Adam Bates
300223031

Project 1 – Concurrent Search Algorithm
NWEN303 – Concurrent Programming

Introduction

This report documents Project 1 in NWEN303 in which a concurrent search algorithm was implemented. This algorithm was implemented in steps as described in the following requirements:

Step 0) Write a program to read a description of a directed graph, consisting of a list of edges(pair of nodes), the start node and goal nodes.

Step 1) Write a program to search a tree, finding all paths from the start node to goal nodes.

Step 2) Extend program so it handles graphs that are not trees.

Step 3) Extend your program so that it can find a single path to a goal node.

Step 4) Optional Extra, investigate other facilities provided by java.util.concurrent which would make it easier to implement the program.

Instructions

1) Import project into eclipse
   Or
   Copy Graphs folder and .java files into same directory. Run javac *.java via the command line.
2) Run FindPaths.java in eclipse or java FindPaths in command line.
3) Program will ask for a graph name. Files are located in Graphs. Type test1 or test2.
4) Program will ask if the graph is directed or non-directed. Type 1 for directed or 2 for non-directed.
5) Program will ask which mode you want to run. Type 1 for all paths (Step 1 and 2) or type 2 for one path(step3).

Step 0

The design of step 0 consisted in following the recommended format of reading a series of lines from a file where each line is either E n n', S n or G n, representing an edge, start node or goal node respectively.

Adam Bates
300223031

As each edge was read in from a file using a scanner, a Node object was created for each node to store the node information in. Each Node object consisted of a node id and an ArrayList to hold the node ids of all the nodes connected from it. A hash map was used to store a node Id mapped to the node object for easy retrieval of a node. A further option was added so a graph could be read in as a directed graph or a non directed graph (ie Links go both ways). The start node was stored in an integer variable and Goal Nodes were stored in an ArrayList so they could be used later.

Step 1 and Step 2

To find all the paths from a start node to goal nodes the following simple concurrent algorithm was used.

For each node connected to a node, compare each node connected to it with the goal nodes. If there are any matches, print the paths. Then for each connected node that hasn't been visited by the path, pass it a list of nodes that have been already visited on the path, including the current node and create a new thread that will follow the same process until all possible paths are found and printed.

The concurrency issues found in this implementation was that an arraylist passed to each child node had to be a fresh copy with the previous node appended to it. This is because java passes array lists as references and multiple paths were modifying the same arraylist causing errors in the paths because some nodes were marked as visited when they weren't visited by a certain path.

Step 3

Step 3 consisted of using the same code as Step 1 and 2 but just adding some conditions to it so when the first path was found it would inform the rest of the threads to terminate.

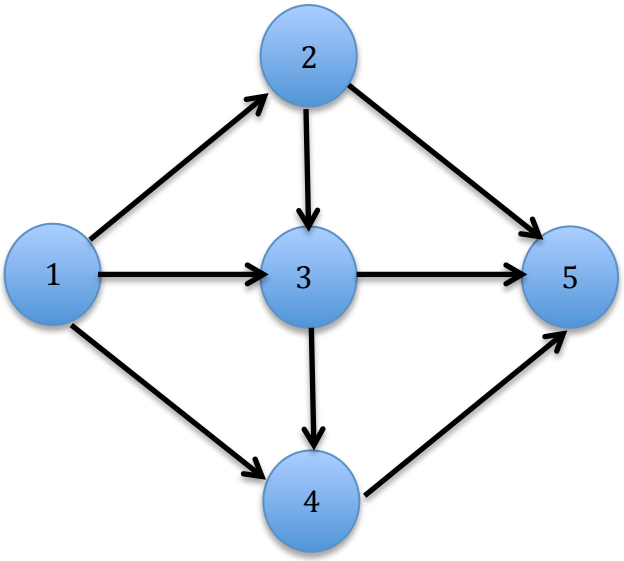
A concurrency issue appears in this step as only one path can be printed. Simply reading and writing to a shared Boolean variable doesn't cut it due to the fact that two or more threads could find a path at the same time and both update the shared variable at the same time and print multiple paths. To solve this problem a synchronized method (write())was used so only one thread could try to update a shared Boolean variable at a time. This means the first thread gets mutual exclusion to change the variable and then print their path and any waiting threads or executing threads that are checking the shared variable are terminated.

