Civil Guardian

אריאל שמילוביץ'
הראל גם זו לטובה
עדי אורגד

רקע ספרותי:

על רקע הסכסוך המתמשך בין ישראל לאיראן, גדל הצורך בתכנון מסלולים חכמים בזמן המלחמה. אנחנו בחרנו בכתבה:
"Fearful of Iranian missiles, many sleep in Israel's underground train stations" מאתר החדשות APNews המתאר כיצד אזרחים ישראלים, הנתונים תחת איום מתמיד מסביב לשעון של למעלה מ – 450 טילים ויותר מ – 1000 רחפנים ששוגרו מאיראן, מנהלים את חייהם וישנים בתחנות רכבת תחתיות ובחניונים תת-קרקעיים. מקומות אלו הפכו למקלטים מאולתרים עבור תושבים רבים שאין להם גישה נוחה לממ"ד / מקלט. משפחות, עובדים ותיירים הפכו את המרחבים הללו לבית זמני, כשהם מביאים עימם מזרנים, מזון וציוד אישי – הכל מתוך רצון כל הזמן להישאר בקרבת מרחב מוגן.

תופעה זו מציגה תובנה מרכזית: כאשר הציבור נמצא תחת איום, הוא צריך לא רק לדעת היכן נמצאים המקלטים בקרבתו, אלא גם לתכנן את מסלולו במהלך תנועה שגרתית בעיר בשעת חירום – הצטיידות בציוד חיוני כמו אוכל ותרופות. הגישה של ריצה למקלט בעת הישמע אזעקה אינה תמיד פרקטית או מהירה מספיק כדי להגיע אליו בזמן, במיוחד בזמן הלילה או לאנשים בעלי מוגבלות וקושי בתנועה. מציאות זו מדגישה את הצורך בפיתוח מערכת לתכנון מסלולים שלא רק מחשבת את המסלול הקצר ומהיר ביותר, אלא גם מקפידה שהמשתמש יישאר לאורך כל הדרך ממרחק סביר ובטוח מאחד המקלטים הפרוסים ברחבי העיר. ע"י שילוב של שני הפרמטרים הללו – איזון בין זמן ההליכה לבין הקרבה של הדרך למקלטים – ניתן ליצור מערכת שמשקפת טוב יותר את הצרכים האמיתיים של האזרחים בשעת חירום.

בפרויקט שלנו, ננסה לפתור את הבעיה הזאת באמצעות כלים של אופטימיזציה: בהינתן לנו מפה של עיר ורשימה של נקודות המייצגות מיקומים של מקלטים בה, ננסה למצוא את הדרך המהירה והבטוחה ביותר להגיע בין שתי נקודות בעיר.

מטרת הפרויקט שלנו היא לבנות מערכת לתכנון מסלולים שיודעת לבנות מסלול כך שכל נקודה בו תמיד תישאר במרחק מקסימלי ממקלט ציבורי מסוים (לדוגמה 500 מטר), ובנוסף יודעת לתעדף את הצרכים של הלקוח: האם הוא מעדיף דרך מהירה או דרך בטוחה?

רקע מתמטי:

1. **מרחב מטרי:** הזוג (X,d) כאשר X – קבוצה ו – $d: X \times X \to R$ יקרא מרחב מטרי אם מתקיימות התכונות הבאות:
    1. $\forall x,y \in X: d(x,y) \geq 0$
    2. $d(x,y) = 0 \leftrightarrow x = y$
    3. $d(x,y) = d(y,x)$
    4. $d(x,z) \leq d(x,y) + d(y,z)$

2. **גרף:** מבנה נתונים המסומן כ – $G = (V,E)$ המרוכב מקבוצה של קודקודים (הנקראים גם צמתים) המסומנת ב – V, וקבוצה של קשתות המחברות בניהם המסומנת ב – E.

ישנם מספר סוגים של גרפים:

- <u>גרף לא מכוון:</u> גרף שבו אין קשתות מכוונות, כלומר שאם קשת מחברת בין הצמתים u,v אז אפשר לנוע על הקשת מ – u ל – v ולהפך. כלומר, קשת מחברת בין שני צמתים ואפשר לנוע בניהם בחופשיות ללא קשר לכיוון התנועה.
- <u>גרף מכוון:</u> גרף שקיימות בו קשתות מכוונות. כלומר שבכל הגדרה של קשת, יש גם להגדיר את כיוון התנועה בין הקשתות – האם אנחנו נעים מ – u ל – v או להפך. בגרף כזה, כל קשת מוגדרת כקבוצה סדורה של שני קודקודים בגרף.
- <u>קשתות מקבילות:</u> קשתות מקבילות הן קשתות שקצותיהן זהים, אך עדיין הן קשתות שונות. גרף בעל קשתות מקבילות נקרא <u>מולטיגרף</u>. כדי להפריד בין שתי קשתות מקבילות, נצמיד לכל אחת אינדקס שונה (לדוגמה: $\{u,v,0\}$).
- <u>גרף ממושקל:</u> גרף שמוגדרת עליו פונקציית משקל $w: E \to R$ שמתאימה לכל קשת בגרף ערך מספרי הנקרא <u>משקל</u>.
- <u>גרף מלא:</u> גרף שקבוצת הקודקודים שלו מהווה קבוצה קשירה – לכל שני קודקודים בגרף, ניתן למצוא מסלול המקשר בניהם.
- <u>גרף מטרי:</u> גרף ממושקל ומלא כאשר פונקציית המשקל שלו משרה מטרי. כלומר, אם נגדיר את המטריקה $d: V \times V \to R_+$ באופן הבא:

$$d(x,y) = \min_{e=\{x,y\} \in E} w(e)$$

אז הזוג (V,d) יהווה מרחב מטרי.

ייצוג של מפה כגרף:
השימוש שלנו בגרף בבעיה היא ייצוג של המפה בעזרתו. אחת הדרכים השימושיות לייצג מפה היא בתור גרף – לכל צומת במפה נגדיר קודקוד בגרף ולכל רחוב במפה נגדיר קשת שמחברת בין שני צמתים. נוכל גם לראות שזהו לעיתים קרובות גם מולטיגרף היות וישנם רחובות שמחברים בין שני צמתים אך הם עדיין רחובות שונים.
יותר מכך, כאשר אנחנו מסתכלים על מפה ומגדירים מרחק בין שתי נקודות **כמרחק הליכה**, אז הגרף הוא גרף מטרי. במילים פשוטות, נגדיר את המטריקה להיות מרחק ההליכה המינימלי בין שני צמתים על המפה.
לכן, במהלך הבעיה ופתרונה, לא נבחין בין המושגים "רחוב" ו – "קשת" כיוון שהם מתייחסים לאותו הדבר בדיוק.
<u>סימון:</u> אורך של רחוב המיוצג ע"י קשת יסומן ב – $\|(u,v,k)\|$.

