UNIVERSITY OF ARKANSAS

CSCE 4133/5133 Algorithms

Homework Assignment #2

Submission Deadline: Mid-Night (11:59PM) February 19, 2026

Instructions

- **Submissions:** Students can use either Google Docs or LaTeX for writing the report. The report template is provided in the Blackboard. Your submission should be a zip file containing your report and the source code, including your implementation. Name the zip file as `lastname_studentID.zip`. Submissions should be made via BlackBoard.
- **Policy:** Students must start working on this assignment independently and review the late-day policy.
- If you have any questions please contact us via Teams or email: kimtran@uark.edu and maalexra@uark.edu.

I. Overview

In homework 1, we have studied the linked list and queue implementation. The linked list is used to build the adjacency matrix of the graph. Meanwhile, the queue is used to implement the graph searching algorithm, i.e., breadth first search. In this homework 2, we are going to implement two common graph searching algorithms studied in the class lecture, i.e. breadth first search (BFS) and depth first search (DFS). In this homework, students are **REQUIRED** to implement these algorithms in Python3. The students are only needed to write their code implementation following the instructions and requirements. In this homework, we are going to implement BFS and DFS to find a path from a given building to another building. The map is represented as a graph structure in which each node represents one building (or a landmark), and an undirected edge between two nodes represents the path between buildings/landmarks. A list of edges in a graph is represented by an adjacency list implemented using a linked list. The BFS algorithm is implemented using a queue and the DFS is implemented by either a stack or a recursive function. You are provided with the implementation of linked list, queue, and stack in this homework. In this homework, you are **REQUIRED** to implement **BFS** and **DFS** algorithms. If you think the homework is too long, don't worry, it is long because you are provided the details of the descriptions. There are some hints that may help to succeed in this homework:

- You should start the homework early.
- You should read the descriptions of homework carefully before you start working on the homework.
- Before implementing the required functions, you should try to compile the source code first.Basically, the source code with an empty implementation can be compiled successfully. This step will help you to understand the procedure of each problem.
- If you forget the BFS and DFS algorithms, you are encouraged to revise them in the lecture slides

The source code is organized as follows:

File or Folder	**Required Implementation**	**Purpose**
assets/	No	Contains the data of homework
src/	No	Contains source code files
bin/	No	Contains the results
`src/linked_list.py`	No	Implements the linked list structure
src/queue.py	No	Implements the queue structure
src/stack.py	No	Implements the stack structure
src/graph.py	No	Implements the graph structure and algorithms
`src/bfs.py`	Yes	Implements the BFS algorithm
`src/dfs.py`	Yes	Implements the DFS algorithm
`src/rdfs.py`	Yes	Implements the Recursive DFS algorithm
src/main.py	No	Implements the main program

II. Implementation

II..1 Queue, Stack, and Graph Usages

Before we start implementing BFS and DFS, we first investigate the usages of **Queue, Stack, and Graph**. These classes have been already implemented in the **src/queue.cpp, src/stack.cpp, and src/graph.cpp**

Class	**Method**	**Meaning**
Queue	Queue	Initialize an empty queue
	pop	Pop out the head value of the queue
	empty	Check the current queue is either empty or not (return true if it's empty)
Stack	Stack	Initialize an empty stack
	push	Push a new value into the head of the stack
	pop	Pop out the head value of the stack
	empty	Check the current stack is either empty or not (return true if it's empty)
Graph	Graph	Initialize a graph with n vertices
	insert_edge	Insert an undirected (or directed) edge into the graph
	is_visited	Return true if the current vertex was already visited by the searching algorithm
	set_visited	Mask a vertex as visited vertex
	trace	Return the previous vertex of the given vertex in the searching order (return -1 if the vertex is the starting vertex in the searching order)
	set_trace	Set the previous vertex of the current vertex in the searching order
	search	Perform the search algorithm to find a path from start to destination

II..2 Breadth First Search

In this section, you are required to implement the BFS algorithm by the queue. You have to implement BFS in **src/bfs.py**

```python
def bfs(g: Graph, start: int, destination: int) -> None:
    queue: Queue[int] = Queue()
    g.reset()
    # YOUR CODE HERE
    while not queue.empty():
        u = queue.pop()
        # YOUR CODE HERE
        number_of_adjacency_nodes = g.e[u].size()
        p = g.e[u].get_root()
        for _ in range(number_of_adjacency_nodes):
            v = p.value
            # YOUR CODE HERE
            p = p.next
```

Given a graph G, a start vertex, and a destination vertex, you have to implement the BFS algorithm so that starting from the given start vertex, the BFS will search for the destination vertex. During the searching process, you have to use G.set_trace to store the previous vertex of all vertices visited by BFS. After this, based on the tracing array, in the file **src/graph.py**, we can trace back the path from the starting vertex to the destination vertex as follows:

```python
def search(self, start: int, destination: int, searchfn: Callable[["Graph", int, int], None
    ]) -> List[int]:
    searchfn(self, start, destination)

    path: List[int] = []
    u = destination
    while u != -1:
        path.append(u)
        u = self.trace(u)

    path.reverse()
    return path
```

