UM-SJTU   JOINT   INSTITUTE

Introduction to Algorithms
(VE477)

Homework #3

Prof. Manuel

Xinmiao Yu

518021910792

Oct. 16, 2020

Q1.

1. Depth-First Search, is used to search or traverse tree or graph. It will explore edges out of the most recently discovered vertex v. Once all the edges of v has been discover, it will backtrack to the vertex that v is discovered. When all the vertex has been explored, the process will end. When use DFS for a graph, it will create a depth-first forest, composing several depth-first trees. Define each vertex initially white, when it is first discovered, grayed. When it is finished, such that all the adjacent list has been explore, blacked. Also use variable *time* to timestamp for the discovering time and finishing time.

Algorithm 1: DFS

Input : graph $G = (V, E)$
Output: depth-first forest

1 Function DFS(G):
2 | **for** *each vertex $u \in G.V$* **do**
3 | | $u.color \leftarrow WHITE$
4 | | $u.\pi \leftarrow NULL$
5 | **end for**
6 | $time \leftarrow 0$
7 | **for** *each vertex $u \in G.V$* **do**
8 | | **if** *$u.color == WHITE$* **then**
9 | | | DFS-VISIT(G, u)
10 | | **end if**
11 | **end for**
12 end
13 Function DFS-VISIT(G,u):
14 | time $\leftarrow$ time + 1
15 | $u.d \leftarrow time$
16 | $u.color \leftarrow GRAY$
17 | **for** *each $v \in G.adj[u]$* **do**
18 | | **if** *$v.color == WHITE$* **then**
19 | | | $v.\pi = u$
20 | | | DFS-VISIT(G, v)
21 | | **end if**
22 | **end for**
23 | u.color $\leftarrow$ BLACK
24 | time $\leftarrow$ time + 1
25 | u.f $\leftarrow$ time
26 end

---

2. Topological sorting is for directed acyclic graph. A linear ordering of vertices such that if a directed edge $u \leftarrow v$, then u should be before v in the ordering. It will print a vertex before its adjacent vertices while DFS will print a vertex then call DFS for its adjacency vertices. It could be easily realized through DFS. Because a vertex with smaller finish time should be

---

2

deeper in the graph, so put each finished vertex at the begin of the result sorting list then could get a topological sorting list.

Algorithm 2: Topological Sorting

Input : graph $G = (V, E)$
Output: topological sorted list

1 DFS(G) to compute each vertex's finish time
2 as each vertex is finished, insert it into the front of the result *list*
3 **return** *list*

---

3. Given a DAG, because Hamiltonian path is a path that includes every vertex, perform a topological sorting and check whether successive vertices are connect in the graph. These edges form a directed Hamiltonian path. The DFS algorithm is modified a bit, only result

---

      3

that would be used for finding Hamiltonian path is kept.

Algorithm 3: Hamiltonian Path in DAG

Input : A DAG graph $G = (V, E)$

Output: whether G contains a Hamiltonian path

1 Function DFS(G):

2 | **for** *each vertex $u \in G.V$* **do**

3 | | $u.color \leftarrow WHITE$

4 | | $u.\pi \leftarrow NULL$

5 | **end for**

6 | $list \leftarrow []$

7 | **for** *each vertex $u \in G.V$* **do**

8 | | **if** *u.color == WHITE* **then**

9 | | | DFS-VISIT(G, u, *list*)

10 | | **end if**

11 | **end for**

12 | **return** *list*

13 end

14 Function DFS-VISIT(G, *u*, *list*):

15 | $u.color \leftarrow GRAY$

16 | **for** *each $v \in G.adj[u]$* **do**

17 | | **if** *v.color == WHITE* **then**

18 | | | v.π = u

19 | | | DFS-VISIT(G, *v*, *list*)

20 | | **end if**

21 | **end for**

22 | u.color $\leftarrow$ BLACK

23 | insert u at the front of *list*

24 end

25 $result \leftarrow$ DFS(G)

26 for $i \leftarrow 0$ **to** $result.size - 1$ **do**

27 | **if** *result[i] do not connect with result[i+1]* **then**

28 | | **return** *False*

29 | **end if**

30 end for

31 return *True*

---

4. Assume the graph $G = (V, E)$. Because in function DFS, first initialize all the vertex (line 2-5), which is in $\Theta(V)$, then the loop on line 7-11 will be performed in $\Theta(V)$ if exclude the time for DFS-VISIT. Using aggregate analysis, because each vertex will be coloured after discovered, so each vertex is discovered for only one time. DFS-VISIT is called one time for each $v \in V$. For each v, its adjacent vertex u will be invoked into this turn if it is white. So during each

---

DFS-VISIT(G, u), the loop on line 16-22 is executed $|Adj[v]|$ times. As we have

$$\sum_{v \in V} |Adj[v]| = \Theta(E),$$

the total time for line 16-22 is $\Theta(E)$. Then the running time for DFS is $\Theta(V + E)$.

