עבודת גמר במקצוע מדעי המחשב

תכנון מסלול עבור רכב אוטונומי בחניה

במסגרת תכנית "אלפא" בטכניון – מכון טכנולוגי לישראל

בוצעה על ידי

אלון קרימגנד אוסובסקי

בהנחיית

מר עומר ניר

במעבדתו של

פרופ"ח אמיר דגני

הפקולטה להנדסה אזרחית וסביבתית, הטכניון.

תודות

אתחיל ואגיד כי העבודה על המחקר הייתה קשה, ארוכה ומאתגרת מאוד. במיוחד בתקופה שאנו נמצאים בה כעת, כשהכל קורה דרך המחשב ולא באופן פרונטלי, לפעמים קשה מאוד לתקשר, לתאם פגישות ולקדם את העבודה. עם זאת, ישנם מספר אנשים שעזרו ותמכו בי במהלך המחקר וכתיבת העבודה הזו, ובלעדיהם כל זה לא היה קורה;

ראשית, אני רוצה להודות למר עומר ניר שהיה המנחה הצמוד שלי במשך יותר משנה שלמה. אפילו שלא יצא לנו להיפגש פעמים רבות פנים מול פנים, תמיד ידעת לכוון אותי לדרך הנכונה וידעת להנחות אותי במקצועיות ובמסירות.

אני מכיר תודה גדולה גם לד״ר אפרת דינרמן, ד״ר סופי כץ, וכמובן לד״ר אורנה עטאר על הנחייה, עזרה טכנית, ואין-ספור תיקונים ושיפורים שדרשתן כדי שהעבודה תצא בדיוק כנדרש. מודה לכן על סבלנותכן ומוקיר לכן תודה מקרב לב!

בנוסף, ברצוני להודות לבית הספר חוגים שבו אני לומד, ובפרט לרכזת אלפא, המחנכת והיועצת אנה בקון. התמיכה האישית, מעקב מקרוב אפילו בפרטים הקטנים ביותר, הדאגה והאהבה שלך היו אמיתיים וכנים מהרגע הראשון. זכיתי בך, ואני שמח ומודה על הקשר שנוצר בינינו.

הייחודיות של תוכנית אלפא היא דווקא החלק החברתי שבה. ברצוני להודות למנחים החברתיים שהפכו את התוכנית הזו לשונה וייחודית – מתן, עמר, חן, וגילי. חיכיתי בציפייה לכל פעילות חברתית שלכם והצלחתם להפתיע, לרגש ולשמח אותי בכל פעם מחדש.

תודה גדולה לפרופ״ח אמיר דגני מהפקולטה להנדסה אזרחית וסביבתית בטכניון שהמחקר כולו נעשה במעבדתו.

ברצוני גם להודות לאבי שהתעניין ותרם מהידע הרב שלו במהלך העבודה. מגיל קטן, אתה גרמת לי להתעניין ולאהוב את תחום המחשבים, ולמדתי ועוד בטוח שאלמד ממך המון בעתיד. אוהב המון!

תודה גם לכל תוכנית אלפא, ובפרט מנהלת התוכנית בטכניון חגית אסף. תודה למרכז מדעני העתיד וקרן מימונידיס, האגף למצטיינים ומחוננים במשרד החינוך, הטכניון והיחידה לנוער שוחר מדע וטכנולוגיה. למדתי בתוכנית זו המון, ואני מאוד שמח שניתנה לי ההזדמנות להשתתף בה.

באהבה, הערכה, והערצה גדולה,
אלון

הקדמה אישית

כבר מגיל קטן, אני זוכר את עצמי מתעניין ברובוטיקה ותכנות. הייתי מתכנת תוכנות שלמות בסקראץ' (סביבת תכנות המאפשרת תכנות בעזרת גרירת וחיבור בלוקים המדמים פקודות), ובונה רובוטים שהיו הורסים את כל הבית ומביאים הרבה צרות להורי.

אני שמח, גאה ומודה שניתנה לי ההזדמנות להשתתף בתוכנית מדהימה כמו אלפא. בעזרת תוכנית זו, הצלחתי סוף סוף להפוך את האהבה העצומה שלי לתכנות לדבר ממשי ומקצועי. כמות המידע שלמדתי לא יכולה להיות מוסברת במילים, והאנשים שהיו סביבי וליוו אותי בכל התהליך הם הטובים ביותר שהייתי יכול לבקש. לפני התוכנית, ללמוד בטכניון הייתה אחת מהשאיפות שלי, וההגעה לטכניון והכניסה למעבדות המחקר באופן קבוע לא מובנת מאליו בעייני עד עצם היום הזה.

אני בטוח שהתוכנית והמחקר ילוו אותי במשך כל חיי.

תוכן עניינים

רשימת איורים

תקציר

הבינה המלאכותית, ובפרט המכוניות האוטונומיות, הוא תחום רחב וחשוב מאוד שתופס תאוצה רבה בעשור האחרון, וחשיבותו ברורה מאליו. טכנולוגיות אלו גם שוות לא מעט כסף: נכון לרגע כתיבת שורות אלו, חברת ייצור ופיתוח הרכבים האוטונומיים המתקדמת בעולם טסלה שווה למעלה מ־640 מיליארד דולר, בעוד ששווין של חברות וותיקות בתחום כמו BMW או Mercedes-Benz עומד על 54 ו־79 מיליארד דולרים בלבד. גם חברת מובילאיי הישראלית, המפתחת מערכות ראיה ממחושבות ובינה מלאכותית למכוניות, נמכרה בשנת 2017 לחברת אינטל תמורת 15.3 מיליארד דולרים.

האמת היא, שליצור תוכנה וחומרה המודעת לסביבה ויכולה לנוע ממקום למקום בצורה יעילה ובטוחה היא משימה מורכבת מאוד. משימה זו הופכת להיות מורכבת פי כמה וכמה כשאנחנו יוצאים מהמעבדה ומדברים על מכוניות אוטונומיות, שצריכות להתמודד עם סוגים שונים של כבישים, חוקי תנועה ומכשולים הנמצאים בתנועה כמו מכוניות אחרות ועוברי אורח.

במחקר, בחנתי והשוואתי אלגוריתמים ידועים וכלליים, כאשר התאמתי אותם לבעיית החנייה. על סמך השוואות אלו, הצעתי אלגוריתם משופר שמיועד באופן ספציפי לבעיית חניית הרכב האוטומטי.

למחקרי חשיבות רבה: יעילות האלגוריתם משפיעה ישירות על מהירות התגובה, הדיוק והאיכות של המכונית. ככל שהאלגוריתם יעיל יותר, כך הוא טוב יותר. מציאת אלגוריתמים ממוקדים ויעילים לבעיות ספציפיות הופך את הפתרון של הבעיות למהיר יותר, ובמכוניות אוטונומיות דבר זה מתרגם למדויק יותר, בטוח יותר (משום שזמן התגובה מהיר יותר), ובסופו של דבר גם זול יותר (מכיוון שאם האלגוריתם יעיל, גם מחשב פחות חזק יוכל להריץ אותו).

ההשוואה בין האלגוריתמים בוצעה על ידי שני פרמטרים – "זמן" ריצת האלגוריתם, ואורך הדרך שנמצאה (אם בכלל נמצאה דרך). תוצאות המחקר הראו כי לכאורה אלגוריתם ה־A* הוא האלגוריתם המתאים והמהיר ביותר לבעיה, שכן הדרך שנמצאה בעזרתו היא הדרך הקצרה ביותר, וגם "זמן" הריצה שלו טוב ביחסית לשאר האלגוריתמים. למרות זאת, אלגוריתם ה־RRT (Rapidly-exploring Random Tree), שנפוץ ונהוג לשימוש בבעיות "צפופות" (עם מכשולים רבים) או בבעיות רב ממדיות דווקא כשל עם 48% הצלחה בלבד.

האלגוריתם שהצענו הוא שדרוג של אלגוריתם ה־RRT ש"ממקד" את החיפוש לכיוון נקודת היעד. למרות שהדרך שאלגוריתם זה מוצא היא לא אופטימלית (מבחינת אורך המסלול), נמצא כי אלגוריתם זה הוא יעיל מאוד ומוצא מסלול ביעילות טובה יותר פי 1.5 מה־A*.

למרות התוצאות המעודדות, עדיין נדרש מחקר נוסף. מכיוון שהאמצעים שלנו מוגבלים, הסימולציה שיצרנו למכונית הייתה "גסה" יחסית, והמודל היה פשטני. כדי לאשר את יעילות האלגוריתם נצטרך בדיקה על מודל מדויק ונאמן יותר.

1. מבוא

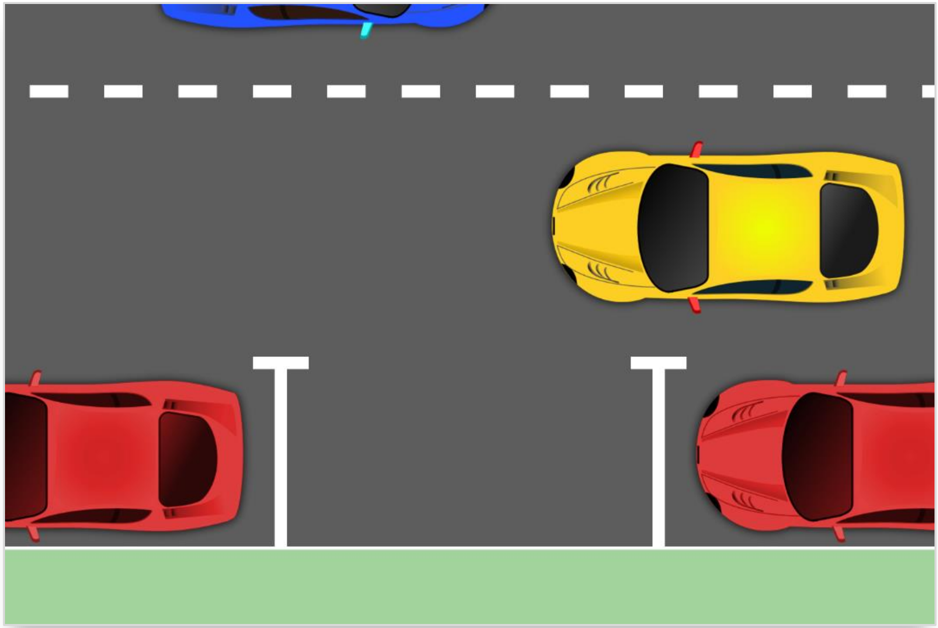
תכנון ומציאת דרכים של רובוטים שונים במרחב הוא תת תחום ברובוטיקה. לתת תחום זה נוכל למצוא אין־ספור שימושים בחיי היום־יום, החל מרובוט "פשוט" במפעל המבצע את אותה הפעולה עשרות אלפי פעמים ומזיז חפצים ממקום למקום בפס היצור, ועד ה־iRobot לדוגמא – שואב אבק חכם היודע להתמצא בבית, ולשאוב את החדרים בבית ולחזור לעמדת הטעינה שלו לבדו. דוגמא נוספת שבה קיים השימוש בתחום הזה בדיוק הוא מכוניות אוטונומיות למיניהן, אשר משתמשות בעשרות חיישנים בו זמנית להבנת וקליטת הסביבה, ניתוח שלה וחישוב הפעולות הבאות של המכונית.

לפתרון בעיות תנועה ותזוזה מנקודת התחלה לנקודת סיום מסוימת (כמו הבעיה שאנו מנסים לפתור) יש אין־ספור גישות שונות. עם זאת, אחת מהגישות הנפוצות ביותר בתחום היא הפיכת בעיית התנועה לגרף, והרצת אלגוריתם חיפוש כלשהוא על הגרף שנוצר למציאת מסלול מנקודה לנקודה. בעזרת שיטה זו ניתן להפוך כמעט כל בעיית תנועה לגרף, ולאחר מכן להפעיל אחת מעשרות (אם לא מאות) אלגוריתמי החיפוש שקיימים על גרפים – שהוכחו כיעילים. שיטה זו אולי נשמעת נהדר במחשבה ראשונית, אך האמת היא שהיא הרבה יותר מסובכת ממה שהיא נראית: לעיתים ישנה יותר מדרך אחת שבה ניתן להפוך בעיית תנועה אחת לגרף, ובעיות מסוימות יכולות להיות מסובכות מידי ל"פישוט" כה אגרסיבי על ידי גרף.

בעיית התנועה שאנחנו מתארים – בעיית תכנון קינמטי עם אילוצים לא הולונומיים, מסובכת יחסית ומכילה שלושה ממדים שונים (מיקום דו־מימדי וסיבוב המכונית). כאשר נהפוך אותה לגרף אנו נגלה שהגרף שקיבלנו גדול מאוד, והחיפוש בו בעזרת אלגוריתמי החיפוש הקיימים לא יעיל במיוחד.

1.1 שאלות המחקר

נפתח סביבה ממוחשבת (סימולציה) שבה ניתן ליצור מכשולים ולהניע את המכונית. בסביבה זו נעצב מספר מקרים שאותם נרצה לחקור המתארים חנייה במקביל (איור 1). את הבעיה נהפוך לגרף, ונחפש מסלולים בגרף זה בעזרת אלגוריתמי החיפוש BFS, A* ו־RRT. מטרת המחקר היא להשוות את האלגוריתמים האלו בבעיית החנייה במקביל, ובנוסף להציע אלגוריתם חדש העוקף את ביצועיהם של האלגוריתמים המוכרים.



איור 1 – המחשת בעיית החנייה במקביל

1.2 השערת המחקר

בין האלגוריתמים שנשווה, אנו מעריכים כי אחד אלגוריתם ה־BFS לא יניב תוצאות טובות מיוחד, וההשוואה המעניינת יותר תתבצע בין ה־A^* ל־RRT. אלגוריתם ה־BFS לא יעיל במיוחד, וזאת מכיוון שסורק את כל הצמתים במרחק n מנקודת ההתחלה לפני שממשיך לסרוק את הצמתים במרחק $n + 1$. לעומתו, לאלגוריתם ה־A^* ישנה היכולת "לדלג" על שלבים שלמים של צמתים, ולכן, ברוב המקרים, הוא נמצא ליעיל יותר. אלגוריתם ה־RRT לעומת זאת אינו מבוסס כלל על Forward Search, ולכן לא קיימת דרך קלה לחזות איך תוצאותיו יראו לעומת תוצאות ה־A^* או ה־BFS. אנו משערים כי בעזרת האלגוריתם החדש שנציע, נשלב בין יתרונות ה־RRT וה־A^*, ונוכל להציג תוצאות טובות יותר.

2. סקירה ספרותית

לפתרון בעיות תנועה ותזוזה מנקודת התחלה לנקודת סיום מסוימת (כמו הבעיה שאנו מנסים לפתור) יש אין-ספור גישות שונות. בתור התחלה, נסתכל על בעיה כללית ופשוטה ביותר: נדמיין מגרש כדורגל ריק לגמרי, שבקצהו האחד ישנו שחקן שכל מטרתו היא להגיע לקצה השני במסלול הקצר ביותר. מחוקי הגאומטריה, ברור שבמקרה כזה הדרך האופטימלית והקצרה ביותר היא הקו הישר היחיד שעובר בין הנקודה ההתחלתית והנקודה הסופית של השחקן במגרש. מצד שני, ברור שישנן אינסוף דרכים שבהם השחקן יכול לבחור ולרוץ לאורכו של המגרש! כעת, נוסיף למגרש מכשולים שונים, שבין היתר, יחסמו את הדרך האופטימלית בין נקודת ההתחלה לסיום שדיברנו עליה קודם לכן. כיצד נוכל למצוא את הדרך האופטימלית כעת?

אחת הגישות הפופולאריות לגישה לבעיות מסוג זה (ואף בעיות מסובכות בהרבה מזאת) היא על ידי הפיכת הבעיה לבעיה פשוטה יותר. כאמור, ישנן אינסוף דרכים שבהן השחקן יכול לבחור ולרוץ לאורכו של המגרש, ולכן לבדוק כל אחת מהן בעזרת מחשב יהיה בלתי אפשרי. נדמיין מצב שבו על המגרש, בנוסף למכשולים, נמקם נקודות מיוחדות במקומות שונים. נגדיר לשחקן את החוק הבא: מכל נקודה שכזו, הוא יכול לרוץ לכל נקודה אחרת שנמצאת בשדה ראייתו במגרש בקו ישר (הוא אינו יכול לרוץ לנקודות שמוסתרות על ידי המכשולים). כמובן שגם נקודת ההתחלה והסיום הן נקודות כאלה. למעשה, הפכנו את הבעיה במגרש לפשוטה הרבה יותר, מכיוון שכעת ישנו מספר "נקודות" מוגבל שבהם השחקן יכול להיות, ומכל נקודה ישנו מספר מוגבל של דרכים שאליהם הוא יכול להמשיך ללכת. האמת היא, שמה שעשינו עכשיו היה העברת הבעיה במגרש לגרף. [1]

2.1 מבוא לתורת הגרפים

גרף הוא צורה לייצוג מספר מצבים של מודל (כמו מודל המכונית הנתון). הגרף מורכב ממצבים (States) שונים הנקראים "צמתים" (Vertices, Nodes) כשבניהם ניצבים קשתות או צלעות (Edges) המייצגים את המעבר או ה"דלתא" בין הצמתים. הגרפים משמשים לתיאור בעיות תנועה מסובכות והפיכתן לפשוטות יותר. בתור דוגמא, ניתן להפשיט את בעיית התנועה במדינה כך שכל הצמתים בגרף ייצגו ערים, והדרכים או הכבישים ביניהם ייוצגו על ידי הקשתות. כדי למצוא דרך בין עיר אחת לאחרת, נסמן את הראשונה כנקודת ההתחלה והשנייה כנקודת הסיום, ונעביר את הגרף לאחד מהאלגוריתמים הרבים לחיפוש בגרף, ונקבל מסלול שיחבר בין שתי הצמתים ויחזיר מסלול שיורכב מקשתות. המסלול האופטימלי בבעיה מסוג זה יהיה המסלול שבו סכום אורכי הקשתות הוא הקטן ביותר. [2]

2.1.1 סוגי גרפים

קיימים שלל סוגי גרפים שונים לבעיות מסובכות יותר ופחות:

גרף ממושקל הוא גרף שבו לכל קשת יש משקל מסוים. המסלול האופטימלי בין שני צמתים שונים יהיה המסלול שבו סכום המשקלים של הקשתות בו הוא הקטן ביותר. בדוגמא הנתונה מעלה, ניתן לסמן את המשקל של כל כביש באורך האוקלידי שלו (בק"מ לדוגמא), או אפילו בזמן שלוקח לרכב ממוצע לעבור דרכו. על ידי הפעלת אלגוריתמים כמו "האלגוריתם של דייקסטרה" (יפורט בהמשך), ניתן למצוא את המסלול האופטימלי בין שתי נקודות, ובדרך דומה פועלות אפליקציות ניווט שונות.

גרף מכוון הוא גרף שבו לפחות קשת אחת היא חד-כיוונית, כלומר, יהיה ניתן לעבור מצומת אחת לשנייה, אך לא להפך.