II..3 Depth First Search

In this section, you are required to implement the DFS algorithm using the stack. You have to implement DFS in **src/dfs.py**

```python
def dfs(g: Graph, start: int, destination: int) -> None:
    stack: Stack[int] = Stack()
    g.reset()
    # YOUR CODE HERE
    while not stack.empty():
        u = stack.pop()
        # YOUR CODE HERE
        number_of_adjacency_nodes = g.e[u].size()
        p = g.e[u].get_root()
        for _ in range(number_of_adjacency_nodes):
            v = p.value
            # YOUR CODE HERE
            p = p.next
```

Similar to BFS, given a graph G, a start vertex, and a destination vertex, you have to implement the DFS algorithm so that starting from the given start vertex, the DFS will search for the destination vertex. During the searching process, you have to use G.set_trace to set the previous vertex of all vertices visited by DFS.

II..4 Recursive Depth First Search

In this section, you are required to implement the Recursive DFS algorithm. You have to implement Recursive DFS in **src/rdfs.py**

```python
def rdfs(g: Graph, start: int, destination: int) -> None:
    # YOUR CODE HERE
    number_of_adjacency_nodes = g.e[start].size()
    p = g.e[start].get_root()
    for _ in range(number_of_adjacency_nodes):
        v = p.value
        # YOUR CODE HERE
        p = p.next
```

Similar to BFS and DFS, given a graph G, a start vertex, and a destination vertex, you have to implement the Recursive DFS algorithm so that starting from the given start vertex, the Recursive DFS will search to the destination vertex. During the searching process, you have to use G.set_trace to set the previous vertex of all vertices visited by Recursive DFS.

II..5 Main Program

The main program is fully implemented in **src/main.py**. This program contains the implementation of the unit test of your search implementation. This program also contains the visualization of the search results on the campus map. The program takes an argument that is the name of the algorithm used to perform the search.

python3 src/main.py bfs → Perform BFS

python3 src/main.py dfs → Perform DFS

python3 src/main.py rdfs → Perform Recursive DFS

III. Compilation and Testing

In this homework 2, we use the same environment (Python3, Make, OpenCV Library) used in homework 1 to compile the source code. For convenience, instead of typing a long command like **python3 src/main.py bfs**, you can use a Makefile and simply type:

- **make bfs**: to perform BFS
- **make dfs**: to perform DFS
- **make rdfs** : to perform Recursive DFS

If you run successfully and your implementation is correct, you should pass all the unit tests designed in the main program with the correct path as follow:

Class	Method
BFS	Perform unit test on your bfs implementation Path from 0 to 5 by bfs: 0 1 3 4 5 Path from JBHT to destination: JBHT - HILL - WJWH - HAPG
DFS	Perform unit test on your dfs implementation Path from 0 to 5 by dfs: 0 1 2 4 5 Path from JBHT to destination: JBHT - HILL - WJWH - HAPG
Recursive DFS	Perform unit test on your rdfs implementation Path from 0 to 5 by rdfs: 0 1 3 4 5 Path from JBHT to destination: JBHT - HILL - WJWH - HAPG

IV. Operation Counting

To practice computing time complexity, in this section, we will practice counting the number of operators (+, -, *, /, >, <, assign, swap...) in an algorithm. Please write your answer in the report.

Algorithm 1: Count the number of operators (+, *) in the algorithm below (no need to compute number of assignments):

```python
sum = 0
for i in range (10):
    for j in range (100):
        sum = sum + i * j + j
```

Algorithm 2: Count the maximum and minimum possible number of assignments in the algorithm below:

```python
def bubble_sort(arr, n: int):
    for i in range(n):
        for j in range(0, n - 1):
            if arr[j] > arr[j + 1]:
                temp = arr[j]
                arr[j] = arr[j + 1]
                arr[j + 1] = temp
```

Algorithm 3: Count the number of **comparisons (>, <, <=,...)** in the algorithm below:

```
// binary_search
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 7

left = 0
right = len(arr) - 1
is_found = False
while left <= right:
    mid = (left + right) // 2
    if arr[mid] == target:
        is_found = True
        break
    elif arr[mid] < target:
        left = mid + 1
    else:
        right = mid - 1
```

V. Submission

Write a report detailing the key ideas for implementing each required function and include the implemented functions themselves. You must also include a screenshot of the final output (from either the command prompt or terminal) after running the program.

Write your full name, email address, and student ID in the report (report is worth 20 points). Answer the above questions in the operation-counting section with explanations (4 points each) in your report.

Your submission should be a zip file containing your report and **the entire source code (NOT just the files you changed)**, including your implementation. Name the zip file as `lastname_studentID.zip`. Submission should be made via BlackBoard.