Algorithm Analysis

getShortestPath

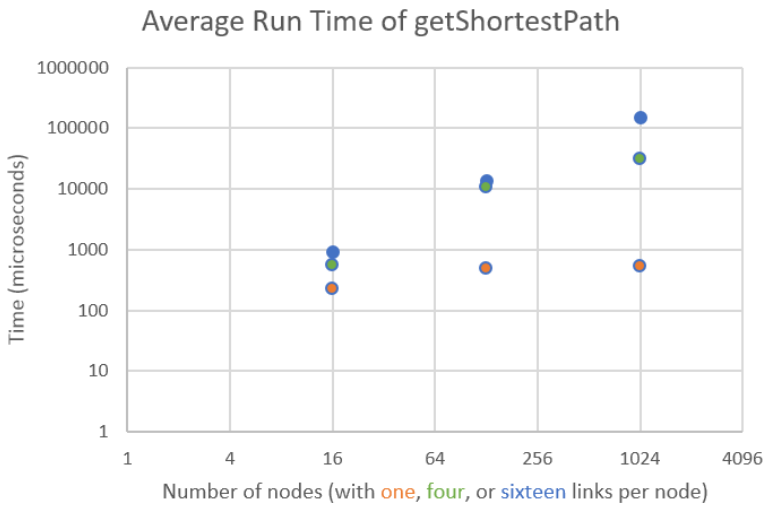
Description of the algorithm:
The method used to find the shortest path was Dijkstra's algorithm implemented for a directed unweighted graph. This works by finding the unvisited adjacent nodes which are at a minimum distance from the start point and working through the graph until a minimum path is found to the finish point. The number of links in the path is returned.

Pseudo code:
Set all distances in the matrix distances to ∞
Set the distance at the start to 0
While there is a vertex that hasn't been visited
    Pick a node (min) which has the smallest distance to it and is not already visited
    For every node
        If the node is adjacent to min and it hasn't been visited and the pre-calculated distance to it has a greater distance than min+1 then
            Set the new distance to this vertex as the distance to min + 1
        End if
    End for
End while
Return the distance to the urlTo

Source of the algorithms
This algorithm was sourced from this website: http://techieme.in/shortest-path-using-dijkstras-algorithm/

Complexity analysis
The complexity of this algorithm is $O(|V|^3)$ as there is a double nested loop iterating $O(|V|)$ times. The second loop contains a method (adjacentTo) which has a complexity of $O(|V|)$ therefore the overall complexity is $O(|V| \times |V| \times |V|)$ which is $O(|V|^3)$.

Performance



This graph shows the runtime of the method getShortestPath for graphs with different amounts of nodes (16, 128, and 1024) each with a different amount of links between them (1, 4, or 16 links per node e.g. the leftmost data points were obtained from a graph with 16 nodes and are coloured orange, green, and blue to represent the number of links in the graph: 16, 64, and 256 respectively). Logarithmic axes were used to more accurately display the difference between the runtime of the smaller graphs as the runtime increases exponentially. A wide variety of graphs were used to test the runtime ranging from 16 nodes with 16 links to 1024 nodes with 16384 links. This shows how the function performs for small, sparse graphs to large, full graphs. The fact that the runtime does not change significantly for significantly different sized graphs (especially for sparse graphs) shows that the main contributing factor to runtime is the number of links. The reason for that is that this method has to find the shortest distance between 2 points, and the more paths there are, the more work the function has to do.

getHamiltonianPath

Description of the algorithm:
The method used to find the Hamiltonian path was backtracking by using a depth first recursive search. The Hamiltonian path was found by checking every path in the graph starting from every point and checking to see if it is a valid Hamiltonian path by making sure it is of the correct length and that no node is repeated.

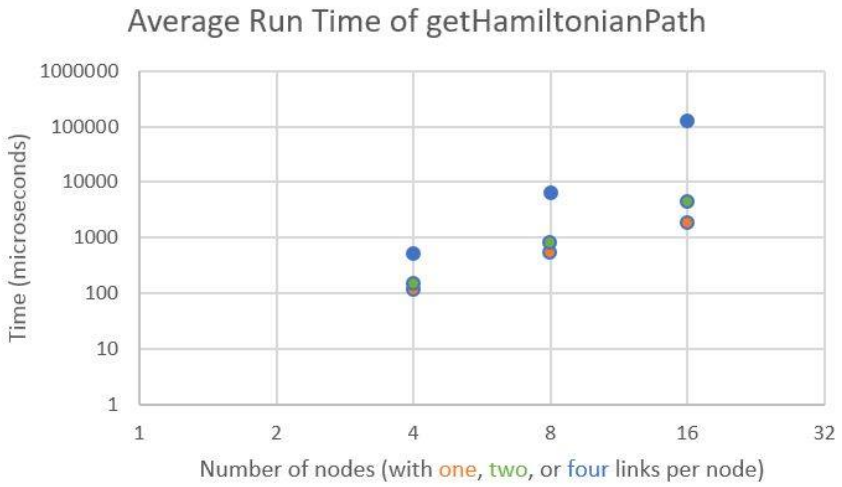
Pseudocode
Create an adjacency matrix graphMatrix
For every node, call the recursive function depthFirstSearch(an ArrayList path with the current node as its only entry)
Return the Hamiltonian path found by depthFirstSearch

