CPT_S 515: Midterm #1

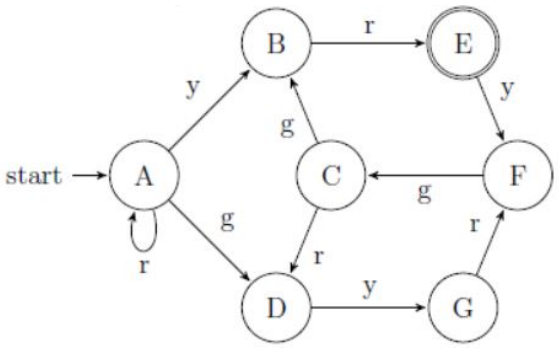
Instructor: Zhe Dang

Sheng Guan

Problem 1

1. Problem 1. Consider a directed graph G where each edge is labeled with one of three colors: green, yellow and red. The graph has a designated initial node and a designated final node. A walk of the walk refers to a path of G that starts with the initial node and ends with the final node. Notice that the path may contain loops and hence its length is unbounded.

(1). Try your best to design an algorithm that can decide (yes/no) on whether there is a walk (from initial to final) on which the number of green edges, the number of yellow edges, and the number of red edges are all equal. This is a very difficult problem therefore a partial solution is also worthwhile for submission.
Please also run your algorithm or even ideas of the algorithm on the following graph (notice that your algorithm should work on general graphs, not just this graph. Below, double curcled node is the final, while the start node is the initial.):

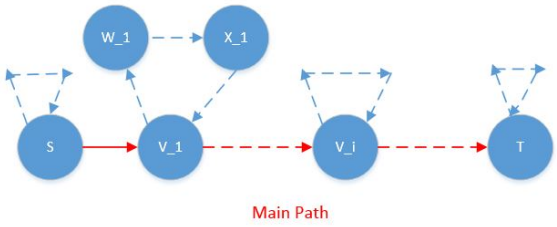


(2). The following is an even harder problem. Give me some thoughts on designing an algorithm that decides (yes/no) on whether there are infinitely many walks such that, on each such walk, the number of green edges, the number of yellow edges, and the number of red edges are all equal.

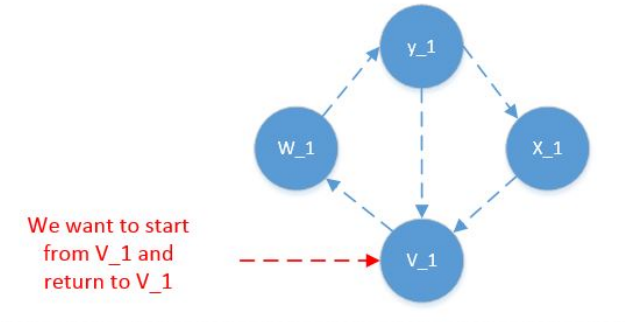
Answer:
(1)
For this question, I have two ideas to solve. The first idea is only related to the existence problem and I consider it as a naive solution; the second idea is related to the problem 1-(2).
Idea 1:
We use the idea of Turing machine definition. we use a stack to keep the records of red edge, yellow edge and green edge. We run the DFS from the starting point in the graph, and there's a tape to record the passed edge, starting point, ending point to keep the record of the path. In the stack, when we encounter a red edge, we push the red edge into the stack. Similarly, we push the yellow edge, green edge into the stack. We keep an eye on the stack, when red edge, yellow edge, green edge exist at the same time in the stack and we don't have the need to roll back(we will talk about roll back later), we eliminate the red edge, yellow edge and green edge in the stack. When the stack is empty, we check whether the current point is the ending point as the input. If not, we let the algorithm keep going. If yes, we just identify one path that meets the requirement and the algorithm stops. Now we talk about the roll back procedure. When we reach a point which has no outgoing edge that means we go to the dead end, we should reverse the direction that kept in the last tape and roll back to the earlier state, then possibly we will according to the DFS to choose another outgoing edge; If the earlier point also has no other outgoing edge, we will continue this roll back procedure until we can find a different outgoing edge that we didn't choose before.
The drawback of this algorithm is clear that it will continue running unless there's a walk meets the requirement then it will stop. When the algorithm is running, you can never say whether such a walk exists or not, in other words, algorithm cannot decide no. One possible approach to enhance this naive solution is we

keep a data structure to organize the cycle (staring point and ending point is the same) the algorithm has already passed, and we treat a cycle as a big node to update the stack. Still this improvement is not good enough, then I have the idea 2.
Idea 2:
(1)Always, let's consider the simplest case as Case 1 first.

