Project2

Name: Zian Wang

(The performance and the memory requirements are at the end of this file.)

Program1: Dijkstra with two-dimension array.

This program has 7 primary functions:

int[][] readArray(String path, int num):

Input: the path of the input file, the number of the nodes in the graph.

Output: A two dimension array which stores the adjacency matrix.

Function: Read the input file and make the data inside a adjacency matrix, and return it.

Map Dijkstra(int n, int[][] W, int startNode):

Input: the number of the nodes in the graph: n, the adjacency matrix, the index of the start node.

Output: a Map that contains two objects: the set of the edge: F, and the touch array.

Function: apply the Dijkstra algorithm, read the adjacency matrix, return F and touch array.

int[] getPath(int[] touch, int numOfNodes, int startNode, int endNode, int[][] adjacency)

Input: the touch array, the number of the nodes in the graph, the index of start node and end node, the adjacency matrix.

Output: an array that stores the path from start node to end node.

Function: get the path from assigned start node and end node.

int[][] completeAdjacency(int[][] adjacency, int numOfNodes)

Input: the adjacency matrix, the number of the nodes.

Output: the completed adjacency matrix.

Function: the adjacency matrix which is returned from readArray() is not complete. The diagonal elements are not 0, and the weight between two points which are not connected is not infinity. So this function is used to make diagonal elements 0 and the infinite length infinity.

int[][] resetAdjacency(int[][] adjacency, int numOfNodes)

Input: the adjacency matrix, the number of the nodes.

Output: a copy of the adjacency matrix.

Function: in Java, input an array into a function is inputting its location, so when we operate the array in the function, it will be change inside and outside the function. Therefore, if we simply input the same adjacency matrix into the Dijkstra function every time, it will be changed, so every time(except the first time) we input the matrix, actually it has already been changed. So we use this function to create a exactly same matrix to input into the function.

int getNumOfNodes(String path)
Input: the path of the input file.
Output: the number of the nodes in the graph.
Function: to get the number of the nodes.

void run(String filePath, int numOfNodes)
Input: the path of the input file, the number of the nodes.
Output: none.
Function: used to call other functions in this program in a proper order and get the results we want.

The core function is the Dijkstra function. The other functions are used to read files, make data an adjacency matrix or some other support functions.

The Dijkstra() function is the function that applies the Dijkstra algorithm. This function is almost as same as the function in the text book. However, in the text book, the Dijkstra function can only treat 0 node as the start node. Therefore, I add few steps to make the function can treat every node as the start node. When we call the Dijkstra function and specify **x** as the start node, the function will exchange positions of 0 and **x** in the adjacency matrix. We will do this by swapping 0th and **xth** row, then swap 0th and **xth** column. After this operation, the **x** node is swapped to the position of 0 node, so the Dijkstra function will take **x** node as start node. In other words, we will call **x** node 0, and call 0 node **x** now.

This operation will also make some trouble in the result, which is all the 0 in the results(touch array and F set) are actually **x**, and all the **x** in the results are actually 0 because we exchange their names before. Therefore, at the end of the function, we will exchange 0 and **x** back in the touch array and F set.

We will then use touch array to find the path. If we want to find, for example, the path from start node to **y** node, we will first find touch[y]. Touch[y] is the penultimate node on the path from start to **y.** Therefore, we will then look for touch[touch[y]], and touch[touch[y]] is the third point from the bottom on the path. We will loop this process until we got the start node from the touch array. All the nodes we get in order in this process will be stored in an array and we will then reverse the order of the nodes in the array and put the length of the path in the latest element of the array. Therefore, the path and the length are stored in the array now.

Test cases:

```
Dijkstra with two-dimension array:
The shortest path from 65 to 280 is:
65-> 216-> 116-> 117-> 201-> 274-> 326-> 24-> 23-> 125-> 140-> 203-> 167-> 197-> 192-> 280
The length of the path is: 4923

The shortest path from 187 to 68 is:
187-> 238-> 229-> 231-> 264-> 247-> 17-> 18-> 242-> 158-> 77-> 78-> 136-> 137-> 332-> 70-> 134-> 176-> 269-> 286-> 300-> 318-> 290-> 302-> 323-> 277-> 175-> 68
The length of the path is: 11199

The shortest path from 197 to 27 is:
197-> 198-> 303-> 293-> 142-> 26-> 27
The length of the path is: 3009


Process finished with exit code 0
```

The order of the test cases below is little different from the .pdf file because I use a for loop to take all the nodes as the start node to input into Dijkstra function, and the for loop is from 0 to the end. Therefore, the for loop will input 65 as start node first, and calculate the shortest paths from 65 to all the other nodes, including the shortest path from 65 to 280. Then, the for loop will input 187 into Dijkstra function, so we then got the shortest path from 187 to 68. Lastly, the for loop will input 197 so we got the shortest path from 197 to 27 at last.

