Determining Cycles in a Graph

Using DFS and BFS approaches

Data Structure

- Array of linked lists representing nodes and their adjacency lists
- Implemented as a Python dictionary
  - Node weight is the key
  - Adjacency list is the value

Depth-First Search Approach

Input size: number of vertices |V| + number of pointers in each adjacency list |E|

Basic operation: comparison

Strategy

- For each node in the graph, traverse its adjacency list and keep track of visited nodes.
- For each node in the adjacency list, visit that node in the graph and traverse that node's adjacency list
- Repeat this process until a node has been visited twice, indicating a cycle.
- Using the location of the first occurrence of the repeated item in the visited list, extract the cycle from the visited list.
- Move to the next node in the current adjacency list or backtrack to the previous list if there are no nodes left in the current list.
- Output unique cycles.

DFS Pseudocode

```
function dfsHelper(node, graph, visited[0..n-1], locations):

// inputs:  a node in the adjacency list,  the entire graph, a list of nodes previously visited, a
dictionary of items in visited with their indicies as values

  while the node exists:

    value <- node value

    if node has not been visited:

      add value to the visited list

      k <- value

      v <- length(visited) - 1

      add an item to the locations dictionary with key k and value v

      dfsHelper(graph[value], graph, visited, locations)

      remove value from visited

      remove value from locations

    else:

      i <- locations[value]

      cycle <- visited[i..n-1]

      add cycle to the global cycles list

  node <- node.next
```

```
cycles <- empty set // global

def dfsApproach(graph):

  for each node in the graph:

    adjList <- the node's adjacency list

    visited <- []

    locations <- empty set

    v <- node value

    add v to visited

    add v to the dictionary with value 0

    dfsHelper(node, graph, visited, locations)
```

DFS Pseudocode

```
function checkCycles():
   // filters the global list of found cycles to ensure distinct cycles
   // output: the list of distinct cycles
   output <- []
   for each cycle in the global cycles list:
      c <- canonical_cycle(cycle)
      if c not in output, add c to the output
```

```
function canonical_cycle(cycle[0..n-1]):
   // determines the smallest canonical rotation of a list
   // input: a list of characters that represents a cycle in the graph
   // output: the smallest lexicographic list in rotationsList
   t <- tuple(cycle)
   rotationsList <- [] // list of lists
   i <- 0
   l <- length(t)

   while i < l:
      rotatedList <- rotate t once to the left
      add rotated list to rotationsList

   return the smallest lexicographic list in rotationsList
```

DFS Implementation

```python
cycles: set[tuple[str]] = set()

def canonical_cycle(cycle):
    '''...

    t = tuple(cycle)
    rotations = [t[i:] + t[:i] for i in range(len(t))]
    return min(rotations)


def checkCycles():
    '''...

    output:set[list[str]] = set()

    for cycle in cycles:
        cycle = canonical_cycle(cycle)
        if cycle not in output:
            output.add(cycle)

    return output
```

```python
def dfsHelper(node: ListNode, adjList: AdjacencyList, visited:list[str], locations:dict[str, int]) -> None:
    '''...

    while node is not None:

        value = node.val
        if value not in set(visited):
            visited.append(value)
            locations[value] = len(visited) - 1
            dfsHelper(adjList[value].head, adjList, visited, locations)
            visited.pop()
            locations.pop(value)

        else:
            valIdx = locations[value]
            cycle = visited[valIdx:]
            cycles.add(tuple(cycle))

        node = node.next
```

```python
def dfsApproach(adjList: AdjacencyList):
    '''...

    for key in adjList.lst.keys():
        list = adjList.lst[key]
        visited = [] # track visited nodes in current iteration
        locations = {}
        visited.append(key)
        locations[key] = 0
        dfsHelper(list.head, adjList, visited, locations)

    print(checkCycles())
```

DFS Analysis

Function	Input Size	Basic Operation	Order of Growth
Canonical_cycle(cycle[0...n-1])	Size of cycle	Comparison	$\Theta\ (n^2)$
CheckCycles()	Number of Cycles	Comparison	$\Theta\ (n^2)$
DfsHelper(node,graph,visited[0..n-1],locations)	Number of vertices	Comparison (Process one adjacency-list entry)	Best Case: $\Theta(1)$ Worst Case: $\Theta\ (n^n)$
DfsApproach(graph)	Number of vertices	Comparison (Process one adjacency-list entry)	$\Theta\ (n)$