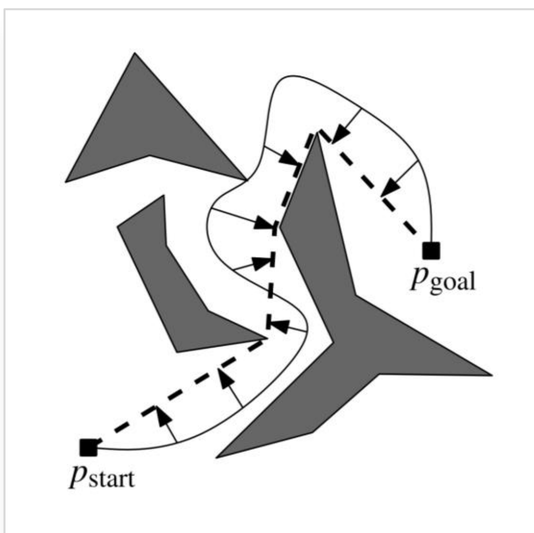
כל גרף אמור לתאר את הבעיה שלשמה הוא קיים בצורה הטובה ביותר, ומהסיבה הזו קיימים אין-ספור וריאציות שונות של גרפים שהורכבו במיוחד בשביל בעיות ספציפיות שונות. [3]

2.2 דרכים שונות לבניית גרפים

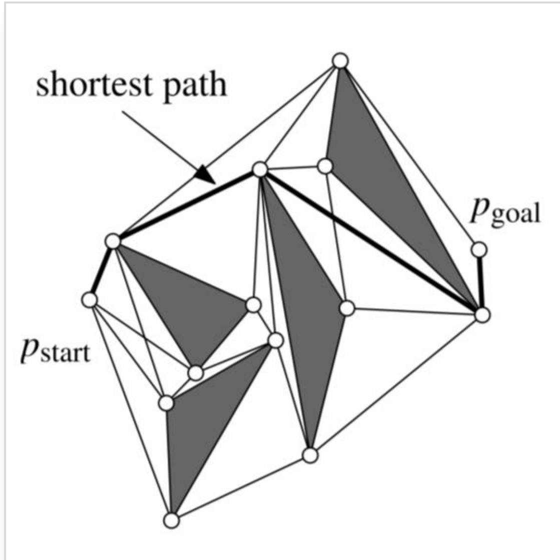
ישנן אין-ספור דרכים שונות לבניית הגרף על סמך הבעיה הנתונה, ואין דרך אחת שמתאימה לכלל הבעיות. נתאר לעצמנו בעיה דו ממדית עם מכשולים מצולעים: אנו יודעים שהדרך המהירה ביותר לחיבור שתי נקודות היא בעזרת קו ישר. נחשוב על זה כמו סוג של גומייה - אם נעביר אותה בין המכשולים היא לא תיגע בהם, אך ברגע שנמתח אותה מההתחלה והסוף נראה שהיא הופכת למקבץ קטעים ישרים, כשהם "נשברים" בקודקודים של המכשולים (

איור 2). מכאן, אנחנו מסיקים שהדרך הקצרה ביותר בין שתי נקודות בבעיה עם מכשולים, חייבת לעבור בין ישרים שהקצוות שלהם הם קודקודים של המכשולים. נוכל ליצור גרף שהצמתים שלו הם קודקודי המכשולים, והקשתות בו הן כל הישרים שמחברים בין קצוות המכשולים ולא עוברות דרך המכשולים עצמם - קשתות אלו כוללות גם את צלעות המכשולים עצמם (

איור 3). לגרף מסוג זה נהוג לקרוא Visibility graph.



איור 2 – הדרך הקצרה ביותר בין שתי נקודות עם מכשולים. [4]

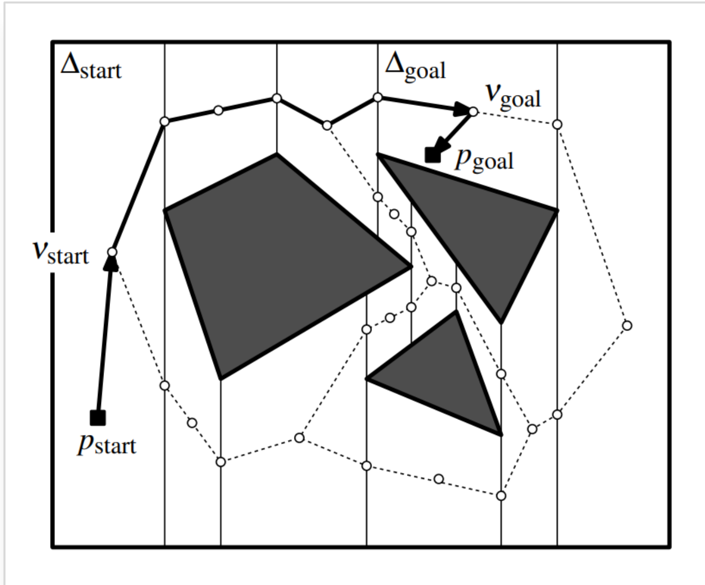


איור 3 – גרף המחבר קצוות מכשולים. [4]

היתרון בגרף מסוג זה הוא ברור: הדרך שתתקבל על ידי חיפוש בגרף בין שתי נקודות, תהיה הדרך הקצרה ביותר ביניהם. אך לשיטה זו, מספר חסרונות: בראש ובראשונה, הדרך שתתקבל תעבור דרך קודקודי המכשולים, ויתכן שגם על צלעותיהם, במקרה אמיתי, מצב שבו רובוט עובר על צלעות המכשולים הוא פשוט לא הגיוני, ורובוטים רבים מסתמכים על סוג של פרמטר "מרחק ביטחון" קבוע ביניהם למכשול בכל מצב.

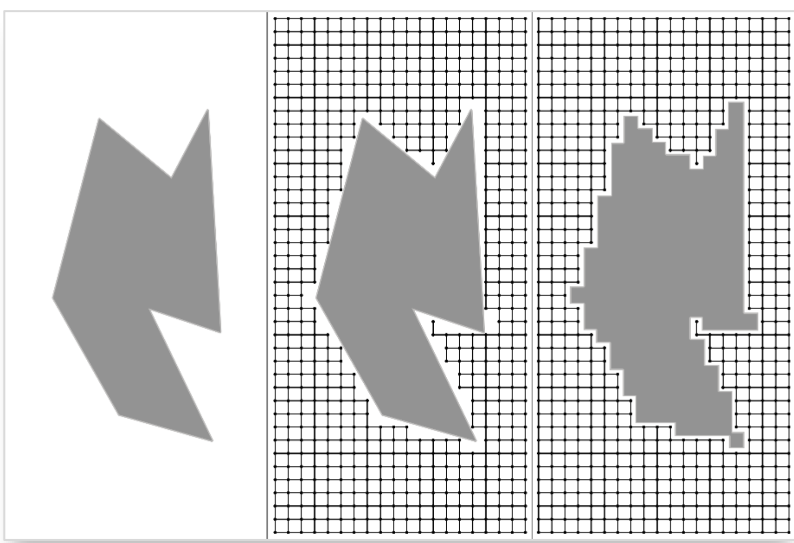
גישה נוספת לבעיה זו היא יצירת גרף מהמסלולים הבטוחים ביותר בין המכשולים, כלומר המסלולים שנמצאים רחוק ביותר מבין כל מכשול. כדי לבנות גרף שכזה, נעביר קו לכיוון קבוע (בדרך כלל אנכי או אופקי) בכל קודקוד של מכשול, ונקבל טרפזים בין המכשולים. במפה שקיבלנו, נסמן את אמצעי הקווים שסימנו ואת אמצעי הטרפזים בתור צמתים בגרף, ואת הדרכים שמחברות את הנקודות בתור קשתות. למפה הזאת, קוראים מפה טרפזית או Trapezoidal Map (

איור 4). [4]



איור 4 – מפה טרפזית, הגרף שנוצר איתה ומציאת דרך בית שני נקודות בעזרתה. [4]

דרך נוספת, ואולי הפופולארית ביותר היא בעזרת הדיסקרטיזציה (Discretization, מהמילה discrete – אינו רציף, בדיד) של המרחב. בתהליך הדיסקרטיזציה, נחלק את המרחב לחלקים שווים (בדרך כלל ברשת), כאשר החיבורים בין המשבצות בו יהיו הקשתות, ומרכז כל משבצת יהיה צומת. כל משבצת אשר נוגעת במכשול לא תהיה חלק בגרף (איור 5). שיטה זו מורידה את איכות החיפוש, אך ככל שהמשבצות יהיו קטנות יותר כך התוצאה תהיה יותר קרובה לחיפוש רציף (שאפשר להגדיר אותו גם בתור דיסקרטיזציה עם משבצות בגודל 0).



איור 5 – הפיכת מרחב דו ממדי עם מכשול לגרף בעזרת דיסקרטיזציה.

2.3 חיפוש דרכים בגרפים

לאחר שהפכנו את בעיית התנועה במרחב לבעיה בגרף, קיימים מספר סוגים של אלגוריתמים שבעזרתם נוכל לסרוק את הגרף ולמצוא דרך מנקודה לנקודה. לכל אלגוריתם יתרונות וחסרונות - חלקם יעילים ומהירים יותר אך התוצאות שיביאו לא מושלמות, ואחרים איטיים הרבה יותר אך מוצאים את המסלול האופטימלי.

תבנית האלגוריתם **Forward Search** (קטע קוד 1) משומשת על ידי כל האלגוריתמים שאציין מטה. בכל שלב במהלך ריצת האלגוריתם הנקודות בשטח יתחלקו לשלושה סוגים: **(1) נקודות שלא בוקרו כלל** (בתחילת הריצה: כל הנקודות X מלבד x_I), **(2) נקודות "חיות"** (שנמצאות במערך Q), כלומר נקודת שבוקרו אבל שיתכן כי הנקודות אחריהן לא בוקרו (כלומר יתכן שלפחות אחת הנקודות x` של הנקודה עוד לא בוקרה), או **(3) נקודות "מתות"** – נקודות שבוקרו וגם כל הנקודות אחריהן בוקרו. האלגוריתם יתחיל ב x_I, כאשר היא נקודה חיה (שורות 1,2). כל האלגוריתם יחזור בתוך לולאה כל עוד ישנם נקודות חיות או עד שנמצא x_G – כלומר האלגוריתם יעבוד עד שכל הגרף נחקר ולא נמצאה הנקודה הסופית, או עד שהיא נמצאה (במקרה שהגרף אין-סופי ואין עליו נקודה סופית, האלגוריתם ימשיך לחפש עד אין-סוף). כל סיבוב של הלולאה מתמקד על נקודה חיה אחת: תחילה, האלגוריתם בודק אם הנקודה הנוכחית היא x_G (שורה 6 בקטע קוד 1). אם זאת לא הנקודה, הוא יוצר נקודות חדשות x'. כל נקודה חדשה שנוצרה ועדיין לא בוקרה (שורה 9 בקטע קוד 1), הופכת לנקודה חיה ונוספת לסוף מערך הנקודות החיות (שורה 11 בקטע קוד 1). אם הנקודה כבר בוקרה, יתכן שבחלק האלגוריתמים נצטרך לטפל בצורה כזאת או אחרת ב x' (שורה 13 בקטע קוד 1). השוני העיקרי בין כל האלגוריתמים המבוססים על הForward Search הוא סדר הסריקה של הנקודות שלהם, או יותר נכון הדרך שבה המערך Q מסודר. [2]

1. נבנה מערך Q ונכניס לתוכו את הנקודה x_I.
2. נסמן את x_I בתור נקודה שכבר ביקרנו בה.
3. בצע עד ש- Q ריק:
4. הצב ב x את הערך הראשון ב Q.
5. מחק את הערך הראשון ב Q.
6. אם x הוא x_G:
7. נמצאה דרך בין x_I ל x_G.
8. עבור כל צומת מהצמתים שאליה אפשר להתקדם מx (נסמן בתור x'):
9. אם x' עדיין לא בוקרה:
10. נסמן את x' בתור נקודה שכבר ביקרנו בה.
11. נוסיף את x' למערך Q.
12. אחרת:
13. נטפל ב x' המשוכפל (יפורט בהמשך).
14. לא נמצאה דרך בין x_I ל x_G.

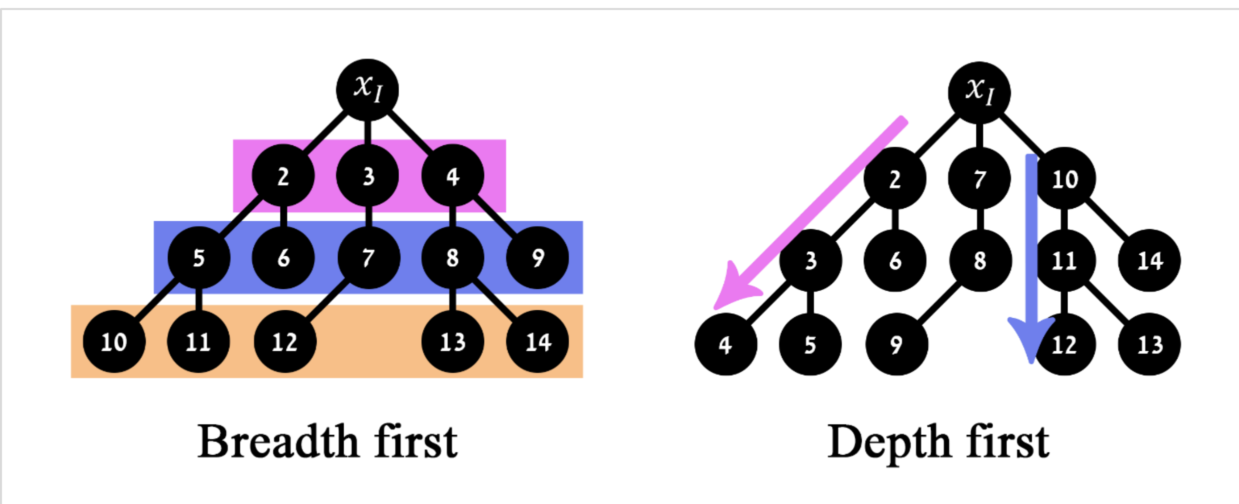
קטע קוד 1 – תבנית כל אלגוריתם חיפוש בסיסי קדימה (Forward Search). [2]

2.3.1 אלגוריתמים הבנויים על ה־Forward Search

Breadth First Search : באלגוריתם זה, המערך Q מסודר בשיטה בשם First-In First-Out (או בקיצור, FIFO). בשיטה זו, הנקודה הראשונה שנוספת למערך תהיה גם הראשונה שתצא. כך, האלגוריתם מבטיח שהדרך הראשון שתצא לנקודה הסופית תהיה גם הדרך הקצרה ביותר, מכיוון שבכל שלב באלגוריתם הדרך לנקודה מסוימת תיקח k צעדים, והדרך לנקודה שאחריה תיקח $k+1$ צעדים. שיטה זו גם אומרת כי בשורה 13 בקטע קוד 1 הנתון לא צריך לעשות כלום, מכיוון שהדרך הקצרה ביותר להגעה לנקודה מסוימת היא המסלול הראשון שנמצא עליה.

Depth First Search: באלגוריתם זה, המערך Q יסודר בשיטה הפוכה לגמרי מהשיטה ב-Breadth First (בשיטה הנקראת First-in Last-out), כלומר הנקודה שתתווסף ראשונה למערך תהיה זאת שתצא ממנו אחרונה. בשיטה זו, האלגוריתם יתקדם בכיוון מסוים, ורק כאשר יגיע לצומת שממנה לא יוכל להתקדם, יחזור שלב אחד אחורה וילך למסלול "מקביל" (איור 6). אלגוריתם זה נחשב ליעיל פחות מכיוון שהוא מתבסס בעיקר על מזל, ויהיו מצבים שבו נקודת הסיום תהיה במרחק קשת אחת מנקודת ההתחלה, אלגוריתם זה יספיק חלק גדול מהגרף עד שימצא את הדרך (אם לא את כולו). בנוסף, אלגוריתם זה עובד על גרפים סופיים בלבד, מכיוון שאם ננסה לממש אותו על גרף אין-סופי, נגלה כי הוא יחקור בכיוון אחד בלבד. גם באלגוריתם זה, אם חזרנו לאותה הנקודה פעמיים אין צורך לעשות דבר נוסף בשורה 13 בקטע קוד 1. [2]

נסתכל על איור 6 : שני האלגוריתמים תמיד יחתרו ללכת שמאלה ולמטה, ובדוגמא לא נמצאת נקודת מטרה. ניתן לראות כי אלגוריתם החיפוש לרוחק סורק קודם את כל הנקודות שנמצאות במרחק 1 מנקודת ההתחלה, ורק לאחר מכן ממשיך לנקודות הנמצאות במרחק 2 (וכך הלאה), בזמן שאלגוריתם החיפוש לעומק יסרוק את כל הנקודות לעומק בכיוון אחד, ורק כשיגיע לדרך ללא מוצא יחזור אחורה.



איור 6 – דוגמא של חיפוש לרוחב מול חיפוש לעומק.

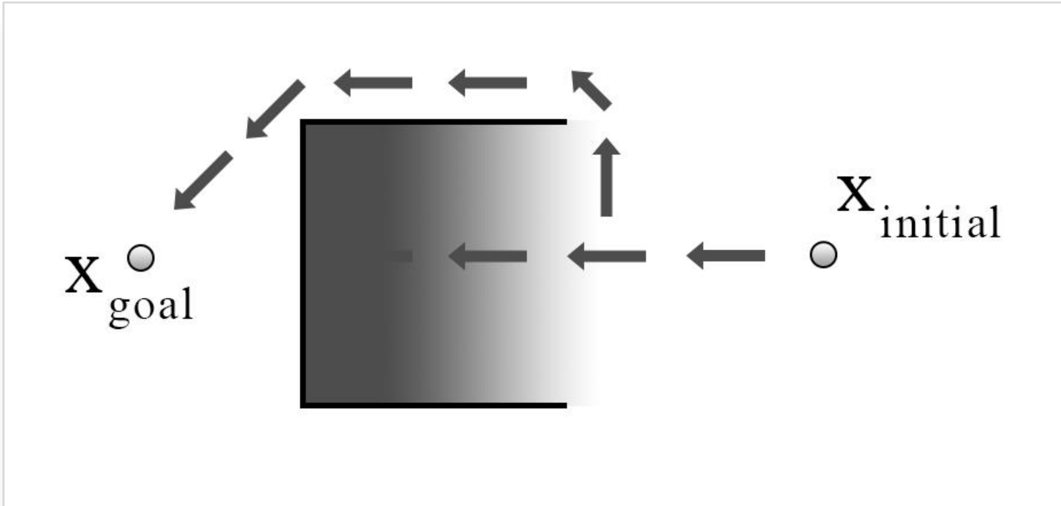
Dijkstra's Algorithm: בניגוד לאלגוריתמים הקודמים, שידעו להתמודד רק עם גרפים דו-כיוונים ולא ממושקלים, אלגוריתם זה יודע להתמודד עם שני סוגי הגרפים אלו. אלגוריתם זה הוא אלגוריתם בסיסי ומפורסם מאוד, כשעליו נשענים הרבה אלגוריתמים חדשים ומתוחכמים הרבה יותר. מטרת האלגוריתם תהיה למצוא את הדרך החסכונית ביותר בין צומת ההתחלה לסיום, כלומר הדרך שבה סכום הקשתות בה יהיה הנמוך ביותר. ניצור לנו פונקציה חדשה $C(x)$ – הפונקציה תגדיר את המחיר הזול ביותר הידוע כרגע שלוקח להגיע מצומת x_I לצומת x. אם אנחנו יודעים בוודאות שהמחיר הזה הוא הזול ביותר, נסמן אותו בתור $C^*(x)$. נתחיל בצומת ההתחלה x_I, כאשר $C^*(x_I) = 0$. משם, עבור כל x' חדש שנוצר (שורה 8 באלגוריתם הנתון מעלה), נריץ פעולה חדשה לחישוב $C(x')$ שתיקח את המחיר הסופי לצומת x (אפשר גם לסמן בתור $C^*(x)$) ותוסיף לו את מחיר הקשת המחברת את שני הצמתים. בגלל שיתכן שנגיע לצומת x' מעוד דרכים ויתכן שהן יהיו יותר חסכוניות, עדיין לא נוכל לסמן את המחיר הסופי C^*. וזה בדיוק מה שנעשה בשורה 13 – אם הגענו לנקודה שכבר ביקרנו בה, נבדוק האם ה C החדש שלנו קטן יותר מה C השמור לנקודה, ואם כן, נעדכן אותו. באלגוריתם הזה, נסדר את רשימת ה Q לפי ה C השמור לכל נקודה, כאשר בכל פעם "נמשוך" מהרשימה את הנקודה עם הערך הקטן ביותר. [1, 3]

A* Search Algorithm: אלגוריתם זה משתמש בשיטה של האלגוריתם של דייקסטרה, אך שבו נשתמש במשתנה נוסף לכל נקודה, $G(x)$, שיציג חסם תחתון לדרך בין צומת x לצומת הסופית x_G. כמובן שהדבר הזה תלוי מאוד בבעיה הנתונה, אך אם נסתכל על בעיה דו ממדית לדוגמא, נוכל לייצג כל נקודה בעזרת שני ממדים (i,j), ונוכל לתת הערכה כזאת בעזרת החישוב $G(x) = \sqrt{\left(i_x - i_{x_G}\right)^2 + \left(j_x - j_{x_G}\right)^2}$ שמייצג את המרחק האוקלידי בין הנקודה x לנקודה x_G ללא התחשבות במכשולים בדרך. האלגוריתם משנה את פונקציית המיון של Q שתתחשב גם בערך החדש של G, כלומר $C(x) + G(x)$. היתרון של אלגוריתם זה הוא היעילות שלו: אלגוריתם זה יתחשב גם בדרך שעבר עד ההגעה לנקודה המסוימת על הגרף וגם על הדרך שיקח לו להגיע מהנקודה הנוכחית ועד לנקודת הסיום, וברוב המקרים דרך זה ייעל אותו פי כמה לעומת האלגוריתמים שהוזכרו כאן קודם. [1, 4]

2.3.2 אלגוריתם ה־RRT

בעיה באלגוריתם ה-A* היא שבמקרים רבים הוא יתנהג בצורה חדה מידיי, ויתכן כי ימשוך יותר מידיי לכיוון נקודת הסיום. כדי לפתור בעיה זו, נצטרך להכניס אלמנט רנדומלי לחיפוש המסלול, בעזרת אלגוריתם חיפוש רנדומלי כמו ה-RRT.

