קורס מונחה עצמים – מטלה 0 – ספריה אלגוריתמית לגרפים לא מכוונים.

כללית: במטלה זו נפתח תשתית של מבנה נתונים, אלגוריתמים ומערכת תצוגה עבור משך המטלות בקורס. המטלה עוסקת בפיתוח של מבנה נתונים של גרף לא ממושקל, (ולא מכוון). לאחר מימוש מבנה הנתונים נממש מספר אלגוריתמים על הגרף לרבות חישוב מסלול קצר (מינימום צלעות), בדיקת קשירות (כמות מחלקות הקשירות), יכולת שכפול, ושמירה וטעינה מקבצים

שלבי עבודה:

1. בשלב הראשון עליכם להכיר את העקרונות של מבנה הנתונים של גרף מכוון, ולאחר מכן לתכנן כיצד אתם רוצים לממש אותו – דוגמאות והסברים נתנו בהרצאות והתרגולים.
2. ממשו את מחלקה Graph שמממשת את הממשק graph, עשו זאת בעזרת מימוש הממשקים של edge_data ו node_data.
3. ממשו את המחלקה Graph_Algo שמממשת את הממשק graph_algorithms ומייצגת אוסף של אלגוריתמים על גרפים.
4. כתבו תיעוד + הסברים מפורטים לגבי מבנה הנתונים, האלגוריתמים, מערכת התצוגה, וכמובן אופן השימוש בפרויקט מבחינת הורדה, והרצה.

איור 1: גרף מכוון וממושקל בעל 20 קדקודים. מספרים במרכז כל צלע מיצגים את המשקל שלה, והעיגולים הצהובים מייצגים את כיוון הצלע: משמע לקדקוד 8 (ו13) לא ניתן להגיע שכן אין לו אף צלע נכנסת – לפיכך גרף זה אינו קשיר.

הנחייה כללית:

- מטלה זו מוגדרת בעיקר ע"י מספר ממשקים שמגדירים את ה api הנדרש ממחלקות, לנוחיותכם מימשנו כבר (באופן ריק) את המחלקות הנדרשות, ועליכם להוסיף מחלקת בדיקה (Junit) לכל מחלקה לוגית שאתם כותבים. עליכם לעשות שימוש בממשקים - אותם אינכם יכולים לשנות, המחלקות המצורפות נועדו לבדיקה שלכם אתם יכולים (וצריכים) לעדכן אותן בהתאם להנחיות.
- שימו לב שעל המימוש להיות יעיל כך שניתן יהיה לבנות גרף של מיליון קדקודים (ופי 10 צלעות) במחשב רגיל בזמן ריצה סביר (נניח בפחות מ 10 שניות).

הנחיות הגשה:

את המטלה יש להגיש כפרויקט למערכת בדיקת המטלות לפי ההנחיות מפורטות שנמצאות באתר – מטלות שלא תוגשנה לפי ההנחיות לא תזכנה בציון מלא.