As we have V elements in $result$, the loop in line 26-30 will be executed in $\mathcal{O}(V)$. So its complexity is $\mathcal{O}(V + E)$.

5. It belongs to NP-class.

Q2.

1. No. To prove a function $f(n)$ is polynomial bounded, it is same to prove $log(f(n))$ is bounded by $\mathcal{O}(log(n^k))$, such that there exist constant c, n_0, k such that for any $n > n_0$, $log(f(n)) \leq cklogn$.

    First prove $log(n!) = \Theta(n \cdot logn)$

    $$\begin{aligned} log(n!) &= log(1) + log(2) + ...log(n) \\ &\leq log(n) + log(n) + ...log(n) = nlog(n) \\ log(n!) &\geq log(n/2) + log(n/2 + 1) + \ldots + log(n-1) + log(n) \\ &\geq log(n/2) + log(n/2) + \ldots + log(n/2) = (n/2)log(n/2) \\ &= (n/2)(logn + log(1/2)) \\ &= (n/2)logn - n/2 \geq 1/4(nlogn). \end{aligned}$$

    Then

    $$log(\lceil logn \rceil!) = \Theta(\lceil logn \rceil \cdot log(\lceil logn \rceil)).$$

    Also it is easy to get $\lceil logn \rceil = \Theta(logn)$. So we have

    $$log(\lceil logn \rceil!) = \Theta(logn \cdot log(logn)).$$

    Then prove $\lceil logn \rceil!$ is not bounded by a polynomial by contradiction. Assume it is bounded by a polynomial, then $log(\lceil logn \rceil!) = \mathcal{O}(logn)$, such that $logn \cdot log(logn) = \mathcal{O}(logn)$, which is obviously wrong. So $\lceil logn \rceil!$ is not bounded by a polynomial.

2. Yes. When $n > 1$, $log^*n = 1 + log^*logn$. log^*n is the number of log need to be applied to n to reach the fixed value 1.
    When $n > 1$, it is easy to obtain $logn < n - 1$ as $\frac{d(logn)}{dn} = \frac{1}{n} < 1 = \frac{d(n-1)}{dn}$. So

    $$loglog^*n = log(1 + log^*logn) < log^*logn.$$

3. The minimized number of weighing could be 2.

Algorithm 4: Find a lighter ball

Input   : eight balls with similar size $a1, a2, ..., a8$, one of which is lighter

Output: the lighter ball $a0$

1 Divide the balls into three groups $(a1, a2, a3), (a4, a5, a6), (a7, a8)$

2 **First Weighing:** compare the weight of $(a1, a2, a3)$ with $(a4, a5, a6)$

3 **if** $(a1, a2, a3).weight == (a4, a5, a6).weight$ **then**

4   **Second Weighing:** compare the weight of $a7$ and $a8$

5 $a0 \leftarrow min(a7.weight, a8.weight)$

/* assume $(a1, a2, a3)$ is lighter without losing generality   */

6 **else**

7   **Second Weighing:** compare the weight of $a1$ and $a2$

8   **if** $a1.weight == a2.weight$ **then**

9 $a0 \leftarrow a3$

10   **else**

11 $a0 \leftarrow min(a1.weight, a2.weight)$

12   **end if**

13 **end if**

Q3.

Rubik's Cube is made up of 26 mini cubes and each of them is called cubie. Turn the cube to make each face (a side of the cube) to have the same color. The center, edge and corner cubie refers to a cubie with one sticker, two stickers and three stickers respectively. Twist if a 90 or 180 degree turn of one of the six face, use F, B, U, D, L, and R designate 90-degree turns of the front, back, up, down, left, and right faces, respectively. Add ' (such that F') is a 90-degree counterclockwise "prime" twist, and add 2 (such that F2) is a 180-degree twist.

