CSE 101: Design and Analysis of Algorithms
Winter 2018

Homework 1 Solutions
Due: 01/19/2018, 11.59 PM

Exercise 1. (DPV 1.3) *D*-ary Tree

A d-ary tree is a rooted tree in which each node has at most d children. Show that any d-ary tree with n nodes must have a depth of $\Omega(\frac{\log(n)}{\log(d)})$ depth. Can you give a precise formula for the minimum depth that it could possibly have

Solution: Any d-ary tree with n nodes must have a depth of $\Omega(log\ n/log\ d)$:
A d-ary tree of height h has at most $1 + d + \ \ ... \ \ + d^h = (d^{h+1} - 1)/(d - 1)$ vertices. We can prove this inductively.

(a) **Claim:** A d-ary tree of height h has at most $1 + d + \ \ ... \ \ + d^h = (d^{h+1} - 1)/(d - 1)$ vertices.

(b) **Base Case:** When the height of h of the tree is 1, the number of vertices is at most $1 + d = (d^{1+1} - 1)/(d - 1)$. Therefore, the statement is true when $h = 1$.

(c) **Induction Hypothesis:** The statement is true when the height h of the tree is $\leq k$.

(d) **Induction Step:** We will show that the statement is still true when $h = k + 1$. A d-ary tree of height $k + 1$ consists of a root that has at most d children, each of which is the root of a subtree with height at most k. By the Induction Hypothesis, the number of vertices in a d-ary tree of height $\leq k$ is at most $1 + d + \ \ ... \ \ + d^k = (d^{k+1} - 1)/(d - 1)$. Therefore, a tree with height $k + 1$ has at most $1 + d(1 + d + \ \ ... \ \ + d^k)$ vertices. Hence a tree with height $k + 1$ has at most $(d^{k+2} - 1)/(d - 1)$ vertices. We have completed the induction.

The height of a tree is equal to the maximum depth D of any node in the tree.
For a tree of height h, the maximum number of vertices $n \leq (d^{h+1} - 1)/(d - 1)$. Therefore,

$$d^{h+1} \geq n \cdot (d - 1) + 1 \tag{1}$$

$$(h + 1) \geq log_d(n \cdot (d - 1) + 1) \tag{2}$$

$$D = h \geq (log_d(n \cdot (d - 1) + 1) - 1) \tag{3}$$

By the definition of Big-Ω, the maximum depth $D \geq (log_d(n \cdot (d - 1) + 1) - 1) = \Omega(log_d\ (n)) = \Omega(log\ (n)/log\ (d))$ is true.

To obtain the specific formula:
Let $d =$ maximum number of children, $h =$ height of tree
Geometric series:

$$\Sigma nodes\ of\ complete\ tree = \frac{1 - d^{h+1}}{1 - d}$$

Using $\Sigma nodes\ of\ complete\ tree = n$:

$$n(1 - d) - 1 = -d^{h+1}$$

$$n(d - 1) + 1 = d^{h+1}$$

$$log_d n(d - 1) + 1 = d^{h+1}$$

$$\frac{log\ [n(d - 1) + 1]}{log\ d} = h$$

This represents the height of a graph of a complete tree. To get the height of an incomplete tree, we take the ceiling of this equation. Incomplete levels of the tree will be rounded up (eg., $h = 2.334 \rightarrow h = 3$)

$$\lceil \frac{log\ [n(d - 1) + 1]}{log\ d} \rceil = h$$

Exercise 2. (DPV 1.5) Harmonic Series

Unlike a decreasing geometric series, the sum of the *harmonic* series 1, 1/2, 1/3, 1/4, 1/5 . . . diverges; that is,

$$\textstyle\sum_{i=1}^{\infty} \frac{1}{i} = \infty.$$

It turns out that for large n, the sum of the first n terms of the above series can be well approximated as

$$\textstyle\sum_{i=1}^{n} \frac{1}{i} \approx \ln(n) + \gamma.$$

where ln is the natural logarithm and γ is a particular constant 0.57721. . . Show that

$$\textstyle\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n)).$$

(Hint: To show an upper bound, decrease each denominator to the next power of two. For a lower bound, increase each denominator to the next power of two.)

Solution:
To obtain an upper bound:

$$\sum_{i=1}^{n} \frac{1}{i} \leq 1 + (\frac{1}{2} + \frac{1}{2}) + (\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}) + \cdots \leq \log_2 n$$

$$\Rightarrow \sum_{i=1}^{n} \frac{1}{i} = \mathcal{O}(\log(n))$$

To obtain a lower bound:

$$\sum_{i=1}^{n} \frac{1}{i} \geq (\frac{1}{2} + \frac{1}{2}) + (\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4}) + \cdots \geq \log_2 n$$

$$\Rightarrow \sum_{i=1}^{n} \frac{1}{i} = \Omega(\log(n))$$

Combining the above, we have $\sum_{i=1}^{n} \frac{1}{i} = \Theta(\log(n))$

Exercise 3. (DPV 1.7) Runtime for Multiplication

(DPV 1.7) How long does the recursive multiplication algorithm (page 25 of DPV) take to multiply an m-bit number by an n-bit number? Justify your answer.

Solution: Without loss of generality, assume we want to multiply the n-bit number x with the m-bit number y. The algorithm must terminate after m recursive calls because at each call y is halved (the number of bits is decreased by one). Each recursive call requires a division by 2, a check for even-odd, a multiplication by 2 (all of those take constant time) and a possible addition of x to the current result (which takes $O(n)$ time). So the total time is $O(m \cdot n)$.

Exercise 4. (DPV 1.10) Modular Arithmetic

(DPV 1.10) Show that if $a \equiv b \pmod{N}$ and if M divides N then $a \equiv b \pmod{M}$.

