המחלקה למתמטיקה ומדעי המחשב

סימולטור אלגוריתמים מבוזרים באמצעות Python והשוואתו ל - Sinalgo

פרויקט גמר מוגש כחלק מדרישות התואר

לקראת תואר שני M.Sc במדעי המחשב

האוניברסיטה הפתוחה

החטיבה למדעי המחשב

נכתב על ידי

סתו אברהם

תחת הנחייה של המנחה

ד״ר לאוניד ברנבוים

תוכן עניינים

אלגוריתמים מבוזרים 3
מבוא 3
אלגוריתם BFS מבוזר 4
Sinalgo 6
רקע 6
ארכיטקטורת על 7
כתיבת פרויקט 10
יתרונות 16
חסרונות 16
הסימולטור החדש שלנו- סימולטור Python 18
רקע 18
ארכיטקטורה 18
תצוגת GUI של הסימולטור 22
כתיבת פרויקט 23
יצירת כפתור מתוך הקוד 25
יתרונות מול Sinalgo 27
חסרונות מול Sinalgo 27
השוואת ביצועים Python-Simulator VS Sinalgo 28
אלגוריתם Flooding 28
מימוש ב Sinalgo 29
מימוש ב Python-simulator 29
השוואה בין הסימולטורים 30
אלגוריתם Shortest Paths 31
מימוש ב Sinalgo 31
מימוש ב Python-simulator 32
השוואה בין הסימולטורים 32
אלגוריתם Random Vertex Coloring 34
מימוש ב Sinalgo 34
מימוש ב Python-simulator 35
השוואה בין הסימולטורים 36
Multi Threaded Python-Simulator 37
רקע 37
השוואה 38
מסקנות 39
מחקר עתידי 41
סיכום 42
מקורות 43

אלגוריתמים מבוזרים

מבוא

כיום עקב התפתחות האינטרנט, המערכות המפותחות הינן גדולות ומורכבות מבעבר.
מערכות אלו כבר אינן עומדות באופן בודד אלא מתקשרות עם מערכות אחרות על מנת לתת מענה פיזיבילי ומהיר.
עקב השינוי ההולך וגובר מבחינת חלוקת המערכות ותקשורת בין המערכות השונות פותח ענף האלגוריתמים המבוזרים.
אלגוריתם מבוזר הינו אלגוריתם הרץ על מספר nodes במקביל.

אלגוריתמים אלו יכולים להתבצע בשני סוגים של מודלים:
מודלים סינכרוניים - מניחים כי קיים שעון גלובאלי המתקדם בקצב אחיד וסופר יחידות זמן,
מודלים אסינכרוניים - בהם אין שעון גלובאלי ולכל קודקוד יש יחידות זמן לוקאליות משלו.

השוני העיקרי בין אלגוריתמים אלו לעומת אלגוריתמים במודל הסדרתי המוכר לנו הינו שבמודל הסדרתי בדר"כ לאלגוריתם נתון גרף G(V,E) המכיל את קבוצת הקדקודים V וקבוצת הצלעות E וניתן לגשת לכל קדקוד/צלע בקבוצות אלו.
במודל המבוזר כל קדקוד הוא יחידה עצמאית ויכול להכיר אך ורק את שכניו.
בכל סיבוב ריצה האלגוריתם ירוץ על כל הקדקודים פעם אחת.

חישוב יעילות זמן ריצה אינו כמו במודל הסדרתי כיוון שלא ניתן להשוות מהירויות בפועל.
בחירת מודל היא לפי מצב המערכת הנתון לנו וכי לא ניתן לומר בוודאות מה ירוץ מהר יותר בפועל, יכול להיות שהמודל המבוזר כיוון שהאלגוריתם רץ על כל node במקביל ולא עוברים על כל אחד בצורה איטרטיבית ויכול להיות שהמודל הסדרתי כיוון שסיבוכיות התקשורת בין הקדקודים גבוהה מאוד.

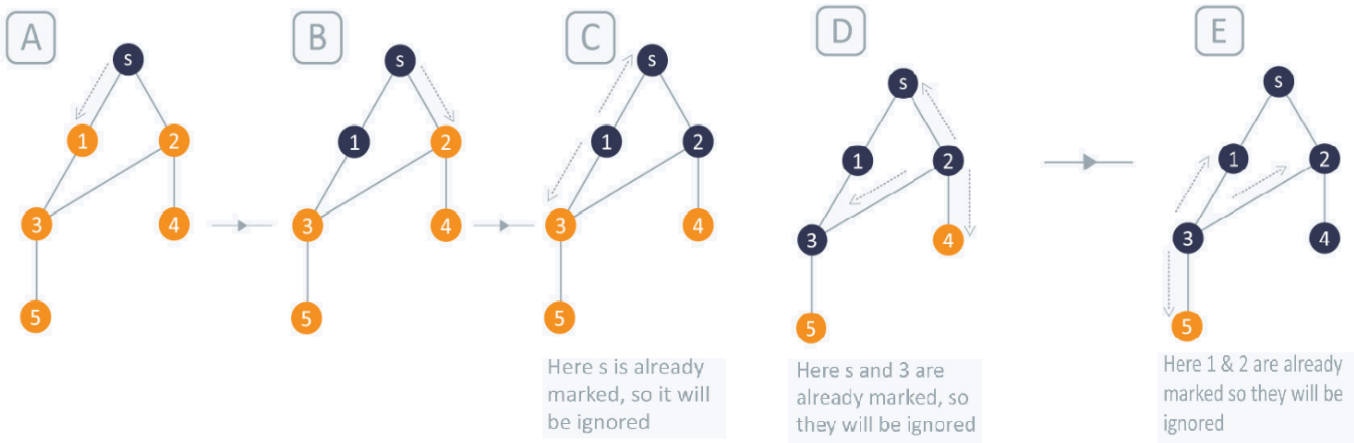
במודל המבוזר יש חשיבות גבוהה יותר למספר הסיבובים שהאלגוריתם רץ עד שהשיג את התוצאה הרצויה ולא לזמן שלקח לו לרוץ בפועל על node ספציפי
ישנם פרמטרים נוספים של יעילות (גודל הודעות/תעבורה, זמן תקשורת ועוד)

אלגוריתם BFS מבוזר

נראה דוגמא לאלגוריתם BFS המוכר לנו היטב גם בצורתו הסדרתית וגם בצורתו המבוזרת:

BFS בצורתו הסדרתית:

```
BFS (G, s):          //Where G is the graph and s is the source node
  let Q be queue
  Q.enqueue( s )
  mark s as visited.
  while ( Q is not empty)
    v  =  Q.dequeue( )
    for all neighbours w of v in Graph G
      if w is not visited
        Q.enqueue( w )
        mark w as visited.
```

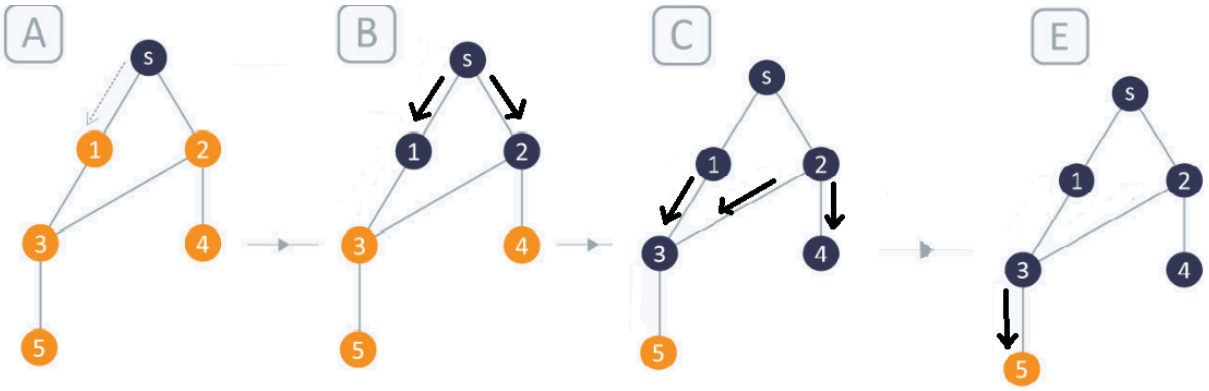


איור 1 אלגוריתם BFS בצורתו הסדרתית

BFS בצורתו המבוזרת (במודל האסינכרוני):

```
States: distance, initially 0 for initiator and inf for all other nodes, internal send buffers

Initiator initialization code:
   send distance to all neighbors

All processes:
   upon receiving d from p:
      if d+1 < distance:
         distance := d+1
         parent := p
         send distance to all neighbors
```

איור 2 אלגוריתם BFS בצורתו המבוזרת במודל אסינכרוני



איור 2 אלגוריתם BFS בצורתו המבוזרת במודל אסינכרוני

Sinalgo

רקע

על מנת לבדוק אלגוריתמים במודל המבוזר באופן נוח ומהיר, פותחו מספר סימולטורים שנותנים את היכולת להריץ על המחשב הבודד את האלגוריתם, אולם להריצו באופן המדמה סביבה מבוזרת הבנוייה עם מספר מעבדים שכל אחד מהם יריץ קטע קוד המוגדר על ידי המשתמש במספר סיבובים שנבחר.

בקורס אלגוריתם מבוזרים לרשתות תקשורת **(22932)** המוצע באוניברסיטה הפתוחה ישנו שימוש באחד הסימולטורים הללו שנקרא Sinalgo.
סימולטור זה פותח במעבדה לאלגוריתם מבוזרים באוניברסיטת ETH בשוויץ.

הסימולטור כתוב בשפת JAVA וממשמש את היחידות השונות של הרשת כמו קדקוד, צלע, הודעה וכו'.
המימוש מתבצע בכתיבת מחלקות השונות בשפת תכנות מונחית עצמים.

לסימולטור זה יכולות רבות כגון :
הרצת אלגוריתמים באופן אסינכרוני/סינכרוני
הרצת אלגוריתמים כאשר הקדקודים חשופים לרעשים
שימוש בצורות שונות של גרפים כמו גרפי דיסק יחידה וכו'
תצוגת משתמש נוחה המאפשרת לראות בצורה ויזואלית את התהליך
ועוד...

סימולטור זה נוח לשימוש ובהמשך נצלול עמוק יותר על מנת להבינו.

ארכיטקטורת על

Sinalgo הינה אפליקציית Desktop הכתובה בשפת מונחית עצמים JAVA.
בסימולטור מפותחות מחלקות בסיס המייצגות אובייקטים בגרף לדוגמא:

- Node
- Message
- Edge

ניתן לרשת ממחלקות אלו וליצור אובייקטים מורחבים
נסביר על המחלקות העיקריות ולאחר מכן איך הסימולטור עובד בכללי
בסימולטור ישנן מספר רב של מחלקות ולכן נציג ברמת את העיקריות שבהן:

1. **מחלקת Main**
נתיב: eclipse-workspace\sinalgo\src\sinalgo\runtime\Main.java
מטרה: EntryPoint של הסימולטור
תפקידים עיקריים:
   a. קובע האם הסימולטור רץ ב Gui mode או ב Batch mode ומאתחל משתנה runtime להיות ב mode הנכון (אם בא מה console מבצע parsing לפקודות מסויימות שהתווספו, לדוגמא שם פרויקט)
   b. מאתחל מודל default שהוא MessageTransmissionModel
   c. טוען פרמטרים מהקונפיגורציה (לדוגמא האם הסימולטור רץ במצב Sync או Async)
   d. מאתחל אובייקט Global שערכיו יהיו נגישים לשאר המחלקות

2. **מחלקת Runtime**
נתיב: eclipse-workspace\sinalgo\src\sinalgo\runtime\Runtime.java
מטרה: מחלקת אב אבסטרקטית של GuiRuntime, BatchRuntime, חלק ליבה של ה framework
תפקידים עיקריים:
   a. פעולות בסיסיות על אובייקט "קדקוד" (מחיקה, הוספה וכו')
   b. Initialize למערכת כולל parsing של פקודות נוספות שהגיעו מהמשתמש (ערכי defaults במידה והגיע מה GUI)
   c. יצירת nodes
   d. מתודת abstract של Run שכל אחד מה-modes יממש בעצמו

3. **מחלקת GuiRuntime**
נתיב: eclipse-workspace\sinalgo\src\sinalgo\runtime\GuiRuntime.java
מטרה: מחלקה האחראית על טיפול ותחילת הריצה במוד GUI
תפקידים עיקריים:
   a. מימוש מתודת Run האחראית לתחילת ריצת הסימולטור
   b. איתחול של ה GUI והמתנה שהסימולטור ימתין עד שכל האובייקטים יצויירו
   c. יצירת קדקודים על ידי שימוש בפונקציית אב

4. **מחלקת BatchRuntime**
נתיב: eclipse-workspace\sinalgo\src\sinalgo\runtime\BatchRuntime.java
מטרה: מחלקה האחראית על טיפול ותחילת הריצה במוד Batch
תפקידים עיקריים:
   a. יצירת קדקודים על ידי שימוש בפונקציית אב
   b. מימוש מתודת Run האחראית לתחילת ריצת הסימולטור

