

**through the openings**

ספר פרויקט - פתרון מבוכים

שם התלמידה: שירה בקשי.

מספר זהות: 325703486.

מקום לימודים: סמינר נוות ישראל.

שם המנחה: הגב' מירי דבורקין.

תאריך הגשה: 06/2024.

תוכן עניינים

[**1.** **הצעת פרויקט** 4](#_Toc168261322)

[**2.** **תקציר** 9](#_Toc168261323)

[**2.1. רקע לפרויקט** 9](#_Toc168261324)

[**2. 2. תהליך מחקר:** 10](#_Toc168261325)

[**2.3. סקירת ספרות** 11](#_Toc168261326)

[**2.4. אתגרים מרכזים:** 11](#_Toc168261327)

[**2.4.1. הבעיה אתה התמודדתי** 11](#_Toc168261328)

[**2.4.2. הסיבות לבחירת הנושא** 12](#_Toc168261329)

[**2.4.3. הצורך שהפרויקט עונה עליו** 12](#_Toc168261330)

[**2.4.4. דרכי פתרון שנבדקו** 12](#_Toc168261331)

[**1.** **יעדים ומטרות** 13](#_Toc168261332)

[**2.** **מדדי הצלחה** 14](#_Toc168261333)

[**3.** **אתגרים** 15](#_Toc168261334)

[**4.** **רקע תאורטי** 16](#_Toc168261335)

[**7.** **מצב קיים** 17](#_Toc168261336)

[**8.** **ניתוח חלופות מערכתי** 22](#_Toc168261337)

[**9.** **תיאור החלופה הנבחרת והנימוקים לבחירתה** 24](#_Toc168261338)

[**10. אפיון מערכת** 27](#_Toc168261339)

[**10.1. ניתוח דרישות המערכת** 27](#_Toc168261340)

[**10.2. מודול המערכת** 27](#_Toc168261341)

[**10.3. אפיון פונקציונלי** 27](#_Toc168261342)

[**10.4. ביצועים עיקריים** 27](#_Toc168261343)

[**10.5. אילוצים** 28](#_Toc168261344)

[**11. תיאור הארכיטקטורה** 29](#_Toc168261345)

[**11.1. הארכיטקטורה של הפתרון המוצע** 29](#_Toc168261346)

[**11.2. תיאור הרכיבים בפתרון** 29](#_Toc168261347)

[**11.3. ארכיטקטורת הרשת** 29](#_Toc168261348)

[**11.4. תיאור פרוטוקולי התקשורת** 29](#_Toc168261349)

[**11.5. שרת-לקוח** 29](#_Toc168261350)

[**12. ניתוח ותרשים UML / Use cases של המערכת המוצעת** 30](#_Toc168261351)

[**12.1. תיאור ה-UC העיקריים של המערכת** 30](#_Toc168261352)

[**12.2. הצגת use caseעבור כל הפונקציות העיקריות בפרויקט** 30](#_Toc168261353)

[**12.3. מבנה נתונים בפרויקט** 31](#_Toc168261354)

[**12.4. הקשרים בין היחידות השונות** 32](#_Toc168261355)

[**12.5. עץ מודולים** 32](#_Toc168261356)

[**12.6. תרשים המחלקות בפרויקט** 33](#_Toc168261357)

[**12.7. תיאור המחלקות המוצעות** 33](#_Toc168261358)

[**13. רכיבי ממשק** 35](#_Toc168261359)

[**14. תיכון המערכת** 36](#_Toc168261360)

[**14.1. ארכיטקטורת המערכת** 36](#_Toc168261361)

[**14.2. תיכון מפורט** 36](#_Toc168261362)

[**14.3. חלופות המערכת** 37](#_Toc168261363)

[**15. תיאור התוכנה** 38](#_Toc168261364)

[**15.1. סביבת עבודה** 38](#_Toc168261365)

[**15.2.**  **שפות תכנות** 38](#_Toc168261366)

[**16 . תיאור מסכים** 39](#_Toc168261367)

[**17. תרשים מסכים המתאר את היררכיית המסכים** 40](#_Toc168261368)

[**18. פרוט המסכים** 41](#_Toc168261369)

[**19. קוד התוכנית + תיעוד** 42](#_Toc168261370)

[**20. תיאור מסד הנתונים** 53](#_Toc168261371)

[**21. מדריך למשתמש** 54](#_Toc168261372)

[**22. בדיקות והערכה** 55](#_Toc168261373)

[**23. ניתוח יעילות** 56](#_Toc168261374)

[**24. אבטחת מידע** 57](#_Toc168261375)

[**25. מסקנות** 58](#_Toc168261376)

[**26. פיתוחים עתידיים** 59](#_Toc168261377)

[**27. ביבליוגרפיה** 60](#_Toc168261378)

# **הצעת פרויקט**

**סמל מוסד:** 141119

**שם מכללה:** סמינר נוות ישראל

**שם סטודנט:** שירה בקשי

**ת.ז**: 325703486

**שם פרויקט:** פותר מבוכים אוטומטי

**פרטי הפרויקט-**

**תיאור הפרויקט :**תוכנה שבה המשתמש יעלה תמונה של מבוך. ויקבל כפלט תמונה שעליה משורטט מסלול היציאה מהמבוך. על התמונה להיות מרובעת בלבד, עם מסגרת שחורה בעלת שני פתחים – יציאה וכניסה. בנוסף גודל המבוך בתמונה לא פחות מ 65% מגודל התמונה.

התמונה נשלחת לעיבוד התמונה, משם לבניית גרף ומשם לאלגוריתם למציאת המסלול הקצר, חוזרת לעיבוד התמונה על מנת סימון המסלול ומשם ללקוח.

המטרה שלי שהתוכנה תצליח לקורא תמונת מבוך ולהחזיר מבוך פתור במהירות גבוהה.

(תמונה זו תשלח לשרת בפיתון, שם היא תעובד למטריצה בינארית שבה המבוך בלבד. לאחר מכן ייבדקו אורך קיר במבוך ואורך מעבר. את המטריצה, המעבר והקיר אכניס לקובץ JSON שאותו אשלח לשרת WEB API , השרת שולח את המטריצה והאורכים למציאת המסלול הקצר. שם אני אעבור על המטריצה ואבנה גרף. הצמתים בגרף יהיו רק נקודות פיצול או שינוי כיון. ( לא כל מעבר הינו צומת) לאחר מכן יורץ האלגוריתם בשני threads מהכניסה והיציאה של המבוך, האלגוריתם ייעצר כששני הthreads יעמדו על אותה צומת. משם יסומן הנתיב על המטריצה והיא תוחזר אל השרת בפיתון ומשם )

**הגדרת הבעיה האלגוריתמית**:

1. זיהוי המבוך בתמונה
2. מציאת מהו גודל של קיר וגודל של מעבר.
3. בניית גרף באופן יעיל.
4. פתירת המבוך בסיבוכיות סבירה.

**רקע תיאורטי בתחום הפרויקט:** כיום ישנם הרבה משחקי מבוך. אך אין אפליקציה לפתרון מבוכים.

את התמונה שהמשתמש יעלה אהפוך אותה לשחור- לבן , וכך אוכל לעקוב יותר טוב על צורת המבוך, אותו אכניס למטריצת 0, 1 כאשר 1 הוא קיר , ו – 0 הוא מעבר.

המטרה היא למצוא את המסלול הקצר ביותר בין שתי נקודות במבוך, שהוא למעשה גרף דו-ממדי עם קירות (קצוות חסומים) ומעברים (קצוות פתוחים). בעיית המבוך יכולה להיות מיוצגת כגרף, שבו כל צומת מייצג נקודה במבוך וכל קשת מייצגת מעבר אפשרי בין נקודות סמוכות.

**תיאור אלגוריתמים ו/או פתרונות הקיימים בעולם:**

בעיית המבוך נפתרת באמצעות אלגוריתמים לחיפוש בגרף. האלגוריתמים הנפוצים כוללים:

* אלגוריתם BFS - אלגוריתם חיפוש לרוחב.
* אלגוריתם DFS - אלגוריתם חיפוש לעומק.
* אלגוריתם דייקסטרה.
* אלגוריתם הליכה בעקבות הקיר.
* אלגוריתם העכבר האקראי.
* אלגוריתם בלמן פורד.
* אלגוריתם DAG.

**הבהרת המושגים בבסיס הבעיה האלגוריתמית**

צומת (Node): נקודה במבוך המייצגת מקום אפשרי למעבר.

קשת (Edge): מעבר בין שני צמתים סמוכים.

גרף: מבנה מתמטי המורכב מצמתים וקשתות.

מסלול (Path): סדרת צמתים המחברת בין נקודת ההתחלה לנקודת הסיום.

אלגוריתם חיפוש: פרוצדורה למציאת מסלול או פתרון במבוך. אלגוריתם A\*.

**הקשר בין הבעיה האלגוריתמית לטכנולוגיות ההנדסה הקיימות**

הפתרון לבעיית המבוך משלב מספר תחומים טכנולוגיים:

עיבוד תמונה: זיהוי והמרת התמונה למטריצה בינארית.

תורת הגרפים: ייצוג המבוך כגרף ועיבודו.

תכנות מקבילי: שימוש ב-threads למציאת המסלול במהירות גבוהה.

מערכות מבוזרות: תקשורת בין שרת ה-API לשרת המחשב המבוזר לפתרון הבעיה.

**מודל תאורטי העומד בבסיס הפרויקט, השלכות וחשיבות הפרויקט**

המודל התאורטי מבוסס על גרפים ואלגוריתמי חיפוש בגרף. הפרויקט חשוב מכיוון שהוא מציע פתרון יעיל ומהיר למציאת מסלולים במבוכים, דבר שיכול להיות שימושי בתחומים כמו רובוטיקה, משחקים ואפליקציות ניווט.

**פתרונות רלוונטיים :**

שיפור ביצועים על ידי תכנות מקבילי: הרצת שני threads מהכניסה והיציאה במקביל.

שימוש בהיוריסטיקות: שילוב אלגוריתם A\* עם פונקציות הערכה מותאמות למבוך.

שיפור זיהוי ועיבוד תמונה: שימוש בטכניקות עיבוד תמונה מתקדמות לזיהוי מדויק יותר של קירות ומעברים.

**ניתוח של הקשרים וההבדלים בין המחקרים/הפתרונות**

פתרונות קיימים כוללים אלגוריתמים לחיפוש בגרף כמו BFS, דייקסטרה ו-A\*. כל אחד מהאלגוריתמים מתאים למצבים שונים: BFS למבוכים בלתי ממושקלים, דייקסטרה למבוכים ממושקלים ו-A\* כאשר ניתן להשתמש בהיוריסטיקה. תכנות מקבילי ושימוש בטכניקות עיבוד תמונה מתקדמות יכולים לשפר את הביצועים.

