COSC2430: Programming and Data Structures
Graph Algorithms: Reachability from a source vertex.

1 Introduction

Reachability from a source vertex s is the problem to find the set of vertices S such that $v \in S$ if exists a path from s to v. You will create a C++ program to find reachable vertices using two alternative approaches: a) Iteration of vector-matrix multiplications. b) Depth First Search. In this assignment, you will exploit several data structures learned in this course, including arrays, linked lists, binary search trees, hash tables, and stacks.

2 Theory

Let $G = (V, E)$ a directed graph, where n vertices and m edges. G can be represented as an adjacency matrix E, $n \times n$. We will manipulate E as a sparse matrix, as done in previous homeworks.

2.1 Reachability from a source vertex with vector-matrix multiplications

Represent the set of reachable vertices from a source s as a row vector S, such that $S[i] = 1$ if i is reachable from s, 0 otherwise. Algorithm 1 compute the reachable vertices for path of length $0, 1, 2...k$.

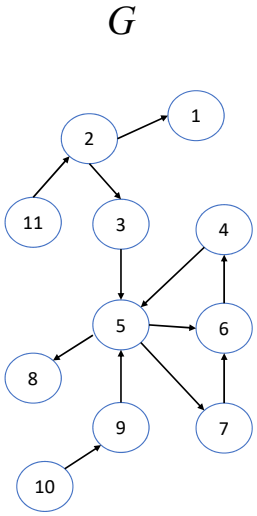
Data: E, source vertex s, max path length k
Result: S
Initialization $S[s] \leftarrow 1$;
`/* Iterations */`
$d \leftarrow 0$;
while $d < k$ **do**
  $d \leftarrow d + 1$;
  $S \leftarrow S \vee (S \cdot E)$;
end
return S

Algorithm 1: Reachability from a single source vertex with vector-matrix multiplication

In Phase 1 of this homework, you will compute iteratively the vector of reachable vertices from a source vertex s , for paths of length 0,1,2 .. k. Your computation shall be based on Algorithm 1.

2.2 Reachability from a source vertex with depth-first search

Depth-first search is a fundamental graph traversal algorithm that has as output the set of vertices reachable from a source vertex s. Please see the textbook for details.



i	j	v
2	1	1
2	3	1
3	5	1
4	5	1
5	6	1
5	7	1
5	8	1
6	4	1
7	6	1
9	5	1
10	9	1
11	2	1

Figure 1: The adjacency matrix E (sparse representation) for a sample graph G

3 Program and input specification

Phase 1:Reachability from a source vertex with vector-matrix multiplications and recursive DFS

Specifically, the input for your program will be the E , the adjacency matrix of G. E shall be manipulated as a sparse matrix. You can consider S as a sparse matrix $1 \times n$. The input is one text file, which contains E, the adjacency matrix of G, in sparse representation. Thus, the input text file will have entries with three values (i, j, v), not necessarely sorted. Note v is always 1. Comments have a # at the beginning of the line. In each iteration your program must perform the vector-matrix multiplication $S_d \cdot E$. After the multiplication, $S[i]$ contains the number of paths from s to i. Although $S[i]$ might be greater than 1, you are require to consider entries as 1 or 0 only.

When the iterative process stops, you shall generate as the output the set of all vertices reachable, for path with length 1 up to$k$. The set of reachable vertices for a path with length $1, 2...$ up to k edges is computed in each iteration as $S = S \vee (S \cdot E)$. NOTE: The vector matrix product $S \cdot E$ computes the vertices reachable for path of length k ONLY. Additionally, your program shall find the reachable vertices from a specific source vertex, using Depth First Search (DFS). In Phase 1, you must program DFS in recursive way. The output shall be generated to a different file.

Phase 2:Reachability from a source vertex with DFS.

In Phase 2, DFS must be programmed with and without recursive calls (iterative process). In the second case, you should implement a stack, to push and pop vertex-ids. In Phase 2, along with E you will be provided of a text file with the labels of the vertices (for instance, user names in a social network), and the output file will show both the vertices IDs and the labels. You should implement a hash table to access the labels in $O(1)$ time.

