

ניתוח ותיכון - פרויקט מסכם

בפרויקט זה תממשו משחק מבוך.

תקציר:

1. חלק א':
 - יצירת ספריות קוד
 - מפסאודו קוד לתכנות מונחה עצמים
 - Demo
2. חלק ב':
 - Streaming
 - שמירה לקבצים
 - ממשק משתמש CLI
3. חלק ג':
 - MVC
4. חלק ד':
 - Caching
5. חלק ה':
 - תוצרים

המטרה:

מטרת העל של הפרויקט היא לאפשר לכם לצבור ידע וניסיון בתכנון ומימוש מערכת משחק מורכבת, להיעזר בעקרונות התכנות מונחה העצמים, ולהטמיע את החומרים שלמדנו בקורס.

זכרו!

זאת ההזדמנות שלכם להשתדרג מבחינת יכולות כתיבת הקוד ומימוש המערכת. נצלו זאת והשקיעו מחשבה בפתרון האתגרים בפרויקט. בניגוד למבחן, הפרויקט מספק לכם דרך Hands-On להטמיע את החומרים הנלמדים וכמובן להוסיף עוד פרוייקט מעניין לפורטפוליו עבודות שלכם לקראת היציאה לתעשייה.

הגשה:

הגשת הפרויקט תתבצע בזוגות בלבד.

חלק א' - מפסאודו קוד לתכנות מונחה עצמים

כאשר אנחנו הופכים פסאודו קוד לתכנות מונחה עצמים ישנם שני כללים חשובים:

כלל ראשון: להפריד את האלגוריתם מהבעיה שהוא פותר

כשנתבונן בפסאודו-קוד של האלגוריתם נסמן את השורות שהן תלויות בבעיה. שורות אלה יגדירו לנו את הפונקציונליות הנדרשת מהגדרת הבעיה. את הפונקציונליות הזו נגדיר בממשק מיוחד עבור הבעיה הכללית. מאוחר יותר מחלקות קונקרטיות יממשו את הממשק הזה ובכך יגדירו בעיות ספציפיות שונות.

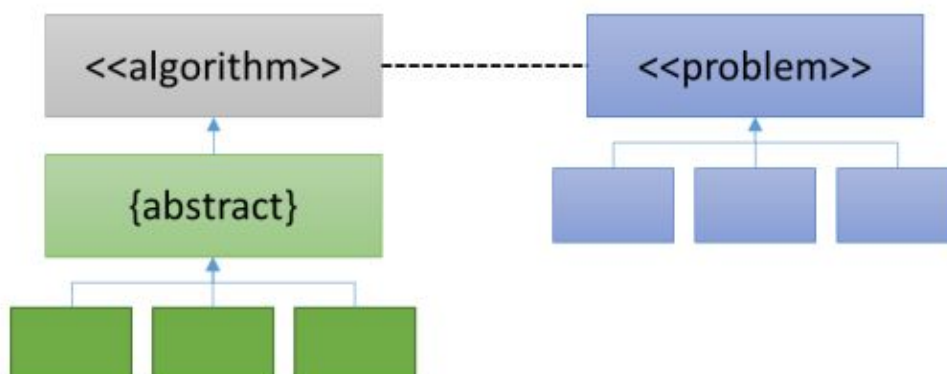
שמירה על כלל זה תאפשר לאלגוריתם לעבוד מול טיפוס הממשק במקום מול טיפוס של מחלקה ספציפית. תכונת הפולימורפיזם תאפשר לנו להחליף מימושים שונים לבעיה מבלי שנצטרך לשנות דבר בקוד של האלגוריתם.



כלל שני: לממש את האלגוריתם באמצעות היררכיית מחלקות.