5. **מחלקת AbstractCustomGlobal**
נתיב: eclipse-workspace\sinalgo\src\sinalgo\runtime\AbstractCustomGlobal.java
מטרה: מחלקה אבסטרקטית הנותנת פונקציונליות אבסטרקטית לשליטה כללית של המשתמש במערכת
תפקידים עיקריים (במחלקה זו יופיעו ההצהרות ולא המימושים עצמם):

a. Interface הוספת כפתור ל main panel
b. אבסטרקציה (צריך לממש במחלקת בת) לפונקציונליות שתרוץ לפני כל סיבוב ולאחר כל סיבוב
c. אבסטרקציה (צריך לממש במחלקת בת) האם הסימולטור נגמר, משתמש יצא וכו'
d. אבסטרקציה (צריך לממש במחלקת בת) לפונקציונליות שתרוץ לאחר הוספת/מחיקת קדקוד
e. טיפול ב Timers גלובליים (שימוש פנימי של הסימולטור, מפתחים לא אמורים להשתמש בזה, קורה לפני כל סיבוב)

6. **מחלקת CustomGlobal**

נתיב : eclipse-workspace\sinalgo\src\projects\< project-name>\CustomGlobal.java

מטרה : מימוש של מחלקת AbstractCustomGlobal, מחזיק את המתודות הכלליות עבור ה framework, נטען בקובץ main ונשמר במשתנה בקובץ Global.java

תפקידים עיקריים :

a. יכולת מימוש כל אחד מפונקציונליות של מחלקת האב (חובה לממש פונקציית hasTerminated)
b. יכולת ליצור את קדקודי הגרף ואת צלעות הגרף
c. הוספת tigger המופעל באמצעות כפתור ב panel

7. **מחלקת Node**

נתיב : eclipse-workspace\sinalgo\src\sinalgo\nodes\Node.java

מטרה : מייצגת קדקוד המכיר את עצמו ואת שכניו

תפקידים עיקריים :

a. אבסטרקציה (צריך לממש במחלקת בת) לפונקציונליות שתרוץ בכל סיבוב עבור כל אחד מהקדקודים (handleMessages)
b. שליחת הודעה לקדקוד מסויים
c. שליחת הודעה לכל השכנים
d. מחזיק משתנה Inbox המכיל את ההודעות שנצברו עבור הקדקוד
e. פונקציית step() הנקראת על ידי המערכת (לא לשימוש מפתח) ומריצה עבור כל קדקוד בכל סיבוב את השלבים שהסימולטור מבצע לכל קדקוד, כולל הפעלה של הפונקציה handleMessages
f. פונקציונליות של תצוגת הקדקוד במוד Gui כולל מיקום, צבע, צורה וכו'

8. **מחלקת Edge**

נתיב : eclipse-workspace\sinalgo\src\sinalgo\nodes\edges\Edge.java

מטרה : מחלקה המייצגת את אובייקט צלע ברשת

תפקידים עיקריים :

a. מחזיקה משתנים המייצגים את הקדקודים בקצוות
b. יכולות גרפיות (עבור Gui mode)
c. קבלת כמה הודעות עברו בצלע (מספר)
d. מחיקת הצלע מהגרף

9. **מחלקת Message**

נתיב : eclipse-workspace\sinalgo\src\sinalgo\nodes\messages\Message.java

מטרה : מחלקה המייצגת את אובייקט הודעה העובר בין קדקודים

תפקידים עיקריים :

a. Clone() , פונקציה אבסטרקטית שצריך לדרוס במחלקת בת, כאשר קדקוד מעביר הודעה ה framework שולח העתק שלה על מנת שלא יהיה ניתן לשנות את ההודעה המקורית, פונקציה זו מחזירה את ההעתק שה framework שולח (המחלקה המשתמשת בפונקציה זו היא מחלקת Node)

10. **מחלקת Inbox**

נתיב : eclipse-workspace\sinalgo\src\sinalgo\nodes\messages\Inbox.java

מטרה : כל ההודעות שהגיעו לקדקוד בסיבוב מסויים נשמרות ב Inbox (בתור Packet List) וניתן לקרוא אותם משם

תפקידים עיקריים :

a. Iterator על גבי ההודעות שנשמרו
b. מחיקה הודעה

c. פונקציה המחזירה את הקדקוד השולח של ההודעה הנוכחית (activePacket)
d. פונקציה המחזירה את הקדקוד המקבל של ההודעה הנוכחית (activePacket)

11. **מחלקת Packet**

נתיב : eclipse-workspace\sinalgo\src\sinalgo\nodes\messages\Packet.java

מטרה : עטיפה לאובייקט הודעה ולמידע הנלווה, את האובייקט הזה Inbox מכיל

תפקידים עיקריים :

a. שומר מידע נלווה כמו זמן הגעה, זמן שליחה וכו'
b. מכיל את אובייקט Message

נסביר את אופן פעולת הסימולטור, נתמקד במצב Sync :

בוחרים מוד (gui/batch)

פונקציית main מופעלת ומאתחלת את כל המשתנים הרלוונטים לסימולטור (runtime, CustomGlobal וכו')

פונקציית run מופעלת על ידי המשתמש

עבור כל סיבוב (במידה ולא נחלץ כפתור – עצור)

מתבצע customGlobal.preRound()

עדכון כל הגרף על מנת שכל קדקודי הגרף יעבדו על הגרף המעודכן (עדכון connections)

עבור כל קדקוד שימוש בפונקציית step() (**באופן איטרטיבי**) שנמצאת במחלקת קדקוד והוזכרה קודם בפירוט המחלקות (כאן כל קדקוד מבצע את פונקציית handleMessages())

מתבצע customGlobal.postRound()

Redraw של כל הגרף במוד Gui

טיפול במקרה יציאה/כתיבת לוגים

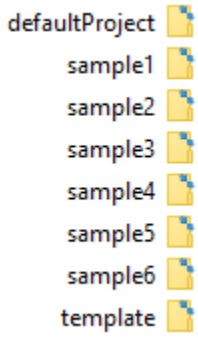
לאחר שכל הסיבובים שהוצהר עליהם נגמרו, enabled לכפתורים וציור הגרף מחדש

הערה:

כפי שניתן לראות בעת הפעלת מוד Gui ישנם מספר פעולות (כמו ציור כל הגרף מחדש ועדכון כל הגרף על מנת שכל הקדקודים יעבדו על הגרף המעודכן) שיכולות לגרום לאיטיות הסימולטור,
מצד שני אם נשתמש במוד Batch לא נוכל להנות מתצוגה ויזואלית נוחה ונאלץ להבין outputs מה console

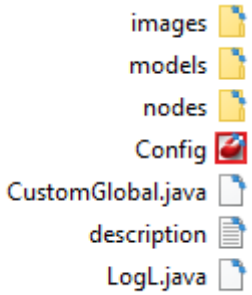
כתיבת פרויקט

כאשר רוצים להתחיל ולכתוב פרויקט בסימולטור sinalgo ולאחר שמורידים את הקבצים הרלוונטים ניתן לראות בתיקיית src - projects> את הפרוייקטים לדוגמא:



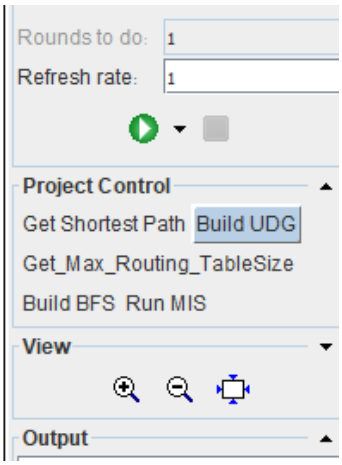
איור 3 פרויקטים לדוגמא בסימולטור sinalgo

ניתן להעתיק תייקת template ולשנות את שמה לשם הפרויקט הרלוונטי, הפרויקט יכיל את הקבצים הבאים:



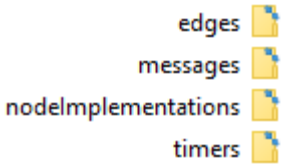
איור 4 רשימת תיקיות וקבצים בתיקיית templates

הקובץ CustomGlobal מייצג את כל מה שגלובאלי לדוגמא: כפתורים שיהיו פעילים בעת הסימולטור יצירת הגרף, בחירת אלגוריתם מסוים להרצה וכו':



איור 5 חלק מתצוגת GUI של סימולטור Sinalgo

בתיקיית nodes יהיו הקבצים הבאים:



איור 6 רשימת קבצים בתיקיית nodes

כאשר בכל תיקייה נוכל ליצור את האוביקט הרלוונטי שיירש מהמחלקת אב המתאימה לדוגמא מחלקות Node ו Message:

```
import java.awt.Color;

public class GraphNode extends Node {

    public GraphNode() {

    }
```

```
import sinalgo.nodes.messages.Message;

/**
 * A message sent to children that should be marked.
 */
public class MarkMessage extends Message {

    public MarkMessage() {

    }

    @Override
    public Message clone() {
        return this; // read-only policy
    }
```

איור 7 קטעי קוד של מחלקות GraphNode, MarkMessage

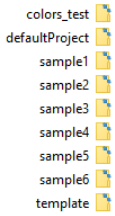
נוכל ליצור גרף שבו כל קדקוד ייצג את המימוש שלנו למחלקת קדקוד (לדוגמא GraphNode)

נראה דוגמה לאלגוריתם הבא הצובע גרף עץ בשני צבעים (3 צבעים כולל השורש):

```
Message type colorA ="yellow", colorB = "blue", rootColor= "red", baseColor = Black
All Color(Nodes) = baseColor

in each round r = 1,2,3,..., do:
 If Color(i) == baseColor:
   if (i == root):
      Color(i) = rootColor
      For each Child k of i:
        send_message(colorA, k)
   else:
      receive  msg(j)
      Color(i) = msg(j)
      if (msg(j) == colorA):
        color_for_sending = colorB
```

```
else:
  color_for_sending = colorA
For each Child k of i:
  If (k != j):
    send_message(color_for_sending, k)
```

1. נשכפל את תיקיית template וניצור פרויקט חדש הנקרא colors_test על ידי שינוי שם התיקייה



איור 8 רשימת תיקיות בסימולטור Sinalgo כולל הקובץ החדש colors_test

2. נשנה בכל הקבצים הרלוונטים את שם ה package מ template ל colors_test:

```
package projects.template;

import javax.swing.JOptionPane;
```

```
package projects.colors_test;

import javax.swing.JOptionPane;
```

איור 9 שינוי בשמות ה packages

3. ניצור מחלקת GraphNode בנתיב nodes -> nodeImplementations, במחלקה זו נממש את האלגוריתם שכתבנו:

```java
import sinalgo.nodes.messages.Message;

public class GraphNode extends Node {

    private GraphNode parent = null;
    private Vector<GraphNode> childs = new Vector<>();
    private Color colorA = Color.YELLOW;
    private Color colorB = Color.BLUE;
    private Color rootColor = Color.RED;
    private Color baseColor = Color.BLACK;

    public GraphNode(int id) {
        this.ID = id;
        this.setColor(this.baseColor);
    }

    @Override
    public void handleMessages(Inbox inbox) {
        this.fixEdges();
        if (this.getColor() == this.baseColor) {
            if (this.checkIfRoot() == true) {
                System.out.println("aaa");
                this.setColor(this.rootColor);
                this.sendMsgToAllNeighbors(new ColorMessage(this.colorA));
            } else {
                if (inbox.hasNext()) {
                    ColorMessage msg = (ColorMessage) inbox.next();
                    this.setColor(msg.color);
                    Color colorForSending = msg.color == this.colorA ? this.colorB : this.colorA;
                    this.sendMsgToAllNeighbors(new ColorMessage(colorForSending));
                }
            }
        }
    }

    public void fixEdges() {
        this.outgoingConnections = new NodeOutgoingConnectionsList(true);
        for (GraphNode node : this.childs) {
            this.addConnectionTo(node);
        }
    }

    public ArrayList<GraphNode> getNeighbors() {
        ArrayList<GraphNode> neighbors = new ArrayList<>();
        for (Edge e : this.outgoingConnections) {
            neighbors.add((GraphNode) e.endNode);
        }
        return neighbors;
    }

    public void sendMsgToAllNeighbors(Message msg) {
        ArrayList<GraphNode> neighbors = this.getNeighbors();
        for (GraphNode neighbor : neighbors) {
            this.send(msg, neighbor);
        }
    }

    public boolean checkIfRoot() {
        return this.parent == null;
    }

    public void setParent(GraphNode parentNode) {
        this.parent = parentNode;
    }

    public void setChild(GraphNode childNode) {
        this.childs.add(childNode);
    }

    public void preStep() {...

    public void init() {...

    public void neighborhoodChange() {...

    public void postStep() {...

    public void checkRequirements() throws WrongConfigurationException {...
```

איור 10 מימוש אלגוריתם במחלקת GraphNode

4. ניצור מחלקת ColorMessage בנתיב nodes -> messages:

```java
package projects.colors_test.nodes.messages;

import java.awt.Color;

import sinalgo.nodes.messages.Message;

public class ColorMessage extends Message {

    public Color color;

    public ColorMessage(Color color) {
        this.color = color;
    }

    @Override
    public Message clone() {
        return this;
    }

}
```

