



שוברים את קטאן

מאת גל ראובני

הקדמה

היום אנחנו הולכים לעשות משהו קצת שונה. אנחנו עומדים לתקוף משחק קופסא ולחפש בו חולשות. משחק הקופסא שניגע בו הוא בסך הכל לוח עם משושים וכמה כלים מפלסטיק... אז מה כל זה ולסייבר?

ראשית, חשוב להבהיר כמה דברים:

בעולמות מדעי המחשב ותורת הגרפים ככלל אנחנו יודעים להמיר בעיות מהחיים האמיתיים למודל של גרפים ועל גביו לפתור בעיות מורכבות. זה בדיוק מה שאנחנו נעשה כאן.

כל אחד שיסיים לקרוא את המאמר הזה יוכל לממש את הפתרון שאני מציע כאן באופן אוניברסאלי כמעין אלגוריתם שתמיד יעבוד לפתרון של בעיות שונות תלויות בהגדרה. בהמשך גם תבינו שהאלגוריתם שאציע הוא מתוכנן בצורה סופר-מוקפדת והוא אינו משתמש כלל ב-Brute Force או ב-AI, מה שמניב לו יעילות חישובית גבוהה במיוחד וביצועים עקביים, ללא תלות במורכבות המפה.

בהינתן מפת משחק נתונה, וטבלת ניצחונות (נרחיב על מה זה בהמשך), האלגוריתם שלי יודע להישאר דבוק על האסטרטגיה העדיפה ביותר לפי ניתוח המפה לאורך כל המשחק מההתחלה ועד הסוף. ובכך באופן הסתברותי לזכות בניצחון ברוב הפעמים.

המשחק "מתיישבי קטאן" הוא אחד מהמשחקים הנחקרים ביותר באקדמיה, במיוחד בתחומי תורת המשחקים, כלכלה ובעיקר היום ב-AI, וכל זאת בעיקר בשל השילוב הייחודי של ניהול משאבים מוגבלים, משא ומתן והסתברות. המשחק נחקר מספר פעמים באוניברסיטאות.

לשם המחשה הנה כמה מחקרים מוכרים:

קישור	תחום מחקר	אוניברסיטה	כותבים	שם המאמר (אנגלית)
קישור	כלכלה ותורת המשחקים	Syracuse University	Dmitriy Ioselevich	The Settlers of Catan and a Study of Trade in Non-cooperative Games
קישור	ניהול משא ומתן ואסטרטגיה	University of Edinburgh / Heriot-Watt University	Markus Guhe, Alex Lascarides, Kevin O'Connor, Verena Rieser	Effects of Belief and Memory on Strategic Negotiation
קישור	למידת מכונה / Deep reinforced learning	Stanford University	Chris C. Kim & Aaron Y. Li	Re-L Catan: Evaluation of Deep Reinforcement Learning for Resource Management Under Competitive and Uncertain Environments

הקדמה (מי שמכיר את המשחק - לדלג ל"ניסוח הבעיה")

למי שלא יודע, "קטאן" הוא משחק קופסא מוכר מאוד שהומצא בשנת 1995. מעבר למיתוסים על "פירוק החברויות" או הריבים העקשניים שהמשחק יוצר בקרב השחקנים (-), המשחק משלב אסטרטגיה רבה וטומן בחובו בעיה מתמטית מרתקת שמושכת אותנו לעולמות של גרפים ואלגוריתמים. את הבעיה שנציג פה ניתן להמיר לבעיות מכל תחומי החיים וזוהי המחשה נהדרת. אז קדימה! בואו נצלול עמוק פנימה.

היום אנו ניקח בעיה פשוטה כמו הצבת הכלים על המפה ובין כיצד ניתן לקבל פתרון פשוט ומהיר ללא שימוש ב-[Brute force](#). (למי שבקיא במשחק מומלץ לדלג ל-"ניסוח הבעיה"):

הסבר קליל על המפה:

המפה עצמה מורכבת ממשושים שמונחים זה לצד זה באופן אקראי לחלוטין ויוצרים משושה גדול יותר. לכל משושה יש סוג משאב שאותו השחקנים מקבלים: חיטה, כבשים, בנייה (חמר), עץ, אבן ומדבר. משושה המדבר הוא שולי כיוון שדרכו אף אחד מהשחקנים לא זוכה במשאב (לא נתייחס כרגע למשושה זה לבעיה שלנו).

בנוסף לכך, על כל משושה מונח מספר שמייצג את המספרים שיכולים לצאת בקוביות (2 קוביות בסה"כ), כלומר הקומבינציות של שתי קוביות יחד (המס' 7 אינו רלוונטי ונתייחס אליו בהמשך). בכל הטלה של הקובייה, אלו שיש להם יישוב או עיר שמוצבים בסמוך למשושה שעליו מוצב המספר שיצא בקובייה, זוכים בקלף מכל משאב של משושה זה (במקרה של עיר מקבלים פעמיים כל משאב). לדוגמא: אם היישוב שלי ממוקם ליד 8 עץ, 5 חיטה ו-8 כבשה, אז ברגע שיצא המס' 8 בקובייה, אני אזכה בשתי קלפים מסוג כבשה ועץ. המטרה - להשיג 10 נקודות ניצחון במשחק.

אז כיצד ניתן לעשות זאת?

מקור הנקודות	נק' ניצחון	הערה
יישוב 	1	כל יישוב בנוי מעניק נקודת ניצחון אחת
עיר 	2	כל עיר בנויה מעניקה שתי נקודות ניצחון
הדרך הארוכה ביותר 	2	בנוסף לשחקן שמחזיק בתואר
הצבא הגדול ביותר 	2	בנוסף לשחקן שמחזיק בתואר
קלף ניצחון 	1	כל קלף ניצחון מחפסת הפיתוח מעניק נקודה אחת

ע"י בניית יישובים, ערים, דרך רציפה הכי ארוכה (כל דרך מחברת בין כל ישוב חדש שנרצה לבנות), בניית הצבא הגדול ביותר וקלפי פיתוח. כל אחד מהם מקנה בונוס של נקודות ניצחון. להלן טבלה המפרטת כמה נק' ניצחון כל אחד מהם מקנה:

למרות הפשטות שבדבר ישנן כמה נקודות שוני - והן קלפי הפיתוח (קלף ניצחון) והצבא הגדול ביותר:



קלף פיתוח הוא קלף מזל, ניתן לקנות אותו ע"י משאבים מסוג: כבשה, חיטה ואבן. מדובר בקלף שמבחינה סטטיסטית יכול לזכות את השחקן בדברים הבאים (כתובים לפי אחוזים וכמות):

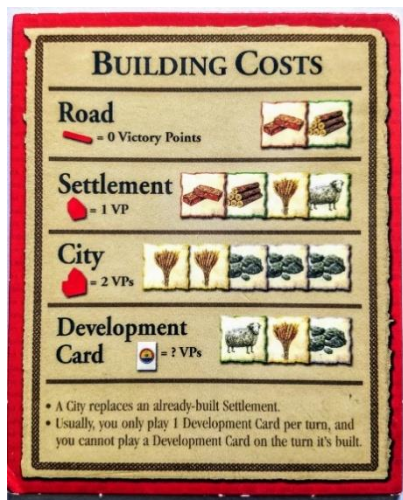
- 56% סיכוי לקבלת קלף אביר (הסבר בהמשך)

- 20% סיכוי לקבלת קלף שמעניק נקודת ניצחון אחת
 - 8% סיכוי לקבלת קלף הצבת שני דרכים לבחירת השחקן בחינם
 - 8% סיכוי לקבלת קלף שמאפשר לקחת שני משאבים לבחירת השחקן
 - 8% סיכוי לקבלת קלף המונופול - מאפשר לגנוב משאב לבחירתך מכל שאר השחקנים.
- אז מה באמת קורה כשיוצא 7 בקובייה? כאשר שחקן הטיל 7 בקוביות, כל אחד שיש לו מעל ל-7 קלפים ביד מוכרח לזרוק חצי ממה שיש לו באותו הרגע לקופה. בנוסף, השחקן יכול לבחור להציב את השודד שבתחילת המשחק ממקום במדבר באיזה משושה שהוא רוצה. השודד למעשה מונע קבלת משאבים מהמשושה שעליו הוא נמצא ברגע נתון, השחקן ממקמו במשושה לבחירתו וגונב קלף אחד אקראי משחקן שיש לו התיישבות על אחד מקודקודי המשושה. ומה לגבי מי שמחזיק בקלף ה-"אביר"? ובכן לו יש את האפשרות להזיז שודד בתורו גם כאשר לא יצא 7 בקובייה.

שחקן המחזיק במעל ל-3 קלפי אביר מחזיק בתואר "הצבא הגדול ביותר" שמזכה אותו ב-2 נקודות ניצחון נוספות.

שחקן המחזיק בדרך הכי ארוכה (מעל ל-5 דרכים סמוכות ברצף), וזו שהכי ארוכה כעת ביחס לשאר על הלוח - מזוכה ב-2 נקודות ניצחון נוספות.

אז כעת לאחר שאנו יודעים את המידע הבסיסי הזה, הבא נתחיל:



בתחילת המשחק, לאחר הגרלה, כל שחקן בתורו מציב ישוב ודרך על מיקום לבחירתו על המפה. כך, עוברים באופן הזה על כל השחקנים עד שכולם הציבו ישוב ודרך במיקום לבחירתם על המפה. בסיבוב השני, מציבים עוד ישוב ודרך במקום מסוים לבחירת השחקנים. שלב ההצבה נסגר כאשר לכל אחד מהמשתתפים יש כ-2 יישובים ו-2 דרכים מפוזרים על המפה במקומות אסטרטגיים.

הבחירה הזו היא **קריטית**, כיוון שבהמשך המשחק השחקן יכול להרחיב את האימפריה שלו רק בסמוך ליישובים שבהם כבר הציב אותם בהתחלה. למעשה, מותר לבנות יישוב נוסף רק במרחק מינימום של 2 דרכים מכל יישוב אחר.

חשוב לציין שיש חשיבה רבה מאחורי בחירת המיקומים הראשוניים של היישובים. היא צריכה להתחשב בכמה אלמנטים: המצב הנתון של המפה, הצבות של שחקנים אחרים, המספרים על גבי המשושים, ההסתברות לקבל אותם ו-סוגי המשאב במשושים לתכנון האסטרטגיה הארוכת טווח במשחק.

בכל מקרה, האסטרטגיה שלנו היא לשאוף ל-10 נקודות ניצחון, היא תלויה במצב הנתון של המפה עליה אנו משחקים. ולאור כל זאת אנו נרצה לפתור את הבעיה הבאה.

ניסוח הבעיה

כשנהגתי לשחק עם חבריי במשחק קטאן, תמיד מצאתי את עצמי מסתכל על הלוח בתחילת המשחק וחושב... "היכן אציב את היישובים"? המצב הוא סופר מורכב! יושבים כ-ארבעה אנשים מסביב ללוח וכל אחד צריך לבחור את המיקום האופטימלי להציב בו את היישובים. תמיד הדחימה אותי כמות ההשפעה שיכולה להיות רק לבחירה אחת במשחק ועד כמה היא עשויה להכריע. המהלכים ששחקן יכול לעשות בהמשך בעקבות הבחירה הראשונה שלו עלולים לעלות לו במשחק.

אז יש שיגידו שלא תמיד זה נכון ושלפעמים אפשר פשוט להיות גאון משא ומתן פסיכולוגי שיודע לנתח כל צעד וכל תגובה, ואז, איכשהו לרכוב על הגל ולנסות להוציא את עצמך מסיטואציות שונות ומורכבות ולנסות ולנצח בהן. אז אולי זה נכון, וכנראה שבחלק מפעמים כן. אך זה לא מה שאעסוק בו כאן.

את הבעיה של ההתבוננות בלוח, חיפוש אחר שטחים מניבים עם מספרי קובייה טובים ומשאבים חיוניים, ודרך זאת, תכנון האסטרטגיה להמשך המשחק - היא זו שאפתור. אז אם כן:

בהינתן מפת משחק נתונה, נרצה לקבוע היכן הוא המיקום האסטרטגי ביותר להציב את שני היישובים והדרכים הראשונים תחילה במשחק. מיקום זה יהיה האופטימלי ביותר בכך שיעזור לנו להגיע ל-10 נקודות ניצחון הכי מהר בהתחשב בבחירת אסטרטגיה נתונה מראש.

החצי הראשון של המשפט נשמע מבטיח, "בוא נמצא את המקום הכי טוב להציב בו את היישוב בהתחלה!" אבל אז, החצי השני: "מיקום זה יעזור לנו להגיע הכי מהר ל-10 נקודות"... פה הוא כבר מעורר חשד. אז תנו לי להסביר: בהמשך למה שכתבתי קודם, המחקר לא יעסוק בבחינת משא ומתן, יחסים בין שחקנים וכו'. המחקר בא לבדוק דבר אחד פשוט - מה קורה כשנשארים דבוקים לאותה אסטרטגיה מתחילת המשחק. הרי בואו נחשבו על זה לרגע אחד... בתחילת המשחק זה השלב שבו המפה תהיה הכי ריקה (באופן יחסי, תלוי מתי אתם מציבים ראשונים). אנחנו יוצאים מנקודת הנחה שהעובדה שיש בידיכם את האפשרות לבחור מיקום אופטימלי ליישוב על פני איזה מקום שתמצאו, מתוך הרבה אפשרויות היא מכרעת, ובהמשך עלולה אפילו לזכות אתכם בניצחון. אנחנו נמצא את המיקום הזה, ולאחר מכן נבחן מה קורה כשנשארים דבוקים לאסטרטגיה הכי טובה לאורך כל המשחק בשיטה דטרמיניסטית זו.

אסטרטגיות ניצחון:

אסטרטגיות ניצחון רבות ככל שיהיו מצריכות משאבים. גם להרחיב ישוב לעיר או לקנות דרך וישוב או אפילו קלף פיתוח... בסופו של דבר דברים אלה מצריכים קלפי משאבים. לדוגמא: אסטרטגיה שמשלבת 4 ערים + דרך הכי ארוכה מצריכה: 11 בנייה, 12 אבן, 4 כבשה, 12 חיטה ו-11 עץ בסה"כ כדי לקנות את מה שצריך. וכן הלאה...

במקרה זה נעבוד עם טבלה. כל שורה i מייצגת דרך ניצחון X , כאשר כל עמודה j מייצגת את סוג המשאבים שדרך ניצחון זו מצריכה. נסיים עם משהו כזה (שימו לב שהשארתי עמודה בסוף לסכום):

```

/* win conditions table:
   each row determines the resource importance for a specific win condition, the last column
   is the total sum (later used for weighted need calculation)
   j corresponds to: Brick | Ore | Sheep | Wheat | Wood | Total
   i corresponds to: Condition index
*/
double[][] winConditionsResources = {
  {11, 12, 4, 12, 11, 0}, // 4 Cities + LR
  {9, 11, 8, 12, 9, 0}, // 2 cities + 1 Settlement + LR & LA (Extra Brick & Wood)
  {14, 9, 5, 11, 14, 0}, // 3 cities + 2 settlements + LR
  {13, 11, 10, 14, 13, 0}, // 2 Cities + 3 Settlements + LR
  {14, 8, 10, 12, 14, 0} // 1 City + 3 Settlements + LR & LA
};
  
```

בנוסף לכך, אנו יודעים שלכל משאב יש אחוזי פריסה קבועים על גבי המפה. למשל:



21.1%	4 משבצות	משבצת כבשה
21.1%	4 משבצות	משבצת חיטה
21.1%	4 משבצות	משבצת עץ
15.8%	3 משבצות	משבצת בריק
15.8%	3 משבצות	משבצת אבן
5.3%	משבצת אחת	משבצת מדבר

ביודענו זאת, אנו יכולים לשלב את טבלת אפשרויות הניצחון יחד עם טבלת פריסת המשבצות ויחד לקבל טבלה שמייצגת עבור כל דרך ניצחון, את כמות המשאבים שהיא מצריכה בהשוואה לשטח של אותם משאבים על המפה. כך נדע מה התדירות שדרך ניצחון מסוימת עשויה להופיע בהתבסס על המשאבים שהיא מצריכה. למעשה, אם טבלת האסטרטגיה שלנו נראית כך:

Condition	Brick	Ore	Sheep	Wheat	Wood	Total
Cities + LR4	11	12	4	12	11	50
Cities + 1 Settlement + LR + LA2	9	11	8	12	9	49
Cities + 2 Settlement + LR3	14	9	5	11	14	53
Cities + 3 Settlements + LA2	13	11	10	14	13	61
City + 3 Settlements + LR + LA1	14	8	10	12	14	58

נוכל לשלב את הטבלאות וע"י חישוב פשוט נקבל את סך המשאבים המשוקללים (שימו לב שסכמתי בעמודה הסופית):

```
// Calculating the total weighted needs for each win condition
// (Resource needed / area coverage)
double weightedNeed;
double totalResourcesNumber;
double resourceAreaCoverage;
double areaProbability;
double totalWeightedNeed;

for (int i = 0; i < 5; i++) {
    resourceAreaCoverage = resourcesAvailability[i][0];
    totalWeightedNeed = 0;
    for (int j = 0; j < 5; j++) {
        totalResourcesNumber = winConditionsResources[i][j];
        // calculate weighted need for each resource
        areaProbability = resourceAreaCoverage / 18;
        weightedNeed = totalResourcesNumber / (areaProbability);
        totalWeightedNeed += weightedNeed;
        // add to cell
        winConditionsResources[i][j] = weightedNeed;
    }

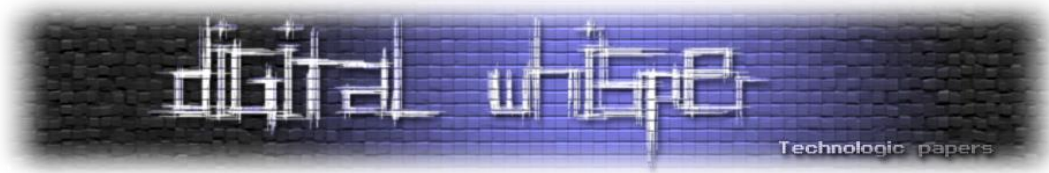
    winConditionsResources[i][5] = totalWeightedNeed;
}
```

לאחר מכן, נוכל לחלק את עמודות המשאבים במספר הכולל של המשאבים שנצטרך עבור כל דרך ניצחון, ובדרך זו נקבל מידע על האחוזים שאנו נרצה מכל אחד מהמשאבים. אחוזים אלה מייצגים עד כמה הם שימושיים לאסטרטגיה המסוימת הזו. הטבלה תראה כך (נחזור אליה בהמשך):

Condition	Brick	Ore	Sheep	Wheat	Wood
4 Cities + LR	25%	28%	7%	21%	19%
2 Cities + 1 Settlement + LR + LA	22%	26%	14%	22%	16%
3 Cities + 2 Settlement + LR	31%	20%	8%	18%	23%
2 Cities + 3 Settlements + LA	25%	21%	14%	20%	19%
1 City + 3 Settlements + LR + LA	29%	16%	15%	18%	21%

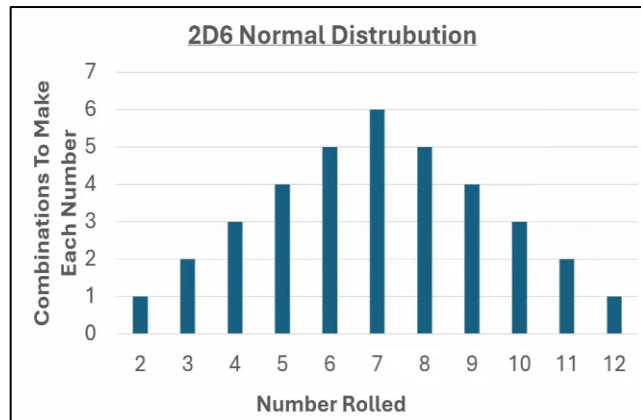
וכך למשל - עבור אסטרטגיה 1, 25% מסך המשאבים שאנו מחפשים כדי לנצח הם יהיו מסוג Brick... וכו'

הדבר האחרון שנותר לעשות זה לקחת את הטבלה האחרונה שקיבלנו ולהמיר אותה לצורה שתוכל להתחשב במספרי הקובייה. למעשה, ע"פ אחוזי החשיבות שהראנו קודם לכן, אנו נוכל גם להכליל ולתת חשיבות גם לתדירות של המספרים שיצאו בקובייה. למה אני מתכוון? מכיוון שבמשחק משתמשים תמיד בשתי קוביות, כל אחת במקסימום 6 ויחד הן יוצרות 12.



[הקומבינציות](#) הנפוצות ביותר שיכולות לצאת בהטלות הן 7, ולאחריה 6 ו-8, ולאחריה 5 ו-9 ואז 4 ו-10 וכן הלאה...

Sum	# of Combinations	Probability	Chance of the difference
2	1	2.8 %	2.8 %
3	2	5.6 %	2.8 %
4	3	8.3 %	2.8 %
5	4	11.1 %	2.8 %
6	5	13.9 %	2.8 %
7	6	16.7 %	2.8 %
8	5	13.9 %	2.8 %
9	4	11.1 %	2.8 %
10	3	8.3 %	2.8 %
11	2	5.6 %	2.8 %
12	1	2.8 %	2.8 %



[הבהרה: הכוונה בקומבינציה היא לדוגמא, כדי שיצא 4 בשתי הקוביות יש לנו 3 קומבינציות שונות והן 1 ו-3, 2 ו-1]

חשוב להזכיר שאנו מחפשים את המספר בקובייה שאליו אנו מכוונים כשאנחנו מחפשים משאב מסוים. כי כפי שציינו בהתחלה, מטרתנו היא לזהות את מיקומי הצבת היישוביים האופטימליים ביותר לדרך הניצחון שנבחר בה (היא מוגדרת ע"פ סך הקלפים שצריך מכל משאב, הסבר מורחב בהתחלה). וכך, ע"י הנוסחה הבאה:

$$Relative\ Token\ Score = 6 - (Highest\ \% - resource)3$$

נוכל להמיר את הטבלה הקודמת לטבלה שמשקפת את המספרים העדיפים בקובייה. לדוגמא: עבור אפשרות ניצחון 1 ומשאב מסוג Brick, נחשב:

$$Relative\ token\ score = 6 - \frac{28 - 25}{3} = 6 - 1 = 5$$

ואכן כך, עבור המשאב Brick אנו נרצה לכוון למספר 5 בקובייה (או המקביל אליו 9).

בתמונה הבאה נוכל לשים לב לטבלאות של מספרי הקובייה העדיפים מכל משאב (כאשר כל שורה מייצגת דרך ניצחון אחרת). כך עבור מספרים שמעל ל-6 ומתחת ל-6 כולל:

```

5 6 0 6 5
3 5 3 6 3
6 2 0 4 6
5 4 3 6 5
6 2 3 4 6

9 8 14 8 9
11 9 11 8 11
8 12 14 10 8
9 10 11 8 9
8 12 11 10 8

```

```

// Main function to calculate win conditions
public ArrayList<int[][]> WinConditionsCalculation(double[][] resourcesAvailability, double[][] winConditionsResources) {
    // Calculate important index table
    winConditionsResources = importantIndex(winConditionsResources, resourcesAvailability);
    double[] highestIndex = highestIndexCalculation(winConditionsResources);
    // Create relative token score table:
    // given the resource importance index table, it then calculates the relative dice token score for each resource.
    // for less than 7
    int[][] relativeTokenScoreBottom = new int[5][5];
    for (int i = 0; i < 5; i++) {
        for (int j = 0; j < 5; j++) {
            // calculate relative token score
            double resourcePercentage = winConditionsResources[i][j];
            relativeTokenScoreBottom[i][j] = (int)(6 - (highestIndex[i]*100 - resourcePercentage*100) / 3); // scale to 1-6
            //System.out.println(relativeTokenScoreBottom[i][j] + " ");
        }
        //System.out.println();
    }
}

```

לפני שמסתיים חלק זה אני מוכרח לתת קרדיט ל-Dwarves & Dice הענק שהעלה על זה [סרטון ביוטיוב](#) ממנו שאבתי חלק מההשראה להתחלת המחקר. מרגע זה והלאה כל מה שתקראו הומצא ומומש על ידי.

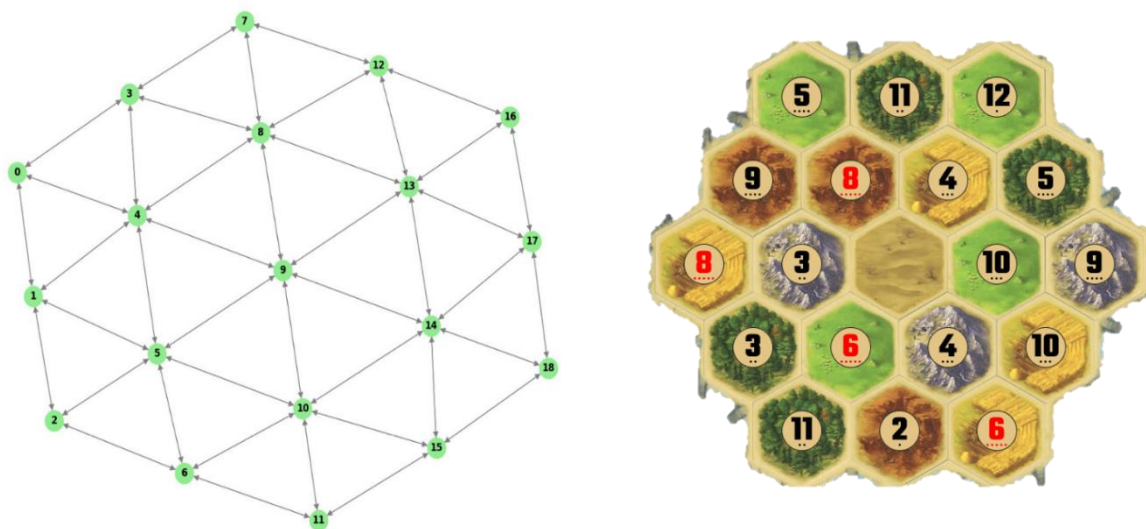
אוקיי. וואו. אז עד לנקודה זו אנו יודעים איזה מספרי קובייה אנו מחפשים עבור כל משאב בדרך ניצחון מסוימת. זה די והותר כדי להתקדם לשלב הבא. ופה באמת לפחות לטעמי מגיע החלק המעניין!

עברנו חישובים הסתברותיים שעזרו לנו להבין באיזה משבצת אנחנו רוצים לבחור במפה בהתחשב ברשימה של אסטרטגיות מוכנות לניצחון. חשוב מאוד להגיד כי הרשימה שאנחנו מכניסים כקלט היא עומדת לרשותנו בלבד. אם דני החליט שיש לו 25 דרכי ניצחון שהוא חושב עליהם, ויוסי חושב שיש אפילו 70, אז אדרבה! אין פה מגבלה מסוימת, ולאורך ההסבר אנחנו נמחיש את פעולת האלגוריתם לפי 5 אסטרטגיות שונות לניצחון. חשוב להגיד שאנו גם מניחים את האלגוריתם שלנו על פי חישוב הסתברותי ולכן חשוב להגיד שהוא יעבוד רק **ברוב הפעמים** שישתמשו בו. ומה לגבי מסחר? ובכן, אם שחקן אחד מחליט שהוא רוצה לסחור קלף בקלף עם שחקן אחר גם זה מותר במשחק הסטנדרטי. אך באלגוריתם שלנו אנחנו משחקים Solo (לחוד), ואנו רוצים לדאוג שהוא יעבוד גם במקרים שבהם כולם שונאים אותנו ולא רוצים לסחור איתנו בכלל. הכל מתבסס על הסתברות לקבל משאבים.

המפה

כיוון שבעולם התוכנה קשה לייצג מפה של 19 משושים ע"י מטריצה ריבועית או מלבנית, פשוטה אנו נאלצים לעבור לייצוג של המפה בגרף.

כל משושה יהיה צומת בגרף, וכל קשת מחברת בין המשושים, ע"י [Adjacency Matrix](#) נקבל את הדבר הבא (ראו בתמונה - הסדר של המספרים שרירותי):



הגדרת הבעיה: אמרנו שאנו רוצים לנסות למצוא את המיקום האופטימלי להציב בו יישוב התחלתי. אז אם כן, נמין את הבעיה לשני סוגים.

1. אנו רוצים למצוא את המשושה עם המספר הקרוב ביותר לטבלת מספרי הקובייה שראינו קודם.
2. אנו רוצים להבין באיזו אסטרטגיה לבחור בהתאם למצב הנתון של המפה.



כדי לטפל בבעיות אלה, הפתרון שלנו ישלב דרך פעולה שפותרת את שתי הבעיות במקביל. מה שנעשה זה את הדבר הבא: נעבור שורה שורה על כל דרכי הניצחון שהצבנו בטבלה, ועבור כל דרך ניצחון כזאת ניצור **מפת חום (Heatmap)** שמתארת את **הקרבה** של כל משבצת אל המספר שאנו מחפשים בטבלת הניצחונות, בהתאם לסוג המשאב שאליו המספר פונה בטבלה.

למה דווקא מפת חום?

מפת החום הזאת תהיה חלק קריטי שיעזור לנו בהמשך להבין היכן למקם את הישוב הראשון שלנו במשחק. בנוסף לכך, אנחנו נראה בהמשך שנבצע מיונים מורכבים שיאפשרו לנו לבחור את האופציה הטובה לנו על פי האסטרטגיה הכי טובה.

האסטרטגיה הכי טובה שנבחרת בהתאם למצב הנתון של המפה היא זאת שנמשיך איתה לכל אורך המשחק, ולכן חשוב מאוד לעבוד במפות חום מסודרות היטב כך שנדע לסווג כל אזור במפה לפי האסטרטגיה הפרטית שלנו.

השלבים הבאים מעט מסתבכים אך קחו נשימה עמוקה והישארו עמי. זה יהיה שווה את זה!

