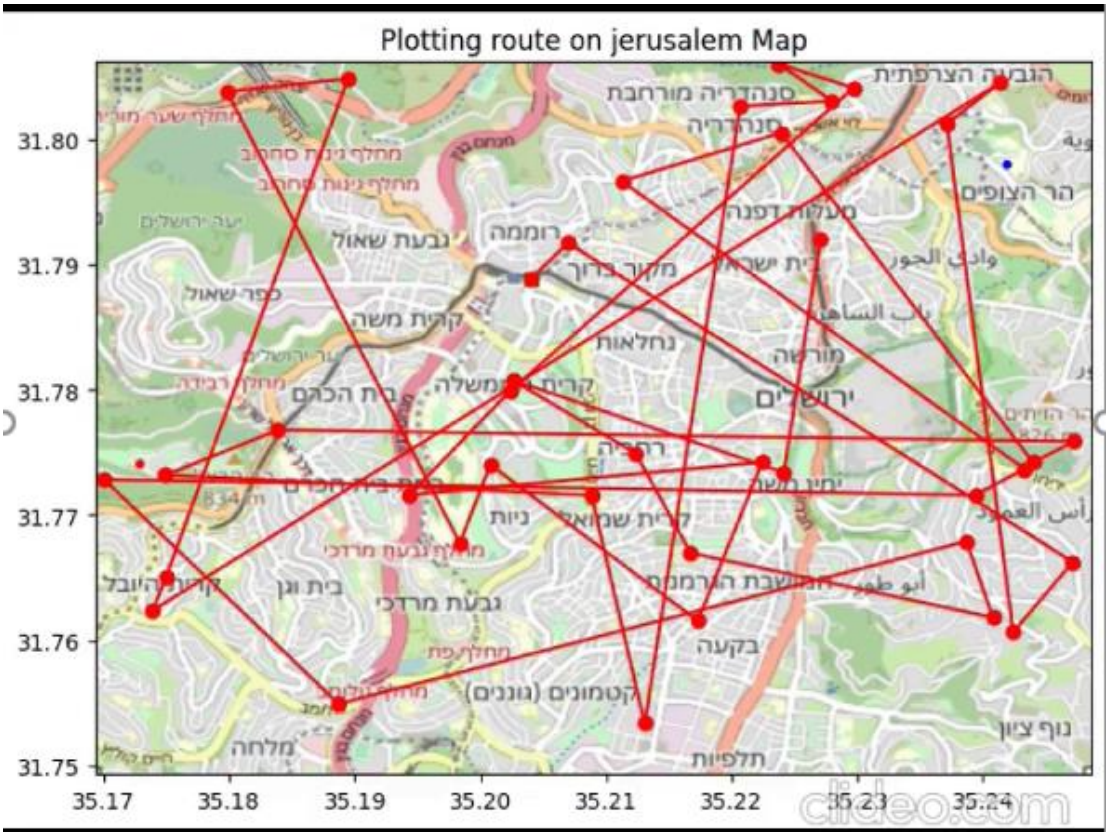
בעיית איסוף הנוסעים



מגישים:

חסין צ'ו יאנג

נריה גרנות

נדב כדורי

שלומי אדלמן

תוכן עניינים

הקדמה

נפתח בשני תיאורים, שלכאורה הקשר ביניהם אינו חזק במיוחד.

בשנים האחרונות, התחדש עולם התחבורה הציבורית בקונספט חדש: "מונית משותפת". במסגרת זאת, כאשר מספר אנשים הנמצאים במרחק לא גדול זה מזה מעוניינים כולם בשירותי הסעה, איש איש ליעדו, באפשרותם להזמין מונית באמצעות אפליקציה. האפליקציה תתזמן מונית (מתוך צי מוניות קיים) שתעבור בין האנשים ותביא כל אחד ליעדו. שיטה זו מאפשרת ניצול טוב יותר של המשאבים הקיימים (קרי, המוניות), ובכך תורמת להפחתת העלות עבור כל אחד מהנוסעים.

בנוסף, חברות שונות כמו Amazon השיקו בשנים האחרונות שירות הכולל איסוף משלוחים ממחסנים שונים והעברתם לבתי הלקוחות באמצעות רחפנים. אין צורך להכביר במילים בדבר הנוחות שיש בכך לציבור הצרכנים, ובדבר החשיבות הקיימת בהבטחת היעילות המרבית של שירותים כגון אלו.

גם אם במבט ראשון מדובר בשני עולמות שונים, לאמיתו של דבר שני התרחישים הללו מהווים מופע של בעיה כללית אחת, שאותה ננתח בפרויקט זה.

הצגת הבעיה הכללית

באופן כללי, הבעיה שננתח היא זו: לרשותנו צי של רכבים (כפי שנראה, במסגרת פרויקט זה הצי כולל רכב אחד בלבד), שלהם קיבולות שונות. מתקבלות קריאות ממספר מסוים של אנשים המעוניינים באיסוף מנקודה מסוימת ובהגעה לנקודה אחרת. עלינו לתזמן את שליחת הרכבים לאנשים הללו באופן שבו עלות הנסיעה תהיה מינימלית, בהתאם למדד עלות מוסכם כלשהו.

נגדיר כעת את הבעיה בצורה פורמלית.

<u>קלט:</u>

- N בקשות הסעה.
- כל בקשה הינה טאפל $((s_1,s_1),(d_2,d_2))$ המכיל שתי נקודות גיאוגרפיות: נקודת מוצא (source) ונקודת יעד (destination).
- K כלי תחבורה, כך שעבור כל i, קיבולתו של k_i היא c_i.
- פונקציה distance המקבלת כקלט שתי נקודות גיאוגרפיות ומחזירה את המרחק ביניהם, לפי מדד כלשהו (למשל, מרחק אוקלידי).
- נקודת מוצא גיאוגרפית עבור צי הרכבים.

<u>מטרה:</u>

- על כל נוסע להגיע ליעדו.
- זאת, מבלי להפר את אילוצי הקיבולת על הרכבים (כלומר, באף שלב אסור שרכב יכיל יותר נוסעים מהקיבולת שלו).
- וזאת, כאשר המרחק הכולל שמבצעים הרכבים במהלך מסלולם הינו מינימלי (ביחס לפונקציית המרחק הנתונה).

<u>פלט:</u>

- K רשימות, אחת עבור כל רכב.
- כל רשימה מכילה את המסלול שמבצע הרכב המתאים לה (מסלול זה תמיד מתחיל בנקודת המוצא של צי הרכבים, הנתונה בקלט הבעיה).

הווריאציה שאנו חקרנו וסיווגה המחקרי

קונקרטית, במסגרת פרויקט זה חקרנו וריאציה ספציפית של הבעיה:

ראשית, עסקנו במקרה בו קיים רכב אחד בלבד בצי הרכבים. כלומר K=1.

שנית פונקציית המרחק שאיתה עבדנו היא פונקציית המרחק האוקלידי.

מבחינת הקיבולת של הרכבים, בדקנו שתי וריאציות אפשריות:

- C=1: הרכב יכול לשאת בכל רגע נתון נוסע אחד בלבד.
- C=∞: הרכב יכול לשאת בכל רגע נתון כמות נוסעים גדולה כרצוננו.

הווריאציות הללו מתאימות כמובן ליישומים שונים, ונרצה לבחון כל אחת מהן. את הנקודות עבור הקלט לבעיה לקחנו מנקודות שונות ברחבי ירושלים.

נציין, כי הווריאציה הראשונה, שבה ברכב יש מקום לנוסע אחד בלבד, מהווה למעשה גרסה של בעיית הסוכן הנוסע (עם שני הבדלים קלים: מחפשים מסילה המילטונית ולא מעגל המילטוני, ובנוסף, הגרף מכוון). נסביר:

- **קודקודים -** נוכל להגדיר כל בקשת הסעה כקודקוד בגרף. בנוסף נגדיר קודקוד עבור נקודת המוצא של כלי הרכב.
- **צלעות -** בנוסף, נגדיר צלע בין כל שני קודקודים (כלומר, מדובר בגרף מלא).
- **משקול -** לכל צלע נשייך משקל: משקל הצלע שמחברת בין קודקוד א' לקודקוד ב' יהיה המרחק בין נקודת הסיום המתאימה לבקשת ההסעה שמשויכת לקודקוד א' (ואם מדובר בנקודת המוצא של הרכבים – נסתכל פשוט על מיקום נקודה זו) לבין נקודת ההתחלה המתאימה לבקשת ההסעה שמשויכת לקודקוד ב'.

במידול זה, קל לראות כי הבעיה שלפנינו היא למעשה מציאת מסלול המילטוני בעל משקל מינימלי בגרף זה (המתחיל בנקודה נתונה).

לעומת זאת, הגרסה השנייה של הבעיה הינה מורכבת יותר, שכן היא משלבת אילוצים נוספים על מסלול הנסיעה, ובנוסף מרחב הפתרונות עבורה גדול יותר. בניגוד לגרסה הפשוטה, שבה אחרי העלאת נוסע בהכרח הרכב ייסע לנקודת היעד של נוסע זה (שכן אין בו מקום לנוסעים נוספים), בגרסה המורכבת ייתכן שיעלו נוסעים נוספים בטרם ירד מי מהם בנקודת היעד שלו. ממילא, לא נוכל להגדיר את משקול הצלעות כפי שהגדרנו קודם.

בעיה זו שייכת לתחום נחקר בשם SOP (sequential ordering problem), העוסק במציאת מסילת המילטון בעלת משקל מינימלי בגרף מכוון, בכפוף לאילוצים באשר לסדר שבו יש לעבור בין הקודקודים. בהקשר שלנו, האילוץ העיקרי הוא שחובה על הרכב להעלות נוסע לפני הורדתו.