3.1 Input file

In the transition matrix, vertices are identified by an integer id. An example of the input file is below:

```
# note corner of the matrix
#i j v
2 1 1
2 3 1
3 5 1
4 5 1
5 6 1
5 7 1
5 8 1
6 4 1
7 6 1
9 5 1
10 9 1
11 2 1
11 11 0
```

Example of vertex labels file

```
#id label
8 john.zhu
6 mary.lee
5 paul.diaz
7 ann.smith
3 someone
.
.
.
11 anyname.abc
```

3.2 Example: Vector Matrix Multiplication

This example is the first computation of $S \cdot E$ for the sample graph in Fig. 1.

```
Vector S                Matrix E
0 1 0 0 0 0 0 0 0 0 0   0 0 0 0 0 0 0 0 0 0 0
                        1 0 1 0 0 0 0 0 0 0 0
                        0 0 0 0 1 0 0 0 0 0 0
                        0 0 0 0 1 0 0 0 0 0 0
                        0 0 0 0 0 1 1 1 0 0 0
                        0 0 0 1 0 0 0 0 0 0 0
                        0 0 0 0 0 1 0 0 0 0 0
                        0 0 0 0 0 0 0 0 0 0 0
                        0 0 0 0 1 0 0 0 0 0 0
                        0 0 0 0 0 0 0 0 1 0 0
                        0 1 0 0 0 0 0 0 0 0 0

S * E =  1 0 1 0 0 0 0 0 0 0 0
```

3.3 Program call parameters

The main program should be called "reachability". Call syntax:

```
reachability "E=<file>;labels=<file>;result=<file>;source=<integer>;dfs=no|recursive|iterative;k=<integer>".
```

Notice the brackets < > indicate the parameter value and they are not part of the input string.

3.4 Output File

The requirements for the output text file are: Phase 1: In each line you must present the vertex id of the reachable vertices, in ascending order. Phase 2: In each line you must present the vertex id of the reachable vertices AND the lables. While the solution must be sorted when vector-matrix multiplication is applied, the output of the DFS algorithm shall not be sorted .

```
# Example for k=3 and source =2 for the sample graph (Phase 1)
# Program call
# reachability "E=A.txt;result=out.txt;source=2;dfs=no;k=3"
# Note that the iterative algorith includes the source  in the output
#vertex-id
1
2
3
5
6
7
8

# Example for k=4 and source =10 for the sample graph (Phase 1)
# Program call
# reachability "E=A.txt;result=out.txt;source=10;dfs=no;k=4"
# Note that the iterative algorith includes the source  in the output
#vertex-id
4
5
6
7
8
9
10

# Example for k=3 and source =2 for the sample graph (DFS Phase 2)
# reachability "E=A.txt;labels=L.txt;result=out.txt;source=2;dfs=iterative;k=3"
# Note: For  DFS do not sort the output; do not present source vertex.
#vertex-id
8 john.zhu
7 ann.smith
6 mary.lee
5 paul.diaz
3 someone
1 xxxx.yyyy
```

4 Data Structures

In phase 1, the main operation in each iteration is sparse matrix multiplication. You must store the matrix E as a single list of non-zero entries in one 1-dimensional array, representing a sparse $n \times n$ matrix. The the row vector S_d **might** be stored as a sparse $1 \times n$ matrix. Another option is to store S as a dense vector, i.e. declared as an one-dimensional array. You should use a hash table for fast access to the node labels (Phase 2). Depth First Search is required both in Phase 1 and Phase 2. For Phase 2, you must use your own stack data structure.

Example:

```
#no dfs; use vector-matrix multiplication; 3 iterations
reachability  "E= A.txt;result=S.txt;source=2;dfs=no;k=3".

# use dfs;
reachability  "E= A.txt;result=S.txt;source=2;dfs=yes;k=8".
```

5 Requirements

- Phase 1: Reachability computation from a source vertex with vector matrix multiplication and recursive DFS. Return all the reachable vertices, sorted by id.

  Phase 2: Reachability computation from a source vertex with Non-recursive Depth First Search. Implement your own stack for DFS and hash table to get the vertex description (vertex name). The output must include both the vertex ids an the labels.

- Storage: arrays of matrix entries: array of triples (i, j, v).

  Arrays: lists stored as arrays of non-zero elements. You can store the matrix as a single 1-dimensional array. You cannot store zero entries, except one corner of the matrix.

  Optional to get credit for HW2 (lists): Sparse Matrix , Sparse vector and result vector stored as linked lists.

- **Optional to get credit for HW3 (sorting):** Sorting Algorithms and Binary Search for fast access to the entries.