Program2: Dijkstra with link list.

This program has 7 primary functions:

link_list[] readLinkList(String path, int num):

Input: the path of the input file, the number of the nodes in the graph.

Output: An array of link list which stores the adjacency matrix.

Function: Read the input file and make the data inside the link lists, and return it.

int getLength(int startNode, int target, link_list[] V)

Input: the index of the start node, the index of the end node, the link list V which stores the graph.

Output: the length between the start node and the end node.

Function: get the length between the start node and the end node.

Map Dijkstra(int n, link_list[] V, int startNode):

Input: the number of the nodes in the graph: n, the link list which stores the graph, the index of the start node.

Output: a Map that contains two objects: the set of the edge: F, and the touch array.

Function: apply the Dijkstra algorithm, read the link lists, return F and touch array.

int[] getPath(int[] touch, int numOfNodes, int startNode, int endNode, link_list[] V)

Input: the touch array, the number of the nodes in the graph, the index of start node and end node, the link lists.

Output: an array that stores the path from start node to end node.

Function: get the path from assigned start node and end node.

link_list[] resetLinkList(link_list[] V, int numOfNodes)

Input: the array of the link lists, the number of the nodes.

Output: a copy of the array.

Function: because when we input the array into a function, if we change the array inside the function, it will be also changed outside the function, so we use this function to create a copy of the array of the link lists.

int getNumOfNodes(String path)

Input: the path of the input file.

Output: the number of the nodes in the graph.

Function: to get the number of the nodes.

void run(String filePath, int numOfNodes)

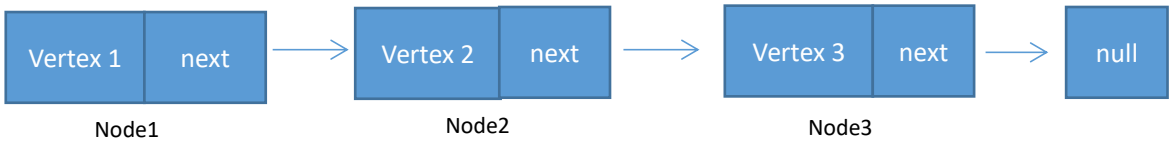
Input: the path of the input file, the number of the nodes.

Output: none.

Function: used to call other functions in this program in a proper order and get the results we want.

The link list is defined in a separate class: link_list. It is singly link list which has 3 members: 1. “vertex” stands for the vertex of the node, 2. “weight” stands for the weight of the path

between current node's vertex and the vertex of the next node, 3. "next" which points to the next node. The structure of the link_list is as the figure following.



This link list stands for there is path from 1 to 2, and its length is stored in the weight of the node 1. Also, there is a path from 1 to 3, and its length is stored in the weight of node 3.

We will store all the data in the file in an array of link list, V. V[x] stores all the path that starts from x node.

The Dijkstra() function is the function that applies the Dijkstra algorithm. The Dijkstra function in this program has the same steps in that in last program to exchange the 0 node and the start node. We will do this by exchanging the index of the link lists of the 0 node and the start node in the array of the link lists. So the function, which only can use 0 node as start node previously, can treat any node as start node. The rest parts of this function is as same as it in the text book.

Also, all other auxiliary functions are basically the same as they are in the first program, except all of them will operate array of link list instead of adjacency matrix.

Test cases:

```
Dijkstra with link list
The shortest path from 65 to 280 is:
65-> 216-> 116-> 117-> 201-> 274-> 326-> 24-> 23-> 125-> 140-> 203-> 167-> 197-> 192-> 280
The length of the path is: 4923

The shortest path from 187 to 68 is:
187-> 238-> 229-> 231-> 264-> 247-> 17-> 18-> 242-> 158-> 77-> 78-> 136-> 137-> 332-> 70-> 134-> 176-> 269-> 286-> 300-> 318-> 290-> 302-> 323-> 277-> 175-> 68
The length of the path is: 11199

The shortest path from 197 to 27 is:
197-> 198-> 303-> 293-> 142-> 26-> 27
The length of the path is: 3009

Process finished with exit code 0
```

The order of the test cases below is little different from the .pdf file because I use a for loop to take all the nodes as the start node to input into Dijkstra function, and the for loop is from 0 to the end. Therefore, the for loop will input 65 as start node first, and calculate the shortest paths from 65 to all the other nodes, including the shortest path from 65 to 280. Then, the for loop will input 187 into Dijkstra function, so we then got the shortest path from 187 to 68. Lastly, the for loop will input 197 so we got the shortest path from 197 to 27 at last.

