Isabel Broome-Nicholson
BROOMEISAB
301020046

NWEN303 – Project 1

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).

Example:

```
> java -jar Search.jar searchType=graph filename=CyclicGraph12.txt numThreads=4 printCycles=false
```

The four different arguments are all optional and can go in any order:

- **searchType**: Possible values are {tree, graph, graphFindOne}. Defaults to graphFindOne. Only 'graph' will pay attention to the numThreads.
- **filename**: Name of file to load in. Defaults to whatever the class default file is
- **numThreads:** Positive int. Max number of threads to run. Defaults to unlimited.
- **printCycles:** Possible values are {true, false}. Whether or not to include the cycles in the printed answer. Defaults to false. I wouldn't recommend making this true for large data sets or you will run off the screen.

Each variation of the search algorithm has its own internal Thread/Runnable 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.

Explanation of code structure

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

- **Search.java** – base class for all search classes, and program entry point.
- **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. It can also generate new graphs.

Isabel Broome-Nicholson
BROOMEISAB
301020046

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>'

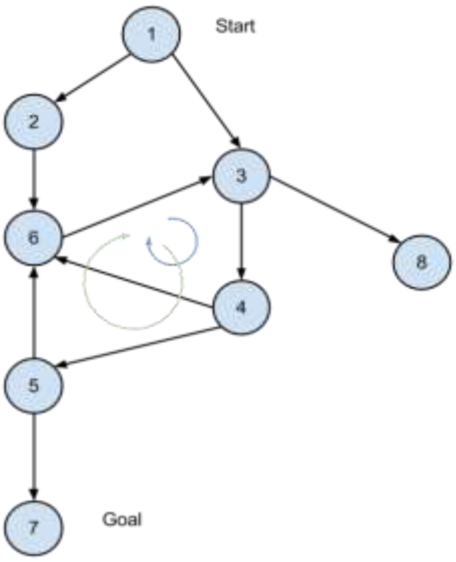
Then:

- Start node denoted 'S <nodeId>'
- Goal node(s) denoted 'G <nodeId>'

Example:

CyclicGraph1.txt

```
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

You can turn cycles off, as they are pretty massive for larger datasets.

Isabel Broome-Nicholson
BROOMEISAB
301020046

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

Concurrency issues

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. For a tree, for every node, there will only ever be one path from the start node to it, so we don't need to worry about locking access to nodes.

However, there is one concurrency issue to deal with:

Working out when the search is complete

In my initial tree solution, I made sure each thread waited until all child threads had stopped executing before terminating. I originally did this so I would know when I the search had finished by calling join() on the first thread created. However, this does not scale well, and uses WAY too much memory for large datasets!

The tricky thing is that each Runnable doesn't know if it is the last one to compute or not. We need to keep track of all threads being started, make sure that all that have been started have finished, and be somehow certain of the point that there aren't any more that will be started.

A solution to this is to have a shared list of all threads that are started, and make sure that when all threads have been added, they have all stopped executing before you try printing the answer.

In order to do this, I created another monitor-like-class in the GraphSearch class. This monitor holds a linked list of threads. Threads can add any newly created threads using the Add(Thread t) method, which is synchronized to make sure the operation is safe. The main thread can also call Wait(), which blocks it until we are confident that:

1. All threads that will be started are in the list, AND
2. All threads in the list are finished

As new threads get added to the list, the 'size' variable is incremented. This means that 'size' is guaranteed to be either the same as the list.size(), or less than it. Because we are only adding threads to the end, we can iterate through the list from the start as long as we don't overtake the size.

Also, because the polling thread is one thread, and there will be more than one thread adding new threads to the list, it is unlikely that the polling thread will catch up before all child threads have been added. However, if each thread took extra time to process, we would want to be certain that if we had caught up with the end of the list, no new threads would be added. This is where the 'waitTime' variable comes in – if we have caught up, we will wait however long we expect to have to wait to give the other threads a chance to add more to the list.

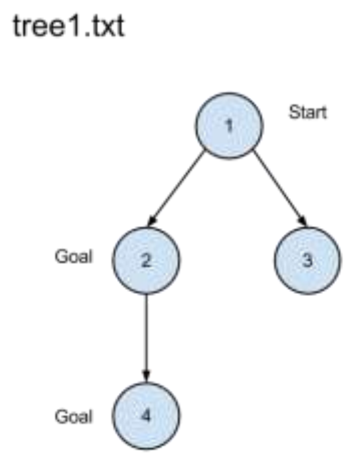
Hopefully this solution will mean we won't run out of memory (until we end up starting too many threads at once).

Isabel Broome-Nicholson
BROOMEISAB
301020046

Testing

Correctness

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
```

Performance + memory use

I didn't test larger files on the TreeSearch class – this was done in later stages on GraphSearch.