נסתכל על איור 7: ניתן לראות כי האלגוריתם מושך את החיפוש לכיוון נקודת הסיום, ונתקע במכשול. בגלל שאלגוריתם ה-A* מושך את החיפוש לכיוון נקודת הסיום, החיפוש יצטרך "למלא" את כל המכשול בצורת ה"C" עד שיצליח לברוח באחד מהצדדים. בבעיה מסוג זה, אלגוריתם ה-A* לא יעיל בעליל, ובדיוק מסיבה זו נצטרך אלגוריתם שלפחות חלק ממנו מסתמך על אקראיות שתמשוך את החיפוש לכל הכיוונים.

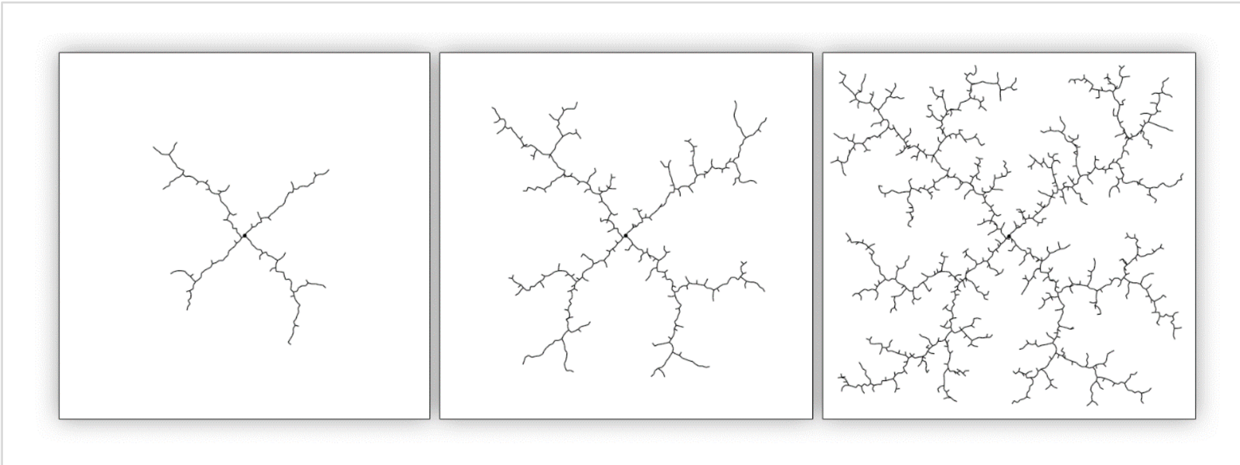


איור 7 – דוגמא לבעיה עם מכשול בצורת "C" ואלגוריתם החיפוש $A*$.

שלבי אלגוריתם ה-RRT: נתחיל כאשר נתון לנו המצב ההתחלתי בלבד $x_{initial}$, ואזור או קבוצת מצבי סיום $X_{goal} \subset X$, כאשר X מכיל בתוכו את כל המצבים האפשריים. בהנחה שהשטח שלנו סופי ובעל גבולות, נזרוק עליו נקודה רנדומלית, ונסמן אותה בתור x_{rand}. מהנקודה ההתחלתית, ננסה להתקדם בכיוון הנקודה הרנדומלית שמצאנו, מרחק קבוע מראש (שיהיה בדרך כלל קטן יחסית). אם הגענו אליה, נוסיף אותה בתור צומת לגרף, אם נתקענו במכשול בדרך, נמשיך מבלי להוסיף אותה בתור צומת, ואם הלכנו בכיוון הנקודה אך לא הגענו אליה ונעצרנו בגלל מגבלת המרחק אך לא נתקענו במכשול, נסמן את הנקודה שעצרנו בה בתור צומת בגרף והדרך שעברנו תהיה הדרך בגרף.

נמשיך לבצע את השלבים האלו באלגוריתם שוב ושוב, רק שעכשיו בכל פעם שנגריל נקודה רנדומלית חדשה x_{rand}, נבדוק מהי הנקודה הקרובה ביותר בגרף שכבר ביקרנו בה, וממנה נתקדם לכיוון הנקודה הרנדומלית. כאמור, נבצע שלבים אלו שוב ושוב עד שנגיע ל X_{goal}. [7]

יתרונות אלגוריתם ה-RRT: כאמור, אלגוריתם חיפוש זה מבוסס על חיפוש רנדומלי, ולכן בהינתן בעיה כמו הבעיה באיור 7, האלגוריתם יצטרך להגריל צמתים בודדים עד שתימצא דרך מנקודת ההתחלה לסיום. דבר זה נובע בגלל ההתפשטות הרחבה שלו לכל הכיוונים (איור 8), וה"רצון" של האלגוריתם להתקדם דווקא לנקודות ולאזורים שלא בוקרו. בגלל שהאלגוריתם לא באמת סורק כל נקודה ונקודה אך בפועל מכסה הרבה שטח תוך זמן קצר, הוא יעיל במיוחד בעבודה על ממדים רבים, מעברים צרים ומכשולים מרובים. בנוסף, אלגוריתם זה הוא שלם, כלומר אם קיימת דרך בין שתי נקודות, היא תימצא בסופו של דבר.

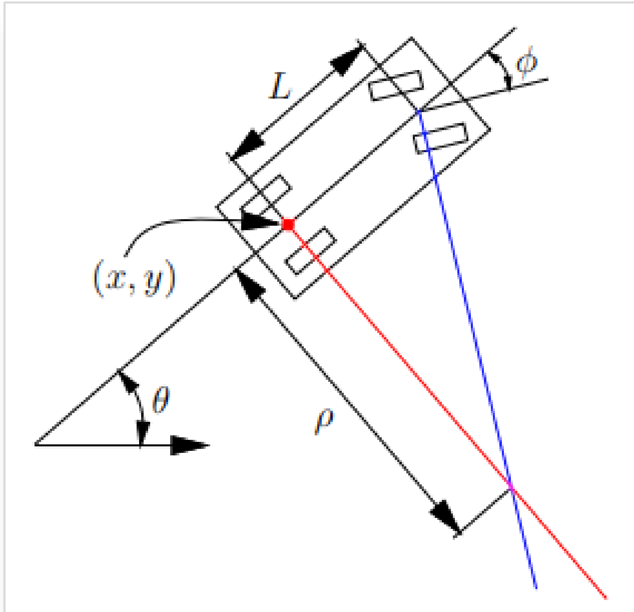


איור 8 – הדגמה של התפשטות החיפוש באלגוריתם ה-RRT. ניתן לראות כי בזמן קצר, החיפוש הגיע לכמעט כל נקודה במשטח, ואפילו לקצותיו ופינותיו, דבר שהיה לוקח יותר זמן באלגוריתם כמו BFS. [7]

חסרונות אלגוריתם ה-RRT: למרות כל היתרונות שאלגוריתם ה-RRT מספק, מכיוון שהוא מבוסס כולו על הגרלה אקראית, הדרכים שימצא באותה הבעיה בדיוק יהיו שונות בכל פעם ולא אופטימליות (כלומר, לא הקצרות ביותר) בניגוד ל-BFS או אפילו *A. [7]

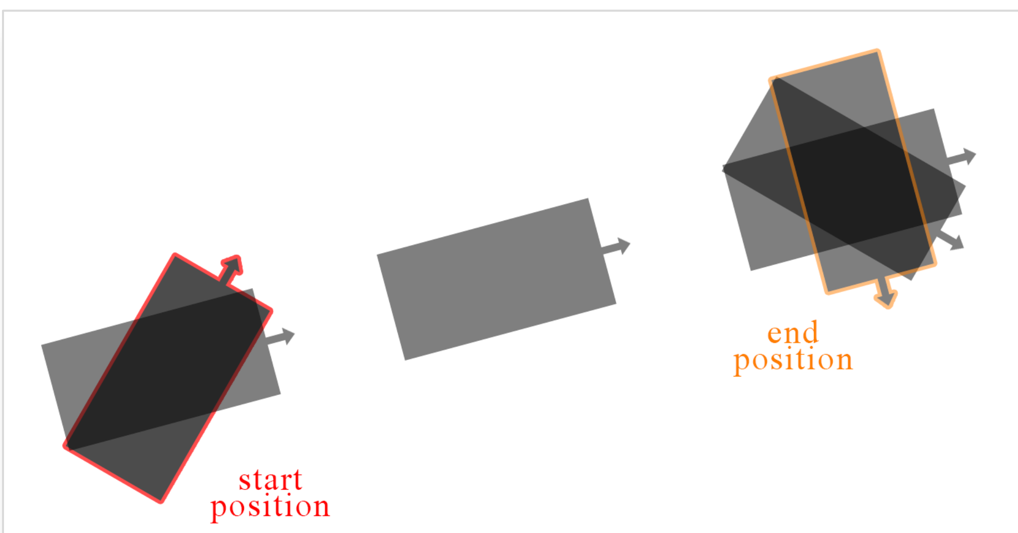
2.4 Dubins Path

נתבונן על איור 9 : למכונית הנתונה ישנן 3 דרגות חופש – מיקום על ציר x, מיקום על ציר y, וסיבוב המכונית המיוצג על ידי θ. אורך המכונית, או ליתר הדיוק המרחק שבין הגלגלים הקדמיים והאחוריים שלה מיוצג על ידי L. זווית הסיבוב של הגה המכונית מיוצגת על ידי ϕ, כאשר ρ הוא הרדיוס של המעגל שסביבו מסתובבת המכונית בזווית ההיגוי בכל מצב נתון. ρ תמיד יהיה מאונך לכיוון המכונית, כאשר את האורך שלו נוכל לקבוע בעזרת המרחק בין נקודת היחוס של הרכב, ונקודת המפגש שבין האנך לכיוון המכונית (מסומן כאדום באיור 9) והאנך לכיוון סיבוב המכונית (מסומן ככחול באיור 9). מכונית זו תוכל לזוז קדימה בלבד, ולמודל מכונית זו, קוראים Dubins Car. מודל זה הוא קינמטי (שבו אנו חוקרים את תנועת המכונית עצמה ללא התייחסות לכוחות הפועלים עליה ושהיא מפעילה על הסביבה), ואילוציו אינם הולונומיים (במקרה זה, לא ניתן לבטא מצב של המערכת בפחות מ-3 משתנים). הדרך או השביל של דובינס (Dubins Path) היא הוכחה מתמטית המתארת ומאפיינת את הדרך הקצרה ביותר של המכונית המתוארת בין שתי נקודות נתונות (התחלה וסיום).



איור 9 – מודל בסיסי של המכונית של דובינס.

נחזור לאיור 9, ונסמן את זווית ההיגוי המקסימלית בתור ϕ_{max}. נוכל להגיד שככל ש ϕ גדולה יותר, כך ρ קטנה יותר, ולכן במצב שבו $\phi = \phi_{max}$, נוכל להגיד ש $\rho = \rho_{min}$. נתאר לעצמנו מקרה שבו $\phi_{max} = 90°$. במצב זה, $\rho_{min} = 0$, והמכונית תוכל להסתובב סביב נקודת היחוס שלה. מסיבה זו הדרך המהירה ביותר בין נקודת התחלה וסיום נתונות תהיה פשוט קו ישר בניהם, מכיוון שהמכונית פשוט תוכל להסתובב במקומה עד שתפנה בדיוק לכיוון נקודת הסיום, תיסע אליה בקו ישר, וכאשר תגיע אליה תסתובב לכיוון הרצוי באותה הדרך (איור 10).



איור 10 – מצב שבו זווית ההיגוי המקסימלית היא 90 מעלות, והדרך הקצרה ביותר בין שני מצבי מכונית היא קו ישר ביניהם.

לסטר דובינס הצליח להוכיח במאמרו [8] כי בסביבה ללא מכשולים, הדרך בין שתי מצבים נתונים של המכונית תהיה מורכבת לכל היותר משלושה תנועות קבועות:

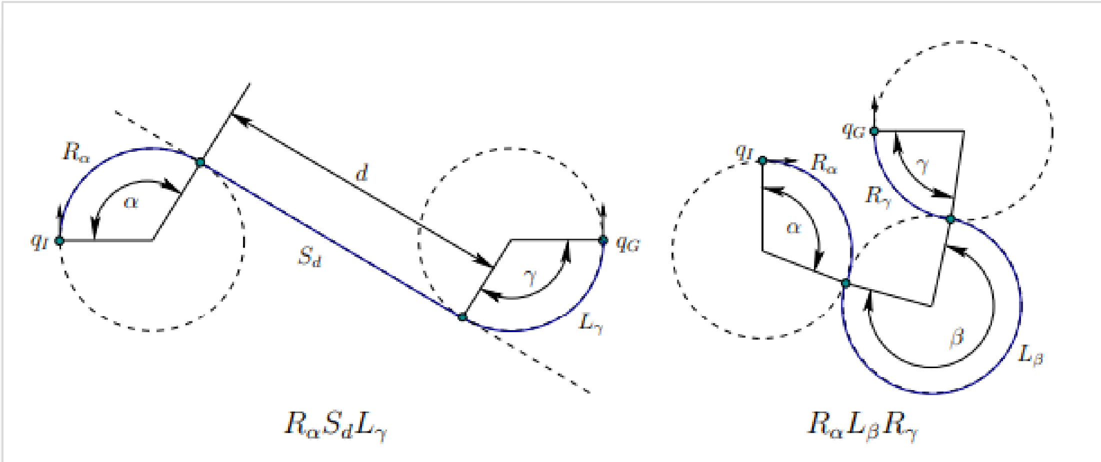
- ***S*** מהמילה Straight. יסמל תנועה של המכונית כאשר $\phi = 0$ (קו ישר).
- ***L*** מהמילה Left. יסמל תנועה של המכונית כאשר $\phi = \phi_{max}$ לצד שמאל. התנועה הזו תייצג את הפנייה החדה ביותר שרכב יוכל לקחת שמאלה, כלומר כאשר ההגה מסובב שמאלה עד שלא ניתן יותר.
- ***R*** מהמילה Right. יסמל תנועה של המכונית כאשר $\phi = \phi_{max}$ לצד ימין. תנועה זו תייצג את התנועה ההפוכה ל"L", כאשר ההגה מסובב עד לקצה הימני שלו.

הסיבוב של המכונית תמיד יתרחש בזווית ההיגוי ϕ_{max}, וזאת מכיוון שבזווית זו כאמור רדיוס המעגל של מסלול הסיבוב יהיה הקטן ביותר, והמרחק הכולל של הסיבוב עד למעבר לנסיעה בקו ישר או ההגעה לנקודת הסיום יהיה הקצר ביותר. כאמור, דובינס הוכיח כי הדרך הקצרה ביותר בין שני מצבי מכונית תהיה מורכבת משלושה מצבים כאלו. באיור 10 לדוגמא, השתמשנו ברצף המצבים *RSR*, וזאת מכיוון שתחילה המכונית הסתובבה ימינה, לאחר מכן נסעה ישר, ולבסוף הסתובבה ימינה פעם נוספת. נוכל לדמיין מצבים נוספים כאלו, אך דובינס הוכיח כי ששת המצבים היחידים שבעזרתם נוכל להרכיב את הדרך האופטימלית הם:

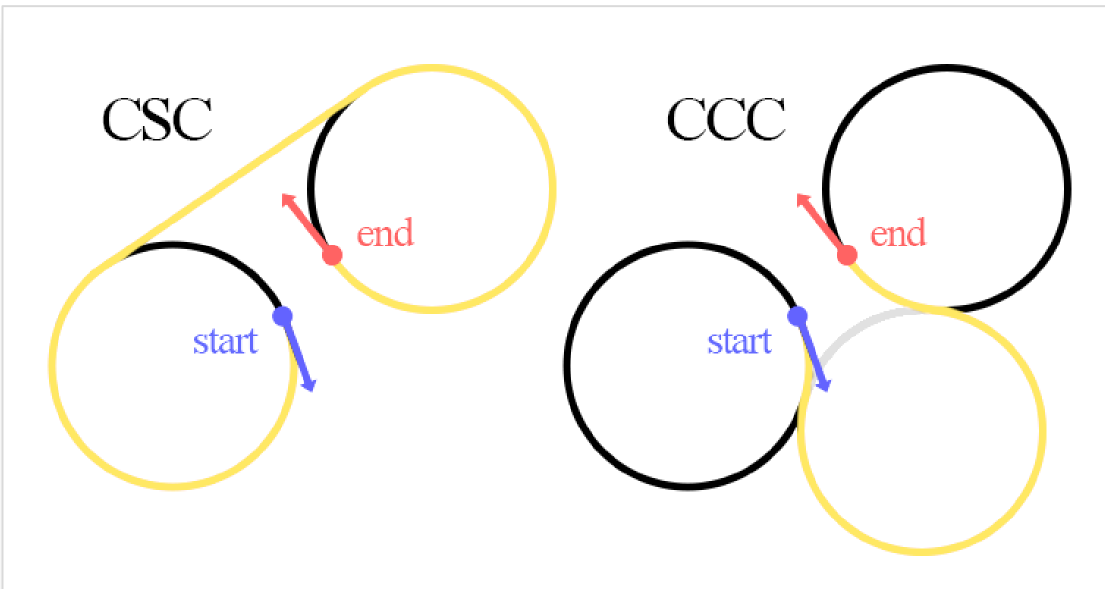
LRL, RLR, LSL, RSR, RSL, LSR

כדי להפוך את ששת המצבים האלו לפשוטים יותר, נוכל להגדיר סימן חדש **C** שייצג תנועה מעוקלת (כלומר יחליף את הסימנים **L** ו-**R**). בעזרת **C**, נוכל להציג את ששת המצבים בעזרת שני מצבים חדשים בלבד: **CCC**, **CSC**.