מתודולוגיה ותהליך עבודה

כשניגשנו לפתרון הבעיה, חשבנו על מספר גישות אפשריות:

- Brute force
- גישה חמדנית
- Local search
  - וריאציות של אלגוריתם hill climbing
  - Genetic algorithm

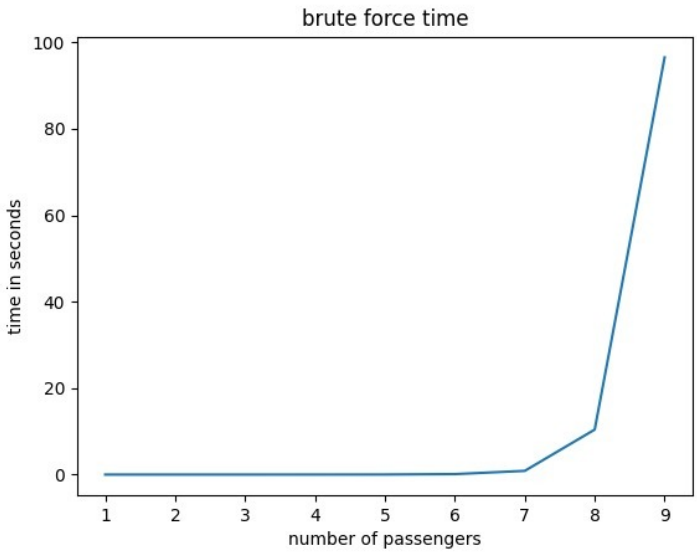
<u>Brute force</u>

הגישה הראשונה, והנאיבית מכולן, היא גישת brute force. במסגרת זאת, עוברים על כל מסלול חוקי אפשרי שיכול לבצע הרכב, ומחזירים את המסלול שלו המשקל המינימלי.

כמובן, גישה זו אינה יכולה לספק תוצאות טובות, שכן הבעיה שלנו היא Np קשה. סיבוכיות אלגוריתם שכזה היא $\Omega(n!)$. נסביר זאת:

נניח שיש לנו n נוסעים. נניח הנחה מפשטת, שתמיד מורידים נוסע ביעדו מיד לאחר שאוספים אותו (הנחה זו היא הנחה מפשטת עבור הגרסה השנייה של הבעיה, אך היא הגדרת הבעיה עבור הגרסה הראשונה). ניתן אם כן לעבור על כל פרמוטציה אפשרית של n הנוסעים, ולבצע מסלול שמסיע אותם ליעדם לפי הסדר. מאחר ומספר הפרמוטציות הוא n!, נקבל את הסיבוכיות הנ"ל.

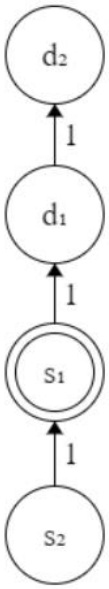
גישה זו, אם כן, אינה פרקטית. ואכן, כבר עבור יותר מ9 נוסעים, האלגוריתם אינו מסיים לרוץ בזמן סביר (ראו גרף מצורף).



<u>גישה חמדנית</u>

הגישה השנייה, המבטיחה יותר, שחשבנו עליה, היא הגישה החמדנית. הרעיון הוא פשוט: בכל פעם נשלח את הנהג לנקודה הקרובה ביותר למקום בו הוא נמצא, וזאת בתנאי ששומרים על אילוצי החוקיות (למשל, שלא מורידים נוסע לפני שאספנו אותו). בסוף יתקבל מסלול, ואותו נחזיר כפתרון. הסיבוכיות במקרה זה היא o(n^2), שהרי בכל פעם עלינו לעבור על כל N נקודות הקלט בשביל למצוא את הנקודה הקרובה ביותר אלינו, וזאת עלינו לעשות N פעמים. עם זאת, הפתרון המושג אינו אופטימלי.

ניווכח בכך בדוגמה הפשוטה הבאה.



נניח שהנקודה הקרובה ביותר לנקודת המוצא היא s_1. אז המסלול שיבצע האלגוריתם החמדן הוא s_1-d_1-s_2-d_2. משקלו של מסלול זה הוא 6 (בהנחה שהנקודות מסודרות בקו ישר, כבדוגמה). לעומת זאת, המסלול האופטימלי הוא s_1-s_2-d_1-d_2, שמשקלו הוא 4.

לכן, לא הסתפקנו בפתרון שנותן האלגוריתם החמדן, וחיפשנו דרכים נוספות.

Local search

הגישה השלישית שבחנו היא שימוש בlocal search. גישה זו מתפצלת לשתי גישות עיקריות: האחת, בוחנת וריאציות שונות על אלגוריתם hill climbing, והשנייה בוחנת את יעילות השימוש באלגוריתמים גנטיים עבור הבעיה.

Hill climbing variations

הרצנו על הבעיה מספר וריאציות אפשריות של hill climbing, שאת כולן מימשנו:

חלקן דטרמיניסטיות:

- Hill climbing – בכל פעם, ממשיכים לשכן שלו הערך המינימלי, ועוצרים כאשר מגיעים לנקודת קיצון.
- Local beam search – הרצת k אלגוריתמי hill climbing המתקשרים ביניהם במקביל, כפי שנלמד בכיתה.
- Late acceptance hill climbing – וריאציה שנחקרה בשנים האחרונות, שבה מקבלים צעד שמרע את איכות הפתרון, אם מדובר בפתרון טוב יותר מזה שהיינו בו לפני k צעדים (עבור k קבוע כלשהו).
- Tabu search – וריאציה נוספת, שבה מתחזקים רשימה של הפתרונות האחרונים במרחב החיפוש שבהם ביקרנו. כל עוד פתרון נמצא ברשימה זו, נדע שלא לבקר בו שוב. כך נמנעים מחזרה על אותו פתרון פעמים רבות. בנוסף, בוריאציה זו תמיד בוחרים את השכן הטוב ביותר וממשיכים ממנו באיטרציה הבאה (גם אם הוא פחות טוב מהמצב שבו אנו נמצאים כעת), ובסוף מחזירים את המצב הכי טוב שביקרנו בו לאורך הריצה.

וחלקן רנדומליות:

- Random restart hill climbing – מבצעים מספר חיפושי hill climbing, כאשר בכל פעם מתחילים את החיפוש מנקודה אחרת, הנבחרת באופן רנדומלי.
- Stochastic beam search – גרסה של local beam search, שבה בוחרים את k הילדים עבור האיטרציה הבאה באופן רנדומלי (ולא פשוט בוחרים את הילדים הכי טובים שנמצאו במהלך האיטרציה).
- Simulated annealing - מאפשרים צעדים מרעים בהסתברות הולכת וקטנה (ההסתברות קטנה הן כפונקציה של הזמן והן כפונקציה של מידת ההרעה שיש בצעד).
- First choice hill climbing – במסגרתו בכל פעם מגרילים שכן וממשיכים ממנו אם הוא טוב יותר מהמצב הנוכחי (אחרת, מגרילים שוב).

נציין, שכדי לאפשר לאלגוריתמים השונים להתמודד עם מצב בו האזור בו נמצאים במרחב החיפוש הינו "מישורי" (כלומר יש אזור שבו לכל הנקודות יש את אותו ערך), לא סיימנו את הריצה מיד כאשר מגיעים לנקודת קיצון, אלא רק לאחר מספר idle iterations שבהן לא מצאנו ערך טוב יותר מזה הנוכחי.

בנוסף, היה עלינו למדל את הבעיה באופן שיתאים לביצוע אלגוריתמים אלו. נציין מספר פרטים הקשורים למידול:

- ייצגנו כל פתרון אפשרי במרחב החיפוש כוקטור של נקודות.
- כל נקודה מתאימה לנקודת מוצא\יעד של נוסע. אם יש שני נוסעים שנקודת המוצא שלהם זהה, למשל, עדיין נייצג זאת בוקטור כשתי נקודות נפרדות.
- כל נקודה בוקטור הינה אובייקט המייצג נקודה גיאוגרפית. נקודה יכולה להיות נקודת source או נקודת destination, וכן שמורים עבורה מה טווח האינדקסים בהם היא יכולה להופיע בוקטור מבלי לפגום בחוקיותו (בהתאם לאילוצי הבעיה – למשל, שנקודת יעד של נוסע לא תופיע לפני נקודת המוצא שלו במסלול).

- כאשר נדרשנו להגריל מספר מצב התחלתי, הגרלנו פרמוטציה על n הנוסעים (כך שתמיד בכל מצב התחלתי נקודת המוצא של כל נוסע סמוכה בוקטור לנקודת היעד שלו), וכך הבטחנו את חוקיות פתרונות אלו.

נקודה מרכזית שהתלבטנו בה היא כיצד לייצר שכנים עבור מצב קיים, באופן שיבטיח את חוקיותם. בחנו שלוש אפשרויות עיקריות:

- Insert and shift – עבור כל נקודה בוקטור, ניתן "להעבירה" לאינדקס אחר בוקטור, המצוי בטווח האינדקסים החוקיים השמורים עבור נקודה זו. במסגרת זאת יש לעדכן את טווחי האינדקסים המתאימים ליתר הנקודות בוקטור. כל החלפה בודדת שכזו מייצרת שכן אחד, וסך כל ההחלפות מייצרות סדר גודל של n^2 שכנים.
- Swap – עבור כל נקודה בוקטור, ניתן להחליפה עם נקודה אחרת בוקטור, בתנאי שאחרי ההחלפה כל נקודה תימצא בטווח האינדקסים החוקי עבורה.
- Swap adjacent – כמו הגישה הקודמת, אך במקום לבדוק את אפשרות ההחלפה לכל זוג נקודות, נבדוק זאת רק עבור כל זוג נקודות סמוכות. בכך קיים פוטנציאל של חיסכון בזמן יצירת השכנים בכל איטרציה (משום שבאופן זה יש סדר גודל לינארי, ולא ריבועי, של שכנים).

האפשרויות השונות עשויות להגדיר טופולוגיה שונה של מרחב החיפוש, ולכן חשבנו שנכון לבדוק כל אחת מהן ולראות האם אכן מתקבלות תוצאות שונות.

