036044 - תכן תנועת רובוטים וניווט ע"י חיישנים
פרויקט - חלק שני

אור קוזלובסקי – 305447476

שיר קוזלובסקי - 204321228

המטרה של חלק זה היא לפתח אלגוריתם על הבסיס החלק הראשון של הפרויקט, לחישוב מסלול של רובוט מצולע קמור מנקודת התחלה ידועה באוריינטציה ידועה לנקודת סיום ידועה, גם היא באוריינטציה ידועה. בעולם דו-ממדי המכיל מכשולים מצולעים קמורים ועל בסיס כך נבחרה שיטת המימוש, החלוקה למחלקות ולפונקציות.

נזכיר רק שהמטרה של החלק הראשון הייתה לממש אלגוריתם לחישוב מרחב הקונפיגורציה של גוף קשיח (רובוט). המימוש נעשה בתוכנת מטלב בשימוש ב-$Object\ Oriented$.

1 אלגוריתם למציאת המסלול

החלק המרכזי של האלגוריתם שלנו בחלק זה של הפרויקט הוא פונקציה למציאת מסלול ניווט בין נקודת ההתחלה לנקודת הסוף. כמובן שנרצה שפונקציה זו תהיה שלמה, כלומר שתדע להגיד אם קיים מסלול או לא, ואם קיים, אז שתספק את המסלול הקצר ביותר. נחלק את הבעיה לשתיים:

1. המרת מרחב הקונפיגורציה הדיסקרטי לגרף.
2. מציאת המסלול הקצר ביותר בגרף בהינתן צומת התחלה וצומת סוף.

נפרט על כל אחת מהבעיות והדרך שבחרנו לפתור אותה.

<u>המרת המרחב קונפיגורציה הדיסקרטי לגרף</u>

הדרך הנאיבית להמיר את מרחב הקונפיגורציה הדיסקרטי לגרף הוא שכל תא במרחב הקונפיגורציה החופשי יוגדר כצומת וקשתות הגרף יוגדרו כמחברות בין כל שני תאי מרחב חופשי סמוכים (המושג "סמוכים" אינו מוגדר היטב בשלב זה אך הוא יוגדר בהמשך העבודה כשנפרט על אופן המימוש).

ישנן דרכים יותר מורכבות להמרה, שמגדילות את הסיבוכיות לבנית הגרף אך מקטינות את מימדי הגרף ובכך מקטינות את סיבוכיות זמן הריצה של השלב הבא למציאת המסלול הקצר ביותר בו.

האלגוריתם שלמדנו בכיתה לייצוג מרחב הקונפיגורציה החופשי על ידי גרף:

1. 3D visibility graph.
2. מפת וורונוי.
3. פונקציית פוטנציאל.

מרחב הקונפיגורציה בבעיה הנתונה אינו מורכב או מכיל מקרי קצה ולכן סיבוכיות הגרף בהמרה הנאיבית אינו עולה בהרבה אל מול השיטות האחרות. לכן, לשם פשטות המימוש, בחרנו לפתור את בעיה זו באמצעות המרה נאיבית.

<u>מציאת המסלול הקצר ביותר בגרף בהינתן צומת התחלה וצומת סוף</u>

בהינתן הגרף שהרכבנו, צומת התחלה וסוף, נוכל להשתמש באחד האלגוריתמים שלמדנו בכיתה למציאת המסלול הקצר ביותר בגרף בין שני צמתים נתונות. האלגוריתמים שלמדנו בכיתה למציאת המסלול הקצר ביותר בגרף הם:

- A^*
- *Dijkstra*
- *BFS (Breadth-first search)*

נציין שאלגוריתם *DFS* אינו אלגוריתם למציאת מסלול קצר ביותר בגרף ולכן אינו עונה על הדרישות שלנו. ל-*DFS* שימושים אחרים כמו מציאת רכיבי קשירות, מיון טופולוגי ועוד.

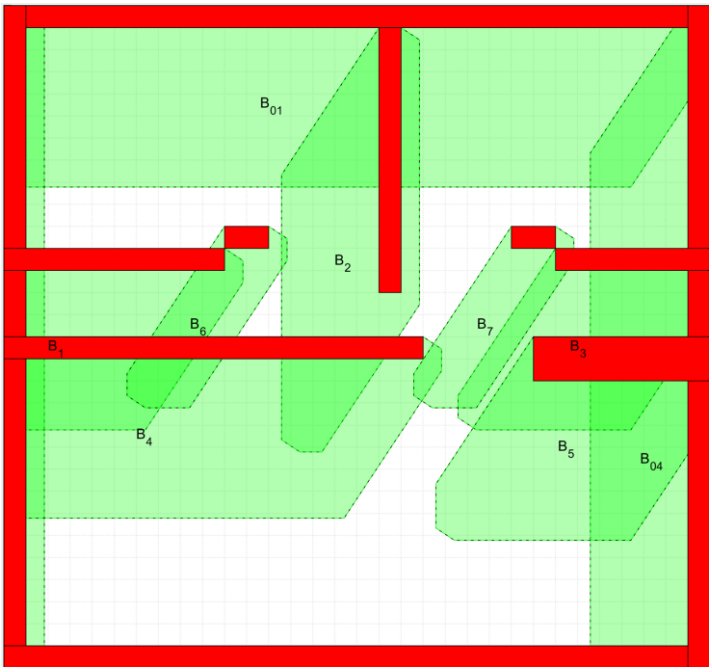
אלגוריתם *BFS* ו-*Dijkstra* מרחבים את החיפוש אחר מסלול אופטימלי במעגלים קונצנטריים ממורכזים בנקודת ההתחלה שהולכים ומתרחבים בכל איטרציה. שיטה זו יעילה אם מחפשים מסלולים קצרים לכמה יעדים בו-זמנית. אך במקרה שבו יש נקודת יעד אחת, כמו במקרה שלנו, נוכל לשפר משמעותית את זמן הריצה אם החיפוש יתרחב יותר כלפי צמתים שקרובות יותר לנקודת היעד.

זאת בדיוק התכונה שאלגוריתם A^* ממש בשימוש בהיוריסטיקה הוקלידית כפי שנלמד בכיתה. בכיתה הוכחנו כי אלגוריתם A^* הוא שלם ומחזיר את המסלול הקצר ביותר בין צומת המקור ליעד, לכן בחרנו בו כפתרון לבעיה זו.

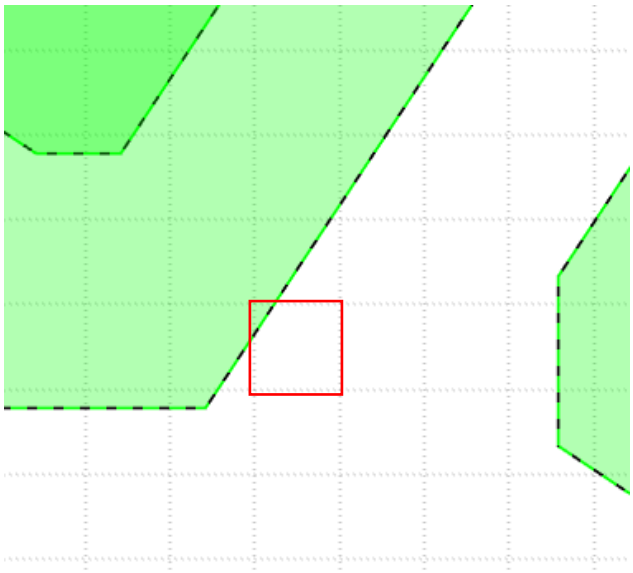
2 רזולוציה

רזולוציה מרחבית

כדי למזער את סיבוכיות הזמן והמקום של האלגוריתם (שאיפה בסיסית בכל אלגוריתם) נרצה לבחור ברזולוציה המינימלית שתאפשר פתרון של הבעיה. קביעת הרזולוציה המרחבית המינימלית תלוי בבעיה. נסביר זאת באמצעות דוגמה. נניח כי זה מרחב הקונפיגורציה שלנו:

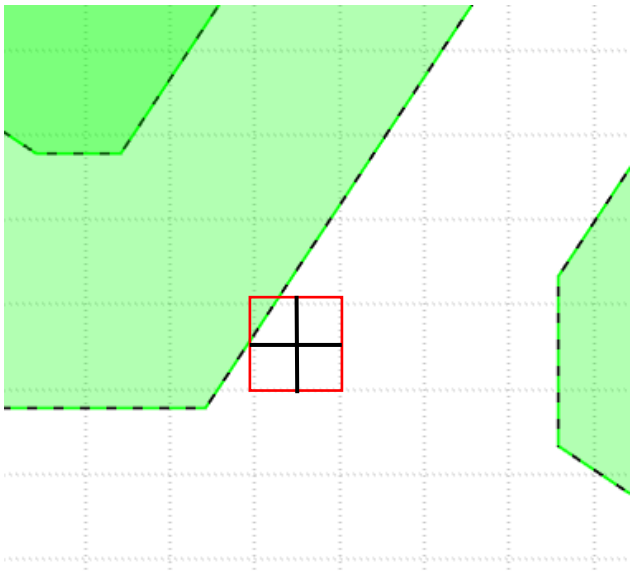


האזור הלבן מסמל מרחב קונפיגורציה חופשית. כל ריבוע במפה הוא יחידה אחת ולכן המפה המוצגת היה ביחס של 1:1 לגדלים הנתונים בשאלה. כעת נתמקד באזור מסוים המציג את הגבול בין מרחב הקונפיגורציה החופשית לשפתו של מכשול במרחב הקונפיגורציה:



נתייחס לריבוע שסימנו באדום. ניתן לראות כי יש בו אזור קטן המכיל מכשול אך רובו מכיל מרחב חופשי. במעבר מהמרחב הרציף לדיסקרטי נצטרך להגדיר לכל משבצת במרחב הדיסקרטי האם היא מכשול או מרחב חופשי.
במעבר זה, אם נבחר רזולוציה מרחבית 1:1 אז המשבצת הזו תוגדר כמכשול – כי למרות שרובה מרחב חופשי,

בהסתכלות דיסקרטית הרובוט לא יכול לעבור בה כי היא מכילה גם מכשול. אך, אם היינו בוחרים ברזולוציה של 2:1 בשני הצירים אז כל משבצת הייתה נחלקת ל-4 במעבר למרחב הדיסקרטי:



במצב זה, רק רבע מן המשבצת המקורית היה מוגדר כמכשול והרובוט היה יכול לנוע דרך המשבצת בכל אחת משלושת הרבעים האחרים שהיו מוגדרים כמרחב חופשי.

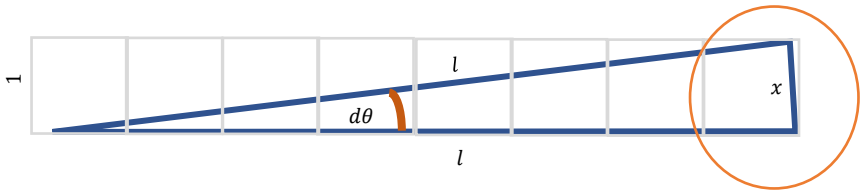
כלומר, הגדלת הרזולוציה המרחבית יכולה לפתוח בפני הרובוט משבצות שלא היו פתוחות בפניו ברזולוציות נמוכות יותר. הבדל היכול להיות המפתח לפתרון. בחירת העידון הנדרש כדי לאפשר פתרון תלוי בפרמטרים של הבעיה כמו מיקום המכשולים, מימדי הרובוט וכו'.

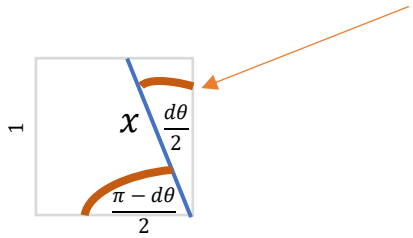
<u>רזולוציה זוויתית</u>

בחלק הראשון של הפרויקט נדרשנו לרזולוציה של $\frac{360}{32}$, ולכן מכשולי מרחב הקונפיגורציה השתנu באופן ניכר בין שתי אוריינטציה עוקבות. כעת, על מנת לפתור את הבעיה שניתנה וכן לתאר את המציאות בצורה הטובה ביותר, עלינו למצוא רזולוציה עדינה יותר, המאפשרת מעבר בכל משבצות המרחב הדיסקרטי ולא מפספסת אף לא אחת במעבר בין שתי זוויות עקובות של הרובוט.

מצד אחד, ככל שהרזולוציה עדינה כך ישנן יותר איטרציות וריצת האלגוריתם יתארך. מצד השני, רזולוציה גסה מדי תדלג על משבצות מסוימות בין שתי זוויות עוקבות ולא תספק תוצאה הייצגת את המציאות.

קודקודי מכשולי מרחב הקונפיגורציה מחושבים על ידי קודקודי המכשולים וקודקודי הרובוט. קודקודי המכשולים אינם משתנים בין התך θ אחד לאחר אלא רק קודקודי הרובוט. לכן, על מנת למצוא את הרזולוציה האופטימלית עבור בעיה זו, עלינו לוודא שבין שני זוויות עוקבות שינוי קודקודי הרובוט הוא לכל היותר משצבת אחת. נמצא את הרזולוציה שמקיימת דרישה זו על ידי כך שנחשב את שינוי הזווית המתקבלת כאשר קצה אחד של הרובוט משנה את מיקומו במשבצת אחת, ואילו הקצה השני הוא ציר הסיבוב (באלגוריתם שלנו הרובוט מסתובב סביב ראשיתו שמוגדרת באחד הקודקודים שלו). להלן שרטוט הבעיה:





נשתמש במשפט הקוסינוס:

$$x^2 = l^2 + l^2 - 2 \cdot l \cdot l \cdot \cos(d\theta) = 2 \cdot l^2 - 2 \cdot l^2 \cdot \cos(d\theta)$$

בנוסף נדרוש שיתקיים:

$$x \cdot \cos\left(\frac{d\theta}{2}\right) = 1 \Longrightarrow x = \frac{1}{\cos\left(\frac{d\theta}{2}\right)}$$

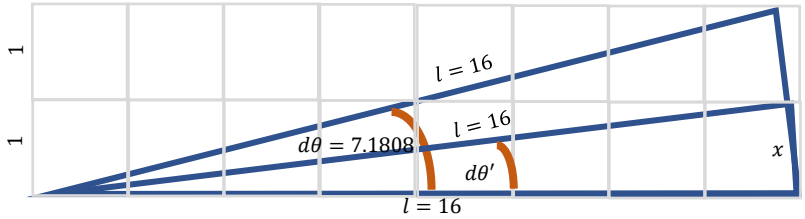
מכאן:

$$\frac{1}{\cos^2\left(\frac{d\theta}{2}\right)} = 128 - 128 \cdot \cos(d\theta) \underbrace{\Longrightarrow}_{\cos\left(\frac{d\theta}{2}\right) = \pm\sqrt{\frac{1+\cos(d\theta)}{2}}} \frac{1}{\frac{1+\cos(d\theta)}{2}} = 2 \cdot l^2(1 - \cos(d\theta)) \Longrightarrow$$

$$\Longrightarrow \frac{1}{l^2} = 1 - \cos^2(d\theta) \Longrightarrow \cos^2(d\theta) = \frac{l^2 - 1}{l^2}$$

הקשר בין הרזולוציה הזוויתית למרחבית

הרזולוציה הזוויתית מחושבת ביחס לרזולוציה מרחבית נתונה הרזולוציה הזוויתית שקיבלנו הגיעה מהדרישה הטריגונומטרית שהוצגה בסעיף הקודם וניתן לראות כי היא תלויה באורך l. נסביר את האינטואיציה לכך. שינוי הרזולוציה המרחבית תשנה את אורכי השוקיים במושלש שווי השוקיים אך, אם עדין נדרוש תזוזה של משבצת אחת בין שתי זוויות עוקבות, בסיסו של המשולש לא ישתנה ובהכרח הזווית המקסימלית שתקיים את הדרישה תקטן, והרזולוציה הזווית תגדל. האיור הבא מדגים זאת:



בגלל שהגדלנו באופן פרופורציונאלי את כל צלעות המשולש אז $d\theta$ לא השתנתה. ניתן לראות כי הזווית המקסימלית $d\theta'$ ברזולוציה המרחבית החדשה קטנה ביחס ל-$d\theta$.