Program3: Floyd with two-dimension array.

This program has 8 primary functions:

int getNumOfNodes(String path)

Input: the path of the .csv file.

Output: the number of the nodes in the file.

Function: get the number of the nodes in the file.

int[][] readArray(String path, int num)

Input: the path of the .csv file, the number of the nodes in the file.

Output: the adjacency matrix of the graph stored in the file.

Function: read the input files and output the adjacency matrix.

int[][] completeAdjacency(int[][] adjacency, int numOfNodes)

Input: the adjacency matrix, the number of the nodes.

Output: the completed adjacency matrix.

Function: the adjacency matrix which is returned from readArray() is not complete. The diagonal elements are not 0, and the weight between two points which are not connected is not infinity. So this function is used to make diagonal elements 0 and the infinite length infinity.

Map Floyd(int n, int[][] adjacency)

Input: the number of the nodes in the file, the adjacency matrix.

Output: the matrix that stores the length of the shortest path: D, the matrix that stores the intermediate vertexes: P.

Function: calculate the shortest path from all the nodes to all the other nodes in the graph, and its length.

void getPath(int[][] P, int startNode, int endNode)

Input: the matrix that stores the intermediate vertexes: P, the start node and the end node of the path.

Output: the shortest path from the start node to the end node.

Function: get the shortest path from the start node to the end node, except the start node and the end node. This function does not print the full path because I want to print "->" to the console, and if I only use this function, there will be an additional "->" at the end of the console output.

void calPath(int[][] P, int startNode, int endNode)

Input: the matrix that stores the intermediate vertexes: P, the start node and the end node of the path.

Output: none.

Function: this function is used to simulate the process that calculate the shortest path. This function is another version of getPath() function, it will calculate but it will not print anything to the console. I use this function to make sure that the running time of my program includes the time that calculating the shortest path.

void printPath(int[][] P, int startNode, int endNode)

Input: the matrix that stores the intermediate vertexes: P, the start node and the end node of the path.

Output: the shortest path from the start node to the end node.

Function: get the full shortest path from the start node to the end node. This function is used to print the full path which does not contain an additional "->" at the end.

void run(String filePath)
Input: the path of the input file.
Output: none.
Function: used to call other functions in this program in a proper order and get the results we want.

public void runTest(String filePath, int[] startTestNode, int[] endTestNode)
Input: the path of the input file, two arrays that store the start nodes and the end nodes that need to be tested(the test cases).
Output: print the shortest path from start nodes to end nodes.
Function: used to print the shortest paths of the test cases.

This program uses Floyd algorithm which is as same as the "Floyd Algorithm for Shortest Paths 2" in the text book. This program will additionally output a matrix that stores the immediate vertexes: P. We will use the same recursive call in the text book to find and print the shortest path in P.

There one point that is different from the Dijkstra algorithm: the Floyd algorithm calculates the shortest paths from all the nodes to all the other nodes in the graph, so the function does not need to specify the start node. Therefore, we do not need to change the position in the adjacency matrix.

Test cases:

```
Floyd with two-dimension array
The path from 197 to 27 is:
197-> 198-> 303-> 293-> 142-> 26-> 27
The length of the path is:3009

The path from 65 to 280 is:
65-> 216-> 116-> 117-> 201-> 274-> 326-> 24-> 23-> 125-> 140-> 203-> 167-> 197-> 192-> 280
The length of the path is:4923

The path from 187 to 68 is:
187-> 238-> 229-> 231-> 264-> 247-> 17-> 18-> 242-> 158-> 77-> 78-> 136-> 137-> 332-> 70-> 134-> 176-> 269-> 286-> 300-> 318-> 290-> 302-> 323-> 277-> 175-> 68
The length of the path is:11199


Process finished with exit code 0
```

Program4: Floyd with link list.

This program has 9 primary functions:

int getNumOfNodes(String path)

Input: the path of the .csv file.

Output: the number of the nodes in the file.

Function: get the number of the nodes in the file.

link_list[] readLinkList(String path, int numOfNodes)

Input: the path of the .csv file, the number of the nodes in the file.

