert(List list, Node newNode) {
  // Special case for the head end
  if (list == NULL || list.data >= newNode.data):
    newNode.next = list;
    list = newNode;
  else:
  // Locate the node before the point of insertion
  Node currentNode = list;
  while (current.next != NULL && current.next.data < newNode.data):
    current = current.next;
  newNode.next = current.next;
  current.next = newNode;
}
```

What is the runtime of your algorithm, using Big-O notation?

Solution: The runtime of the given algorithm is $O(n)$. In the best case scenario, the list is empty, or tha new node should be added at the beginning of the list. However, if this is not the case, we may have to traverse through the whole list to find where to place the new node.

Here is a blank page for you to use, if you need it.

Problem 3 ***(20 pts)***

(a) Please consider the following "natural language" description of a simple algorithm: ***(7 pts)***

Input: an array

Output: a modified array

For every element in the array, if the value (not index!) of the current element is even, then the current element becomes equal to the sum of values of two elements in the array: the previous and the next. Otherwise, if the current element is odd, then we multipy the value of the current element by three.

If there is no previous element, we do not modify the value of the current element. If there is no next elements, the algorithm is done.

Your task: Assuming 0-based indexing for arrays, write the pseudocode for the descirbed algorithm:

Solution: A pseudocode represnting one possible solution to this problem is represented below. In the given solution, the original array is preserved (it is not destroyed and/or modified). Additionally, all array-based values for modification are always drawn from the original array.

The pseudocode does nothing about the first and the last elements, and simply skips processing them.

```
Array modifiedArray modifyArray(Array array){
  Array modifiedArray = new Array();

  //Start the loop induction variable i from 1 to skip the first element
  //Finish the loop at length() - 2 to skip the last element
  for (i = 1; i < array.length() - 2; i++){
    //Even number
    if(array[i] \% 2 == 0){
      modifiedArray[i] = array[i - 1] + array[i + 1];
    }
    //Odd number
    else {
      modifiedArray = array[i]*3;
    }
  }
return modifiedArray;
}
```

What is the running time of this algorithm?

Solution: The runtime of the given algorithm is equal to the worst case O(n) runtime of the algorithm, and it equals $O(n)$.

An example of a worst case input is virtually any array longer than 1. If we are being very precise, the worst case input is an array of the maximum allowed length (i.e., the maximum length we are able to store into the memory of our machine).

What is the best case running time of this algorithm? Provide an example of a best-case input.

Solution: The best case runtime of the given algorithm is still $O(n)$, but for any array of size 1, that $O(n)$ becomes $O(1)$.

What is the memory requirement of this algorithm, if we assume that we preserve the original array?

Solution: In order to preserve the original array, we need to create a copy of it, which we then modify and return. Therefore, the memory requirement of the given algorithm is $O(n)$.

(b) Consider the following simple algorithm, searching for the maximum value in an unsorted array: ***(5 pts)***