קודם כל, הנה הפסאודו-קוד (**קוד במילים**) של האלגוריתם שעונה על בעיות 1 ו-2:

```
Given a matrix (num of win conditions) x (18 tiles) & winconditions dice numbers matrix
Initialize all values to 0
for each win condition scenario:
  for each tile on the map:
    if the resource on the tile & dice - number matches EXACTLY the win condition relative to the score table
      (look up the score using resource type and dice number):
        then, add +1 to the score of that win condition for that tile area.
    if the dice number in the according resource in the score table has an error range of 1:
      then, add +0.75 to the score of that win condition for that tile area.
    if the resource on the tile & dice number has error range of 2:
      then, add +0.5 to the score of that win condition for that tile area
    if the resource on the tile & dice number has error range of 3:
      then, add +0.25 to the score of that win condition for that tile area.
  ter going through all matrix entries, we then sort the win conditions (i.e. rows) based on their total score
  across all 18 tiles.
  The win condition with the highest score is considered the most suitable for the current given map
  configuration.
```

הסבר על הפסאודו-קוד: למי שהתקשה להבין, אנו מבצעים ריצה על טבלת האסטרטגיות שלנו. עבור כל "אסטרטגית ניצחון", נעבור בלולאה על כל המשושים שעל המפה (18 בסה"כ, חוץ מהמדבר), ונבדוק את הדברים הבאים:

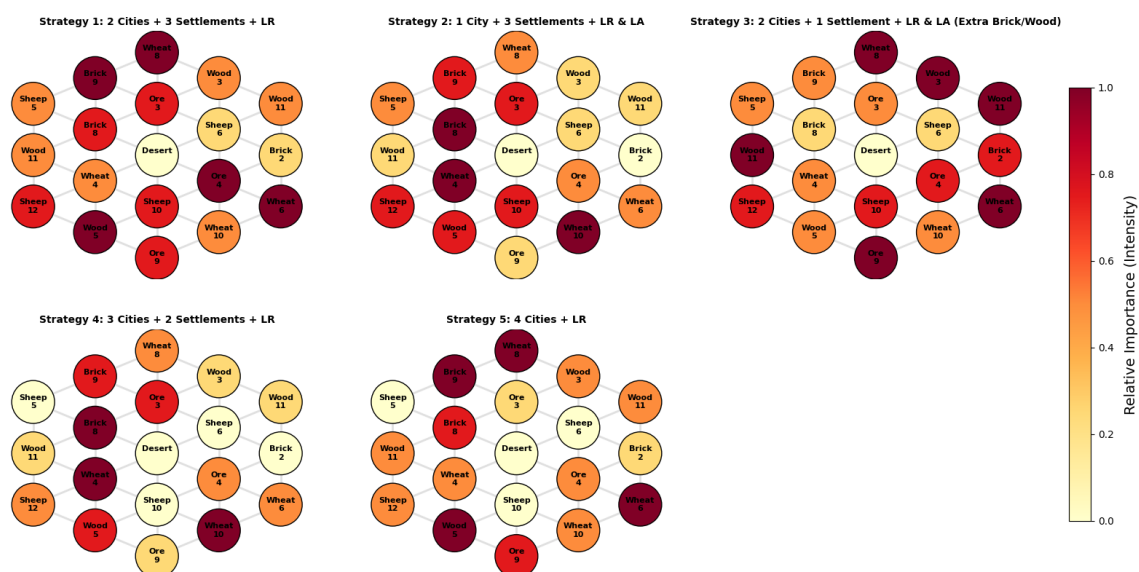
1. במידה והמס' בקובייה של המשאב שנמצא על המשושה תואם בדיוק למס' הקובייה במשאב בטבלת האסטרטגיות, נוסיף ציון של +1 לאותו משושה (במערך מוכן מראש).
2. אחרת, במידה והמס' בקובייה של המשאב שנמצא על המשושה סוטה ב-1 מהמס' המייצג שלו בטבלת האסטרטגיות, נוסיף ציון של +0.75 לאותו משושה.
3. אחרת, במידה והמס' בקובייה של המשאב שנמצא על המשושה סוטה ב-2 מהמס' המייצג שלו בטבלת האסטרטגיות, נוסיף ציון של +0.5 לאותו משושה.
4. אחרת, במידה ונקבל סטייה של 3 נוסף בדומה לקודם +0.25 במיקום של המשושה במערך.

לאחר מכן, נסכום את כל הציונים שנתנו עבור כל שורה בטבלה (תזכורת: שורה מייצגת סוג אסטרטגיה). נמייין את השורות (אסטרטגיות) לפי הסכום הגבוה ביותר. השורה בעלת הסכום הגבוה ביותר היא תהיה האסטרטגיה האופטימלית ביותר עבורנו.

שימו לב שבסוף האלגוריתם אנו מבצעים סכימה של כל "מפת חום" שכזו עבור כל דרך ניצחון בנפרד. אנו עושים זאת על מנת למיין את דרכי הניצחון ולהשיג את דרך הניצחון האופטימלית ביותר.

ובכך, בעיה 1 ובעיה 2 שהצגנו קודם נפתרו. השלב הבא יהיה לזהות היכן להציב את הישובים בתור התחלה. מכאן הדברים נעשים מעט מסובכים, אך לפני שניגע בזה אראה לכם את מפות החום שמצאנו:

Catan Strategy Heatmaps



ניתן לראות שהאסטרטגיה הכי פחות מתאימה שנבחרה היא האסטרטגיה של 3 ערים + דרך ארוכה ביותר. הסיבה לכך טמונה בזה שהמספרי קובייה האופטימליים לנו הם:

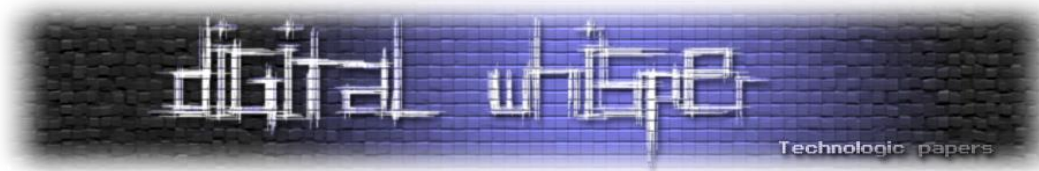
מתחת ל-6 (כולל):

Brick	Ore	Sheep	Wheat	Wood
5	6	0	6	5

מעל ל-6:

Brick	Ore	Sheep	Wheat	Wood
9	8	14	8	9

*המספרים 0 ו-14 כמובן לא קיימים ולכן הם יקבלו את ההתאמה הכי גרועה (צומת של 2 או 12).



אסטרטגיה כזו שמצריכה כל כך הרבה משאבים מסוג "אבן" במפה שבה האבן ממוקמת במספרים לא הכי מוצלחים הסתברותית כמו: 4, 3, 9... נבחרה אחרונה.

אך חשוב לזכור כי באסטרטגיה 5 במקום האחרון: מודגשות באדום כהה רק המשבצות בהם יש התאמה ישירה לטבלה. אך מה עם המשבצות האחרות שיש להן טווח טעות של 1? או 2? ובכן יש רק 2 מאלה. ומה לגבי טווח טעות של 3? יש 7 מאלה.

ומה אם נתבונן באסטרטגיה הזוכה במקום הראשון (בצד שמאל למעלה)? להלן הטבלה:

Brick	Ore	Sheep	Wheat	Wood
5	4	3	6	5
Brick	Ore	Sheep	Wheat	Wood
9	10	11	8	9

ובכן, למרות שנראה כי אסטרטגיה 3 גם לוקחת מקום שווה על סמך סך הצמתים באדום כהה, **אין אנו בהכרח מסתמכים פה על התאמות מדויקות**. חשוב לציין ולהזכיר שהחישוב נעשה על פי **כמותיות**.

למשל: באסטרטגיה 1, אם נרצה משבצת כבשים עם המספר 11, נראה שבמפה יש רק משבצת כבשים של 10 (טווח טעות של 1) ולכן היא נצבעה באדום. מסיבה זו, האלגוריתם שלנו יודע להניב את סך ההתאמות כאשר לכל צומת ניתן ציון הסתברותי על פי הקרבה שלו בטבלת ההתאמה למספרי הקובייה העדיפים.

גישה זו שמסתמכת על **הקרבה** מניבה לנו בסופו של דבר אזורי גישה עתידיים טובים יותר שאליהם נוכל **להרחיב** את האימפריה שלנו על הלוח. זאת בניגוד לאסטרטגיה 5 שבה גם אם נבחר במשבצות העדיפות ביותר (5 עץ, 6 חיטה...) לא בטוח שבהמשך המשחק יהיו לנו מקומות מספיק טובים כדי להרחיב בהם את האימפריה. הבה לא נשכח שבמפה מוצבים עוד 2 או במקרה הגרוע 3 שחקנים נוספים שגם כן רוצים להציב ישובים! (נרחיב על כך בהמשך), לכן לשטח הפוטנציאלי לאימפריה שלנו יש חשיבות מכרעת.

אוקיי, אז עד כה הבנו כיצד בוחרים את מפות החום. אז נחזור כעת לשאלה הבאה, מה לגבי הצבת הישובים? כדי לפתור בעיה זו אנו משתמשים בטכניקות יפות מאוד של גרפים. הבה נפרט.

מיון לרכיבי קשירות בגרף עבור כל אסטרטגית ניצחון, ודירוגם על פי גודל

בואו ננשום עמוק לפני חלק זה כי חשוב שנתרכז כי דברים הולכים ונעשים מסובכים:

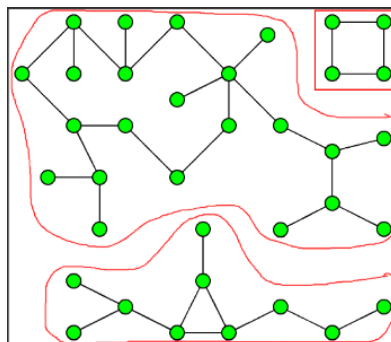
בשלב זה אנו עובדים עם כל מה שהיה לנו עד עכשיו ומטריצה שנראית כך:

Win Condition Per Tile Scores Matrix:																								
0.5	0.5	0.75	1.0	0.75	0.5	1.0	1.0	0.75	0.0	0.75	0.75	0.5	0.25	1.0	0.5	0.5	0.25	1.0	3.0					
0.5	0.25	0.75	0.75	1.0	1.0	0.75	0.5	0.75	0.0	0.75	0.25	0.25	0.25	0.5	1.0	0.25	0.0	0.5	4.0					
0.5	1.0	0.75	0.5	0.25	0.5	0.5	1.0	0.5	0.0	0.75	1.0	1.0	0.25	0.75	0.5	1.0	0.75	1.0	1.0					
0.0	0.25	0.5	0.75	1.0	1.0	0.75	0.5	0.75	0.0	0.0	0.25	0.25	0.0	0.5	1.0	0.25	0.0	0.5	2.0					
0.0	0.5	0.5	1.0	0.75	0.5	1.0	1.0	0.25	0.0	0.0	0.75	0.5	0.0	0.5	0.5	0.5	0.25	1.0	0.0					

תזכורת: כל שורה מייצגת אסטרטגית ניצחון, כל עמודה מייצגת את אינדקס המשושה, והעמודה הכי ימנית מייצגת את מס' האסטרטגיה שאנו נבחר בה. אם תשימו לב נוכל לראות שהאסטרטגיות לא מסודרות לפי סדר כרונולוגי בהכרח, זאת משום שהטבלה הזו מופיעה בסדר ממוין לפי הסכומים כפי שהסברנו קודם לכן. כלומר היא מופיעה כעת כבר לאחר שמיינו את הטבלה לפי האסטרטגיה העדיפה.

כעת, אנו נרצה לקחת את הגרף שאנו עובדים אתו שמשויך לכל המפה ולחלק אותו לרכיבי קשירות ([Connected Components](#)).

אז מה זה רכיבי קשירות? אם נתבונן [בגרף](#) הבא (צומת בירוק וקשת בשחור):



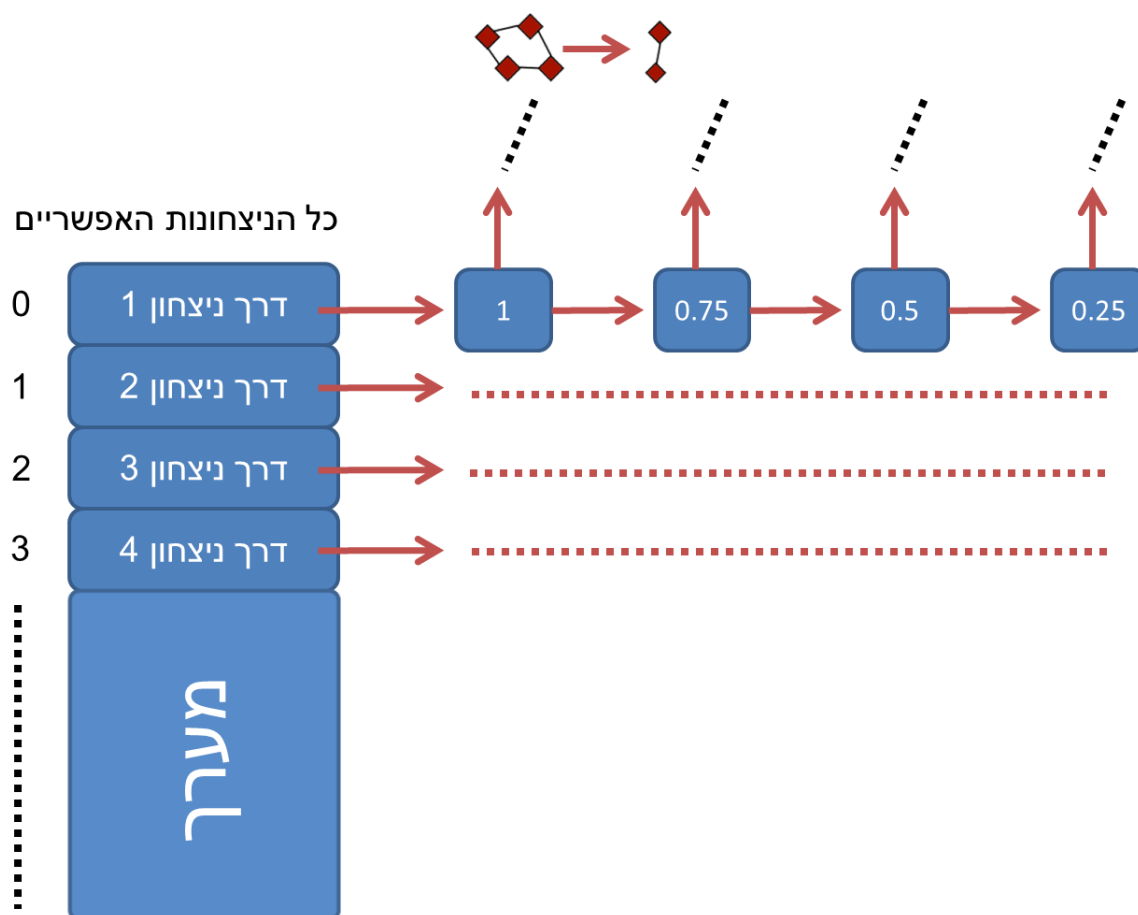
נוכל לשים לב שהוא מחולק לקבוצות שונות של רכיבי קשירות, אלה קבוצות נפרדות שאין אף קשת שמחברת ביניהם. את אותו הדבר בדיוק אנו ניישם על הגרף שלנו, רק שמה שיגדיר כל קבוצה יהיה ה-"Relative importance", שזה האינדקס שהשתמשנו בו קודם לכן בהדמיה של מפת החום.

באינדקס שלנו יש מספר מוגבל של מצבים אפשריים (כפי שראינו בפסאודו-קוד קודם לכן), ואלה- 0.25, 0.5, 0.75, 1.0.

אז בוא נגדיר יחד את הבעיה הבאה שלנו:

אנו רוצים ליצור לכל אפשרות ניצחון בנפרד רשימה שמחזיקה את כל הרכיבי קשירות שקיימים במפה לפי סדר ממוין מהגדול לקטן, כל רכיבי קשירות באינדקס מסוים יהיה מסודרים גם בעצמם מהגדול לקטן.

בקריאה ראשונה הגדרת הבעיה הזו עלולה לבלבל. לכן, כדי לעזור לכם להבין אנסה להמחיש את זה בתרשים הבא:



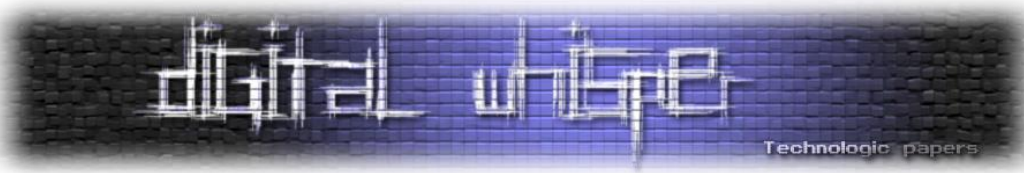
כפי שאמרתי, המספרים 1, 0.75, 0.5... מסודרים מהגדול לקטן. בנוסף, תשימו לב שגודל רכיבי הקשירות גם הם מסודרים מהגדול לקטן - למשל: באינדקס החשיבות (Importance Index) בספרה 1, אנו תחילה סופרים רכיב קשירות בעל 4 צמתים, לאחר מכן 2 וכן הלאה... בסדר יורד.

בדוגמא שהצגתי רואים רק ארבעה דרכי ניצחון, אך כמובן שכל זה ניתן להרחבה וזה כמובן תלוי בערך שהכניס המשתמש, כמספר דרכי הניצחון שהוא רוצה להכניס.

אז למה בעצם כל זה חשוב?

אז בואו נחשוב על זה, כשאנו ממיינים את המשושים מהגדול לקטן אנו מתפרסים על פני שטח גדול יותר. בהמשך אנו נרצה שכל שחקן יבחר את המיקום המועדף ביותר לבניית יישוב. אם נמין גם את דרכי הניצחון מהטוב לגרוע, גם את מידת החשיבות של המשושה (המשבצת), ובנוסף עבור כל מידת חשיבות נמין גם את השטחים שלהם מהגדול לקטן, באופן הזה אנו תמיד נדע בכל מצב נתון של המפה מהי האפשרות העדיפה ביותר להתפרס בה ולבנות יישובים בהתאם לאסטרטגיה שלנו.

הערה: במובן מסוים המיון שלנו הוא וריאציה של Bucket Sort - מיון דליים ([להרחבה](#)).



את המידע הזה אנחנו לוקחים איתנו הלאה להמשך האלגוריתם. כשנרצה להתאים בגרף גם את הישובים אנחנו נרצה לדעת את האזור המועדף ביותר עלינו, וזה בדיוק מה שנעשה!

כעת לאחר שהסתיים שלב ההסבר התיאורטי, אסיים חלק זה בפירוט טכני (הצצה מהקוד) ואיור נחמד:

הערה: הקוד נכתב בשפת Java.

1. המערך עם הרשימות יוגדר כך: `ArrayList<ArrayList<ConnectedComponentTile>>` כאשר

`ConnectedComponentTile` זו מחלקה שבה נשמר מידע על כל משבצת/משושה:

- `TILE_INDEX` - האינדקס של הצומת בגרף (אין למספר זה קשר למספרי הקובייה)
- `TILE_VALUE` - מידת החשיבות במפת החום (0.25, 0.5, 0.75, 1).
- `TILE_RESOURCE_TYPE` - סוג המשאב.
- `TILE_DICE_NUMBER` - המספר בקובייה.

2. להלן הצצה ל-פסאודו-קוד:

```
for each win scenario:
  for each importance_index:
    We want to create a set (List) of sets of connected components, and rank
    them from the biggest to smallest.
    Then, we output an array of a linked list of linked lists consisting of
    tiles within the class of "ConnectedComponenetTile"
```

למעשה, לאחר שניצור רשימה של רשימות מקושרות, אנו נרצה למיין אותה, בזה משתמשים ב:

```
componentsFound.sort((c1, c2) -> Integer.compare(c2.size(), c1.size()));
```

אבל רגע אחד, כיצד אנו נמצא את הצמתים (משושים) של כל רכיב קשירות בנפרד? לשם כך ניעזר באלגוריתם *DFS* שידע למצוא את כל רכיבי הקשירות שלנו בגרף. (כדאי לקרוא על האלגוריתם כאן: https://en.wikipedia.org/wiki/Depth-first_search).

קופצים מכל צומת לצומת הבאה שקרובה אליה אנו משתמשים ב-*Adjacency matrix* שבנינו מקודם:

```
// for 18 tiles, the numbers are counted from top left to bottom right, left to right, top to bottom.
// the rotation of the board does not matter for this analysis
public int[][] adjacencyMatrix = {
  {0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0}, // 0
  {1,0,1,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0}, // 1
  {0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0}, // 2
  {1,0,0,0,1,0,0,1,1,0,0,0,0,0,0,0,0,0}, // 3
  {1,1,0,1,0,1,0,0,1,1,0,0,0,0,0,0,0,0}, // 4
  {0,1,1,0,1,0,1,0,0,1,1,0,0,0,0,0,0,0}, // 5
  {0,0,1,0,0,1,0,0,0,0,1,1,0,0,0,0,0,0}, // 6
  {0,0,0,1,0,0,0,0,1,0,0,0,1,0,0,0,0,0}, // 7
  {0,0,0,1,1,0,0,1,0,1,0,0,1,1,0,0,0,0}, // 8
  {0,0,0,0,1,1,0,0,1,0,1,0,0,1,1,0,0,0}, // 9
  {0,0,0,0,0,1,1,0,0,1,0,1,0,1,0,0,0,0}, // 10
  {0,0,0,0,0,0,1,0,0,0,1,0,0,0,1,0,0,0}, // 11
  {0,0,0,0,0,0,0,1,1,0,0,0,0,1,0,0,1,0}, // 12
  {0,0,0,0,0,0,0,0,1,1,0,0,1,0,1,0,1,0}, // 13
  {0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,1,1,1}, // 14
  {0,0,0,0,0,0,0,0,0,1,1,0,0,1,0,0,0,1}, // 15
  {0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1}, // 16
  {0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1,1}, // 17
  {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,1}}; // 18
```

3. כפי שהבטחתי, אספק תצלומים מהקוד עצמו. נתחיל ב-Adjacency matrix:
שימו לב שהמספרים עצמם נספרים על הלוח מהפינה השמאלית ביותר העליונה אל הפינה הימנית ביותר התחתונה. סדר הכנסת המספרים לא משנה במקרה הזה.
וכעת נעבור לפונקציה המרכזית:

```
ArrayList<ArrayList<ConnectedComponentTile>>[] StronglyConnectedComponentsSorting(double[][] winConditionPerTileScores)
{
    int numWinConditions = winConditionPerTileScores.length;
    // get adjacency matrix
    Tile graph = new Tile();
    int[][] adjacencyMatrix = graph.adjacencyMatrix;
    // create a container for the file result: used to hold strongly connected components for each win condition
    // Structure: Array[WinCondition] -> List [Ranked Groups] -> List [Tiles in that group]
    @SuppressWarnings("unchecked")
    ArrayList<ArrayList<ConnectedComponentTile>>[] sortedComponents = new ArrayList[numWinConditions];
    // for each win condition scenario:
    for (int i = 0; i < numWinConditions; i++) {
        // Initialize the list for this specific win condition
        sortedComponents[i] = new ArrayList<>();
        // Get the specific row of scores for this win condition
        double[] currentScores = winConditionPerTileScores[i];
        // loop through all ranking types: 1, 0.75, 0.5, 0.25
        double ranking = 1.0;
        while (ranking > 0) {
            // Call the helper method with the Adjacency Matrix and Score Array
            ArrayList<ArrayList<ConnectedComponentTile>> componentsFound = getComponents(adjacencyMatrix, currentScores, ranking);
            // sort the componentsFound list in descending order
            componentsFound.sort((c1, c2) -> Integer.compare(c2.size(), c1.size()));

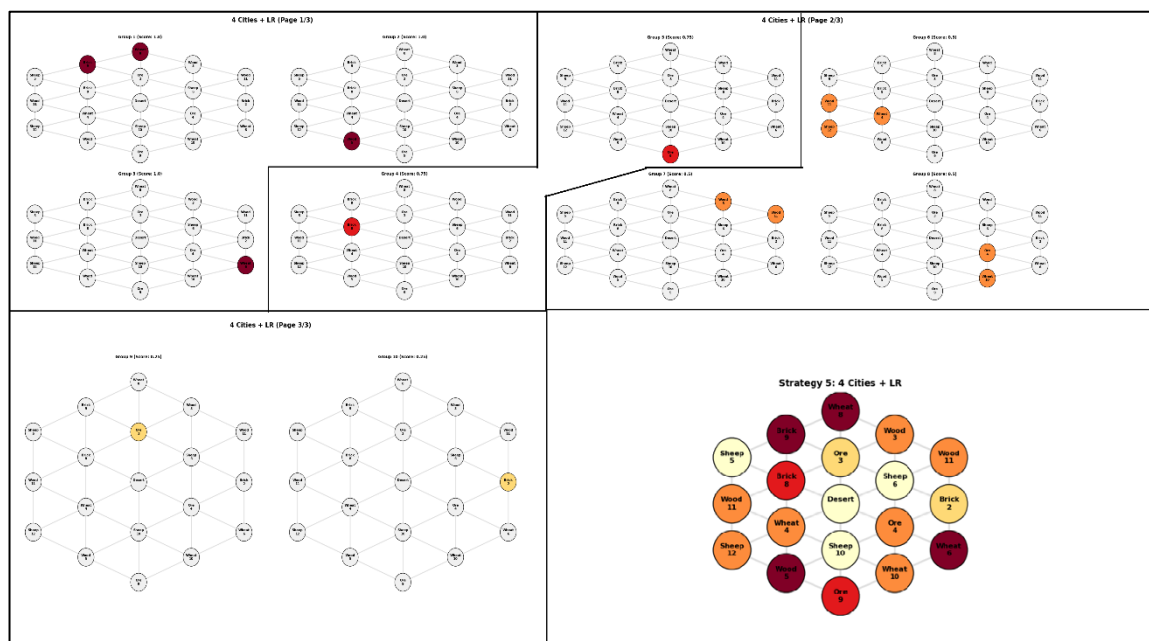
            // Add the groups found to our main list
            sortedComponents[i].addAll(componentsFound);
            ranking -= 0.25;
        }
    }
    return sortedComponents;
}
```

פונקציה ה-DFS:

```
// Helper method to get components of a graph using DFS algorithm depending on node ranking.
private ArrayList<ArrayList<ConnectedComponentTile>> getComponents(int[][] adj, double[] scores, double targetRanking) {
    int numTiles = adj.length;
    boolean[] visited = new boolean[numTiles];
    // initialize the result list the new class
    ArrayList<ArrayList<ConnectedComponentTile>> res = new ArrayList<>();
    // Loop through all tiles (0 to 18)
    for (int i = 0; i < numTiles; i++) {
        // Start search if unvisited AND matches score
        if (!visited[i] && scores[i] == targetRanking) {
            // Create a new connected component tile list
            ArrayList<ConnectedComponentTile> component = new ArrayList<>();
            // Run DFS to fill this list
            DFS(adj, scores, visited, i, component, targetRanking);
            res.add(component);
        }
    }
    return res;
}

// Helper method for DFS finding connected components of a graph
private void DFS(int[][] adj, double[] scores, boolean[] visited, int currentTile, ArrayList<ConnectedComponentTile> res, double targetRanking) {
    visited[currentTile] = true;
    int resourceType = this.mapTiles[currentTile].resourceType;
    int diceNumber = this.mapTiles[currentTile].diceNumber;
    res.add(new ConnectedComponentTile(currentTile, scores[currentTile], resourceType, diceNumber));
    // Iterate over neighbors
    // "neighbor" is the index of the adjacent tile
    for (int neighbor = 0; neighbor < adj.length; neighbor++) {
        // Check connection, visited status, and score match
        if (adj[currentTile][neighbor] == 1 && !visited[neighbor] && scores[neighbor] == targetRanking) {
            DFS(adj, scores, visited, neighbor, res, targetRanking);
        }
    }
}
```

4. ונסיים עם ויזואליזציה נחמדה שמראה את מה שקיבלנו:



תשימו לב כיצד במקרה של אסטרטגיה 5 אנחנו מקבלים בדיוק את מה שרצינו, כל מחלקת שקילות (Connected component) של מספר חשיבות מחולקת בנפרד למחלקות שקילות בעצמה. אחרי שסיימנו חלק זה אפשר להגיד שאנחנו מוכנים דיו כדי לעבור לשלב הבא והמעניין!

הצבת הישובים

המפות שיש ברשותנו כעת לא מאפשרות לנו לדעת היכן הוא המיקום האופטימלי להציב בו את הישוב. מה יקרה אם נציב את הישוב הראשוני בסמוך ל-3 משושים מאותו סוג משאב? או בסמוך ל-2 משושים מאותו סוג משאב... הבעיה הזאת מכריחה אותנו למצוא פתרון טוב ויעיל כדי להבין איך אנחנו מציבים שני ישובים במקום שגם יניב לנו תפוקה גבוהה וגם פיזור מספיק רחב של משאבים מהסוגים שאנחנו הכי צריכים.

בשביל לגשת לבעיה זו אנו מוכרחים לבנות גרף חדש לגמרי מאפס, זה יהיה גרף הישובים. גרף זה יכול ל-54 ישובים כמספר הישובים בכל המפה. אנחנו נרצה לחבר כל ישוב לאחר ובכך ליצור רשת עצומה שנוכל לטייל בה ולהגיע מישוב לישוב במהירות.

הדבר הנוסף שאנו נזדקק אליו הוא מידע לגבי כל יישוב. נצטרך לדעת לאיזה משושה על המפה קרוב כל ישוב, כדי שנדע לבחון איזה משאבים ישוב יוכל לקבל (ובכך להתאים את זה לאסטרטגיה שבחרנו). יש כ-3 אפשרויות: ישוב בפינת המפה והוא נוגע במשאב אחד בלבד, ישוב בצד המפה או בפנים שנוגע ב-2 משאבים, וישוב באמצע המפה שנוגע ב-3 משאבים שונים.

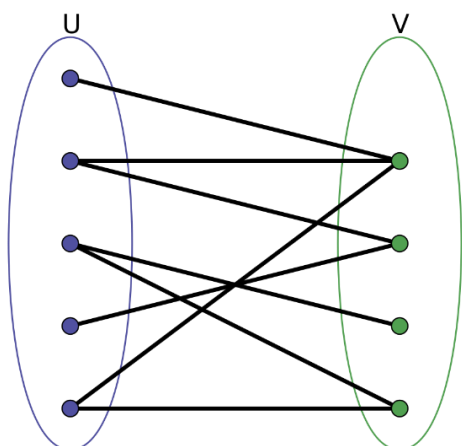
אנו נתקוף את הבעיה בשני אופנים:

1. ניצור את הגרף שמחבר בין ישוב לשוב - זה קל.
 2. השכבה הבאה תהיה להוסיף גרף נוסף מעל, שמחבר בין כל מיקומי ישוב למשוישים שהוא נוגע בהם, אך הוא יהיה **נפרד** מהגרף הקודם. אחרת, פשוט נגיע למצב בו נטייל ממשושה לשוב ובחזרה (מצב שנוגד את חוקי המשחק).
- לפני שנמשיך, נסו להתבונן בתמונה הבאה ותוודאו שאתם מבינים היטב את העובדה שמתוך כ-54 מיקומים אפשריים להצבת ישוב במפה, יש רק 3 מצבים אפשריים ביחס לכמות המשוישים שנוגעים בו (הבהרה: המדבר לא מניב משאבים כלל):



כעת, על מנת לפתור את בעיה מס' 1 ו-2 נצטרך לעבור על מושג חדש וחשוב: גרף דו-צדדי.

"בתורת הגרפים, **גרף דו-צדדי** (נקרא גם גרף דו-חלקי) הוא גרף שבו ניתן לחלק את הקודקודים לשתי קבוצות זרות, כך שלא קיימת קשת בין שני קודקודים השייכים לאותה הקבוצה." - (מתוך ויקיפדיה).



אם נרצה להקביל את הבעיה הזו למצב שלנו, נבין שבהחלט אנו נדרשים להשתמש בגרף דו-צדדי על מנת לפתור את בעיה מס' 2. בצד שמאל (U) נציב את כל הקודקודים של הישובים בזה אחר זה (בסה"כ 54). ובצד השני (V) נציב את כל המשוישים (בסה"כ 19). לאחר מכן, נחבר קו בין כל ישוב למשושה שבסמוך אליו.

בסופו של דבר נקבל מן גרסה ענקית של "חברו את הקווים" שמפרידה בין הישובים לכל המשוישים. ההתאמה הזו ביניהם תוכל לעזור לנו בהמשך לגנוב סטטיסטיקות ומידע על כל ישוב בנפרד ולשמור אותו. בנוסף, האינדקסים של המשוישים בצד הימני יהיו אותם האינדקסים ממפות החום שיצרנו. השילוב של שני אלה ביחד יניב לנו מפה מאורגנת ענקית שנוכל להריץ עליה בדיקות חשובות בהמשך.



