CS21120 Word Ladder Assignment Document

**Module Code:**

**CS21120 Program Design, Data Structures and Algorithms**

**Author:**

**Chris Savill –** [**chs17@aber.ac.uk**](mailto:chs17@aber.ac.uk)

Table of Contents

[Class Diagram 3](#_Toc338793770)

[Design 4](#_Toc338793771)

[Graph Class 4](#_Toc338793772)

[Vertex Class 4](#_Toc338793773)

[UnboundedStack and UnboundedQueue 4](#_Toc338793774)

[LinkedList and Node 5](#_Toc338793775)

[WordReader 5](#_Toc338793776)

[LadderCycler 5](#_Toc338793777)

[GenerateCycler and DiscoveryCycler 5](#_Toc338793778)

[GenerateLadder 5](#_Toc338793779)

[DiscoverLadder 5](#_Toc338793780)

[Menu 5](#_Toc338793781)

[WordLadderDriver 5](#_Toc338793782)

[Description of Algorithms 6](#_Toc338793783)

[Pseudo-code 7](#_Toc338793784)

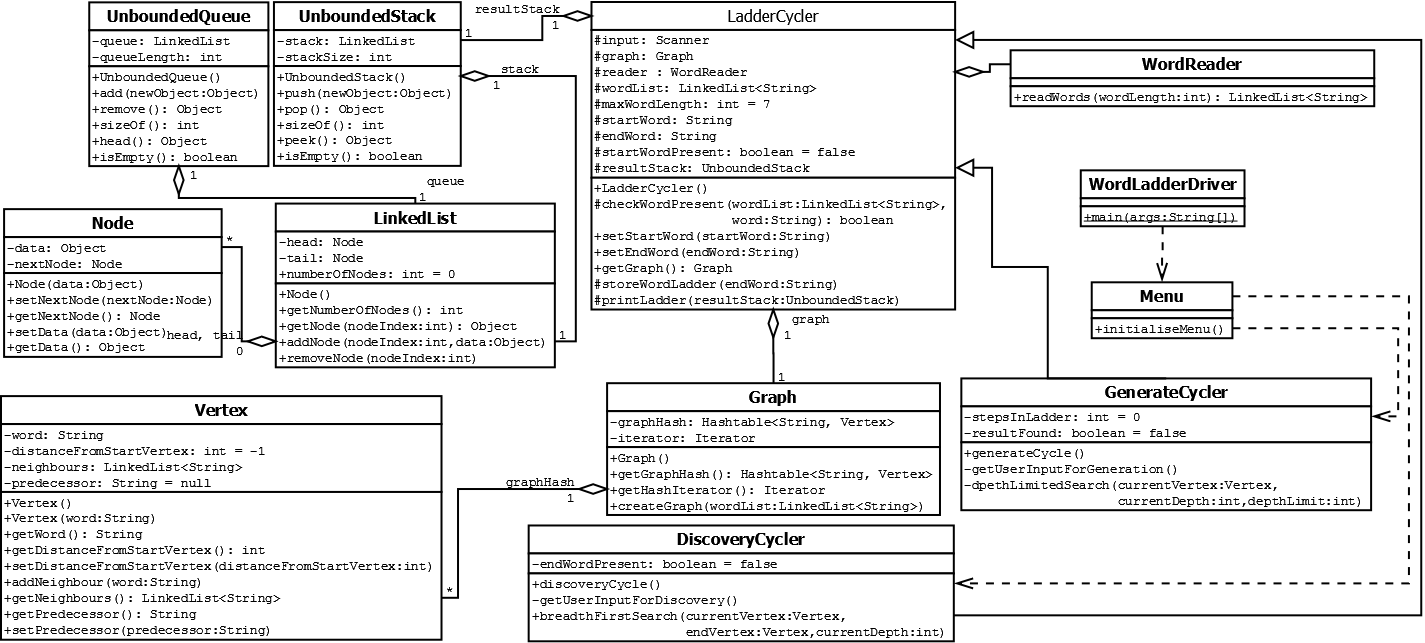
[Initial start-up 7](#_Toc338793785)