3. **בעיית אופטימיזציה:** בעיה שבה אנחנו מחפשים את ערך ה – $x \in X$ שיוביל למיקסום או למזעור של פונקציית המטרה $f(x)$, כאשר X הוא התחום הפיזיבילי – תחום המגדיר את ערכי הפתרונות האפשריים. התחום הפיזיבילי נתון ע"י האילוצים של הבעיה: שיוונות ואי-שיוונות המגדירים את תחום הערכים האפשריים של x.
בעיית אופטימיזציה סטנדרטית מוגדרת בצורה הבאה:

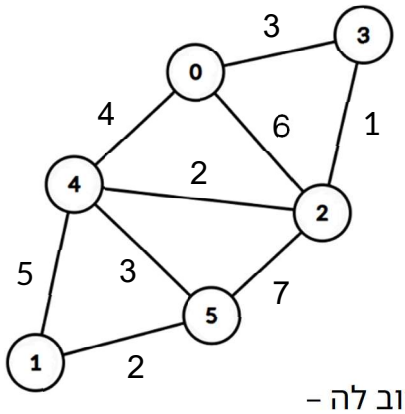
$$\min_{x \in X} f(x)$$
$$\forall x \in X: g_i(x) \leq 0 \qquad i = 0,1,2,\dots$$
$$\forall x \in X: h_i(x) = 0 \qquad i = 0,1,2,\dots$$

אם פונקציית המטרה $f(x)$ וכל האילוצים הם קומבינציה לינארית של הרכיבים של x, נאמר כי זאת **בעיית אופטימיזציה לינארית**.
בנוסף, נוכל להגביל את הרכיבים של x להיות ערכים שלמים, ואז הבעיה הזו תיקרא **בעיית תיכנון בשלמים**.

4. **אלגוריתם דייקסטרה:** אלגוריתם אופטימיזציה שפועל על גרף משוקלל בעל פונקציית משקל חיובית שמטרתו היא למצוא את הדרך הקצרה ביותר בין שני קודקודים בגרף. אורך של דרך בגרף נתון בתור סכום המשקלים של הקשתות שבהן אנחנו עוברים במסלול. האלגוריתם פותח ע"י המתמטיקאי ההולנדי Edsger W. Dijkstra ומאז נחשב לאלגוריתם היעיל ביותר לעשות זאת.
הסיבוכיות של האלגוריתם היא $O(|V|\log|V| + |E|)$.

נדגים את אופן פעולתו של האלגוריתם באמצעות דוגמה בסיסית:

דוגמה 1:



נתון הגרף הלא-מכוון המטרי הבא בעל האורכים הנתונים בשרטוט. המטרה שלנו היא למצוא את הדרך הקצרה ביותר בין הקודקודים 1 ו – 3.

מטרת הטבלה היא למצוא לכל צומת בגרף את השכן הקרוב לה – הערך של פונקציית המשקל הוא הנמוך ביותר, וכך נוכל למזער את סכום המשקלים.

בעמודת המרחק, אנחנו מכניסים בכל פעם את המרחק המינימלי בין נקודת המוצא את הנקודה הנוכחית, וההורה הוא הקודקוד הקודם שנמצא במסלול הזה. נתחיל עם המצב הבסיסי שבו המרחק של צומת ההתחלה משאר הצמתים הוא אינסופי, למעט המרחק שלו מעצמו שהוא 0.

צומת	מרחק	הורה
0	∞	
1	0	
2	∞	
3	∞	
4	∞	
5	∞	

ניצור רשימה של הצמתים בגרף: $open = [0,1,2,4,5]$ ונעבור עליהם בלולאה. בכל פעם, נחשב את המרחק של כל צומת מהשכנים שלו ונבדוק האם אנחנו יכולים לשפר את משקל המסלול המלא ביחס למסלול שכבר אצל השכן. אין לנו סיבה לעבור את אותו הצומת פעמיים כיוון שאז אנחנו מבצעים לולאה ואנחנו יודעים שהלולאה הוסיפה למשקל הכולל והאריכה את הדרך.

נתחיל עם השכנים של קודקוד 1: נמצא את המרחק בניהם לבין קודקוד 1 (די פשוט), ונראה שאנחנו יכולים להקטין את המשקל הכולל של הדרך, ולכן נציב את הערכים אל תוך הטבלה.

צומת	מרחק	הורה
0	∞	
1	0	
2	∞	
3	∞	
4	0+5	1
5	0+2	1

לאחר מכן, נסיר את 1 מרשימת הצמתים כיוון שביקרנו בו כבר.

$open = [0,2,4,5]$

לאחר מכן, תמיד נמשיך בדרך שהיא כרגע הקצרה ביותר. לכן, נבחר להמשיך לקודקוד 5. לקודקוד 5 יש שני שכנים שלא ביקרנו בהם והם 2, 4. בקודקוד 4, המשקל נשאר אותו הדבר אז אין משמעות לשינוי. בקודקוד 2, נשנה את הערך ל – $2 + 7 = 9$.

וגם, לא נשכח להסיר את צומת 5 מרשימת הצמתים.

צומת	מרחק	הורה
0	∞	
1	0	
2	9	5
3	∞	
4	5	1
5	2	1

כעת, הרשימה היא $open = [0,2,4]$.

עכשיו, נעבור אל קודקוד 4: יש לו שני שכנים שעוד לא ביקרנו בהם: 0, 2. אם מגיעים לקודקוד 2 דרך קודקוד 4, אז המשקל קטן ($5 + 2 < 9$) ולכן נשנה את הטבלה.
בקודקוד 0 נשנה את המשקל ל – $5 + 4 = 9$.

צומת	מרחק	הורה
0	5+4	4
1	0	
2	7	4
3	∞	
4	5	1
5	2	1

עכשיו, הרשימה היא $open = [0,2]$.

נבחר את קודקוד 2: השכנים שעוד לא ביקרנו בהם הם 3 ,0. נעדכן את הערך של קודקוד 3 ל – $7 + 1 = 8$. אין סיבה לעדכן את צומת 0 כיוון ש – $7 + 6 > 9$ מה שרק יגדיל את המשקל של המסלול.