**ארכיטקטורת הפרויקט**

הפרויקט מורכב ממספר שלבים וטכנולוגיות המקושרות ביניהן ליצירת פתרון כולל לפתרון מבוכים. להלן פירוט הארכיטקטורה:

1. **קלט תמונה ועיבוד תמונה**
   * **קלט משתמש**: המשתמש מעלה תמונת מבוך לשרת.
   * **עיבוד תמונה ראשוני**: התמונה עוברת תהליך עיבוד ראשוני הכולל הסרת רעשים, תיקון עיוותים וזיהוי הקירות והמעברים במבוך.
   * **המרת תמונה למטריצה בינארית**: התמונה מומרת למטריצה בינארית, שבה 1 מייצג קיר ו-0 מייצג מעבר.
2. **שליחת הנתונים ל-API**
   * **יצירת JSON**: המטריצה הבינארית יחד עם מידע נוסף על אורכי קירות ומעברים נארזים לקובץ JSON.
   * **שליחת JSON לשרת API**: הנתונים נשלחים לשרת Web API לצורך עיבוד נוסף.
3. **עיבוד נתונים ובניית גרף**
   * **קבלת נתונים**: שרת ה-API מקבל את קובץ ה-JSON.
   * **בניית גרף**: המטריצה הבינארית מומרת לגרף, שבו כל צומת מייצג נקודת פיצול או שינוי כיוון.
   * **הגדרת אלגוריתם חיפוש**: האלגוריתם למציאת המסלול הקצר ביותר מוגדר ומוכן להרצה.
4. **מציאת המסלול הקצר ביותר**
   * **הרצת אלגוריתם בשני threads**: האלגוריתם רץ משני כיוונים (כניסה ויציאה) במקביל, במטרה למצוא את הצומת המשותפת.
   * **חיבור המסלול**: כאשר שני ה-threads נפגשים, המסלול הקצר ביותר מחובר ונשמר.
5. **סימון המסלול על התמונה המקורית**
   * **עדכון המטריצה**: המסלול הקצר מסומן במטריצה הבינארית.
   * **החזרת המטריצה :** המטריצה המעודכנת מוחזרת לשרת בפיתון, שם היא תיהפך חזרה לתמונה ומסלול שסומן יצבע באדום.
6. **שליחת הפתרון למשתמש**
   * **שליחת הפתרון למשתמש**: התמונה נשלחת בחזרה למשתמש דרך השרת .

**תיאור פרוטוקולי תקשורת:** HTTP

**פיתוחים עתידים:** האתר ידע גם לייצר מבוכים ולסווגם לפי רמות קושי, מציאת פתרון לכל מבוך שהו בכל צורה. התאמת התוכנה לפאלון נייד.

**תיאור טכנולוגיה הנדסה:**  עיבוד התמונה אכתוב בשפת פיתון, היות שבפיתון יש הרבה ספריות על עיבוד תמונות. את הפתרון המבוך אעשה בשפת c#, היות ושפה זו מאוד מסודרת , מבחינתי לכתיבת מחלקות שבהם אשתמש לפתרון המבוך.

**מסד נתונים:** אין מסד נתונים.

**פרטים פורמליים- לוחות זמנים:**

אלול-כסלו: זיהוי המבוך בתמונה

טבת- שבט: ייצוג המבוך במבנה נתונים מתאים

אדר א' – כתיבת אלגוריתם הפתרון

אדר ב' – ניסן: הקמת שרת לקוח

**מנחות פרויקט:** מרים דבורקין

**חתימת הסטודנט:** שירה בקשי

**חתימת רכז המגמה:** רחל פנסטר

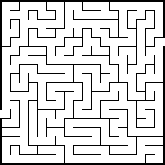
# **תקציר**

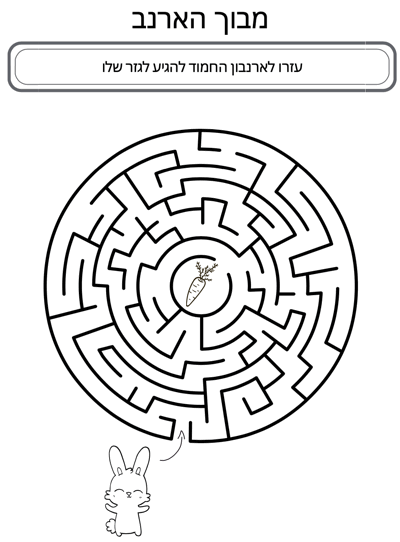
## **2.1. רקע לפרויקט**

הפרויקט שלי עוסק בפתרון מבוכים תוך עיבוד התמונה שהביא המשתמש.

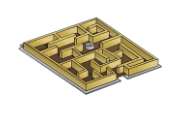
מבוך- מבוך הוא חידה הבנויה ממעברים מתפצלים, וקירות – מחסומים, על הפותר למצוא נתיב מתחילת המבוך לסופו.

ישנם סוגים רבים של מבוכים:

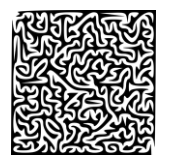
* המבוך הקלאסי – מרובע שבו כניסה ויציאה.
* מבוך עגול – סיום המבוך נמצא במרכז המעגל.



* מבוכי תלת ממד – אלו סוגי מבוכים שנראים כתלת ממד, הקירות גבוהים, ויתכן שלא יראו את המעבר, אך הוא קיים. מבוכים אלו יכולים להיות מיוצגים גם כבתים או מפות עם מדרגות וקירות.



* מבוכי צורות – ישנם מבוכים שקירותיהם מיוצגים כחוטים כאלו, גלים, צורות גאומטריות וכד'.



בפרויקט שלי אפתור מבוכים מרובעים ע"י עיבוד תמונה:

עיבוד התמונה:

המשתמש מכניס תמונה.

ייתכן ובתמונה יהיו עצמים נוספים, ולכן יש להסירם.

לצורך ההסרה, יצאתי מנקודת הנחה, שהמבוך בתמונה יהיה לפחות 60% מן המבוך, ולכן ביצעתי אלגוריתם לזיהוי שחור ארוך שהוא יהיה המסגרת של המבוך.

לאחר זיהוי קטעים ארוכים, שמרתי את הקודקודים וחתכתי את התמונה.

לאחר שיש לי נטו את המבוך, אני בודקת מהו אורך של קיר במבוך, ומהו אורך מעבר.

האלגוריתם ירוץ על התמונה לאורך ולרוחב, וכל קטע שחור או לבן הוא יסכום אותו.

במערך של HASH TABLE אני מוסיפה אחד באינדקס הסכום. ככה שהאורך בעל השכיחות הגבוה יהיה אורך הקיר או המעבר.

## **2. 2. תהליך מחקר:**

כיום בשוק ישנו מוצר ממשי – [רובוט](https://www.out-of-the-box.co.il/items/1557440-%D7%A8%D7%95%D7%91%D7%95%D7%98-%D7%97%D7%9B%D7%9D-%D7%94%D7%95%D7%9C%D7%9A-%D7%90%D7%97%D7%A8%D7%99%D7%99-%D7%A7%D7%95-MazeBreaker-%D7%A4%D7%95%D7%AA%D7%A8-%D7%9E%D7%91%D7%95%D7%9B%D7%99%D7%9D) הנכנס פיזית לתוך מבוך ופותר אותו. ישנן גם אפליקציות משחק של מבוכים, הן עוסקות ביצירת מבוכים והמשתמש פותר, ישנה עוד [אפליקציה היוצרת](https://puzzlemaker.discoveryeducation.com/maze) ופותרת אך אינה נעימה ברורה ונוחה לשימוש, בנוסף היא מוגבלת עד לגודל של 100X100.  
האפליקציה שלי עוסקת בעיקרה בפתרון מבוכים גדולים ואף גדולים מאוד במהירות גבוהה.

את המבוך ניתן לייצג ע"י גרף.

גרף הוא ייצוג מופשט של קבוצת אובייקטים, כאשר כל זוג אובייקטים בקבוצה עשויים להיות מקושרים זה לזה. האובייקטים הניתנים לקישור מכונים קדקודים או צמתים, ומקושרים ע"י קשתות.

במבוך כל התפצלות היא קדקוד, והמרחק בין הקדקודים הוא הקשת לקדקוד הבאה.

ישנם אלגוריתמים רבים למציאת המסלול הקצר בגרף:

* אלגוריתם DFS - אלגוריתם חיפוש לעומק.
* אלגוריתם BFS - אלגוריתם חיפוש לרוחב.
* אלגוריתם הליכה בעקבות הקיר.
* אלגוריתם העכבר האקראי.
* אלגוריתם דייקסטרה.
* אלגוריתם בלמן פורד.
* אלגוריתם DAG.
* אלגוריתם A STAR.

עיבוד התמונה:

* אלגוריתם לזיהוי קצוות.
* אלגוריתם מורבץ לזיהוי קצוות.
* אלגוריתם לזיהוי גבול ורק"חים בתמונה.

## **2.3. סקירת ספרות**

במהלך כתיבת הפרויקט נעזרתי בהרבה אתרי אינטרנט המביאים דרכים שונות לפתירת מבוכים, נעזרתי בקודים בgit בעיקר בהסברים אודותיהם, ובעזרתם גיבשו לי הבנה איך אני רוצה לפתור את המבוך.

* על אלגוריתם A STAR : <https://rotter.name/nor/prog/15871.shtml>
* אלגוריתמים לפתרון מבוכים: <https://he.wikipedia.org/wiki/%D7%90%D7%9C%D7%92%D7%95%D7%A8%D7%99%D7%AA%D7%9E%D7%99%D7%9D_%D7%9C%D7%A4%D7%AA%D7%A8%D7%95%D7%9F_%D7%9E%D7%91%D7%95%D7%9B%D7%99%D7%9D>
* פרויקט מ- gitub לפתרון מבוכים: <https://github.com/mikepound/mazesolving>
* הסבר על האלגוריתם : <https://www.youtube.com/watch?v=-L-WgKMFuhE>

## **2.4. אתגרים מרכזים:**

### **2.4.1. הבעיה אתה התמודדתי**

* ניקוי התמונה מעצמים.
* זיהוי המבוך בתוך תמונה.
* הפיכת המבוך למטריצה.
* זיהוי צמתים במבוך.
* כתיבת אלגוריתם בעל סיבוכיות נמוכה ככל האפשר, גם במבוכי ענק.

### **2.4.2. הסיבות לבחירת הנושא**

תמיד אהבתי לפתור מבוכים גדולים, וכשהגעתי ללמוד תכנות עניין אותי לנסות לכתוב אלגוריתם לפתירת מבוכים, ובמסגרת זו של פרויקט גמר ראיתי בזה הזדמנות.

### **2.4.3. הצורך שהפרויקט עונה עליו**

הפרויקט עונה על הצורך שמשתמש הרוצה לראות פתרון למבוך מסוים שלו יוכל להעלות את התמונה שלו, ולקבל את הפתרון.

### **2.4.4. דרכי פתרון שנבדקו**

* אלגוריתם למציאת פיקסלי הגבול/קונטור וההיקף ע"י מציאת נקודת ההתחלה השחורה ומעבר על שמונת שכיניו החל מהשכן המערבי (עם כיוון השעון) עד למציאת פיקסל שחור.
* אלגוריתם מורבץ לזיהוי פינות )נקודות העניין ( בתמונה ע"י מדידת השונות בנקודה מסוימת של התמונה. מגדירים חלון ריבועי סביב הנקודה ואז מזיזים את החלון בשמונה כיוונים שונים )אופקי אנכי ושני אלכסונים( ואזה מסכמים את ריבועי ההפרשים בין הנקודות המתאימות בחלון המקורי והמוזז .השונות המינימאלית מאלו שחושבו תהיה השונות של נקודה P.
* לפתרון, אלגוריתם דייקסרטה, BFS , על גרף.