- C++ programming requirements:

  Sparse storage: you cannot store zeroes in your matrix data structure, other than the lower right corner a_{mn}. You cannot use simple 2-dimensional arrays to store the matrix (e.g. double A[10][10]). It is preferable your storage takes space $O(q)$, where q is the number of non-zero entries.

  The code of your hash table must be in a hash.h file. Likewise, the code of your stack implementation must be written in a stack.h file.

  Subscripts in input and output matrices are based on a standard mathematical definition, indexed from 1. Matrices of size $m \times n$ start on entry [1][1] and go up to [max(i),max(j)] $\forall a_{ij}, 1 \leq i \leq m, 1 \leq j \leq n$. However, in your C++ code you can use arrays indexed from [0], from [1] or indexed by your own functions or pointers making a subscript conversion, but the input/output matrices always have subscripts starting at 1,1.

- Input:

  One file per matrix. In general you should assume input matrix entries are not sorted, but you can make one pass to detect it.

- Output:

  Reachability with vector-matrix multiplication requires the output sorted by vertex id. Do not sort the outport for DFS. In Phase 2, the output requires both vertex ids and vertex labels, as in the previous example./

- Testing:

  Your program will be tested for dynamic memory allocation and memory leaks (new, delete operators correctly programmed). In other words, the TAs will verify you do not exploit 2-dimensional arrays. Optional items (counting credits for HW2 & HW2) will be graded manually, including source code inspection.

  The number of iterations k will not be greater than the number of edges. The vertices will have (at least) either one incoming edge or one outgoing edge.

  Notice the program will be tested with sparse matrices whose theoretical size goes up to 10000000×10000000 (i.e. they will have large subscripts in some entries). In other words, it will be impossible to use C++ 2-dimensional arrays. Additional clarifications: matrix subscripts can be large, matrix files may have incorrect entries (skip them), matrix files may be empty or non-existent (report error).

- Grades:

  Phase 1 grade: you will receive a final grade for Phase 1 and a preliminary grade for Phase 2. It is highly recommended you make an effort to attempt Phase 2 on the 1st deadline so that you get feedback and a better grade on Phase 2.

  Phase 2 grade: you can fix errors from Phase 1 and make Phase 1 and Phase 2 requirements fully functional. Phase 2 is an opportunity for resubmission of a fixed program. Resubmissions or late submissions will not be allowed (30% penalty for each day).

6 Phase 2 updated specification

Some Phase 2 specifications has been updated. Please check this section carefully. If there is a discrepancy with the rest of the document, what has been specified in this section remains official.

Parameters

Test cases WILL NOT provide the parameter `k` . Your algorithm shall output all the reachable vertices in any case (vector-matrix multiplication and DFS). The `label` parameter will be provided in most test cases.

Output File

The output file MUST be sorted by vertex-id in ascending order, in any case (vector-matrix multiplication and DFS). The source vertex MUST be presented, in any case (vector-matrix multiplication and DFS). The output file MUST present both the vertex-id and labels of reachable nodes, including the source vertex, in any case (vector-matrix multiplication and DFS).

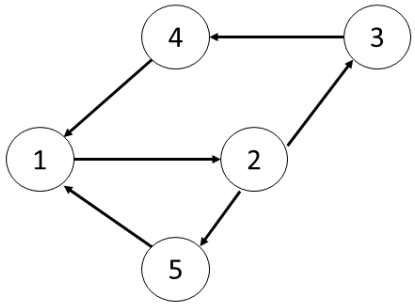


Figure 2: A sample graph G, with adjacency matrix E

7 STEP BY STEP EXAMPLE

Below is presented the sequence of vector-matrix multiplications to compute reachability from source node $s = 1$, up to $k = 3$ iterations.

```
S
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0


E
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    0    0    0
[2,]    0    0    1    0    1
[3,]    0    0    0    1    0
[4,]    1    0    0    0    0
[5,]    1    0    0    0    0
-----------------------------
1st iteration; S * E =
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    0    0    0

S<- S + ( S * E )
S=
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    0    0    0
-----------------------------
2nd iteration; S * E =
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    1    0    1

S<- S + ( S * E )
S=
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    0    1
-----------------------------
```

```
3rd iteration; S * E =
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1

S<- S + ( S * E )
S=
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
-----------------------------
```