CS/ECE 374 B Fall 2019
Homework 7 Problem 1

Jin Yucheng (yucheng9)
Fan Wenyan (wenyanf2)
Lu Hanxiao (hl4)

Solution:

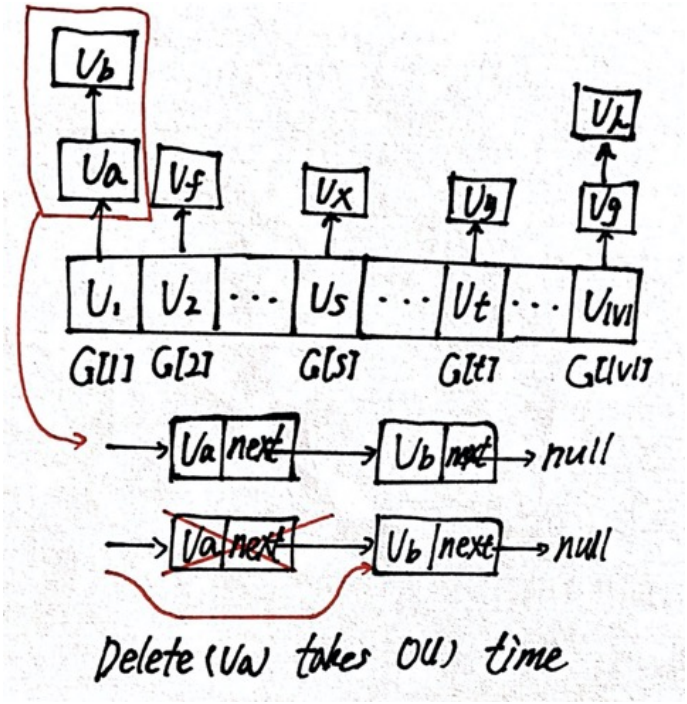
(a) For simplicity, denote the starting vertex s as v_s, and the destination vertex t as v_t, a vertex v_i can represent both the vertex itself and the subscript i.

The general strategy is to construct a new directed graph G', and call BFS/DFS on G'.

Suppose v_j is an arbitrary vertex in G, and v_k is one of its neighbors, $v_j \to v_k$ will be preserved in G' if and only if:

- v_j % 3 = 0 (labeled as 3) and v_k % 3 = 1 (labeled as 7)
- v_j % 3 = 1 (labeled as 7) and v_k % 3 = 2 (labeled as 4)
- v_j % 3 = 2 (labeled as 4) and v_k % 3 = 0 (labeled as 3)

After G' is constructed as a new adjacency list, if v_s is labeled as 3 and there is a path from v_s to v_t, then all vertices in this path must follow the above rules, which ensures the labels of vertices to follow the 374374... pattern.



The above figure shows the data structure of a adjacency list, from $G[1]$ to $G[|V|]$, $G[i]$ stores the neighbors of a vertex v_i, and $G[i]$ itself is a linked list. To delete an element in a linked list takes $O(1)$ time since we just need to change the pointer of its predecessor to point its successor (or null at the end of the linked list). Suppose there is a subroutine Delete(v_i) that can delete v_i from a linked list.

The algorithm to construct a new directed graph G' works as follows,

def ConstructDirectedGraph(G):

```
1 : for i ← 1 to |V|:
2 :     G'[i] ← G[i]
3 :     CurrentState ← v_i % 3
4 :     for each v_j in G'[i]:
5 :         NextState ← v_j % 3
6 :         if CurrentState = 0:
7 :             if NextState ≠ 1: Delete(v_j)
8 :         else if CurrentState = 1:
9 :             if NextState ≠ 2: Delete(v_j)
10 :        else:
11 :            if NextState ≠ 0: Delete(v_j)
12 : return G'
```

It takes $O(V+E)$ space to store G', and it takes $O(V+E)$ time to construct G', since we iterate through all vertices and all edges.