כעת נתמקד במצב **CCC**, שבו כל התנועה מורכבת מעיקולים. חשוב לציין שהעיקול האמצעי בתנועה זו, יהיה תמיד העיקול בסיבוב השונה משני העיקולים האחרים. בנוסף, עיקול זה יהיה גדול מ $180°$ ואם הוא קטן יותר, סימן שישנה דרך אחרת יותר קצרה בין שני המצבים (ניתן לראות דוגמא באיור 11). איור 12 ממחיש בצורה טובה למה ובאילו מצבים אנחנו צריכים את רצף התנועות **CCC**, כאשר במצבים כמו באיור 12, הדרך שתיווצר בעזרת **CSC** תהיה ארוכה מאוד ותעשה כמעט שני סיבובים מלאים. [1, 7]



איור 11 – דוגמא לשניים מתוך ששת המצבים של הדרך הקצרה ביותר בין שני מצבי מכונית שונים. [2]



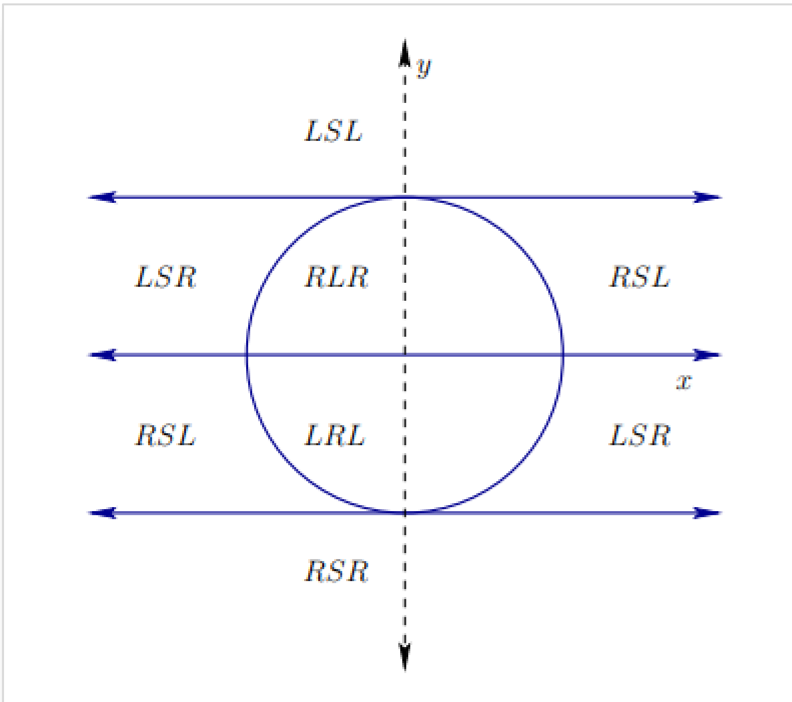
איור 12 – דוגמא של CCC ו-CSC על אותו זוג מצבי מכונית התחלתי וסופי. ניתן לראות שהמסלול הנוצר עם CCC קצר באופן משמעותי מהמסלול הנוצר בעזרת CSC.

אך גם עם כל המידע הזה, עדיין נשארה שאלה אחת באוויר: בהינתן שני מצבי מכונית, איך נדע באיזה מצב של הדרך יהיה האופטימלי ביותר, ומה היו זוויות העיקולים ואורך הדרך הישרה (אם קיימת). ובכן, למציאת הדרך ישנן מספר דרכים.

הדרך הראשונה ו"הפרימיטיבית" יותר היא יצירת כל ששת הדרכים לפי שני מצבי המכונית הנתונים, חישוב המרחק האוקלידי של כל דרך, ובחירה בדרך הקצרה ביותר. היצירה של כל דרך היא פשוטה יחסית: תחילה ניצור את שני מעגלי הסיבוב של המכונית בנקודת ההתחלה והסיום, ולאחר מכן ננסה לחבר את שניהם בעזרת

קו ישר (חשוב לשים לב לכיוון התנועה, ולא לחבר את שני המעגלים בכיוונים מנוגדים). ישנה אפשרות שחלק מהמסלולים יפסלו כבר בשלב הזה, וזאת מכיוון ששני המעגלים יחצו אחד את השני.

אפשרות נוספת לחישוב הדרך האופטימלית היא בעזרת אפיון של כל האזורים שלהם כל רצף של תנועות הוא הקצר ביותר. אם ניקח לדוגמא את נקודת ההתחלה, ונגדיר אותה כ-(0, 0, 0), נוכל לייצר מפת קונפיגורציה שבה עבור כל נקודת סיום שקיימת, נוכל לדעת מהו סוג המסלול האופטימלי בלי לבדוק את כל האפשרויות. איור 13 מציג דוגמא למפה מהסוג הזה.



איור 13 – דוגמא למפה שבעזרתה ניתן לקבל את רצף התנועות האופטימלי לפי (x, y) של נקודת הסיום. במפה זו, נקודת ההתחלה היא (0, 0, 0). [2]

2.5 הסתברות הבטא

הסתברות הוא תחום מתמטי שעיקרו הוא תרגום הסיכויים של אירוע מסוים להתקיים, לביטויים מתמטיים שניתן לחשבם. בדרך כלל, נתאר הסתברות של אירוע כלשהוא במספר בין 0 ל-1, כאשר 0 משמעותו שאין סיכוי שהאירוע התקיים כלל, ו־1 משמעותו שהאירוע יתקיים בוודאות. כך לדוגמא, הסיכוי שבהטלת מטבע יחידה הצד "עץ" ייפול כלפי מעלה הוא בדיוק 0.5 (בתנאי אידיאליים ובהנחה שהמטבע הוגן). בעזרת ההסתברות ניתן לחזות אירועים שקרוב לוודאי יתרחשו בעתיד, ואין ספק שההסתברות הוא ענף חשוב במתמטיקה.

נתבונן בדוגמא הבאה: נניח שיש לנו מטבע, אך אנו לא יודעים עד כמה הוא הוגן (אם בכלל). למטבע כמובן, שני צדדים בלבד – "עץ" ו"פלי". בהטלת המטבע הראשונה, אנו לא יודעים כלום על המטבע, ולכן אין לנו ברירה אלא להניח שהסתברות של שני צדי המטבע לנחות כלפי מעלה היא שווה. נמשיך להטיל את המטבע, ולאחר

מאה הטלות בדיוק נעצור. אם קיבלנו לדוגמא 30 הטלות בלבד שבהן צד ה־"עץ" נחת כלפי מעלה ו-70 הטלות שבהן ה־"פלי" ניצח, כנראה שנתחיל לחשוד שהמטבע לא כל כך הוגן כמו שחשבנו בהתחלה... אך בכל זאת, גם במטבע הוגן לגמרי, ההסתברויות השוות של שני צדי המטבע לא "מכריחה" את המטבע לנחות בכל פעם בצד אחר, ובהחלט קיימת האפשרות שגם בעבור מטבע הוגן לחלוטין לאחר 100 הטלות נקבל תוצאה זו.

כיצד נדע מתי המטבע הוא הוגן ומתי לא? ובכן, התשובה לשאלה זו, באופן כללי, היא אף פעם. כאמור, גם אם המטבע הוגן, יתכן מצב שבו 1,000 הטלות רצופות יראו את ה"עץ" בלבד. עם זאת, בעזרת ההסתברות, נוכל להביא ביטויים מתמטיים וממש לחשב את הסיכויים של המטבע להתנהג בדרך כזאת או אחרת.

2.5.1 הנוסחה

נסתכל על הנוסחה להסתברות הבטא:

$$beta(\theta|a,b) = \frac{\theta^{(a-1)} * (1-\theta)^{(b-1)}}{B(a,b)}$$

כאשר הביטוי $B(a,b)$ הוא קבוע ה"מנרמל" את הפונקציה ומקפיד על כך שהשטח מתחת לפונקציית הצפיפות לעיל שווה לאחת (כמו בכל פונקציות הצפיפות של הסתברויות שונות). כלומר:

$$B(a,b) = \int_0^1 \theta^{(a-1)} * (1-\theta)^{(b-1)}\, d\theta$$

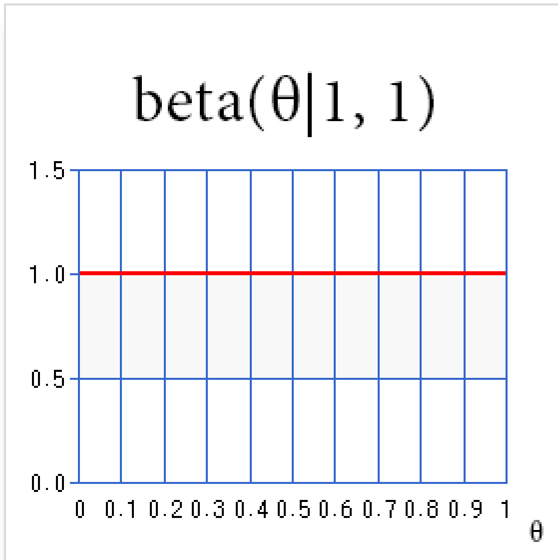
מעתה והלאה נקרא ל־$beta(\theta|a,b)$ בשם "הסתברות הבטא" ול־$B(a,b)$ בשם "פונקציית הבטא". נשים לב כי פונקציית הבטא לא תלויה ב־θ מכיוון שהוא "נזרק החוצה" בפעולת האינטגרציה. בנוסף, נשים לב כי הפונקציה $beta(\theta|a,b)$ מקבלת שלושה "משתנים". האמת היא ש־a ו־b הם קבועים שנגדיר מראש, ולכן הסתברות הבטא היא בעצם פונקציה של θ [10].

הסתברות הבטא מוגדרת בעבור $\theta \in [0,1]$ בלבד, ובעבור כל $a,b \in \mathbb{Z}^+$.

2.5.2 אז מה אנחנו בעצם רואים כאן?

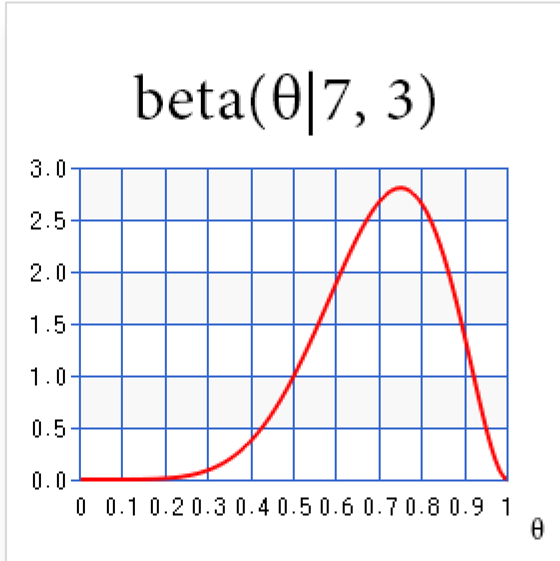
נחזור לדוגמת המטבעות מקודם לכן. בעזרת הסתברות הבטא, נוכל לגלות מידע על "הוגנות" המטבע, מבלי לקבל אותו מראש, אך ורק בעזרת הניסויים וההטלות שעשינו. נחשוב על a ועל b בתור מידע שכבר יש לנו על האירוע, ובדוגמא שלנו, נסמן את a בתור מספר הפעמים שצפינו במטבע מוטל כאשר "עץ" כלפי מעלה, ונסמן בתור b את מספר הפעמים שבו "פלי" הוא הצד שנחת כלפי מעלה. נחשוב על θ בתור "הוגנות" המטבע, כאשר θ מתקרב ל־1 ההסתברות ל־"עץ" (סומן כ־a) גבוהה יותר, וכאשר הוא מתקרב ל־0 ההסתברות ל־"פלי" (סומן כ־b) גדלה. הסתברות הבטא, תוכל לספר לנו בעצם מה הסיכוי שהגינות המטבע היא כזאת או אחרת, על סמך ההטלות הקודמות של המטבע.

בתור התחלה, כדי להגיד שאנו לא יודעים כלום על המטבע, נוכל להגיד כי "הטלנו אותו פעמיים, ופעם אחת יצא עץ ובפעם השנייה יצא פלי", ולכן נקבע $a = b = 1$. כעת נסתכל על $beta(\theta|1,1)$ ונראה כי גרף ההסתברות קבוע – כלומר ההסברות ש־θ יהיה כל אחד מבין הערכים שבין 0 ל־1 שווה [10]. כאמור, הסתברות הבטא מראה בעצם את ההוגנות של המטבע, ומכיוון שהגרף קבוע, לא ניתן להגיע על המטבע כלום בשלב זה, מכיוון שהסיכוי שהמטבע יקבל את כל אחד מערכי ה"הוגנות" האפשריים, זהה (איור 14).



איור 14 – גרף הסתברות הבטא כאשר $a = b = 1$.

נמשיך להטיל את המטבע, וכמו בדוגמא הקודמת, נעצור לאחר 10 הטלות, כאשר סה"כ הטלנו 7 פעמים "עץ" ו־3 פעמים "פלי". נסמן $a = 7, b = 3$ בהתאם, ונסתכל על $beta(\theta|7,3)$. כעת, אם נסתכל על גרף ההסתברות, נוכל לקבל קצת יותר מידע (איור 15): כאשר θ נמצא בסיבות 0.7-0.8, אנו בהחלט רואים שערך ההסתברות היא הגבוהה ביותר. משמעות הדבר היא שהסיכוי ש"הוגנות" המטבע היא בסביבות 0.7-0.8 היא הגדולה ביותר. נשים לב כי אין הדבר מעיד דבר על הוגנות המטבע עצמו, ובשום שלב לא נוכל להעיד בוודאות של מאת האחוזים על הוגנות המטבע. אנו מדברים על הסיכוי ש"הוגנות" המטבע היא כזאת או אחרת, אך לא על ההוגנות עצמה.

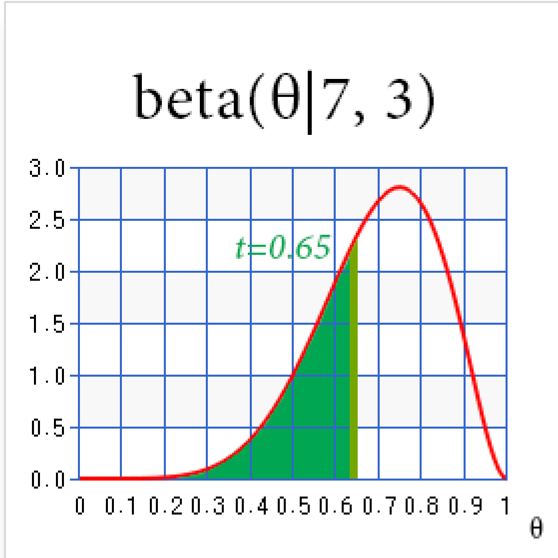


איור 15 – הסתברות הבטא כאשר $a = 7, b = 3$.

כעת, נרצה לבדוק דבר נוסף. נשאל את השאלה הבאה - לאחר 10 ההטלות שתיארנו מעלה, מהו הסיכוי שהמטבע לא הוגן לטובת פלי? ובכן, על השאלה הזו, נוכל לענות במדויק בעזרת חישוב המסתמך על הסתברות הבטא. נשים לב כי כדי שהמטבע יהיה "לא הוגן לטובת העץ" נרצה שערך ה"הוגנות" שלו יהיה גדול מ־0.5, ואם נרצה שהמטבע יהיה "לא הוגן לטובת פלי" נרצה שערך ה"הוגנות" שלו יהיה קטן מ־0.5. את ההסתברות של המטבע לכל "ערכי ההוגנות" נוכל לקבל מהסתברות הבטא (איור 15). נסתכל על הפונקציה הבאה:

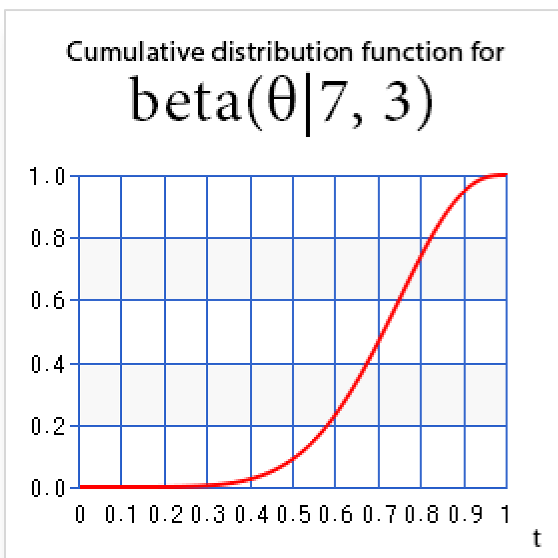
$$\int_0^t beta(\theta|a,b)\,d\theta, \qquad t \in [0,1]$$

נקרא לפונקציה לעיל "פונקציית התפלגות" או "פונקציית התפלגות מצטברת" של הסתברות הבטא [10]. נשים לב שהמשתנה היחיד של פונקציה זו הוא t, מכיוון ש־θ נזרק החוצה לאחר האינטגרציה. בעצם, קיבלנו פונקציה המתארת את השטח הנמצא בין גרף ההסתברות וציר x עד מ־$x = 0$ ועד $x = t$. אם ניקח לדוגמא את $t = 0.65$, ערך פונקציית ההתפלגות המצטברת בעבור $t = 0.65$ יהיה השטח שבין גרף ההסתברות עד $x = 0.65$ (איור 16).



איור 16 – גרף הסתברות הבטא כאשר $a = 7$, $b = 3$, עם הדגשה רפרזנטציה וויזואלית של פונקציית ההתפלגות המצטברת.

כעת נסתכל על גרף פונקציית ההתפלגות המצטברת ביחס ל-t (איור 17). נשים לב שכאשר $t = 1$ גם ערך הפונקציה שווה לאחד, וזאת בגלל שבהגדרת הסתברות הבטא חילקנו את כולה בפונקציית הבטא $B(a, b)$, שהייתה גם היא אינטגרל.



איור 17 – גרף פונקציית ההתפלגות המצטברת בעבור $a = 7, b = 3$ בהסתברות הבטא.

נחזור לשאלה: לאחר 10 ההטלות שתיארנו מעלה, מהו הסיכוי שהמטבע לא הוגן לטובת פלי? ובכן, נשתמש בשיטה הדומה למה שעשינו באיור 16, ונבחר $t = 0.5$. מאיור 17 נוכל להסיק כי כאשר $t = 0.5$ פונקציית ההתפלגות המצטברת היא 0.1 לערך, ולכן נוכל להגיד כי הסיכוי לא הוגן לטובת הפלי הוא 0.1, בעוד שהסיכוי שהמטבע לא הוגן לטובת עץ הוא 0.9 ($1 - 0.1$).

2.5.3 הסתברות הבטא בכלליות

נסתכל על איור 18 המציג גרפים להסתברות הבטא בעבור a, b שונים. נשים לב שכאשר ערך a גודל (משמאל לימין), נקודת המקסימום של גרף ההסתברות זזה ימינה. לעומת זאת, כאשר ערך b גדל (מלמעלה למטה), נקודת המקסימום של גרף ההסתברות זזה שמאלה. את התופעה הזו קל להסביר עם הדוגמא שנתתי קודם עם המטבע: כאשר אנו מגדילים את a, נוכל להגיד שראינו שהמטבע נחת יותר פעמים על "עץ" ולכן הסיכוי שהמטבע הוא לא הוגן לטובת "עץ" גדול יותר (ולהפך עם הגדלת b ו"פלי"). בנוסף, נשים לב כי אם נגדיל את a ו־b יחדיו, גרף ההסתברות יראה "צר" יותר (לדוגמא, $a = b = 4$ לעומת $a = b = 2$). גם תופעה זו ניתן להסביר בעזרת דוגמת המטבעות, שכן לאחר שהטלנו 7 פעמים "עץ" ו־3 בלבד "פלי", הסיכוי שהמטבע הוא הוגן עדיין סביר (ישנו סיכוי סביר שבהטלת מטבע הוגן 10 פעם נקבל 7 פעמים "פלי"), אך אם הטלנו 7,000 פעמים "עץ" ו־3,000 פעמים "פלי" הסיכוי שהמטבע הוגן הוא קלוש.