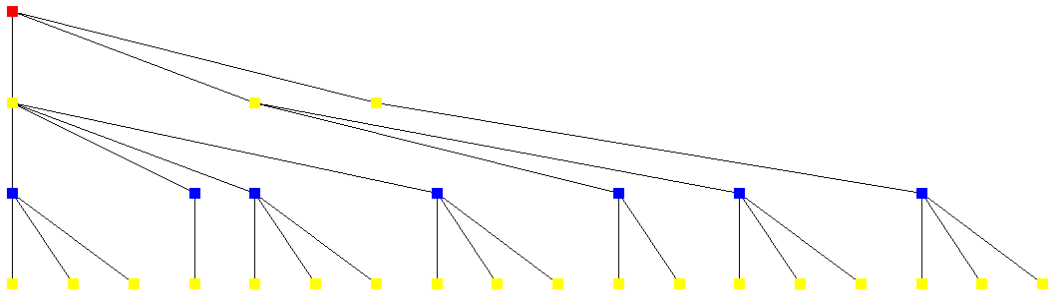
איור 11 מימוש אלגוריתם עבור מחלקת ColorMessage

5. נבנה את גרף העץ בקובץ CustomGlobal,
כאשר ניתן למשתמש להכניס מספר עלים ודרגה מקסימלית:

```
import sinalgo.tools.Tools;

public class CustomGlobal extends AbstractCustomGlobal {

    @Override
    public boolean hasTerminated() {
        return false;
    }

    @AbstractCustomGlobal.CustomButton(buttonText = "Build Tree", toolTipText = "Builds a tree")
    public void buildTreeButton() {
        int numLeaves = Integer.parseInt(Tools.showQueryDialog("Number of leaves:"));
        int maxDegree = Integer.parseInt(Tools.showQueryDialog("Max Degree:"));
        buildTree(maxDegree, numLeaves);
    }

    public void buildTree(int maxDegree, int numLeaves) {
        if (maxDegree < 1) {
            Tools.showMessageDialog("The max degree needs to be at least 1.\nCreation of tree aborted.");
            return;
        }
        if (numLeaves <= 0) {
            Tools.showMessageDialog("The number of leaves needs to be at least 1.\nCreation of tree aborted.");
            return;
        }
        if (maxDegree == 1 && numLeaves > 1) {
            Tools.showMessageDialog(
                    "Cannot create tree with max degree 1 and more than 1 leaves.\nCreation of tree aborted.");
            return;
        }

        double dx = ((double) Configuration.dimX) / (numLeaves + 1);
        double dy = Configuration.dimY - 30;
        int idCounter = 0;
        Vector<GraphNode> leaves = new Vector<>();

        // remove all nodes (if any)
        Runtime.clearAllNodes();
        leaves.clear();

        // create the leaves
        for (int i = 0; i < numLeaves; i++) {
            GraphNode leave = new GraphNode(++idCounter);
            leave.setPosition((i + 1) * dx, dy, 0);
            leave.finishInitializationWithDefaultModels(true);
            leaves.add(leave);
        }

        // build the parents by max degree
        int level = 0;
        int childPosInLvl = 0;
        Vector<GraphNode> currentLevelNodes = new Vector<>();
        Vector<GraphNode> prevLevelNodes = leaves;
        while (prevLevelNodes.size() > 1) {

            GraphNode node = new GraphNode(++idCounter);
            node.setPosition((childPosInLvl + 1) * dx, dy - ((level + 1) * 80), 0);
            node.finishInitializationWithDefaultModels(true);
            int degree = (int) (1 + Math.random() * ((maxDegree + 1) - 1));
            int limit = degree < prevLevelNodes.size() - childPosInLvl ?
                    degree : prevLevelNodes.size() - childPosInLvl;
            for (int i = 0; i < limit; i++) {
                (prevLevelNodes.get(childPosInLvl)).setParent(node);
                node.outgoingConnections.add(node, prevLevelNodes.get(childPosInLvl), true);
                node.setChild(prevLevelNodes.get(childPosInLvl));
                childPosInLvl++;
            }
            currentLevelNodes.add(node);

            if (childPosInLvl == prevLevelNodes.size()) {
                level++;
                prevLevelNodes = currentLevelNodes;
                currentLevelNodes = new Vector<>();
                childPosInLvl = 0;
            }
        }
        // Repaint the GUI as we have added some nodes
        Tools.repaintGUI();
    }
}
```

איור 12 מימוש יצירת גרף עץ במחלקת CustomGlobal

6. ולבסוף נקבל צביעה של העץ (מספר הסיבובים יהיה כגובה העץ):



איור 13 תוצאת האלגוריתם לצביעת עץ

יתרונות

לסימולטור Sinalgo ישנם מספר גדול של יתרונות כגון:

- תומך בשני מצבים של אלגוריתמים – מצב סינכרוני/אסינכרוני
- מודולרי – כתוב בגישת מונחית עצמים, קיימת מחלקת אב לכל אובייקט בגרף ויכולת לעשות customization לאובייקטים אלו
- תומך ביכולת לסמלץ רעשים ש Node יהיה חשוף אליהם
- יכולת ליצור מודלי גרף שונים כמו גרף דיסק יחידה
- תצוגת GUI נוחה בעלת יכולות רבות
  - יצירת גרף (קדקודים, צלעות) בעזרת ה GUI
  - יצירת Buttons המפעילים Event בהתאם למה שהמשתמש רוצה
  - הרצת Round בודד או הרצה מתמשכת
  - תצוגה דינאמית של הגרף (יכולת להזיז(קאורדינאטות)/למחוק/להוסיף אובייקטים של הגרף כמו קדקודים/צלעות)
  - תצוגה גרפית של קדקודי הגרף, לדוגמא גודל/צבע/צורה וכו'
- יצירת פרויקט חדש באופן פשוט
- סוגים שונים של מודלים: MessageTransmissionModel, ConnectivityModel, DistributionModel etc
- פרדיגמה אינטואיטיבית וקלה להבנה (ביחס לפרדיגמות אחרות בתחום לדוגמא MapReduce)
- קוד פתוח – ניתן להרחיב/לשנות את הסימולטור כרצוננו כיוון שה Source code נמצא אצלנו בעת ההורדה

חסרונות

לסימולטור Sinalgo מספר חסרונות, כגון:

- במידה ובוחרים במוד GUI אין הפרדה בין המנוע המריץ את הקוד הכתוב לבין האובייקטים הגרפים, כל צד ה GUI מוטמע בקוד ולא ניתן באופן מובנה לשינוי/שיפור התצוגה (נניח תצוגה על ידי דפדפן ולא ב Desktop)
- Sinalgo מציג את הגרף בשלמותו בעת היצירה ולא בצורה מצטברת (בסימולטור לא נוכל להריץ עשרות אלפי/ מאות אלפי קדקודים שייטענו בזמן סביר)
- Sinalgo מריץ בכל סיבוב את פונקציית הטיפול של כל קדקוד באופן איטרטיבי וללא שימוש מבוקר ב - Threads לצורך שיפור מהירות זמן הסימולציה (ניתן לראות בקובץ GUIRuntime.java בפונקציית run כי בהינתן ש thread רץ

לא ייפתחו עוד)

- במידה ובוחרים במוד GUI הסימולטור מעדכן בכל סיבוב את תצוגת ה GUI (נניח צבע/מיקום קדקוד וכו') דבר שעלול להיות מורכב מבחינת זמן ריצה כאשר מדובר בגרף גדול המשתנה לעיתים קרובות ואלגוריתם הדורש מספר לא קטן של סיבובים – בעיה זו יכולה לגרום לסימולטור לפעול באיטיות רבה

- במידה ובוחרים במוד Batch על מנת להרוויח מבחינת ביצועים לא נוכל לראות ויזואליזציה של המצב ונאלץ להפעיל את הפרויקט מחדש.

- הסימולטור מכיל את כל ה components (קוד, Gui וכו') בזיכרון ללא אפשרות להפריד, בעיה זו גורמת לסימולטור במצב Gui לקרוס (זיכרון מתמלא) בעת הרצה של אלגוריתם על גרפים גדולים

הסימולטור החדש שלנו- סימולטור Python

רקע

כפי שראינו בפרק הקודם לסימולטור Sinalgo הינו סימולטור עשיר ביכולות ובעל מורכבות גם בהיבט הלוגי וגם בהיבט התכנותי.
אולם לסימולטור מספר חסרונות שלעיתים יכולים לפגוע באלגוריתמיקאי/סטודנט המעוניין לבדוק את האלגוריתם שכתב.

לכן בפרויקט זה עלה הרעיון של כתיבת סימולטור בגישה קצת שונה משאר הסימולטורים שפותחו בפרדיגמה דומה ל Sinalgo.

ראינו שבמידה ורוצים להרוויח יכולות ויזואליות ברמה גבוהה צריך להתפשר על ביצועים
(כפי שראינו ב flow של Sinalgo),
ובמידה ולא רוצים להתפשר על ביצועים נאלץ לעבוד במוד ללא הצגה ויזואלית ולנסות להבין את ה outputs השונים.

לכן העיקרון כאן הוא שונה, בדומה לגישת MVC הידועה בתחום ה-Architecture pattern,
גם כאן נעשתה הפרדה בין "מנוע" הסימולטור לתצוגה שלו, על מנת להרוויח כמה שיותר יתרונות מארכיטקטורה זו (כמו לדוגמא שינוי אחד מהצדדים מבלי להשפיע על האחר)

בכך אין מודים שונים (כמו בסימולטור Sinalgo) ובמידה ורוצים ליצור מוד חדש צריך לעשות שינויים לא קטנים.
כאן ישנו הסימולטור שהוא בצד ה Server side, לסימולטור יש API שכל Client באשר הוא (Desktop, Browser, Service וכו') יכולים לפנות אליו ולצרוך את המידע בצורה נוחה מבלי להפריע לסימולטור לרוץ.
בהנחה שלא מעוניינים כלל ב Client ניתן לדגום את ה outputs של האלגוריתם מהסימולטור בדומה למוד Batch.

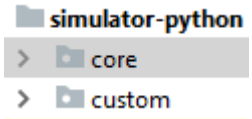
בפרק הזה נרחיב על הסימולטור, על יכולותיו, יתרונותיו וחסרונותיו מול Sinalgo

ארכיטקטורה

נרחיב כאן על ארכיטקטורה בצד השרת (מנוע הסימולטור),
הסימולטור נכתב בשפת Python עקב כך שהפיתוח בה מהיר, מספר שורות קוד מועט (ובכך יכול לפשט את מורכבות הפרויקט) ומגוון עשיר של ספריות צד שלישי שיכולות לעזור במידת הצורך.

ישנו דימיון בין הסימולטור Sinalgo לסימולטור הנוכחי מבחינת המחלקות השונות ודמיון זה נבע מכך שחלוקה זו היא די אינטואיטיבית.

תיקיית השורש מחולקות ל core ול custom כאשר core מכיל את הליבה של הסימולטור וה-custom מכיל את פיתוח המשתמש (אלגוריתמים שונים), נבין כל אחד מהחלקים השונים:



איור 14 היררכיה ראשונית של הפרוייקט

<u>נתמקד תחילה ב core:</u>

מכיל את מחלקות הבסיס (בדומה ל Sinalgo):

<u>Abstract custom global</u> – מחלקה המכילה מימוש default של הפונקציות pre_round, post_round אשר מופעלות לפני ואחרי הרצת אלגוריתם על ידי כל קדקודי הגרף בכל סיבוב

<u>Custom global manager</u> – המחלקה המספקת את ה custom global המתאים, כאשר אנו בוחרים פרויקט ה custom global של הפרויקט הנבחר ישמר במקום ה abstract custom global הדיפולטי, מחלקה זו גם אחראית לספק את ה custom global המתאים

<u>Edge</u> – מחלקה המייצגת אוביקט בגרף, המחלקה מכילה פונקציונליות כמו השמה של משקל על הצלע, ופרמטרים כמו משקל קדקוד מקור וקדקוד יעד בצלע

<u>Fifo_queue</u> – מכיל מימוש של תור לפי צרכינו (לדוגמא במימושים אחרים של תור ב python אין את הפונקציונליות של top)

<u>Inbox</u> – המכילה:
packets queue שהוא מופע של fifo_queue,
active packet שהינה ה packet הנוכחי שנקרא מה inbox
הפונקציונליות של הוספת packet,

פונקציה בוליאנית המחזירה האם יש עוד הודעות has next (בדומה לsinalgo),
גודל תור ההודעות,
שולח ההודעה הנוכחית ומקבל ההודעה הנוכחית

Message – מימוש אבסטרקטי של מחלקת הודעה, לא מכיל דבר נדרש לממש מחלקה היורשת ממנה

Node – מכיל :
מזהה קדקוד,
inbox עבור ההודעות
edges רשימה של כל הצלעות שהקדקוד מחובר אליהן
צבע הקדקוד
store לצורך אופטימיזציות של שליחת הודעות (2 הודעות לא ישלחו לאותו קדקוד באותו סיבוב, רק אחת תשלח)
פונקציונליות handle messages
הוספת צלע לקדקוד נתון
שידור הודעה לכל השכנים חוץ משכנים הנמצאים ב blacklist שמועבר במתודה זו
שליחת הודעה לשכן ספציפי
הוספת packet ל inbox
רישום קדקוד (קורה רק באתחול) למנוע הסימולטור
get color

packet – מכילה קדקוד יעד, מקור, אובייקט מסוג message ומשתנה המציג מאיזה סיבוב ה packet פעיל

