Question 1

Let $f(n)$ and $g(n)$ be asymptotically positive functions. Prove or disprove each of the following conjectures.

1. **$f(n) = O(g(n))$ implies $g(n) = O(f(n))$.**
2. **$f(n) + g(n) = \Theta(min(f(n), g(n)))$.**
3. **$f(n) = O(g(n))$ implies $lg(f(n)) = O(lg(g(n)))$, where $lg(g(n)) \geq 1$ and $f(n) \geq 1$ for all sufficiently large n.**
4. **$f(n) = O(g(n))$ implies $2^{f(n)} = O(2^{g(n)})$.**
5. **$f(n) = O((f(n))^2)$.**
6. **$f(n) = O(g(n))$ implies $g(n) = \Omega(f(n))$.**
7. **$f(n) = \Theta(f(n/2))$.**

1. **False**. Given an example that $f(n) = n$ and $g(n) = n^2$, we have $f(n) = O(g(n))$ but $g(n) \neq O(f(n))$.

2. **False**. Given an instance that $f(n) = n$ and $g(n) = n^2$, we have $f(n) + g(n) = n^2 + n$. Because $\Theta(min(f(n), g(n))) = \Theta(n)$, we can conclude that $f(n) + g(n) \neq \Theta(min(f(n), g(n)))$.

3. **True**. Due to the condition that $f(n) = O(g(n))$, we can find a positive number N and C_1 satisfy that when $n > N$, $f(n) \leq C_1 \cdot g(n)$. Then we have $lg(f(n)) \leq lg(C_1 \cdot g(n)) = lg(C_1) + lg(g(n)) \leq C_2 \cdot lg(g(n))$, where C_2 is a positive number and $C_2 > 1$. Therefore, we have $lg(f(n)) = O(lg(g(n)))$.

4. **False**. Given an example that $f(n) = 2log(n)$ and $g(n) = log(n)$, we have $f(n) = O(g(n))$. $2^{f(n)} = n^2$, $2^{g(n)} = n$. Because $n^2 \neq O(n)$, we can conclude that $2^{f(n)} \neq O(2^{g(n)})$.

5. **False**. Suppose that $f(n) = \frac{1}{n}$, then $(f(n))^2 = \frac{1}{n^2}$. Because $\frac{1}{n} \neq O(\frac{1}{n^2})$, we can conclude that $f(n) \neq O((f(n))^2)$.

6. **True**. Due to the condition that $f(n) = O(g(n))$, we can find a positive number N and C_1 satisfy that when $n > N$, $f(n) \leq C_1 \cdot g(n)$. That is we can find a positive number $C_2 = \frac{1}{C_1}$ satisfy that when $n > N$, $g(n) \geq C_2 \cdot g(n)$. Therefore we have $g(n) = \Omega(f(n))$.

7. **False**. Suppose that $f(n) = 2^{2n}$, then we have $f(n/2) = 2^n$. It is obviously that $2^{2n} \neq O(2^n)$. That is $f(n) \neq \Theta(f(n/2))$.

Question 2

Show that when all elements are distinct, the best-case running time of HEAP-SORT is $\Omega(nlgn)$

We first prove the lower bound of sorting-based comparison algorithm is $\Omega(nlogn)$.

We can construct a decision-making tree whose node represents a comparison between two elements. Just like the instance showed in figure 2.1， let k be the number of layer of the total decision-making tree. Notice that there are $n!$ kinds of orders and the number of nodes is 2^k, so we have $2^h \geq n!$.

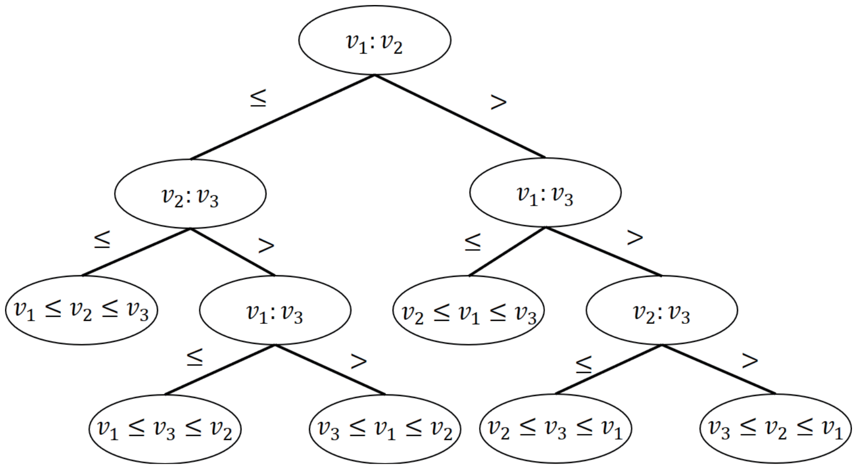


Figure 2.1 An instance of decision-making tree

Then we can get $h \geq log(n!)$.