[Generation function: 7](#_Toc338793786)

[Java Source Code 8](#_Toc338793787)

[Testing 9](#_Toc338793788)

# Class Diagram



# Design

The design approach for this project was aimed to enforce low-coupling and high-cohesion, however, due to time constraints and problems encountered during the implementation, the program may not be as optimised for low-coupling and high-cohesion as it could be. Below are the class descriptions, refer to class diagram above where necessary. Justifications of design are given for each class where appropriate.

## Graph Class

To accomplish the task at hand guidance was given that a graph data structure consisting of a hash table and vertices would be the best method. The hash table data structure would be implemented using the Hashtable.java class. It would use the words of type String as keys and vertices would be the data that the keys were hashed to.

As the class diagram indicates, the Graph class contains methods relating to building/creating the graph as well as returning the hash table an iterator (used for iterating through the hash table) and the default constructor.

## Vertex Class

The vertex class was designed to act as nodes/vertices of the graph class. The class contains four variables:

1. ‘word’ which is a string to store the word to which the vertex refers to;
2. ‘distanceFromStartVertex’ which is an int to label how far from the start word the vertex lies (initialised to -1 to mark as unexplored);
3. ‘neighbours’ which is a linked list of strings to store all of the words that have only a one letter difference;
4. ‘predecessor’ which is a string to store the word of the vertex that the vertex was expanded from (initialised to null as there is no predecessor when generated).

Having the linked list of neighbours essentially creates a network of words which represents a graph data structure, along with the hash table to access the graph’s vertices.

## UnboundedStack and UnboundedQueue

The UnboundedStack and UnboundedQueue classes were implemented instead of the Stack and Queue classes provided by the java libraries because those classes contain many methods not required by this program. Also as both the UnboundedStack and UnboundedQueue classes were already built when worksheet 5 was completed , no extra work would be required to implement them over the over choices. The two classes used contained what was needed and not much else so it seemed appropriate to implement them over the other choices.

As the class diagram indicates, the UnboundedStack class is used by the DiscoveryCycler and GenerateCycler classes (through the super LadderCycler class) as a stack to store the result/ladder. The UnboundedQueue class is only used by the DiscoverLadder class as a queue data structure for the frontier of vertices/nodes to be expored.

## LinkedList and Node

These two classes were built in worksheet 4 and are used by the UnboundedStack and UnboundedQueue classes so are part of the design and choice of the data structures.

## WordReader

The WordReader class as shown by the class diagram is used by the the DiscoveryCycler and GenerateCycler classes (through the super LadderCycler class) for reading in the word lists from the data files provided. The list created is then used by the Graph class to add to the hash table and create new vertices from each word in the word list to create the graph.

## LadderCycler

The LadderCycler abstract class is a super class of the DiscoveryCycler and GenerateCycler classes. This was created because there were many common variables and methods between the two sub classes so it made sense to encapsulate them in this abstract class to reduce code duplication. This also aids in high cohesion as all of the variables and methods relevant are contained in the one class, also to be used by the sub classes. As this class is acting like a hub for all of the main data structures to come together, it allows for all of the other classes to facilitate low coupling as they do not have to all connect to each other.

## GenerateCycler and DiscoveryCycler

These two classes as indicated in the class diagram are sub classes of the LadderCycler super class. They both inherit all of the variables and methods as well as incorporating their own variables and methods related to the generation/discovery functionality. They have methods to launch the generation/discovery cycle including calling other methods to get the user inputs for the purpose of the ladder generation/discovery and the search algorithms.

## Menu

This class is just a simple menu class that gives the user three options, generation, discovery and exit.

