COM1005 Assignment 1 2019 Report

**Name:** Jamie Huddlestone **CICS userID:** aci17jh

# Testing LatticeSearch

In order to test my implementation of LatticeSearch, it was necessary to create a new WordLattice and load a lattice file using the latticeFromFile method of that class. Using the default lattice file (latt1.txt), my test harness prints the following:

----------------------------------------

WORD LATTICE

please 0 10 20

lettuce 10 30 100

know 30 36 50

flea 0 9 30

use 16 29 120

throw 29 36 70

freeze 0 10 40

let 10 18 30

useless 18 31 200

throw 31 36 80

us 18 30 90

End time: 36

Words at time 0:

[please 0 10 20, flea 0 9 30, freeze 0 10 40]

----------------------------------------

Unfortunately, the implementation of RunLatticeSearch prohibits a straightforward instantiation of the LM class, as the vocabulary list and cost matrix are defined within the scope of the main method; updating the LM class to initialise itself from a CSV file with headers (representing the vocab with the header row and the matrix below) instead, in the fashion of WordLattice, would allow for more flexible instantiation and testing.

# Testing LatticeState

I encountered similar object-hierarchy issues in writing a test harness for LatticeState: to check the output of the goalP and getSuccessors methods would require prior instantiation of several other classes dependent on the language model represented by the LM class. Fortunately, they are sufficiently trivial that the expected behaviour can be determined by inspection if we assume that the implementations of all the classes used within are correct.

However the sameState method was not too onerous to check in isolation, and because it depends on the implementation of *SearchState*.toString(), I believe it is worth having a test for. The test harness performs a simple equality test over instances that differ by one parameter:

----------------------------------------

zero 0 0 0 == zero 0 0 0

----------------------------------------

zero 0 0 0 != one 0 0 0

zero 0 0 0 != zero 1 0 0

zero 0 0 0 != zero 0 1 0

zero 0 0 0 != zero 0 0 1

zero 0 0 0 == zero 0 0 0

----------------------------------------

one 0 0 0 != zero 0 0 0

zero 1 0 0 != zero 0 0 0

zero 0 1 0 != zero 0 0 0

zero 0 0 1 != zero 0 0 0

zero 0 0 0 == zero 0 0 0

----------------------------------------

# Experiments with search strategies

As we were given only one word lattice with which to test the two bigram language models provided, I needed to find a way of making some more: to this end, I have written a self-contained class called MakeRandomLatticeFile that generates dummy lattice files from header-CSV files representing bigrams (of the kind described above). With this program, it was possible to generate a lattice from the “TVLM” model and a further five test lattices from the “please let us know” model, which are provided in the accompanying zip file.

In order to more easily test the RunLatticeSearch class with these different lattices, and with the different search algorithms implemented in the Search class, I have modified its main method slightly to take command line parameters for the location of the lattice file and the name of the algorithm: in this way, it is now easier to test different lattices and algorithms by obviating the need for recompilation. However, the LM class has not yet been changed to facilitate CSV files representing bigram models (see above).

In the course of the adjustment to RunLatticeSearch, I noticed that the selectNode method of the Search class began with a faulty comparison between strings using the == operator, which ensured fall-through to the final else case and subsequent execution of the branchAndBound method on *every* occasion; this has now been rectified to perform the necessary string comparisons using the equals method when determining the algorithm.

The efficiency of each algorithm on five randomly-generated lattices is as follows (2 sig fig):

|  |  |  |  |
| --- | --- | --- | --- |
| *lattice / algorithm* | **depthFirst** | **breadthFirst** | **branchAndBound** |
| **lattice1** | 0.67 | 0.50 | 0.57 |
| **lattice2** | 0.83 | 0.45 | 0.45 |
| **lattice3** | 1.0 | 0.27 | 0.44 |
| **lattice4** | 1.0 | 0.33 | 0.36 |
| **lattice5** | 0.15 | 0.75 | 0.75 |

Table : comparison of state-space search performance over randomly-generated word lattices

Figure : comparison of state-space search performance over randomly-generated word lattices

As the table and graph show, the performance of branchAndBound was at least equal to or superior to breadthFirst on each occasion, whereas the massive variance shown by the results for depthFirst reflect the fact that over relatively short lattices, it is possible that the solution path may coincide with the outside path of the generated search tree; in other words, the efficiency of depthFirst has more to do with luck than admissibility.

# Further lattice search investigation

Considering the applicability of the A\* search algorithm over this problem domain, it seems to me that there may be no appropriate heuristic by which to easily calculate estimated global costs for candidate partial solution paths. The bigram language models provided give us an indication of the ‘cost’ incurred between two words – in this case, a measure of how surprising it would be to hear the first word followed immediately by the second in a naturally-occurring English sentence – but only over an extremely limited subset of the lexicon from which I assume this measure of surprisal is derived, so it is not possible to calculate a probability distribution for the frequency of occurrence of these words in spoken language from the data provided. This in turn would prevent us from making a reasonable estimate of the global cost of a candidate sentence, as we have no way of making a sensible underestimate of a candidate path’s global cost without knowledge of the likelihood of words following that carry unhelpfully high or low costs.