<u>Genetic algorithm</u>

בחנו גם את האפשרות לפתור את הבעיה בעזרת אלגוריתם גנטי. לשם כך, השתמשנו באלגוריתם כפי שנלמד בכיתה, בצורה הבאה כאשר ניתן לכייל את הפרמטרים בכל צורה:

- גודל אוכלוסייה: 100 מסלולים חוקיים (המוגרלים בהתחלה).
- הסתברות ליצירת ילד מהורה: 0.8
- הסתברות לעשות מוטציה על מסלול כלשהו:0.7
- Fitness function: המרחק הכולל של המסלול כאשר הוא מחולק בסכום מרחקי כלל המסלולים השמורים (לשם נרמול).
- Crossover: מימשנו שלושה סוגים שונים של פונקציות crossover (ראו פירוט בנספח):
  - Cyclic Crossover.
  - partially mapped crossover.
  - ordered mapped crossover.
- מוטציה: החלפת זוג נקודות אקראיות במסלול.

נבחין שעבור בעיית אינסוף הנוסעים רק ordered mapped crossover יוצר ילדים תקינים, ואילו עבור נוסע אחד כל הCrossover שומרים על חוקיות הילדים. לכן, עבור גרסת האינסוף השתמשנו רק באחת מהפונקציות הנ"ל.

תוצאות וניתוחן

בדקנו את התוצאות באמצעות הגרלת קלטים רבים עבור כל גודל קלט, והרצת כל אחד מהאלגוריתמים עליהם. באופן זה, קיבלנו את זמן הריצה הממוצע של כל אלגוריתם על כל גודל קלט שנבדק, וכן את איכות הפתרון הממוצעת שהתקבלה, וכך ניתן היה לנתח את התוצאות.

כאשר הרצנו את האלגוריתמים השונים, ראינו שבאופן יחסי לאחרים, האלגוריתם החמדן נותן תוצאות טובות. לכן, כדי לשפר את תוצאותיהם של שאר האלגוריתמים, כאשר ניתן היה הפכנו את הפתרון שנותן האלגוריתם החמדן למצב ההתחלתי שממנו מתחילים שאר האלגוריתמים. בגרפים שלהלן נראה כיצד שינוי זה משפיע על הביצועים.

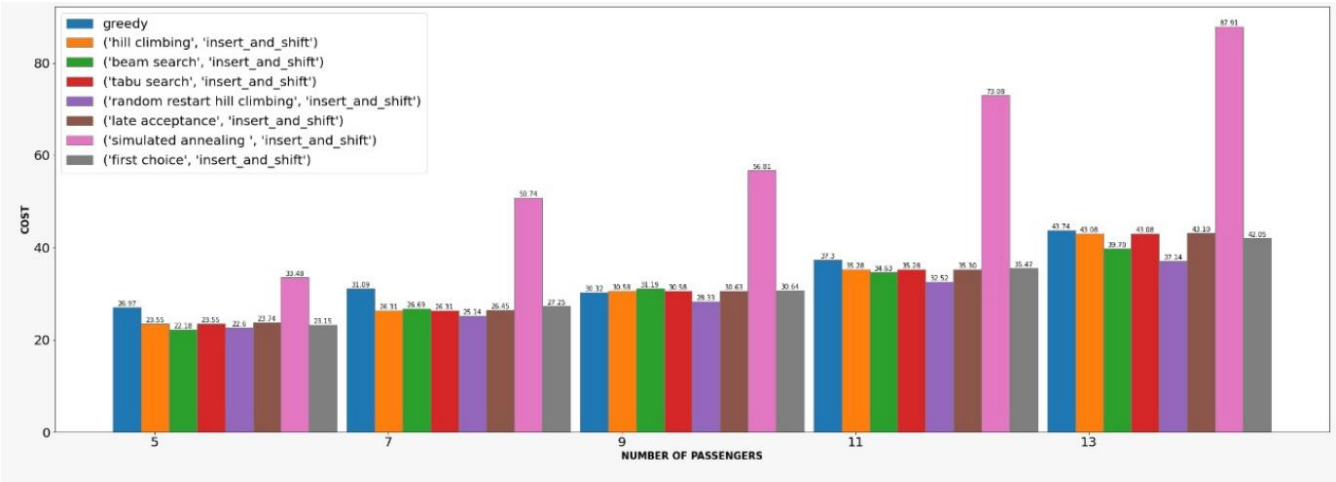
בחנו מספר היבטים של תוצאות הרצת האלגוריתמים השונים על הבעיה. נציג את התוצאות הקשורות בכל היבט, ואת הניתוח הנוגע בהן.

ללא התחלה מהפתרון החמדני

בתחילה, בדקנו את התוצאות של הרצת הווריאציות השונות של hill climbing כאשר המצב ההתחלתי הוא הקלט עצמו, בהתאם לסדר בקשות הנוסעים המופיעות בו.

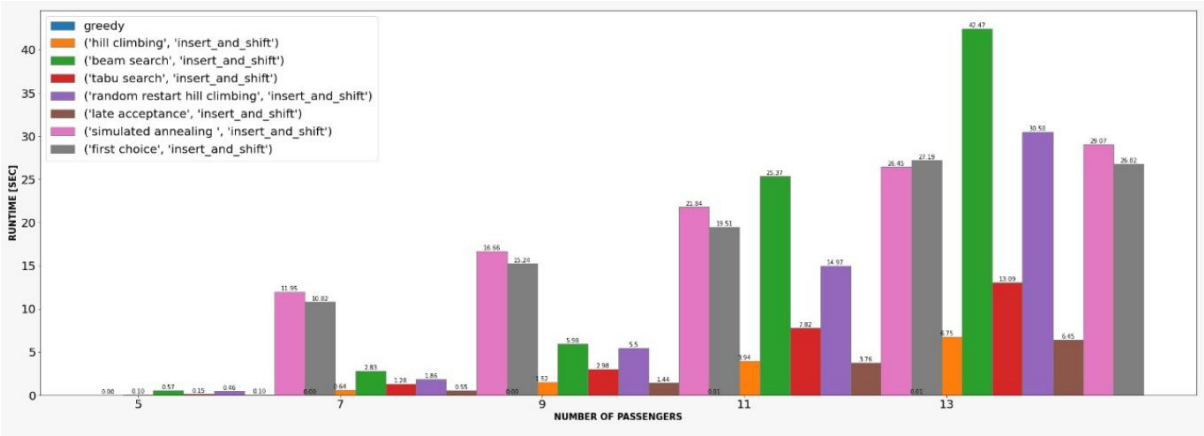
כאשר מתחילים מהקלט כפי שהוא, ולא מהפתרון החמדני, מתקבלות התוצאות הבאות (עבור קיבולת אינסופית):

מבחינת איכות הפתרון המתקבל:



ניתן לראות כי האלגוריתמים משיגים תוצאות דומות למדיי (למעט simulated annealing, ראו לקמן), אך random restart משיג את התוצאה הטובה ביותר.

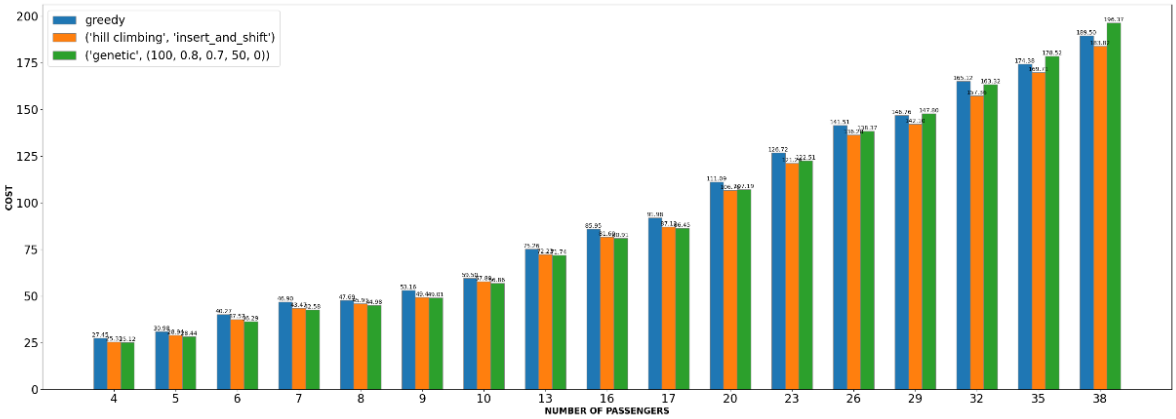
ומבחינת זמני הריצה:



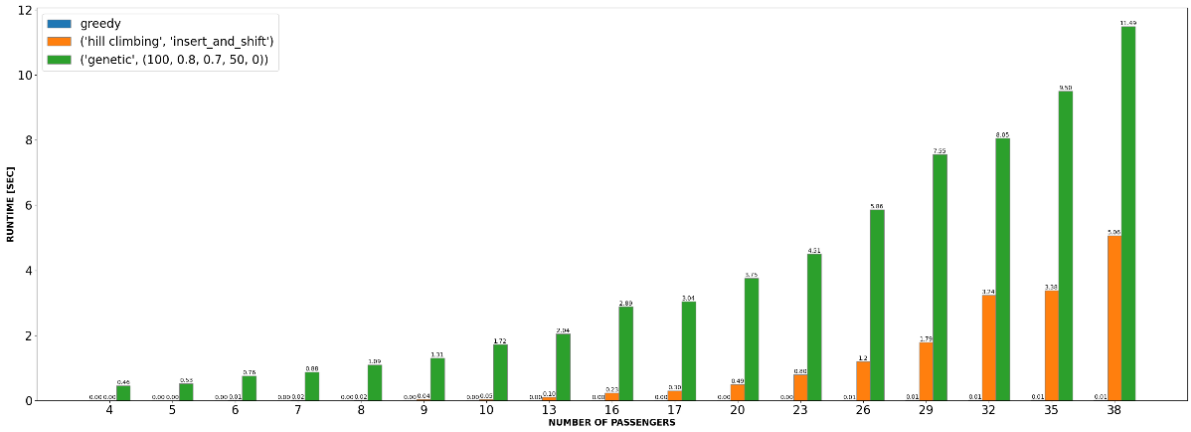
השוואה בין המייצגים העיקריים של שתי הגישות

נראה את ההבדל בין הביצועים של genetic algorithm לבין הביצועים של hill climbing (כאשר הוא מאותחל עם הפתרון החמדן), עם מתודת היורשים insert, ראשית עבור הבעיה של קיבולת 1.