1. the beginner's method[1]. The whole step is shown in the Figure 1. It divides the cube into layers and solve each layer without messing up the pieces that already in place.
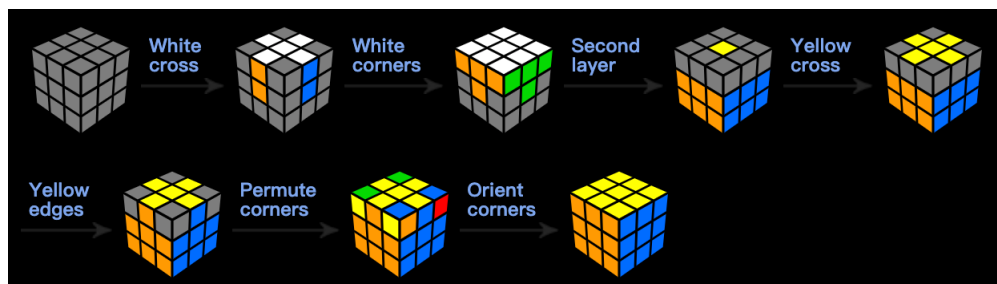

Figure 1: Step by step solution for beginner's method

   a. First layer: choose white (or any other color) to get white cross, then get white corners

---

to finish the first layer. An algorithm that would always be valid to bring a white corner piece to the top layer without break the solved pieces is R'D'RD.

b. Second layer: put the top layer face down now. Two algorithms to make the front-up edge to the right-front or left-front positions, called right algorithm (URU'R'U'F'UF) and left algorithm (U'L'ULUFU'F').

c. Last layer: make a yellow cross first (FRUR'U'F') then for yellow edges (RUR'URU2R'U to switch the front and left yellow edges). Premutation for the last layer (URU'L'UR'U'L) then do the orientation.

2. CFOP method. The advanced Fridrich (CFOP) method still divides the problem into different layers, and apply different algorithm for each step shown in Figure 2.
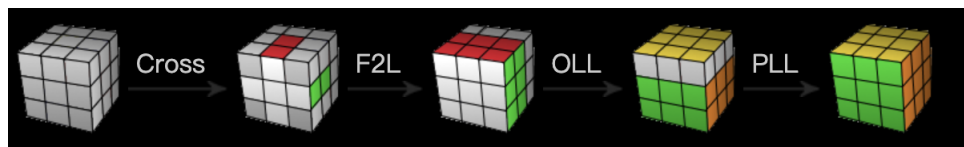

Figure 2: Steps of CFOP method

a. Complete the white cross (or choose any other color), but need to foresee 7 steps to succeed,

b. Solve the first two layers (F2L) through pairing white corner and second layer edge pieces. It could be done intuitively or memorize different algorithm for different situations.

c. Orienting the last layer (OLL). Solve the yellow face without matching the side colors.

d. Permutate the last layer (PLL).

References

[1] Ruwix. How to solve the rubik's cube?, 2019.

Q4.

To prove a problem is in $\mathcal{NP}$, show that a certificate could be used with a verifier running in deterministic polynomial time to recognize the given problem.

1. Certificate: a simple path y. Verifier: traverse the vertices to see whether y is a simple path, running in $\mathcal{O}(|V|)$.

2. Certificate: a factor y of the given integer n. Verifier: do $n\%y$, easily implement in polynomial time.

3. Certificate: a vertex cover of size k. Verifier: For each edge $e = (u, v) \in E$, check whether either u or v is in the vertex cover, running in $\mathcal{O}(|E| \cdot k)$.

Q5.

It is not sufficient to prove that PRIMES is in $\mathcal{P}$. It is obvious that by dividing n using incremented integers, the process could be stopped when the integer is larger than $\sqrt{n}$.

Algorithm 5: trivial division for PRIMES

Input : integer n
Output: whether n is a prime number

1 $i \leftarrow 2$
2 **while** $i \leq \sqrt{n}$ **do**
3   **if** $n \% i == 0$ **then**
4   **return** *True*
5   **end if**
6 $i \leftarrow i + 1$
7 **end while**
8 **return** *False*

The loop will be executed for at most $\sqrt{n}$ times, **Prime Number Theorem:** $\pi(N) \sim \frac{N}{log(N)}$, where $\pi(N)$ is the function counting the prime number smaller or less than N and $log(N)$ is the natural logarithm of N.