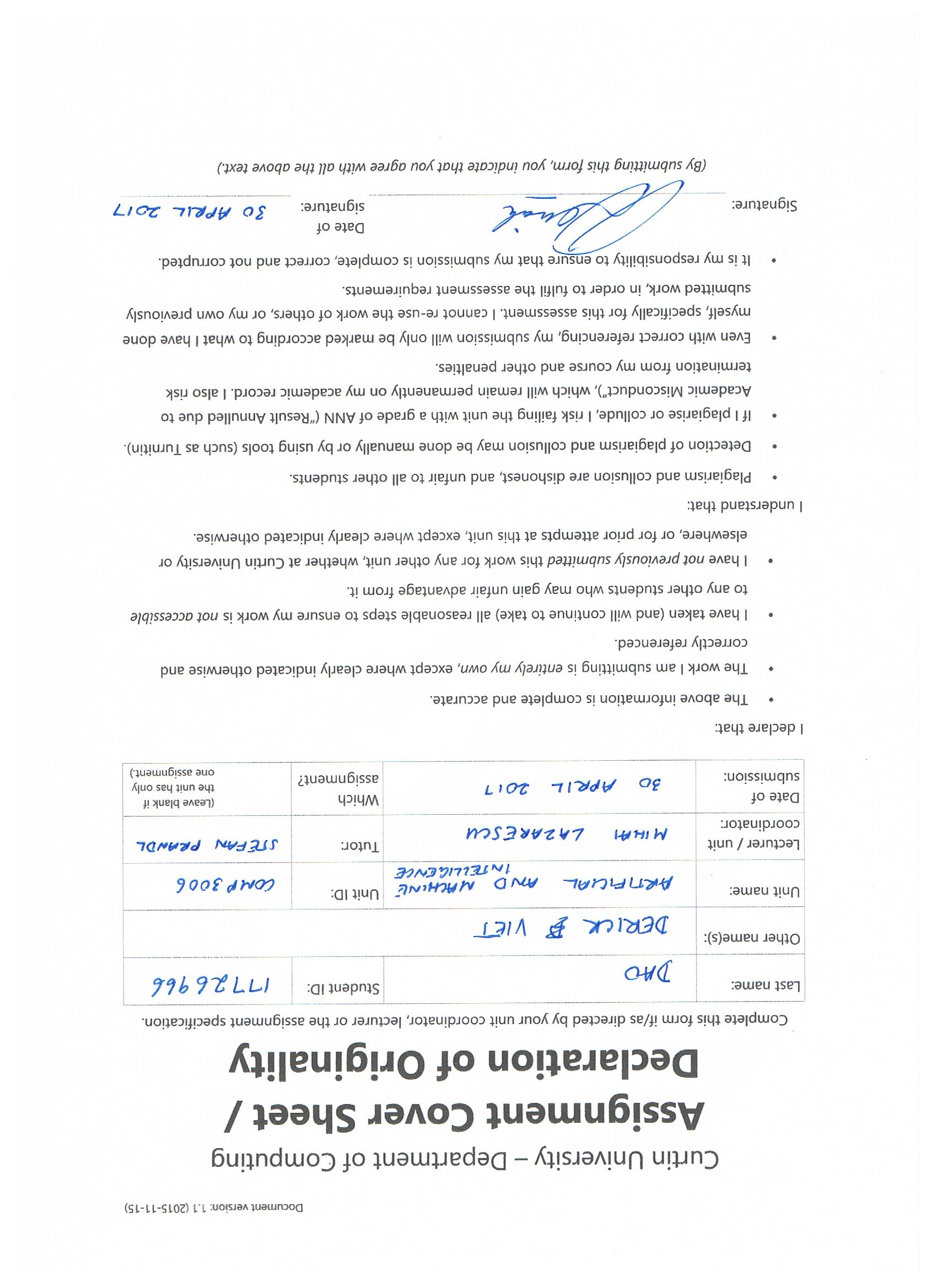
******

***Introduction***

This report covers my bugs and problems when creating the programs for my memory limited A\* search and my beam search where the value for k can range from one to three. It also describes the ways I tried to overcome the problems and states what current problems I still have in the program. The report displays the successes and failures of my testing with the graphs provided in the tutorial and a graph which consisted of 16 nodes to test against my memory limited A\* search.

***Beam Search Report***

Some of the various problems that have occurred while creating my beam search program was keeping track of the current path the algorithm is attending to and how to keep the other paths in memory while working on one path. I decided to implement a List<Node> array list which allowed me to store nodes that are being searched to be added to the list to be printed out when the goal node is reached.

The other paths, however, because my algorithm is defined to go only one way and backtrack if one path doesn’t work, nodes that have been searched are added to the path list and if it cannot be expanded, it will be removed from the path list and other children nodes are then expanded.

*For example:*

1-3-6-10-12 is the current path that is being assessed to find the node from 1 to 15 (goal node). Previously, we had to assess 1-2 to be able to obtain the k children nodes at the next depth. 2 was added to the path list but we had to expand 6 next because it was the lower heuristic. Due to the fact 6 is to be expanded, 2 was removed from the path list, 3 was added and then 6 was expanded. My program follows only one path, but by the end of the program, it will backtrack to the nodes it hasn’t expanded and find alternate paths.

The way the program has been created, I couldn’t find a way to print out partial paths when a solution is found. The algorithm assesses all parent nodes before finding the k number of children nodes of the next depth but will only add to the path list nodes that it expands while trying to find the solution. This is a major problem/bug that is still present in my program. Ways that I have tried to go about this problem is to put various lists into a list of nodes (List<List<Node>>) and importing them each time I run my recursive algorithm, but each time the algorithm runs, it will create segments of paths rather than a whole partial path (i.e. 1-3, 1-3-6, 1-3-6-10 etc.).