```
Input: Unsorted array A

FindMax(A){
  max = 0;
  for(i = 0; i < A.length()/2; i++){
    if (A[0] >= A[A.length()-1]){
      max = A[0];
      {
    else {
      max = A[A.length() - 1];
      }
}
if(A[A.length()/2 + 1] > max){
  max = A.length()/2 + 1;
  }
}
```

Disprove that the given algorithm is correct, using **proof by counterexample** technique:

Solution: The given pseudocode is supposed to find the maximum value in an unsorted array. One way to do so would be to use a `for` loop, and check the value of every element of an array, while keeping a record of the current maximum.

The given pseudocode may resemble such an implementation, but the given loop is useless, beucase the induction variable is never used within the loop. Instead, in the given pseudocode, we repeatedly check whether or not the value of the first element is greater than the value of the last element, and set the current maximum accordingly. This approach, altough wasteful, could work for a sorted array, but our array is not sorted.

The pseudocode sort of takes into account the fact that an array is not necessarily sorted by comparing the current max with an element at the position $halfLen + 1$. However, we are not guaranteed that the element with the maximum value will always be stored there.

Therefore, the given algorithm works correctly only for arrays where the maximum value is stored at index 0, length -1, and halfLen +1. An array where maximum value is stored at any other position is a counterexample. One such example is [1, 10, 2, 3, 4, 5, 6, 7]

(c) Prove the following statement using the **proof by induction** technique: ***(8 pts)***

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1} \tag{8}$$

Solution:

Base case: When $n = 1$, the left side of equation (8) becomes equal to:

$$\frac{1}{i(i+1)} = \frac{1}{2} \tag{9}$$

Similarly, the right side of the same equation becomes equal to:

$$\frac{n}{n+1} = \frac{1}{2} \tag{10}$$

Therefore, the given statement holds for the base case.

Inductive hypothesis: Let assume that the given statement holds true for some value $k \in \mathbb{Z}_+$.

Inductive step: Let now set $n = k + 1$. The left-hand side of the given statement (equation (8)) can now be rewritten as:

$$\sum_{i=1}^{k+1} \frac{1}{i(i+1)} = \sum_{i=1}^{k} \frac{1}{i(i+1)} + \frac{1}{(k+1)(k+2)} \tag{11}$$

Using the inductive hypothesis, we can rewrite equation (11) as:

$$\begin{aligned} \sum_{i=1}^{k+1} \frac{1}{i(i+1)} &= \sum_{i=1}^{k} \frac{1}{i(i+1)} + \frac{1}{(k+1)(k+2)} = \frac{k}{k+1} + \frac{1}{(k+1)(k+2)} \\ &= \frac{k(k+2)+1}{(k+1)(k+2)} = \frac{k^2+2k+1}{(k+1)(k+2)} = \frac{(k+1)^2}{(k+1)(k+1)} \\ &= \frac{k+1}{k+2} \end{aligned} \tag{12}$$

Equation (12) show that we have successfully shown that the left-hand side of the starting expression (11) equals its right-hand side. Therefore, the statement is satisifed in the inductive step. This proves that, by the principle of induction, statement (8)holds for all $n \in \mathbb{Z}_+$.

Here is a blank page for you to use, if you need it.

Problem 4 ***(12 pts)***

(a) Consider the following recurrence relation: ***(7 pts)***

$$T(n) = T\left(\frac{n}{5}\right) + T\left(\frac{4n}{5}\right) \tag{13}$$

Please draw the recurrence tree (to at least 2 levels including the root). todo

Solution: The recurrence trees for the two levels are depicted in Figure 1.

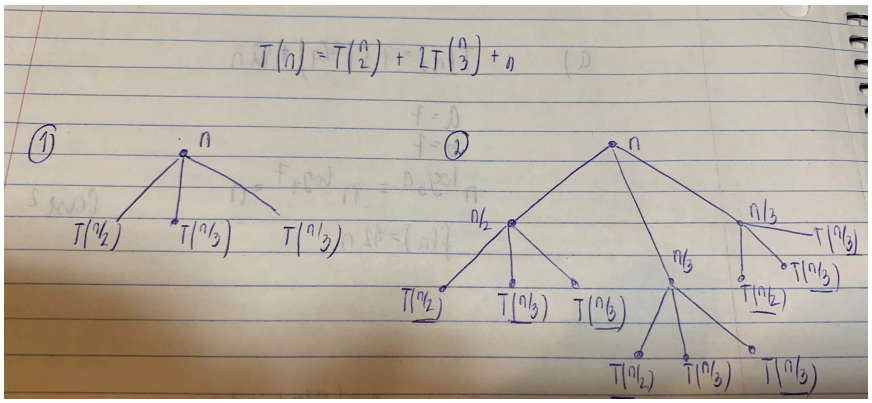

Figure 1: Recurrence tree created in Problem 4, part (a).

Assuming that the root is the first level of the three, what is the total amount of work done on the 2nd level of the tree?

Solution: The total amount of work done on the 2nd level of the tree can be found as:

$$T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{3}\right) = \frac{n}{2} + \frac{2n}{3} = \frac{7n}{6} \tag{14}$$

The total amount of work done on the 3rd level of the tree can be found as:

$$T\left(\frac{n}{2}\right) + 2T\left(\frac{n}{3}\right) + 2T\left(\frac{n}{2}\right) + 4T\left(\frac{n}{3}\right) = \frac{n}{4} + \frac{2n}{6} + \frac{2n}{6} + \frac{4n}{9} = \frac{49n}{36} \tag{15}$$

How deep is the tree?

Solution: To find the height of the tree, we take the height of the longest branch in the tree, which is the branch that reduces the size of the problem the slowest (the largest denominator). The height of the tree in this problem is $\log_2 n$; there are $\log_2 n + 1$ levels.

(b) Use the Master Theorem to find a bound for these recurrences. For each, indicate $a, b, f(n), n^{\log_b a}$ and which case applies. ***(5 pts)***

- $T(n) = 4T\left(\frac{n}{7}\right) + 12n$

Solution: To solve this problem, we apply the Master theorem, and recall that a runtime of some algorithm p can be expressed with a recurrence relation $T(n)$, given as:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n) \tag{16}$$

Using equation (16), we can find $a, b, f(n)$ and $n^{\log_b a}$ as follows:

$$\begin{aligned} a &= 4 \\ b &= 7 \\ f(n) &= 12n \\ n^{\log_b a} &= n^{\log_7(4)} \end{aligned}$$

Since $7 > 4$, we know that $\log_7(4)$ is strictly smaller than zero. Therefore, $n^{\log_b a} = n^{\log_7(4)} = n^{0.xxx}$ Applying now the Master theorem, we observe that:

$$f(n) = 12n > n^{0.xxx} \tag{17}$$

and we observe that we are in the case 3 of the Master theorem: If $f(n)$ is bigger than $n^{\log_b^a}$ then $T(n) = \Theta(f(n))$.

- $T(n) = 4T\left(\frac{n}{2}\right) + 3n$

Solution:

$$\begin{aligned} a &= 4 \\ b &= 2 \\ f(n) &= 3n \\ n^{\log_b a} &= n^{\log_2(4)} = n^2 \end{aligned}$$

Applying now the Master theorem, we observe that:

$$f(n) = 3n < n^2 \tag{18}$$

and we observe that we are in the case 1 of the Master theorem again: If $f(n)$ is smaller than $n^{\log_b^a}$ then $T(n) = \Theta(n^{\log_b^a})$.

- $T(n) = 5T\left(\frac{n}{5}\right) + n$

Solution:

$$\begin{aligned} a &= 5 \\ b &= 5 \\ f(n) &= n \\ n^{\log_b a} &= n^{\log_5(5)} = n \end{aligned}$$

Applying now the Master theorem, we observe that:

$$f(n) = n < n \tag{19}$$

and we observe that we are in the case 2 of the Master theorem again: If $f(n)$ is about the same as $n^{\log_b^a}$ then $T(n) = \Theta(n^{log_b a} \lg n))$.

Here is a blank page for you to use, if you need it.

Problem 5 ***(25 pts)***

(a) Consider the following graph, depicted in Figure 2. Represent the given graph using the **adjacency matrix**. ***(5 pts)***

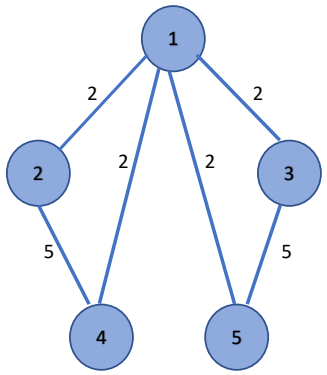

Figure 2: Graph used in Problem 5, part (a).

Solution: The adjaceny matrix for the given graph is a 5×5 matrix, given as:

$$M[\mathcal{G}] = \begin{bmatrix} 0 & 2 & 2 & 2 & 2 \\ 2 & 0 & 0 & 5 & 0 \\ 2 & 0 & 0 & 0 & 5 \\ 2 & 5 & 0 & 0 & 0 \\ 2 & 0 & 5 & 0 & 0 \end{bmatrix} \tag{20}$$

The given adjacency matrix is symmetric. We therefore conclude that the given graph is undicrected.

Is the given graph weighted or unweighted?

Solution: The given graph is weighted.

(b) Consider the following adjacency matrix below, representing some graph $\mathcal{G}$. Please transform the given adjacency matrix into the adjacency list. ***(4 pts)***

$$M(\mathcal{G}) = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \tag{21}$$

Solution: Let's recall that an adjacency list represents a graph as follows:

- Every node in the graph represents a row in our adjacency list.
- Every row n of our adjacency list is a separate liniked list, whose values represent all the nodes that node n is connected to via an edge.

So, for the given adjacency matrix, we get the following adjacency list:

Node 1: { Node 2, Node 3 }

Node 2: { Node 4 }

Node 3: $\{\emptyset\}$

Node 4: { Node 5, Node 6 }

Node 5: $\{\emptyset\}$

Node 6: $\{\emptyset\}$

(c) Consider the following pseudocode, representing the **stack-based** implementation of the BFS (breadth-first search), used to print the values of all the nodes in the tree. Please modify the given pseudocode to find the maximum value among all the nodes in the tree. You can assume that the tree is an integer tree. ***(5 pts)***