After G' is constructed, we can implement either BFS or DFS to find if there is a path from v_s to v_t, let's take DFS:

def DFS(G', v_s, v_t):

```
1 : if v_s % 3 ≠ 0: return FALSE
2 : else:
3 :     Push(v_s)
4 :     while the stack is not empty:
5 :         v_i ← Pop
6 :         if v_i = v_t: return TRUE
7 :         else:
8 :             if v_i is unmarked:
9 :                 Mark(v_i)
10 :                for each edge v_i → v_j: Push(v_j)
11 : return FALSE
```

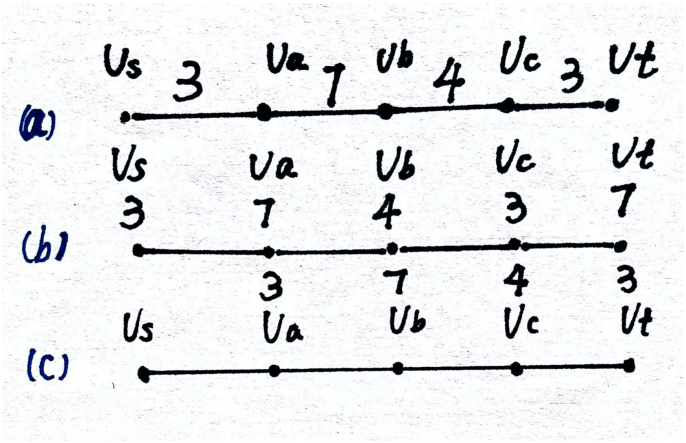
In the worst-case, DFS takes $O(V+E)$ time to iterate through all edges and vertices.

Combine the above two subroutines, it takes both $O(V+E)$ time and space to find if there is a path that follows the 374374... patern from s to t.

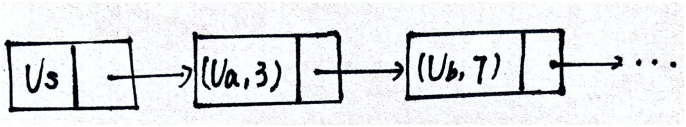
def SearchPathPart1(G, s, t):

```
1 : G' ← ConstructDirectedGraph(G)
2 : return DFS(G', s, t)
```

(b)



The above figure shows how we can connect the first part of Problem 1 and the second part of Problem 1. Case (b) shows exactly an example in the first part where each vertex is labeled with a digit, case (a) shows how each edge is labeled with a digit, case (c) shows how we can align (b) and (a), namely, we do not label the first vertex, and assign v_a with label 3, assign v_b with label 7, and so forth.



The data structure is also an array of linked lists, the above figure shows the linked list of v_s. Assume there is a subroutine GetEdge(v_i, v_j) that can get the label of edge (v_i, v_j). The first step is to construct a new graph $G^{'}$ that labels vertices, the algorithm is defined as follows (suppose v_s is automatically labeled as 0):

def LabelVertices(G, v_s):

1 : Push(v_s)
2 : $v_s \leftarrow 0$
3 : **while** the stack is not empty:
4 : $v_i \leftarrow$ Pop
5 : **for** each v_j that is adjacent to v_i:
6 : $v_k \leftarrow$ GetLabel(v_i, v_j)
7 : Push(v_j)
8 : Add v_k to $G^{'}$
9 : return $G^{'}$

Then, combine it with the algorithm in the first part,

def SearchPathPart2(G, s, t):

```
1 : G' ← LabelVertices(G, s)
2 : return SearchPathPart1(G', s, t)
```

To construct a new graph that labels vertices will take both $O(V+E)$ time and space, as we've discussed above, it takes both $O(V+E)$ time and space to search a path from s to t, so both the time and space complexities for part 2 are also $O(V+E)$.

(c) This time, we need to construct an array that stores all paths from s to t, and check if there is a path in this array that matches the regular expression r.

def AllPaths(G, v_s, v_t):

```
1 : Construct an empty stack that temporarily stores paths
2 : Construct an empty array that stores all paths as AllPaths
3 : Push(v_s)
4 : while the stack is not empty:
5 :     Path ← Pop
6 :     Node ← the last vertex of Path
7 :     if Node = v_t: Add Path to AllPaths
8 :     for each v_i that is adjacent to Node:
9 :         NewPath ← Path + v_i
10 :        Add NewPath to the stack
11 : return AllPaths
```

It takes $O(V+E)$ time to find all paths, after we have found all paths. We constructed a backtracking algorithm that decides whether an input string w matches a parsed regular expression r in Homework 5, Problem 1. For a fixed regular expression, it takes $O(n^2)$ time to check whether an input string matches r, n is the length of the string. We can use the same logic and treat the paths like strings, so it takes $O(E^2)$ time in the worst-case.