Output: An array of link list which stores the adjacency matrix.

Function: Read the input file and make the data inside the link lists, and return it.

int getLength(int startNode, int target, link_list[] V)

Input: the index of the start node, the index of the end node, the link list V which stores the graph.

Output: the length between the start node and the end node.

Function: get the length between the start node and the end node.

Map Floyd(int n, link_list[] V)

Input: the number of the nodes in the file, the array of the link lists: V.

Output: the matrix that stores the length of the shortest path: D, the matrix that stores the intermediate vertexes: P.

Function: calculate the shortest path from all the nodes to all the other nodes in the graph, and its length.

void getPath(int[][] P, int startNode, int endNode)

Input: the matrix that stores the intermediate vertexes: P, the start node and the end node of the path.

Output: the shortest path from the start node to the end node.

Function: get the shortest path from the start node to the end node, except the start node and the end node. This function does not print the full path because I want to print "->" to the console, and if I only use this function, there will be an additional "->" at the end of the console output.

void calPath(int[][] P, int startNode, int endNode)

Input: the matrix that stores the intermediate vertexes: P, the start node and the end node of the path.

Output: none.

Function: this function is used to simulate the process that calculate the shortest path. This function is another version of getPath() function, it will calculate but it will not print anything to the console. I use this function to make sure that the running time of my program includes the time that calculating the shortest path.

void printPath(int[][] P, int startNode, int endNode)

Input: the matrix that stores the intermediate vertexes: P, the start node and the end node of the path.

Output: the shortest path from the start node to the end node.

Function: get the full shortest path from the start node to the end node. This function is used to print the full

path which does not contain an additional “->” at the end.

void run(String filePath)
Input: the path of the input file.
Output: none.
Function: used to call other functions in this program in a proper order and get the results we want.

public void runTest(String filePath, int[] startTestNode, int[] endTestNode)
Input: the path of the input file, two arrays that store the start nodes and the end nodes that need to be tested(the test cases).
Output: print the shortest path from start nodes to end nodes.
Function: used to print the shortest paths of the test cases.

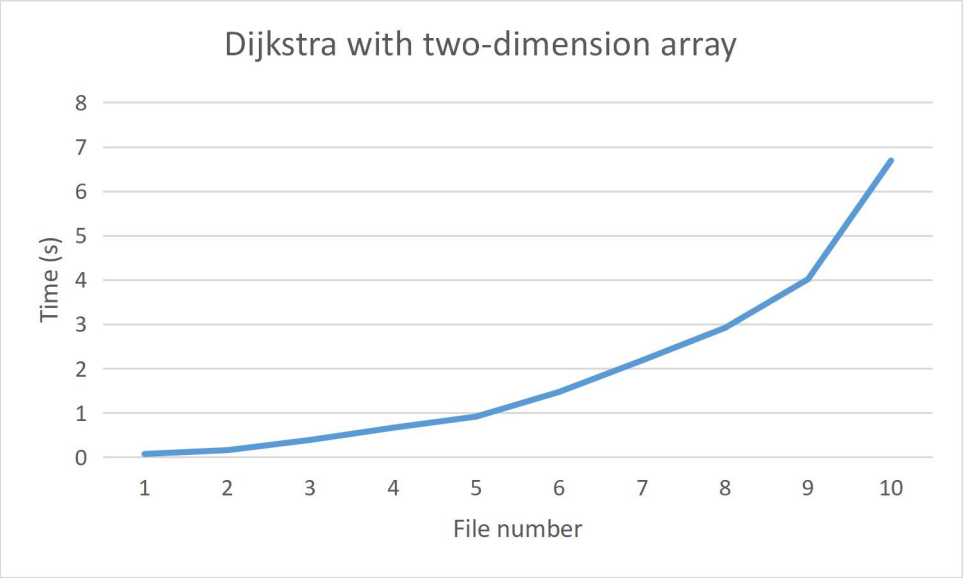
The definition of the link list used in this program is as same as it in Dijkstra with link list. The Floyd function is the same as it is in the book.

