Soham Chatterjee
Email: soham.chatterjee@tifr.res.in
Course: Algorithms

Assignment - 3
Dept: STCS
Date: November 8, 2024

Problem 2 (15 marks)

Given a directed graph $G = (V, E)$ with special vertices s, t, we define the following sets. Let X be the set of vertices that *always* lie on the side of s in any minimum cut (e.g., $s \in X$). Let Y be the set of vertices that *always* lie on the side of t in any minimum cut (e.g., $t \in Y$). Let $Z = V \setminus (X \cup Y)$. Give an O(time for max-flow computation)-time algorithm to partition V into X, Y and Z

Solution: Consider the $s-t$ cut $(S, V \setminus S)$ with smallest number of vertices where $s \in S$. We will show that the set of vertices that always lie on the side of s in any minimum cut $(X, V \setminus X)$ is actually S. First we define the notation $cap_T(T') = \sum_{e=(u,v):u\in T, v\in T'} c_e$ for any subset $T, T' \subseteq V$ and $T \cap T' = \emptyset$

Lemma 1. *S is contained in every $s-t$ mincut $(X, V \setminus X)$ where $s \in X$.*

Proof: Suppose S is not in every $s-t$ mincut. Suppose there exists a mincut $(X, V \setminus X)$ such that $S \not\subseteq X$. Then consider the sets $X \setminus S$, $S \setminus X$. Since $S \not\subseteq X$ and S has smallest size we have $X \setminus S \neq \emptyset$. Now for all $v \in X \setminus S$ we have $cap_S(v) \geq 0$ since otherwise $f(S \cup \{v\}) = f(S) + cap_S(v) < f(S)$ which is not possible as S is a min cut. Now consider the set $X \cap S$. $X \cap S \neq \emptyset$ as $s \in X \cap S$ and $|X \cap S| < |S|$. Hence $X \cap S$ is not a mincut. Therefore $f(X \cap S) > f(S)$. Now notice that

$$\begin{aligned}
f(S) &= cap_{V\setminus(X\cup S)}(X \cap S) + cap_{V\setminus(X\cup S)}(S \setminus X) + cap_{X\setminus S}(X \cap S) + cap_{X\setminus S}(S \setminus X) \\
f(X) &= cap_{V\setminus(X\cup S)}(X \cap S) + cap_{V\setminus(X\cup S)}(X \setminus S) + cap_{S\setminus X}(X \cap S) + cap_{S\setminus X}(X \setminus S) \\
f(X \cap S) &= cap_{V\setminus(X\cup S)}(X \cap S) + cap_{X\setminus S}(X \cap S) + cap_{S\setminus X}(X \cap S) \\
f(X \cup S) &= cap_{V\setminus(X\cup S)}(X \cap S) + cap_{V\setminus(X\cup S)}(X \setminus S) + cap_{V\setminus(X\cup S)}(S \setminus X)
\end{aligned}$$

Therefore we have

$$f(S) > f(S) + f(X) - f(X \cap S) = f(X \cap S) + cap_{X\setminus S}(S \setminus X) + cap_{S\setminus X}(X \setminus S) \geq f(X \cup S)$$

Hence we obtain a $s-t$ cut $X \cup S$ which has capacity lesser than the S which is not possible. Hence contradiction. Therefore S is contained in every $s-t$ mincut $(X, V \setminus X)$ where $s \in X$. □

Hence by Lemma 1 we have that the set of vertices which lies on every $s-t$ mincut is S since S is the smallest $s-t$ mincut where $s \in S$. This lemma also suggests that there in an unique smallest size $s-t$ cut. So all we have to do now is find the mincut which has the smallest number of vertices. Now we will show that for any max-flow the set of vertices reachable in the corresponding residue graph is indeed the set S which also forms a mincut.

