ADA homework 4 b03902089 林良翰

Problem 1

1.1 Find a tree with *pre-order* and *post-order* :

```
function buildTree(*node, *pre, *post)
	add pre[0] to the node
	for prec = 1 to maxPre
		ancestor = findAncestor(post[postc])
		for postc = maxPost to 0
			if post[postc] == pre[prec]
				if left is null
					add to ancestor's left, node= ancestor's node->left
				else
					add to ancestor's right,node= ancestor's node->right
				end if
				break
			else if post[postc] appear in the current tree
				ancestor = findAncestor(post[postc])
			end if
		end for
	end for
end function
```

1.2 Find a tree with *pre-order* and *in-order* :

```
start = 0, end = maxPre
function buildTree(*node, *pre, *in, start, end)
	if start > end
		return null
	else if start == end
		node->value = in[start]
		return node
	end if
	for prec = 0 to maxPre
		for inc = start to end
			if in[inc] == pre[prec]
				node->value = in[inc]
				node->left = buildTree(node->left, pre, in, start, inc-1)
				node->right = buildTree(node->right, pre, in, inc+1, end)
				break
			end if
		end for
	end for
end function
```

1.3 Find a tree with *post-order* and *in-order* :

```
start = 0, end = maxPost
function buildTree(*node, *post, *in, start, end)
	if start > end
		return null
	else if start == end
		node->value = in[start]
```

```
            return node
      end if
      for postc = maxPost to 0 // find root
            for inc = start to end
                  if in[inc] == post[postc]
                        node->value = in[inc]
                        node->left= buildTree(node->left, post, in, start, inc-1)
                        node->right= buildTree(node->right, post, in, inc+1, end)
                        break
                  end if
            end for
      end for
end function
```

problem 2

2.1

Divide the numbers into two groups :

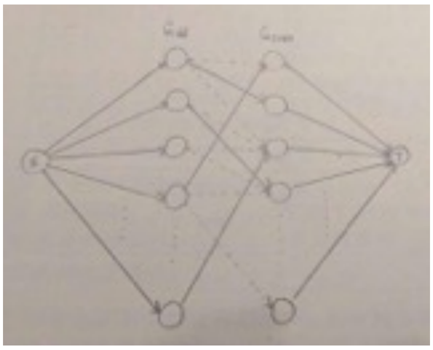
$$\textit{transfer into binary form}\begin{cases} G_{odd} & \textit{have odd ones} \\ G_{even} & \textit{have even ones} \end{cases}$$

```
function separate(*odd, *even, *num)
      for i = 0 to maxNum
            n = transfer num[i] into binary form
            if n have odd ones
                  add to odd array
            else
                  add to even array
            end if
      end for
end function
```

2.2 reference: Wikipedia (*Edmonds-Karp* algorithm)

Edmonds-Karp algorithm is a method for computing the maximum flow of a flow network, which has V vertices and E edges. It will first find the shortest path with available capacity, and make at least one of the edges of the path saturated with the flow, and then find another shortest path with available capacity repeatedly until there is no available path. Finally the result is the maximum flow of the flow network.
To find a path, we need $O(E)$ time. In this path we have one of $O(E)$ edges to be saturated, beside we have the path length $O(V)$. As a result, the complexity will be $O(VE^2)$.

2.3 reference: 簡瑋德



⇒ Construct the two groups to be a graph with a source and a sink, which is a network flow.
⇒ The edges = $\{e \mid v_{odd}, v_{even}$ which can be eliminated by the magic power at once and its weight is $1\}$
⇒ Use *Edmonds-Karp* algorithm to find the maximum flow of the graph, which is the value f.
⇒ Count the number of vertices r which are not connected between two groups G_{odd} and G_{even}.
⇒ The minimum number of the magic power we used $= f + r$.

Problem 3

3.1

According to the condition, G is a *con-word* graph which $\mu_*(G) = 0$

Suppose that G have negative cycles, then $\mu_*(G) < 0$, contradict !

Thus, G must not have any negative cycles.