צומת	מרחק	הורה
0	9	4
1	0	
2	7	4
3	7+1	2
4	5	1
5	2	1

$open = [0]$

הקודקוד האחרון שנשאר הוא קודקוד ה – 0 שהשכן היחיד שלו שאנחנו צריכים לבקר בו הוא צומת 3. שוב, $9 + 3 > 8$ מה שיגדיל את המשקל ולכן לא נשנה דבר.

לבסוף קיבלנו את הטבלה הבאה שמייצגת מסלול אופטימלי שמסתיים בצומת 3. נלך אחורנית בטבלה ונקבל את המסלול ההפוך הבא:

צומת	מרחק	הורה
0	9	4
1	0	
2	7	4
3	8	2
4	5	1
5	2	1

$3 \to 2 \to 4 \to 1$

$1 \to 4 \to 2 \to 3$

הוא המסלול האופטימלי בעל **משקל 8**.

חשוב לציין כי לא כל הגרפים מלאים, ולכן לא תמיד נוכל למצוא מסלול ישיר בין שני צמתים בגרף. לכן, חשוב לשים לב מתי אלגוריתם דייקסטרה יכול להיכשל:

- הראשונה, ברגע שאנחנו נמצאים בקודקוד שכל שכניו הם קודקודים אחרים שכבר ביקרנו בהם.
- או ברגע שאנחנו מוקפים בקודקודים שבהם משקלי הקשתות שמחברים בניהם הם אינסופיים.

מעתה, נתחיל לעסוק בבעיה ופתרון שלה.

בניית הגרף:

1. **הגדרת המפה כגרף:** נתאר את המפה של העיר חולון באמצעות מבנה נתונים של גרף $G = (V, E)$. לכל רחוב במפה, נגדיר קשת בגרף, ונוסיף לה שדה של אורך הרחוב. לכל צומת במפה שמחברת שני רחובות או יותר, נגדיר קודקוד בגרף.

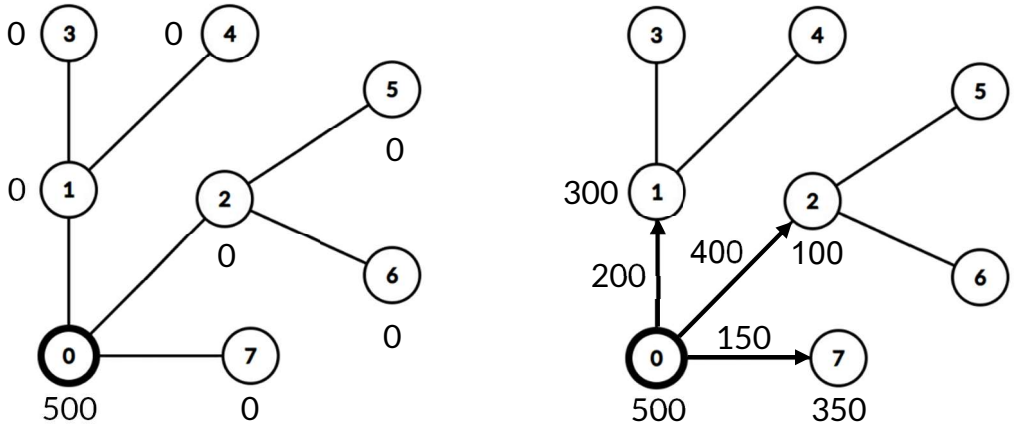
2. **קליטת המקלטים:** נתונה לנו רשימה של מקלטים בעיר. לכל מקלט, נחצה את הרחוב שבו הוא נמצא לשניים, ונגדיר קודקוד חדש בגרף שייקרא "קודקוד מקלט", ונשנה את הערכים של המרחקים המוכלים בקשתות החדשות בהתאם.

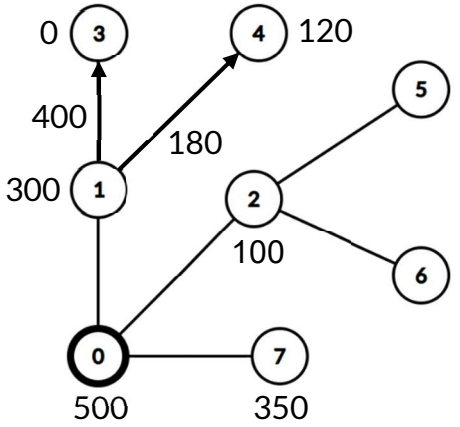
לכל קודקוד (מקלט / צומת) נגדיר שדה חדש בשם $remain$ שהוא יהיה המרחק שנשאר בין הקודקוד למקלט הקרוב ביותר. לכל מקלט, נכניס ערך של פרמטר קבוע שיהיה המרחק המקסימלי שיהיה לנו מותר להתרחק ממקלט. המוטיבציה היא שבזמן אזעקה, יהיה לנו מספיק זמן להגיע מנקודה מסוימת על המסלול עד למקלט הקרוב ביותר בזמן.

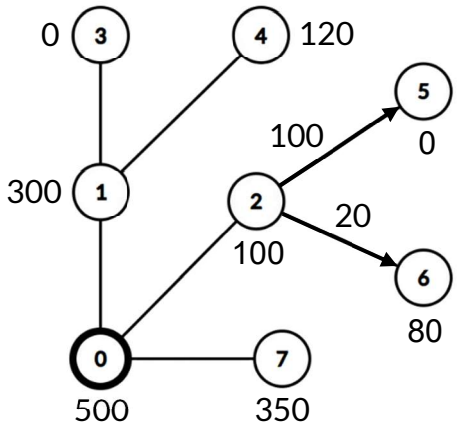
לאחר מכן, נרוץ בלולאה על כל מקלט ונבצע עליו אלגוריתם רקורסיבי שלבסוף יעדכן את כל הערכים של כל הקודקודים במפה הנמצאים במרחק מתאים ממקלט: נעדכן למקלט את הערך MAX (לדוגמה: 500 מ'), ואז נעדכן לכל אחד מהשכנים שלו את הערך של $remain$ ע"פ אורך הרחוב המחבר בניהם. כך נמשיך לכל אחד מהשכנים עד ש - $remain$ יהיה קטן יותר מאורך הרחוב. אם קודקוד מסוים לא נמצא בטווח קרוב למקלט, ערך המשתנה $remain$ יהיה 0.