MatchRegex($w[1...n], r$):

```
1  : if r[0] = ·:
2  :     for i ← 1 to n:
3  :         if MatchRegex(w[1...i], r[1]):
4  :             return MatchRegex(w[i+1...n], r[2])
5  :         return FALSE
6  : else if r[0] = +:
7  :     if MatchRegex(w[1...n], r[1]):
8  :         return TRUE
9  :     return MatchRegex(w[1...n], r[2])
10 : else if r[0] = *:
11 :     for i ← 1 to n:
```

12 : **if** MatchRegex($w[1...i], r[1]$):

13 : return MatchRegex($w[i+1...n], r[1]$)

14 : return FALSE

15 : **else**: # base case happens when the string could no longer be divided

16 : return w = $r[0]$ # return TRUE if the character matches a ($a \in \Sigma$ or ϵ)

Combine the above two subroutines,

def SearchPathPart3(G, s, t):

1 : AllPaths $\leftarrow$ AllPaths(G, s, t)

2 : **for** each Path in AllPaths:

3 : **if** MatchRegex(Path, r): return TRUE

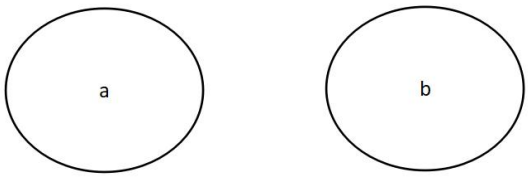
4 : return FALSE

Totally it takes $O(E^2 \cdot (V+E))$ time.

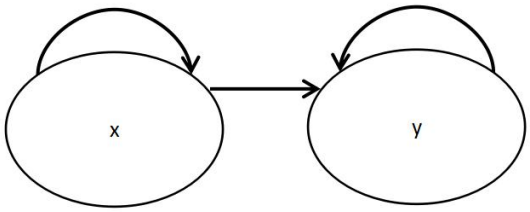
■

2.

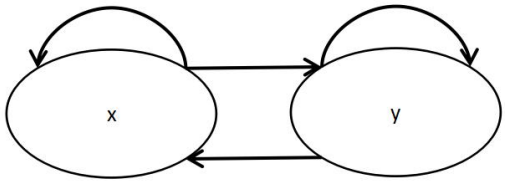
(a) We need to build a dependency directed graph that shows how each variable on the left hand side is related to variable on the right hand side for both part a and part b. For instance, for expression a,b =1,0 , a is related only to a constant and b is also only related to a constant. Therefore the dependency directed graph is disconnected directed graph shown below, where vertices show variables:



For expression x,y = x+1,x+y, the x on the left hand side is related to x itself and y on the left side is related to x and y. Therefore the dependency directed graph is shown below, where vertices show the variables and arrow edges show dependency:



For expression x,y = x+y,x-y, the x on the left hand side is related to x and y and y on the left hand side is also related to x and y. Therefore the dependency directed graph is shown below, where vetices show the variable and edges show dependency:



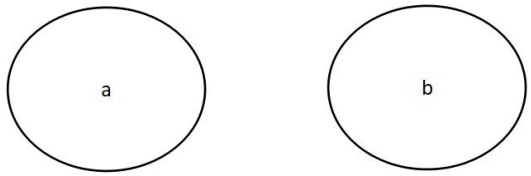
To detect whether we need additional temporary variables, we only find whether there is a loop exited in the dependency directed graph. We can use the following algorithm to find whether there is a loop:

```
Def Isacyclic(G)
    For all vertices v in G:
        v.status <- New
    For all vertices v in G:
        If v is new:
            If Isacyclicdfs(v) == False
                Return false
    Return True
```

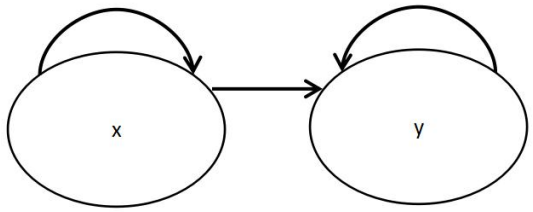
```
Def Isacyclicdfs(v):
    v.status <- Active
    For all edge v->w:
        If w==v:
            Ignore since it is self looping
        if w.status == Active
            Return false
        Else if w.status == New
            If Isacyclicdfs(w) == Flase
            Return false
    V.status <- finished
    Return true;

    Main ():
        Construct the graph G according to the above logic
        If Isacyclic(G) == true
            Return true
        Else
            Return false
```

