CS 161 / Luca Trevisan

Design and Analysis of Algorithms

Midterm
Due: 4:00pm Tuesday 2013-05-14

No Collaboration Permitted

- This is NOT a group project.
- This is NOT a PSet.
- This is a take-home Midterm.
- You should never show anyone else your code.
- You should never read anyone else's code.

```
Our submission script automatically emails
(from your account) your submission to
  cs161-spr1213-hw@lists.stanford.edu
  and your-sunet-id@stanford.edu

If you did not receive such an email,
  we did not receive your solution.
```

Submitting Solutions

- **Grading Environment**

  Your code must compile and run on both `corn.stanford.edu` and `myth.stanford.edu`.[1]

  - If your code fails to run on either `corn.stanford.edu` or `myth.stanford.edu`, it's your fault. (We won't fix your code.)
  - If your code runs on `corn.stanford.edu` and `myth.stanford.edu` but fails to run on our test setup, it's our fault. (You won't lose points; we will make it work.)
  - Your code will read from `input.txt` and write to `output1.txt` and `output2.txt`. Your code should not touch any other files, make other system calls, or make network connections.

- **Languages**

  We only accept C++[2] and Java submissions. We do not accept Ruby, Python, Scala, Matlab, Mathematica, or other languages.

- **Submission Script**

  `ssh` into `corn.stanford.edu` and run one of the following commands:

  - `/usr/class/cs161/scripts/submit-and-send-email-confirmation.sh Midterm.cpp`
  - `/usr/class/cs161/scripts/submit-and-send-email-confirmation.sh Midterm.java`

  Your entire submission must fit in a single file, either `Midterm.cpp` or `Midterm.java`.

- **C++ Submissions**

  - Your entire submission must fit in a single file: `Midterm.cpp`.
  - Our submission script only copies `Midterm.cpp` and ignores all other files in your directory. Therefore, you may not use `#include "local_header_file.h"`
  - You may include the header files from C++ Standard Library. You may not use any other external dependencies.
  - We will compile your code with `g++ -O3 Midterm.cpp -o Midterm` and run your code with `./Midterm`. We will not change compiler flags for you.

- **Java Submissions**

  - Your entire submission must fit in a single file: `Midterm.java`.
  - Our submission script only copies `Midterm.java` and ignores all other files in your directory. Therefore, do not create helper classes `Foo.java` or `Bar.java`.
  - You may import from `java.*`. You may not use any other external dependencies.
  - We will compile your code with `javac Midterm.java` and run your code with `java -Xmx1G Midterm`. We will not change `javac/java` flags for you.

[1] To ensure fair timings, your code will be run on a machine we can monopolize. This may or may not be a corn/myth machine.

[2] You may also submit C code that compiles with `g++ -O3 Midterm.cpp`

Problem Statement

Motivation

We have a directed graph G which models some probabilistic process. The starting state is sampled from a uniform distribution over the vertices. Edge e_i has the form (a_i, b_i, l_i, p_i) and means that if we are at vertex a_i, we have a p_i probability of generating label l_i and moving to vertex b_i. Given a sequence $l_0, ..., l_{T-1}$ of labels, what is the most likely starting point and path taken through graph G?

Modification for Midterm

In the above problem, we have to multiply floating numbers. We don't want to do this for the midterm since order of operations effect rounding errors, which makes everyone's life unpleasant. Instead of multiplying floating point numbers, we will add integral values.

Informal Problem Statement

We have a directed graph G. Edge e_i has the form (a_i, b_i, l_i, w_i) and means that there is an edge from a_i to b_i with label l_i and weight w_i. There may be parallel edges. We are given a trace of labels $l_0, ..., l_{T-1}$. We want to find a path through G (starting at any point, ending at any point) such that (1) the labels of the path match the labels of the trace and (2) the path has minimal weight.

Formal Problem Statement

Your code will read from `./input.txt` the following:

- `int N` `// vertices take on values {0, ..., N-1}`
- `int M` `// there are M edges`
- `int T` `// the trace has T labels`
- `int L` `// labels take on values in {0, ..., L-1}`
- `int traces[T]` `// the trace; encoded as T labels`
- `Edge edges[M]` `// M edges; there may be parallel edges`

We say that $e_0, ..., e_{T-1}$ is a valid path if:

- $\texttt{traces}[i] = \texttt{edges}[e_i]\texttt{.label}$ forall $0 \leq i < T$
- $\texttt{edges}[e_i]\texttt{.to} = \texttt{edges}[e_{i+1}]\texttt{.from}$ forall $0 \leq i < T-1$

The weight of a valid path $e_0, ..., e_{T-1}$ is $\sum_{i=0}^{T-1} \texttt{edges}[e_i]\texttt{.weight}$

For some $e_0, ..., e_{T-1}$ path of minimum weight. Your code should output the following:

- `./output1.txt`: (a single number followed by a newline)
  $\sum_{i=0}^{T-1} \texttt{edges}[e_i]\texttt{.weight}$