# **יעדים ומטרות**

מטרות:

* האתר יספק פתרון של מבוך גדול במהירות.
* זיהוי מבוך מתוך תמונה
* מציאת אלגוריתם מהיר לפתרון מבוך.
* חסכון בזמן ומאמץ מיותר מצד המשתמש.
* המשתמש ירגיש שייכות לאתר ע"י העלאת התמונות שבחר.

יעדים:

* בנית אתר נעים ונוח למשתמשים.
* האתר יכיל מסכים מעוצבים למשתמשים.
* חקירת הנושא של גילויי קצוות בתמונה בצורה הטובה ביותר.
* האתר יכיל תמונות של מבוכים ואפשרות של העלאת תמונה מסוימת של מבוך.
* פתרון למבוכי ענק.

# **מדדי הצלחה**

* התוצר הסופי – האתר יאפשר למשתמש להעלאות תמונה, יפענח את המבוך בתמונה ויחזיר פתרון.
* האתר יצליח לפתור מבוכים בגדלים גדולים.
* האתר יזהה את במבוך התמונה.
* האתר יאפר ליצור מבוכים, ולהורידם.

# **אתגרים**

במהלך כתיבת הפרויקט ניצבו בפני אתגרים שונים אשר אתגרו אותי לא מעט אבל כמובן

רכשתי בעקבתן ניסיון רב ומיומנויות חדשות. כמו:

* למידה עצמית.
* כתיבת ממשק משתמש נוח.
* כתיבת אלגוריתם מהיר ויעיל ככל הניתן גם במבוכים מורכבים.
* חקירת נושא הפרויקט.
* פתירת באגים באלגוריתם.
* לימוד קודים מוכנים.
* קריאת התמונה.

# **רקע תאורטי**

פרויקט זה עוסק במבוכים, מהי הדרך הקצרה ביותר למציאת מסלול במבוך.

לצורך הפתרון חקרתי אלגוריתמים שונים על מנת שהפתרון למבוך יהיה המהיר ביותר. גם כאשר מדבור על מבוכים בסדרי גודל גדולים מאוד.

היות והמבנה הנתונים הנכון ביותר לייצוג מבוך זהו גרף התחלתי לבדוק אלגוריתמים שונים למציאת המסלול קצרים משני נק' בגרף.

אלגוריתם הנכון ביותר שראיתי לנכון לפתור בעזרתו את המבוך זהו **אלגוריתם A STAR!**

מה שמיוחד באלגוריתם זה שהוא מוצא את המסלול הקצר ביותר על גרף מנקודה לנקודה ביעלות טובה.

הוא מתבסס על אלגוריתם דייקסטרה אך יעיל ממנו, כיון שהוא מוצא את המסלול מקדקוד מסוים לקדקוד אחר, ולא כמו דייקסטרה מקדקוד אחד לכל הקדקודים.

בנוסף A\* מסתמך על הערכה.

האלגוריתם הוא מסוג אלגוריתם חיפוש לרוחב ומוצא את המסלול הזול ביותר מצומת המקור לאחד מצומתי היעד. כאשר \*A עובר על הגרף הוא מנסה לפתח בכל פעם את המסלול שנראה הזול ביותר תוך כדי שימוש בתור עדיפויות ממוין. האלגוריתם משתמש בפונקציית עלות F(X) המשלבת היוריסטיקה ((X)H) בצירוף מידע ידוע ((X)G) כדי לבחור איזה צומת לפתח, לבחור בה להמשך. פונקציית העלות היא סכום של שתי הפונקציות:

* ((X)G) - שהיא המרחק הידוע בין צומת ההתחלה לצומת הנוכחי x.
* ((X)H) - שהיא הערכה למרחק בין x לבין צומת היעד.

כדי שהאלגוריתם יחזיר פתרון אופטימלי, פונקציית ההיוריסטיקה, ((X)H) צריכה להיות קבילה, כלומר לא יותר יקרה מהמרחק האמיתי לצומת היעד.

מבני הנתונים שאשתמש עם אלגוריתם זה הם:

1. תור קדימויות(ערימת מינימום).
2. רשימה על מנת שהם מחלק מהמסלול.

אלגוריתם זה ירוץ מפתח המבוך והיציאה ממנו במקביל , ויעצור כשיפגשו שני הריצות.

# **מצב קיים**

ישנם מספר גדול של אלגוריתמים לפתרון מבוכים, המיוצגים בגרף.  
אלגוריתם DFS, BFS, דייקסטרה, האלגוריתם של בלמן פורד, DAG.

**אלגוריתם DFS - אלגוריתם חיפוש לעומק:**

אלגוריתם זה משמש למעבר/חיפוש על גרף. הוא מתחיל מנקודת ההתחלה של המבוך ועובר על כל הצמתים השכנים – אם יש כאלו בסדר כלשהו אם אין – חוזר אחורה, כך הוא עובר על כל המצתים עד שמגיע לצומת הסיום המבוך.

הדרך שבא משתמשים באלגוריתם חיפש לעומק היא רקורסיבית. אלגוריתם חיפוש לעומק בודק כל צומת במסוך עליו הוא הופעל ולאחר מכן מפעיל את עצמו רקורסיבית על כל אחד מהצמתים – השכנים שלו אם הוא עדיין לא ביקר בהם. זאת הוא בודק באמצעות סימון של כל צומת בלבן – טרם נבדק, אפור – עדין הוא או צאצאיו בבדיקה ובשחור – הצומת כבר נבדקה.

קטע פסאדו קוד של האלגוריתם:

**function** DFS(Start, Goal)

Color(Start, Grey)

**if** Start = Goal

**return** True

**for** Child **in** Expand(Node)

**if** not Colored(Child)

**if** DFS(Child, Goal)=True

**return** True

Color(Start, Black)

**return** False

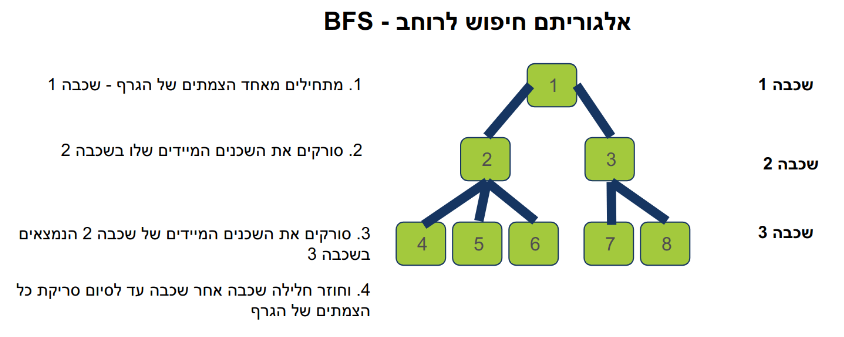
הסיבוכיות של אלגוריתם זה היא:

היא O(V + E), כאשר V הוא מספר הצמתים ו-E הוא מספר הקשתות.

סיבוכיות זו הינה גבוה, כשמדובר במבוכים גדולי מאוד זה ממש לא יעיל, ולכן בחרתי שלא להשתמש באלגוריתם זה.

**אלגוריתם BFS - אלגוריתם חיפוש לרוחב:**

אלגוריתם סריקה בסיסי של גרפים הוא פועל באופן שיטתי ובשכבות. הוא מתחיל בצומת ההתחלה של המבוך ואז חוקר את שכניו ואז עובר לשכבת השכנים הבאה בתור וזאת ע"י מבנה נתונים של תור, כדי לעקוב אחר סדר הסריקה.



פסאדו קוד:

**function** breadthFirstSearch (Start, Goal)

{

enqueue(Queue,Start)

setVisited(start)

**while** notEmpty(Queue)

{

Node := dequeue(Queue)

**if** Node = Goal

{

return Node

}

**for each** Child **in** Expand(Node)

{

**if** notVisited(Child)

{

setVisited(Child)

enqueue(Queue, Child)

}

}

}

}

סיבוכיות זמן הריצה של אלגוריתם זה היא:

O(V + E), כאשר V הוא מספר הצמתים (הקדקודים) בגרף, ו-E הוא מספר הקשתות. וביתר פירוט: ביקור בכל צומת פעם אחת: הלולאה while תמשיך עד שהתור יתרוקן. במקרה הגרוע ביותר, היא תעבור על כל הצמתים בגרף פעם אחת, וזה O(V).

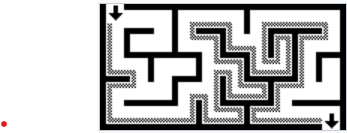
סיבוכיות כאמור גדולה מידיי כאשר מדובר בגרפים ענקיים, שייצגו מבוכים גדולים..  
ולכן לא אשתמש באלגוריתם זה בפתרון המבוך.

**אלגוריתם הליכה בעקבות הקיר:**

אלגוריתם זה מייצג שיטה המבטיחה את הפתרון של המבוך.

הרעיון הוא שבעצם של קירות המבוך מחוברים. זה אומר שניתן לסדר באמצעות לולאה, וכך יוצא שהליכה בעקבות הקיר היא בעצם הליכה מסביב למעגל מנקודת ההתחלה לסוף.

להלן ציור הממחיש את ההליכה בעקבות הקיר:



סיבוכיות של אלגוריתם זה היא גבוה מאוד היות וייתכן שנצטרך לעבור על כל הצמתים הגרף O(N)  
ולכן גם אלגוריתם זה לא יעיל לפתרון מבוכים בצורה המהירה ביותר.

**אלגוריתם העכבר האקראי:**

זוהי שיטה טריוויאלית שיכולה להיות מיושמת על ידי רובוט מאוד לא מפותח או לחלופין על ידי עכבר. באלגוריתם זה, על מבצע האלגוריתם להמשיך בקו ישר עד שהוא מגיע לצומת, ואז להחליט החלטה אקראית לאן לפנות.

על אף ששיטה כזו תביא תמיד את המשתמש בה לפתרון, אלגוריתם זה יכול להיות מאוד איטי ולא יעיל, ולכן לא בחרתי להשתמש בשיטה זו.

**אלגוריתם דייקסטרה:**

זהו אלגוריתם למציאת המסלול ה'קל' ביותר(כלומר שסכום משקולות הקשתות שלו האו המינימאלי האפשרי)

האלגוריתם עובד על גרף מכוון או לא מכוון, בעל משקולות אי-שליליות על הקשתות. המשקולות בגרף מסמלות מרחק. משמעותו של המסלול הקצר ביותר בין שתי נקודות היא המסלול בעל סכום המשקולות הנמוך ביותר בין שתי הנקודות.

פעילותו:

עבור כל צומת, מסומן האם ביקרו בו או לא ומה המרחק צומת המקור(תחילת המבוך) בהתחלה כל הצמתי מסומנים שלא בקרו בהם, והמחק מוגדר אינסופי.

לולאת האלגוריתם:

-כל עוד נותרו קדקודים שלא ביקרנו בהם:

-מסמנים את X (הקדקוד הנוכחי. באיטרציה הראשונה זהו קדקוד המקור S) כקדקוד שביקרו בו.