Solution: We are given that $a \equiv b \pmod{N}$, which means that N divides $a - b$ or

$$a - b = k_1 \cdot N$$

for some integer k_1. Since M divides N, we have,

$$N = k_2 \cdot M$$

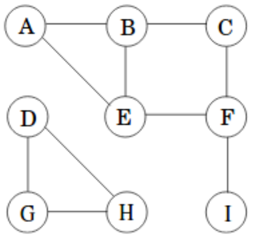
where k_2 is some integer. Therefore,

$$a - b = k_1 \cdot N = k_1 \cdot k_2 \cdot M$$

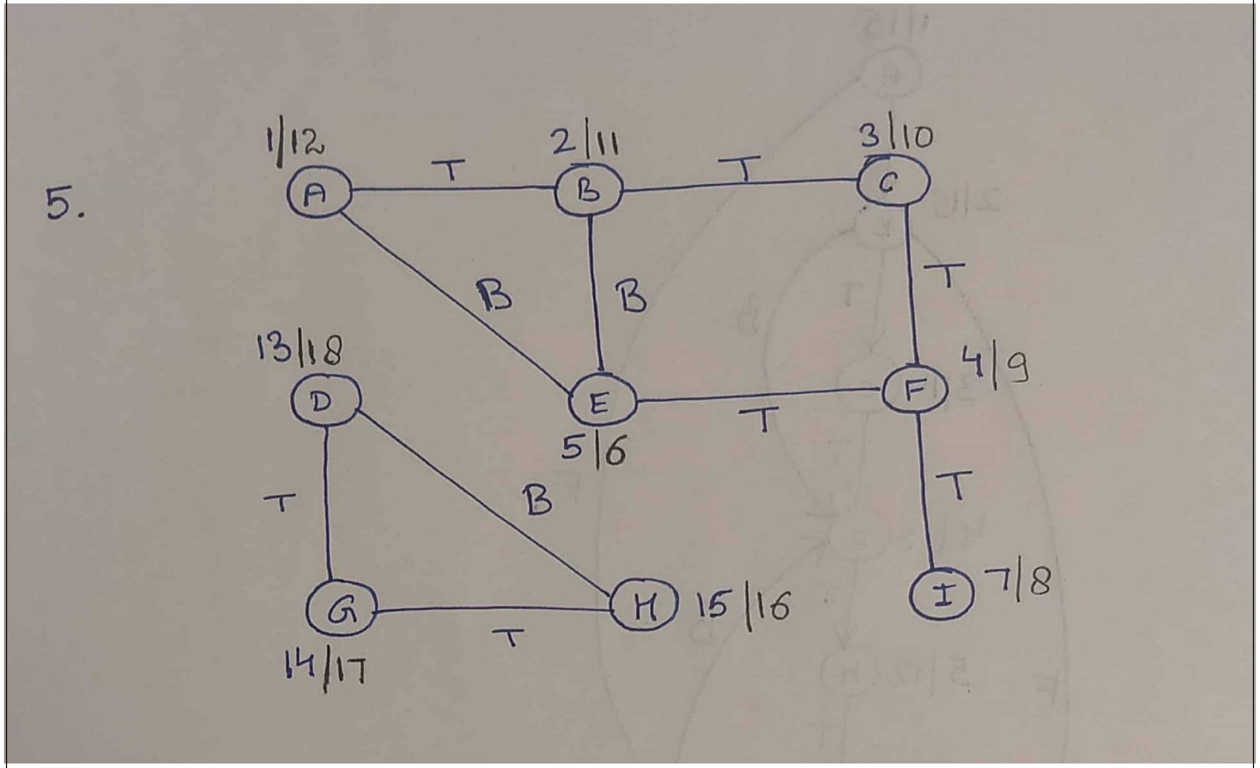
Hence, we have proved that M divides $a - b$ or $a \equiv b \pmod{M}$.

Exercise 5. (DPV 3.1) Depth-First Search 1

(DPV 3.1) Perform a depth-first search on the following graph; whenever there is a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge or a back edge, and give the *pre* and *post* number of each vertex.

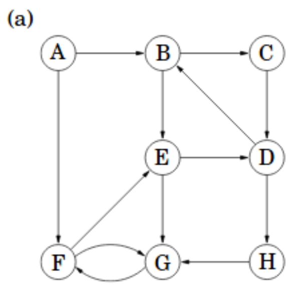


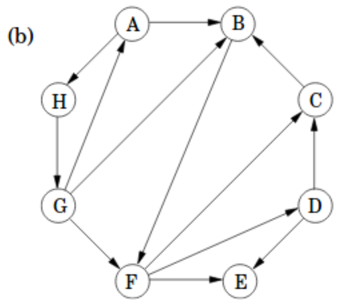
Solution: In the figure below, the *pre* and *post* numbers are shown against each node in the form *pre*/*post*. The edge type is also mentioned against each edge; tree edge is denoted by T and back edge is denoted by B.



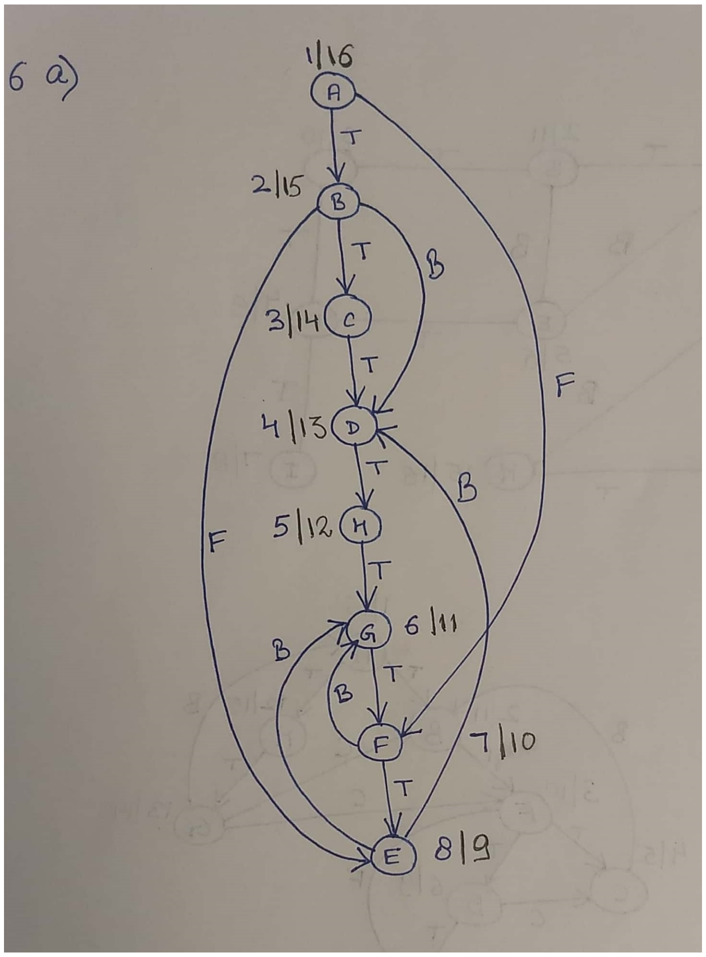
Exercise 6. (DPV 3.2) Depth-First Search 2

(DPV 3.2) Perform a depth-first search on each of the following graphs; whenever there is a choice of vertices, pick the one that is alphabetically first. Classify each edge as a tree edge, back edge, forward edge, or cross edge and give the *pre* and *post* number of each vertex.