הטעינה של הגרף הדו צדדי הייתה מאוד מורכבת. קשה היה למצוא קורלציה מתמטית נוחה בין הישובים שמקיפים כל משושה לבין האינדקסים של המשושים, ומנגד, להכניס את החיבורים שלהם ידנית היה מאוד מורכב. בעיות נוספות הן למשל: מאיזה כיוון יש להסתכל על הגרף, או מה עושים כאשר אותו יישוב מתחבר לכמה משושים?

בסופו של דבר החלטתי לעשות את הדבר הבא: במקום ליצור גרף חדש, החלטתי ללכת בגישה אחרת, הכנסתי אל תוך האובייקט במחלקה שאחראית על כל יישוב בנפרד רשימה ובה כל המשושים שמחוברים אליו, וכן ערך מספרי שמתאר את מספר המשושים שמחוברים לכל יישוב (כזכור, ישנם 1, 2 או 3) - הערך הזה ישמש אותנו בהמשך. נתחיל עם הצגה טכנית מעט ולאחר מכן נמשיך:

בתמונה הבאה תוכלו לראות את הפונקציה המרכזית שמחברת בין כל יישוב למשושה, והיא עושה זאת ע"י שימוש במטריצה:

```
// for each town index, this function connects the town to all the close hexagons that touches it
private void mapTownsToHexes() {

    // each rows represents a hexagon, from top to down, each hexagon contains the 6 town IDs that touches it
    int[][] hexToTownsMap = {
        // (Hexes 0, 1, 2)
        {0, 3, 4, 7, 8, 12}, // Hex 0
        {1, 4, 5, 8, 9, 13}, // Hex 1
        {2, 5, 6, 9, 10, 14}, // Hex 2

        // (Hexes 3, 4, 5, 6)
        {7, 11, 12, 16, 17, 22}, // Hex 3
        {8, 12, 13, 17, 18, 23}, // Hex 4
        {9, 13, 14, 18, 19, 24}, // Hex 5
        {10, 14, 15, 19, 20, 25}, // Hex 6

        // (Hexes 7, 8, 9, 10, 11)
        {16, 21, 22, 27, 28, 33}, // Hex 7
        {17, 22, 23, 28, 29, 34}, // Hex 8
        {18, 23, 24, 29, 30, 35}, // Hex 9
        {19, 24, 25, 30, 31, 36}, // Hex 10
        {20, 25, 26, 31, 32, 37}, // Hex 11

        // (Hexes 12, 13, 14, 15)
        {28, 33, 34, 38, 39, 43}, // Hex 12
        {29, 34, 35, 39, 40, 44}, // Hex 13
        {30, 35, 36, 40, 41, 45}, // Hex 14
        {31, 36, 37, 41, 42, 46}, // Hex 15

        // (Hexes 16, 17, 18)
        {39, 43, 44, 47, 48, 51}, // Hex 16
        {40, 44, 45, 48, 49, 52}, // Hex 17
        {41, 45, 46, 49, 50, 53} // Hex 18
    };

    // loop through each hexagon
    for (int hexIndex = 0; hexIndex < hexToTownsMap.length; hexIndex++) {

        // get indices for each hexagon
        int[] townIndices = hexToTownsMap[hexIndex];
        // find and load the hexagon tile object by the index
        ConnectedComponentTile hexagon = getMapTilesIndex(hexIndex);

        // add the tile
        for (int townIndex : townIndices) {
            this.townArray[townIndex].connectedHexagons.add(hexagon);
        }
    }
}
```



לחדי העין שביניכם, תוכלו לשים לב שה-"חיבור של המשושה" מוגדר על ידי חיבור האובייקט של המשושה מהמחלקה של רכיבי הקשירות שיצרנו קודם לכן.

פעולת החיבור הזו דרשה ממני ליצור פונקציה נוספת שתהיה אחראית לחזור אל תוך מבנה הנתונים הנ"ל ולשלוף ממנו את המידע.

לאחר הרבה בדיקות ותיקוני באגים, הצלחתי לייצר את החיבור הזה לכל יישוב בנפרד, ולאחר מכן הכנסתי את הערך המספרי של סך המשושים המקיפים את היישוב, לכל אובייקט של יישוב במחלקה. את התוצאה תוכלו לראות בתמונה הבאה:

```
--- Starting Town Verification ---
Town [0] => Hexagon [0]      Total of [1] hexagons attached
Town [1] => Hexagon [1]      Total of [1] hexagons attached
Town [2] => Hexagon [2]      Total of [1] hexagons attached
Town [3] => Hexagon [0]      Total of [1] hexagons attached
Town [4] => Hexagon [0]      Total of [2] hexagons attached
Town [4] => Hexagon [1]      Total of [2] hexagons attached
Town [5] => Hexagon [1]      Total of [2] hexagons attached
Town [5] => Hexagon [2]      Total of [2] hexagons attached
Town [6] => Hexagon [2]      Total of [1] hexagons attached
Town [7] => Hexagon [0]      Total of [2] hexagons attached
Town [7] => Hexagon [3]      Total of [2] hexagons attached
Town [8] => Hexagon [0]      Total of [3] hexagons attached
Town [8] => Hexagon [1]      Total of [3] hexagons attached
Town [8] => Hexagon [4]      Total of [3] hexagons attached
Town [9] => Hexagon [1]      Total of [3] hexagons attached
Town [9] => Hexagon [2]      Total of [3] hexagons attached
Town [9] => Hexagon [5]      Total of [3] hexagons attached
Town [10] => Hexagon [2]      Total of [2] hexagons attached
Town [10] => Hexagon [6]      Total of [2] hexagons attached
Town [11] => Hexagon [3]      Total of [1] hexagons attached
Town [12] => Hexagon [0]      Total of [3] hexagons attached
Town [12] => Hexagon [3]      Total of [3] hexagons attached
Town [12] => Hexagon [4]      Total of [3] hexagons attached
Town [13] => Hexagon [1]      Total of [3] hexagons attached
Town [13] => Hexagon [4]      Total of [3] hexagons attached
Town [13] => Hexagon [5]      Total of [3] hexagons attached
Town [14] => Hexagon [2]      Total of [3] hexagons attached
Town [14] => Hexagon [5]      Total of [3] hexagons attached
Town [14] => Hexagon [6]      Total of [3] hexagons attached
Town [15] => Hexagon [6]      Total of [1] hexagons attached
Town [16] => Hexagon [3]      Total of [2] hexagons attached
Town [16] => Hexagon [7]      Total of [2] hexagons attached
Town [17] => Hexagon [3]      Total of [3] hexagons attached
Town [17] => Hexagon [4]      Total of [3] hexagons attached
Town [17] => Hexagon [8]      Total of [3] hexagons attached
Town [18] => Hexagon [4]      Total of [3] hexagons attached
Town [18] => Hexagon [5]      Total of [3] hexagons attached
Town [18] => Hexagon [9]      Total of [3] hexagons attached
Town [19] => Hexagon [5]      Total of [3] hexagons attached
Town [19] => Hexagon [6]      Total of [3] hexagons attached
Town [19] => Hexagon [10]      Total of [3] hexagons attached
Town [20] => Hexagon [6]      Total of [2] hexagons attached
Town [20] => Hexagon [11]      Total of [2] hexagons attached
Town [21] => Hexagon [7]      Total of [1] hexagons attached
Town [22] => Hexagon [3]      Total of [3] hexagons attached
Town [22] => Hexagon [7]      Total of [3] hexagons attached
```



כל זה כמובן קורה לאחר שיצרנו מערך בן 54 מקומות כ-סך כל הישובים, כל אינדקס כזה יגדיר יישוב ויוגדרו לו תכונות (Attributes) שיתנו לנו עוד מידע עליו, המידע הוא:

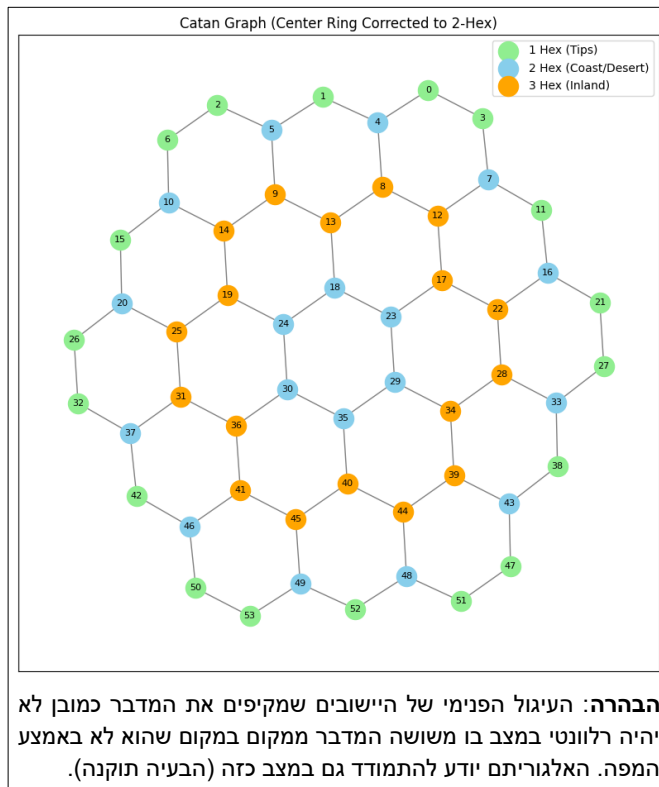
- TOWN_INDEX - יסמל את האינדקס של העיר
 - TOWN_ADJUSTED_HEXAGONS - יסמל את סך המשושים שצמודים ליישוב זה (כפי שהסברנו קודם יש רק 3 אופציות כאלה).
 - *connectedHexagons* - רשימה שמסמלת את כל המשושים שמקיפים את היישוב, בזה אחר זה.
- וכעת ניתן להכריז ש-בעיה מס' 2 נפתרה!**

כעת, על מנת לפתור את בעיה מס' 1, נרצה לחבר בין כל יישוב ליישוב. לשם כך ניצור גרף ענק ע"י מטריצת קשירות (Adjacency Matrix) פעם נוספת. הפעם המטריצה תהיה ריבועית והיא תחבר בין כל יישוב ליישוב. בתמונה הבאה תוכלו לראות כיצד המטריצה נראית במבט מהיר:

[illegible]

כפי ששמתם לב, המטריצה ע-נ-ק-י-ת. לשם יצירת המטריצה השתמשתי בסקריפט שייצר אותה באופן אוטומטי, בניגוד למטריצת הקשירויות הקודמת שהזנתי באופן ידני.

בעמוד הבא, תוכלו להביט בתוצאה של המטריצה לאחר שהכנסתי אותה לקוד בשפה פייתון שמבצע ויזואליזציה. **בעיה 1 נפתרה**, וכעת, לאחר שיצרנו את הגרף שמחבר בין כל יישוב ליישוב אנחנו קרובים אף יותר כדי להתקדם לשלב הבא.



לאחר שסיימנו לעשות זאת אפשר להכריז באופן רשמי **שאנחנו יודעים בהחלט איזה יישוב צמוד לאיזה משושה קריטי שיש לנו במפת החום**. זו כבר הצלחה מסחררת! אבל רגע אחד, שכחנו חולשה אחת סופר קריטית.

אז נניח שזיהינו את כל היישובים הרלוונטיים - כלומר אלה שנוגעים במשושה שאנחנו הכי רוצים, ונניח שכעת אנחנו ממש יודעים לתת אופציות לבחירת יישוב אופטימלי. אבל אנחנו שוכחים מצב לא בלתי סביר שבו יופיעו כמה משושים מאותו סוג משאב אחד ליד השני. אותו גוש (Connected Component) שבחרנו בו יכול

להכיל כמה משאבים מאותו הסוג, ואז במקרה הזה אם נרצה להציב יישוב התחלתי ביניהם אנחנו עלולים לבחור בחירה שגויה שתיתן לנו פשוט **יותר מדי מאותו משאב שאנחנו הכי רוצים**, גם אם במבט ראשון זה נשמע טוב, הבעיה היא שנסתכם **בפחות מגוון** של משאבים אחרים שאנחנו עדיין צריכים בהם בשביל האסטרטגיה שלנו.

מעבר לרצון שלנו לבחור מספרים טובים שיממשו את האסטרטגיה שלנו, אנחנו גם נרצה כמות מסוימת של קלפים מכל משאב, ולכן הבחירה של היישובים מוכרחת להיות מעט יותר מתוחכמת. לא מספיק רק לזהות את היישובים האופטימליים, אלא שאנחנו גם נדרשים למצוא מקום שנותן לנו מספיק Diversity - (מגוון) של משאבים שנזדקק אליהם. אז בוא ניקח את כל מה שהגענו אליו עד כה ונשים את זה בינתיים בהמתנה. וכעת,

בוא ננסח את הבעיה:

"לכל אסטרטגיה מובילה, עלינו למצוא נקודה להצבת יישוב שמוקפת במשושים עם כמה שיותר מגוון של משאבים, ובאותה נשימה גם במספרי הקובייה הרצויים ביותר".

אפשר להגיד בחיוב שאת החלק השני של הבעיה - "ובאותה נשימה גם במספרי הקובייה הרצויים ביותר" כבר פתרנו. אנו נשתמש במערך העצום שבנינו מקודם. על המערך הזה אנחנו נטייל מהגדול לקטן ונדע כך לאתר בצורה מחושבת את הנקודה שבה יש מגוון רחב יותר של משאבים. הכל טוב ויפה (לכאורה), אבל אנחנו מפספסים פה נקודה סופר חשובה!

חשוב להבין ש"מגוון" זה לא הדבר היחיד שחסר פה. אנחנו צריכים לקחת גם בחשבון שלא בטוח שנמצא מגוון להכל, ובמידה וכן, לפחות נרצה שהנקודה שנבחר בה, יהיה לה מגוון שמדורג לפי המשאבים להם אנו הכי זקוקים (תלוי אסטרטגיה). אז בעצם, נרצה לנסח את הבעיה מחדש בצורה טובה יותר:

"לכל אסטרטגיה מובילה, עלינו למצוא נקודה להצבת יישוב שמוקפת במשוישים עם כמה שיותר מגוון של משאבים המדורגים לפי חשיבות, ובאותה נשימה גם במספרי הקובייה הרצויים ביותר".

הערה: שימו לב שהשתמשתי ב-"לכל אסטרטגיה מובילה", זאת משום שכעת, אנחנו עלולים לזרוק אסטרטגיות שלמות רק בגלל חוסר התאמה בין המשאבים ומיקומי היישובים, אך זה בסדר! זכרו שאנחנו עדיין מבצעים את הבחירות הכי משתלמות ואנחנו לא בהכרח נעבוד עם מפה מאורגנת של 54 מקומות להצבת יישובים. (בהמשך הפתרון נוכל גם לחסום כמה קשתות בגרף ולהריץ את האלגוריתם מחדש).

אז כיצד נפתור את הבעיה?

הנוסחה למציאת היישוב הראשון

לפני שנתחיל, אגיד שזה מעט מסתבך מפה, אז קחו נשימה עמוקה. הבא נתבונן שוב בטבלה שהצגתי מקודם, אני מזכיר, היא יודעת את אחוז החשיבות של כל משאב לפי כמות הקלפים הרצויה לכל אסטרטגיה:

Condition	Brick	Ore	Sheep	Wheat	Wood
Cities + LR4	%25	%28	%7	%21	%19
Cities + 1 Settlement + LR + LA2	%22	%26	%14	%22	%16
Cities + 2 Settlement + LR3	%31	%20	%8	%18	%23
Cities + 3 Settlements + LA2	%25	%21	%14	%20	%19
City + 3 Settlements + LR + LA1	%29	%16	%15	%18	%21

במבט חטוף מיד אפשר להבין שהטבלה הזאת עשויה לתרום לנו לפתרון הבעיה הזו. אז אם כן, נעשה את הדבר הבא:

1. קודם כל נזהה אילו יישובים מחוברים לאילו משושים ברכיבי קשירות, ונחבר אותם בגרף דו-צדדי לפי האינדקסים של המשושים. מספיק לחבר את היישובים לאובייקט שיצרנו (שם מרוכז הכל)
2. לכל יישוב יהיה מערך בגודל 5 (כמספר המשאבים שקיימים במשחק), נקרא לו $OptLocation[]$.

אנחנו יכולים ללכת כאן בגישה מסורתית של ללכת לכל היישובים שיש על הלוח ולהבין מה קורה איתם, אבל אנחנו נעשה משהו קצת שונה הפעם. במקום, נלך למערך של הרשימות של הרשימות של האובייקטים (כן תיאור ארוך אני יודע...), ואז בלולאות מקוננות, ממש נמין כל אחד לפי הסדר המתאים. הסיבה המרכזית לבחירה בשיטה הזו היא כדי לשמור על compatibility (תאימות) במידה ונרצה למחוק נתון מסוים בעתיד, לכן כדאי להסתמך על השלב האחרון ביותר שהגענו אליו באלגוריתם.

המימוש של זה נעשה על ידי הצבעה דו צדדית בין כל אובייקט "עיר" אל אובייקט "משושה", כך שכל אחד מהם מצביעים אחד לשני. זה נעשה על ידי רשימות מקושרות בכל מחלקה נפרדת.

3. נעבור על כל דרך ניצחון,

נעבור על כל אינדקס חשיבות (לפי מפת החום),

נעבור על כל רכיבי הקשירות באותו האינדקס,

נעבור על כל המשושים ברכיבי הקשירות,

נעבור על כל היישובים שמחוברים למשושה,

לכל יישוב נטען ערך (ציון) למערך שמשוך לו. (4)

הציון הוא מה שבעתיד יעזור לנו לדעת היכן הוא המיקום האופטימלי לשים בו את היישוב.

4. הערך נטען בצורה הבאה: נדרג את המשאבים הכי חשובים לאסטרטגיה לפי המשקל שלהם, מהכי גבוה לנמוך, נקרא לערך בשם $resWeight$. המשקל ידוע ונמצא בטבלה לעיל. ונוסיף זאת למערך:

$$OptLocation[resource] += resWeight$$

חדי העין בינו שיש פה חזרה על טריק שביצענו כבר מקודם. אך שכחנו דבר מאוד חשוב, אנו מדרגים את היישוב כרגע רק לפי המשאב בלבד. אך מה עם מידת החשיבות של המשושה? אם כן, נוסיף לנוסחה שלנו אלמנט נוסף והוא מידת חשיבות המס' שעל המשושה לדרך הניצחון שלנו. למזלנו שמרנו את הערך הזה כבר באינדקס החשיבות במפת החום (אנחנו יודעים אותו).

נבצע תזכורת קטנה: יש לנו 4 כאלה (1, 0.75, 0.5, 0.25). מעתה המספר של אינדקס החשיבות יכולה בשם: $winIndex$.

ניקח את הנוסחה מקודם ונכפיל את הערך שקיבלנו באחוזים של כל אינדקס חשיבות ונקבל את הדבר הבא:

$$OptLocation[resource] += resWeight \cdot winIndex$$

התוצאה שקיבלנו תהיה הציון שיתווסף לאינדקס של היישוב במערך $OptLocation$.

4.1 ייתכנו מצבים בהם נעבור על יישובים כמה פעמים, רק שניכנס אליהם ממשושים אחרים בלולאה. האם זה לא יצור בעיית התנגשות של הוספת ערך שכבר קיים? ובכן, התשובה היא לא. אנחנו מוסיפים ערכים ולא מחליפים (ישנה חשיבות רבה לטעינת הערך והוספתו במקום שנשנה אותו).

4.2 אנחנו יכולים לדרג כך את כל היישובים אבל אנחנו שוכחים דבר מאוד חשוב. גודל הגושים עצמם של המשושים - כלומר גודל רכיבי הקשירות, גם את זה נרצה לכלול בנוסחה שלנו. למזלנו הרב, דאגנו כבר למיין את רכיבי הקשירות בסדר יורד. לכן נחזיק מס' קבוע בכל מעבר על אינדקס חשיבות (*importance Index*) ששומר את הגודל המקסימלי של רכיב הקשירות שאנו עוברים עליו, נקרא לו *connectedCompSum*. בכל פעם שנתקדם לרכיב קשירות אחר נחלק אותו בסך כל המשושים בקטאן (19). את התוצאה נכפיל בנוסחה שלנו. ויצא משהו כזה:

$$clusterSizeFactor = \frac{connectedCompSum}{19}$$

לאחר מכן, נכפיל אותו בנוסחה שלנו וכך באופן הזה נדע לתת חשיבות גם לגודל רכיבי הקשירות לפי ציון הסתברותי שמייצג את גודלם:

$$OptLocation[resource] += resWeight \cdot winIndex \cdot clusterSizeFactor$$

4.3 עד כאן נדמה שהכל טוב ויפה, אבל אנחנו שוכחים שוב! דבר אחד מאוד מאוד חשוב. מה שאנחנו עושים בנוסחה לעיל זה לאזן בין מספר גורמים שונים מבלי לתת לכל אחד מהם חשיבות בפני עצמה. למה אני מתכוון? אם אנחנו בוחרים באסטרטגיה של דרך הכי ארוכה אז בוודאי שגודל הגושים הרבה יותר חשוב לנו ונרצה להגדיל את *clusterSizeFactor*, ולחילופין, מה אם נרצה לתת יותר תשומת לב למספרי קובייה *winIndex*, או למשאב הכי רלוונטי *resWeight*?

בשביל כל אלה נצטרך להחליט החלטות קשות מאוד. זה דבר שקשה לקבוע ככה על רגל אחת. ראשית, אפשר להסכים של-*clusterSizeFactor* יש חשיבות משתנה בגלל ששטח זה יחסי. שנית, מידת הניצחון *winIndex* היא הקרבה שלנו למספר המתאים בקובייה (לפי מפת החום), זה אולי הדבר הכי חשוב שאנו יכולים להסתמך עליו, כיוון שכך גם מיינו את האסטרטגיות עצמן! ושלישית, ה-*resWeight* עצמו מושפע מהמספר שבחרנו בקובייה - בתחילת החישובים, משתמשים בו בעיקר כדי לפתור בעיית כפילויות של משאבים (נרחיב על זה בהמשך).

הבעיה הזו מצריכה פתרון היוריסטי ([ראו בוויקיפדיה](#)), אחרת, אנו ניאלץ להרחיב את הבעיה למקומות אפלים ביותר של סיבוכיות מקום קשה ואלגוריתמים ארוכים ומסובכים. לכן, ניצור פונקציית הערכה שתעזור לנו לאזן בין גורמים שונים. ננקוט בשיטת *Tuning* (נורמליזציה, להרחבה בנושא [קראו כאן](#)) שבה ניתן לכל איבר בנוסחה ציון מעריכי:

$$OptLocation[resource] += (resWeight^\gamma) \cdot (winIndex^\beta) \cdot (clusterSizeFactor^\alpha)$$

כל אחד מה- α , β , γ יהיה ציון מעריכי שייקבע את מידת החשיבות של כל אלמנט בנוסחה. נקבע "בסיס אסטרטגי", זו תהיה החזקה החשובה ביותר שעליה כל השאר מתבססים. נבחר את *winIndex* להיות אחראי על כך (אם הייתי במקומו הייתי מתרגש!). מדוע נבחר בו?

המספר הזה הוא הערך במפת החום שלנו. מפת החום לא רק עושה שקלול של המספרים בקובייה, אלא היא עושה שקלול של מספרים בקובייה שמתאימים בהתאם למשאים שאנחנו צריכים, כל זאת בהתאם לאסטרטגיה שאנו נמצאים בה! לכן החזקה הזו צריכה לקבל ציון גבוה במיוחד בגלל החשיבות שלה והעובדה שהיא פותרת לנו בעיות רבות, במקרה זה נבחר בציון $\beta = 1.5$. חזקה זו תתפקד כ-"מענישה" שתיתן סטייה חזקה מאוד.

כעת, נבחר "בסיס לינארי" ([להרחבה בנושא](#)), זה יהיה ערך במידת חשיבות נמוכה יותר מ-*winIndex*, אך עדיין גבוה מכל השאר, זה ה-*resWeigh* (ציון לחשיבות המשאב). הסיבה שהוא נמוך יותר מהערך במפת החום היא משום שהערך במפת החום נקבע בכל מקרה לפי חשיבות המשאב. אנו כן נרצה לייחס לערך זה חשיבות מסוימת, משום שיש צורך לעלות לפגוש בכמה משושים עם חשיבויות שונות בו זמנית, ובמקרה כזה אנחנו לא נרצה לעולם שזה יבוא על חשבון בחירה במיקום שסמוך למשושה עם מספר בקובייה ומשאב שהכי טוב לאסטרטגיה שלנו. לכן נקבע לו $\gamma = 1$.

בנוסף, במידה והאסטרטגיה שלנו מסתמכת על ה-"דרך הארוכה ביותר", פה מדובר במקרה קצה שבו דווקא נרצה שהמערך שלנו יהיה משמעותית גבוה יותר, לכן נבחר ב- $\alpha = 0.75$.

אך במידה ולא, נשאיר אותו בעמדה נמוכה יותר. $\alpha = 0.5$. איך נדע זאת? נשמור ערך בוליאני בפונקציה שיעביר לנו את המידע האם אנו בוחרים באסטרטגיה של הדרך הארוכה ביותר.

מזה הרבה זמן שאנחנו נאלצים להשתמש במידע שהמשתמש ייתן לנו, לגבי האם הוא בחר באסטרטגיה שמשלבת את "הדרך הארוכה ביותר".

כמובן שאופציה זו ניתנת לביטול והמשתמש לא חייב לבחור בה (זה יהיה false כברירת מחדל), אך היא תדייק את הפתרון משמעותית. ולסיכום, נקבל את הדבר הבא:

$$OptLocation[resource] + (resWeight) \cdot (winIndex^{1.5}) \cdot (clusterSizeFactor^{0.75 \text{ or } 0.5})$$

4.4 ומה יקרה אם נעבור באיטרציה על אותו היישוב מתוך-משושה מסוג "עץ" ולאחר מכן שוב פעם ניגש אליו מ-משושה מסוג "עץ" (משום ששניהם צמודים אליו)? במקרה שבו עץ זה המשאב שאנחנו מאוד צריכים לאסטרטגיה שלנו, אנחנו פשוט ניקלע למצב שבו נוסיף ליישוב ציון מאוד גבוה למרות שהוא ליד משאבים חוזרים, **ובדרך זו אנו נפסיד משאבים אחרים ולא יתקיים גיוון!** אז אם כן, בואו נדבר על זה...



"תפוקה שולית פוחתת":

בגדול האלגוריתם כמעט הסתיים, הידד! נותרנו עם מערך עבור כל יישוב בנפרד. נותר לנו לעשות רק דבר אחד אחרון, וזה לפתור את בעיה 4.4 שהצגנו זה עתה. הפתרון הוא דווקא פשוט ממה שזה אולי נשמע.

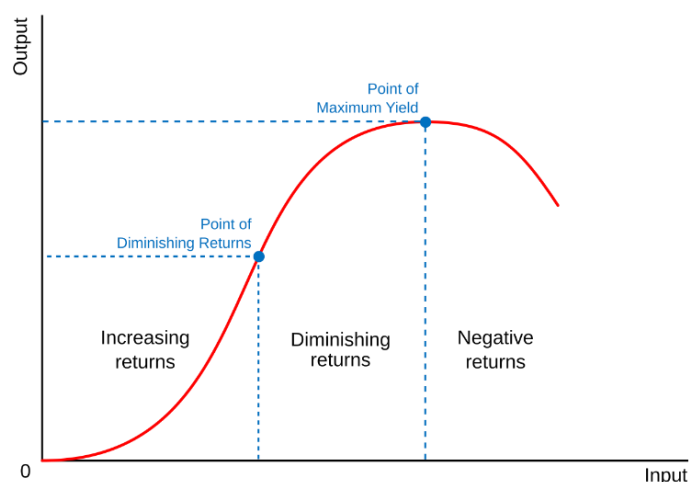
בכלכלה, המונח "תפוקה שולית פוחתת" (להרחבה: [Diminishing returns](#)) מתאר רעיון כללי שבו ככל שנוסיף עוד ועוד מאותו הדבר (משאב) בזמן שהתנאים האחרים נשארים קבועים, **בשלב מסוים** כל תוספת כזו תתרום לנו פחות מהקודמת. ומתי זה השלב המסוים הזה? ואילו תנאים נשארים קבועים?

אז לשם הדוגמא, בוא נגיד שיש לנו מטבח של פיצרייה:

העובד הראשון נכנס למטבח ואופה פיצות, בסה"כ הוא אפה 10 פיצות בשעה. לאחר מכן, נכנס עובד שני נוסף, העובד השני עוזר להכין את הצ'ק בזמן שהעובד הראשון אופה, יחד הם מצליחים להעלות את התפוקה ליעילות של 22 פיצות בשעה! התוספת שלו היא 12 (יעילות גבוהה!). לאחר מכן נכנס עוד עובד שלישי כדי לעזור, אבל פתאום מתחיל להיות צפוף וחם, יחד הם עולים למספר של 28 פיצות בשעה (אבל התוספת שלו היא 6 בלבד, פחות מהעובד השני). לבסוף, כשמגיע העובד הרביעי, כבר נהיה כל כך צפוף וחם שהם נתקלים זה בזה ומחכים לתנור שיתפנה, התפוקה עלתה ל-30 פיצות בסה"כ שזה יותר מ-28! אך התוספת של העובד הרביעי הייתה בסך הכל רק 2.

חשוב שתשימו לב כיצד התפוקה הכללית לא הפסיקה לעלות (10, 22, 28, 30). אך התוספת השולית דווקא המשיכה לרדת ככל שהצטרפו עוד (2, 6, 12). הבעיה הזו מקבילה מאוד למצב שלנו ב-"קטאן".

התוספת השולית של מיקומי יישובים פוטנציאליים למספרים גבוהים, אך מאותו משאב היא מניבה יותר משאבים, אך בטווח הארוך נוצר לנו מצב של תוספת שולית פוחתת, וזה כי למרות "הגדילה המסתמנת", חוסר הגיוון יניב לנו מצב של מיעוט במשאבים אחרים חיוניים - במיוחד בשלבים הראשונים של המשחק!



"אחת ממשמעויות החוק היא כי אין לצפות שהגדלת כמות העובדים בייצור כלשהו תגדיל את התפוקה ביחס בו הוגדל מספר העובדים. התפוקה תגדל בשיעור נמוך יותר משיעור תוספת העובדים, ושיעור זה ילך ויקטן ככל שיתווספו עוד עובדים" (מתוך ויקיפדיה)

אני מזכיר לכם שעצרנו באלגוריתם עם מערך שמתאר חשיבות המיקום האופטימלית לכל יישוב. במערך זה יש כ-5 מקומות בסך הכל כמספר המשאבים שיש במשחק (אנו עובדים כך כדי לסווג בין החשיבות של כל משאב בנפרד).

הפתרון לתפוקה השולית הפוחתת יהיה פשוט למדי: אנו פשוט נסכום את כל המשאבים שקיבלנו במערך המיקום האופטימלי, ונעביר כל תא בפונקציית שורש. כך קצב השינוי ילך ויקטן. הפעלת שורש על כל תא בנפרד לפני הסכימה מתגמלת פיזור שווה יותר יבן המשאבים, על פני ריכוז גבוה במשאב בודד.

***הערה:** אשמור על צניעות ואגיד שאני לא מתמטיקאי דגול, אילו הייתי אולי הייתי בוחר בדרך אחרת יותר טובה, אך מהמחקר שביצעתי זה הדבר הקרוב ביותר שהצלחתי להגיע אליו. כדי להגיע לרמות דיוק גבוהות יותר מה שהייתי עושה זה מריץ מלא סימולציות ועושה Tuning בהתאם לזאת.