- Case1. If from the start node to the final node is a simple path where any node is not the part of any cycle but the path does exist. We only need to scan this path and accumulate the edge information. Let's say the number of red edges is r_1, the number of yellow edges is y_1, the number of green edges is g_1. In the end we do the comparison, if $r_1 = y_1 = g_1$, then the algorithm decides yes. Otherwise, the algorithm decides no. To get such this path, we can run DFS with a hashset to make sure that duplicated nodes won't be appeared in this path.
- Case2. If from the start node to the final node, there's a path where any visited node on the path has a cycle as shown in Figure 1-1. We can run a DFS algorithm first with a hashset to make sure that any node won't be visited twice in the found path and when the DFS algorithm reaches the final node, the algorithm terminates. In this case, we can get a path which we call it the Main Path. then along this Main Path, actually we already can move from the start node to the final node. However, the cycles on the Main Path offers us more flexibility to add red or yellow or green edges in the final path to meet the requirement that the number of them must be equal.



Then we run the SCC algorithm on the graph to detect all the cycles shown as Figure 1-1. Since SCC algorithm will only group together all the points into a set where in such set every point can reaches everyone. the SCC algorithm doesn't necessarily guarantee this set of nodes has only one cycle path starting from A and returning to A, we need to further consider the below scenarios and we use the Figure 1-2 here as an example to illustrate our idea.

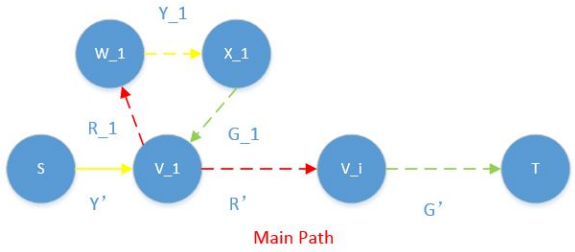


In this example the SCC algorithm will group v_1, w_1, y_1,x_1 together. However, we need to consider $\{v_1, w_1, y_1\}$, $\{v_1, w_1, y_1, x_1\}$these two cases. We can run DFS starting from v_1, this time we use a

two-level hashset to make sure that in the second level, the set DFS traversed is unique and every edge is traversed only once. In this way, we will get $\{v_1, w_1, y_1\}$ and $\{v_1, w_1, y_1, x_1\}$ respectively. At the same time, we will use a three-element tuple to store the number of edges for red, yellow, green of the set $\{v_1, w_1, y_1\}$ and $\{v_1, w_1, y_1, x_1\}$ correspondingly. We will assign x_1 for the first three-element tuple, x_2 for the second three-element tuple.x_i here means the number of cycles should be run in the final path. After this, we continue this process for all points along the Main Path which are involved within cycles, we will get a n-by-1 matrix x_i and 3-by-n matrix {R,Y,G}shown as Figure 1-3. We will try to formalize this problem as a linear programming problem.

	R	Y	G
x_1	1	1	1
x_2	2	1	1
...	...	...	...
...	...	...	...
...	...	...	...
x_n	3	4	5

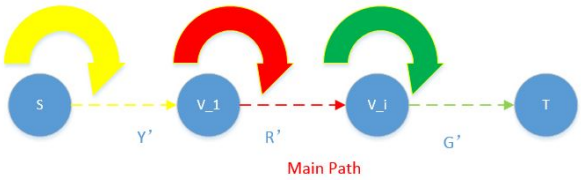
We transpose {R,Y,G} as 3-by-n matrix first we call it A, then for $A \bullet x$, if we can find a bound y_i for x_i, we can formalize this problem as a linear programming problem. To illustrate our idea about how to set the bound y_i, we use another example shown in Figure 1-4.



Here we only consider one possible cycle that would be appeared in the final path and the Main Path at one time. Suppose now in the Main Path, we will get a tuple $\{R', Y', G'\} = \{5, 3, 2\}$, which means we need contributions from cycles as {2,3} extra for yellow edges and green edges.
$3 = max\{\|5-3\|, \|5-2\|, \|3-2\|\}$
Ideally, we hope in this cycle we can make the green edge and red edge equal. For example if we get the $cycle_1$ and the tuple value is $\{R_1, Y_1, G_1\} = \{1, 2, 2\}$. for this cycle, it will contribute 1 more for green edge if we run it once. So, we at most run it three times to make the number of red edges equal to the number of yellow edges. We should not run it more than 3 times since it will re-imbalance the objective.In this case,y_1 as 3. However, sometimes the cycle will contribute to another edge in this case the yellow edge. For example,if we get a tuple $\{R_1, Y_1, G_1\} = \{1, 2, 1\}$, for this cycle, it will contribute 1 more for yellow edge if we run it once. So, we at most run it twice to make the number of red edges equal to the number of yellow edges instead. In this case,y_1 as 2. For the green edge matter, we will look for another cycle. In other words, we can use a greedy approach that we always maximize our effort to make it equal among two variables (R,Y,G) and leave the third one along to set the bound y_i by only considering current cycle and Main Path difference. If we cannot make it tend to balance. For example, if we get $\{R_1, Y_1, G_1\} = \{2, 1, 1\}$ this time, running this cycle will not be helpful for the objective, we set the bound y_1 as 1. In this way, we also can get y_2, ...,y_n (total n cycles). Then if