- `./output2.txt`: (T+1 numbers separated by newlines)
  `edges`$[e_0]$`.from`
  `edges`$[e_0]$`.to`
  `edges`$[e_1]$`.to`
  ...
  `edges`$[e_{T-1}]$`.to`

Concrete Example

- Let us work through a concrete example with `/usr/class/cs161/public/test-0/input.txt`.

```
5 8 5 2          // N M T L
1 0 0 0 0        // int traces[T]
3 4 1 4          // edges[0].from, to, label, weight
4 0 0 0
0 4 0 5
2 0 1 5
2 0 0 0
4 1 0 4
0 3 0 5
3 2 1 4
```

  - The first line contains four numbers, separated by spaces: `N M T L`.
  - The next line contains `T` integers, separated by spaces, `int traces [T]`
  - The next `M` lines each contains 4 integers, representing `edge[i].from, edge[i].to, edge[i].label, edge[i].weight`.
  - There may be parallel edges in the data.

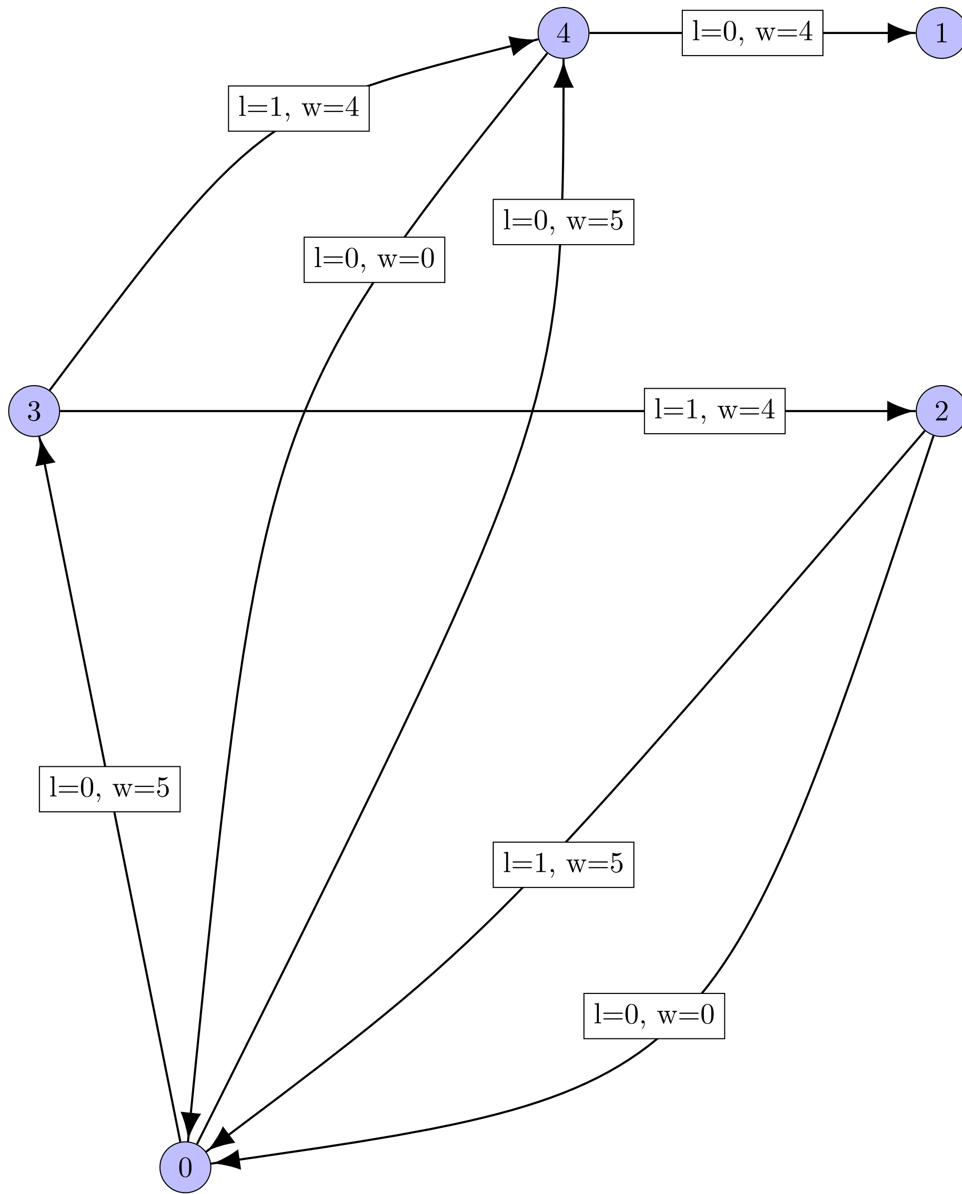
- We need to output a path that satisfies the labels `1 0 0 0 0`.