ייתכנו אלגוריתמים נוספים שנצטרך לממש בעתיד, או מימושים שונים לאותו האלגוריתם שלנו. לכן כבר עכשיו ניצור היררכיית מחלקות שבה:

- את הפונקציונליות של האלגוריתם נגדיר בממשק משלו
 - את מה שמשותף למימושים השונים נממש במחלקה אבסטרקטית
 - את מה שלא משותף – נשאיר אבסטרקטי
 - את המימושים השונים ניצור במחלקות שירשו את המחלקה האבסטרקטית
 - הם יצטרכו לממש רק את מה ששונה בין האלגוריתמים
- את ההיררכיה הזו ניתן כמובן להרחיב ע"פ הצורך.
- קבלנו את המבנה הבא:



משימה א' - אלגוריתם ליצירת מבוך:

צרו מחלקה בשם Maze2d המייצגת מבוך דו-ממדי כבתיאור לעיל. הוסיפו מתודות למחלקה זו כרצונכם ע"פ הצורך והדרישות בהמשך. מי שהולך ליצור מופעים של Maze2d יהיה הטיפוס . Maze2dGenerator.

במשימה זו נתרגל את כתיבת ההיררכיה של המחלקות עבור אלגוריתם. את החלק של הבעיה נתרגל בחלק הבא של הפרויקט. 1

1. הגדירו ממשק בשם Maze2dGenerator שמגדיר

a. מתודה בשם generate שמחזירה מופע של Maze2d. תחליטו לבד מהם

הפרמטרים שהמתודה הזו צריכה לקבל. זכרו שעלינו להגדיר מבוך כללי.

b. מתודה בשם measureAlgorithmTime המחזירה מחרוזת. חישבו לבד מה

צריכים להיות הפרמטרים.

2. ממשו מחלקה אבסטרקטית כסוג של: Maze2dGenerator

a. היא תשאיר את המתודה generate אבסטרקטית. כל אלג' יממש זאת בעצמו.

b. לעומת זאת, פעילות מדידת הזמן זהה לכל האלגוריתמים ולמעשה אינה תלויה

באלגוריתם עצמו. לכן אותה דווקא כן נממש כאן (במקום לממש אותה כקוד כפול

בכל אחת מהמחלקות הקונקרטיות). המתודה measureAlgorithmTime תדגום

את שעון המערכת, תפעיל את generate, ותדגום את הזמן שוב מיד לאחר מכן.

הפרש הזמנים מתאר את הזמן שלקח להפעיל את generate. החזירו את זמן זה

כמחרוזת שנוחה לקריאה.

3. ממשו מחלקה בשם SimpleMaze2dGenerator (שתירש את המחלקה האבסטרקטית)

שפשוט מפזרת קירות בצורה אקראית. צרו מסלול אקראי של חללים ריקים (ערך 0)

מאיזושהי כניסה אקראית לאיזושהי יציאה אקראית כדי להבטיח שלמבוך יש פתרון.

4. למידה עצמית – עיקר התרגיל. היכנסו לעמוד הבא בוויקיפדיה:

https://en.wikipedia.org/wiki/Maze_generation_algorithm

בחרו את אחד האלגוריתמים שאתם מתחברים אליו יותר. באמצעותו ממשו מחלקה בשם

MyMaze2dGenerator המייצרת מבוך תלת-ממדי ע"פ הייצוג לעיל.

משימה ב' - בדיקות:

בדיקת תקינות האלגוריתמים.

הסיקו מכאן מה עליכם לממש על מנת לתמוך ביכולות המבוך והאלגוריתם.

```
class TestMazeGenerator
{
public:
    TestMazeGenerator() {}

    void testMazeGenerator(Maze2dGenerator& mg)
    {
        // prints the time it takes the algorithm to run
        cout << mg.measureAlgorithmTime(**/) << endl;

        // generate another 2d maze
        Maze2d maze = mg.generate(/* your parameters */);

        // get the maze entrance
        Position p = maze.getStartPosition();

        // print the position - format {x, y}
        cout << p << endl;

        // get all the possible moves from a position
        String[] moves = maze.getPossibleMoves(p);

        for (String move : moves)
            cout << move << endl;

        cout << maze.getGoalPosition() << endl;

        cout << maze << endl;
    }
};

int main()
{
    TestMazeGenerator t;
    TestMazeGenerator t2;
    t.testMazeGenerator(new simpleMaze2dGenerator());
    t2.testMazeGenerator(new MyMaze2dGenerator());

    return 0;
}
```