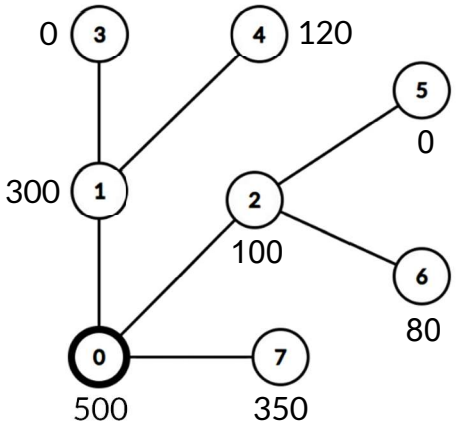
דוגמה 2:

נניח שבשרטוט הבא קודקוד 0 הוא קודקוד מקלט, והרדיוס המקסימלי 500:









<u>הערה:</u> את החלק הזה בתוכנית נצטרך לבצע פעם אחת בלבד ואת הערכים האלה נוכל לשמור ולהשתמש בהם בכל פעם מחדש.

המודל המתמטי:

3. **מציאת המרחק הקצר ביותר:** כעת, נתרכז בחלק הראשון של הבעיה: מציאת המרחק הקצר ביותר בין שני צמתים בגרף. לא נוכל להשתמש באלגוריתם דייקסטרה למציאת המרחק כיוון שבנוסף למזעור אורך הדרך, עלינו גם להתייחס לאילוצי המקלטים וגם עלינו לקחת בחשבון שיש גם משקל מסוים למרחק המסלול מהמקלטים הפרוסים כיוון שעל הדרך להיות מהירה אך גם בטוחה. לכן, נגדיר את הבעיה הזו כבעיית תכנון לינארי:

נגדיר את קבוצת המשתנים X בעלת $2|E|$ איברים:

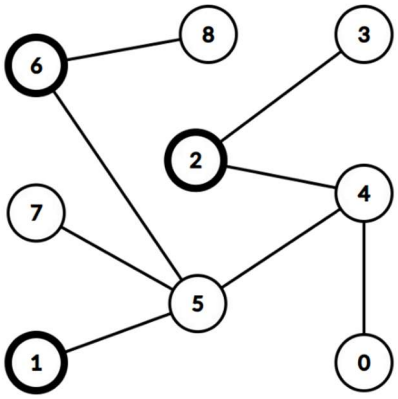
$$X_{u,v,k} = \begin{cases} 1\,, & \text{המסלול עובר בקשת } (u,v,k) \text{ מצומת } u \text{ ל- } v \\ 0\,, & \text{אחרת} \end{cases}$$

אז, פונקציית המטרה שמתארת את האורך של הדרך תהיה מוגדרת להיות:

$$f_1(X) = \sum_{(i,j,k)\in E} X_{i,j,k} * \|(i,j,k)\|$$

כעת, נגדיר את האילוצים על כל צומת. לכל צומת, נוכל להגדיר חוק שימור: מספר הכניסות לצומת אליו = מספר היציאות ממנו. זאת בפרט לצומת ההתחלה וצומת היעד: מספר היציאות מצומת ההתחלה = מספר הכניסות לצומת היעד = 1.

<u>דוגמה 3:</u>



נניח שצומת ההתחלה היא 2, וצומת היעד היא 6. לכן, נוכל להגדיר את שני האילוצים הבאים:

$$X_{2,3} + X_{2,4} = 1$$

$$X_{5,6} + X_{8,6} = 1$$

בנוסף לכך, נוכל להגדיר את חוק השימור על כל אחד מהצמתים. נראה זאת על צומת מספר 5:

$$X_{5,7} + X_{5,1} + X_{5,4} = X_{7,5} + X_{1,5} + X_{4,5}$$

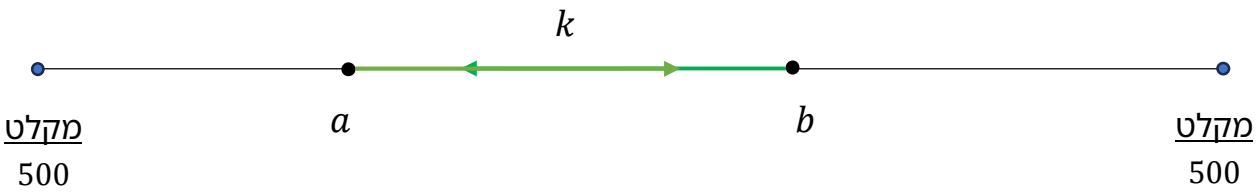
עד כה, הצגנו בעיית הדרך הקצרה ביותר קלאסית ואת המבנה הכללי שלה.

4. **בעיית אילוצי המקלטים:**

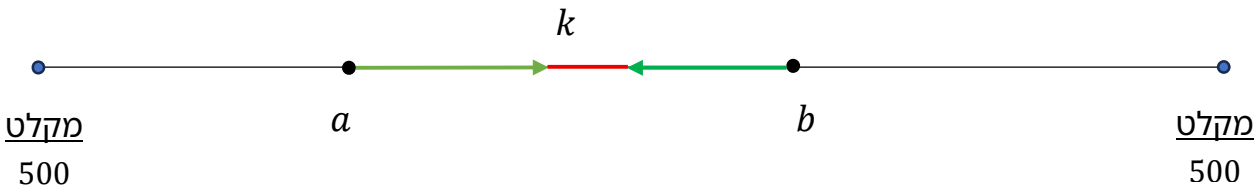
החלק השני בבעיה הוא האינטגרציה של המקלטים לבעיית התכנון.
ראשית כל, עלינו לבדוק האם אנחנו בכלל יכולים להיכנס לקשת מסוימת. אם אנחנו נעים מצד אחד של קשת לצידה השני וקיים בה חלק שלא מוגן כלל, כלומר חלק ממנה לא נמצא בטווח קרוב למקלט, אסור לנו כלל להיכנס אליה.
לשם כך, נטען את הטענה שהקשת (u,v,key) מוגנת כולה אם:

$$u[remain] + v[remain] > \|(u,v,key)\|$$

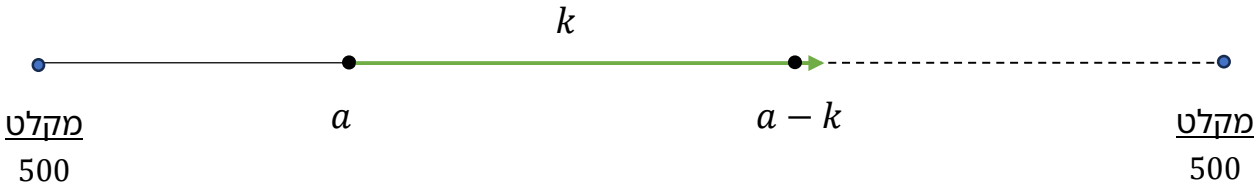
מדוע? ראשית כל, נשים לב שכל צומת מגדירה סביבה "מעגל מוגן" בעל רדיוס של גודל ה – $remain$ שלו ביחס למטריקת הרחובות שהגדרנו.
במקרה הראשון, נניח ששני המקלטים נמצאים בשני צדדים שונים של הרחוב, כלומר:



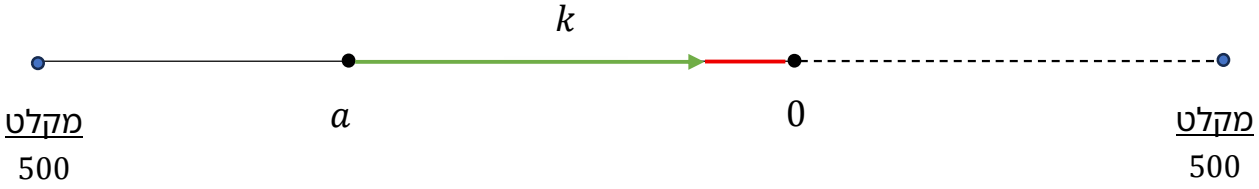
נוכל לראות שרק כאשר $a + b \geq k$, הרחוב כולו מוגן כיוון שהוא מוכל באיחוד של שני המעגלים המוגנים של שני הצמתים, והכניסה אליו מותרת.
ואילו במקרה ההפוך שבו $a + b < k$, נוכל לראות שקיים חלק ברחוב שאינו מוכל אף אחד מהם, לכן הוא לא מוגן ולא נוכל להיכנס אליו:



המקרה השני הוא שבו יש מקלט רק בצד אחד הרחוב בעוד שאין מקלט בצד השני שהטווח שלו מגיע עד לשם:



במצב הזה, אם $a + a - k \geq k$ אז $a \geq k$ וזה מבטיח שכל הרחוב מוכל ב"מעגל המוגן" של הצומת השמאלית. לעומת זאת, במצב ההפוך שבו:



אז נקבל $a + 0 < k$ מה שאומר שיהיה חלק ברחוב שלא יהיה מוכל באף מעגל בטוח של אף צומת ונקבל רחוב שלא מוגן באחד מצדיו.

החלק השני הוא התייחסות למרחק המסלול מהמקלטים הפרוסים ברחבי המפה. לכן, נגדיר את קבוצת המשתנים Y_u להיות:

$$Y_u = \begin{cases} 1\,, & \text{המסלול עובר דרך הצומת ה – } u \\ 0\,, & \text{אחרת} \end{cases}$$

לכן, נוכל להגדיר פונקציית מטרה נוספת $f_2(Y)$ המוגדרת להיות:

$$f_2(Y) = \sum_{u \in V} Y_u * u[remain]$$

שמתארת את סכום המרחקים כל קודקוד בגרף מהמקלט הקרוב אליו ביותר.

נוכל להגדיר את פונקציית המטרה הכוללת כממוצע משוכלל של שתי פונקציות המטרה שהגדרנו למעלה. כדי שהדרך תהיה יותר מהירה, ניתן יותר משקל לפונקציית אורך הדרך $f_1(X)$. באותה צורה, כדי שהדרך תהיה יותר בטוחה, ניתן משקל גדול יותר לפונקציה $f_2(Y)$ שמתארת את מרחק המסלול מהמקלטים. המטרה שלנו היא למצוא את המינימום של הפונקציה הזו, כלומר:

$$\min_{X,Y} M * f_1(X) + (1 - M) * f_2(Y) \qquad 0 < M < 1$$

ובנוסף, אם נמצא שרחוב מסוים הוא אינו מוגן כולו, אז נאמר ש:

$$X_{u,v,k} = X_{v,u,k} = 0$$

5. **קשר בין המשתנים X ו – Y**:

קודם כל, אם הדרך מתחילה מקודקוד s ומסתיימת בקודקוד d, אז אוטומטית נוכל לומר ש: $Y_s = Y_d = 1$ כיוון שהדרך חייבת לעבור בהם. לאחר מכן, נסתכל על חוק השימור על כל אחת משאר הצמתים:

אם הדרך עוברת בצומת u, אז מספר הכניסות לצומת = מספר היציאות = 1.

אחרת, מספר הכניסות = מספר היציאות = 0.

המספר שנקבל לעולם לא יהיה גדול יותר $(2,3,\dots)$ כיוון שאם עברנו דרך צומת מסוימת, אין לנו אף סיבה לחזור אליה כיוון שאז אנחנו עוברים ב"לולאה" בגרף שרק מאריכה את הדרך ואת סכום המרחקים מהמקלטים – מגדילה את פונקציית המטרה.

אז, נוכל לומר לסיכום שאם נשתמש בחוק השימור על הצומת u:

מספר הכניסות ל – u = מספר היציאות מ – u = Y_u.

לדוגמה: אם נסתכל על דוגמה 3, נוכל לומר ש:

$$Y_5 = X_{5,7} + X_{5,1} + X_{5,4} = X_{7,5} + X_{1,5} + X_{4,5}$$

לכן, נוכל להביע את כל אחד מהמשתנים Y_u כקומבינציה לינארית של משתנים מקבוצה X, להציבם ישירות בפונקציית המטרה ולהעלים לחלוטין את המשתנים Y_u. נפשט את הביטוי שהתקבל ונקבל פונקציית מטרה מהצורה הבאה:

$$f(X) = \sum_{(i,j,k)\in E} X_{i,j,k} * \hat{d}_{i,j,k}$$

כפי שאנחנו רואים, קיבלנו פה פונקציית מטרה במבנה של בעיית הדרך הקצרה ביותר כפי שהגדרנו אותה בשלב 3, אך הפעם שאורך הרחוב הוא אינו אורכו האמיתי, אלא משקל אחר $\hat{d}$.