- We see valid output in `/usr/class/cs161/public/test-0/output1.txt`.

```
14               // minimum weight of a path satisfying labels
```

- We see valid output in `/usr/class/cs161/public/test-0/output2.txt`.

```
3                // .from of edge (3 2 1 4)
2                // edge 3 2 1 4; matches 1; weight = 4
0                // edge 2 0 0 0; matches 0; weight = 0
4                // edge 0 4 0 5; matches 0; weight = 5
0                // edge 4 0 0 0; matches 0; weight = 0
3                // edge 0 3 0 5; matches 0; weight = 5
```

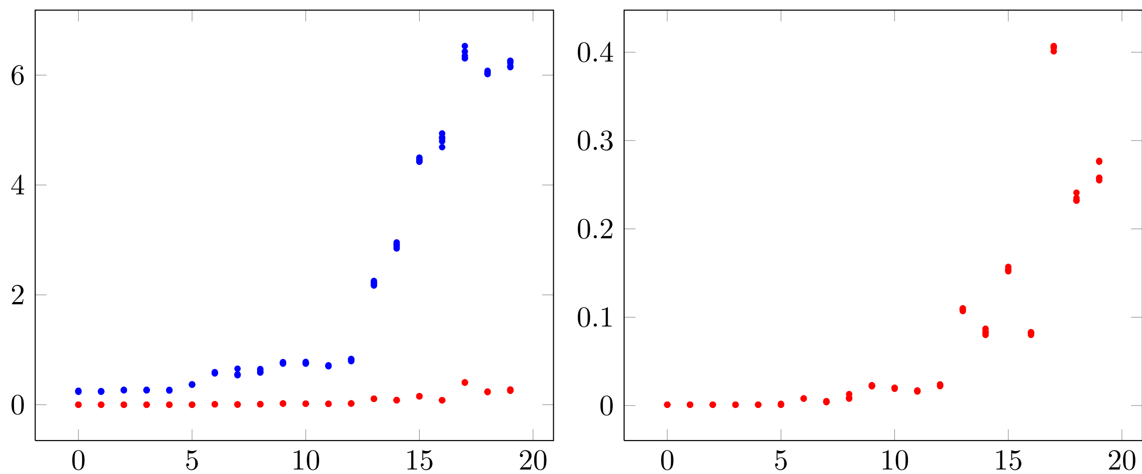

4
l=0, w=4
1
l=1, w=4
l=0, w=5
l=0, w=0
3
l=1, w=4
2
l=0, w=5
l=1, w=5
l=0, w=0
0

Performance Requirements

- We guarantee that `N` $\leq 10,000$, `M` $\leq 100,000$, `T` ≤ 1000, `L` ≤ 50.
  We also guarantee that weights are between 0 and 1,000,000 (inclusive).

- We have 100 datasets (5 random test cases at 20 parameter settings).
  The public data is generated by taking 3 random test cases at parameter settings 0-9 and 3 random test cases at parameter setting 13.

- On a 2.4 GHz Core i7, our single threaded[3] performs as follows:
  Red = C++; Blue = Java; y-axis is seconds; x-axis is parameter setting



- We will take 1/3 of the data from parameter settings 5-9, 1/3 of the data from parameter settings 10-14, and 1/3 of the data from parameter settings 15-19.

- At each parameter setting, your code will be allowed to run for 20x the time of our code. (We will re-time our code on the machine your code is tested on.)

Grading

- We will grade on 80 (private) test cases.

- Each output file is worth 1 point.

- The max score is 160.

[3]Our C++ code is singled threaded. Our Java code is also single threaded; however, the garbage collector automatically runs in another thread.

Details

- Public Test Data

```
corn08 /usr/class/cs161/public> ls
Stest-0   Stest-15  Stest-4  test-1   test-16  test-22  test-29  test-6
Stest-1   Stest-16  Stest-5  test-10  test-17  test-23  test-3   test-7
Stest-10  Stest-17  Stest-6  test-11  test-18  test-24  test-30  test-8
Stest-11  Stest-18  Stest-7  test-12  test-19  test-25  test-31  test-9
Stest-12  Stest-19  Stest-8  test-13  test-2   test-26  test-32
Stest-13  Stest-2   Stest-9  test-14  test-20  test-27  test-4
Stest-14  Stest-3   test-0   test-15  test-21  test-28  test-5
```

  `test-X` is the initial public data we generated.
  `Stest-X` is a "slice" similar to our private test data.
  Each directory contains a `input.txt`, `output1.txt`, `output2.txt` triplet.

- Your code must compile and run on both `./output1.txt` and `./output2.txt`.
- Your code must read from `./input.txt`.
- Your code must write to `./output1.txt` (weight) and `./output2.txt` (path).
- We're grading entirely on correctness and performance. (You don't have to submit a written report; and we don't care about coding style.)
- Our `output2.txt` writes the lexicographically first path of minimum weight. Your `output2.txt` can be any path of minimum weight. (However, it must be ints separated by newlines).
- Your `output1.txt` must consist of a single int followed by a newline.
- Your `output2.txt` must consist of exactly T+1 ints and T+1 newlines; with exactly one newline after each int.
- See our 53 public cases of `output1.txt` and `output2.txt` for concrete examples.