Adam Bates
300223031

Testing

Test 1 – Directed Graph

E 1 2
E 1 4
E 1 3
E 2 3
E 3 4
E 4 5
E 2 5
E 3 5
S 1
G 5



Start = 1, Goal = 5
Expected Output All Paths:

1->3->5
1->3->4->5
1->4->5
1->2->5
1->2->3->5
1->2->3->4->5

Actual Output All Paths : Same as expected

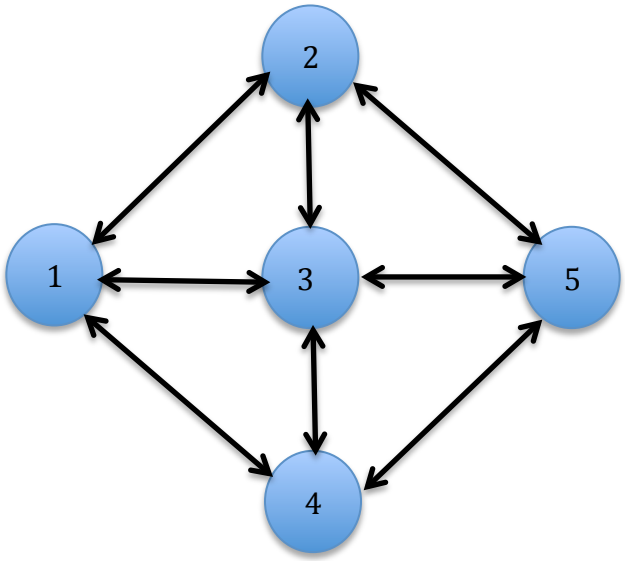
Expected Output One Path: Any one path from list of all paths.
Actual Output: 1 ->2 ->3 ->4 -> 5

****
Test Passed.
****

Adam Bates
300223031

Test 1 – Non directed Graph (2 way)

E 1 2
E 1 4
E 1 3
E 2 3
E 3 4
E 4 5
E 2 5
E 3 5
S 1
G 5



Start = 1, Goal = 5.
Expected Output - All Paths:

1->2->5
1->2->3->5
1->2->3->4->5
1->3->2->5
1->3->5
1->3->4->5
1->4->3->2->5
1->4->3->5
1->4->5

Actual Output – All Paths = Same as Expected

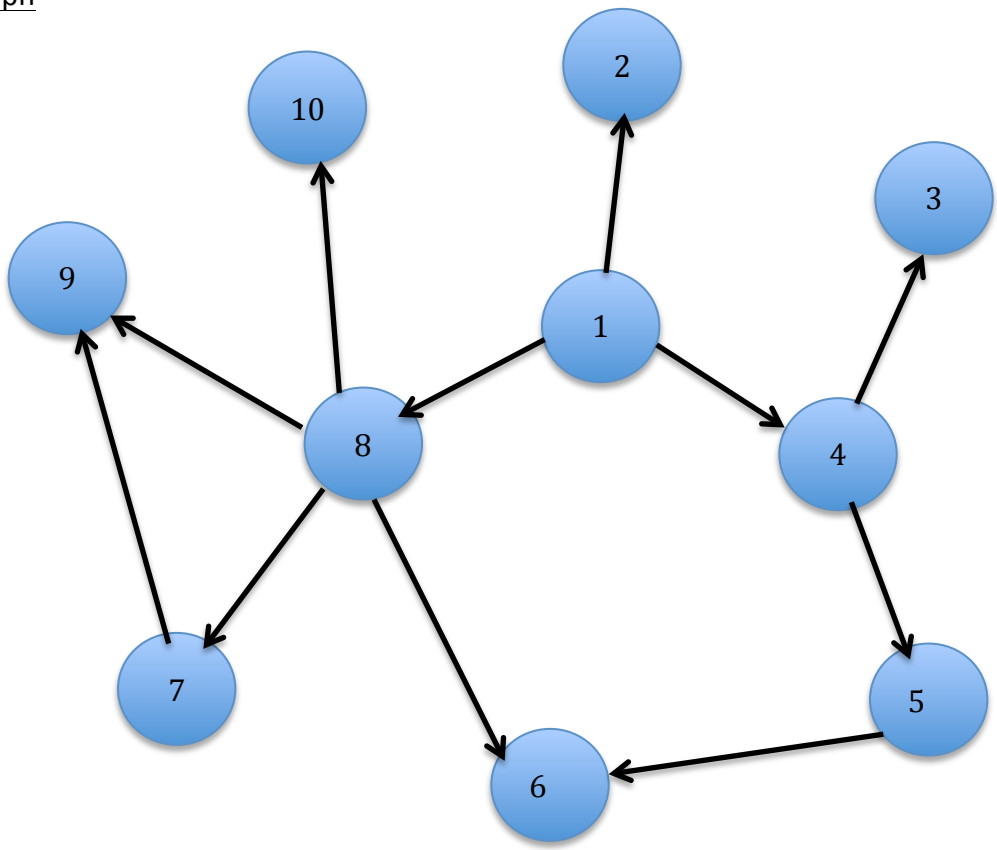
Expected Output – One Path: One path from all paths list.
Actual Output: 1->2->3->4->5