מבחינת עלות המסלול המתקבל:



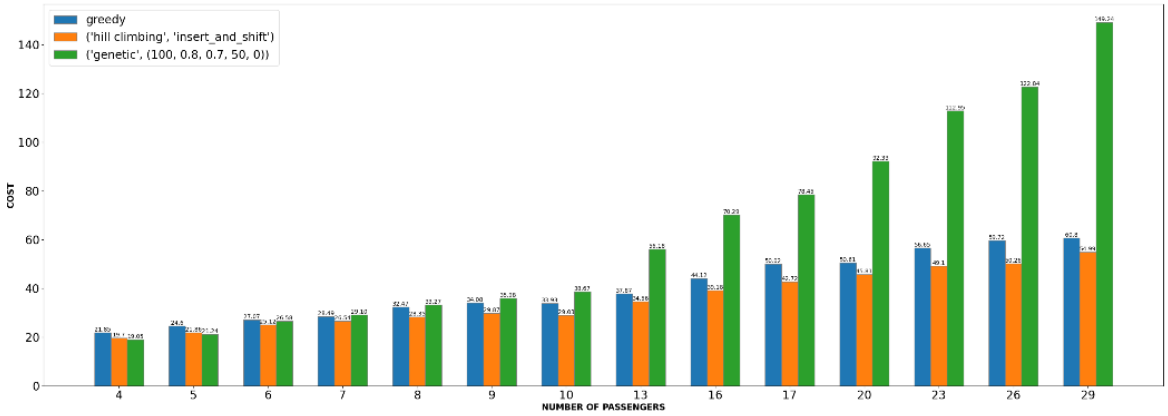
ומבחינת זמני הריצה:



ניתן לראות, שלhill climbing יש יתרון קל על פני genetic algorithm מבחינת איכות הפתרון המתקבל (הפרש של 7% בממוצע עבור קלט של 38 נוסעים), ויתרון משמעותי יותר כשמדובר בזמן הריצה (פי 12.5 עבור קלט של 16 נוסעים, ומעט יותר מפי 2 עבור קלט גדול יותר של 38 נוסעים). עם זאת, יתרון זה הולך ונשחק ככל שמגדילים את מספר הנוסעים: נראה שהסיבה לכך היא שכאשר מגדילים את מספר הנוסעים, העלות של יצירת היורשים (שהיא, כזכור, ריבועית בגודל הקלט) נותנת את אותותיה על זמני הריצה של hill climbing.

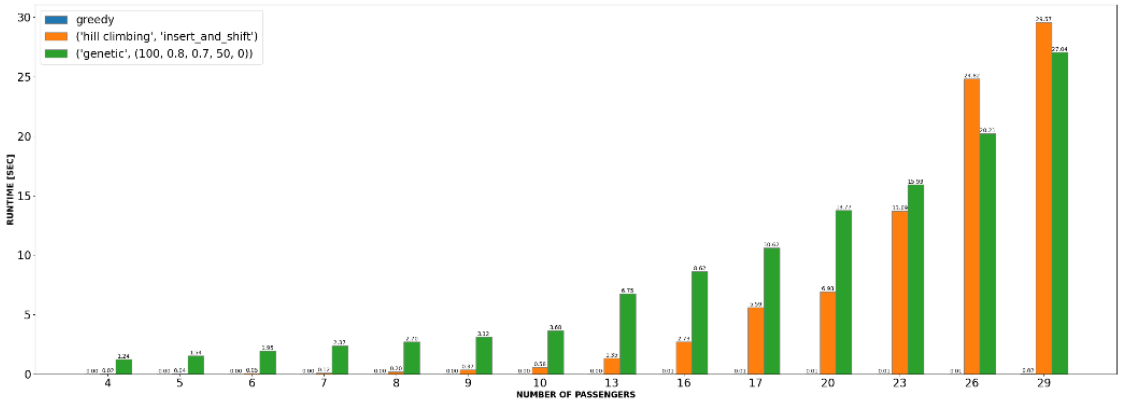
נראה גם את התוצאות עבור קיבולת אינסופית.

מבחינת איכות הפתרון המתקבל:



ניתן לראות שבגרסה זו האלגוריתם הגנטי משיג תוצאות משמעותית גרועות יותר ביחס לhill climbing, ופער זה גדל ככל שמגדילים את מספר הנוסעים. נראה שהקרוסובר הפשוט שבו משתמשים עבור מקרה מורכב זה אינו מצליח לייצר פתרונות שהולכים ומשתפרים עם הזמן.

מבחינת זמני הריצה:

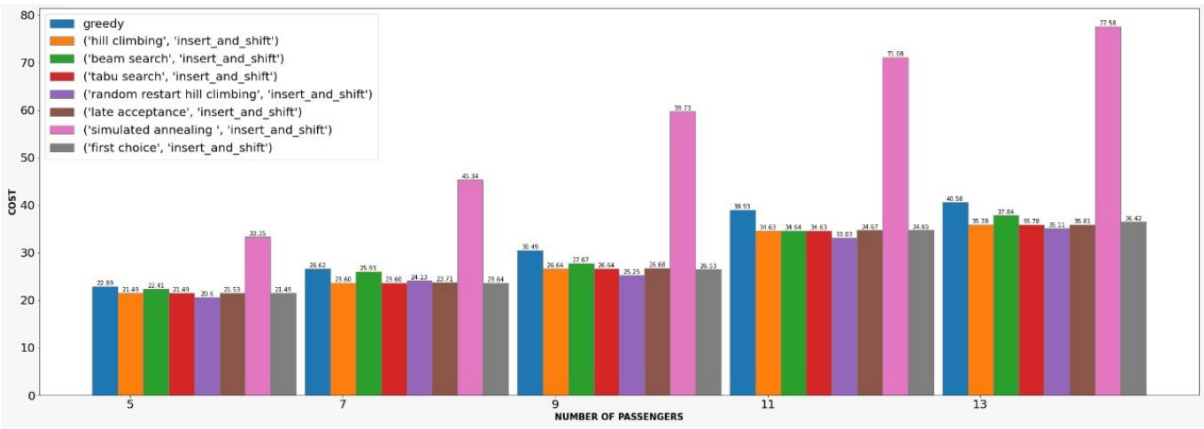


כאן, ניתן לראות שכאשר מגדילים את מספר הנוסעים, hill climbing נעשה יקר בזמני הריצה אפילו ביחס לgenetic algorithm. זאת, שכן עבור מספר גדול מספיק של נוסעים, פונקציית יצירת השכנים הינה יקרה בזמן הריצה (בהמשך נראה לאור התוצאות שניתן לבחור בפונקציה איכותית פחות, אך מהירה יותר).

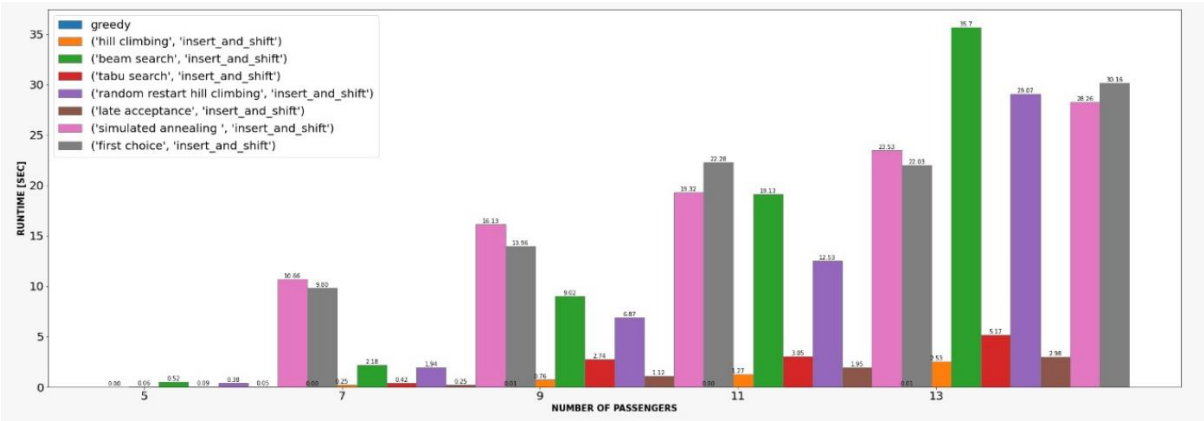
<u>השוואה בין הווריאציות השונות של hill climbing</u>

כעת נשווה בין הווריאציות השונות של אלגוריתם hill climbing שהזכרנו (**עם** אתחול עם הפתרון החמדן). ראשית, עבור קיבולת אינסופית:

מבחינת מחיר הפתרון המתקבל:

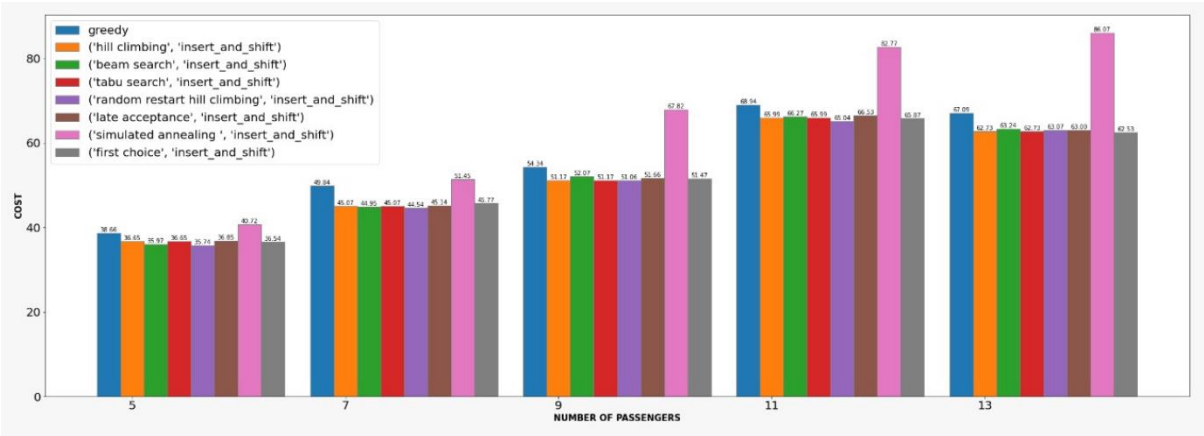


ומבחינת זמני הריצה:

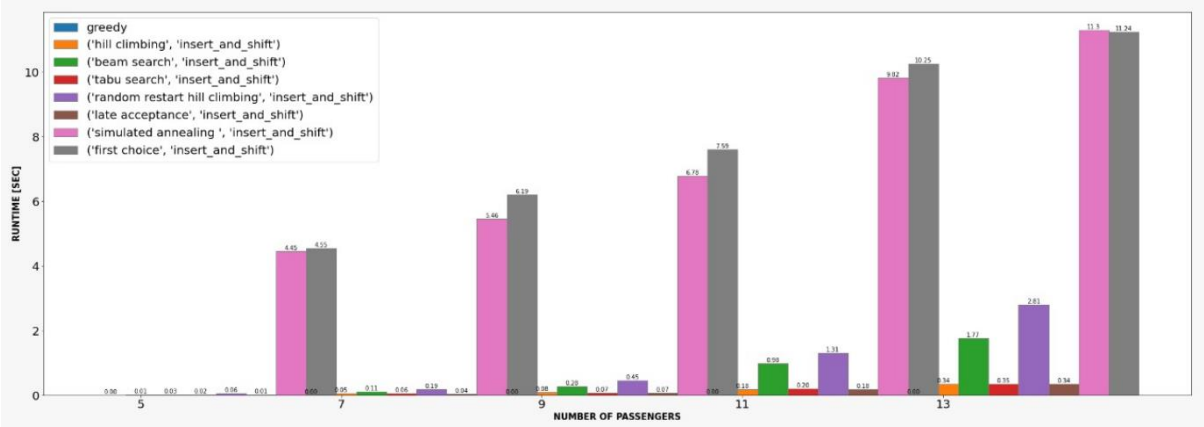


נביא גם את התוצאות עבור התרחיש של קיבולת 1:

מבחינת איכות הפתרון המתקבל:



ומבחינת זמני הריצה:



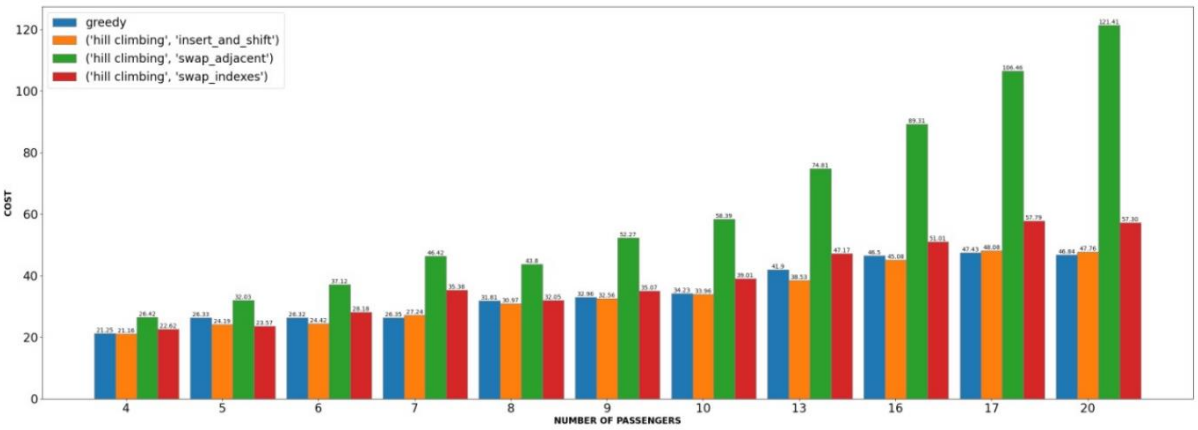
ניתן להבחין במספר נקודות:

- באופן כללי, התוצאות המתקבלות טובות יותר מאלו שקיבלנו לעיל, כאשר לא התחלנו מהפתרון החמדני כמצב התחלתי. כאשר לא התחלנו מהפתרון החמדני, אלגוריתמים כhill climbing הגיעו בממוצע לאיכותו של הפתרון החמדן. כאשר מתחילים מהפתרון החמדן, מצליחים לקבל שיפור של כ5% ביחס אליו. בנוסף, זמני הריצה של hill climbing משתפרים כאשר מתחילים מהפתרון החמדן (זה בולט עבור הממוצע עבור 13 נוסעים: שם זמני הריצה כמעט מכפילים את עצמם) – ככל הנראה מפאת נטייתו להיות בקרבת נקודת קיצון מקומית.

- הווריאציות השונות של hill climbing אינן נותנות תוצאות שונות במיוחד מבחינת איכות הפתרון המתקבל. Random restart ו tabu search נוטים לתת תוצאות מעט טובות יותר, אך לא באופן מאוד משמעותי (בנוסף, היתרון היחסי של random restart נשחק ככל שמגדילים את מספר הנוסעים). הן מצליחות לשפר את הפתרון שנותן החמדן בשיעור של כ5% בממוצע (עבור קיבולת אינסופית, ומעט פחות עבור הגרסה של הקיבולת האינסופית).
- חריג לכלל זה הוא אלגוריתם simulated annealing, שאינו מתכנס כיאות, ונותן באופן עקבי תוצאות גרועות ביחס לאלגוריתם החמדן. נציין כי בדקנו את ריצת אלגוריתם זה עבור קצבי קירור רבים, שונים ומגוונים, ועבור מספר שיטות קירור (קירור בקצב של טור גיאומטרי עם מספר קצבי התכנסות, וכן קירור בקצב של טור הרמוני), ובכולם התוצאה הייתה פחות טובה (לרוב משמעותית) מאשר תוצאת האלגוריתם החמדן. הטוב ביותר שהצלחנו להגיע אליו עם אלגוריתם זה היה כאשר הרצנו אותו בקצב קירור איטי ביותר. במקרה זה האלגוריתם רץ במשך הרבה מאוד זמן, ולבסוף הגיע לתוצאה שגבוהה רק במעט מזו שנותן החמדן, ועדיין לא שיפר.
- מבחינת זמני הריצה, ניכר בבירור כי האלגוריתמים שמתחזקים מספר חיפושים (כגון random restart, local beam) הם הגובים את זמן הריצה הגדול ביותר, בפער גדול, כפי שאכן ניתן היה לצפות. לעומתם, האלגוריתמים שמבוססים על hill climbing פשוט יותר מסיימים לרוץ במהירות. גם אלו, לוקים בזמן ריצה לא קטן במיוחד ככל שגדל הקלט, וזאת בשל עלות יצירת השכנים בכל איטרציה (כפי שנראה בהמשך, גם בכך ניתן לייצר tradeoff בין שיטות שונות ליצירת יורשים).
- בניגוד למה שמקובל לטעון ביחס לאלגוריתמי חיפוש לוקאלי באופן כללי, במקרה שלנו לא ניכר שיפור משמעותי כשהוספנו את ממד הרנדומיות לדרך הפתרון (הדוגמה הבולטת לכך היא simulated annealing, שכאמור נותן תוצאות לא טובות כאן).
- ניתן להבחין בהבדל ניכר בביצועים בין גרסת הקיבולת האינסופית לבין גרסת קיבולת 1. בקיבולת האינסופית, זמני הריצה משמעותית ארוכים יותר (כפי 3, עבור ממוצע על קלט בגודל 13 נוסעים). מסתבר שהסיבה לכך היא שעלות יצירת היורשים יקרה יותר בגרסה זו (שכן יש קושי בהבטחת חוקיות הפתרונות המתקבלים). מבחינת איכות הפתרונות, אין זה פלא שבגרסת הקיבולת האינסופית הפתרונות ככלל טובים יותר, שכן לרכב יש יותר גמישות בבחירת המסלול (אין הכרחי לנסוע ישירות מנקודת מוצא של נוסע ספציפי לנקודת הסיום שלו).

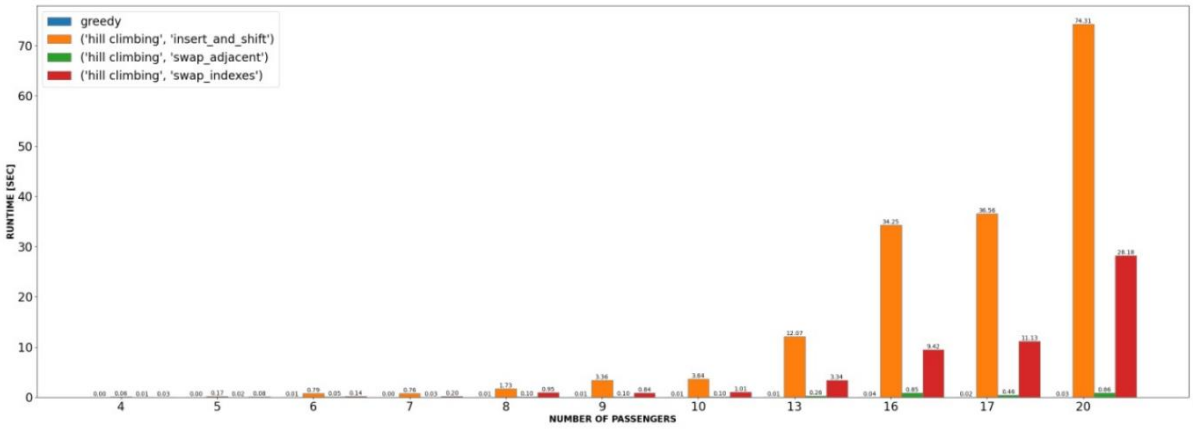
<u>השיטות השונות ליצירת יורשים – הטופולוגיה של מרחב החיפוש</u>

נמחיש את ההבדל בביצועים בין השיטות השונות ליצירת יורשים. הרצנו את השיטות השונות עם אלגוריתם hill climbing, עבור הבעיה שבה קיבולת הרכב היא אינסופית (התוצאות עבור הווריאציה השנייה דומות בהקשר הזה, וכדי להימנע מאריכות יתרה לא נציג אותן כאן), וקיבלנו את התוצאות הבאות (כממוצע על פני מספר הרצות עבור קלטים רנדומיים).