Lemma 2. *For any max flow f in the residue graph G_f the set of vertices reachable from s is the set S.*

Proof: We know the set of vertices reachable from s in the residue graph G_f is a $s-t$ mincut. Let the set of vertices reachable from s in G_f is the set X. By Lemma 1 $S \subseteq X$. Suppose $S \neq X$. Then $S \subsetneq X \implies X \setminus S \neq \emptyset$. So $\exists\ x \in X \setminus S$. Now since f is max flow we have

$$\sum_{e=(u,v):u\in S, v\notin S} c_e = |f| = \sum_{e=(u,v)u\in S, v\notin S} f(u,v)$$

Since $c_e \geq f(e)$ for all $e \in E$ we have that $c_e = f_e$ for all $r = (u, v) \in E$ where $u \in S$ and $v \notin S$. Therefore all the forward edges between S and $V \setminus S$ in G_f are saturated and therefore all the edges between S and $V \setminus S$ in G_f are backward edges. Hence no vertex of $V \setminus S$ are reachable from s in G_f. Hence x is not reachable. But $x \in X$ and X is the set of vertices in G_f reachable from s. Hence contradiction. Therefore S is the set of vertices reachable from s in G_f. Hence we have the lemma. □

So now we have proved that if we run the max flow algorithm on the graph with the capacities we get the maxflow

corresponding to the minimum cut which has the smallest size. Now from a maxflow we can get the corresponding mincut by running a Depth First Search Algorithm on the residue graph of the max flow and the cut set contains all the vertices reachable from s. Now to find the the set of vertices which always lie on the side of t in any minimum cut we reverse the direction of edges in the graph then run the same maxflow algorithm on the new graph with the same capacities. This will return the minimum cut $(V \setminus T, T)$ where $t \in T$ and T has the smallest size. So the algorithm steps follows like this:

Algorithm 1: Partitioning V

Input: $G = (V, E)$, $c_e \in \mathbb{Z}_0$ for all $e \in E$
Output: (X, Y, Z) such that $V = X \sqcup Y \sqcup Z$

1 **begin**
2 **for** $(u, v) \in E$ **do**
3 $\tilde{c}_{v,u} \longleftarrow c_{u,v}$
4 $f \longleftarrow$ Edmond-Karp$(G = (V, E), \{c_e : e \in E\})$
5 **for** $e = (u, v) \in E$ **do**
6 **if** $f(u, v) > 0$ **then**
7 $(v, u) \in E_f$
8 **if** $f(u, v) < c_{u,v}$ **then**
9 $(u, v) \in E_f$
10 $X \longleftarrow$ Set of visited vertices on running DFS on the graph $G_f = (V, E_f)$
11 **for** $(u, v) \in E$ **do**
12 $(v, u) \in E'$
13 $g \longleftarrow$ Edmond-Karp$(G = (V, E'), \{\tilde{c}_e : e \in E'\})$
14 **for** $e = (u, v) \in E'$ **do**
15 **if** $g(u, v) > 0$ **then**
16 $(v, u) \in E'_g$
17 **if** $g(u, v) < c_{u,v}$ **then**
18 $(u, v) \in E'_g$
19 $Y \longleftarrow$ Set of visited vertices on running DFS on the graph $G_g = (V, E'_g)$
20 **return** $(X, Y, V \setminus (X \cup Y))$

For the for loops at line 2,5,11,14 it takes $O(|E|)$ time. And the DFS at lines 10,19 takes time $O(|V|+|E|)$ time. But we know Edmond-Karp algorithm takes time $O(|E|^2 \times |V|)poly(c_e)$. Hence the algorithm takes O(Time for Maxflow algorithm). Hence the total runtime of the algorithm O(Time for max-flow computation). ■

[Me, Soumyadeep and Soumyajit discussed the problem together. I also discussed with Vivek and Shubham]

Problem 3 (5 marks)

Given a set S of n items, a function $f : 2^S \to \mathbb{R}$ is said to be *submodular* if, for all sets $A \subseteq B$ and elements $x \notin B$,

$$f(A \cup \{x\}) - f(A) \geq f(B \cup \{x\})) - f(B)$$

That is, the marginal value of an element ot a smaller set, is at least it's marginal value to a larger set. Prove that a function f is submodular if and only if it satisfies, for any sets $X, Y \subseteq S$,

$$f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$$

Solution: Suppose f is submodular. Let $B \setminus A = \{v_1, \ldots, v_k\}$. Now denote $T_i = \{v_1, \ldots, v_i\}$ for all $i \in \{0, \ldots, k\}$ where

