Hamiltonian Cycle Part 2

Assessment Task 3 - Group Programming Task

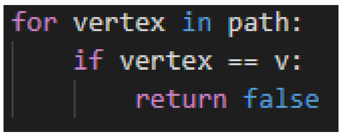
Euan Mendoza
Jonathan Nguyen 13963133

Table of Contents

The complexity of your implementation (indicating the level of confidence in any external components used as applicable and necessary).

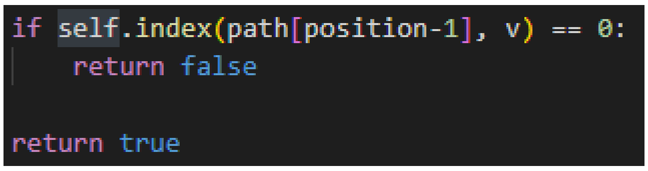
Backtracking

Path validity function:



```
for vertex in path:
    if vertex == v:
        return false
```

- For loop traverses the entire sequence and outputs the data, it is O(n)
- If statement runs in constant time O(1)

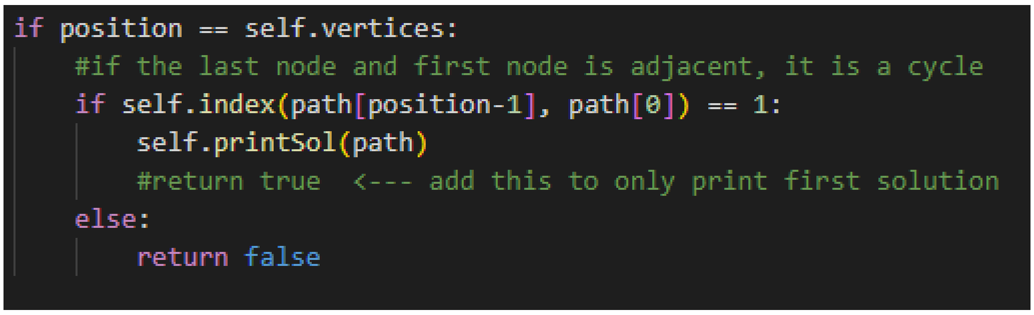


```
if self.index(path[position-1], v) == 0:
    return false

return true
```

- If statements runs in constant time O(1)
- The statement runs in constant time as well

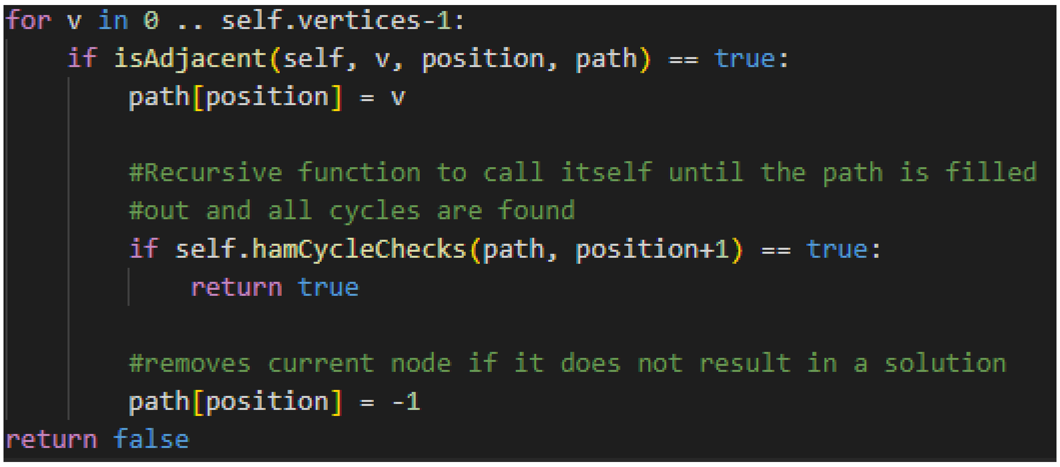
Therefore, the worst case time complexity for the path validity function is **O(n)**

Hamiltonian cycle utilities function:



```
if position == self.vertices:
    #if the last node and first node is adjacent, it is a cycle
    if self.index(path[position-1], path[0]) == 1:
        self.printSol(path)
        #return true  <--- add this to only print first solution
    else:
        return false
```

- If statements run in constant time O(1)
- Linear search inside array is O(n)

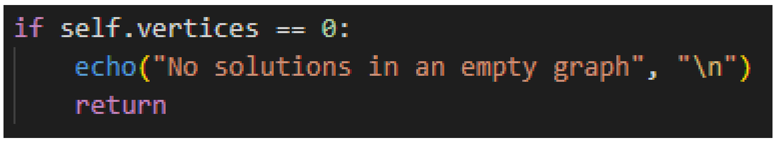


```
for v in 0 .. self.vertices-1:
    if isAdjacent(self, v, position, path) == true:
        path[position] = v

        #Recursive function to call itself until the path is filled
        #out and all cycles are found
        if self.hamCycleChecks(path, position+1) == true:
            return true

        #removes current node if it does not result in a solution
        path[position] = -1
return false
```