הרזולוציה הנבחרת

לאחר ניסוי וטעיה, מצאנו כי עבור רזולוציה של 1:1 (כלומר מרחב דיסקרטי בגדול [32 30]) לא מתקבל פתרון אך עבור רזולוציה של 1:1.5 בכל ציר (כלומר מרחב דיסקרטי בגודל [48 45]) מתקבל פתרון.

כדי להגיע לסיבוכיות זמן ומקום מינימליות באלגוריתם שלנו וכדי לאפשר פתרון קבענו את רזולוציה המרחבית להיות ביחס 1:1.5 בכל ציר, כלומר [48 45]. בהתאם, וכפי שהוסבר בסעיפים הקודמים, קבענו את הרזולוציה הזוויתית להיות $d\theta = \frac{360}{76}$ (רזולוציה זו נובעת מהצבה $l = 12$ בנוסחה שחישבנו לרזולוציה הזוויתית).

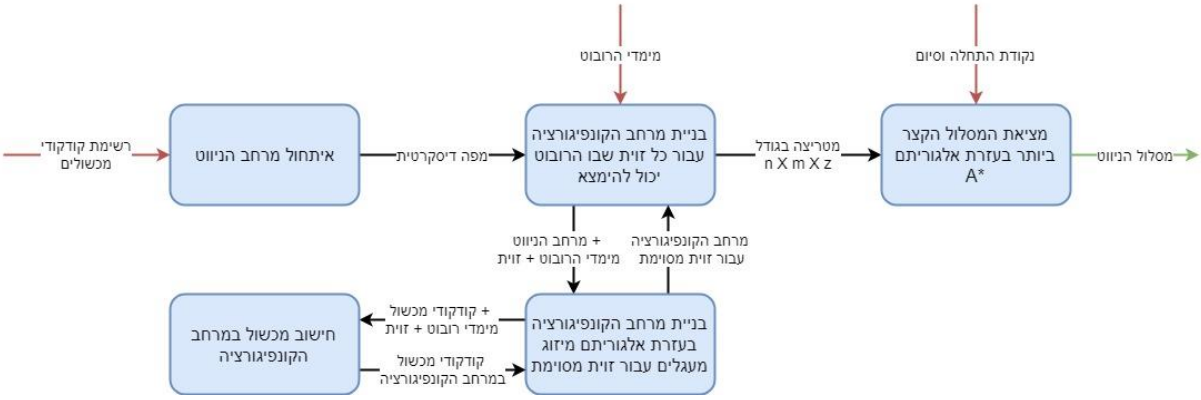
3 רעיון מימוש

בהינתן רשימת קודקודים של מכשולים נבנה מפה דיסקרטית של מרחב הניווט (ברזולוציה מרחבית שנבחרה בסעיף הרזולוציה). בהינתן מימדי הרובוט ומפת מרחב הניווט נחשב את מרחב הקונפיגורציה החופשית באמצעות אלגוריתם מיזוג מעגלים לכל זווית שבה הרובוט יכול להיות (ברזולוציית הזווית שנבחרה בסעיף רזולוציה). עבור זוית מסוימת של הרובוט, נייצג את מרחב הקונפיגורציה על ידי מטריצה דו-מימדית שבה '1' מסמל מכשול ו-'0' מסמל מרחב קונפיגרציה חופשי. לכן, אם ישנן z זוויות אפשריות, ומרחב הניווט הוא בגודל $m \times n$ אז התוצר של שלב זה הוא מטריצה תלת-ממדית בגודל $m \times n \times z$. נקרא למטריצה זו M.

כעת, שיש בידנו את המטריצה M, נוכל להתייחס אליה כגרף באופן הבא - כל תא שמכיל '1' יוגדר כצומת בגרף והקשתות יוגדרו בין כל זוג תאי '1' סמוכים לפי קישוריות 26, כלומר התאים הסמוכים לתא מסוים הם כל התאים בקובייה $3 \times 3 \times 3$ שמרכזה באותו פיקסל.

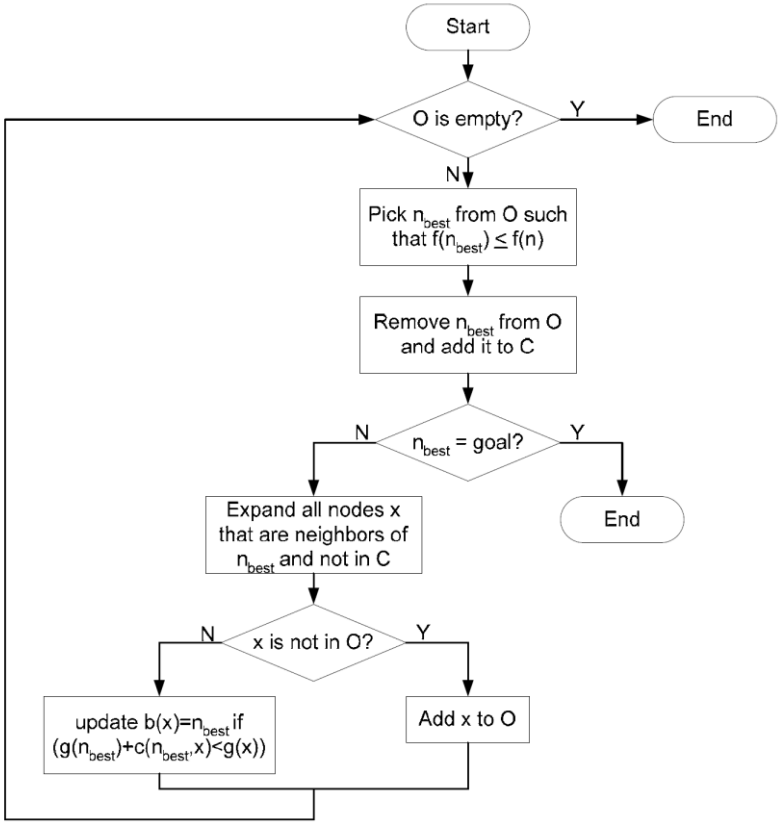
על הגרף המתקבל, ובהינתן נקודות התחלה וסיום, נריץ את האלגוריתם A^* למציאת המסלול הקצר ביותר בין הנקודות הנתונות כפי שנלמד.

סכמת הבלוקים של המימוש המתואר:



4 אופן פעולת הפונקציה למציאת המסלול הקצר ביותר

כפי שנלמד בכיתה, נגדיר שני מבני נתונים לצורך מימוש הפונקציה, מבנה נתונים של צמתים פתוחות O ומבנה נתונים של צמתים סגורות C. נאתחל את O להכיל את צומת ההתחלה ואת C להיות ריקה ונפעל לפי הסכמה הבאה לחישוב המסלול:



את O ממשנו כמערך ואת C בחרנו לממש כמטריצה תלת ממדית זהה בגודלה ל-M שבה כל תא עם ערך '1' מסמל צומת סגורה ו-'0' מסמל מומת שאינה סגורה. מימוש זה אולי אינו יעיל מבחינת סיבוכיות מקום לעומת מימוש רשימה אך סיבוכיות גישה לאיבר במבנה היא $O(1)$ והוא מאפשר מימוש פשוט יותר.

5 מבנה הנתונים של האלגוריתם

כפי שצוין בהקדמה, בחרנו לממש את האלגוריתם בתכנות מונחה עצמים.
לצורך פתרון הבעיה ממשנו כמה מחלקות:

1. מחלקת Map_object - מממשת אבסטרקציה של אובייקט מפה כללי (רובוט או מכשול). מקבלת רשימה של קודקודים, כיוון של הנורמלים (כדי לאפשר נורמלים כלפי פנים האובייקט עבור הרובוט וכלפי חוץ עבור המכשולים) ושם של אובייקט. מאפשר חישוב של הנורמלים של האובייקט והדפסה של האובייקט.
2. מחלקת $Robot$ – הרחבה של מחלקת Map_object המוסיפה לאבסטרקציה תכונה של זווית ומיקום הניתנות לשינוי. המחלקה מממשת סיבוב של אובייקט באמצעות מטריצת סיבוב.
3. מחלקת Map – מחלקה המייצגת את מרחב ניווט ומרחב קונפיגורציה. מכילה מופעים של מחלקת $robot$ ומחלקת Map_object. מקבל רשימה של קודקודי רובוט ורשימה של רשימות קודקודי מכשולים. הפונקציות המרכזיות הן חישוב מכשולי מרחב קונפיגורציה ודיסקריטיזציה של המרחב הרציף למרחב בדיד. לאחר אתחול מרחב הניווט המחלקה מאפשרת שינוי של מרחב הקונפיגרוציה ביעילות ונוחות.
4. מחלקת $Route$ – מחלקה המממשת את הפתרון לבעיה בחלק זה של הפרויקט. מקבל כקלט אוביקט מאותחל מסוג Map, נקודות התחלה ונקודת סוף. הפונקציה מרכזית בה היא חישוב המסלול הקצר ביותר בMap בין נקודות ההתחלה לסוף. בנוסף מאפשר הדפסה של מרחב הקונפיגורציה ושל המסלול שחושב.