ולבסוף, הנוסחה הסופית לדירוג יישוב לפי המיקום האופטימלי תיראה כך:

$$FinalOptLocation = \sum_{i=1}^5 \sqrt{OptLocation_{array}[i]}$$

כל הדרך הארוכה שעשינו נועדה בשביל דבר אחד ☺. לקבוע את המיקום האופטימלי להציב בו את היישובים בהתחלה. אבל חשוב שנעלה פה נקודה למחשבה.

עכשיו יש לנו שתי דרכים לגשת למציאת היישוב הכי אופטימלי:

המידע על רכיבי הקשירות והיישובים שמחוברים אליהם נשמר כבר בזיכרון. כל מה שצריך זה לגשת אליו ולשלוף ממנו נתונים. לכאורה נתנו ציון ודירוג מספיק טוב לכל יישוב. אז מאיפה באמת נשלוף את היישובים?

- אופציה א' - נעבור על הדירוג של כל 54 היישובים, ונשלוף את השניים הכי גבוהים. אך באפשרות זו עלולה להיות בעיה, יש מצב מסוים סביר שבו נבחר יישוב שדווקא לא נמצא ברכיב הקשירות עם דירוג הניצחון הכי גבוה. האם זה באמת מה שאנחנו רוצים?

- אופציה ב' - לעבור על רכיבי הקשירות שגם ככה ממוינים, ובכל רכיב קשירות ממוין נעבור על כל היישובים ש"דבוקים" אליו, נמיין אותם בנפרד, ואז נשלוף את ה-הכי גבוהים. (למה גבוהים? כי בואו לא נשכח שאולי יהיו אפשרויות שייעלמו בעתיד במקרה ששחקן אחר בחר בהן, לא נתעמק בבעיה זו כרגע).

אז במה כדאי לבחור?

נחשוב רגע ונשים לב שיש לנו באופציה ב' סתירה לוגית קשה. אנחנו בעצם אומרים שהחשיבות של רכיבי הקשירות היא אינסופית, כלומר היישוב הגרוע ביותר ברכיב קשירות 1 עדיף על היישוב הטוב ביותר ברכיב קשירות 2. מה שאנחנו מפספסים פה זה למעשה שכל מפת החום שיצרנו מלכתחילה, היא זאת

שעזרה לנו לתת דירוג אמיתי לכל יישוב. ויותר מכך, אף אפשרות לא הולכת "להיעלם" אם שחקן אחר יבנה במקום אחר - כפי שתראו בהמשך.

הנוסחה שבנינו לפני רגע כבר מכילה בתוכה את השיקול של רכיבי הקשירות! לכן אם רכיב קשירות הוא גרוע, הציון של היישוב יצנח כבר באופן אוטומטי. אז אפשר להסכים אחרי הכול שנבחר באופציה א'.

1. מבין כל 54 היישובים, נשלוף את האחד עם ה-FinalOptLocation הכי גבוה.

2. זה היישוב האופטימלי להציב על הלוח בתחילת המשחק.

אז האם זה הכול? ובכן, התשובה היא לא. במשחק קטאן עלינו לבחור דרך בנוסף להצבת היישוב. לעיתים, המשתתפים מגרילים קובייה וכל אחד מטיל בתורו את הקובייה. מי שמקבל את המספר הכי גבוה זוכה להציב את היישוב ראשון יחד עם דרך שצריכה להיות צמודה אליו, וכך באופן הזה מי שיצא לו את המספר השני הכי גבוה בקובייה, ממקם את היישוב והדרך אחריו וכן הלאה. רק לשם ההבנה:

נניח שיש לנו ארבעה שחקנים לפי הסדר הבא בקובייה: שחקן א' -> שחקן ב' -> שחקן ג' -> שחקן ד'

במצב כזה, שחקן ד' יתחיל את הסבב, לאחריו שחקן ג' וכו'... כאשר שחקן א' יסיים את תורו, חוזרים חלילה אל שחקן ב', שחקן ג' ולבסוף שחקן ד'. אז, בסבב השני השחקנים זוכים לבחור ביישוב השני. לפיכך, שחקן א' זוכה להציב שני יישובים ושתי דרכים במקביל.

אז לסיכום, יש לנו בחירה של שני יישובים ושתי דרכים. עוד מעט נדבר על הצבת היישוב הנוסף. אבל בינתיים בואו נדבר על הדרכים. אז קדימה לדרך!

היכן להציב את הדרכים

חשוב לי להבהיר כמה דברים לפני שנתחיל את החלק המעניין הזה. ראשית, מדובר בחלק יחסית מורכב ולעיתים אף מאוד מתמטי, מי שמרגיש שזה יותר מדי בשבילו מוזמן לדלג ל-"הצבת היישוב השני". אך מי שמוכן לעמוד באתגר, אז היכוננו. בנוסף, אנו נגלה שאם עד כה הצלחנו לפתור את שאר הבעיות באמצעות נוסחה סדורה או התחשבות בכל ההיבטים הכוללים, פה, אנו נכיר בעיה שאין לה פתרון מיטבי כלל לא באף גישה. היוריסטית או Brute Force בפרט.

בתחילת המשחק קטאן אנו נדרשים להציב דרך בסמוך ליישוב. את הדרך אנו בוחרים מבין כמה אפשרויות הדרכים שעומדות בצמוד ליישוב אותו כבר הצבנו. יש לנו שתי אפשרויות: להציב דרך אחת או שתי דרכים. ברגע שבחרנו דרך כלשהי, אנו נצטרך להסתמך עליה לאורך המשך המשחק בכך שלכל יישוב אחר שנרצה לבנות, נהיה חייבים לחבר אותו אל המסלול שמוביל לדרך הראשונית שכבר בנינו (בדרך כך או אחרת). המשמעות היא שקודם כל הבחירה הזאת היא מאוד קריטית לנו, במידה מסוימת אנחנו תולים תקווה בדרך הראשונה בעבור שאר היישובים שנרצה לבנות (ואת אלה האלגוריתם שלנו כבר יודע לחשב).

אך הבעיה המרכזית מתחילה ברגע שאנו בוחרים כיוון. כאשר בחרנו כיוון מסוים של דרך אנו מיד מתקדמים לתור של השחקן הבא, ואיננו יכולים לחזות היכן הוא יציב את הדרך שלו על המפה או את היישוב. נניח ששחקן ב' בחר דרך מסוימת, הבחירה שלו תלויה בדרך של שחקן א', ג' ואף ד' לאורך כל הסיבוב, הרי שרמת הסיכון שמציבים לנו שאר השחקנים או אף העובדה שאלו מהם יתפסו לנו כמה יישובים פוטנציאליים היא סכנה ממשית.

יש פה למעשה מצב מתמטי מסובך שבו אין אנו יכולים לחזות את הבחירה של שאר השחקנים שעל המפה ב+1, 2+ ואף 3+ תורות קדימה. הרי שבכל תור יש מספר אינסופי של בחירות או החלטות ששחקן מסוים יכול לקבל, אלה יכולות להיות מסחר, החלפת קלפים, הצבת שודד, קנייה של דברים וכמובן התרחבות על המפה.

בניגוד לבעיית בחירת היישוב שהתמודדנו איתה עד כה, פה, הבעיה מתרחבת ומאלצת אותנו להתחשב בהיבטים ה-לא היוריסטיים הללו. ואני אסביר:



נניח שאני שחקן מסוים על המפה ואני בוחר להתרחב לצד ימין. אך למרבה הצער, בעד כמה תורות אני אגלה ששחקן אחר מצד ימין חוסם את דרכי ולא מאפשר לי להתקדם קדימה אל עבר המקום ליישוב האופטימלי אליו תכננתי להתרחב. וכך, במובן מסוים בזבזתי משאבים רבים על בניית דרך שהציבה אותי בבעיה. אז מה עליי לעשות מפה? להתקדם למקום אחר?

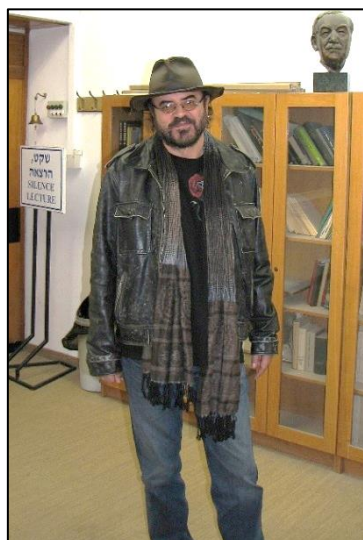
ובכן, אנו נתעלם מהשאלה הזו מכיוון שאנו עוסקים ב-**מניעת מקרים** ולא בהתמודדות עם מצב בעייתי קיים.

אבל כדי להרחיב את הבעיה הזו ולהבין עד כמה שזה כל כך מורכב לפתור את זה בגישה היוריסטית, תנו לי לפתוח בסיפור מעניין על מדען ממוצא יווני וסופת שלגים:

דמיינו שאתם יושבים לכם בבית קפה בטורונטו, השעה 6 בבוקר, בחוץ מינוס 20 מעלות, כיף חיים. יש לכם פגישה סופר חשובה בוונקובר. ואתם צריכים להגיע מנקודה א' לנקודה ב'. אך יש בעיה קטנה, בחוץ משתוללת סופת שלגים מטורפת, הסופה כה עוצמתית שהיא חוסמת חלק מהכבישים מפעם לפעם באופן אקראי.

למזלכם, יש בידיכם מפה מפורטת של כל הכבישים במדינה, אך המפה הזו אינה מעדכנת בזמן אמת את הכבישים החסומים. לרוע מזלכם, הדרך היחידה שלכם לדעת איזה כביש נחסם היא רק להגיע אל הצומת ממנה הוא נחסם ולראות זאת בעיניכם.

אז אתם מחליטים להתחיל לנסוע, ואז בום! נתקלתם בחומת שלג שחוסמת את הדרך, מה תעשו עכשיו? האם כדאי לכם לחזור על עקבותיכם ולנסות דרך ארוכה יותר שאולי היא יותר פתוחה? או להמשיך לצומת קרוב סמוך ולקוות לטוב?



פאפאדימטריו באוניברסיטה העברית

אתם מנסים למקסם את הסיכויים שלכם להגיע אל היעד לפגישה בזמן הקצר ביותר תוך כדי שאתם לוקחים בחשבון שבכל פעם שנתקלתם במחסום, אתם מבזבזים זמן יקר על נסיעה הלך ושוב. זוהי תמצית של בעיית ה-CTP ([Canadian Traveler Problem](#)). אז איך כל זה מתחבר למפה שלנו בקטאן? ובכן, ברגע שאנו בוחרים דרך אנו בוחרים מסלול מסוים אל יישוב אופטימלי, אך בדרך ייתכן ששחקן אחר יחסום אותנו. וכך בדיוק הבעיה מתחברת למצבנו.

הבעיה הזו היא קלאסית בתורת הגרפים ובמדעי המחשב, היא הוצגה לראשונה על ידי מדען ממוצא יווני בשם פאפדימטריו ויאנאקאקיס. הבעיה היא קשה מאוד, למעשה, היא מסוג [complete-PSPACE](#), או במילים פשוטות: היא קשה מאוד לפתרון חישובי במציאות שבה כל החלטה קטנה משנה את מרחב האפשרויות העתידה בסביבה שמשתנה "מתחת לרגליים". בתמונה: דימטריו באוניברסיטה העברית.

עכשיו אנו כן רוצים בכל זאת לחפש איזושהי דרך מספיק טובה, היוריסטית, לפתרון הבעיה. אז ננסה ללכת פה על חבל דק ולמצוא פתרון שאולי לא יהיה מושלם, אך הכי טוב בנסיבות שלנו. ואני מזכיר - הנסיבות שלנו הן **בתחילת המשחק** למצוא את המקום הטוב ביותר להציב בו את הדרך (לאחר שבחרנו ביישוב האופטימלי). בהמשך המשחק דברים נהיים יותר ויותר מסובכים ולכן לא ניגע בזה כלל.

הפתרון לבעיה זו יהיה משולב. ראשית, חשוב להגביל את הבעיה ולהבין שכל שאנו מחפשים זה **כיוון**. המפה היא דבר דינאמי, במהלך המשחק ייתכן ושחקנים מסוימים יחסמו דרכים שונות. לכן, אנו חייבים להיצמד לגישה שבה התחלנו את המסע שלנו פה. הגישה ההיוריסטית. במאמר זה אנו בוחרים את הפתרון ההיוריסטי למשחק קטאן, פתרון שבו ננצל את היתרון ההתחלתי של ההצבה הראשונית ונבחן את השפעתו למשך שארית המשחק.

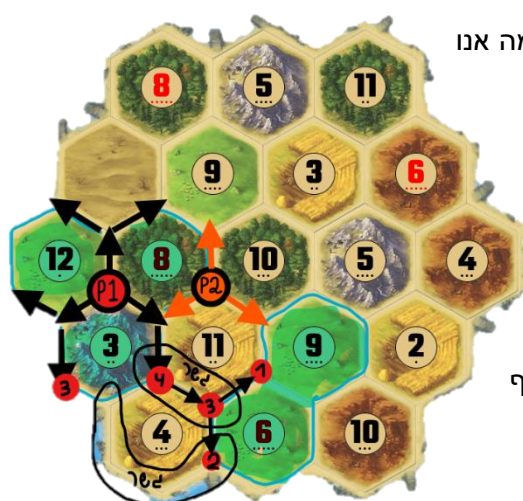
אפשר להגיד ש-"כיוון" כזה יהיה זווית שמובילה אל מיקום פוטנציאלי מספיק טוב להציב בו יישוב עתידי. בשלב זה אנו יודעים כיצד לבחור את המיקום האופטימלי ליישוב, ולכן ננצל עובדה זו כדי לדעת מהו הכיוון העדיף. בנוסף למציאת הכיוון, ניתן לאלגוריתם שלנו גם יכולת למצוא יישוב כזה במרחק "מספיק קרוב".

מרחק קרוב הוא דבר סובייקטיבי ומשתנה. אנו לא יכולים לבחור יישוב פוטנציאלי להתפתח אליו במרחק רחוק משום שאנו פותחים את עצמנו לסיכון שמישהו יחסום אותנו עם דרך משלו בדרך לשם. ואני שב ומזכיר שאיננו יכולים לחזות את העתיד של המשחק. אך אנו כן יכולים להסתמך במשהו מסוים שייתן לנו פתרון שהוא יחסית שלם. **זוה השטחים שבהם שחקנים כבר הציבו יישובים**, משם הם עלולים להתפתח לחסום אותנו. לכן, נרצה גם להימנע או לכל הפחות להתרחק כמה שאפשר מלהתפתח למקומות בהם עלולים לחסום אותנו, ומשם נוכל לפתח את האימפריה שלנו.

האימפריה שלנו מתבססת על גושי (רכיבי קשירות) בעלי עדיפות. הסברתי כבר קודם לכן שהמשקל שנכלל הוא גם הגודל הפיזי במפה של הרכיב קשירות. פה אנחנו נתבסס על זה כדי לפתור את שלל הבעיות שהצגנו. וחשוב להבהיר עוד דבר קטן, נוכל לדעת היכן שחקנים אחרים כבר הציבו דרכים או יישובים אחרים על ידי כך שנחסום בקלט (input) שלנו את אותם משושים או דרכים וניתן להם ערך סתמי. האלגוריתם שלנו ידע לעבוד על מפה חלקית שלא כוללת אותם בדיוק כפי שהוא היה רץ על מפה מלאה.

אז נסכם, ברצוננו:

1. למצוא כיוון מספיק טוב.
 2. להבין מהם המיקומים הפוטנציאליים להציב בהם יישוב שמסביבנו (יישוב פוטנציאלי).
 3. לתעדף מרחק ליישוב פוטנציאלי שהוא מספיק קרוב למיקומנו הנוכחי, אך כזה שיש אליו מסלולים חלופיים.
 4. להתרחק כמה שניתן ממקומות שבהם עלולים להציב דרך ולחסום אותנו (נקודות תורפה וצווארי בקבוק).
- עכשיו תראו. "ממקומות שבהם עלולים לחסום אותנו" זו לא אמירה היוריסטית בכלל. אנחנו צריכים להגדיר פתרון יותר טוב העונה על בעיה זו. הרשו לי להסביר זאת באיור נחמד.



באיור זה אנו נציג סימולציה מעניינת שתעזור לנו להבין עם מה אנו מתמודדים:

יש לנו שני שחקנים על המפה, P1 ו-P2. נניח שאנחנו שחקן מס' 1. שחקן 2 כבר הציב יישוב בקרבתנו. שחקן 1 מעוניין להגיע אל היישובים 1, 2 ו-3 בצד ימין מכיוון שהם מהווים לו דירוג גבוה. אך ישנה בעיה, השחקן השני (הכתום) עומד בדרכו ולא מאפשר לו להגיד אל יישוב מס' 1 בצד ימין במעקף מלמעלה.

מה גם שיש לנו שני גשרים שדרכם נוכל להגיע אל היישובים 2 ו-3 ברכיב הקשירות הימני. גשר אחד צפוני משתלם יותר, ואחר דרומי משתלם פחות. אך טמונה בחובנו בעיה חמורה! למרות שהגשר הצפוני נראה מפתה ביותר, השחקן P2 עלול בקלות לחסום לנו את הגישה אליו בעתיד אילו היה בוחר לבנות דרך באלכסון אליו, במהלך התורות הבאים.



אז לכאורה, הפתרון ה"בטוח" הוא פשוט לברוח דרומה, לעשות עיקוף ענק של 7 דרכים ולהגיע ליעד ללא סכנת חסימה. אבל כל שחקן קטאן מנוסה יגיד לכם שזה רעיון רע! אנו לא יכולים לבחור יישוב פוטנציאלי להתפתח אליו במרחק רחוק משום שאנו פותחים את עצמנו לסיכון שמישהו יחסום אותנו עם דרך משלו בדרך לשם.

בניית 7 דרכים דורשת 14 משאבים, ועד שנשלים אותה המשחק כנראה ייגמר. גרוע מכך, היישובים הפוטנציאליים לאורך החוף הדרומי הם לרוב "עניים" ובדוגמא הזו נוגעים רק במשאב אחד עם מספר חלש (כמו 4). אז כן. כפי שהסברתי, הבעיה לא פשוטה.

כפי שניתן לראות, שוב קונפליקט "הנוסע הקנדי". אז עלינו לחשוב על דרך מסוימת וקבועה לפתור את הבעיה הזו בכל שלב של המפה. אם כן, כמו תמיד, במצבים כאלה נהוג להציג פתרון הסתברותי/סטטיסטי וכפי שעסקנו בזאת עד כה, זה בדיוק מה שנעשה.

האלגוריתמים שפותרים את בעיית ה-CTP לרוב לא מחפשים בצורה עיוורת את "הדרך הקצרה ביותר" או ה"בטוחה ביותר" (מה שהיינו יכולים לעשות באמצעות חישוב דרך ליישוב אופטימלי נוסף), אלא מחשבים את תוחלת העלות. הם משקללים את הסיכוי לחסימה אל מול העלות האסטרטגית של מסלול חלופי.

מבלי להרחיב יותר מדי, ישנו אלגוריתם מאוד ידוע למציאת המסלול הקצר הזול ביותר מקודקוד מקור לקודקוד יעד בגרף ממושקל בעל משקלים חיוביים (המשמעות היא שכל קשת בגרף בעלת מס' כזה). מדובר באלגוריתם מאוד מפורסם בשם "דייקסטרה" ([להרחבה קראו כאן](#)).

אנו ננסה "להתלבש" על האלגוריתם הזה ולשנות אותו כך שנמצא מסלול מיישוב מקור ליישוב יעד אופטימלי. וכמובן, חשוב להזכיר שמציאת היישוב האופטימלי, או השני האופטימלי, או במקום השלישי האופטימלי... היא פעולה שכבר פתרנו בעמודים הקודמים.

אז, ובכן, במקום שכל דרך במפה תשקול פשוט "1", ניצור גרף סכנה והזדמנויות ממושקל. כדי שאלגוריתם דייקסטרה יוכל לרוץ, עלינו להגדיר תחילה את המשקל של כל קשת בודדת (כל דרך) בגרף. נגדיר את המשקל $W(u, v)$ של הדרך מצומת הנוכחי u לצומת היעד v כך:

$$W(u, v) = \max(\epsilon, \text{BaseCost} + \text{Danger}(v) - \text{FinalOptLocation}(v))$$

הבה נפרק את זה ונסביר את הנוסחה:

• עלות דרך בסיסית $BaseCost$:

ניתן לכל דרך בגרף מחיר קבוע וגבוה יחסית (למשל 10). מכיוון שכל צעד יקר, מסלול של 7 דרכים יעלה אוטומטית המון (70). ולחילופין, מסלול באורך שני דרכים, יעלה רק (20). זה פוסל מיד רעיונות תלויים כמו לבצע עיקוף מוגזם, ומאפשר לנו לעקוב אחר אורך המסלול אל היישוב האופטימלי.

• הפקטור על הסכנה $Danger(v)$:

בהשראת בעיית ה-CTP, קשתות המובילות ליעד v שסמוך ליריב (כמו הגשר הצפוני הקרוב ל-P2) יקבלו "קנס" של תוספת סיכון המתווסף לעלות, זאת משום שיש סיכוי סטטיסטי סביר ש-P2 יבנה שם דרך לפנינו. או במילים אחרות: אנו יודעים היכן נמצאים ענני הסערה כעת ברשת הכבישים אך לא בדיוק באיזה כביש השלג יערם כאשר נגיע לשם.

• בנוס צמתי הביניים $FinalOptLocation(v)$:

זהו ייצוג של "עלות ההזדמנות". אם ביעד v נמצא יישוב אופטימלי שמציע הזדמנות בנייה מעולה (את הציון עבור היישוב כבר מצאנו קודם לכן), אז אנו מפחיתים את הציון של היישוב מעלות הדרך! כך באופן הזה, ככל שהציון של היישוב גדול יותר כך אנו מפחיתים יותר מהציון של הדרך. וכזכור, דייקסטרה תמיד יעדיף מסלולים זולים יותר. לכן היכן שנפחית יותר (יישובים טובים יותר) כך נרצה ללכת יותר במיקום זה.

כן ראוי לציין נקודה מאוד חשובה. ה-Clusters (גושים) שלנו הם צמודים אחד לשני בהתאם לערך הציון שלהם. כך גם אנו באופן הזה מבטיחים שלא נסטה החוצה לגושים פחות טובים, הרי שתמיד נתעדף את המסלולים באותו גוש שבו אנו נמצאים (או לחילופין השני הכי טוב).

• הגנת ה- ϵ (אפסילון):

אלגוריתם דייקסטרה קלאסי קורס כאשר הוא נתקל בקשתות בעלות משקל שלילי, מדובר בעניין טכני. פונקציית ה-max מוודאת שגם אם הסבסוד של היישוב ($FinalOptLocation$) הוא אדיר, משקל הדרך אף פעם לא ירד מתחת לערך חיובי מינימלי ϵ (למשל 0.1).

הערת אגב: הערך המקסימלי של $OptLocation$ תמיד יישאר 1, ולכן מדובר רק בפעולה טכנית כדי למנוע קריסה של האלגוריתם, למקרה ו...!

בסופו של דבר, באמצעות הנוסחה הזו, כשהאלגוריתם יריץ את דייקסטרה, הוא יבחן את המעקף הדרומי ויבין: "זה מסלול ארוך מאוד שיעלה לי המון משאבים, ובדרך אין לי שום הזדמנויות בנייה טובות שיפצו על כך". לעומת זאת, הוא יבחן את הגשר הצפוני ויבין: "אמנם יש פה רמת סיכון מסוימת בגלל P2, אבל המסלול קצר, ויש בו תחנות ביניים מעולות לאימפריה שלי ששווה להילחם עליהן".

אז נסכם, בשביל למצוא את הדרך עלינו:

- **בשלב הראשון:**
ניקח את הציון $FinalOptLocation$ שכבר חישובנו בשלבים הקודמים עבור כלל היישובים הפוטנציאליים (ציון המייצג את מידת ההתאמה המושלמת ליישוב בעבור האסטרטגיה שלנו).
- **בשלב השני:**
ניצור גרף סכנה והזדמנויות. נגדיר את משקלי הדרכים בגרף כך שישקפו את אורך המסלול וסכנת החסימה (בהשראת CTP), כאשר הערך של $FinalOptLocation$ ישמש כ"סבסוד" שמפחית את עלות המסלול ככל שהוא עובר בצמתים איכותיים יותר.
- **בשלב השלישי:**
נפעיל את אלגוריתם דייקסטרה על הגרף כדי למצוא את המסלול בעל העלות המשווקללת הנמוכה והמשתלמת ביותר.
- **בשלב הרביעי והאחרון:**
נחזיר את הדרך האופטימלית (הכיוון הראשוני) אל המשתמש.

הערה טכנית: אם יש לנו שני יישובים a ו- b , אז הדרך ביניהם תוגדר כך: (a,b) . ולכן המשמעות האמיתית של להחזיר דרך למשתמש היא למעשה להחזיר זוג סדור של שני אינדקסים של יישובים. מדובר בפרט טכני בקוד שנעשה על ידי מטריצת קשירויות בין יישוב ליישוב.

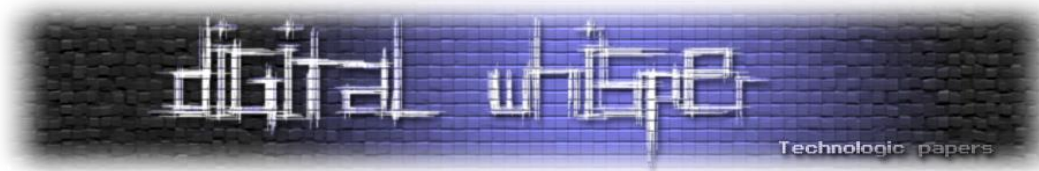
אז עד פה המצב נראה מבטיח. האם סיימנו? אז זהו, **שלא בדיוק**.

האם שמתם לב שהציון $Danger(v)$ שניתן הוא לא הוגדר בכלל? כל מה שעשינו זה להגיד שניתן "מספר שרירותי", "נגיד 10". ובנוסף, אנו מפחיתים ממנו את $FinalOptTownLocation$ שהוא ערך מספרי קבוע שידוע לנו הטווח שלו. ובכן, כמובן שזה לא הגיוני להשאיר כך את $Danger(v)$ בצורה לא מוגדרת. עלינו למצוא פתרון מסוים שיתחשב בכך שהקשר ביניהם יהיה סביר והגיוני. (רק כדי להזכיר, $Danger(v)$ נועד על מנת לקבוע את מידת הסכנה עבור רדיוס הקרבה ליריב על המפה).

או במילים אחרות, הקשר בין $Danger$ ל- $FinalOptTownLocation$ צריך להיות **מנורמל ודינאמי**.

למה הכוונה? נסביר: מדובר בבעיה בה שני פקטורים מסוימים צריכים להיות תלויים זה בזה. במדע מצב זה מכונה כ-"בעיית קנה המידה" ([להרחבה קראו כאן](#)). אם הטווח של $FinalOptLocation$ נע בין 0 ל-5 ואנו זורקים פתאום "קנס" שרירותי של 15, אז פשוט מחקנו לחלוטין את כל ההשפעה של הציון ליישוב שעשינו עד כה. או לחילופין, האלגוריתם שלנו יהפוך לפרנואיד ורק יתחמק מאויבים בדרך אך יתעלם לחלוטין ממשאבים.

ובכן, הפתרון יהיה לבצע דבר שנקרא "[נורמליזציה](#)" (Normalization) בסטטיסטיקה. ובמקרה הזה: **הפקטור של הסכנה חייב להיות הנגזרת של הציון המקסימלי ליישוב האפשרי על הלוח באותו הרגע**. נגדיר את הציון המקסימלי של יישוב על הלוח בתור $MaxOpt$.



הסכנה תלויה במרחקו של היריב הכי קרוב לצומת v מסוימת. נקרא למרחק הזה d_{opp} . ונשתמש בפונקציית דעיכה ([ראו כאן](#)) שמבוססת על הציון המקסימלי שיש ברגע הנוכחי על הלוח:

$$Danger(v) = \frac{MaxOpt \cdot \lambda}{d_{opp}^2}$$

כאשר λ הוא "מקדם רתיעה מסיכון" (למשל $\lambda = 1.5$).

וכיצד זה עובד?

כשהיריב ממש קרוב ($d_{opp} = 1$): רמת הסכנה תהיה $1.5 \cdot MaxOpt$. זה אומר שאפילו אם הצומת הזה הוא הכי טוב על הלוח (ציון $MaxOpt$), החישוב של הסכנה פחות הבנוס ($1.5MaxOpt - 1.0MaxOpt$) ישאיר אותנו עם קנס חיובי של $0.5MaxOpt$. כלומר, במקרה כזה הסכנה מנצחת את הפיתוי, וכך דייקסטר יימנע מהצומת הזה.

כשהיריב במרחק "סביר" ($d_{opp} = 2$): רמת הסכנה צונחת ל- $0.375 \cdot MaxOpt$. ובמצב כזה, אם הצומת באמת מדהים (ציון קרוב ל- $MaxOpt$), הבנוס ינצח את הסכנה וניקח את הסיכון המחושב. אבל אם הצומת בינוני, הסכנה עדיין תרחיק אותנו משם.

ומה אם היריב רחוק ($d_{opp} \geq 3$): הסכנה שואפת לאפס וזניחה לחלוטין.

לאחר שכל זה הובהר, אל לנו להתעלם מפרט קטן שלא נגענו בו, וזה הערך של λ . על מנת להבהיר, המשמעות של ערך זה היא למעשה "אופי השחקן" או רמת האגרסיביות של האלגוריתם שלנו. ערך זה הוא למעשה פרמטר כיוון בדומה לזה שהגדרנו קודם (לגבי מידת החשיבות לדרך הארוכה ביותר). משתנה זה מגדיר עד כמה אנו מוכנים להסתכן כדי להשיג יישוב מעולה, או לחילופין, עד כמה אנו מעדיפים לשחק "על בטוח".

כאשר λ גבוה: אנו יוצרים אלגוריתם שמרן וזהיר במיוחד. "רתיעה מסיכון" גבוהה אומרת שכל נוכחות של שחקן יריב ברדיוס קרוב תקפיץ את משקל ה- $Danger$ לשמיים.

כאשר λ נמוך: הסכנה מהיריבים מקבלת משקל נמוך יותר בהשוואה לפוטנציאל של היישוב, מה שאומר שהפיתוי של היישוב האופטימלי ינצח כמעט תמיד.

ובכן, כפי שניתן לראות, בניגוד לפרמטר ה-"דרך הארוכה ביותר" שכיוונו קודם לכן במאמר, במקרה הזה הפרמטר מוכרח להיות מובנה בתוך האלגוריתם של מציאת הדרכים. כלומר אין פה אפשרות לכבות או להדליק אותו ע"י המשתמש - אחרת כל החישוב יתפספס.

לכן, אנו מוכרחים למצוא ל- λ ערך מספיק מאוזן והוגן עבור "מידת האגרסיביות של האלגוריתם שלנו" במקום שניתן סתם מספר שרירותי. למשל, אם נרצה שהסכנה תגבר על הפיתוי, כלומר במקרה שבו

סכנה במרחק 1 צריכה להיות חזקה ב- $X\%$ מהפיתוי המקסימלי, נוכל כמובן לכוון את הפרמטר שלנו ל- $\lambda = X\% \cdot 100$. כלומר במקרה של 50% לצורך הדוגמא, סיכון הערך יוכל להישאר ב- $\lambda = 1.5$.