****
Test Passed.
****

Adam Bates
300223031

Test 2 – Directed Graph

E 1 4
E 1 2
E 1 8
E 8 10
E 8 9
E 8 7
E 7 9
E 8 6
G 6
E 4 3
E 4 5
E 5 6
S 1
G 5



Start = 1, Goal = 5, 6

Expected Output All Paths:

1->4->5
1->4->5->6
1->8->6

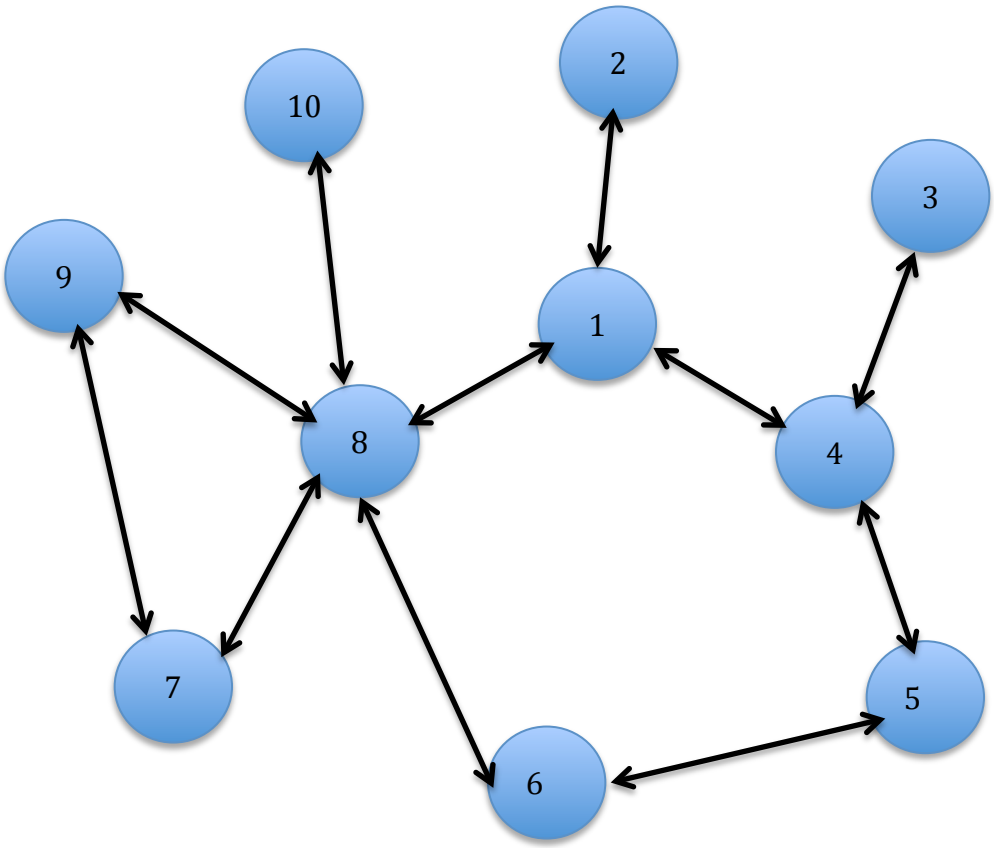
Actual Output: Same as Expected

Expected Output One Path: Can't have more than one goal for one path (step3)
Actual Output One Path: Error message "Can't have multiple goal nodes"