6 פונקציית מחיר

כלל ההתרחבות בA^* (הכלל לבחירת הצומת מתוך רשימת הצמתים הפתוחות) הוא לפי פונקציית המחיר $f = h + g$, כאשר h הינו המרחק האוקלידי לנקודת הסוף ו-g היא סכום הקשתות מנקודת ההתחלה עד לצומת הנוכחית.

קבענו את מחיר הקשתות באופן הבא:

חלקנו את תנועת הרובוט לשלושה סוגי תנועה אפשריים: ללכת באותה זווית נתונה ימינה, משאלה, קדימה אחורה ובאלכסון (נגדיר תנועה זו כ- $type\ 1$), לשנות את זוuית הרובוט מבלי לשנות את מיקמו ($type\ 2$) וכן ישנה אפשרות לשנות גם את המיקום וגם את הזווית ביחד ($type\ 3$).

לפיכך חילקנו את סוגי השכנים לשלושה סוגים. נתנו לקשתות מחיר לפי סוג השכן. קיוונו שבאמצעות כך נוכל ליעיל את התוכנית שלנו, הן מבחינת כמות האטרציות והם מבחינת כמות הצעדים שהרובוט עושה במסלול הסופי. בנוסף, ניתן לתת פקטור כולל לכל סוגי התנועה יחדיו וכך לשנות את היחס בין h ו- g (זהו המשתנה $factor$ בביטוי מטה). באופן טבעי, ללא משקל זה, ככל ש-g קטן יותר, כך h גדול יותר – כלומר ככל שאנחנו מתקרבים ליעד עשינו יותר דרך מאז המקור. לכן, ללא פקטור, תינתן עדיפות לנקודות שיותר קרובות להתחלה ולכן זמן החישוב יכול לגדול.

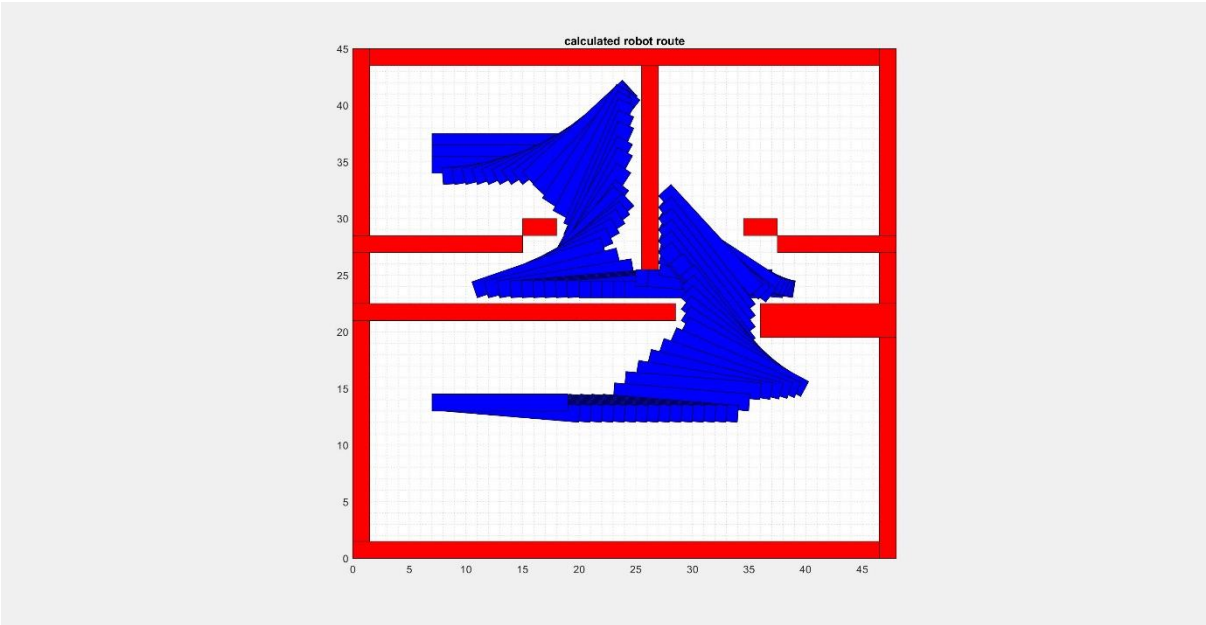
המשקלים שנבחרו הינם:

$$[type\ 1, type\ 2, type\ 3] \cdot factor = [1,1, sqrt(2)] \cdot 0.01$$

7 תוצאות

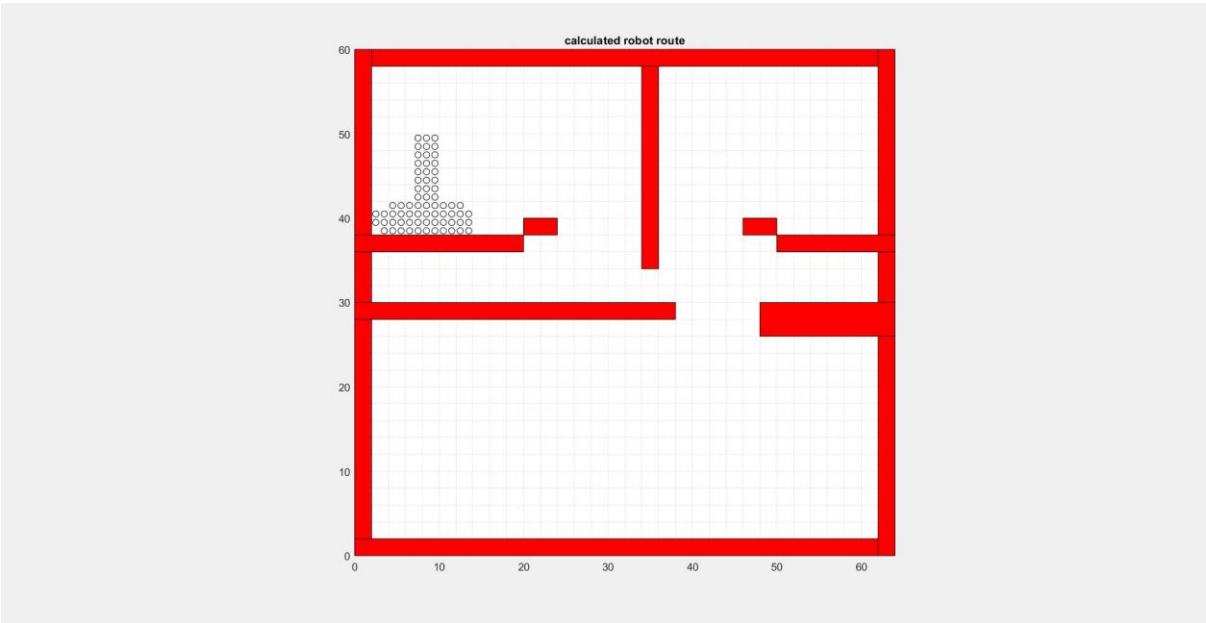
לאחר הרצת הקוד ומציאת המסלול נציג כעת את תוצאות ההרצה.