Brute-Force

Strategy

- Generate all permutations and partial permutations (ordered selections of elements where not all elements are necessarily used)
    - For example {A, B, C} would be checked, as well as just {A, C} and {A, C, B} and every other partial permutation
- Check if each node links to the next and if the final node links to the first
- If this is true, the permutation/partial permutation is a cycle and should be added to the output

Pseudocode

```
brute_force_algo(adj: AdjacencyList)
   (global) cycles <- (empty set)
   testPermutations(adj, [ ], adj.nodes())
   return cycles


check_cycle(adj: AdjacencyList, path: list(Node))
   if path.length = 0
      return false
   else if path.length = 1
      return path[0].linksto(path[0])
   else
   for n in (0 to path.length - 2)
      if not path[n].linksto(path[n+1])
         return false
   if path[path.length-1].linksto(path[0])
      return true
   else
      return false
```

```
test(adj: AdjacencyList, path: list(Node))
   if check_cycle(adj, path)
   # unique will be a function that ensures all equivalent cycles are represented the same way
   # ensuring no duplicates
      cycles.add(unique(path))


testPermutations(adj: AdjacencyList, curPath: list(Node), remainingVertices: set(Node))
   test(adj, curPath)
   for n in remainingVertices:
      newRemaining <- remainingVertices
      newRemaining.add(n)
      newPath <- curPath
      newPath.append(n)
      testPermutations(adj, newPath, newRemaining)
```

Brute Force Analysis

Function	**Input Size**	**Basic Operation**	**Order of Growth**
combinationHelper()	Number of vertices (n)	Checks all permutations & calls test()	$\Theta(n! \times n^2)$
test(current path)	Length of path (m)	Calls checkCyles(current path) n! amount of times	$\Theta(m \times n)$ Worst case: $\Theta(n^2)$
checkCycles(current path)	Length of path (m)	Calls searchLinkedList() m amount of times	$\Theta(m \times n)$ Worst case: $\Theta(n^2)$
searchLinkedList()	Size of adjacency list (<= n)	Iterate through every node in adjacency list	$\Theta(n)$

Implementation

```python
from data_types import *
from dfsApproach import canonical_cycle

cycles: set[tuple[str]] = set()

def bruteForceAlgo(adj: AdjacencyList):
    permutationHelper(adj, [], [i for i in list(adj.lst.keys())])
    return cycles

# returns true if val is in linked
def searchLinkedList(linked: LinkedList, val: str):
    cur = linked.head
    while cur != None:
        if cur.val == val:
            return True
        cur = cur.next
    return False
```

```python
# checks if the vertices in the path all connect, and if the last vertex connects to the start
def checkCycle(adj, path):
    if len(path) == 0:
        return False

    elif len(path) == 1:
        if searchLinkedList(adj[path[0]], path[0]):
            return True
        else:
            # print(f"{path[0]} does not point to itself ({path[0]} -> {adj[path[0]]})")
            return False
        # checking vertices in path
    for i in range(len(path) - 1):
        curItem = path[i]
        curItemReachable = adj[curItem]
        nextItem = path[i + 1]
        if not searchLinkedList(curItemReachable, nextItem):
            # print(f"{nextItem} is not reachable from {curItem} ({curItemReachable})")
            return False

    # checking that last item connects to first item
    lastItem = path[-1]
    lastItemReachable = adj[lastItem]
    firstItem = path[0]
    if (searchLinkedList(lastItemReachable, firstItem)):
        return True
    else:
        # print(f"{firstItem} is not reachable from {lastItem} ({lastItem} -> {lastItemReachable})")
        return False
```

Implementation

```
# calls test() on every path starting with curPath, using some permutation of the vertices in remainingVertices
def permutationHelper(adj: AdjacencyList, curPath: list(str), remainingVertices: set(str)):
    test(adj, curPath)

    for v in remainingVertices:
        newRemaining = set(remainingVertices)
        newRemaining.remove(v)
        newCurPath = [i for i in curPath]
        newCurPath.append(v)
        permutationHelper(adj, newCurPath, newRemaining)
```

```
# checks if the path is a cycle, adds it to cycles if so
def test(adj, path):
    if checkCycle(adj, path):
        # print(f"Cycle Found: {path}")
        cycles.add(tuple(canonical_cycle(path)))
    else:
        # print(f"Not a cycle: {path}")
        return
```