אך האם אנו באמת רוצים שהסכנה תגבר על הפיתוי? האם זו באמת האופציה הנכונה?

אולם שאלות פילוסופיות אלה היו מתאימות יותר לפיתוח בוטים ב-Bruteforce ([ראו כאן](#)), אז היינו יכולים לתת לבוטים לשחק אחד כנגד השני ולבדוק המון הרצות ולנסות לשער את מידת הסכנה. אך מנגד, משחק קטאן הוא משחק דינאמי ביותר, וכפי שהתוודה לנו מקודם בבעיית ה-CAS, ייתכן שגם אם נריץ מיליארד סימולציות, לא בהכרח שנמצא פתרון אופטימלי לרוב המצבים. לכן אנו מוכרחים לנסות לחפור עוד יותר עמוק על מנת לבסס את אותו הערך.

למעשה, נרצה לחלק את הבעיה ל-2:

אפשרות א': אם אנו נמצאים כרגע בצומת מסוימת ואנו רוצים לבנות דרך אל היישוב הכי טוב שיש על המפה, **אך הקרבה שלנו אל היריב היא 1!** אנו במצב "מסוכן אך מצוין". כלומר אפשר להגיד שיש פה מצב מסוים של מרוץ בין שני שחקנים, אבל מרוץ ששווה להקריב למענו.

אפשרות ב': לעומת זאת, אם אנו בהזדמנות להרחיב את הדרך אל יישוב שהציון שלו הוא יחסית ממוצע כאשר אין סביבנו אף שחקן שיכול לחסום אותנו, אנחנו במצב בינוני אך עם סיכון בטוח.

למעשה, לשתי האפשרויות שהצגנו יש יתרון מסוים אך הן ההפכים המנוגדים מבחינת העקרון שלהם.

לכן, הפתרון שלנו יהיה **אדיש**, כלומר ננסה למצוא דרך לתת משקל זהה לשתי האפשרויות האלה בהתאם. חשוב לשים לב, מה שזהה זה "המשקל" כלומר מידת החשיבות של האפשרויות. אנו למעשה באים ואומרים שאם שחקן נמצא בסיטואציה של אפשרות א' או אפשרות ב', הוא נמצא באותו משקל בדיוק של דרגת חשיבות. שם יש מרוץ ששווה מאוד להקריב למענו, ושם יש מצב שהוא כה משתלם ובטוח שעבורו גם שווה להקריב למענו.

אז תכירו (מושג חדש) - "נקודת האדישות":

נקודת האדישות (או עקרון האדישות) בתורת המשחקים, מתייחסת לשיווי משקל של אסטרטגיות מעורבות במשחקים, למצב בו שחקן אינו מעדיף אסטרטגיה אחת על פני אחרת, שכן תוחלת התשלום (התועלת) שווה עבור כל האסטרטגיות הטהורות בהן הוא משתמש בהסתברות חיובית. זהו כלי למציאת "שיווי משקל נאש", מצב שבו אף שחקן לא ירוויח משינוי חד צדדי של האסטרטגיה שלו ([להרחבה](#)).

ובמקרה שלנו, אנו רוצים למצוא את באיזה ערך של λ האלגוריתם שלנו יהיה אדיש - כלומר ייתן משקל זהה לשתי אפשרויות קלאסיות בתחילת המשחק. על מנת למצוא את הנוסחה המתאימה ל- λ בבעיה זו, נבצע כמה חישובים מתמטיים.

לפני שנתחיל, נגדיר כמה משתנים:

μ - הציון של יישוב בטוח ממוצע על הלוח באותו הרגע. החישוב מתבצע על ידי מעבר על כל היישובים שנמצאים במרחק בטוח מאיתנו במפה - כלומר לא קרוב מדי, ואז חישוב של הממוצע של ה- $optLocation$ שלהם. (מרחק בטוח זה מצב מרוץ בו אנו קרובים יותר אל הצומת מאשר היריב שהכי קרוב אליו. כך למעשה עוברים על כל צומת במפה ויודעים איזה צמתים הם נמצאים במכה שמכונה "מרחק בטוח").

MaxOpt - הציון של היישוב הפנוי הכי אופטימלי שנמצא על המפה ברגע הנתון, לפי האסטרטגיה שלנו.

d_{ref} - המרחק בין היריב שהכי קרוב אל הצומת **MaxOpt** (הכי טובה על המפה).

נציב את שתי האפשרויות שלנו בנוסחת משקל הדרך שמצאנו קודם לכן. כזכור, האלגוריתם של דייקסטרה מחפש את המשקל הנמוך ביותר, לכן נשווה ביניהם כדי למצוא את נקודת האיזון של λ .

- עבור אפשרות א' (מצב מעולה עם סיכון גבוה):

נניח שהמרחק של היריב מהצומת הכי טובה על המפה הוא 2, כלומר $d_{ref} = 2$:

$$W(Risky) = W_{BaseCost} + \frac{MaxOpt \cdot \lambda}{2^2} - MaxOpt$$

- עבור אפשרות ב' (מצב בינוני עם סיכון בטוח):

בצומת הבטוח היריב רחוק מאוד ולכן סכנת החסימה מתאפסת.

$$W(Safe) = W_{BaseCost} + 0 - \mu$$

- נשווה כעת בין המשוואות על מנת לבודד את λ :

$$W_{BaseCost} - \mu = W_{BaseCost} + \frac{MaxOpt \cdot \lambda}{4} - MaxOpt$$

- נצמצם את $W_{BaseCost}$ משני האגפים ונעביר אגפים:

$$MaxOpt - \mu = \frac{MaxOpt \cdot \lambda}{4}$$

- נכפיל ב-4 ונחלק ב- $MaxOpt$ כדי לבודד סופית את λ :

$$\lambda = 4 \cdot \left(\frac{MaxOpt - \mu}{MaxOpt} \right)$$

- או בצורה האלגנטית יותר, בסופו של דבר נקבל:

$$\lambda = 4 \cdot \left(1 - \frac{\mu}{MaxOpt} \right)$$

והנה מצאנו את הערך של λ . ומה זה בעצם אומר לנו? זה אומר שהערך הוא אינו מספר קבוע, אלא פונקציה של התפלגות היישובים האופטימליים על המפה בהתאם לדרגת הסיכון.

חשוב לשים לב לכמה דברים מעניינים שקורים פה:

ראשית, אם המיקום הממוצע (μ) הוא כמעט באותה האיכות כמו המיקום המקסימלי (MaxOpt), הרי שהיחס ביניהם שואף ל-1. ובמצב כזה ה- λ ישאף ל-0. המשמעות המתמטית היא שבמצב כזה, אין סיבה להילחם ולהסתכן על המיקום הכי טוב, כי גם המקומות הבטוחים מצוינים. כלומר במצב כזה קרבה לאויב לא שווה את הסיכון כלל.

שנית, אם המקומות הממוצעים גרועים בהרבה בהשוואה ל-MaxOpt, אז היחס $\frac{\mu}{MaxOpt}$ יהיה קטן יותר, למשל: (0.2), במצב כזה נקבל $\lambda = 4 \cdot (1 - 0.2) = 3.2$, ואכן המשמעות המתמטית היא ש-"המקומות הבטוחים הם גרועים, ולכן עלינו לקחת סיכון גבוה פי כמה על מנת להגיע למקום טוב".

$$\lambda = d_{ref}^2 \cdot \left(1 - \frac{\mu}{MaxOpt}\right) \quad \text{והרי לנו:}$$

הנוסחה שלנו התקדמה שלב אחד קדימה אל עבר הסיום, נציב את λ במשוואה:

$$W(u, v) = \max \left(\epsilon, 1 + \left(\frac{MaxOpt \cdot d_{ref}^2 \cdot \left(1 - \frac{\mu}{MaxOpt}\right)}{d_{opp}^2} \right) - FinalOptLocation(v) \right)$$

נצמצם (נפתח סוגריים ונכפיל את MaxOpt פנימה):

$$W(u, v) = \max \left(\epsilon, 1 + \left(\frac{d_{ref}^2 \cdot MaxOpt - MaxOpt \cdot \frac{\mu}{MaxOpt}}{d_{opp}^2} \right) - FinalOptLocation(v) \right)$$

והנה קיבלנו את הנוסחה שלנו עבור כל דרך במפה:

$$W(u, v) = \max \left(\epsilon, 1 + \left(\frac{d_{ref}^2 \cdot (MaxOpt - \mu)}{d_{opp}^2} \right) - FinalOptLocation(v) \right)$$

אבל רגע אחד (אחרון, מבטיח (לפרק זה))!

עד כה למדנו כיצד לתת מחיר לדרך שלנו מבחינת רמת הסיכון של הקרבה ליריב שלנו ביחד עם האופטימליות של היישוב. אבל עוד לא טיפלנו בצורה כירורגית מספיק ברדיוס החיפוש מאיתנו.

הרי בואו נחשוב, אם אנחנו נמצאים בנקודה מסוימת ואנו רצים על כל הדרכים האפשריות, אנו מוכרחים בכל זאת לתת מחיר למרחק הריצה שלנו, כדי שלא נבחר ביישוב אופטימלי שעונה על הצרכים - אבל נמצא במרחק של 7 דרכים מאיתנו. אחרי הכל, אנו צריכים לתעדף רק את היישובים שבקרבתנו, יחד עם השקלול של כל שאר הגורמים.

כדי לגרום לאלגוריתם להבין שהסיכויים פוחתים ככל שמתרחקים, אנו לא משנים את ה- λ שחישבנו זה עתה (שהיא מדד לרמת הסיכון הכללית של הלוח), אלא שואלים מושג קלאסי בשם "פקטור דעיכה" ([ראו כאן](#)) מעולם ה-[reinforcement Learning](#) (למידת חיזוק), או באות היוונית γ (גמא).

"למידת חיזוק" הוא תחום במדעי המחשב של למידת מכונה הנוגע לאופן שבו סוכנים פועלים באופן שימקסם את הרווח המצטבר כתוצאה מהפעולות הללו. זה אולם תיאור כללי, אך אני ממליץ לכם להתעמק בזה עוד בקישורים שמופיעים למעלה.

בכל אופן, הרעיון הוא שערך של יישוב עתידי יהיה "שווה פחות" ככל שהוא רחוק מאיתנו במרחב. אנו נגדיר את גמא γ כשבר חיובי קטן מ-1 (למשל 0.85). כעת, נעדכן את הציון של היישוב ($OptLocation$) כך שידעך מערכית ככל שהמרחק שלנו אליו (d_{us}) גדל:

$$AdjustedOpt(v) = OptLocation(v) \cdot \gamma^{d_{us}}$$

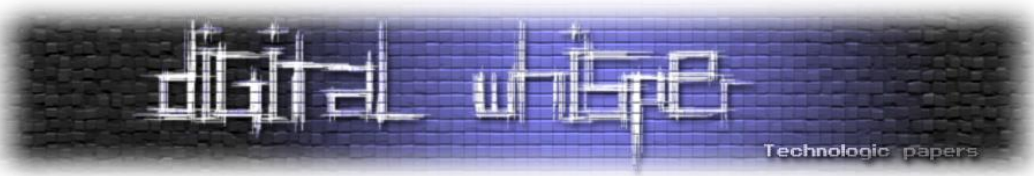
חשוב כמובן לזכור שהערך של $OptLocation$ הוא (כפי שהסברתי קודם) תלוי בגוש/רכיב הקשירות שבו הוא נמצא. לכן מצב שבו נטייל 7 דרכים על המפה למציאת יישוב אופטימלי הוא מאוד לא סביר. מנגד, במצבים בהם רכיבי הקשירות כשלעצמם (כלומר ברכיב קשירות נפרד) ענקיים מאוד, אנו נמצא את עצמנו במצב שנדדקק מאוד לפקטור הדעיכה, כדי שלא נכון לזוויות של יישובים שרחוקים מאוד מאיתנו באותו הרכיב קשירות.

אז כעת, יש לנו פקטור סכנה λ . ופקטור דעיכה γ .

אך כמו קודם לכן, נרצה גם לקבוע את פקטור הדעיכה שלנו כך שהוא לא ייבחר באופן שרירותי. לכן, בדיוק כפי שעשינו קודם, נרצה להגדיר "נקודת אדישות" נוספת. במקרה זה, נרצה להגדיר חסם עליון כדוגמת מהו המרחק המקסימלי (בכמות הדרכים) שנסיים לסלול בתחילת המשחק בשביל היישוב המושלם. לצורך כך, נגדיר פרמטר שנקרא D_{max} (למשל, 4 דרכים).

הבהרה קטנה: כמובן שלשחקן קטאן מנוסה יש אינטואיציה כללית לגבי זה. בתחילת המשחק ניסיון לרצות לבנות 4 או 5 דרכים רצופות אל עבר יעד רחוק הוא לרוב התאבדות אסטרטגית (זה שורף המון משאבים וחושף אותך לחסימות עתידיות של שחקנים אחרים). מה גם שבחירת היישובים שלנו קודם לכן הסתמכה על מקבץ (רכיב קשירות) של יישובים טובים שקרובים זה לזה. לכן מאוד לא סביר שבכלל נרצה לבנות מספר גבוה של דרכים. למעשה, המצב הכי מתאים למקרה זה יהיה כאשר המפה מסודרת בצורה כל כך גרועה, כך שסידור המספרים והמשאבים על המפה מגבילים אותנו עד כדי כך. אך מכיוון שאנו שואפים למוצלחות ב-"לשבור את קטאן", אנו נפתור גם את הבעיה הזאת בצורה מתמטית.

כעת, אנו רוצים שבאותו מרחק מקסימלי (D_{max}), הציון של היישוב המושלם ($MaxOpt$) ידעך עד כדי כך שהוא יהיה שווה בדיוק לציון של יישוב בטוח וממוצע על המפה u (כפי שהגדרנו קודם). במילים אחרות, גם במרחק הזה אנו נהיה אדישים בין האפשרויות ונתאים את הדעיכה שלנו לסגנון של המפה, כמו מקודם.



- לכן נציב זאת במשוואה:

$$MaxOpt \cdot \gamma^{D_{max}} = \mu$$

- כעת נחלץ את γ :

$$\gamma = \sqrt[D_{max}]{\frac{\mu}{MaxOpt}}$$

והנה מצאנו את הערך של γ !

לשם הדוגמא: אם ניקח לוח קטן ממוצע, בו הציון הממוצע הבטוח ליישוב הוא $u = 1.217$, הציון המושלם ליישוב על הלוח הוא $MaxOpt = 3.32$, ואנו קובעים שהמרחק המקסימלי המוצדק שכדאי להשקיע בתחילת המשחק הוא 4 דרכים ($D_{max} = 4$), נציב את הנתונים בנוסחה:

$$\gamma = \sqrt[4]{\frac{1.217}{3.32}} = \sqrt[4]{0.366} = 0.778$$

בעמוד הבא ניתן לראות את הבדיקה (האמיתית!) שערכתי על לוח של קטן לדוגמא:

```
---[Final Optimal Location Values for Each Town]---
```

```
--- Starting Final Optimal Location Verification ---
```

```
Win Condition Index: 3
```

Town [0]	Final Opt Location Value: 0.0
Town [1]	Final Opt Location Value: 0.3779553478878975
Town [2]	Final Opt Location Value: 0.6356425943298072
Town [3]	Final Opt Location Value: 0.0
Town [4]	Final Opt Location Value: 0.3779553478878975
Town [5]	Final Opt Location Value: 1.0135979422177046
Town [6]	Final Opt Location Value: 0.6356425943298072
Town [7]	Final Opt Location Value: 1.0245636288796802
Town [8]	Final Opt Location Value: 1.6492405365475118
Town [9]	Final Opt Location Value: 2.284883130877319
Town [10]	Final Opt Location Value: 1.4971944770540675
Town [11]	Final Opt Location Value: 1.0245636288796802
Town [12]	Final Opt Location Value: 2.2958488175392944
Town [13]	Final Opt Location Value: 2.9205257252071264
Town [14]	Final Opt Location Value: 2.7684796657136816
Town [15]	Final Opt Location Value: 0.8615518827242603
Town [16]	Final Opt Location Value: 1.602062232094874
Town [17]	Final Opt Location Value: 3.320412446418975
Town [18]	Final Opt Location Value: 2.5425703773192287
Town [19]	Final Opt Location Value: 2.1328370713838747
Town [20]	Final Opt Location Value: 1.2395072306121577
Town [21]	Final Opt Location Value: 0.6356425943298072
Town [22]	Final Opt Location Value: 2.6847698520891674
Town [23]	Final Opt Location Value: 2.2958488175392944
Town [24]	Final Opt Location Value: 1.2712851886596144
Town [25]	Final Opt Location Value: 1.2395072306121577
Town [26]	Final Opt Location Value: 0.3779553478878975
Town [27]	Final Opt Location Value: 0.6356425943298072
Town [28]	Final Opt Location Value: 2.109673412071104
Town [29]	Final Opt Location Value: 1.0245636288796802
Town [30]	Final Opt Location Value: 0.7559106957757951
Town [31]	Final Opt Location Value: 1.4469745057969092
Town [32]	Final Opt Location Value: 0.3779553478878975
Town [33]	Final Opt Location Value: 1.0851097831914236
Town [34]	Final Opt Location Value: 1.4740308177412966
Town [35]	Final Opt Location Value: 0.7559106957757951
Town [36]	Final Opt Location Value: 1.8249298536848069
Town [37]	Final Opt Location Value: 1.4469745057969092
Town [38]	Final Opt Location Value: 0.44946718886161635
Town [39]	Final Opt Location Value: 0.898934377232327
Town [40]	Final Opt Location Value: 0.7559106957757951
Town [41]	Final Opt Location Value: 2.5808405494606017
Town [42]	Final Opt Location Value: 1.0690191579090118
Town [43]	Final Opt Location Value: 0.898934377232327
Town [44]	Final Opt Location Value: 0.44946718886161635
Town [45]	Final Opt Location Value: 1.5118213915515901
Town [46]	Final Opt Location Value: 1.8249298536848069
Town [47]	Final Opt Location Value: 0.44946718886161635
Town [48]	Final Opt Location Value: 0.44946718886161635
Town [49]	Final Opt Location Value: 0.7559106957757951
Town [50]	Final Opt Location Value: 0.7559106957757951
Town [51]	Final Opt Location Value: 0.44946718886161635
Town [52]	Final Opt Location Value: 0.0
Town [53]	Final Opt Location Value: 0.7559106957757951

ואכן קיבלנו שקצב הדעיכה שלנו במרחק עבור בחירת הדרך ליישוב האופטימלי, תלוי בהתפלגות של הציון של היישובים על גבי המפה שלנו (u מול $MaxOpt$).

ואכן, אם הלוח יהיה שוויוני יותר (ה- u קרוב מאוד ל- $MaxOpt$), אז המשוואה תניב γ גדול יותר (קרוב ל-1), והאלגוריתם "יתעצל" ללכת רחוק כי למעשה אין בכך באמת טעם. אך לעומת זאת, אם הלוח יהיה פחות שוויוני עם רק פינה בודדת טובה, ה- γ יהיה נמוך יותר, והאלגוריתם יסכים לנדוד למרחקים כדי להגיע לשם.

אז הנה לנו הנוסחה:

$$AdjOpt(v) = \frac{OptLocation(v)}{MaxOpt} \cdot \gamma^{d_{us}}$$

ה- $OptLocation(v)$ צריך להיות מנורמל יחסית ל- $MaxOpt$ כאן, כך שהתמריץ יישאר תמיד באותה הסקאלה כמו ה- $BaseCost$ ולא יעלה עליו באופן בלתי מבוקר.

באופן דומה, מכיוון ש- $AdjustedOpt(v)$ כבר מנורמל יחסית ל- $MaxOpt$ (כפי שהגדרנו קודם), עלינו לבצע את אותה נורמליזציה גם ל- $Danger$ בשלב הזה. אחרת, שני האיברים לא יהיו על אותה הסקאלה.

שוברים את קטן

www.DigitalWhisper.co.il

$$\frac{\text{Danger}(v)}{\text{MaxOpt}} = \frac{\lambda}{d_{\text{opp}}^2}$$

את הגרסה המנומלת נציב בהמשך בנוסחה הסופית.

$$\text{AdjOpt}(v) = \frac{\text{OptLocation}(v)}{\text{MaxOpt}} \cdot \left(d_{\text{max}} \sqrt{\frac{\mu}{\text{MaxOpt}}} \right)^{d_{us}}$$

כאשר שוב, d_{us} מציין את המרחק היחסי אל היישוב.

אז אנו עוברים על כל היישובים הפוטנציאליים ה**סובבים** את היישוב ההתחלתי שכבר בחרנו בו, ואז הרדיוס (מס' הדרכים שמפרידים ביננו) אליהם מאיתנו נקבע עבור כל אחד על ידי הערך d_{us} .

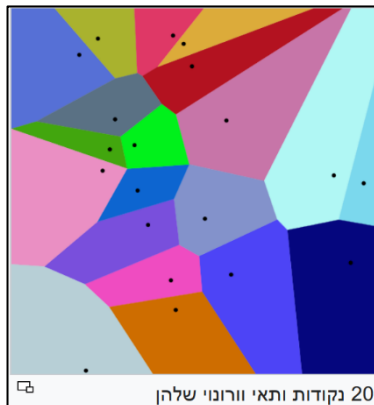
ובנוסף, הערך D_{max} כפי שאמרנו, קובע את המרחק המקסימלי של הדרכים שאנו מוכנים לבנות על מנת להגיע ליישוב יעד כלשהו. אנו נסיים את החישובים עם דבר אחד אחרון, וזה מציאת הערך של D_{max} .

כפי שעשינו עד כה, ננסה גם לתחם את D_{max} :

(נזכיר, D_{max} מייצג את המרחק המקסימלי, במספר הדרכים, שאנו מוכנים להשקיע ולסלול בתחילת המשחק כדי להגיע ליישוב האופטימלי). על אף שלכל שחקן בר דעת ברור שבזמן הבחירה הראשונית של דרך במשחק, לא יהיה חכם לבסס בחירה זו על גבי אותו יישוב פוטנציאלי שנמצא במרחק של 7 או 8 דרכים מאיתנו. 4 דרכים הוא מרחק מאוד הגיוני בשביל למקם כמקסימלי. אבל כפי שהסברתי קודם, ייתכנו מצבים מאוד מאוד מסוימים, בהם הגוש (רכיב קשירות) שלנו יתפרס על פני מרחקים יחסית גדולים, ובמקרה הזה נרצה למנוע טעות חישוב או תיעדוף של יישובים מאוד רחוקים.

אז על מנת לפתור בעיה זו, נעשה את מה שעשינו עד כה, ונבסס את הערך הזה על "נקודת האדישות". אם נקבע את הערך באופן הזה, נוכל אף להפעיל את האלגוריתם שלנו על מפות מגוונות, בגדלים שונים, ואפילו להתמודד עם שינוי המפה אחרי התורות הראשונים, כמו למשל בתור ה-3 או ה-50, שם הבחירות נעשות כבר מסובכות יותר ויותר מבחינה לוקאלית (מקומית).

כדי לעשות את זה נכון, ניתן ללוח להגדיר לנו מה הגבול המקסימלי האפשרי ונעשה זאת באמצעות דבר שנקרא "**דיאגרמת וורוני**" (או בהקשרנו: חלוקה טריטוריאלית).



לרקע כללי, במתמטיקה, דיאגרמת וורוני (Voronoi diagram) היא חלוקה של המישור לאזורים המבוססת על מרחק לנקודות השייכות לחלק מסוים של המישור. קבוצת הנקודות הזו (הנקראות גם זרעים, אתרים או מחוללים) מוגדרת מראש, ולכל "זרע" יש אזור מתאים המורכב מכל הנקודות הקרובות יותר לזרע הזה מאשר לזרעים אחרים. אזורים אלו נקראים "תאי וורוני". דיאגרמת וורוני של קבוצת נקודות היא דואלית לשילוש דלוני (Delaunay triangulation) של אותה קבוצת נקודות. לדיאגרמת וורוני יש שימושים רבים בסטטיסטיקה מרחבית.

אז איך זה עובד?

זוכרים שקודם לכן, על מנת למצוא את הממוצע (μ) סיננו מתוך כל הלוח את ה"צמתים הבטוחים" - כלומר אלו שאנחנו קרובים אליהם יותר מכל יריב אחר?

ובכן, הרשימה הזו של הצמתים מתארת, הלכה למעשה, את "אזור השליטה התיאורטי" על הלוח באותו הרגע. המשמעות של אותו "אזור שליטה" זה ש-אילו היה לנו מספר בלתי מוגבל של אפשרויות בחירה, יכולנו להגיע אל כל אחד מהם בפוטנציה **עוד לפני שאר היריבים**, כך בלי שאף אחד היה מפריע לנו.



ה- μ נתן לנו את הציון הממוצע של הטריטוריה הזו, כזכור: "הציון של ישוב בטוח ממוצע על הלוח באותו הרגע". כעת, כל מה שנעשה זה פשוט לשאול שאלה אחרת: "מה היישוב הכי רחוק שנמצא בתוך הטריטוריה הבטוחה שלנו?"

מבחינה מתמטית (לסקרנים שביניכם), נגדיר את D_{max} כך:

$$v \in SafeNodes$$

ועכשיו, ברגע שאנו תולים את ערך ה- D_{max} במרחב הבטוח שלנו, פקטור הדעיכה γ הופך להיות רגיש לצפיפות של הלוח. ההתאמה הזאת כאמור מאפשרת לנו לפעול במצבים דומים גם בשלבים מאוחרים יותר של המשחק (על אף שלא בזה נדון במאמר).

במקרה של לוח צפוף ואגרסיבי (לרוב בשלב מאוחר של המשחק): המרחב הבטוח יהיה קטן יותר, נניח שהיישוב הבטוח הכי רחוק מאויבים הוא רק במרחק של 2 דרכים ($D_{max} = 2$), נציב 2 בנוסחת השורש ונקבל γ קטן מאוד. כלומר אנו נקנוס "באכזריות" כל מחשבה על מסע ארוך, ונבחר להתבצר ולנצל את מה שקרוב יותר.

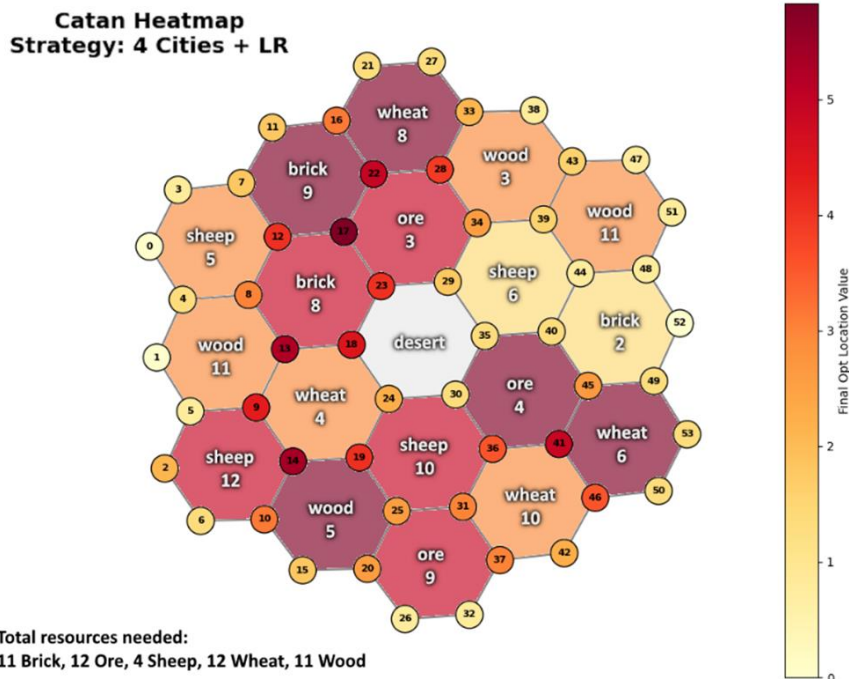
אך במקרה של לוח פתוח ויחסית ריק (לרוב בתחילת המשחק): המרחב הבטוח שלנו משתרע הרחק קדימה. נניח שהיישוב הבטוח הכי רחוק נמצא במרחק של כ-6 דרכים מאויב ($D_{max} = 6$). כאשר נציב 6 בנוסחת השורש נגלה ש- γ (פקטור הדעיכה) קרוב מאוד ל-1. כלומר האלגוריתם מזהה שיש לו אופק תכנון רחב יותר ולכן הוא לא יוגבל לתכנון מסלול יחסית יותר ארוך אל עבר יישוב מדהים.

הבהרה קטנה על החשיבות וההבדל:

כמו שאמרנו קודם לכן, הערך של היישובים $AdjustedOpt(v)$ יקטן באופן טבעי ככל שנתרחק ממנו, בגלל מקדם הדעיכה ($\gamma^{d_{us}}$) שמופעל עליו. הוא הכי אופטימלי מבחינת הציון שמתחשב גם בגוש של רכיב הקשירות שבו אנו נמצאים.

ובכן זה בוודאי כבר ברור לנו, הרי שהיישוב אותו בחרנו כבר בשלב הקודם הוא במובן מסוים מסתמך כבר מראש על ההנחה שזה "מרכז האימפריה שלנו". זה לא מרכז האימפריה במובן שבו באמת חישבנו את

המרחק מיריבים אחרים, אלה כי תיעדפנו יישובים שסובבים אותם רכיבי קשירות עם ציון גבוה יחסית. וזה שם אותנו בסבירות יחסית גבוהה להיות מרכז האימפריה - אך בזמן סטטי ונתון בסריקה של המפה. את זה כבר פתרנו בפרק הקודם (כפי שניתן להבחין ביישוב מס' 17 שהוא הכי אופטימלי וגם מרכז האימפריה, בתמונה המצורפת).



עם ההצבה של היישוב הראשון, אנו מיד מוגבלים לנקודה סטטית אשר ממנה אנחנו מוכרחים לבחור בין כמה כיוונים שונים. אולם, בבחירת הדרך, אנו נמצאים במצב אחר לגמרי, בו אנו נאלצים לחזות מצב דינמי ומשתנה על המפה (גם בפוטנציה, כלומר בעוד X תורות), מה שמקשה עלינו לתת פתרון סטטי. המצב הזה נקבע על פי בחירתם העתידית של שחקנים אחרים.

ובעקבות זאת, ודווקא בניגוד למה שנעשה בפרק הקודם, האופטימיזציה שנעשה פה, נוגעת לשילוב בין "מרכז האימפריה של היישוב הראשון" (רכיב קשירות) שלנו (שמוגדר על ידי הציון שקבענו ליישוב *OptLocation*) ולבין "המרחב הבטוח שלנו" מיריבים אחרים. שני המרחבים הנפרדים הללו אולם, אינם תלויים זה בזה כלל, ולכן נדרש אותו איזון ביניהם שעשינו עד כה.

נסגור עם פרט קטן אך חשוב שממנו התעלמנו. וזה עלות ה-BaseCost שלנו (או כזכור, 1). עלות זו עד כה מגלמת עלות דרך בסיסית. אך פה טמונה נקודה קריטית, האסטרטגיה שבה כבר בחרנו מגלמת בתוכה את עלות המשאבים הנדרשים לסלילת דרכים (כפי שהצגתי בטבלת האסטרטגיות בתחילת המאמר).