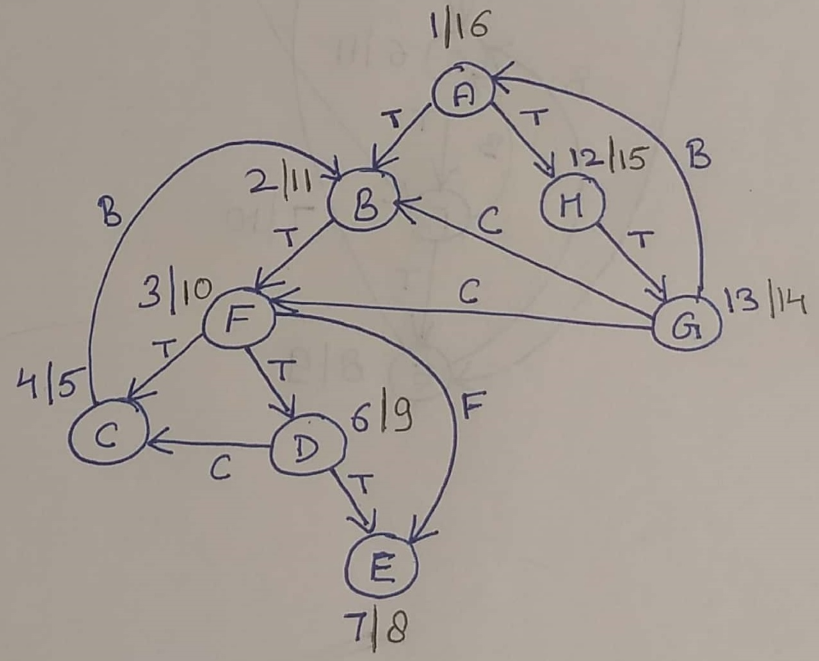




Solution: In the figures below, the *pre* and *post* numbers are shown against each node in the form *pre*/*post*. The edge type is also mentioned against each edge; tree edge is denoted by T and back edge is denoted by B, forward edge is denoted by F and cross edge is denoted by F.



6. b.)



Problem 1. Asymptotic Notation

i. $f(n) = n \cdot log(n)$, $g(n) = n^2$

Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \mathcal{O}(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n \cdot \log(n)}{n^2} = \lim_{n\to\infty} \frac{\log(n)}{n}$$

By L'Hospital's Rule:

$$\lim_{n\to\infty} \frac{\log(n)}{n} = \lim_{n\to\infty} \frac{\frac{1}{n\ln(2)}}{1} = 0$$

. Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in \mathcal{O}(g(n))$.

ii. $f(n) = n^3 + 10 \cdot n^2 + n$, $g(n) = 100 \cdot n^3$

Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \Theta(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^3 + 10 \cdot n^2 + n}{100 \cdot n^3} = \frac{1}{100}$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{1}{100}$, $f(n) \in \Theta(g(n))$.

iii. $f(n) = n \cdot log(n)$, $g(n) = n \cdot \sqrt{n}$

Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \mathcal{O}(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n \cdot \log(n)}{n^{3/2}} = \lim_{n\to\infty} \frac{\log(n)}{n^{1/2}}$$

By L'Hospital's Rule:

$$\lim_{n\to\infty} \frac{\log(n)}{n^{1/2}} = \lim_{n\to\infty} \frac{\frac{1}{n\ln(2)}}{\frac{1}{2\cdot n^{1/2}}} = \lim_{n\to\infty} \frac{2 \cdot n^{1/2}}{n\ln(2)} = \lim_{n\to\infty} \frac{2}{n^{1/2}\ln(2)} = 0$$

. Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in \mathcal{O}(g(n))$.

iv. $f(n) = log^2(n)$, $g(n) = log^3(n)$

Solution: Since the log function is monotone and increasing, for any k such that $\log(k) \geq 1$, $\log^2(k) \leq \log^3(k)$. So we can pick constants $C = 1, k = 2$ and $f(n) \leq C \cdot g(n)$ for all $n > k$. By the definition of $\mathcal{O}$, $f(n) \in \mathcal{O}(g(n))$.

v. $f(n) = log_a(n)$, $g(n) = log_b(n)$ where $a, b > 0$ and $a, b \neq 1$

Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \Theta(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{log_a(n)}{log_b(n)}$$

By rules of logarithms:

$$\lim_{n\to\infty} \frac{log_a(n)}{log_b(n)} = \lim_{n\to\infty} \frac{\frac{log_2(n)}{log_a(2)}}{\frac{log_2(n)}{log_b(2)}} = \lim_{n\to\infty} \frac{log_b(2)}{log_a(2)} = \frac{log_b(2)}{log_a(2)}$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \frac{log_b(2)}{log_a(2)}$, $f(n) \in \Theta(g(n))$.

vi. $f(n) = 2^n \cdot n^2$, $g(n) = 2^{2n}$

Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \mathcal{O}(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n \cdot n^2}{2^{2n}} = \lim_{n\to\infty} \frac{n^2}{2^n}$$

By L'Hospital's Rule:

$$\lim_{n\to\infty} \frac{n^2}{2^n} = \lim_{n\to\infty} \frac{2 \cdot n}{2^n \cdot \ln(2)} = \lim_{n\to\infty} \frac{2}{2^n \cdot \ln^2(2)} = 0$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in \mathcal{O}(g(n))$.

vii. $f(n) = e^n$, $g(n) = n!$

Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \mathcal{O}(g(n))$.

$$0 \le \frac{e^n}{n!} = \frac{e}{n} \cdot \frac{e}{n-1} \dots \frac{e}{1} \le \frac{e}{1} \cdot \frac{e}{2} \cdot (\frac{e}{3})^{n-2} \to 0, \text{when } n \to \infty$$

Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in \mathcal{O}(g(n))$.

viii. $f(n) = n^a$, $g(n) = b^n$ where $a > 0, b > 1$

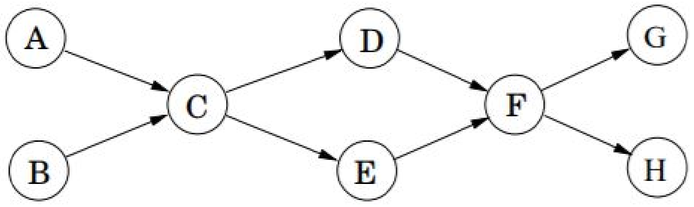
Solution: We take the limit as $n \to \infty$ to determine that $f(n) \in \mathcal{O}(g(n))$.