לכל צומת, נציב בפונקציית המטרה: Y_u = מספר הכניסות ל – u. המטרה שלנו היא למצוא את המקדם של כל אחד מהמשתנים $X_{u,v,k}$. המקדם מורכב מסיכום של שני ערכים: הראשון, $M * \|(u,v,key)\|$ (תרומה מ – $M * f_1(X)$), והשני, $(1-M) * u[remain]$ (התרומה מ – $(1-M) * f_2(Y)$). אז:

$$\hat{d}_{u,v,key} = M * u[remain] + (1-M) * \|(u,v,key)\|$$

קיבלנו כעת בעיית תכנון לינארית בשלמים (בעיה שאינה קלה לפתרון) ובנוסף עלינו להוסיף לכל משתנה אילוץ שאומר ש – $0 \le X_{u,v,k} \le 1$ כדי להבטיח שהמשתנה יקבל רק ערכים בינאריים.
אבל בסופו של דבר, קיבלנו כאן בעיית מציאת הדרך הקצרה ביותר סטנדרטית כפי שהגדרנו למעלה, ובשביל בעיות כאלה יש לנו את אלגוריתם דייקסטרה והבעיה שמנעה מאיתנו להשתמש בו עכשיו נעלמה.

סיכום האלגוריתם:

ההסבר הארוך שהצגנו מסתכם לבסוף לאלגוריתם קצר וקומפקטי:

א. נמיר את מפת העיר שלנו לגרף כך שכל צומת בעיר יהפוך לקודקוד וכל רחוב שמחבר בין שני צמתים יהפוך לקשת המחברת שני קודקודים בגרף.

ב. נוסיף לגרף שיצרנו את המקלטים: לכל מקלט, נוסיף קודקוד מתאים בגרף ונחצה לשניים את הרחוב שהוא נמצא בו ונעדכן את אורכי הקשתות החדשות בהתאם.

ג. לכל קודקוד בגרף, נוסיף שדה $remain$ שמתאר את המרחק של הקודקוד מהמקלט הקרוב ביותר (ביחס למטריקת הרחובות) ואז נבצע אלגוריתם רקורסיבי שיעדכן את הערכים של שדה $remain$ של כל אחד מהקודקודים.

ד. נקלוט מהמשתמש את צומת המקור והיעד, ובנוסף את "אחוז הבטיחות" שמציין את המשקל של בטיחות בזמן תכנון המסלול – עד כמה תהיה קרובה הדרך ממקלטים.

ה. לכל אחד מהקשתות בגרף, נגדיר שני משקלים חדשים. אם נסתכל על הקשת בגרף (u, v, key), אז:

$$\hat{d}_{u,v,key} = M * u[remain] + (1 - M) * \|(u, v, key)\|$$

$$\hat{d}_{v,u,key} = M * v[remain] + (1 - M) * \|(u, v, key)\|$$

כאשר $0 \leq M \leq 1$ הוא "מקדם הבטיחות" שמציין כמה הדרך שאנחנו רוצים לקבל תהיה בטוחה.

ו. נשתמש באלגוריתם דייקסטרה בשביל למצוא את הדרך הקצרה ביותר ביחס למרחקים המדומים שחישבנו.

<u>הערה:</u> האלגוריתם שנשתמש בו לחישוב הדרך הוא לא אלגוריתם דייקסטרה המדויק כיוון שאורך הדרך משתנה בהתאם לכיוון התנועה – האם אנחנו נעים מצומת A ל – B או להפך. אבל, נוכל להסתכל על קשת מסוימת כאל שני קשתות חד-כיווניות בעלות אורכים שונים ואז נקבל גרף רגיל עם אורכים חד-ערכיים שאפשר לעשות עליו את האלגוריתם הבסיסי.

מימוש האלגוריתם:

נשתמש בספריית $OpenStreetMap$ ב – $Python$ בשביל לייבא את המפה של העיר חולון. הספרייה יוצרת אוטומטית גרף מהמפה שהוריד לפי הכללים שקבענו בשלב א'.

לאחר מכן, נקרא לפונק' $build\_map()$ שמייבאת רשימה של מקלטים נתונים בעיר חולון וממשת את שלבים ב' – ג': להוסיף את המקלטים לגרף ולחשב את ערכי שדות ה – $remain$ בכל אחד מהצמתים.

הפונקציה שומרת את הערכים הללו בקובץ $graph.json$.

פונקציות עזר

פיצול הרחוב לשני חלקים להוספת מקלט

הוספת המקלטים לגרף

פונק' רקורסיבית לחישוב שדה $remain$

יצירת שדות ה – $remain$

כתיבת קובץ $json$ לשמירת הנתונים

```python
from shapely.geometry import LineString
import json
import osmnx as ox

# Utility functions for graph manipulation
def distance(loc1, loc2):
    return ox.distance.great_circle(lat1=loc1[0], lon1=loc1[1],
                                    lat2=loc2[0], lon2=loc2[1])

def address_to_coords(address):
    try:
        full_address = f"{address}, Holon, Israel"
        location = ox.geocode(full_address)
        return location

    except Exception as e:
        print(f"Error converting address '{address}': {e}")
        return None

# Functions to adding the shelters to the graph
id = 0
def split_street(G, u, v, key, loc):
    global id
    u_loc = (G.nodes[u]['x'], G.nodes[u]['y'])
    v_loc = (G.nodes[v]['x'], G.nodes[v]['y'])
    shelter_loc = (loc[1], loc[0])

    new_node = f"{id}"
    id += 1
    G.add_node(new_node, y=loc[0], x=loc[1])
    G.add_edge(u, new_node, length=distance((G.nodes[u]['y'], G.nodes[u]['x']), loc),
               geometry=LineString([u_loc, shelter_loc]))
    G.add_edge(new_node, v, length=distance((G.nodes[v]['y'], G.nodes[v]['x']), loc),
               geometry=LineString([shelter_loc, v_loc]))
    G.remove_edge(u, v, key)
    return new_node

def add_shelter_nodes(G, shelters):
    shelter_nodes = list()

    for shelter in shelters:
        loc = shelter['coords']
        u, v, key = ox.nearest_edges(G, loc[1], loc[0])
        new_node = split_street(G, u, v, key, loc)
        G.nodes[new_node]['shelter'] = True
        G.nodes[new_node]['name'] = shelter['name']
        shelter_nodes.append(new_node)

    return shelter_nodes

# Proccessing the graph to create circles around shelters
def create_circle_rec(G, node):
    for u, v, data in G.edges(node, data=True):
        other = u if v == node else v
        remain = G.nodes[node]['rem'] - data['length']

        if remain < G.nodes[other]['rem']:
            continue
        elif remain < data['length']:
            G.nodes[other]['rem'] = 0
        else:
            G.nodes[other]['rem'] = remain
            create_circle_rec(G, other)

def create_circles(G, nodes, radius):
    for node in G.nodes:
        G.nodes[node]['rem'] = 0

    for node in nodes:
        G.nodes[node]['rem'] = radius
        create_circle_rec(G, node)

def calc_lengths(G):
    lengths = {}
    for u, v, key, data in G.edges(keys=True, data=True):
        lengths[f"{u} {v} {key}"] = {"length": data['length']}
        lengths[f"{v} {u} {key}"] = {"length": data['length']}
        if G.nodes[u]["rem"] + G.nodes[v]["rem"] > data['length']:
            lengths[f"{u} {v} {key}"]["rem"] = G.nodes[u]["rem"]
            lengths[f"{v} {u} {key}"]["rem"] = G.nodes[v]["rem"]
        else:
            lengths[f"{u} {v} {key}"]["rem"] = -1
            lengths[f"{v} {u} {key}"]["rem"] = -1
    return lengths

def build_map():
    G = ox.graph_from_place("Holon, Israel", network_type="drive")
    with open("shelters.json", encoding="utf-8") as f:
        shelters = json.load(f)
    RADIUS = 500

    shelter_nodes = add_shelter_nodes(G, shelters)
    create_circles(G, shelter_nodes, RADIUS)
    lengths = calc_lengths(G)

    with open("processed.json", "w") as f:
        json.dump(lengths, f, indent=4)

    print(f"Saved {len(lengths)} edge lengths to processed.json")
    return G

if __name__ == "__main__":
    G = build_map()
```