However, I did add the "Working out when search is complete" feature nearer the end, so this was actually tested with the later graph solutions. In the original version, I kept all threads alive until children were done. With the old version, I noticed that I would get out of memory exceptions at about thread 10,000 when running roadsFile.txt (massive dataset). With the added feature (and without thread pools), exceptions don't happen until you get up to about 30,000 threads.

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

Concurrency issues

There are two new 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. This was done in the Node.AddAnswer(String answer) method, by locking on the list of answers within the node, which is basically a really simple monitor.

I originally locked on the Node object, but because cycles need a separate lock, I changed it.

Isabel Broome-Nicholson
BROOMEISAB
301020046

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 entry point node id 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.

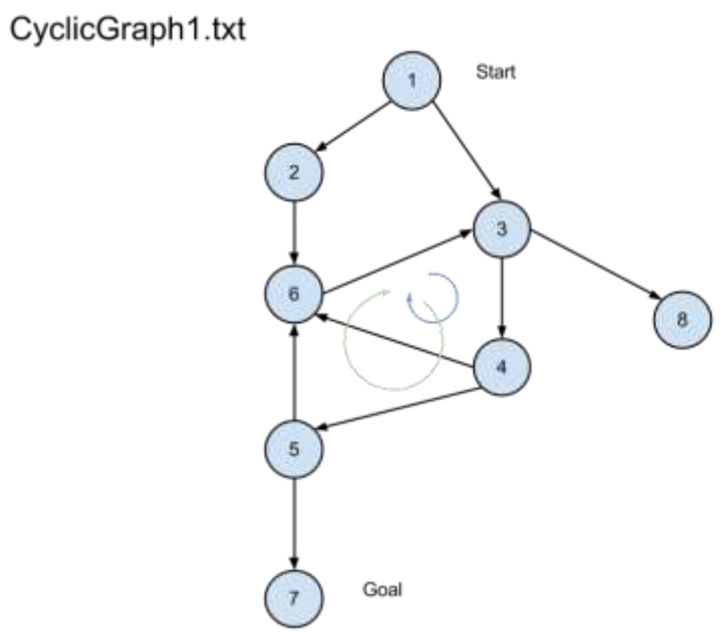
This needs to be synchronized somehow because multiple threads could be adding cycles at once. As with recording answers, we probably don't want to restrict all nodes from adding cycles at once if the nodes are different. It should also be fine to add a cycle to a node at the same time as adding an answer.

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

Correctness

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



I made sure that this would find the two different paths with correct cycles:

- 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

Isabel Broome-Nicholson
BROOMEISAB
301020046

Speed

In order to test the speed performance, I used CyclicGraph50.txt, which has 50ish nodes and 50ish double sided edges. It results in 193 solutions, and takes a few seconds:

Machine	vivo (2 cores)	home(4 cores)	lighthouse (64)
test 1	1916	1264	2284
test 2	1908	1218	2303
test 3	1903	1246	2418
test 4	1918	1257	2216
Average	**1911.25**	**1246.25**	**2305.25**

With an unlimited number of threads, it is interesting to see the differences don't seem to have much to do with the number of cores available. It seems to actually take longer to run on lighthouse than it does on a normal 2 core processor, or my 4 core processor at home. I wonder whether this is to do with the CPU processing speed rather than the concurrency.

When I think about the algorithms I have implemented, it doesn't seem like there would be much blocking happening, as the only time it would happen is when two threads are recording an answer, or a cycle in the same node at the same time, which wouldn't happen that often. Even when it does happen, it wouldn't block for long at all.

Memory use

In order to push the memory load, I ran search on roadsFile.txt, which is the COMP261 roads data. It has about 40,000 nodes, and 40,000 edges, and the start and end haven't been found by any of my solutions yet.

When the search is run, exceptions happen at about thread 30,000:

```
Exception in thread "Thread-30732" java.lang.OutOfMemoryError: unable to create new native thread
	at java.lang.Thread.start0(Native Method)
	at java.lang.Thread.start(Unknown Source)
	at GraphSearch$GraphSearchThread.run(GraphSearch.java:108)
```

I think this means that too many new threads are being created faster than they are completing, and indicates some kind of thread pool is necessary to limit this.

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.

However, there is an added concurrency issue:

Isabel Broome-Nicholson
BROOMEISAB
301020046

Global found variable

The difference 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.

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 the GraphSerachFindOne class. 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 code space, but this way is more encapsulated.

Testing

Correctness

Again, I tested this with CyclicGraph1.txt to make sure it finds a correct solution, which it does.

Speed

Testing it against CyclicGraph50.txt is very quick, and it generally finds a solution in 2ms.

Memory use

Testing it against roadFile.txt, it crashes at about 30,000 threads as it can't find a solution below that. This indicates I need to implement some kind of thread limiting.