הציון של *FinalOptLocation* כבר "יודע" מראש האם משתלם לנו כלכלית לסלול דרכים ארוכות או לא בהתאם לאסטרטגיה שבחרנו בה. מסיבה זאת בחרנו להגדיר פשוט שהעלות הבסיסית של דרך היא $BaseCost = 1$. מטרת ערך זה היא אך ורק לספור צעדים פיזיים במרחב, כך שאלגוריתם דייקסטרה יעדיף באופן טבעי מסלולים קצרים יותר, אלא אם כן עקיפה תציל אותנו מסכנת חסימה כבדה (כפי שפירטתי בתחילת הפרק).

וכעת סוף כל סוף הגיע הזמן לחגוג עם שילוב של כל הנוסחאות יחד וכל הנקודות שחיברנו עד כה!

כעת, לאחר שעברנו על כל המשתנים, וכאשר הם מנומלים ומתחשבים כולם במידע הדטרמיניסטי שקיים על הלוח, נוכל להגדיר את המשקל הסופי של כל קשת (דרך) בגרף $W(u, v)$, מצומת u ליעד v , שעליו ירוץ אלגוריתם דייקסטרה בפועל. נעבוד בשלבים:

1. נמצא את ה-"מרחב הבטוח שלנו", ונגדיר:

$$D_{max} = \max(d_{us}(v)) \text{ for all } v \in \text{SafeNodes}$$

d_{us} - המרחק של כל יישוב מאיתנו.

2. נעבור על כל היישובים שנמצאים במסלול בטווח ההגעה מאיתנו, ונשנה את הערך:

$$\text{AdjustedOpt}(v) = \frac{\text{OptLocation}(v)}{\text{MaxOpt}} \cdot \left(D_{max} \sqrt{\frac{\mu}{\text{MaxOpt}}} \right)^{d_{us}}$$

MaxOpt - ה- OptLocation המירבי שיש ליישוב על המפה.

μ - ה- OptLocation הממוצע של כל היישובים ב-"מרחב הבטוח" מהגבול עם יריביהם.

3. עבור כל דרך (קשת) במפה (גרף), נגדיר את הערך הבא:

$$W(u, v) = \max(\epsilon, \text{BaseCost} + \text{Danger}(v) - \text{AdjustedOpt}(v))$$

$$W(u, v) = \max\left(\epsilon, 1 + (d_{ref}^2 \cdot \frac{1 - \frac{u}{\text{MaxOpt}}}{d_{opp}^2}) - \text{AdjustedOpt}(v)\right)$$

או בצורה הפשוטה יותר, מכיוון ש- $\lambda = d_{ref}^2(1 - \frac{u}{\text{MaxOpt}})$,

$$W(u, v) = \max\left(\epsilon, 1 + \frac{\lambda}{d_{opp}^2} - \text{AdjustedOpt}(v)\right)$$

d_{ref} - מרחקו של היריב הקרוב ביותר אל הצומת הטוב ביותר על הלוח (MaxOpt).

ϵ - ערך מינימלי שואף ל-0.

d_{opp} - מרחקו של היריב הכי קרוב לצומת v .

הבהרה: השימוש בפונקציית ה- $\max$ (כאשר ϵ הוא מספר חיובי קטן, למשל 0.1) הוא קריטי אלגוריתמית: הוא שמשקל הדרך אף פעם לא יהיה שלילי (וכזכור, אלגוריתם דייקסטרה קלאסי קורס עם משקלים שליליים).

4. נריץ את אלגוריתם דייקסטרה מהיישוב הראשון אל כל שאר היישובים הפוטנציאליים:
מבין כל המסלולים שמצאנו, נבחר את המסלול בעל העלות המצטברת הכי נמוכה (הזול ביותר).

5. הצעד הראשון במסלול שמצאנו הוא יהיה הכיוון שבו נציב את הדרך שלנו, בסמוך ליישוב הראשון.

ולבסוף, זו תהיה הדרך האופטימלית.

ובזאת אני גאה להכריז שאנו יכולים לעבור למימוש עצמו שכל כך חיכינו אליו!

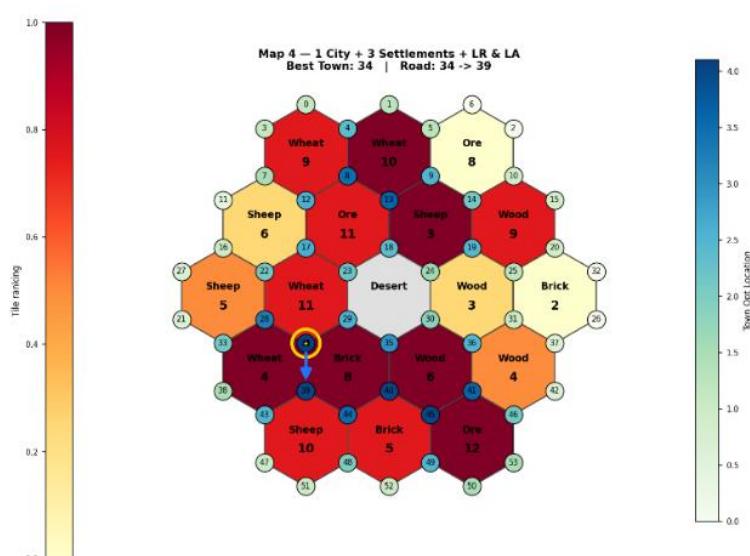


המימוש:

לשם המימוש השתמשתי לראשונה בפרויקט זה ב-AI. במקרה הזה בחרתי ב-Claude (השתמשתי לרוב ב-Sonnet 5-High) והעברתי לו את המסמך הזה ואת שאר הקוד שכתבתי. תוך כדי השיחה עם הצ'אט, הרצתי דגימות, סימולציות, בדיקות ובעיקר בדקתי שמה שהוא מייצר תואם בדיוק לחזון ולאגוריתם שנכתב פה. בחרתי לא לרדת לעומקי הקוד כמו בפרקים הקודמים, משום שאין בקוד היבט מיוחד או מחדש מלבד לכך שהוא בסך הכל מממש את הפסאודו-קוד בדיוק כפי שנכתב פה למעלה, פשוט בשפת Java (בפועל ניתן לממש זאת בכל שפת תכנות אחרת לבחירתכם). למי שמעוניין, פירוט מלא נמצא ב-GitHub בסוף המאמר.

התוצאות:

במפה זו תוכלו לראות את המשושים מדורגים לפי מפת החום שיצרנו מקודם, את היישובים האופטימליים לפי דירוג, ואיך לא, כמובן הדרך האופטימלית (בחץ כחול). בדוגמא זו, היישוב האופטימלי שלנו הוא בין "11 חיטה", "8 בנייה" ו-"4 חיטה", פה נבחרה הדרך דווקא לכיוון הדרומי.

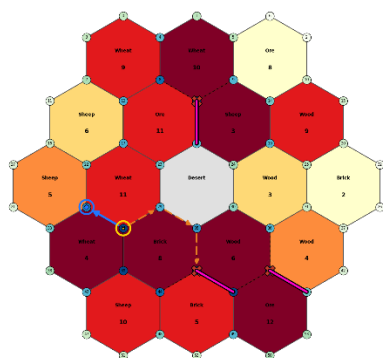


אפשר לתת לכך כמה סיבות: הראשונה שבהן היא שדרך דרומית מאפשרת לנו לאגף את "8 בנייה" מדרום כדי שתהיה לנו האפשרות להציב יישוב נוסף מסביבו, כזה שלא גובל במשושה המדבר (חשוב להזכיר ש-8 הוא מספר נחשב מאוד בקובייה). סיבה נוספת שניתן לחשוב עליה היא עצם העובדה שבניגוד לדרך צפון מזרחית לכיוון יישוב 29 (שגם שם מקיפים את "8 בנייה"), במקרה הזה ההתקדמות למטה תאפשר לנו לגשת דווקא ל-"5 בנייה" בעתיד, המס' 5 הוא במקום השני מבחינת החשיבות בקובייה.

ומעבר לכך, לפי הצבע של היישובים, ניכר כי המסלול העתידי שהאלגוריתם "תכנן" הוא כזה שמתעדף מעבר ארוך יותר בין יישובים נחשבים כדי להגיע אל היעד "הדרך הארוכה ביותר" (שהיא גם חלק מהאסטרטגיה) וגם על מנת להשיג עוד מה-"6 עץ" ו-"8 בנייה", שאלו מספרי קובייה טובים ביותר.

ומה קורה ברגע שמוסיפים אויבים ללוח?

את ההמחשה לרעיון הזה תוכלו לראות בתמונה הבאה שמציגה את אותה המפה בדיוק ובה המסלול דייקסטרה שהאלגוריתם בחר בו (על סמך מסלול זה נבחר הכיוון להציב בו את הדרך) והוא מסורבל יותר. בנוסף, בדוגמא הזו **אנו לראשונה מוסיפים אויבים ללוח**, זאת משום שהאלגוריתם שכתבנו למציאת הדרכים יודע להתמודד איתם. בשלב הזה לראשונה, אנחנו לא בוחרים את היישוב ואת הדרך ראשונים במעגל, ויריבים אחרים בחרו אותם לפנינו, אותן מקומות מוסתרים במפה מבחינת האלגוריתם.



פה "בעיית הנוסע הקנדי" חוזרת ובגדול. התמונה הזו נהדרת כיוון שהיא ממחישה את השוני שנוצר ברגע שמוצבים יריבים על הלוח. במקרה הזה נבחרה דווקא הדרך השמאלית, זאת למרות שבפועל הוכחנו בתמונה הקודמת שהיינו מעדיפים דווקא להתקדם למטה (אילו המפה הייתה ריקה). ולמרות זאת, אילו היינו מתקדמים מטה על מנת להקיף את "8 בנייה" וחושבים להקים את יישוב 44, זה היה לא חוקי כיוון שישנו יישוב במרחק דרך אחת שצמוד אלינו. לעומת זאת, אילו היינו מתקדמים ימינה, היינו יכולים להציב יישוב אך ורק ב-23, מה שאומר שהיינו מוותרים על משאב נוסף כיוון שזה צמוד למשושה המדבר.

לכן האפשרות האלטרנטיבית היחידה שהאלגוריתם למציאת הדרכים בחר בשלב הזה הוא החץ השמאלי העליון - היישובים 33 או 22. אולם חשוב לציין כי עד כאן מדובר בפרשנות אנושית שלנו לבעיה, אך היופי שבדבר זה שהאלגוריתם שלנו יודע להתמודד עם השאלות הללו באופן אוטומטי לגמרי. ואת זה הראנו בעמודים הקודמים.

הדיבור על היריבים שעל המפה מוביל אותנו לנושא הבא והחשוב ביותר, והוא...

מציאת היישוב השני

בשלב הזה נרצה למצוא את היישוב השני. כלומר אנחנו מניחים שכל השחקנים (כולל אנחנו) כבר סיימו להציב את היישובים והדרכים שלהם על המפה. ישנם כמה אתגרים להתמודד איתם (אולם לא ברמה שהייתה עד עכשיו, אך בכל זאת...).

למעשה, כעת המפה עמוסה ביישובים ובדרכים של יריבים, מה שמצמצם משמעותית את טווח הפעולה שלנו. לכן אנחנו נאלצים לגזור 3 אלמנטים עיקריים מהמפה שלנו:

1. היישובים של היריבים
2. הדרכים של היריבים
3. הדרכים הסמוכות ליישובים של היריבים (שהרי לפי החוקים כל שני יישובים במרחק שני דרכים ומעלה)

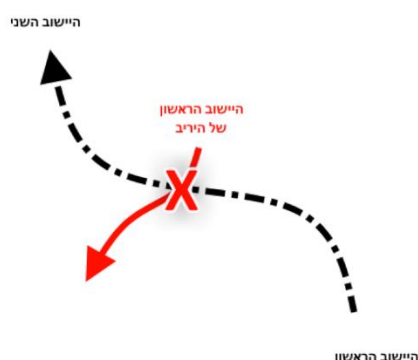
בשלב הזה אנחנו לראשונה מתמודדים עם מצב חדש במאמר שלא עסקנו בו כלל. והוא **להחליט החלטה שמתבססת על החלטה קודמת שביצענו** (וזאת לאחר שעברו מס' תורות בין ההחלטות שאנו ביצענו).

בואו ננסה לתקוף את זה בדיון תיאורטי:

המצב הזה בעייתי שכן בהתחלה הגדרנו את המחקר שלנו להיות היוריסטי, כלומר כזה שבוחן את המצב הנתון כרגע ורק זה בלבד. כעת, כשאנו מרחיבים את המחקר שלנו לממדים של קביעת החלטות שמבוססות על החלטות קודמות, אנו מתחרים עם שיטות אלגוריתמיות חזקות הרבה יותר, ומדוע?

הרי נניח שהחלטנו שתי החלטות שונות (הראשונה - מציאת היישוב הראשון, והשנייה מציאת היישוב השני), פה אנו מניחים שהמחשבה על היישוב השני התבססה על המצב הכי כדאי באותו הרגע על בסיס המפה. אך מנגד, ייתכן ואם היינו יודעים לקבל את המידע הזה בזמן הצבת היישוב הראשון, היה לנו יותר מידע חבויה שהיינו יכולים להשתמש בו מראש ולשנות את הבחירה שלנו בהתחלה - מה שהיה משפיע על הבחירה השנייה.

אבל פה אפילו לא נמצאת הבעיה המרכזית. הבעיה העיקרית שלנו היא היתרון שהיריבים שלנו מקבלים במרווח הזמן הקטן שבין הבחירה הראשונה והשנייה שלנו. בזמן הזה הם יכולים להרוס לנו בחירות פוטנציאליות שעליהן התבססה כל האסטרטגיה שלנו להצבת הדרכים והיישובים בהתחלה.



למשל: אם תכננו ליצור מסלול בין שני יישובים כבר בתור הראשון (לפי מפת החום של האלגוריתם), הרי שהרעיון נכשל כי היריב הרס לנו אותו באמצע, עוד לפני שהספיק להגיע התור שלנו (הערה: זה מצב נפוץ מאוד).

אז מה הפתרון?

הפתרון:

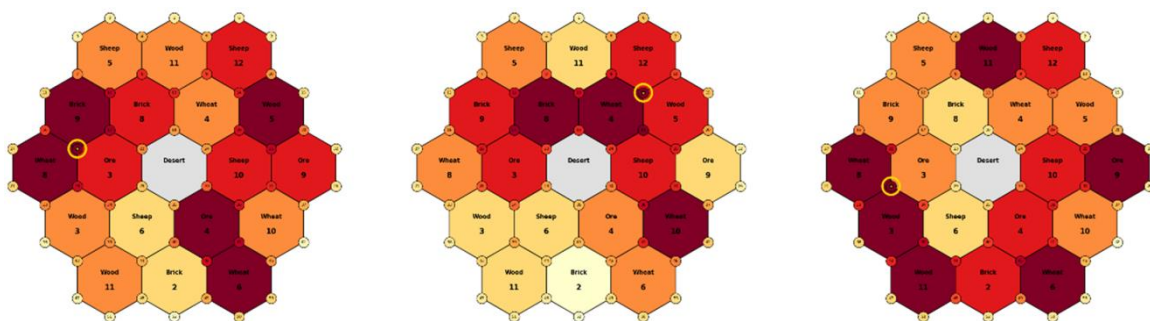
הפתרון הוא לא לנסות לחזות שום דבר שישפיע על ההחלטה הראשונה באופן עקיף (הרי שזה יפגע בעקרון היוריסטי שלנו), אלא במקום זאת, אנו פשוט נעבוד בשלבים הבאים:

1. נחתוך מהמפה את כל היישובים, הדרכים, והדרכים התפוסות שצמודות לדרכים של היריבים, בכל רחבי הלוח.
2. נפעיל שוב את אלגוריתם מפת החום ובידוד המפה לרכיבי קשירות, ונדאג שהוא יעבוד על האסטרטגיה שבחרנו בשלב הקודם.
3. לאחר מכן, נפעיל שוב את אלגוריתם מציאת היישוב הראשון על המפה שקיבלנו, נבחר ביישוב האופטימלי.
4. נפעיל את אלגוריתם מציאת הדרך על היישוב שמצאנו בשלב הקודם.

עד כה הפתרון נראה פשוט ממבט ראשוני. כל מה שאנחנו עושים זה מפעילים את מה שעשינו בשלב הקודם שוב פעם. אז אם כן היכן באמת השינוי? ובכן, השינוי המרכזי קיים בשלב 2. פה אנחנו מחליטים באופן נחרץ להישאר עם האסטרטגיה שבחרנו בשלב הקודם, גם אחרי שמפת המשחק עברה אינספור טרנספורמציות שונות שיכולות לעלות לנו בשאלה "האם עדיין כדאי לנו לסמוך על האסטרטגיה הזו?"

למה בחרנו להישאר עם האסטרטגיה אותה בחרנו בהתחלה?

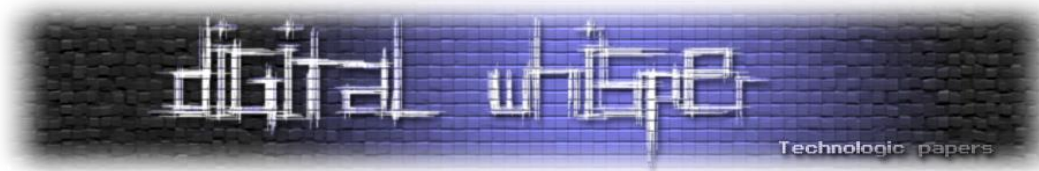
אפשר לענות על השאלה הזו בכמה דרכים שונות, ובוודאי ברור לכם כבר בשלב הזה של המאמר, שינוי של האסטרטגיה בשלב השני באופן כזה נחרץ זה למעשה לתקוע אצבע בעין של כל הרעיון היוריסטי הזה שלשמו אנחנו כאן. אבל בכל זאת, כדי לחזק את ההיגיון היוריסטי שבחרנו בו מלכתחילה, אבחר להמחיש את תשובתי בתמונה יפה שמסכמת את הכל:



בתמונה הזו נוכל לראות מפת חום עבור 3 אסטרטגיות שונות שנבחרו לאותה מפה. נשים לב שלכל אחד מהם יש מכנה משותף, והוא העובדה שרכיבי הקשירות במפת החום מתחלקים **או למקבצים שונים, או למקבצים גדולים בפני עצמם.**

העיקרון הזה ממחיש שבשני המקרים, בין אם בחרנו במקבץ הגדול בהתחלה, יש לנו יותר אופציות להתרחב באותו המקבץ בזמן בחירת היישוב השני. אך לעומת זאת, גם אם כל המקבץ שלנו נחסם על ידי היריבים, אנחנו עדיין נוכל להציב את היישוב השני על גבי המקבץ הנוסף (או לפחות על מה שנשאר ממנו). וגם אם לא נשאר ממנו הרבה, נוכל להציב את היישוב השני באחד מהאזורים שנותרו בו.

בסופו של דבר, המשחק עבר אבולוציה של סיבוב אחד שכלל במקרה הכי גרוע שתי הצבות של יישובים ושתי הצבות דרכים לכל אחד מהשחקנים. תמיד יהיה מקום מסוים במפה שיהיה אופטימלי עבור האסטרטגיה הראשונה שבחרנו ללכת איתה. ולמה בחרנו דווקא בה? ובכן, בשביל זה תצטרכו לחזור אחורה לתור הראשון ולהתבונן שוב במפת המשחק בזמן ההצבה הראשונית, אז המפה הייתי הכי נקייה עבורנו, התבוננו אז בראייה היוריסטית מתוך הבנה שזו הזדמנות נדירה לנצל את העובדה שמפת המשחק נקייה דיה והאפשרות נתונה בידינו, אז מן הסתם שלא ננטוש אותה בשלב הזה! ☺



ישוב ראשון - דרך ראשונה - ישוב שני - דרך שנייה

בפרק הזה נשלב את כל מה שעשינו בפרקים הקודמים לכדי מנוע משחק אחד שידע לקבל מפה במצב נתון כלשהו, להפעיל עליה את אלגוריתם היישוב הראשון + הדרך הראשונה. וכך לחזור על אותה פעולה שוב ושוב בכל תור במהלך המשחק.

למעשה, בכל שלב שבו השחקן ירצה להציב יישוב או דרך על המפה, הוא יאלץ להשתמש באלגוריתם שלנו למציאת הדרכים והיישובים (רק שהפעם זה לא יהיה רק בתור הראשון, אלא גם בתורות הבאים). אז איך בדיוק זה עומד לקרות?

כיצד נתמודד עם יריבים:

בין התור הראשון והשני (כפי שציינו קודם לכן), חלק מהמפה תפוסה על ידי יריבים, ואלה תופסים גם מקומות פוטנציאליים שאיננו יכולים להציב בהם יישוב עתידי פוטנציאלי. הפתרון הוא פשוט שכן כל מה שנצטרך לעשות זה להגיד לאלגוריתם מציאת היישוב שלנו שעליו להתעלם לחלוטין מכל היישובים על המפה שנמצאים במרחק של דרך אחת מיישוב של יריב אחר (שהרי לפי החוקים, כל יישוב מופרד בין שני דרכים). את המידע הזה נשמיט מהאלגוריתם שלנו רק בזמן החיפוש אחר היישובים.

כיצד נבחר שוב יישובים:

שינוי נוסף יהיה כמובן לגרום לאלגוריתם לבחור דרכים/יישובים רק בצמוד לדרכים/יישובים שכבר נבנו על ידינו בתורות הקודמים, והם במרחק של לפחות 2 מיישובים קודמים אחרים שבנינו או שאחד היריבים בנה (לפי חוקי המשחק). הדרך שבה נעשה זאת תהיה להתייחס ל-OptLocation שוב, רק שהפעם, נבחר את היישובים עם הדירוג הכי גבוה שנמצאים דווקא במקומות החוקיים, ואלו כמובן תואמים לאותה האסטרטגיה שלנו שנבחרה מתחילת המשחק.

כיצד נבחר שוב דרכים:

שינוי נוסף במציאת הדרכים יהיה להתאים את חיפוש מסלולי הדייקסטרה שלנו באלגוריתם למציאת הדרך, כך שבמקום שהחיפוש יתחיל רק מיישוב אחד שבנינו (כפי שעבדנו עד כה, בעמודים הקודמים), כעת הם יתחילו בו-זמנית מכל שאר היישובים שהצבנו במפה. כך נקבל כמה מסלולים אופטימליים שונים מכל יישוב שלנו. ומתוכם, נדע לברור איזה מהם הוא הכי אופטימלי בהתחשב בדירוג שלו.

השיטה שבה אנו בוררים בין כמה מסלולים מנקודות שונות על המפה נקראת Multi-source Dijkstra והיא נפוצה מאוד. ניתן לקרוא עליה [בהרחבה פה](#).



בניית מנוע המשחק:

```
--- Debug Menu ---  
Select options to add to the queue:  
1: Win Conditions upper dice & lower dice table  
2: Win Conditions table per strategies  
3: Showing heatmaps cluster propriety based on strategy, per each tile  
4: Debugging used for Python visualization  
5: Win Conditions (WIP...)  
6: Connection between each town to its adjusted hex  
7: Final Optimal Location Values for Each Town  
8: Connection between indices in ConnectedComponentTile class and Tile Class  
9: Final Optimal Town Location  
Commands: 'r' to reset | [Empty Enter] to Run | 'q' to quit | '0' to select all  
Current Selection [1] > |
```

גילוי נאות, עד השלב הזה בזמן כתיבת המאמר אני עבדתי במערכת די מסורבלת ב-CLI שנראתה כך: פה הבנתי שהגיע הזמן בפעם השנייה בפרויקט זה להפעיל את Claude ולבקש ממנו לעשות את מה שהוא יודע לעשות הכי טוב, לקחת את כל מה שעשינו עד כה ולחבר אותו לכדי "מערכת אינטראקטיבית" (מונחים שה-AI אוהב) שיודעת לקבל:

1. מפה נתונה

2. טבלת אסטרטגיות מוכנה מראש

והיא מחזירה:

1. מקום להצבת יישוב אופטימלי

2. מקום להצבת דרך אופטימלית

כך באופן הזה האלגוריתם שלנו ידע להתמודד עם מצב המשחק לא רק בתור הראשון והשני, **אלא בכל התורות של המשחק ככלל.**

פה חשוב לי לציין שבנושא זה נתמקד בעיקר בפרק הבא ובו נבחן את היעילות היוריסטית של האלגוריתם כולו בדרך לניצחון, בהשוואה לאלגוריתמים לא היוריסטים אחרים. זהו! עד כאן הסקופ, ועכשיו בואו נחזור לעניינים.

לשם המימוש, בחרתי בבניית תוכנה לוקאלית (מקומית) שלוקחת את כל מה שנעשה עד כה וממירה את זה לממשק נוח לשימוש וברור.

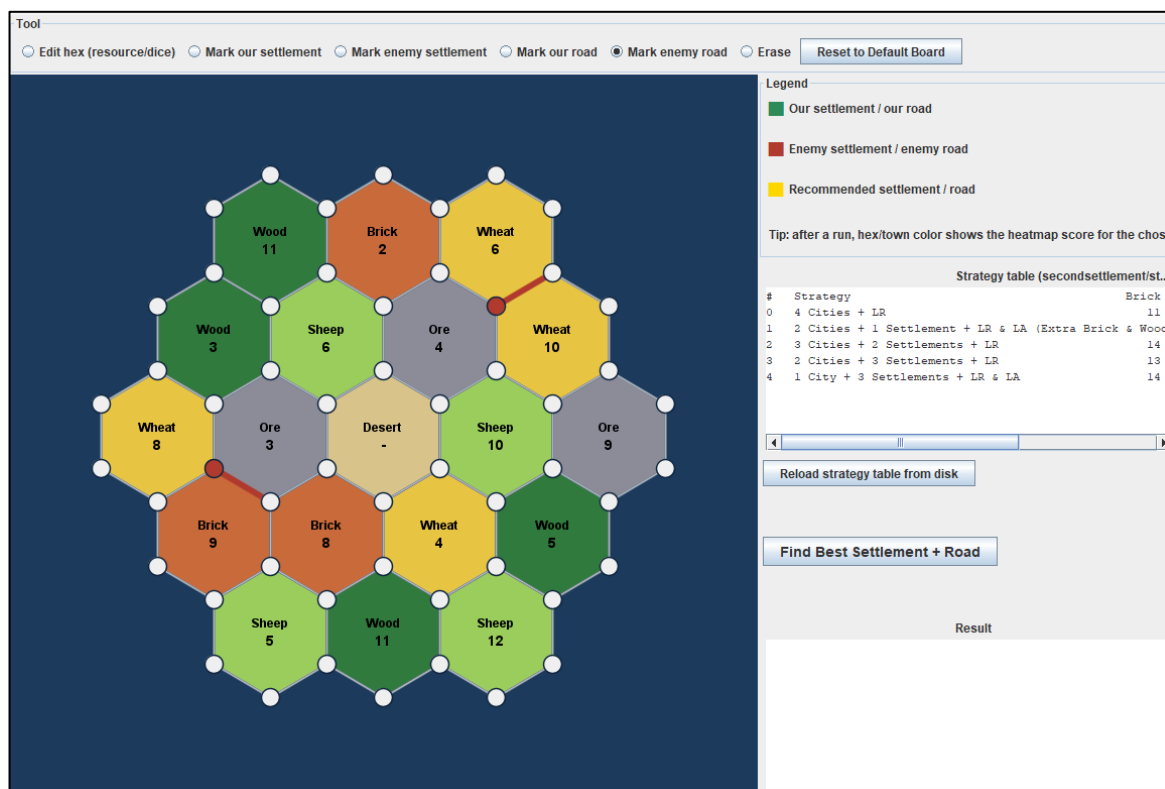
בדיקת הרצה ראשונית וניתוח סטטיסטי

בדיקת הרצה ראשונית:

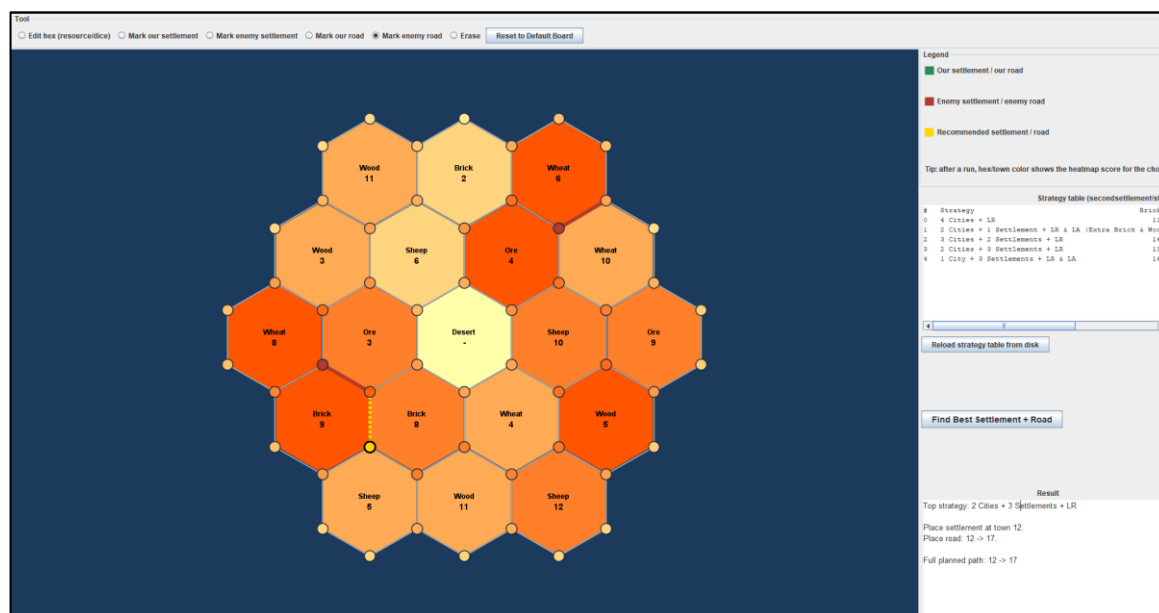
בשלב הזה הרצתי את התוכנה על פני מפה אקראית והנחתי שאנחנו נמצאים בתור השלישי בזמן הצבת היישוב והדרך הראשונה (ניתן להבחין ביישובים והדרכים של שאר היריבים על הלוח).

כדי לקבוע היכן למקם את היישובים והדרכים של היריבים, בחרתי בתבונה להריץ פעמיים את האלגוריתם שלנו על מנת למצוא את המיקומים האופטימליים ביותר לדרכים וליישובים של היריבים, בזה אחר זה. ואז לבסוף, פשוט הרצתי שוב כדי למצוא את המקום השלישי שבו אנו נציב את המיקומים שלנו.