כל אחד מצוי ברשימה בקובץ $json$ ומוגדרים איתו שני שדות: אורך הקשת וערך ה – $remain$ של הצומת ממנו הקשת מתחילה:

```
"279963239 983107895 0": {
        "length": 7.586324890242367,
        "rem": 208.69654656274918
    },
```

בדוגמה זו, ניתן לראות את הקשת $(279963239,\ 983107895,\ 0)$ שאורך הרחוב הוא $\approx 7.5(m)$ וערך שדה ה – $remain$ של צומת 279963239 הוא $\approx 208.69(m)$.

בחלק הבא, נבקש מהמשתמש את "אחוז הבטיחות" ואת צומת המוצא והיעד. לאחר מכן, נבצע לכל קשת ממוצע משוכלל בין האורך שלה לערך ה – $remain$ שלה, וכך נקבל את המשקל של כל קשת בגרף. ולבסוף, נבצע את אלגוריתם דייקסטרה ונמצא את הדרך האופטימלית לפי רצוננו.

בקטע קוד הבא, נכתוב פונקציה שמקבלת שמות של צומת המוצא והיעד ואת אחוז הבטיחות, ויודעת להחזיר רשימה של צמתים המייצגים את המסלול המבוקש:

הגדרת ערכי המשקלים

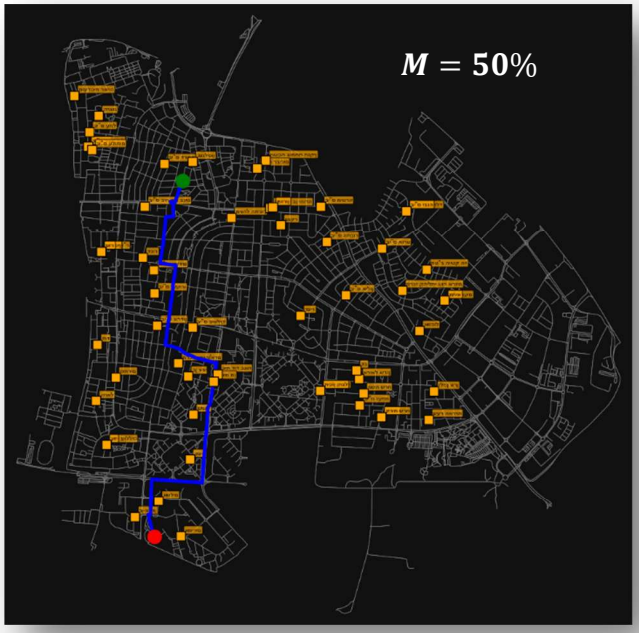
מימוש אלגוריתם דייקסטרה

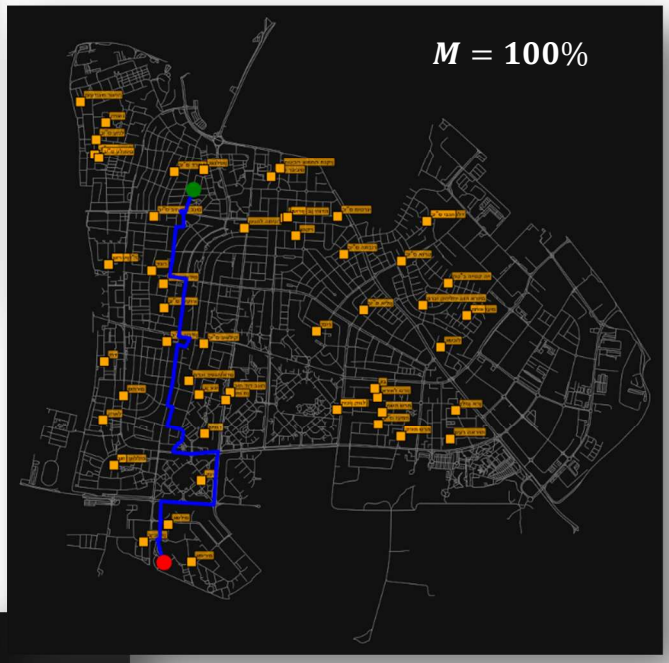
```python
import json
import heapq
from collections import defaultdict

def find_way(start, dest, M):

    with open('graph.json', 'r') as f:
        graph_data = json.load(f)

    graph = defaultdict(list)

    for edge_key, edge_data in graph_data.items():
        parts = edge_key.split()
        if len(parts) >= 2:
            source = parts[0]
            target = parts[1]

            length = edge_data['length']
            rem = edge_data['rem']

            if rem == -1:
                weight = float('inf')
            else:
                weight = M * length + (1 - M) * rem

            graph[source].append((target, weight))

    # Initialize Dijkstra table: Node | Distance | Parent
    dijkstra_table = {}
    visited = []

    all_nodes = set()
    for node in graph.keys():
        all_nodes.add(node)
    for node_neighbors in graph.values():
        for neighbor, _ in node_neighbors:
            all_nodes.add(neighbor)

    for node in all_nodes:
        dijkstra_table[node] = {
            'distance': float('inf'),
            'parent': None
        }

    dijkstra_table[start]['distance'] = 0

    pq = [(0, start)]

    while pq:
        current_dist, current_node = heapq.heappop(pq)

        if current_node in visited:
            continue

        visited.append(current_node)

        if current_node == dest:
            path = []
            node = dest
            while node is not None:
                path.append(node)
                node = dijkstra_table[node]['parent']
            path.reverse()

            return dijkstra_table[dest]['distance'], path

        for neighbor, weight in graph[current_node]:
            if neighbor not in visited:
                new_dist = current_dist + weight

                if new_dist < dijkstra_table[neighbor]['distance']:
                    dijkstra_table[neighbor]['distance'] = new_dist
                    dijkstra_table[neighbor]['parent'] = current_node
                    heapq.heappush(pq, (new_dist, neighbor))

    return float('inf'), []
```

