NWEN303 – Project 1

# Explanation of code structure

I have implemented five main classes (along with Node.java):

* **Search.java** – base class for all search classes.
* **TreeSearch.java** – subclass of Search.java. Concurrently searches a tree finding all solutions.
* **GraphSearch.java** – subclass of Search.java. Concurrently searches a tree / cyclic graph, finding all solutions.
* **GraphSearchThreadPool**.java – subclass of Search.java. Concurrently searches a tree/cyclic graph finding all solutions, but with a limited number of threads.
* **GraphSearchFindOne.java** – subclass of Search.java. Concurrently searches a cyclic graph, finding just one solution.
* **FileReader.java –**used by Search.java, and is responsible for reading in the graph from a file.

# Explanation of file format

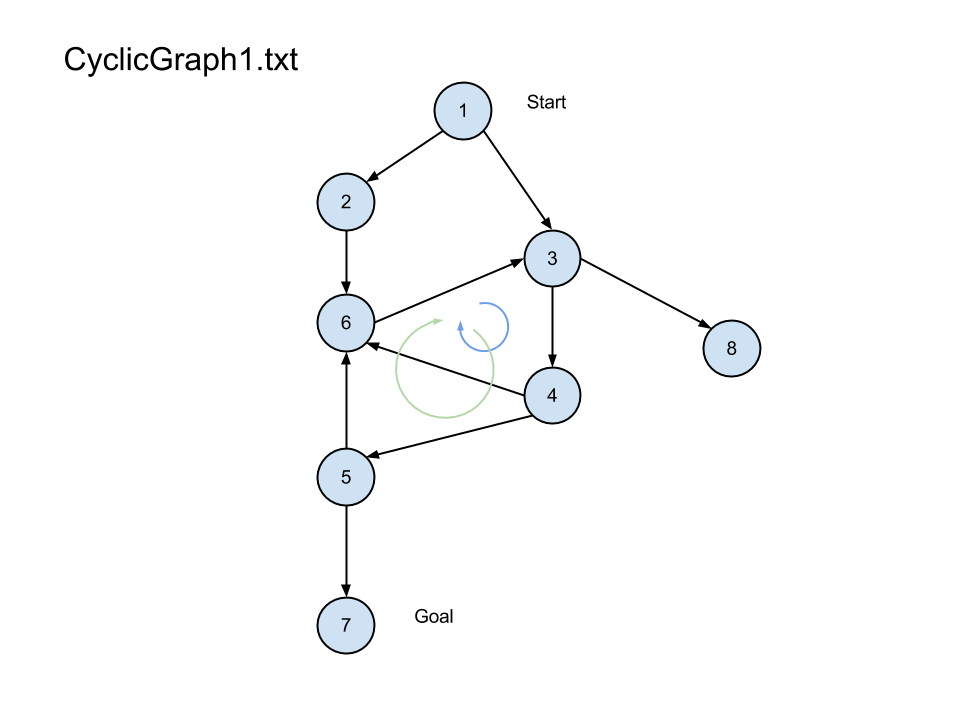
The file format consists of:

* First, all edges between node numbers on a separate line, separated by a space

‘<nodeId1> <nodeId2>’

* Then:
  + Start node denoted ‘S <nodeId>’
  + Goal node(s) denoted ‘G <nodeId>’

## Example:



1 2

1 3

2 6

3 8

3 4

4 5

4 6

5 6

5 7

6 3

S 1

G 7

# Explanation of output

In the above example, there are two cycles, and two paths to the goal:

* 1 3 (4,6,3)\* (4,5,6,3)\* 4 5 7
* 1 2 6 (3,4,6)\* (3,4,5,6)\* 3 4 5 7

Each cycle is counted as the same if it contains all the same nodes, and duplicates are not recorded. I am assuming that connections are not ever symmetric to simplify this – this doesn’t make the search algorithm any different, but it is just more complicated to compare cycles to see if you’ve found it or not (eg. 3 (4,6,3)\* and 3(6,4,3)\* are counted as the same in my solution).

Cycles start from their entry point.

* In the above example, there is no cycle starting from 4 because threads that find a cycle will record it and stop, and the cycle starting at 3 would be found first.
* Both solutions above contain the same cycles, but starting at different entry points

# How to run the different search algorithms

I have implemented separate classes for each version of the search. You can run all types of search through Search.jar (the main method is on Search.java class).

* Java –jar Search.jar <searchType> <filename> <numThreads>

**SearchTypes:**

* tree – TreeSearch.java
* graph – GraphSearch.java
* graphFindOne – GraphSearchFindOne.java
* graphThreadPool – GraphSearchThreadPool.java – requires the number of threads.

Each variation of the search algorithm has its own internal Thread/Task class that does the searching. The Search() method in each class initiates the search, and then it calls the superclass to interpret and print the results when done.

# Step 1 – Implementing a concurrent tree search (TreeSearch.java)

## Explanation

The TreeSearchThread takes the nodeid, and the path so far.

For a tree, for every node, there will only ever be one path from the start node to it. I decided to save the answer in the node each time. This means that there is more freedom – if I had one shared set that contained all answers, then no answers could be recorded simultaneously, even if they were recording an answer to completely different goal nodes.

Although this takes a bit of extra work to compile the answers at the end, it would be worth it if there were lots of goal nodes, and the recording of answers was something that was frequent.

My solution doesn’t implement any locking mechanisms, as I don’t need to worry about multiple threads recording an answer in one node at the same time, or about cycles.

## Testing

I tested using Tree1.txt.