משימה ג' - אלגוריתמי פתרון:



1. ממשו את התשתית שראינו בתרגול והשלימו את המימוש של אלגוריתם BFS
2. ממשו אלגוריתם חיפוש נוסף בשם (A* - A Star) קראו למחלקה AStar

אלגוריתם A* הוא אלגוריתם חיפוש מסוג search first best. כאלגוריתם כזה, הוא מסדר את המצבים השונים בתור עדיפויות. ככל שהעדיפויות מוגדרות יותר טוב, כך האלגוריתם יצטרך לפתח פחות מצבים (להוציא פחות מצבים מהתור ולבצע חישובים) ולגלות את המסלול הטוב ביותר אל המטרה. אם היינו משתמשים בתור רגיל, אז היינו מפתחים את כל המצבים... BFS הגדיר את העדיפות של כל מצב כמשקל (הנמוך ביותר עד כה) של המסלול שמוביל מהמצב ההתחלתי אל המצב הזה. לכן BFS מפתח פחות מצבים מהאלגוריתם הקודם.

A* לעומתו, מגדיר את העדיפות של כל מצב ע"פ שני קריטריונים: א. כמו ב BFS, משקל המסלול עד לאותו המצב. ב. צפי \ הערכה מה יהיה המשקל של המסלול מהמצב הנוכחי אל מצב המטרה. לקריטריון הראשון מתייחסים בפסאודו קוד כפונקציה בשם $g()$ ואילו לשני כפונקציה בשם $h()$. השם h נבחר משום שמדובר ביוריסטיקה (heuristic) אמצעי עזר להערכה או לפתרון בעיות שלא בעזרת אלגוריתם. העדיפות תקבע ע"י $g+h=f$. ובפשטות, ככל שהמחיר להגעה למצב הוא זול יותר (g קטן יותר), וככל שאנו מעריכים שמהמצב הזה נגיע אל המטרה במחיר זול יותר (h קטן יותר), כך העדיפות של המצב בתור העדיפויות תהיה גבוהה יותר.

ההיגיון הוא שככל הנראה מצב מוצלח שכזה יהיה בוודאי חלק מהפתרון ולכן כדאי לנו לפתח אותו לפני מצבים אחרים. ישנה הוכחה לכך ש A* הוא הכי יעיל מבין אלגוריתמי החיפוש שבקטגוריה, כל עוד h היא אדמיסבילית כלומר קבילה. ומתי h היא קבילה? כאשר היא לא נותנת הערכה מופרזת, כלומר ההערכה שלה תמיד תהיה קטנה או שווה למשקל של המסלול האמיתי שייבחר בסוף.

לדוגמא: נניח ויש לנו את המבוך הבא שאסור לנוע בו באלכסון ומשקל כל תנועה היא 10.

			
		ב	
ג		א	

המצבים האפשריים הם א' ו ג'. לכל אחד מהם $g=10$. ולכן ב BFS תהיה להם אותה העדיפות. באיזה h נשתמש? יוריסטיקה המתאימה לבעיה זו היא "מרחק מנהטן", כלומר מרחק השורות +

מרחק העמודות בין המצב הנתון לבין למטרה. ממצב א' נהיה במרחק של שתי שורות ועמודה אחת $h=3*10$ ובסך הכל העדיפות של מצב א' תהיה $f=30+10=40$. לעומת זאת מצב ג' נמצא במרחק מנהטן של 5 ולכן ה f שלו תהיה $f=10+5*10=60$. אלג' A^* יכניס את מצב א' לתור העדיפויות לפני מצב ג' ולכן יצטרך לפתח פחות מצבים מה BFS. נשים לב שמרחק מנהטן לעולם לא יחזיר לנו הערכה גבוהה יותר מהפתרון האמתי שעשוי להכיל פיתולים בגלל קירות וכו'. ואם היה מותר לנוע באלכסון? נניח בעלות של 15? הפעם היוריסטיקה שיכולה לשמש אותנו היא "קו אווירי" הרי ברור שזה המרחק הקצר ביותר בין שתי נקודות וכל מסלול אמתי יהיה במחיר יקר יותר מההערכה הזו.