3.2.(a)

If we suppose that we have a path s to v with length $\geq N$ (N is the number of edges)

Then we must create a cycle c in the path. the mean weight of the cycle $\mu(c) \geq 0$ ($\because$ G is con-word)

Because the the weight of the cycle larger than 0 will enlarge the weight of the path,

we may avoid creating a cycle when finding a shortest path in G.

Thus, we will have a shortest path from s to v with at most $N-1$ edges.

3.2.(b)

According to problem 3.2.(a), $d_N(v)$ is not the real shortest path from s to v, $d_k(v)$ is the real shortest path.

Thus :

$$\max_{1 \leq k \leq N-1} \frac{d_N(v) - d_k(v)}{N-k} \geq 0$$

3.3.(a) reference: 簡瑋德

$d(u) + w(e) \geq d(v)$, (s to u), e is one of a path from s to v

$d(v) - w(e) \geq d(u)$, (s to v), *other* is one of a path from s to u

$\Rightarrow d(u) \geq d(v) - w(e) \geq d(u) \Rightarrow d(v) - w(e) = d(u) \Rightarrow d(v) = d(u) + w(e)$

3.3.(b) reference: 簡瑋德

There exist vertex such that : $d_k(v) = d(v)$

Set e to be the edge connect from v to u

$d(u) = d(v) + w(e) = d_{k+1}(u)$ exist, which can proved by problem 3.3.(a)

Thus, d(v') = $d(v) + w(e) + \ldots = d_{N-|c|}(v')$ exist

d(v') = $d_{N-|c|}(v') = d_{N-|c|}(v') + w(e_1) + w(e_2) + \ldots w(e_c) = d_N(v')$, which $e_1 \sim e_c$ are edges in c

3.3.(c) Proved directly by 3.3.(a) and 3.3.(b)

3.4.(a)

pf : p exist at least one cycle

If path p have no cycles with N edges, then we must have N+1 distinct vertices to be connected. But we only have N distinct vertices in $G \Rightarrow$ contradict ! Thus, we can assert that p must have at least one cycle.

pf : $d_N(v) - d_{N-l}(v) \geq W$

If path p have c cycles ($c \geq 1$, proved above), we can define $d_{N-c\times l}(v)$ is the shortest path without cycles.

$\Rightarrow d_{N-c\times l}(v) + cW = d_N(v)$

$\Rightarrow cW = d_N(v) - d_N(v)$

$\Rightarrow d_N(v) - d_{N-l}(v) \geq W$

3.4.(b)

Let $k = N-l$, we have known : $\max_{0 \leq k \leq N-1} \frac{d_N(v) - d_k(v)}{N-k} = 0$

By problem 3.4.(a)

$\Rightarrow d_N(v) - d_{N-l}(v) = d_N(v) - d_k(v) = 0 \geq W$

$\Rightarrow G$ is *con-word* graph, so there isn't exist *negative-weight* cycle

$\Rightarrow W = 0$, thus path p doesn't contain any *positive-weight* cycle.

3.5.(b) reference: 簡瑋德

```
dk[N][M] = {0} // N vertices, at most length M
d[N] = {0} // shortest length from s to v
function minCycle(*vertex, *edge, **dk, *d)
	mu = 0
	path = null
	for i = 0 to N-1
		build a DFS-tree with root vertex[i] and create a list tmp_dk[M]
		for j = 0 to M-1
			if tmp_dk[j] < dk[i][j]
				dk[i][j] = tmp_dk[j]
			end if
			if dk[i][j] < d[i]
				d[i] = dk[i][j]
			end if
		end for
	end for
	for i = 0 to N-1
		build a BFS-tree with root vertex[i]
		use BFS-tree find a path p with length d[i], k edges
		for j = 0 to N-1
			use BFS-tree to find a another path q with M-k edges
			if p and q disjoint except head and end && mu <= d[i] + dk[j][M-k]
				mu = d[i] + dk[j][M-k]
				path = p + q
			end if
		end for
	end for
	return mu and path
end function
```

We can show that we have two for loops in the function. One is $O(NM)$, which uses DFS-tree to find d(v) and $d_k(v)$, the other is $O(N^2)$, which uses BFS-tree to find the path of $d(v)$, $d_k(v)$, and the minimum mean cycle. Consequently, the total time complexity is $O(N^2 + NM)$.