VE477 Introduction to Algorithms
Homework 3

Taoyue Xia, 518370910087

2021/10/18

Ex. 1 — Hamiltonian path

1. Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node, and explores as far as possible along each branch before backtracking or finding the exact route. The pseudocode can be shown below:

Algorithm 1: Depth-first Search

Input : A graph $G = \langle V, E \rangle$, V for vertices, E for edges
Output: A tree of vertices reachable from

```
1  Function DFS(G, v):
2      S ← a stack;
3      push v into S;
4      while S is not empty do
5          u ← S.pop();
6          if u is not visited then
7              Label u as visited;
8              for each vertex w adjacent to u do
9                  S.push(w);
10             end
11         end
12     end
13 end
```

2. Topological sort of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering. A topological sort is possible if and only if the graph has no directed cycles, which in other words, if it is a directed acyclic graph (DAG). Kahn's algorithm can explain topological sort clearly:

Algorithm 2: Topological Sort (Kahn's Algorithm)

Input : A directed graph $G = \langle V, E \rangle$
Output: A list L containing vertices in order after topological sort

```
/* All vertices have an attribute n for number of incoming edges      */
Function topological(G):
    L ← an empty list which will contain sorted vertices;
    S ← Queue of all vertices which initially have no incoming edge;
    while S is not empty do
        u ← S.pop();
        add u to L;
        for all vertices w which have an incoming edge from u do
            w.n ← w.n − 1;
            remove the edge uw from G;
            if w.n = 0 then
                push w into S;
            end
        end
    end
    if G still has some edges then
        return error ;                                  // graph has circle, not DAG
    end
    return L;
end
```

3. For this problem, since a directed acyclic graph is given, we can use topological sort to sort the vertices in order. After that, we can check whether there exists an edge for each two consecutive vertices in the list of sorted vertices. If true, then there is a Hamiltonian path. If false, then the answer would be no since we cannot go back in a Hamiltonian path in a DAG. Since the time complexity of topological sort is $\mathcal{O}(V + E)$, and the time complexity of check adjacency between consecutive vertices is $\mathcal{O}(V)$, the total complexity of determining a Hamiltonian path is $\mathcal{O}(V + E)$.

Algorithm 3: Hamiltonian Path Detection

Input : A directed acyclic graph $G = \langle V, E \rangle$
Output: a **flag** deciding whether the graph contains a Hamiltonian path or not

```
1 Function Hamiltonian(G):
2     L ← topological(G);
3     for i = 0 to L.length − 2 do
4         if no edge connecting L[i] and L[i + 1] then
5             return false;
6         end
7     end
8     return true;
9 end
```

4. For the while loop in topological sort, it will loop V times until all the vertices are visited. For the for loop in it, since the graph is a DAG, all the edges are taken into account, so the operation in for loop will be executed for at most E times. Therefore, the time complexity of topological sort is $\mathcal{O}(V + E)$. Furthermore, the time complexity of check of edges between all consecutive vertices is $\mathcal{O}(V)$. Therefore, the overall complexity is $\mathcal{O}(V + E)$.

5. The problem belongs to NP class.

Ex. 2 — Critical thinking

1. Firstly, by taking the logarithm of $n!$, we can calculate the time complexity of $\log(n!)$.

$$\begin{aligned}\log(n!) &= \log(1) + \log(2) + \cdots + \log n \\ &\leq \underbrace{\log n + \log n + \cdots + \log n}_{n} \\ &= n \log n\end{aligned}$$

Therefore $\log(n!) = \mathcal{O}(n \log n)$. Furthermore, we can calculate whether it has a lower bound:

$$\begin{aligned}
\log(n!) &= \log(1) + \log(2) + \cdots + \log n \\
&\geq \underbrace{\log(\lceil \frac{n}{2} \rceil) + \cdots + \log(\lceil \frac{n}{2} \rceil)}_{n/2} \\
&= \frac{n}{2} \log \frac{n}{2} \\
&= \frac{n}{2}(\log n - \log 2) \\
&= \frac{1}{2} n \log n - \frac{\log 2}{2} n \\
&= \Theta(n \log n)
\end{aligned}$$

Therefore, we can prove that $\log(n!) = \Omega(n \log n)$. Therefore, $\log(n!) = \Theta(n \log n)$.