-עבור כל קדקוד Y שהוא שכן של X ועדיין לא ביקרנו בו:

Y- מעודכן, כך שמרחקו יהיה שווה לערך המינימלי בין שני ערכים: בין מרחקו הנוכחי, לבין משקל הקשת המחברת בין X לבין Y ועוד המרחק בין S ל-X.

-אם כל הקדקודים בוקרו האלגוריתם עוצר. אחרת בוחרים קודקוד X חדש בתור הקודקוד שעדיין לא בוקר ושמרחקו בשלב הזה מצומת המקור S הוא הקל ביותר מבין כל הקודקודים בגרף שטרם ביקרנו בהם.

הסיבוכיות היא:

O(|V|2) אם מדובר ברשימה או מערך.

O(|E|log|V|+|V|log|V|) אם משתמשים בערימה בינארית.

ואם משתמשים בערימת פיבונאצ'י הסיבוכיות משתפרת ל- O(|E|+|V|log|V|) .

זהו אלגוריתם טוב, אך הוא בהכרח יביא לי את התוצאה המהירה ביותר.

**אלגוריתם בלמן פורד:**

זהו אלגוריתם ממש כמו דייקסטרה רק שהוא פועל גם על קשתות שליליות , זה אינו נחוץ במבוך היות וכל הקשתות חיוביות.

**אלגוריתם DAG :**

זהו אלגוריתם שפועל על גרף חסר מעגלים, ובמבוכים ייתכן שיהיו מעגלים.

ולכן לא השתמשתי באלגוריתם זה.

בעקבות כך בחרתי באלגוריתם A STAR.

# **ניתוח חלופות מערכתי**

עיבוד תמונה מול ראיה ממוחשבת:

קיימים יישומים רבים לראיה ממוחשבת ולזיהוי צורות, וניכרת חדירה חזקה של תחום זה

לתעשייה ולמוצרים בעשור האחרון.

"עיבוד תמונה" מתמקדת במשימות בהן הן הכניסה והן היציאה הם תמונות. "ראיה ממוחשבת" לעומת זאת דנה במשימות בהן הכניסה היא תמונה והמוצא אינו תמונה.

ניתן גם להציע חלוקה אחרת של פרקים הקשורים בעיבוד תמונות, המבוססת על תהליך עיבודה של תמונה מרגע יצירתה ועד סיום הטיפול בה במערכת דמוית ראיית אנוש.

בשלב ראשון מצלמה (העין) אמורה לרכוש את התמונה, וזאת ע"י אופטיקה מתאימה למשל במצלמה, או CCD (העדשה בעין), לאחר מכן התקנים אלקטרו-אופטיים (אותם גלאי ובסיום עיבוד אותות אלמנטרי של תיקון תחום) , Cones - ו Rods החיישנים בעין הקרויים

דינאמי, דגימה וקוונטיזציה. ואמנם, לאופטיקה, אלקטרו-אופטיקה ועיבוד אותות קשר ישיר וחשוב לעיבוד תמונות )שלבים אלו מתבצעים ברטינה בעין(. Stream Main - לאחר בנייתה של התמונה, יבואו אלגוריתמי דחיסה ושיפור איכות השייכים לעיבוד תמונות (פעולות אלו מבוצעים בדרך מהעין למוח, ובמוח הראיה עצמו). אחריהם תתחיל פעולת ניתוח (ראייה ממוחשבת) לשם הפקת מידע מהתמונות, כגון זיהוי האנשים, אבחנה בתנועה וכו'.

כך אנו רואים מעבר מאופטיקה ועד ראייה ממוחשבת ולמעשה בינה מלאכותית. הציור הבא סוקר את המסלול הנ"ל על הפרקים אותם הוא רותם.



בפרויקט שלי עסקתי בחילוץ מאפיינים מתוך התמונה אך החזרת התמונה ולא הסקת מסקנות ממנה, כפי שנעשה בראייה ממוחשבת לכן בחרתי בעיבוד תמונה ולא בראיה ממוחשבת המבוססת על מודל. כמו כן רציתי לכתוב אלגוריתם שייתן לי סיפוק בו אני אכתוב את הקודים ואשתפשף כמה שיותר בכך. עצם הבחירה בכתיבת אלגוריתם ולא בלקיחת מודל בנוי שיפרה מאד את היכולת שלי להתמודד עם כתיבת קודים ארוכים ומסובכים.

ניתוח חלופות טכנולוגיות

את עיבוד התמונה כתבתי בשפת פיתון בסביבת העבודה PYCHARM ואת כתיבת האלגוריתמים לפתרון בשפת C# בסביבת העבודה VISUAL STUDIO .

# **תיאור החלופה הנבחרת והנימוקים לבחירתה**

עיבוד התמונה:

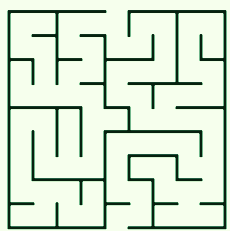
בתחילה המרתי את התמונה לאפור, ניקיתי ממנה רעשים, והפכתי אותה למטריצה בינארית.

על המטריצה הזו אני אעבוד.

לדוגמא:

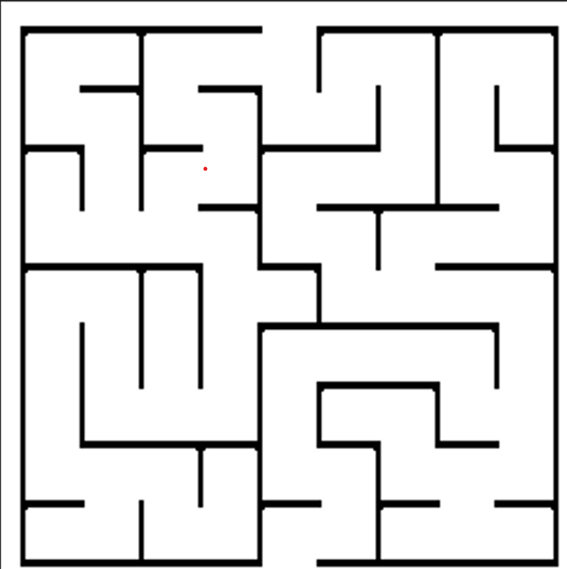
המרה לאפור:

התמונה המקורית:



התמונה בינארית:

לאחר ניקוי רעשים:



לאחר שהפכתי את התמונה הבינארית למטריצה, יכולתי לעבוד עליה ולזהות את המבוך.

היות וייתכן שמשתמש יכניס תמונה שמלבד המבוך יש עוד עצמים, ע"מ לזהות את המבוך ללא התייחסות לעצמים. יצאתי מנקודת הנחה שגודל המבוך בתמונה יהיה לפחות 60% מכלל התמונה, ולכן כתבתי אלגוריתם לזיהוי נקודה שממנה והלך יש רצף של מעל ל60% מאורך \גובה התמונה של פיקסלים שחורים. ונקודה נוספת מהצד השני של המבוך למטה.

כך יש לי 2 נקודות שיכולות לחתוך לי את הריבוע הזה שנוצר, שהוא המבוך.

את מטריצה זו אני שולחת לאלגוריתם אחר שתפקידו למצור לי מהו גודל של תא במבוך. (מעבר , וקיר).

אלגוריתם זה מתבצע כך:

האלגוריתם סוכם כל רצף של פיקסלים שחורים או לבנים, וע"י מבנה של HASH מוסיף 1 באינדקס של הסכום.  
לאחר מכן אני בודקת את הסכום השכיח ביותר שיהיה גודל הקיר/מעבר).

את הנתונים הללו : מטריצה בינארית , גודל של קיר ומעבר אני שולחת לC# לבניית גרף ופתרון המבוך.

**האלגוריתם לפתרון:**

לאחר חקר גדול אודות אלגוריתם A\* נמצא שהאפליקציות המובילות כגון GPS, WESE

מתבססות את אלגוריתם זה, זהו אלגוריתם מאוד פופולרי למציאת מסלול קצר בגרף.

בנוסף הוא האלגוריתם בעל הסיבוכיות הנמוכה ביותר.

* כך עובד אלגוריתם A\*:  
    
   אתחול:  
  - האלגוריתם מתחיל באתחול שתי רשימות: רשימה פתוחה ורשימה סגורה. הרשימה הפתוחה מכילה צמתים שטרם נחקרו, והרשימה הסגורה מכילה צמתים שכבר הוערכו.  
  - האלגוריתם גם מקצה עלות להגיע לכל צומת מצומת ההתחלה ועלות משוערת להגיע לצומת המטרה מכל צומת. עלויות אלו משמשות לקביעת עדיפות הצמתים ברשימה הפתוחה.  
    
  לולאה ראשית:  
  - בעוד שהרשימה הפתוחה אינה ריקה, האלגוריתם ממשיך לחזור על השלבים הבאים:  
  - בחר את הצומת עם העלות הכוללת הנמוכה ביותר (עלות הגעה לצומת מההתחלה + עלות משוערת להגיע ליעד) מהרשימה הפתוחה. הצומת הזה הוא הצומת הנוכחי.  
  - אם הצומת הנוכחי הוא צומת המטרה, האלגוריתם משחזר את הנתיב מצומת ההתחלה לצומת המטרה באמצעות צמתי האב ומסתיים.  
  - אחרת, האלגוריתם מרחיב את הצומת הנוכחי על ידי התחשבות בצמתים הסמוכים לו (צמתים סמוכים) וחישוב העלויות שלהם.  
  - עבור כל צומת שכן:  
  - אם הצומת אינו ברשימה הפתוחה, הוסף אותו לרשימה הפתוחה והגדר את הצומת הנוכחי כהורה שלו. עדכן את העלויות עבור הצומת.  
  - אם הצומת נמצא ברשימה הפתוחה והנתיב החדש אליו קצר יותר, עדכן את העלויות והאב של הצומת.  
  - לבסוף, העבר את הצומת הנוכחי לרשימה הסגורה.  
    
  סיום:  
  - האלגוריתם מסתיים כאשר מגיעים לצומת המטרה (נמצא נתיב) או שהרשימה הפתוחה ריקה (לא קיים נתיב).

# **10. אפיון מערכת**

## **10.1. ניתוח דרישות המערכת**

סביבת פיתוח: visual studio , pycharm

חומרה: 8GB i7

עמדת פיתוח: מחשב נייד DELL

מערכת ההפעלה: 11 windows

מסד נתונים: אין מסד נתונים.

חיבור לרשת: יש צורך.

תוכנות: PyCharm, visual studio code

תוספים טכנולוגיים: כגון מצלמה

דרישות חומרה, מערכת הפעלה ודפדפן בצד הלקוח:

חומרה: 8GB i5 ומעלה.

מערכת הפעלה: windows 7 ומעלה.

* דפדפנים:
* Google chrome- גרסה 60 ומעלה
* Mozilla Firefox- גרסה 55 ומעלה
* Safari- גרסה 11 ומעלה

Microsoft Edge- גרסה 16 ומעלה

## **10.2. מודול המערכת**

* המערכת עוסקת בזיהוי מבוך בתמונה.
* המערכת עוסקת בבניית הגרף ע"פ התמונה.
* המערכת עוסקת בפתרון המבוך.