ניתן לראות שבאופן עקבי, שיטת swap_adjacent נותנת תוצאות הרבה פחות טובות משתי השיטות האחרות. עבור בעיות של 20 נוסעים, מתקבל הפרש בסדר גודל של יותר מ100%. מבין שלוש השיטות, התוצאות הטובות ביותר הן של שיטת הinsert (בפער של כ17% משיטת swap_indexes, עבור עשרים נוסעים).

מאידך, כאשר בוחנים את זמני הריצה, מתקבל:

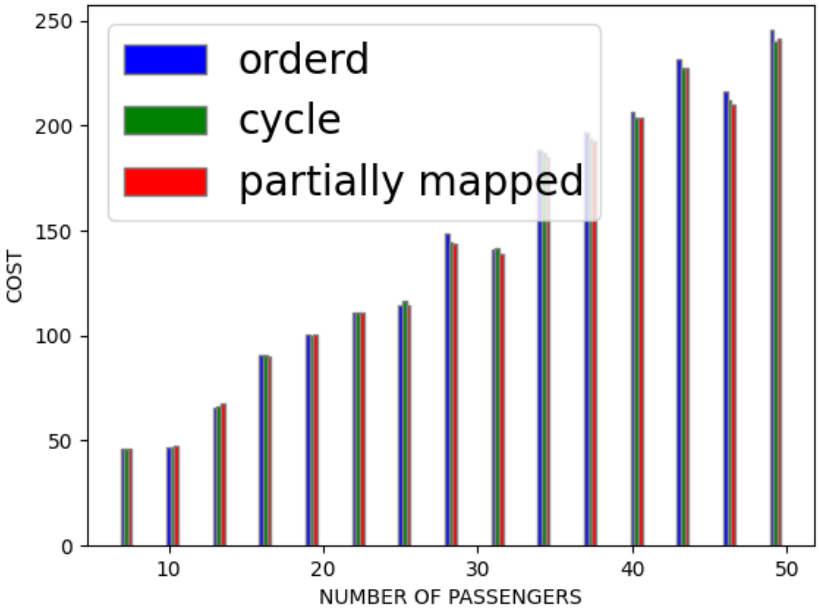


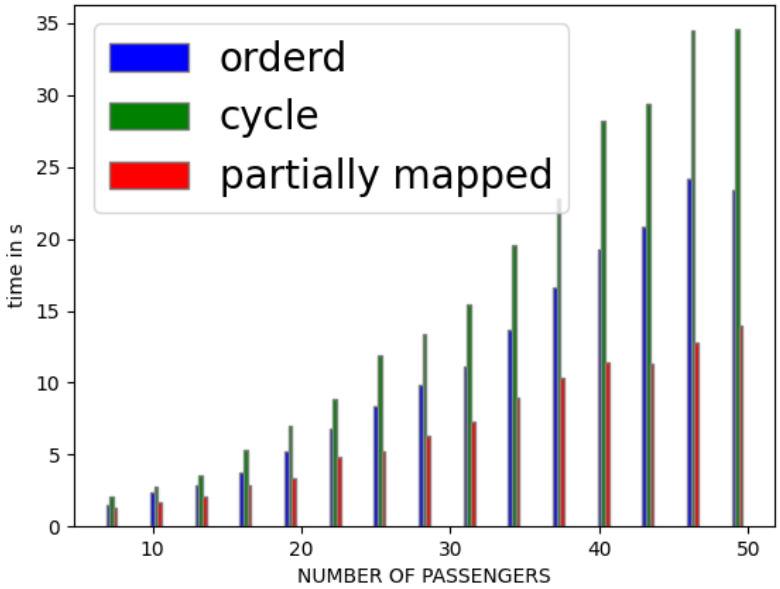
כאן ניתן לראות את הצד השני של המטבע: מאחר ובשיטת swap_adjacent בוחנים פחות שכנים – זמני הריצה קצרים הרבה יותר מאשר בשתי השיטות האחרות. השיטה שלה הביצועים הטובים ביותר מבחינת המחיר, insert, נותנת את הביצועים הכי פחות טובים כאשר בוחנים יעילות זמן ריצה.

כלומר, כפי שניתן היה לצפות, פתיחת פחות שכנים משפרת את זמני הריצה מצד אחד, אך מעלה את הסיכוי להיתקלות מוקדמת בנקודת קיצון מקומית שתוביל לביצועים פחות טובים, מצד שני. בנוסף, שיטת הinsert, שדורשת עדכון של ערכים בחלקים נרחבים בווקטור (ולא רק בשתי הנקודות שמחליפים, בשיטת הswap) אמנם נותנת ביצועים טובים יותר, אך עולה יותר בזמן הריצה.

<u>השיטות השונות לפונקציית crossover</u>

ראינו כמה דרכים ליצור crossover באלגוריתמיים הגנטיים, אבל כפי הנראה מהגרף, אין שיפור משמעותי בניהם, חוץ מהפרש בזמני ריצתם. חשבנו שהסיבה לכך היא בגלל שיש הנחה סמויה שעומדת מאחורי אלגוריתמים גנטיים ולפיה הורים טובים ייצרו ילדים טובים. אצלנו, אין הדבר נכון בהכרח, שכן שילוב של שני מסלולים קצרים עלול להוביל ליצירת מסלול ארוך, ולכן אין שיפור משמעותי בסוגי יצירת הילדים.





סיכום

בפרויקט זה הראינו כיצד שיטות שונות של חיפוש לוקאלי ואלגוריתם גנטי מצליחות להשיג תוצאות סבירות עבור קלטים לא גדולים מדי בבעיית איסוף הנוסעים. אמנם, השיפור המתקבל ביחס לאלגוריתם חמדני קלאסי שניתן להפעיל על הבעיה אינו גדול במיוחד, אך בכל זאת הוא קיים. יעילות האלגוריתמים השונים, מבחינת זמני ריצה, מושפעת לא מעט מאסטרטגיית יצירת היורשים הננקטת.

קיימים מספר אתגרים להמשך מחקר בנושא: ראשית, ניתן לבחון את התרחיש בו קיבולת הרכב הינה **סופית** וגדולה מאחד. מקרה זה מאתגר יותר ומהווה תרחיש ביניים בין שני המקרים שחקרנו. בנוסף, יש מקום לבחון את הרחבת הבעיה למספר רכבים ולא רק לרכב אחד. אפשרות נוספת למחקר עתידי היא שינוי פונקציית העלות שלפיה מודדים את איכות הפתרון. למשל, ייתכן שעדיף לבדוק לא את המרחק הכולל שעברו הרכבים, אלא את הזמן הכולל שחיכו הנוסעים, וכן על זו הדרך.

נספח – פירוט פונקציות הcrossover בהן השתמשנו

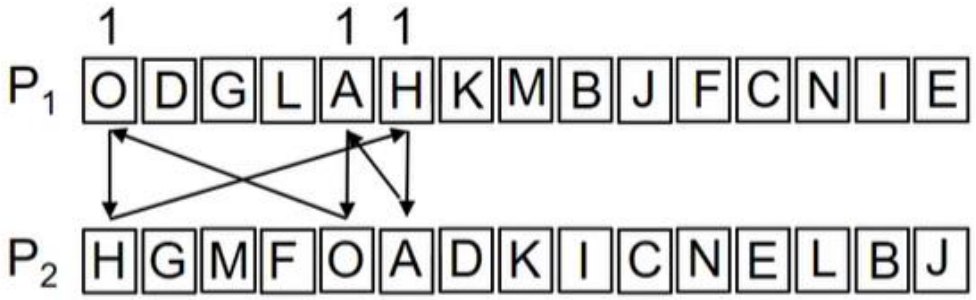
נפרט את סוגי פונקציות הCrossover שבהן השתמשנו (פונקציות אלו הן בשימוש עבור בעיית TSP)[1]:

Cyclic Crossover

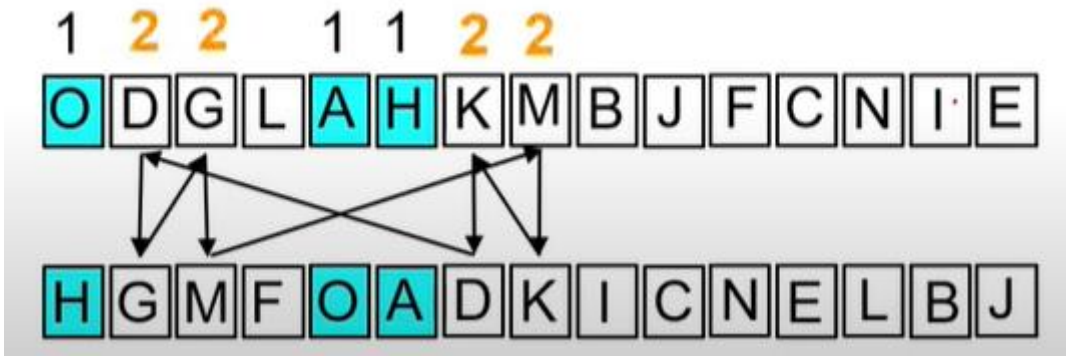
צעד 1 - זיהוי המעגלים:

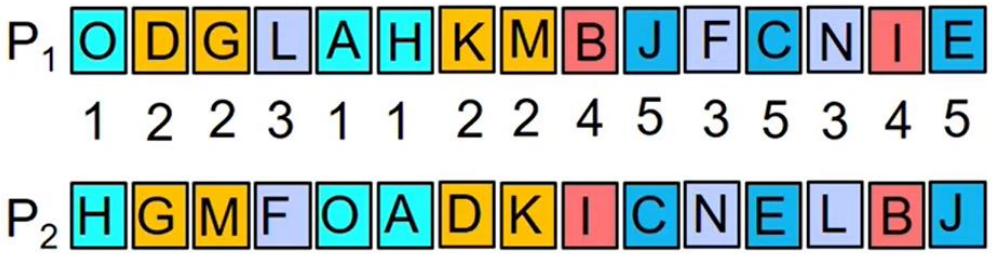
i. התחל מאינדקס 1 בהורה הראשון.
ii. ראה מה הנקודה השמורה באינדקס 1 בהורה השני.
iii. מצא את הנקודה משלב 2 באינדקס i כלשהו בהורה הראשון.
iv. מצא את הנקודה משלב 3 באינדקס j כלשהו בהורה השני.
v. חזור על התהליך המתואר בשני השלבים האחרונים עד שייווצר מעגל.