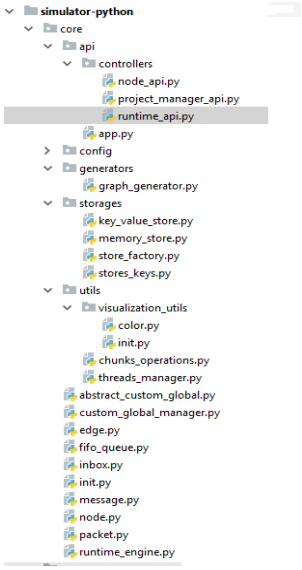
Runtime_engine – המחלקה העיקרית של הסימולטור, מכילה :
את מספר ה rounds
משתנה המציין האם הסימולטור בזמן ריצה (כל עוד משתנה זה פועל הסימולטור יריץ סיבובים בלי הפסקה)
משתנה המציין האם המנוע הופעל (על מנת לא להפעילו מספר פעמים במקביל באותו ה process)
מחיקת כל הקדקודים
הרצה רציפה של האלגוריתם (ללא הפסקה בין סיבובים)
הרצה של סיבוב אחד של האלגוריתם
הפסקה של סימולטור במידה והוא בזמן ריצה (לפי הפרמטר המציין האם הסימולטור בזמן ריצה)
מיפוי בין מזהה קדקוד ל id שלו לצורך אופטימיזציה

מכיל בנוסף פונקציות עזר כמו:

- מחלקת צבע (מכיל מספר צבעים)
- chunks operation המכיל פונקציונליות של הפרדת רשימה ל chunks
- threads manager פונקציה הניתנת לשימוש בתור decorator ונותנת את האפשרות להריץ מתודה בעזרת multithreads
- key value storage הנותן אפשרות לשמור מיפיים/ אובייקטים במידה וצריך (מכיל מימוש אחד כרגע של memory store אך ניתן להרחבה ולהוספה של stores נוספים ובחירתם על ידי הקונפיגורציה.
- generators מכיל graph generator הנותן אפשרות ליצירת גרף מכוון או לא מכוון ובחירת צלע בין שני קדקודים בהסתברות מסוימת שהמשתמש בוחר

מכיל בנוסף api המכיל את ה controllers הבאים:

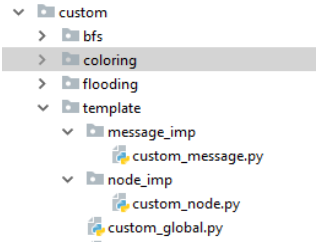
- Node api מכיל את ה api לקבלת הקדקודים של הגרף עם המידע הנלווה להם
- project manager api מכיל api לקבלת רשימת הפרויקטים שמומשו
- runtime api - מכיל api לקבלת מידע runtime כמו קבלת מספר הסיבובים שעברו באלגוריתם, invoke לפונקציה הראשית ב custom global של הפרויקט, invoke לפונקציה כלשהי ב custom global, עצירת הסימולטור, הרצה מתמשכת שלו והרצה של סיבוב בודד



איור 15 היררכיה מלאה של core

כעת נפרט את ה custom:

כאן למעשה אנו פותחים פרויקט בשם שנרצה (נוכל להעתיק לדוגמא את פרויקט template ואז לשנותו)
שתי המחלקות העיקריות הן custom_node ו custom_global שבדומה ל sinalgo כאן נוכל לרשמת ממחלקת node עבור custom node וממחלקת abstract custom global עבור custom global.
ב custom global נהיה חייבים לממש פונקציה main
ב custom node נצטרך לממש handle messages (אחרת זה ייקח את המימוש של האב שלא עושה דבר)



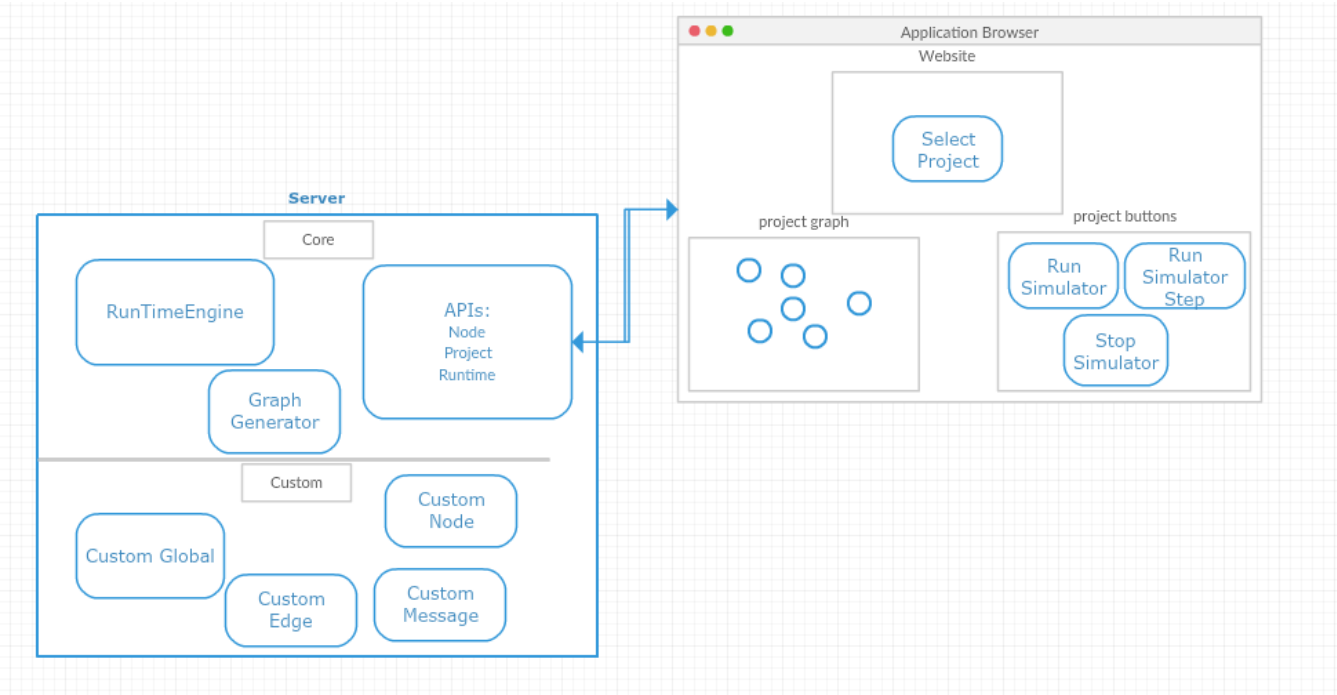
איור 16 היררכיה של custom

כעת לאחר שהצגנו את הקבצים/אובייקטים השונים בפרויקט ואת האחריות שלהם, נסביר איך הסימולטור עובד:

כפי שציינו כבר השוני כאן הוא ההפרדה בין הרצת המנוע לבין תצוגת הגרף, השוני נובע מההנחה שביצוע האלגוריתם בזמן טוב היא חשובה ביותר, אולם התצוגה שהמשתמש רואה היא בעדיפות פחותה.

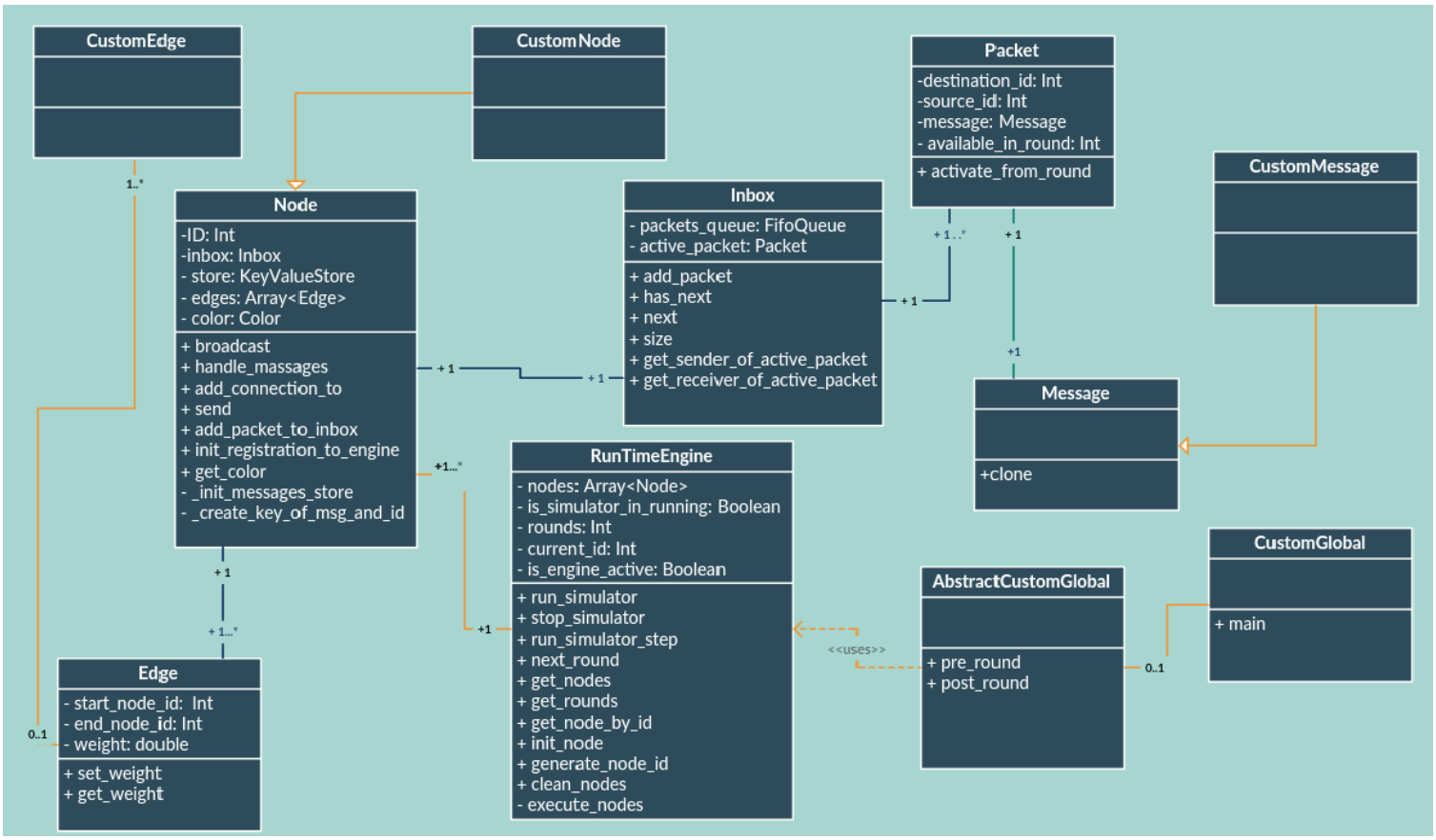
לכן בדומה לגישת MVC יצרנו צד server side המכיל את הסימולטור והמנוע הרץ מאחורי הקלעים
ובנוסף יצרנו client side בסיסי ב java script על מנת לקבל אב טיפוס של GUI.
נוכל לממש איזה GUI שנרצה כל עוד הוא יתמוך בתעבורת Http, שינוי זה נותן גמישות נוכחית ועתידית ב frameworks בין אם הם ברמת browser או ברמת desktop (כמו sinalgo).