Then substitute n with $\lceil \log n \rceil$, we can have:

$$\log(\lceil \log n \rceil !) = \Theta(\lceil \log n \rceil \log(\lceil \log n \rceil)) = \Theta(\log n \cdot \log \log n)$$

Suppose that $\lceil \log n \rceil !$ is bounded by a polynomial, then it means that $\log(\lceil \log n \rceil !) = \mathcal{O}(\log n)$. However, $\log(\lceil \log n \rceil !) = \Theta(\log n \cdot \log \log n) \neq \mathcal{O}(\log n)$. Then it raises a contradiction.

Therefore, $\lceil \log n \rceil !$ is not bounded by a polynomial.

2. From slide c2, we have the definition of the iterated logarithm function $\log^*$ as:

$$\log^* n = \begin{cases} 0 & \text{if } x \leq 1 \\ 1 + \log^* \log n & \text{if } x > 1 \end{cases}$$

To make $\log \log^* n$ have sense, $n > 1$ should be satisfied. Then, we can express $\log \log^* n$ as:

$$\log \log^* n = \log(1 + \log^* \log n)$$

Set $\log^* \log n = t$. If a function $f(x)$ is asymptotically larger than $g(x)$, then it means that:

$$\lim_{x \to \infty} \frac{f(x)}{g(x)} = \infty$$

Therefore, let $f(t) = t = \log^* \log n$ and $g(t) = \log(1 + t)$, then set $h(t) = \frac{f(t)}{g(t)} = \frac{t}{\log(1+t)}$. Then we can calculate the derivative of $h(t)$ as:

$$\frac{dh(t)}{dt} = \frac{\log(1+t) - \frac{t}{1+t}}{\log^2(1+t)} \approx \frac{\log(1+t) - 1}{\log^2(1+t)}$$

When t is greater than 1, the derivative is greater than 0 and converge to $\frac{1}{\log(1+t)}$, which means that:

$$\lim_{t\to\infty} \frac{f(t)}{g(t)} = \frac{\log^* logn}{\log\log^* n} = \infty$$

when $t = \log^* \log n$. Proof done.

3. The minimum number of weighing is 2, and the pseudocode is shown below

Algorithm 4: 8 balls weighing algorithm

Input : 8 balls $a_1 \dots a_8$, and one ball is lighter while others are of same weight
Output: The light ball

```
1  Function weighBalls(a_1, ..., a_8):
2      Split the 8 balls into three groups g_1 = (a_1, a_2, a_3), g_2 = (a_4, a_5, a_6), g_3 = (a_7, a_8);
3      Weigh g_1 and g_2;
4      if weight(g_1) = weight(g_2) then
5          Weigh a_7 and a_8;
6          if weight(a_7) < weight(a_8) then
7              return a_7
8          else
9              return a_8;
10         end
11     else if weight(g_1) < weight(g_2) then
12         Weigh a_1 and a_2;
13         if weight(a_1) = weight(a_2) then
14             return a_3;
15         else if weight(a_1) < weight(a_2) then
16             return a_1;
17         else
18             return a_2;
19         end
20     else
21         Do the same procedure as for g_1;
22     end
23 end
```

From the above algorithm, we can optimize the weighing number to 2.

Ex. 3 — Rubik's Cube

A Rubik's cube is a cube with six faces, each of different color, usually white, yellow, red, blue, green, orange. Then, each face is split into nine small squares, constructing three rows and three columns. Each face with one row can be rotated so that colors can be mixed, and create different permutations. The figure of a Rubik's cube (Fig. 1)

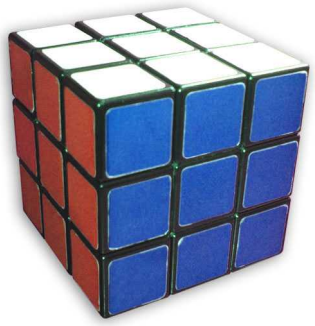

Figure 1: Rubik's cube

Solution Algorithms

To begin, first denote different symbols. Suppose that we face directly to a face of the cube. 'F' means rotating the front face clockwise for 90°. Also, 'L', 'R', 'U', 'D' denote for rotating the left, right, upper and downside face clockwise for 90°. When adding a $'$ to the symbol, it means counterclockwise.