- The for-loop runs in linear time O(n)
- The first if statement runs the path validity function which is shown to be at minimum O(n)
- The second if statement is a recursive function that calls upon the hamiltonian cycle utilities function until the path is fully filled out, this means that it is O((n-1)!) since the first vertex in the path is already filled out in the hamiltonian cycle function

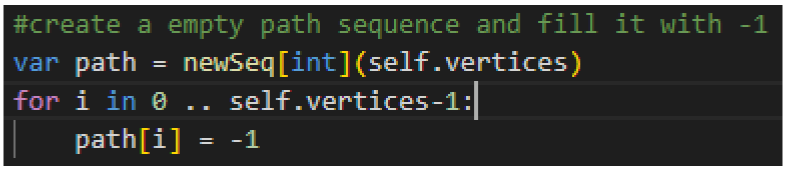
Therefore, the worst case time complexity for the hamiltonian cycle utilities function is **O((n-1)!)**.

Main hamiltonian cycle function:



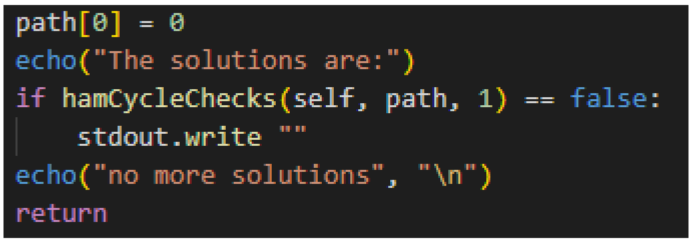
```
if self.vertices == 0:
    echo("No solutions in an empty graph", "\n")
    return
```

- If statements run in constant time O(1)



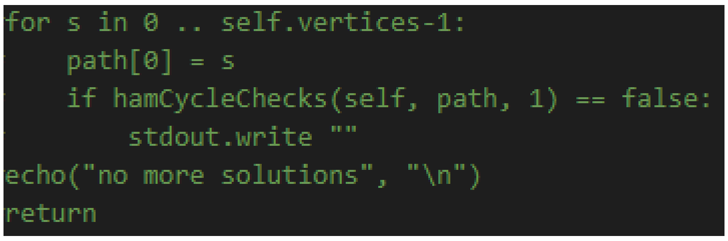
```
#create a empty path sequence and fill it with -1
var path = newSeq[int](self.vertices)
for i in 0 .. self.vertices-1:
    path[i] = -1
```

- Creating the path sequence should be O(n)
- The for loop is linear time, O(n), to traverse the amount of vertices
- Replacing the data inside the sequence is O(1)



```
path[0] = 0
echo("The solutions are:")
if hamCycleChecks(self, path, 1) == false:
    stdout.write ""
echo("no more solutions", "\n")
return
```

- Replacing the data inside the sequence is O(1)
- The hamcycleCheck (hamiltonian cycle utilities function) is shown to be O((n-1)!)



```
for s in 0 .. self.vertices-1:
    path[0] = s
    if hamCycleChecks(self, path, 1) == false:
        stdout.write ""
echo("no more solutions", "\n")
return
```

- This code is an optional piece of code that checks all the permutations of the hamiltonian cycle function and replaces the previous section.
- As there is a for-loop traversing the amount of vertices as well as the hamCycleCheck (hamiltonian cycle utilities function) which was shown to be O((n-1)!), this means that this section of code results in O(n(n-1)!) which can be simplified to O(n!)

Therefore, the worst case time complexity for the main hamiltonian cycle function is **O((n-1)!)** for unique solutions (since starting from another vertex results in a permutation of the solution) and **O(n!)** for all permutations of the cycle.

Matrix:
The matrix object was created so that the data is stored in a single array and is therefore at most **O(n)** for all the functions required (initialising the matrix and accessing the i-th element)

This means that the worst-case time complexity for the implementation is **O((n-1)!)** or **O(n!)** depending on whether the “all permutations” section of code is used

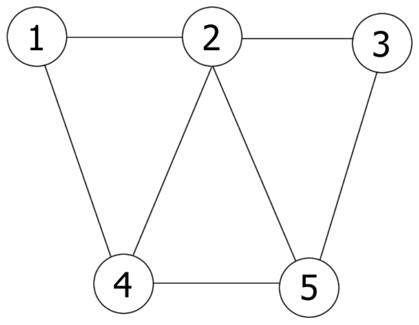
The **space complexity** of the implementation is

Still either O(1) or O(n) depending on whether the use of a new path array/sequence is counted, not sure