נציג דיאגרמה של המערכת:



איור 17 חלוקת רכיבים מרכזיים בפרויקט

נציג דיאגרמה של מחלקות הגרף של הסימולטור :



איור 18 חלוקת מחלקות מרכזיות בפרויקט

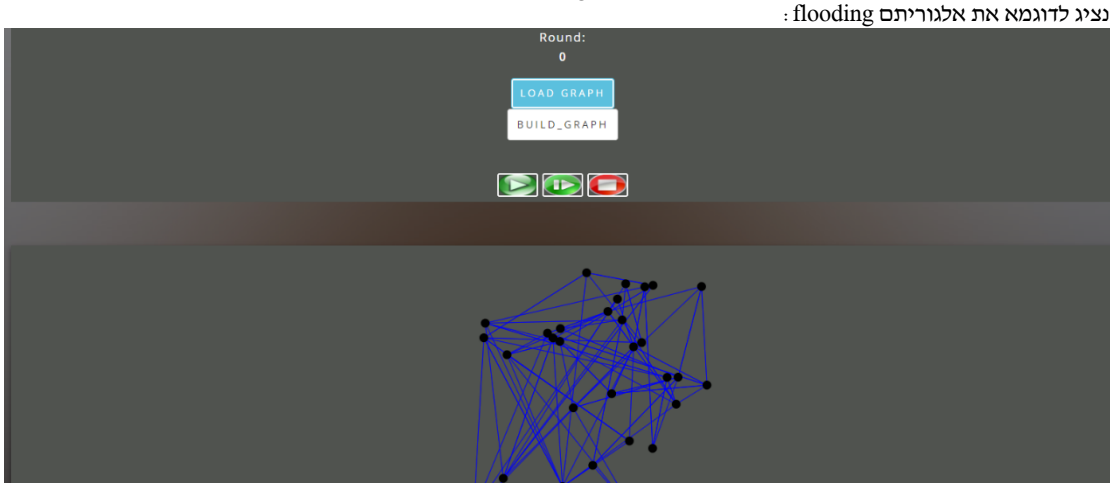
סדר הרצה:

כאשר אנו יוצרים פרויקט בנתיב המוגדר שם שלו יופיע בממשק ה GUI (ה API יחזיר אותו)
כאשר אנו מריצים סיבוב מתבצע:

- ה RunTimeEngine מריץ פונקציית pre_round
- עובר כל כל קדקוד (בצורה איטרבית או מקבילית)
  - כל קדקוד מריץ את הפונקציה handle_messages
- ה RunTimeEngine מריץ את הפונקציה post_round

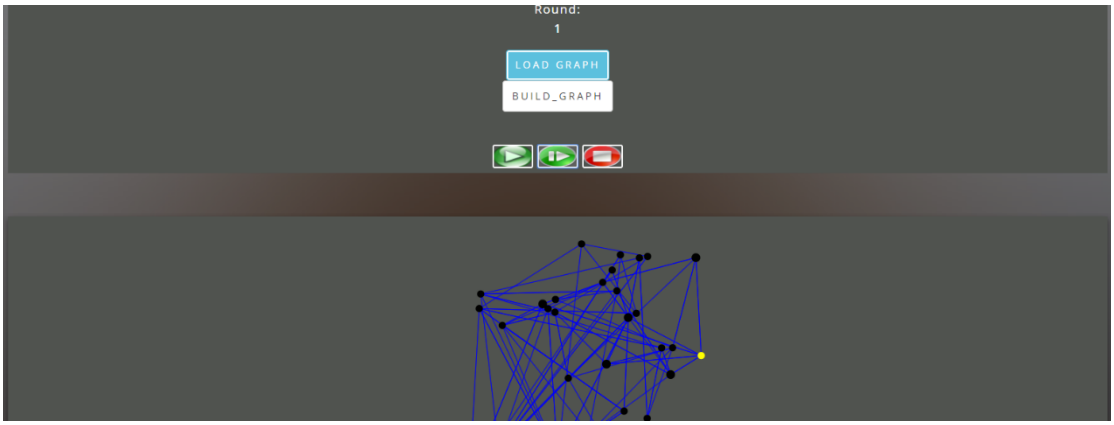
תצוגת GUI של הסימולטור

על מנת לתת את היתרון של GUI בעת הרצת הסימולטור כמו ב sinalgo, פותח ממשק בסיסי בשפת javascript המראה את הגרף ומציג את פעולותיו.
לצורך הצגת הגרף כאן נעשה שימוש ב framework של SigmaJS
נציג לדוגמא את אלגוריתם flooding:



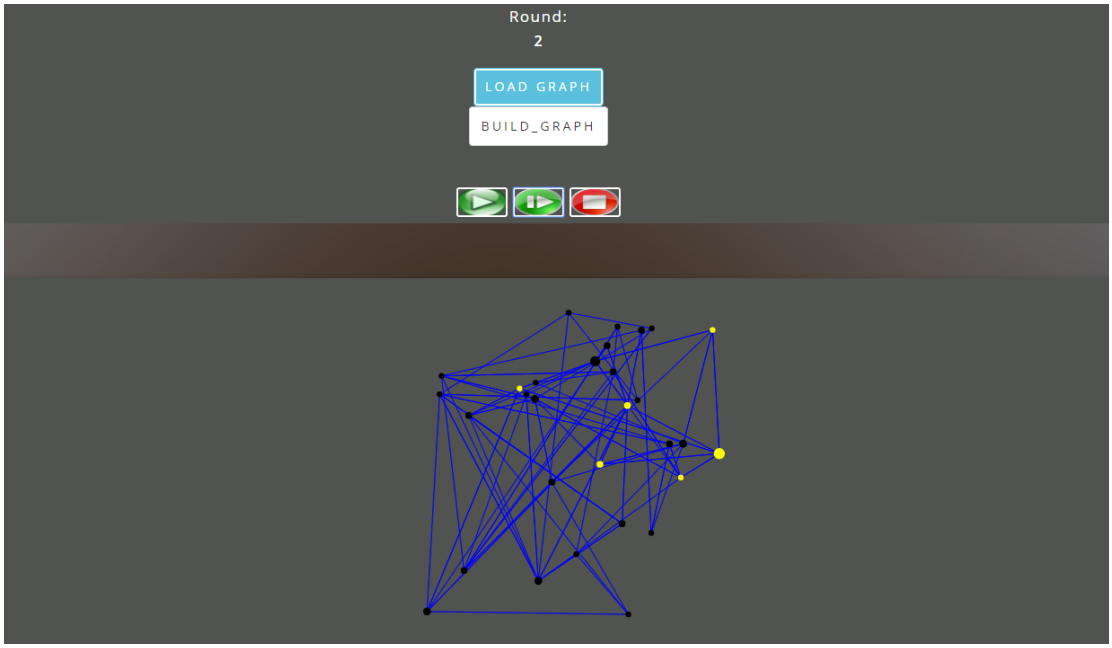
איור 19 הצגת גרף באלגוריתם flooding

כעת לאחר שנלחץ על הרצת סיבוב בודד (הכפתור האמצעי מבין 3 הכפתורים מצד שמאל) נקבל:



איור 20 הרצה 1 של אלגוריתם flooding

נוכל לראות כי הסיבוב עלה ב -1, מודגש כי קדקוד ראשון נצבע בצהוב וכעת ישלח הודעות לכל שכניו, נריץ סיבוב נוסף ונקבל צביעה של כל השכנים של הקדקוד הצבוע:

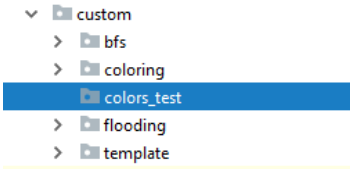


איור 21 הרצה 2 של אלגוריתם flooding

כתיבת פרויקט

על מנת להתחיל בכתיבת הפרויקט נבצע את השלבים הבאים:

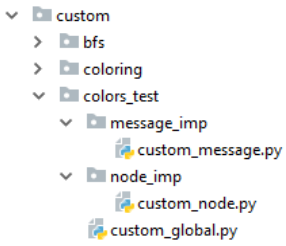
- ניצור תקייה חדשה תחת custom עם שם הפרויקט שנרצה (לצורך הדוגמא נקרא לפרויקט colors_test):



איור 22 פרויקט colors_test

- תחת התיקייה החדשה:
  - באותו נתיב ניצור את הקובץ custom_global.py
  - ניצור תיקייה node_imp
    - תחת תיקייה זו ניצור את הקובץ custom_node.py
  - ניצור תיקייה message_imp
    - תחת תיקייה זו ניצור את הקובץ custom_message.py

כעת היררכיית התיקיות והקבצים תראה כך:



איור 23 מבנה התיקיות והקבצים בפרויקט colors_test

- כפי שעשינו בדוגמת הפרויקט של Sinalgo ניצור גרף עץ בתיקיית custom_global:

```python
class CustomGlobal(AbstractCustomGlobal):
    def main(self):
        self.build_tree(6, 15)

    def build_tree(self, max_degree, leaves_num):
        leaves = []
        for i in range(leaves_num):
            leave = CustomNode()
            leave.init_registration_to_engine()
            leaves.append(leave)

        level = 0
        child_pos_in_lvl = 0
        current_level_nodes = []
        prev_level_nodes = leaves

        while len(prev_level_nodes) > 1:
            node = CustomNode()
            node.init_registration_to_engine()
            degree = random.randint(1, max_degree)
            limit = degree if degree < len(prev_level_nodes) - child_pos_in_lvl else len(prev_level_nodes) - child_pos_in_lvl
            for i in range(limit):
                prev_level_nodes[child_pos_in_lvl].parent = node
                node.add_connection_to(prev_level_nodes[child_pos_in_lvl])
                node.childs.append(prev_level_nodes[child_pos_in_lvl])
                child_pos_in_lvl +=1
            current_level_nodes.append(node)
            if(child_pos_in_lvl == len(prev_level_nodes)):
                level +=1
                prev_level_nodes = current_level_nodes
                current_level_nodes = []
                child_pos_in_lvl = 0
```

איור 24 מחלקת customGlobal

- ניצור custom_message בסיסי בקובץ custom_message.py:

```python
class MarkMessage(Message):
    def __init__(self, msg):
        self.msg = msg
```

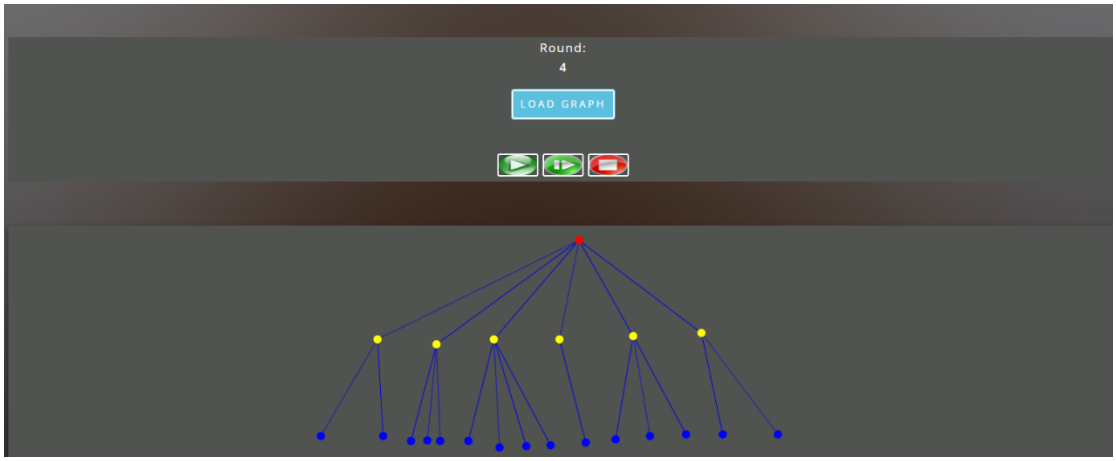
איור 25 מחלקת MarkMessage

- נממש את הפונקציה handle_messages בקובץ custom_node.py:

```python
class CustomNode(Node):
    def __init__(self):
        Node.__init__(self)
        self.base_color = BLACK
        self.color_A = YELLOW
        self.color_B = BLUE
        self.root_color = RED
        self.childs = []
        self.parent = None

    def handle_messages(self):
        if self.get_color() == self.base_color:
            if self.parent is None:
                self.color = self.root_color
                self.broadcast(MarkMessage(self.color_A))
            elif self.inbox.has_next():
                msg = self.inbox.next()
                self.color = msg.msg
                color_for_sending = self.color_B if msg.msg == self.color_A else self.color_A
                self.broadcast(MarkMessage(color_for_sending))
```

איור 26 מחלקת CustomNode

- ונקבל:



איור 27 צביעת עץ בשני צבעים וצבע נוסף עבור השורש

יצירת כפתור מתוך הקוד

כמו ב Sinalgo גם בסימולטור שלנו ניתן לייצר כפתור דרך הקוד ולהטריגו ממסך ה GUI

נציג כיצד הדבר מתבצע:

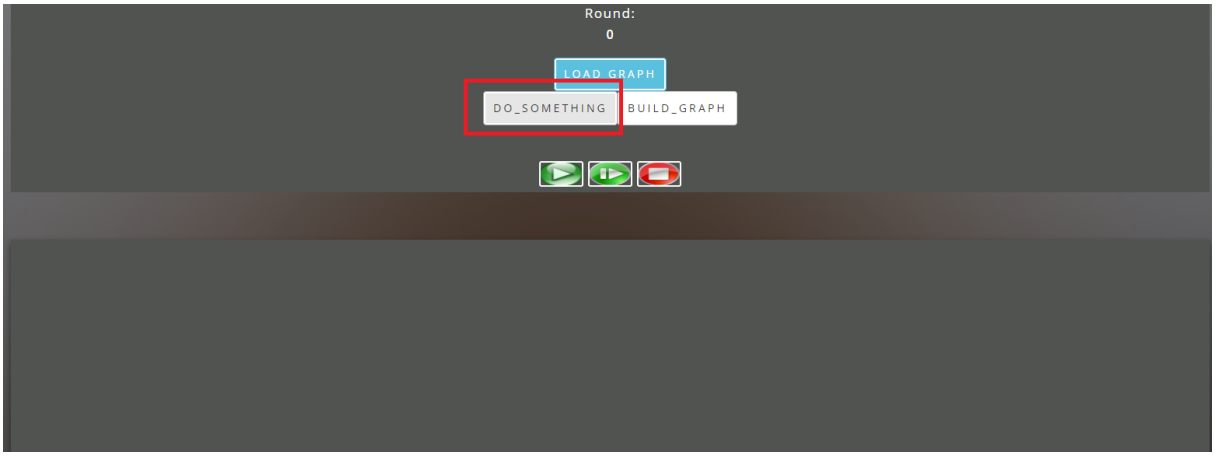
- בפרויקט מסוים נכנס למחלקת CustomGlobal
- נכתוב את הפונקציה אותה אנו רוצים להטריג על ידי כפתור
- נשים Decorator הנקרא `@store_method(...)`
  - יקבל את שמות הפרמטרים ותיאורם שנמצאים ב Inputs של הפונקציה, לדוגמא:

```
@store_method(nodes_count="The number of nodes", prob="probability of edge between 2 nodes")
def build_graph(self, nodes_count, prob):
```

