COSC2430
HW4: Expression Evaluation Using Stacks and Exploring Graphs

1 Introduction

You will create a C++ program to process a large set of files representing many interconnected pages on the web. In Phase 1 you will evaluate set operations and In Phase 2 you will explore the web page interconnections.

You will initially understand if pages have similar or diverse content with set operations. You will evaluate expressions combining set union, set intersection and parentheses. The program must exploit a stack to convert the input expressions from infix notation to postfix notation and a second stack to evaluate the postfix expression, as seen in class. Then you will extend your program to manipulate a set of files as if it is a set of web pages (i.e. linked with each other).

In order to maintain similar notation to arithmetic expressions, + will denote union and * will be intersection. You will evaluate expressions like $A * B + C$, $(A + B) * (D + E * F)$ or $A * B + (C * D)$, where A, B, C and D are sets. We will define the operation " $+$ " of two sets A and B as the union of the contents of both sets:

$$A + B \equiv A \cup B \tag{1}$$

Similarly, we will define the operation " $*$ " of A and B as the intersection of A and B:

$$A * B \equiv A \cap B \tag{2}$$

These definitions allow us to write long algebraic equations. We assign to the " $+$ " a lower precedence than " $*$ " (in the same way as we do it with numbers) and parentheses change the precedence of an expression. That is $A * B + C \neq A * (B + C)$.

2 Input

The input is one script file, and two or more text files containing words. The script file will have read(), write() and set operations expressions. You will need to write a simple script parser in order to accept expressions such as

```
R0=(A+B)*C
```

There will be **one** expression per line, and there might be multiple expressions in each script file. You are expected to store the result of an expression and then reuse the results from previous expressions in future expressions.

Graph: For the graph part some lines will contain "links" using simplified html. The goal will be to identify: graph sizes $n = |V|$, $m = |E|$, the 3 most popular vertices (highest indegree), and all isolated vertices (nobody points to them). Display one feature per line in an output graph.txt file.

3 Examples

INPUTS

```
   A.txt
   ------------------------------------------------
   alpha beta gamma delta Alpha

   <EOF>

   B.txt
   ------------------------------------------------
   cat dog bird

   <EOF>

   C.txt
   ------------------------------------------------
   alphA

   dog
   <EOF>
```

```
input.script1
------------------------------------------------
read(A,'A.txt')
read(B,'B.txt')
R1=A+B
write(R1,'out1.txt')
read(C,'C.txt')
R2=A+B*C
write(R2,'out2.txt')
```

```
input.script2
------------------------------------------------
read(A,'A.txt')
read(B,'B.txt')
read(C,'C.txt')
R1=A+B
R2=A*B
R3=A+B*C
R4=(A+B)*C
R5=(A+B)*R4+A
write(R3,'out3.txt')
write(R4,'out4.txt')
write(R5,'out5.txt')
```

```
input.script3, only for Phase 2, leave G exploration in file 'graph.txt'
-----------------------------------------
exploregraph('filelist.txt')
```

Input file list

You can create the input filelist with Unix command:

```
ls *.txt>filelist.txt.
```

Files

Graph exploration input example (each file represents a html web page).

```
filelist.txt
---------------
1.txt
2.txt
3.txt
4.txt
5.txt
10.txt


1.txt
-----------------------------------------------
BC
<a href="2.txt">Link1</a>
A
<a href="5.txt">Link2</a>
ABC
<EOF>

2.txt
-----------------------------------------------
<a href="10.txt">Link 3</a>
XYZ
<a href="5.txt">Link 4</a>
<EOF>

5.txt
-----------------------------------------------
cat
beta
alpha
<a href="10.txt">Link1</a>
dog
delta
<EOF>
```

```
4.txt
----------
nothing


10.txt
------------------------------------------------
facebook
<EOF>
```