ברור כי צורתו של גרף פונקציית ההסתברות תלוי אך ורק בפרמטרים a ו־b. נביא ביטוי המתאר את מיקומה של נקודת המקסימום של גרף ההסתברות (נסמן בתור μ) ול"רוחב" של גרף ההסתברות (נסמן בתור v):

$$v = a + b$$

$$\mu = \frac{a}{a + b - 1} = \frac{a}{v - 1}$$

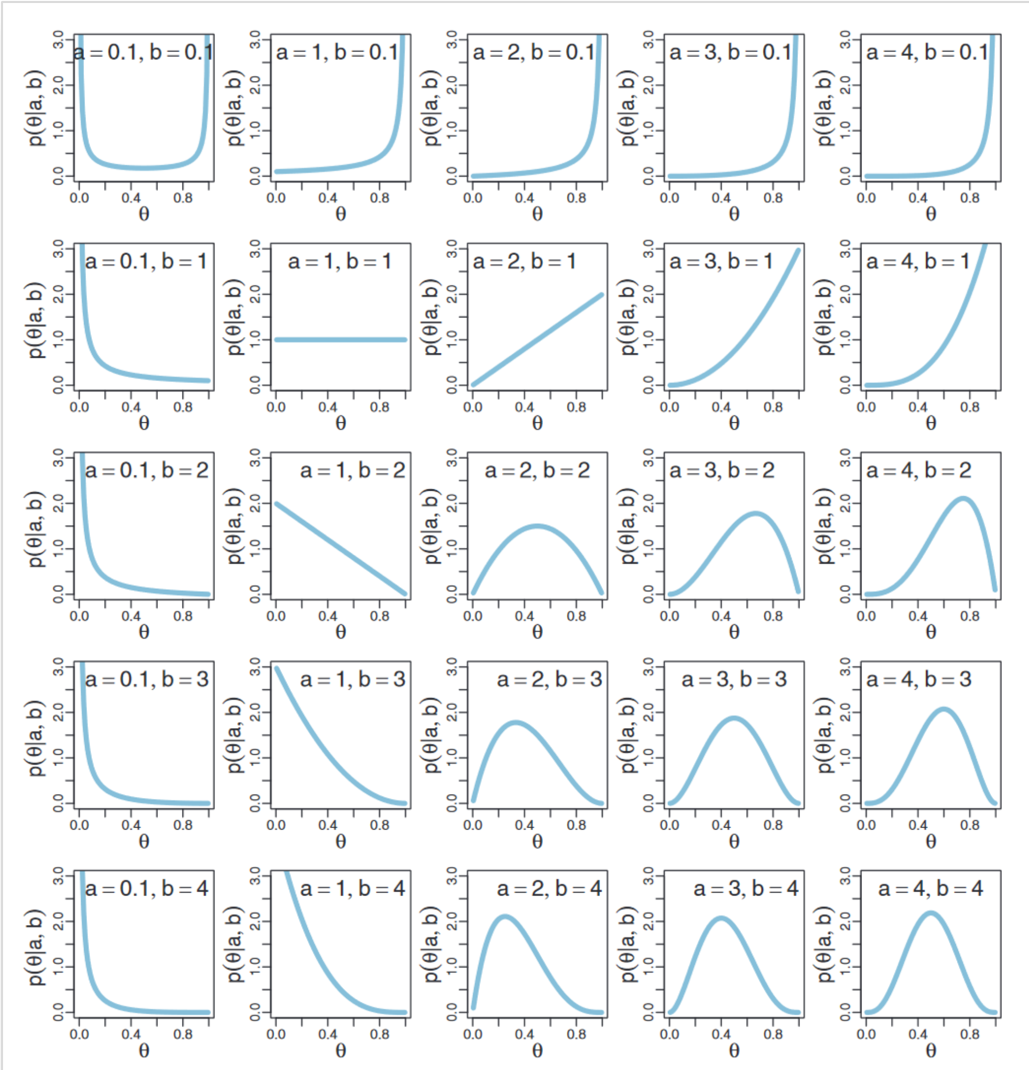
כאשר $a > 1, b > 1$. נשים לב שהביטוי μ הוא לא ה־x המדויק של נקודת המקסימום. למען האמת, ישנו ביטוי המתאר בדיוק זאת והוא מסומן בתור ω:

$$\omega = \frac{a - 1}{a + b - 2}$$

אך מכיוון ששני הביטויים מתנהגים באופן דומה, נעבוד בעיקר עם μ מכיוון שהוא יותר פשוט ונוח יותר. בעזרת אלגברה נוכל להגיד:

$$a = \mu * v, \qquad b = (1 - \mu) * v$$

שני הביטויים μ, v הם חשובים מאוד, ונשתמש בהם בהרחבה במהלך המחקר. [11]



איור 18 – גרף הסתברות הבטא בעבור ערכי a, b שונים. [10]

2.5.4 מספר הערות לגבי הסתברות הבטא

בדוגמאות שהבאתי, השתמשתי במטבע בעל שתי מקרים אפשריים בלבד. כמובן שהסתברות הבטא תקפה להרבה מאוד מקרים וניתן להשתמש בה באופן כללי הרבה יותר.

בנוסף, השתמשתי בפרמטרים a, b כאילו חייבים להיות טבעיים. למען האמת, אלו יכולים להיות כל מספר ממשי חיובי שהוא, ובפרט גם מספרים בין אפס לאחת (כמו באיור 18 לדוגמא). התנהגות ההסתברות כאשר אחד מהפרמטרים בטווח זה שונה במעט, ובמחקר זה נשתמש בפרמטרים a, b כאשר הם גדולים מ־1.

2.6 לסיכום

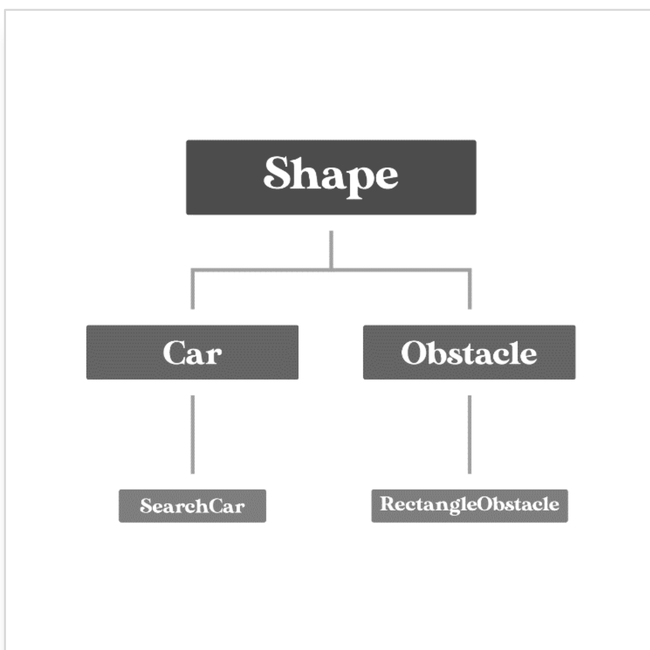
בעיית התנועה שאנו רוצים לפתור היא בעיה מסובכת, כאשר פתרון בדרך "ישירה" (לדוגמא, הזזת המכונית לכל מצב אפשרי שקיים, עד ההגעה לנקודה הסופית) לא יעיל, ויבזבז זמן ומשאבים רבים. לכן, נשתמש בשיטות שהוזכרו מעלה כדי לפשט את הבעיה לבעיות קטנות ופשוטות יותר. נייצג את בעיית התנועה בעזרת גרפים, ועל הגרפים נבצע חיפושים שונים לחיפוש מסלול בין הצומת ההתחלתית לצומת הסופית בגרף. על הגרף, נפעיל את אלגוריתמי חיפוש שונים כמו BFS, *A, ועוד. לאחר השוואה מעמיקה שנעשה בין האלגוריתמים האלו, נציע אלגוריתם חדש המתבסס על אלגוריתם ה־RRT עם שילוב של הסתברות הבטא.

3. שיטות וחומרים

לתכנות מודל המכונית, השתמשנו בשפת התכנות MATLAB. בנינו מודל מכונית בעל שלוש דרגות חופש המורכב ממיקום דו ממדי וסיבוב המכונית. במהלך התכנות, ניתן דגש על תכנות בשיטה "מונחת עצמים", כאשר התוכנה מחולקת לאובייקטים רבים, כמו אובייקט המכונית, המפה, המכשולים, ואפילו חלקי אלגוריתמים שונים. לכל אובייקט קיימים פונקציות ומשתנים משלו, והאובייקטים "מתקשרים" אחד עם השני בעזרתם.

בנוסף, בוצע ניסיון לפצל את הקוד לכמה שיותר חלקים עצמאיים – כלומר אובייקטים היכולים לעבוד בלי תלות באובייקטים אחרים. ניתן לראות כי הקוד שנכתב מפוצל לעשרות קבצים, אך כל קובץ מכיל עשרות שורות בלבד, כאשר תפקידו של כל קובץ ברור וידוע מראש. גם בתוך הקבצים, הקוד מחולק לכמה שיותר פונקציות עצמאיות, כאשר כל פונקציה מבצעת פעולה קטנה יחסית, אך בעזרת שילוב של כל הפונקציות ביחד, מתקבלת התוצאה הסופית. במחקר ניתן דגש גדול על הסדר בקוד, ובתוכנית לא ימצא קוד מסורבל או ארוך ולא ברור (מה שנהוג לעיתים לכנות כ"קוד ספגטי"). כל הקוד מתועד בעזרת Comments.

3.1 מבנה כללי של התוכנית



איור 19 – היררכיית הצורות בתוכנית.

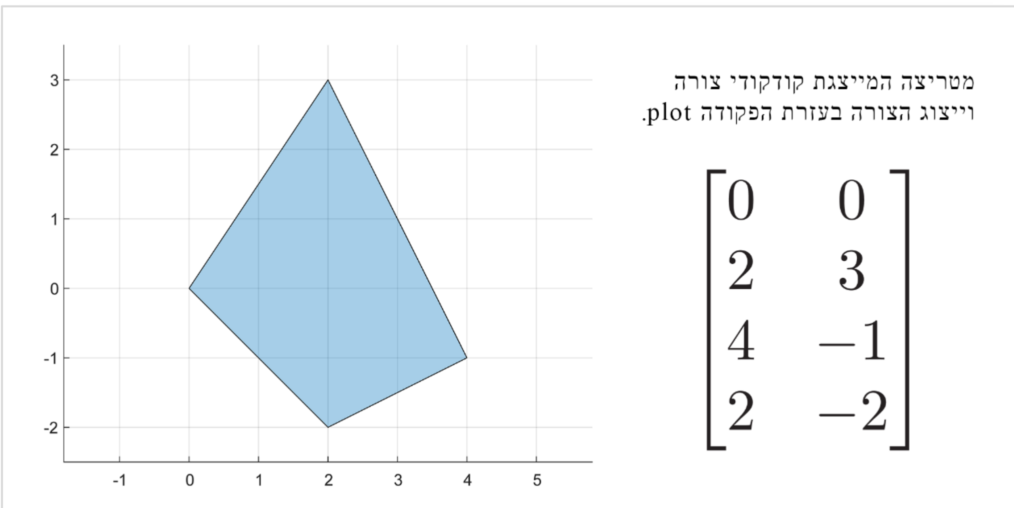
נשים לב כי באיור 19 מתואר אובייקט הצורה ("Shape") כאובייקט ה"אב" – כאשר שאר האובייקטים בתרשים הם בניו. דבר זה נקרא בשפה המקצועית "ירושה" ומשמעותו היא שכל המאפיינים והפונקציות הקיימות באובייקט הצורה, קיימות גם באובייקטים היורשים אותו. נדגיש שוב כי התוכנית כולה מורכבת מעשרות אובייקטים שונים, שחלקם משמעותיים יותר וחלקם פחות, ולכן על חלקם אדבר בהרחבה ואת חלקם לא אזכיר כלל.

בנוסף, באיור 19 מוזכר האובייקט "SearchCar" – אובייקט מכונית החיפוש. אובייקט זה מתנהג בדיוק כמו אובייקט המכונית הרגילה (שאליו אפרט בהמשך), אך בנוסף לכך מכיל פעולות שעוזרות למכונית לעבוד עם אובייקטים של אלגוריתמים שונים, ובכך לתת למחשב לשלוט במכונית, במקום שהמשתמש ישלוט בה.

גם אובייקט ה-"RectangleObstacle" הוא שיפור קל לאובייקט המכשול הרגיל, רק שמכשול זה הוא מלבן (כמו שנראה ברוב הדוגמאות הבאות). הצורה מלבן נבחרה מכיוון שהיא אחת הצורות הבסיסיות ביותר, ובעזרתה קל מאוד לבנות את הבעיה הסופית שלנו – חנייה במקביל.

3.2 אובייקט הצורה

אובייקט הצורה – Shape, הוא אחד מהאובייקטים הבסיסיים והחשובים ביותר בתוכנית. אובייקט זה משמש כ"אב" של כל אובייקט אחר שמופיע על המסך בתוכנית – המכונית, המכשולים וכו', כאשר כל מופע של האובייקט (Instance) מכיל תכונה אחת בלבד השומרת את מיקומי הקודקודים של הצורה. למעשה, אובייקט הצורה יכול לייצר כל מצולע שהוא, עם כמות קודקודים שאינה מוגבלת.



איור 20 – דוגמא לאובייקט צורה בסיסי, והייצוג שלו בתור מטריצה.

אובייקט זה מומש בצורה הפשוטה ביותר שניתן, ומכיל בסך הכל כ-20 שורות קוד. חשוב לציין כי אובייקט הצורה אינו יכול להתקיים לבדו – כלומר אי אפשר ליצור צורה שאינה מוגדרת. צורה חייבת להיות תחת קטלוג מסוים – צורה שהיא מכונית, צורה שהיא מכשול וכו'. אך צורה לבדה, אינה יכולה להתקיים.

3.2.1 הפונקציה get_shape

פונקציה זו מחזירה לקורא לה אובייקט מטיפוס polyshape (טיפוס המובנה ומוגדר כחלק מהשפה) המייצג את הצורה של המופע (Instance).

3.2.2 הפונקציה plot

פונקציה זו מציגה את המופע של הצורה על המסך. כלומר, לדוגמא, אם ישנו אובייקט מטיפוס Shape הנקרא car, כדי להציג אותו על המסך נשתמש בפקודה ()car.plot. הפונקציה הזו היא אחת מהפונקציות שנקראות מספר הפעמים הרב ביותר בתוכנית – כל תזוזה של מכונית, או כל עדכון אחר של המפה, תיקרא הפונקציה הזו למופע המתאים.

3.2.3 הפונקציה update

הפונקציה update היא פונקציה המקבלת מטריצה בעלת שתי עמודות (המייצגים x, y) של קודקודים, ושומרת את הקודקודים שמועברים כקודקודים של הצורה. פונקציה זו אולי לא נראית שימושית, כי כל צורה לא משנה את הקודקודים שלה, אבל למען האמת שזוהי אחת הפונקציות השימושיות ביותר באובייקט הצורה – וזאת מכיוון שכל אובייקט אחר שיורש את תכונות הצורה, מחליף את הפונקציה הזו (Overwriting) כפונקציה של האובייקט היורש. לדוגמא, הפונקציה update של אובייקט המכונית מבצעת חישובים על בסיס מיקום המכונית המרחב ומחשבת את מיקומי הקודקודים המדויקים של המכונית על סמך זאת.

בנוסף, הפונקציה update נקראת לפני כל הרצה של הפעולה get_shape, או של הפעולה plot.

3.3 אובייקט המכשול

אובייקט המכשול (Obstacle), כשמו כן הוא – מייצג מכשול אחד על המפה. אובייקט זה יורש את כל התכונות והפונקציות מאובייקט הצורה, Shape, שהוזכר למעלה, והוא בעצם מעיין "מקרה פרטי" של צורה, או צורה מיוחדת בעלת תכונות מיוחדות.

לשם פשטות, כל המכשולים במפה הם מלבנים, אך מכיוון שאין הגבלה על מספר המכשולים במפה ניתן ליצור מצולעים מסובכים יותר בעזרת צירוף של כמה מכשולים מלבניים אחד ליד השני.

3.3.1 הפעולה הבונה (Constructor)

למכשול פעולה אחת בלבד, והיא הפעולה הבונה. הפעולה הבונה מקבלת כפרמטר שתי זוגות של קואורדינטות – כלומר שתי נקודות המייצגות שתי פינות של המלבן. בעזרת שתי הקואורדינטות, מחושבים בפשטות כל מיקומי ארבעת הקודקודים של המלבן, ואלו נשמרים ומיוצגים כצורה.

3.4 אובייקט המכונית

אובייקט המכונית (מופיע בקוד תחת השם Car), מייצג את המכונית בסימולציה. אובייקט זה יורש את התכונות והפונקציות של הצורה, Shape. המכונית היא אובייקט יחסית מורכב יותר משאר האובייקטים במפה.

להלן תכונות המכונית:

1. xPos – המיקום של המכונית על ציר ה-x.

2. yPos – המיקום של המכונית על ציר ה-y.
3. Rotation – סיבוב המכונית, במעלות. הסיבוב מתבצע נגד כיוון השעון, כאשר כשערך התכונה הוא אפס, המכונית פונה ישירות ימינה. הסיבוב מתבצע סביב הנקודה (0,0) כאשר שתי התכונות הקודמות שוות לאפס. כלומר, הסיבוב מתבצע סביב מרכז הציר כאשר המכונית נמצאת במיקום (0,0).
4. defaultVertices – מטריצה עם שתי עמודות כאשר העמודה הראשונה מייצגת את כל ערכי ה-x של קודקודי המכונית, והעמודה השנייה מייצגת את כל ערכי ה-y של קודקודי המכונית. קודקודי המכונית כאן הם הקודקודים כאשר שלוש התכונות האחרונות, שוות ל0. תכונה זו היא קבועה ולא ניתן לשנותה במהלך ריצת התוכנית.

מכיוון שהמכונית היא אחד מהאובייקטים המרכזיים בתוכנית, הוא מכיל הרבה מאוד פונקציות שלא בהכרח משומשות באלגוריתמים השונים – כאשר יתכן שחלק מהפונקציות לא מצאו שימוש כלל בתוכנית הסופית. דבר זה נעשה במכוון: כוונתי הייתה לבנות אובייקט מכונית שיוכל לבצע מגוון פעולות שונות. לאחר שכתבתי את כל הפונקציות למכונית שחשבתי שאצטרך בעתיד, המשכתי לפתח את שאר התוכנית ומכיוון "שהבסיס" היה יציב, היה לי הרבה יותר קל לפתח את שאר התוכנית מאוחר יותר.

3.4.1 הפעולה הבונה (Constructor)

הפעולה הבונה של המכונית מקבלת את המיקום ההתחלתי והסיבוב ההתחלתי של המכונית, ושומרת את הערכים בתכונות xPos, yPos ו-Rotation.

3.4.2 הפונקציה rotate

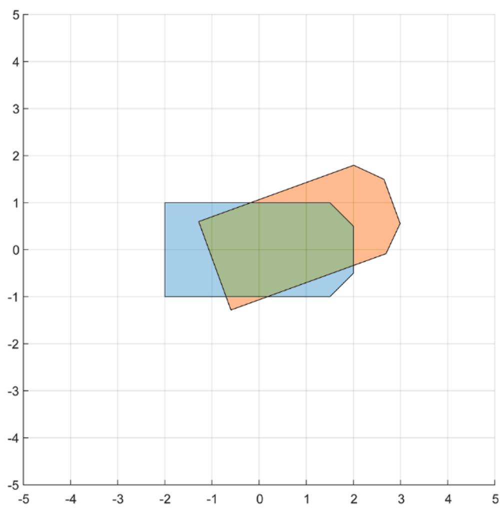
פונקציה זו מקבלת פרמטר אחד – סיבוב, ומסובבת את המכונית במקום הנוכחי שלה (כלומר, משנה את ערך התכונה Rotation בהתאם). כלומר, היא מחברת את הסיבוב הנוכחי של המכונית, והסיבוב שמעובר כפרמטר.

3.4.3 הפונקציה jump

פונקציה זו מקבלת כפרמטר את אורך הקפיצה, ובעזרת חישובים טריגונומטריים עם זווית הסיבוב של המכונית מזיזה (מסיעה) את המכונית את המרחק הנתון ישר, ביחס לסיבוב המכונית. לדוגמא, נניח שקיים אובייקט מטיפוס המכונית שנקרא *car*. אם המכונית פונה מעלה, כלומר car.Rotation = 90 ונריץ את הפקודה car.jump(5), המכונית תזוז 5 יחידות מעלה.