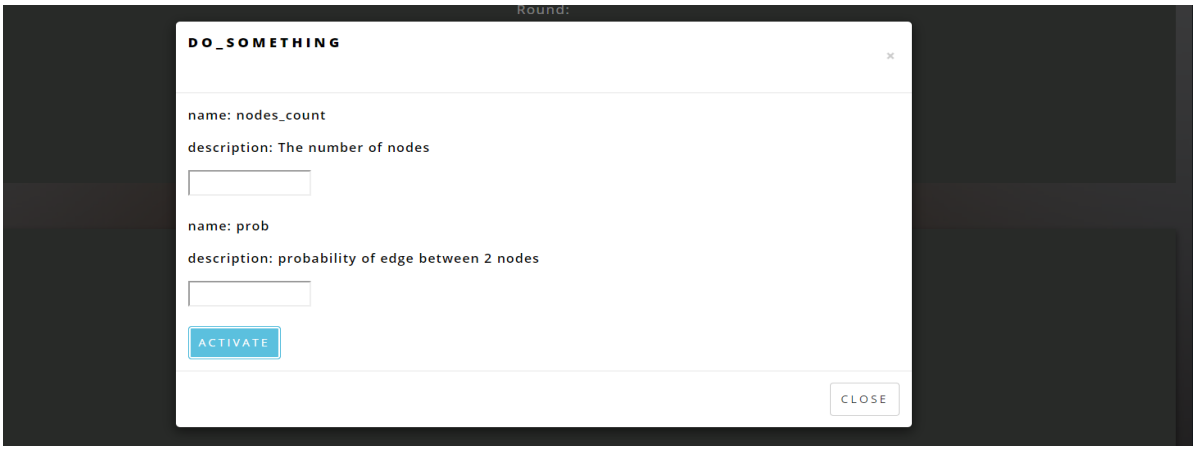
- הכפתור יופיע במסך ה GUI

```
@store_method(nodes_count="The number of nodes", prob="probability of edge between 2 nodes")
def do_something(self, nodes_count, prob):
    print("Hello World")
```

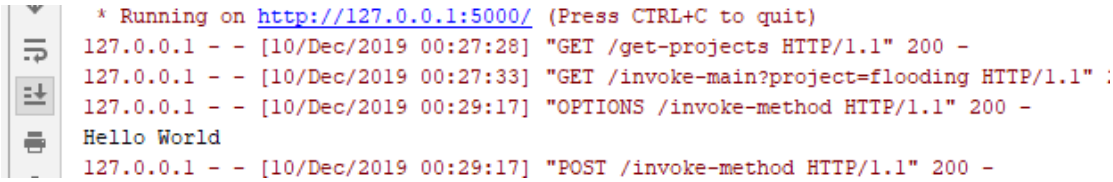
איור 28 decorator עוטף פונקציה שתופיע בתור כפתור



איור 29 מסך GUI עם הכפתור שנוצר



איור 30 מסך מודל נפתח עם הפרמטרים שהועברו ב Decorator



איור 31 console שבו רואים שהפונקציה אכן נקראה כאשר מפעילים אותה ממסך ה GUI

יתרונות מול Sinalgo

- פשטות בפיתוח – יתרונות של python מבחינת כמות קוד וכו'

- גמישות של מודל MVC – במודל זה ישנה הפרדה בין ממשק ה GUI לבין הגדרת הלוגיקה והמודל, בכך נוכל להחליף כל אחד מהחלקים מבלי לפגוע באחר.
בדוגמא שלנו אנחנו יכולים לעבור לממשק GUI אחר (להחליפו ב framework נוכחי ועתידי שעונה טוב יותר לתצוגת גרפים) בצורה נוחה מבלי לפגוע בסימולטור, לעומת Sinalgo שהממשק הוא חלק ממנו ולכן הפרדה כזו היא מורכבת
בנוסף נוכל גם לבזר כוחות ולהעביר את ממשק ה GUI או את הסימולטור לשרת ייעודי (אין צורך שהרכיבים ישבו באותו השרת ויעמיסו עליו. זהו **יתרון משמעותי**

- יכולת MultiThreading – הסימולטור יכול להיות מבוסס על MultiThreading (בהמשך נבצע בדיקה אם זה עוזר או לא, אבל היכולת קיימת)

- תצוגה של גרף קבועה – בסימולטור נוכל לקבוע את הגרף שהמשתמש יקבל, זאת כיוון שאין טעם להציג למשתמש גרף עמוס וגדול יתר על המידה, ב Sinalgo זה אינו אפשרי ולכן כל הגרף יוצג (כבד מבחינת משאבים)

- פונקציוליות נוספת – בסימולטור הוספנו פונקציות עזר היכולות לתרום למהירות הפיתוח, כמו לדוגמא יצירת גרף לפי מספר קדקודים והסתברות לצלע בין כל שני קדקודים

- שיפור יעילות בשליחת הודעות – בסימולטור שלנו כאשר באותו סיבוב קדקוד שולח יותר מהודעה אחת זהה לעבר קדקוד מסוים, עבור אותו קדקוד ההודעות הזהות יאוחדו להודעה אחת.
בכך לא יהיו הודעות כפולות זהות באותו סיבוב וזה יחסוך לקדקוד היעד בסיבוב הבא מעבר על אותה הודעה שהגיעה באותו קדקוד שוב ושוב

- הסימולטור הינו Web server ולכן ניתן לתשאל מידע (לדוגמא כמות קדקודים או מספר סיבובים) בכל זמן אפילו בזמן ריצת הסימולטור

- עקב ההגבלה בגודל הגרף, הסימולטור יוכל להציג גרפים גדולים יותר, ב Sinalgo אין הגבלה כפי שתארנו ולכן עבור גרף גדול מאוד יוקצה זיכרון רב לצורך ציור כל הרכיבים (קדקודים, צלעות) ואילו בסימולטור שלנו יש יכולת להציג גרף בגודל קבוע על מנת לתת למשתמש אינדיקציה אולם מאוחרי הקלעים הסימולטור יעבוד על הגרף העצום ויבצע את האלגוריתם כנדרש.

חסרונות מול Sinalgo

- זמן ריצה – Sinalgo כתוב ב JAVA שעל פי בדיקות benchmark היא מהירה יותר במספר מקרים מאשר CPython

- יכולות רבות – Sinalgo תומך במספר סוגי גרפים, ובמספר מודים בעוד שהסימולטור בצורתו הנוכחית לא תומך בכל כך הרבה יכולות.

השוואת ביצועים Python-Simulator VS Sinalgo

תנאים מקדימים של סביבת ההשוואה:

- מחשב נייד אישי
- מעבד Intel core i5 1.7GHz (מעבד מבוסס 64x)
- כרטיס RAM 4.00 GB
- Python 3.6
- Python implementation: CPython
  - ניתן לבדוק את המימוש כך:

```
import sys
print(sys.implementation.name)
```

- החישובים נלקחים עם סביבות העבודה Eclipse, PyCharm (כאשר ב pycharm נבצע חיסור בין ה ram שנקלח ללא פעולה לבין ה ram שנקלח בזמן הרצה, ב eclipse לא נצטרך לעשות זאת כיוון שנפתח תהליך נפרד בעת ההרצה ואותו אנחנו יכולים לנטר)

הנחות:

- החישוב הינו בממוצע על צריכת משאבים
- החישובים ייעשו על גרפים בגודל 100, 500, 1,000, 3,000
- החישובים ייעשו על גרפים לא מכוונים
- ההסתברות לצלע בין 2 קדקודים הינה 0.08
(ניקח בחשבון שזהו גרף לא מכוון ולכן צלע בין 2 קדקודים תוכל להתקיים עבור הגרלה של הקדקוד הראשון ולאחר מכן הגרלה של הקדקוד השני)

<u>הערה:</u> בהמשך נראה כי מבחינת זמני ריצה לרב Sinalgo יהיה מהיר יותר, אולם "יסבול" מבעיות אחרות כמו Memory, אנו נבין בפרקים הבאים סיבות לתוצאה.

אלגוריתם Flooding

אלגוריתם Flooding הינו אלגוריתם שידור הודעות לשאר הקדקודים בגרף המתחיל מקדקוד ספציפי.
הקדקוד הנבחר v שולח הודעה לכל שכניו, כל שכן המקבל את ההודעה מעביר אותה לכל שכניו פרט לשולח ההודעה.
בהינתן תוספת שבה כל קדקוד שקיבל את ההודעה לראשונה מסמן את השולח כאביו, אנו נקבל עץ פורש של הגרף.
יעילותו של האלגוריתם הינה O(Diam(G))

האלגוריתם ממומש בשני הסימולטורים באופן הבא:

כל קדקוד מריץ את הפונקציה handleMessages בכל סיבוב
נתאר מה הפונקציה עושה:

- הקדקוד v מתחיל בסיבוב הראשון (קדקוד זה מסומן כ isRoot == True) שולח הודעה לכל שכניו ונצבע בצהוב
- כל קדקוד אחר שאינו קדקוד התחלה, קורא את ההודעה הראשונה שקיבל במידה ויש, שולח אותה לשאר שכניו, מסמן את שולח ההודעה כאביו וצובע את עצמו בצהוב
- קדקוד שכבר סימן את אביו לא מבצע דבר

מימוש ב Sinalgo

נציג את מימוש מסוים של האלגוריתם בסימולטור Sinalgo:

```
public class Fnode extends BaseNode {

    private int parentId = -1;
    public boolean isRoot = false;

    public Fnode() {
    }

    @Override
    public void handleMessages(Inbox inbox) {
        if (this.isRoot && CustomGlobal.currentRound == 1) {
            this.setColor(Color.YELLOW);
            this.sendMsgToAllNeighbors("", new ArrayList<Integer>());
        } else {
            if (this.parentId == -1 && this.isRoot == false) {
                if (inbox.hasNext()) {
                    Message msg = inbox.next();
                    Node sender = inbox.getSender();
                    this.parentId = sender.ID;
                    this.setColor(Color.YELLOW);
                    this.sendMsgToAllNeighbors("aa", new ArrayList<Integer>());
                }
            }
        }
    }
```

איור 32 מימוש אלגוריתם Flooding ב sinalgo

מימוש ב Python-simulator

נציג את מימוש מסוים של האלגוריתם בסימולטור python:

```
class CustomNode(Node):
    def __init__(self):
        Node.__init__(self)
        self.is_root = False
        self.parent = None

    def handle_messages(self):
        if self.is_root and RunTimeEngine.getInstance().rounds == 1:
            self.color = YELLOW
            self.broadcast(MarkMessage("hello world"))
        else:
            if(self.parent == None and self.is_root is False):
                if self.inbox.has_next() == True:
                    self.color = YELLOW
                    msg = self.inbox.next()
                    self.parent = self.inbox.get_sender_of_active_packet().ID
                    print ("node:{} recieved msg".format(self.ID))
                    self.broadcast(msg)
```

השוואה בין הסימולטורים

ההשוואה של האלגוריתם Flooding תהיה מורכבת מהפרמטרים הבאים:

- זמן ריצה בפועל (seconds)
- CPU (אחוזים)
- RAM (MB)
- Delta של הגרף

גודל גרף: 100

Python-simulator	Sinalgo	
0.25	0.3	זמן ריצה
65	68	CPU
10(pychram) +30(chrome)	90	RAM
27	23	Delta

גודל גרף: 500

Python-simulator	Sinalgo	
5.8	1.3	זמן ריצה
63	56	CPU
50(pychram) +45(chrome)	115	RAM
114	105	Delta

גודל גרף: 1000

Python-simulator	Sinalgo	
23.6	5.5	זמן ריצה
41	43	CPU
120(pychram) +47(chrome)	207	RAM
209	189	Delta

גודל גרף: 3000

Python-simulator	Sinalgo	
501	73	זמן ריצה
44	50	CPU
860(pychram) +47(chrome)	837	RAM
594	527	Delta

אלגוריתם Shortest Paths

אלגוריתם Shortest Paths מחשב את המסלולים הקצרים ביותר מכל קדקוד בגרף לכל קדקוד אחר בגרף.
האלגוריתם עובד כמו אלגוריתם BFS הרגיל רק שאנו מריצים אותו מכל קדקוד בגרף,
כל קדקוד מקור השולח את ההודעה הראשונה מכניס לתוכה את המזהה שלו, כל קדקוד אחר המקבל את ההודעה הזו במהלך הסיבובים יכיל טבלת ניתוב שבה יסמן את האב עבור המסלול הספציפי של אותו קדקוד מקור.
יעילותו של האלגוריתם הינה O(Diam(G)) כיוון שכל הקדקודים מריצים במקביל את האלגוריתם