Step 4 – Implementing with graph search with a thread pool (GraphSearchThreadPool.java)

Concurrency issues

Up until here, I had made sure that threads aren't being kept alive for longer than necessary, which greatly improved the amount of memory my solution used. However, obviously if you create a new thread for each recursive step, with a large dataset, you are still going to run out of space for threads (just not as fast).

To improve this, I implemented GraphSearch using a thread pool. The code itself is straightforward, and similar to that of GraphSearch. The only real difference is that you make the search class a Runnable, rather than a thread, and used a fixed thread pool to execute the runnable tasks.

I didn't end up implementing GraphSearchFindOne.java with thread pools as I ran out of time, but it would stop it crashing on the large datasets.

Isabel Broome-Nicholson
BROOMEISAB
301020046

Testing

Correctness

I tested this with CyclicGraph1.txt again to check it was correct, which it was.

Speed

I tested my implementation with CyclicGraph50.txt for speed on my computer (4 cores):

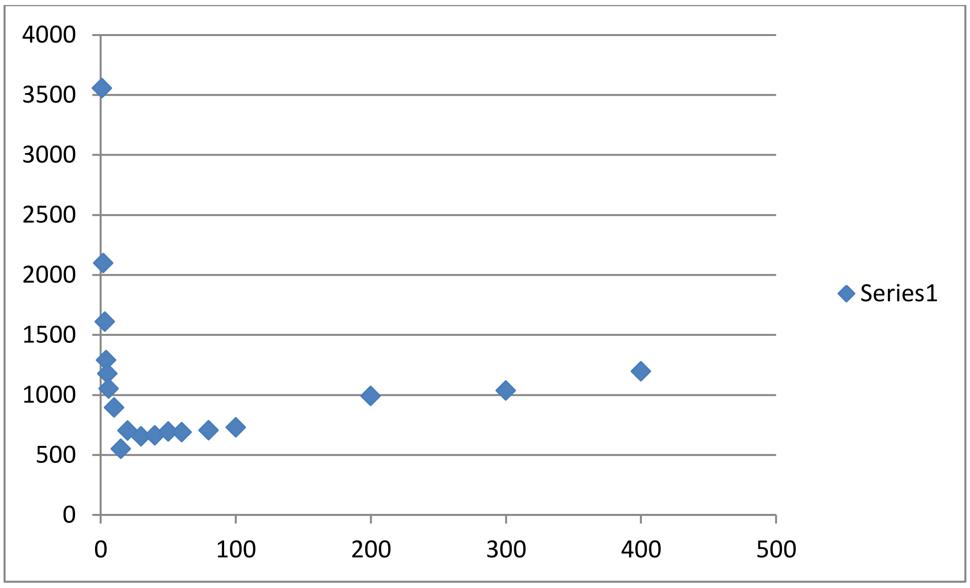
NumThreads	unlimited	1	2	3	4	5	6	7	8	9	10	11	12	13	14
test 1	1264	226	158	264	264	292	262	193	220	226	199	181	179	197	205
test 2	1218	226	155	272	273	265									
test 3	1246	227	155	256	250	275									
test 4	1257	225	155	267	248	277									
Average	1246.25	226	155.75	264.75	258.75	277.25	262	193	220	226	199	181	179	197	205

It seems like 2 threads is optimal, but it fluctuates quite a bit as the threads increase. The fastest speed I recorded with limited threads was with 2 cores, which was about 87% faster than unlimited threads. I'm guessing this is to do with the overhead of creating, waking and pausing all the threads.

I also tested on lighthouse with a different number of threads:

Testing on lighthouse (64 cores)

NumThreads	unlimited	1	2	3	4	5	6	10	15	20	30	40	50	60	80	100	200	300	400
test 1	2284	3555	2098	1610	1288	1175	1050	893	548	701	652	662	693	688	704	729	990	1036	1194



Interestingly, the minimum time seems to happen at around 15 cores.

Isabel Broome-Nicholson
BROOMEISAB
301020046

Memory use

I used roadsFile.txt (which is massive) to test memory use. With the normal GraphSearch with unlimited threads, exceptions happen at about 30,000:

```
Exception in thread "Thread-30732" java.lang.OutOfMemoryError: unable to create new native thread
	at java.lang.Thread.start0(Native Method)
	at java.lang.Thread.start(Unknown Source)
	at GraphSearch$GraphSearchThread.run(GraphSearch.java:108)
```

When I ran search on roadFile.txt with 4 threads, it just keeps searching, rather than crashing, so I count that as success.

Final thoughts

The synchronisation seemed fairly basic, and it didn’t seem like there was much benefit to implementing a strict monitor with all methods synchronised. I think all my monitor-type-things only ever had a max of one locked method, which made it almost simpler to implement using an object to lock on, and synchronized blocks.