An explanation and justification of the effectiveness of the tests provided in assuring algorithmic correctness.

testMatrix1

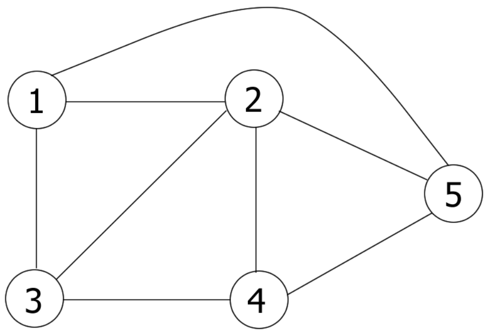
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0



This test involves a 5-vertex graph with indirect connections between vertices. It effectively tests the algorithms by providing multiple paths that result in a hamiltonian path, but not a hamiltonian cycle (i.e., 1, 2, 4, 5, 3). By including such a graph, we test whether or not the algorithm can find the paths that satisfies the requirement of both including each path only once as well as the last vertex being adjacent to the first vertex.

testMatrix2

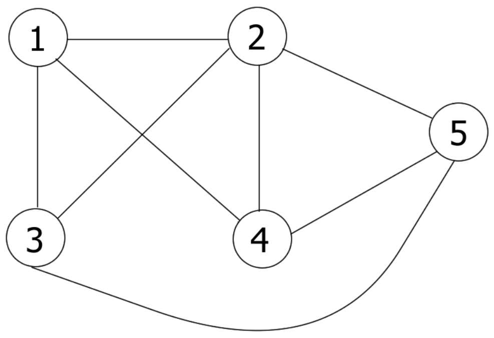
	1	2	3	4	5
1	0	1	1	0	1
2	1	0	1	1	1
3	1	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0



This test was included to showcase whether or not the algorithm successfully found the solution identified in the backtracking demonstration outlined earlier (pg 4 - 10).

testMatrix3

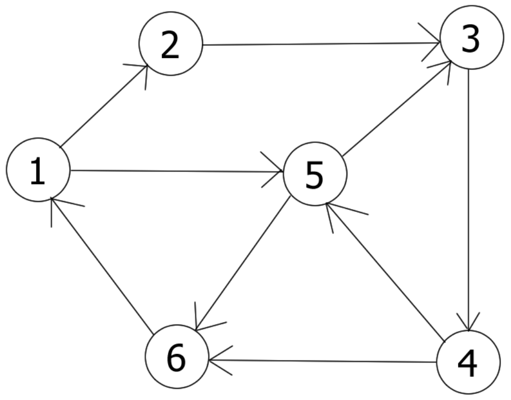
	1	2	3	4	5
1	0	1	1	1	0
2	1	0	1	1	1
3	1	1	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0



Similar to the previous test, this graph was included to identify whether or not the algorithm found a solution similar to what was outlined earlier in the report (pg 8).

testDirectedMatrix

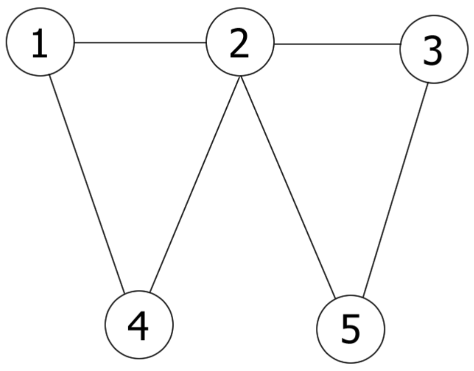
	1	2	3	4	5	6
1	0	1	0	0	1	0
2	0	0	1	0	0	0
3	0	0	0	1	0	0
4	0	0	0	0	1	1
5	0	0	1	0	0	1
6	1	0	0	0	0	0



This test involves a 6-vertex graph with directed connection between vertices. It aims to identify whether the algorithm successfully accounts for directed graphs as well as identifying the single hamiltonian path amongst multiple paths with “dead-ends”.

testNoCycle

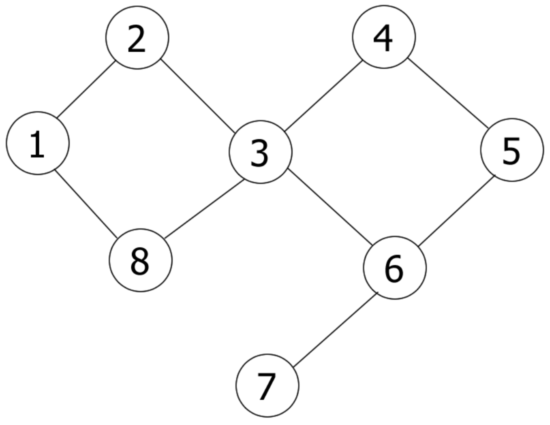
	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	1	1
3	0	1	0	0	1
4	1	1	0	0	0
5	0	1	1	0	0



