Curtin University

Department of Computing

COMP3001

Design and Analysis of Algorithms Assignment

Author: 15560955
Chris Garland

Lecturer:
Dr. Sie Teng Soh

March 29, 2017

Question 1

a) **(10 marks).** Use the Master method to solve the following recurrence function:

$$T(n) = 3T(\sqrt[2]{n}) + \log_2 n \tag{1}$$

Solution:

Given the master theorem:

$$T(n) = aT(n/b) + f(n) \tag{2}$$

We can see that $a = 3$, $b = \sqrt{n}$ and $f(n) = \log_2 n$. As b does not conform to the master theorem, we will use a change of variable:

let: $n = 2^m \therefore \log_2 n = m$
$\therefore \sqrt[2]{n} = 2^{m/2}$

$$T(2^m) = 3T(2^{m/2}) + m \tag{3}$$

Now, we perform another substitution ...

let: $T(2^m) = S(m)$
let: $T(2^{m/2}) = S(m/2)$

$$S(m) = 3S(m/2) + m \tag{4}$$

We now have a recurrence equation that conforms to the format of the Master Theorem ... $a = 3$, $b = 2$ and $f(m) = m$. Lets compare $m^{\log_b a}$ with $f(m)$ ...

$m^{\log_b a} = m^{\log_2 3} > f(m)$
$f(m) = O(m^{\log_2 3 - \epsilon})$, where $\epsilon > 0$

By case 1 of Master Theorem:

$$S(m) = \Theta(m^{\log_2 3}) \tag{5}$$

We know that $S(m) = T(2^m)$ and $2^m = n \therefore$

$$T(n) = \Theta(m^{\log_2 3}) \tag{6}$$

Given that $m = \log_2 n$ ...

$$\begin{aligned} T(n) &= \Theta((\log_2 n)^{\log_2 3}) \\ &= \Theta(\log_2^{\log_2 3} n) \end{aligned} \tag{7}$$

Question 2

Consider the following communication network that is represented by a weighted graph $G = (V, E)$ in which the non-negative number $r_{u,v}$ represents the *operational probability* or *reliability* of link $(u, v) \in E$ for $0 \leq r_{u,v} \leq 1.0$. Recall that a path $P_{a,b}$ is a sequence of links from a given source node a to its destination node b. The reliability of a path (called *path reliability*), $r_{a,b}$, is computed by multiplying the reliability of each link in path $P_{a,b}$. For example of path $P_{A,E}$ = (A, D, B, E) from source node A to destination node E is $R_{A,E} = (0.9 * 0.85 * 0.8) = 0.612$. We define *the most reliable path* from a source node s to a destination node t as the path with the highest reliability among all possible paths form s to t, i.e., the maximum $R_{s,t}$.

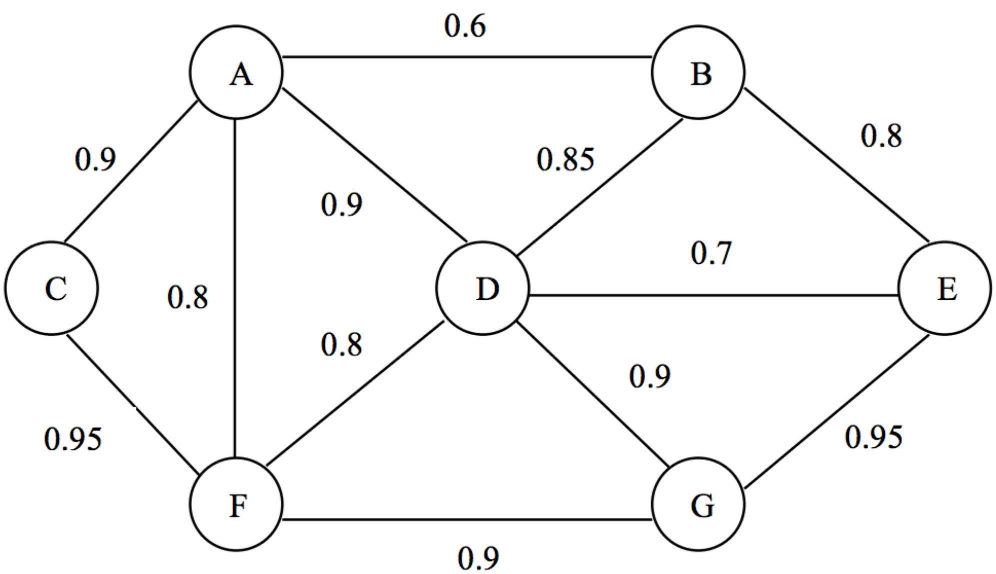

Figure 1: Weighted graph representation of a communication network

Question 2 continued on next page . . .

a) **(25 marks).** Design a *greedy* algorithm that generates the most reliable path from a source node s to every destination node t in the network.