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{n^a}{b^n} = \lim_{n\to\infty} \frac{a \cdot (a-1) \dots (a-k) \cdot n^{a-k-1}}{b^n \cdot \ln^k(b)} = 0$$

where k is the smallest integer, such that $a - k - 1 \le 0$
Since $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, $f(n) \in \mathcal{O}(g(n))$.

Problem 2. **(DPV 3.3, 3.4) Strongly Connected Components**

A— Run the DFS-based topological ordering algorithm on the following graph. Whenever you have a choice of vertices to explore, always pick the one that is alphabetically first.



(a) Indicate the *pre* and *post* numbers of the nodes.

Solution:

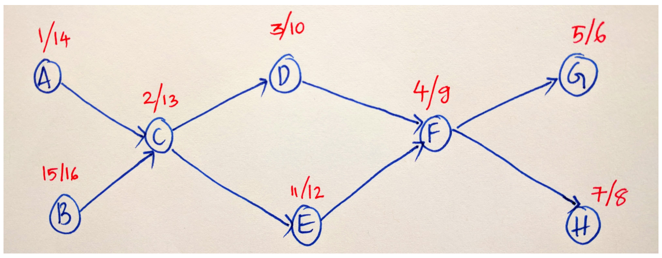


Figure 1: pre/post numbers are denoted in red.

(b) What are the sources and sinks of the graph?

Solution: Based on the execution of DFS algorithm on the given graph, vertex A and vertex B have the highest post numbers, and only have outgoing edges. Additionally, DFS finishes explore() on these vertices after visiting all nodes reachable from them. Hence, these are the source vertices.

Similarly, vertex G and H have the smallest post numbers. Therefore, these are sinks of the graph.

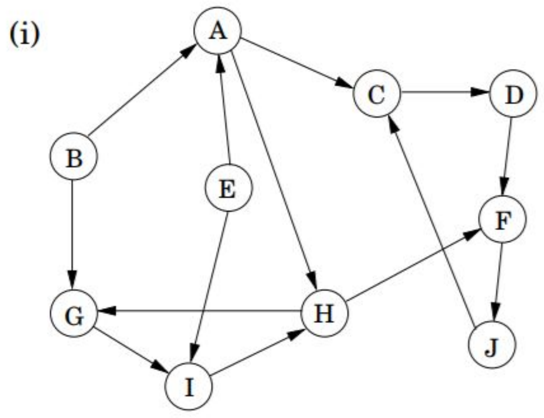
(c) What topological ordering is found by the algorithm?

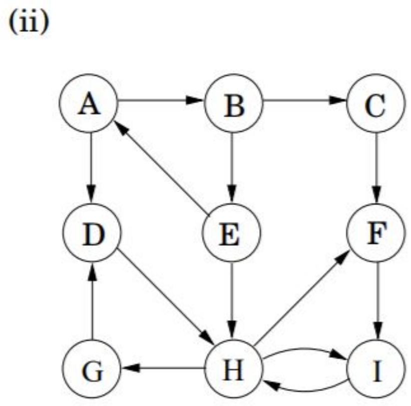
Solution: The topological ordering found by the algorithm is B,A,C,E,D,F,H,G

(d) How many topological orderings does this graph have?

Solution: The topological ordering can either begin with B or A. Likewise, after processing C, we can process D or E first. After processing F, we can process G or H first. Therefore, we have $2 * 2 * 2 = 8$ possible topological orderings for the given graph.

B— Run the strongly connected components algorithm on the following directed graphs G. When doing DFS on G^R: whenever there is a choice of vertices to explore, always pick the one that is alphabetically first.





In case (i) answer the following questions.

(a) In what order are the strongly connected components (SCCs) found?

Solution: The SCCs are found in the following order: {CDFJ}, {HGI}, {A}, {E}, {B}.

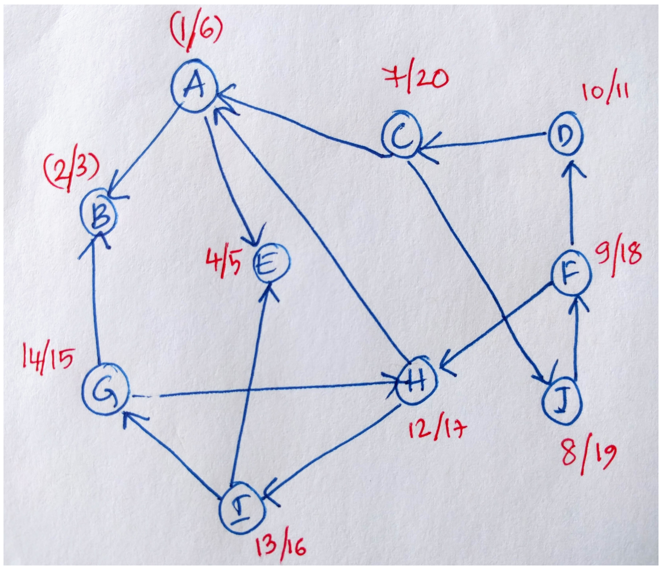


Figure 2: pre/post numbers are indicated in red for DFS on G^R

(b) Which are source SCCs and which are sink SCCs?

Solution: The source SCCs are {B} and {E}. The only sink SCC is {CDFJ}.

(c) Draw the "metagraph" (each meta-node is an SCC of G).