$T_0 = \emptyset$. Now for T_i where $i < k$ we have $(A \cap B) \cup T_i \subseteq A \cup T_i$. Hence

$$f\Big(((A \cap B) \cup T_i) \cup \{v_{i+1}\}\Big) - f\Big((A \cap B) \cup T_i\Big) \geq f\Big((A \cup T_i) \cup \{v_{i+1}\}\Big) - f(A \cup T_i)$$
$$\implies f\Big((A \cap B) \cup T_{i+1}\Big) - f\Big((A \cap B) \cup T_i\Big) \geq f(A \cup T_{i+1}) - f(A \cup T_i)$$

Now we sum these expressions for all $i = 0$ to $i = k-1$ and we get

$$f\Big((A \cap B) \cup T_k\Big) - f\Big((A \cap B) \cup T_0\Big) \geq f(A \cup T_k) - f(A \cup T_0) \implies f(B) - f(A \cap B) \geq f(A \cup B) - f(A)$$

Hence we have $f(A) + f(B) \geq f(A \cup B) + f(A \cap B)$.

Now we will show that if f satisfies the condition that $f(X) + f(Y) \geq f(X \cup Y) + f(X \cap Y)$ for any two subsets $X, Y \subseteq S$ then f is submodular. Let $A, B \subseteq S$ where $A \subseteq B$. Now let $x \notin B$ and $x \in S$. Then we have

$$f(A \cup \{i\}) + f(B) \geq f(A \cup \{i\} \cup B) + f\Big((A \cup \{i\}) \cap B\Big) \implies f(A \cup \{i\}) + f(B) \geq f(B \cup \{i\}) + f(A)$$

Hence we have $f(A \cup \{i\}) - f(A) \geq f(B \cup \{i\}) - f(B)$. Therefore f is submodular. ■

Problem 4 (5 marks)

Let $G = (V, E)$ be a directed graph with nonnegative integral capacity c_e on each edge. Define the *cut function* $f : 2^V \to \mathbb{Z}_+$ as

$$f(S) = \sum_{e=(u,v):u \in S, v \notin S} c_e$$

Show that cut function f is submodular.

Solution: Using Problem 3 it suffices to show that for any $S, T \subseteq V$ we have $f(S) + f(T) \geq f(S \cup T) + f(S \cap T)$. Like in Problem 2 we define the notation $cap_T(T') = \sum_{e=(u,v):u \in T, v \in T'} c_e$ for any subset $T, T' \subseteq V$ and $T \cap T' = \emptyset$.

Then we have

$$f(S) = cap_{V \setminus (S \cup T)}(S \cap T) + cap_{V \setminus (S \cup T)}(S \setminus T) + cap_{T \setminus S}(S \cap T) + cap_{T \setminus S}(S \setminus T)$$
$$f(T) = cap_{V \setminus (S \cup T)}(S \cap T) + cap_{V \setminus (S \cup T)}(T \setminus S) + cap_{S \setminus T}(S \cap T) + cap_{S \setminus T}(T \setminus S)$$
$$f(S \cup T) = cap_{V \setminus (S \cup T)}(S \cap T) + cap_{V \setminus (S \cup T)}(S \setminus T) + cap_{V \setminus (S \cup T)}(T \setminus S)$$
$$f(S \cap T) = cap_{V \setminus (S \cup T)}(S \cap T) + cap_{S \setminus T}(S \cap T) + cap_{T \setminus S}(S \cap T)$$

Hence we have $f(S) + f(T) - (f(S \cup T) f(S \cap T)) = 2cap_{T \setminus S}(S \setminus T) \geq 0$. Hence by Problem 3 f is submodular. ■

Problem 5 (10 marks)

Let $G = (V, E)$ be a directed graph with nonnegative integral capacities on the edges, and let s, t, be two special vertices in the graph. Let $(S, V \setminus S)$ be a minimum $s - t$ cut with vertices u, v in S, **so that there exists a minimum** $u - v$ **cut** $(U, V \setminus U)$ **with** $t \notin U$**.** Then show that there exists a minimum cut $u - v$ cut $(U', V \setminus U')$ so that $U' \subseteq S$ ~~or $V \setminus U \subseteq S$~~.
Problems 3, 4 may be useful in solving this.