Solution:

In this design we will modify Dijkstra's algorithm to *greedily* determine the most reliable path from a source node s to every destination node in the network. Original time complexity is maintained: $O(|E| + |V| \log_2 |V|)$ ...

DIJKSTRA-MODIFIED(G, w, s)

1 INITIALIZE-SINGLE-SOURCE(G, s)
2 $S = \emptyset$
3 $Q = G.V$
4 **while** $Q \neq \emptyset$
5 $\quad u =$ EXTRACT-MAX(Q) // Modification *from*: $u =$ EXTRACT-MIN(Q)
6 $\quad S = S \cup \{u\}$
7 $\quad$ **for** each vertex $v \in G.Adj[u]$
8 $\quad\quad$ RELAX(U, V, W)

Line 1 initializes the d and π values as shown below. Line 2 initializes the set S to the empty set. Line 3 initializes the *max-priority* queue Q to contain all the vertices in V. Each time through the **while** loop of lines 4–8, line 5 extracts a vertex from Q and line 6 adds it to S. Lines 7–8 relax each edge (u, v) and updates the estimate $v.d$ if the path reliability can be improved.

INITIALIZE-SINGLE-SOURCE(G, s)

1 **for** each vertex $v \in G.V$
2 $\quad v.d = 0$ // Modification *from*: $v.d = \infty$
3 $\quad v.\pi = \text{NIL}$ // v's predecessor
4 $s.d = 1$ // Modification *from*: $s.d = 0$

We modify by initializing the *reliability* attribute $v.d$, of all $v \in V - \{s\}$ to 0, which is a lower bound on the weight/reliability of a path from source s to v. We call $v.d$ a path reliability estimate. The path-reliability attribute $s.d$ of the source node s is initialized to 1 (max reliability).

RELAX(u, v, w)

1 **if** $v.d < u.d \times w(u, v)$ // Modification *from*: $v.d > u.d + w(u, v)$
2 $\quad v.d = u.d \times w(v, v)$ // Modification *from*: $v.d = u.d + w(u, v)$
3 $\quad v.\pi = u$ // v's predecessor

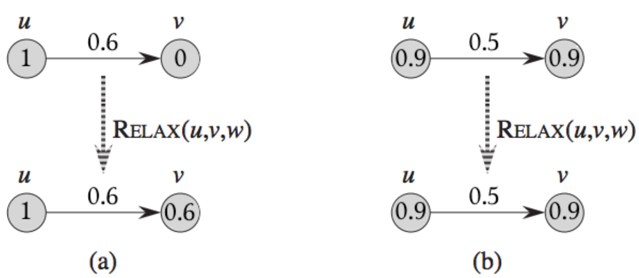

Figure 2: **(a)** Because the value of $v.d < u.d \times w(u, v)$ prior to relaxation, the value of $v.d$ is updated. **(b)** Here, $v.d \geq u.d \times w(u, v)$ before relaxation, and so $v.d$ remains unchanged.

b) **(15 marks)**. Use your algorithm in part a) to generate the most reliable path from node A to every other node in the given graph. List the most reliable paths and their corresponding path reliabilities.

Solution:

On Next Page . . .