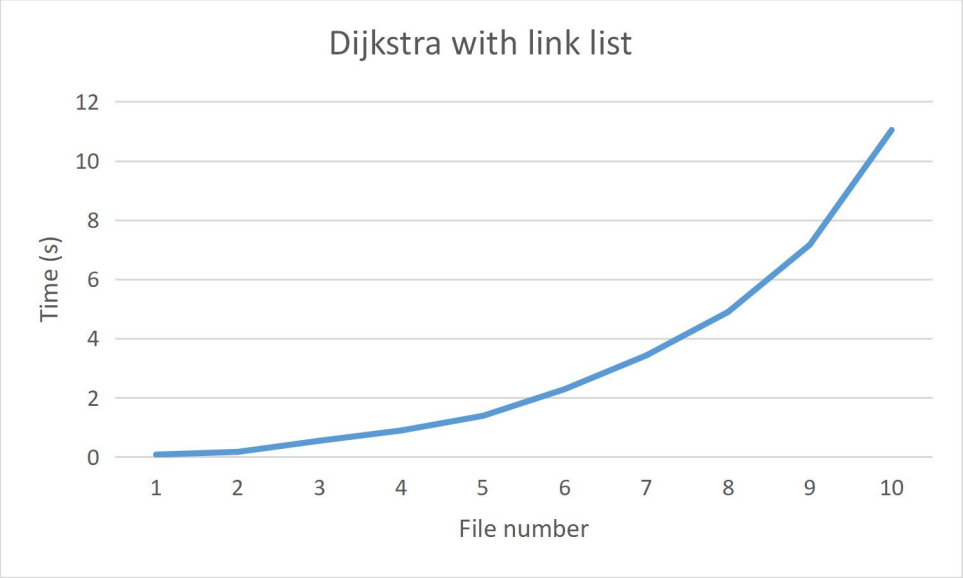
Test cases:

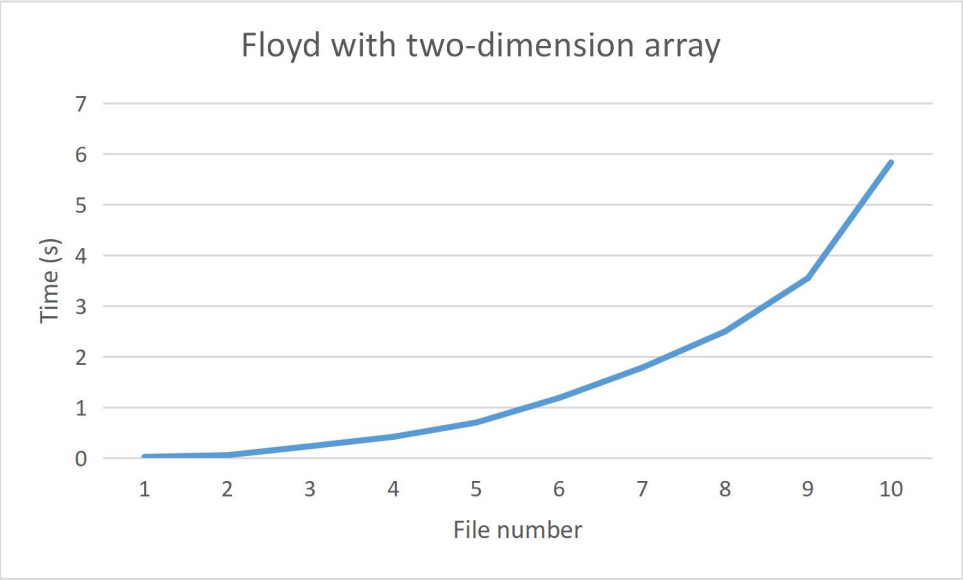
```
Floyd with link list
The path from 197 to 27 is:
197-> 198-> 303-> 293-> 142-> 26-> 27
The length of the path is:3009

The path from 65 to 280 is:
65-> 216-> 116-> 117-> 201-> 274-> 326-> 24-> 23-> 125-> 140-> 203-> 167-> 197-> 192-> 280
The length of the path is:4923

The path from 187 to 68 is:
187-> 238-> 229-> 231-> 264-> 247-> 17-> 18-> 242-> 158-> 77-> 78-> 136-> 137-> 332-> 70-> 134-> 176-> 269-> 286-> 300-> 318-> 290-> 302-> 323-> 277-> 175-> 68
The length of the path is:11199


Process finished with exit code 0
```