Solution: Since $t \notin U$ and $u, v \in S$, $S \cup U$ forms a $s - t$ cut and $S \cap U$ forms a $u - v$ cut. Now since the *cut function* is submodular by Problem 4. Hence we have using Problem 3

$$f(S) + f(U) \geq f(S \cup U) + f(S \cap U)$$

Now since $S \cap U$ is a $s - t$ cut and S is $s - t$ min cut we have $f(S \cup U) \geq f(S)$. And similarly since U is $u - v$ mincut and $S \cap U$ is a $u - v$ cut we have $f(S \cap U) \geq f(U)$. Therefore

$$f(S \cup U) + f(S \cap U) = f(S) + f(U) \implies f(S \cup U) + f(S \cap U) = f(S) + f(U)$$

Now if $f(S \cup U) > f(S)$ then we have $f(S \cup U) + f(S \cap U) > f(S) + f(U)$ but this is not true because of the submodularity of the cut function. Hence $f(S \cup U) = f(S)$. Therefore we get $f(S \cap U) = f(U)$. Hence $S \cap U$ is a $u - v$ mincut which is a subset of S. ■
[Me and Soumyadeep discussed with the instructor]

Problem 6 Exercise 21-2 (Depth determination) from CLRS (25 marks)

In the ***depth-determination problem***, we maintain a forest $\mathcal{F} = \{T_i\}$ of rooted trees under three operations:

MAKE-TREE(v) creates a tree whose only node is v.
FIND-DEPTH(v) returns the depth of node v within its tree.
GRAFT(r, v) makes node r, which is assumed to be the root of a tree, become the child of node v, which is assumed to be in a different tree than r but may or may not itself be a root.

a. Suppose that we use a tree representation similar to a disjoint-set forest: $v.p$ is the parent of node v, except that $v.p = v$ if v is a root. Suppose further that we implement GRAFT(v) by setting $r.p = v$ and FIND-DEPTH(v) by following the find path up to the root, returning a count of all nodes other than v encountered. Show that the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH , and GRAFT operations is $\Theta(m^2)$.

By using the union-by-rank and path-compression heuristics, we can reduce the worst-case running time. We use the disjoint-set forest $\mathcal{D} = \{S_i\}$, where each set S_i (which is itself a tree) corresponds to a tree T_i in the forest $\mathcal{F}$. The tree structure within a set S_i, however, does not necessarily correspond to that of T_i. In fact, the implementation of S_i does not record the exact parent-child relationships but nevertheless allows us to determine any node's depth in T_i. The key idea is to maintain in each node v a "pseudodistance" $v.d$, which is defined so that the sum of the pseudodistances along the simple path from v to the rot of its set S_i equals the depth of v in T_i. That is, if the simple path from v to its root in S_i is $v_0, v_1, \ldots, v_k$ where $v_0 = v$ and v_k is S_i's root, then the depth of v in T_i is $\sum_{j=0}^{k} v_j.d$.

b. Give an implementation of MAKE-TREE

c. Show how to modify FIND-SET to implement FIND-DEPTH. Your implementation should perform path compression, and its running time should be linear in the length of the find path. Make sure that your implementation updates psedodistances correctly.

d. Show how to implement GRAFT(r, v) which combines the sets containing r and v, by modifying UNION and LINK procedures. Make sure that your implementation updates pseudodistances correctly. Note that the root of a set S_i is not necessarily the root of the corresponding tree T_i.

e. Find a tight bound on the worst-case running time of a sequence of m MAKE-TREE, FIND-DEPTH , and GRAFT operations, n of which are MAKE-TREE operations.

