DSP 2019 Spring HW1 Discrete HMM

B05901105 陳矩翰

- Environment:
  $ uname -a
  Linux t480 5.0.4-200.fc29.x86_64 #1 SMP Mon Mar 25 02:27:33 UTC 2019 x86_64 x86_64 x86_64 GNU/Linux

- compiler: g++ and gcc. Note that POSIX threading support is needed on the system.
  - gcc -v:

```
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/libexec/gcc/x86_64-redhat-linux/8/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-redhat-linux
Configured with: ../configure --enable-bootstrap --enable-languages=c,c++,fortran,objc,obj-c++,ada,go,lto --prefix=/usr
--mandir=/usr/share/man --infodir=/usr/share/info --with-bugurl=http://bugzilla.redhat.com/bugzilla --enable-shared --enable-
threads=posix --enable-checking=release --enable-multilib --with-system-zlib --enable-__cxa_atexit --disable-libunwind-exceptions --
enable-gnu-unique-object --enable-linker-build-id --with-gcc-major-version-only --with-linker-hash-style=gnu --enable-plugin --enable-
initfini-array --with-isl --enable-libmpx --enable-offload-targets=nvptx-none --without-cuda-driver --enable-gnu-indirect-function --
enable-cet --with-tune=generic --with-arch_32=i686 --build=x86_64-redhat-linux
Thread model: posix
    gcc version 8.3.1 20190223 (Red Hat 8.3.1-2) (GCC)
```

- how to train and test default testing material?
  $ chmod u+x run_train.sh; make clean; make; bash run_train.sh
  # the script completes the training with iteration 100, test “testing_data1.txt” and “testing_data2.txt” with newly-trained model, and then store the results in “result1.txt” and “result2.txt” respectively.

- how to train and test non-default testing material?
  Please refer to the homework PDF. Or simply type
  $ ./train; ./test
  would give a usage example.

- how to compare testing_answer and result1?
  $ ./acc
  # acc is a simple small program compiled with g++, which would compare “testing_answer.txt” and “result1.txt” by default; No command-line options available. Consider change acc.cc and type
  $ make clean;make
  again if you want to compare other results.

- Some perks of this implementation:
  - 比較直覺性的實作可能會在每個 gamma 跟 epsilon 的 entry 都重新用 summation 做一次 P(O|lambda) ，個人在這個作業的 P(O|lambda) 都是使用該 observation sequence O 產生的 alpha 的最後一個 column 之 summation 作為值，因此對於每個 input sequence O 來說可以省下 Θ(m*n) 這麼多的計算量，其中 m 是 HMM 的 hidden state 數量、 n 是 input sequence O 的長度。並且，直接用這個值進行計算的話，其實際背後代表的意義應該也是較為精確的，因為不需要做一些額外的「某些事件會是獨立事件」的假設。（詳情請參照上課投影片 lecture 4.0）
  - 在 training 以及 testing 的部份都有使用 POSIX pthreads 以進行加速:
    traing 時，對於給定的一個 input sequence O，很明顯可以看出 alpha 跟 beta 可以同時計算而不會有 data-racing 問題，同理 gamma 跟 epsilon 的累計、Baum Welch 最後計算新的(π’, A’, B’)

的部份也都可以平行化運算，因此在 train.c 中這三個部份都是平行處理的，alpha/beta 開兩條 pthread、gamma/epsilon 的部份開兩條、B-W algorithm 的部份則開三條;
testing 時，因為我們是針對每一個 HMM 進行運算、取值，對於 testing data 檔案只是唯讀操作，因此在 test.c 中有多少個 HMM 就一次開了多少個 thread。

- 有對每個 Observation sequence O 的長度進行 normalization，因此雖然這個作業測資的 O 長度固定都在 50，本程式可以處理檔案中有不同長度的測資的情況，並且儘管長度可能不同，其對結果的貢獻都是大致相同的。
- test.c 以及 train.c 都能應對不同的 hmm 總數、不同的 hidden state 總數、不同的 possible observation count from a hidden state 的 initial HMM，例如可以丟有 10 個各自有 8 個 hidden state 的 HMM、以及共有 ABCDEFGHI 的測資給這兩隻程式訓練及測試
- iteration 的時候會對 hmm model 本身進行防止 degeneration 的處理，也就是會在一些 observation/transition/inital 的 entry 降到低於一個靠近 0 的閾值的時候進行修正，輸出 model 的時候也會另外進行一次。
- 在 test.c 的 Viterbi 演算法實作部份採用 long double 而非 double，以盡量減少誤差

- Some defects of this implementation:
  - 利用 run_train.sh 做出來的 model 其正確率並不高，針對 testing_data1.txt 僅 0.788。
  - 測資只能是 char 型別，並且其 ascii 值必須大於等於(A)，否則會產生記憶體存取越位。
  - 數值計算的時候的機率 normalization、以及 anti-degeneration 的方法都非常 naive，亦即只是把超出閾值的部份丟到最大的 entry 上，並不是非常精確的方法，尤其是在輸出 model 的時候受限於輸出只有到小數後 5 位，誤差的最大值可能高達(0.00001*(state_count-1))，對於一些 state 數量比較大的 hmm 並不是一個很好的辦法。
- Some observation:
  - hmm.h 裡面提供了 loadHMM 函式讀取 model 的資料，但是 loadHMM 裡面只有開檔沒有對應的關檔，因此如果不把 hmm.h 改掉的話必然會有 memory leak 產生;
    另外，load_models 的 counter 邏輯也有些奇怪，如果想要讀 n 個 model 而把 n 當作最後一個參數丟進去的話，就算 loadHMM 沒有 memory leak，load_models 自己也會因為過早的 return statement 而產生一個 leak。
    個人的作業中沒有改動 hmm.h，而只是把 load_models 以及 loadHMM 複製到對應的 train.c、test.c 並更動名稱成 myload_models、myloadHMM，並修改 load_models 其中 counter 的邏輯、在 loadHMM 最後加上 fclose。用 valgrind (valgrind-3.14.0)測試會發現經過這樣的改動後就沒有發現 memory leak 的情況了。