Thus, my program can still find the complete path and the alternate solutions but can’t print out the partial paths that are currently held in memory when a solution is found.

***Alim Search (Memory Limited A\* Search)***

Some problems that have occurred while creating my memory limited A\* search program was finding a way to keep the search tree under the limit of 15 and removing certain nodes from the search tree to add new searched nodes.

I had originally started with having the search tree as an underlying function with my Node class. I found that defining the search tree is near impossible to do that if I have it as a structure made in my Node class. So I moved it as a List<Node> in starSearch() method to keep track of the search tree. The list will specify the nodes that are visited in the search tree and will use the function memoryCheck() to deal with having a memory limit of 15. The function memoryCheck() will try to find the highest f\_cost node that isn’t part of the path of the node dequeued from the queue and remove it to make space for the insertion of the dequeued node.

Printing out the partial paths is also another struggle for me when designing this program. Given a search tree with multiple nodes that aren’t part of a path, I tried to think of a way to grab the nodes deepest of the other branches but if I am dealing with a tree with more than 2 branches extending from the root, it will be hard to dictate which nodes are from where. This resulted in me being unable to implement a way to print out partial paths when a solution is found. Even though this is a bug/problem still evident in my program, my program still functions properly and tries to find the optimal solutions and tries to find alternate solutions as well.

Some of the few problems I had previously was having invalid nodes in my search tree, removing the wrong nodes from the search tree to add a new one when the memory limit was reached and having incorrect calculations of the f\_score for each of the nodes.

Following the occurrence of the invalid nodes, I created a method in the ReadFile class named printContentsN() which prints out the content of a particular list, so in this case, it’s the list I used for the search tree. Printing out the contents of the search tree allowed me to see which nodes were out of place. I would use the function to print after every change that was made to see which bits of the program would affect the invalid node. Tracking down the bits that would allow the invalid node to stay in the search tree, I had picked out that setting the parents of the nodes being added would affect the search tree and how the search tree would recreate itself if it were to switch from one side of the tree to another.

When removing the wrong nodes from the search tree was a problem, I had to print out the object ids of all the nodes that were in the search tree currently and I found out that multiple nodes had the same object id. This was the reason why it wouldn’t delete because my algorithm will not delete a node if it is in the path of the node being added and due to having multiple nodes that were in the node’s path, I needed some way to separate each node from each other. This lead to me “remaking” the node before adding the nodes to the search tree or when I expand the node and find its children, I would remake the children as a whole new node which eliminated the mutual object id problem and allowed for every node to be unique.

Having incorrect calculations of the f\_score allowed for some nodes to be removed invalidly or unintentionally. I found that if I left the node to have the same object id, various nodes with that node being its parent will affect the other nodes with the same object id in the search tree or queue. This lead to me concluding to remake nodes to eliminate the fact that if one node were to be changed, then others would be changed.

***Testing***

For testing, I have used the graphs that were provided in the tutorial (a total of four undirected graphs) which fueled for editing and finding bugs in my program.

I started using the graph which was given to us on Blackboard (along with node to node values and heuristics) for testing because it was accessible to me and I didn’t need to write up any .txt files to use it. Basing my testing on this file, the memory limit of 15 for the memory limited A\* search was impossible because the graph was small. This led to me changing the memory limit of the graph to 10 temporarily to deal with the small graph and see if my program will pick up the memory overflow if I were to add a node to a full memory search tree. This allowed me to also test what nodes were being removed when I attempt to insert a node to a full memory search tree and print out the node’s contents to display heuristics and scores before and after the insertion. In turn, this enabled me to eradicate various problems in my program which were explained previously.

I used the same graph when testing my beam search algorithm for k = 1, k = 2 and k =3. Tracing by hand which nodes the search will reach, my algorithm will also print out the path it is taking each time the recursion happens. This allowed me to check whether the algorithm is doing what it is supposed to be doing. I tried for k = 4 on the same graph and my algorithm could deal with it due to my FOR loop I had implemented in the program which allowed various nodes to find its children at each depth and only k amount of nodes at each depth. This proved to have succeeded in testing beam search against one graph.

I tried both my programs for memory limited a\* and beam search against the other three graphs which I had to write up and the searches worked fine. I also tried a graph which was a straight line of 16 nodes and tested my memory limited a\* if it detected that the memory would be full when trying to add the goal (16th) node into the search tree.

***Conclusion***

In conclusion to this assignment, I was a very adventurous way to design a program which allows for searching a graph. Designing beam search felt more of a hassle due to having nothing to start off with and building from the ground up. Once the creation of beam search had finished and it was time to move on to memory limited A\* search, it felt less heavy because some of the framework had already been implemented in the previously created search. Utilising the Node, ReadFile and Weights class allowed the creation of my memory limited A\* search a whole lot easier. It was just the body of the searching function which made each of the searches harder and easier from each other. Overall, I felt that I have learnt quite a bit of the Java API while undertaking this assignment and I had found ways to be able to implement various things better such as printing out paths in my A\* search which I could have done better in my beam search by the use of recusion. But in the end, I don’t want to be touching something that works and go with whatever works and take what I have implemented. I didn’t consider time complexity or space complexity in this assignment other than the memory limit of 15 because I found that it wasn’t something the assignment was asking for. It might have been good to consider it, but I decided not to worry about something that isn’t going to affect the outcome of my assignment mark. It might have been wise to implement a fast algorithm, but again, I don’t want to be touching something that already works.