Solution:

a. MAKE-TREE and GRAFT both takes constant time. FIND-DEPTH takes time linear in the depth of the node. Now the depth can be at most the number of nodes present in the tree. So if k of the m commands are MAKE-TREE and GRAFT operations and rest are FIND-TREE operations then each FIND-TREE takes at most $O(k)$ time. Hence the total time is $\theta(k) + (m-k)\theta(k) = \theta(k(m-k))$. Now $k(m-k)$ is largest when $k = m-k \implies k = \frac{m}{2}$. Therefore it takes $\theta(m^2)$ time in worst case for a sequence of m MAKE-TREE, FIND-DEPTH and GRAFT operations.

b. Sine MAKE-TREE creates a new tree with only one node, we create a new set with only a single node.

Algorithm 2: MAKE-TREE(v)

```
1 v.p ⟵ v;
2 v.rank ⟵ 0;
3 v.d ⟵ 0
```

c. Like suggested in the question we will do a path compression and while doing a path compression we update

the add $v.d$ to the value obtained by doing a path compression from $v.p$ and this will give the depth. So

Algorithm 3: FIND-SET(v)

```
if v ≠ v.p then
    (v.p.d) ⟵ FIND-SET(v.p);
    v.d ⟵ v.d + d;
return (v.p, v.d)
```

FIND-DEPTH(v) = FIND-SET(v)[1]

d. Now to implement GRAFT we will need to find the roots of the tree r, v is in and then find the depths of r, v. We will add the trees according to their ranks and then update the pseudodistances.

Algorithm 4: GRAFT(r, v)

```
x ⟵ FIND-SET(v)[0];
(y, depth) ⟵ FIND-SET(v);
if x.rank > y.rank then
    y.p ⟵ x, y.d ⟵ y.d − x.d
else
    x.p ⟵ y, x.d ⟵ x.d + depth + 1 − y.d;
    if x.rank == y.rank then
        y.rank ⟵ x.rank + 1
```

e. MAKE-TREE takes constant time. For FIND-DEPTH takes O(FIND-SET) time and GRAFT takes O(UNION) time. Hence after a sequence of m opertions, n of which are MAKE-TIME operations, takes $O(n)$ time. Now other $m - n = O(m)$ operations are FIND-DEPTH and GRAFT. So it takes $O(m \log^* n)$ time in worst case.

■

Problem 7 Exercise 16-4(a) from CLRS (Scheduling variations) (10 marks)

Consider the following algorithm for the problem from section 16.5 of scheduling unit-time tasks with deadlines and penalties. Let all n time slots be initially empty, where time slot i is the unit-length slot of time that finishes at time i. We consider the tasks in order of monotonically decreasing penalty. When considering task a_j, if there exists a time slot at or before a_j's deadline d_j that is still empty, assign a_j to the latest such slot, filling it. If there is no such slot, assign task a_j to the latest of the as yet unfilled slots. Argue that this algorithm always gives an optimal answer.

Solution: Suppose $E = \{a_j : j \in [n]\}$ are the jobs with deadline $d_1, \ldots, d_n$. Let $S \subseteq J$ is independent i.e. $S \in \mathcal{I}$ if S has a feasible schedule. Now we know $(E, \mathcal{I})$ forms a matroid. Define $N_t(S) = |\{j \in S : d_j \leq t\}|$. We know $S \in \mathcal{I} \iff N_t(S) \leq t$ for all t. For any $j \in [n]$, if there exists a time slot at d_j or before the algorithm assigns a_j to the latest such slot and otherwise the algorithm a_j assigns a_j to the latest of the as yet unfilled slots. Therefore at any iteration the set of jobs that are filled in the time slots at or before their deadlines is an independent set.