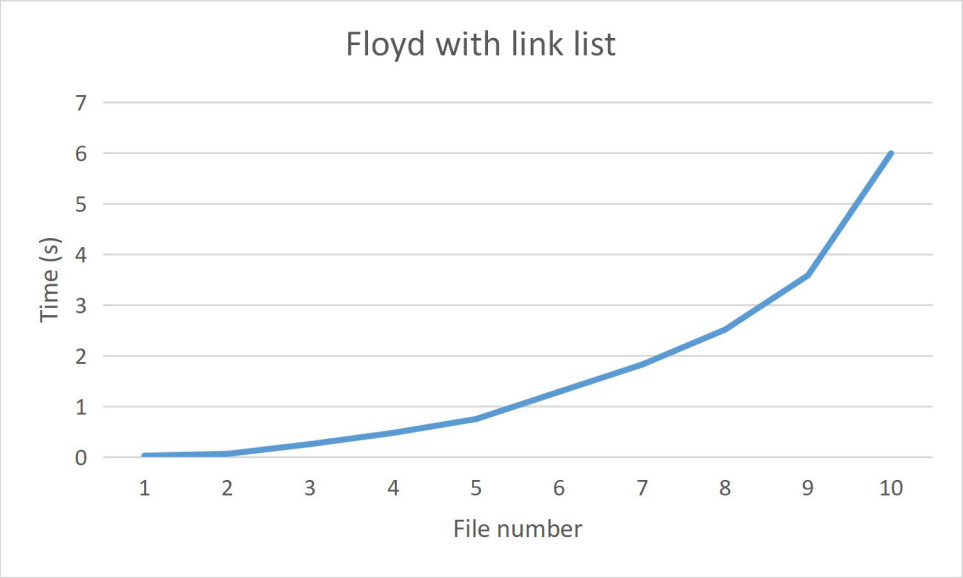
Performance and Memory Requirements.

Performances:









My platform:

Language: Java
IDE: IntelliJ IDEA (Community Edition 2020.3)
OS: Win10 (19042)
CPU: Intel i5-8300 H
RAM: 16G

The time taken is a little strange. Each running time for all 4 programs and all 10 input files does not exceed 10 seconds. However, all the programs works well, and all these times include the time to run the algorithms and get the result(touch array and F set for Dijkstra, D and P matrix for Floyd), also include the time to get all-pair shortest paths from touch array or P matrix of graphs. The times maybe are too short, but the trend is good and all programs are

working really well and have calculated all the results we need. However, if the programs are asked to print all the paths to the console, the time will become much longer. But if we only print the test cases, the time is short.

Memory requirements:

My IDE produce the memory requirements. However, I think this number includes the memory used by other parts of the jvm and IDE because it is pretty high. When the programs are running, it will fluctuate around 370M and when they are not, it will fluctuate around 240M.

Running

Not running

Because the number is keeping fluctuating all the time and also, the unit is M so the differences between these 4 programs is little, I will analyze the memory requirements from the codes.

We will assume there are n nodes and m edges in the graph. $\left[(n\times n)+(n\times n)+(n\times n)\right]\times 4=12n^2$

Dijkstra with two-dimension array: the adjacency matrix is a matrix of size n*n, the edge set: F is a matrix of size (n-1)*2, the touch array is an array of length (n-1), and the array which stores the shortest path and the length of the path is an array of size n. All the data type is int, and int type in java takes 4 bytes. Therefore, the memory used in Dijkstra with two-dimension array for each input file is $\left[(n\times n)+((n-1)\times 2)+(n-1)+n\right]\times 4=4n^2+16n-12$ bytes.

Dijkstra with link list: the array of link lists is an array of size n-1. The edge set: F is a matrix of size (n-1)*2, the touch array is an array of length (n-1), and the array which stores the shortest path and the length of the path is an array of size n. Each link list includes two int type variables: vertex and weight and a pointer points to the next node. So in the other words, the array of link lists has m nodes, and each node has 2 int type variables. Therefore, the memory used in Dijkstra with link list for each input file is $\left[(m\times 2)+((n-1)\times 2)+(n-1)+n\right]\times 4=8m+16n-12$ bytes.

Floyd with two-dimension array: the adjacency matrix is a matrix of size n*n, the matrix P and D are two matrices of size n*n. Printing the path does not need variables to store the path and length. Therefore, the memory used in Floyd with two-dimension array for each input file is $\left[(n\times n)+(n\times n)+(n\times n)\right]\times 4=12n^2$ bytes.

Floyd with link list: the array of link lists contains m edges, each edge corresponds to a node which contains 2 int type variables. The the matrix P and D are two matrices of size n*n. Printing the path also does not need variables to store the path and length in this program. Therefore, the memory used in Floyd with two-dimension array for each input file is $[(m \times 2) + (n \times n) + (n \times n)] \times 4 = 8n^2 + 8m$ bytes.