Objective(s):

a. To practice representing a graph using adjacency matrix
b. To understand the process of Dijkstra algorithm

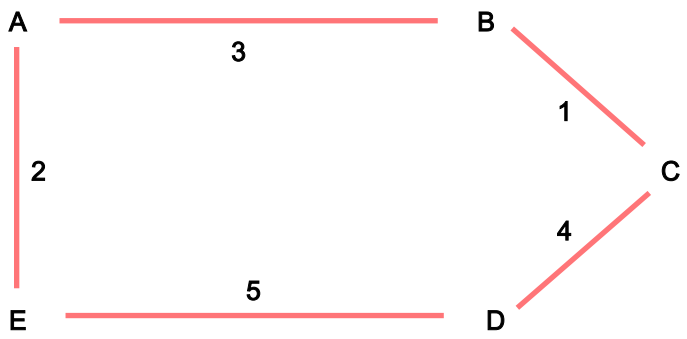
Task 1:

Given thisGraph = (V,E) which V = {A, B, C, D, E} and E = { (A,B,3), (A,E,2), (B,C,1), (C,D,4), (D,E,5) }

1.1 write int [][] thisGraph =

```
int[][] thisGraph = {
 //  A  B  C  D  E
    { 0, 3, 0, 0, 2 }, // A
    { 3, 0, 1, 0, 0 }, // B
    { 0, 1, 0, 4, 0 }, // C
    { 0, 0, 4, 0, 5 }, // D
    { 2, 0, 0, 5, 0 }  // E
};
```

1.2 draw thisGraph



Task 2:

In order to find a subgraph tree for thisGraph in Task1, we can use depth-first-search (depth-first-traversal) to perform the task.

```
//final static int inf = Integer.MAX_VALUE;
  static void q2() {
                        // A  B  C  D  E
  int[][] thisGraph = { { _, _, _, _, _ },  // your 1.1
                        { _, _, _, _, _ },
                        { _, _, _, _, _ },
                        { _, _, _, _, _ },
                        { _, _, _, _, _ } };
        System.out.println("computing dfs");
        q2_dfs(thisGraph);
    }
  private static void q2_dfs(int[][] thisGraph) {
    ArrayList<Integer> stack = new ArrayList<>();
    ArrayList<Integer> visited = new ArrayList<>();
    stack.add(0); // root is at A, we'll suffix next city
    while (notEmpty(stack)) {
      int parent = Integer.parseInt(processing);
      visited.add(parent);
      for (int x = 0; x < thisGraph.length; x++) {
        if (0 < thisGraph[parent][x]
            && thisGraph[parent][x] < inf && /* your code 1a */) {
                stack.add(parent + "" + x);
                /* your code 1b */
                System.out.println("Edge " + parent + ", " + x);
        }
      } //for
    }
  }
  private static boolean notEmpty(ArrayList<Integer> stack) {
        return !stack.isEmpty();
  }
```

We can simply use an ArrayList to be our stack because adding is appending by default (push) thus we only need to call remove (arraylist.size() -1) for pop(). This will allow the loop to come back to unexplored subtree. With the strategy, we can say the parent – child (x in the code) is the selected edge. When the algorithm terminates we obtain one of many possible tree for the graph.

The main pitfall when performing a dfs on a graph is that the code must prevent the dfs going into a loop (infinite, if so). We do this by keep a visited city list. (Let's call city 0 – 4 for A – E.

Instruction:

2.1 Complete the code and capture it to space provided.

2.2 draw the tree.

```
computing dfs
Edge 0 -> 1
Edge 0 -> 4
Edge 4 -> 3
Edge 3 -> 2
Edge 2 -> 1
```

```java
private static void q2_dfs(int[][] thisGraph) {
    ArrayList<Integer> stack = new ArrayList<>();
    ArrayList<Integer> visited = new ArrayList<>();
    stack.add(e:0); // Start from vertex A (index 0)

    while (!stack.isEmpty()) {
        int parent = stack.get(stack.size() - 1); // Get the top element from the stack
        stack.remove(stack.size() - 1); // Remove it from the stack
        visited.add(parent);

        for (int x = 0; x < thisGraph.length; x++) {
            // System.out.println("Checking edge : " + parent + " -> " + x);
            // System.out.println("Value : " + thisGraph[parent][x]);
            if (thisGraph[parent][x] != 0 && thisGraph[parent][x] != inf && !visited.contains(x)) {
                stack.add(x); // Add unvisited neighbors to the stack
                System.out.println("Edge " + parent + " -> " + x);
            }
        }
    }
}
private static boolean notEmpty(ArrayList<Integer> stack) {
    return !stack.isEmpty();
}
```

Task 3:

Given a structure of code for set for Dijkstra computation on single source shortest path below:

```
public class L11_GraphRep_Main {
  final static int inf = Integer.MAX_VALUE;
                                        //  A    B    C    D    E    F
  static int [][] distanceBetween = { {   0,   4,   5, inf, inf, inf},
                                      {   4,   0,  11,   9,   7, inf},
                                      {   5,  11,   0, inf,   3, inf},
                                      { inf,   9, inf,   0,  13,   2},
                                      { inf,   7,   3,  13,   0,   6},
                                      { inf, inf, inf,   2,   6,   0} };
  public static void main(String[] args) {
    q3();
  }
  Static void q3() {
    int A,B,C,D,E,F;    A = 0; B = 1; C = 2; D = 3; E = 4; F = 5;
    System.out.println("dijkstra from A");
    dijkstra(distanceBetween, A);

  }
  static void dijkstra(int [][] graph, int source) {
    int [] distance = initialize_distance_from_source(graph.length,source);
    boolean [] visited = new boolean[graph.length];

    while (moreCityToExplore(visited)) {
      int exploring = nextExplore(visited, distance);
         // dijkstra details
      visited[exploring] = true;
      println("exploring " + exploring + " " + Arrays.toString(distance));
    }   //while
  }
  private static int[] initialize_distance_from_source(int numCities, int source) {
    int [] distance = new int[numCities];
    for (int i = 0; i < numCities; i++)
        distance[i] = Integer.MAX_VALUE;
    /* your code 2*/; //start from source!!
    return distance;
  }
  private static boolean moreCityToExplore(boolean [] visited) {
    for (int i = 0; i < visited.length; i++)
        if (!visited[i])
            return /* your code 3*/;
    return /* your code 4*/;
  }
  private static int nextExplore(boolean [] visited, int [] dist) {
        int city_index = -1;
        /* int random_index = -1;
        while (city_index < 0) {
            random_index = (int)(Math.random() * 100) % visited.length;
            if (!visited[random_index])
                city_index = random_index;
        } */
        /* your code 5*/
        return city_index;
    }
}
```

Important!! /* your code 2 – 5 */ are not the main idea of Dijkstra algorithm. We focus on the flow for traversing the graph for now.

For readability, we define `inf` to be the largest possible integer value. We also define A, B, C, D, E F for the same reason. The adjacency matrix representing input graph is `int [][] distanceBetween` which is setup to be static so that it can be accessed from any inside method. (It's access modifier, of course, could be private.) Brief steps on computing Dijkstra is as follows:

Given a graph, create `int [] distance` for storing distance for each city from source. Initially, the only distance we know is the distance[source] = 0 because, analogically we are in the city. Next, `boolean [] visited` to keep track of computed city. The algorithm terminates when all cities are taken into consideration. Thus the algorithm iteratively pick an unvisited city for updating int [] distance until to more city to process. (Note that we'll leave `int nextExplore()` unchanged for now (Theoretically, the algorithm still terminates with the random selection policy).

Instruction: complete /* your code 2 */ to /* your code 4 */. For /* your code 5 */ simply uncomments the code. We are testing whether the code will terminate without any problem. Capture the code to the space provided.