we do $A \bullet y$, we will get b, If we make $C^T \bullet x = sum_{i=1}^{n} x_i$ by letting C is full of 1s, then the existence problem will become an linear programming problem that we would like to know $maxC^T \bullet x$ with the constraints that
$A \bullet x \leq b$,
$R' + A_1 \bullet x = Y' + A_2 \bullet x$,
$R' + A_1 \bullet x = G' + A_3 \bullet x$,
$Y' + A_2 \bullet x = G' + A_3 \bullet x$,
$x \geq 0$.
where A_i is the i^{th} row of the matrix A. We can further use two inequality equation to replace the equality equation in the last three rows.
$R' + A_1 \bullet x \leq Y' + A_2 \bullet x$,
$Y' + A_2 \bullet x \leq R' + A_1 \bullet x$,
$R' + A_1 \bullet x \leq G' + A_3 \bullet x$,
$G' + A_3 \bullet x \leq R' + A_1 \bullet x$,
$Y' + A_2 \bullet x \leq G' + A_3 \bullet x$,
$G' + A_3 \bullet x \leq Y' + A_2 \bullet x$,
We run the linear programming algorithm, then if we can get one set of $\{x_i\}$ to solve the objective function $maxC^T \bullet x$, we decide yes.
Worth to mention that in the end , since the DFS with a hashset algorithm will give us more than one Main Path. For each Main Path, we need to repeat the above procedure. In the end, we have a outer loop to loop for all the possible Main Paths. If for all the corresponding linear programming problems we cannot find a solution, we decide no and the algorithm terminates

Applying our idea 2 algorithm on the Problem 1's figure. Because A contains a self-loop, the scenario 1 doesn't satisfy. In scenario 2, for the Main Path A $\rightarrow$ D $\rightarrow$ G $\rightarrow$ F $\rightarrow$ C $\rightarrow$ B $\rightarrow$ E, we will first get one SCC {B,E,C,F,D,G}. However, the further DFS with hash structure will get us {B,E,C,F} ,{C,D,G,F}, {A}to form cycles and we will get $\{R', Y', G'\} = \{2, 1, 3\}$ for the Main Path, However, to solve the linear programming problem we cannot satisfy all the constraints, which means along this Main Path A $\rightarrow$ D $\rightarrow$ G $\rightarrow$ F $\rightarrow$ C $\rightarrow$ B $\rightarrow$ E, the walk doesn't exist.
Then our algorithm will find another Main Path A $\rightarrow$ B $\rightarrow$ E, {B,E,C,F} , {A}will form cycles.We will get $\{R', Y', G'\} = \{1, 1, 0\}$ for the Main Path. {B,E,C,F} will make the cycle to contribute one more to green edge. The main path needs 1 for the green edge. We have the bound as 1 for cycle {B,E,C,F}. Applying the linear programming, we get the solution that A $\rightarrow$ B $\rightarrow$ E $\rightarrow$ F $\rightarrow$ C $\rightarrow$ B $\rightarrow$ E, for this Main Path, the walk exists.
Our algorithm will output yes in the end.
(2)
Continue with the above idea (2), from the problem 1-(1), we have already found a path that satisfies the requirement that the number of green edges, red edges and yellow edges are equal. However, slightly different that in problem 1-(1), the algorithm terminates when we find the first satisfied path if any. In problem 1-(2), the algorithm needs to enumerate all the possible satisfied paths. Then for each path i,
We can use this path as the new Main Path we defined earlier. Then in this Main Path, $R' = Y' = G'$. Again we use an example to illustrate our idea as shown in Figure 1-5.