## **10.3. אפיון פונקציונלי**

* העלאת תמונת מבוך ע"י המשתמש.
* שליחת התמונה לעיבוד תמונה, משם שליחת לפתרון המבוך, החזרתו לצד עיבוד תמונה משם למשתמש .

## **10.4. ביצועים עיקריים**

המערכת משתמשת בעיבוד תמונה ע"י אלגוריתמים , ועיבוד התמונה , אלגוריתמים לבניית גרף ולפתרון המבוך - אלגוריתם A סאטר.

## **10.5. אילוצים**

* המערכת מקבלת מבוכים מרובעים בלבד.
* על הכניסה והיציאה מהמבוך להיות פתוחים – מסגרת המבוך אינה סגורה.
* על המעברים במבוך להיות בהירים.
* אין להכניס תמונה עם מסגרת.

# **11. תיאור הארכיטקטורה**

## **11.1. הארכיטקטורה של הפתרון המוצע**

* בכניסת המשתמש בפעם הראשונה הוא יכניס את שמו ואת הסיסמה.
* לאחר מכן הוא יופנה לדף של העלאת תמונה מבוך.
* התמונה תישלח לשרת שיפנה לעיבוד וזיהוי המבוך.
* המבוך נשלח לפתרון
* הפתרון חוזר ומציג אותו בתמונה.
* השרת מחזיר את התמונה חוזרת למשתמש.

## **11.2. תיאור הרכיבים בפתרון**

שרת- flask:

Flask הוא סביבה לפיתוח web בשפת python, היא מיקרו-מסגרת- שומרת על גרעין מינימלי, מה שמאפשר למפתחים להרחיב את יכולות המערכת בעזרת תוספים.

מאפיינים עיקריים של flask:

* גמישות- flask נותן למפתחים חופש בבחירת הכלים והספריות שרוצים להשתמש בהם. הוא לא כופה ארכיטקטורה מסוימת כך שהמפתח יכול לבחור איך לארגן את הקוד.
* קלות השימוש- סביבה זו נחשבת לפשוטה וקלה לשימוש, היא מספקת דרך פשוטה ומהירה לבניית יישומים בלי הרבה סיבוכים.
* מינימליזם- כמיקרו-מסגרת flask מספקת רק את הכלים הבסיסיים הדרושים לפיתוח web, כמו ניתוב URL ויכולת לטפל בבקשות HTTP, תכונות מתקדמות יותר ניתן להוסיף באמצעות תוספים.

שימושים נפוצים של Flask:

* יצירת API.
* פיתוח אתרים ודפי אינטרנט.
* יישומים מבוססי נתונים.
* פיתוח מערכות קטנות עד בינוניות שצריכות גמישות ותפעול מהיר.

## **11.3. ארכיטקטורת הרשת**

אין

## **11.4. תיאור פרוטוקולי התקשורת**

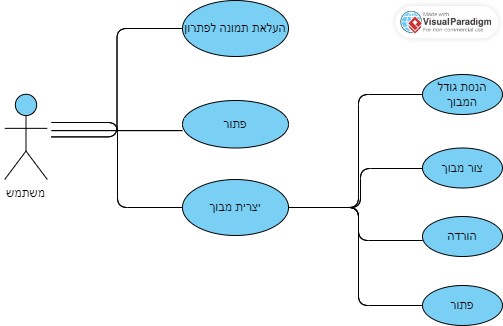
HTTP

## **11.5. שרת-לקוח**

angular

# **12. ניתוח ותרשים UML / Use cases של המערכת המוצעת**

## **12.1. תיאור ה-UC העיקריים של המערכת**



## **12.2. הצגת use caseעבור כל הפונקציות העיקריות בפרויקט**

1. **העלאת תמונה לפתרון**:

* המשתמש מעלה תמונת מבוך למערכת לצורך פתרון.

1. **פתור**:

* המערכת לוקחת את התמונה שהועלתה, מעבדת אותה, מוצאת את הפתרון ומחזירה את התוצאה למשתמש.

1. **יצירת מבוך**:

* המשתמש מזין את גודל המבוך ומבקש ליצור מבוך חדש.
* המערכת יוצרת את המבוך לפי הגדרות המשתמש.
* לאחר יצירת המבוך, המשתמש יכול להוריד את המבוך או לבקש מהמערכת לפתור אותו.

**התייחסות לפונקציות העיקריות בפרויקט:**

* העלאת תמונה לפתרון: קלט ראשוני מהמשתמש.
* פתרון המבוך: עיבוד נתונים מורכב שכולל עיבוד תמונה, בניית גרף והרצת אלגוריתם.
* יצירת מבוך: יכולת נוספת שמאפשרת למשתמש ליצור מבוך מותאם אישית.

## **12.3. מבנה נתונים בפרויקט**

**המטריצה**:

* + מטריצה בינארית בגודל MxN (M שורות ו-N עמודות).
  + ערך התא: 1 מייצג קיר, 0 מייצג מעבר.
  + משמשת לאחסון תצורת המבוך לאחר עיבוד התמונה.

**שימושים**:

* + המרת תמונת המבוך למטריצה בינארית.
  + זיהוי הקירות והמעברים במבוך.
  + שליחת המטריצה לשרת ה-API לצורך עיבוד נוסף.

**גרף (Graph)**

גרף הוא מבנה נתונים המורכב מצמתים וקשתות, ומשמש לייצוג המבוך בצורה המאפשרת חיפוש מסלול בצורה יעילה.

* **תיאור הגרף**:
  + צמתים (Vertices): נקודות פיצול או שינוי כיוון במבוך.
  + קשתות (Edges): מעברים בין הצמתים.
  + מיוצג כגרף בלתי ממושקל, כאשר כל קשת מייצגת מעבר אפשרי בין שני צמתים סמוכים.
* **שימושים**:
  + ייצוג המבוך כגרף כדי לאפשר חיפוש מסלול.
  + בניית הגרף מהמטריצה הבינארית על ידי זיהוי צמתים וקשתות.
  + הרצת אלגוריתמי חיפוש בגרף למציאת המסלול הקצר ביותר.

**רשימה של ערמת מינימום (Priority Queue)**

רשימה של ערמת מינימום היא מבנה נתונים המשמש לאחסון ולשליפה יעילה של הצומת עם המשקל הנמוך ביותר (המרחק הקצר ביותר) במהלך הרצת האלגוריתם למציאת המסלול הקצר.

* **תיאור רשימת ערמת מינימום**:
  + מאחסנת צמתים עם המשקל הנמוך ביותר בעדיפות ראשונה.
  + מעדיפה את הצמתים לפי מרחק או עלות מצטברת מהצומת ההתחלתי.
* **שימושים**:
  + במהלך הרצת האלגוריתם דייקסטרה או A\* למציאת המסלול הקצר ביותר.
  + שמירה ושליפה יעילה של הצומת הבא לבדיקה.

## **12.4. הקשרים בין היחידות השונות**

בפרויקט זה ישנן 5 יחידות:

* **צד לקוח - Angular**:
  + ממשק משתמש להעלאת תמונה וקבלת פתרון.
* **צד שרת - Python**:
  + קבלת התמונה מהלקוח, המרתה למטריצה בינארית, עיבוד תמונה.
* **צד שרת WEB C#:**
  + קבלת המטריצה וגודל התאים.
* **מודל בניית גרף**:
  + המרת המטריצה לגרף לצורך חיפוש מסלול.
* **אלגוריתם חיפוש מסלול**:
  + מציאת המסלול הקצר ביותר בין נקודת הכניסה והיציאה במבוך.

**הקשר ביניהן:**

* **צד לקוח** שולח את התמונה ל**צד שרת**.
* **צד שרת** מעבד את התמונה ושולח את המטריצה ל**מודל בניית גרף**.
* **השרת WEB** מקבל את המטריצה**.**
* **מודל בניית גרף** שולח את הגרף ל**אלגוריתם חיפוש מסלול**.
* **אלגוריתם חיפוש מסלול** מוצא את המסלול הקצר ושולח חזרה את המידע ל**צד שרת**.
* **צד שרת** מחזיר את התמונה עם המסלול המסומן ל**צד לקוח**.

## **12.5. עץ מודולים**

אין

## **12.6. תרשים המחלקות בפרויקט**

class PriorityQueue<vertax>

}

Vertex[] heap;

int size;

int capacity;

{

class MazeRequest

}

List<List<int>> Matrix\_maze

int Wall

Int Transition

{

class Edge

}

int weight

int indexV

{

class Vertex

}

int x\_value

int y\_value

int x\_value

int father

int fatherEnd

List<Edge> Neighbours

int costTotal

int costEnd

int costStart

int index

bool visited

{

class graph

}

List<Vertex> LValue

int countdV

{

## **12.7. תיאור המחלקות המוצעות**

class graph

}

List<Vertex> LValue

int countdV

{

**מחלקת גרף** מכילה רשימה של צמתים ואינקס. במחלקה כתובים הפונקציות הסטנדרטיות לגרף, וכן האלגוריתם למציאת המסלול הקצר.

**Edge ומחלקת Vertex מחלקת**

מכילות את המתודות הסטנדרנטיות לצומת וקשת בגרף.

**PriorityQueue מחלקת**

HEAPFY מכילה את כל המתודות הקשורות לערימה, הכנסה, הוצאה החלפה,

MazeRequest **מחלקת**

מחלקה זו משמשת לקבלת קובץ משרת בפיתון, שהוא מכיל את המטריצה המתקבלת מעיבוד התמונה   
את גודל הקיר וגודל מעבר.

# **13. רכיבי ממשק**

בפרויקט זה ממשק המשמש נבנה ב angular.

**הרכיבים בממשק זה:**

* מסך הבית
* מסך אודות בו המשתמש קורא אודות המבוך.
* מסך פתירת מבוך בו המשתמש מעלאה תמונה של מבוך והמערכת תחזיר לו אותו פתור.
* מסך יצירת מבוך בו המשתמש מכניס אורך ורוחב למבוך והמערכת תיצור לו את המבוך , ותאפשר לו להוריד אותו וכן להחזיר את הפתרון.

ישנו תפריט המאפשר למשתמש לנווט בין המסכים.

# **14. תיכון המערכת**

## **14.1. ארכיטקטורת המערכת**

במערכת זו ישנם 5 רכיבים:

* צד לקוח- Angular.
* צד שרת- Python.
* כיבוד תמונה - python.
* צד שרת – web api
* פתרון המבוך – C#

תיאור:

* צד לקוח – רכיב זה אחראי לממשק המשתמש, איסוף התמונה ושליחתה לשרת, ומקבל תוצאות מהשרת.
* צד שרת- מקבל בקשות מצד הלקוח, מעבד את התמונה ושולח לשרת web
* שרת web מעבד נתונים מוצא את המסלול ומחזיר לשרת python.
* פיתון מקבל את המטריצה ממיר לתמונה ומחזיר ללקוח.