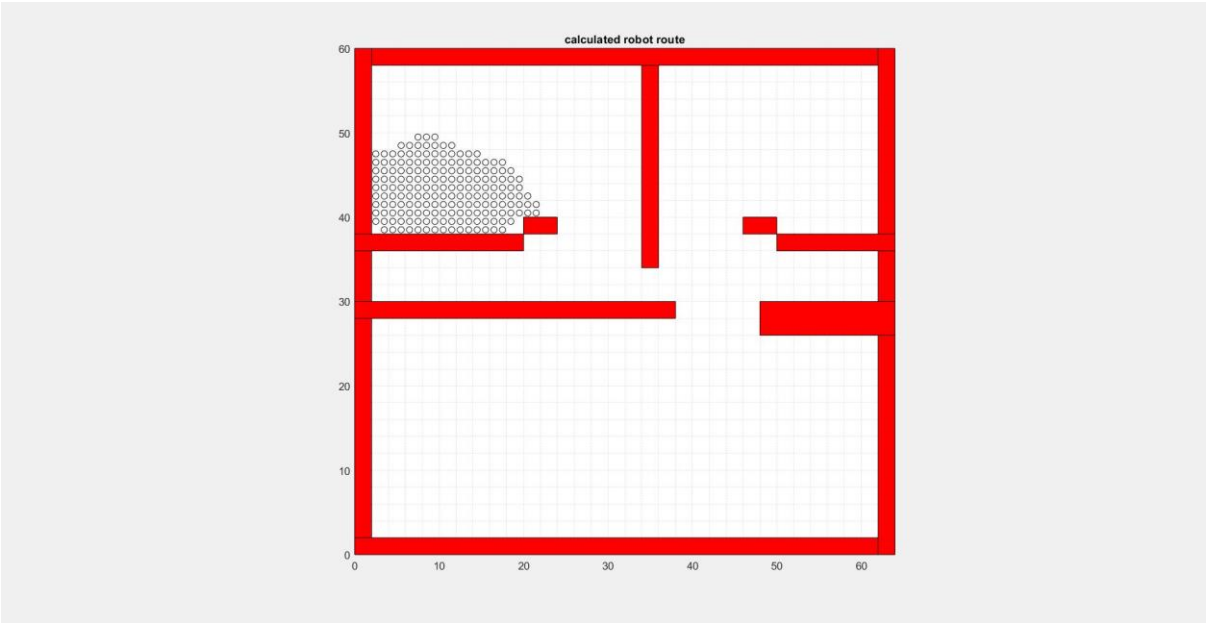
המסלול הנמצא הינו:

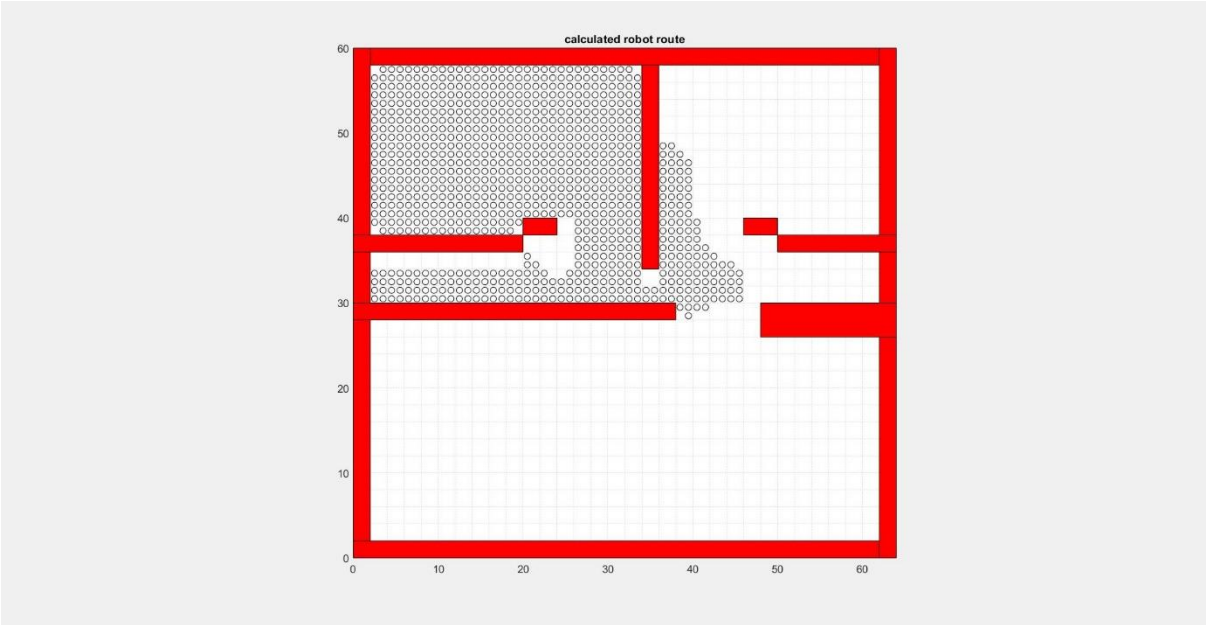


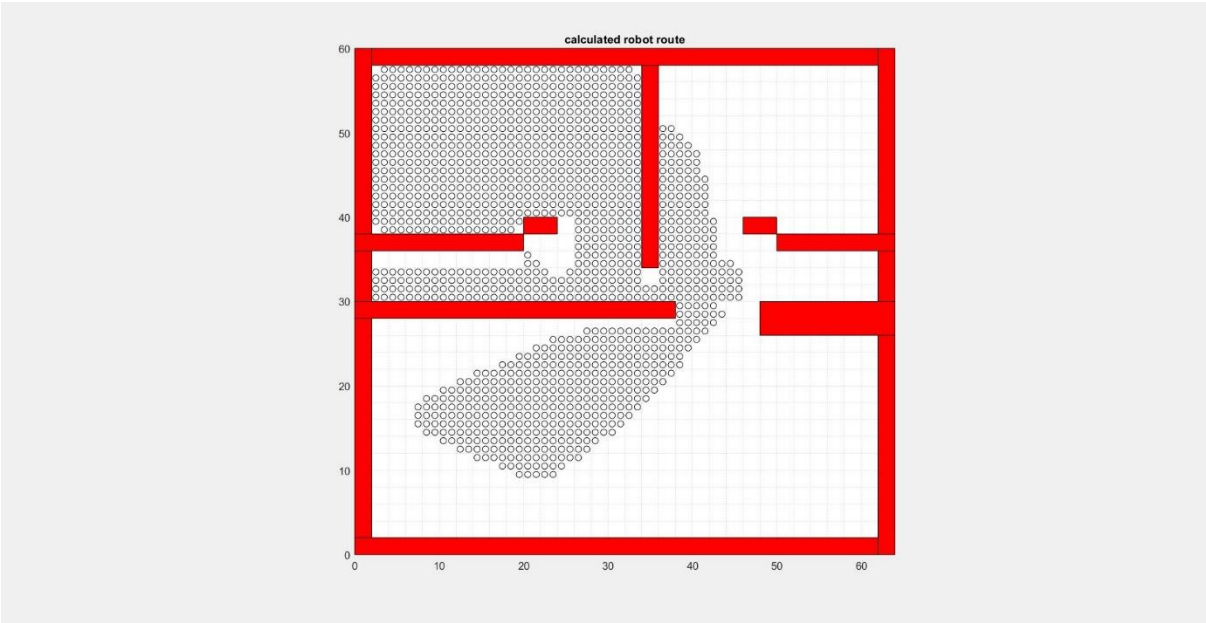
כאשר הרובוט עשה 89 צעדים.

על פי משפט האליפסה, אלגוריתם A^* סוגר שטחים בצורה של אליפסות הולכות וגדלות כאשר מרכזי האליפסה הגדולה ביותר הם נקודות ההתחלה והסיום, וסכום אורכי הרדיוסים הוא אורך המסלול הקצר ביותר. ניתן לראות שזה אכן מתקיים גם בתוכנית שלנו:




calculated robot route
60
50
40
30
20
10
0
0
10
20
30
40
50
60


calculated robot route
60
50
40
30
20
10
0
0
10
20
30
40
50
60



התנסנו בפונקציות משקל שונות ושמנו לב כי הן משפיעות על הטרייד-אוף בין אורך המסלול הסופי לבין זמן ריצת האלגוריתם. כפי שהוסבר בסעיף פונקציית המחיר, שינוי פונקציית המחיר משפיע על סדר בחירת המצתים מתוך רשימת הצמתים הפתוחות. ראה סעיף פונקציית מחיר להסבר נוסף.