האלגוריתם ממומש בשני הסימולטורים באופן הבא:

כל קדקוד מריץ את הפונקציה handleMessages בכל סיבוב
נתאר מה הפונקציה עושה:

- בסיבוב הראשון כל קדקוד (שבסיבוב זה נחשב קדקוד מקור) שולח הודעה המכילה String ואת מספר הזיהוי שלו על מנת שכל מקבל ההודעה יידע כי הוא מקור המסלול)
- בסיבובים לאחר מכן כל קדקוד המקבל הודעה מחלץ את קדקוד המקור של המסלול ואת האב ששלח את ההודעה
- לכל קדקוד יש טבלת ניתוב, במידה ואין בטבלה key של קדקוד ראש המסלול יוצר אחד ושומר את הקדקוד ששלח את ההודעה כאביו במסלול זה
- במידה ויש בטבלת הניתוב כבר key, לא מבצע דבר

מימוש ב Sinalgo

נציג את מימוש מסוים של האלגוריתם בסימולטור Sinalgo:

```
public class BfsNode extends BaseNode {

    public HashMap<Integer, Integer> parentPerPathRootId = new HashMap<>();

    public BfsNode() {
    }

    @Override
    public void handleMessages(Inbox inbox) {
        if (CustomGlobal.currentRound == 1) {
            this.sendMsgToAllNeighbors("", this.ID, new ArrayList<Integer>());
        } else {
            while (inbox.hasNext()) {
                StrMessage msgObj = (StrMessage) inbox.next();
                int rootId = msgObj.senderId;
                int senderId = inbox.getSender().ID;
                if (this.parentPerPathRootId.containsKey(rootId) == false) {
                    ArrayList<Integer> blacklist = new ArrayList<>();
                    blacklist.add(rootId);
                    blacklist.add(senderId);
                    this.parentPerPathRootId.put(rootId, senderId);
                    this.sendMsgToAllNeighbors(msgObj.msg, rootId, blacklist);
                }
            }
        }
    }
```

איור 34 מימוש אלגוריתם Shortest Paths ב Sinalgo

מימוש ב Python-simulator

נציג את מימוש מסוים של האלגוריתם בסימולטור python:

```python
class CustomNode(Node):
    def __init__(self):
        Node.__init__(self)
        self.parent_per_path_root_id = {}

    def handle_messages(self):
        if RunTimeEngine.getInstance().rounds == 1:
            print("node: {} starting".format(self.ID));
            self.broadcast(MarkMessage("", self.ID))
        else:
            while self.inbox.has_next() == True:
                msg = self.inbox.next()
                root_id = msg.path_root_id
                if self.parent_per_path_root_id.get(root_id) is None:
                    self.parent_per_path_root_id[root_id] = self.inbox.get_sender_of_active_packet().ID
                    print("node: {} recieve msg from node: {}".format(
                        self.ID, self.inbox.get_sender_of_active_packet().ID)
                    )
                    self.broadcast(msg, [root_id, self.inbox.get_sender_of_active_packet().ID]);
```

איור 35 מימוש אלגוריתם Shortest paths ב python simulator

השוואה בין הסימולטורים

ההשוואה של האלגוריתם Shortest Paths תהיה מורכבת מהפרמטרים הבאים:

- זמן ריצה בפועל (seconds)
- CPU (אחוזים)
- RAM (MB)
- Delta של הגרף

➔ סימון זה מציין שהתהליך שמריץ את הסימולטור קרס על שגיאת זיכרון, הכוונה כאן שהסימולטור צרך יותר מידי זיכרון (יותר מאשר מתאפשר לתהליך המריץ אותו לצרוך) זאת כיוון שהסימולטור צורך המון זיכרון גם מבחינת המידע שהוא מכיל וגם מבחינת ה GUI אותו הוא מנהל באותו התהליך.
לכן התהליך נעצר והיה צריך לסגור אותו, בכך למעשה אנו לא יכולים לראות את תוצאות הסימולטור.

גודל גרף: 100

	Sinalgo	Python-simulator
זמן ריצה	1.45	25.5
CPU	68	61
RAM	160	100(pychram) +30 (chrome)
Delta	24	26

גודל גרף : 200

Python-simulator	Sinalgo	
214.5	5.9	זמן ריצה
50	66	CPU
350(pychram) +45(chrome)	530	RAM
51	47	Delta

גודל גרף : 500

Python-simulator	Sinalgo	
3054		זמן ריצה
39		CPU
780(pychram) +52(chrome)		RAM
115		Delta

גודל גרף : 1000

Python-simulator	Sinalgo	
Too long		זמן ריצה
41.5		CPU
1.5G+(pychram) +50 (chrome)		RAM
210		Delta

גודל גרף : 3000

Python-simulator	Sinalgo	
		זמן ריצה
		CPU
		RAM
		Delta

אלגוריתם Random Vertex Coloring

אלגוריתם Random Vertex Coloring הינו אלגוריתם רנדומי לצביעת קדקודים.
האלגוריתם רץ בכל אחד מהקדקודים, בהנחה שצבעו אינו נכון (יש לו שכן עם צבע כמו שלו) יבחר צבע חדש בטווח 1..delta+1
מספר הצבעים שהגרף ייצבע בהם הוא delta+1
נשים לב שהאלגוריתם יעבוד כאשר ישנו גרף שכל קדקוד בו עם צבע שונה,
בכך עבור קלט גרף כזה בסיום האלגריתם נקבל אותו מספר צבעים, זה לא מה שאנו רוצים ולכן נחליט כי לא משנה איזה גרף קלט נקבל, כל הקדקודים בגרף זה יהיו בצבע אחיד (Default) וכמובן שצביעה זו היא אינה צביעה חוקית, לאחר הרצת האלגוריתם נקבל צביעה חוקית בלכל היותר delta+1 צבעים.

האלגוריתם ממומש בשני הסימולטורים באופן הבא:

כל קדקוד מריץ את הפונקציה handleMessages בכל סיבוב
נתאר מה הפונקציה עושה:

- בכל שלב של האלגוריתם, קדקוד שיש לו לפחות שכן אחד עם צבע כמו שלו ייחשב כפעיל
- כל קדקוד פעיל יגריל בהסתרות 1/2 האם הוא ינסה למצוא צבע חדש
- במידה ויצא שהוא צריך למצוא צבע חדש, יבחר צבע מהטווח 1...delta +1
- בהנחה שהצבע החדש שונה מצבע כל שכניו קדקוד זה ייצבע בצבע החדש ויהפוך ללא פעיל

מימוש ב Sinalgo

נציג את מימוש מסוים של האלגוריתם בסימולטור Sinalgo (כאן נניח כי הצבע הינו ID וכל הקדוקדים מתחילים עם אותו ה ID):

```
public class Cnode extends BaseNode {

    public Cnode() {

    }

    @Override
    public void handleMessages(Inbox inbox) {
        if (this.isDifferFromAllNeighbors(this.ID) == false) {
            if ((Math.random() <= 0.5) == true) {
                int newColor = this.getRandom(1, CustomGlobal.delta + 1);
                if (this.isDifferFromAllNeighbors(newColor)) {
                    this.ID = newColor;
                }
            }
        }
    }

    public boolean isDifferFromAllNeighbors(int currColor) {
        ArrayList<BaseNode> neighbors = this.getNeighbors();
        ArrayList<Integer> colors = new ArrayList<>();
        for (BaseNode node : neighbors) {
            colors.add(node.ID);
        }
        return colors.contains(currColor) ? false : true;
    }
```

מימוש ב Python-simulator

נציג את מימוש מסוים של האלגוריתם בסימולטור python:

```
class CustomNode(Node):
    graph_delta = 0
    colors_mapping = {}

    def __init__(self):
        Node.__init__(self)
        self.graph_delta = 0
        self.colors_mapping = {}

    def handle_messages(self):
        if self.is_different_from_neighbors(self.get_color()) is False:
            if random.random() <= 0.5:
                new_color = random.randint(1, self.graph_delta + 1)
                if self.is_different_from_neighbors(new_color):
                    self.color = self.colors_mapping[new_color]

    def get_neighbors(self):
        neighbors_array = []
        for neighbor in self.edges:
            neighbors_array.append(neighbor.end_node_id)
        return neighbors_array

    def is_different_from_neighbors(self, color):
        neighbors_array = self.get_neighbors();
        get_color_from_nodes = lambda neighbor: RunTimeEngine.getInstance().get_node_by_id(neighbor).get_color()
        neighbors_colors_map = map(get_color_from_nodes, neighbors_array)
        colors = list(neighbors_colors_map)
        if color in colors:
            return False;
        return True;

    def generate_colors(self):
        for i in range(self.graph_delta + 5):
            r = lambda: random.randint(0, 255)
            self.colors_mapping[i] = '#%02X%02X%02X' % (r(), r(), r())
```

איור 37 מימוש אלגוריתם Random Vertex Coloring ב Python simulator

השוואה בין הסימולטורים

ההשוואה של האלגוריתם Shortest Paths תהיה מורכבת מהפרמטרים הבאים :

- זמן ריצה בפועל (seconds)
- CPU (אחוזים)
- RAM (MB)
- Delta של הגרף
- מספר סיבובי ריצה

גודל גרף: 100

	Sinalgo	Python-simulator
זמן ריצה	0.25	0.14
CPU	66	60
RAM	130	17(pychram) +22(chrome)
Delta	27	24
סיבובי ריצה	10	8

גודל גרף : 500

	Sinalgo	Python-simulator
זמן ריצה	2.7	1.8
CPU	45	40
RAM	130	40(pychram) +33(chrome)
Delta	103	111
סיבובי ריצה	21	23

גודל גרף : 1000

	Sinalgo	Python-simulator
זמן ריצה	16.4	6.6
CPU	42	41
RAM	240	92(pychram) +32(chrome)
Delta	192	216
סיבובי ריצה	32	25

גודל גרף : 3000

	Sinalgo	Python-simulator
זמן ריצה	538	210
CPU	46	42.5
RAM	450	340(pychram) +44(chrome)
Delta	528	598
סיבובי ריצה	29	56

רקע

בפסקה הקודמת הרצנו את הסימולטור מבוסס על multi treaded, זאת אומרת שבתהליך אנו יוצרים מספר threads על מנת שכל אחד יעשה יחידת עבודה בצורה מבוזרת ויבצע זאת מהר יותר

לעומת Sinalgo הסימולטור שלנו תומך ב multi treaded באופן די נוח.
את הפונקציונליות הזו הוצאנו ל Decorator שהוא מעין מעטפת לפונקציה הנקראת :

```python
@staticmethod
@use_threads
def execute_nodes(nodes):
    for node in nodes:
        node.handle_messages()
```

```python
def collect_active_threads(threads):
    for thread in threads:
        thread.join()

def get_threads_num(collection):
    threads_num = int(NUMBER_OF_THREADS)
    if threads_num > len(collection):
        threads_num = len(collection)
    return threads_num

def use_threads(func):
    def inner(*args, **kwargs):
        threads_list = []
        collection = args[0]
        chunks = list(split_to_chunks(collection, int(len(collection) / get_threads_num(collection))))
        for chunk in chunks:
            thread = Thread(target=func, args=(chunk,))
            threads_list.append(thread)
            thread.start()
        collect_active_threads(threads_list)
    return inner
```

איור 38 מימוש חלוקה ל Threads בתור Decorator

כפי שאנו ראינו כבר במחלקת RuntimeEngine ישנה פונקציה שנקראת בכל סיבוב של האלגוריתם,
פונקציה זו הינה execute_nodes שרצה על כל הקדקודים ומריצה את הפונקציה handle_messages שאנו מממשים.

כאשר לעומת Sinalgo שאכן רץ בצורה איטרטיבית ב thread אחד, כאן ממומשת הרחבה לריצה ב thread אחד, הרחבה זו נותנת לנו את היכולת להריץ במספר threads שנקבע בקונפיגורציה.

הפונקציה הראשית לצורך ניהול ה threads הינה use_threads שמקבל את הפונקציה שאותה היא תריץ במספר ה threads שנקבע.
הפונקציה מחלקת את ה collection שברצוננו לעבור עליו למספר chunks שכל אחד ירוץ במקביל על ידי thread אחר (במקרה הזה ה collection הוא כל ה nodes).

לאחר מכן אנו יוצרים מערך של threads פעילים, על כל chunk יוצרים thread ייעודי ומתחילים אותו.
על מנת שלא נסיים סיבוב לפני שחלק מה nodes סיימו את פעולתם, אנו מפעילים את הפונקציה collect_active_threads,
פונקציה זו מקבלת מערך של threads ועבור כל אחד מבצעת join ובכך ה thread הראשי לא ימשיך הלאה עד שכל השאר לא יסיימו.

השוואה

נרצה לבצע השוואה ולראות האם הסימולטור שלנו במצב Multi threaded יהיה מהיר יותר ממצב רגיל ואם כן בכמה.