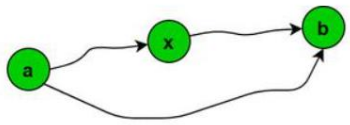
```
private static int[] initialize_distance_from_source(int numCities, int source) {
    int [] distance = new int[numCities];
    for (int i = 0; i < numCities; i++) distance[i] = Integer.MAX_VALUE;
    distance[source] = 0; /* Your code 2 */
    return distance;
}

private static boolean moreCityToExplore(boolean [] visited) {
    for (int i = 0; i < visited.length; i++) if (!visited[i]) return true; /* Your code 3 */
    return false; /* Your code 4 */
}
```

Task 4:

The updated (shorter) distance between a to b is demonstrated as shown in the accompanying figure. Given that dist(a,b) is known. Is there (city) x such that dist(a,x) + dist(x,b) such that it is less than existing dist(a,b). If so, update dist(a,b) value. (back-tracking the path of this shortest distance value is not part of this algorithm.

Triangle Inequality

- Let ***d(a, b)*** be the length of the shortest path from ***a*** to ***b*** *in graph* ***G_1***. Then,
  - ***$d(a, b) \leq d(a, x) + d(x, b)$***



Graph G_1

```
static void dijkstra(int [][] graph, int source) {
    int [] distance = initialize_distance_from_source(graph.length,source);
    // int [] prev = new int [distance.lenth];
    boolean [] visited = new boolean[graph.length];

    while (moreCityToExplore(visited)) {
        int exploring = nextExplore(visited, distance);
        boolean neighbor_of_exploring = false;
        for (int x = 0; x < distance.length; x++) {
           neighbor_of_exploring = 0 < distanceBetween[exploring][x]
                                   && distanceBetween[exploring][x] < inf ;
           if (neighbor_of_exploring) { // x is neighbor
                if (distance[x] /* your code 6 */) {
                    distance[x] /* your code 7 */;
                    // prev[x] = exploring;
                }
           }
        }
        visited[exploring] = true;
        // println("exploring " + exploring + " " + Arrays.toString(distance));
    }   //while
    // println("prev " + Arrays.toString(prev));
}
```

The difference between accompanied figured to the lecture's pseudo code is that this implementation adopts visited instead of pseudo code's Q. Noted that [] prev stores incoming city to the city.

Instruction: Correct the /* your code 5 */. Complete /* your code 6 */, /* your code 7 */ capture the code to the space provided.

```
if (neighbor_of_exploring) { // x is neighbor
    if (distance[x] > distance[exploring] + distanceBetween[exploring][x]) {/* your code 6 */
        distance[x] = distance[exploring] + distanceBetween[exploring][x];
        /* your code 7 */
        prev[x] = exploring;
    }
}
private static int nextExplore(boolean [] visited, int [] dist) {
    int city_index = -1;
    int random_index = -1;
    // while (city_index < 0) {
    //     random_index = (int)(Math.random() * 100) % visited.length;
    //     if (!visited[random_index]) city_index = random_index;
    // }
    // System.out.println("exploring city " + city_index + " with distance " + dist[city_index]);

    int min = Integer.MAX_VALUE;
    for (int i = 0; i < dist.length; i++) {
        if (!visited[i] && dist[i] < min) {
            min = dist[i];
            city_index = i;
        }
    }
    /* Your code 5 */

    return city_index;
}
```

Key Remark: One famous Dijkstra algorithm application is to find the best network connectivity (internet) route. Its implementation, in fact, use priority queue to expand the connectivity from the source. The priority queue abstracts the complexity of identifying the next node.

Dijkstra(Graph *G*, Vertex *s*)

1. Initialize(*G*, *s*);
2. Priority_Queue minQ = {all vertices in *V*};
3. **while** (minQ ≠ Ø) **do**
4. Vertex *u* = ExtractMin(minQ); // minimum est(*u*)
5. **for** (each *v* ∈ minQ such that (*u*, *v*) ∈ *E*)
6. Relax(*u*, *v*);
7. **end for**
8. **end while**

https://www.sciencedirect.com/topics/computer-science/dijkstra-algorithms

Submission: this pdf

01286222 Lab 11 Name.Chanasorn Howattanakulphong. id .....65011277.....

Due date: TBA