This test involves a 5-vertex graph with indirect connections between vertices. It showcases a clear “articulation point” at node 2 in which the path requires a minimum of 2 visits to node 2 in order to find a path that would result in a cycle. However, as stated earlier, a hamiltonian cycle requires for the path to visit each vertex exactly once, testing whether or not the algorithm will reject the path found.

testNoCycle2

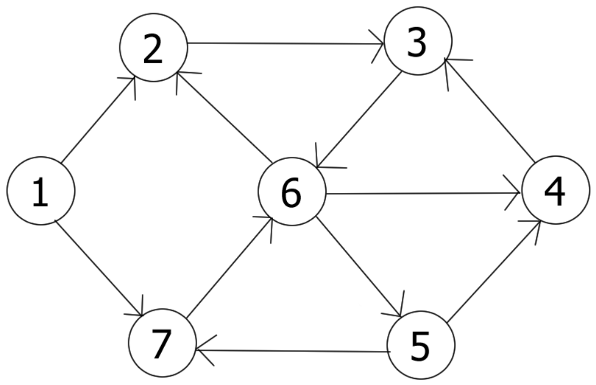
	1	2	3	4	5	6	7	8
1	0	1	0	0	0	0	0	1
2	1	0	1	0	0	0	0	0
3	0	1	0	1	0	1	0	1
4	0	0	1	0	1	0	0	0
5	0	0	0	1	0	1	0	0
6	0	0	1	0	0	0	1	0
7	0	0	0	0	0	1	0	0
8	1	0	1	0	0	0	0	0



This test involves a 8-vertex graph with indirect connections between vertices. It includes a clear “pendant” vertex, meaning a vertex with a degree of 1, at point 7 as well as an “articulation point” at node 3. This tests whether or not the algorithm can catch that there is no possible solution due to the requirement of only visiting each vertex exactly once and requiring a full cycle across all nodes.

testDirectedNoCycle

	1	2	3	4	5	6	7
1	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0
3	0	0	0	0	0	1	0
4	0	0	1	0	0	0	0
5	0	0	0	1	0	0	1
6	0	1	0	1	1	0	0
7	0	0	0	0	0	1	0



This test involves a 7-vertex graph with directed connections between vertices. It includes several paths, however, does not include any hamiltonian cycles or paths, resulting in no solutions, thereby confirming algorithm accuracy through outputting “no solutions”.

testEmptyMatrix

This test involves inputting an empty adjacency matrix into the algorithm to see whether or not the code can account for errors that may occur due no input.

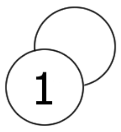
testInvalidMatrix

	1	2	3
1	3	7	5
2	1	1	1
3	1	1	1

This test involves inputting a matrix that contains numbers that are not 0s and 1s, therefore not being an adjacency matrix. If the algorithm successfully runs, that means that any output/solution is invalid as the algorithm is not supposed to run on non-adjacency matrices.

testSelfLoop

	1
1	1



This test involves inputting a graph that only contains 1 node which loops to itself. Therefore, any solution as a result of this does not pass as a hamiltonian cycle as it does not fulfil the requirement of only visiting each node once.

Any insights or observations that would help someone else solving the same problem.

Brute Force: Backtracking Depth First Search

It is important to check the adjacency of each new vertex with the previous vertex to ensure that the path is valid.
Lol idk

check that the last vertex is adjacent to the first vertex as that is a requirement of a hamiltonian cycle rather than a path.

Ensure that each vertex is only in the path once

References

GeeksForGeeks. (2022, October). *What is backtracking?*. Backtracking Algorithms.
https://www.geeksforgeeks.org/backtracking-algorithms/

Bari, A. (2018, April). *6.4 Hamiltonian Cycle - Backtracking.* Algorithms.
https://www.youtube.com/watch?v=dQr4wZCiJJ4&ab_channel=AbdulBari

Programiz. (Accessed 2022, November). *Adjacency Matrix*. Graph based DSA.
https://www.programiz.com/dsa/graph-adjacency-matrix

Ramandeep8421. (2022, June) *Count all Hamiltonian paths in a given directed graph*. GeeksForGeeks.
https://www.geeksforgeeks.org/count-all-hamiltonian-paths-in-a-given-directed-graph/

Msharma04. (2021, October) *Print all Hamiltonian Cycles in an Undirected Graph*. GeeksForGeeks.
https://www.geeksforgeeks.org/print-all-hamiltonian-cycles-in-an-undirected-graph/?ref=rp

GeeksForGeeks.(2022, January) *Hamiltonian Cycle | Backtracking-6*. Backtracking Algorithms.
https://www.geeksforgeeks.org/hamiltonian-cycle-backtracking-6/