Suppose till some iteration the set of operations that are filled in the time slots at or before their deadlines is S. Hence $S \in \mathcal{I}$. Therefore in the next iteration if the algorithm assigns a_j within it's deadline then $S \cup \{a_j\} \in \mathcal{I}$. Now let $S \cup \{a_j\} \in \mathcal{I}$. Suppose the algorithm is not able to place a_j in the time slot at or before d_j. That means all the time slots till d_j are filled before assigning a_j in a time slot. That means $N_{d_j}(S) = d_j$ and $N_{d_j}(S \cup \{a_j\}) = d_j + 1$. But since $S \cup \{a_j\} \in \mathcal{I}$ we have $N_t(S \cup \{a_j\}) \leq t$ for all t. Hence contradiction. Therefore the algorithm assigns a_j in the time slot at or before d_j if and only if $S \cup \{a_j\} \in \mathcal{I}$.

Now in order to show that the algorithm outputs a optimal answer we have to show that the penalty is minimum. Since $(E, \mathcal{I})$ is a matroid. Hence the greedy algorithm works to find the minimum weight base. And

in the given algorithm each a_j is added in time slot at or before it's deadline if and only if adding to the set of jobs completed within deadline in previous iteration is an independent set. Hence the algorithm forms the greedy approach to minimize the penalty. Therefore the algorithm outputs an optimal solution. ■
[I discussed with Vivek]

Problem 8 (30 marks)

We are given two red-black trees T_1 and T_2 and an element x, with the guarantee that, for any $x_1 \in T_1$ and $x_2 \in T_2$, $x_1.key < x.key < x_2.key$. Our problem is to implement the procedure RB-Join that forms a single red-black tree from the elements in T_1, T_2, and x. Let n be the total number of nodes in T_1 and T_2.

(i) (5 marks) Given a red-black tree with n' nodes, show that the black-height of the tree can be obtained in time $O(\log n')$. Let $T.bh$ store this information for each red-black tree T.

(ii) (5 marks) Assume that $T_1.bh \geq T_2.bh$. Give an $O(\log n)$ times algorithm that finds a black node y in T_1 with the largest key from among all nodes in T_1 with black-height $T_2.bh$.

(iii) (5 marks) Let T_y be the subtree rooted at y. Describe how $T_y \cup \{x\} \cup T_2$ can replace T_y in $O(1)$ time without destroying the binary search tree property.

(iv) (10 marks) What color should x be so that the red-black properties 1,3,4 (from section 13.1 of CLRS) are maintained? Describe how to enforce properties 2 and 4 in $O(\log n)$ time.

(v) (5 marks) Complete the description of RB-Join, and show the running time.

Solution:

(i) Since in a Red-Black tree in every path from root to any leaf the number of black nodes visited is same. Therefore for any vertex if we start to go up till we reach the root and go down till we reach a leaf the total number of visited black nodes is the black-height of the red-black tree.

Algorithm 5: Find-Black-Height(T)

```
Input: Red-Black tree with n' nodes
Output: Find the black-height of the tree
1  begin
2      v ⟵ root of T, bh ⟵ 0
3      while True do
4          if v.color == black then
5              bh ⟵ bh + 1
6          if v.left ≠ NIL then
7              v ⟵ v.left
8          else if v.right ≠ NIL then
9              v ⟵ v.right
10         else
11             return bh
```

Since the height of a red-black tree with n' nodes is $O(\log n')$ nodes both the for loops run for $O(\log n')$ many iterations. Now this returns the black-height correctly since from the root on every encounter of a black nodes it increments bh by 1 and at the end when it reaches a leaf it increments bh and then returns the value. So the black-height of the tree can be obtained in time $O(\log n')$.