Step #	**Unvisited(Q)**	**Visited(S)**	**Current(u)**	**Reliability of Path to Vertex(v): (reliability[s-v], predecessor(π))$_{iteration}$**						
				A(s)	**B**	**C**	**D**	**E**	**F**	**G**
Init	{A, B, C, D, E, F, G}	{-}	N/A	$(1, \text{-})_0$	$(0, \text{-})_0$	$(0, \text{-})_0$	$(0, \text{-})_0$	$(0, \text{-})_0$	$(0, \text{-})_0$	$(0, \text{-})_0$
1	{B, C, D, E, F, G}	{A}	A	$(1, \text{-})_0$	$(0.6, \text{A})_1$	$(0.9, \text{A})_1$	$(0.9, \text{A})_1$	$(0, \text{-})_0$	$(0.8, \text{A})_1$	$(0, \text{-})_0$
2	{B, D, E, F, G}	{A, C}	C	$(1, \text{-})_2$	$(0.6, \text{A})_1$	$(0.9, \text{A})_1$	$(0.9, \text{A})_1$	$(0, \text{-})_0$	$(0.855, \text{C})_2$	$(0, \text{-})_0$
3	{B, E, F, G}	{A, C, D}	D	$(1, \text{-})_3$	$(0.765, \text{D})_3$	$(0.9, \text{A})_1$	$(0.9, \text{A})_1$	$(0.63, \text{D})_3$	$(0.855, \text{C})_3$	$(0.81, \text{D})_3$
4	{B, E, G}	{A, C, D, F}	F	$(1, \text{-})_4$	$(0.765, \text{D})_3$	$(0.9, \text{A})_4$	$(0.9, \text{A})_4$	$(0.63, \text{D})_3$	$(0.855, \text{C})_3$	$(0.81, \text{D})_3$
5	{B, E}	{A, C, D, F, G}	G	$(1, \text{-})_4$	$(0.765, \text{D})_3$	$(0.9, \text{A})_4$	$(0.9, \text{A})_5$	$(0.7695, \text{G})_5$	$(0.855, \text{C})_5$	$(0.81, \text{D})_3$
6	{B}	{A, C, D, E, F, G}	E	$(1, \text{-})_4$	$(0.765, \text{D})_6$	$(0.9, \text{A})_4$	$(0.9, \text{A})_6$	$(0.7695, \text{G})_5$	$(0.855, \text{C})_5$	$(0.81, \text{D})_6$
7	{-}	{A, B, C, D, E, F, G}	B	$(1, \text{-})_7$	$(0.765, \text{D})_6$	$(0.9, \text{A})_4$	$(0.9, \text{A})_7$	$(0.7695, \text{G})_7$	$(0.855, \text{C})_5$	$(0.81, \text{D})_6$

Question 3

Consider an undirected graph $G = (V, E)$, where V is a set of nodes and E is a set of links. As an example, consider the following graph, where each link is labeled by a lower case letter, e.g., link a connects nodes A and C. As defined in Chapter 23 of the textbook (Introduction to Algorithms by Cormen, et al), a cut (S, VS) is a partition of nodes in V. Further, a link $(u, v) \in E$ *crosses* the cut (S, VS) if **either** node $u \in S$ and $v \in (VS)$ **or** node $u \in (VS)$ and $v \in S$; i.e., one of its end points is in S and the other is in VS. As an example, $S_1 = \{C\}$ and $S_2 = V - S_1 =$ A, B, D, E, F, G is a cut. The weight of a cut is defined as the number of links **crossing** the cut. As an example, the weight of the cut (S_1, S_2) is two; there are two crossing links in the cut. The **maximum cut** (called **Max-Cut**) is a cut with the **maximum weight**. The problem of finding a maximum cut in a graph is known as the **Max-Cut Problem**, a well known NP-complete problem.

