Algorithmics I – Assessed Exercise

Status and Implementation Reports

Jack Tickoo
2397224T

November 15, 2020

Status report

I believe both programs are working correctly based on the methodology I have implemented and the output of the tests I have run. Both programs return the same paths where paths can be found, and I have checked the Dijkstra's algorithm path lengths by hand.

Implementation report

(a) In the first program I created a utility function to check whether two words were adjacent by nature of the specification. I created a list containing all the words in the dictionary. I created a graph of nodes containing indexes corresponding to the words in the dictionary. I.e. for the dictionary size 1638 the graph contains 1638 nodes with indexes from 0 to 1637. Each can therefore represent a word. Then for every node using the same index referencing technique I created a corresponding adjacency list.

Once graph is created and all the adjacencies have been established the algorithm can be executed. I start with the corresponding node to the start word. I mark this node as visited push it to the queue. I then repeat the following algorithm until the queue is empty or the finishing node is found: 1) pop a node from the queue 2) push all its unvisited adjacent nodes to the queue 3) mark node as visited and remove from queue

This algorithm will repeat until the node with the corresponding end word is found or until all the nodes in the graph reachable from the starting vertex have been visited. I set the nodes predecessors as I go, this is reminiscent of descending rank in a tree. In order to get the path and the ladder size, if such a ladder exists, I traverse up through the predecessors until I reach the starting node adding the nodes to a list. I then return the size of the list. I could not think to make this any more efficient, the elapsed time was already very fast however I could perhaps have cleaned the code in places be utilising a node-visiting function.

The time complexity of BFS is O(V + E) where V is the number of vertices and E is the total number of edges in the graph.

(b) For program 2 I used the same technique to construct the graph as described in program 1. I made a utility function to determine the edge weight between nodes and stored the respective weights in the adjacency list nodes. I devised another utility function determining the node with the closest weight. I then implemented Dijkstra's algorithm setting the source node to

"visited", setting it's predecessor node to -1 then updating the weights of the adjacent nodes. Then iterating through the all the nodes finding the closest node and visiting it. I repeated the algorithm updating the adjacent nodes distance to source and predecessor if the path through the node I am currently visiting is smaller than the existing. I repeated this process until the entire graph was traversed. To find the shortest path back I used the same technique as in program 1 by accessing the end node and traversing the predecessors back to the source. If the end node's distance to the source is still set to infinity, I can determine that no path exists as the entire graph has been traversed.

The time complexity of this implementation is approximately $O(v^2)$ as for every node I am iterating through all the nodes in the graph trying to find the closest node to move to next.

I improved on this implementation by creating my new implementation where everything is the same except, I create two lists for visited and unvisited. I add the source node to visited and then all the adjacent nodes to unvisited. As I visit nodes, I add nodes to visited and remove them from unvisited. This meant that I was searching through far less nodes when searching for the closest next node as all the nodes that have yet to be added to unvisited still had source distances of infinity and did not need to be traversed when finding the nest closest. This implementation also meant that if I visit all the reachable nodes from the source node, I do not then unnecessarily visit of all the unreachable nodes in the graph. This implementation had the improved times you can view below. The time complexity of this implementation is on average less than $O(v^2)$ if all the vertices are not adjacent to one another and providing a path exists the path is not the furthest in the graph.

Empirical results

```
first program tests:

size of dictionary: 1638
start word: print
end word: paint
print
paint
Shortest Word Ladder length: 2
elapsed time: 1522100 nanoseconds

size of dictionary: 1638
start word: forty
end word: fifty
forty
forth
firth
fifth
fifty
Shortest Word Ladder length: 5
elapsed time: 3031500 nanoseconds

size of dictionary: 1638
start word: cheat
```

```
end word: solve
cheat
chert
chart
charm
chasm
chase
cease
lease
leave
heave
helve
halve
salve
solve
Shortest Word Ladder length: 14
elapsed time: 8170900 nanoseconds

size of dictionary: 1638
start word: worry
end word: happy
Word ladder from worry to happy is not possible.
elapsed time: 6922600 nanoseconds

size of dictionary: 1638
start word: smile
end word: frown
smile
smite
spite
spice
slice
slick
click
clock
crock
crook
croon
crown
frown
Shortest Word Ladder length: 13
elapsed time: 5821300 nanoseconds

size of dictionary: 1638
start word: small
end word: large
small
shall
```

```
shale
share
shard
chard
charm
chasm
chase
cease
tease
terse
verse
verge
merge
marge
large
Shortest Word Ladder length: 17
elapsed time: 6539199 nanoseconds

size of dictionary: 1638
start word: black
end word: white
black
blank
blink
brink
brine
trine
thine
whine
white
Shortest Word Ladder length: 9
elapsed time: 4203700 nanoseconds

size of dictionary: 1638
start word: greed
end word: money
Word ladder from greed to money is not possible.
elapsed time: 10579700 nanoseconds

-------------------------------------------------------------------------------------------------
second program tests:

size of dictionary: 1638
start word: blare
end word: blase
blare
blase
Dijkstra's shortest path length: 1
```

```
elapsed time: 14844900 nanoseconds

size of dictionary: 1638
start word: blond
end word: blood
blond
blood
Dijkstra's shortest path length: 1
elapsed time: 11023300 nanoseconds

size of dictionary: 1638
start word: allow
end word: alloy
allow
alloy
Dijkstra's shortest path length: 2
elapsed time: 1227900 nanoseconds

size of dictionary: 1638
start word: cheat
end word: solve
cheat
chert
chart
charm
chasm
chase
cease
lease
leave
heave
helve
halve
salve
solve
Dijkstra's shortest path length: 96
elapsed time: 16663200 nanoseconds

size of dictionary: 1638
start word: worry
end word: happy
Word ladder from worry to happy is not possible.
elapsed time: 8500599 nanoseconds

size of dictionary: 1638
start word: print
end word: paint
print
```

```
paint
Dijkstra's shortest path length: 17
elapsed time: 9629400 nanoseconds

size of dictionary: 1638
start word: small
end word: large
small
shall
shale
share
shard
chard
charm
chasm
chase
cease
tease
terse
verse
verge
merge
marge
large
Dijkstra's shortest path length: 118
elapsed time: 13609200 nanoseconds

size of dictionary: 1638
start word: black
end word: white
black
slack
shack
shank
thank
thane
thine
whine
white
Dijkstra's shortest path length: 56
elapsed time: 13420200 nanoseconds

size of dictionary: 1638
start word: greed
end word: money
Word ladder from greed to money is not possible.
elapsed time: 18844499 nanoseconds
```