$$\text{מצב א': } f = 10 + \sqrt{1^2 + 2^2} * 10$$

$$\text{מצב ב': } f = 15 + \sqrt{1^2 + 1^2} * 10$$

$$\text{מצב ג': } f = 10 + \sqrt{3^2 + 2^2} * 10$$

המרחק האווירי נלקח ע"י מספר העמודות ומספר השורות שיש לנו, העלנו אותם בריבוע, חיברנו אותם והוצאנו שורש. את התוצאה הכפלנו בעלות של תנועה רגילה, שכן הקו האווירי כבר מיצע את התנועה באלכסון עם התנועה הרגילה ולא נרצה לעבור בהערכה שלנו את המשקל האמתי. ואכן ניתן לראות שהמרחק האווירי לעולם אינו עובר בהערכתו את משקל המסלול האמתי, וכן שמצב ב' יהיה כצפוי בעדיפות הראשונה.

לקריאה נוספת ולפסאודו-קוד האלגוריתם לחצו [כאן](#). מה אנו לומדים מהדוגמא הזו עבור המימוש בקוד? שימו לב שמצד אחד h היא פרמטר של האלגוריתם, הרי האלגוריתם תלוי ב h (בדיוק כמו שהוא תלוי בשאר המידע שבעיית חיפוש צריכה לספק), ובנוסף המימוש של היוריסטיקה תלוי בבעיית החיפוש (תלוי domain) ומצד שני לא לכל בעיית חיפוש צריך להגדיר יוריסטיקה. רק כשנרצה לפתור בעיית חיפוש כלשהי באמצעות A^* נצטרך להזריק לו את היוריסטיקה תלוית הדומיין עבור הבעיה הספציפית שנרצה לפתור. באמצעות איזה pattern design נפתור את הבעיה?

רמז: בהינתן State מסוים, ומצב המטרה, הפונקציה h תיתן הערכה כמה יעלה לנו להגיע מהמצב המסוים הזה למצב המטרה. מחלקה המממשת יוריסטיקה ספציפית תבקש גם סוג ספציפי של State. ממשו את שתי היוריסטיקות עבור העולם הדו-ממדי שהגדרנו (אין לנו תנועה באלכסון):

מרחק מנהטן = מרחק השורות + מרחק העמודות
מרחק אווירי = שורש סכום הריבועים של המרחקים בציר ה X , וה Y .

משימה ד' - Demo:

1. צרו adapter Object שמבצע אדפטציה ממבוך (מופע של Maze2d) לבעיית חיפוש (Searchable)

2. כתבו מחלקה בשם Demo עם מתודה בשם run ש:

a. יוצרת מבוך דו-ממדי באמצעות MyMazeGenerator

b. מדפיסה אותו

c. פותרת אותו באמצעות BFS

d. פותרת אותו באמצעות AStar באמצעות כל יוריסטיקה, ומדפיסה את הפתרון

e. מדפיסה למסך כמה מצבים כל אלגוריתם פיתח. עליכם ליצור מבוך מספיק גדול כדי שתחושו בהבדל.

שימו לב לתיעוד המתודות והמחלקות לפי הצורך!

חלק ב' - Streaming ושמירה לקבצים

כדי לקצר את זמן השמירה והטעינה יהיה עלינו לדחוס את המידע של המבוך. בשמירה, נדחוס את נתוני המבוך ובטעינה נפתח את הדחיסה ונהנה מהמידע. בנוסף נשמור פתרונות שכבר חישבנו, כך שאם נתבקש לפתור בעיה שכבר פתרו נשלוף את הפתרון מהקובץ במקום לחשב אותו מחדש. תהליך זה מכונה כ caching. לשם כך, בחלק זה של המטלה אנו נתרגל עבודה עם קבצים, ונממש אלג' דחיסה פשוט. בנוסף, ניצור ממשק משתמש טקסטואלי - CLI ונקשר אותו לשאר הרכיבים בתוכנית באמצעות ארכיטקטורת MVC.