כך זה נראה לפני ההרצה השלישית:

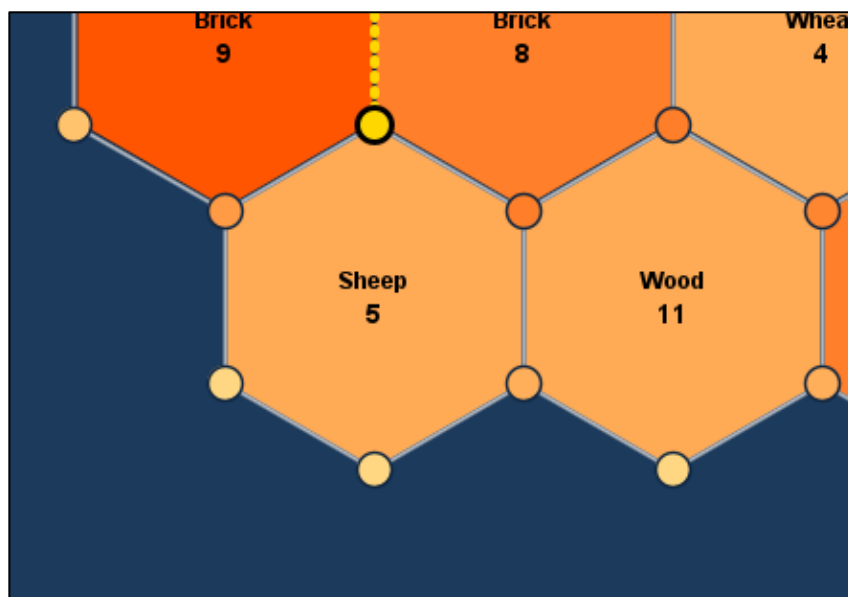


וכך זה נראה לאחריה:



חדי העין (או אלה שיעשו זום ב-PDF שאתם קוראים ממנו) ישימו לב שהאסטרטגיה שנבחרה לטובה ביותר כתובה בצד ימין למטה, והיא "2 ערים + 3 יישובים + הדרך הארוכה ביותר".

נקודה נוספת ומעניינת היא עצם העבודה שהאלגוריתם שלנו בחר להציב יישוב ודרך במקום מסוכן יחסית, צמוד לדרך של אחד היריבים. אך כנראה שבמקרה זה מדובר בעלות ששווה את התועלת, שהרי האופציות האחרות להתרחב אליהן למטה פחות טובות (תשימו לב לצבעים הבהירים של היישובים הפוטנציאליים):



ניתוח סטטיסטי:

החלק הזה הוא למעשה אחד הפרקים המשמעותיים ביותר בכל המסמך הזה. הרגע הגדול הגיע והוא למעשה השלב שבו ניקח את כל מה שבנינו עד כה ונבחן אותו מול אלגוריתמים אחרים כדי להבין עד כמה הפתרון היוריסטי באמת יעיל?

בעולם קיימים אלגוריתמים רבים לפתרון הבעיה שהצגתי ובאופן כללי לפתרון בעיית הניצחון במשחק קטאן. ישנם אלגוריתמים שמתעסקים בכל שלבי המשחק, החל מהנמלים, המסחר, אפילו עד לרמת האינטראקציה הפסיכולוגית של המשא ומתן בין שני יריבים. אך פה אני מוכרח לשים "סטופ!" ולהגדיר את הגבולות שאיתם אנו מתעסקים.

אבל לפני שנגדיר את הגבולות, יש משהו אחד קטן שחייכתי לעשות הרבה זמן...

טבלת אסטרטגיות קבועה:

לפני שנמשיך הלאה, עד כה הקלט לאלגוריתם שלנו היה רשימת אסטרטגיות שהמשתמש צריך להכניס. לשם מיצוי הבדיקה הסטטיסטית עד הסוף, נרצה לעבוד עם טבלת אסטרטגיות קבועה ומיטבית שמכילה את כל האסטרטגיות האפשריות במשחק. שזה למעשה כל הקומבינציות שניתן לחשוב עליהם לרכישת יישוב, עיר או דרך (ללא קלפי פיתוח) שבסופו של דבר יובילו לכך שנגיע ל-10 נקודות ניצחון.

הרשימה הזאת תישמר באופן קבוע במערכת והשינוי היחיד שיהיה ניתן לעשות בה זה לקבוע האם אנו רוצים להתחשב באסטרטגיה שמשלבת את "הדרך הכי ארוכה" או שלא. חשוב להזכיר כי עסקנו באפשרות הזו בפרקים הקודמים וכעת זה יהיה השלב שבו ניתן למשתמש את האפשרות לבחור זאת בעצמו באופן בוליאני (כפי שהוסבר בפרקים הקודמים).

לכל שחקן יש במחסן האישי שלו עד 5 יישובים, 4 ערים ו-15 דרכים להשתמש בהם באופן תיאורטי לכל אורך המשחק. לכן לפי שקלול כל הקומבינציות האפשריות, עולה כי ישנן רק 5 קומבינציות אפשריות על מנת להגיע ל-10 נק' ניצחון.

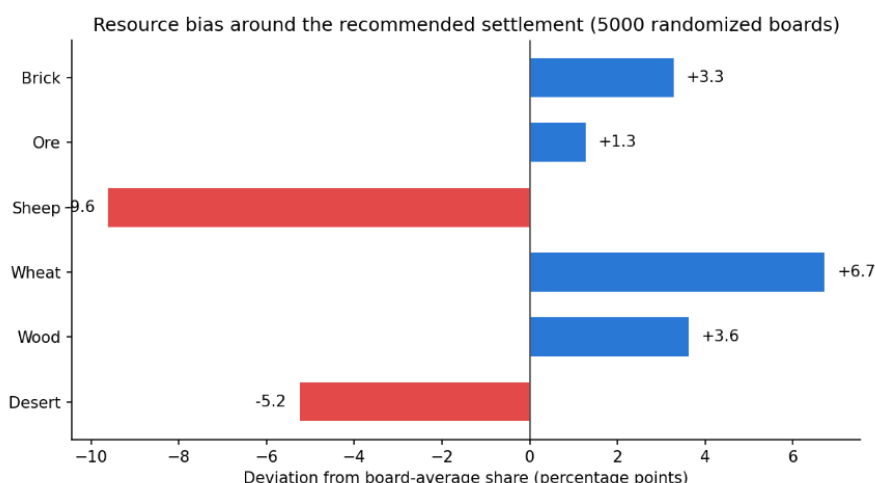
#	Condition	Brick	Ore	Sheep	Wheat	Wood
0	Cities + LR4	9	12	4	12	9
1	Cities + 2 Settlements + LR3	10	9	5	11	10
2	Cities + 4 Settlement + LR2	11	6	6	10	11
3	Cities + 2 Settlements (no LR)4	6	12	6	14	6
4	cities + 4 Settlements (no LR)3	7	9	7	13	7

האם יש נטייה למשאב ספציפי?

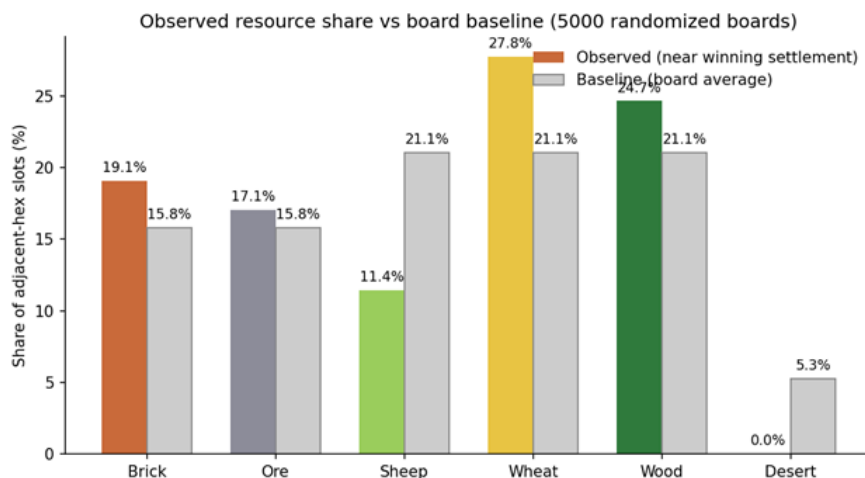
לפני שנמשיך עם השוואה מול מודלים אחרים, נרצה לבדוק האם טבלת האסטרטגיות הקבועה יוצרת מצב של הטיה ברורה בין משאבים ([להרחבה קראו על Selection bias](#)). ואם כן, האם היא אכן הגורם לכך? שהרי במידה וכן, נוכל להישאר מרוצים ובסך הכל נגדיר את מקור ההשוואה להיות לכזה שדומה לשלנו.

לאחר הרצה של 5,000 מפות אקראיות, נבדק איזה משאב נגע ביישוב שנבחר בכל פעם שבדקנו את מיקום היישוב האופטימלי, בהשוואה לשכיחות הטבעית שלו על הלוח. ונמצאה הטיה ברורה אך לא אקראית.

הטיה לעומת הבסיס:



אחוזי נוכחות בפועל מול הבסיס לכל משאב:



בפועל, נראה שישנה נטייה ברורה. למשל, החיטה גבוהה פי-2 (לערך) מהצמר. אחת מהמסקנות החשובות ביותר שמצאנו היא שההטיה הזו היא עקבית והיא נובעת מהעובדה הפשוטה שטבלת האסטרטגיות שהתבססנו עליה מצריכה 3 ערים ומעלה כמעט כחלק מכל אפשרות ניצחון, מה שגורר מצב שבו יש פחות תיעדוף לצמר וכו'...

הנטייה הזאת נוצרת מכיוון שבחרנו להגביל את השימוש בקלפי פיתוח, מסחר ובכל שאר האלמנטים האפשריים במשחק. ההגבלה הזו נובעת מעצם בחירתנו באלגוריתם היוריסטי. ולכן, כעת כשאנו מבינים שיש לנו הטיה ברורה במשאבים, חשוב להיצמד לכללים שקבענו לעצמנו כדי שנוכל לבחור מקור השוואה אמין שיבחן את אמינות האלגוריתם שלנו.

אז איך נבחר מקור השוואה אמין?

הגבולות שלנו בבחירת אלגוריתם מתחרה להשוואה הם הדברים הבאים:

1. נרצה התעלמות מוחלטת ממסחר בין שחקנים שונים
 2. נרצה התעלמות מוחלטת מהנמלים שנמצאים על המפה
 3. נרצה התעלמות מוחלטת משימוש בקלפי פיתוח
- הסעיף הראשון ברור, שהרי האלגוריתם שלנו יוצא מנקודת הנחה שכולם שונאים אותנו ולא רוצים לבצע איתנו מסחר. אבל לעומת זאת, אנחנו מוכרחים לצאת מנקודת הנחה שגם המסחר בין שאר השחקנים (ולא איתנו) לא יהיה חוקי, שהרי מסחר כזה יכול להסב להם יתרון משמעותי עלינו שלא נוכל להתגבר עליו באופן הוגן.
 - לגבי הסעיף השני, שימוש בנמלים נועד גם לשם מטרות מסחר אך בעיקר לצורך המרת קלפים בקלפים אחרים. למשל, 2 חיטה בעבור קלף 1 מכל סוג אחר. הבעיה המרכזית פה עומדת בכך שאם ננסה לחזות הסתברותית מתי תהיה לנו את האופציה להמיר 2 קלפים בעתיד, המשמעות היא

שנצטרך לשער מתי נקבל את אותם 2 קלפים, ומכאן טבלת ההסתברויות שלנו בהתחלה תהיה מושפעת מגורמים סטטיסטיים מורכבים מדי שיורידו את ההשפעה של המפה על בחירת האסטרטגיה שלנו (שזה עיקר המבחן במנגנון שיצרנו). מהסיבה הזו אנו לא משתמשים בנמלים - לפחות לא בגרסה הזו של האלגוריתם.

- לגבי הסעיף השלישי, הסברתי על כך בתחילת המאמר, אך בקצרה, גם פה הסיבה קשורה לסיבה השנייה בכך שהוספת תמיכה בקלפי פיתוח אומרת שברגע שנבחן את הסיכויים לקבלת קלף פיתוח מסוים, אנו ניאלץ להתבסס על מידע "פוטנציאלי מדי" שעלול להוריד מסך האמינות של החישובים שלנו.

השודד:

נקודה נוספת שלא עסקנו בה היא השודד.

במשחק קטאן סטנדרטי, בכל פעם שיוצא המס' 7 בקובייה, על השחקן בתורו לקחת את השודד ולהציב אותו על מספרי הקובייה באחד מהמשושים לבחירתו. הצבה שכזאת למעשה תחסום את האפשרות לקבל משאבים מאותו המשושה בכל פעם שיצא את אותו המס' בקובייה, כך עד הפעם הבאה שלמישהו יצא 7 בקובייה. (או במקרים חריגים בהם נשלף קלף הפיתוח מסוג "האביר". אך מכיוון שאנו לא עוסקים בקלפי פיתוח, נוותר כרגע על האפשרות הזאת).



מאחר ואנו עוסקים בסימולציות משחק קטאן מלאה, האלגוריתם שלנו איננו יודע להתמודד עם האפשרות של מס' 7 בקובייה והוא איננו יודע כיצד להגיב לכך בהתאם. לכן, נדרש מאיתנו למצוא פתרון לבעיה הזו כדי לדאוג שהמגבלה הזו של האלגוריתם שלנו לא תפגע בהשוואה בין המודלים השונים.

אז מה הפתרון? הפתרון הוא פשוט מאוד - לא לעשות כלום. העקרון המרכזי שלנו בהשוואה הוא השוני שעומד בין אופן בחירת היישובים והדרכים שלנו ולבין האופן שבו האלגוריתם המתחרה מממש את זה. במידה וכל שאר היבטי המשחק יתנהלו באופן שוויוני בין כל שאר הבוטים שבמשחק, רק כך נוכל באמת לקבוע האם השוני באלגוריתם שלנו מניב יתרון, חיסרון או שהוא איננו משנה כלל. לכן הדבר היחיד שיבדיל ביננו לבין שאר השחקנים זה שאנחנו נשתמש באלגוריתם שלנו, ולעומת זאת כל שאר השחקנים לא.

אז אם כן, הבה ניגש לבחירת האלגוריתם המתחרה. ופה חשוב להדגיש שכעת אנו מחפשים אלגוריתם מכל סוג שהוא, בין אם מדובר ב-Brute Force (לנסות את כל האפשרויות בבת אחת ולראות מה עובד), לבין מודלי AI מתוחכמים. הכל קביל כל עוד זה עומד בדרישות לעיל, כאשר המטרה המרכזית היא לעשות בדיוק את מה שהאלגוריתם שלנו עושה. וזה מציאת יישוב, דרך ולראות מי מצליח להגיע ראשון לניצחון.

בחירת מודל להשוואה:

אז אם כן הבה נציג את המתחרים שלנו:

1. אלגוריתם ה-Pip Count (ראו כאן):

פה מדובר בנוסחה הבסיסית והמוכרת ביותר לתכנון מיקום ויישובים בקטאן. הרעיון שעומד בבסיסה הוא שלכל אחד מתוך 11 האסימונים המספריים של הקוביות במשחק (2 עד 12, בלי 7) יש הסתברות שונה להיכלל מתוך 36 צירופי קוביות אפשריים. מעצבי המשחק הדפיסו את ההסתברות הזו ישירות על הספרות שמוצבות על המשושים. ואז, כדי לדרג מיקום פוטנציאלי ליישוב פשוט סוכמים את מספר הנקודות של המשושים שמקיפים את היישוב שלנו. השיטה הזו פשוטה ולא מתוחכמת והיא נגזרת מחישוב הסתברות בסיסי. אך היא הבסיס שעליו כמעט כל שיטות ה-AI המתוחכמות של קטאן מתבססות עליו. השיטה קלה ליישום והיא לא מפרה אף אחת מהמגבלות שהצבנו (קלפי פיתוח, מסחר, נמלים).

2. אלגוריתם ה-JSettlers (ראו כאן):

במקורו ממחקר אקדמי מתחילת שנות ה-2000, הוא התחיל כעבודת הדוקטורט של רוברט ס. תומאס ומאז הוא הפך לפרויקט קוד-פתוח קהילתי תחת רישיון (GPL). האלגוריתם נפוץ מאוד במאמרי מחקר על Baseline. ה-AI של "JSettlers" מובנה כמוח עם שתי אסטרטגיות מובנות:

- הראשונה Fast (מהירה, היוריסטית ופשוטה).
- והשנייה Smart (איטית יותר אך מנתחת יותר לעומק).

שתיהן פועלות תחת מחלקה שמעריכה בהתבסס על הסתברויות הקוביות וכמות המשאבים שהשחקן כבר צפוי לקבל מהמיקומים שלו - כמה תורות יידרשו לו כדי לבנות כל חלק אפשרי בתוכנית שלו (דרך, יישוב, עיר, קלף פיתוח). לאחר מכן, בהתאם לחישובים אלה היא בוחרת את הפעולה עם זמן ההגעה הקצר ביותר לניצחון, תוך כדי שהיא עוקבת אחרי זמן ההגעה הכולל לניצחון של כל השחקנים במשחק, כדי לזהות מי מוביל כרגע במרוץ וכיצד להגיב לכך בהתאם. הגישה הזו שונה במהותה מהאלגוריתם שלנו (שמבוסס על ניקוד סיכון סטטי ועל דייקסטרה, ולכן זה הופך אותו למתחרה אמיתי ובלתי תלוי שגם שומר על החוקים (כפי שנראה בהמשך) ולא לווריאציה על אותו הרעיון.

(יש לציין כי שיטה זו שמתבססת על מעקב אחר הקרבה לניצחון של היריבים והניסיון שלנו להשיג ולהכשיל אותם, היא נפוצה מאוד בקרב שחקנים מקצועיים בקטאן).

הערה: במקרה הזה, בחרנו להשתמש דווקא באלגוריתם ה-Jsettlers משום שהוא מתוחכם יותר, מאתגר וכי משתמשים בו רבות במאמרי מחקר.

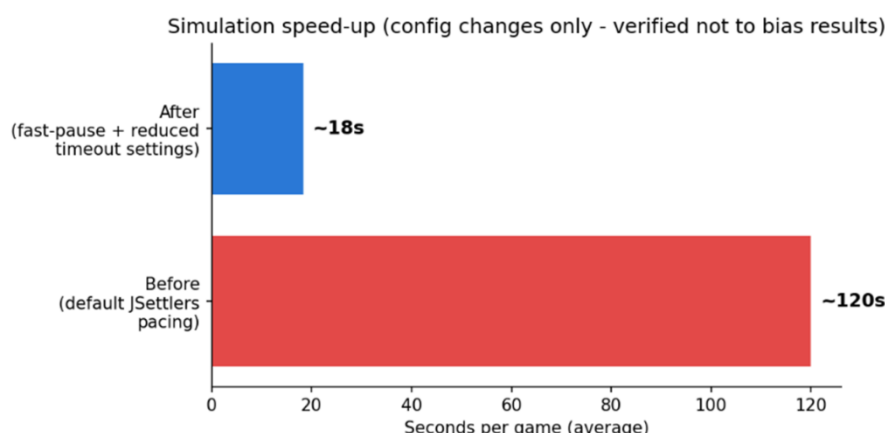
בניית הסימולציה:

בשלב הזה ביקשתי מ-Claude לממש סימולציית בוטים שבה יש 3 שחקנים בוטים ועוד שחקן אחד נוסף שזה אנחנו (בסה"כ 4 שחקנים במשחק). כל הבוטים עובדים לפי האלגוריתם של Jsettlers, בעוד שאנחנו (הבוט ה-4 הנותר) עובדים לפי האלגוריתם שלנו בלבד. כמובן שהתכנון לא היה פשוט שכן נדרשתי לדאוג לכך ש-Jsettlers יבטל את האופציה המובנית להתייחס למסחר, קלפי פיתוח וכל שאר הגורמים שאינם חוקיים בהשוואה שלנו. אך בסופו של דבר, לאחר שהגעתי לתוצאות מספקות והרצתי את הסימולציה הראשונה, זאת לקחה המון זמן (כ-17 דק'), הסיבה לכך כרוכה בבאג של טיימר בין כל תור שלא היה לנו נחוץ בו).

בסופו של דבר בניתי מערכת שמריצה 2,000 משחקים על 12 ליבות במקביל (40דק' עבודה בפועל), כאשר כל סימולציית משחק מציגה מפה אקראית לחלוטין. בחצי מהסימולציות שעריתי משושה המדבר מופיע באמצע, ובחצי השני הוא מופיע בכל מקום אקראי אחר.

(הערה: ישנם שחקנים שבחרים לשחק רק באחד מן שני האופנים הללו, לכן היה לי חשוב לבדוק גם את זה)

פה ניתן לראות את השוואה לפני האופטימיזציה של אורך סימולציית המשחק, ולאחריה:

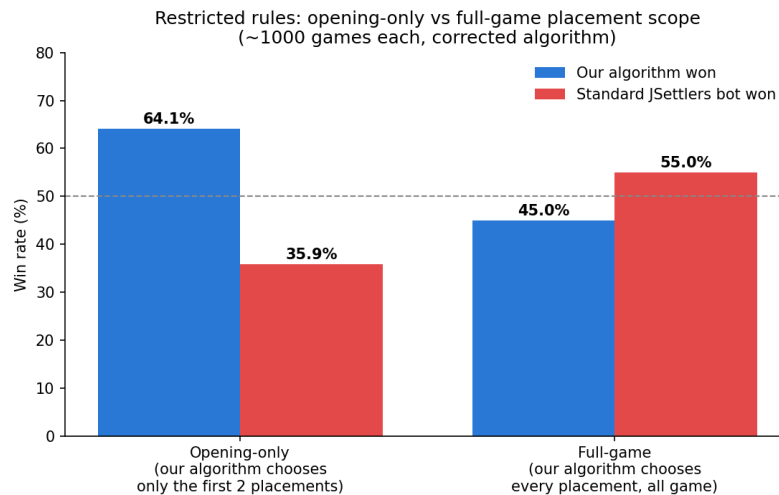


התוצאות:

כפי שהוסבר קודם לכן, אנחנו בודקים את סיכויי של האלגוריתם שלנו להביא לניצחון תחת מגבלות על חוקי המשחק. האלגוריתם שלנו עוזר למצוא את היישוב והדרך האופטימליים. הפעולה הזאת נעשית בתחילתו של כל משחק, אך גם במהלך כל תור, כאשר השחקן מעוניין לבנות יישוב או דרך הוא יכול גם כן להיעזר באלגוריתם שלנו. אם כן, נשאלת השאלה מי מחליט **מתי** תיבנה הדרך או היישוב? (ולא היכן) ובכן, התשובה לכך היא האלגוריתם של Jsettlers. כלומר, האלגוריתם המתחרה משאיר את כל ההחלטות במהלך המשחק זהות בין כל הבוטים באופן שווה. אך בזמן שהאלגוריתם של Jsettlers מחליט בתורו שזהו הזמן המתאים לבנות דרך או יישוב, רק אז האלגוריתם שלנו ייכנס לפעולה ויפעל - כאשר בשלב הזה הוא ינסה לאתר את המיקום האופטימלי עבור אותו יישוב/דרך. **אותו שוני, הוא זה שיוצר את נקודת ההשוואה שממנה ניתן לבדוק איזה אלגוריתם יעיל יותר.**

עם זאת ערכתי בדיקה נוספת, ובדקתי מה יקרה כאשר בוחרים להשתמש באלגוריתם שלנו דווקא בתחילת המשחק בלבד (כשצריכים לבחור יישובים ודרכים) וכלל לא במהלכו. והנה התוצאות.

מה הסיכוי לנצח בעת הצבה בתחילת משחק לעומת במהלכו, כאשר חוקי המשחק מוגבלים:



בכחול - האלגוריתם שלנו

באדום - האלגוריתם המתחרה Jsettlers

ראשית כל, חשוב להגיד שמתוך 4 שחקנים בסה"כ, הסיכוי הבסיסי שלנו (כלומר "נקודת המוצא" - להרחבה קראו על [התפלגות אחידה בדידה](#)) לנצח עומד על 25%. וכעת, הממצאים הם די מדהימים. כאשר משתמשים באלגוריתם שלנו רק בתחילתו של המשחק בלבד, הוא מצליח לנצח את האלגוריתם המתחרה Jsettlers ב-64.1% מהמקרים.

לעומת זאת, כאשר משתמשים באלגוריתם שלנו בעת כל הצבה שמתרחשת במהלך המשחק, היעילות שלנו צונחת ל-45% (אם כי היא עדיין נמצאת ביתרון של פי 1.8 מנקודת המוצא ההוגנת) ומכאן נשאלת השאלה, מדוע?

אם כל מה שששתנה בין שתי הבדיקות זה "היכן" שאנו מציבים את היישובים והדרכים, אז אנחנו עדיין מצליחים לשמור על יתרון יחסי של פי 1.8 מנקודת המוצא (כפי שראינו), אבל זה עדיין לא עונה לנו על השאלה מדוע.

הסבר פוטנציאלי: ובכן, כפי שהוסבר כאן קודם לכן, "המרחב הבטוח" הוא המרחב שבו האלגוריתם שלנו



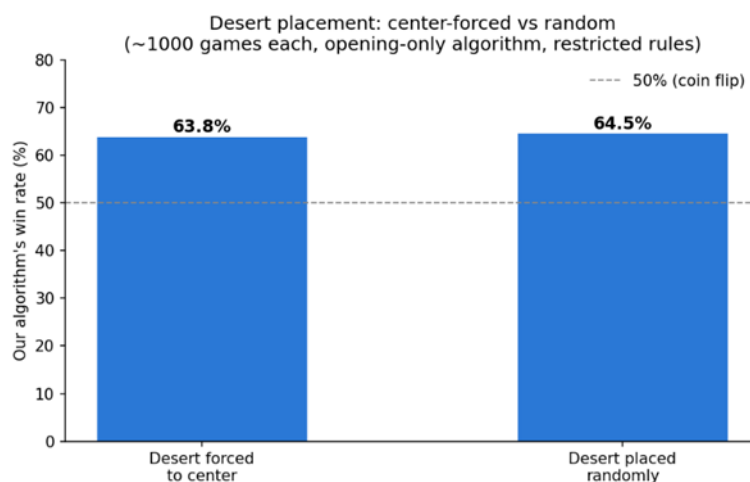
יודע לבחון את המפה ועל בסיסו להבין באיזה מקום מתאים להציב דרך ליישוב. ככל שהמשחק ממשיך (ובואו לא נשכח שמדובר ב-4 שחקנים שמשחקים במקביל), נבנים עוד ועוד דרכים ויישובים. כך המרחב הבטוח מצטמצם, היריבים בוחרים

במקומות הטובים (באופן יחסי, גם אם הם לא הכי טובים) וכך טווח האפשרויות שלנו "למרחב הבטוח" קטן. למעשה, בגלל שאנו נדבקים לאותה האסטרטגיה מתחילת המשחק, נוצר מצב שהאלגוריתם שלנו

לא נותן ביטוי למצב היד בפועל של היריבים או שלנו בכלל. ובגלל שאנחנו לא נותנים לכך ביטוי, הבחירה של היישובים והדרכים שלנו אינה מושפעת כלל מהגורמים הרלוונטיים האלה, לכן זה מיד עלול להציב אותנו בחיסרון מסוים.

האם מיקום משושה המדבר משנה את הסיכוי שלנו לנצח?

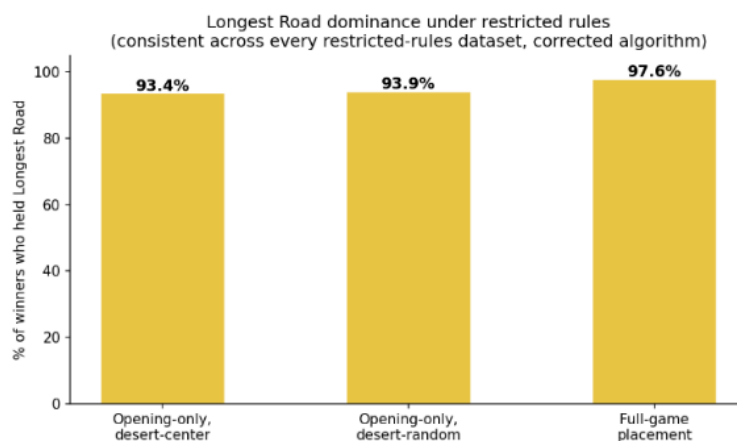
בגלל שהאלגוריתם שלנו מבסס את החלטותיו על מצב המפה בפועל, הבנתי שיהיה נכון למדוד גם מה קורה אם מציבים את משושה המדבר רק באמצע של המפה (כמו שרבים נוהגים לעשות), לעומת בכל מקום אחר. ולהלן התוצאות:



כפי שניתן לראות, אין הבדל ממשי.

הדרך הארוכה ביותר:

ממצא נוסף ומעניין היה שברוב המובהק של הפעמים שהאלגוריתם שלנו ניצח, הוא השתמש באסטרטגיה שמשלבת את "הדרך הארוכה ביותר". ניתן לשער כי הסיבה לכך היא עצם העובדה שאנחנו יודעים לשלב בצורה היוריסטית אסטרטגיה שמתחשבת בבניית דרכים עוד מההצבה הראשונית.



מה קורה ברגע שפותחים את כל חוקי המשחק? (מסחר, קלפי פיתוח, נמלים):

חשוב לציין שעד לשלב הזה האלגוריתם שלנו לא הוכן מראש. בכל המאמר לא התייחסנו למצב של נמל, קלפי פיתוח ולבטח שלא למסחר. בתחילתו של המאמר ציינתי שיש לכך כמה סיבות והראשונה שבהם שזה יפגע בעיקרון היוריסטיות שהצבנו לעצמנו בהתחלה. אך עם זאת יש לסייג, במשחק אמיתי אלו הם חוקי המשחק הרגילים! בהחלט ייתכן מצב שבו אף אחד לא ירצה לסחור איתך, אבל נמלים וקלפי פיתוח הם בהחלט גורמים שנכנסים לשיקולי ההחלטות של השחקנים סביב המפה. כשניסיתי לחשוב ולתערך את הבעיה הזאת, ישנן כמה דרכים בהן ניתן לפתור אותה:



1. קלפי הפיתוח - מכיוון שחפיסת קלפי הפיתוח הינה קבועה בכל משחק קטאן, ניתן להכניס אותם לחישוב ההסתברותי (הוסבר

על זה בתחילת המאמר), ולחשב טבלת אסטרטגיות מוכנה מחדש על זה בתחילת המאמר). ולחשב טבלת אסטרטגיות מוכנה מחדש החשיבות שלהם במשחק. הבעיה המרכזית היא שאחוזי ההסתברות יורדים ככל שמנסים לנחש "האם נקבל את המשאבים לקניית קלף הפיתוח?" וביחד עם זאת "האם בכלל נקבל קלף פיתוח מהסוג שאנו רוצים?".

הבא נסביר: למעשה, ישנו חישוב הסתברותי כפול שמתבסס קודם כל על השאלה "מה הסיכוי שנקבל את המשאבים שאנו צריכים לקניית קלף הפיתוח?" ואז, על גבי אותו סיכוי "מה ההסתברות שנקבל בכלל את מה שאנחנו מחפשים בתוך חפיסת הקלפים?".

אך בכל זאת, בגלל שאנו רוצים לבדוק את האלגוריתם שלנו בראייה היוריסטית, נצטרך להשתמש בחישוב בכפול הזה רק לצורך הבדיקה, **זה המעט שנוכל לעשות כדי לבחון את העניין.**

להלן האסטרטגיות החדשות שנוספו לטבלת האסטרטגיות לאחר החישוב ההסתברותי הכפול:

עץ	חיטה	כבשה	ברזל	בנייה (חמר)	אסטרטגיה
4	17.36	9.36	17.36	4	4 ערים + הצבא הגדול ביותר
5	16.36	10.36	14.36	5	3 ערים + 2 יישובים + הצבא הגדול ביותר
6	15.36	11.36	11.36	6	2 ערים + 4 יישובים + הצבא הגדול ביותר
5	18	10	17	5	4 ערים + יישוב אחד + קלף נקודת ניצחון
6	17	11	14	6	3 ערים + 3 יישובים + קלף נקודת ניצחון

הבהרה: השינויים המרכזיים הם במשאבים ברזל, חיטה וכבשה משום שאלו הם המשאבים הנדרשים לקניית קלף פיתוח.