בהצלחה.

```
import java.util.Collection;
/**
 * This interface represents an undirectional unweighted graph.
 * It should support a large number of nodes (over 10^6, with average
degree of 10).
 * The implementation should be based on an efficient compact
representation
 * (should NOT be based on a n*n matrix).
 */

public interface graph {
   /**
    * init this graph with a deep copy of g
```

```
     * @param g
     */
    public void init(graph g);
    /** constructs a deep copy of this graph
     *
     * @return
     */
    public graph copy();
    /**
     * returns the node_data by the node_id,
     * @param key - the node_id
     * @return the node_data by the node_id, null if none.
     */
    public node_data getNode(int key);
    /**
     * returns the data of the edge (src,dest), null if none.
     * Note: this method should run in O(1) time.
     * @param node1
     * @param node2
     * @return
     */
    public edge_data getEdge(int node1, int node2);
    /**
     * add a new node to the graph with the given node_data.
     * Note: this method should run in O(1) time.
     * @param n
     */
    public void addNode(node_data n);
/**
 * Connects an edge with weight w between node src to node dest.
 * Note: this method should run in O(1) time.
 * Note2: if the edge node1-node2 already exists - the the method
simply does nothing.
 */
    public void connect(int node1, int node2);
    /**
     * This method return a pointer (shallow copy) for the
     * collection representing all the nodes in the graph.
     * Note: this method should run in O(1) time.
     * @return Collection<node_data>
     */
    public Collection<node_data> getV();
    /**
     * This method return a pointer (shallow copy) for the
     * collection representing all the edges getting out of
     * the given node (all the edges starting (source) at the given
node).
     * Note: this method should run in O(1) time.
     * @return Collection<edge_data>
     */
    public Collection<edge_data> getE(int node_id);
    /**
     * Deletes the node (with the given ID) from the graph -
     * and removes all edges which starts or ends at this node.
     * This method should run in O(n), |V|=n, as all the edges should
be removed.
     * @return the data of the removed node (null if none).
     * @param key
     */
    public node_data removeNode(int key);
    /**
```

```
     * Deletes the edge from the graph,
     * Note: this method should run in O(1) time.
     * @param src
     * @param dest
     * @return the data of the removed edge (null if none).
     */
    public edge_data removeEdge(int src, int dest);
    /** returns the number of vertices (nodes) in the graph.
     * Note: this method should run in O(1) time.
     * @return
     */
    public int nodeSize();
    /**
     * returns the number of edges (assume directional graph).
     * Note: this method should run in O(1) time.
     * @return
     */
    public int edgeSize();
/**
 * returns the Mode Count - for testing changes in the graph.
 * Any change in the inner state of the graph should cause an
increment in the ModeCount
 * @return
 */
    public int getMC();
}
/**
 * This interface represents the set of operations applicable on a
 * directional edge(src,dest) in an (undirectional, unweighted) graph.
 * @author boaz.benmoshe
 *
 */
public interface edge_data {
  /**
   * The id of the source node of this edge.
   * @return
   */
  public int getSrc();
  /**
   * The id of the destination node of this edge
   * @return
   */
  public int getDest();
  /**
   * return the remark (meta data) associated with this edge.
   * @return
   */
  public String getInfo();
  /**
   * Allows changing the remark (meta data) associated with this edge.
   * @param s
   */
  public void setInfo(String s);
  /**
   * Temporal data (aka color: e,g, white, gray, black)
   * which can be used be algorithms
   * @return
   */
  public int getTag();
```

```java
 /**
  * Allow setting the "tag" value for temporal marking an edge - common
  * practice for marking by algorithms.
  * @param t - the new value of the tag
  */
  public void setTag(int t);
}

/**
 * This interface represents the set of operations applicable on a
 * node (vertex) in an (undirectional) unweighted graph.
 * @author boaz.benmoshe
 *
 */
public interface node_data {
   /**
    * Return the key (id) associated with this node.
    * @return
    */
   public int getKey();
   /**
    * return the remark (meta data) associated with this node.
    * @return
    */
   public String getInfo();
   /**
    * Allows changing the remark (meta data) associated with this
node.
    * @param s
    */
   public void setInfo(String s);
   /**
    * Temporal data (aka color: e,g, white, gray, black)
    * which can be used be algorithms
    * @return
    */
   public int getTag();
   /**
    * Allow setting the "tag" value for temporal marking an node -
common
    * practice for marking by algorithms.
    * @param t - the new value of the tag
    */
   public void setTag(int t);
}
/**
 * This interface represents the "regular" Graph Theory algorithms
including:
 * 0. clone();
 * 1. init(String file_name);
 * 2. save(String file_name);
 * 3. isConnected();
 * 5. int shortestPathDist(int src, int dest);
 * 6. List<Node> shortestPath(int src, int dest);
 *
 * @author boaz.benmoshe
 *
 */

import java.util.List;
```

```java
public interface graph_algorithms {
    /**
     * Init this set of algorithms on the parameter - graph.
     * @param g
     */
    public void init(graph g);
    /**
     * Compute a deep copy of this graph.
     * @return
     */
    public graph copy();
    /**
     * Init a graph from file
     * @param file_name
     */
// public void init(String file_name);
    /** Saves the graph to a file.
     *
     * @param file_name
     */
    //public void save(String file_name);
/**
 * Returns true if and only if (iff) there is a valid path from EVREY
node to each
 * other node. NOTE: assume directional graph - a valid path (a-->b)
does NOT imply a valid path (b-->a).
 * @return
 */
    public boolean isConnected();
    /**
     * returns the length of the shortest path between src to dest
     * @param src - start node
     * @param dest - end (target) node
     * @return
     */
    public int shortestPathDist(int src, int dest) ;
    /**
     * returns the the shortest path between src to dest - as an
ordered List of nodes:
     * src--> n1-->n2-->...dest
     * see: https://en.wikipedia.org/wiki/Shortest_path_problem
     * @param src - start node
     * @param dest - end (target) node
     * @return
     */
    public List<node_data> shortestPath(int src, int dest);
}
```