depthFirstSearch(path, the current path to try)
If no Hamiltonian path has been found, then
    If path is the right size, then set it to be the answer and stop computation
    For every child of the last node in the path
        If the child is not already in the path
            Add the child to the path
            Call depthFirstSearch(path)
            Remove the last node in the path
        End if
    End for
End if

Source of the algorithms
The algorithm was adapted from an explanation about backtracking in Hamiltonian cycles found here https://www.youtube.com/watch?v=7VilgF4B9Tg

Complexity analysis
The complexity of the Hamiltonian path algorithm is $O(|V| \times |V|!)$ as the depth first search of complexity $|V|!$ has to be called V times, once for every starting point.

Performance



The graphs used to test the running time of the getHamiltonianPath method where all the graphs used did not have a Hamiltonian path. This was done to find the worst-case runtime of the method. As with the other graphs a logarithmic axis was used as the complexity of the method is polynomial and therefore the runtime increases in such a way that the data is best shown on a logarithmic scale. The size of the graphs tested range from 4 nodes and 4 links to 16 nodes and 64 links. Graphs with 4 links per node had a much longer runtime than graphs of 1 or 2 links especially for graphs of more nodes, this shows that while number of nodes has a contributing factor number of links have a much higher contributing factor. This is because the increase in possible paths is much higher when several links are added compared to adding a few nodes. Therefore the depth first search will check more paths, and it will take longer.

getStronglyConnectedComponent

Description of the algorithm:
The method used to find the strongly connected components of the graph was Kosaraju's algorithm. Kosaraju's algorithm works by doing a depth first traversal and putting the results onto a stack. It then does another depth first traversal but uses the elements from the stack instead of from the graph. This effectively reverses the order of the vertices.

Kosaraju's algorithm works by doing a recursive depth first traversal of the graph, pushing the vertices onto the stack from the bottom depth up once a particular branch has finished. All of the vertices will be entered into the stack.
Every link in the graph is now reversed, so the reverse graph is obtained.
Pop the top vertex from the stack, and traversal the graph from that vertex. Every node visited is part of one simply connected component.
Pop elements from the stack for every vertex in the simply connected component obtained.
Repeat until the stack is empty. Every simply connected component will have been obtained.

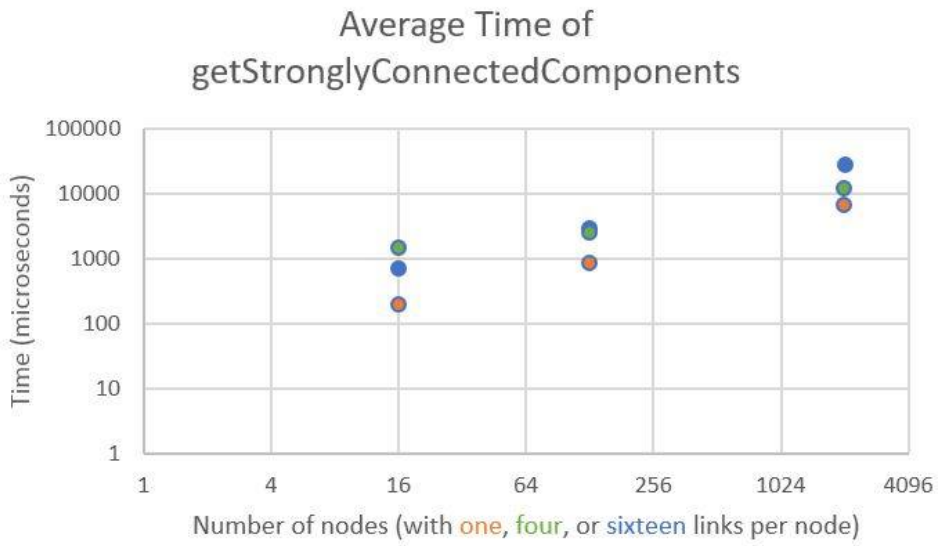
Pseudo code:

```
Create an adjacency list graphMatrix
Create a boolean array of size v called visited, all false
For each vertex
        Perform depthFirstRecursion1 on graphMatrix adding elements to a stack
End for
Set every value of visited to false again
Create reversedGraph, where all the links are reversed
Set an integer count to 0
While the stack is not empty
        Perform depthFirstRecusion2 on the stack
        Iterate count
End while
Return a v by v array of strings with the results of the DFS

depthFirstRecursion1(node)
Set the value of visited at the current node to true
For every child of node
        If the child is not visited
                depthFirstRecursion1(the child)
        End if
End for
Push the current node to the stack

depthFirstRecursion2(node, count)
Set the value of visited at the current node to true
For every child of node in the reversed graph
        If the child has not been visited already
                depthFirstRecursion2(child of the node, count)
        End if
End for
Add the node to the current strongly connected component
```