חישובו איך לייעל את העברת המידע בין השכבות באמצעות **Observer pattern**.

משימה א' - דחיסה:

כמה מידע באמת מחזיק מופע של Maze2D?

הוא מחזיק את נק' הכניסה והיציאה מהמבוך, וכמובן את הגדרת המבוך. הגדרה זו די בזבזנית. כך שאם Maze2D תהיה serializable ונשלח מופע שלה בתקשורת או נשמור אותו בקובץ, בזבזנו המון מקום מיותר. איך ניתן לכווץ את המידע?

דמיינו את המערך הדו-ממדי שלנו כמערך חד-ממדי. יש בו המון רצפים שחוזרים על עצמם, לדוגמא: 1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1. במקום לשמור את הרצף עצמו, נוכל לשמור את הסימן ואז את מספר ההופעות שלו ברצף. עבור הדוגמא לעיל נקבל:

...,1,7,0,10,1,6 נצטרך משמעותית פחות בתים כדי לייצג את אותו המידע בדיוק. אם ניצור

מחלקה שדוחסת את המידע של המבוך, שומרת את ממדיו התלת-ממדיים, וכן שומרת את מיקומי הכניסה והיציאה מהמבוך, אז נוכל באמצעותה לשמור \ להעביר את המבוך בכמות פחותה של בתים באופן משמעותי, ולאחר מכן לשחזר בדיוק את אותו המבוך. צרו את מחלקת MazeCompression שתפקידה יהיה לקבל מידע גולמי של המבוך לדחוס אותו ולכתוב אותו לקובץ, וכמובן גם לשחזר אותו בחזרה. כעת, במחלקה Maze2D נוסיף שני דברים:

1. את המתודה getData שתחזיר מערך המייצג את כל המידע (הלא מכווץ) של המבוך מיקום כניסה ויציאה, ממדי המבוך, ותוכן המבוך
2. בנאי שמקבל מערך של ובונה באמצעותו את Maze2D.

בצעו הניסוי על מופע קיים של Maze2D, מה גודל המבוך? מה גודל הקובץ שנשמר? האם הקריאה מהקובץ הניבה מבוך זהה?

משימה ב' - CLI:

צרו מחלקה בשם CLI. אלו ראשי תיבות של Interface Line Command. מחלקה זו תממש לנו ממשק משתמש של שורת פקודה. נרצה לממש את המחלקה הזו בצורה הכי גנרית שאפשר. לשם כך נצטרך להשאיר ללקוח שלנו את כל ההחלטות, ואנו נספק לו רק את המנגנון. ההחלטות שלו הן:

1. מהו מקור הקלט. הוא יכול להחליט שזה ה input standard למשל, קובץ או אפילו ערוץ תקשורת.
2. מהו מקור הפלט. כנ"ל.
3. מהו סט הפקודות שעלינו לזהות
4. מה תרצה שנריץ כשנזהה פקודה

איך ניגשים לזה? את מקור הקלט והפלט נוכל לייצג כ members data מוג Istream ו Ostream בהתאמה. נוח מאד לעבוד איתם וגם הם יכולים לעטוף כל stream שנרצה. נקרא להם in ו out. נבקש לאתחל אותם בבנאי של המחלקה. הלקוח כבר יבחר מה הם יעטפו.