3.4.4 הפונקציה move

פונקציה זו מקבלת שני פרמטרים, סיבוב ומרחק קפיצה, וקוראת לשתי הפעולות rotate ו־jump. המכונית תחילה מסתובבת במקומה, ולאחר מכן זזה ישר. פעולה זה לא באמת מדמה תזוזה אמיתית של מכונית, מכיוון שמכונית לא מסתובבת במקום ואז נוסעת, אלא "מסתובבת תוך כדי נסיעה". הפונקציה הזו הייתה בשימוש בשלבים ההתחלתיים של התוכנית, והוחלפה בפעולות תנועה מסובכות ומציאותיות יותר (שאתאר בהמשך).




```
1 car = Car(0, 0, 0)
2 car.plot()
3 car.move(1, 20)
4 car.plot()
```

בשורה הראשונה, אנו יוצרים אובייקט מכונית, כאשר המכונית ממוקמת ב*(0,0)* והיא פונה ישירות ימינה. בשורה השנייה, אנו משתמשים בפקודה *plot* שקיימת לכל אובייקט צורה בתוכנית, ומציגה את המכונית על המסך. בשורה השלישית אנו מזיזים את המכונית 20 מעלות נגד כיוון השעון וצעד אחד קדימה, ובשורה הרביעית אנו מציגים את המכונית על המסך שוב.

איור 21 – דוגמא של שימוש בפונקציה הבונה, הפונקציה move והפונקציה plot של אובייקט המכונית.

3.4.5 הפונקציה update

כל פונקציות התזוזה של המכונית מבצעות פעולות על תכונות המכונית שהוזכרו מעלה – מיקום המכונית וסיבובה, אך אף אחד מהן לא משנה ומזיזה את מיקום המכונית בפועל. מכיוון שאובייקט המכונית יורש את כל תכונותיו מאובייקט הצורה (Shape), אובייקט המכונית בעצם מכיל תכונה אחת נוספת, והיא רשימת הקודקודים של הצורה של המכונית. הפונקציה update מבצעת חישובים ולוקחת את רשימת הקודקודים ב־defaultVertices ובעזרת מיקום וסיבוב המכונית בשאר התכונות, ומעדכנת את הקודקודים כך שייצגו מכונית במקומה הנוכחי.

3.4.6 הפונקציה move_xy

פונקציה זו מקבלת שני פרמטרים, תזוזה בציר ה-x ותזוזה בציר ה-y, ומזיזה את המכונית לפי הנוסחה הבאה: *מיקום המכונית הנוכחי + המיקום שמועבר לפי הפרמטר*. סיבוב המכונית אינו משתנה בהרצה של פונקציה זו, ואם נריץ את הפונקציה עם הפרמטרים (0,0), המכונית תישאר במקומה.

3.4.7 הפונקציה move_8directions

פונקציה זו מקבלת כפרמטר מרחק, ומזיזה את המכונית בהתאם. אם המכונית פונה בדיוק לכיוון אחד הצירים – כלומר סיבוב המכונית הוא בדיוק 0, 90, 180, או 270, המכונית תזוז את המרחק המועבר כפרמטר לכיוון זה בלבד. אחרת, המכונית תזוז את המרחק המועבר כפרמטר בשני הצירים, והכיוון של הקפיצה יוחלט לפי סיבוב המכונית באופן הבא:

- אם זווית הסיבוב היא בין 0 ל-90, כלומר המכונית פונה צפון מזרח, המכונית תזוז את המרחק המועבר כפרמטר על ציר ה-x וציר ה-y.
- אם זווית הסיבוב היא בין 90 ל-180, כלומר המכונית פונה צפון מערב, המכונית תזוז את המרחק המועבר כפרמטר על ציר y, ומינוס המרחק המועבר כפרמטר על ציר x.
- אם זווית הסיבוב היא בין 180 ל-270, כלומר המכונית פונה דרום מערב, המכונית תזוז מינוס המרחק המועבר כפרמטר על ציר ה-x ועל ציר ה-y.
- אם זווית הסיבוב היא בין 270 ל-360 (או 0), כלומר המכונית פונה דרום מזרח, המכונית תזוז את המרחק המועבר כפרמטר על ציר x, ומינוס המרחק המועבר כפרמטר על ציר ה-y.

פונקציה זו לא משפיעה על סיבוב המכונית.

3.4.8 הפונקציה teleport

פונקציה זו מקבלת שלושה פרמטרים – מיקום על שני הצירים וסיבוב המכונית, ו"משגרת" את המכונית ישירות למיקום שמועבר בפרמטרים.

3.5 אובייקט המפה

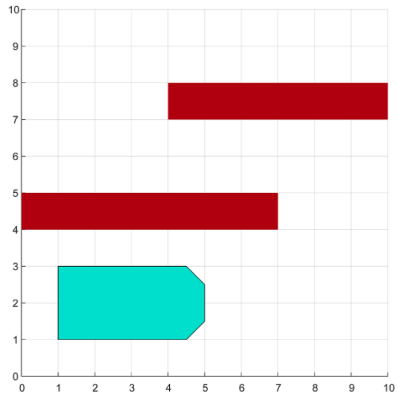
אובייקט המפה (מופיע בקוד תחת Map) הוא המאגד של כל רכיבי המכשולים והמכונית, ומכיל פונקציות היודעות לקשר את שני סוגי הרכיבים האלו ביחד. כשמו כן הוא – אובייקט זה הוא המפה, ובעצם מייצג את כל האזור שמוצג על המסך למשתמש, ומשמש אותנו במחקר.

לכל האובייקטים בתוכנית, ובמיוחד לאובייקט המפה יש גם פעולות (פונקציות) פרטיות. אלו הן פונקציות שניתן לקרוא להן אך ורק מפונקציות אחרות של אותו האובייקט, ובדרך כלל כתובות לביצוע פעולות קטנות בתוך האובייקט עצמו, שמשתמש חיצוני לא אמור לדעת עליהן. באובייקט המפה, מתבצעים חישובים רבים לבדיקת התנגשויות בין המכונית למכשולים, לקצוות המפה ועוד.

3.5.1 הפעולה הבונה (Constructor)

הפעולה הבונה של אובייקט המפה מקבלת שלושה פרמטרים – אובייקט מטיפוס מכונית (שתהיה המכונית במפה), רשימה של אובייקטים מטיפוס מכשול (שיהוו את כל המכשולים במפה), ואת גודל המפה. שלושת הפרמטרים נשמרים כתכונות באובייקט המפה. אם רשימת המכשולים לא מועברת כפרמטר, המפה תיווצר ללא מכשולים.

המפה תמיד תהיה מרובעת, כאשר הפינה השמאלית תחתונה תהיה הנקודה (0,0). גודל צלעות הריבוע יהיה הגודל המועבר כפרמטר לפעולה הבונה.




```
1 car = Car(3, 2, 0)
2 obs1 = RectangleObstacle(0, 4, 7, 5)
3 obs2 = RectangleObstacle(10, 7, 4, 8)
4 map = Map(car, 10, [obs1, obs2])
5 map.generate()
```

בשורה הראשונה, ניצור משתנה מטיפוס *Car* ונקרא לו *car*. המכונית תהיה במיקום (3,2) כאשר היא פונה בדיוק מזרחה. בשתי השורות הבאות, ניצור שני מכשולים ונשמור אותם במשתנים *obs1* ו-*obs2*. בשורה הרביעית ניצור את המפה - נקבע שהמכונית תהיה *car* שיצרנו בשורה הראשונה, גודל המפה יהיה בדיוק 10, ורשימת המכשולים תהיה המכשולים שיצרנו בשורות 2 ו-3. בשורה החמישית נייצר את המפה - שאפשר לראות בצד שמאל.

איור 22 – דוגמא של שימוש באובייקט המפה יחד עם אובייקט המכונית והמכשול.

3.5.2 הפונקציה generate

כאשר פונקציה זו נקראת, היא מציגה את מצב המפה הנוכחי בעזרת גרף (אובייקט הבנוי בשפה MATLAB), ומדפיסה את מיקום המכונית המדויק, והמכשולים סביבה. בפועל, פונקציה זו קוראת לפונקציה plot של אובייקט המכונית ששמור כתכונה במפה, ולפונקציה *plot* של כל מכשול ששמורים גם הם כרשימה כתכונה של המפה.

3.5.3 הפונקציה checkObstacleCarIntersect

פונקציה זו מחזירה ערך בוליאני (אמת או שקר). אמת יוחזר אם ורק אם המכונית נוגעת באחד מהמכשולים במפה. פעולה זו מתבצעת בעזרת הפונקציה overlaps, פונקציה שהיא חלק מהשפה MATLAB, אשר בודקת האם שני אובייקטים מטיפוס polyshape "עולים אחד על השני". מתבצעת בדיקה של המכונית מול כל מכשול ומכשול בנפרד, ומוחזר אמת אם לפחות אחד מהמכשולים נוגע במכונית.

3.5.4 הפונקציה checkIfOutOfGraph

פונקציה זו מחזירה ערך בוליאני (אמת או שקר), כאשר אמת יוחזר אם ורק אם המכונית (או חלק ממנה) נמצא מחוץ למפה. בדיקה זו מתבצעת ע"י בדיקת כל קודקוד של מכונית בנפרד, ובדיקה האם הוא נמצא מחוץ לגבולות המפה. אם חלק מהמכונית נמצא מחוץ למפה, בהכרח שגם אחד מהקודקודים נמצא מחוץ למפה – בגלל שמדובר במצולע.

3.5.5 הפונקציה checkDead

פונקציה זו משלבת את שתי הפונקציות checkIfOutOfGraph ו- checkObstacleCarIntersect, ומחזירה אמת אם אחת מהפונקציות האלו מחזירה אמת. כלומר, הפונקציה checkDead מחזירה אמת אם המכונית (או חלק ממנה) נמצא מחוץ למפה, או שהמכונית (או חלק ממנה) נמצא בתוך מכשול.

3.5.6 פונקציות נוספות

אובייקט המפה מכיל בתוכו פונקציות נוספות, שהפעולות שמבוצעות בהן ברורות מאליהן. להלן רשימת הפונקציות, עם תיאור קצר של כל אחת מהן:

- getCar – פעולה המחזירה את המכונית השמורה כתכונה באובייקט המפה.
- setSize – פעולה המקבלת כפרמטר גודל, וקובעת אותו להיות גודל המפה.
- getSize – פעולה המחזירה את גודל המפה.
- addObstacle – פעולה המקבלת כפרמטר אובייקט מטיפוס מכשול, ומוסיפה אותו למפה.

3.6 שיטת המחקר

מטרת הניסוי הייתה השוואת כל האלגוריתמים על בעיית התנועה של חנייה במקביל. ההשוואה נעשה בין שלושה אלגוריתמים בסיסיים – Breadth First Search, A* Search וה־RRT. לאחר ההשוואה ויצירת שתי גרסאות האלגוריתם המשופר, הוספנו אותם להשוואה ואימתנו את יעילותם.

3.6.1 השוואת האלגוריתמים

במהלך הניסוי ביצענו יותר מ־200 הרצות של האלגוריתמים הנתונים מעלה על אותה בעיית תנועה, אך מנקודות התחלה שונות (קודם הוגרלה נקודת ההתחלה, ולאחר מכן כל האלגוריתמים שברשימה הורצו על אותה נקודת התחלה). בכל ריצה שכזאת (על כל אלגוריתם בנפרד), נבחנו מספר פרמטרים:

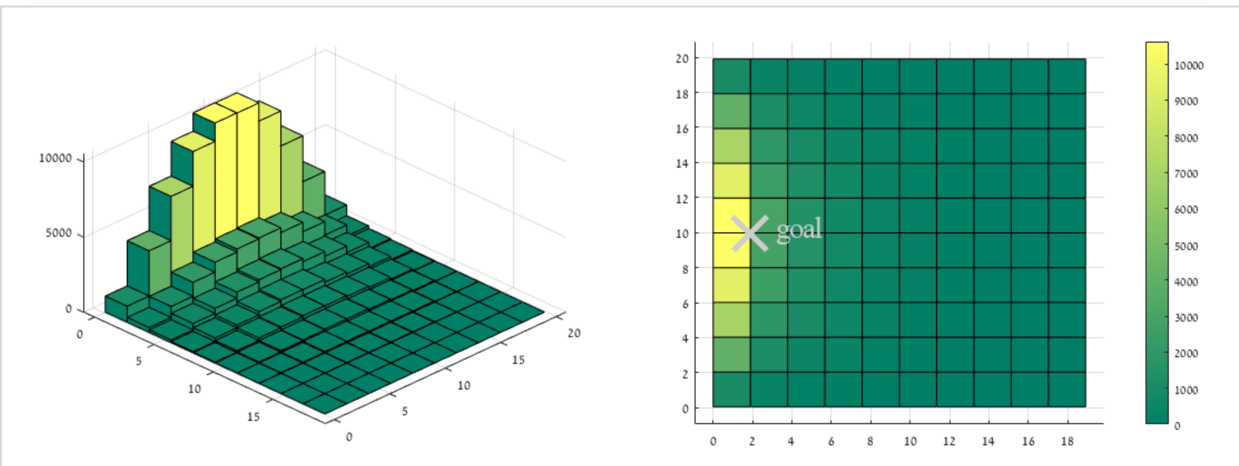
1. האם נמצא מסלול מנקודת ההתחלה לסיום?
2. מספר הצמתים במסלול מנקודת ההתחלה לסיום (אם נמצא מסלול).
3. מספר הצמתים שהאלגוריתם ביקר בהם (חקר אותם).

3.6.2 יצירת האלגוריתם המשופר

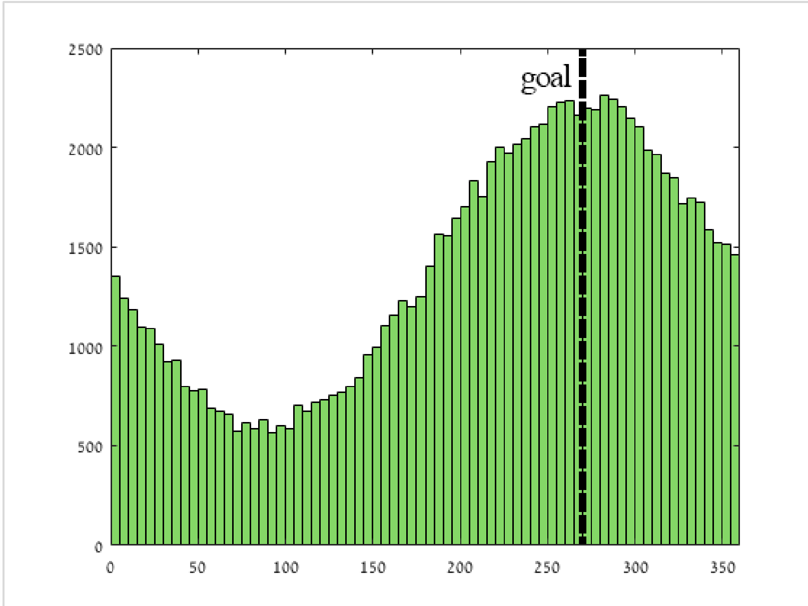
באלגוריתם המשופר, נרצה לנסות ולשמור על יתרונותיו ויעילותו של אלגוריתם ה־RRT, אך להפוך אותו למדויק יותר ו"מתפזר" פחות. כידוע, חלק גדול מאלגוריתם ה־RRT הוא הגרלת נקודות אקראיות על משטח התנועה. בכל הגרלה של נקודה כזו, המכונית תנסה להתקדם לכיוונה של נקודה זו. באלגוריתם ה־RRT נקודות אלו יכולות להיות מוגרלות בכל רחבי המשטח, וזוהי הסיבה שהמכונית תתקדם ותחקור את כל המסלול. אם

נשתמש בהסתברות הבטא להגרלת הנקודות על המשטח, נוכל לצמצם את כמות הנקודות שמוגרלות בקצוות המשטח, ולהגדיל את כמות הנקודות שמוגרלות בקרבת נקודת המטרה. אותו הדבר קורה גם לגבי סיבוב המכונית – באלגוריתם ה־RRT הרגיל, כל נקודה שמוגרלת מקבלת גם זווית סיבוב, והמכונית שואפת להגיע לזווית הסיבוב הרנדומלית הזאת. באופן דומה, נוכל לצמצם את הסיכוי להגרלת נקודות בזווית ההפוכה לזווית שבה נרצה לחנות, ולהגדיל את מספר הנקודות שמוגרלות בסביבת הזווית שנרצה שבה המכונית תחנה.

השליטה בהסתברות הבטא מתבצעת בעזרת הפרמטרים a ו־b, אך נוכל להמיר פרמטרים אלו גם ל־μ ו־υ. כך, בעזרת υ נוכל לשלוט על מיקומה של נקודת הקיצון בגרף ההסתברות, ולהחליט באיזה אזור הסיכוי להגדלת הנקודות גדול יותר, ובעזרת μ נגדיר את הפתיחות של ההסתברות – כלומר, כמה אנחנו רוצים שהנקודות יהיו קרובות לנקודת המטרה. באיור 23 ובאיור 24 נוכל לראות את תוצאות התפלגות הנקודות בעבור בדיקה שביצענו עם הגרלת 100,000 נקודות שונות בשיטה זו, בדומה לבעיית התנועה הסופית שאנו חוקרים.



איור 23 – היסטוגרמה דו-ממדית של הגרלת 100,000 נקודות סביב נקודת הסיום $x = 2, y = 10$ בעזרת הסתברות הבטא.



איור 24 – היסטוגרמה של הגרלת 100,000 נקודות סביב זווית הסיום 270° בעזרת הסתברות הבטא.

את השיטה החדשה הזו להגרלת הנקודות מימשנו באלגוריתם ה־RRT, במקום ההגרלה של נקודות אקראיות לגמרי. ניצור שני אלגוריתמים חדשים: הראשון, "אלגוריתם ה־RRT המשופר", הוא אלגוריתם RRT שמשתמש בהגרלת הנקודות בצורה החדשה בלבד. השני, "אלגוריתם ה־RRT המאוזן", הוא אלגוריתם RRT שמשלב בין שתי שיטות הגרלת הנקודות – הטריוויאלית והגרלה בעזרת הסתברות הבטא. השילוב של שני השיטות הוא פשוט – אנו מגרילים נקודה בשתי השיטות, ולוקחים את הנקודה שנמצאת בידוק ביניהן, בקו ישר (הנקודה הממוצעת). בפרק התוצאות, נקרא לאלגוריתמים החדשים גם "Improved RRT" ו־"Balanced RRT" בהתאמה.

4. תוצאות

כאמור, במהלך המחקר יצרנו סביבה ממוחשבת המדמה סביבה בעלת מכשולים ומכונית, כאשר המכונית צריכה למצוא את דרכה למקום היעד ממקומות התחלתיים שונים בעזרת שלל אלגוריתמים. במחקר, ביצענו השוואה בין אלגוריתמים אלו, וכתוצאה מהשוואה זו הצענו שני אלגוריתמים חדשים שהתבססו על אלגוריתם ה-RRT. לאחד מהם נקרא "אלגוריתם ה-RRT המשופר" (או גם Improved RRT), ולשני נקרא "אלגוריתם ה-RRT המאוזן" (או גם "Balanced RRT").