Comparison

- Adjacency List:
- A -> B -> C -> E -> F -> None
- B -> C -> D -> J -> None
- C -> E -> F -> G -> A -> None
- D -> A -> C -> E -> E -> J -> B -> None
- E -> C -> H -> None
- F -> I -> I -> None
- G -> H -> G -> None
- H -> H -> F -> I -> C -> None
- I -> C -> F -> F -> None
- J -> B -> None

- running DFS algorithm…
- {('C', 'G', 'H'), ('B', 'D', 'J'), ('H',), ('C', 'E'), ('F', 'I'), ('A', 'B', 'C'), ('C', 'E', 'H', 'I'), ('A', 'B', 'D', 'E', 'C'), ('C', 'G', 'H', 'I'), ('C', 'E', 'H'), ('G',), ('A', 'E', 'H', 'F', 'I', 'C'), ('B', 'J'), ('A', 'B', 'D', 'C'), ('A', 'E', 'H', 'C'), ('A', 'F', 'I', 'C'), ('C', 'G', 'H', 'F', 'I'), ('C', 'F', 'I'), ('A', 'B', 'D', 'E', 'H', 'C'), ('A', 'B', 'D'), ('C', 'E', 'H', 'F', 'I'), ('B', 'D'), ('A', 'C'), ('A', 'B', 'D', 'E', 'H', 'F', 'I', 'C'), ('A', 'E', 'H', 'I', 'C'), ('A', 'B', 'D', 'E', 'H', 'I', 'C'), ('A', 'E', 'C')}
- DFS algorithm complete, time taken: 0.0017981529235839844s
- running brute force algorithm…
- {('B', 'D', 'J'), ('C', 'G', 'H'), ('H',), ('C', 'E'), ('F', 'I'), ('A', 'B', 'C'), ('C', 'E', 'H', 'I'), ('A', 'B', 'D', 'E', 'C'), ('C', 'G', 'H', 'I'), ('C', 'E', 'H'), ('G',), ('A', 'E', 'H', 'F', 'I', 'C'), ('B', 'J'), ('A', 'B', 'D', 'C'), ('A', 'E', 'H', 'C'), ('A', 'F', 'I', 'C'), ('C', 'G', 'H', 'F', 'I'), ('C', 'F', 'I'), ('A', 'B', 'D', 'E', 'H', 'C'), ('A', 'B', 'D'), ('C', 'E', 'H', 'F', 'I'), ('A', 'C'), ('B', 'D'), ('A', 'B', 'D', 'E', 'H', 'F', 'I', 'C'), ('A', 'E', 'H', 'I', 'C'), ('A', 'B', 'D', 'E', 'H', 'I', 'C'), ('A', 'E', 'C')}
- brute force algorithm complete, time taken: 9.27213978767395s

Brute force time taken: 9.27s
DFS time taken: 0.000180s
DFS is about 50,000x faster!
Only 10 List elements

Conclusion

- Brute-force may be simpler to understand, but it is wildly inefficient compared to the DFS algorithm
  - This is shown both in the asymptotic analysis and in the testing
- For this reason, the DFS approach is highly preferred, as the brute force algorithm scales factorially with input size (* n^2 as well!)
  - This is wildly inefficient, as we saw for an input size of only 10, the algorithm took 10 seconds to complete
  - We can guess based on this, that a graph with 20 vertices and a similar proportion of connections could take approximately 2.68 * 10^12 seconds to complete (10s * 20! / 10! * 2^2)
  - In contrast, we would guess that the DFS approach would take about 0.0036s to complete for n = 20 (0.0018s * 2)

Students in Presentation

- Jacob Croket – DFS pseudocode and implementation
- Krutika Patil – DFS analysis
- Dakota DeGolyer – Brute Force Code
- Ariq Chowdhury – Brute Force Analysis
- Philo Salama – Brute Force Pseudocode