RESULTS

Phase 1:

```
out1.txt
------------------------------------------------
alpha
beta
bird
cat
delta
dog
gamma
<EOF>

out2.txt
------------------------------------------------
alpha
beta
delta
dog
gamma
<EOF>

out3.txt
------------------------------------------------
alpha
beta
delta
dog
gamma
<EOF>

out4.txt
------------------------------------------------
alpha
dog
<EOF>
```

```
out5.txt
-------------------------------------------
alpha
beta
delta
dog
gamma
<EOF>
```

4 Program and output specification

The main program should be called "web". Syntax:

```
web script=input.script
```

When asked to output a list to a file you must write one word per line. The output should be sorted. If the number of occurrences is 0 you must not output the word. Similarly, when the result word list is empty, the file created will be empty. When calculating the intersection of two lists you must only keep one copy of each word in the output list.

5 Requirements

- Phase 1: arithmetic expressions of sets. Evaluate each set operation as efficiently as possible, meaning that you should reuse and improve your previous search/sort algorithms.

  Phase 2: Build a graph G and explore it. Important: you cannot use 2D arrays (dense matrix not allowed). The files may contain references to other files (i.e. file names, in addition to words). The goal is to understand the characteristics of $G(V, E)$: n, m, top-3 popular vertices, isolated vertices (display one characteristic per line as simple as possible e.g. "n=10", "m=20", "top-3=5.txt", "isolated=2.txt"). You can solve "top-3" and "isolated" with either: (1) matrix-vector multiplication (multiplying by a vector of 1s), (2) scanning E by column and adding indegrees.

  Phase 2 optional (20 points extra credit, choose either option (1) or (2) and explain in README file): (1) use a hash table to manipulate set/result names (assume there are up to $r = 3$ collisions or use a linked list with chaining-open hashing). (2) use hash table to compute intersection and union (use hashing on both sets to determine intersection with 2 strings per bucket, manage collisions; union works similarly with 1/2 strings per bucket; closed hashing preferred).

- Your program must use stacks to evaluate algebraic expressions, one for infix to posfix conversion and another for posfix evaluation.

  Notice the stack allows you to detect invalid expressions, at conversion time or during evaluation time. Your program must be able to detect invalid expressions. Your program must not crash if such an error is detected. You must send an error message to the log file (and optionally to the screen), and continue reading the script file. There will be two hard test cases with invalid expressions.

- You are allowed to build your stacks in any way you prefer, as long as the fundamental stack operations "push", "pop" and "top" exist. Your program must not crash due to stack overflows or underflows.

- Storage and data structures: You can use 1D arrays or linked lists in this homework (indicate it in your README file). That is, store E as a list of edges ((i, j) vertex pairs (notice edges have a direction).

G is assumed to be sparse. Therefore, it is unacceptable to store it as a 2D array (i.e. like a dense matrix); this solution will receive 60 points maximum (i.e., minimum passing grade).

- Limits

  If you use arrays to keep track of results you can assume there will be up to 100. Also, you can assume there can be up to 100 input files (pages, sets of words).

  The number of words a set (file) may contain can be large (say 100k).

- The program should not halt when encountering errors in the script. It should just send a message to the log file and continue with the next line. The only error that is unrecoverable is a missing script file.

- Your program should write error messages to a log file (and optionally to the screen). Your program should not crash, halt unexpectedly or produce unhandled exceptions. Consider empty input, zeroes and inconsistent information. Each exception will be -10.

- Test cases: Your program will be tested with 10 test scripts, going from easy to difficult. You can assume 80% of test cases will be clean, valid input files. If your program fails an easy test script 10-20 points will be deducted. A medium difficulty test case is 10 points off. Difficult cases with specific input issues or complex algorithmic aspects are worth 5 points.

- A program not submitted by the deadline is zero points. A non-working program is worth 10 points. A program that does some computations correctly, but fails several test cases (especially the easy ones) is worth 50 points. Only programs that work correctly with most input files that produce correct results will get a score of 80 or higher. In general, correctness is more important than speed.