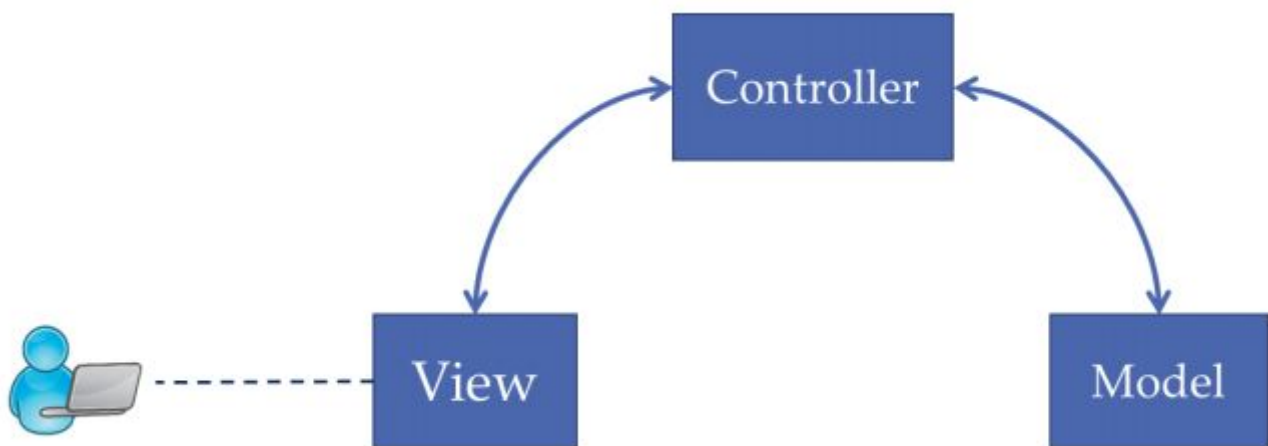
במתודה start נריץ לולאה שבכל איטרציה קולטת מ in מחרוזת עד לקבלה של מחרוזת היציאה .exit.

כעת, על כל מחרוזת שנקלטה מ in נצטרך לגלות מי היא מבין סט של מחרוזות שהלקוח קבע. ונרצה להריץ את הפקודה המתאימה במקרה זה. לשם כך נוכל להיעזר ב Command design pattern

במחלקה CLI (שב view) ניצור member data עבור מבנה נתונים שממפה בין מחרוזת ל Command כך, לאחר כל קלט של מחרוזת נוכל לנסות לשלוף באמצעות מופע של Command מה HashMap באמצעות כל המחרוזת, אם הצלחנו סימן שהקלט הוא חלק מהפקודות בהן אנחנו תומכים, ונוכל מיד להריץ את doCommand מהמופע שחזר. אם לא, נכתוב ל out הודעת שגיאה מתאימה.

חלק ג' - MVC

כדי לא ליצור קוד עם שכבה אחת בלבד, חילקנו את הקוד ל view ו controller (עבור כל שכבה ניצור תחילה ממשק שיגדיר את הפונקציונליות) כעת נוסיף שכבה חדשה עבור המודל. המודל יהיה אחראי לפעולות של האלגוריתמים שונים. ה- view יהיה אחראי על קלט המשתמש והתצוגה עבורו. ה- controller יהיה אחראי להעברת פקודות ביניהם. כל ממשק כזה משמש אותנו כ façade עבור שאר המחלקות שיהיו יחד אתו באותה השכבה. בכל ממשק עליכם להגדיר בעצמכם את המתודות ע"פ הנראה לכם לנכון. כעת בכל שכבה צרו מחלקה המממשת את הממשק `MyController imp' Controller` `MyModel imp' Model` `MyView imp' View` אפשרו במחלקות אלו, ע"י הכלה מתאימה, את התרשים שראינו בשיעור:



בצעו את האתחול במתודת ה main הראשית של התוכנית. לאחר החיבורים תפעילו מתודה בשם start של CLI שתתחיל במלאכת הקלט מהמשתמש והתוכנית שלנו תוכל לרוץ. מה התוכנית מריצה? הרי ה CLI צריכה את ה HashMap של ה Commands כדי לדעת מה לעשות... בהמשך תופיע הגדרה של פרוטוקול. כל הפעולות המתוארות עבור פקודה מסוימת בפרוטוקול צריכות להיות במחלקה משלהן מסוג Command

שבשכבת ה controller. את הפעולות המתוארות הן יבצעו כמימוש של doCommand.
 חלק מהפעולות האלה יפעילו פעולות במודל ואחרות יפעילו פעולות ב view.
 **שימו לב שמחלקות אלה לא אמורות להכיר אף אחד פרט לממשק של ה View ולממשק של המודל. בעת הפעלתה מה main, המחלקה MyController צריכה ליצור את ה HashMap הממפה בין String ל Command שמאותחל באמצעות מחלקות Command אלה.