According to Stirling's approximation, we have:

$$n! = \sqrt{2\pi n} \cdot (\frac{n}{e})^n \cdot (1 + \Theta(\frac{1}{n}))$$

And we can get:

$$h \geq \frac{1}{2}log(2\pi) + \frac{1}{2}log(n) + nlog(n) - nlog(e) + log(1 + \Theta(\frac{1}{n}))$$

where

$$\Theta(\frac{1}{n}) \in [e^{\frac{1}{12n+1}}, e^{\frac{1}{12n}}]$$

So we get the asymptotical lower bound is $\Omega(nlog(n))$.

Because HEAPSORT is one kind of sorting-based comparison algorithm. So we can easily get the best-case running time of HEAPSORT is $\Omega(nlgn)$.

Question 3

Give an algorithm to detect whether a give undirected graph contains a cycle. If the graph contains a cycle, then your algorithm should output one. (It should not output all cycles in the graph, just one of them.) The running time of your algorithm should be $O(m + n)$ for a graph with n nodes and m edges.

We design the algorithm based on depth-first traversal. As shown in Alg 3.0.1,we mark the difference between our cycle detected algorithm and DFS algorithm as blue. It is clearly that, when we find a repeatedly accessing vertex, there is a cycle in the graph.

Algorithm 3.0.1 Cycle Detected

```
Input: $s$
 1: Initialize $S$ to be a stack with one elements $s$
 2: while $S \neq \phi$ do
 3:     Take a node $u$ from $S$
 4:     if Explored[$u$]=$false$ then
 5:         Set Explored[$u$]=$true$
 6:         for each edge($u, v$) incident to $u$ do
 7:             Add $v$ to the stack $S$
 8:         end for
 9:     else
10:         Return TRUE
11:     end if
12: end while
13: Return $FALSE$
```

Because we only modify one step in the "if" judgement in the while cycle, it won't affect the time complexity of the original DFS time. Therefore, the time complexity is $O(m + n)$.

Question 4

We have a connected graph $G = (V, E)$, and a specific vertex $u \in V$. Suppose we compute a depth-first search tree rooted at u, and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u, and obtain the same tree T. Prove that $G = T$. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

Consider an example in figure 4.1. The tree T in figure 4.1 is the depth-first and breadth-first tree of ten vertex. We firstly focus on breadth-first search tree. We note two

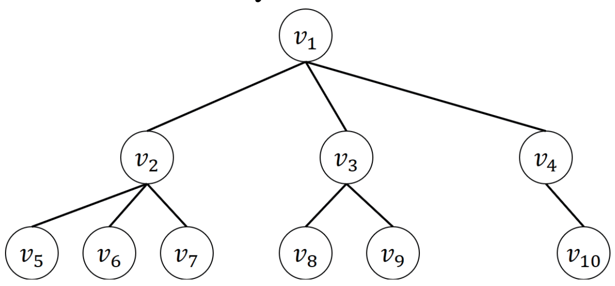
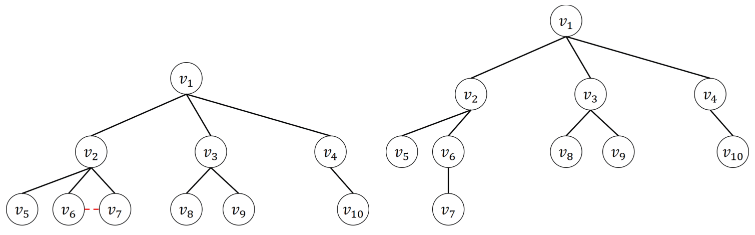


Figure 4.1 depth-first and breadth-first tree

vertexes in the same layer(BFS search tree) and satisfy v_i is before v_j as $v_i \prec v_j$. We note prt(v) as the parent node of v. Supposed that there is an edge e which is not belong to T, then e cannot across two or more layers in T. And e cannot be (prt(u), v) satisfying prt(u) $\neq$ prt(v) and $u \prec v$. Then e must be the edge of the two following cases:

1) e is between two vertexes v_i and v_j satisfying prt(v_i)=prt(v_j). As show in figure 4.2, in this case, the DFS tree is different from the BFS tree because v_7 will be access as the son of v_6 not the son of v_2 in the BFS tree.