****
Test Passed.
****

Adam Bates
300223031

Test 2 – Non-Directed (Two way)



Start = 1, Goal = 5, 6
Expected Output All Nodes:

1->4->5
1->4->5->6
1->8->6
1->8->6->5

Actual Output All Nodes: Same as expected.

Expected Output One Path: Error two goal nodes
Actual Output One Path: Error Message “Multiple goal nodes”.

****

Test Passed.

****

Adam Bates
300223031

Performance

Auckland roads (file = auck)

4508 ->4534 ->4568 ->4606 ->4605 ->4590 ->4623 ->4636 ->4668 ->4655 ->4779 ->5040 ->5240 ->3 ->5694 ->5440 ->5437 ->5417 ->5482 ->5544 ->5566 ->5597 ->5605 ->5614 ->41403 ->5702 ->5767 ->5776 ->5839 ->5851 ->5890 ->5956 ->5975 ->6081 ->6110 ->6195 ->6244 ->39110 ->6306 ->6326 ->6335 ->6379 ->6423 ->42954 ->6676 ->6696 ->42094 ->42092 ->42093 ->6591 ->6476 ->6407 ->6381 ->6368 ->6346 ->6338 ->31028 ->6258 ->6245 ->6216 ->6267 ->6279 ->6280 ->6217 ->6215 ->6204 ->6197 ->6169 ->6150 ->6149 ->6163 ->6171 ->6152 ->6105 ->6097 ->40499 ->6104 ->6099 ->28696 ->6106 ->6063 ->6023 ->6015 ->40378 ->6011 ->5955 ->5886 ->5831 ->5877 ->5917 ->5992 ->6107 ->6157 ->6720 ->6813 ->39723 ->39719 ->7048 ->7102 ->7153 ->7171 ->7204 ->7248 ->7274 ->7332 ->7261 ->7206 ->7184 ->7168 ->7260 ->7249 ->7185 ->7129 ->7116 ->7101 ->6619 ->6259 ->6471 ->6281 ->6246 ->6200 ->6128 ->6090 ->6121 ->6074 ->6040 ->5991 ->5895 ->5858 ->5852 ->5824 ->41888 ->6737 ->7002 ->7309 ->7915 ->7957 ->8266 ->39751 ->39752 ->39753 ->39754 ->8899 ->8900 ->8880 ->8867 ->8722 ->8679 ->8611 ->8565 ->8444 ->8274 ->8236 ->8235 ->8178 ->8225 ->8273 ->8308 ->8329 ->8264 ->8176 ->8147 ->8052 ->7970 ->7823 ->7813 ->7842 ->7843 ->7873 ->7872 ->7913 ->7972 ->7993 ->8011 ->8115 ->8116 ->8179 ->8227 ->8226 ->8265 ->8239 ->8275 ->8295 ->8330 ->8508 ->8535 ->8636 ->8680 ->8748 ->8749 ->8750 ->8751 -> 8777
8777

No delays at node:

Lighthouse: 124.0 ms
Mac i5 2.4Ghz Duel Core: 16.0 ms

With delay of 500ms per node.

Lighthouse: 226258.0 ms
Mac i5 2.4Ghz Duel Core: 226546.0 ms

Lighthouse faster by 112 ms.

Adam Bates
300223031

Results and Conclusion

The results of the testing in this project showed that the implemented concurrent graph search algorithm successfully worked. The performance analysis was interesting because the concurrent search was slower on the 5 core, 16 processor machine when new threads were created with little workload. As soon as more workload was added per thread the theoretically faster machine (Lighthouse) was the fastest which was expected because it had more processing power to process more threads at once.