(ii) So to find a black node which has the largest key we will follow the right most path from the root and we will return the node on encountering the $(T_1.bh - T_2.bh + 1)^{th}$ black node since that black node has height $T_2.bh$. So the algorithm for finding the black node with largest key and height $T_2.bh$:

Algorithm 6: LARGEST-KEY-HEIGHT($T_1, T_2.bh$)

Input: Red-black trees T_1, T_2
Output: Find the black node with largest key with height $T_2.bh$

```
1  begin
2      if T_1.bh < T_2.bh then
3          return None
4      v ⟵ root of T_1
5      b ⟵ 0
6      while b < T_1.bh − T_2.bh + 1 do
7          if v.color == black then
8              b ⟵ b + 1
9          if v.right ≠ NIL then
10             v ⟵ v.right
11         else if v.left ≠ NIL then
12             v ⟵ v.left
13         else
14             return None
15     return v
```

(iii) Since for all $x_1 \in T_1$ we have $x_1.key < x$ and for all $x_2 \in T_2$ we have $x_2.key > y.key$ we will put x in place of y and then make y the left child of x and T_2 the right child of x. So we do the following operation:

Algorithm 7: REPLACE-SUBTREE(y, x, T_2)

```
1  v ⟵ root of T_2
2  x.left ⟵ y
3  x.right ⟵ v
4  v.p ⟵ x
5  y.p ⟵ x
6  x.p ⟵ y.p
```

(iv) Since we know the color of y is black and the root of T_2 is also black we set $x.color = red$. If $x.p.color = black$ then we are okay. Otherwise we have $x.p.color = red$. Now a red node can not have a red child. In this case we will do recolor and rotations as necessary like we did for INSERT operation in a red-black tree in order to balance the coloring. First we define the LEFT-ROTATE and RIGHT-ROTATE and RECOLOR:

Algorithm 8: RIGHT-ROTATE(u)

```
1  v ⟵ u.left
2  v.p ⟵ u.p
3  u.p ⟵ v
4  u.left ⟵ v.right
5  v.right ⟵ u
```

Algorithm 9: LEFT-ROTATE(u)

```
1  v ⟵ u.right
2  v.p ⟵ u.p
3  u.p ⟵ v
4  u.right ⟵ v.left
5  v.left ⟵ u
```

Algorithm 10: RECOLOR-BLACK(u)

```
1  u.color ⟵ black
2  u.left ⟵ red
3  u.right ⟵ red
```

Algorithm 11: RECOLOR-RED(u)

```
1  u.color ⟵ red
2  u.left ⟵ black
3  u.right ⟵ black
```

Now we use the pointer $v.uncle$ to point to the uncle of v i.e. sibling of $v.p$. If $v.uncle.color = red$ then we do on RECOLOR-RED($v.p.p$) and then we move to $v.p.p$ and check if the colors of $v.p.p$ and $v.p.p.p$ are same

or not other wise if $v.u.color = black$ then we do a LEFT-ROTATE($v.p$) and then a RIGHT-ROTATE($v.p.p$) and then RECOLOR-BLACK(v) and then we have a proper red-black tree. S we have the following algorithm: In

Algorithm 12: COLOR-NEW-NODE(x)

```
1  begin
2      x.color ⟵ red
3      v ⟵ x
4      while True do
5          if v.p.color == black then
6              return
7          if v.uncle.color == red then
8              RECOLOR-RED(v.p.p)
9              v ⟵ v.p.p
10         else if v.uncle.color == black then
11             LEFT-ROTATE(v.p)
12             RIGHT-ROTATE(v.p.p)
13             RECOLOR-BLACK(v)
14             return
```

the algorithm in each iteration the we move upwards till we reach the root and there we stop since the root has the color black. And we know height of the red-black tree is $O(\log n)$. Hence the number of iterations of the while loop is $O(\log n)$. Each iteration takes constant time. Therefore this algorithm takes $O(\log n)$ time and it ensures all the properties of the red-black tree are maintained.

(v) So the final complete description for the RB-JOIN is following:

Algorithm 13: RB-JOIN(T_1, x, T_2)

```
1  begin
2      bh_1 ⟵ FIND-BLACK-HEIGHT(T_1);
3      bh_2 ⟵ FIND-BLACK-TREE(T_2);
4      if bh_1 < bh_2 then
5          return None
6      y ⟵ LARGEST-KEY-HEIGHT(T_1, bh_2);
7      REPLACE-SUBTREE(y, x, T_2);
8      COLOR-NEW-NODE(x);
9      return T_1
```

■