Solution:

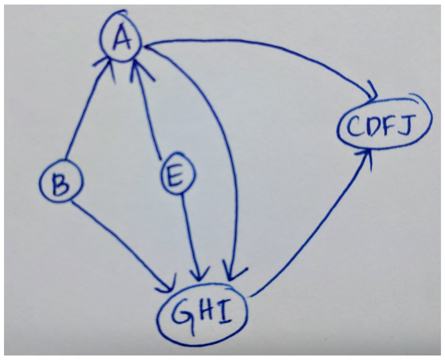


Figure 3: Topological ordering for the given graph

(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

Solution: You must add 2 edges at minimum. {C, D, F, J} to {E} and {E} to {B}. The edge can be added to any node in the SCC. Because both E and B have edges to {A} and {H, I, G} we only need to connect {B}, {E}, and {C, D, F, J} to create a strongly connected graph. We can check this by running explore() from any node, and we should be able to reach every other node.

In case (ii) answer the following questions.

(a) In what order are the strongly connected components (SCCs) found?

Solution: The SCCs are found in the following order: {DGHIF}, {C}, {AEB}.

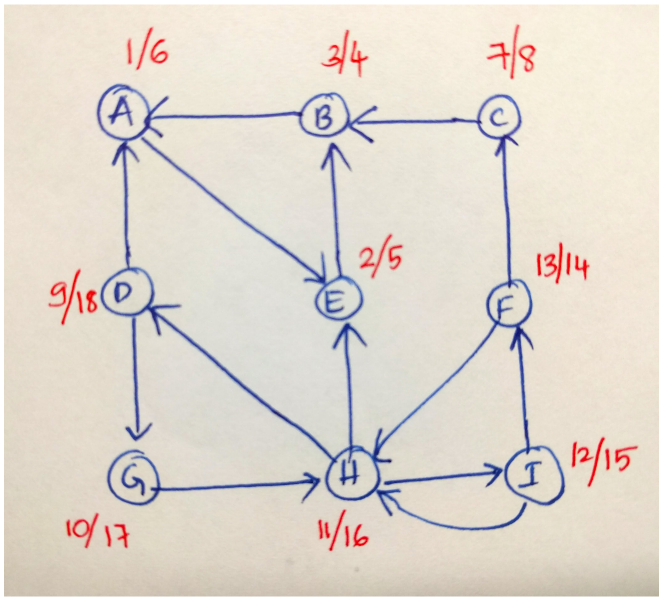


Figure 4: pre/post numbers are indicated in red for DFS on G^R

(b) Which are source SCCs and which are sink SCCs?

Solution: The only source SCC is are {AEB}. The only sink SCC is {DGHIF}.

(c) Draw the "metagraph" (each meta-node is an SCC of G).

Solution:

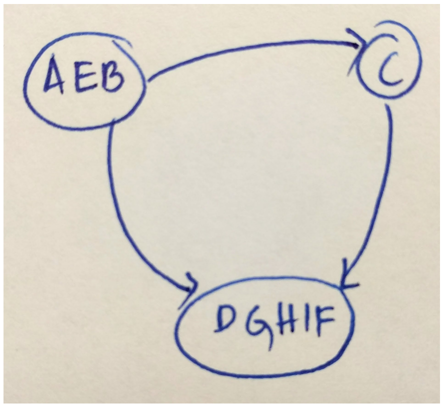


Figure 5: Topological ordering for the given graph

(d) What is the minimum number of edges you must add to this graph to make it strongly connected?

Solution: You must add 1 edges at minimum, from sink to source to make it strongly connected.

Problem 3. (DPV 3.7) Bipartite Graph

A bipartite graph is a graph $G = (V, E)$ whose vertices can be partitioned into two sets ($V = V1 \cup V2$ and $V1 \cap V2 = \emptyset$) such that there are no edges between vertices in the same set (for instance, if $u, v \in V1$, then there is no edge between u and v).

1. Give a linear-time algorithm to determine whether an undirected graph is bipartite.

Solution:
High Level Description:
Our algorithm will recursively partition the vertices of G into two sets and check that they maintain the two-coloring property of a bipartite graph as it proceeds. If a vertex is found that violates the coloring property we immediately halt and return false. If we completely explore the graph without finding such a violation we return true.

Let {-1, 1} be the 2 colors. Then **visited[u]** will be 0 if u has not been visited yet, 1 if u has been colored with the color 1 and -1 if it has been with the color -1.

Pseudo code:

```
Input: graph G = (V, E)
Output: true if the graph G is bipartite, false otherwise
procedure isBipartite(G):
  for each v in V:
    visited[v] = 0

  for each v in V:
    if explore(G, v, 1) == false:
      return false

  return true
end

procedure explore(G, v, color):
  visited[v] = color

  for each edge (v, u) in E:
    if visited[u] <> 0 && visited[u] + color <> 0:
       return false
    else
       explore(G, u, color * (-1))
end
```

Running time:
Our algorithm mimics DFS on a graph with a constant amount of additional work to check for a valid coloring of the graph for each edge. Therefore the running time of our algorithm is the same as DFS, $\mathcal{O}(|V| + |E|)$.

Proof of correctness:
A graph is bipartite if and only if a valid two-coloring of the graph exists. Clearly, our algorithm will find a two-coloring if one exists since any alternating of the coloring of connected vertices will be a valid two-coloring of a bipartite graph.

Now, consider the case when a valid two-coloring does not exist (i.e. the graph is not bipartite). Then, there must exist some cycle such that node u of the cycle is colored with one color of {1, -1} and node v is colored with the other color and there is an edge connecting v and u. When our algorithm reaches node v it will check the edge connecting v and u, see the conflict and return false.

2. There are many other ways to formulate this property. For instance, an undirected graph is bipartite if and only if it can be colored with just two colors.
   Prove the following formulation: an undirected graph is bipartite if and only if it contains no cycles of odd length.