Beginner's Method

Then, referring to the figure below (Fig. 2)

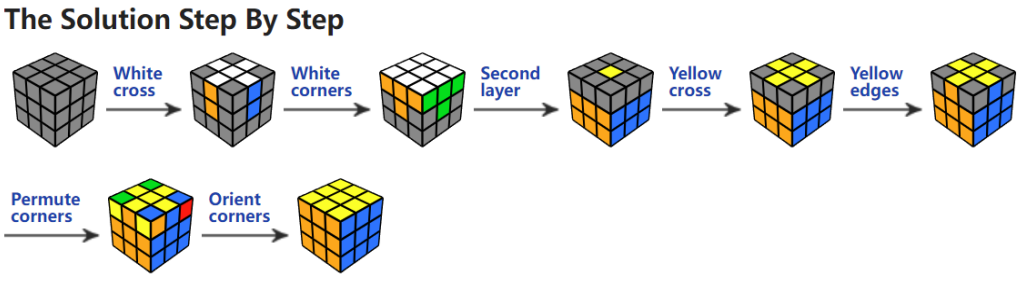

Figure 2: Solution for beginners. [1]

- Firstly, build a white cross at the top layer, and fit each corner with the correct corresponding color to each face by using $R'D'R$, then turn the top layer to the bottom.
- Then, to finish the second layer, in two different situations to replace a block with another, we can use $U'L'ULUFU'F'$ or $URU'R'U'F'UF$.
- Then we can use the formula $FRUR'U'F'$ to build a yellow cross in the top layer.
- After that, use $URU'L'UR'U'L$ to fit the corners.
- Finally, use $R'D'RD$ to get the cube all done.

Zbigniew Zborowski (ZZ) Method

This method is originally proposed by Zbigniew Zborowski in 2006, which is comparatively faster than CFOP. Refer to the figure below (Fig. 3):

(a) EOLine (b) F2L (c) LL

Figure 3: Procedure of ZZ [2]

- First, one should fit a line at the bottom face of a cube, which is the EOLine part (Fig. 3a).
- In the F2L stage, one should complete the first two layers by building two 1x2x3 blocks on either side of the Line made in the previous stage. Since all edges are oriented, we just need R, U and L moves to complete the stage (Fig. 3b).
- Finally, we can just simply apply methods done on the top layer, and will get the cube recovered (Fig. 3c).

References

[1] "How to solve the Rubik's Cube," How to solve the Rubik's Cube - Beginners Method. [Online]. Available: `https://ruwix.com/the-rubiks-cube/how-to-solve-the-rubiks-cube-beginners-method/`. [Accessed: 19-Oct-2021].

[2] C. Rider, ZZ Method Tutorial. [Online]. Available: `http://cube.rider.biz/zz.php`. [Accessed: 19-Oct-2021].

Ex. 4 — The $\mathcal{NP}$ class

A problem is in $\mathcal{NP}$ if and only if given a certificate, the correctness of the answer can be determined with a polynomial time algorithm. So considering the three problems:

1. Finding a simple path is equivalent to find a Hamiltonian path. Set the certificate to be a simple path generated by topological sort, It can be decided in polynomial time whether it is a simple path (according to Ex1.4). Therefore, it is $\mathcal{NP}$.

2. Checking whether an integer is composite is in $\mathcal{NP}$ class. The certificate can be a small integer i, suppose that the given integer is n, we can simply use Euclidean algorithm to check whether it is prime or not.

3. The vertex cover determination problem is also in $\mathcal{NP}$ class. Set the certificate to be a vertex cover of size k. Then we just need to check in the given graph $G = \langle V, E\rangle$, for each $e = \langle u, v\rangle \in E$, whether all u and v are in the vertex cover. It only takes time complexity of $\mathcal{O}(E)$.

Ex. 5 — *PRIMES* in $\mathcal{P}$

According to the **Prime Number Theorem**, given an integer x, there are about $\pi(x)$ primes less than it, where $\pi(x)$ is defined as:

$$\pi(x) \approx \frac{x}{\log x}$$

Trivial division can be simplified in the way that if N is given to test its primality, we just need to take the primes less than or equal to$\sqrt{N}$ into account, since if we can find a prime factor larger than $\sqrt{N}$, you can always find a corresponding prime less than it. However, it doesn't mean that trivial division only have the time complexity of $\mathcal{O}(\sqrt{n})$.

For example, let's assume that we need to check the primality of an integer x with n binary digits. Since we just need to take $\sqrt{x}$ into account, which is $2^{n/2}$, then according to the prime number theorem, we will have

$$\pi(2^{\frac{n}{2}}) = \frac{2^{\frac{n}{2}}}{\frac{n}{2}\log 2}$$

primes to be taken into account, will is not bounded in polynomial, but exponential.

Therefore, we can not prove that *PRIMES* is in $\mathcal{P}$. Furthermore, one of the fastest algorithm to determine a number's primality, the quadratic sieve algorithm, can run with time complexity $e^{(1+\mathcal{O}(1))\sqrt{\ln n \ln\ln n}}$, which is still not pseudo polynomial.