## WordLadderDriver

As indicated by the class diagram this class just contains the main method, it is the first class launched on start-up and just creates an instance of the Menu class and calls the initialiseMenu method.

# Justification of Algorithms

## Generation Algorithm

For the generation part of the program it seemed apparent that a Depth-Limited Search (DLS) would be the most appropriate algorithm to use. This is because the user would enter in the number of steps in the ladder which is basically the depth at which to go down. As a Depth-Limited Search algorithm acts like a Depth-First Search (DFS) but with a limit on the depth, it made sense just to use a Depth-Limited Search instead of a Depth-First Search. As soon as the depth limit was reached the resulting ladder would have been generated successfully. It must be referenced that the book ‘Artificial Intelligence: A Modern Approach’ Third Edition aided me in the decision and implementation of the Depth-Limited Search algorithm for the generation part of the program.

## Discovery Algorithm

For the discovery part of the program it was hinted that Dijkstra’s Algorithm would be one of the best ways to go (at least for an uninformed search). However as Dijkstra’s Algorithm relies on having a weighted graph (of which the graph implemented here is not), there is no point in using it. Without the priority queue due to a weighted graph the algorithm would act exactly like a Breadth-First Search (BFS), this is why a Breadth-First Search was chosen for the discovery algorithm. The Breadth-First Search algorithm is complete and is guaranteed to find the shortest word ladder/path to a solution as it checks the shallowest vertices/nodes first then the next depth below etc. Breadth-First Search is the most efficient uninformed search algorithm to use. There may be a better and more optimal/efficient informed heuristic search algorithm but due to time constraints it was a good idea to get a simpler algorithm working first. It must be referenced that the book ‘Artificial Intelligence: A Modern Approach’ Third Edition aided me in the decision and implementation of the Breadth-First Search algorithm for the discovery part of the program.

# Pseudo-code

## Initial start-up

Algorithm for the initial start-up of the program. Involves:

* Displaying the menu to the user;
* Getting their option input:
* Checking if their selected option is valid
  + If not get input again and recheck.
  + If yes run the corresponding code.

**Display** "Welcome to the Word Ladder Generator"

**While** (selected option is not equal to the exit option) {

**Prompt** a menu to run either the generation function, discovery function or exit function.

**Read** user's input for option.

**Switch** (selected option) {

**Case 1** (Generation function):

**Run** generation function

**Break** out of switch case

**Case 2** (Discovery function):

**Run** discovery function

**Break** out of switch-case

**Case 3** (Exit):

**Display** "Exiting program"

**Break** out of switch-case

**Default:**

**Display** "Invalid option selected, please select a valid option"

**Break** out of switch case

}

}

## Depth-Limited Search (DLS) Algorithm for Generation:

Recursive DLS-

* Set current word being looked at as the word passed in (start word if at beginning);
* Set distance of current word to current depth (0 for start word);
* Check if current word’s distance from start word is higher than the current depth;
  + If yes, return false.
* If no, check if current word is at the depth limit;
  + If yes, return true.
* If no, for each neighbour word unexplored, if any (distance less than 0):
  + Set predecessor of neighbour word to the current word;
  + Check if a call to the recursive DLS method returns true to a result being found (passing in the neighbour word, the current depth + 1 and the depth limit);
  + If result found equals true, return true.
* If no result found through any path to the depth required, return false (failure).

## Breadth-First Search (BFS) Algorithm for Discovery:

Breadth-First Search-

* Create new frontier queue;
* Set distance from start word to current depth (0 if at beginning/start word);
* Add word to frontier queue;
* While frontier queue is not empty:
  + If current word matches end word, return true.
  + Else set current word to word at the front of the frontier queue and remove front of the queue;
  + For every neighbour of current word unexplored, if any (distance less than 0):
    - Add neighbour to back of frontier queue;
    - Set distance from start word of neighbour to the current depth + 1;
    - Set the predecessor of the neighbour to the current word;

# Java Source Code

# Testing