Solution:

- First, we will show that an undirected graph is bipartite if it contains no cycles of odd length. Let $G = (V, E)$ be a graph with no odd length cycles. Consider a run of DFS on G. For every edge $e = (u, v) \in E$, e is either a tree edge or it is a back edge. If e is a DFS tree edge then the bipartite property will be maintained since we can alternate colors along the DFS search tree. If e is a back edge then there is a cycle with an even number of vertices in G. Therefore u and v must be an odd number of edges away in the DFS search tree and thus colored with different colors. So, our graph is successfully two-colored and therefore bipartite.
- Second, we will show that an undirected graph contains no cycles of odd length if it is bipartite. Let $G = (V, E)$ be a bipartite graph and suppose to the contrary that G has at least one cycle of odd length. Pick some node u along this cycle. Let node v be the last node that we reach along the cycle before returning to u. If we start coloring u with color 1 and alternate along the cycle we will also color v with color 1 since the cycle is odd in length. Thus, we cannot have a valid two-coloring of G and this contradicts our assumption that G is bipartite. Therefore, G cannot contain a cycle of odd length.

3. At most how many colors are needed to color in an undirected graph with exactly *one* odd-length cycle?

Solution: 3
If a graph has exactly one odd cycle, it can be colored by 3 colors. To obtain a 3-coloring, delete one edge from the odd cycle. The resulting graph has no odd cycles and can be 2-colored. We now add back the deleted edge and assign a new (third) color to one of its end points.

Problem 4. (DPV 3.10) DFS Using Stack

Rewrite the *explore* procedure (DPV Figure 3.3) so that it is non-recursive (that is, explicitly use a stack). The calls to *previsit* and *postvisit* should be positioned so that they have the same effect as in the recursive procedure.

Solution:

```
explore (Graph g, Vertex v)
  visited(v) = true
  previsit(v)
  Stack s
  s.push(v)
  while(s.size() > 0)
    v' = s.top()
    flag = 0
    for each edge (v',u) in E:
      if not visited(u):
        previsit(u)
        visited(u) = true
        s.push(u)
        flag = 1
        break from for loop
    if flag == 0:  // all neighbors visited
      stack.pop()
      postvisit(v')
```

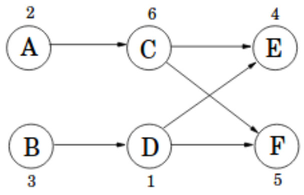
We use an explicit stack S to handle the visit to the vertices in a graph. We first push a vertex to the stack to mark it as visited. We observe that we pop the vertex from the stack after all of its neighbors (i.e. successors) have been visited and popped from the stack. Therefore, our non-recursive algorithm behaves identical to the recursive *explore* procedure.

Problem 5. (DPV 3.25) Minimize Cost

You are given a directed graph in which each node $u \in V$ has an associated *price* p_u, which is a positive integer. Define the array *cost* as follows: for each $u \in V$,

$$cost[u] = \text{price of the cheapest node reachable from } u \text{ (including } u \text{ itself)}$$

For instance, in the graph below (with the prices shown for each vertex), the *cost* value of the nodes A, B, C, D, E, F are 2, 1, 4, 1, 4, 5 respectively.

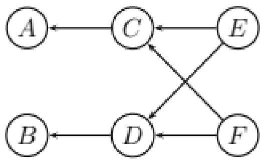


Your goal is to design an algorithm that fills the entire *cost* array (i.e. for all vertices).

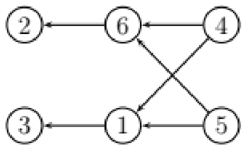
(a) Give a linear time algorithm that works for directed *acyclic* graphs. (*Hint*: Handle the vertices in a particular order).

Solution: Our algorithm makes use of a simple property of a directed graph: To find which nodes can reach a certain target node v we can run DFS from v on G^R, the reversed graph of G (the same graph as G but with all the edges reversed). All explored in this DFS run will be nodes with a path to v in G. The outline of our algorithm is as follows:

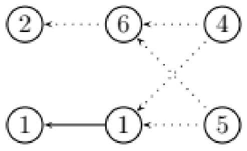
- Generate G^R from G. Here is how the graph given in example in the text would be inverted:



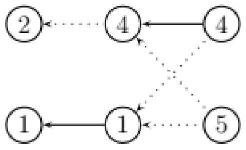
- Relabel nodes according to their cost. Here is how the previous graph would be relabeled:



- Run DFS on the reversed graph. Note that D comes first now in our example since we have relabeled all the nodes.



- When we start DFS on some node u we store the cost of that node. As we build our DFS search tree, the entire tree is labeled with the cost of u. This happens until we cannot reach any more nodes from u. On the example, starting by 1, we can only reach 3 and we relabel it with 1:



- Now we move on to the next unexplored node v and store the cost of that node. We repeat the previous step for the entire DFS tree starting from v. This continues until DFS terminates, and we have stored costs for all nodes. Here is what we would have at the end of the algorithm for the graph given:

Pseudocode:

```
procedure find_costs(G, s, t):
   Let R be the reversed graph of G
   Relabel nodes in R with their costs
   Run DFS on R as follows
end

procedure dfs(G)
   for all v in V:
      visited(v) = false
   for all v in V:
      if not visited(v): explore(v, v.cost)
   end

procedure explore(v, cost):
   visited(v) = true
   v.cost = cost
   for each edge (v, u) in E:
     if not visited(u): explore(u, cost)
end
```

Running time:
Given our edges are specified in an adjacency list format, we can reverse each list in linear time and so generate G^R in linear time. We then run a standard DFS but instead of *pre* and *post* labels we use a variable cost label as described above. Thus, the running time of our algorithm is the same as DFS: $O(|V| + |E|)$.

Proof of correctness:
Consider a run of DFS on the reversed graph. Since we are starting with node u of the lowest cost and visiting all nodes that have a path to u, we will label each node with the lowest possible cost. Since we explore all DFS trees starting with the lowest cost node in that tree, we will propagate up the lowest possible costs.

(b) Extend this to a linear time algorithm that works for all directed graphs. (*Hint*: Recall the "two-tiered" structure of directed graphs).

Solution: For part (b) we use the property of directed graphs that they are DAGs of their strongly connected components. Given this, the algorithm is very simple: Construct the meta-graph of SCCs of G, label each node in the meta-graph with the cheapest cost of any node in the represented SCC. Now, we can run our algorithm from part (a) on the meta-graph. We have seen from Chapter 3, Section 3.4.2 that we can create a graph of SCCs of G in linear time. Therefore, our algorithms running time is still linear as with part (a).