על מנת לשנות את מספר הטריידים במערכת נשנה את הפרמטר NUMBER_OF_THREADS בקובץ config.py בנתיב core/config/config.py

לכן ניקח את האלגוריתמים שנבדקו ונציג טבלת ביצועי זמן ריצה בגרפים בגדלים שונים.
נשווה את הסימולטור שלנו במצב של single threaded למצב של multi threaded

אלגוריתם Flooding

גודל גרף : 500

	Ps-multi (6 threads)	Ps-multi (3 threads)	Ps-single
זמן ריצה	6.10	6.12	5.9
Delta	116	112	110

אלגוריתם Shortest Paths

גודל גרף : 100

	Ps-multi (6 threads)	Ps-multi (3 threads)	Ps-single
זמן ריצה	27.42	26.94	26.14
Delta	27	27	24

אלגוריתם Coloring

גודל גרף : 1000

	Ps-multi (6 threads)	Ps-multi (3 threads)	Ps-single
זמן ריצה	10.31	7.89	7.46
Delta	206	210	213

נוכל לראות כי Multi Threaded לא שיפר את הביצועים כפי שציפינו.
כפי שראינו מימוש Python שלנו הינו CPython,
במימוש זה ישנו מנגנון הנקרא GIL (Global Interpeter Lock) או בשם אחר Mutex
מנגנון זה מריץ את ה threads באופן איטרטיבי ולא מקבילי ולכן אין שיפור בביצועים ולעיתים אפילו החמרה (threads מבזבזים משאבים במקרה הזה ללא תרומה)

מסקנות

לאחר ניתוח זמני ריצה של מספר אלגוריתמים בשני הסימולטורים נוכל לטעון כי:

- Sinalgo דורש זיכרון רב, דבר הגורם לבעיה כאשר רוצים להריץ אלגוריתם כבד על גרף גדול
- Sinalgo מהיר יותר ברב המקרים בהנחה שההרצה אפשרית ואינו קורס (עבור האלגוריתם האחרון נוכל להניח שהשוני נבע מתלות במימוש או מצריכת משאבים ספציפית עבור אותו אלגוריתם)
  - בחישוב ממוצע משוערך באלגוריתם Flooding זמן הריצה של Sinalgo הוא כ 20 אחוזים מזמן ריצת האלגוריתם של Python-Simulator
  - בחישוב ממוצע משוערך באלגוריתם Shortest Paths זמן הריצה של Sinalgo הוא כ 0.04 אחוזים מזמן ריצת האלגוריתם של Python-Simulator (המדגם היה קטן עקב קריסת האלגוריתם בסימולטור Sinalgo)
  - בחישוב ממוצע משוערך באלגוריתם Random Vertex Coloring זמן הריצה של Sinalgo **איטי** יותר וכי זמן ריצתו של Python-Simulator הוא כ 0.5 אחוזים מזמן ריצתו של Sinalgo
- Python-simulator גמיש יותר מהבחינה שישנה הפרדה בין החלקים השונים (client, sever)
- Python-simulator משתמש ב Multi Threads אולם הדבר אינו משפר את זמני הריצה עקב GIL
- שני הסימולטורים זהים פחות או יותר מבחינת צריכת CPU עם יתרון קל ל python-simulator
- Python-simulator צורך פחות RAM מ sinalgo גם בהתחשב בזכרון ש chrome צורך עבור הצגת הגרף בדפדפן
  - בחישוב ממוצע משוערך באלגוריתם Flooding הצריכה של Python-Simulator כולל צריכת הזיכרון של הדפדפן היא כ 80 אחוזים מהצריכת זיכרון של Sinalgo
  - בחישוב ממוצע משוערך באלגוריתם Shortest Paths הצריכה של Python-Simulator כולל צריכת הזיכרון של הדפדפן היא כ 77 אחוזים מהצריכת זיכרון של Sinalgo
  - בחישוב ממוצע משוערך באלגוריתם Random Vertex Coloring הצריכה של Python-Simulator כולל צריכת הזיכרון של הדפדפן היא משתנה אבל נעה בין 0.4-0.8 אחוזים מהצריכת זיכרון של Sinalgo

אז מדוע Sinalgo לרב ניצח מבחינת זמן ריצת האלגוריתם?

זה נובע בין היתר מהשפה ולאו דוקא מאיך שהסימולטור פועל.

נסתכל על Benchmark בין השפות:

regex-redux

source	secs	mem	gz	busy	cpu load
Python 3	18.39	453,960	512	36.53	35% 39% 32% 93%
Java	10.25	634,884	740	30.23	68% 95% 67% 65%

reverse-complement

source	secs	mem	gz	busy	cpu load
Python 3	16.52	1,004,988	814	22.37	28% 41% 20% 47%
Java	3.26	738,800	2183	7.49	44% 63% 73% 48%

k-nucleotide

source	secs	mem	gz	busy	cpu load
Python 3	73.39	217,080	1967	279.23	96% 95% 96% 94%
Java	9.06	435,240	1812	28.35	98% 73% 72% 69%

binary-trees

source	secs	mem	gz	busy	cpu load
Python 3	80.71	448,276	589	288.85	87% 88% 87% 96%
Java	8.29	966,764	835	27.76	86% 80% 86% 83%

fannkuch-redux

source	secs	mem	gz	busy	cpu load
Python 3	493.19	47,864	950	1,942.93	98% 100% 100% 96%
Java	17.93	34,616	1282	70.89	98% 100% 98% 99%

fasta

source	secs	mem	gz	busy	cpu load
Python 3	66.22	843,344	1947	141.79	87% 13% 47% 66%
Java	2.24	44,876	2473	6.13	18% 79% 79% 97%

mandelbrot

source	secs	mem	gz	busy	cpu load
Python 3	258.69	48,416	688	1,033.06	100% 100% 100% 100%
Java	6.90	78,956	796	27.14	99% 98% 98% 98%

n-body

source	secs	mem	gz	busy	cpu load
Python 3	857.90	8,160	1196	881.88	25% 76% 1% 1%
Java	21.91	35,356	1429	22.28	100% 0% 0% 1%

spectral-norm

source	secs	mem	gz	busy	cpu load
Python 3	645.00	9,476	328	652.53	65% 36% 0% 0%
Java	4.20	36,636	950	16.37	97% 98% 97% 97%

pidigits

source	secs	mem	gz	busy	cpu load
Python 3		Failed			
Java	3.07	39,184	938	3.29	99% 3% 4% 1%

איור 39 טבלת ביצועים בין Java לבין Python בסוגים שונים של בדיקות

נוכל לראות כי java יהיה מהיר יותר, הרבה מכך הודות לאופטימזציות שמבצע ה-JVM.
כאשר אנו מריצים קוד שקומפל ל byte-code ה JVM מתרגם אותו בעת הריצה ל native machine code זהו מנגנון ה JIT (just in time compiler)

בנוסף java גם תומך ב Multi threaded ואינו מוגבל כמו CPython ב GIL.

מחקר עתידי

בסימולטור שבנינו התמקדנו בהרצת אלגוריתמים במוד סינכרוני, הכוונה שיש שעון גלובאלי האחראי שכל הקדקודים מתואמים איתו, זאת אומרת שכאשר קדקוד שולח הודעות לשכניו מובטח כי הן יגיעו בסיבוב הבא.
נוכל להרחיב את הסימולטור שלנו בכך שנוסיף מוד אסינכרוני, מוד בו לכל קדקוד יש שעון משלו, נוכל לתת זמן רנדומי להודעה להגיע לשכן בתור מימוש default לאסינכורניות,
בסימולטור שלנו ישנה הכנה למצב default זה, כאשר אצלנו לאחר שקדקוד שולח הודעה לשכנו היא תהיה זמינה לאחר סיבוב אחד (מצב סינכרוני), נוכל לשנות זאת ולממש פונקציית רנדום על הסיבוב בו ההודעה תהיה זמינה, כך למעשה אנו מסמלצים אסינכורניות ושעון לוקאלי לכל קדקוד.

נוכל גם להרחיב את הפונקציונליות שלנו ולתת לאלגוריתמיקאי או המפתח אפשרות לממש בעצמם לוגיקה לאסינכרוניות שתשפיע על כל הודעה שנשלחת, כל מה שהם יצטרכו לעשות זה רק לדרוס את הפונקציה האחראית על כך, כל זאת כמובן כאשר בקונפיגורציה נחליט על מצב Async

הסימולטור שפותח מופרד ממסך התצוגה והמידע מועבר בפרוטוקול Http, כאשר מדובר על גרף גדול לא נוכל להעבירו בבקשה אחת ונצטרך לעשות זאת בצורה מצטברת כאשר ב API המחזיר את קדקודי הגרף עם המידע הנלווה להם נעביר אינדקס, צד ה client יוכל כל פעם לשלוח מאיזה אינדקס הוא רוצה לקבל את המידע ובכך יוכל לבנות את הגרף בצד שלו.

ראינו כי python במימושה הנוכחי (CPython) אינה תומכת ב Multithreading אמיתי כפי שציפינו, החידוש כאן הוא הגישה המפרידה את הסימולטור עם התצוגה ולא השפה שבה נכתב הפרויקט.
לכן נוכל לשכתב את הפרויקט בשפה אחרת נניח JAVA אולם הפעם נרוויח מהיתרונות של שימוש ב Multi threaded וככל הנראה מביצועים טובים יותר.
לחילופין נוכל להשתמש במימוש אחר של python דוגמת PyPy או Jyrhon וכו'.

סיכום

ישנם מספר פרדיגמות להרצה של אלגוריתמים מבוזרים,
בפרויקט הזה התמקדנו בפרדיגמה דומה לזו של Sinalgo

הפרויקט הכיל מורכבות גם מבחינת למידת הסימולטור הקיים (Sinalgo) והן מבחינת התמודדות עם כתיבת סימולטור חדש, מתחרה, עם דימיון בחלוקה של מחלקות הבסיס אולם שוני בגישה הארכיטקטונית.

לאחר כתיבת הפרויקט ישנם מספר נקודות חוזקה ומספר נקודות חולשה שראינו, אך ניתן לומר שישנו כיוון לשיפור גדול של הסימולטור, שיפור זה יכול להתעלות בהרבה פרמטרים על הסימולטור Sinalgo (לדוגמא כתיבת הסימולטור ב JAVA עם תמיכה ב MultiThreading)

החידוש כאן מעבר לשיפורים הקטנים (כמו MultiThreading) היה ההפרדה בין המנוע לתצוגה והאפשרות לתת לכל רכיב כזה לרוץ בנפרד אפילו על גבי שרת שונה ובכך לא להעמיס על אותו שרת.
הפרדה זו נותנת לנו יתרונות רבים ואפשרויות נרחבות להמשך ש Sinalgo לא נותן

בפרויקט הזה ראינו גם כמה הסימולטור Sinalgo הוא מורכב, מכיל הרבה לוגיקה ומכיל הרבה יכולות שלא פשוט לממש.

בפרויקט התנסינו גם בכתיבת מספר אלגוריתמים מבוזרים ובכך סיגלנו לנו את הגישה הרלוונטית לתחום, השונה מהגישה לפיתוח אלגוריתמים מסורתיים שבהם הגרף כולו מובא לאלגוריתם כקלט.
גישה זו היא כמובן רלוונטית כיום יותר עם התפתחות הטכנולוגיה וההפרדה בין רכיבים (מעבדים, שרתים וכו')

בפאן האישי – הפרויקט הכניס אותי יותר עמוק לתחום שמאוד התחברתי אליו, סימולטור להרצת אלגוריתמים מבוזרים הוא חלק חשוב מבדיקה וכתיבה של אלגוריתמים שמפתחים, הוא נותן את האפשרות גם לצפות ברמה ויזואלית וגם לוודא שהאלגוריתם שכותבים אכן עושה מה שהאלגוריתמיקאי התכוון שיעשה.
מעבר לכך שדרך הפרויקט למדתי עובדות טכנולוגיות בנושאים שהפרויקט נכתב גם סיגלתי לי הגישה שצריך עבור התחום הזה.
מעבר למסגרת הפרויקט אני מאמין שאמשיך לשפר את הסימולטור ולכניס את מרבית השיפורים העתידיים שצויינו על מנת להפוך אותו לסימולטור שימושי ומוביל בתחום.

מקורות

1) Anton Malakhov, "Composable Multi-Threading for Python Libraries", Proc. of the 15th Python in Science Conf. (SCIPY 2016), July 11-17, 2016

2) K. Erciyes. Distributed Graph Algorithms for Computer Networks. Springer Publishing Company, Incorporated, 2013

3) Nagoya Institute of Technology, Gokiso-cho, Showa-ku, Distributed Graph Algorithms and their Complexity: An Introduction, Nagoya 466-8555, Japan, 2015

4) Benchmark Python Vs Java:
https://benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python.html

5) JavaScript Graphs Framework: SigmaJS
http://sigmajs.org/

6) Python GIL:
https://realpython.com/python-gil/

7) Python Framework: Flask
http://flask.palletsprojects.com/en/1.1.x/

8) SigmaJS Tutorial :
https://www.bsimard.com/2018/04/25/graph-viz-with-sigmajs.html

9) Tutorial Sinalgo:
https://sinalgo.github.io/tutorial/Documentation.html