## **14.2. תיכון מפורט**

**תיאור הרכיבים הפנימיים:**

* **עיבוד תמונה**: מודול זה מטפל בעיבוד תמונת המבוך שהמשתמש מעלה. התמונה מומרת למטריצה בינארית שבה הקירות והמעברים מוגדרים לאפס ואחד.
* **בניית גרף**: מודול זה משתמש במטריצה הבינארית שנוצרה בעיבוד התמונה כדי לבנות גרף של המבוך. כל צומת בגרף מייצג נקודת פיצול או שינוי כיוון, והקשתות מייצגות את המעברים בין הצמתים.
* **אלגוריתם חיפוש מסלול**: A star אלגוריתם זה משתמש בגרף המבוך כדי למצוא את המסלול הקצר ביותר בין נקודת הכניסה לנקודת היציאה. האלגוריתם מבוסס על שני threads שרצים מהכניסה ומהיציאה עד שהם נפגשים, ומסמנים את הנתיב הקצר ביותר.

**ממשקים:**

**ממשק בין צד הלקוח לצד השרת:**

בשרת ובלקוח ישנן פונקציות השולחות ומקבלות נתונים ומחזירות תוצאות. להלן הפונקציות הקיימות:

* **העלאת תמונה לפתרון**: צד הלקוח מעלה תמונת מבוך בפורמט JSON. צד השרת מקבל את התמונה, מעבד אותה למטריצה בינארית ושולח חזרה ללקוח.
* **עיבוד תמונה**: צד השרת מקבל את המטריצה הבינארית, מזהה את הקירות והמעברים, ושומר את הנתונים במסד נתונים.
* **בניית גרף**: צד השרת מקבל את המטריצה ומבצע בניית גרף מהמבוך, כאשר כל צומת מייצג נקודת פיצול או שינוי כיוון.
* **חיפוש מסלול**: צד השרת מפעיל את אלגוריתם חיפוש המסלול, שמוצא את המסלול הקצר ביותר בין נקודת הכניסה לנקודת היציאה. האלגוריתם מחזיר את המסלול המסומן.
* **הצגת פתרון**: צד השרת שולח את התמונה עם המסלול המסומן בחזרה ללקוח, והלקוח מציג את הפתרון למשתמש.

## **14.3. חלופות המערכת**

# **15. תיאור התוכנה**

## **15.1. סביבת עבודה**

עבדתי ב – 2 תוכנות:

PyCharm ו- visual studio

את עיבוד התמונה כתבתי בסביבת העבודה pycharm

ואת בנית גרף, ומציאת המסלול הקצר ב vusual studio 2022

## **15.2. שפות תכנות**

שפות התכנות איתן השתמשתי הינם:

Python- עיבוד התמונה.

Angular- בנית הגרף ומציאת המסלול הקצר בc#.

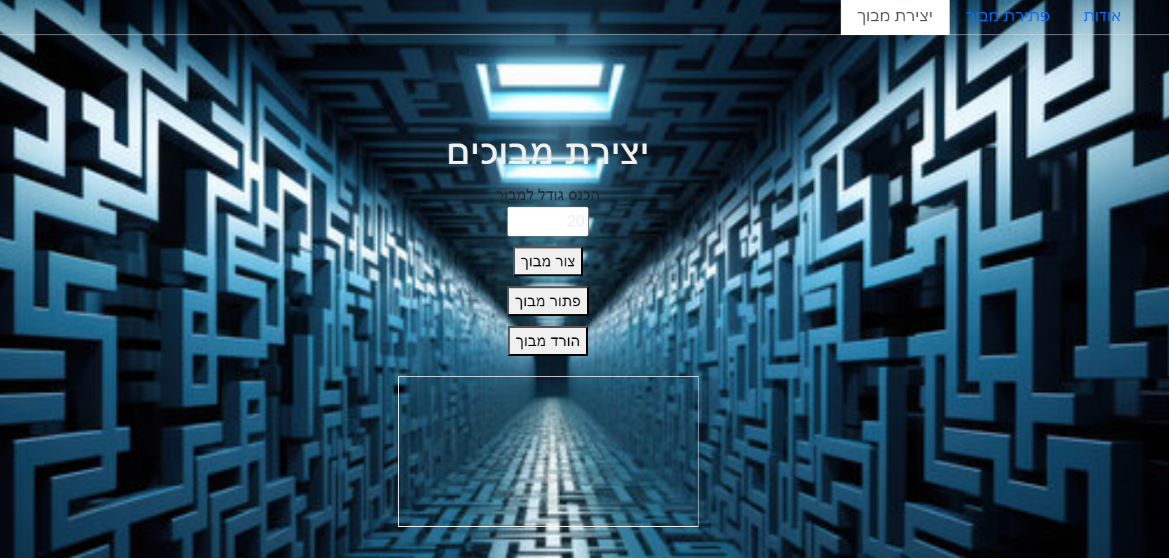
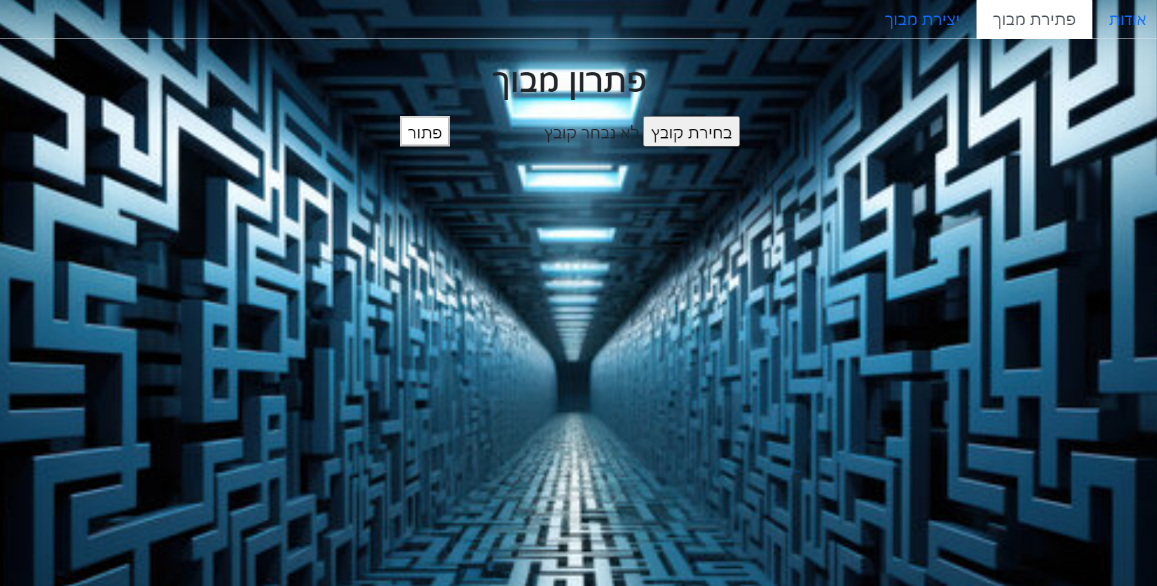
# **16 . תיאור מסכים**

המסכים שיש בפרויקט:

* מסך הבית.
* מסך אודות- הסבר למשתמש על האתר.
* מסך פתרון מבוך – בו המשתמש מעלאה תמונה לפתרון.
* מסך יצרית מבוכים בו המשתמש יוצר מבוך לפי גודל .

# **17. תרשים מסכים המתאר את היררכיית המסכים**

# **18. פרוט המסכים**



# **19. קוד התוכנית + תיעוד**

עיבוד תמונה:

* פונקציה המקבלת נתיב לתמונה, פותחת אותה באפור ומחזירה אותה ואת המטריצה. סיבוכיות o(m\*n)

def GetMetrixImages(pathImages):  
  
 image = Image.open(pathImages).convert("L")  
  
 binary\_image = image.point(lambda pixel: 0 if pixel < 240 else 1, mode="1")  
  
 pixel\_matrix = binary\_image.load()  
  
 return pixel\_matrix, binary\_image

* מחזירה את את אורך הקיר ואורך המעבר

def GetSub(maze):  
 width, height = maze.shape  
 for i in range(width):  
 for j in range(height):  
 print(maze[i][j], end=" ")  
 print()  
 wall = 0  
 common\_wall = [0] \* height  
 for x in range(width - 1):  
 for y in range(height - 1):  
 if maze[x, y] == 0:  
 i = x  
 j = y  
 wall\_i = 0  
 wall\_j = 0  
 while maze[i, y] == 0:  
 if i > width - 2:  
 break  
 wall\_i += 1  
 i += 1  
 while maze[x, j] == 0:  
 if j == height - 2:  
 break  
 wall\_j += 1  
 j += 1  
 if wall\_i > wall\_j:  
 wall = wall\_j  
 else:  
 wall = wall\_i  
 break  
 if wall != 0:  
 common\_wall[wall] += 1  
 max = 0  
 i\_max = 0  
 for i, h in enumerate(common\_wall):  
 if i > 0:  
 if h > max:  
 max = h  
 i\_max = i  
 wall = i\_max

transition = 0  
common\_t = [0] \* height  
for x in range(width - 1):  
 for y in range(height - 1):  
 if maze[x, y] == 1:  
 i = x  
 j = y  
 transition\_i = 0  
 transition\_j = 0  
 while i < width - 1 and maze[i, y] == 1:  
 if i > width - 1:  
 break  
 transition\_i += 1  
 i += 1  
 while maze[x, j] == 1:  
 if j == height - 1:  
 break  
 transition\_j += 1  
 j += 1  
 if transition\_i > transition\_j:  
 transition = transition\_j  
 else:  
 transition = transition\_i  
 break  
 if transition != 0:  
 common\_t[transition] += 1  
max = 0  
i\_max = 0  
c = 1  
if wall == 1:  
 c = 0  
for i, h in enumerate(common\_t):  
 if i > c:  
 if h > max:  
 max = h  
 i\_max = i  
transition = i\_max  
return transition, wall

wall: עובי הקיר הנפוץ ביותר במבוך.

transition: רוחב המעבר הנפוץ ביותר במבוך.

חישוב עובי הקירות (wall):

אתחול משתנים.

מעבר על כל הפיקסלים במבוך.

כאשר מוצאים תא שהוא חלק מקיר (maze[x, y] == 0), הפונקציה מודדת את עובי הקיר לאורך צירי ה-X וה-Y.

מעדכנת את המערך common\_wall, הסופר את תדירות כל עובי קיר שנמצא.

בסיום המעבר, מוצאת את עובי הקיר הנפוץ ביותר.

חישוב רוחב המעברים (transition):

אתחול משתנים.

מעבר נוסף על כל הפיקסלים במבוך.

כאשר מוצאים תא שהוא חלק ממעבר (maze[x, y] == 1), הפונקציה מודדת את רוחב המעבר לאורך צירי ה-X וה-Y.

מעדכנת את המערך common\_t, הסופר את תדירות כל רוחב מעבר שנמצא.

בסיום המעבר, מוצאת את רוחב המעבר הנפוץ ביותר.

בסוף, הפונקציה מחזירה את הערכים transition (רוחב המעבר הנפוץ ביותר) ו-wall (עובי הקיר הנפוץ ביותר).

סיבוכיות:  
O(width^2×height+width×height^2)

* פונקציות המוצאת את נק' ההתחלה והסיום של מסגרת המבוך.

#פונקציה

def find\_point\_start(pixel\_matrix, width, height):  
 start = (-1, -1)  
 for i in range(width):  
 for j in range(height):  
 if pixel\_matrix[i, j] == 0:  
 k = j  
 while (k < height and pixel\_matrix[i, k] == 0):  
 k += 1  
 l = i  
 while (l < height and pixel\_matrix[l, j] == 0):  
 l += 1  
 if k - j > height \* 0.6 or l - i > width \* 0.6:  
 start = (i, j)  
 break  
 if start != (-1, -1):  
 break  
 return start

def find\_point\_end(pixel\_matrix, width, height):  
 finish = (-1, -1)  
 for i in range(width - 1, -1, -1):  
 for j in range(height - 1, -1, -1):  
 if pixel\_matrix[i, j] == 0:  
 k = j  
 while k > 0 and pixel\_matrix[i, k] == 0:  
 k -= 1  
 l = i  
 while l > 0 and pixel\_matrix[l, j] == 0:  
 l -= 1  
 if (width - (j - k)) >= width \* 0.6 or (height - (i - l)) >= height \* 0.6:  
 finish = (i, j)  
 break  
 if finish != (-1, -1):  
 break  
 return pixel\_matrix, finish

O(width×height2) הסיבוכיות במקרה הגרוע ביותר- לא נמצא התחלה \ סיום.

פונקציית בניית הגרף:

public static void build\_graph(int[,] wallx, int[,] wally,Vertex s,Vertex f)

{

Vertex t = new Vertex();

int c = 0, c1 = 0, int indexGraph=0;

for (int i = 0; i < mazeMatrix.GetLength(0); i++)//מציאת התחלה וסיום

{

if (mazeMatrix[0, i] == 1)

{

s = new Vertex(0, i, indexGraph++);

graph.add(s);

}

if (mazeMatrix[mazeMatrix.GetLength(0), i] == 1)

{

f = new Vertex(mazeMatrix.GetLength(0), i, 0);

}

}

bool flag = false;

for (int i = 0; i < mazeMatrix.GetLength(0); i++)//מציאת שאר הצמתים

{

c = 0;

c1 = 0;

for (int j = 0; j < mazeMatrix.GetLength(1); j++)

{

if (i > 0 && i < mazeMatrix.GetLength(0) && j > 0 && j < mazeMatrix.GetLength(0))

if (mazeMatrix[i, j] == 1)

{

flag = false;

//אם הנקודה היא רק מעבר בין צמתים היא לא צומת

if ((mazeMatrix[i - 1, j] == 1 && mazeMatrix[i + 1, j] == 1 &&

mazeMatrix[i, j - 1] == 0 && mazeMatrix[i, j + 1] == 0) ||

(mazeMatrix[i - 1, j] == 0 && mazeMatrix[i + 1, j] == 0 &&

mazeMatrix[i, j - 1] == 1 && mazeMatrix[i, j + 1] == 1))

flag = true;

// אם היא צומת

if (flag == false)

{

Vertex v = new Vertex(i, j, indexGraph++);

v.DistanacPoint(s,f);

graph.add(v);

}

}

if (mazeMatrix[i, j] == 0)

{

wallx[i, c++] = j;

}

if (mazeMatrix[j, i] ==0)

{

wally[i, c1++] = j;

}

}

}

f.index = indexGraph;

s.DistanacPoint(s,f);

f.DistanacPoint(s,f);

graph.add(f);

}

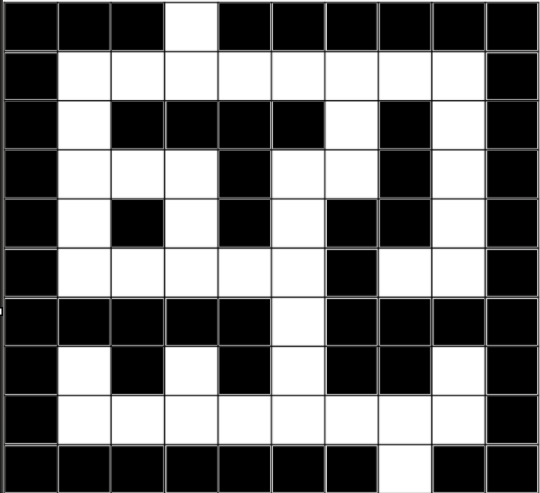
אל פונקציית בניית הגרף נשלחים 4 עצמים ריקים שיתעדכנו במהלך בניית הגרף.

שני העצמים wallx, wally הינם מטריצות בגודל המבוך, והתעדכנו בהם אינדקסי קירות במבוך (לצורך פונקציה מציאת השנים של כל צומת).

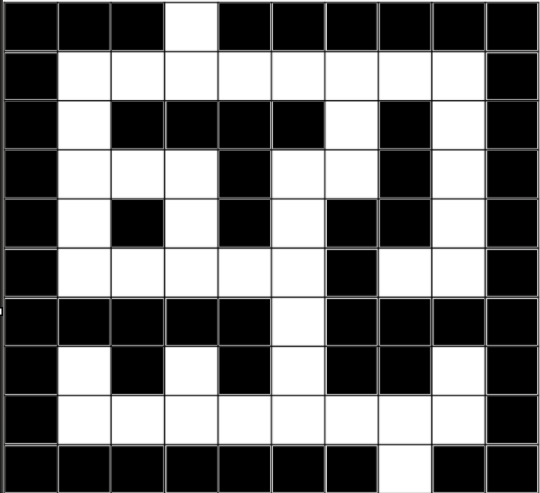
בצורה כזו שקיר שנמצא באינדקס (4,5) יעודכן בwallx כך: בשורה שבהi = 4 , יכנס 5 .   
וב- wally : בשורה שבה i = 5 יכנס 4. כמובן הקירות יכנסו לפי סדר וכך כל שורה תהיה ממוינת.

העצמיםs ו- f הינם התחלה וסיום המבוך.

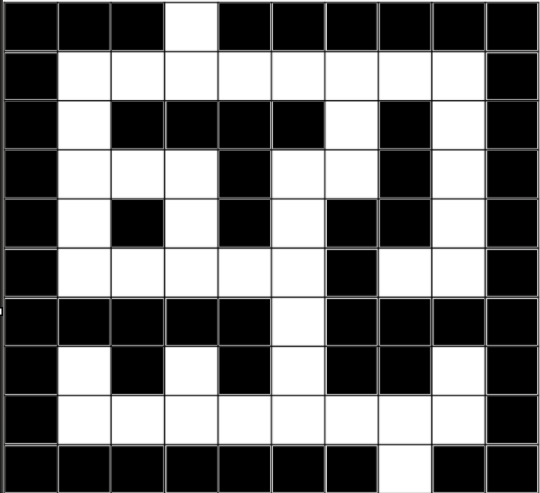
בלולאה הראשונה הפונקציה מוצאת את התחלת וסיום המבוך.

הלולאה השנייה רצה על כל מטריצת המבוך הבינארית, כאשר היא מגיעה ל 1 – למעבר במבוך, היא בודקת האם המעבר הזה הוא צומת, צומת הוא מעבר שאינו אנכי או אופקי, לדוגמא:

כפי הניתן לראות המשבצות המודגשת הינן מעבר אנכי / אופקי - שכניהם נמצאים רק בצדדים או רק למעלה ולמטה, ולכן היא אינן צומת.



לעומת זאת המשבצות הללו – יש להן שכנים גם מכיוונים שונים , ולכן הן צמתים.

בפונקציה זו אני מזהה את הצמתים ורק אותם מכניסה לגרף שלי , בצורה כזו נחסך המון צמתים שהינם רק מעבר בגרף.

לצורך ההבנה אדגים כמה צמתים אמורים היו להיות בגרף אילולי הינו משתמשים בדרך להעיל,  
43 צמתים (!) – כל משבצת לבנה היא צומת.  
עם השיטה שלי למעלה, ישנן רק 22 צמתים.

סיבוכיות זמן ריצה O(n×m)

פונקציות בניית השכנים:

public static void findAllNeighbours(int[,] wallx, int[,] wally,List<Vertex> lSX, List<Vertex> lSY)

List<Vertex> temp ;

int i = 0;

int vTemp = lSX[i].x\_value;

//מציאת שכנים לצדדים

while (i < lSX.Count)

{

temp = lSX.Where(x=>x.x\_value==vTemp).ToList();

temp.Sort((a, b) => a.y\_value.CompareTo(b.y\_value));

find(wallx, temp,'x');

if (i+temp.Count < lSX.Count)

{

vTemp = lSX[i+temp.Count].x\_value;

i+=temp.Count;

}

else

i=lSX.Count;

}

i = 0;

vTemp = lSY[i].y\_value;

temp = new List<Vertex>();

//מציאת שכנים למעלה ולמטה

while (i < lSY.Count)

{

temp = lSY.Where(x => x.y\_value == vTemp).ToList();

temp.Sort((a, b) => a.x\_value.CompareTo(b.x\_value));

find(wally, temp, 'y');

if (i + temp.Count < lSX.Count)

{

vTemp = lSY[i + temp.Count].y\_value;

i += temp.Count;

}

else

i = lSY.Count;

}

}

פונקציה זו מקבלת את הפרמטרים :  
wallx, wally - שני המטריצות שנבנו בבניית הגרף.

Lsx, lsy – שתי רשימות המכילות העתק של הגרף , רק שהן ממוינות על פי האינדקסים שלהן. Lsx - ממוין על פי הx - , ו lsy - ממוין על פי הy - .   
בעזרתם אמצא שכנים לצדדים – X ולמעלה ולמטה – Y.

האלגוריתם רץ על הרשימה הממוינת LSX) ) ויוצרת תת רשימה (temp) בה בכל הצמתים אינדקס X הינו זהה. את הרשימה הזו ממיינים כעת על פי Y .   
כעת אני שולחת לפונקציה find את ה'תת רשימה' למצוא לכל איבר בה שכנים.

את אותו הדבר גם לרשימה lsy.

סיבוכיות: O(n2logn)

הפונקציה find:

public static void find(int[,] wall, List<Vertex> lv, char type\_)

{

for (int i = 0; i < lv.Count; i++)

{

if (i==0 && i+1 < lv.Count)

isNeighbors (wall, lv[i],null, lv[i+1],type\_);

else if(i==lv.Count-1 && i > 0)

isNeighbors (wall, lv[i],lv[i - 1] ,null, type\_);

else if (i>0 && i < lv.Count)

{

isNeighbors(wall, lv[i], lv[i-1], lv[i + 1], type\_);

}

}

}

הפונקציה מקבלת את מטריצת העזר של הקירות במבוך, התת רשימה וסוג – כעת אני מחפשת שכנים מסוג X או סוג Y.

תפקידה של פונקציה זו היא לשלוח לכל צומת את השכנים שלו מצדדיו לפונקציה isNeighbors שתבדוק האם הם שכנים.

public static void isNeighbors(int[,] wall, Vertex v, Vertex NRight, Vertex Nleft,char type\_)

{

if (type\_=='x')

{

if (NRight!=null && binaryFind(wall, v.x\_value, NRight.y\_value, v.y\_value) == false)

v.addNeighbours(NRight.index, v.y\_value - NRight.y\_value);

if (Nleft!=null && binaryFind(wall, v.x\_value, v.y\_value, Nleft.y\_value) == false)

v.addNeighbours(Nleft.index, Nleft.y\_value-v.y\_value);

}

else

{

if (NRight != null && binaryFind(wall, v.y\_value, NRight.x\_value, v.x\_value) == false)

v.addNeighbours(NRight.index, v.x\_value - NRight.x\_value);

if (Nleft != null && binaryFind(wall, v.y\_value, v.x\_value, Nleft.x\_value) == false)

v.addNeighbours(Nleft.index, Nleft.x\_value - v.x\_value);

}

}

הפונקציה מקבלת שכן נוכחי ואת שכניו מימין ושמאל ובודקת באמצעות הפונקציה binaryFind האם הם שכנים או שמפריד ביניהם קיר. אם הם שכנים היא מוסיפה לצומת הנוכחי שכן.

הפונקציה :binaryFind

public static bool binaryFind(int[,] Iwall, int i, int IVarMIn, int IVarMax)

{

int countArr = 1;//המספר הראשון הוא 0 , במקרה שזהו פתח המבוך אז אז יש קיר בסוף

while (countArr < Iwall.GetLength(1) && Iwall[i, countArr] != 0)

countArr++;

int left = 0, right = countArr - 1;

while (left <= right)

{

int middle = left + (right - left) / 2;

if (Iwall[i, middle] > IVarMIn && Iwall[i, middle] < IVarMax)//בדיקה האם המספר בטווח

return true;

else if (Iwall[i, middle] < IVarMIn)

left = middle + 1;

else

right = middle - 1;

}

return false;

}

פונקציה זו משתמשת במטריצת הקירות שיצרנו עם בניית הגרף, ומבצעת על השורה המתאימה חיפוש בינארי בין שני אינדקסים של שכנים האם יש מספר ביניהם (קיר) – אם כן הם אינם שכנים.

הסבר על האלגוריתם A star :

את אלגוריתם A\* הפעלתי בשני threads האחד יתחיל מתחילת המבוך, והשני מסוף המבוך.

סיבוכיות כוללת לכל הפונקציות הללו: O(n2logn)היות והפונקציה findAllNeighbours קוראת לכל הפונקציות ומשתמשת בהם.

לצורך כך הוספתי למחלקה של צומת Vertex עוד משתנה fatherEnd בנוסף על המשתנה father, את הפונקציה כתבתי פעמים שההבדל ביניהם הוא בהתייחסות לאבא.

List<int> startPath = new List<int>();

List<int> endPath =new List<int> ();

Mutex mutex = new Mutex();

// יצירת תהליכים

Thread thread1 = new Thread(() =>

{

// פונקציה שתחשב את הנתיב מההתחלה לסוף

startPath = graph.startAStar(0, graph.countdV - 1);

});

Thread thread2 = new Thread(() =>

{

// פונקציה שתחשב את הנתיב מהסוף להתחלה

endPath = graph.endAStar(graph.countdV - 1, 0);

});

// הפעלת התהליכים

thread1.Start();

thread2.Start();

// המתנה לסיום התהליכים

thread1.Join();

thread2.Join();

סיבוכיות O(ElogV). שימוש בפונקצית A\*

public List<int> startAStar(int start, int end)

{

List<int> path = new List<int>();

PriorityQueue<Vertex> openSet = new PriorityQueue<Vertex>(countdV);

PriorityQueue<Vertex> closeSet = new PriorityQueue<Vertex>(countdV);

this.LValue[start].costStart = 0;

this.LValue[start].costTotal = this.LValue[start].costEnd;

openSet.Insert(this.LValue[start]);

int cost = 0;

while (!openSet.IsEmpty)

{

Vertex currentV = openSet.ExtractMin();

if (currentV.index == end || currentV.visited)

{

currentV = this.LValue[currentV.father];

while (currentV.index != start)

{

path.Add(currentV.index);

currentV = this.LValue[currentV.father];

}

path.Add(start);

path.Reverse();

return path;

}

//not goal:

currentV.visited = true;

closeSet.Insert(currentV);

if (currentV.father != -1)

cost = this.LValue[currentV.father].costStart;

foreach (Edge n in currentV.Neighbours)

{

Vertex neighbor = this.LValue[n.indexV];

int newCost = cost + n.weight;

if (openSet.Contains(neighbor) || closeSet.Contains(neighbor))

{

if (neighbor.costStart > newCost)

{

neighbor.costStart = newCost;

neighbor.costTotal = neighbor.costEnd + neighbor.costStart;

neighbor.father = currentV.index;

if (openSet.Contains(neighbor))

openSet.update(neighbor);

if (closeSet.Contains(neighbor))

closeSet.update(neighbor);

}

}

else

{

neighbor.costStart = newCost;

neighbor.costTotal = neighbor.costEnd + neighbor.costStart;

neighbor.father = currentV.index;

openSet.Insert(neighbor);

}

}

}

return new List<int>();

}

הסבר על האלגוריתם:  
אתחול:  
- האלגוריתם מתחיל באתחול שתי רשימות: רשימה פתוחה ורשימה סגורה. הרשימה הפתוחה מכילה צמתים שטרם נחקרו, והרשימה הסגורה מכילה צמתים שכבר הוערכו.  
- האלגוריתם גם מקצה עלות להגיע לכל צומת מצומת ההתחלה ועלות משוערת להגיע לצומת המטרה מכל צומת. עלויות אלו משמשות לקביעת עדיפות הצמתים ברשימה הפתוחה.  
  
לולאה ראשית:  
כל עוד שהרשימה הפתוחה אינה ריקה, האלגוריתם ממשיך לחזור על השלבים הבאים:  
- בחר את הצומת עם העלות הכוללת הנמוכה ביותר (עלות הגעה לצומת מההתחלה + עלות משוערת להגיע ליעד) מהרשימה הפתוחה. הצומת הזה הוא הצומת הנוכחי.  
- אם הצומת הנוכחי הוא צומת המטרה או שביקרתי בו – ע"י הthread השני , האלגוריתם משחזר את הנתיב מצומת ההתחלה לצומת המטרה או לצומת הנוכחי באמצעות צמתי האב ומסתיים.  
- אחרת, האלגוריתם מרחיב את הצומת הנוכחי על ידי התחשבות בצמתים הסמוכים לו (צמתים סמוכים) וחישוב העלויות שלהם.  
- עבור כל צומת שכן:  
- אם הצומת אינו ברשימה הפתוחה, הוסף אותו לרשימה הפתוחה והגדר את הצומת הנוכחי כהורה שלו. עדכן את העלויות עבור הצומת.  
- אם הצומת נמצא ברשימה הפתוחה והנתיב החדש אליו קצר יותר, עדכן את העלויות והאבא של הצומת.  
- לבסוף, העבר את הצומת הנוכחי לרשימה הסגורה.  
  
סיום:  
- האלגוריתם מסתיים כאשר מגיעים לצומת המטרה או ששני הtreads נפגשו (נמצא נתיב) או שהרשימה הפתוחה ריקה (לא קיים נתיב).  
סיבוכיות: O(ElogV)

# **20. תיאור מסד הנתונים**

אין מסד נתונים.

# **21. מדריך למשתמש**

לאחר כניסת המשתמש לאתר יש לו 2 אפשרויות:

* העלאת תמונה של מבוך.
* יצירת מבוך בגודל הרצוי.

לאחר שישנה תמונה לשליחה, המשתמש לוחץ על "פתור", התמונה תשלח לפתרון, ולבסוף תחזור למשתמש תמנה של המבוך פתורה בה דרך הפתרון צבועה באדום.

למשתמש יש אפשרות להוריד לעצמו את תמונת המבוך הפתורה.

משחק מהנה!!!

# **22. בדיקות והערכה**

עיבוד תמונה:

מציאת המבוך בתמונה שהעלה המשתמש.

האלגוריתם:

האלגוריתם מוצא את המסלול הקצר לפתירת המבוך ומחזיר אותו פתור למשתמש.

# **23. ניתוח יעילות**

עיבוד תמונה:

* GetMetrixImages - סיבוכיות o(m\*n)
* GetSub – סיבוכיות O(width^2×height+width×height^2)
* find\_point\_start, find\_point\_end – סיבוכיות: O(width×height2)

בניית גרף, והאלגוריתם:

* build\_graph – סיבוכיות: O(n×m)
* findAllNeighbours – סיבוכיות: O(n2logn)
* findAllNeighbours – סיבוכיות: O(n2logn) פונ' זו כוללת את הפונ': binaryFind, isNeighbors, find
* startAStar - O(ElogV)

# **24. אבטחת מידע**

אין צורך

# **25. מסקנות**

המסקנות שלי מהפרויקט:

* מבחינה אישית:
* למדתי שאני אוהבת ויודעת ללמוד באופן עצמאי.
* גיליתי שיש לי סבלנות.
* מבחינה מקצועית:
* למדתי הרבה על קריאת תמונה, עיבוד תמונה, וראיה ממוחשבת.
* למדתי לכתוב קודים באופן מדויק יותר ונכון יותר.
* למדתי לכתוב קוד מסודר וקריא.
* למדתי על מבנה הנתונים גרף על כל חלקיו.
* למדתי לחבר בין סביבות עבודה שונות.

# **26. פיתוחים עתידיים**

* התאמת האפליקציה גם לפלאפון נייד.
* פתירת מבוכים בצורות שונות.
* התאמת יצירת מבוכים לפי רמות קושי.
* האתר ישמור עבור כל משתמש את המבוכים שפתר.

# **27. ביבליוגרפיה**

* סרטון הסבר על האלגוריתם:

"A\* Pathfinding Algorithm." YouTube, 27 בפברואר 2018. זמין ב: https://www.youtube.com/watch?v=ySN5Wnu88nE&t=403s&pp=ygUX16TXqteo15XXnyDXnteR15XXm9eZ150%3D. [גישה בתאריך 2 ביוני 2024].

* אלגוריתמים לפתרון מבוכים:

"אלגוריתמים לפתרון מבוכים." ויקיפדיה, 16 בינואר 2022. זמין ב: https://he.wikipedia.org/wiki/%D7%90%D7%9C%D7%92%D7%95%D7%A8%D7%99%D7%AA%D7%9E%D7%99%D7%9D\_%D7%9C%D7%A4%D7%AA%D7%A8%D7%95%D7%9F\_%D7%9E%D7%91%D7%95%D7%9B%D7%99%D7%9D. [גישה בתאריך 2 ביוני 2024].

* קריאת תמונה:

"קריאת תמונה באמצעות Pillow." רשתכיינג, 10 במרץ 2021. זמין ב: https://reshetech.co.il/python-tutorials/pillow-imaging-library.

"פייתון למתחילים." דאטה קוד, 5 ביולי 2021. זמין ב: https://data.cyber.org.il/python/python\_book.pdf.

* אלגוריתם A\*:

"על אלגוריתם A\*." רוטר, 3 בספטמבר 2019. זמין ב: https://rotter.name/nor/prog/15871.shtml.

* פרויקט ב-GitHub לפתרון מבוכים:

Pound, Mike. "Maze Solving." GitHub, 28 ביוני 2020. זמין ב: https://github.com/mikepound/mazesolving.

* סרטון הסבר על האלגוריתם:

"A\* Pathfinding Algorithm Explanation." YouTube, 23 ביוני 2015. זמין ב: https://www.youtube.com/watch?v=-L-WgKMFuhE. [גישה בתאריך 2 ביוני 2024].