Problem 6. (DPV 3.23) Paths in a DAG

Give an efficient algorithm that takes as input a directed acyclic graph $G = (V, E)$ and two vertices $s, t \in V$ and outputs the number of different directed paths from s to t in G.

Solution:
The main idea behind the solution is that the number of $s - t$ paths in a DAG is equal to the sum of the different s-t paths from the children of s to t. This is critical in building a recurrence relation used to count the total number of simple paths from u to v for any two vertices $u, v \in G$. The recurrence relation can be defined as:

$$Paths(u) = \begin{cases} 0 & \text{if } u \text{ is a leaf node} \\ 1 & \text{if } u == b \\ \sum\limits_{(u,v) \in E} Paths(v) & \text{otherwise} \end{cases} \tag{1}$$

Pseudo-Code:

```
procedure pathCount(s, t):
  INPUT: Source and destination vertices s and t
  OUTPUT: Number of simple paths from s to t
  We will keep an array numpath[] which holds the number of simple paths
  from each node to t in G.
  Set numpath[u] = -1 when the number of simple paths from u has not been
  revealed
  for all u in V :
     numpath[u] = -1
     numpath[t] = 1
  return pathCountHelper(s, numpath)

procedure pathCountHelper(u, numpath)
  INPUT: Source vertex u, array numpath which holds the number of
    simple paths from each node to t in G
  OUTPUT: Number of simple paths from u to t
  if numpath[u] < 0:
      numpath[u] = 0
  for all (u,v) in E:
      numpath[u] = numpath[u] + pathCountHelper(v, numpath)
  return numpath[u]
```

Proof by induction:
Proof. We will give a proof by induction over the longest $s - t$ path length. A longest path length is well defined because the given graph is a DAG and hence contains no cycles.

Base case: Consider the case where no paths exist from s to t, i.e., t is not reachable from s. In this case, the algorithm, beginning at s, recurses over all the children of s, then the children of children of s and so on eventually covering all nodes reachable from s, summing up the paths from each of these nodes to t along the course of its operation. However, as t is not reachable from s, the algorithm correctly terminates with a total path count of 0 from s to t.

Consider the case where there exists only one $s - t$ path, which is of length one, i.e. s and t are connected by an edge (s, t), and t is not reachable from any other child of s. In this case, the algorithm returns 1 according to the case $(u == v)$ of the recursion, and all other children of s return 0 as proved above.

Inductive hypothesis Let us assume that the algorithm returns the correct number of $s - t$ paths in a graph where the longest $s - t$ path is of length k.

Inductive step We prove that the algorithm returns the correct number of $s - t$ paths for a graph where the longest $s - t$ path is of length $k + 1$: In this case, the longest path from any of the children of s to t is of length k. Any path from s to t must go through one of the children of s thus by induction hypothesis, all paths will be counted. Now suppose that some path is counted twice, i.e., two children u, V of s return the same path. However, this cannot be true since the paths split to u and v after starting from s and are thus distinct. Thus, all paths are counted exactly once. Thus we would correctly count all paths from s to t when the longest $s - t$ path has length $k + 1$.

Time complexity
The initial processing on the nodes takes $\mathcal{O}(|V|)$ time. In the recursion, a particular node v is only updated k times, where k is the in-degree of the node v. Thus, the total running time of the $pathCountHelper$ function is the sum of the in-degrees over all nodes. This adds $\mathcal{O}(|E|)$ operations to the running time. Thus, the total running time of the algorithm is $\mathcal{O}(|V| + |E|)$.

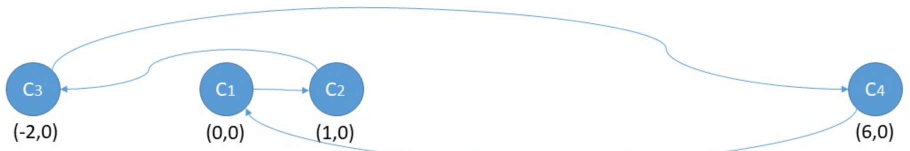
Problem 7. A Greedy Heuristic for the Traveling Salesperson Problem

You are given a set of n points $(x_i,\, y_i)$ for $i = 1, 2, \dots, n$ respectively corresponding to cities c_1, c_2, c_3, $\dots$, c_n in the Euclidean plane. The travel distance between any pair of cities is the Euclidean distance between the corresponding points. City c_1 located at $(x_1,\, y_1)$ is your *home* city.
The Traveling Salesperson Problem (TSP) seeks to find a Hamiltonian cycle, or "tour", over the n cities (i.e., a closed path that begins and ends at c_1, and that visits every other city exactly once) that has minimum total travel distance.
Consider the following "greedy" approach to finding a minimum-cost tour over the n cities:

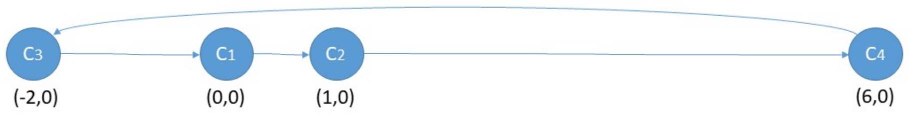
- Initialize all cities to "unvisited".
- Set *current_city* $= c_1$; mark c_1 as "visited".
- For $j = 1, \dots, n-1$:
  - Travel from *current_city* to the nearest city, *city_next*, that is "unvisited".
  - Mark *city_next* as "visited".
  - Set *current_city* $=$ *city_next*
- Travel from *current_city* to c_1.

1. Show that this "greedy" approach to TSP is not optimal by exhibiting a small counterexample.

Solution: Figure (a) shows the greedy solution will construct a tour that has cost 18. Figure (b) shows that the optimal tour has cost 16. Therefore, the greedy solution is not optimal.



(a)



(b)

2. How badly suboptimal do you think the greedy approach can be, relative to optimal? Please clearly explain your definition of "badly suboptimal".