2. סחר - במקרה הזה, לא נתנו ביטוי היוריסטי לסחר אבל אנחנו גם לא צריכים. חשוב להדגיש שבבדיקה שלנו אנו לא מנסים לבדוק האם יהיה לנו יתרון יחסי בסחר או שלא, הבוט שישחק את האלגוריתם שלנו מקבל את אותה יכולת הסחר של כל שאר הבוטים בסימולציה (כפי שהסברנו קודם לכן) והיא היכולת של Jsettlers. **לכן אין פה באמת שוני** וזה לא מייחד או מבדיל אותנו מהשאר. לכן נתעלם מכך לצורך ההשוואה.

3. **נמלים** - במקרה הזה, הנמלים משפיעים באופן נחרץ על המיקום שבו נרצה להציב את היישוב. לתת ביטוי לנמל על גבי המפה זה למעשה יצירת קשר בין מיקום פיזי לבין חישוב ציוני ההסתברויות לאסטרטגיה. זה משהו שעשינו קודם לכן במאמר והיה ניתן להרחיב אותו.



הסיבה שלא התמקדתי בזה היא מטעמי פשטות.

לכן בשלב הזה ניתן להגיד שהאלגוריתם שלנו לוקה בחסר בכך שהוא

אפילו לא מייחס משמעות לאפשרות הזאת והוא לחלוטין לא מודע

לכך שהיא קיימת. זה מציב בפנינו בעיה אמיתית,

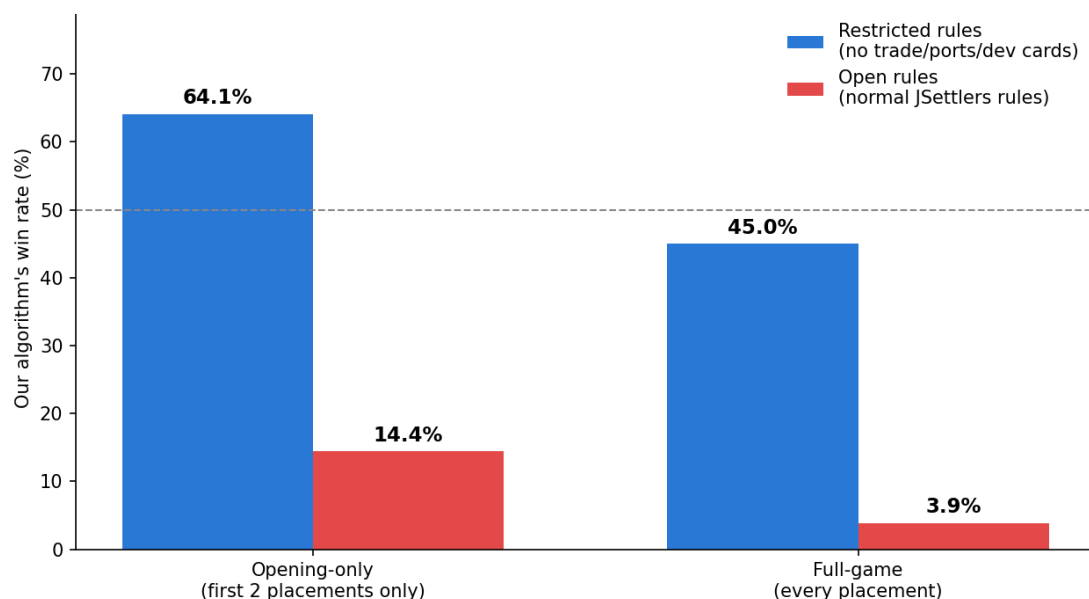
מפני ששחקנים אחרים יכולים לגבור עלינו במהירות הגישה למשאבים

מסוימים בשלבים המאוחרים של המשחק וזאת בזכות זה שהם מייחסים את חשיבות המיקום גם לנמלים.

אלו שנמצאים ליד נמל יהיו ביתרון יחסי עלינו, מה שיקשה על האלגוריתם שלנו לגבור עליהם.

בסופו של דבר, לאחר הרצת 4,000 סימולציות משחק שונות, בדקתי בדומה לקודם מה היה קורה אילו היינו מציבים רק בתחילת המשחק את היישובים והדרכים ע"פ האלגוריתם שלנו, או לחילופין, בכל תור במשחק, רק שהפעם אנו מבצעים את הבדיקה הזו בעוד שכל החוקים בסימולציות פתוחים.

השוואה בין משחק עם חוקים פתוחים לחוקים מוגבלים (והצבה ראשונית לבין כל המשחק):



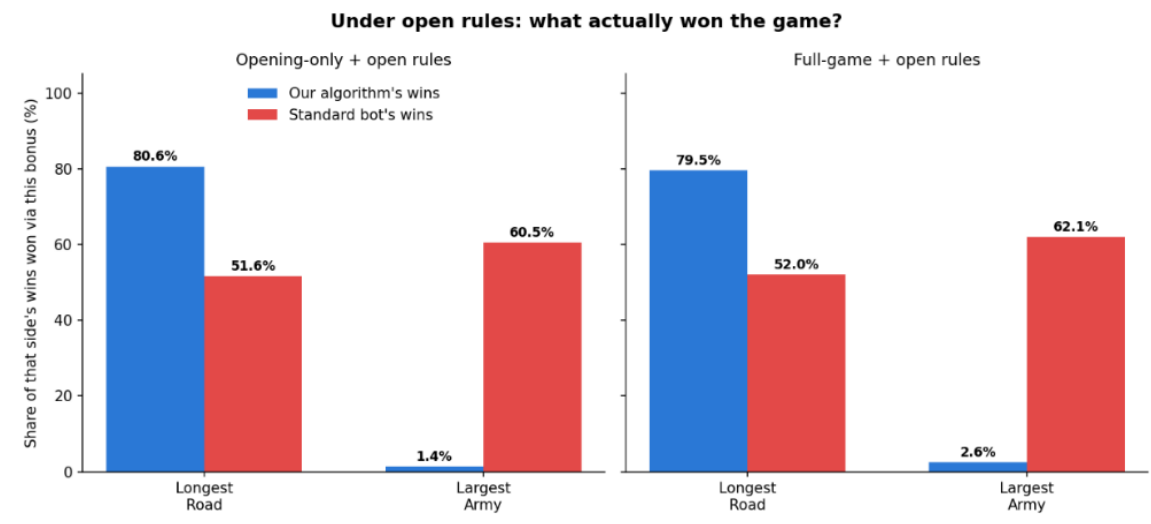
בכחול - סימולציה עם חוקים מוגבלים (כפי שבדקנו קודם)

באדום - סימולציה עם חוקים פתוחים

הממצאים בשלב הזה הם די מדהימים (אך גם לא הכי מפתיעים). קודם לכן ראינו שהיתרון שלנו היה 64% במשחק עם חוקים מוגבלים כאשר אנו מציבים רק בהתחלה, אבל כעת, הוא צנח ל-14.4%! בנוסף, כאשר אנו משתמשים באלגוריתם שלנו לכל אורך המשחק, היתרון צנח מ 45% קודם לכן ל-3.9%, שזה כבר חיסרון משמעותי.

למעשה, אנו למדים מכך שברגע שאנו מחזירים את כל חוקי המשחק, אחוז הניצחון שלנו קורס משמעותית. ומכאן נשאלת השאלה, **למה זה בעצם קורה?**

על מנת לנסות ולענות על שאלה זו, פירקנו כל סימולציית משחק לפי האסטרטגיות שהכריעו אותנו:



כפי שניתן לראות, כאשר אנו מפצלים כל תרחיש לפי האפשרויות "הדרך הכי ארוכה" ו-"הצבא הגדול ביותר" (שזכור כל אחד מאלה מוסיפים +2 נק' ניצחון), אנו מתבשרים לגלות כי בכל אחד מהמקרים, מתוך הניצחונות של האלגוריתם שלנו, **"הדרך הארוכה ביותר"** לוקחת באופן מובהק על פני האלגוריתם המתחרה. אך לעומת זאת, כאשר מדובר בניצחון שבו מעורב קלף הניצחון **"הצבא הגדול ביותר"**, האלגוריתם המתחרה מצליח לגבור על שלנו באופן מובהק.

מחד, ניתן אולי לחשוב כי הסיבה לכך נובעת מכך שהאלגוריתם שלנו מתמקד בניית תוח המפה בעיקר. אך במקרה הזה, כזכור, כל הבוטים שלנו במשחק זהים באופי ההחלטות שלהם. השוני היחיד הוא במיקום הצבת היישוב והדרך כאשר הבוט שלנו מחליט שזה הזמן המתאים לעשות זאת. אז אם כן, כיצד זה ייתכן אם הקוד הזה בכל המקרים?

ובכן למעשה הקוד שמנהל את קבלת ההחלטות הינו זהה, אך המשאבים שהוא מקבל כקלט אינם זהים. כל שלושת הבוטים האחרים מריצים את הקוד קנייה על גבי מיקום שנבחר ע"י ה-AI המובנה של Jsettlers שתוכנן מלכתחילה, עם ידיעה מלאה שקלפי פיתוח קיימים (בעוד שהקוד שלנו נועד לפתור רק מצבי מיקום ודרך על גבי המפה). בנוסף, נבדק גם הסבר חלופי אפשרי שבו אולי הטיית המושבים (סדר התורות) שהבוט שלנו בוחר אולי הוא זה שגורם לחיסרון המובהק הזה, אך גם לאחר בדיקה הוכח כי מקום מושבו של הבוט שלנו מתפלג באופן 24%/25%/24%/27% על פני 4 המושבים/התורות השונים בסימולציות. כלומר זה שולל את ההסבר הזה.

למעשה, אותו הפער מ-25% הוא בדיוק המדד הכמותי לעלות של הרצת אלגוריתם מיקום ש"אינו יודע" שהמשחק כולל יותר מהצבת יישובים.

לסיכום: במקרים בהם ההצבה הראשונית היא הגורם המכריע היחיד (חוקים מוגבלים), האלגוריתם שלנו בוחר יישוב פתיחה בצורה טובה יותר מהיוריסטיקה המובנית של Jsettlers, פי 2.5 בערך מקו הבסיס ההוגן. אך ברגע שאנו מרחיבים את סמכותו למשחק מלא (עדיין תחת חוקים מוגבלים), היתרון מצטמצם משמעותית אך עדיין נשאר יתרון אמיתי (פי 1.8 בערך). לעומת זאת, ברגע שמחזירים את חוקי המשחק המלאים (קלפי פיתוח, מסחר ונמלים), גם בפתיחה בלבד היתרון נעלם לחלוטין והופך לחיסרון. ניתוח מסלולי הניצחון בפועל מצביע על קלפי הפיתוח (ובפרט "הצבא הגדול ביותר") כמנגנון הדומיננטי מאחורי הקריסה, שכן זה בדיוק הערוץ שבו האלגוריתם המתחרה של "Jsettlers" עוקף אותנו.

הבהרה חשובה: עם זאת, מאחר וכל שלוש ההגבלות (קלפי פיתוח, מסחר וכו'..) הוסרו יחדיו בסימולציות, עדיין לא ניתן לשלול לחלוטין תרומה חלקית שמהווה יתרון לאלגוריתם שלנו, של מסחר או של נמלים בלי שנקיים ניסוי שמבודד לנו כל גורם בנפרד. (זאת משום ש"קורלציה אינה מעידה על נסיבתיות" - [להרחבה קראו כאן](#)).

הרחבות

את הבדיקות שנעשו קודם לכן בשלב הסטטיסטיקה ניתן היה להרחיב לאלגוריתמי AI משופרים ואף עדכניים יותר. הסיבה שלא בחרתי לעשות זאת היא משום שזה היה דורש עבודה תכנותית רבה במיוחד, ובנוסף, רציתי דווקא להתמקד בהיבט הפשוט והיוריסטי, ולבחון אותו אל מול השוואה של מודל שמתמקד באופן קבלת ההחלטות (משהו שמנוגד למודל שלנו שמתבסס על מיקומים בלבד). ההשוואה הזאת טובה כיוון שהיא מראה לנו שישנה עוד פרספקטיבה לדברים מסוימים, אך היא איננה שוללת בהכרח את ההצלחה של האלגוריתם שלנו, אלא היא רק מוסיפה לעובדה שהוא מאוד טוב בלפתור בעיה ספציפית. הבעיה שאליה הוא נועד, או שהוא תוכנן לפתור מלכתחילה.

בכל מקרה, ישנן גם בעיות ספציפיות שכאלה במגוון תחומים מחיי היום-יום. הנה רשימה של כמה הרחבות מעניינות שחשבתי עליהן. את אף אחת מהן לא יצרתי או מימשתי בעצמי, אך אלו רעיונות כלליים ובסיסיים ביותר שעוזרים לנו להבין כיצד האלגוריתם שבנתי נוסק לעולמות נוספים שאינם בהכרח רק המשחק "קטאן".

נתחיל מהרחבה ברורה ופשוטה - ראיית תמונה באפליקציה

אפליקציה שתאפשר למשתמשים לצלם את מפת המשחק שלהם בתחילת המשחק, כך שהאלגוריתם שלנו ידע באופן מיידי להמליץ לשחקן היכן למקם את היישובים והדרכים בתחילת המשחק. בתרחיש שבו השחקן ישתמש באלגוריתם שלנו לאורך משחק שלם (כלומר יצלם את המפה בכל פעם שמגיע התור שלו), הדבר המעניין לדעת הוא שזמן הריצה של האלגוריתם שלנו הוא בעצם אפסי, זאת בשל היותו היוריסטי וקבוע ([constant time](#)). אחד מהאתגרים יהיו לעדכן את אזורי האויבים בהתאם ולשלוח מפה מתוקנת בכל פעם לאלגוריתם כדי לעקוב ולהבין היכן המקום האופטימלי לשים בו דרך ויישוב.

דוגמאות מהחיים האמיתיים

האלגוריתם שלנו אמנם נוצר בשביל המשחק "קטאן", אבל השימושים הרעיוניים שלו בעולם האמיתי הם אדירים ונוגעים להמון תחומים שונים. כזכור, בתחילת המאמר למדנו להמיר את הבעיה שלנו לבעיית גרפים. מובן לגמרי שבדומה להמרה של "קטאן" לבעיית גרפים, כך נוכל גם להמיר את האלגוריתם שיצרנו עבור קטאן לכל בעיה אחרת שקיימת בעולם. הנה כמה דוגמאות:

1. מיקום תחנות חירום (אמבולנס/כיבוי אש) תחת ביקוש לא ודאי:

בתכנון תפעולי, קיים תחום ותיק בשם [\(MCLP\)](#) Maximal Covering Location Problem, יש לו שימושים רבים, וביניהם גם תכנון מיקום של תחנות אמבולנסים וכיבוי אש. למעשה, כל שכונה בעיר מסוימת מייצרת תמהיל שונה של סוגי אירועי חירום שונים (למשל אירועים קרדיולוגיים, נשימתיים, טראומה ועוד..). בקצב שאינו ידוע מראש ומשתנה מעיר לעיר (בדומה למשושה במפת קטאן עם משאבים משתנים). תחנה מסוימת מכסה קבוצה קבועה וקטנה של שכונות סמוכות בטווח זמן נסיעה נתון מראש



(בדיוק כמו יישוב שנוגע ב-2/3 משושים שנדרש אליו כמות מסוימת של דרכים כדי להגיע אליו).

במקרה שלנו, נוכל לסווג כל צוות או ציוד אפשרי לתחנה (למשל יחידת טראומה מול יחידה קרדיולוגית) כ-"אסטרטגיית ניצחון" הדורשת שילוב שונה של אותם משאבים. כך למשל אם ישנו אירוע נדיר בעיר כולה, כיסוי שכונה עם סיכון כלשהו לאותו סוג שווה יחסית יותר (בדיוק באותו עיקרון שגרם לאלגוריתם שלנו להעדיף משושים נדירים בשל הנחיצות שלהם לאסטרטגיה שלנו). ומיקום התחנה, כמו הצבת יישוב, הוא החלטת שמשתנה בפועל אחת לכמה שנים.

אך במקרה הזה, התרומה הפוטנציאלית של האלגוריתם שלנו היא דווקא הגישה שאיתה אנו מגיעים לתחום. השיטה הסטנדרטית ב-MCLP פותרת אופטימיזציה משוקללת אחת על פני כל סוגי הביקוש בבת אחת (יקר חישובית). לעומת זאת, האלגוריתם שלנו מציע גישה שונה, והיא לזהות קודם כל לפי מפת הביקוש הספציפית, איזו אסטרטגיה יחידה הכי ריאלית להשגה (על פי נדירות מקומית), ורק אז לפתור בעיה חד-יעדית פשוטה בהרבה - והיא קירוב מהיר שיכול לשמש כשלב ראשוני זול או כשאין זמן לפתור אופטימיזציה מלאה (למשל בפריסת חירום זמנית או באסון טבע).

2. פריסת תשתית סלולרית/רשת בשוק חדש:

גם בעולמות התקשורת האלגוריתם שלנו נכנס לעניינים חזק מאוד. למשל, חברות תקשורת הנכנסות לתחום חדש (בעיקר בכלכלות מתפתחות) מזמינות סקרי ביקוש לפני פריסת אנטנות ראשונות, זאת מכיוון שתמהיל המשתמשים בכל אזור גאוגרפי אינו ידוע מראש ומשתנה מאוד ממדינה למדינה. אך גם כאן: אזור = משושה, עם תמהיל משתמשים משלו. למעשה, אנטנה מוצעת מכסה קבוצה קטנה וקבועה של אזורים סמוכים, ותצורת רשת אפשרית (בנייה כפרית מול בנייה מהירה ועירונית) היא "אסטרטגיה" הדורשת תמהיל משתמשים שונה. מיקום האנטנה הראשונה במקרה הזה היא השקעת הון יקרה ובלתי הפיכה, והיא מתבצעת לפני שדפוס השימוש בפועל מתברר.

לכן גם במקרה הזה אסטרטגיה ידועה מראש יכולה לעזור לנו לפרוס רשת של מיקומים פוטנציאליים על גבי מפת ביקושים משתנה, או לחילופין, לפרוס דרכים שמחברות בין אנטנה לאנטנה (בדומה לדרכים



שמחברות בין ישוב לישוב).

חשוב מאוד לסייג ולהגיד שבשני המקרים, האלגוריתם שלנו רלוונטי אך ורק להחלטת המיקום הראשונית ולא לניהול השוטף (למשל איזה אמבולנס לשלוח לאיזו קריאה בזמן אמת או כיצד לאזן עומסי רשת מרגע לרגע), זו בדיוק הבעיה שמצאנו כשניסינו להרחיב את האלגוריתם שלנו למשחק קטאן מלא עם חוקים פתוחים, האלגוריתם שלנו אינו בנוי להסתגלות מתמשכת, אלא הוא מיועד אך ורק לבחירת המיקום הראשוני והחד-פעמי, או לחילופין, לבחירת מיקום מתמשכת בעוד שהחוקים הם מוגבלים ומיטיבים עם האלגוריתם שלנו.

וכמובן שלא חסרים רעיונות נוספים על מנת להקביל אלגוריתם כללי שכזה לעוד בעיות בחיי היום-יום.

סיכום

טוב אז איך נסכם... העבודה הזו היא פרי הדמיון שלי, סיפור שלם שהתחיל ממשחק קטן בסלון הבית של חבר טוב שלי, ומשם התפתח ללילות ארוכים של כתיבה מול המחשב ומחקר. המחקר הזה כולו לקח 7 חודשים בסה"כ (עם הפסקות גדולות באמצע שנוצרו עקב לימודים ועבודה), ועשרות שעות של מחקר וכתיבה. אחד העקרונות החשובים שבחרתי בהם בתחילת העבודה היא מימוש עצמי. בחצי שנה בלבד עולם הבינה המלאכותית השתכלל כמו שלא קרה בעשורים שלמים במהלך ההיסטוריה. היה לי חשוב להיצמד לאלגוריתם היוריסטי שלא מבוסס על AI או Brute Force ולהקליד את הקוד באופן עצמאי מכמה סיבות: הראשונה שבהן היא העובדה שאני מאוד מעריך התנסות עצמית והתחבטות מחשבתית. בפועל, הקוד יכל היה להיכתב ע"י AI בלי שום בעיה ובהמשך הפרויקט הבנתי שאני מוכרח לעשות זאת כדי שאוכל להתמקד במחקר ולסיים את הפרויקט (וגם כי הייתי מאוד עמוס בחיי האישיים). אבל את הבסיס עד לשלב מציאת הדרכים ובכלל, כל הפסאודו-קוד לאלגוריתמים מימשתי באופן עצמאי לחלוטין.

אחד מהלקחים המרכזיים שלמדתי מתוך הפרויקט זה העובדה שלא משנה כמה תבקשו מהצ'אט שיבצע את המטלות שלכם, בסופו של דבר מטלה מסובכת כמו זאת היא בסך הכל עניין של מחשבה ארכיטקטורית מורכבת שבבסיסה עומד המוח האנושי. כל בקשה, תכנון, חשיבה וחיבור של כל כך הרבה אפשרויות ביחד, הן אלו שגרמו להצלחה של זה בסופו של דבר.

אני לא יכול אפילו לתאר לכם כמה פעמים ה-AI הציע לי גיבוב שטויות של רעיונות לא הגיוניים ומקרי קצה שהוא לא קלט אותם, או לחילופין, כאלה שחרגו מהרעיון הכללי עליו מבוסס האלגוריתם. **ככל שהפרויקט הזה התקדם אני הייתי למנחה היוצר וזה הפך לעובד זוט שמשלב מסיום כבר לא היה בו כלל צורך.** מה שחשוב לי שתיקחו מפה זה את העובדה שאין תחליף ליצירתיות, התנסות וחשיבה אנושית. ושכל אחד ייקח את זה לאיזה מקום שהוא רוצה.

כפי שהוצג כאן קודם לכן, את האלגוריתם הזה ניתן להרחיב וכמובן לבנות ממנו סימולטור או צ'יטר ([cheater](#)) שלם לכל מי שחפץ בכך. אך זו איננה המטרה. המטרה העיקרית לכל מי שהגיע עד לחלק זה במאמר היא להבין כיצד לפרק קונספט מופשט של בעיה מחיי היום-יום ולבנות ממנה פתרון היוריסטי. וכמובן שהדברים העיקריים שניתן לקחת מתהליך המחקר הזה זה הלמידה, העומק, ידיעת נושאים חדשים, ההבנה כי לא תמיד המטרה שלכם תביא לפתרון המיוחל (אך זה אינו שולל שהיא תפתור פתרונות נוספים אחרים בנושאים שונים לגמרי) וכמובן איך לא - להיות מסוגלים לנצח את קטאן!

לכל מי שהגיע עד לפה, אני מודה לכם על הקריאה ועל ההעמקה. ומפה יש לי בקשה אחת אליכם, אל תשכחו ללכת לסלון הבית של חברים שלכם, לעצבן את כולם מסביב, להגיע למצב שאף אחד לא רוצה לסחור אתכם, לטשטש את הנמלים שעל המפה של קטאן, להגניב את חפיסת קלפי הפיתוח מתחת לשולחן. ואני יכול להבטיח לכם **אתם תנצחו!**

למעבר לפרויקט ב-GitHub: [לחצו כאן](#)

הבהרה: כל המאמר שקראתם הוקלד על ידי בלבד ולא נכתב על ידי אף צ'אט AI. התמונות בחלקן נבנו ע"י יוצר התמונות ב-ChatGPT או באמצעות תוכנת "הצייר".



על המחבר

גל ראובני, בן 22. כיום סטודנט לתואר ראשון במדעי המחשב באוניברסיטה הפתוחה, רכז הדרכה ארצי בתוכנית "מגשימים AI", ראש צוות קורס פרויקט, מדריך שפת C בתוכנית "מגשימים סייבר" ויוצר הערוץ "[החזית הביטחונית](#)" (ערוץ תוכן OSINT, קהילה בת אלפי עוקבים עם למעלה ממיליון צפיות, בכל הפלטפורמות).

• אימייל: reuvenigal100@gmail.com

• [לינקדאין](#)

קצת עליי: את דרכי בעולם המחשבים התחלתי בכיתה ה', כשקיבלתי את המחשב הראשון שלי. מהר מאוד מצאתי את עצמי מפרק, בודק, לומד ובונה דברים בעצמי, והסקרנות הזאת נשארה איתי מאז.

הדרך שלי ללימודי מדעי המחשב לא הייתה שגרתית. במהלך הילדות עברתי בין מספר בתי ספר, ובסוף כיתה י"א עזבתי את מערכת החינוך. כמה שנים לאחר מכן, אחרי שהתחלתי ללמוד באופן עצמאי וסיימתי בין השאר את הקורס [CS50](#) של אוניברסיטת Harvard, החלטתי להתחיל תואר במדעי המחשב באוניברסיטה הפתוחה.

במקביל ללימודי עסקתי במשך שנים גם בעולמות הגאופוליטיקה וה-OSINT ([מדיעין ממקורות גלויים](#)). הקמתי את "החזית הביטחונית", שהחל כערוץ דיווחי חדשות ובהמשך הפך לפלטפורמה שבה עסקתי בניתוח אירועים ביטחוניים מורכבים, תצלומי לוויין והפקת מידע ממקורות גלויים.

כיום אני רכז הדרכה ארצי בתוכנית "מגשימים AI", רש"צ פרויקט, מדריך C בתוכנית "מגשימים סייבר" וסטודנט למדעי המחשב. אחד הדברים שאני אוהב במיוחד בעבודה שלי זה האפשרות לתת לבני נוער הזדמנות להיכנס לעולם הטכנולוגי, לפעמים בדיוק בנקודה שבה מישו צריך לפתוח עבורם את הדלת, משהו שלי לא היה בזמנו.

אני לומד ולא מפסיק לחקור דברים חדשים, והמאמר הזה הוא רק אחד מהם. אבל מעל לכל, אני נכון לכל אתגר. אתגר זה חלק ממני.

"Fear is a luxury for men who have somewhere to run."

אם אתם מחפשים מישו שחושב על דברים בצורה קצת אחרת ולא מפחד מאתגרים, אז מצוין, אני כנראה מחפש גם אתכם. תדעו שיש לכם אל מי לפנות.

זמין לכם בפרטי.



מקורות מידע

- (p. 1) Academic honors thesis (Syracuse) - https://surface.syr.edu/honors_capstone/186/
- (p. 1) Academic paper on Catan strategies (Guhe & Lascarides) - <https://homepages.inf.ed.ac.uk/alex/papers/semdial2013.pdf>
- (p. 1) Stanford CS230 project on Catan - https://cs230.stanford.edu/projects_fall_2021/reports/103176936.pdf
- (p. 2) Catan (the game) - <https://en.wikipedia.org/wiki/Catan>
- (p. 2) Brute-force search - https://en.wikipedia.org/wiki/Brute-force_search
- (p. 6) Normal distribution - https://en.wikipedia.org/wiki/Normal_distribution
- (p. 7) Dwarves & Dice (YouTube) - <https://www.youtube.com/@DwarvesAndDice>
- (p. 7) Demonstration video (YouTube) - <https://www.youtube.com/watch?v=b46lzp8Fw8&list=LL&index=10>
- (p. 7) Adjacency matrix - https://en.wikipedia.org/wiki/Adjacency_matrix
- (p. 8) Heat map - https://en.wikipedia.org/wiki/Heat_map
- (p. 8) Pseudocode - <https://en.wikipedia.org/wiki/Pseudocode>
- (p. 11) Component (graph theory) - [https://en.wikipedia.org/wiki/Component_\(graph_theory\)](https://en.wikipedia.org/wiki/Component_(graph_theory))
- (p. 11) Graph theory - https://en.wikipedia.org/wiki/Graph_theory
- (p. 12) Bucket sort - https://en.wikipedia.org/wiki/Bucket_sort
- (p. 12) Depth-first search - https://en.wikipedia.org/wiki/Depth-first_search
- (p. 16) Bipartite graph - https://en.wikipedia.org/wiki/Bipartite_graph
- (p. 21) Heuristic (computer science) - [https://en.wikipedia.org/wiki/Heuristic_\(computer_science\)](https://en.wikipedia.org/wiki/Heuristic_(computer_science))
- (p. 21) Performance tuning - https://en.wikipedia.org/wiki/Performance_tuning
- (p. 21) Hyperparameter optimization - https://en.wikipedia.org/wiki/Hyperparameter_optimization
- (p. 23) Diminishing returns - https://en.wikipedia.org/wiki/Diminishing_returns
- (p. 26) Canadian traveler problem - https://en.wikipedia.org/wiki/Canadian_traveller_problem
- (p. 26) PSPACE-complete - <https://en.wikipedia.org/wiki/PSPACE-complete>
- (p. 28) Dijkstra's algorithm - https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- (p. 30) Feature scaling - https://en.wikipedia.org/wiki/Feature_scaling
- (p. 30) Normalization (statistics) - [https://en.wikipedia.org/wiki/Normalization_\(statistics\)](https://en.wikipedia.org/wiki/Normalization_(statistics))
- (p. 30) Exponential decay - https://en.wikipedia.org/wiki/Exponential_decay
- (p. 31) Proof by exhaustion - https://en.wikipedia.org/wiki/Proof_by_exhaustion
- (p. 31) Strategy (game theory) - [https://en.wikipedia.org/wiki/Strategy_\(game_theory\)](https://en.wikipedia.org/wiki/Strategy_(game_theory))
- (p. 31) Nash equilibrium - https://en.wikipedia.org/wiki/Nash_equilibrium
- (p. 33) Exponential decay - https://en.wikipedia.org/wiki/Exponential_decay
- (p. 33) Reinforcement learning - https://en.wikipedia.org/wiki/Reinforcement_learning



- (p. 36) Voronoi diagram - https://en.wikipedia.org/wiki/Voronoi_diagram
- (p. 36) Delaunay triangulation - https://en.wikipedia.org/wiki/Delaunay_triangulation
- (p. 43) Multi-source Dijkstra (NetworkX docs) - https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.shortest_paths.weighted.multi_source_dijkstra.html
- (p. 46) Selection bias - https://en.wikipedia.org/wiki/Selection_bias
- (p. 49) Dice Odds for Settlers of Catan (Stanford) - <https://cs.stanford.edu/people/nick/settlers/DiceOddsSettlers.html>
- (p. 49) JSettlers2 (GitHub) - <https://github.com/jdmonin/JSettlers2>
- (p. 51) Discrete uniform distribution - https://en.wikipedia.org/wiki/Discrete_uniform_distribution
- (p. 54) Correlation does not imply causation - https://en.wikipedia.org/wiki/Correlation_does_not_imply_causation
- (p. 55) Time complexity - https://en.wikipedia.org/wiki/Time_complexity
- (p. 55) Maximum coverage problem - https://en.wikipedia.org/wiki/Maximum_coverage_problem
- (p. 57) Cheating in video games - https://en.wikipedia.org/wiki/Cheating_in_video_games
- (p. 57) Breaking Catan (GitHub repo) - <https://github.com/OctaPakta/Breaking-Catan>
- (p. 57) Personal link - <https://linktr.ee/TheSecurityFront>
- (p. 58) Contact (email) - reuvenigal100@gmail.com
- (p. 58) LinkedIn - <https://www.linkedin.com/in/gal-r-59b82b1ab/>
- (p. 58) CS50 (Harvard) - <https://pll.harvard.edu/course/cs50-introduction-computer-science>
- (p. 58) Open-source intelligence - https://en.wikipedia.org/wiki/Open-source_intelligence