Source of the algorithms
This algorithm was adapted from an explanation about Kosaraju's algorithm which can be found here http://scienceblogs.com/goodmath/2007/10/30/computing-strongly-connected-c/

Complexity analysis
The time complexity of Kosaraju's algorithm is $O(|V| + |E|)$ due to two consecutive DFS recursions, one of $O(|V|)$ and one of $O(|E|)$.

Performance



This graph shows to running time for the method getStronglyConnectedComponents. The range of graphs tested were 16 nodes with 16 links to 2048 nodes with 32768 links. One thing to note here is that while the running time still increases rapidly enough that a logarithmic scale is needed, the runtime changes by less than a factor of 10 between an increase of one to sixteen links per node (e.g. the leftmost data points are run on graphs with 16 nodes with 16, 64, and 256 links from each node). This indicates that the method is quite efficient at handling an increase in links and this is reflected by the complexity being $O(|V| + |E|)$.

getCenters

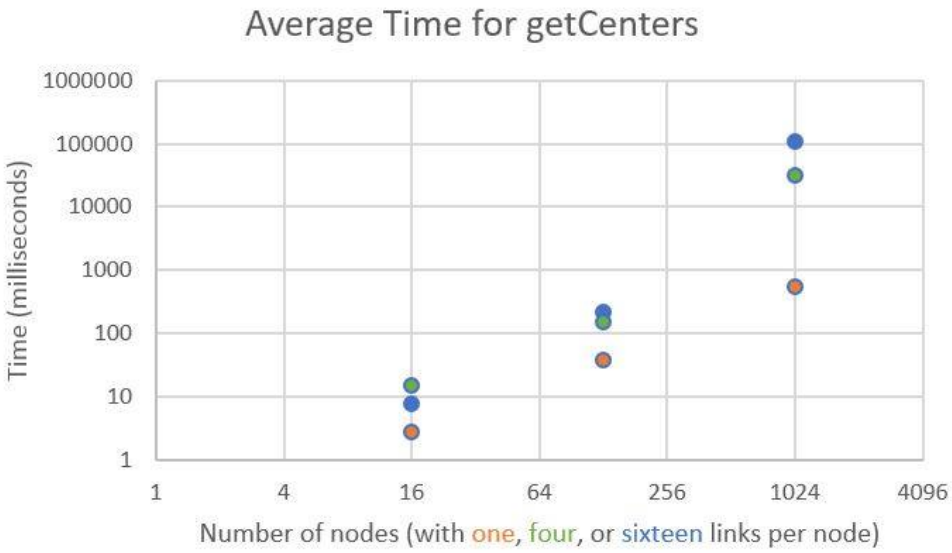
Description of the algorithm:
This method is used to find the vertex or vertices that have the shortest path to all other vertices in the graph. It works by creating a matrix of shortest paths then removing all nodes that can't reach every other node. The remaining nodes are scanned to find out how long it takes to get to the node furthest away and then shortest of these distances are returned.

Pseudo code or descriptions of algorithms:
For every node
    Get the array of shortest paths from the current node to every other node
    Add this array to a 2D ArrayList called distances
End for
A: For each row in distances
    Set an array of size 2 called max to 0, 0
        For each element in the current row of distances
            If distance = ∞ then return to A
            If the current node is greater than the first entry of max then set it to the current node
        End for
    Add max to an array of max distances for each node
End for
Set an integer min to ∞
For each element in the array of max distances
    If the distance at the current element of max is less than min then
        Set min to the distance at the current element of max
    End if
    If the distance at the current element of max is equal to min, then iterate count
End for
Put all the groups of strongly connected components into an array, centres
Return centres

Source of the algorithms
No source was used for the construction of this algorithm

Complexity analysis
The complexity of getCentres is $O(|V|^4)$ because getShortestPath (which has a time complexity of $O(|V|^3)$) is run V times thus giving the overall complexity as $O(|V| \times |V|^3)$.

Performance



This graph shows the running time of getCentres for graph sizes ranging from 16 nodes and 16 links to 1024 nodes and 16384 links. One interesting thing to note is that the running time for graphs of 16 nodes and 64 links is actually greater than that of 16 nodes and 256 links. This is because there are more connections between nodes and therefore the shortest path can be found more quickly. The data shows exponential increase for more nodes and therefore a logarithmic scale must be used. An interesting trend in the data is that while for sparse graphs the runtime increases linearly on the log axis whereas for full graphs the runtime increases more exponentially. This shows that the number of links have a greater impact than the number of nodes.