Algorithmics 3 Assessed Exercise

Status and Implementation Reports

Nikolay Ivanov
2115451i

November 14, 2016

Status report

Both programs were implemented and are working correctly. Using the test data provided, some of the output was checked manually by hand, and then for the more complex test data I compared my output with the output of other students.

Implementation report

(a) The Dijkstra algorithm was implemented as follows:

First I create a list of all the words from the input file, then create a graph with the size of the number of words. After I iterate over the words, I add each word to the graph as a vertex, and create an adjacency list for it. While iterating, if the current word equals either the start or end word, I save the index. When the Dijkstra method is called, I create two sets (one for "visited", the other for "unvisited"), and a map with distances to each vertex from the starting one. While going through the helper methods, I get the min distance from vertices in the "unvisited" set, add the found node to the "visited" set, then I perform relaxation and update the distances map, and then I set the predecessors. After dijkstra's algorithm is finished, in order to get the path with shortest distance from the start vertex to the end vertex, I simply get each predecessor for the end vertex, add it to a list, and return the list in reversed order. To get the shortest distance as an integer, I simply get the value from the distances map where it's key is the end vertex.

A step I used to improve the efficiency of the program, was to fix my data reading. At the beginning I was creating unnecessary list of Vertices which I was then copying in the graph. I noticed that this was useless and that I can just get the vertex from the graph with given index and set its word, and later append to its adjacency list.

(b) The backtrack algorithm was implemented as follows:

Similar to the Dijkstra implementation, I create a list of all the words from the input file, and again iterate over them adding each word as a vertex to the graph, saving the indexes for the start and end vertices. Then I do a bfs by giving the method the start and end vertices. In my implementation of the bfs I have a list, a queue, and a helper list. I begin by adding the start vertex to the list and setting it to "visited". I then add it to the queue and then loop while the queue is not empty. If the last element in the list is equal to the end vertex, the ladder is found and is printed out. If not I go through the adjacency list of the element in the ladder, and if the current vertex from the adjacency list is not "visited", I add it to the ladder, update the queue, and set that vertex to "visited". In the end when all vertices are visited, and no vertex in the ladder is equal to the end vertex, that means that there is no ladder between the two words.

A step I used to improve the efficiency was the same step I did in the Dijkstra's implementation.

Empirical results

The programs run in between the range of 80 to 115 ms.

Output from the Wordladder program:
(Start word: print, End word: paint)
Steps in ladder: 1
Ladder: print → paint
Elapsed Time: 88 milliseconds

(Start word: forty, End word: fifty)
Steps in ladder: 4
Ladder: forty → forth → firth → fifth → fifty
Elapsed Time: 98 milliseconds

(Start word: cheat, End word: solve)
Steps in ladder: 13
Ladder: cheat → chert → chart → charm → chasm → chase → cease → lease → leave → heave → helve → halve → salve → solve
Elapsed Time: 93 milliseconds

(Start word: worry, End word: happy)
No ladder
Elapsed Time: 91 milliseconds

(Start word: smile, End word: frown)
Steps in ladder: 12

Ladder: smile → smite → spite → spice → slice → slick → click → clock → crock → crook → croon → crown → frown

Elapsed Time: 92 milliseconds

(Start word: small, End word: large)

Steps in ladder: 16

Ladder: small → shall → shale → share → shard → chard → charm → chasm → chase → cease → tease → terse → verse → verge → merge → marge → large

Elapsed Time: 91 milliseconds

(Start word: black, End word: white)

Steps in ladder: 8

Ladder: black → blank → blink → brink → brine → trine → thine → whine → white

Elapsed Time: 88 milliseconds

(Start word: greed, End word: money)

No ladder

Elapsed Time: 92 milliseconds

Output from the Dijkstra program:

(Start word: blare, End word: blase)

Minimum Distance: 1

Words in path: 1

Path: blare → blase

Elapsed time: 113 milliseconds

(Start word: blond, End word: blood)

Minimum Distance: 1

Words in path: 1

Path: blond → blood

Elapsed time: 102 milliseconds

(Start word: allow, End word: alloy)

Minimum Distance: 2

Words in path: 1

Path: allow → alloy

Elapsed time: 90 milliseconds

(Start word: cheat, End word: solve)

Minimum Distance: 96

Words in path: 13

Path: cheat → chert → chart → charm → chasm → chase → cease → lease → leave → heave → helve → halve → salve → solve

Elapsed time: 117 milliseconds

(Start word: worry, End word: happy)
No Ladder
Elapsed time: 93 milliseconds

(Start word: print, End word: paint)
Minimum Distance: 17
Words in path: 1
Path: print → paint
Elapsed time: 102 milliseconds

(Start word: small, End word: large)
Minimum Distance: 118
Words in path: 16
Path: small → shall → shale → share → shard → chard → charm → chasm → chase → cease → tease → terse → verse → verge → merge → marge → large
Elapsed time: 107 milliseconds

(Start word: black, End word: white)
Minimum Distance: 56
Words in path: 8
Path: black → slack → shack → shank → thank → thane → thine → whine → white
Elapsed time: 110 milliseconds

(Start word: greed, End word: money)
No Ladder
Elapsed time: 104 milliseconds