1 2

2 3

2 4

S 1

G 4

G 2

The paths to the goal are, with the path to the first goal being within the path to the second:

* 1 2
* 1 2 4

To test, I made sure that there were no errors thrown, and that the expected paths were found:

Time taken: 1ms

NumAnswers: 2

1 2

1 2 4

# Step 2 – Implementing a concurrent graph search (GraphSearch.java)

## Concurrency issues

There are three areas that need to be dealt with regarding concurrency.

#### Recording an answer

The first is recording an answer in a node. Unlike with a tree, a Graph can contain multiple paths to the same node, so we need to make sure we synchronise recording an answer in a goal node. In this solution, we just need to synchronize on the Node object. This was done by synchronising the RecordSolution(String solution) method in the Node class.

#### Recording a cycle

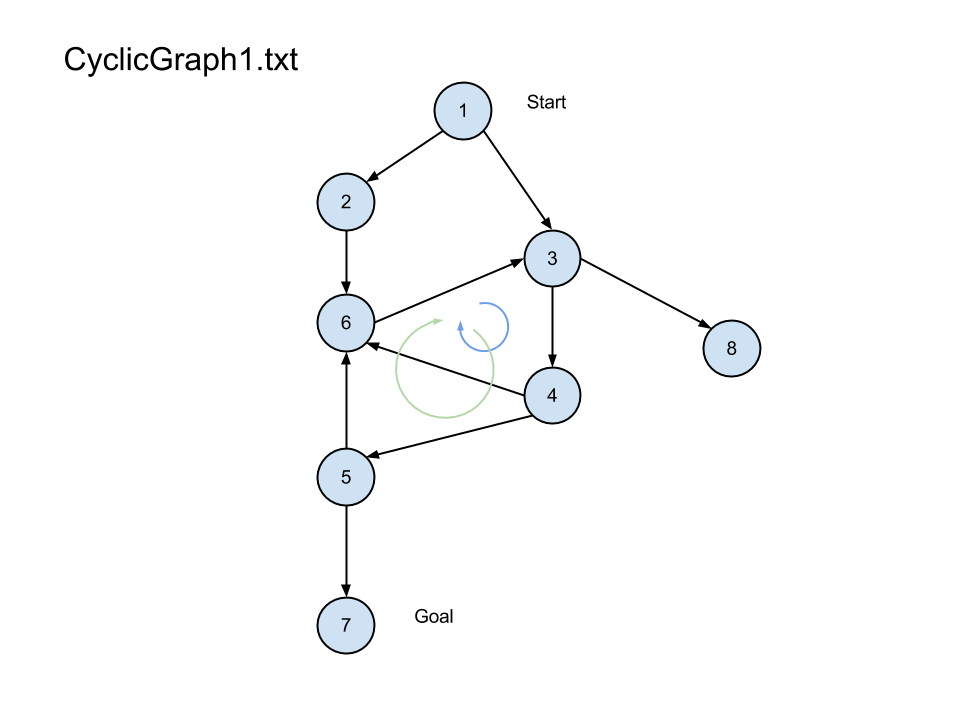
The second is dealing with cycles. The way I decided to do it is to record each cycle found in a shared map from the entrypoint nodeid to a string describing all cycles through that node.

Because each path has its own set of visited, whenever a thread is started for a node already visited in the path, it knows the path contains a cycle, so it records it if it is new, and then stops searching.

Because multiple threads could be adding cycles at once, this needs to be synchronized somehow. As with recording answers, we probably don’t want to restrict all nodes from adding cycles at once if the nodes are different. It should also be fine to add a cycle to a node at the same time as adding an answer. The solution I came up with is for each node to have two separate objects to lock on. One to lock for adding cycles and one to lock for adding answers. This can be seen in the Node class.

What I’ve done is effectively create two monitors, without the extra classes.

## Testing

The bulk of the testing was done using CyclicGraph1.txt.

When running CyclicGraph20.txt on vivo (2 cores)

Timetaken: 1997ms

NumAnswers: 193

Time taken: 1948ms

NumAnswers: 193

When running CyclicGraph50.txt on Lighthouse (80 cores)

Time taken: 2583ms

NumAnswers: 193

Time taken: 2570ms

NumAnswers: 193

Time taken: 2601ms

NumAnswers: 193

It seems to actually take longer to run on lighthouse than it does on a normal 2 core processor.

# Step 3 – Implementing a concurrent graph search to find just one solution (GraphSearchFindOne.java)

## Concurrency issues

This solution is similar to that of Step 2. The difference here is that we need a global variable to keep track of whether a solution has been found or not.

The extra issue with concurrency here is that we don’t want multiple answers to be recorded at once, otherwise, we will get more than one answer.

## How it was solved

The runnable method was edited so that the thread checks whether the answer has been found when it first starts, and if it has, it exits.

In order to do this safely, I added an AnswerMonitor class in GraphSerachFindOne. If a thread wants to record an answer, it calls AnswerMonitor.RecordAnswer(Node n, String path). Within the method, it checks that found is still false, and if so, records the answer in the node, and sets found to true.

I was going to just have an object to synchronize over, and just put the logic in a synchronized block. It would have taken up less space, but this is more encapsulated.

## Testing and results

I tested this with CyclicGraph1.txt to make sure it finds a correct solution:

# Step 4 – Implementing with graph search with a thread pool

There was a problem with the Step 2 solution to find all paths, where all threads created had to remain active while their child threads were still active, and if there is a new thread for each child unexplored in the path, then that is a lot of threads, and the memory use will severely limit the size of graph that can be searched.

I tried implementing