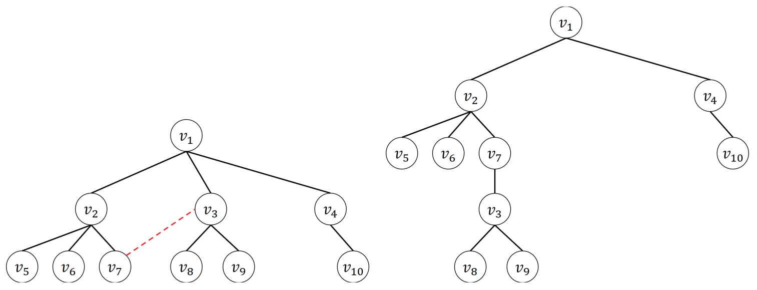


(a) BFS (b) DFS

Figure 4.2 e is between two vertexes v_i and v_j satisfying prt(v_i)=prt(v_j)

2) e is between two vertexes v_i and v_j satisfying prt(v_i) $\neq$ prt(v_j) and $v_j \prec v_i$. As show in figure 4.3, in this case, the DFS tree is different from the BFS tree because v_3 will be

access as the son of v_7 not the son of v_1 in the BFS tree.



(a) BFS (b) DFS

Figure 4.3 e is between two vertexes v_i and v_j satisfying $\text{prt}(v_i) \neq \text{prt}(v_j)$ and $v_j \prec v_i$

Question 5

There's a natural intuition that two nodes that are far apart in a communication network ——separated by many hops ——have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.

Suppose that an n-node undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all $s - t$ paths.(In other words, the graph obtained from G by deleting v contains no path from s to t.) Give an algorithm with running time $O(m + n)$ to find such a node v.

According to BFS algorithm and the given conditions, as shown in figure **??**, there exists an separated layers satisfying that the count of layers is m and $m - 1 > n/2$. The problem is equal to Pigeon nest problem. Notice that there remain at most $n - 2$ vertexes which will be put into $m - 1$ cells. And notice that $2(m - 1) > n > n - 2$. So there must be at least one cell(actually two cells) that contains only one vertex. That is to say there exists some node v that satisfying our goal.

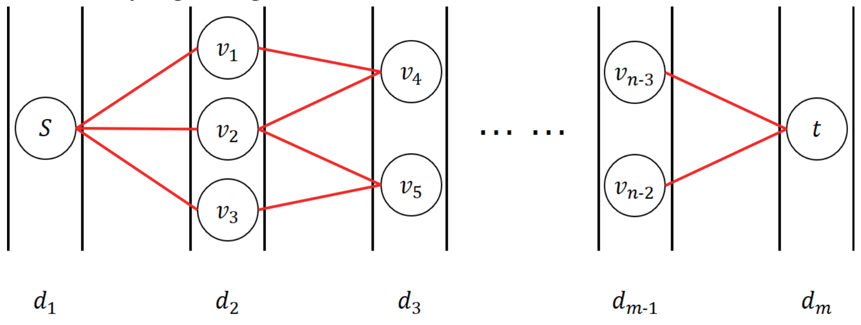


Figure 5.1 The layers of breadth-first search

We next give an algorithm to find one of these vertex. Generally speaking, we just need to execute depth-first traversal rooted at S and get tree T. Because s and t are connected, t is definitely in T. Besides, it is sure that there is at leat one layer that contains one vertex. So we just need to find this vertex. We give the detail process in Alg 5.0.2.

As showed in Alg 5.0.2, we test the number of each layer(line 6). When the count of layer number is one and it is not the first layer, we set the result node as the single element in this layer and break the loop.

Algorithm 5.0.2 The pivotal vertex discovering

```
Input: $s$
1:  Set Discovered[$s$]=$true$ and Discovered[$v$]=$false$ for all other $v$
2:  Initialize $L[0]$ to consist of the single element $s$
3:  Set the layer count $i$=0
4:  Set result node rnode = $null$
5:  while $L[i] \neq \phi$ do
6:    if Size($L[i]$) = 1 and $i > 0$ then
7:      Set rnode = $L[i][0]$
8:      break
9:    end if
10:   Initialize an empty list L[$i + 1$]
11:   for each node $u \in L[i]$ do
12:     Consider each edge $(u, v)$ incident to $u$
13:     if Discovered[$v$]=$false$ then
14:       Set Discovered[$v$]=$true$
15:       Add $v$ to the list $L[i + 1]$
16:     end if
17:   end for
18:   Increment the layer counter $i$ by one
19: end while
20: Return rnode
```