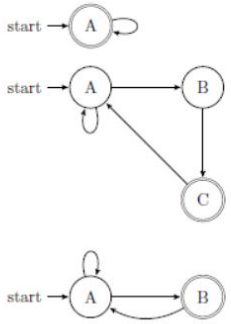


In this example, we can see after the Main Path satisfies the requirement, if we can find cycles combination such that
$thenumberofrededges = thenumberofyellowedges = thenumberofgreenedges$
then we can start from the Main Path and go into this combination and do the cycling n times, we still satisfy the requirement. As Figure 1-5 shows, this is a very special case. Each cycle here only contains 1 edge with 1 color, we need to combine three cycles to make
$thenumberofrededges = thenumberofyellowedges = thenumberofgreenedges$, then we can repeat cycling the Red, Green,Yellow cycle n times at the same time along with the Main Path, we will get n groups of Paths and all n groups results satisfy the requirement. Hence, based on the algorithm we proposed before in idea 2, after we determine the Main Path, we can still do the SCC and DFS with two-level hashset to get all the small cycles we need to consider. Here we assume we in total get the m cycles in the end. Then we brute-force all the combinations of these m cycles starting from C_m^1 , C_m^2, ... , ...,C_m^m. If in one combination, we calculate that the
$thenumberofrededges = thenumberofyellowedges = thenumberofgreenedges$
then there are infinitely many walks such that, on each such walk, the number of green edges, the number of yellow edges, and the number of red edges are all equal since we can adjust the looping number for this combination, the algorithm decides yes. Otherwise, the algorithm needs to go through all possible paths $i = 1, 2, ..n$ we get from problem 1-(1), and if no such combination exists, the algorithm decides no.

Problem 2

2.

Problem 2 (Bonus). I use (G, v, u) to denote a directed graph G where v and u are two designated nodes. Clearly, when G has loops, the number of paths from v to u, written $\#(G, v, u)$, can be ∞. This causes a problem. If we are given two graphs (G_1, v_1, u_1) and (G_2, v_2, u_2) and if it happends to be the true that $\#(G_1, v_1, u_1) = \infty$ and $\#(G_2, v_2, u_2) = \infty$, then, in mathematics (since both infinities are countable infinity), it really says that $\#(G_1, v_1, u_1)$ and $\#(G_2, v_2, u_2)$ are the same. Apparently, there is a better way to say that, even though both are infinity, it is indeed true that one contains more path than the other. That is, you are asked to define and show an algorithm to compute a real-valued function $\lambda(G, v, u)$ such that $\lambda(G_1, v_1, u_1) > \lambda(G_2, v_2, u_2)$ if (G_1, v_1, u_1) ***has more paths than*** (G_2, v_2, u_2) ***does*** (you need define the meaning of the boldface sentence!). (Hint: consider the following three graphs where the start points to the v and the double circled node is the u. Which one is likely to have the most paths, though all have infinitely many paths? It is the last one for sure. But why?)



Answer:

If we assign each edge a variable and use regular expression to express this problem according to the three graphs below,

The first is a^*;

The second is $(a^*bc(da^*bc)^*)$

The third is $(a^*b(ca^*b)^*$

The rough idea is we use the symbolic graph representation. If for one node, we fix the level of bits we need to represent this node, For example, here we can choose level of bits as 2. In the reality, we choose m that $2^m = number of nodes in the graph$ A $\rightarrow$ 00, B $\rightarrow$ 01, C $\rightarrow$ 10 by using two variables x_1 and x_2. If we increase the level of bits by 1, we need to add one more variable x_3.At the same level of bits representation (number of variables x_i),my intuition to define the number of paths in (Gi; vi; ui) is suppose we have a boolean formula representation R, then to make this representation R stays true, the possible values choice means the number of paths, we call it NU(Gi; vi; ui), then for the below three pictures we will have

$NU(G1; v1; u1) < NU(G2; v2; u2) < NU(G3; v3; u3)$

In this case, figure 1 can be represented as 4 pictures; figure 2 can be represented as 2 pictures; figure 3 can be represented as 3 pictures. Then we can use the function of transitive closure to get the boolean formula representation R. We define (G, v, u) with transitive closure algorithm:

Step1: Transfer the graph G to symbolic format.

Step2: Use loop to compute transitive closure R^* for R. Set H as R firstly.

Step3: Let $H = H$.

Step4: Let H as $H \cup (H \circ R)$

Step5: Simplify H.

Step6: If H doesnt equal H, repeat steps 3-5.

Step7: H is R^* now, return H

After we get the R, we enumerate all the possible choices for Variables $X = \{x_1, x_2\}, Y = \{y_1, y_2\}$ and the number of choices correspondingly. If at the same level, we will get :

$NU(G1; v1; u1) < NU(G2; v2; u2) < NU(G3; v3; u3)$