We define Suboptimality as ratio between the worst possible greedy cost (NN) and optimal cost (OPT)

(Refer to slide 33 in Lecture 2 slides for a related version of Suboptimality and the a discussion of TSP approximation), defined as,

$$NN = max_{all\ possible\ greedy\ tours}(\sum_{i=1}^{n} Distance(g_i, g_{i+1}))$$

$$OPT = \sum_{i=1}^{n} Distance(o_i, o_{i+1})$$

where $g_1, g_2, g_3, \cdots g_{n+1}$ is the tour (sequence of cities) chosen by the Greedy algorithm (with $g_{n+1} = g_1$) and $o_1, o_2, o_3, \cdots o_{n+1}$ is the optimal tour (sequence of cities, with $o_{n+1} = o_1$) and $Distance(c_i, c_{i+1})$ is the Euclidean distance between cities c_i and c_{i+1}.

$$Suboptimality = \frac{NN}{OPT}$$

Note: In the context of this question we are more interested in the asymptotic growth of the above defined Suboptimality rather than the exact value.

As mentioned in the FAQ for HW1, it's actually known that the NN heuristic can be forced to take up to a $\Theta(\log n)$ factor more tour cost than the optimal solution, in Euclidean geometry. Essentially, it's possible to construct a pointset that forces NN to keep wasting distance by – informally – 'reversing direction back over itself'. (It's perfectly fine to stop reading here... :-).)

The following is adapted from a paper by Hougardy and Wilde: `https://pdfs.semanticscholar.org/24fa/c595d92569643972bd20a259805505e565ee.pdf`.

Consider the following construction of a family of metric TSP instances G_k on which the nearest neighbor rule (NNR) yields tours that are much longer than optimum tours. As the set of cities V_k we take the points of a 2x$(8 \cdot 2^k - 3)$ subgrid of $\mathbb{Z}^2$. Thus we have $|V_k| = 16 \cdot 2^k - 6$. Let G_k be any TSP-instance defined on the cities V_k. The graph for G_0 is shown in the figure below.

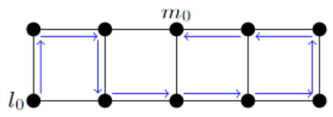


Fig. 1. The graph defining the graphic TSP G_0 together with a partial NNR tour that connects l_0 with m_0.

We label the lower left vertex in V_k as l_k and the top middle vertex in V_k as m_k.

We can prove by induction on k that the nearest neighbor rule can find a rather long tour in G_k. For this we need to prove the following slightly more general result.

Lemma 1. *Let the cities of G_k be embedded into G_m with $m > k$. Then there exists a partial NNR tour in G_m that*

- *visits exactly the cities in G_k,*
- *starts in l_k and ends in m_k, and*
- *has length exactly $(12 + 4k) \cdot 2^k - 3$.*

Proof. We use induction on k to prove the statement. For $k = 0$ a partial NNR tour of length $12 \cdot 2^0 - 3 = 9$ that satisfies is shown in Figure 1. Now assume we already have defined a partial NNR tour for G_k. Then we define a partial NNR tour for G_{k+1} recursively as follows. As

$|V_{k+1}| = 16 \cdot 2^{k+1} - 6 = 2 \cdot (16 \cdot 2^k - 6) + 6 = 2 \cdot |V_k| + 6$ we can think of G_{k+1} to be the disjoint union of two copies $G^{‘}_k$ and $G^{“}_k$ of G_k separated by a 2x3 grid. This is shown in Figure 2.

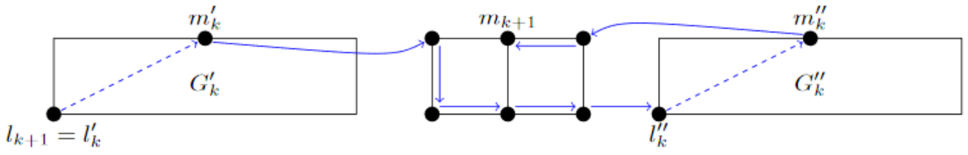


Fig. 2. The recursive construction of a partial NNR tour for the instance G_{k+1}. The dashed lines indicate partial NNR tours in G'_k and G''_k.

Now we can construct a partial NNR tour for G_{k+1} as follows. Start in vertex $l^{‘}_k$ and follow the partial NNR tour in $G^{‘}_k$ that ends in vertex $m^{‘}_k$. The leftmost top vertex of the 2x3-grid is now a closest neighbor of $m^{‘}_k$ that has not been visited so far. Go to this vertex, visit some of the vertices of the 2x3-grid as indicated in Figure 2, and then go to vertex $l^{‘‘}_k$. From this vertex follow the partial NNR tour in $G“_k$ which ends in vertex $m“_k$. A nearest neighbor for this vertex now is the rightmost top vertex of the 2x3-grid. Continue with this vertex and go to the left to reach vertex m_{k+1}. This partial NNR tour is also a partial NNR tour when G_{k+1} is embedded into some G_l for $l > k+1$.

The length of this partial NNR tour is twice the length of the partial NNR tour in G_k plus five edges of length 1 plus two edges of length $\frac{1}{2}(\frac{1}{2} \cdot |V_k| + 1)$. Thus we get a total length of

$$2((12 + 4k) \cdot 2^k - 3) + 5 + 8 \cdot 2^k - 2 = (12 + 4(k+1)) \cdot 2^{k+1} - 3$$

Theorem 1. For the family of TSP instances we defined, the nearest neighbor rule is no better than $\Omega(\log n)$.

Proof. The instance G_k defined above has $n = 16 \cdot 2^k - 6$ cities and an optimum TSP tour in G_k has length n. As shown in Lemma 1 there exists a partial NNR tour in G_k of length at least $(12+4k) \cdot 2^k - 3$. Thus the approximation ratio of the nearest neighbor rule is no better than

$$\frac{(12 + 4k) \cdot 2^k - 3}{16 \cdot 2^k - 6} \geq \frac{12 + 4k}{16} = \frac{3 + 4k}{4} = \frac{3 + \log \frac{n+6}{16}}{4} = \Omega(\log n)$$

NOTE: The above proof addresses the case when the final destination city is not the same as the start city. However, even for that case, we just need to add a distance term to both the numerator and the denominator of the suboptimality ratio. Knowing the fact that the extra term added to the numerator will be greater or equal to that added to the denominator (which is 1 for the family of TSP instance we defined), the $\Omega()$ bound will still hold true.

Again, if you'd like to dig more deeply into this question, you can look up the paper: `https://pdfs.semanticscholar.org/24fa/c595d92569643972bd20a259805505e565ee.pdf` and `http://epubs.siam.org/doi/pdf/10.1137/0206041`