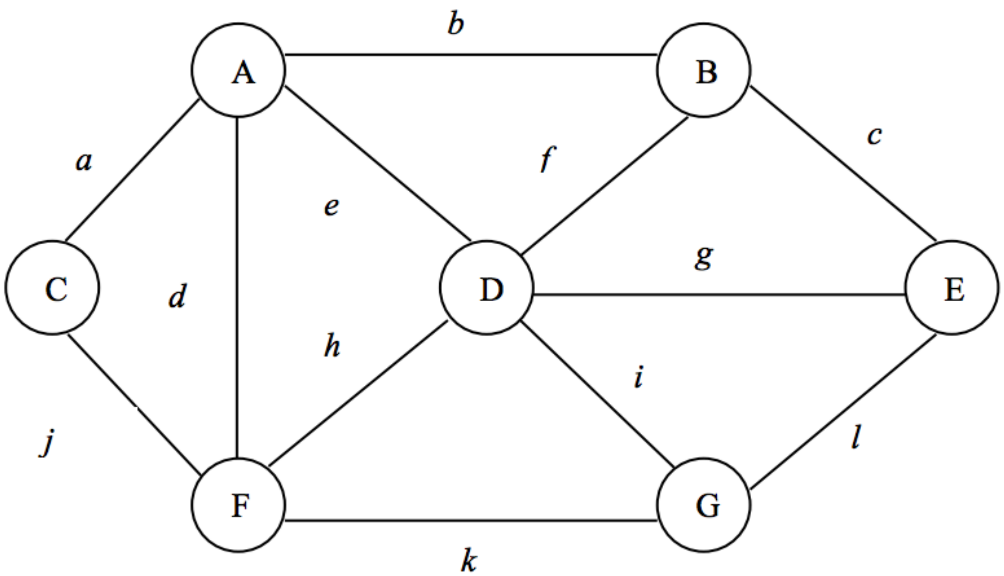

Figure 3: Weighted graph representation of a communication network

a) **(5 marks).** Generate all possible cuts in the given graph, and determine its maximum cut.

Solution:

In order to determine the number of all possible cuts, we will use the formula ...

$$2^n - 2 \tag{8}$$

Here we have n as the number of vertices. This will give us the number of *all* possible combinations including the empty *and* full sets. This is why we subtract 2. This will give us the answer ...

$$\begin{aligned} 2^n - 2 &= 2^7 - 2 \\ &= 126 \end{aligned} \tag{9}$$

This is the number of all possible cuts.

In the figure below will provide the Max-Cut ...

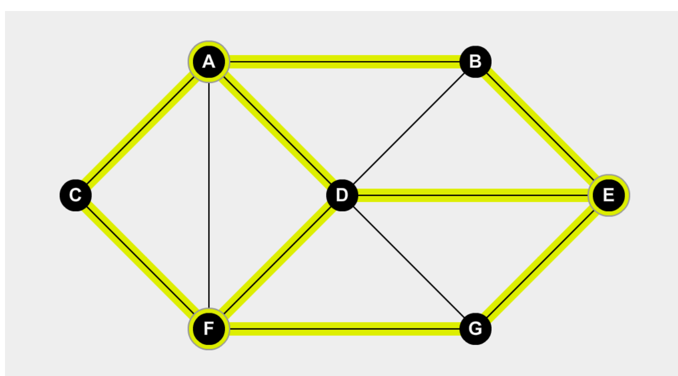

Figure 4: Max-Cut of graph illustrated in previous figure

b) **(20 marks)**. Design a *greedy* algorithm to solve the Max-Cut problem. As part of your solution, you must state your *greedy criteria*. Further, show your algorithm in a concise but clear pseudo-code. You must explain in detail each line of the pseudo-code and show how to implement the algorithm so that it has the best possible time complexity.

Solution:

In this design we follow the greedy approach of making the locally optimal choice at each stage with the hope of finding a global optimum. the idea behind this algorithm is that we have two sets S_1 and S_2. S_1 will be initialized to contain the first vertex $v \in G.V$ (A in this case). S_2 will be initialized to $G.V - S_1$ (all other vertices). Now we have $(S_1 = \{A\}, S_2 = \{B, C, D, E, F, G\})$. Lets consider vertex A $\in S_1$. The number of cuts is determined by the number of adjacent vertices in S_2. In this case we have 4, $\{B, C, D, F\}$. We will call this set *external vertices*. Any vertices in the same set, we will call *internal vertices*. The algorithm runs through each $v \in G.V$ and determines whether the number of *internal vertices* $\geq$ the number of *external vertices* ***(greedy criteria)***. If so, then swap sets and maintain a boolean value to be use as a loop terminator. Once there is no more improvement, we have our final cut.

Input: Graph (all vertices $G.V$, adjacency list $G.adj[u]$)
Output: MAX-CUT

```
MAX-CUT(G)
1   S1 = 1st ∈ G.V
2   S2 = G.V − S1
3   do
4       improvement = FALSE
5       for each vertex u ∈ G.V
6           if (num v ∈ G.adj[u]) ∈ current set ≥
               (num v ∈ G.adj[u]) ∈ other set
7               SWAP-SETS(u, S1, S2)
8               improvement = TRUE
9   while improvement
10  return (S1, S2)
```

Lines 1–2 initialize the sets. In this case, to $(\{A\}, \{B, C, D, E, F, G\})$. The **do - while** loop on lines 3–9 will terminate if the boolean value on Line 4 is not adjusted to TRUE. The **for** loop on lines 5–8 will iterate over every vertex $v \in G.V$. The **if** statement on line 6 is what makes this algorithm greedy. **if** number of *internal vertices* $\geq$ number *external vertices* then swap sets and maintain bool value to keep the loop going. Finally when there does not exist a vertex that has a greater number of *internal vertices*, Line 10 will **return** the max-cut.

```
SWAP-SETS(u)
if u.currentSet == S1
    S1 = S1 − {u}
    S2 = S2 ∪ {u}
    u.currentSet = S2
else
    S2 = S2 − {u}
    S1 = S1 ∪ {u}
    u.currentNode = S1
```

c) **(10 marks)**. Use your algorithm in part b) to generate the Max-Cut of the given graph. Does your algorithm generate an optimal result?

Solution:

On Next Page . . .

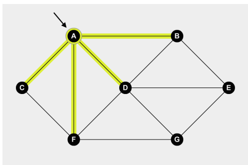

Figure 5: $S_1 = \{A\}; S_2 = \{B, C, D, E, F, G\}$

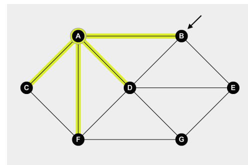

Figure 6: Internal edges $\geq$ External edges

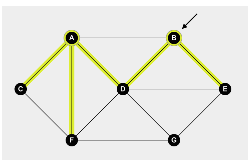

Figure 7: $S_1 = \{A, B\}; S_2 = \{C, D, E, F, G\}$

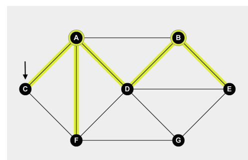

Figure 8: Internal edges $\geq$ External edges

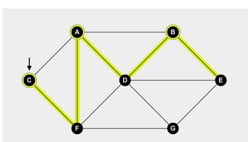

Figure 9: $S_1 = \{A, B, C\}; S_2 = \{D, E, F, G\}$

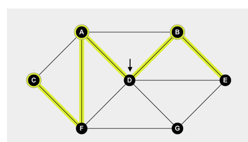

Figure 10: Internal edges $\geq$ External edges

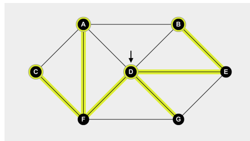

Figure 11: $S_1 = \{A, B, C, D\}; S_2 = \{E, F, G\}$

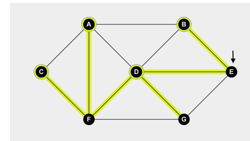

Figure 12: Internal edges $\ngeq$ External edges

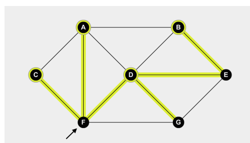

Figure 13: Internal edges $\ngeq$ External edges

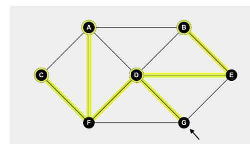

Figure 14: Internal edges $\geq$ External edges

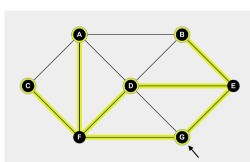

Figure 15: $S_1 = \{A, B, C, D, G\}; S_2 = \{E, F\}$

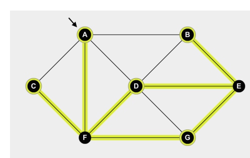

Figure 16: Internal edges $\geq$ External edges

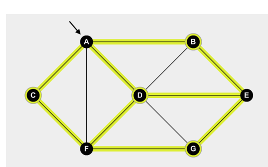

Figure 17: $S_1 = \{B, C, D\}; S_2 = \{A, E, F\} =$ Max-Cut

d) **(5 marks)**. Give a counter example to show that your greedy algorithm does not always generate an optimal result.

Solution:

The algorithm may not produce the optimal result in the case of a bipartite graph.