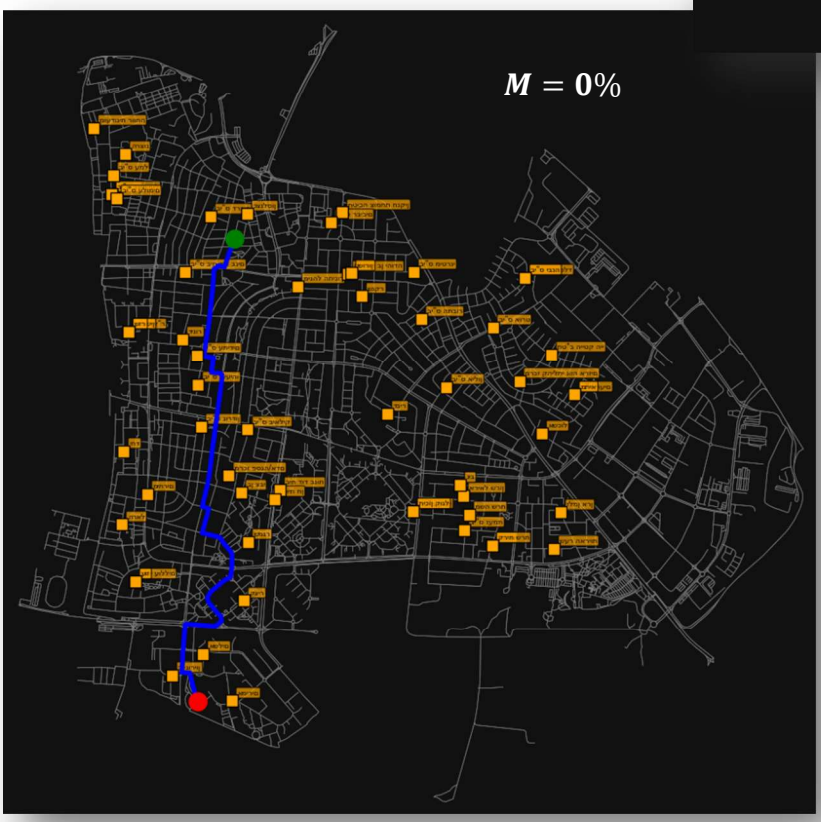
דוגמת הרצה:

להלן קוד לדוגמה שמשתמש בפונקציות שכתבנו לצורך תכנון הדרך. בחרנו בשני צמתים בצדדים שונים של העיר ובדקנו את תכנון הדרך עם אחוז בטיחות שונה בכל פעם: 0% , 50% , 100%. ניתן לראות שקיבלנו דרכים דומות אך עדיין טיפה שונות בכל פעם:

```python
import matplotlib.pyplot as plt
import osmnx as ox
import json
from find_way import find_way

def get_holon_graph_and_coordinates():
    print("Downloading Holon map...")
    G = ox.graph_from_place("Holon, Israel", network_type="walk")

    coordinates = {}
    for node, data in G.nodes(data=True):
        coordinates[str(node)] = (data['y'], data['x'])  # (lat, lon)

    return G, coordinates

def plot_route_matplotlib(path, G, coordinates):
    if not path or len(path) < 2:
        print("No valid path to display")
        return

    with open('shelters.json', 'r', encoding='utf-8') as f:
        shelters = json.load(f)

    fig, ax = ox.plot_graph(G, figsize=(15, 15), node_size=0, edge_linewidth=0.5,
                            edge_color='gray', show=False, close=False)

    route_coords = []
    for node in path:
        if node in coordinates:
            coord = coordinates[node]
            route_coords.append(coord)

    if len(route_coords) < 2:
        print("Not enough coordinate data to plot route")
        return

    route_lats = [coord[0] for coord in route_coords]
    route_lons = [coord[1] for coord in route_coords]

    ax.plot(route_lons, route_lats, 'b-', linewidth=4, label='Route', zorder=10)
    ax.plot(route_lons[0], route_lats[0], 'go', markersize=15, label='Start', zorder=11)
    ax.plot(route_lons[-1], route_lats[-1], 'ro', markersize=15, label='End', zorder=11)

    # Add shelters to the plot
    shelter_lats = [shelter['coords'][0] for shelter in shelters]
    shelter_lons = [shelter['coords'][1] for shelter in shelters]
    ax.scatter(shelter_lons, shelter_lats, c='orange', s=100, marker='s',
               label='Shelters', zorder=12, edgecolors='black', linewidth=1)

    for i, shelter in enumerate(shelters):
        ax.annotate(shelter['name'], (shelter['coords'][1], shelter['coords'][0]),
                    xytext=(5, 5), textcoords='offset points', fontsize=6,
                    bbox=dict(boxstyle="round,pad=0.2", facecolor="orange", alpha=0.7))

    plt.tight_layout()
    plt.show()

def main():
    print("Finding route from 719832096 to 8025915366 with M=0.7...")

    # Every time we change the M value to something else.
    distance, path = find_way("1279298397", "1743699231", 0.5)

    print(f"Distance: {distance:.2f}")
    print(f"Path length: {len(path)} nodes")
    print(f"Path: {' -> '.join(path)}")

    G, coordinates = get_holon_graph_and_coordinates()

    print("\nCreating route visualization...")

    plot_route_matplotlib(path, G, coordinates)

if __name__ == "__main__":
    main()
```


$M = 50\%$


$M = 100\%$


$M = 0\%$