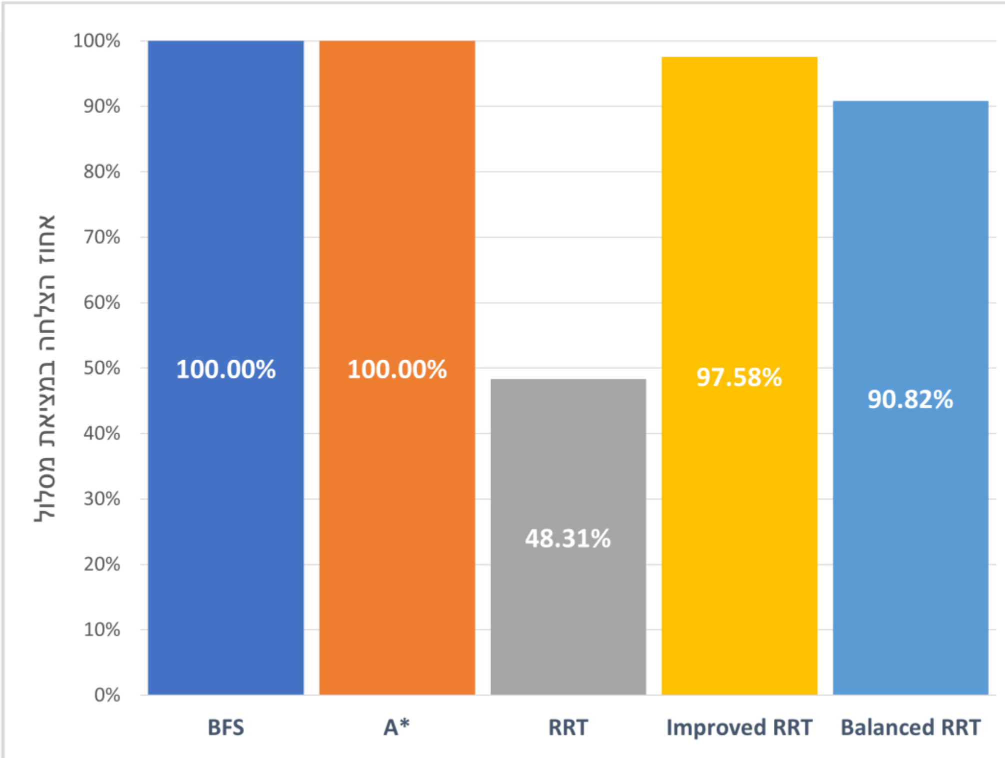
נזכיר: בהשוואה בין האלגוריתמים אספנו את שלושת הפרמטרים הבאים:

1. האם נמצא מסלול מנקודת ההתחלה לסיום?
2. מספר הצמתים במסלול מנקודת ההתחלה לסיום (אם נמצא מסלול).
3. מספר הצמתים שהאלגוריתם ביקר בהם (חקר אותם).

נסתכל על היחסים שבין הפרמטרים האלו, על הקשרים ביניהם ועל ההבדלים בקשרים וביחסים אלו בין האלגוריתמים שנבדקו.

4.1 יעילות במציאת מסלול

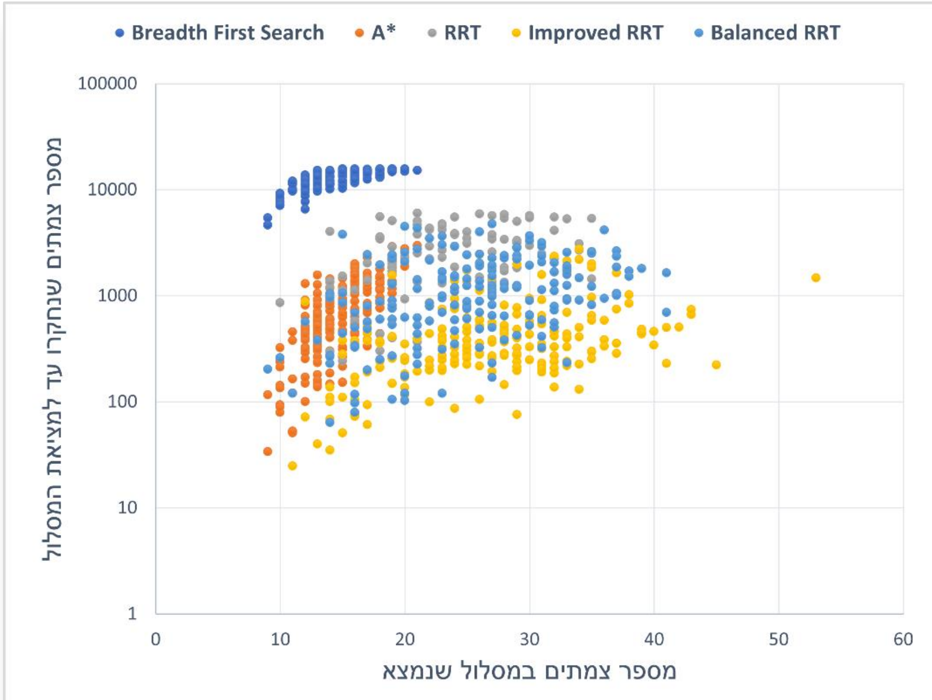
למרות שהאלגוריתמים BFS וה־*A מבטיחים כי מסלול בין נקודת ההתחלה לסיום יימצא בסופו של דבר (אם קיים כזה), אלגוריתם ה־RRT, מכיוון שמתבסס בעיקר על הגרלות אקראיות, לא מבטיח לנו כי הדרך הזו תימצא בזמן סופי. לכן, במימוש האלגוריתמים במחקר, כאשר החיפוש נמשך זמן ממושך ולא הניב תוצאות, נוכל להגיד כי החיפוש הזה פשוט לא מצא דרך בין נקודת ההתחלה לסיום. באיור 25 נוכל לראות את אחוזי מציאת המסלול בין האלגוריתמים השונים. ניתן לראות כי לשני האלגוריתמים BFS ו־*A 100 אחוזי הצלחה כצפוי, בעוד שהגרסה החזקה לאלגוריתם שהצענו מצאה דרך ב־97.58%, והגרסה המאוזנת מצאה דרך ב־90.82% מההרצות. אלגוריתם ה־RRT שעליו מבוססים האלגוריתמים שהצענו מצא דרך ב־48.31% מההרצות בלבד.



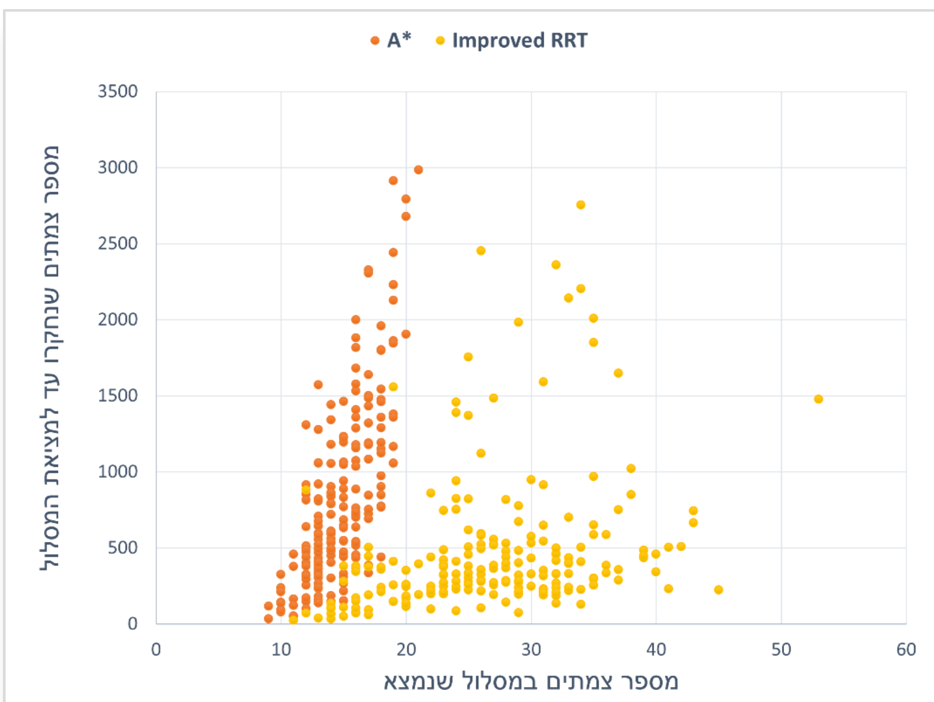
איור 25 – אחוז ההצלחה במציאת מסלול באלגוריתמים שנבדקו

4.2 מכלול ההרצות

באיור 26 ובאיור 27 נראה קיבוץ של כל ההרצות בניסוי, כאשר באיור 26 נראה מידע על כל חמשת האלגוריתמים, ובאיור 27 נתמקד על אלגוריתם ה-RRT המשופר ואלגוריתם ה-*A. כל הרצה של אלגוריתם מיוצגת על הגרפים על ידי נקודה בודדת, כאשר צבעה של הנקודה מסמל את האלגוריתם שאליו שבה נעשה שימוש בהרצה. ההרצות שלהן לא נמצא מסלול, לא מיוצגות בגרף. מאיורים אלו ניתן לקבל רעיון כללי לכלל ההרצות, מהירות מציאת המסלול, ואורך המסלול הנמצא באלגוריתמים השונים. ניתן לראות את ההתקבצות של ההרצות מסווגים לפי האלגוריתם: כך לדוגמא, אלגוריתם ה־BFS מוצא דרכים יעילות (ולכן ההרצות שלו מקובצות בצד שמאל), אך הוא לא יעיל במיוחד ולכן בגרף ההרצות עבורו מקובצות מעלה. כך גם באיור 27 – באופן כללי נוכל להגיד כי האלגוריתם המשופר יעיל יותר, אך מוצא מסלולים ארוכים יותר מה־*A.



איור 26 - מספר הצמתים שבוקרו עד למציאת המסלול ביחס למספר הצמתים במסלול שנמצא (סרגל לוגריתמי)

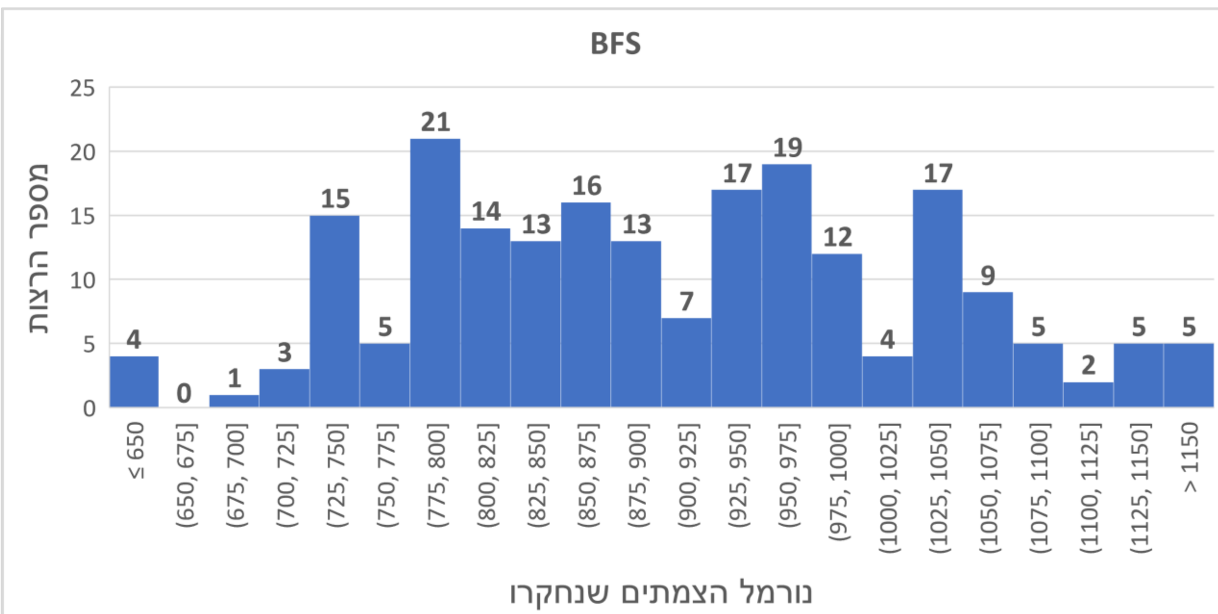


*איור 27 – מספר הצמתים שבוקרו עד למציאת המסלול ביחס למספר הצמתים במסלול שנמצא (על ידי האלגוריתמים *A והאלגוריתם המשופר בלבד)*

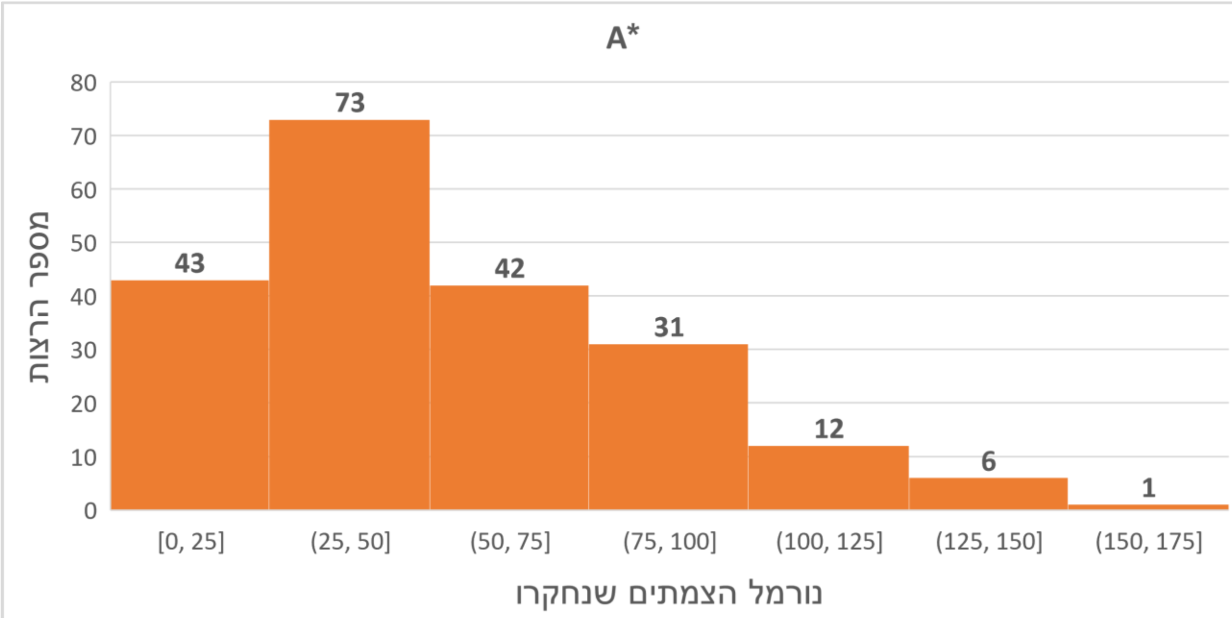
4.3 נורמל הצמתים שנחקרו ויעילות החיפוש

כעת, נגדיר מושג שנשמש בו לתיאור התוצאות הבאות: עבור כל הרצה של האלגוריתמים, תמיד יהיה מסלול שהוא המסלול הקצר ביותר בין נקודת ההתחלה לסיום, וזאת מכיוון שיצרנו את בעיית החנייה כך שמסלול מההתחלה לנקודת הסיום יהיה אפשרי בכל מקרה. האלגוריתמים A^* ו-BFS מתחייבים למצוא את המסלול היעיל (אם קיים), ולכן ניקח את מספר הצמתים שחקרנו בעבור הרצה מסוימת, נחלק אותו באורך המסלול המינימלי (הקצר ביותר), ונקבל ממוצע שאומר לנו כמה צמתים חקרנו "בתמורה" למציאת צומת אחת במסלול הסופי. לנתון זה, נקרא "**נורמל הצמתים שנחקרו**", וברור שככל שערך זה קטן יותר, כך זמן ריצת האלגוריתם מהיר יותר (אך לא בהכרח אורך המסלול עצמו). באיורים 28-32 נוכל לראות את מספר המסלולים שלהן נמצא מסלול בעבור טווחי "נורמל הצמתים שנחקרו" שונים, בעבור כל אלגוריתם שנחקר (מוצג בעזרת היסטוגרמות).

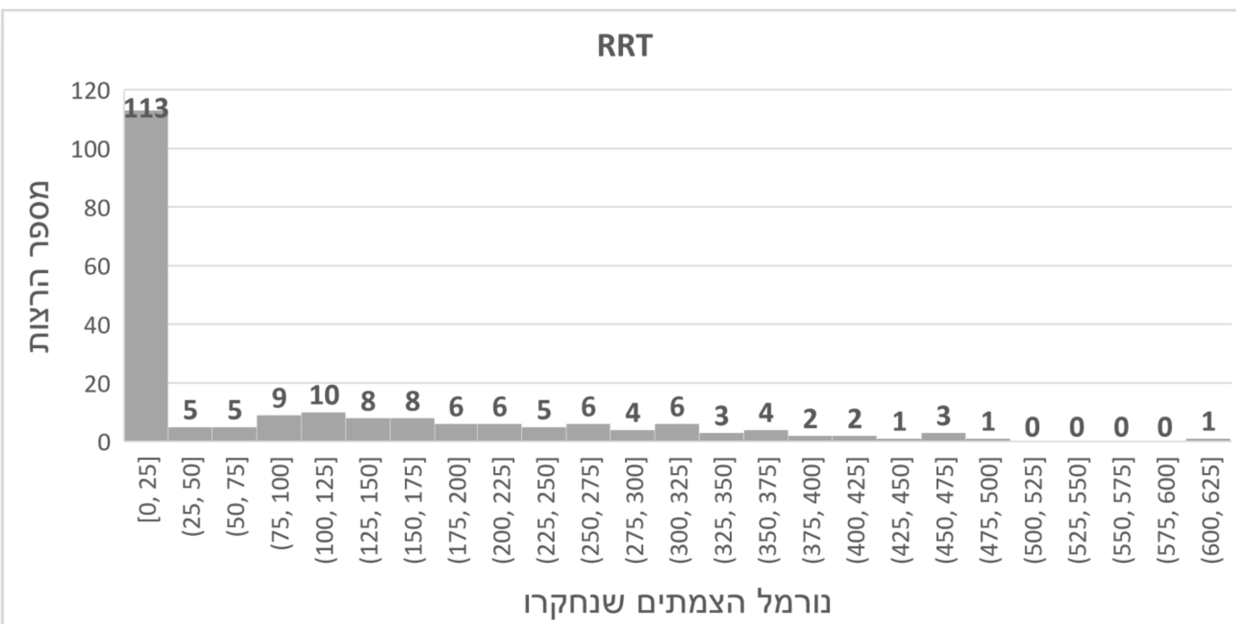
באלגוריתם ה־BFS לדוגמא (איור 28), נוכל לראות כי מספר נורמל הצמתים שנחקרו מתפזר לאורך תחום רחב של ערכים. לעומת זאת, באלגוריתם ה־A^* (איור 29), אף הרצה לא העלתה את נורמל הצמתים שנחקרו למעל 175. באלגוריתם ה־RRT (איור 30), נורמל הצמתים שנחקרו של כמחצית מההרצות נמצא בתחום $[0, 25]$, ורוב החצי השני לא מצא מסלול כלל ולכן לא מופיע עליו מידע באיור. באלגוריתם המשופר שהצענו (איור 31) נוכל לראות בדומה ל־RRT כי מחצתי מההרצות נמצאות בתחום $[0, 25]$, אך שאר ההרצות נמצאות בתחומים הצמודים לתחום זה. באלגוריתם המאוזן שהצענו (איור 32), נראה תופעה הדומה לאלגוריתם המשופר, אך כזאת שמתפרסת על גבי תחומים רחבים הרבה יותר.



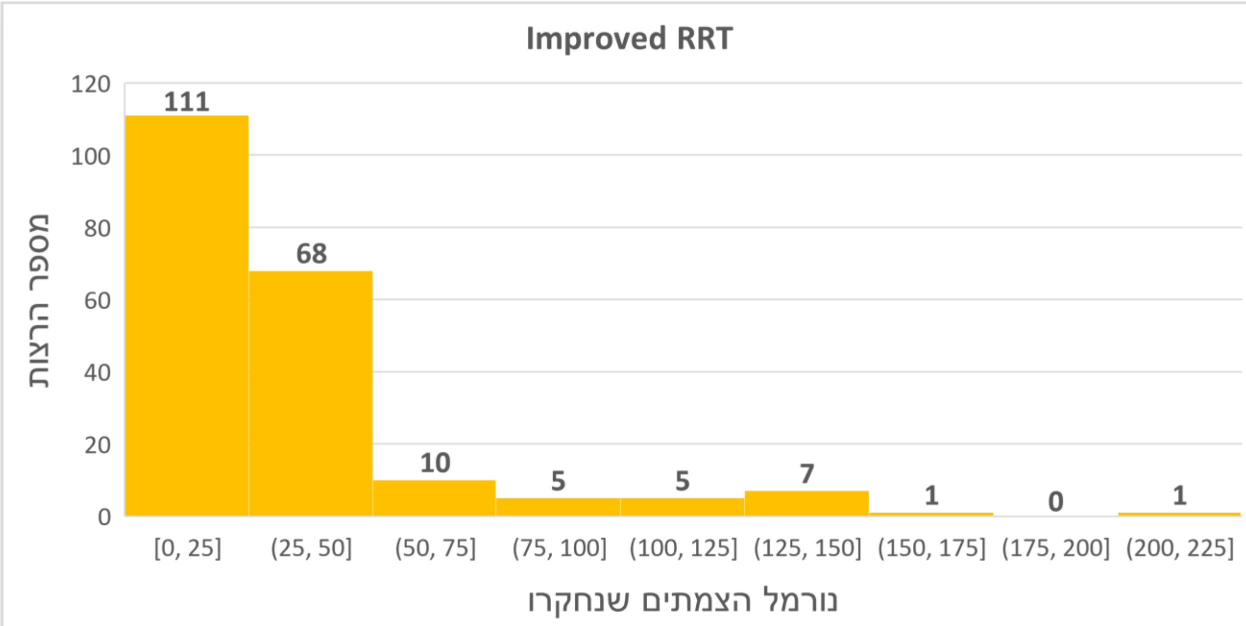
איור 28 – מספר המסלולים שלהם נמצא מסלול, מחולקים לפי נורמל הצמתים שנחקרו (היסטוגרמה) עבור אלגוריתם ה־BFS בלבד.