[1] https://www.youtube.com/watch?v=3lc_Fcga5z8&ab_channel=NPTEL-NOCIITM



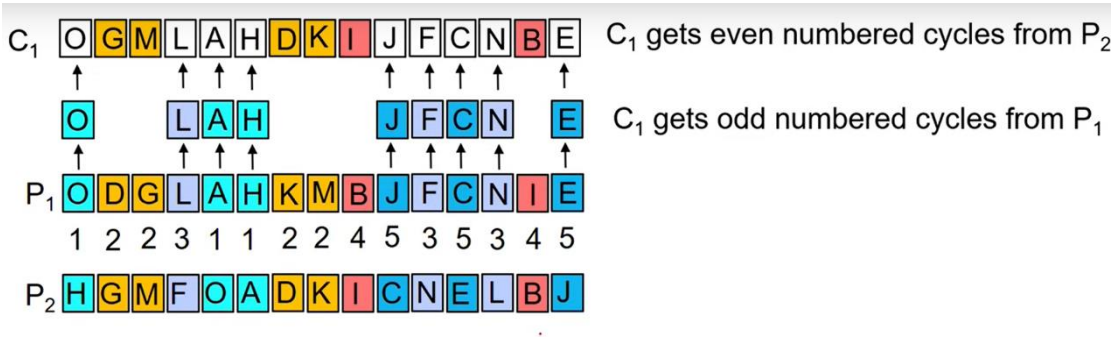
צעד שני: עוברים על הווקטור של ההורה הראשון. בכל פעם שהגענו לאינדקס של נקודה שלא משתתפת במעגל שמצאנו, מריצים עליה את האלגוריתם מהשלב הקודם (כאשר משנים את נקודת ההתחלה בהתאם לנקודה בהורה הראשון שאליה הגענו). באופן זה, אפשר לוודא שכל איבר בווקטור יהיה שייך רק פעם אחת למעגל.





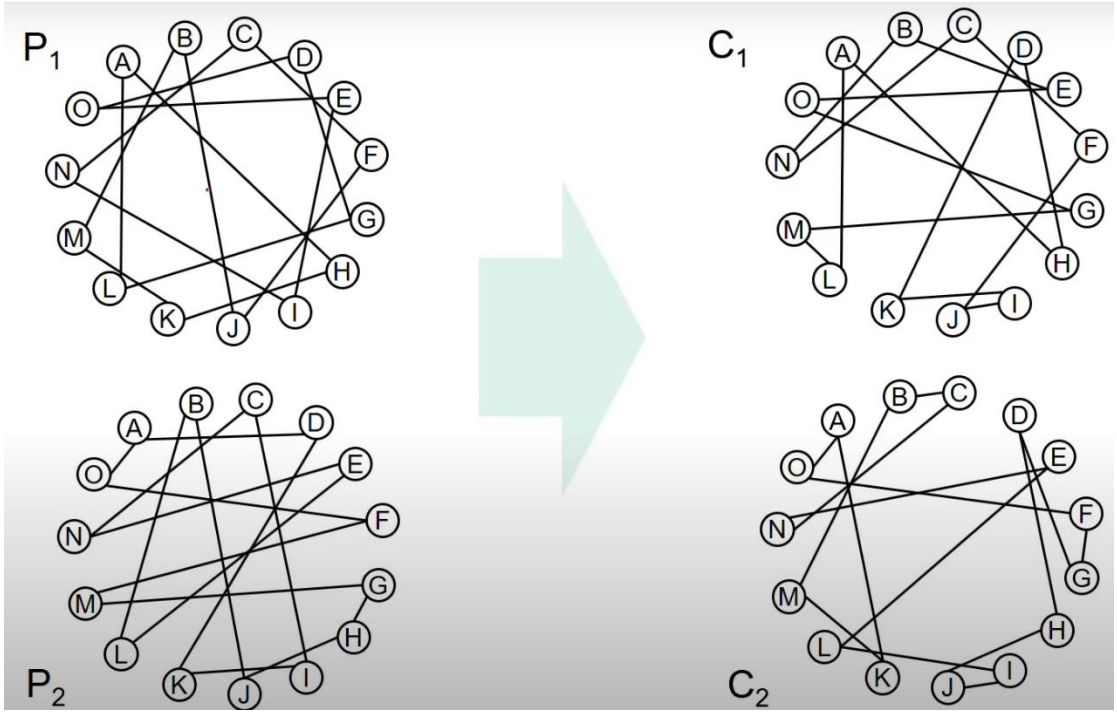
צעד שלישי - יצירת הילד:

את הילד יוצרים מהמעגלים שהתקבלו באופן הבא:



לאחר מכן, יוצרים ילד נוסף באמצעות הרצת אלגוריתם זהה, כאשר הופכים את תפקידיהם של שני ההורים.

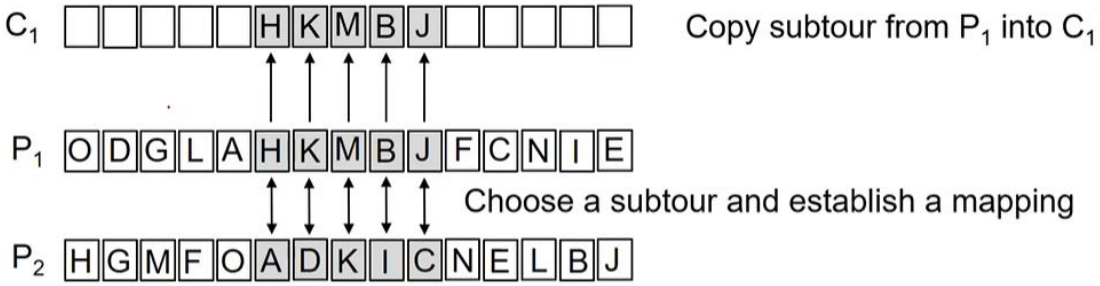
לאחר שתי הלידות הילדים יראו בצורה הבאה:



partially mapped crossover

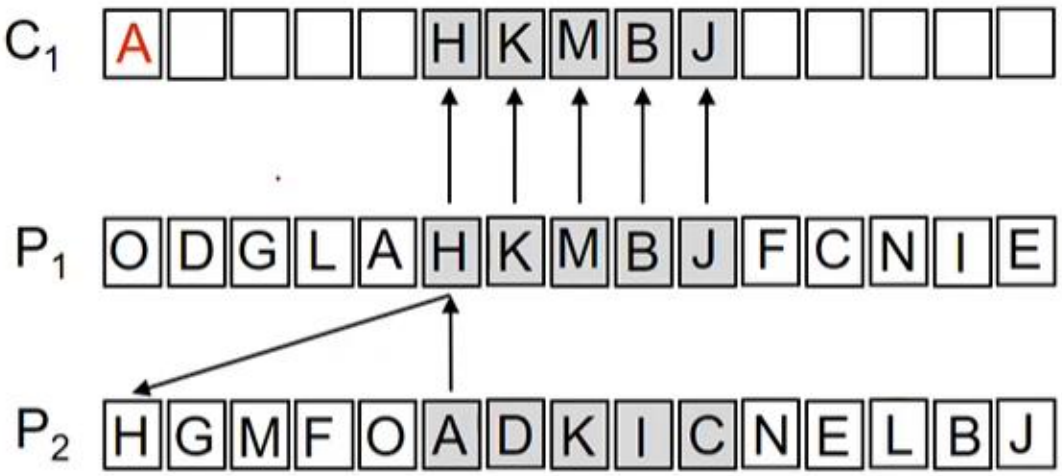
צעד ראשון:

בוחרים אזור כלשהו מהורה 1 ויוצרים מיפוי לאותו אזור בהורה 2 בצורה הבאה:

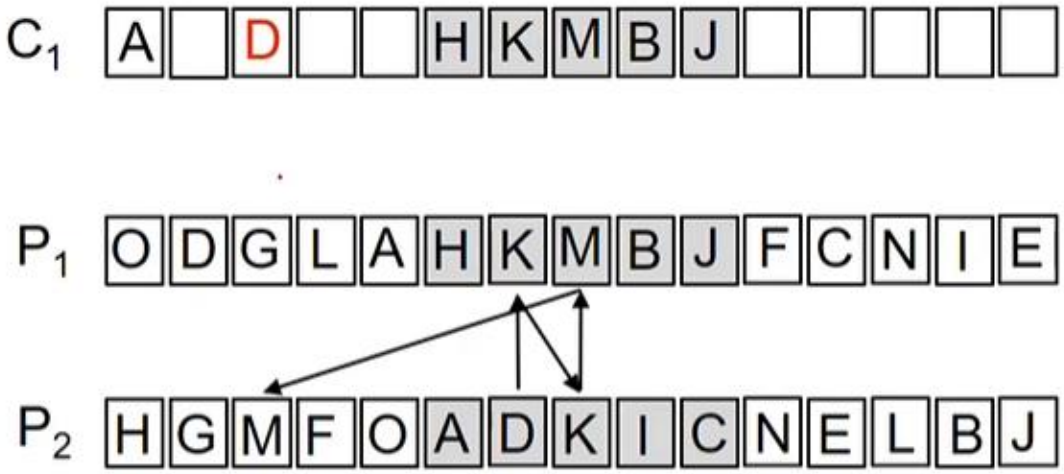


צעד שני:

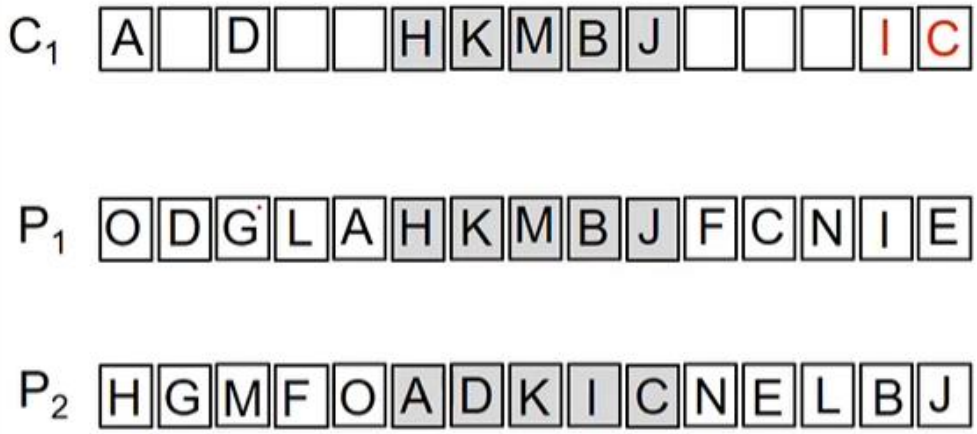
מעתיקים את האזור מהורה 1 לילד, ואז מעתיקים את כל מה שנשאר מההורה השני. כדי שהילד המתקבל יהיה חוקי, נעשה זאת בצורה הבאה, כמתואר בתמונה:



i. למעשה, עוברים על הנקודות המתאימות מהאזור הנ"ל בהורה השני. בכל פעם, בוחרים את האינדקס המתאים להכנסת כל נקודה ונקודה בהתאם לאלגוריתם המעגלי שתואר לעיל, כפי שרואים בתמונה.



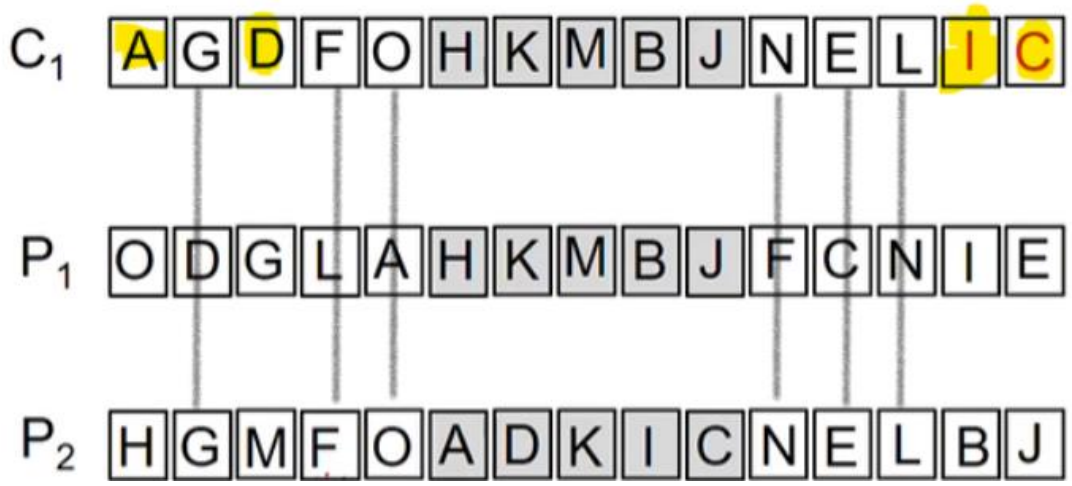
ii. בשביל מיקומים שמופיעים בשני האיזורים של ההורים - נימנע מלהעתיק אותם שוב, ובמקום זאת פשוט נמשיך את התהליך עד שכל הילדים מההורה השני גם יועתקו ללא כפילויות.



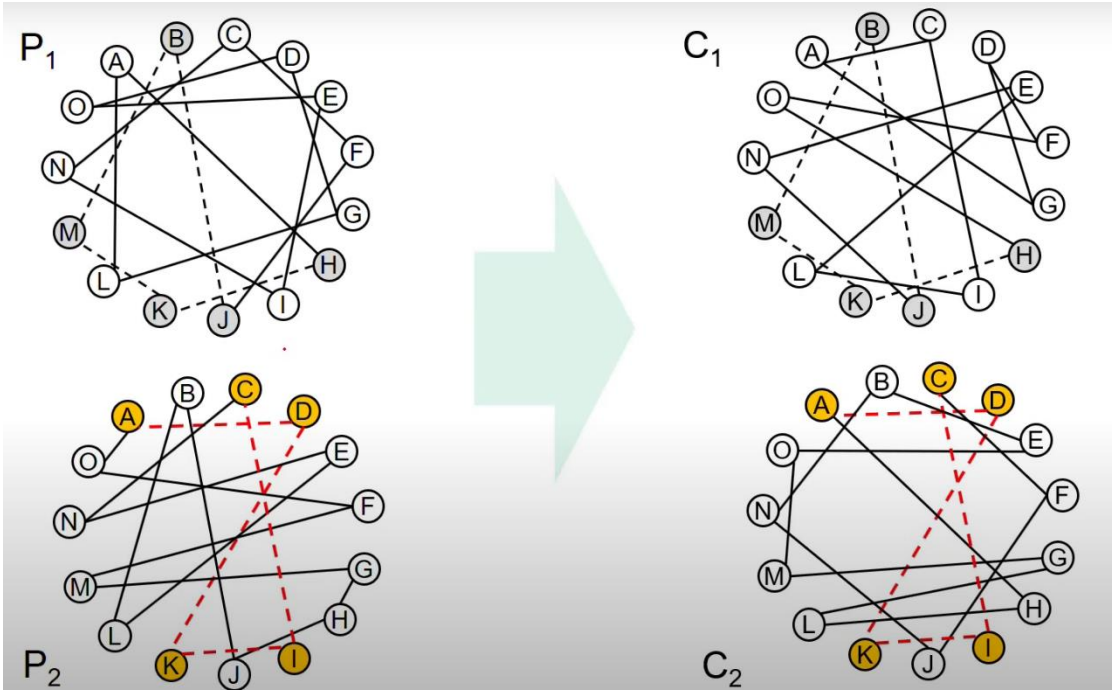
צעד שלישי:

אחרי שסיימנו להעתיק את כל המיקומים מההורה השני מאותו אזור של ההורה הראשון, נעתיק את כל המיקומים שנשארו בשביל למלא את הילד מההורה השני, בסדר היחסי בו הם נמצאים בהורה

השני.



הילד ייראה כך:



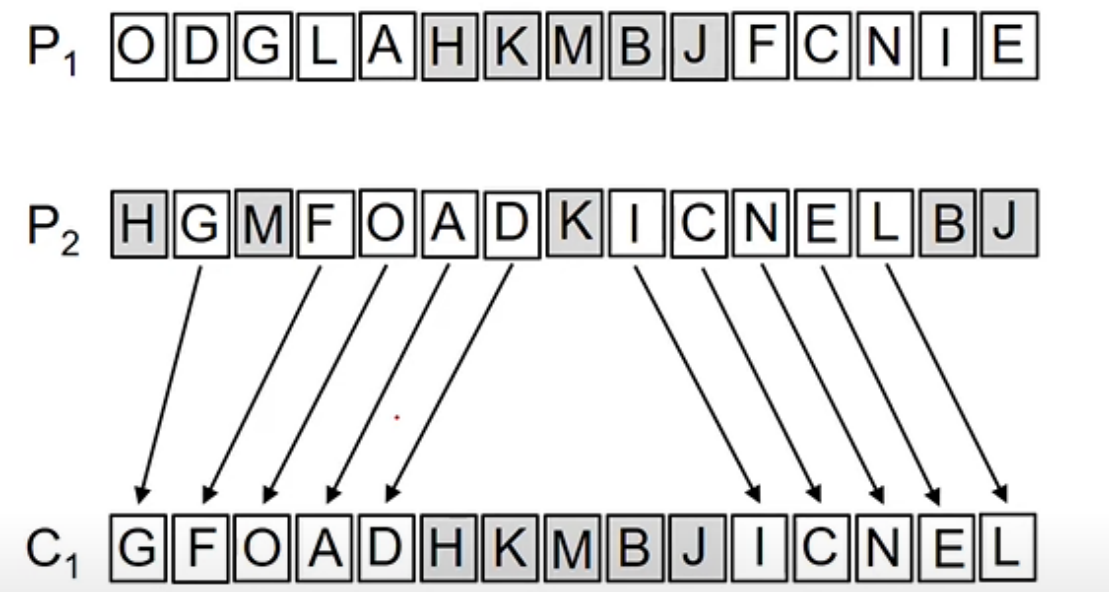
ordered mapped crossover

צעד ראשון:

בוחרים איזור כלשהו מהורה אחד ומעתיקים אותו לילד (באותם מיקומים יחסיים בווקטור).

צעד שני:

מעתיקים את כל שאר הצאצאים מההורה השני בסדר היחסי שבו הם נמצאים בו (תוך הימנעות מהכנסת נקודה זהה מספר פעמים לילד, כמתואר בתמונה).



בבעיה שלנו, האזור שבוחרים מההורה חייב להתחיל **בתחילת** הווקטור, בשביל שהילד שיתקבל יהיה תקין (אחרת, ייתכן שנקבל מסלול שבו מורידים נוסעים לפני שמעלים אותם).

ביבליוגרפיה

מקורות בדבר אלגוריתם גנטי (כולל התמונות להמחשה):

https://towardsdatascience.com/evolution-of-a-salesman-a-complete-genetic-algorithm-tutorial-for-python-6fe5d2b3ca35

https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/

https://www.youtube.com/watch?v=3lc_Fcga5z8&ab_channel=NPTEL-NOCIITM

מקורות בדבר local search:

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

https://github.com/perrygeo/simanneal

https://www.sciencedirect.com/science/article/pii/S0377221716305495