```
stack s;
push(s, root);
Node currentNode;
while (!isEmpty(currentNode)){
  currentNode = pop(s);
  print(currentNode);
  push(currentNode.rightChild);
  push(currentNode.leftChild);
  }
```

Solution: To modify the given pseudocode, presenting the stack-implementation of DFS, we observe that we only need to modify the code within the while loop, to make sure that it correctly searches for the node with the maximum value in the tree.

The modified code is given below:

```
stack s;
Node currentNode;
integer maxValue = 0;
Node maxNode;

push(s, root);

while (!isEmpty(s)){
  currentNode = pop(s);
  if(currentNode.data > maxValue){
    maxValue = currentNode.data;
    maxNode = currentNode;
  }
  print(currentNode);
  push(currentNode.rightChild);
  push(currentNode.leftChild);
}
{\tiny }        print("The maximum found value is", maxValue);
  return maxNode;
```

What is the time complexity (runtime) of your modified algorithm, considered using Big-O notation?

Solution: The time complexity of our modified algorithm did not change, compared to the original DFS algorithm - it is still driven by pushing nodes on the stack, and then removing nodes from the stack. The added change is just a simple comparison. So, the complexity of the modified algorithm is equal to $O(V + E)$, where V represents the number of vertices, and E the number of edges.

(d) Consider the following tree, depicted in Figure 3. Please write down what gets printed when the given tree is traversed using: ***(6 pts)***

- Pre-order traversal
- In-order traversal
- Post-order traversal

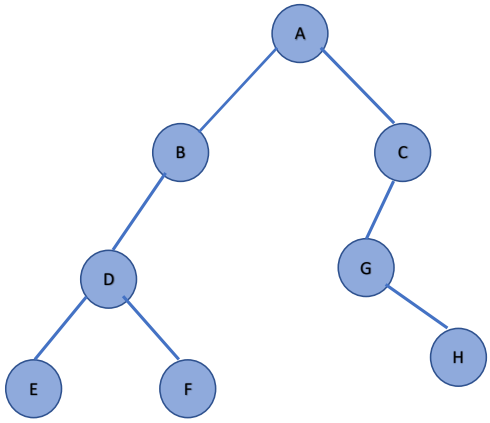

Figure 3: Graph used in Problem 5, part (d).

Solution: **Pre-order traversal:** A – B – D – E – F – C –G – H
In-order traversal: E – D – F – B – A – G – H – C
Post-order taversal: E – F – D – B – H – G – C – A

(e) Use Dijkstra's algorithm to find the shortest path from A to F for the graph represented in Figure 4. Please, list all steps of the algorithm. ***(5 pts)***

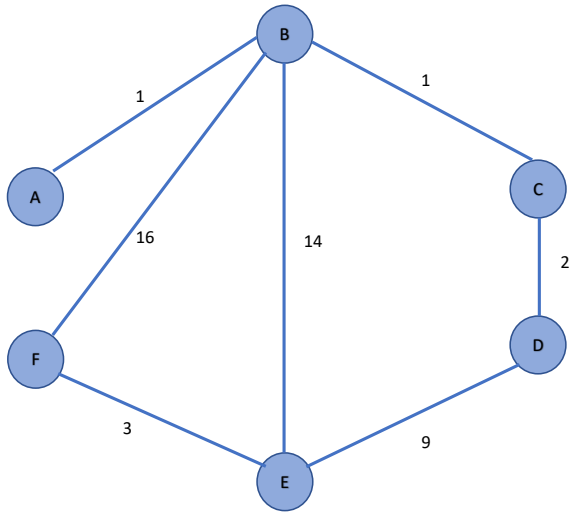

Figure 4: Graph used in Problem 5, part (e).

Solution: The steps of the Dijkstra's algorithm are presented below:

Step 1:

Source	*Via*	*Destintion*	*Cost*
A	A	A	0
A	A	B	1
A	A	C	∞
A	A	D	∞
A	A	E	∞
A	A	F	∞

(22)

Step 2:

Source	*Via*	*Destintion*	*Cost*
A	A	A	0
A	A	B	1
A	B	C	2
A	B	D	∞
A	B	E	15
A	B	F	17

(23)

Step 3:

Source	*Via*	*Destintion*	*Cost*
A	A	A	0
A	A	B	1
A	B	C	2
A	C	D	4
A	B	E	15
A	B	F	17

(24)

Step 4:

Source	*Via*	*Destintion*	*Cost*
A	A	A	0
A	A	B	1
A	B	C	2
A	C	D	4
A	D	E	13
A	B	F	17

(25)

Step 5:

Source	*Via*	*Destintion*	*Cost*
A	A	A	0
A	A	B	1
A	B	C	2
A	C	D	4
A	D	E	13
A	E	F	16

(26)

The algorithm terminates after step 5, because no other reductions in cost are possible. From the table represented in step 5, we find the optimal path from node A to node F to be a path $A--B--C--D--E--F$, with the associated cost equal to 16.

Here is a blank page for you to use, if you need it.