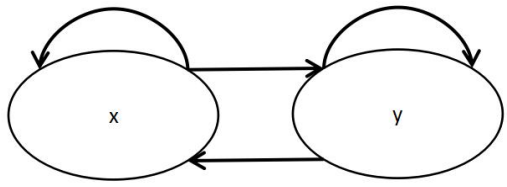
(b) We need to build a dependency directed graph that shows how each variable on the left hand side is related to variable on the right hand side for both part a and part b. For instance, for expression a,b =1,0 , a is related only to a constant and b is also only related to a constant. Therefore the dependency directed graph is disconnected directed graph shown below, where vertices show variables:



For expression x,y = x+1,x+y, the x on the left hand side is related to x itself and y on the left side is related to x and y. Therefore the dependency directed graph is shown below, where vertices show the variables and arrow edges show dependency:



For expression x,y = x+y,x-y, the x on the left hand side is related to x and y and y on the left hand side is also related to x and y. Therefore the dependency directed graph is shown below, where vetices show the variable and edges show dependency:



To detect whether we need exactly one additional temporary variables, we only find whether there is only one loop exited in the dependency directed graph. We can use the following algorithm to find whether there is exactly one loop:

```
Def Isacyclic(G)
    For all vertices v in G:
        v.status <- New
    For all vertices v in G:
        If v is new:
            If Isacyclicdfs(v) == False
                Return false
    Return True

Def Isacyclicdfs(v):
    v.status <- Active
    For all edge v->w:
        If w==v:
            Ignore since it is self looping
        if w.status == Active
            Count_number++;
        Else if w.status == New
            If Isacyclicdfs(w) == Flase
            Return false
    V.status <- finished
    Return true;

Main():
    Construct the graph G according to the above logic
    Count_number = 0;
    Isacyclic(G);
    If (Count_number == 1)
        Return true
    Else
        Return false
```

Homework 7 Problem 3

Jin, Yucheng (yucheng9) Fan, Wenyan (wenyanf2) Lu, Hanxiao (hl4)

November 12, 2019

Solution

Let G = (V, E) denote a dag.

We reduce the problem to depth-first search in the following directed graph:

V contains vertices from all n points in the set S.

E contains directed edges from (x, y) → (x', y') if x < x' and y < y'.

The edges are weighted with $L_{ENGTH}(x, y, x', y')$.

Let llmip[i] denote the **L**ength of the **L**ongest **M**onotonically **I**ncreasing **P**ath (LLMIP) beginning with the vertex $v_i = (v_i.x, v_i.y)$. The values of $v_i.x$ and $v_i.y$ are defined by the index i in the arrays X[1 .. n] and Y [1 .. n].

To solve the problem, we can compute the length of the longest monotonically increasing path by computing the length of the monotonically increasing path from each vertex, and then get the largest one of them.

To reduce the time complexity, we can use DFS to do topological sort before computing the length of the monotonically increasing path from each vertex, and dynamic programming when computing the length of the monotonically increasing path from each vertex.

```
def LENGTHLONGESTMONOTONICALLYINCREASINGPATH(G):
    TOPOLOGICALSORT(G)
    for each vertex v_i ∈ V in reverse Topological Order:
        length = 0
        for each vertex v_j after v_i in Topological Order:
            # llmip[j] denote the length of the monotonically increasing path from j
            length = max((LENGTH(v_i.x, v_i.y, v_j.x, v_j.y) + llmip[j]), length)
        # llmip[i] denote the length of the monotonically increasing path from i
        llmip[i] = length
    # max(llmip) denote the length of the longest monotonically increasing path
    return max(llmip)
```

We can evaluate this recursive function in $O(V + E) = O(n + n^2) = O(n^2)$ time by performing a depth-first search of S. The dynamic programming runs in $O(n^2)$ time for the same reason. The operation of max(llmip) runs in O(n) time. Thus, the algorithm runs in $O(n^2)$ time.