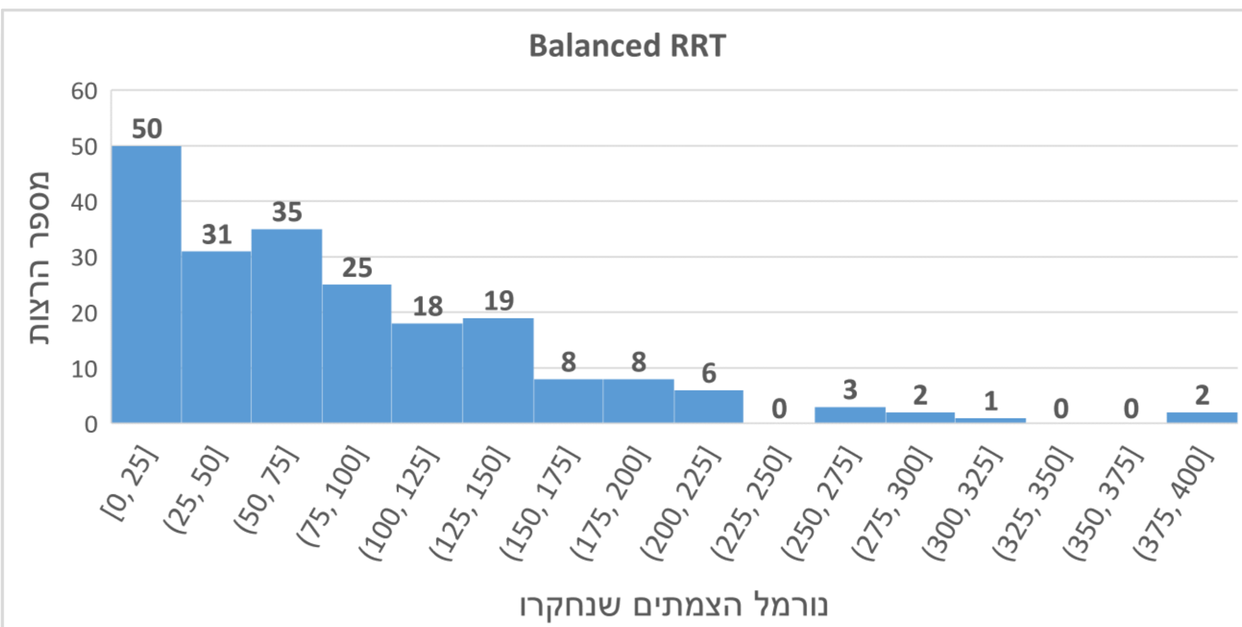
איור 29 - מספר המסלולים שלהם נמצא מסלול, מחולקים לפי נורמל הצמתים שנחקרו (היסטוגרמה) עבור אלגוריתם ה־A בלבד.*



איור 30 – מספר המסלולים שלהם נמצא מסלול, מחולקים לפי נורמל הצמתים שנחקרו (היסטוגרמה) עבור אלגוריתם ה־RRT בלבד.

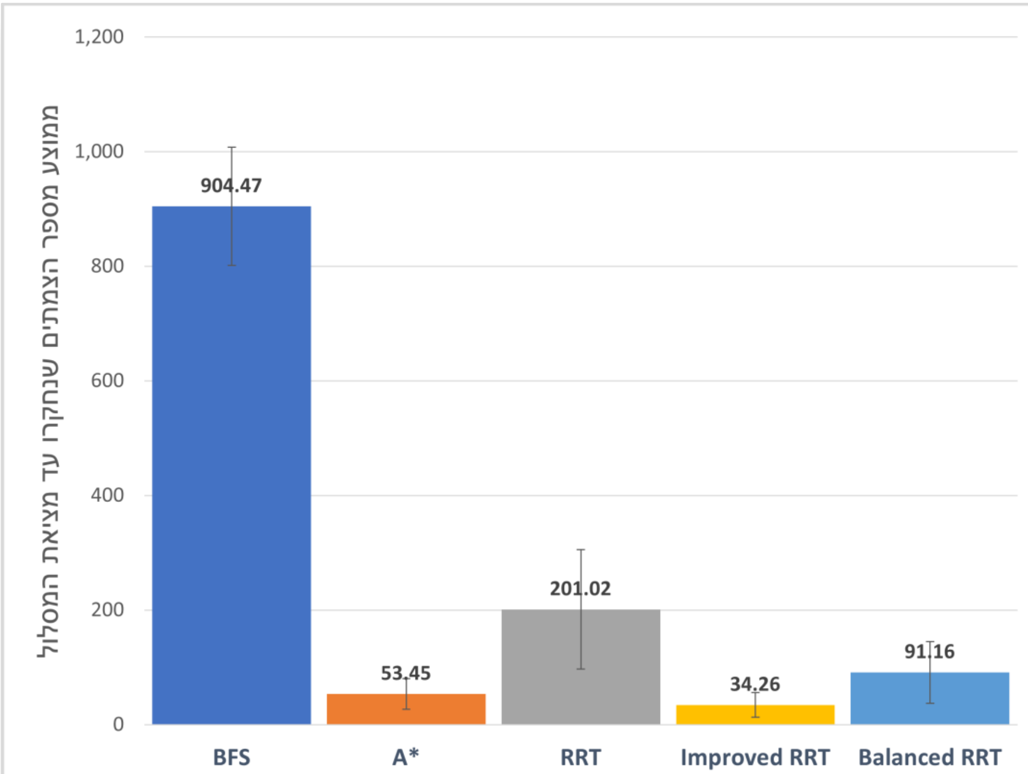


איור 31 - מספר המסלולים שלהם נמצא מסלול, מחולקים לפי נורמל הצמתים שנחקרו (היסטוגרמה), עבור אלגוריתם ה־Improved RRT בלבד.



איור 32 - מספר המסלולים שלהם נמצא מסלול, מחולקים לפי נורמל הצמתים שנחקרו (היסטוגרמה), עבור אלגוריתם ה־Balanced RRT בלבד.

באיור 33 נראה את ממוצע של נורמל הצמתים שנחקרו עבור כל ההרצות שביצענו, עבור כל אלגוריתם בנפרד. ניתן לראות כי אלגוריתם הBFS הוא האלגוריתם ה״יקר״ ביותר מכל האלגוריתמים שנבדקו, וזאת מכיוון שכדי למצוא מסלול באורך 20 לדוגמא, הוא יצטרך לעבור ולבדוק את כל המסלולים האפשריים באורך 19, 18, 17 ... ניתן לראות גם שהאלגוריתם המשופר שהצענו הוא האלגוריתם היעיל ביותר מבין האלגוריתמים שנבדקו – כאשר בממוצע, בעבור מציאת צומת אחת במסלול הסופי נחקרו בו 34.26 צמתים בלבד. נדגיש כי המסלול שנמצא על ידי אלגוריתם זה הוא לא המסלול היעיל ביותר, אלא מסלול כלשהוא מנקודת ההתחלה לסיום.



איור 33 – ממוצע נורמל הצמתים שנחקרו בכל אחד מהאלגוריתמים שנחקרו

5. דיון

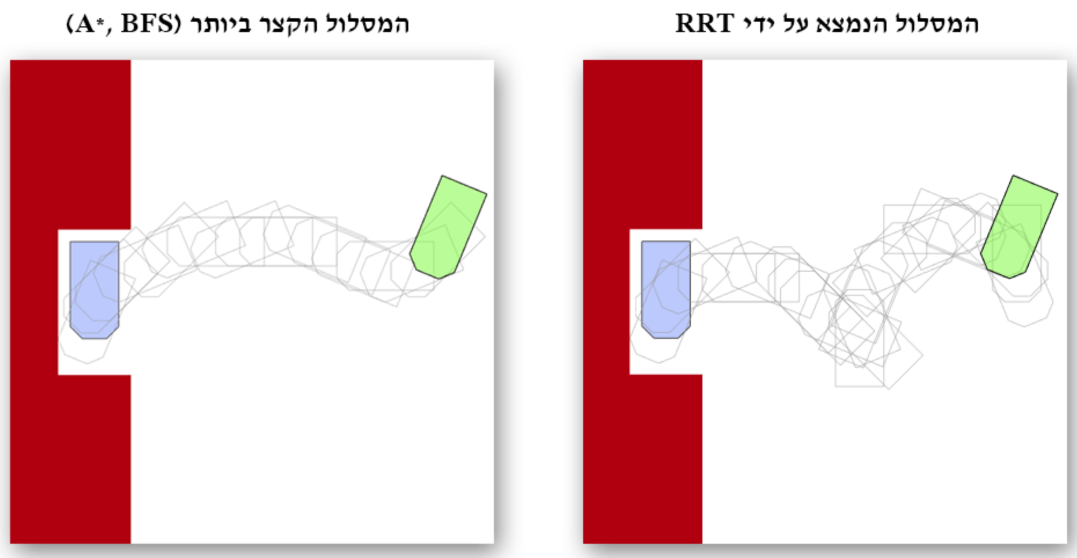
מטרתנו הייתה לחקור התנהגויות של אלגוריתמים ידועים על התרחיש החנייה במקביל שבנינו, ובסופו של דבר גם להציע אלגוריתם חדש, שיותאם לחנייה במקביל. האלגוריתם שאותו שיפרנו הוא אלגוריתם ה־RRT, כאשר בעזרתו יצרנו שני אלגוריתמים חדשים – Improved RRT ו־Balanced RRT.

5.1 דיון התוצאות

כבר בהשוואה הראשונית, ניתן היה להבחין כי לאלגוריתם ה-Breadth First Search אין חשיבות בבעיה שלנו – ואלגוריתם ה-*A יעיל מהשני (איור 33), כאשר אחוזי ההצלחה במציאת מסלול בין שניהם זהים (איור 25 – 100%). זאת מכיוון שאפילו שהבעיה שאנחנו מתארים נראית לנו, בני האדם, מסובכת, היא מכילה יחסית מספר קטן של מכשולים. מאותה הסיבה, ניתן לראות גם כי אלגוריתם ה-RRT הפשוט לא מתאים למשימה – כאשר באיור 25 ניתן לראות שיותר ממחצית מההרצות כלל לא מצאו מסלול לחניית המכונית, וגם ממוצע הצמתים שנחקרו ביחס למספר הצמתים במסלול המינימלי (איור 33) גדול באופן משמעותי מאותו הנתון באלגוריתם ה־*A. יתרונותיו של אלגוריתם ה-RRT הפשוט הן בסביבות מרובות מכשולים [7], שבדיוק בהן אלגוריתמים כמו ה־*A מתקשים.

באיור 27 ניתן לראות כי האלגוריתם המשופר שהצענו אומנם מוצא מסלולים ארוכים יותר מהמסלולים המינימליים, אך הרוב המוחלט של ההרצות שמצאו מסלול חקרו פחות מ-500 צמתים לפני מציאת המסלול הסופי. נתון זה מתבטא גם באיור 33, כאשר ממוצע מספר הצמתים שנחקרו עד למציאת המסלול ביחס לאורך המסלול הקצר ביותר קטן פי 1.5 באלגוריתם המשופר שהצענו לעומת האלגוריתם *A (34 לעומת 53).

קבוצת האיורים המעניינת ביותר בתוצאות היא איורים 28-32. איורים אלו בעצם מציגים את יעילות האלגוריתמים השונים, ומהם ניתן לראות בצורה מובהקת ופשוטה יחסית את היתרונות והחסרונות של כל אלגוריתם. נתחיל דווקא באלגוריתם ה־RRT: כשהצגנו אותו בסקירה הספרותית, הזכרנו כי היתרון הגדול שלו הוא גם החיסרון שלו. באיור 30 ניתן לראות כי מספר ההרצות שנורמל הצמתים שנחקרו בהן נמצא בטווח $[0, 25]$ הוא הגדול ביותר מכל שאר האלגוריתמים. כלומר בפועל, חלק גדול מההרצות של ה־RRT מוצאות מסלול בין נקודת ההתחלה לסיום, ואפילו ביעילות טובה יותר משאר האלגוריתמים. הבעיה ב־RRT אצלנו נובעת מה"רצון" של ה־RRT להתפשט לכל הכיוונים מהר מאוד, אך לא להתעמק בחיפוש באזורים מסוימים, ובמיוחד לא בחיפוש סביב יותר "אינטנסיבי" סביב נקודת המטרה (איור 8). ברגע שה־RRT מילא את אזור החיפוש במספיק נקודות, שאר החיפוש יהיה הרבה פחות יעיל מתחילתו, ולכן אם נקודת המטרה לא נמצאה בשלב החיפוש הראשוני של ה־RRT, רוב הסיכויים הם שנקודת הסיום לא תמצא בכלל, או תמצא אחרי זמן רב (מידי). זוהי הסיבה שבאיור 30 בטווחים שאחרי הטווח הראשון אנו רואים הרצות בודדות שמוצאות דרך, וזוהי גם הסיבה שאחוז ההצלחה במציאת הדרך באלגוריתם ה־RRT באופן כללי הוא פחות מ־50% (איור 25). בנוסף לכך, מכיוון שאלגוריתם ה־RRT חוקר את לכל הכיוונים בו־זמנית, ויכול לחקור לכיוון מנוגד לחלוטין מכיוונה של הנקודה הסופית, המסלול שיימצא על ידי האלגוריתם יכול להיות מסורבל ולא פרקטי במיוחד, כמו באיור 34.



איור 34 – השוואה בין המסלול שנמצא על ידי אלגוריתם ה־RRT והמסלול הקצר ביותר שנמצא על ידי A ו BFS. מתוך תוצאות ההשוואה (הרצה מספר 13).*

את הבעיה הזו בדיוק של ה־RRT אנו מנסים לתקן בעזרת האלגוריתמים שהצענו. כאמור, במקום להגריל את הנקודות באלגוריתם ה־RRT באופן רנדומלי לגמרי, השתמשנו בהתפלגות הבטא כדי להגריל יותר נקודות סביב נקודת המטרה (מיקום החנייה של המכונית), ובכך "לכוון" את אלגוריתם ה־RRT לחיפוש אינטנסיבי יותר סביב נקודה זו, וחיפוש אינטנסיבי פחות בכיוונים האחרים. נסתכל על איור 31: בטווח $[0, 25]$ ישנם 111 הרצות שונות, כאשר זהו הפרש של שני הרצות בלבד בין אלגוריתם זה ל־RRT. נוכל להגיד כי השינוי שעשינו באלגוריתם לא "הרס" את היתרון של אלגוריתם ה־RRT הבסיסי, וחלק גדול מההרצות עדיין מוצאות מסלול ביעילות טובה מאוד. מצד השני, שאר הצמתים, הצלחנו לשפר באופן משמעותי את שלמותו של האלגוריתם, ואחוז ההרצות שמצאו מסלול באלגוריתם המשופר עומד על 97.5% (איור 25), וזוהי כמובן תוצאה טובה בהרבה מזו של אלגוריתם ה־RRT הרגיל.

5.2 לסיכום

למרות שלא מוצא את המסלול האופטימלי ביותר, האלגוריתם המשופר שהצענו יעיל פי 1.5 במציאת דרך לחנייה. ברור כי שיפור זה יכול להיות משמעותי – מכוניות אוטונומיות צריכות לקבל החלטות בזמן אמת, ולהיות מתוכנתות במידת היעילות המקסימלית. בעזרת אלגוריתם יעיל יותר בחנייה, מחשב הרכב יוכל לפנות כוח חישוב חשוב למשימות אחרות, או לחילופין, ניתן יהיה אפשר להשתמש במחשב חזק פחות (וכתוצאה מכן, להוזיל את עלות הרכב) בעזרת האלגוריתם היעיל יותר.

5.3 לאיפה ניתן להתקדם מכאן?

למרות התוצאות המעודדות, עדיין נדרש מחקר נוסף. מכיוון שהאמצעים שלנו מוגבלים, הסימולציה שיצרנו למכונית הייתה "גסה" יחסית, והמודל היה פשטני. כדי לאשר את יעילות האלגוריתם נצטרך בדיקה על מודל מדויק ונאמן יותר. בנוסף, בחלק מהמקרים יתכן וניתן להפעיל אלגוריתם ל"החלקת" המסלול שנמצא, ובכך להפוך את המסלול שנמצא על ידי האלגוריתם, שלעיתים יכול להיות מסורבל, לחלק ופשוט יותר במאמץ יחסית קטן.

6. רשימת מקורות ספרות

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning - 1st Edition*. Morgan Kaufmann, 2004.

[2] S. M. LaValle, *Planning algorithms*. Cambridge University Press, 2006.

[3] R. J. Trudeau, *Introduction to Graph Theory*. Dover Publications, 1994.

[4] P. Giblin, M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, vol. 85, no. 502. 2001.

[5] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numer. Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959, doi: 10.1007/BF01386390.

[6] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, 1968, doi: 10.1109/TSSC.1968.300136.

[7] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," p. 4, 1998.

[8] L. E. Dubins, "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents," *Am. J. Math.*, vol. 79, no. 3, p. 497, Jul. 1957, doi: 10.2307/2372560.

[9] J. A. Reeds and L. A. Shepp, "Optimal paths for a car that goes both forwards and backwards," *Pacific J. Math.*, vol. 145, no. 2, pp. 367–393, 1990, doi: 10.2140/pjm.1990.145.367.

[10] J. Kruschke, *Doing Bayesian Data Analysis - 2nd Edition*, 2nd ed. Academic Press, 2014.

[11] C. Forbes, M. Evans, N. Hastings, and B. Peacock, *Statistical Distributions, 4th ed.* 2010.

7. נספחים

7.1 תוצאות ההרצות

הרצת המחקר (ההשוואות בין האלגוריתמים) התבצעו למשך שלושה ימים, בין ה־17 ועד ל־19 בנובמבר 2020. במהלך הרצות אלו, נאסף מידע בגודל של 100MB לערך, כשרובו קבצי וידיאו המתארים כל מסלול שנמצא על ידי כל אחד מהאלגוריתמים. כל המידע על ההרצות נאסף והוכנס לטבלאות xlsx באופן אוטומטי, וכל המידע זמין ונגיש להורדה מהכתובת הבאה:

https://github.com/RealA10N/car-pathfinding-problem/releases/tag/raw-runs-release

לשם הנוחות, ניתן לגשת לדף זה גם על ידי סקירת הברקוד המצורף בצד שמאל.

7.2 קוד מקור

את הקוד המלא שנכתב במהלך מחקר זה ניתן למצוא בדף ה־GitHub הבא:

https://github.com/RealA10N/car-pathfinding-problem

לשם הנוחות, ניתן לגשת לדף זה גם על ידי סקירת הברקוד המצורף בצד שמאל.
בנוסף, ניתן להוריד (clone) את הקוד בעזרת תוכנת git ע״י הפקודה:

git clone https://github.com/RealA10N/car-pathfinding-problem.git