נסכם את התוצאות בטבלה:

כמות איטרציות	אורך מסלול	$[type\ 1, type\ 2, type\ 3] \cdot factor$
3438	90	$[1,1,1] \cdot 0.01$
7720	86	$[1,1,1]$
3462	89	$[1,1, sqrt(2)] \cdot 0.01$
8828	85	$[1,1, sqrt(2)]$
3427	90	$[1,1,0.9]$

ניתן לראות כי פונקציית המחיר משפיעה באופן ישיר על אורך המסלול וזמן החישוב. לפי דעתנו שיפור של יותר מ-100 אחוז בכמות האיטרציות עדיף על פני תשלום של אחוזים בודדים באורך המסלול ולכן פונקציית המחיר שבחרנו היא זאת שמוצגת בסעיף פונקציית מחיר.

8 פתרון בעיית אונליין

פתרון A^* לבעיית האונליין הוא חישוב מפת הקונפיגורציה ומסלול מחדש כל פעם שהרובוט נפגש במכשול, כאשר נקודת ההתחלה משתנה ומוגדרת להיות הנקודה בה הרובוט, כל זאת כמובן לאחר המכשול החדש התווסף למפת הניווט.

ראינו כי זמן החישוב של אלגוריתם A^* ארוך וכי הוא לא מתבסס כלל על מידע קודם. פתרון זה לא יעיל ולכן נציע פתרון אחר לפתירת בעיה זו, המבוסס על A^*, אך עובד בצורה מעט שונה. אלגוריתם D^*lite. אלגוריתם זה יכול לעבוד בסביבה ידועה ובסביבה שאיננה ידועה, כאשר בסביבה השניה הוא מבטיח את המסלול הקצר ביותר וכן הוא יעיל יותר מ-A^* בסביבה כזאת. (חשוב לציין כי A^*, D^*lite יעילים במידה שווה בסביבה ידועה מראש, כתלות בסביבה).

כמו A^* הוא שומר "מחיר" לכל משבצת (צעד) לפי המרחק שלו מהמטרה, אך בנוסף ל-A^* הוא שומר ערך נוסף של "מחיר" הצעד הבא.

כאשר הוא מגיע למכשול לא מוכר, הוא מחשב את מסלול לפי ערכים אלו, כאשר הוא בוחן על כל השכנים שלו, ובוחר את השכן הרלוונטי ביותר למציאת המסלול האופטימלי.

9 מסקנות

1. האלגוריתם A^* סיפק מסלול לרובוט לפי נתוני הבעיה. הרובוט ביצע 89 צעדים למטרה.
2. הרזולוציה המוצגת בבעיה (חלוקה של 32 זוויות לסיבוב מלא) לא מאפשרת תנועה רציפה וחלקה ותמרון בין המכשולים (מפספסת צעדים בגלל גסותה) ולכן טווח הרזולוציה המינימלי המספק תוצאות הינו חלוקה ל-76 זוויות לסיבוב שלם. בנוסף כל משבצת חולקה ב-1.5 כדי ליצור עידון מרחבי וכן קיבלנו מרחב ניווט בגודל $48 * 45$, כמובן שכל היחסים נשמרו. (בין הרובוט ובין המכשולים). להגדלת הרזולוציה יש חסרון – הגדלת סיבוכיות הזמן והמקום.
3. קביעת משקל הקשתות משפיע באופן ישיר על זמן החישוב ועל אורך המסלול הסופי כאשר הראנו כי קיים טרייד אוף בין השניים. בחרנו בפונקציית המחיר האופטימלית לדעתנו המביעה לשיפור משמעותי בזמן הריצה לעומת תשלום קטן באורך המסלול.
4. מספר האטרציות למציאת המסלול שהתוכנית מבצעת הוא 3462. כאשר מספר האטרציות המקסימלי שהתוכנית יכולה לעשות הוא: $45 \cdot 48 \cdot 76 = 164{,}160$. פי 47 ממספר האטרציות שבוצעו באמת.
5. ניתן לפתור את בעיית ה- $online$ של A^* ע"י חישוב מחדש של מסלול הניווט. אך פתרון זה לא יעיל ולכן הוצע פתרון אחר, אשר מתבסס על עקרונות A^* ונקרא $D^*\ lite$.

מחלקת Route

```
classdef Route < handle

    properties
        config_space
        theta_res
        start
        dest
        map
        route
    end

    methods
        function obj = Route(map, start, dest)
            obj.map = map; % intilaized Map object
            obj.start = start;
            obj.dest = dest; % 3D-point (x,y,theta)
            obj.theta_res = 76;
        end

        function calc_config_space(obj)
            dt= 360/obj.theta_res;
            obj.config_space = zeros([30*obj.map.size_factor 32*obj.map.size_factor obj.theta_res]);
            i=0;

            for theta=0:dt:360-dt
                i=i+1;
                obj.map.robot.set_theta(theta);
                obj.map.calc_c_obstacles;
                obj.map.update_matrix_map;
                obj.config_space(:,:,i) = obj.map.matrix_map;
            end
        end

        function A_Star(obj, saveDir)
            % A star algorithm
            start = obj.start+1;
            target = obj.dest+1;

            path = 1;
            key= distance([start,target], obj.theta_res);
            O = [key,start,0,start]'; %#ok<*PROP>
            C = obj.config_space;

            fig = figure('visible', 'on', 'Units', 'normalized', 'position', [0 0 1 1]); hold on;
            with_norms = false; %set to true will print red lines to mark the normals from each line.

            for z=1:length(obj.map.obstacls)
                obj.map.obstacls{z}.print_obstacle(with_norms);
            end

            axis equal;
            fig.Children.XLim = [0 obj.map.map_size(2)];
            fig.Children.YLim = [0 obj.map.map_size(1)];
            grid minor; title(['calculated robot route']);
            l=1;

            while (path == 1)
                x_best_i = find(O(1,:) == min(O(1,:)), 1);
                x_best = O(2:4,x_best_i);
                g_x_best = O(5,x_best_i);
                plot_A_Star(obj, O, x_best(1:2), saveDir);
                pause(0.1);
                C(x_best(2), x_best(1), x_best(3)) = 1;
                potentional_routh(1:6,l) = O([2:4,6:8],x_best_i);
                l=l+1;
                O(:,x_best_i) = [];

                if (all(x_best' == target))
                    break;
                end

                neighbors = FindNeighbors(x_best, obj.theta_res);
                for i=1:size(neighbors,2)
                    r = neighbors(1:3,i);
                    r_type = neighbors(4,i);
                    if i ==1
                        g=0;
                    end
                    dist = distance([r',target], obj.theta_res);
                    k = is_x_not_in_O(r);
                    if isempty(k)
```

```
                    O(5,end+1) = g_x_best + neighbor_cost(r_type);
                    O(1,end) = O(5,end) + dist;
                    O(2:4,end) = r;
                    O(6:8,end) = x_best;
                elseif g_x_best+neighbor_cost(r_type)<O(5,k)
                    O(5,k)= g_x_best+neighbor_cost(r_type) ;
                    O(1,k)= O(5,k)+ dist;
                    O(6:8,k) = x_best;
                end
            end
            if isempty(O)
                path = 0;
            end
        end

        save([saveDir '\potentional_routh.mat'], 'potentional_routh');
        calc_route(obj, potentional_routh)

        function y = is_x_not_in_O(r)
            y = find(all(O(2:4,:) == r), 1);
        end

        function plot_A_Star(obj, O, x_best, saveDir)
            persistent n

            if isempty(n)
                n = 1;
            end

            if mod(n,100) == 0
                open_list = O(2:3,:);
                scatter(open_list(1,:)-0.5, open_list(2,:)-0.5, 'MarkerFaceColor',[1 1
1],'MarkerEdgeColor',[0 0 .0]);
                imwrite(frame2im(getframe(gcf)), [saveDir '\' num2str(n) '.jpg']);
            end
            n = n+1;
        end

        function neighbors = FindNeighbors(r_vec, theta_res)
            xi = r_vec(1);
            yi = r_vec(2);
            zi = r_vec(3);
            neighbors_class = cat(3, [4 3 4; 3 2 3; 4 3 4], ones(3,3), [4 3 4; 3 2 3; 4 3 4]); % helps
to define the cost for each neigbor
            if zi==1
                neighborhood = C(yi-1:yi+1, xi-1:xi+1, [theta_res 1 2]);
            elseif zi==theta_res
                neighborhood = C(yi-1:yi+1, xi-1:xi+1,[theta_res-1 theta_res 1]);
            else
                neighborhood = C(yi-1:yi+1, xi-1:xi+1,zi-1:zi+1);
            end
            ind = find(neighborhood == 0);
            [y,x,z] = ind2sub([3 3 3],ind);
            neighbors = [xi+x'-2;yi+y'-2;zi+z'-2;neighbors_class(ind)']; % the '-2' is to center the
neighborhood at [xi, yi, zi]

            neighbors_zeros = neighbors(3,:)==0;
            neighbors(3,neighbors_zeros) = theta_res;

            neighbors_zeros = neighbors(3,:)==theta_res+1;
            neighbors(3,neighbors_zeros) = 1;

        end

        function cost = neighbor_cost(type)
            p = [1 , 1, 1, 1];
            cost = p(type);
        end

        function dist = distance(r,theta_res)
            %This function calculates the distance between any two cartesian coordinates
            dist=sqrt((r(1)-r(4))^2 + (r(2)-r(5))^2 + (min(r(3)-r(6),(theta_res + 1 - r(3))-r(6)))^2);
        end
    end

    function calc_route(obj, potentional_routh)
        j=1;
        i=length(potentional_routh);
        while i>1
            obj.route(j,:) = potentional_routh(1:3,i);
            t(j)=find( potentional_routh(1,1:i-1) == potentional_routh(4,i) &...
                potentional_routh(2,1:i-1) ==potentional_routh(5,i) & ...
                potentional_routh(3,1:i-1) == potentional_routh(6,i));
            i=t(j);
            j=j+1;
        end
        disp(['route length: ' num2str(j) ]);
    end
```

```
        function plot_config_space(obj)
            [x,y,z] = ind2sub(size(obj.config_space),find(obj.config_space == 0));
            figure;
            scatter3(x,y,z, 40, 'o', 'MarkerEdgeColor','k','MarkerFaceColor',[0 .75 .75]);
        end

        function plot_route(obj, saveDir)
            with_norms = false; %set to true will print red lines to mark the normals from each line.

            fig = figure('visible', 'on', 'Units', 'normalized', 'position', [0 0 1 1]); hold on;

            for i=1:length(obj.map.obstacls)
                obj.map.obstacls{i}.print_obstacle(with_norms);
            end

%             obj.map.robot.org_vertices = [0 0; 8 0; 8 1; 0 1]*obj.map.size_factor;
            for i=size(obj.route,1):-1:1
                obj.map.robot.set_location(obj.route(i, 1:2));
                obj.map.robot.set_theta((obj.route(i, 3)-1)*(360/obj.theta_res));
                obj.map.robot.print_obstacle(with_norms);
            end

            axis equal;
            fig.Children.XLim = [0 obj.map.map_size(2)];
            fig.Children.YLim = [0 obj.map.map_size(1)];
            grid minor; title(['calculated robot route']);

            imwrite(frame2im(getframe(gcf)), [saveDir '\final route.jpg']);
            savefig('final route.fig')
        end
    end

    methods (Static)
        function obj_route = main()
            saveDir = datestr(datetime, 'mm-dd-yy HH-MM-SS');
            mkdir(saveDir);

            obj_map = Map.init_question_map();
            obj_route = Route(obj_map, [4 24 0]*obj_map.size_factor, [4 8 0]*obj_map.size_factor);
            obj_route.calc_config_space()
            obj_route.plot_config_space()
            obj_route.A_Star(saveDir)
            obj_route.plot_route(saveDir)
        end
    end
end
```

מחלקת Map

```
classdef Map < handle
    properties
        map_size
        size_factor

        matrix_map
        obstacls
        c_obstacls
        robot

        non_obstacle_i
        obstacle_i
        robot_i
        wall_i
    end

    methods
        function obj = Map(robot, obstacls)
            %UNTITLED Construct an instance of this class
            %   Detailed explanation goes here
            obj.size_factor = 1.5;
            obj.map_size = [30 32]*obj.size_factor; % detarmine in the question

            %inside map index
            obj.obstacle_i = 1;
            obj.non_obstacle_i = 0;
            obj.robot_i = 2;

            % create the robot
            obj.robot = Robot(robot{1}*obj.size_factor, robot{2});

            % creat the map obstacles and c-obstacles
            obj.obstacls = cell(1,length(obstacls));

            % saving the original obstacles objects and the c_obstacls list
```

```
            % for print_map
            for i=1:length(obstacls)
                obj.obstacls{i} = Map_Object(obstacls{i}{1}*obj.size_factor,1, obstacls{i}{2});
            end
        end

        function mat = init_matrix_map(obj)
            mat = zeros(obj.map_size(1), obj.map_size(2));
        end

        function update_matrix_map(obj)
            % the 'imfill' works on binery images. thus we fill the matrix
            % map with the robot (index '2') only after we activate the 'imfill'.
            obj.matrix_map = obj.init_matrix_map();

            for i=1:length(obj.obstacls)
                tmp_mat = obj.draw_object_on_matrix_map(obj.c_obstacls{i}.vertices, obj.obstacle_i,
obj.c_obstacls{i}.norms);
                obj.matrix_map = min((obj.matrix_map + imfill(tmp_mat)),1);
            end
        end

        function mat = get_obstacle_matrix_map(obj)
            % the 'imfill' works on binery images. thus we fill the matrix
            % map with the robot (index '2') only after we activate the 'imfill'.
            mat = obj.init_matrix_map();

            for i=1:length(obj.obstacls)
                tmp_mat = obj.draw_object_on_matrix_map(obj.obstacls{i}.vertices, obj.obstacle_i);
                mat = min((obj.matrix_map + imfill(tmp_mat)),1);
            end
        end

        %%%% print functions %%%
        function fig = get_map(obj, show)
            with_norms = false; %set to true will print red lines to mark the normals from each line.

            fig = figure('visible', show); hold on;
            obj.robot.print_obstacle(with_norms);
            for i=1:length(obj.c_obstacls)
                obj.c_obstacls{i}.print_obstacle(with_norms);
            end
            for i=1:length(obj.obstacls)
                obj.obstacls{i}.print_obstacle(with_norms);
            end
            for i=1:length(obj.c_obstacls)
                obj.c_obstacls{i}.add_label_to_plot;
            end
            axis equal;
            fig.Children.XLim = [0 obj.map_size(2)];
            fig.Children.YLim = [0 obj.map_size(1)];
            grid minor; title(['Configuration space for fixes-\theta = ' num2str(obj.robot.theta, '%.3f')
'^o']);
        end

        function matrix_map = get_map_matrix(obj, show)
            update_matrix_map(obj);

            c = [1 1 1; 0 1 0; 1 0 0]; %obstacles in red, wall in black and robot in blue;

            matrix_map = figure('visible', show);
            [a,b] = size(obj.matrix_map);
            imagesc(0.5,0.5, obj.matrix_map); ax = gca; ax.YDir = 'normal';
            grid minor; axis equal; xlim([0 b]); ylim([0 a]); title(['Discrete configuration space for
fixes-\theta = ' num2str(obj.robot.theta, '%.3f') '^o']);
            colormap(c);
        end

        function print_maps(obj)
            get_map(obj, 'on');
            get_map_matrix(obj, 'on');
        end

        %%%% draw objects and lines %%%%
        function mat = draw_object_on_matrix_map(obj, map_obj, val, norms)
            mat = init_matrix_map(obj);
            %the objects vertices are initilized so the last vertix is the
            %same as the first so a line will be draw between them in this
            %loop - thats why the loop is with '-1';
            max_x = max(map_obj(:,1))-1;
            max_y = max(map_obj(:,2))-1;
            min_x = min(map_obj(:,1))+1;
            min_y = min(map_obj(:,2))+1;

            for i=1:length(map_obj)-1
                if norms(i) >= 0 && norms(i) <= 90
                    x = 1; y=1;
                elseif norms(i) > 90 && norms(i) <= 180
```

```
                    x = -1; y=1;
                elseif norms(i) > -90 && norms(i) < 0
                    x = 1; y=-1;
                elseif norms(i) >= -180 && norms(i) <= -90
                    x =-1; y=-1;
                end
                mat = obj.drwa_line(mat, map_obj(i,:)+1, map_obj(i+1,:)+1, val, [max_x max_y min_x min_y],
x, y);
            end
        end

        function mat = drwa_line(obj, mat, start_point, end_point, val, max_val, x ,y)
            start_point = convret_point_to_map_size_point(obj, start_point, max_val);
            end_point = convret_point_to_map_size_point(obj, end_point, max_val);
            nPoints = 100;

            switch y
                case 1
                    rIndex = ceil(linspace(start_point(2), end_point(2), nPoints));  % Row indices
                case -1
                    rIndex = floor(linspace(start_point(2), end_point(2), nPoints));  % Row indices
                otherwise
                    rIndex = round(linspace(start_point(2), end_point(2), nPoints));  % Row indices
            end
            switch x
                case 1
                    cIndex = ceil(linspace(start_point(1), end_point(1), nPoints));  % Column indices
                case -1
                    cIndex = floor(linspace(start_point(1), end_point(1), nPoints));  % Column indices
                otherwise
                    cIndex = round(linspace(start_point(1), end_point(1), nPoints));  % Column indices
            end

            index = sub2ind(size(mat), rIndex, cIndex);     % Linear indices
            mat(index) = val;
        end

        function map_point = convret_point_to_map_size_point(obj, point, max_val)
            x = max([min([point(1) obj.map_size(2) max_val(1)]), max_val(3),1]);
            y = max([min([point(2) obj.map_size(1) max_val(2)]), max_val(4),1]);
            map_point = [x y];
        end

        %%%% c obstacles %%%%
        function calc_c_obstacles(obj)
            obj.c_obstacls = cell(1,length(obj.obstacls));
            for i=1:length(obj.obstacls)
                obj.c_obstacls{i} = Map_Object(obj.calc_c_obstacle(obj.obstacls{i}), 1,
obj.obstacls{i}.name);
            end
        end

        function c_obstacle = calc_c_obstacle(obj, obstacle)
            robot_dim = obj.robot.vertices; %the method works with the robot relative vertices (the robot
dims)
            m = size(obj.robot.norms,1);
            n = size(obstacle.norms,1);

            var_axes = [1:m 1:n; ones(1,m), ones(1,n)*2; obj.robot.norms(:,1)', obstacle.norms(:,1)'];
%help matrix for the method implementation.
            [sort_var_axes(3,:), i] = sort(var_axes(3,:));
            sort_var_axes(1:2,:) = var_axes(1:2,i);

            x = find(sort_var_axes(2,:) == 2,1);
            y = find(sort_var_axes(2,:) == 1,1);
            sort_var_axes = [sort_var_axes, sort_var_axes(:,1:x+1) + [0 0 360]'];

            c_obstacle = [];
            tmp = [];

            for i=y:length(sort_var_axes)-1
                if sort_var_axes(2,i) == 1
                    tmp = [tmp; robot_dim(sort_var_axes(1,i),:)];
                else
                    if sort_var_axes(3,i) == sort_var_axes(3,i-1)
                        c_obstacle = [c_obstacle; obstacle.vertices(sort_var_axes(1,i),:) - tmp];
                    else
                        tmp = [tmp; robot_dim(sort_var_axes(1,i+1),:)];
                        c_obstacle = [c_obstacle; obstacle.vertices(sort_var_axes(1,i),:) - tmp];
                    end
                    tmp = [];
                end
            end
        end
    end

    methods (Static)
```

```
        function obj = init_question_map()
            robot = {[0 0; 8 0; 8 1; 0 1], 'robot'};
            B0_1 = {[0 0; 32 0; 32 1; 0 1]+[0 29], 'B_{01}'};
            B0_2 = {[0 0; 1 0; 1 30; 0 30]+[0 0], 'B_{02}'};
            B0_3 = {[0 0; 32 0; 32 1; 0 1]+[0 0], 'B_{03}'};
            B0_4 = {[0 0; 1 0; 1 30; 0 30]+[31 0], 'B_{04}'};
%
            B1 = {[0 0; 10 0; 10 1; 0 1]+[0 18], 'B_1'};
            B2 = {[0 0; 1 0; 1 12; 0 12]+[17 17], 'B_2'};
            B3 = {[0 0; 7 0; 7 1; 0 1]+[25 18], 'B_3'};
            B4 = {[0 0; 19 0; 19 1; 0 1]+[0 14], 'B_4'};
            B5 = {[0 0; 8 0; 8 2; 0 2]+[24 13], 'B_5'};
            B6 = {[0 0; 2 0; 2 1; 0 1]+[10 19], 'B_6'};
            B7 = {[0 0; 2 0; 2 1; 0 1]+[23 19], 'B_7'};

            obstacles = {B0_1, B0_2, B0_3, B0_4, B1, B2, B3, B4, B5, B6, B7};

            obj = Map(robot, obstacles);
        end
    end
end
```

מחלקת Map_object

```
classdef Map_Object < handle
    properties
        vertices
        norms
        InOut
        name
    end

    methods
        function obj = Map_Object(vertices, InOut, name)
            % assumse the vertices are counter-clock wish order
            set_vertices(obj, vertices);
            obj.InOut = InOut;
            calc_norms(obj);
            obj.name = name;
        end

        function set_vertices(obj, vertices)
            obj.vertices = vertices;
            obj.vertices(end+1,:) = vertices(1,:);
        end

        function calc_norms(obj)
            % Inout = 1 means out mormals.
            % Inout = -1 means in mormals.
            num = size(obj.vertices,1);
            obj.norms = zeros(num-1, 5);
            for i=1:(num-1)
                x = obj.vertices(i:i+1,1);
                y = obj.vertices(i:i+1,2);
                obj.norms(i,:) = get_norm(x,y,obj.InOut);
            end

            function obj_norm = get_norm(x,y,InOut)
                dy = diff(y);
                dx = diff(x);
                m = (dy/dx);
                % Slope of new line
                L = 0.3*sqrt(dy^2+dx^2);
                minv = -1/m;
                if abs(minv) == Inf
                    obj_norm = [atan2d(L*sign(minv)*sign(InOut),0) , mean(x), mean(x), mean(y), mean(y)-
L*sign(minv)*sign(InOut)];
                else
                    obj_norm = [atan2d(L*minv*sign(dy)*sign(InOut),L*sign(dy)*sign(InOut)), mean(x),
mean(x)-L*sign(dy)*sign(InOut), mean(y), mean(y)-L*minv*sign(dy)*sign(InOut)]; %#ok<*CPROP>
                end
            end
        end

        function add_label_to_plot(obj)
            ver_x = obj.vertices(:,1);
            ver_y = obj.vertices(:,2);
            text(mean(ver_x), mean(ver_y), obj.name , 'HorizontalAlignment' , 'center' ,
'VerticalAlignment' , 'middle')
        end

        function print_obstacle(obj, with_norms)
            ver_x = obj.vertices(:,1);
            ver_y = obj.vertices(:,2);

            if obj.InOut == -1
```

```
                fill(ver_x,ver_y,'b');
                line(ver_x, ver_y, 'Color', 'k');
            elseif obj.InOut == 1
                fill(ver_x,ver_y,'r');
                line(ver_x, ver_y, 'Color', 'k');
            else
                fill(ver_x,ver_y,'g', 'facealpha',.3);
                line(ver_x, ver_y, 'Color', 'g', 'LineStyle', '--');
            end

            if with_norms
                for i=1:size(obj.norms,1)
                    line(obj.location(1) + [obj.norms(i,2); obj.norms(i,3)], obj.location(2) +
[obj.norms(i,4); obj.norms(i,5)], 'Color', 'r');
                end
            end
        end
    end
end
```

מחלקת Robot

```
classdef Robot < Map_Object
    properties
        location
        theta
        org_vertices
    end

    methods
        function obj = Robot(vertices, name)
            % theta needs to be in degree
            % theta > 0 - counterclockwise
            obj@Map_Object(vertices, -1, name);
            obj.org_vertices = vertices;
            obj.location = 0;
            obj.theta = 0;
        end

        function set_theta(obj, theta)
            obj.theta = theta;
            obj.set_vertices(rotate_robot(obj.org_vertices, theta)+obj.location);
            calc_norms(obj)

            function new_vertices = rotate_robot(vertices, theta)
                R = [cosd(theta) -sind(theta); sind(theta) cosd(theta)];
                robot_dim = vertices - vertices(1,:);
                new_vertices = (R*robot_dim')' + vertices(1,:);
            end

        end

        function set_location(obj, location)
            obj.location = location;
            obj.set_vertices(obj.org_vertices+location);
        end
    end

    methods (Static)
        function test_robot()
            obj = Robot([0 0; 0 5; 2 5; 2 0], -1, -30);
            print_obstacle(obj)
        end
    end
end
```