כעת, צרו את המחלקות הנדרשות כדי לקיים את הפרוטוקול הבא:
זכרו: כל הפעלה נעשית במחלקה ייעודית מסוג Command ב controller. כל בקשת תצוגה תמומש ב View. וכל בקשת חישוב או עיסוק בנתונים תתבצע במודל. (תזכורת - שלבו את Observer pattern בין השכבות)

dir <path>	תציג את כל הקבצים והתיקיות הנמצאים בתוך <path> נתיב מסוים במערכת הקבצים של המחשב. טיפ: המחלקה File.
generate maze <name><other params>	זו פקודה ליצור מבוך עם השם name. מותר לכם לבקש פרמטרים נוספים כרצונכם. יצירת המבוך תתבצע בשכבת המודל. לכשהחישוב יסתיים המודל יבקש מה controller להדפיס " maze <name> is ready" כמובן controller יעביר את הבקשה ל View.
display <name>	נדפיס למשתמש את המבוך בשם name. לאורך כל הריצה של התוכנית נאפשר למשתמש להציג את כל המבוכים שהוא ביקש ליצור. טיפ: שימרו את המבוכים שנוצרו במבנה נתונים מתאים. חישבו היכן לשמור אותו
save maze <name> <file name>	תשמור את המבוך name בצורה מכווצת בקובץ file name
load maze <file name> <name>	תטען מהקובץ file name מבוך חדש שישמר תחת השם name
maze size <name>	תציג את גודל המבוך בזיכרון

file size <name>	תציג את גודל המבוך בקובץ
solve <name> <algorithm>	תפתור את המבוך בשכבת המודל כמובן, באמצעות האלגוריתם הנתון. לכשהיה פתרון המודל יבקש מה Controller להציג - "Solution for <name> is ready" והוא כמובן יעביר את הבקשה ל- View
display solution <name>	יגרום לתצוגת צעדי הפתרון
exit	יציאה מסודרת מהתוכנית ללא זליגות זיכרון, קבצים ושאריות אחרות

חלק ד' - Caching

המתודות ליצירה של מבוך תלת-ממדי או פתרונו הן מתודות חישוב שעלולות לקחת זמן רב.

MyModel תחזיק HashMap הממפה Maze2d ל Solution. כל פתרון שיחושב נכנס למפה זו. כך שבהינתן בקשה לפתרון מבוך, נוודא תחילה אם הוא נמצא ב HashMap.

אם ה Maze נמצא במפה אז נחזיר מיד את ה Solution. כלומר במקום להתחיל חישוב חדש נשלח נוטיפיקציה שהפתרון מוכן. כמו כן, נרצה לשמור את המפה בקובץ. זאת כדי שבתחילת הריצה הבאה נטען הקובץ ישירות למפה, וכך ננצל גם חישובים שבוצעו בריצות קודמות. את המידע של המפה יש לכווץ לפני השמירה (ולפענח לאחר הקריאה).

חלק ה' - תוצרים

1. יש לצרף דיאגרמת מחלקות המתארת את הקשרים, והפונקציונליות של מחלקות הפרויקט.
 2. סרטון הסבר על הפרוייקט - 5-8 דק':
 - a. תיאור כללי של הפרויקט
 - b. האלגוריתמים
 - c. דרך העברת המידע ושמירתו
 - d. כל ערך מוסף נוסף שאתם מוצאים לנכון
 3. קובץ זיפ - המכיל את קבצי הקוד הרלוונטיים
 - a. מומלץ להפריד את חלקי הפרויקט השונים לתיקיות
 4. יש לוודא שהפרויקט עובד, ומתקמפל ללא שגיאות או הערות.
 5. מומלץ (לא חובה) בעבודה הזוגית להשתמש במנהל גרסאות Git הזדמנות נפלאה לתרגל עבודה צוותית ואתגרי סנכרון קוד בהם תתקלו בהמשך הקריירה בתעשייה.
- ניתן לצלם את הסרטון דרך הזום, עם שיתוף מסך או באמצעות תוכנת הקלטה ייעודית לבחירתכם (OBS עושה עבודה טובה והיא חינמית)

בהצלחה: