אלגוריתמים קומבינטוריים ־ הרצאות

גדי אלכסנדרוביץ'

תוכן עניינים

1 מבוא

מטרת קורס זה היא לתת היכרות בסיסית עם מדעי המחשב מנקודת מבט שהיא "מתמטית" באופיה. לצורך כך נציג מספר נושאים אשר נלמדים בקורסי הבסיס של מדעי המחשב העוסקים במבני נתונים ואלגוריתמיים: אלגוריתמי מיון, אלגוריתמים שפועלים על גרפים ואלגוריתמים של קידוד. בסיוע הנחה נוספת על ידע בסיסי באלגברה מופשטת נוכל להציג גם אלגוריתמים בסיסיים של תורת המספרים וקריפטוגרפיה, שבמדעי המחשב נלמדים בשלב מתקדם יותר כחומר בחירה.

2 אלגוריתמי מיון

2.1 מבוא

2.1.1 מהו אלגוריתם מיון?

מטרתם של אלגוריתמי מיון היא לקחת כקלט רשימה של איברים ולהחזיר רשימה חדשה המסודרת "מהקטן לגדול". הנה כמה דוגמאות לקלטים כאלו:

1. רשימה של מספרים טבעיים, למשל $[13, 53, 5, 18]$, כשהפלט הצפוי הוא $[5, 13, 18, 53]$.
2. רשימה של קלפי משחק, כשהסדר ביניהם נקבע על בסיס חוקי הברידג', למשל $[K\diamondsuit, 2\spadesuit, 5\diamondsuit, 8\clubsuit]$ כשהפלט הצפוי הוא $[8\clubsuit, 5\diamondsuit, K\diamondsuit, 2\spadesuit]$.
3. רשימה של כוכבי לכת במערכת השמש כשהסדר ביניהם נקבע על בסיס הקרבה לשמש.
למשל [Earth, Jupyter, Mars, Venus] כשהפלט הצפוי הוא [Venus, Earth, Mars, Jupyter]

בדוגמאות הללו אנו רואים כי זהות האובייקטים שאנו מסדרים יכולה להיות מורכבת, וגם הכלל שקובע את הסדר ביניהם עשוי להיות מורכב, אך בכל המקרים שבהם נעסוק נניח כי בין כל אברי הקבוצה קיים יחס סדר **לינארי**, כלומר לכל זוג איברים שונים a, b מתקיים או $a < b$ או $b < a$. במדעי המחשב נהוג לבטא את יחס הסדר הזה באמצעות **מפתח** שמתואם לכל איבר כך שההשוואה מתבצעת בין המפתחות. כך למשל עבור כוכבי לכת, המפתח יהיה מרחקם מהשמש; עבור קלפי משחק ניתן להתאים לכל קלף מספר בין 1 ל־52 באופן ייחודי בהתאם לסדרה שלו ולערכו, וכן הלאה.

באופן כללי ייתכן שלאיברים שונים יהיו מפתחות זהים. דרישה אפשרית אחת מאלגוריתם מיון היא שיהיה **יציב**, במובן זה שהסדר בין איברים בעלי אותו מפתח לא ישתנה בעקבות הפעלת אלגוריתם המיון. כל עוד לא נעסוק בדרישה זו נוכל להניח כי לכל האיברים מפתחות שונים (כי עבור כל זוג איברים בעלי אותו מפתח אפשר לבחור באופן שרירותי מי יהיה הקטן יותר ולשנות את המפתח בהתאם).

על מנת לפשט את התיאורים בהמשך נניח שכל המפתחות הם מספרים ממשיים, אך כל מה שנציג יהיה תקף לכל קבוצה של מפתחות שהיא סדורה לינארית.

הגדרה 2.1 בעיית המיון: בהינתן סדרה של n מספרים ממשיים שונים $a = (a_0, \ldots, a_{n-1})$, בעיית המיון היא בעיית המציאה של התמורה היחידה $\pi \in S_n$ המקיימת $a_{\pi(1)} < \ldots < a_{\pi(n)}$ (כאשר S_n היא קבוצת התמורות על האיברים $\{0, \ldots, n-1\}$). התמורה π נקראת **התמורה הממיינת** של a.

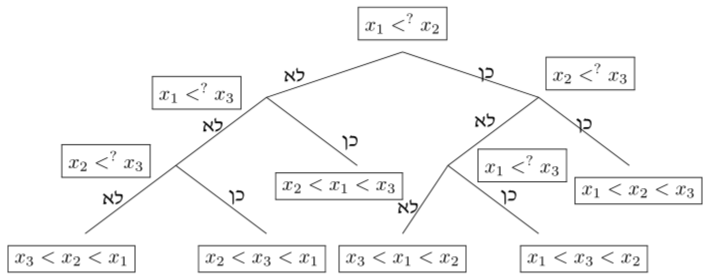
קיימים אלגוריתמים שעושים שימוש בתכונות ספציפיות של המפתחות על מנת לייעל את תהליך המיון. כך ניתן לציין את **מיון מניה** ואת **מיון בסיס** שמתבססים על ההנחה שהמפתחות הם מספרים טבעיים בתחום חסום וידוע, ואת **מיון סלים** שמתבסס על ההנחה שהמפתחות הם מספרים ממשיים המתפלגים בצורה אחידה. לעת עתה ננתח אלגוריתמים שהשימוש היחיד שלהם במפתחות הוא **בהשוואה** של מפתחות של איברים שונים זה לזה. במילים אחרות, האלגוריתמים שנציג יתבססו על שתי פעולות בסיס בלבד:

- **העתקה** של איברים ־ החלפת המקום של איברים בתוך רשימה קיימת או העתקת איבר מרשימה קיימת למקום ברשימה חדשה.
- **השוואה** של המפתחות של איברים: בהינתן שני איברים x_i, x_j, בדיקה האם $x_i < x_j$ וקבלת תשובה של "כן/לא".

כמו כן על מנת לשמור את הדיון פשוט ככל הניתן, האלגוריתמים שלנו יהיו **דטרמיניסטיים** (לא יבצעו הגרלות).

2.1.2 עץ השוואות - מבוא אינטואיטיבי

ניתן לתאר באופן קומפקטי את פעולת אלגוריתם דטרמיניסטי מבוסס השוואות על קלט מגודל n בעזרת **עץ** המכונה "עץ השוואות". המסלולים בעץ מייצגים ריצות אפשריות שונות של האלגוריתם, וצמתיו הפנימיים מייצגים את ההשוואות שאותן ריצות מבצעות. העלים כוללים את הפלט של האלגוריתם. הנה דוגמא לעץ השוואות אפשרי עבור $n = 3$:



האלגוריתם המתואר על ידי האיור פועל על קלט $a = (a_0, a_1, a_2)$ באופן הבא: הוא בונה מסלול החל משורש העץ, כך שבכל צומת פנימי בעץ עם הסימון $x_i <^? x_j$, האלגוריתם בודק האם $a_i < a_j$. אם התשובה היא "כן", האלגוריתם מוסיף למסלול את הבן הימני של הצומת וממשיך אליו; אם התשובה היא "לא" הוא מוסיף למסלול את הבן השמאלי וממשיך אליו. כאשר האלגוריתם מגיע לעלה, הוא מוציא פלט בהתאם לסימון של העלה: אם הסימון הוא $x_i < x_j < x_k$, הפלט של האלגוריתם הוא הרשימה (a_i, a_j, a_k).

כל מסלול בעץ הוא בעל שלושה צמתים פנימיים ולכן בכל ריצה של האלגוריתם מתבצעות שלוש השוואות. האם קיים אלגוריתם שיכול להסתפק תמיד רק בשתי השוואות? נראה זאת בהמשך.

2.1.3 עץ השוואות - הגדרות פורמליות

נזכיר כי **עץ בינארי** T הוא גרף מכוון עם שורש r כך שדרגת היציאה של כל צומת היא לכל היותר 2, ואנו משיתים על העץ מבנה נוסף בכך שאנו מבדילים בין "בן ימני" ובין "בן שמאלי" של צומת (לכל צומת יש ארבע אפשרויות: או שאין לו בנים כלל, או שיש לו שני בנים שאחד ימני והשני שמאלי, או שיש לו ימני בלבד, או שיש לו שמאלי בלבד). עץ בינארי נקרא **מלא** אם כל צומת בו הוא בעל דרגת יציאה 0 או 2.

הבנים השמאלי והימני של צומת v (אם הם קיימים) מסומנים על ידי $\text{left}(v), \text{right}(v)$ בהתאמה. האב של צומת v (אם הוא קיים) מסומן על ידי $\text{parent}(v)$.

עלה בעץ הוא צומת בעל דרגת יציאה 0.

גובה העץ $h(T)$ הוא מספר הקשתות המקסימלי במסלול המכוון המחבר את השורש r אל עלה בעץ (דהיינו, האורך המקסימלי של מסלול מכוון בעץ).

הגדרה 2.2 יהא n טבעי ויהא T עץ בינארי מלא שבו כל קודקוד פנימי v מתויג על ידי השוואה $q(v) = x_i <^? x_j$ עבור $1 \leq i, j \leq n$ וכל עלה v מתויג על ידי תמורה $\pi(v) \in S_n$. לכל וקטור $a = (a_0, \ldots, a_{n-1}) \in \mathbb{R}^n$ נגדיר את **מסלול החישוב** של a על העץ T בתור הסדרה $v_0, v_1, \ldots, v_m$ המוגדרת באופן אינדוקטיבי:

- **בסיס**: $v_0 \triangleq r$ (הסדרה מתחילה בשורש העץ)
- **צעד**: אם v_k הוא עלה, הסדרה מסתיימת באיבר זה. אחרת v_k הוא קודקוד פנימי המתויג על ידי $q(v_k) = x_i <^? x_j$ ונגדיר

$$v_{k+1} \triangleq \begin{cases} \text{right}(v_k) & a_i < a_j \\ \text{left}(v_k) & a_i \not< a_j \end{cases}$$

נסמן את מסלול החישוב של a על העץ T על ידי $\text{Path}_T(a)$ (נשמיט את T כאשר הוא ברור מההקשר).

בהגדרה שלעיל אין התחייבות לנכונות של החישוב שמבצע T; לשם כך נשתמש בהגדרה נוספת.

הגדרה 2.3 יהא T עץ מתויג כמו בהגדרה 2.2. נאמר ש-T הוא **עץ השוואות למיון** n איברים אם לכל $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ מתקיים שהתמורה הממיינת (הגדרה 2.1) של a שווה ל-$\pi(v_m)$, כך ש-v_m הוא האיבר האחרון בסדרה $\text{Path}_T(a)$ ו-$\pi(v_m)$ הוא התיוג של העלה v_m בעץ T.

טענה 2.4 כל אלגוריתם מיון המבוסס על השוואות מגדיר לכל מספר טבעי n עץ השוואות יחיד המתאר את פעולתו על רשימות מאורך n.

לא נוכיח טענה זו פורמלית אך האינטואיציה מאחוריה ברורה: כדי למצוא את עץ ההשוואות פשוט נריץ את האלגוריתם מספר פעמים כך שעבור כל השוואה שהאלגוריתם מבצע נראה פעם אחת לפחות את תגובתו לתשובת "כן" ופעם אחת לפחות את תגובתו לתשובת "לא".

הגדרה 2.5 יהא A אלגוריתם מיון. **סיבוכיות האלגוריתם** A היא פונקציה $c_A(n):\mathbb{N}\to\mathbb{N}$ המחזירה לכל n את מספר ההשוואות המקסימלי שיבצע A על קלט שהוא רשימה מאורך n. פורמלית: $c_A(n)\triangleq h(T)$ כאשר T הוא עץ ההשוואות למיון n איברים שמגדיר A.

נשים לב לכך שהגדרה זו שונה מעט מההגדרות ה"רגילות" של סיבוכיות אלגוריתמים שמודדות שימוש בזמן חישוב או זיכרון; כאן אנחנו מודדים בצורה ישירה את **מספר ההשוואות** שמבצע האלגוריתם, מתוך הנחה שממילא מספר השוואות זה עומד ביחס ישר עם זמן החישוב הכולל של האלגוריתם.

2.1.4 חסם תחתון לסיבוכיות של אלגוריתמי מיון

מודל עץ ההשוואות מאפשר לנו לזהות בקלות יחסית את מספר ההשוואות המינימלי שנדרש מאלגוריתם מיון. ראשית נזכיר תוצאה בסיסית על עצים בינאריים:

טענה 2.6 בעץ בינארי מגובה k יש לכל היותר 2^k עלים.

הוכחה: באינדוקציה על k. המקרה של $k=0$ הוא עץ בעל צומת בודד (שהוא גם עלה). ניקח כעת עץ T מעומק $k+1$ ונסיר ממנו את כל העלים. נקבל עץ T' מעומק k ולכן מהנחת האינדוקציה יש בו לכל היותר 2^k עלים. כעת, כל עלה בעץ T הוא בן של אחד מהעלים בעץ T'. מכיוון שב־T' יש לכל היותר 2^k עלים, ולכל עלה כזה יכולים להיות לכל היותר 2 בנים ב־T, אז ב־T יש לכל היותר $2\cdot 2^k=2^{k+1}$ עלים. ■

כעת, עץ השוואות למיון של n איברים חייב לכלול בעלים שלו כל אחת מ־$n!$ התמורות ב־S_n, כי אם חסרה תמורה $\sigma\in S_n$ אז העץ לא יחזיר תשובה נכונה על הקלט $a=(\sigma(0),\dots,\sigma(n-1))$. בשילוב עם הטענה על הקשר בין גובה של עץ בינארי ומספר העלים שלו נקבל שכל עץ השוואות למיון n איברים T מקיים $n!\le 2^{h(T)}$, כלומר $\lg(n!)\le h(T)$ (lg הוא סימון מקוצר של $\log_2$).

אפשר להעריך את $\lg(n!)$ בעזרת **נוסחת סטירלינג**, $n!\sim\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$, אבל ניתן לקבל חסם תחתון על $\lg(n!)$ בצורה פשוטה יותר.

ראשית, נזכור כי $\lg(x)=\frac{\ln x}{\ln 2}$ ולכן מספיק למצוא חסם על $\ln(n!)$ ולכפול בקבוע $\ln 2$ - זה לא משנה את החסם האסימפטוטי.

שנית, $\ln(n!)=\sum_{j=1}^{n}\ln(j)=\sum_{j=2}^{n}\ln(j)$, לכן מספיק שנמצא חסם תחתון על הסכום הזה.

מכיוון ש־ln היא פונקציה מונוטונית עולה וחיובית בקטע $[1,\infty)$ ניתן לתת לסכום חסם בעזרת התבססות על הגדרת האינטגרל בעזרת **סכומי דארבו**: נחלק את הקטע $[1,n]$ ל־$n-1$ אינטרוולים מאורך 1 ($[1,2],[2,3],\dots,[n-1,n]$). ערך הפונקציה ln על נקודת הקצה **הימנית** של הקטע היא חסם עליון לערכה בקטע כולו, ומכאן שמתקיים

$$\begin{aligned}\sum_{j=2}^{n}\ln j &\ge \int_1^n \ln x\,dx\\ &=\left[x\ln x-x\right]_1^n\\ &=n\ln n-n+1\\ &=\Omega(n\log n)\end{aligned}$$

קיבלנו כי $h(T)=\Omega(n\log n)$ עבור **כל** עץ השוואות עבור n איברים, ומכאן ש־$c_A(n)=\Omega(n\log n)$ עבור **כל** אלגוריתם מיון מבוסס השוואות בלבד. בהמשך נראה אלגוריתמי מיון שאכן מממשים את החסם התחתון הזה.

2.2 מיון הכנסה

2.2.1 תיאורו של מיון הכנסה

מיון הכנסה אינו אלגוריתם יעיל במיוחד במימוש על ידי מחשב, אבל הוא טבעי יחסית לבני אדם ורבים משתמשים בו כדי למיין יד של קלפים, למשל. אינטואטיבית, בכל שלב מרימים קלף חדש, מוצאים את המקום שמתאים לו ביד (בין אילו שני קלפים להכניס אותו או האם לשים אותו בהתחלה/בסוף), מכניסים אותו ועוברים לקלף הבא.

במימוש אלגוריתמי, המיון יפעל כך על רשימה $(a_0, a_1, \ldots, a_{n-1})$: על האיבר הראשון a_0 נחשוב כאילו הוא לבדו מהווה רשימה ממויינת. כעת נניח באינדוקציה שכל האיברים $a_0, a_1, \ldots, a_{k-1}$ כבר ממויינים ב־k המקומות הראשונים ברשימה, ונתבונן על האיבר הבא, a_k. כל עוד הוא קטן מהאיבר שבא מייד לפניו ברשימה, נחליף ביניהם; נעצור כאשר a_k יגיע לראשית הרשימה (במקרה שבו הוא הקטן ביותר מבין $k+1$ האיברים הראשונים) או כאשר האיבר שלפניו קטן ממנו.

בשיטה זו, האלגוריתם אינו זקוק למקום נוסף כדי למיין את הרשימה ־ המיון מתבצע **במקום** (In-Place), תוך שינוי הרשימה שהתקבלה כקלט (אפשר לשמר את הרשימה המקורית על ידי יצירת עותק שלה והפעלת המיון על העותק).

בשפת Python האלגוריתם נראה כך:

```
1 def insertion_sort(a):
2     for k in range(1, len(a)):
3         key = a[k]
4         i = k - 1
5         while i >= 0 and a[i] > key:
6             a[i+1] = a[i]
7             i = i - 1
8         a[i+1] = key
```

2.2.2 הסיבוכיות של מיון הכנסה

מהי סיבוכיות האלגוריתם, שנסמן $c_{\mathrm{IS}}(n)$? כפי שראינו, המקרה הגרוע ביותר מבחינת האלגוריתם הוא זה שבו a_k קטן מכל האיברים שלפניו, כי אז מספר ההשוואות הוא הגדול ביותר. כלומר, הקלט הגרוע ביותר מבחינת האלגוריתם הוא הרשימה $(n, n-1, n-2, \ldots, 1)$. בעזרת קלט זה נוכל לחשב במדויק את הסיבוכיות:

משפט 2.7 $c_{\mathrm{IS}}(n) = \binom{n}{2}$

הוכחה: בריצת האלגוריתם על רשימה מאורך n, הלולאה החיצונית (שורה 2) מתבצעת בדיוק $n-1$ פעמים עבור הערכים $k \in \{1, 2, \ldots, n-1\}$. עבור הערך k מתבצעות לכל היותר k השוואות (בשורה 5, עם האיברים במקומות ה־$0, 1, \ldots, k-1$) כך שהאלגוריתם מבצע לכל היותר $1 + 2 + \ldots + n - 1 = \frac{n(n-1)}{2} = \binom{n}{2}$ השוואות ומכאן $c_{\mathrm{IS}}(n) \le \binom{n}{2}$.

מצד שני, על הקלט $(n, n-1, n-2, \ldots, 1)$ אכן מתבצעות בדיוק k השוואות בשלב k ולכן $c_{\mathrm{IS}}(n) \ge \binom{n}{2}$ (כי החישוב על קלט זה הוא דוגמא אחת למסלול בעץ ולכן לחסם תחתון על עומק העץ).

משני אלו נסיק את $c_{\mathrm{IS}}(n) = \binom{n}{2}$ כמבוקש. ■

אסימפטוטית, זמן ריצה זה הוא $\Theta(n^2)$, שאינו נחשב לזמן ריצה טוב במיוחד. ישנם אלגוריתמי מיון שזוהי סיבוכיות זמן הריצה שלהם ועדיין נחשבים טובים בזכות ריצה מהירה במקרה **הממוצע** (למשל, האלגוריתם **מיון־מהיר**) אך לא נציג גישה זו כאן.

2.3 מיון מיזוג

2.3.1 תיאורו של מיון מיזוג

כפי שראינו, מיון הכנסה הוא אלגוריתם איטי למדי. על קלט שהוא רשימה של 1,000,000 איברים (סדר גודל ריאליסטי) הוא עלול להזדקק ל־$\binom{10^6}{2} \approx 500,000,000,000$ השוואות ־ מספר מופרז לגמרי. זאת מכיוון שהגודל המקורי של הרשימה הוכפל ב־$\frac{1,000,000}{2}$ בערך. האם ניתן לצמצם מאוד את המספר בו מכפילים, למשל, להכפיל רק ב־40? התשובה היא חיובית ודוגמא לאלגוריתם שמבצע את הקסם הזה ורץ בסיבוכיות $O(n \lg n)$ היא **מיון מיזוג** שנציג כעת.

מיון מיזוג הוא דוגמא טיפוסית לאלגוריתם שפועל בשיטת **הפרד ומשול** ־ הוא מחלק את הקלט שלו לשני חלקים מגודל זהה וממיין כל אחד מהחלקים הללו בנפרד (על ידי קריאה רקורסיבית לעצמו). לאחר מכן הוא **ממזג** את שתי הרשימות הממויינות לרשימה ממויינת אחת. הביצוע היעיל של שלב מיזוג זה הוא המפתח ליעילות האלגוריתם.

נעסוק אם כן בשאלה הבאה: בהינתן שתי רשימות **ממוינות** $a = (a_0, a_1, \dots, a_{n-1})$ ו־$b = (b_0, b_1, \dots, b_{m-1})$ אנו רוצים לחשב את הרשימה הממוינת $c = (c_0, c_1, \dots, c_{n+m-1})$ שאבריה הן בדיוק אברי שתי הרשימות a, b, כלומר $\{c_0, \dots, c_{n+m-1}\} = \{a_0, \dots, a_{n-1}\} \cup \{b_0, \dots, b_{m-1}\}$.

המיזוג מתבצע בצורה אינטואיטיבית: אפשר לחשוב על עלייה של אנשים למטוס, כך שבעלי המקומות המרוחקים יותר נכנסים קודם. נניח כי יש לנו שני תורים שבהם האנשים כבר עומדים בצורה ממויינת. בכל רגע נתון בודקים את מספר המקום של שני האנשים שבראש התור ומכניסים אל המטוס את זה מביניהם שאמור להיכנס קודם. אל ראש התור של האדם שנכנס מגיע אדם חדש וכעת משווים אותו עם האדם שבראש התור השני וכן הלאה. בסופו של דבר אחרי שתור אחד מתרוקן נותנים לאנשים בתור השני להיכנס לפי הסדר, בלי צורך בהשוואות נוספות.

במימוש במחשב נפעל כך: בכל שלב אנו מחזיקים את האינדקסים של האיברים הקטנים ביותר ב־a, b שטרם הכנסנו אל c, משווים את שני האיברים הללו, מכניסים אל c את הקטן יותר ומקדמים את האינדקס של הסדרה של האיבר שהכנסנו. כאשר בשלב מסויים אחת הרשימות מתרוקנת (דהיינו, האינדקס שלה עובר את האיבר האחרון בה) אנחנו עוברים להכנסה סדרתית של אברי הרשימה השניה.

בשפת Python האלגוריתם נראה כך:

```
1   def merge(a,b):
2       n = len(a)
3       m = len(b)
4       i, j = 0, 0
5       c = []
6       while i < n and j < m:
7           if a[i] < b[j]:
8               c.append(a[i])
9               i = i + 1
10          else:
11              c.append(b[j])
12              j = j + 1
13      if i == n:
14          while j < m:
15              c.append(b[j])
16              j = j + 1
17      else:
18          while i < n:
19              c.append(a[i])
20              i = i + 1
21      return c
```

שורות 6־12 אחראיות לשלב הראשון של המיזוג שבו טרם רוקנו אף אחת מהרשימות; שורות 13־16 מכסות את המקרה שבו הרשימה a התרוקנה ושורות 17־20 מכסות את המקרה שבו הרשימה b ריקה.

נשים לב כי באופן שבו מימשנו את האלגוריתם, הוא יוצר רשימה חדשה c ומחזיר אותה; אנחנו לא מבצעים מיון "במקום" כפי שקרה במיון מיזוג.

כעת נשתמש בפונקציה merge שהגדרנו כדי לממש את אלגוריתם מיון מיזוג המלא. להבדיל מהמימוש הארוך יחסית של merge, המיון כולו ניתן לתיאור קומפקטי, כי המורכבות שלו מוחבאת בתוך merge ובתוך הקריאה הרקורסיבית, וכל שנותר לו לעשות הוא "ניהול חשבונות" - לבדוק אם אנחנו במקרה הקצה של רשימה ריקה או בת איבר אחד (ואז ניתן להחזיר אותה), לפרק את הרשימה לשתי רשימות ולמיין אותן רקורסיבית, ואז למזג את התוצאה:

```
1 def merge_sort(a):
2     n = len(a)
3     if n == 0 or n == 1:
4         return a
5     k = n // 2
6     b = merge_sort(a[:k])
7     c = merge_sort(a[k:])
8     return merge(b,c)
```

- בשורה 5 מחושב אינדקס "אמצע" הרשימה, $k = \left\lfloor \frac{n}{2} \right\rfloor$.
- בשורה 6 נבנית תת־הרשימה $b = (a_0, a_1, \dots, a_{k-1})$.

- בשורה 7 נבנית תת־הרשימה $c = (a_k, a_{k+1}, \dots, a_{n-1})$.

2.3.2 הסיבוכיות של מיון מיזוג

הסיבוכיות של מיון מיזוג מתקבלת משילוב של שני מרכיבים: ניתוח הסיבוכיות של הפונקציה merge, וניתוח הסיבוכיות של הקריאה הרקורסיבית. נתחיל עם הפונקציה merge.

טענה 2.8 נסמן ב־$c_M(n,m)$ את הסיבוכיות של merge על רשימות מאורכים n,m. אז $c_M(n,m) \le n+m-1$

הוכחה: בתוך merge מתבצעת השוואה רק בשורה 7, כחלק מהלולאה המרכזית. הלולאה הזו מתחילה כאשר $i=0, j=0$ ומסתיימת כאשר $i=n-1$ או $j=m-1$, ובכל איטרציה שלה מגדילים ב־1 או את i או את j.

נתבונן על המספר $i+j$. המספר הזה שווה בתחילת הלולאה ל־$0+0=0$ ובסוף הלולאה הוא קטן או שווה ל־$(m-1)+(n-1)=n+m-2$. כמו כן, בכל איטרציה $i+j$ גדל בדיוק ב־1. לכן מספר האיטרציות הכולל (ומכאן גם מספר ההשוואות הכולל) הוא לכל היותר $n+m-1$ (עבור הערכים $0,1,2,\dots,n+m-2$ של $i+j$). ■

נעבור אל מיון מיזוג עצמו:

משפט 2.9 נסמן ב־$c_{MS}(n)$ את הסיבוכיות של מיון מיזוג על רשימה מאורך n, אז $c_{MS}(n) = O(n\log n)$

הוכחה: ראשית נשים לב לכך שעבור $n>1$ ניתן לחסום את הסיבוכיות של מיון מיזוג על ידי סכום הסיבוכיות של שתי ההפעלות הרקורסיביות שלו ועוד פעולת המיזוג:

$$\begin{aligned}
c_{MS}(n) &\le c_{MS}\left(\left\lfloor \frac{n}{2}\right\rfloor\right) + c_{MS}\left(\left\lceil \frac{n}{2}\right\rceil\right) + c_M\left(\left\lfloor \frac{n}{2}\right\rfloor, \left\lceil \frac{n}{2}\right\rceil\right)\\
&\le 2c_{MS}\left(\left\lceil \frac{n}{2}\right\rceil\right) + \left(\left\lfloor \frac{n}{2}\right\rfloor + \left\lceil \frac{n}{2}\right\rceil - 1\right)\\
&\le 2c_{MS}\left(\left\lceil \frac{n}{2}\right\rceil\right) + n - 1
\end{aligned}$$

נשתמש באי השוויון שקיבלנו כדי להוכיח באינדוקציה שלכל k טבעי מתקיים $c_{MS}\left(2^k\right) \le k\cdot 2^k$.

- בסיס: עבור $k=0$, $c_{MS}\left(2^k\right) - c_{MS}(1) = 0$ כי במקרה זה המיון מחזיר מייד את המערך ללא ביצוע השוואות (שורה 3 באלגוריתם).
- צעד: נניח כי $c_{MS}\left(2^k\right) \le k\cdot 2^k$. כעת:

$$\begin{aligned}
c_{MS}\left(2^{k+1}\right) &\le 2c_{MS}\left(2^k\right) + 2^{k+1} - 1\\
&\le 2\cdot k\cdot 2^k + 2^{k+1} - 1\\
&\le k\cdot 2^{k+1} + 2^{k+1}\\
&\le (k+1)\cdot 2^{k+1}
\end{aligned}$$

כמבוקש.

כעת יהא n כלשהו. אז קיים k טבעי כך ש־$2^{k-1} < n \le 2^k$, ולכן $2^k < 2n$, כלומר $k < \lg(2n) = \lg n + 1$, וכעת:

$$\begin{aligned}
c_{MS}(n) &\le c_{MS}\left(2^k\right)\\
&\le k\cdot 2^k\\
&< (\lg n + 1)\,2n\\
&= 2n\lg n + 2n
\end{aligned}$$

ומכאן שסיבוכיות מיון מיזוג היא $O(n\log n)$, כנדרש. ■

אם נצרף לחסם הסיבוכיות העליון את החסם התחתון $\Omega\left(n\log n\right)$ שהוכחנו באופן כללי עבור אלגוריתמי מיון מבוססי השוואות, נראה שסיבוכיות מיון מיזוג היא $\Theta\left(n\log n\right)$. על פניו מדובר על סיבוכיות אופטימלית, ואכן מיון מיזוג הוא מיון טוב למדי, אבל יש לזכור שבמציאות יש שיקולים נוספים בנוסף לסיבוכיות האסימפטוטית של המקרה הגרוע ביותר (האלגוריתם **מיון מהיר** שלא יילמד בקורס הוא בעל סיבוכיות $\Theta\left(n^{2}\right)$ במקרה הגרוע אבל בפועל מביס על פי רוב את מיון מיזוג).

2.4 מיון ערימה

2.4.1 מבוא אינטואיטיבי

מיון ערימה הוא אלגוריתם מיון יעיל בעל סיבוכיות $\Theta\left(n\log n\right)$ בדומה למיון מיזוג, אך להבדיל ממיון מיזוג הוא מבוצע כולו **במקום** (כלומר, בלי שימוש בזכרון נוסף). הקסם הזה מתבצע באמצעות שימוש חכם במבנה נתונים הנקרא **ערימה** (Heap).

במיון מיזוג, היו לנו שני "תורים" ממוינים של איברים ובכל שלב הוספנו אחד מהם לרשימה הממויינת וקידמנו את התור. בערימה יש לנו **עץ בינארי** שאיננו ממויין אבל **כל מסלול** בו ממוין. בכל שלב, השורש של העץ הוא האיבר הגדול ביותר שנותר בו, וניתן להוציא את השורש, להכניס אותו לרשימה הממויינת ואז לבצע "תיקון" של העץ שמפעפע אל השורש שלו את האיבר הגדול ביותר בו.

האלגורים פועל בשני שלבים:

1. שלב בניית הערימה: בשלב זה לוקחים את רשימת האיברים המבולגנת ויוצרים בתוכה את הסדר החלקי שמתקיים בתוך ערימה. שלב זה דורש רק $O\left(n\right)$ השוואות.

2. שלב המיון: בשלב זה חוזרים שוב ושוב על הצעד הבא: שולפים מראש הערימה את האיבר הגדול ביותר ומתקנים את הערימה בהתאם. "שליפה" כזו ותיקון הערימה בהתאם דורשת רק $O\left(\log n\right)$ השוואות ומכיוון שחוזרים עליה n פעמים, מקבלים סיבוכיות של $O\left(n\log n\right)$.

2.4.2 ערימה וייצוג שלה באמצעות מערך

נפתח בהגדרה פורמלית:

הגדרה 2.10 ערימה היא עץ בינארי T שצמתיו ניתנים להשוואה כך שלכל צומת $v\in T$ שאינו השורש מתקיים $v\le\text{parent}\left(v\right)$.

בפרט, משמעות ההגדרה היא שבכל מסלול מעלה בעץ חזרה אל השורש, הצמתים ממויינים בסדר עולה.

ניתן לייצג עצים בשלל דרכים במחשב, אך מכיוון שמטרתנו כאן היא מיון יעיל של מערך, נציג אופן שבו אפשר לחשוב על מערך $\left(a_{1},a_{2},\dots,a_{n}\right)$ בתור ייצוג של עץ בינארי. בגישה זו, הבנים של הצומת a_{i} יהיו a_{2i} ו־a_{2i+1}, ואילו ההורה של הצומת a_{i} יהיה $a_{\left\lfloor \frac{i}{2}\right\rfloor }$.

נשים לב שכדי שהתעלול המספרי יעבוד, התחלנו את המערך מ־a_{1} ולא מ־a_{0}, כרגיל. כאשר נכתוב קוד מחשב נצטרך להתייחס בצורה כלשהי לכך שכן בדרך כלל מערכים מתחילים מאינדקס 0.

בשפת Python נוכל להשתמש בפונקציות הבאות כדי לבצע את האבסטרקציה הזו:

```
def left(i):
    return 2*i + 1
def right(i):
    return 2*i + 2
def parent(i):
    return (i-1) // 2
```

איננו מציגים כאן אופטימיזציות שניתן לבצע במימוש האלגוריתם (למשל, אפשר לתאר את הפעולות לעיל בתור פעולות על סיביות בודדות, כך למשל כפל ב־2 או חלוקה ב־2 הן פעולות shift של סיביות) אלא מסתפקים בהבנת הרעיון שמאחורי האלגוריתם עצמו.

איור

כפי שניתן לראות, הערימה מיוצגת כך: האיבר הראשון במערך הוא השכבה הראשונה של הץ. שני האיברים הבאים הם השכבה הבאה, וכן הלאה. השכבה האחרונה בעץ מיוצגת על ידי רצף האיברים האחרונים במערך. לכן, אם נקטין את המערך באיבר בודד, הדבר יתבטא בהסרת עלה מהעץ. ניעזר בכך בהמשך, כאשר נעביר לסוף המערך את האיברים שכבר מוינו ונקטין את הערימה בהתאם.

עץ בינארי שבנוי בשכבות כאלו, כך שכל השכבות מלאות פרט אולי לאחרונה, נקרא **עץ בינארי כמעט שלם**.

2.4.3 שמירה על תכונת הערימה

"תכונת הערימה" היא התכונה שתיארנו קודם - כל צומת בערימה גדול מבניו. גם בשלב הבניה הראשוני של הערימה וגם בשלב הוצאת האיברים לצורך המיון, הכלי המרכזי שלנו יהיה פונקציה ש"מתקנת" ערימה שהיא תקינה למעט אולי האפשרות שהשורש שלה **קטן** מאחד מבניו. תיקון של בעיה כזו מבוצע באופן רקורסיבי: השורש מוחלף עם הבן הגדול יותר, וכעת נותר לתקן את תת־הערימה של אותו בן. בצורה הזו השורש המקורי של הערימה "מפועפע" לתחתית הערימה עד שהוא נעצר במקום שבו הוא גדול משני בניו - באופן שמזכיר את מיון הכנסה. מכיוון שבעץ עם n צמתים עומק העץ הוא $\Theta(\log n)$, שלב ה"פעפוע" הזה הוא קצר יחסית.

בשפת Python נממש את הפונקציה הזו בצורה הבאה:

```
def heapify(a, i, n):
    children = []
    left_child = left(i)
    right_child = right(i)
    max_child_value = None
    max_child_index = None
    if left_child < n:
        max_child_value = a[left_child]
        max_child_index = left_child
    if right_child < n and (max_child_value is None
                            or max_child_value < a[right_child]):
        max_child_value = a[right_child]
        max_child_index = right_child
    if max_child_value is not None and a[i] < max_child_value:
        a[i], a[max_child_index] = max_child_value, a[i]
        heapify(a, max_child_index, n)
```

הפונקציה מקבלת שלושה קלטים: את המערך a, את האינדקס i של הצומת שנמצא בשורש תת־הערימה שאנחנו רוצים לתקן, ואת הגודל n של הערימה. בשפת Python אין בדרך כלל צורך לקבל בנפרד את האורך של מערך (להבדיל מבשפת C, למשל) כי הפונקציה len(a) תחזיר את אורכו, אולם במקרה הנוכחי ייתכן שהערימה **אינה** משתמשת בכל המערך (כזכור, הרעיון יהיה שבסוף המערך יהיו האיברים שכבר מוינו, והאיברים שטרם מוינו יהיו בערימה).

הפונקציה מבצעת לכל היותר שתי השוואות (בין שני הבנים של הצומת ובין הצומת לגדול מבין הבנים הללו) וקוראת לעצמה רקורסיבית עבור תת־עץ שהשורש שלו הוא בן של i, כך שסיבוכיות הפונקציה חסומה מלמעלה על ידי $2\cdot h(v_i)$ כאשר v_i הוא הצומת בעץ שהאינדקס שלו במערך הוא i, ו־$h(v_i)$ הוא עומק העץ ש־v_i נמצא בשורש שלו.

2.4.4 בניית ערימה

בהינתן הפונקציה heapify, בניית ערימה היא פשוטה: נעבור על המערך a **מהסוף להתחלה** ועל כל צומת נפעיל את heapify בתורו. המעבר מהסוף להתחלה מבטיח שתמיד תתקיים הנחת היסוד של heapify: כאשר אנו מפעילים אותה על צומת, מובטח לנו שתתי־העצים של בניו הם כבר ערימות.

בשפת Python זה נראה כך:

```
1 def make_heap(a):
2     n = len(a)
3     for i in range(n-1, -1, -1):
4         heapify(a, i, n)
```

כאן בשורה 3 מתבצעת הלולאה $i=n-1,\ldots,0$ (ה־-1 השני אומר לעצור עם ההגעה ל־$i=-1$, לא כולל הצעד הזה; וה־-1 השני אומר שיש להפחית 1 מ־i בכל איטרציה במקום להוסיף 1 כרגיל).

מכיוון שהלולאה מתבצעת n פעמים ובכל פעם מתבצעת קריאה ל־heapify שהסיבוכיות שלה חסומה על ידי $O(\log n)$, קל לראות כי סיבוכיות make_heap היא לפחות $O(n\log n)$. אבל למעשה, ניתוח זהיר יראה לנו כי היא קטנה יותר:

משפט 2.11 סיבוכיות make_heap היא $O(n)$.

המפתח לניתוח הוא פירוק העץ לשכבות. ככל ששכבה היא יותר גבוהה, כך יש בה אקספוננציאלית פחות צמתים, מה שמוביל לסיבוכיות נמוכה משמעותית. **הוכחה:** לכל n טבע, יהא k המספר הטבעי הקטן ביותר כך ש-$2^{k-1} \leq n \leq 2^k$. אז בעץ בינארי כמעט שלם עם n צמתים יש $k+1$ שכבות. שנמספר על ידי $i = 0, 1, \ldots, k$. בשכבה ה־i יש 2^i צמתים לכל היותר וגובהה הוא $k - i$.

כעת, סיבוכיות heapify לכל צומת v_i חסומה כפי שראינו על ידי $2 \cdot h(v_i)$, כך שסיבוכיות make_heap חסומה על ידי:

$$\begin{aligned} \sum_{v \in T} 2h(v) &\leq 2 \sum_{i=0}^{k} 2^i \cdot (k - i) \\ &= 2 \sum_{j=0}^{k} j \cdot 2^{k-j} \\ &= 2^{k+1} \sum_{j=0}^{k} \frac{j}{2^j} \end{aligned}$$

נרצה לחסום את $\sum_{j=0}^{k} \frac{j}{2^j}$. לשם כך ניתן להשתמש בתעלול מתורת טורי החזקות:

$$\begin{aligned} \sum_{j=0}^{\infty} j x^j &= x \sum_{j=0}^{\infty} j x^{j-1} \\ &= x \left(\sum_{j=0}^{\infty} x^j \right)' \\ x \cdot \left(\frac{1}{1-x} \right)' &= \frac{x}{(1-x)^2} \end{aligned}$$

אם נציב $x = \frac{1}{2}$ נקבל $\sum_{j=0}^{k} \frac{j}{2^j} \leq \sum_{j=0}^{\infty} \frac{j}{2^j} = 2$. מכאן שסיבוכיות make_heap חסומה על ידי 2^{k+2}. מכיוון ש-$2^{k+2} \leq 8n$ קיבלנו שסיבוכיות make_heap היא $O(n)$. ■

2.4.5 שימוש בערימה למיון

אחרי עבודת ההכנה שביצענו, מיון ערימה עצמו הוא טריוויאלי: בונים ערימה, ואז במשך $n - 1$ צעדים מעבירים את השורש שלה אל התא האחרון במערך שטרם מוין, מקטינים את גודל הערימה ב־1 ומתקנים אותה בעזרת heapify.

בשפת פייתון הקוד נראה כך:

```
1 def heap_sort(a):
2     n = len(a)
3     make_heap(a)
4     for i in range(1,n):
5         a[0], a[n-i] = a[n-i], a[0]
6         heapify(a, 0, n-i)
```

כאן בשורה 3 נוצרת הערימה, בשורה 5 מתבצעת החלפה בין האיבר בראש הערימה (הגדול ביותר מאלו שטרם מוינו) ובין האיבר בעל האינדקס הגדול ביותר מבין אלו שטרם מוינו, ובשורה 6 מתבצע תיקון הערימה (החל מהשורש שלה ולכן ה־0 שמועבר לפונקציה, וכשגודלה מעודכן בהתאם ולכן ה־$n - i$ שמועבר לפונקציה).

משפט 2.12 הסיבוכיות של מיון ערימה היא $O(n\log n)$.

הוכחה: הסיבוכיות של מיון ערימה מורכבת מהסיבוכיות של make_heap ו־$n-1$ הפעלות של heapify ולכן

$$\begin{aligned} c_{\text{HEAP\_SORT}}(n) &\le c_{\text{MAKE\_HEAP}}(n) + (n-1)\,c_{\text{HEAPIFY}}(n) \\ &\le O(n) + (n-1)\,O(\log n) \\ &\le O(n\log n) \end{aligned}$$

■

2.5 בעיית הבחירה

2.5.1 מבוא

לעתים בהינתן רשימה a אין לנו צורך למיין את כולה, אלא רק למצוא איברים כלשהם שמאופיינים על פי המיקום שלהם ברשימה הממויינת. שלוש הדוגמאות הבסיסיות הן האיבר המינימלי ברשימה (שנמצא במקום הראשון ברשימה הממויינת), האיבר המקסימלי ברשימה (שנמצא במקום האחרון ברשימה הממויינת) ו**החציון** של הרשימה (שנמצא באמצע הרשימה הממויינת).

מציאת מינימום היא עניין פשוט: מאתחלים את המינימום להיות האיבר הראשון ברשימה ואז עוברים סדרתית על הרשימה ומשווים כל איבר למינימום. אם נמצא ברשימה איבר קטן יותר, הוא הופך למינימום החדש. בצורה הזו נזקקים ל־$n-1$ השוואות, ואלגוריתם דומה עובד עבור מציאת מקסימום.

לעומת זאת, הרבה פחות ברור כיצד למצוא חציון ברשימה לא ממויינת. כמובן, ניתן למיין את הרשימה בסיבוכיות $O(n\log n)$ ולבדוק את האיבר באמצע הרשימה, אך כפי שנראה, ניתן לפעול בצורה יעילה יותר ולמצוא את החציון בסיבוכיות $O(n)$. החציון אינו ייחודי כאן ־ נראה שלכל רשימה מאורך n ולכל $0\le k\le n-1$ ניתן למצוא את האיבר במקום ה־k־י ברשימה הממויינת תוך ביצוע לכל היותר $30n$ השוואות.

נגדיר פורמלית את המטרה שלנו:

הגדרה 2.13 בעיית הבחירה: בהינתן סדרה $a=(a_0,\dots,a_{n-1})$ עם תמורה ממיינת $\pi\in S_n$, $a_{\pi(0)}<\dots<a_{\pi(n-1)}$, נגדיר

$$\operatorname{sel}(a,k) = a_{\pi(k)}$$

למשל:

- $\operatorname{sel}(a,0)=\min a$
- $\operatorname{sel}(a,n-1)=\max a$
- החציון של a מוגדר להיות $\operatorname{sel}\left(a,\left\lfloor\frac{n}{2}\right\rfloor\right)$ (אם הרשימה מאורך זוגי זהו **החציון העליון** של a למשל עבור הרשימה הממויינת (a_0,a_1,a_2,a_3) מוחזר a_2).

2.5.2 תיאור האלגוריתם

השיטה שלנו תהיה רקורסיבית: בכל איטרציה נמצא איבר $x\in a$ שהוא "קרוב מספיק להיות החציון" של הרשימה ונבנה שני מערכים חדשים, $c=\{a_i\in a\mid a_i\le x\}$ ו־$d=\{a_i\in a\mid a_i>x\}$. אם יצא שגודל c הוא בדיוק k, הרי שהאיבר במקום k ברשימה הממויינת אמור להיות x, וסיימנו; אם גודל c **גדול** מ־k אפשר להמשיך את החיפוש רקורסיבית בתוך c, ואם גודל c **קטן** מ־k אפשר להמשיך את החיפוש בתוך d. כל עוד האיבר x נבחר בצורה כזו שגם c וגם d יהיו קטנות בצורה מהותית ביחס ל־a, זה מספיק כדי לקבל סיבוכיות לינארית.

הדרך שבה מוצאים את x נדמית קצת קסומה במבט ראשון. ראשית, מחלקים את a לסדרות של חמישה איברים:

$$F_1=(a_0,a_1,a_2,a_3,a_4)$$
$$F_2=(a_5,a_6,a_7,a_8,a_9)$$

וכן הלאה עד לקבלת סדרה $F_1, F_2, \ldots, F_t$, כשה"חמישייה" האחרונה F_t יכולה לכלול גם פחות מחמישה איברים, ומתקיים $t = \lceil \frac{n}{5} \rceil$.

כל חמישייה כזו ניתן למיין בזמן קבוע ולכן ניתן למצוא את החציון שלה בצורה ישירה. נקבל רשימה $b = (b_1, \ldots, b_t)$ של החציונים של של החמישיות. נבחר את x שלנו להיות החציון של הרשימה הזו. נשים לב לכך שלא ניתן למיין את הרשימה הזו כדי למצוא את החציון שלה, בניגוד לתעלול שביצענו עם הרשימות בנות חמש האיברים. תחת זאת, נקרא רקורסיבית לשיטת החיפוש שלנו כדי לטפל ברשימה זו.

בשפת פייתון הקוד נראה כך:

```
1   def select(a, k):
2       n = len(a)
3       if n == 1:
4           return a[0]
5       b = [find_median(a[5*i:5*(i+1)]) for i in range(math.ceil(n / 5))]
6       x = select(b, len(b) // 2)
7       c = [ai for ai in a if ai < x]
8       d = [ai for ai in a if ai > x]
9       if len(c) == k:
10          return x
11      if len(c) > k:
12          return select(c, k)
13      if len(c) < k:
14          return select(d, k - len(c) - 1)
```

כאשר המימוש שלנו של find_median מתבסס על מיון הכנסה:

```
def find_median(f):
    insertion_sort(f)
    return f[len(f) // 2]
```

נשים לב לכך שהקוד אינו אופטימלי ואין בו בדיקת תקינות של הקלטים; זאת כדי לשמור על פשטות הקריאה שלו.

נתאר פרטנית את הקוד של select:

- בשורה 3 נבדק מקרה הקצה של מערך בן איבר בודד שאותו מחזירים בלי קשר לערכו של k.
- בשורה 5 מחולק המערך לסדרה של תתי־מערכים מגודל 5 ועבור כל אחד מהם מוצאים את החציון שלו בצורה ישירה. התוצאה היא רשימת החציונים b.
- בשורה 6 קוראים רקורסיבית ל־select על מנת לחשב את החציון x של הרשימה b.
- בשורות 7־8 נבנים המערכים c, d מתוך האיברים הקטנים והגדולים מ־x בהתאמה (שלב זה מבוצע בצורה לא אופטימלית, עם שתי סריקות של a במקום סריקה בודדת).
- בשורות 9,11,13 נבדקים שלושת המקרים האפשריים לגבי המיקום של האיבר ה־k־י (או שהוא x, או שהוא בתוך c, או שהוא בתוך d). במקרים שבהם האיבר טרם נמצא, מתבצעת קריאה רקורסיבית ל־select עבור המערך המתאים.

2.5.3 ניתוח סיבוכיות

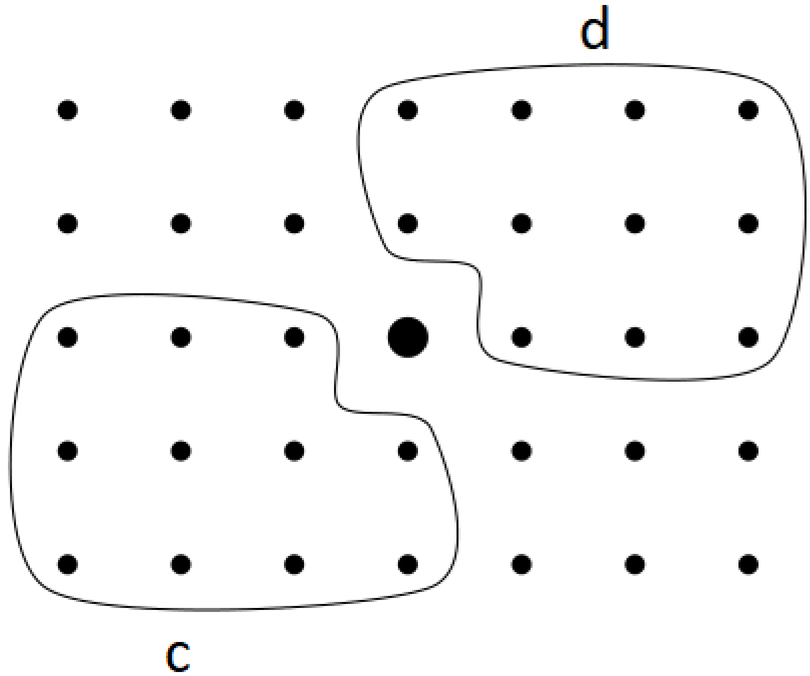
המפתח לניתוח הסיבוכיות של select הוא במציאת חסם על גודל המערכים שעליהם פועלים רקורסיבית. מציאת החציון של b בשורה 5 מתבצעת על מערך מגודל $\lceil \frac{n}{5} \rceil$, אבל מה ניתן לומר על גודלן של c, d? הקבוצות הללו נוצרות על ידי לקיחת איבר

בודד x ופירוק שאר האיברים לקטנים וגדולים ממנו. האם לא קיימת האפשרות שניקח בטעות את x להיות אחד מהאיברים הקטנים במערך ולכן d תהיה עצומה?

התשובה היא לא, כפי שמראה הטענה הבאה:

טענה 2.14 אם $|a| = n$ אז גודל הקבוצות c, d שמתקבלות בשורות 7־8 של select מקיים $|c|, |d| \leq \frac{7n}{10} + 6$

לפני שנוכיח פורמלית את הטענה, הנה האינטואיציה שמאחוריה. נסתכל על המקרה הקונקרטי של $n = 35$ שבו יש $7 = \frac{n}{5} = |b|$ קבוצות של חמישה איברים. נצייר אותן בתור עמודות של 5 נקודות, כך שהעמודות מסודרות על פי הסדר בקבוצת החציונים b (השורה האמצעית). הנקודה הבולטת במרכז היא x ־ התוצאה של מציאת החציון של b בשורה 6.



כעת נתבונן על איברים ש**ידוע לנו בודאות** שיהיו שייכים ל־c, כלומר שידוע לנו בודאות שהם קטנים מ־x. ראשית, כל החציונים ב־b שבאים **לפני** x יהיו כאלו. שנית, כל האיברים בטור של החציונים הללו שקטנים מהם יהיו כאלו. יש 3 חציונים כאלו וכל חציון כזה תורם 3 איברים (הוא עצמו ושני אלו שקטנים ממנו בטור שלו) כך שנקבל 9 איברים לפחות. בנוסף לכך יש גם את האיברים בטור של x אבל כאן יהיה מדובר רק על שני איברים ולא שלושה.

באופן כללי יש בעיה נוספת שלא מתבטאת באיור ־ אחד הטורים עלול לכלול פחות מחמישה איברים. בשתי בעיות אלו נתחשב בהוכחה הכללית. **הוכחה:** גודל הקבוצה b הוא $\left\lceil\frac{n}{5}\right\rceil$, ומספר אברי b שהוא קטן או שווה לחציון x הוא לפחות $\left\lceil\frac{b}{2}\right\rceil$. פירוש הדבר הוא שיש בין הקבוצות $F_1, F_2, \ldots, F_t$ לפחות $\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil$ קבוצות שבהן החציון קטן או שווה ל־x. כל קבוצה כזו תורמת 3 איברים שקטנים מ־x למעט אולי הקבוצה שכוללת את x עצמו וקבוצה אחת נוספת שיש פה פחות מ־5 איברים, ולכן יש לכל הפחות

$$3\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{5}\right\rceil\right\rceil - 2\right) \geq 3\left(\frac{n}{10} - 2\right) = \frac{3n}{10} - 6$$

איברים שקטנים מ־x. מכאן שכמות האיברים ש**גדולים** מ־x היא לכל היותר $n - \left(\frac{3n}{10} - 6\right) = \frac{7n}{10} + 6$, כלומר $|d| \leq \frac{7n}{10} + 6$. אותו הטיעון מראה גם ש־$|c| \leq \frac{7n}{10} + 6$. ■

נעבור כעת לניתוח הסיבוכיות של select כולה. נסמן ב־$c_{\text{SEL}}(n)$ את מספר ההשוואות שמבצעת select במקרה הגרוע על קלט מגודל n. כעת נחסום את $c_{\text{SEL}}(n)$ על ידי בחינת שלבי האלגוריתם.

- בשורה 5 מבוצע מיון הכנסה ל־$\left\lceil\frac{n}{5}\right\rceil$ קבוצות שכל אחת מהן היא מגודל 5 ולכן דורש $\binom{5}{2} = 10$ השוואות לכל היותר,

ולכן שלב זה דורש לכל היותר

$$\begin{aligned}\left\lceil\frac{n}{5}\right\rceil\cdot 10 &\le \frac{10n}{5}+10\\ &= 2n+10 = O(n)\end{aligned}$$

השוואות.

- בשורה 6 מתבצעת קריאה רקורסיבית ל־select שדורשת לכל היותר $c_{\text{SEL}}\left(\left\lceil\frac{n}{5}\right\rceil\right)$ השוואות.
- בשורה 7 משווים את כל אברי a ל־x, כך ששורה זו דורשת n השוואות, וכך גם שורה 8 (ניתן לבצע את שתי השורות במעבר אחד על המערך לצורך שיפור היעילות). בכל מקרה שורות אלו לוקחות $O(n)$ השוואות.
- בשורות 12 ו־14 מתבצעת קריאה רקורסיבית ל־select שדורשת לכל היותר $c_{\text{SEL}}\left(\frac{7n}{10}+6\right)$ השוואות. רק אחת מבין שתי הקריאות הללו יכולה להתבצע בכל פעם.

קיבלנו את הנוסחה הרקורסיבית הבאה:

$$c_{\text{SEL}}(n)\le c_{\text{SEL}}\left(\left\lceil\frac{n}{5}\right\rceil\right)+c_{\text{SEL}}\left(\frac{7n}{10}+6\right)+O(n)$$

משפט 2.15 קיים קבוע t כך ש־$c_{\text{SEL}}(n)\le tn$ לכל $n>0$.

הוכחה: יהא a קבוע כך שהחלק בנוסחה ל־c_{SEL} שלעיל שהוא $O(n)$ יהיה חסום על ידי an לכל n טבעי. כעת נשתמש ב"מספרי קסם" שנבין מהיכן הגיעו בהמשך ונגדיר $t=\max\{20a, c_{\text{SEL}}(140)\}$. נוכיח באינדוקציה כי $c_{\text{SEL}}(n)<tn$.

ראשית, לכל $1\le n\le 140$ מתקיים $c_{\text{SEL}}(n)\le c_{\text{SEL}}(140)$ (כי ממבנה האלגוריתם אפשר לראות שהסיבוכיות היא מונוטונית עולה, דהיינו הגדלה של הקלט יכולה רק להביא להגדלת הסיבוכיות). לכן $c_{\text{SEL}}(n)\le c_{\text{SEL}}(140)\le t\le tn$ כנדרש.

נניח כעת באינדוקציה שלמה ש־$c_{\text{SEL}}(k)\le tk$ לכל $k<n$ ונשתמש בנוסחה הרקורסיבית שראינו קודם:

$$\begin{aligned}c_{\text{SEL}}(n) &\le c_{\text{SEL}}\left(\left\lceil\frac{n}{5}\right\rceil\right)+c_{\text{SEL}}\left(\frac{7n}{10}+6\right)+O(n)\\ &\le t\left\lceil\frac{n}{5}\right\rceil+t\left(\frac{7n}{10}+6\right)+an\\ &\le \frac{tn}{5}+t+\frac{7tn}{10}+6t+an\\ &= \frac{9tn}{10}+7t+an\\ &= tn+\left(-\frac{tn}{10}+7t+an\right)\end{aligned}$$

דהיינו, נקבל את התוצאה המבוקשת בתנאי שמתקיים ש־$-\frac{tn}{10}+7t+an\le 0$. נחלץ את t מאי השוויון הזה ונקבל:

$$\begin{aligned}t\left(7-\frac{n}{10}\right)+an &\le 0\\ t\left(\frac{n-70}{10}\right) &\ge an\\ t &\ge 10a\left(\frac{n}{n-70}\right)\end{aligned}$$

כעת תתברר הבחירה שלנו בערכו של t. כזכור, בחרנו את t כך שאפשר להניח ש־$n \geq 140$, מה שמוביל לכך שהביטוי $\frac{n}{n-70}$ יקיים $\frac{n}{n-70} \leq 2$ (כי זו פונקציה יורדת ששווה ל־2 ב־$n = 140$). מכאן ש־$10a\left(\frac{n}{n-70}\right) \leq 20a \leq t$ מהבחירה שלנו את t להיות לפחות $20a$. זה מסיים את ההוכחה. ■

3 אלגוריתמי חיפוש בגרף ושימושיהם

3.1 ייצוג גרפים

אנחנו מגדירים גרף בתור זוג $G = (V, E)$ כך ש־V היא קבוצה של **צמתים** ו־E היא קבוצה של **קשתות** כאשר כל קשת מאופיינת בכך שיש לה שתי **נקודות קצה** שהן צמתים ב־V. אם G הוא גרף **לא מכוון** אז נקודות הקצה מהוות קבוצה בת שני איברים, בזמן שאם G הוא גרף **מכוון** אז נקודות הקצה הן זוג סדור (צומת היציאה של הקשת וצומת הכניסה של הקשת).

בהמשך נעסוק בגרפים שמגיעים עם מידע נוסף ־ למשל, משקל מספרי על הקשתות. אפשר לייצג מידע נוסף זה בתור פונקציות על קבוצת הצמתים או הקשתות.

מעבר לייצוג המתמטי, שאלת הייצוג של גרפים במחשב אינה טריוויאלית ויש שיטות ייצוג רבות ושונות. נבחר לצורך הקורס הזה את שיטת הייצוג הנוחה עבורנו להצגת האלגוריתמים שלנו ולא נעסוק בשיטות האחרות או בהשוואה בין היעילות שלהן. שיטת הייצוג תתבטא רק בקטעי הקוד שנכתוב.

אנו נניח שהגרף G נתון על ידי **אובייקט** שהשדות שלו הם קבוצת הצמתים V וקבוצת הקשתות E

3.2 אלגוריתם חיפוש לרוחב (BFS)

3.2.1 האלגוריתם הבסיסי

המטרה בחיפוש לרוחב היא לעבור על כל צמתי הגרף הישיגים מצומת התחלתי כלשהו, כך שהסדר שבו מגיעים לצמתים תלוי במרחק שלהם מהצומת ההתחלתי: ראשית עוברים על כל הצמתים שמרחקם ממנו 1, אחר כך על הצמתים שמרחקם ממנו הוא 2 וכדומה, כך שאפשר לחשוב על האלגוריתם בתור מעין "גל" שמתחיל להתפשט מהצומת ההתחלתי.

בפני עצמו BFS מאפשר את גילוי כל הצמתים הישיגים מהצומת ההתחלתי, אך הרחבות שונות שלו מאפשרות לחשב דברים נוספים, ונראה דוגמאות לכך.

אופן הפעולה של BFS הוא כדלהלן: בכל רגע נתון האלגוריתם מתחזק שתי רשימות צמתים ־ הרשימה Q שהיא תור של הצמתים שעדיין צריכים להיסרק, ורשימה S של הצמתים שכבר נסרקו. בתחילת פעולתו האלגוריתם מקבל צומת $s \in V$ ומכניס אותו אל Q, ומכאן ואילך פועל כך כל עוד Q איננו ריק: הוא מוציא את האיבר שבראש הרשימה Q, עובר על שכניו של הצומת הזה ולכל שכן שאינו ב־S או Q ־ מכניס אותו לסוף הרשימה Q.

בשפת Python האלגוריתם נראה כך:

```
1  def BFS(G, s):
2      Q = [s]
3      S = []
4      while len(Q) > 0:
5          u = Q.pop(0)
6          for v in G.adjacency(u):
7              if v not in Q and v not in S:
8                  Q.append(v)
9          S.append(u)
10     return S
```

האלגוריתם עשוי לעבור על כל הצמתים בגרף ולכן הלולאה החיצונית עשויה להתבצע $O(|V|)$ פעמים. לכל צומת האלגוריתם עובר על כל שכניו, כלומר על כל הקשתות שמחוברות אליו, ולכן בסך הכל הוא עשוי לעבור על $O(|E|)$ קשתות. על פניו הדבר גורר סיבוכיות של $O(|V| + |E|)$ ובמימוש יעיל של האלגוריתם זה גם המצב; לרוע המזל המימוש שהצגנו אינו יעיל כי בשורה 7 מתבצע חיפוש בקבוצות Q, S שתלוי גם בגודל הנוכחי שלהן.

הפתרון הפשוט ביותר לבעיית יעילות זו היא לשמור את הצומת בתור מבנה נתונים מורכב שמסוגל להכיל מידע נוסף מעבר לשם הצומת ־ למשל, "צבע" שמסמן אם הצומת טרם הוכנס ל־Q (ואז הוא לבן), אם הוא כבר הוכנס ל־Q אבל טרם הוצא ממנו (ואז הוא אפור) ואם הוא כבר הוצא מ־Q (ואז הוא שחור). נראה גישה זו בהמשך.

3.2.2 מציאת מרחקים ועץ BFS

נשתמש כעת ב־BFS כדי לקבל שני סוגי מידע על הצמתים שישיגים מ־s: את **המרחק** שלהם מ־s ואת המסלול הקצר ביותר מהם אל s. פורמלית, נגדיר שתי פונקציות, $d:V\to\mathbb{N}\cup\{\infty\}$ כך שלכל צומת $v\in V$, $d\left(v\right)$ הוא מרחק v מ־s או ∞ אם v לא ישיג מ־s. בנוסף, נגדיר פונקציה $\pi:V\to V\cup\{\text{None}\}$ כך שלכל צומת $v\in V$, $\pi\left(v\right)$ הוא הצומת שבא **לפניו** במסלול מ־s אל v, או $\emptyset$ אם לא קיים צומת כזה. באופן הזה, לכל צומת $v\in V$ המסלול הקצר ביותר אליו מ־s הוא המסלול הבא:

$$s=\pi^{d(v)}\left(v\right)\to\pi^{d(v)-1}\left(v\right)\to\ldots\to\pi^{2}\left(v\right)\to\pi\left(v\right)\to v$$

בשפת Python האלגוריתם נראה כך:

```
1   def BFS_shortest_paths(G, s):
2       Q = [s]
3       S = []
4       for v in G.V:
5           v.color = 'white'
6           v.d = math.inf
7           v.pi = None
8       s.color = 'gray'
9       s.d = 0
10      while len(Q) > 0:
11          u = Q.pop(0)
12          for v in G.adjacency(u):
13              if v.color == 'white':
14                  Q.append(v)
15                  v.color = 'gray'
16                  v.d = u.d + 1
17                  v.pi = u
18          u.color = 'black'
19          S.append(u)
20      return S
```

ההבדל בין האלגוריתם הזה לקודם הוא בניהול שדות המידע בתוך האובייקט שמייצג את הצמתים עצמם.

- בשורות 4־7 מאותחל המידע של כל הצמתים לערך ברירת המחדל (צבע לבן, מרחק אינסופי, צומת קודם ריק)
- בשורות 8־9 ישנו אתחול מיוחד של הצומת שממנו מתחיל ה־BFS (המרחק מעצמו הוא 0, הצבע שלו הוא אפור כי הוא כבר בתוך התור Q)
- בזמן שסורקים את הגרף מהצומת u (שורה 11) עוברים על הצמתים שמחוברים אליו (שורה 12) ולכל צומת כזה בודקים אם זו הפעם הראשונה שראינו אותו ולכן צבעו לבן (שורה 13). אם כן, מעדכנים את הצבע שלו לאפור (שורה 15) והמרחק שלו מ־s מוגדר להיות מרחק v מ־s ועוד 1 (שורה 16). הצומת הקודם במסלול, $\pi\left(v\right)$, מוגדר להיות u (שורה 17).
- כאשר מסיימים לעבור על הצומת u משונה צבעו לשחור (שורה 18).

על פניו אין טעם בהפרדה בין הצבעים שחור ואפור ואפשר היה להסתפק במשתנה בוליאני שאומר אם כבר פגשנו את הצומת או לא, אלא שבהמשך נראה שההפרדה הזו בין אפור ושחור מקילה על ההוכחה המרכזית שלנו.

נוכיח את נכונות האלגוריתם. לשם כך נפתח עם הגדרה טבעית:

הגדרה 3.1 יהא $G=(V,E)$ גרף ויהיו $s,v\in V$. **המרחק** מ־s אל v שמסומן ב־$\delta\left(s,v\right)$ הוא מספר הקשתות המינימלי במסלול כלשהו מ־s אל v. אם לא קיים מסלול מ־s אל v אז $\delta\left(s,v\right)=\infty$. **מסלול קצר ביותר** מ־$s$ אל v הוא מסלול מאורך $\delta\left(s,v\right)$.

מטרתנו היא להראות שבסיום ריצת אלגוריתם ה־BFS_shortest_paths(G,s), שדה ה־d של כל צומת $v\in V$ מכיל את $\delta\left(s,v\right)$ ושהמסלול שמוגדר באמצעות ה־π־ים הוא מסלול קצר ביותר מ־s אל v.

ראשית נוכיח ששדה ה־d הוא **חסם מלמעלה** על δ:

טענה 3.2 בסיום ריצת האלגוריתם על s, לכל $v\in V$ מתקיים ש־$\delta\left(s,v\right)\le v.d$.

הוכחה: אם $v.d=\infty$ זה ברור, אחרת v הוכנס מתישהו אל Q במהלך ריצת האלגוריתם. נוכיח את הטענה באינדוקציה על ההכנסות ל־Q. הבסיס הוא עבור s שהוכנס ראשון (בשורה 2), ועבורו נקבע $s.d=0$ (בשורה 9) ולא משתנה עוד, ולכן מתקיים $\delta\left(s,s\right)=0\le s.d$.

נעבור אל צעד האינדוקציה. אם $v\ne s$ התווסף ל־Q זה חייב להיות בשורה 14, כחלק מסריקת השכנים של צומת u שהתווסף קודם אל Q ולכן ניתן להפעיל עליו את הנחת האינדוקציה ולקבל $\delta\left(s,u\right)\le u.d$. במקרה זה, בשורה 16 נקבע ערכו של $v.d$ להיות $v.d=u.d+1$.

כמו כן, נשים לב לכך ש־$\delta\left(s,v\right)\le\delta\left(s,u\right)+1$ מאי־שוויון המשולש על מרחקים בגרף (המסלול מ־s אל v שעובר דרך מסלול קצר ביותר מ־s אל u ואז בקשת מ־u אל v הוא מסלול אפשרי אחד מ־s אל v, לאו דווקא הקצר ביותר, ואורכו $\delta\left(s,u\right)+1$). נחבר את שלושת פרטי המידע הללו ונקבל:

$$\begin{aligned}\delta\left(s,v\right)&\le\delta\left(s,u\right)+1\\&\le u.d+1=v.d\end{aligned}$$

■

אנו רוצים כעת להוכיח את ההפך, $v.d\le\delta\left(s,v\right)$. לצורך כך נזדקק לטענת עזר אחת:

טענה 3.3 אם u הוכנס ל־Q לפני v, אז $u.d\le v.d$.

הוכחה: תהא $v_0,v_1,\ldots,v_t$ סדרת כל הצמתים שהוכנסו ל־Q במהלך האלגוריתם על פי סדר ההכנסה שלהם. מספיק להוכיח ש־$v_i.d\le v_{i+1}.d$ לכל $0\le i<t$ כדי להסיק את המסקנה המבוקשת. נוכיח זאת באינדוקציה שלמה על i.

אם $v_i=s$ אז על פי שורה 9, $v_i.d=0\le v_{i+1}.d$ שכן שדה ה־d של **כל** צומת הוא מספר טבעי. לכן נניח מעתה ש־$v_i\ne s$ (ולכן גם $v_{i+1}\ne s$). כלומר, v_i הוכנס ל־Q במהלך סריקה של צומת אחר, שנסמן u_i, ו־v_{i+1} הוכנס ל־Q במהלך סריקה של צומת שנסמן u_{i+1}.

ייתכן ש־ $u_i=u_{i+1}$ ואז $u_i.d=u_{i+1}.d$, ואם לא, אז בהכרח u_i הוכנס ל־Q לפני u_{i+1} אחרת v_{i+1} היה מתווסף ל־Q לפני v_i ואז ניתן להשתמש בהנחת האינדוקציה ולקבל $u_i.d\le u_{i+1}.d$.

כעת, על פי שורה 16 נקבל $v_i.d=u_i.d+1$ וגם $v_{i+1}.d=u_{i+1}.d+1$. לכן קיבלנו:

$$\begin{aligned}v_i.d&=u_i.d+1\\&\le u_{i+1}.d+1\\&=v_{i+1}.d\end{aligned}$$

כנדרש. ■

כעת ניתן להוכיח את החצי השני של הטענה על המרחקים:

טענה 3.4 בסיום ריצת האלגוריתם על s, לכל $v\in V$ מתקיים ש־$v.d\le\delta\left(s,v\right)$.

הוכחה: אם $\delta(s,v)=\infty$ אי השוויון מתקיים תמיד, לכן מספיק להוכיח את הטענה עבור v־ים שעבורם $\delta(s,v)$ הוא מספר טבעי. נוכיח זאת באינדוקציה שלמה על הגודל של $\delta(s,v)$.

עבור $\delta(s,v)=0$ בהכרח $v=s$ ואכן $s.d=0$ (שורה 9 באלגוריתם).

יהא כעת $v\neq s$ ונתבונן במסלול מ־s אל v מאורך $k=\delta(s,v)$. נסמן ב־u את הצומת שלפני v במסלול (קיימת כזו כי אורך המסלול גדול מאפס). כלומר, המסלול הוא $s\to\ldots\to u\to v$ ויש עליו $\delta(s,v)$ קשתות. המסלול הזה בפרט כולל בתוכו מסלול מאורך $k-1$ מ־s אל u כך ש־$\delta(s,u)<\delta(s,v)$, ולכן ניתן להשתמש על u בהנחת האינדוקציה ולהסיק ש־$u.d\le\delta(s,u)$.

אם נוכל להוכיח ש־$v.d\le u.d+1$ סיימנו, כי אז יתקיים:

$$\begin{aligned}v.d&\le u.d+1\\&\le\delta(s,u)+1\\&\le\delta(s,v)\end{aligned}$$

מכיוון שהקשת $u\to v$ קיימת בגרף, ועל פי הנחת האינדוקציה ערכו של u נקבע בשלב מסויים (ולא נשאר ∞) פירוש הדבר הוא ש־u התווסף אל Q, ולאחר מכן הוצא ממנה. כחלק מתהליך ההוצאה של u מ־Q האלגוריתם עבר על שכניו, ובפרט עבר על v. צבע v בזמן הזה יכול להיות או לבן, או אפור או שחור, ונבדוק מה נובע מכל אחת מהאפשרויות הללו.

- אם v היה לבן בזמן הסריקה (שורה 13) אז בשורה 16 תבוצע ההשמה $v.d=u.d+1$ וסיימנו.
- אם v היה שחור בזמן הסריקה, המשמעות היא ש־v הוצא מ־Q לפני ש־u הוצא מ־Q. לכן בהכרח v נכנס אל Q לפני u (כי Q הוא תור שפועל בשיטת First-in-first-out) ומטענה 3.3 נקבל ש־$v.d\le u.d$.
- אם v היה אפור בזמן הסריקה, אז הוא הפך לאפור (בשורה 15) בזמן סריקה שביצע צומת **אחר**, w, שכבר הוצא מ־Q לפני u, כלומר גם הוכנס ל־Q לפני u ולכן מטענה 3.3 עולה ש־$w.d\le u.d$. אחרי שצבעו של v הפך לאפור גם נקבע $v.d=w.d+1$ (בשורה 16). משני אלו נסיק ש־$v.d=w.d+1\le u.d+1$, כפי שרצינו.

■

ראינו כי $v.d\le\delta(s,v)$ ו־$\delta(s,v)\le v.d$ ולכן נוכל להסיק ש־$v.d=\delta(s,v)$.

נותר רק להשתכנע בכך ש־π אכן מגדירה את המסלול הקצר ביותר מ־s אל צמתי הגרף.

משפט 3.5 אם v אינו ישיג מ־s אז $v.\pi=\mathrm{None}$. אחרת, המסלול הבא הוא מסלול קצר ביותר מ־s אל v:

$$s=\pi^{d(v)}(v)\to\pi^{d(v)-1}(v)\to\ldots\to\pi^2(v)\to\pi(v)\to v$$

הוכחה: ראשית, אם v אינו ישיג מ־s אז $d(v)=\infty$ כפי שראינו קודם. כלומר, מרגע האתחול של v בשורות 5־7 לא השתנה בו דבר, ולכן בפרט גם $v.\pi=\mathrm{None}$.

נניח כעת כי v ישיג מ־s, כלומר $d(v)$ הוא מספר טבעי. ההוכחה היא באינדוקציה על $d(v)$. עבור הבסיס, $d(v)=0$ גורר ש־$v=s$ ואכן $s=\pi^0(s)$ כמבוקש.

יהא כעת $s\neq v$ כלשהו ויהא $u=v.\pi$. ערך זה נקבע בשורה 17 של האלגוריתם ונובע מכך שבשורה 16 נקבע $v.d=u.d+1$, וכמו כן קיימת קשת מ־u אל v. כעת, מכך ש־$d(u)=d(v)-1$ עולה שניתן להשתמש בהנחת האינדוקציה על u ולקבל מסלול

$$s=\pi^{d(u)}(u)\to\ldots\to\pi(u)\to u$$

נציב $u=\pi(v)$ ו־$d(u)=d(v)-1$ ונקבל:

$$s=\pi^{d(v)-1}(\pi(v))\to\ldots\to\pi(\pi(v))\to\pi(v)$$

נרכיב את ההפעלות של π, נוסיף לסוף המסלול את v בהתבסס על הקשת $u \to v$ שראינו את קיומה, ונקבל את המבוקש:

$$s = \pi^{d(v)}(v) \to \ldots \to \pi^2(v) \to \pi(v) \to v$$

■

3.3 אלגוריתם המסלולים הקלים ביותר של דייקסטרה

3.3.1 בעיית המסלול הקלים ביותר

ראינו כיצד אלגוריתם BFS מאפשר לנו למצוא את **המרחק** בין צומת s נתון בגרף ובין כל צומת אחר שישיג ממנו. המרחק הזה נמדד באמצעות **מספר הקשתות** במסלול מ־s לצמתים אחרים, אבל זהו מקרה פרטי של בעיה כללית יותר ־ הבעיה של מציאת **מסלולים קלים ביותר** בגרף.

דוגמא פשוטה לאופן שבו בעיה זו מתעוררת נתונה על ידי **רשת כבישים** שמחברת ערים (או נקודות כלשהן במפה). נניח שאנו רוצים למצוא את המסלול שבו נגיע במהירות הגדולה ביותר מהעיר A אל העיר B. ייתכן למשל שיש כביש ישיר שמחבר את A אל B אבל הוא ארוך ומתפתל והנסיעה בו איטית, ולעומת זאת קיימים כבישים ישירים מהירים מ־A אל עיר אחרת C ומ־C אל B. במקרה זה עדיף להתבסס על המסלול שכולל שתי קשתות במקום אחת.

כדי למדל את הסיטואציה הזו בפורמליזם של תורת הגרפים אנו מגדירים **גרף ממושקל**:

הגדרה 3.6 גרף מכוון ממושקל הוא שלשה $G = (V, E, w)$ כך ש־(V, E) הוא גרף מכוון ו־$w : E \to \mathbb{R}^{\geq 0}$ היא **פונקציית משקל** שמתאימה לכל איבר של E משקל שהוא מספר ממשי אי־שלילי.

כעת קל להגדיר משקל של מסלול:

הגדרה 3.7 משקל של מסלול: יהא $p : v_0 \to v_1 \to \ldots \to v_n$ מסלול בגרף ממושקל . אז משקל המסלול הוא $w(p) = \sum_{i=0}^{n-1} w((v_i, v_{i+1}))$.

לבסוף, אפשר להגדיר מהו המסלול הקל ביותר בין שני צמתים:

הגדרה 3.8 יהיו $u, v \in V$ בגרף מכוון ממושקל. אם v אינו ישיג מ־u נגדיר $\delta(u, v) = \infty$ ואחרת נגדיר את $\delta(u, v) \triangleq \min\left\{w(p) \mid u \overset{p}{\leadsto} v\right\}$.

אחת הסיבות שבגללן דרשנו שהמשקלים על הקשתות יהיו **אי־שליליים** היא שאם יש קשתות בעלות משקל שלילי, ייתכן ש־$\delta(u, v)$ יהיה לא מוגדר גם אם v ישיג מ־u; זה יקרה אם קיים מסלול מ־u אל v שבדרך אליו עוברים על **מעגל** שסכום המשקלים של קשתותיו **שלילי** (ולכן ככל שחוזרים על המעגל יותר פעמים, משקל המסלול יהיה קטן יותר). בנוסף, האלגוריתם של דייקסטרה שאותו נתאר כאן לא עובד כלל על גרף שיש בו קשתות שליליות (גם אם אין בו מעגל שלילי). אלגוריתם **בלמן־פורד** מסוגל להתמודד עם קשתות שליליות (בתנאי שאין מעגל שלילי) במחיר זמן ריצה גדול יותר, אבל לא נציג אותו כאן.

הבעיה שלנו כעת פשוטה לניסוח:

הגדרה 3.9 בעיית המסלולים הקלים ממקור יחיד: בהינתן G גרף מכוון ממושקל ו־$s \in V$, למצוא עבור כל $v \in V$ את $\delta(s, v)$ ומסלול $s \overset{p}{\leadsto} v$ שזהו משקלו.

על פניו זו בעיה שאפתנית יותר מאשר הבעיה של מציאת מסלול קל ביותר בין שני צמתים: במקום זוג צמתים, אנחנו מבקשים לדעת משהו על צומת אחד **וכל יתר הגרף**. בפועל, לא ידוע אלגוריתם שפועל בסיבוכיות אסימפטוטית יעילה יותר מאשר האלגוריתמים שפותרים את הבעיה הכללית יותר ולכן אנו מתמקדים בה.

3.3.2 תיאור האלגוריתם של דייקסטרה

האלגוריתם של דייקסטרה דומה מאוד בבסיסו לאלגוריתם BFS, עם שני הבדלים חשובים:

התור: באלגוריתם BFS הצמתים שמתגלים במהלך האלגוריתם מוכנסים אל **תור** Q שפועל בשיטת FIFO ־ הראשון שנכנס הוא שיוצא. האינטואיציה מאחורי גישה זו היא שככל שצומת מוכנס מוקדם יותר כך הוא קרוב יותר אל s ולכן בכל איטרציה נשלף מ־Q צומת שהוא מינימלי מבחינת המרחק שלו מ־s.

בסיטואציה של מסלולים קלים ביותר ההנחה הזו כבר לא בהכרח נכונה ־ כפי שראינו, מספר הקשתות על מסלול אינו אינדיקציה מלאה למשקל שלו. זה מאלץ את התור Q להיות ממומש בצורה חכמה יותר שמאפשרת בכל איטרציה את שליפת הצומת בעל המרחק **המינימלי** מ־s. תור כזה נקרא **תור עדיפויות**. במסגרת הזו לא נוכל להציג את המימוש היעיל ביותר לתור העדיפויות שאלגוריתם דייקסטרה נזקק לו (מימוש שמכונה **ערימת פיבונאצ'י**) ונסתפק בהסבר כללי על מימוש באמצעות ערימה.

השכנים: באלגוריתם BFS בכל איטרציה עוברים על כל השכנים של הצומת u. שכנים v שהם צמתים שהאלגוריתם טרם נתקל בהם ("לבנים") מסומנים כבעלי מרחק $d(v) = d(u) + 1$ מהצומת s, ושכנים שהאלגוריתם כבר נתקל בהם קודם ("אפורים" או "שחורים") לא זוכים לטיפול כלל. באלגוריתם דייקסטרה לא ניתן לנקוט באותה גישת התעלמות מצמתים אפורים, כי ייתכן שהמסלול מ־s אל u ועוד הקשת (שטרם נבדקה עד כה) מ־u אל v גם יחד מהווים מסלול קל יותר מהמסלול הקל ביותר שידוע כרגע מ־s אל v. לכן יש לבצע השוואה שבודקת אם המסלול הזה אכן קל יותר ואם כן ־ לתקן בהתאם.

נציג מימוש בשפת Python של האלגוריתם שבו התור Q ממומש בצורה נאיבית על ידי רשימה. את הוצאת האיבר המינימלי מהרשימה נממש על ידי פונקציה שמשתמשת ב"שורת קסם" בודדת בפייתון (לא קריטי להבין מה קורה בה):

```python
def pop_min(Q):
    return Q.pop(min(range(len(Q)), key = lambda i: Q[i].d))
```

נעבור לאלגוריתם דייקסטרה עצמו:

```python
def djikstra(G, s):
    for v in G.V:
        v.d = math.inf
        v.pi = None
    s.d = 0
    S = []
    Q = [v for v in G.V]
    while len(Q) > 0:
        u = pop_min(Q)
        S.append(u)
        for v in G.adjacency(u):
            if v.d > u.d + G.w[(u,v)]:
                v.d = u.d + G.w[(u,v)]
                v.pi = u
    return S
```

אלגוריתם דייקסטרה הוא דוגמא קלאסית לאלגוריתם **חמדני**, כלומר כזה שבכל שלב שלו מבצע את השיפור שנראה הטוב ביותר באותו הרגע. במקרה של דייקסטרה, הגישה החמדנית מבטיחה הצלחה; לא כל אלגוריתם חמדני זוכה להבטחה כזו.

3.3.3 סיבוכיות אלגוריתם דייקסטרה

האלגוריתם מבצע $|V|$ חזרות על הלולאה שבשורה 8 (כי בכל צעד של האיטרציה מוצא איבר בודד מ־Q). הלולאה הנוספת שבשורה 11 מתבצעת פעם אחת לכל קשת בגרף, כלומר $|E|$ פעמים בסך הכל, ולכן המרכיב הרלוונטי הנוסף בקביעת זמן הריצה של דייקסטרה היא פעולת הוצאת המינימום שבשורה 9. במימוש הנאיבי שלנו פעולה זו דורשת זמן של $|V|$, ולכן סיבוכיות האלגוריתם היא $O\left(|V|^2+|E|\right)=O\left(|V|^2\right)$ כאשר הרכיב של $|E|$ נעלם שכן בגרף פשוט מספר הקשתות חסום בידי $|V|^2$.

מימוש בעזרת ערימה של Q ישנה את הסיבוכיות באופן הבא: ראשית, בניית הערימה בשורה 8 תדרוש זמן של $O\left(|V|\right)$ שלא משפיע על סיבוכיות האלגוריתם. שנית, פעולת הוצאת המינימום בשורה 9 תדרוש זמן של $O\left(\log|V|\right)$ בלבד, כפי שלקחה במיון־ערימה (זהו הזמן של ביצוע heapify על הערימה לאחר הוצאת האיבר המינימלי בה).

לרוע המזל, אלגוריתם דייקסטרה מניב סיבוך של שימוש בערימה שלא ראינו עד כה ־ בשורה 13 עשוי להתעדכן ערכו של איבר שנמצא **בתוך הערימה**. פירוש הדבר הוא שיש לשמור מכל צומת v מצביע אל האיבר שמייצג אותו בתוך הערימה, וכשערכו של $v.d$ מתעדכן, לתקן את הערימה בהתאם על ידי "פעפוע למעלה" של האיבר שערכו השתנה. תיקון זה עשוי לדרוש זמן של $O\left(\log|V|\right)$ גם הוא.

בסך הכל נקבל שמימוש על ידי ערימה מניב זמן ריצה של $O\left(\left(|V|+|E|\right)\log|V|\right)$, כלומר $O\left(|V|\log|V|+|E|\log|V|\right)$. זמן ריצה זה הוא שיפור משמעותי ביחס ל־$O\left(|V|^2\right)$ בתנאי שהגרף דליל, כלומר מספר הקשתות בו קטן דיו ־ $|E|=o\left(|V|^2/\log|V|\right)$.

כפי שהוזכר קודם, שימוש במבנה נתונים שנקרא **ערימת פיבונאצ'י** למימוש Q יכול לשפר עוד יותר את זמן הריצה. במבנה נתונים זה **הסיבוכיות המשוערכת** של פעולת הקטנת המפתח היא רק $O\left(1\right)$ (אך לא נוכל להסביר כאן מהי סיבוכיות משוערכת...) וכתוצאה מכך סיבוכיות האלגוריתם היא $O\left(|V|\log|V|+|E|\right)$.

3.3.4 נכונות אלגוריתם דייקסטרה

הטענה שאנו מבקשים להוכיח כעת היא:

משפט 3.10 בסיום ריצת אלגוריתם דייקסטרה על הקלט G,s מתקיים לכל צומת $v.d=\delta\left(s,v\right)$.

הוכחה: נוכיח באינדוקציה טענה חזקה יותר: שבכל שלב של ריצת האלגוריתם, לכל $v\in S$ מתקיים $v.d=\delta\left(s,v\right)$ ולכל $v\in Q$ מתקיים $v.d\ge\delta\left(s,v\right)$. מכיוון שבשורה 7 מאותחל Q לכלול את כל צמתי V, ובכל פעם שצומת מוצא (שורה 9) מוסיפים אותו ל־S (שורה 10) והאלגוריתם עוצר כאשר התור Q ריק (שורה 8) נסיק שבסיום ריצת האלגוריתם $S=V$ ולכן הטענה על S גוררת את מה שצריך להוכיח.

בסיס האינדוקציה קל: בתחילת ריצת האלגוריתם $S=\emptyset$ (שורה 6) ולכן הטענה נכונה עבור S באופן ריק. כמו כן, לכל $v\in Q$ השונה מ־s מתקיים $v.d=\infty\ge\delta\left(s,v\right)$ ועבור s מתקיים $s.d=0=\delta\left(s,s\right)$.

ראשית נראה שבכל שינוי של $v.d$ עבור $v\in Q$, עדיין מתקיים $v.d\ge\delta\left(s,v\right)$. המקום היחיד שבו $v.d$ משתנה לאחר שלב האתחול הוא שורה 13, שבה הוא משתנה ל־$v.d=u.d+w\left(u,v\right)$ כאשר u הוא צומת ששייך ל־S (שכן הוא התווסף בשורה 10) ומהנחת האינדוקציה, $u.d=\delta\left(s,u\right)$. מכאן ש־$u.d+w\left(u,v\right)$ הוא משקל המסלול $s\rightsquigarrow u\to v$ שבו עוברים מ־s אל u במסלול קל ביותר, ואז עוברים אל v דרך הקשת $u\to v$. מהגדרת $\delta\left(s,v\right)$ בתור מינימום נובע ש־$\delta\left(s,v\right)\le u.d+w\left(u,v\right)$, כמבוקש.

נותר להראות שבכל פעם שבה מוסיפים צומת u ל־S (שורה 10), אותו צומת u מקיים $u.d=\delta\left(s,u\right)$. יש שני מקרים טריוויאליים: זה שבו $u=s$ וזה שבו u לא ישיג מ־s ולכן $\delta\left(s,u\right)=\infty$. בתחילת ריצת האלגוריתם עבור צמתים אלו מתקיים $u.d=\delta\left(s,d\right)$ וכפי שכבר ראינו, לא יכול עבור אף צומת להיווצר מצב שבו $v.d<\delta\left(s,v\right)$ ולכן גם בסיום ריצת האלגוריתם צמתים אלו מקיימים $u.d=\delta\left(s,d\right)$.

נותר לטפל בצומת u שהוא ישיג מ־s אך שונה מ־s. נתבונן במצב האלגוריתם בתחילת שורה 10, לפני הכנסת u אל S: ניקח מסלול קל ביותר מ־s אל u, $s\rightsquigarrow u$. מכיוון ש־$s\in S$ ואילו $u\notin S$ בשלב זה, הרי שקיים בתוך המסלול צומת y שהוא הצומת הראשון כך ש־$y\notin S$. נסמן את הצומת שלפניו במסלול ב־x. כלומר המסלול הקל ביותר הוא מהצורה $s\rightsquigarrow x\to y\rightsquigarrow u$.

מכיוון ש־y הוא הצומת הראשון במסלול שאינו שייך ל־S נקבל ש־$x\in S$ והוא הוכנס לשם לפני u ולכן הנחת האינדוקציה תקפה עבורו: $x.d=\delta\left(s,x\right)$.

כעת, מכיוון ש־$s\rightsquigarrow x\to y\rightsquigarrow u$ הוא מסלול קל ביותר מ־s אל u הרי ש־$s\rightsquigarrow x\to y$ הוא מסלול קל ביותר מs אל y (אחרת היינו יכולים להחליף אותו במסלול קל יותר ולהקל על המסלול מs אל u). מכאן ש־$\delta\left(s,y\right)=\delta\left(s,x\right)+w\left(x,y\right)$.

כעת, כאשר x הוכנס אל S בשורה 10 מיד לאחר מכן האלגוריתם עבר סדרתית על שכניו כולל y. הבדיקה בשורה 12 האם מתקיים $y.d \geq x.d + w\left(x,y\right)$ בהכרח מצליחה, שכן

$$\begin{aligned} x.d + w\left(x,y\right) &= \delta\left(s,x\right) + w\left(x,y\right) \\ &= \delta\left(s,y\right) \leq y.d \end{aligned}$$

ולכן בשורה 13 מעודכן $y.d$ כך ש־$y.d = \delta\left(s,x\right) + w\left(x,y\right) = \delta\left(s,y\right)$.

כעת נתבונן פעם אחרונה במסלול $s \rightsquigarrow x \to y \rightsquigarrow u$. מכיוון ש־$u$ נמצא בהמשך המסלול אחרי y **ואין קשתות שליליות בגרף** אז

$$y.d = \delta\left(s,y\right) \leq \delta\left(s,u\right) \leq u.d$$

כעת נשתמש בכך שבשורה 9 נשלף מ־Q איבר בעל ערך d **מינימלי**. מכיוון שבחרנו את y כך ש־$y \notin S$ הרי ש־$y \in Q$, ולכן המינימליות של u גוררת ש־$u.d \leq y.d$. אם נוסיף את זה לשרשרת אי השוויונים שקיבלנו קודם, נקבל שהאיבר הראשון והאחרון בה זהים:

$$y.d = \delta\left(s,y\right) \leq \delta\left(s,u\right) \leq u.d \leq y.d$$

כלומר, כל השרשרת היא למעשה של שוויונים. מכאן נסיק שברגע ההכנסה שלו אל S, $u.d = \delta\left(s,u\right)$, כמבוקש. ■

כמו ב־BFS, גם כאן פשוט להוכיח ש־π מקודד את המסלולים הקלים ביותר:

משפט 3.11 אם v אינו ישיג מ־s אז $v.\pi = \mathrm{None}$. אחרת, המסלול הבא הוא מסלול קל ביותר מ־s אל v:

$$s = \pi^{d(v)}\left(v\right) \to \pi^{d(v)-1}\left(v\right) \to \ldots \to \pi^{2}\left(v\right) \to \pi\left(v\right) \to v$$

הוכחה: ההוכחה דומה מאוד להוכחה של משפט 3.5 עבור אלגוריתם BFS ולא נציג אותה כאן. ■

3.4 אלגוריתם המסלולים הקלים ביותר של פלויד־וורשאל

3.4.1 מבוא לאלגוריתם פלויד־וורשאל

אלגוריתם דייקסטרה פתר את בעיית המסלולים הקלים ביותר **ממקור יחיד**. כעת אנו רוצים לעסוק בבעיית המסלולים הקלים ביותר בין כל זוג צמתים בגרף:

הגדרה 3.12 בעיית המסלולים הקלים בין כל הזוגות: בהינתן G גרף מכוון ממושקל, למצוא עבור כל $u,v \in V$ את $\delta\left(u,v\right)$ ומסלול $u \overset{p}{\rightsquigarrow} v$ שזהו משקלו.

דרך פתרון אפשרית אחת היא להריץ את אלגוריתם דייקסטרה $\left|V\right|$ פעמים, פעם אחת לכל צומת בגרף. זה פתרון שמניב זמן ריצה של $O\left(\left|V\right|^{2}\log\left|V\right| + \left|V\right|\left|E\right|\right)$ עבור המימוש הטוב ביותר של דייקסטרה.

גישה אחרת היא להשתמש באלגוריתם פלויד־וורשאל, שסיבוכיות זמן הריצה שלו היא $O\left(\left|V\right|^{3}\right)$ - לכאורה פחות טובה מזו של דייקסטה, אך יש מספר סיבות להציג אותו:

- פלויד־וורשאל יודע להתמודד עם הסיטואציה של קיום קשתות שליליות בגרף (אך לא קיום מעגל שסכום המשקלים עליו שלילי). במצב כזה לא ניתן להשתמש בדייקסטרה, ואלגוריתם חלופי כמו בלמן־פורד דורש סיבוכיות זמן ריצה גבוהה יותר.
- זמן הריצה בפועל של פלויד־וורשאל עשוי להיות טוב משל השימוש הזה בדייקסטרה עקב האופי השונה של האלגוריתמים.
- הרעיון שעומד מאחורי פלויד־וורשאל מעניין בפני עצמו (ובמובן מסויים, פשוט מזה של דייקסטרה) ומצדיק היכרות עמו.

3.4.2 תיאור אלגוריתם פלויד־וורשאל

לצורך האלגוריתם נמספר את צמתי V מ־1 עד n: $V=\{v_1,v_2,\dots,v_n\}$. כמו כן נגדיר מטריצה $W\in M_{n\times n}\left(\mathbb{R}\cup\{\infty\}\right)$ של **משקלים** שנותנת משקל לכל זוג סדור של צמתים בגרף, באופן הבא:

$$[W]_{ij}=\begin{cases}0 & i=j\\ w\left(v_i,v_j\right) & i\neq j,\left(v_i,v_j\right)\in E\\ \infty & i\neq j,\left(v_i,v_j\right)\notin E\end{cases}$$

הרעיון מאחורי האלגוריתם הוא לתחזק מטריצה $D\in M_{n\times n}\left(\mathbb{R}\cup\{\infty\}\right)$ של **מרחקים**: מטרתנו היא להבטיח שבסוף ריצת האלגוריתם יתקיים $[D]_{ij}=\delta\left(v_i,v_j\right)$. האלגוריתם מבצע זאת על ידי סדרת שיפורים הדרגתיים שמניבה סדרה של מטריצות: $D^0,D^1,D^2,\dots,D^n$ כך שמתקיים העיקרון הכללי הבא:

$\left[D^k\right]_{ij}$ שווה למשקל המסלול הקל ביותר מ־v_i אל v_j ש**צמתי הביניים** שבהם הוא עובר הם בעלי אינדקס k לכל היותר (צמתי הביניים של מסלול הם כל הצמתים בו למעט הראשון והאחרון).

קל לראות ש־$D^0=W$. כיצד ניתן לחשב את D^k מתוך D^{k-1}? באמצעות האבחנה הבאה: אם המסלול הקל ביותר מ־v_i אל v_j שצמתי הביניים שלו הם בעלי אינדקס לכל היותר k **לא כולל** את v_k בתור צומת ביניים, אז $\left[D^k\right]_{ij}=\left[D^{k-1}\right]_{ij}$.

אם המסלול **כן כולל** את v_k אז הוא יכלול אותו רק פעם אחת, ואפשר לפצל את המסלול לשלושה חלקים:

$$v_i\rightsquigarrow^p v_k\rightsquigarrow^q v_j$$

כך שהמסלולים p,q **אינם** משתמשים בצומת v_k ולכן כל צמתי הביניים שלהם הם בעלי אינדקס $k-1$ לכל היותר. מכאן שהמסלול הקל ביותר שעושה שימוש ב־v_k נבנה מתוך שני מסלולים קלים ביותר השייכים כבר ל־D^{k-1}: מ־v_i אל v_k ומ־v_k אל v_j. כלומר מתקיים במקרה זה:

$$\left[D^k\right]_{ij}=\left[D^{k-1}\right]_{ik}+\left[D^{k-1}\right]_{kj}$$

ועל כן אם נגדיר

$$\left[D^k\right]_{ij}=\min\left\{\left[D^{k-1}\right]_{ij},\left[D^{k-1}\right]_{ik}+\left[D^{k-1}\right]_{kj}\right\}$$

כלל העדכון הזה יאפשר לנו לחשב אינדוקטיבית את D^n.

חישוב המסלול הקצר ביותר ניתן לביצוע במקביל לחישוב D (או לאחר מכן, מתוך D). גם לצורך כך נגדיר מטריצה: $[\Pi]_{ij}$ תהיה None אם אין מסלול מ־i אל j, ואם קיים מסלול כזה, אז היא $[\Pi]_{ij}=t$ כך ש־v_t הוא הקודם המיידי של v_j במסלול קל ביותר מ־v_i אל v_j, כלומר המסלול הקל ביותר הוא מהצורה $v_i\rightsquigarrow v_t\to v_j$.

כמו עם D, כך גם את Π נחשב איטרטיבית על ידי סדרה $\Pi^0,\Pi^1,\dots,\Pi^n$ שמציייתת לאותו עיקרון: $\left[\Pi^k\right]_{ij}$ הוא הקודם המיידי של v_j במסלול הקל ביותר מ־v_i אל v_j שעובר דרך צמתי ביניים עם אינדקס k לכל היותר. האתחול מתאים לסיטואציה של מסלול שעובר בצעד בודד מ־v_i אל v_j כך שאין בו צמתי ביניים:

$$\left[\Pi^0\right]_{ij}=\begin{cases}i & [W]_{ij}<\infty\\ \text{None} & [W]_{ij}=\infty\end{cases}$$

והתיקון האיטרטיבי מתבצע בהתאם לבחירה אם הועדף המסלול שלא עובר דרך v_k או שהועדף המסלול שעובר דרכו:

$$\left[\Pi^k\right]_{ij}=\begin{cases}\left[\Pi^{k-1}\right]_{ij} & \left[D^{k-1}\right]_{ij}\leq\left[D^{k-1}\right]_{ik}+\left[D^{k-1}\right]_{kj}\\ \left[\Pi^{k-1}\right]_{kj} & \left[D^{k-1}\right]_{ij}>\left[D^{k-1}\right]_{ik}+\left[D^{k-1}\right]_{kj}\end{cases}$$

נציג מימוש בשפת Python של אלגוריתם פלויד־וורשאל:

```python
def floyd_warshall(W):
    n = len(W)
    D = [[W[i][j] for j in range(n)] for i in range(n)]
    Pi = [[i if W[i][j] != math.inf else None for j in range(n)] for i in range(n)]
    for k in range(n):
        D_new = [[D[i][j] for j in range(n)] for i in range(n)]
        Pi_new = [[Pi[i][j] for j in range(n)] for i in range(n)]
        for i in range(n):
            for j in range(n):
                if D[i][j] > D[i][k] + D[k][j]:
                    D_new[i][j] = D[i][k] + D[k][j]
                    Pi_new[i][j] = Pi[k][j]
```

```
13              D = D_new
14              Pi = Pi_new
15          return D, Pi
```

בשורות 3,4 מאתחלים את D^0, Π^0. העדכון עצמו מתבצע בצורה הבאה: בשורות 6,7 מעתיקים את D^{k-1}, Π^{k-1} כמות שהם ומשנים אותם (בשורות 11,12) רק אם התקיים התנאי $\left[D^{k-1}\right]_{ij} > \left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj}$ (שנבדק בשורה 10).

המימוש שלנו מניח שאנו מקבלים את W המורחב. ניתן בפייתון לבנות אותו מתוך הגרף G באמצעות הקסם הבא, שלא קריטי להבין:

```
1 def build_W(G):
2     n = len(G.V)
3     W = [[0 if i == j
4             else w[(G.V[i],G.V[j])] if (G.V[i],G.V[j]) in G.w
5             else math.inf
6             for j in range(n)]
7             for i in range(n)]
8     return W
```

3.4.3 ניתוח סיבוכיות ונכונות של אלגוריתם פלויד־וורשאל

סיבוכיות אלגוריתם פלויד־וורשאל פשוטה מאוד: הוא מבצע קינון של שלוש לולאות (שורות 5,9,10) שכל אחת מהן רצה $|V|$ צעדים, כך שזמן הריצה שלו הוא $\Theta\left(|V|^3\right)$. נעבור להוכחת הנכונות שלו, שמפרמלת את ההסבר שהצגנו קודם.

משפט 3.13 בסיום ריצת אלגוריתם פלויד־וורשאל, $[D]_{ij} = \delta\left(v_i, v_j\right)$

הוכחה: נוכיח באינדוקציה טענה חזקה יותר: אם D^k היא המטריצה המתקבלת לאחר k הפעלות של הלולאה שבשורה 5, אז $\left[D^k\right]_{ij}$ הוא משקל המסלול הקל ביותר מ־v_i אל v_j שאינו עובר דרך צמתי ביניים בעלי אינדקס גדול מ־k, או ש־$\left[D^k\right]_{ij} = \infty$ אם מסלול כזה אינו קיים.

עבור $k = 0$, מכיוון שצמתי הגרף אונדקסו בתור $\{v_1, \ldots, v_n\}$, כל צומת הוא בעל אינדקס גדול מ־k ולכן המסלולים היחידים מ־v_i אל v_j שבאים בחשבון הם אלו מאורך 0 (רק כאשר $i = j$) או 1.

אם $i = j$ אז המסלולים מ־v_i לעצמו כוללים את המסלול הריק שמשקלו 0 ואולי מסלול מהצורה $v_i \to v_i$ אם קיימת קשת מ־v_i אל עצמו. מכיוון שהנחנו שאין בגרף מעגלים עם משקל שלילי, משקל המסלול הזה הוא אי־שלילי ולכן משקל המסלול הקל ביותר הוא 0. ואמנם $[W]_{ii} = 0$ (שורה 3 באלגוריתם build_W).

אם $i \neq j$ וקיימת הקשת $v_i \to v_j$ אז המסלול היחיד מ־v_i אל v_j שמקיים את התנאי הוא המסלול $v_i \to v_j$ שמשקלו $w\left(i, j\right)$ והוא אכן שווה ל־$[W]_{ij}$ (שורה 4 באלגוריתם build_W). אם לא קיימת הקשת $v_i \to v_j$ אז לא קיים מסלול כלל ואכן $[W]_{ij} = \infty$ (שורה 5 באלגוריתם build_W).

נעבור אל צעד האינדוקציה. יהיו i, j כלשהם. אם לא קיים מסלול מ־v_i אל v_j שצמתי הביניים בו הם כולם מאינדקס לכל היותר k אז בפרט לא קיים מסלול כזה גם עבור צמתי ביניים מאינדקס לכל היותר $k-1$ כך שעל פי הנחת האינדוקציה $\left[D^{k-1}\right]_{ij} = \infty$. בנוסף, לא ייתכן שקיים גם מ־v_i אל v_k ומ־v_k אל v_j שמקיימים את תנאי האינדקסים עבור $k-1$ כי שרשור המסלולים מניב מסלול מ־v_i אל v_j שמקיים את תנאי האינדקסים עבור k. כלומר, $\left[D^{k-1}\right]_{ik} = \infty$ או $\left[D^{k-1}\right]_{kj} = \infty$ ובכל מקרה $\left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj} = \infty$. לכן התנאי בשורה 10 אינו מתקיים, ועל פי ההשמה בשורה 6 נקבל $\left[D^k\right]_{ij} = \left[D^{k-1}\right]_{ij} = \infty$ כנדרש.

נניח כעת כי קיים מסלול כזה וניקח את המסלול הקל ביותר מ־v_i אל v_j עם אינדקסים לכל היותר k. אם v_k לא מופיע במסלול הזה אז על פי הנחת האינדוקציה $\left[D^k\right]_{k-1}$ שווה למשקל מסלול זה ובהכרח יתקיים $\left[D^k\right]_{k-1} \leq \left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj}$, אחרת המסלול הקל ביותר יתקבל משרשור שני המסלולים בעלי המשקלים $\left[D^{k-1}\right]_{ik}, \left[D^{k-1}\right]_{kj}$. לכן התנאי בשורה 10 אינו מתקיים גם במקרה זה ונקבל $\left[D^k\right]_{ij} = \left[D^{k-1}\right]_{ij}$ כנדרש.

אם לעומת זאת v_k כן מופיע במסלול הקל ביותר, נתבונן במסלול זה: $v_i \rightsquigarrow^p v_k \rightsquigarrow^q v_j$. אם v_k מופיע בו יותר מפעם אחת זה יוצר מעגל, ועל פי ההנחה שלנו אין בגרף מעגלים בעלי משקל שלילי כך שאפשר להסיר את המעגל מהמסלול ולקבל מסלול עם אותו משקל (או קל יותר), ולכן ניתן להניח שבמסלול שלקחנו v_k מופיע בדיוק פעם אחת. לכן p, q מקיימים את תנאי האינדקסים עבור $k-1$ ובהכרח משקלם שווה ל־$\left[D^{k-1}\right]_{ik}, \left[D^{k-1}\right]_{kj}$ (אחרת היה אפשר להחליף אותם במסלולים קלים יותר).

כעת, אם $\left[D^{k-1}\right]_{ij} = \left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj}$ אז התנאי בשורה 10 לא יופעל ונקבל $\left[D^{k}\right]_{ij} = \left[D^{k-1}\right]_{ij} = \left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj}$, כנדרש.

אם לעומת זאת $\left[D^{k-1}\right]_{ij} > \left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj}$ התנאי בשורה 10 כן יופעל וההשמה בשורה 11 תגרום לכך שיתקיים $\left[D^{k}\right]_{ij} = \left[D^{k-1}\right]_{ik} + \left[D^{k-1}\right]_{kj}$, כנדרש. ■

כרגיל, יש להוכיח גם כי האלגוריתם מצליח לא רק לחשב את משקל המסלולים הקלים ביותר אלא גם למצוא אותם:

משפט 3.14 לכל i נגדיר פונקציה $\pi_i(v_j) = [\Pi]_{ij}$ עבור ערכו של Π בסיום ריצת האלגוריתם. אם לא קיים מסלול מ־v_i אל v_j אז $\pi_i(j) = \text{None}$. אחרת, קיים t כך ש־$v_i = \pi^t(v_j)$ והמסלול $\pi^t(v_j) \to \pi^{t-1}(v_j) \to \cdots \to \pi^0(v_j)$ הוא ממשקל $\delta(v_i, v_j)$.

הוכחה: כמו במקרה של דייקסטרה, ההוכחה דומה באופיה להוכחות שכבר ראינו ולא נציג אותה כאן. ■

3.5 אלגוריתם חיפוש לעומק (DFS)

3.5.1 תיאור אלגוריתם DFS

הרעיון באלגוריתם חיפוש לעומק הוא לבצע סריקה של גרף החל מצומת התחלתי כדי למצוא את כל הצמתים הישיגים ממנו, אך לעשות זאת בצורה שונה מ־BFS שהיא אולי טבעית יותר בביצוע בידי בני אדם: להתחיל ללכת לאורך אחד מהמסלולים בגרף תוך סימון הצמתים שאנו עוברים בהם, עד שאנו "נתקעים" בצומת שאין ממנו יציאה לצומת שביקרנו ממנו. במצב שכזה חוזרים צעד אחד אחורה ובודקים אם קיימת שם קשת אל צומת שטרם ביקרנו בו, וכן הלאה.

ב־BFS מבנה הנתונים ששימש לניהול סדר העדיפויות בבדיקת צמתים היה **תור**: בכל שלב, הכנסנו אל התור את הצמתים שטרם ביקרנו בהם מבין שכניו של הצומת הנוכחי אותו בדקנו, והצמתים הוצאו מהתור על פי עיקרון First-in-first-out. ב־DFS לעומת זאת מבנה הנתונים שבו נשתמש הוא **מחסנית** שפועל על פי עיקרון Last-in-first-out; בכל שלב נוסיף למחסנית את כל השכנים שטרם ביקרנו בהם של הצומת שאנו בודקים באותו שלב.

ברוב שפות התכנות ניתן לממש DFS באמצעות **רקורסיה**, כלומר פונקציה שקוראת לעצמה. מאחורי הקלעים של שפת התכנות מנגנון הרקורסיה ממומש באמצעות מחסנית, שבה נשמר המידע הרלוונטי לכל קריאה של הפונקציה (בפרט, ייצוג כלשהו של הצמתים שנבדקים במהלך קריאה זו). זה הופך את המימוש של DFS בשפות כאלו לפשוט יחסית.

אנו נציג כיצד ניתן להשתמש ב־DFS כדי למצוא **מיון טופולוגי** של גרף וכדי למצוא **רכיבים קשירים היטב** של גרף. לצורך כך נממש את ה־DFS בצורה ששומרת בצמתים מידע נוסף שיהיה שימושי לצרכים הללו.

נציג מימוש של אלגוריתם DFS בשפת פייתון:

```
def DFS(G):
    for u in G.V:
        u.color = 'white'
        u.pi = None
    time = 0
    for u in G.V:
        if u.color == 'white':
            time = DFS_visit(G, u, time)
```

הקוד של DFS הזה הוא מעטפת לחלק הרקורסיבי של האלגוריתם, שיתבצע ב־DFS_visit. ב־DFS מתבצעים שני דברים:

1. אתחול המידע של האלגוריתם (שכולל בשלב זה את צבעי הצמתים, ה־π שלהם שיצביע על הצומת שממנו נכנסו אליהם בסריקה ומשתנה בשם time שיאפשר לנו לעקוב אחר הסדר שבו צמתים התגלו והטיפול בהם הסתיים)

2. מעבר סדרתי על צמתי G והפעלת האלגוריתם הרקורסיבי על כל אחד מהם שטרם טופל בהפעלות קודמות. זה שונה מ־BFS שבו הסתפקנו בהפעלת האלגוריתם על צומת בודד ולא התעניינו בצמתים שאינם ישיגים ממנו.

נציג כעת את מימוש החלק הרקורסיבי של האלגוריתם:

```
1   def DFS_visit(G, u, time):
2        time = time + 1
3        u.d = time
4        u.color = 'gray'
5        for v in G.adjacency(u):
6             if v.color == 'white':
7                  v.pi = u
8                  time = DFS_visit(G, v, time)
9        u.color = 'black'
10       time = time + 1
11       u.f = time
12       return time
```

האלגוריתם פועל כך: הוא מסמן את זמן תחילת הטיפול בצומת u בתוך $u.d$ ואת זמן סיום הטיפול בו ב־$u.f$, כשבכל פעם הזמן מוגדל ב־1 בדיוק לפני ההשמה הזו. הוא משנה את צבעו של u לאפור עם תחילת הטיפול בו ומשנה את הצבע לשחור לאחר סיום הטיפול.

הטיפול עצמו כולל מעבר סדרתי על כל הצמתים שיש קשת מ־u אליהם, ולכל צומת כזה שטרם ביקרו בו (כלומר, צבעו לבן) מסומן $v.\pi = u$ ולאחר מכן האלגוריתם ממשיך לטפל רקורסיבית ב־v (כשהוא מעדכן את משתנה הזמן time בהתאם למה שקרה בקריאה הרקורסיבית).

הבדל מהותי אחד בין BFS ובין DFS הוא בכך שב־BFS הטיפול בצומת הסתיים **לפני** שהחל הטיפול בצמתים שנתגלו במהלך הסריקה שלו; כאן אנו משהים את סימון סיום הטיפול עד **לאחר** שטופלו הצמתים הבאים בתור.

סיבוכיות זמן הריצה של DFS היא $O(|V| + |E|)$ שכן האלגוריתם עובר סדרתית על כל הצמתים והקשתות בגרף.

3.5.2 תכונות אלגוריתם DFS

על מנת להיעזר ב־DFS נרצה להבין טוב יותר מה מאפיין את אופן פעולתו.

האבחנה הראשונה היא כי כל הפעלה של DFS_visit סורקת את הגרף במבנה של **עץ** שמקודד על ידי הפונקציה π שמתאימה לכל צומת את ההורה שלו בעץ. מכיוון ש־DFS_visit עשוי להיקרא מספר פעמים מתוך DFS על צמתים שונים, תת־הגרף שמקודד על ידי π הוא **יער**, כלומר אוסף של עצים. למרות זאת, נמשיך לדבר בחופשיות על "עץ ה־DFS".

כל צומת מתווסף לעץ ה־DFS כאשר שורה 7 באלגוריתם מופעלת עבורו. באותו רגע, הצומת שאליו מחברים אותו הוא אפור (זו השפעת שורה 4) ואפשר להראות באינדוקציה שכל סדרת הצמתים שתתקבל באמצעות π על הצמתים הללו תהיה אפורה, ושאלו הצמתים האפורים היחידים בשלב זה של האלגוריתם. המסקנה היא שאם צומת מתווסף לעץ ה־DFS אז אבותיו בעץ הם כל הצמתים האפורים, והוא אינו צאצא של אף צומת שאינו אפור בעץ.

אבחנה נוספת שנשתמש בה בחופשיות בהמשך הוא שכל הזמנים שמוכנסים לצמתים ($u.d$ ו־$u.f$) הם שונים זה מזה עבור כל הצמתים בגרף. זאת מכיוון שהמשתנה time הוא "גלובלי" במובן זה שכל שינוי בו בתוך DFS_visit מוחזר החוצה ומועבר לקריאות הבאות של DFS_visit, ומכיוון שלפני כל השמה ב־$u.d, u.f$ ערכו של המשתנה מוגדל ב־1.

נתאר כעת את מה שמכונה **תכונת הסוגריים** של DFS. בשימוש "אמיתי" בסוגריים, ביטוי כמו $([)]$ אינו חוקי כי הסוגריים העגולים נסגרים לפני הסוגריים המרובעים; האינטואיציה היא כי מכיוון שהסוגריים המרובעים נפתחו **אחרי** שהסוגריים העגולים נפתחו אבל לפני שהסוגריים העגולים נסגרו, הם צריכים להיסגר **לפני** שהסוגריים העגולים נסגרים. אפשרות אחרת היא שהסוגריים המרובעים ייפתחו **אחרי** שהסוגריים העגולים ייסגרו. כלומר, שתי האפשרויות הבאות הן חוקיות:

- $([])$
- $()[]$

תכונת הסוגריים של DFS אומרת שדבר דומה קורה כאשר מסתכלים על זמני תחילת וסיום הטיפול בצמתים:

משפט 3.15 יהיו $u, v \in V$ ונניח בלי הגבלת הכלליות ש־$u.d < v.d$. אז בדיוק אחת משתי האפשרויות הבאות מתקיימת:

1. $u.d < v.d < v.f < u.f$ ("המקרה $([])$"). במקרה זה, v הוא צאצא של u בעץ ה־DFS.

2. $u.d < u.f < v.d < v.f$ ("המקרה $[]()$"). במקרה זה, אף צומת אינו צאצא של השני בעץ ה־DFS.

הוכחה: יש בדיוק שתי אפשרויות: או ש־$v.d < u.f$ או ש־$u.f < v.d$. כל אפשרות תניב את אחד המקרים שלנו.

מקרה א': $v.d < u.f$ (הטיפול ב־v מתחיל לפני שהטיפול ב־u מסתיים)

במקרה זה עלינו להוכיח ש־$v.f < u.f$ וש־v הוא צאצא של u בעץ ה־DFS. נשים לב לכך שערכו של $u.d$ נקבע בשורה 3 וערכו של $u.f$ נקבע בשורה 11, כך שהגעת האלגוריתם אל שורה 3 עבור v מתבצעת בין הפעלת שתי שורות אלו עבור u. כדי לחזור אל u האלגוריתם צריך לסיים את הפעלת DFS_visit עבור v, כלומר להגיע לשורה 11 של v ולקבוע את ערכו של $v.f$, ולכן $v.f < u.f$.

מכיוון ש־$v.f < u.f$ הרי שכאשר v התווסף לעץ ה־DFS, הצומת u היה אפור ולכן (מההערה שאמרנו קודם) v יהיה צאצא שלו.

מקרה ב': $u.f < v.d$ (הטיפול ב־v מתחיל אחרי שהטיפול ב־u מסתיים)

במצב זה מכיוון ש־ $v.d < v.f$ ו־$u.d < u.f$ שרשרת אי השוויונים מתקבלת מעצמה. מכיוון שבזמן הטיפול ב־v הצומת u כבר לא היה אפור, נקבל ש־v אינו צאצא של u. ■

נוכל כעת להסיק משפט שימושי נוסף:

משפט 3.16 ("משפט המסלול הלבן") הצומת v הוא צאצא של u בעץ DFS אם ורק אם כאשר האלגוריתם הופעל לראשונה על u היה מסלול בגרף ממנו אל v שכולל כולו צמתים לבנים.

הוכחה: כיוון ראשון ("צאצא" גורר "קיום מסלול לבן"): אם $u = v$ המשפט בבירור נכון כי כאשר האלגוריתם מופעל על u, הוא עדיין לבן (שורות 7 ב־DFS ו־6 ב־DFS_visit דורשות זאת במפורש). אם v הוא צאצא כלשהו של u ששונה ממנו, אז לפי המשפט הקודם, $u.d < v.d$. מכיוון שצבעו של צומת משתנה (שורה 4) רק אחרי שערכו של d שלו נקבע (שורה 3) נובע מכך שכאשר האלגוריתם מופעל על u, מכיוון ש־$v.d$ טרם נקבע גם צבעו של v טרם השתנה והוא לבן. מכיוון שהדבר נכון **לכל** הצאצאים של u, הוא נכון בפרט לאלו שנמצאים על המסלול בינו לבין כל v קונקרטי שהוא צאצא שלו בעץ ה־DFS.

כיוון שני ("קיום מסלול לבן" גורר "צאצא"): יהא u צומת כלשהו ונתבונן על מסלול לבן כלשהו בגרף בזמן הגילוי של u. נניח שלא כל צמתי המסלול הם צאצאים של u בעץ ה־DFS וניקח בתור v את הראשון שמביניהם על המסלול. בהכרח $v \neq u$ כי u הוא תמיד צאצא של עצמו. לכן קיים צומת w שהוא הקודם המיידי ל־v במסלול ו־w **כן** שייך לעץ ה־DFS של u.

ממשפט תכונת הסוגריים נובע ש־$u.d < w.d < w.f < u.f$. כעת, מכיוון ש־v הוא בן של w הוא יהיה אחד מהצמתים שעוברים עליהם בשורה 5 של DFS_visit. לכן האלגוריתם יבקר בו בשלב זה, אם לא ביקר בו עוד מוקדם יותר, ולכן יתקיים $v.f < w.f < u.f$. כמו כן, מכיוון ש־v היה לבן כאשר u התגלה, $u.d < v.d$. כלומר אנחנו בסיטואציה $u.d < v.d < v.f < u.f$ של משפט תכונת הסוגריים ולכן v הוא צאצא של u בעץ ה־DFS. ■

3.5.3 סיווג הקשתות באלגוריתם DFS

בריצתו, DFS בונה **עץ DFS** (למעשה, יער). הקשתות בעץ הזה הן קשתות בגרף המקורי G שעליו רץ האלגוריתם. נשאלת השאלה ־ מה בדבר הקשתות האחרות? מתברר שמשתלם לסווג אותן לשלושה סוגים נוספים.

הגדרה 3.17 בהינתן עץ DFS עבור גרף G נסווג את קשתות G ביחס לעץ:

1. **קשת עץ** תהיה כל קשת השייכת לעץ ה־DFS. כלומר, כל קשת (u, v) כך ש־$v.\pi = u$.

2. **קשת אחורית** תהיה קשת (u, v) כך ש־v הוא **אב קדמון** של u בעץ ה־DFS. גם קשת מ־u לעצמו תיחשב קשת אחורית.

3. **קשת קדמית** תהיה קשת (u, v) כך ש־v הוא צאצא של u בעץ ה־DFS אך הקשת אינה שייכת לעץ ה־DFS.

4. **קשת חוצה** תהיה כל קשת אחרת בגרף (כזו שמחברת שני צמתים שאף אחד מהם אינו אב קדמון של השני בעץ ה־DFS).

על מנת להיות פורמליים יש להוכיח כי כל קשת בגרף משתייכת בדיוק לאחת מהקבוצות הללו; נדלג על ההוכחה כאן.

אלגוריתם DFS יכול להיעזר בצבע הצמתים שהוא פוגש בגרף כדי לבצע סיווג חלקי של הקשתות עוד במהלך ריצתו. לצורך כך אפשר להרחיב את הבדיקה שמתבצעת בשורה 6, שבו במהלך סריקת צומת u אנו בודקים את צבע השכן שלו v:

- אם צבע v לבן אז (u,v) תהיה קשת עץ (בשורה 7 אנחנו מוסיפים אותה לעץ)
- אם צבע v אפור אז (u,v) היא קשת אחורית (כי הצבע האפור של v מצביע על כך שאנחנו כרגע במהלך סריקת תת־עץ ה־DFS שלו)
- אם צבע v שחור אז (u,v) היא או קשת קדמית (במקרה שבו כבר סרקנו את v, חזרנו קצת אחורה בעץ ה־DFS ואז נתקלנו בה שוב) או קשת חוצה (במקרה שבו v בכלל הייתה שייכת לעץ DFS אחר ביער ה־DFS שנסרק קודם).

3.5.4 מיון טופולוגי

כזכור, מיון "רגיל" של קבוצת איברים הוא מספור שלה, $(v_0,v_1,\ldots,v_{n-1})$ כך ש־$v_0<v_1<\ldots<v_{n-1}$. במיון כזה אנחנו מניחים שאפשר להשוות כל זוג איברים על פי יחס סדר לינארי, וכתוצאה מכך הרשימה הממוינת היא יחידה.

כעת נעסוק במקרה שבו התנאים מעט שונים: ייתכן שחלק מהאיברים לא יהיו ניתנים להשוואה כלל, והדרישה היחידה מהרשימה היא שאם $v_i<v_j$ אז $i<j$.

הדוגמא ה"קלאסית" למיון טופולוגי היא לבישת בגדים: אין חשיבות של ממש לשאלה אם לובשים את המכנסיים או את החולצה קודם, אך אי אפשר ללבוש את התחתונים לאחר לבישת המכנסיים, או את הגרביים לאחר לבישת הנעליים. פתרון לבעיית המיון הוא רשימת הבגדים שיש ללבוש, מהראשון עד לאחרון, כך שלבישת הבגדים לא תהיה מלווה באסונות.

הדרך הטבעית לתאר רשימת אילוצים שכזו היא באמצעות גרף מכוון: האיברים שיש למיין הם צמתי הגרף, והקשת $u\to v$ פירושה ש־u חייב לבוא לפני v במיון הטופולוגי. נגדיר זאת פורמלית:

הגדרה 3.18 מיון טופולוגי של גרף מכוון $G=(V,E)$ הוא פונקציה חח"ע $\varphi:V\to\mathbb{Z}$ כך שאם קיימת קשת $u\to v$ אז $\varphi(u)<\varphi(v)$.

אם בגרף מכוון קיימים מעגלים, בבירור לא יכול להיות לו מיון טופולוגי, שכן אם φ היא מיון טופולוגי של גרף שכזה ו־$u_1\to u_2\to\cdots\to u_k\to u_1$ הוא מעגל, אז על ידי הפעלת φ על הצמתים נקבל $\varphi(u_1)<\varphi(u_2)<\ldots<\varphi(u_k)<\varphi(u_1)$ ובפרט נקבל $\varphi(u_1)\ne\varphi(u_1)$ ־ סתירה.

לעומת זאת, לגרף מכוון חסר מעגלים (מה שמכונה באנגלית DAG: Directed Acyclic Graph) קיים מיון טופולוגי וניתן להשתמש באלגוריתם DFS כדי למצוא כזה:

משפט 3.19 יהא G גרף מכוון חסר מעגלים. לאחר הפעלת DFS על G, הפונקציה $\varphi(v)=-v.f$ היא מיון טופולוגי של G.

במילים אחרות ־ אם נסדר את הצמתים בגרף על פי הזמן שבו האלגוריתם סיים את הטיפול בהם, **מהסוף להתחלה**, נקבל מיון טופולוגי של הגרף. **הוכחה:** תהא (u,v) קשת בגרף. עלינו להוכיח כי $v.f<u.f$. לצורך כך נפריד בין האפשרויות השונות לקשת זו:

- אם (u,v) היא קשת עץ או קשת קדמית, אז v הוא צאצא של u בעץ ה־DFS וממשפט תכונת הסוגריים 3.15, $v.f<u.f$.
- אם (u,v) היא קשת חוצה, אז בעת הגעת האלגוריתם אל קשת זו, v כבר יהיה שחור, כלומר ערכו של $v.f$ כבר נקבע. עם זאת, ערכו של $u.f$ טרם נקבע (כי האלגוריתם באמצע הסריקה של שכניו) ולכן $v.f<u.f$ גם במקרה זה.
- אם (u,v) היא קשת אחורית אז קיים בעץ ה־DFS מסלול מ־u אל v שאינו משתמש בקשת זו. אם נשרשר את (u,v) לסוף המסלול נקבל מעגל בגרף, בסתירה לכך שהגרף חסר מעגלים. מכאן שמקרה זה אינו יכול להתרחש כלל.

■

זמן הריצה של אלגוריתם זה הוא כזמן הריצה של DFS ־ $O(|V|+|E|)$. עבור מיון "רגיל" שבו כל זוג איברים ניתנים להשוואה נקבל ש־$|E|=\binom{|V|}{2}=\Theta\left(|V|^2\right)$ כך שנקבל אלגוריתם מיון תת־אופטימלי; האלגוריתם שלעיל אפקטיבי בסיטואציות שבהן יש מיעוט יחסי של קשתות ב־G.

3.5.5 מציאת רכיבים קשירים היטב בגרף

נציג כעת שימוש מתוחכם יותר ב־DFS שמאפשר לנו למצוא **רכיבים קשירים היטב** בגרף.

כזכור, גרף לא מכוון הוא **קשיר** אם קיים מסלול בין כל שני צמתים. בגרף מכוון הסיטואציה מורכבת יותר כי ייתכן שיש מסלול מ־u אל v אבל אין מסלול מ־v אל u; המושג של **קשירות היטב** בא לאפיין את הסיטואציה בה יש מסלולים בשני הכיוונים.

הגדרה 3.20 (**רכיבים קשירים היטב**) יהא $G = (V, E)$ גרף מכוון. נגדיר יחס שקילות על V באופן הבא: $u \sim v$ אם קיים מסלול $v \rightsquigarrow u$ וגם קיים מסלול $u \rightsquigarrow v$. מחלקות השקילות של יחס שקילות זה נקראות **הרכיבים הקשירים היטב** של G.

קל לראות כי היחס שהגדרנו לעיל הוא אכן יחס שקילות ולכן מושג הרכיבים הקשירים היטב מוגדר היטב. בנוסף, עולה מכך שניתן לפרק את הגרף לאוסף של תתי־גרפים קשירים היטב, כך שכל צומת בגרף שייך לאחד הרכיבים (אפשר גם לחשוב עליהם בתור תתי־גרפים מקסימליים ביחס לתכונת הקשירות־היטב). האתגר שלנו הוא למצוא את הפירוק הזה.

האלגוריתם כולל שתי הפעלות של DFS ־ אחת פשוטה ואחת מעט יותר מחוכמת. ראשית, בהינתן גרף $G = (V, E)$ נגדיר את $G^R = \left(V, E^R\right)$ בתור הגרף המתקבל באמצעות היפוך הקשתות: $E^R \triangleq \{(v, u) \mid (u, v) \in E\}$.
כעת האלגוריתם ניתן לתיאור הבא:

1. הפעילו את DFS על הגרף G
2. מיינו את V על פי $u.f$ בסדר יורד.
3. הפעילו את DFS על G^R עם קבוצת הצמתים הממויינת.
4. רכיבי הקשירות של G הם העצים השונים ביער ה־DFS שמתקבל משלב 3.

כלומר, האלגוריתם ניתן לביצוע על ידי שימוש בודד בגרסה הרגילה של DFS (שורה 1) ואז שימוש בוריאציה קלה על DFS הרגיל:

```
1  def DFS(G):
2      for u in G.V:
3          u.color = 'white'
4          u.pi = None
5      time = 0
6      for u in G.V:
7          if u.color == 'white':
8              time = DFS_visit(G, u, time)
```

בשורה 6 של אלגוריתם ה־DFS, נוודא שהמעבר על הצמתים מתבצע על פי המיון של שלב 2, ובעת הפעלת DFS_visit בשורה 8 נפעיל את הפונקציה עם מידע נוסף שמאפשר סימון של רכיב השקילות ־ למשל, את הצומת שממנו הבדיקה מתחילה, או מערך שלתוכו יוכנסו איברי הרכיב הקשיר.

נציג את מימוש האלגוריתם המורחב בשפת פייתון:

```
1   def SCC_DFS(G):
2       for u in G.V:
3           u.color = 'white'
4       time = 0
5       for u in G.V:
6           if u.color == 'white':
7               time = DFS_visit(G, u, time)
8       V_sorted = sorted(G.V, key=lambda u: -u.f)
9       E_R = [(v,u) for (u,v) in G.E]
10      G_R = Graph(V_sorted, E_R, directed=True)
11      for u in G_R.V:
12          u.color = 'white'
13      scc = []
14      for u in G_R.V:
15          if u.color == 'white':
```

```
16                  component = []
17                  SCC_DFS_visit(G_R, u, component)
18                  scc.append(component)
19       return scc
```

```
1  def SCC_DFS_visit(G, u, component):
2       component.append(u)
3      u.color = 'gray'
4      for v in G.adjacency(u):
5           if v.color == 'white':
6                SCC_DFS_visit(G, v, component)
```

הקוד דומה מאוד לקוד של DFS (ומשתמש ב־DFS_visit המקורי). ההבדל המהותי הוא הורדה של שורות "מיותרות" (למשל, מעקב הזמנים ב־SCC_DFS_visit), מיון V שמתבצע בשורה 8 והיפוך E שמתבצע בשורה 9 של SCC_DFS והכנסת צמתים לרכיבי הקשירות שמתבצעת בשורה 2 של SCC_DFS_visit.

לא נוכיח פורמלית את נכונות האלגוריתם, אבל נסביר את האינטואיציה שמאחוריו. בהינתן גרף $G=(V,E)$ ניתן להגדיר את גרף הרכיבים הקשירים היטב שלו $G^{\text{SCC}}=\left(V^{\text{SCC}},E^{\text{SCC}}\right)$ באופן הבא: V^{SCC} הוא אוסף רכיבי הקשירות היטב של G, ואילו $E^{\text{SCC}}=\{(A,B)\mid \exists a\in A,b\in B:(a,b)\in E\}$.

קל לראות כי G^{SCC} הוא גרף חסר מעגלים כי מעגל היה מוביל למיזוג כל הרכיבים שעליו (כי הצמתים שמרכיבים אותם שייכים כולם לאותו רכיב קשירות היטב). בנוסף ניתן להוכיח (זה החלק המרכזי של ההוכחה) כי אם נגדיר $f(C)=\max\{u.f\mid u\in C\}$ אז קיום הקשת $A\to B$ גורר ש־$f(A)>f(B)$. מכאן נובע שסריקת הגרף G^R בחלק השני של האלגוריתם כשמיון הצמתים הוא על פי f פירושה שאנחנו עוברים על רכיבי G^{SCC} על פי **מיון טופולוגי**.

מכיוון שאנחנו עוברים על G^{R}, הרי שבמקום הקשת $A\to B$ תהיה לנו הקשת $B\to A$. מכיוון שאנו עוברים על הרכיב A קודם, כאשר האלגוריתם יגיע אל קשת זו הרכיב A כבר יהיה מסומן והאלגוריתם לא ייכנס אליו, כך שהאלגוריתם יישאר במסגרת הרכיב B.

פרטי ההוכחה המלאים מורכבים יותר אבל נסתפק כאן בנפנוף ידיים זה.

4 עצים פורשים מינימליים

4.1 מבוא והגדרות

עץ הוא גרף לא מכוון $G=(V,E)$ שהוא **קשיר** ו**חסר מעגלים**. ניתן להוכיח שעץ הוא מקסימלי ביחס לתכונת חוסר המעגלים במובן זה שהוספת כל קשת לעץ יוצרת בו מעגל; בדומה, עץ הוא מינימלי ביחס לתכונת הקשירות במובן זה שהסרת כל קשת הופכת אותו ללא־קשיר. כאשר העץ סופי אז מתקיים $|E|=|V|-1$, וכל גרף קשיר עם $|V|-1$ קשתות הוא עץ.

לכל גרף קשיר $G=(V,E)$ קיים תת־גרף $G'=(V,E')$ עם $E'\subseteq E$ כך ש־G' הוא עץ. תת־גרף כזה נקרא **עץ פורש** של G. עבור גרף סופי קל לראות שקיים עץ פורש: כל עוד קיימת בגרף קשת שהסרה שלה אינה הופכת את הגרף לבלתי קשיר, נסיר אותה; כשנגיע למצב שבו אין קשתות כאלו הגרף שהתקבל יהיה עץ (עבור גרף אינסופי הטיעון מורכב יותר אך איננו עוסקים בגרפים אינסופיים בקורס זה).

הבעיה שנעסוק בה היא של מציאת **עץ פורש מינימלי:** בבעיה זו יש פונקציית משקל על הקשתות, ואנו רוצים למצוא עץ פורש שסכום המשקלים של קשתותיו הוא מינימלי (בכל העצים הפורשים יהיה את אותו מספר קשתות - $|V|-1$, כך שהמינימליות מתבטאת רק בפונקציית המשקל).

דוגמת שימוש קלאסית לבעייה זו היא פרישת רשת כבישים בין ערים: יש קשת בין כל זוג ערים שניתן לסלול בינם כביש, ומשקל הקשת הוא מחיר סלילת הכביש. אנו רוצים למצוא את המחיר המינימלי שנדרש לסלילת רשת כבישים שתאפשר מסע בין כל שתי ערים.

ננסח פורמלית את הבעיה:

הגדרה 4.1 (בעיית עץ פורש מינימלי): יהא $G=(V,E)$ גרף לא מכוון וקשיר ו־$w:E\to\mathbb{R}$ פונקציית משקל. לכל $E'\subseteq E$ נסמן $w(E')=\sum_{e\in E'}w(e)$. בעיית העץ הפורש המינימלי היא למצוא את $\text{argmin}_{E'\subseteq E}\{w(E')\}$ כאשר $E'\subseteq E$ רץ על כל קבוצות הקשתות כך ש־(V,E') הוא עץ.

4.2 האלגוריתם הגנרי למציאת עץ פורש מינימלי

נציג שני אלגוריתמים לפתרון בעיית העץ הפורש המינימלי: אלגוריתם קרוסקל ואלגוריתם פרים. אפשר לחשוב על שני האלגוריתמים הללו בתור מימושים שונים של אלגוריתם כללי יותר - "גנרי" - לפתרון בעיית העץ הפורש. האלגוריתם בונה את העץ הפורש "מלמטה למעלה" - מתחילים עם תת־גרף בלי קשתות כלל, ומוסיפים אליו קשתות עד לקבלת העץ הפורש המינימלי.

האלגוריתם מתחיל עם הקבוצה $A=\emptyset$. במצב זה, A היא בעלת התכונה שהיא תת־קבוצה של קבוצת הקשתות בעץ פורש מינימלי כלשהו. בכל איטרציה של האלגוריתם, הוא מוצא קשת $e\in E$ כך ש־$A\cup\{e\}$ גם היא תת־קבוצה של קבוצת הקשתות בעץ פורש מינימלי כלשהו. קשת כזו נקראת **קשת בטוחה** עבור A.

מכיוון שנשמרת האינוריאנטה לפיה A היא תת־קבוצה של קבוצת הקשתות בעץ פורש מינימלי, כאשר מגיעים למצב שבו $|A|=|V|-1$, אז A תהווה את קבוצת הקשתות המלאה של עץ פורש מינימלי כלשהו.

האלגוריתם הגנרי, אם כן, הוא בסך הכל:

1. אתחלו $A=\emptyset$
2. בצעו $|V|-1$ פעמים:
   (א) מצאו קשת בטוחה e עבור A
   (ב) הגדירו $A\leftarrow A\cup\{e\}$
3. החזירו את A

כמובן, האלגוריתם כולו נשען על השאלה "כיצד למצוא קשת בטוחה עבור A". הן קרוסקל והן פרים מוצאים קשת כזו באמצעות בחירת **חתך** בגרף. כזכור, **חתך** $(S,V\backslash S)$ הוא פירוק של צמתי הגרף לשתי קבוצות זרות ומשלימות. קשת $e=(u,v)$ **חוצה** את החתך אם $u\in S,v\in V\backslash S$ או $v\in S,u\in V\backslash S$. נאמר שחתך **מכבד** את A אם אף קשת של A לא חוצה את החתך (כלומר, תת־הגרף שפורשות הקשתות ב־A מוכל כולו באחת הקבוצות בחתך).

המשפט המרכזי שלנו נוגע לקשת הקלה ביותר שחוצה חתך:

משפט 4.2 יהא $G=(V,E)$ גרף לא מכוון וקשיר ו־$w:E\to\mathbb{R}$ פונקציית משקל. תהא $A\subseteq E$ קבוצה שמוכלת בקבוצת הקשתות של עץ פורש מינימלי של G עבור w. יהא $(S,V\backslash S)$ חתך של G שמכבד את A ותהא $e=(u,v)$ קשת שחוצה את החתך ומשקלה מינימלי מבין הקשתות שחוצות את החתך, אז $A\cup\{e\}$ גם כן מוכלת בקבוצת הקשתות של עץ פורש מינימלי של G עבור w.

הוכחה: יהא $T=(V,E_T)$ העץ הפורש המינימלי של G שמכיל את A שקיומו מובטח מתנאי המשפט. אם $e\in E_T$ סיימנו, כי גם $A\cup\{e\}$ מוכלת באותה קבוצת קשתות. נניח אם כן כי $e\notin E_T$ ונראה כי קיים עץ פורש מינימלי **אחר**, T', שכן מכיל את $A\cup\{e\}$, שיתקבל על ידי "תיקון" של העץ T.

נתבונן בקשת $e=(u,v)$. מכיוון שהקשת חוצה את החתך, מתקיים בלי הגבלת הכלליות ש־$u\in S$ ו־$v\in V\backslash S$. מכיוון ש־T הוא עץ פורש, קיים מסלול ב־T מ־u אל v, והמסלול חייב לחצות את החתך בשלב כלשהו (כי הצומת הראשון הוא ב־S והאחרון אינו ב־S ולכן קיימת קשת שמעבירה מ־S אל מחוץ ל־S). נסמן ב־(x,y) את הקשת על המסלול שחוצה את החתך, כך ש־$x\in S,y\in V\backslash S$. בפרט, קיימים המסלולים $x\rightsquigarrow u,v\rightsquigarrow y$.

מכיוון שהחתך מכבד את A, מתקיים ש־$(x,y)\notin A$. לכן A מוכלת בקבוצה $E_{T'}=(E_T\backslash\{(x,y)\})\cup\{e\}$ שמתקבלת מהחלפת (x,y) ב־(u,v). נשאר רק להראות שקבוצה זו מגדירה עץ פורש מינימלי. מכיוון ש־$e\notin E_T$ הרי שאחרי הסרת הקשת (x,y) והוספת e קיבלנו $|E_{T'}|=|E_T|=|V|-1$ ולכן אם הגרף T' קשיר, הוא עץ.

ניקח שני צמתים כלשהם בגרף. היה קיים ב־T מסלול שמחבר את שניהם. אם הוא לא עבר דרך (x,y) הוא קיים גם ב־T'. אם הוא עובר דרך (x,y) אז נתקן את המסלול כך: עם ההגעה ל־x ימשיך המסלול אל v דרך המסלול $x\rightsquigarrow u$ שאת קיומו ראינו קודם, יעבור אל v דרך הקשת $e=(u,v)$ ולבסוף ימשיך אל y דרך $v\rightsquigarrow y$ ומשם ימשיך כמו במסלול המקורי. הראינו כי T' קשיר ולכן עץ.

נותר להראות כי T' הוא עץ פורש מינימלי. מכיוון שהוא התקבל מהעץ הפורש המינימלי T על ידי הסרת (x,y) והוספת (u,v) די להראות ש־$w(u,v)\le w(x,y)$, כלומר החלפת הקשת יכולה רק להקטין את משקל העץ. מכיוון שגם (x,y) וגם (u,v) חוצות את החתך $(S,V\backslash S)$ ו־(u,v) נבחרה להיות בעל משקל מינימלי מבין כל הקשתות שחוצות את החתך, קיבלנו $w(u,v)\le w(x,y)$ כמבוקש. ■

4.3 אלגוריתם קרוסקל

הרעיון באלגוריתם קרוסקל הוא לבנות את העץ הפורש על ידי **מיזוג** של תתי־עצים שונים יחד. בתחילת ריצת האלגוריתם, כל צומת בגרף הוא "עץ" בפני עצמו, ללא קשתות. בכל צעד של האלגוריתם נבחרת הקשת הקלה ביותר שהוספתה לעץ תאחד שני רכיבי קשירות שעד כה היו נפרדים. ניתן לראות כי זוהי קשת בטוחה עבור קבוצת הקשתות שנבחרו עד כה, ולכן האלגוריתם תואם את המבנה של האלגוריתם הגנרי ומניב עץ פורש מינימלי.

על מנת לממש את האלגוריתם יש צורך במבנה נתונים של **קבוצת זרות**. המבנה הזה כולל אוסף של קבוצות זרות של איברים, וצריך לתמוך ביעילות בשתי פעולות: **מציאת** הקבוצה שבה נמצא איבר נתון, **ואיחוד** שתי קבוצות זרות. מבנה נתונים כזה נקרא "מבנה נתונים Disjoint-set" או "מבנה נתונים Union-find" על שם שתי הפעולות שבהן הוא נועד לתמוך. קיימים מימושים מחוכמים למדי למבנה נתונים זה אך אנו נסתפק כאן במימוש נאיבי בשפת Python:

```
class DisjointSets():
    def __init__(self, elements):
        self.sets = {element: element for element in elements}
    def find(self, element):
        if self.sets[element] == element:
            return element
        return self.find(self.sets[element])
    def union(self, element_1, element_2):
        self.sets[self.find(element_1)] = self.find(element_2)
```

פרטי המימוש אינם חשובים כאן ־ הרעיון הוא לייצג כל קבוצה באמצעות נציג, כאשר המידע שנשמר בפועל, עבור כל איבר, הוא מה נציג הקבוצה שלו. איחוד קבוצות מתבצע על ידי שינוי הנציג של אחת הקבוצות כך שיצביע על נציג הקבוצה השניה במקום על עצמו.

בסיוע מבנה נתונים זה, הקוד ב־Python של אלגוריתם קרוסקל הוא פשוט:

```
1 def kruskal(G):
2     A = []
3     sets = DisjointSets(G.V)
4     sorted_E = sorted(G.E, key=lambda e: G.w[e])
5     for (u,v) in sorted_E:
6         if sets.find(u) != sets.find(v):
7             A.append((u,v))
8             sets.union(u,v)
9     return A
```

בשורה 2 מאותחלת רשימת הקשתות בעץ להיות הרשימה הריקה; בשורה 3 מאותחלת הקבוצות שלנו כך שבהתחלה כל צומת הוא קבוצה בפני עצמה, ובשורה 4 ממויינת רשימת הקשתות בסדר עולה על פי המשקל של כל קשת.

לאחר מכן, בשורה 5 עוברים על הקשתות על פי הסדר, בשורה 6 בודקים האם הקשת הנוכחית מחברת בין שני רכיבי קשירות נפרדים, ואם כן מוסיפים אותה ל־A בשורה 7 ומאחדים את רכיבי הקשירות בשורה 8.

מדוע קשת (u,v) שמקיימת את התנאי בשורה 6 היא קשת בטוחה עבור A? נתבונן בחתך $(S,V\backslash S)$ שבו S היא רכיב הקשירות של u. החתך מכבד את A שכן קשת מ־A שחוצה אותו משמעותה ששני צדי הקשת שייכים לרכיב הקשירות של u וזוהי סתירה לכך שהצד השני של הקשת אמור להיות ב־$V\backslash S$. כמו כן, הקשת (u,v) חוצה את החתך (כי התנאי בשורה 6 מתקיים) וכל קשת אחרת שחוצה את החתך חייבת להיות בעל משקל גדול או שווה לה, כי אם לקשת היה משקל נמוך יותר האלגוריתם היה עובר עליה קודם ומוסיף אותה (כי שני צדדיה הם כעת בשני רכיבי קשירות שונים ולכן בוודאי שהיו כך גם קודם). מכאן שעל פי משפט 4.2 אלגוריתם קרוסקל הוא מימוש אפשרי של האלגוריתם הגנרי לעץ פורש מינימלי.

נותר להבין את הסיבוכיות של האלגוריתם. שלב מיון הקשתות 4 ניתן לביצוע בסיבוכיות $O\left(|E|\log|E|\right)$. סיבוכיות המשך הקוד תלויה במימוש של מבנה הנתונים שבו אנו משתמשים עבור DisjointSets. במימוש הנאיבי שהצגנו, הסיבוכיות של פעולות find ושל union היא $O\left(|V|\right)$. מכיוון שפעולה זו מתבצעת $O\left(|E|\right)$ פעמים אנו מקבלים סיבוכיות של $O\left(|V|\cdot|E|\right)$.

קיים למבנה הנתונים DisjointSet מימוש יעיל בהרבה שלא נציג כאן שסיבוכיות הריצה שלו עבור ביצוע m פעולות היא $O\left(m\cdot\alpha\left(n\right)\right)$ כאשר α היא פונקציה עם קצב גידול קטן מאוד. במקרה שלנו שימוש במבנה הנתונים הזה יניב שסיבוכיות כל השלב הזה היא $O\left(|E|\log|E|\right)$ ולכן זוהי גם סיבוכיות האלגוריתם כולו.

4.4 אלגוריתם פרים

הרעיון באלגוריתם פרים הוא לבנות את העץ הפורש על ידי **הרחבה הדרגתית** של עץ יחיד. האלגוריתם מתחיל מצומת r שישמש בתור שורש העץ, ובכל שלב בוחר להוסיף לעץ צומת חדש על ידי חיבור שלו לצומת שכבר בעץ, כאשר הצומת נבחר בצורה חמדנית, בתור הצומת שמחיר ההוספה שלו לעץ (כלומר, משקל הקשת שמחברת אותו לעץ) הוא מינימלי.

המבנה של האלגוריתם כמעט זהה לזה של אלגוריתם דייקסטרה - ניתן להשתמש באותו קוד כמו עבור דייקסטרה עם שינויים מינוריים. בפרט, הרכיב המרכזי באלגוריתם הוא **תור עדיפויות** שבו מוחזקים צמתי הגרף כשהם ממויינים, ובכל איטרציה מוצא ממנו הצומת שמשקלו מינימלי. המשקל של כל צומת, כאמור, הוא משקל הקשת הקלה ביותר שמחברת אותו לצומת כלשהו בעץ; כדי לוודא שמספר זה מעודכן, עם כל הוספת צומת לעץ עוברים על שכניו שמחוץ לעץ ומעדכנים את המשקל שלהם במידת הצורך.

נציג מימוש של האלגוריתם בשפת Python:

```
1   def prim(G, s):
2       for v in G.V:
3           v.d = math.inf
4           v.pi = None
5       s.d = 0
6       Q = [v for v in G.V]
7       while len(Q) > 0:
8           u = pop_min(Q)
9           for v in G.adjacency(u):
10              if v in Q and G.w[(u,v)] < v.d:
11                  v.d = G.w[(u,v)]
12                  v.pi = u
```

במימוש שאנו מציגים פה, בדומה לדייקסטרה, אנו מתחילים מצומת s בגרף (בחירת צמתים שונים עשויה להניב עצים פורשים שונים). לכל צומת אנו מגדירים את $v.d$ בתור ה"מחיר" של הוספת v לעץ (בהתחלה $v.d=\infty$ לכל צומת למעט s שמחירה 0). את הקשתות של העץ נשמור בתוך $v.\pi$ שעבור כל צומת מצביע על אביו בעץ, בדומה למה שעשינו עבור דייקסטרה.

בשורה 6 נבנה תור עדיפויות ובשורה 8 מוצא ממנו האיבר המינימלי u (אנו משתמשים באותה פונקציה pop_min כמו בדייקסטרה). הוספת u לעץ מתבצעת באופן מובלע על ידי הסרתו מתור העדיפויות; בזמן ההוצאה של u, הקשת $u.\pi$ היא זו שמחירה מינימלי מבין כל אלו שמחברות את u לעץ, ולכן זוהי הקשת ש"מתווספת לעץ".

לבסוף, שורות 10-12 מעדכנות עבור כל שכני הצומת שהוספנו לעץ את משקל הקשת המינימלית שמחברת אותם לעץ - זאת בהינתן והקשת שמחברת אותם לצומת החדש בעץ קלה יותר מאשר המשקל המינימלי הקודם.

נכונות האלגוריתם מתבססת על משפט 4.2. כדי לראות זאת, נסתכל בשלב כלשהו באלגוריתם על חתך $(S,V\backslash S)$ שבו S כוללת את כל הצמתים שהתווספו עד כה אל העץ. הקשת שנבחרת בשלב זה להוספה לגרף היא זו של הצומת שנשלף בשורה 8. משקל הצומת הזה הוא משקל הקשת שאנו מוסיפים לעץ, ומשקל זה הוא מינימלי מבין זה של כל הקשתות שמחברות את צמתי העץ לצמתים שב-Q, מה שממלא את תנאי המשפט.

סיבוכיות האלגוריתם תלויה במימוש של Q. במימוש באמצעות ערימה, שלב האתחול בשורה 6 דורש זמן $O\left(|V|\right)$. פעולת הוצאת האיבר המינימלי בשורה 8 מתבצעת $|V|$ פעמים בסיבוכיות $O\left(\log|V|\right)$ פעמים כך שאנחנו מקבלים סיבוכיות $O\left(|V|\log|V|\right)$.

לבסוף, שורות 10-12 מבוצעות פעמיים עבור כל קשת בגרף, כלומר $2|E|$ פעמים (כי הן מתבצעות פעם אחת לכל נקודת קצה של קשת). בכל פעם כזו יש לבדוק שייכות ל-Q, מה שניתן למימוש בזמן $O\left(1\right)$ במימוש חכם (למשל שדה נוסף לכל צומת שאומר האם הוא ב-Q או לא ומעודכן בהתאם). בנוסף, שינוי הערך בשורה 11 מצריך עדכון של הערימה, בסיבוכיות $O\left(\log|V|\right)$, כך שבסך הכל אנחנו מקבלים שזמן ריצת האלגוריתם הוא $O\left(|E|\log|V|\right)$, וזה הגורם הדומיננטי בסיבוכיות זמן הריצה (מכיוון שהגרף קשיר יש בו לפחות $|V|-1$ קשתות, כלומר $|V|=O\left(|E|\right)$).

כמו באלגוריתם דייקסטרה, מימוש Q באמצעות **ערימת פיבונאצ'י** יכול לשפר את זמן הריצה עוד יותר, אך לא נציג זאת כאן.

5 קידוד חסר רעש וקוד האפמן

5.1 מבוא

טקסטים במחשב מיוצגים על ידי **קידוד** של התווים בטקסט לסדרות של ביטים. דוגמא פשוטה לקידוד הזה היא קוד ASCII: בקוד הזה מיוצגים 256 תווים שונים על ידי 8 ביטים. כך למשל התו "F" מיוצג על ידי המחרוזת 01000110 ואילו התו "E" הקודם לו מיוצג על ידי המספר הקודם, 01000101.

שיטה זו ודומות לה הן שימושיות ומועילות באופן כללי, אך בבירור יש בהן **בזבוז** של ביטים. זאת מכיוון שלא כל התווים מופיעים בטקסט באותה תדירות. הסימן { כנראה יופיע בטקסט פחות מאשר האות e. איך ניתן לנצל זאת? אם e תיוצג באמצעות **פחות ביטים**, מה שיתבטא בכך שתווים פחות נפוצים יהיו מיוצגים באמצעות **יותר ביטים**.

נדגים זאת באמצעות דוגמא פשוטה. המילה GATTACAA היא בעלת 8 אותיות שכולן בקבוצה $\{G, A, C, T\}$. מכיוון שיש רק ארבע אותיות, קידוד נאיבי שלהן יתבסס על שני ביטים:

אות	קידוד
G	00
A	01
C	10
T	11

הייצוג של GATTACAA בצורה זו יהיה מאורך 16 ביטים (8 תווים כפול 2 ביטים לתו): 0001111101100101.

ננסה למצוא קידוד חסכוני יותר.

על ידי מעבר בודד על המילה ניתן לספור את המופעים של כל אות ואת התדירות שבה היא מופיעה במילה (מספר המופעים שלה חלקי אורך המילה:

אות	מופעים	תדירות
G	1	$\frac{1}{8}$
A	4	$\frac{1}{2}$
C	1	$\frac{1}{8}$
T	2	$\frac{1}{4}$

בבירור A היא האות הנפוצה ביותר ולכן כדאי לתת לה "עדיפות", בזמן ש־T היא בעדיפות שניה ואילו G,C חולקות את המקום האחרון. נציע אם כן שיטת קידוד שנדמה ששלפנו מהשרוול אך בהמשך נראה כיצד ניתן למצוא אלגוריתמית:

אות	קידוד
G	110
A	0
C	111
T	10

בשיטה זו קיצרנו את קידוד A בביט והמחיר ש"שילמנו" על כך הוא הגדלת אורך הקידוד של G,C. נחשב מה כעת יהיה אורך הקידוד של המילה: $4\cdot 1+2\cdot 2+1\cdot 3+1\cdot 3=14$ - כלומר, חסכנו שני ביטים (12.5 אחוז מאורך המילה). קידוד המילה בשיטה החדשה יהיה 11001010011100.

באופן כללי, כל טקסט שמורכב מארבע אותיות אלו בהתפלגות התדירות הזו ואורכו הוא N ידרוש $2N$ ביטים בקידוד נאיבי ו־$\left(\frac{1}{2}+\frac{2}{4}+\frac{3}{8}+\frac{3}{8}\right)N=1.75N$ ביטים בקידוד המשופר שהצענו. ככל שיש אותיות רבות יותר שהתפלגות התדירויות שלהן רחוקה מלהיות אחידה, כך שיטת קידוד באורך משתנה הופכת לחסכונית יותר ויותר.

5.2 קודי רישא

קידוד מחרוזת מתבצע בקלות על ידי החלפת כל תו במחרוזת בסדרת הביטים שמקודדת אותו, שנקראת **מילת הקוד** או **הקידוד** של המחרוזת. **פיענוח** קידוד הוא המרה של הייצוג באמצעות ביטים חזרה למחרוזת. כאשר מספר הביטים לכל תו הוא קבוע, הקידוד ניתן לביצוע בקלות: מחלקים את מילת הקוד לסדרות בהתאם לאורך (למשל, סדרות מאורך 8 בקוד ASCII) ומתרגמים כל סדרה בנפרד חזרה לתו שהיא מקודדת.

לרוע המזל, כשעוברים לקידוד באמצעות סדרות מאורך משתנה, נהיה פחות ברור כיצד לפענח ובבחירה לא טובה של קוד הפיענוח עשוי להיות לא חד־ערכי כלל. למשל, אם האות A מקודדת על ידי 1010 והאות B מקודדת על ידי 10 אז הסדרה 101010 מקודדת הן את AB והן את BA ואין לנו דרך לדעת מה היה במקור.

דרך פתרון פשוטה לבעיה זו, שמבטיחה גם שלא תהיה "התנגשות" של קידודים וגם פיענוח קל, היא שימוש ב**קודי רישא**. בקודים הללו, הקידוד של אף אות אינו רישא של קידוד של אות אחרת, ולכן כאשר קוראים את מילת הקוד אין שלב שבו אנו עומדים בפני הדילמה האם הסתיימה אות בקידוד ומתחילה אות חדשה, או שמא אנחנו רק באמצע האות הנוכחית.

הקוד שהצגנו קודם הוא דוגמא לקוד רישא שכזה:

אות	קידוד
G	110
A	0
C	111
T	10

פענוח מתבצע בצורה הבאה: אם הביט הבא הוא 0, התו הבא הוא A; אם הוא 1 אנו משהים את השיפוט להמשך. אם הביט השני הוא 0 אז התו הבא הוא T, ואם הוא 1 אנו משהים את השיפוט להמשך, והביט השלישי קובע האם מדובר על G או C. ניתן לתאר את התהליך בצורה פשוטה באמצעות **עץ בינארי** שקשתותיו מסומנות ב־0 ו־1 וכל עלה בו מייצג את אחת האותיות, כך שהקידוד של האות רשום על הקשתות במסלול מהשורש אל העלה הזה.

נעבור להגדרות פורמליות. מטעמי נוחות ה"אותיות" שלנו יהיו פשוט המספרים בקבוצה $\{1,2,\ldots,M\}$ עבור M טבעי חיובי כלשהו.

הגדרה 5.1 קוד רישא הוא קבוצת מילים בינאריות $w_1,\ldots,w_M\in\{0,1\}^*$ כך שלכל $i\neq j$, w_i אינה רישא של w_j (כלומר, $w_j\neq w_iu$ עבור $u\in\{0,1\}^*$).

משפט 5.2 יהא $\{w_1,\ldots,w_M\}$ קוד רישא, אז כל קידוד של מילה באמצעות הקוד הוא יחיד. כלומר אם עבור שתי סדרות $(i_1,\ldots,i_k)$ ו־$(j_1,\ldots,j_t)$ מתקיים $w_{i_1}\cdots w_{i_k}=w_{j_1}\cdots w_{j_t}$ אז $k=t$ ו־$i_r=j_r$ לכל $1\le r\le k$.

הוכחה: נוכיח את הטענה באינדוקציה על k. נניח ש־$w_{i_1}\cdots w_{i_k}=w_{j_1}\cdots w_{j_t}$ ונתבונן ב־w_{i_1},w_{j_1}. אם אחת מהמילים הללו ארוכה מהשניה, אז בגלל ששתיהן מופיעות החל מתחילת המילה האותיות שלהן זהות עד לשלב שבו המילה הקצרה יותר נגמרת ־ כלומר, מילה אחת היא רישא של המילה השניה, בסתירה להנחה שלנו שזהו קוד רישא.

אם כן, $|w_{i_1}|=|w_{j_1}|$ ומכיוון ששתי המילים מופיעות בתחילת מילת הקוד, האותיות שלהן זהות, כלומר $w_{i_1}=w_{j_1}$. כעת נפעיל את הנחת האינדוקציה על $w_2\cdots w_{i_k}=w_{j_2}\cdots w_{j_t}$ ונקבל את המבוקש. ■

משפט 5.3 יהא $\{w_1,\ldots,w_M\}$ קוד רישא, אז קיים אלגוריתם שלכל סדרה $u=b_1\cdots b_n$ של n ביטים, אם קיימת סדרה $(i_1,\ldots,i_k)$ כך ש־$u=w_{i_1}\cdots w_{i_k}$ אז הוא מוצא אותה ואחרת מחזיר שאין כזו ופועל בסיבוכיות $O(n)$.

הוכחה: נתבונן בעץ שצמתיו הם כל הרישות של מילים מ־$\{w_1,\ldots,w_M\}$. השורש מייצג את הרישא ε (המילה הריקה). לצומת שמייצג את u יהיו לכל היותר שני בנים ־ הצמתים שמייצגים את $u\cdot 0$ ואת $u\cdot 1$, והקשתות אל בנים אלו יסומנו ב־0 ו־1 בהתאמה. עלי העץ הם בדיוק המילים $\{w_1,\ldots,w_M\}$ שכן העובדה שאף מילה בקבוצה היא רישא של מילה אחרת בה מבטיחה שלא יצאו קשתות מהצמתים שמתאימים למילים אלו.

נשים לב שלכל צומת $u=\sigma_1\cdots\sigma_k$, סדרת סימוני הקשתות שמובילה מ־ε אל u היא בדיוק $\sigma_1,\ldots,\sigma_k$ (קל להוכיח זאת באינדוקציה).

כעת בהינתן סדרה $u=b_1\cdots b_n$ של ביטים, האלגוריתם שלנו יפעל באופן הבא:

1. התחל מהצומת ε בעץ.
2. קרא את האות b_i הבאה מהקלט u.
3. אם לא קיימת קשת יוצאת מ־u עם הסימון b_i, החזר "לא קיים פיענוח"
4. אם קיימת קשת יוצאת מ־u עם הסימון b_i, עבור אל הצומת שמתאים לקשת זו.
5. אם הצומת אליו הגעת הוא עלה המתאים למילה w, הוסף את w לפלט וחזור לצומת ε בעץ.
6. אם הקלט נגמר והאלגוריתם אינו בצומת ε, החזר "לא קיים פיענוח"

7. אם הקלט נגמר והאלגוריתם בצומת ε החזר את רשימת האינדקסים של ה־w שהתווספו לפלט במהלך הריצה.

8. חזור לשלב 2.

ניתן להוכיח כי אלגוריתם זה אכן מפענח את קידוד המילה ומכיוון שהוא קורא כל אות ב־u בדיוק פעם אחת ומבצע פעולות בסיבוכיות קבועה לכל אות כזו, סיבוכיות הריצה שלו היא $O\left(n\right)$. ■

5.3 קוד האפמן

ראינו את השימושיות של קודי רישא ואת האופן שבו ניתן לייצג אותם באמצעות עץ. כעת נרצה לעסוק בשאלה ־ מה קוד הרישא **האופטימלי** בהינתן שאנו יודעים את תדירות האותיות בטקסט שאנחנו רוצים לקודד? (למשל, על ידי ביצוע מעבר בודד על הטקסט וחישוב שלהן)

נפתח בהגדרה:

הגדרה 5.4 יהא C אלפבית כלשהו ונניח כי לכל $c\in C$ נתונה לנו תדירות, $f\left(c\right)\in\left[0,1\right]$ כך ש־$\sum_{c\in C}f\left(c\right)=1$. יהא קוד רישא עבור C שמיוצג באמצעות עץ T, ולכל $c\in C$ נסמן ב־$d_{T}\left(c\right)$ את עומק העלה של c בעץ T. אז **מחיר** הקוד T ביחס להתפלגות הנתונה מוגדר בתור $B\left(T\right)=\sum_{c\in C}d_{T}\left(c\right)\cdot f\left(c\right)$.

במילים אחרות, $B\left(T\right)$ הוא **ממוצע משוקלל** של המחיר בביטים של כל $c\in C$ כשהמשקלות הם התדירויות של האותיות ב־C. קל לראות כי עומק העלה של c ב־T אכן שווה למספר הביטים שבהם מיוצג c בעץ.

המשימה העומדת לפנינו כעת היא פשוטה לניסוח: בהינתן התדירויות $f\left(c\right)$ של כל $c\in C$, ברצוננו לבנות קוד רישא T עבור C כך ש־$B\left(T\right)$ הוא מינימלי. האלגוריתם של האפמן משיג מטרה זו בגישה **חמדנית** שבונה את העץ של הקוד "מלמטה למעלה". אנו מתחילים כשבגרף שלנו נכללים רק עלי העץ עם כל האותיות ב־C, ובכל איטרציה אנחנו מאחדים את שני הצמתים בעלי תדירות **מינימלית** (ככל שצומת מאוחד מהר יותר, כך הוא יהיה נמוך יותר בעץ ולכן בעל משקל גבוה יותר). האיחוד מתבטא בהוספת צומת פנימי חדש בעץ שהצמתים המאוחדים הם בניו; מכאן אפשר להמשיך באינדוקציה כשמתייחסים לצומת הפנימי בתור "עלה" חדש, עם תדירות ששווה לסכום התדירויות של בניו.

מבחינה רעיונית האלגוריתם מזכיר את האלגוריתם של קרוסקל לעץ פורש מינימלי במובן זה שאנחנו מתחילים עם איבר מבודדים ומבצעים סדרה של "איחודים" עד שכולם נמצאים באותו עץ. עם זאת, האלגוריתם שונה בכך שהקשתות אינן נתונות מראש אלא נתונות לבחירתו של האלגוריתם, והאופן שבו מזהים את הצמתים בעלי תדירוות מינימלית היא באמצעות **תור עדיפויות** כמו זה ששימש אותנו באלגוריתם של פרים ודייקסטרה.

נציג מימוש של האלגוריתם בשפת Python:

```
1   def huffman(C):
2       Q = []
3       for c, freq in C.items():
4           v = Vertex(c)
5           v.freq = freq
6           Q.append(v)
7       for i in range(len(C) - 1):
8           x = pop_min(Q)
9           y = pop_min(Q)
10          z = Vertex(i)
11          z.left = x
12          z.right = y
13          z.freq = x.freq + y.freq
14          Q.append(z)
15      return pop_min(Q)
```

בקוד אנחנו מניחים שנתון לנו מילון C שכולל זוגות של תו (c) והתדירות שלו (freq). לכל זוג כזה אנחנו מייצרים צומת חדש בגרף עם שדה freq; צמתים אלו ישמשו בתור העלים. בלולאה שמתחילה בשורה 7 אנו מייצרים את הצמתים הפנימיים בגרף. תור העדיפויות Q נבנה כך שהשליפה ממנו מתבצעת על פי שדה ה־freq של האיברים.

בשורות 8-9 אנו מוצאים את הצמתים בעלי התדירויות הנמוכות ביותר. בשורות 10-13 אנחנו מייצרים צומת חדש z שישמש בתור האב של שני צמתים אלו בעץ, וקובעים את התדירות שלו להיות סכום התדירויות של הבנים.
סיבוכיות האלגוריתם תלויה במימוש Q. במימוש סטנדרטי באמצעות ערימה, האתחול דורש זמן $O\left(n\right)$ והלולאה הפנימית מתבצעת n פעמים כשבכל פעם מופעלות על Q פעולות שדורשות זמן $O\left(\log n\right)$ כך שסיבוכיות הזמן הכוללת היא $O\left(n\log n\right)$.
נותר להשתכנע בכך שהקוד שהאלגוריתם מחזיר הוא אכן קוד הרישא האופטימלי עבור רשימת התדירויות שהתקבלה כקלט. לשם כך נשים לב לאופן הפעולה של האלגוריתם בכל איטרציה שלו:

- בונה מהמילון C מילון חדש C' שמתקבל על ידי הסרת $x,y\in C$ והוספת $z\in C'$ כך ש-$f\left(z\right)=f\left(x\right)+f\left(y\right)$ ואז ממשיך את הריצה עליו.
- בונה עץ T עבור C על ידי כך שהוא מחבר אל העץ T' שנוצר מתוך C' את הצמתים עבור x,y בתור בנים של הצומת שנוצר עבור z (וכעת מאבד את הסימון הזה שלו ונשאר צומת פנימי חסר סימון).

אם נוכיח שבניית T באופן הזה מתוך עץ **אופטימלי** עבור C' מניבה עץ אופטימלי עבור C ההוכחה תסתיים באמצעות אינדוקציה פשוטה. ראשית ננסה להעריך את "הפרש המחיר" בין T ו-T'. כזכור, הגדרנו $B\left(T\right)=\sum_{c\in C}d_{T}\left(c\right)\cdot f\left(c\right)$. כעת:

- $C=\left(C'\backslash\left\{ z\right\} \right)\cup\left\{ x,y\right\}$
- $f\left(z\right)=f\left(x\right)+f\left(y\right)$
- $d_{T}\left(x\right)=d_{T}\left(y\right)=d_{T'}\left(z\right)+1$
- $d_{T}\left(c\right)=d_{T'}\left(c\right)$ לכל $c\in C\cap C'$ (כי לא שינינו את הצמתים של תווים אלו בעץ).

בהינתן כל אלו, נבצע חישוב אלגברי פשוט:

$$\begin{aligned}B\left(T\right) & =\sum_{c\in C}d_{T}\left(c\right)f\left(c\right)\\
 & =\sum_{c\in C\cap C'}d_{T}\left(c\right)f\left(c\right)+d_{T}\left(x\right)f\left(x\right)+d_{T}\left(y\right)f\left(y\right)\\
 & =\sum_{c\in C\cap C'}d_{T'}\left(c\right)f\left(c\right)+\left(d_{T'}\left(z\right)+1\right)\left(f\left(x\right)+f\left(y\right)\right)\\
 & =\sum_{c\in C\cap C'}d_{T'}\left(c\right)f\left(c\right)+d_{T'}\left(z\right)f\left(z\right)+f\left(z\right)\\
 & =\sum_{c\in C'}d_{T'}\left(c\right)f\left(c\right)+f\left(z\right)\\
 & =B\left(T'\right)+f\left(z\right)
\end{aligned}$$

בהמשך נוכיח כי עבור C קיים עץ T_{opt} כך ש-$B\left(T_{\text{opt}}\right)$ מינימלי ו-x,y הם אחים, כלומר בנים של אותו צומת. נראה כיצד ניתן להשתמש בעץ כזה כדי להוכיח ש-$B\left(T\right)=B\left(T_{\text{opt}}\right)$.
נבנה מתוך T_{opt} עץ חדש T'_{opt} על ידי הסרת x,y והחלפתם בצומת z עם $f\left(z\right)=f\left(x\right)+f\left(y\right)$.
מהשוויון שראינו קודם, נקבל $B\left(T_{\text{opt}}\right)=B\left(T'_{\text{opt}}\right)+f\left(z\right)$.
כמו כן, T' נבחר להיות אופטימלי, ולכן $B\left(T'\right)\le B\left(T'_{\text{opt}}\right)$. נקבל:

$$\begin{aligned}B\left(T\right) & =B\left(T'\right)+f\left(z\right)\\
 & \le B\left(T'_{\text{opt}}\right)+f\left(z\right)\\
 & =B\left(T_{\text{opt}}\right)\le B\left(T\right)
\end{aligned}$$

כלומר $B\left(T_{\text{opt}}\right)=B\left(T\right)$, כמבוקש.

נותר להוכיח כי אכן קיים עץ T_{opt} אופטימלי עבור C שבו x,y הם אחים. הרעיון הוא לקחת עץ אופטימלי T_{opt} **כלשהו** ואם הוא לא מתאים לקריטריון - לתקן אותו בהתאם.

יהא T_{opt} עץ אופטימלי ונסתכל על צומת a מעומק מקסימלי בעץ. אם לא קיים ל-a אח אז אפשר להחליף את אביו של a ב-a עצמו ולקבל עץ בעל מחיר קטן יותר, כך שקיים ל-a אח b. כזכור, x,y נבחרו כך שערך ה-f שלהם מינימלי. בלי הגבלת הכלליות נניח ש-$f\left(x\right)\le f\left(y\right)$ ו-$f\left(a\right)\le f\left(b\right)$ ולכן נוכל להסיק:

- $f\left(x\right)\le f\left(a\right)$
- $f\left(y\right)\le f\left(b\right)$

(ההסקה השניה נובעת מכך שאם $f\left(b\right)<f\left(y\right)$ אז מתקבלת השרשרת $f\left(x\right)\le f\left(a\right)\le f\left(b\right)<f\left(y\right)$ שמראה ש-y אינו אחד משני הצמתים המינימליים בעץ).

מכיוון ש-a,b נבחרו להיות בעלי עומק מקסימלי ב-T_{opt}, נקבל:

- $d_{T_{\text{opt}}}\left(x\right)\le d_{T_{\text{opt}}}\left(a\right)$
- $d_{T_{\text{opt}}}\left(y\right)\le d_{T_{\text{opt}}}\left(b\right)$

כעת נחליף את מיקומי הצמתים a,x ונקבל עץ חדש T'_{opt}. נשווה את מחירו למחיר T_{opt}:

$$\begin{aligned}
B\left(T_{\text{opt}}\right)-B\left(T'_{\text{opt}}\right) &= \sum_{c\in C}d_{T_{\text{opt}}}\left(c\right)f\left(c\right)-\sum_{c\in C}d_{T'_{\text{opt}}}\left(c\right)f\left(c\right)\\
&= d_{T_{\text{opt}}}\left(a\right)f\left(a\right)+d_{T_{\text{opt}}}\left(x\right)f\left(x\right)-d_{T'_{\text{opt}}}\left(a\right)f\left(a\right)-d_{T'_{\text{opt}}}\left(x\right)f\left(x\right)\\
&= d_{T_{\text{opt}}}\left(a\right)f\left(a\right)+d_{T_{\text{opt}}}\left(x\right)f\left(x\right)-d_{T_{\text{opt}}}\left(x\right)f\left(a\right)-d_{T_{\text{opt}}}\left(a\right)f\left(x\right)\\
&= \left(d_{T_{\text{opt}}}\left(a\right)-d_{T_{\text{opt}}}\left(x\right)\right)\left(f\left(a\right)-f\left(x\right)\right)\\
&\ge 0
\end{aligned}$$

כאשר אי השוויון האחרון נובע מכך ש-$d_{T_{\text{opt}}}\left(a\right)\le d_{T_{\text{opt}}}\left(x\right)$ ומצד שני $f\left(a\right)\ge f\left(x\right)$ כפי שראינו.

מכאן שפעולת ההחלפה בין a,x יכלה רק להקטין את משקל העץ ומכיוון ש-T_{opt} היה אופטימלי, משקלם שווה וגם T'_{opt} אופטימלי.

כעת נחליף את y,b באותו אופן ונקבל בעזרת אותו חישוב שהעץ החדש שקיבלנו הוא אופטימלי. עץ חדש זה הוא גם בעל התכונה ש-x,y הם אחים בו, כמבוקש.

6 רשתות זרימה

6.1 מבוא והגדרות

רשתות זרימה משמשות אותנו למידול סיטואציות שבהן קיים "חומר" כלשהו שיש להעביר בין שתי נקודות - מנקודת **המקור** אל נקודת **היעד**, כאשר יש מסלולים שונים להעברת החומר וניתן לפצל אותו ביניהם. במסלולים קיימים אילוצי **קיבולת** שמגבילים את כמות החומר שניתן להעביר בכל מסלול, והבעיה המרכזית שבה נעסוק היא מציאת הדרך האופטימלית לפצל את החומר בין מסלולים שונים כך שכמות החומר שעוברת מהמקור אל היעד ביחידת זמן היא אופטימלית.

אפשר להשתמש ברשתות זרימה כדי למדל סיטואציות רבות - זרימת נוזלים, זרימה חשמלית, העברת מידע ברשת האינטרנט, תנועת רכבים וכדומה, אך לא נעסוק בשימושים אלו כאן אלא רק במידול האבסטרקטי של הבעיה.

הגדרה 6.1 רשת זרימה היא גרף מכוון $G=\left(V,E\right)$ כך שקיימים שני צמתים מובחנים בגרף, $s,t\in V$ הנקראים **מקור** ו**יעד** בהתאמה וקיימת פונקציה $c:V\times V\to\mathbb{R}$ הנקראת **פונקציית קיבול** כך שמתקיים:

- כל צומת $v\in V$ נמצא על מסלול $s\rightsquigarrow v\rightsquigarrow t$ מהמקור אל היעד.
- אם קיימת קשת $\left(u,v\right)\in E$ אז לא קיימת קשת בכיוון ההפוך: $\left(v,u\right)\notin E$. בנוסף לא קיימות לולאות עצמיות, כלומר $\left(u,u\right)\notin E$.

- לכל $u, v \in V \times V$ מתקיים $c(u,v) \geq 0$ ואם $(u,v) \notin E$ אז $c(u,v) = 0$.

רשת הזרימה מגדירה את הקלט לבעיה שלנו. ה"פתרון" לבעיה הזו היא זרימה שמתאימה לרשת:

הגדרה 6.2 תהא $G = (V, E)$ רשת זרימה עם פונקציית קיבול c. **זרימה** $f : V \times V \to \mathbb{R}$ היא פונקציה המקיימת:

- **אילוץ הקיבול**: $0 \leq f(u,v) \leq c(u,v)$ לכל $u, v \in V$.
- **שימור הזרימה**: לכל $u \in V \backslash \{s,t\}$ מתקיים שהזרימה שנכנסת ל־u שווה לזרימה שיוצאת ממנו, כלומר $\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v)$

לבסוף, מכיוון שאנו רוצים לפתור בעיית אופטימיזציה, אנחנו צריכים להגדיר את פונקציית המטרה שלנו ־ הערך שאנחנו מנסים במקרה זה למקסם:

הגדרה 6.3 תהא f זרימה עבור רשת זרימה $G = (V, E)$. אז נגדיר את **ערך הזרימה** $|f|$ בתור כמות הזרימה שיוצאת מ־s פחות כמות הזרימה שנכנסת ל־s: $|f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$

אינטואיטיבית נראה שאל s לא צריכות להיכנס קשתות כלל ואין הגיון בהכנסת זרימה אל s; בהמשך נראה שמועיל לאפשר זאת בתור כלי עזר עבור אלגוריתמים שמוצאים זרימה מקסימלית.

6.2 שיטת פורד־פולקרסון

הבעיה שאיתה אנו מתמודדים היא זו: בהינתן רשת זרימה, למצוא זרימה עבורה כך שערך הזרימה הוא מקסימלי. דרך אחת להתמודד עם הבעיה היא **שיטת פורד־פולקרסון**, שהיא שיטה כללית למציאת זרימה מקסימלית בגרף שיש לה מימושים שונים רבים, בדומה לאלגוריתם הגנרי למציאת עץ פורש מינימלי.

השיטה נסמכת על שני מושגים:

- **רשת שיורית**: בהינתן רשת זרימה G וזרימה f עבורה, הרשת השיורית G_f היא רשת (שאינה עונה על כל הדרישות מרשת זרימה) שבה קיבולי הקשתות מתארות במובן מסויים עד כמה ניתן **להגדיל** או **להקטין** את זרימת f ברשת המקורית.
- **מסלול שיפור**: הוא מסלול מהמקור אל היעד בתוך רשת שיורית G_f, שאפשר להסיק ממנו שינויים בזרימה f **שישפרו** את ערך הזרימה של f.

שיטת פורד־פולקרסון כעת מתוארת בצורה הכללית מאוד הבאה:

1. אתחלו זרימה $f = 0$.
2. כל עוד ברשת השיורית G_f קיים מסלול שיפור p:
   (א) שפרו את f על פי p.
3. החזירו את f.

כדי שהשיטה לעיל תהיה בעלת משמעות עבורנו עלינו להגדיר במדויק רשת שיורית ומסלול שיפור, ולהציע אלגוריתם קונקרטי שמוצא מסלול שיפור ברשת השיורית. נפתח עם ההגדרות.

הרעיון מאחורי **רשת שיורית** G_f הוא לבנות רשת שמייצגת שני פרטים: כמה ניתן עוד **להגדיל** זרימה על קשת ב־G (זה מתבטא בהפרש בין הקיבול של הקשת והזרימה הנוכחית) וכמה ניתן **להקטין** זרימה על קשת (זה מתבטא בגודל הזרימה הנוכחי).

הגדרה 6.4 תהא G רשת זרימה עם פונקציית קיבול c ו־f זרימה ברשת. נגדיר פונקציית קיבול חדשה, **הקיבול השיורי** $c_f : V \times V \to \mathbb{R}$, באופן הבא:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(v,u) & (v,u) \in E \\ 0 & \text{else} \end{cases}$$

נגדיר כעת $E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$, ונגדיר את הרשת השיורית G_f בתור הגרף $G_f = (V, E_f)$ עם פונקציית הקיבול c_f.

נשים לב לכך ש־c_f מוגדרת היטב, שכן כפי שהנחנו, ברשת זרימה E אם קיימת הקשת $(u,v)\in E$ אז $(v,u)\notin E$ ולכן שני המקרים הראשונים בהגדרת c_f אינם יכולים להתקיים בו־זמנית. עם זאת, ברשת השיורית G_f עצמה **יכולות** להופיע קשתות בשני הכיוונים ולכן זוהי אינה רשת זרימה בעצמה אך פרט להבדל זה רשת שיורית זהה לרשת זרימה, ובפרט ניתן להגדיר זרימה גם עבורה באותו אופן שבו הגדרנו עבור רשת זרימה.

המטרה של הרשת השיורית היא להצביע על שינויים אפשריים בזרימה f. זאת נעשה באמצעות פונקציית זרימה f' עבור הרשת השיורית, שמשפיעה על הזרימה המקורית באופן הבא:

הגדרה 6.5 תהא f זרימה ברשת זרימה G ו־f' זרימה ברשת השיורית G_f. נגדיר את **השיפור** $f\uparrow f'$ של f באמצעות f' על ידי

$$f\uparrow f'(u,v)\triangleq\begin{cases}f(u,v)+f'(u,v)-f'(v,u) & (u,v)\in E\\ 0 & \text{else}\end{cases}$$

ההגדרה הזו תופסת את המהות של הזרימה השיורית מ־u אל v כמתארת **הגדלה** של הזרימה המקורית ושל הזרימה השיורית מ־v אל u כמתארת **הקטנה** של הזרימה המקורית. האילוץ שלנו שלא יתקיים $(u,v)\in E$ וגם $(v,u)\in E$ מונע "בלבול בתפקידים" בין שני סוגי הזרימות השיוריות הללו.

לא מובן מאליו שההגדרה "עובדת" ־ יש להוכיח זאת:

משפט 6.6 $f\uparrow f'$ היא זרימה ברשת G ו־$|f\uparrow f'|=|f|+|f'|$

הוכחה: ■

ראינו כי כל זרימה ברשת השיורית G_f מאפשרת לנו "לתקן" את הזרימה f ברשת המקורית G. נדבר כעת על סוג ספציפי של זרימה ברשת השיורית ־ כזו שנובעת מ**מסלול שיפור**.

הגדרה 6.7 תהא G רשת זרימה עם זרימה f. מסלול מ־s אל t ברשת השיורית G_f נקרא **מסלול שיפור**.

מסלול השיפור לכשעצמו לא נותן לנו זרימה ־ הרעיון הוא להגדיל את הזרימה ככל הניתן לאורך המסלול הזה. מה שמגביל אותנו הוא "צוואר הבקבוק" של המסלול ־ הקשת בעלת הקיבולת המינימלית שעליו:

הגדרה 6.8 בהינתן מסלול שיפור p ב־G_f נגדיר את **הקיבול השיורי** של מסלול השיפור בתור $c_f(p)\triangleq\min\{c_f(u,v)\mid (u,v)\in p\}$.

כעת ניתן להשתמש במושג זה כדי להגדיר את הזרימה שאנו יוצרים מתוך מסלול שיפור:

טענה 6.9 תהא G_f רשת שיורית עם מסלול שיפור p. נגדיר פונקציה $f':V\times V\to\mathbb{R}$ על ידי $f_p(u,v)=\begin{cases}c_f(p) & (u,v)\in p\\ 0 & \text{else}\end{cases}$

אז f_p היא זרימה ברשת השיורית G_f ו־$|f_p|=c_f(p)$.

הוכחה: ■

נשים לב לכך שהקיבול השיורי של מסלול שיפור הוא תמיד מספר חיובי (גדול מאפס) כי מלכתחילה הגדרנו את קשתות הרשת השיורית בתור הקשתות שהקיבול שלהן חיובי. מכאן נקבל:

מסקנה 6.10 אם p מסלול שיפור ברשת השיורית G_f אז $|f\uparrow f_p|=|f|+|f_p|>|f|$

במילים אחרות ־ מסלול שיפור אכן מאפשר לנו לשפר את f.

כעת סיימנו להגדיר במפורש את כל המרכיבים של שיטת פורד־פולקרסון ־ עדיין נותר להבין מדוע היא עובדת (מחזירה את הזרימה המקסימלית ברשת) וכיצד ניתן למצוא מסלול שיפור ב־G_f בפועל.

6.3 משפט חתך־מינימלי־זרימה־מקסימלית

אלגוריתם פורד־פולקרסון מסתיים כאשר הוא מגיעה לזרימה f שעבורה ברשת השיורית G_f לא קיימים יותר מסלולי שיפור. אנו רוצים להוכיח כי בשלב זה, f היא זרימה מקסימלית (לא קיימת ב־G זרימה עם ערך גבוה יותר). כדי להוכיח זאת, נוכיח שזרימה היא מקסימלית אם ורק אם ערכה שווה ל**ערך הקיבול של חתך** ברשת. לצורך כך נצטרך להגדיר את מושג ה**חתך** והקיבול שלו.

כשעסקנו בעצים פורשים, **חתך** היה חלוקה של צמתי הגרף לשתי קבוצות. כאן ההגדרה תהיה דומה עם התוספת לפיה החתך חייב להפריד בין s ו־t:

הגדרה 6.11 תהא G רשת זרימה. **חתך** של G הוא זוג קבוצות (S,T) כך ש־$s \in S, t \in T$ ו־$V = S \cup T$.
קיבול של חתך הוא סכום הקיבולים של הקשתות שחוצות את החתך מ־S אל T (**ללא** הקשתות בכיוון ההפוך):

$$c\left(S,T\right)=\sum_{u\in S}\sum_{v\in T}c\left(u,v\right)$$

חתך מינימלי ב־G הוא חתך שערך הקיבול שלו מינימלי.
תהא f זרימה ב־G. **הזרימה הכוללת** של f מ־S אל T היא:

$$\sum_{u\in S}\sum_{v\in T}f\left(u,v\right)-\sum_{u\in S}\sum_{v\in T}f\left(v,u\right)$$

הזרימה הכוללת של f מ־S אל T, עבור כל חתך ב־G, שווה ל־$|f|$. האינטואיציה היא ש"אין לזרימה לאן לברוח". המקום היחיד שאליו הזרימה שיוצאת מ־s יכולה להתנקז הוא t (כי זרימה שמתנקזת אל s לא נספרת כחלק מ־$|f|$) ולכן היא תהיה חייבת לעבור מ־S אל T, בלי תלות בחתך. מכיוון שהזרימה הכוללת מ־S אל T חסומה על ידי קיבול החתך, המסקנה היא ש־$|f|$ חסום על ידי קיבול החתך (S,T), וזאת **לכל** חתך של רשת הזרימה. מכאן שכשהערך $|f|$ יהיה **מקסימלי** הוא יהיה שווה לכל היותר לערך **המינימלי** של קיבול חתך כלשהו של G.

פורמלית:

משפט 6.12 יהא (S,T) חתך של רשת הזרימה G ו־f זרימה כלשהי. אז $|f|$ שווה לזרימה הכוללת של f מ־S אל T.

הוכחה: ■

מסקנה 6.13 לכל זרימה f ב־G ולכל חתך (S,T) של G מתקיים $|f| \le c\left(S,T\right)$

הוכחה: ישירות מההגדרה ובהסתמך על המשפט הקודם:

$$\begin{aligned}|f| &= \sum_{u\in S}\sum_{v\in T}f\left(u,v\right)-\sum_{u\in S}\sum_{v\in T}f\left(v,u\right)\\ &\le \sum_{u\in S}\sum_{v\in T}f\left(u,v\right)\\ &\le \sum_{u\in S}\sum_{v\in T}c\left(u,v\right)\\ &= c\left(S,T\right)\end{aligned}$$

■

כעת נוכל להוכיח את המשפט המרכזי שלנו:

משפט 6.14 ("חתך־מינימלי־זרימה־מקסימלית") תהא G רשת זרימה עם זרימה f. התנאים הבאים שקולים:

1. f היא זרימה מקסימלית ב־G.
2. ברשת השיורית G_f לא קיים מסלול שיפור.
3. $|f| = c(S,T)$ עבור חתך (S,T) כלשהו של G.

הוכחה: נוכיח על ידי שרשרת הגרירות $1 \Rightarrow 2 \Rightarrow 3 \Rightarrow 1$

$1 \Rightarrow 2$:

אם ב־G_f קיים מסלול שיפור p אז כפי שראינו במסקנה 6.10, הזרימה $f \uparrow f_p$ ב־G מקיימת $|f \uparrow f_p| > |f|$, בסתירה למקסימליות f. מכאן שלא קיים מסלול שיפור ב־G_f.

$2 \Rightarrow 3$:

נגדיר חתך של G באופן הבא: S תכלול את כל הצמתים ב־V שקיימים מסלול אליהם מ־s בגרף השיורי G_f. נגדיר $T = V \backslash S$. מכיוון שלא קיים מסלול שיפור ב־G_f, בפרט $t \notin S$ ולכן $t \in T$ ומכאן ש־(S,T) הוא חתך.

יהיו $u \in S, v \in T$ כלשהם. נפריד בין שלושה מקרים:

אם $(u,v) \in E$ אז הקיבול השיורי של קשת זו הוא $c_f(u,v) = c(u,v) - f(u,v)$. אם $f(u,v) < c(u,v)$ אז $c_f(u,v) > 0$ ולכן הקשת (u,v) תהיה שייכת ל־G_f ומכאן שקיים מסלול מ־s אל v (המסלול אל u והצעד הנוסף אל v דרך הקשת (u,v)), בסתירה לשייכות v אל T. מכאן ש־$f(u,v) = c(u,v)$ במקרה זה, ואילו $f(v,u) = 0$.

אם $(u,v) \notin E$ אז $c(u,v) = 0$ ולכן $f(u,v) = 0 = c(u,v)$. נעבור לבדוק מה קורה במקרה זה ל־$f(v,u)$ ונפצל לשני מקרים:

אם $(v,u) \in E$ אז $c_f(u,v) = f(v,u)$ ולכן אם $f(v,u) \neq 0$ קיימת הקשת (u,v) ב־G_f ונגיע לסתירה כמו קודם. מכאן ש־$f(v,u) = 0$.

אם $(v,u) \notin E$ אז בגלל שגם $(u,v) \notin E$ נקבל $c_f(v,u) = 0$, כלומר $f(v,u) = 0$ גם במקרה זה.

נסכם: בכל שלושת המקרים קיבלנו ש־$f(u,v) = c(u,v)$ ואילו $f(v,u) = 0$, ומכאן

$$\begin{aligned}
|f| &= \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) \\
&= \sum_{u \in S} \sum_{v \in T} c(u,v) - \sum_{u \in S} \sum_{v \in T} 0 \\
&= c(S,T)
\end{aligned}$$

$3 \Rightarrow 1$:

כפי שראינו במסקנה 6.13, לכל זרימה ב־G ערכה חסום מלעיל על ידי $c(S,T)$. מכיוון ש־$|f| = c(S,T)$ זוהי זרימה בעלת ערך מקסימלי. ■

6.4 אלגוריתם אדמונדס־קארפ

אלגוריתם אדמונדס־קארפ הוא מימוש יעיל של שיטת פורד־פולקרסון. האלגוריתם מתחזק זרימה שמאותחלת להיות 0 על כל קשתות הרשת, ובכל איטרציה מבצע הרצת BFS על הגרף השיורי G_f למציאת מסלול מ־s אל t משפר את f בהתאם למסלול זה, ומסיים כאשר לא קיים מסלול כזה בגרף (מה שמוכיח את מקסימליות f בעזרת משפט חתך־מינימלי־זרימה־מקסימלית). כזכור, BFS מחזיר את המסלול **הקצר ביותר**, ונראה כי זה מבטיח התכנסות מהירה של האלגוריתם.

נציג מימוש של האלגוריתם בשפת Python:

```
def edmonds_karp(flow_network):
    G,s,t,c = flow_network
    f = {(u,v): 0 for (u,v) in G.E}
    max_found = False
    while not max_found:
        p = find_augmenting_path(flow_network, f)
        if p is None:
```

```
8                 max_found = True
9             else:
10                f = augment_path(flow_network, f, p)
11        return f
```

```
1   def find_augmenting_path(flow_network, f):
2       G,s,t,c = flow_network
3       E_f, c_f = residual_network(flow_network, f)
4       G_f = Graph(G.V, E_f, directed=True)
5       return BFS_shortest_path(G_f, s, t)
```

```
1   def augment_path(flow_network, f, p):
2       G,s,t,c = flow_network
3       E_f, c_f = residual_network(flow_network, f)
4       path_edges = [(p[i], p[i+1]) for i in range(len(p) - 1)]
5       min_c_p = min([c_f[(u,v)] for (u,v) in path_edges])
6       for (u,v) in path_edges:
7           if (u,v) in G.E:
8               f[(u,v)] = f[(u,v)] + min_c_p
9           else:
10              f[(v,u)] = f[(v,u)] - min_c_p
11      return f
```

```
1   def residual_network(flow_network, f):
2       G,s,t,c = flow_network
3       c_f = {}
4       for (u,v) in G.E:
5           c_f[(u,v)] = c[(u,v)] - f[(u,v)]
6           c_f[(v,u)] = f[(u,v)]
7       E_f = [(u,v) for (u,v) in product(G.V, G.V) if c_f.get((u,v), 0) > 0]
8       return (E_f,c_f)
```

אדמונדס־קארפ מאתחל את f להיות 0 לכל קשת בשורה 3. בשורה 6 נקראת הפרוצדורה שמחפש מסלול שיפור, ואם נמצא כזה f משופרת בשורה 10. אם לא נמצא כזה, האלגוריתם מסתיים.

מציאת מסלול שיפור מתבצעת באמצעות בניה של הגרף השיורי והרצת BFS רגיל עליו החל מ־s, תוך בניית המסלול מ־s אל t על ידי מעבר על הסדרה $t.\pi, t.\pi.\pi, \ldots$ עד להגעה אל s.

שיפור f מתבצע על ידי חישוב $c_f(p)$ (הקיבול המינימלי בגרף השיורי על מסלול השיפור) והגדלת/הקטנת f בהתאם.

לבסוף, הרשת השיורית עצמה נבנית ישירות מההגדרה שהצגנו קודם עבור $c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & (u,v) \in E \\ f(u,v) & (v,u) \in E \end{cases}$ ו־$E_f = \{(u,v) \mid c_f(u,v) > 0\}$.

נעבור להוכחת נכונות האלגוריתם. על פניו הנכונות של שיטת פורד־פולקרסון ברורה, כי ראינו שהתנאי "אין מסלול שיפור" שקול לתנאי "הזרימה מקסימלית", אולם ההנחה המובלעת שלנו הייתה ששיטת פורד־פולקרסון בהכרח מסתיימת מתישהו, מה שעלול לא לקרות כלל אם קיבולות הקשתות אינן מספרים טבעיים ־ הזרימה אמנם תגדל בכל איטרציה, אבל בצורה שלא מתכנסת אל הזרימה המקסימלית.

למרבה המזל, באלגוריתם אדמונדס־קארפ הגדלת הזרימה מתקשרת בצורה חזקה אל **אורך המסלולים הקצרים ביותר** בגרף השיורי, מה שמאפשר לנו לחסום את מספר האיטרציות.

טענה 6.15 בכל איטרציה של אדמונדס־קארפ על רשת זרימה f ולכל $v \notin \{s,t\}$, המרחק המינימלי $\delta_f(s,v)$ של v מ־s ברשת השיורית עולה מונוטונית (כלומר, אחרי איטרציה ערך זה אינו יכול לרדת).

הוכחה: נניח בשלילה ש־$v \notin \{s,t\}$ הוא צומת כך שבשלב מסויים של אדמונדס־קארפ, שיפור f מוביל להקטנת $\delta_f(s,v)$.

נסמן ב־f את הזרימה לפני ההגדלה הראשונה שמובילה להקטנה כזו וב־f' את הזרימה לאחריה, כלומר $\delta_{f'}(s,v) < \delta_f(s,v)$. בלי הגבלת הכלליות, מבין כל הצמתים ששיפור f מקטין את δ_f עבורם נבחר את v להיות זה שעבורו $\delta_{f'}(s,v)$ הוא מינימלי. נסמן ב־$p = s \rightsquigarrow u \rightarrow v$ מסלול קצר ביותר מ־s אל v, כלומר $(u,v) \in E_{f'}$ ו־$\delta_{f'}(s,u) = \delta_{f'}(s,v) - 1$.

מכיוון שבחרנו את v להיות בעל ערך $\delta_{f'}(s,v)$ מינימלי מבין כל הצמתים ש־δ_f שלהם מוקטן, u אינו צומת כזה. כלומר $\delta_f(s,u) \leq \delta_{f'}(s,u)$.

כעת נראה כי $(u,v) \notin E_f$, שכן אחרת היינו מגיעים לסתירה הבאה:

$$\begin{aligned}\delta_f(s,v) &\leq \delta_f(s,u) + 1\\ &\leq \delta_{f'}(s,u) + 1\\ &= \delta_{f'}(s,v)\end{aligned}$$

כלומר, הקשת $(u,v) \in E_f$ הייתה מבטיחה את הגדלת $\delta_f(s,v)$, בסתירה להנחה שלנו.

אם כן, במעבר מ־E_f אל $E_{f'}$ הקשת (u,v) **התווספה** אל הגרף. אם $(u,v) \in E$ זה יכול לקרות אם השיפור **מקטין** את $f(u,v)$ (כדי שיפסיק להתקיים $c(u,v) - f(u,v) = 0$), כלומר אם הזרימה השיורית במסלול השיפור p היא מ־v אל u. אם $(u,v) \notin E$ אז השיפור צריך **להגדיל** את $f(v,u)$ (כדי שיפסיק להתקיים $f(v,u) = 0$) ־ כלומר גם במקרה זה הזרימה השיורית היא מ־v אל u.

כעת נשתמש באופן שבו אלגוריתם אדמונדס־קארפ עובד: מסלול השיפור שהוא מוצא הוא תמיד המסלול הקצר ביותר מ־s אל t, כך שהוא גם המסלול הקצר ביותר מ־s אל כל צומת שעל המסלול. בפרט המסלול מ־s אל u הוא כזה, כלומר הקשת (v,u) היא הקשת האחרונה במסלול הקצר ביותר מ־s אל u ב־G_f. מכאן נסיק:

$$\begin{aligned}\delta_f(s,v) &= \delta_f(s,u) - 1\\ &\leq \delta_{f'}(s,u) - 1\\ &= \delta_{f'}(s,v) - 2\end{aligned}$$

וזאת בסתירה להנחה המקורית שלנו ש־$\delta_{f'}(s,v) < \delta_f(s,v)$. מכאן שלא קיים צומת v שהאיטרציה של אדמונדס־קארפ מקטינה את המרחק המינימלי עבורו. ■

בעזרת טענת העזר נוכל להוכיח את הסיבוכיות (ולכן גם את הנכונות) של אדמונדס־קארפ:

משפט 6.16 סיבוכיות זמן הריצה של אלגוריתם אדמונדס־קארפ על רשת זרימה $G = (V,E)$ היא $O(|V||E|)$.

הוכחה: נאמר שקשת (u,v) ברשת שיורית G_f היא **קריטית** על מסלול שיפור p אם היא שייכת למסלול והקיבול שלה שווה לקיבול המסלול, כלומר $c_f(p) = c_f(u,v)$. בכל מסלול שיפור קיימת קשת שכזו (כי קיבול המסלול נקבע על ידי המינימום מבין קיבולי הקשתות שעליו) ואחרי שיפור הזרימה בהתאם למסלול השיפור, הקשת תיעלם מהרשת השיורית (כי אם $(u,v) \in E$ אז השיפור גרם ל־f להגיע למלוא הקיבולת של (u,v) ואילו אם $(v,u) \in E$ אז השיפור איפס את f ולכן ביטל את הקשת השיורית בכיוון השני).

נוכיח כי במהלך ריצת אדמונדס־קארפ, כל קשת עשויה להפוך לקריטית לכל היותר $\frac{|V|}{2} + 1$ פעמים. מכיוון שבכל איטרציה קשת כלשהי חייבת להפוך לקריטית, ובגרף השיורי מספר הקשתות הוא $O(|E|)$ (כי הוא שווה למספר הקשתות ב־G ואולי גם חלק מהקשתות בכיוון ההפוך, כתלות ב־f) הרי שמספר האיטרציות חסום על ידי $O(|V||E|)$.

יהיו $u,v \in V$ צמתים שמחוברים ביניהם בקשת (ולכן יכולה להיות קשת ביניהם בשני הכיוונים בגרף השיורי). בפעם הראשונה שבה הקשת (u,v) היא קריטית, מכיוון שהיא נחה על מסלול שיפור ובאדמונדס־קארפ מסלול שיפור הוא מסלול קצר ביותר, הרי ש־

$$\delta_f(s,v) = \delta_f(s,u) + 1$$

אחרי ביצוע השיפור הקשת (u,v) מוסרת מהגרף השיורי. על מנת להפוך שוב לקשת קריטית, היא צריכה להתווסף בהמשך. זה יכול לקרות רק אם יהיה בהמשך מסלול שיפור שבו מופיעה הקשת בכיוון הנגדי, (v,u) (לאו דווקא מופיעה בתור קשת קריטית).

תהא f' הזרימה ב־G כאשר מופיע מסלול השיפור הזה. אז מאותו נימוק על מסלולים קצרים, נקבל

$$\delta_{f'}(s,u) = \delta_{f'}(s,v) + 1$$

כעת נשתמש בטענה 6.15 שהוכחנו קודם ולפיה בכל איטרציה של אדמונדס־קארפ המרחק המינימלי של כל צומת מ־s לא יכול לקטון. כלומר

$$\delta_f(s,v) \le \delta_{f'}(s,v)$$

ומכאן נסיק:

$$\begin{aligned}\delta_{f'}(s,u) &= \delta_{f'}(s,v) + 1\\ &\ge \delta_f(s,v) + 1\\ &= \delta_f(s,u) + 2\end{aligned}$$

כלומר, בין שתי נקודות הזמן שבהן (u,v) הופכת לקריטית, המרחק של u מ־s גדל לפחות ב־2.

כעת, המרחק המקסימלי של u מ־s הוא חסום על ידי $|V|$, ולכן מספר הפעמים הנוספות שבהן (u,v) עשויה להפוך לקריטית חסום על ידי $\frac{|V|}{2}$ ובסך הכל מספר הפעמים שבהן הקשת עשויה להפוך לקריטית חסום על ידי $\frac{|V|}{2}+1$ כנדרש. זה מסיים את ההוכחה. ■

6.5 מציאת שידוך מקסימום בגרף דו־צדדי

נציג כעת שימוש של רשתות זרימה כדי לפתור את הבעיה הקומבינטורית של מציאת **שידוך מקסימום בגרף דו צדדי**. שימוש זה הוא דוגמא ל**רדוקציה** ־ האופן שבו ממירים את הקלט של בעיה אחת בקלט של בעיה אחרת, כך שפתרון לבעיה האחרת מתורגם ישירות לפתרון של הבעיה המקורית.

נפתח בחזרה על ההגדרות הרלוונטיות מתורת הגרפים:

הגדרה 6.17 יהא $G=(V,E)$ גרף לא מכוון. **שידוך** ב־G הוא תת־קבוצה $M\subseteq E$ של קשתות כך שלכל $v\in V$, הוא מופיע לכל היותר באחת מהקשתות ב־M. **שידוך מקסימום** ב־G הוא שידוך ב־G כך שלכל שידוך M' ב־G מתקיים $|M'|\le|M|$.

המושג מכונה "שידוך" שכן הוא יוצר זוגות מבין צמתי הגרף (כל זוג הוא נקודות הקצה של אחת מהקשתות בשידוך) תוך שמירה על הקריטריון שהשידוך הוא "מונוגמי" ־ אין צומת שמשודך לשני צמתים שונים.

לשידוכים יש שימושים רבים, כששימוש בולט אחד הוא ב**הקצאת משאבים**: למשל, אם יש לנו אוסף L של משימות חישוביות ואוסף R של מחשבים שניתן להריץ משימות חישוביות עליהם, כשיש קשת בין משימה מ־L לכל מחשב ב־R שמסוגל להתמודד איתה, ואנו מעוניינים להקצות מחשבים לפתרון כמות גדולה ככל הניתן של משימות חישוביות.

בדוגמא זו ובדוגמאות דומות לה יש בגרף שלנו שתי קבוצות מובחנות של צמתים, כשיש קשת רק בין איברי קבוצה אחת לשניה ולא בתוך הקבוצות עצמן. סוג מיוחד זה של גרפים זוכה לשם משל עצמו:

הגדרה 6.18 גרף $G=(V,E)$ נקרא **דו־צדדי** אם $V=L\cup R$ כך ש־$L\cap R=\emptyset$ ולכל $(u,v)\in E$ מתקיים $u\in L, v\in R$ או ההפך, $u\in R, v\in L$.

ניתן לבדוק האם גרף הוא דו־צדדי באמצעות הרצת אלגוריתם BFS; אנו נניח כאן כי הגרפים כבר נתונים עם תיאור מפורש של הקבוצות L,R.

נרצה לפתור את הבעיה הבאה: בהינתן קלט של גרף דו־צדדי $G=(L\cup R,E)$, למצוא שידוך מקסימום M עבור G. נפתור בעיה זו באמצעות **רדוקציה** לבעיית מציאת זרימה מקסימלית:

מתוך G נבנה רשת זרימה $G'=(V',E')$ כך ש־$V'=L\cup R\cup\{s,t\}$ עבור צמתים חדשים s,t, וקשתות

$$E' = \{(s,u)\ \mid u\in L\}\cup\{(u,v)\ \mid\ (u,v)\in E\}\cup\{(v,t)\ \mid v\in R\}$$

דהיינו, אנחנו מוסיפים קשת מ־s לכל צמתי L, קשת מכל צמתי R אל t, ומכוונים את כל הקשתות ב־E כך שילכו מ־L אל R.

לבסוף, נגדיר פונקציית קיבול $c: E' \to \mathbb{R}$ על ידי $c(e)=1$ לכל $e \in E'$.

בהינתן זרימה f עבור G', נגדיר באמצעותה שידוך באופן הבא: $M_f = \{(u,v) \in L \times R \mid f(u,v)=1\}$.

טענה 6.19 M_f הוא שידוך ב־G.

הוכחה: ראשית נשים לב לכך ש־$(u,v) \in E$ אחרת היה מתקיים $c(u,v)=0$ ואז $f(u,v)=1$ היה מפר את אילוץ הקיבול.
נניח כעת בשלילה שקיימים $v_1 \neq v_2 \in R$ כך ש־$(u,v_1) \in M_f$ וגם $(u,v_2) \in M_f$. אז $\sum_{v \in V} f(u,v) \geq 2$ אבל $\sum_{v \in V} f(v,u) = f(s,u) \leq 1$ כי הקשת הנכנסת היחידה אל u היא מ־s. מכאן שאילוץ שימור הזרימה מופר. לכן לא קיימים v_1, v_2 כנ"ל. בדומה מטופל גם המקרה של התנגשות עבור צומת ב־R. ■

ראינו שמכל זרימה f ניתן לקבל שידוך M_f. גם ההפך נכון: אם M שידוך, אפשר לקבל ממנו זרימה על ידי העברת 1 בכל קשת השייכת לשידוך, תוך הזרמת 1 מ־s ואל t מהצמתים שמופיעים בשידוך. בצורה זו מתקבלת זרימה f_M כך ש־$|f_M| = |M|$.

מה שטרם הוכחנו הוא את ההפך: שבמקרה שבו בונים שידוך מתוך זרימה, $|M_f| = |f|$. אם נוכיח זאת נסיים באופן הבא: אם f היא זרימה מקסימלית אז נובע מכך ש־M_f הוא שידוך מקסימום שכן אם היה קיים שידוך M עם $|M_f| < |M|$ היינו מקבלים סתירה למקסימליות f: $|f| = |M_f| < |M| = |f_M|$.

הקושי היחיד שלנו הוא בהוכחה ש־$|M_f| = |f|$. הבעיה עשויה לנבוע מכך שבנוסף לזוגות (u,v) שעבורם $f(u,v)=1$ יהיו עוד זוגות שדרכם מועברת זרימה **שאינה מספר שלם** ומאפשרת הגדלה נוספת של הזרימה, כך שמתקבל $|M_f| < |f|$. . בחינה מדוקדקת של אלגוריתם אדמונדס־קארפ מעלה שדבר כזה לא יכול להתרחש בו במקרה (כמו זה שלנו) שבו הגרף כולל רק קיבולים שהם מספרים שלמים ־ במקרה כזה, כל השינויים בזרימה יהיו עם ערכים שהם מספר שלם.

7 אלגוריתמים בסיסיים בתורת המספרים

7.1 הסיבוכיות של אלגוריתמים על מספרים

מספר טבעי הוא **ראשוני** אם הוא מתחלק רק בעצמו וב־1. אם $n = ab$ אז לא ייתכן ש־$a > \sqrt{n}$ וגם $b > \sqrt{n}$ כי אז היינו מקבלים $n = ab > n$ וזו סתירה. לכן אם מחפשים ל־n פירוק מהצורה $n = ab$ מספיק לבדוק רק **עד השורש** של n. זה מניב את אלגוריתם "עד השורש" לבדיקת ראשוניות:

```
def is_prime(n):
    for a in range(2, math.ceil(math.sqrt(n) + 1)):
        if n % a == 0:
            return False
    return True
```

האלגוריתם עונה נכון לכל $n \geq 2$, וסיבוכיות זמן הריצה שלו היא $O(\sqrt{n})$ / לכאורה זהו זמן ריצה טוב, אך בפועל זהו אלגוריתם גרוע ביותר מבחינת סיבוכיות, שאין טעם להפעיל אלא עבור קלטים קטנים.

הסיבה לכך היא שסיבוכיות של אלגוריתמים שפועלים על מספרים נמדדת על פי רוב ביחס **לגודל הייצוג** של המספר ־ למספר הספרות שלו. בכל בסיס שאינו אונרי, מספר הספרות של n הוא מסדר גודל של $\ln(n)$, ולכן סיבוכיות האלגוריתם נמדדת ביחס לגודל זה. כך למשל הסיבוכיות של אלגוריתם **חיבור ארוך** היא $O(\log n)$, והסיבוכיות של אלגוריתם כפל ארוך היא $O(\log^3 n)$. לעומת זאת, הסיבוכיות של אלגוריתם "עד השורש" היא $O\left(\left(\sqrt{2}\right)^{\lg n}\right) = O\left(\sqrt{2^{\lg n}}\right) = O(\sqrt{n})$ ־ אקספוננציאלית ב־$\lg n$.

אלגוריתם בדיקת ראשוניות שהוא פולינומי ב־$\lg n$ היה אתגר מהותי עבור מדעי המחשב. בהמשך נראה אלגוריתם **הסתברותי** לבדיקת ראשוניות שפועל בזמן פולינומי; אלגוריתם **דטרמיניסטי** כזה התגלה רק ב־2002 (אלגוריתם AKS) ולא נציג אותו כאן.

לצורך פשטות, נמדוד סיבוכיות של אלגוריתמים על מספרים באמצעות ספירת **פעולות אטומיות** שכולן ניתנות למימוש בזמן יעיל (פולינומי בגודל הייצוג עם פולינום קטן) ־ חיבור, חיסור, כפל ו**חילוק עם שארית**. נזכיר את האחרון:

בהינתן זוג מספרים טבעיים a, b ניתן למצוא q, r טבעיים כך ש־$a = qb + r$ כך ש־$0 \leq r < b$.

אפשר להכליל את מעט את המשפט אם מרשים ל־a, q להיות **שלמים** (כלומר, לאפשר גם שליליים).

7.2 אלגוריתם למציאת המחלק המשותף המקסימלי

7.2.1 הגדרה והאלגוריתם הבסיסי

המחלק המשותף המקסימלי (gcd, ראשי תיבות של greatest common divisor) של זוג מספרים טבעיים a, b הוא המספר הטבעי d המקסימלי כך ש־$d|a$ וגם $d|b$ (לכל a, b לכל הפחות $d = 1$ מקיים תכונה זו כך שה־gcd קיים תמיד). נסמן $d = \gcd(a, b)$ ולרוב כשזה יהיה ברור מההקשר נשמיט את ה־gcd ונכתוב פשוט (a, b). קיום אלגוריתם יעיל לחישוב ה־gcd של שני מספרים היה ידוע כבר לאוקלידס, והיעילות שלו היא הבסיס לרוב התוצאות שנראה בהמשך.

לצורך חישוב ה־gcd, נשים לב לאבחנה הבאה:

טענה 7.1 אם $a = qb + r$ אז $(a, b) = (b, r)$

הוכחה: נסמן $d = (a, b)$ ו־$d' = (b, r)$. מכך ש־$d|a$ ו־$d|b$ נובע שהוא מחלק כל צירוף לינארי שלהם, בפרט $d|a - qb = r$. מכיוון ש־d' הוא המקסימלי שמחלק את b, r אז $d \le d'$. באופן דומה מוכיחים ש־$d' \le d$, ושני אלו מראים ש־$d = d'$. ■

אבחנה זו מניבה אלגוריתם רקורסיבי פשוט. נציג אותו בשפת Python:

```
def gcd(a,b):
    if b == 0:
        return a
    return gcd(b, a % b)
```

תנאי העצירה של האלגוריתם הוא המקרה שבו $b = 0$, ובמקרה זה $a = (a, 0)$ (כי $a|0$ ו־$a \ge 0$). בכל מקרה אחר, מחושב ה־r בביטוי $a = qb + r$ והאלגוריתם ממשיך רקורסיבית עבור הזוג b, r.

7.2.2 ניתוח סיבוכיות האלגוריתם הבסיסי

על מנת למצוא את זמן הריצה של האלגוריתם, נתחיל מלשאול את עצמנו מה המספרים שעבורם הביצוע שלו הוא הגרוע ביותר ביחס לגודל הייצוג שלהם ־ לכל k, מה המספרים הקטנים ביותר שעדיין דורשים k צעדי חישוב? התשובה היא **מספרי פיבונאצ'י** שנזכיר כאן:

הגדרה 7.2 **סדרת פיבונאצ'י** היא סדרת המספרים $0, 1, 1, 2, 3, 5, 8, 13, \ldots$ המוגדרת על ידי נוסחת הנסיגה הבאה:

$$F_k = \begin{cases} 0 & k = 0 \\ 1 & k = 1 \\ F_{k-1} + F_{k-2} & k \ge 2 \end{cases}$$

לסדרת פיבונאצ'י שימושים רבים ושונים, וההופעה שלה בהקשר של אלגוריתם ה־gcd היא טבעית למדי כפי שנוכיח כעת:

טענה 7.3 אם $a > b \ge 1$ והרצת $\gcd(a, b)$ ביצעה $k \ge 1$ קריאות רקורסיביות, אז $a \ge F_{k+2}$ ו־$b \ge F_{k+1}$.

הוכחה: נוכיח את הטענה באינדוקציה על k.

בסיס: עבור $k = 1$, מכיוון ש־gcd ביצעה קריאה רקורסיבית לא ייתכן ש־$b = 0$ ולכן $b \ge 1 = F_2$. מכיוון ש־$a > b$ הרי ש־$a \ge 2 = F_3$.

צעד: נניח נכונות עבור $k - 1$ ונוכיח עבור k. נניח ש־$\gcd(a, b)$ ביצעה $k > 1$ קריאות רקורסיביות, אז $\gcd(b, r)$ ביצעה $k - 1 \ge 1$ קריאות רקורסיביות עבור $a = qb + r$ ולכן ניתן להשתמש עליה בהנחת האינדוקציה ולקבל:

- $b \ge F_{(k-1)+2} = F_{k+1}$
- $r \ge F_{(k-1)+1} = F_k$

כעת נסיק: $a = qb + r \ge b + r = F_{k+1} + F_k = F_{k+2}$, כמבוקש. ■

נשים לב שההוכחה גם מצביעה לנו על הסיטואציה שבה החסם מושג: המעבר $qb + r \ge b + r$ שהוא שוויון כאשר $q = 1$ הוא המקרה הגרוע ביותר (כי אז ה"הקטנה" שאנחנו מקבלים במעבר מ־a אל r היא הנמוכה ביותר - מחסרים את b בדיוק פעם אחת).

מסקנה 7.4 לכל $k \ge 1$, אם $a > b \ge 1$ ו־$b < F_{k+1}$ אז $\gcd(a,b)$ מבצע פחות מ־k קריאות רקורסיביות.

כעת נוכל לחסום את זמן הריצה של $\gcd(a,b)$. ראשית, מספר צעדי החישוב ש־$\gcd(a,b)$ מבצע הוא לינארי במספר הקריאות הרקורסיביות (כי אין לולאה פרט לקריאה הרקורסיבית). שנית, אם $a \ge b$ אז אחרי הקריאה הרקורסיבית הראשונה נקבל $a > b$ (במקרה זה $a = b$ של הסיבוב הקודם ו־$b = r$ של הסיבוב הקודם) ולכן תוספת של 1 למספר הקריאות הרקורסיביות מבטיחה שניכנס לתנאי הטענה שהוכחנו קודם. לבסוף, אנו יודעים במדויק מה קצב הגידול של F_k, מכיוון שידועה לנו נוסחה סגורה לסדרה זו: $F_k = \frac{\phi^k + \phi_{-1}^k}{\sqrt{5}}$ כאשר $\phi = \frac{1+\sqrt{5}}{2}$ הוא **יחס הזהב** ו־$\phi_- = \frac{1-\sqrt{5}}{2}$. עם זאת, ניתן להסתפק גם בחסם תחתון נאיבי יותר:

$$F_k = F_{k-1} + F_{k-2} \ge 2F_{k-2} \ge \ldots \ge 2^{\left\lfloor \frac{k}{2} \right\rfloor} \ge \frac{1}{2}\left(\sqrt{2}\right)^k$$

כעת, בהינתן b, יהא F_k מספר פיבונאצ'י המקסימלי כך ש־$F_k \le b$. כלומר, מצד אחד $b < F_{k+1}$ ולכן $\gcd(a,b)$ מבצע פחות מ־k קריאות רקורסיביות; מצד שני, $b \ge F_k \ge \frac{1}{2}\left(\sqrt{2}\right)^k$ ולכן $\lg(b) \ge \lg\left(\frac{1}{2}\left(\sqrt{2}\right)^k\right) = k\lg\left(\sqrt{2}\right) - 1$, כלומר $k = O(\log b)$ וזהו גם חסם זמן הריצה של האלגוריתם.

7.2.3 האלגוריתם האוקלידי המורחב

בשימוש פרקטי באלגוריתם האוקלידי על פי רוב אנחנו מעוניינים לא רק במציאת ה־gcd אלא גם במציאת ייצוג שלו בתור **צירוף לינארי** של a, b, מה שניתן לביצוע על ידי הרחבה קלה של האלגוריתם האוקלידי שאינה כרוכה בהגדלת הסיבוכיות.

משפט 7.5 אם $d = (a,b)$ אז קיימים שלמים x, y כך ש־$d = ax + by$

הוכחה: נוכיח באינדוקציה שלמה על b. אם $b = 0$ אז $(a,b) = a = a \cdot 1 + b \cdot 0$. אחרת, $a = qb + r$ כך ש־$0 \le r < b$ ומהנחת האינדוקציה ומכך ש־$d = (b,r)$, קיימים x', y' כך ש־$d = bx' + ry'$.

נציב בנוסחה עבור d את $r = a - qb$ ונקבל:

$$d = bx' + (a - qb)\,y' = ay' + b\,(x' - qy')$$

על כן, נגדיר:

- $x = y'$
- $y = x' - qy'$

ונקבל $d = ax + by$ כמבוקש. ■

ההוכחה כוללת בתוכה גם את השיטה שבה ניתן לחשב את x, y מתוך הקריאה הרקורסיבית. נשתמש בכך על מנת להרחיב את אלגוריתם ה־gcd שראינו:

```
def extended_gcd(a,b):
    if b == 0:
        return (a, 1, 0)
    d, x, y = extended_gcd(b, a % b)
    return (d, y, x - (a // b) * y)
```

סיבוכיות האלגוריתם זהה לזו שכבר ראינו, כי לא שינינו את המבנה הרקורסיבי שלו אלא רק הוספנו חישוב אריתמטי לכל איטרציה.

7.2.4 חישוב הכפולה המשותפת המינימלית

הכפולה המשותפת המינימלית $\operatorname{lcm}(a,b)$ היא מעין מושג אנלוגי ל־gcd. בעוד ה־gcd הוא המספר d **הגדול ביותר** כך ש־$d|a$ וגם $d|b$, lcm הוא המספר הטבעי האי שלילי f **הקטן ביותר** כך ש־$a|f$ וגם $b|f$. למרבה המזל, תכונה פשוטה שקושרת בין ה־gcd וה־lcm מאפשרת לנו חישוב יעיל של השני:

משפט 7.6 $\operatorname{lcm}(a,b)=\frac{ab}{\gcd(a,b)}$

הוכחה: די להוכיח כי $\operatorname{lcm}(a,b)\cdot\gcd(a,b)=ab$.
נכתוב $a=p_1^{k_1}\cdots p_n^{k_n}$ ו־$b=p_1^{t_1}\cdots p_n^{t_n}$ כאשר $p_1,\dots,p_n$ הם הראשוניים שמחלקים את a ואלו שמחלקים את b (אם ראשוני מחלק רק אחד משניהם, הוא יופיע בחזקת 0 במכפלה השניה). נשים לב לכך שמתקיים

- $\gcd(a,b)=p_1^{\min\{k_1,t_1\}}\cdots p_n^{\min\{k_n,t_n\}}$
- $\operatorname{lcm}(a,b)=p_1^{\max\{k_1,t_1\}}\cdots p_n^{\max\{k_n,t_n\}}$
- $ab=p_1^{k_1+t_1}\cdots p_n^{k_n+t_n}$

המשפט כעת נובע מיידית מכך ש־$x+y=\min\{x,y\}+\max\{x,y\}$ לכל זוג מספרים x,y (שכן אחד מהם שווה למינימום והשני שווה למקסימום). ■

נסיים באבחנה על תכונה שימושית של ה־gcd וה־lcm:

משפט 7.7 יהיו a,b טבעיים כלשהם. אז $\gcd(a,b)$ מתחלק בכל מספר שמחלק גם את a וגם את b, ואילו $\operatorname{lcm}(a,b)$ מחלק כל מספר שמתחלק גם ב־a וגם ב־b.

הוכחה: ראינו כי $\gcd(a,b)=ax+by$ עבור x,y כלשהם. אם c הוא מספר כך ש־$c|a$ וגם $c|b$ אז הוא מחלק גם את המכפלה שלהם בקבוע ואת סכומם, כלומר $c|ax+by=\gcd(a,b)$.
יהא כעת c כך ש־$a|c$ וגם $b|c$. נסמן $m=\operatorname{lcm}(a,b)$ ונחלק את c ב־m: $c=mq+r$ כאשר $0\le r<m$. מכיוון ש־$a|c$ וגם $a|m$ אז $a|c-mq=r$. בדומה גם $b|r$. כלומר, r הוא כפולה משותפת של a,b שקטנה מ־m ולכן בהכרח $r=0$ אחרת נקבל סתירה להגדרת m כמינימלי. מכאן ש־c מתחלק ב־m ללא שארית. ■

7.3 אריתמטיקה מודולרית

תורת המספרים אינה עוסקת רק בחוג השלמים $\mathbb{Z}$ אלא בחוגים רבים נוספים בעלי תכונות פחות או יותר דומות לאלו של $\mathbb{Z}$. הדוגמא השימושית והנפוצה ביותר היא החוג $\mathbb{Z}_n$ שבו נעסוק כאן.
בהינתן n טבעי כלשהו אפשר לחלק את המספרים השלמים ל**מחלקות שקילות** על פי השארית שהם מחזירים בחלוקה ב־n (שהיא מספר בתחום $0,1,\dots,n-1$). למשל, עבור $n=3$ נקבל את מחלקות השקילות:

$$\{0,3,-3,6,-6,\dots\}$$
$$\{1,4,-2,7,-5,\dots\}$$
$$\{2,5,-1,8,-4,\dots\}$$

אוסף מחלקות השקילות של היחס הזה מסומן ב־$\mathbb{Z}_n$. נהוג לסמן את אבריו פשוט בתור $\mathbb{Z}_n=\{0,1,2,\dots,n-1\}$ ולהגדיר עליהם פעולות חיבור וכפל שמושרות מהפעולות המתאימות על $\mathbb{Z}$. אנו אומרים ש־$\mathbb{Z}_n$ הוא **חוג** מכיוון שהוא קבוצה שמוגדרות עליה פעולות חיבור וכפל שמקיימות חלק מהתכונות הבסיסיות שמתקיימות עבור פעולות אלו ב־$\mathbb{Z}$.
פורמלית, לבקיאים בתורת החוגים, $\mathbb{Z}_n=\mathbb{Z}/n\mathbb{Z}$ עם הפעולות המושרות $[a]+[b]=[a+b]$ ו־$[a]\cdot[b]=[a\cdot b]$.
ביצוע פעולות ה**חיבור** וה**כפל** של איברים ב־$\mathbb{Z}_n$ הוא פשוט: מבצעים עבורם את פעולות החיבור והכפל הרגילות, ולאחר מכן מוצאים את השארית של חלוקת התוצאה ב־n.
נעסוק כעת בשתי פעולות נוספות שניתן לבצע ב־$\mathbb{Z}_n$: חילוק והעלאה בחזקה.

7.3.1 הופכי כפלי וחילוק ב־$\mathbb{Z}_n$

בהינתן $a\in\mathbb{Z}_n$, נאמר ש־$x\in\mathbb{Z}_n$ הוא **הופכי כפלי** של a אם $ax=1$. למשל, בחוג $\mathbb{Z}_7$, ההופכי הכפלי של 3 הוא 5 כי $3\cdot5=15$ ואחרי חלוקה ב־7 ולקיחת שארית נקבל 1.
תופעה זו, שבה מוכפלים a,b ששונים מ־±1 ומתקבל 1 לא מתרחשת ב־$\mathbb{Z}$. זה מאפשר לנו להגדיר **חלוקה** עבור מספרים הפיכים: אם a הפיך עם הופכי b, אז חלוקה ב־a מוגדרת ככפל בהופכי שלו, b. עם זאת, לא לכל מספר ב־$\mathbb{Z}_n$ קיים הפיך:

משפט 7.8 יהא n טבעי כלשהו ו־$a \in \mathbb{Z}_n$. אז a הפיך אם ורק אם $(a,n)=1$.

הוכחה: בכיוון אחד, אם $\gcd(a,n)=1$ אז קיימים x,y כך ש־$ax+yn=1$. אם נתבונן על משוואה זו מודולו n, נקבל $ax=1$ כך ש־x הוא ההופכי של a.

בכיוון השני, אם $d=\gcd(a,n)>1$ נתבונן ב־$b=\frac{n}{d}$. זהו מספר טבעי המקיים $0<b<n$ ובפרט $b\in\mathbb{Z}_n$. עם זאת, $ba=a\frac{n}{d}=\frac{a}{d}n$ ו־$\frac{a}{d}n$ הוא 0 מודולו n כי הוא מתחלק ב־n.

כעת, נניח בשלילה כי a הפיך, כלומר קיים x כך ש־$ax=1$. אז נקבל $b=b\cdot 1=b\cdot(ax)=(ba)x=0\cdot x=0$ וקיבלנו ש־$b=0$ בסתירה לכך שראינו כי $b>0$. ■

בלשון תורת החוגים, ההוכחה הראתה שכל איבר שונה מ־0 ב־$\mathbb{Z}_n$ הוא או הפיך או **מחלק אפס**, כלומר קיים איבר שונה מאפס כך שמכפלתם שווה לאפס (גם תופעה זו אינה קיימת ב־$\mathbb{Z}$). תכונה זו נותנת מוטיבציה להגדרה הבאה:

הגדרה 7.9 יהא n טבעי. **חבורת ההפיכים** מודולו n היא הקבוצה $\mathbb{Z}_n^* \triangleq \{a\in\mathbb{Z}_n \mid (a,n)=1\}$.

אנו מכנים את $\mathbb{Z}_n^*$ בשם **חבורה** כי אם משמיטים ממנה את פעולת החיבור ונותרים עם פעולת הכפל, מתקבל המבנה האלגברי **חבורה**.

אם $(a,b)=1$ אנו אומרים ש־a,b הם **זרים**. אם כן, התוצאה שראינו היא כי כדי להיות הפיך מודולו n מספיק והכרחי להיות זרים אליו.

מכיוון שבידינו אלגוריתם יעיל לחישוב gcd, ניתן להשתמש בו למציאת הופכי מודולורי או קביעה שאין כזה:

```
def modular_inverse(a,n):
    d, x, y = extended_gcd(a,n)
    if d > 1:
        return None
    return x
```

7.3.2 העלאה מהירה בחזקה

כאשר אנו עובדים במספרים שלמים, ביצוע אלגוריתמי של העלאה בחזקה גבוהה, למשל חישוב $2^{10^{10}}$, הוא עניין קשה בגלל **גודל הייצוג** הגדול של התוצאה. שיקולי זמן ריצה הופכים להיות משניים בחישוב כזה. לעומת זאת, אם אנו מבצעים חישובים מודולו n כאשר n מסדר גודל סביר (גוגול, 10^{100} הוא עדיין סדר גודל סביר שכזה, למשל) הרי שתוצאה של העלאה בחזקה גבוהה k שכזו תהיה בעלת ייצוג בגודל סביר אבל חישוב פעולת ההעלאה בחזקה על ידי ביצוע סדרתי של k פעולות כפל הוא איטי שלא לצורך. נראה כאן כיצד ניתן לחשב את a^k בסיבוכיות $O(\log k)$ פעולות כפל. האלגוריתם תקף גם במספרים שלמים ־ אך כאמור, השימושיות שלו מוגבלת יותר שם עקב הגודל העצום של התוצאה.

בהינתן a, ניתן לכפול אותו בעצמו לקבלת a^2. מספר זה ניתן לכפול בעצמו לקבלת a^4, וכן הלאה. נקבל את הסדרה: $a,a^2,a^4,a^8,a^{16},a^{32},\ldots,a^{2^t},\ldots$

שבה חישוב a^{2^t} דרש רק t פעולות כפל.

על מנת לחשב את a^k עבור כל מספר k, ניזכר ראשית בכך שניתן לייצג את k ב**ייצוג בינארי**, בתור סכום חזקות של 2:

$k=\sum_{i=0}^{t}b_i2^i$

כאשר $b_i\in\{0,1\}$ ו־$t=\lfloor\lg k\rfloor$.

ולכן ניתן לכתוב את a^k כך:

$$a^k=a^{\sum_{i=0}^{t}b_i2^i}=\prod_{i=0}^{t}a^{b_i2^i}=\prod_{b_i=0}a^{2^i}$$

כלומר, כדי לחשב את a^k אנחנו יכולים לאתחל איבר $x=1$ ולחשב את הסדרה $a,a^2,a^4,\ldots,a^{2^t}$ כך שבכל פעם שבה הגענו אל i עבורו $b_i=1$, אנו כופלים את האיבר a^{2^i} שהגענו אליו ב־x.

נקבל את האלגוריתם הבא:

```python
def modular_power(a,k,n):
    x = 1
    a_power = a
    while k > 0:
        if k % 2 == 1:
            x = (x * a_power) % n
        k = k // 2
        a_power = (a_power * a_power) % n
    return x
```

7.3.3 משפט השאריות הסיני

משפט השאריות הסיני נותן לנו את היכולת לפתור **מערכת משוואות מודולריות**. נפתח עם דוגמא, ברוח "פגישת התאומים" של אפרים קישון: לצורך תיקון דליפה בצינור בדירה עלינו להביא אליה בו זמנית את שטוקס האינסטלטור שיתקן את הצינור ואת גדעון הבנאי שיפתח את הקיר. לרוע המזל שטוקס פנוי רק ביום ג' כל שבוע ואילו גדעון פנוי רק ביום ה' כל שבועיים, כך שנראה שאין דרך להפגיש אותם. למרבה המזל, לאחר מעט שכנוע הצלחנו לגרום לגדעון לחשוב שיום שבת מגיע רק אחת לשבועיים, כך שהם כוללים רק 13 ימים. האם כעת נוכל לגרום להם להיפגש?

בתרגום למתמטיקה, השאלה היא האם קיים x כך ש־

$x \equiv_7 3$

$x \equiv_{13} 5$

על מנת למצוא את הפתרון נשתמש בתעלול הבא: נניח שהצלחנו למצוא זוג מספרים a,b המקיימים

$a \equiv_7 1$

$a \equiv_{13} 0$

ו־

$b \equiv_7 0$

$b \equiv_{13} 1$

במקרה זה, אם נגדיר $x = 3a + 5b$ נקבל את התוצאה המבוקשת:

$x \equiv_7 3 \cdot 1 + 5 \cdot 0 = 3$

$x \equiv_{13} 3 \cdot 0 + 5 \cdot 1 = 5$

יתר על כן, אם נמצא a,b כאלו נוכל לפתור גם **כל משוואה אחרת** מודולו 7 ו־13 ־ פשוט נכפיל את a,b בתוצאות המבוקשות, בהתאם.

כיצד נמצא a,b מתאימים? ראשית, מכיוון ש־$(7,13) = 1$ הרי ש־13 הפיך מודולו 7, כלומר קיים s כך ש־$13s \equiv_7 1$. בנוסף, $13s \equiv_{13} 0$. אם נשתמש באלגוריתם האוקלידי נקבל $s = -1$, ועל כן נגדיר $a = -13$.

באופן דומה, מכיוון ש־7 הפיך מודולו 13 עם הופכי 7, נגדיר $b = 14$.

כעת נקבל $x = 3 \cdot (-13) + 5 \cdot 14 = -39 + 70 = 31$. זהו מספר הימים שנחכה עד ששטוקס וגדעון יוכלו להיפגש.

נעבור כעת למקרה הכללי. מהדוגמא שראינו עולה שאם המודולוסים אינם זרים (כמו במקרה של 7 ו־14) אז ייתכן שלמערכת המשוואות לא תהיה פתרון, אבל אם הם זרים אז לכל בחירת ערכים לאגף ימין של המשוואה, קיים פתרון למשוואה. זה נכון גם באופן כללי:

משפט 7.10 (משפט השאריות הסיני) יהיו $n_1, \ldots, n_k$ מספרים טבעיים חיוביים כך ש־$(n_i, n_j) = 1$ לכל $i \neq j$, ויהיו $b_1, \ldots, b_k$ מספרים שלמים כלשהם. אז למערכת המשוואות

$$\begin{gathered} x \equiv_{n_1} b_1 \\ \vdots \\ x \equiv_{n_k} b_k \end{gathered}$$

קיים פתרון והוא יחיד מודולו $n = \prod_{i=1}^{k} n_i$.

הוכחה: לכל $1 \le i \le n$ נסמן $a_i = \frac{n}{n_i}$. בהכרח $(a_i, n_i) = 1$ כי אם $p | a_i = \frac{n}{n_i}$ אז בהכרח $p | n_j$ עבור $j \neq i$ ואז נקבל ש־p הוא גורם משותף של n_i, n_j בסתירה לכך ש־$(n_i, n_j) = 1$.

מכיוון ש־$(a_i, n_i) = 1$ קיים ל־a_i הופכי מודולו n_i שנסמן a_i^{-1}. נגדיר $d_i = a_i^{-1} a_i$, אז מתקיים

- $d_i \equiv_{n_i} 1$
- $d_i \equiv_{n_j} 0$ לכל $j \neq i$

כאשר המשוואה השנייה נובעת מכך ש־$a_i = \frac{n}{n_i}$ ובפרט $n_j | a_i$.

נגדיר כעת $x = \sum_{i=1}^{n} b_i d_i$, ונקבל לכל i $x \equiv_i b_i$, כמבוקש.

נותר להראות כי הפתרון הוא יחיד מודולו n. ראשית ניתן את הדעת לטענה הבאה: אם a, b זרים, כלומר $\gcd(a,b) = 1$, אז $\operatorname{lcm}(a,b) = \frac{ab}{\gcd(a,b)} = ab$. כלומר, כל מספר שמתחלק ב־a, b זרים מתחלק גם במכפלתם ובאינדוקציה הדבר נובע לכל n מספרים זרים.

נניח ש־x_1, x_2 שניהם פתרונות למערכת המשוואות. אז לכל i מתקיים $x_1 \equiv_{n_i} x_2$, כלומר, $x_1 - x_2$ מתחלק על ידי n_i. מכיוון שהוא מתחלק על ידי מספרים זרים, הוא מתחלק גם על ידי מכפלתם, כלומר על ידי n, וקיבלנו $x_1 \equiv_n x_2$. ■

ניתן לממש את האלגוריתם בשפת Python באופן הבא:

```
def crt(pairs):
    n = 1
    x = 0
    for (ni,bi) in pairs:
        n *= ni
    ds = []
    for (ni,bi) in pairs:
        ai = n // ni
        di = modular_inverse(ai, ni) * ai
        x = x + di*bi
    return x
```

מסקנה אחת ממשפט השאריות הסיני נוגעת למבנה של $\mathbb{Z}_n$:

משפט 7.11 אם $(a,b) = 1$ אז $\mathbb{Z}_{ab} \cong \mathbb{Z}_a \times \mathbb{Z}_b$ וגם $\mathbb{Z}_{ab}^* \cong \mathbb{Z}_a^* \times \mathbb{Z}_b^*$

הוכחה: נגדיר פונקציה $\varphi : \mathbb{Z}_{ab} \to \mathbb{Z}_a \times \mathbb{Z}_b$ על ידי $\varphi(x) = (x \bmod a, x \bmod b)$. הפונקציה היא על בגלל טענת **הקיום** של משפט השאריות הסיני, והיא חח"ע בגלל טענת **היחידות** של משפט השאריות הסיני.

אותה פונקציה מראה ש־$\mathbb{Z}_{ab}^* \cong \mathbb{Z}_a^* \times \mathbb{Z}_b^*$; רק יש להוסיף את האבחנה לפיה $(x, ab) = 1$ אם ורק אם $(x,a) = 1$ וגם $(x,b) = 1$. ■

7.3.4 פונקציית אוילר

נגדיר את **פונקציית אוילר:** $\varphi(n) \triangleq |\mathbb{Z}_n^*|$. כלומר, $\varphi(n)$ היא מספר המספרים הקטנים מ־n וזרים לו.

העניין המרכזי שלנו בפונקציית ינבע מאחת מתכונותיה:

משפט 7.12 (משפט אוילר): אם $(a,n) = 1$ אז $a^{\varphi(n)} \equiv_n 1$.

הוכחת המשפט מתבססת על משפט מרכזי בתורת החבורות האלמנטרית הנקרא **משפט לגראנז'**. לפיו, אם G חבורה ו־$a \in G$ אז $a^{|G|} = e$ כאשר e הוא איבר היחידה של החבורה. לא נוכיח את משפט לגראנז' כאן. משפט אוילר נובע ממנו מיידית בזכות היות $\mathbb{Z}_n^*$ חבורה.

בשל השימושיות של פונקציית אוילר, מעניין אותנו לחשב אותה. נפתח במספר אבחנות פשוטות:

טענה 7.13 לכל ראשוני p, $\varphi(p) = p - 1$.

הוכחה: מכיוון ש־p ראשוני, לכל מספר a כך ש־$(a,p)\neq 1$ חייב להתקיים $(a,p)=p$ (שכן אין עוד מחלקים של p), כלומר בפרט $p|a$. לכן אם $1\le a<p$ בהכרח $(a,p)=1$ ולכן $a\in\mathbb{Z}_p^*$, כלומר $\mathbb{Z}_p^*=\{1,2,3,\dots,p-1\}$. ■

כאשר מסתכלים על משפט אוילר במקרה הפרטי של p ראשוני, מקבלים את התוצאה $a^{p-1}\equiv_p 1$. בשל חשיבותה ההיסטורית, התוצאה הזו זכתה לשם מיוחד ־ **המשפט הקטן של פרמה**.

טענה 7.14 לכל ראשוני p, $\varphi\left(p^n\right)=p^{n-1}\left(p-1\right)$

הוכחה: באותו אופן כמו קודם, נסיר מהקבוצה $\{1,2,3,\dots,p^n\}$ את כל הכפולות של p. הכפולות הללו נמצאות בהתאמה חח"ע ועל עם הקבוצה $\left\{1,2,3,\dots,p^{n-1}\right\}$ באופן הבא: $n\mapsto p\cdot n$. לכן מקבוצה מגודל p^n הסרנו קבוצה מגודל p^{n-1} וקיבלנו קבוצה מגודל $p^n-p^{n-1}=p^{n-1}\left(p-1\right)$. ■

טענה 7.15 אם $(a,b)=1$ אז $\varphi\left(ab\right)=\varphi\left(a\right)\varphi\left(b\right)$, כלומר φ כפלית על מספרים זרים.

הוכחה: זו תוצאה מיידית ממשפט השאריות הסיני: $\mathbb{Z}_{ab}^*=\mathbb{Z}_a^*\times\mathbb{Z}_b^*$ ולכן $|\mathbb{Z}_{ab}^*|=|\mathbb{Z}_a^*|\cdot|\mathbb{Z}_b^*|$. ■

מתוצאות פשוטות אלו נובעת נוסחה כללית לחישוב φ. יהא $n>1$ ויהא $n=p_1^{k_1}\cdots p_n^{k_n}$ הפירוק לגורמים ראשוניים שלו. אז

$$\begin{aligned}
\varphi\left(n\right)&=\varphi\left(p_1^{k_1}\cdots p_n^{k_n}\right)\\
&=\varphi\left(p_1^{k_1}\right)\cdots\varphi\left(p_n^{k_n}\right)\\
&=p_1^{k_1-1}\left(p_1-1\right)\cdots p_n^{k_n-1}\left(p_n-1\right)\\
&=p_1^{k_1}\left(1-\frac{1}{p_1}\right)\cdots p_n^{k_n}\left(1-\frac{1}{p_n}\right)\\
&=p_1^{k_1}\cdots p_n^{k_n}\left(1-\frac{1}{p_1}\right)\cdots\left(1-\frac{1}{p_n}\right)\\
&=n\prod_{p|n}\left(1-\frac{1}{p}\right)
\end{aligned}$$

כאשר המכפלה בסיום נלקחת על כל הגורמים הראשוניים של n.

נוסחה זו נותנת דרך פשוטה לחשב את $\varphi\left(n\right)$ כאשר ידועים לנו כל הגורמים הראשוניים של n. לרוע המזל, איננו מכירים דרך יעילה למצוא את הגורמים הראשוניים הללו. למרבה המזל, אנחנו יודעים לנצל את חוסר הידע הזה לצורך מערכת ההצפנה RSA, שנראה בהמשך.

7.4 בדיקת ראשוניות וייצור ראשוניים

7.4.1 אלגוריתם מילר־רבין

ראינו כבר כי אלגוריתם בדיקת ראשוניות "עד השורש" הוא בלתי יעיל בצורה קיצונית. נשאלת השאלה מה **כן** יעיל. לצורך כך נתחיל מאבחנה פשוטה לגבי מה **לא יכול** להיות יעיל. כיום לא ידוע שום אלגוריתם יעיל עבור **פירוק לגורמים** של מספר; במילים אחרות, לא מוכר אלגוריתם יעיל אשר בהינתן מספר $n=ab$ כך ש־$a,b>1$ מוצא את a או את b.

בהינתן שזה המצב, לא ייתכן שמבחן ראשוניות שמתבסס על מציאת פירוק של n יהיה יעיל. על מבחן הראשוניות להכריע בצורה כלשהי לגבי הפריקות/אי הפריקות של n בצורה "עיוורת". כיצד ניתן לעשות זאת?

התשובה היא שלראשוניים יש נטייה ליצור מבנים אלגבריים "יפים" בזמן שעבור מספרים פריקים חלק מהתכונות היפות משתבשות, ומספיק לאתר תכונה אחת שהשתבשה כדי להגיע למסקנה שהמספר שהמבנה יצר אינו ראשוני. המבנה האלגברי שעליו נתבסס הפעם הוא $\mathbb{Z}_n$. ספציפית, נתבסס על שתי התכונות הבאות:

- המשפט הקטן של פרמה: עבור p ראשוני, לכל $a\in\mathbb{Z}_p^*$ מתקיים $a^{p-1}=1$ (כשהחשבון הוא ב־$\mathbb{Z}_p^*$).
- עבור p ראשוני, $\mathbb{Z}_p$ הוא **שדה**.

התכונה הראשונה נותנת לנו מבחן בדיקת ראשוניות מיידי, שנקרא **מבחן פרמה**:

```
def fermat_primality_test(n):
    a = random.randint(2,n-1)
    if modular_power(a, n-1, n) != 1:
        return False
    return True
```

לרוע המזל, המבחן אינו אמין דיו לכשעצמו. זאת מכיוון שהמשפט הקטן של פרמה איננו "אם ורק אם"; בהחלט ייתכן שעבור n פריק יהיו קיימים ערכי a כך ש־$a^{n-1}=1$. גרוע מכך, יש מספרים פריקים שעבורם **לכל** a שזר להם מתקיים $a^{n-1}=1$ ־ מספרים אלו מכונים **מספרי קרמייקל** והקטן שבהם הוא 561. בגלל בעיות אלו מבחן פרמה אינו מספיק טוב לכשעצמו.

עם זאת, שינוי קטן באופן שבו מחושבת החזקה a^{n-1} במהלך מבחן פרמה מאפשר לנו להוסיף **עוד** בדיקה "על הדרך", והשילוב של הבדיקה של משפט פרמה והבדיקה הנוספת מאפשר לנו לזהות מספרים פריקים בהסתברות טובה: אם מספר הוא פריק, אז ההסתברות שלו "לרמות" ולעבור בהצלחה את המבחן היא $\frac{1}{4}$.

בהינתן n, ניתן לחשב במהירות את הפירוק $n-1=2^k\cdot d$ כאשר d הוא אי זוגי ־ פשוט מחלקים את $n-1$ ב־2 כל עוד לא קיבלנו מספר אי־זוגי. כעת ניתן לשנות את אופן חישוב a^{n-1}. ראשית כל מחשבים את a^d בעזרת אלגוריתם ההעלאה בחזקה הרגיל. כעת מציבים $x\leftarrow a^d$ ובמהלך k איטרציות מבצעים את הפעולה $x\leftarrow x^2$. כל העלאה בחזקה של x משולה לכפל החזקה של a^d ב־2, כך שלאחר k צעדים נקבל בדיוק את $a^{2^kd}=a^{n-1}$ המבוקש.

מה שאנו מחפשים בשלב ההעלאה־בריבוע־המתמשכת הוא **שורש לא טריוויאלי של 1**. נסביר זאת: במספרים ממשיים, אנו יודעים שלמשוואה $x^2=1$ קיימים בדיוק שני פתרונות: $x=\pm1$. זה מקרה פרטי של טענה כללית יותר ־ מעל שדה, לפולינום ממעלה n יש לכל היותר n שורשים. כאשר p הוא ראשוני, $\mathbb{Z}_p$ הוא שדה והפתרונות היחידים של המשוואה $x^2=1$ הם $1,p-1$ (כאן $p-1$ הוא שקול ל־-1). לעומת זאת, אם n הוא פריק, אז $\mathbb{Z}_n$ איננו שדה ובהחלט יכולים להיות פתרונות נוספים. כך למשל עבור $\mathbb{Z}_8$ מתקיים $1^2=3^2=5^2=7^2=1$, כך ש־$3,5$ שניהם שורשים "לא טריוויאליים" של 1.

בהתאם לכך, כאשר האלגוריתם מבצע את החישוב $x\leftarrow x^2$, אם התוצאה שהתקבלה הייתה 1 אבל x עצמו לא היה ±1, ניתן לדעת מיידית ש־n הוא פריק ולעצור את ריצת האלגוריתם.

קיבלנו את האלגוריתם הבא:

```
def miller_rabin(n):
    a = random.randint(1,n-1)
    k = 0
    d = n-1
    while d % 2 == 0:
        k += 1
        d //= 2
    x = modular_power(a,d,n)
    for i in range(k):
        x_new = (x*x) % n
        if x != 1 and x != n-1 and x_new == 1:
            return False
        x = x_new
    if x != 1:
        return False
    return True
```

7.4.2 סיבוכיות ונכונות אלגוריתם מילר־רבין

סיבוכיות אלגוריתם מילר רבין זהה לסיבוכיות של העלאה בחזקה מודולרית. אנו נזקקים רק ל־$O\left(\lg n\right)$ פעולות כפל ומציאת שארית מודולרית, כך שהאלגוריתם יעיל. השאלה המהותית יותר היא לגבי נכונותו.

ראשית, אם n ראשוני, האלגוריתם עונה True בודאות; הדרך היחידה שבה האלגוריתם עונה False היא אם המשפט הקטן של פרמה נכשל (שורה 15) או שהתגלה שורש לא טריוויאלי של 1 (שורה 12) ושני אלו לא יכולים לקרות עבור n ראשוני.

נותר להוכיח שאם n לא ראשוני, אז עבור כמות גדולה מספיק של a־ים ב־$\mathbb{Z}_n$, האלגוריתם יחזיר False על n.

נשים לב לדבר המדויק שאנו מוכיחים כאן: מטרתנו היא להוכיח ש**לכל** n פריק, קיימת הסתברות גדולה מספיק להחזיר False עליו. תוצאה זו חזקה משמעותית מתוצאות כמו "עבור n אקראי בהסתברות טובה האלגוריתם מחזיר עליו את התשובה הנכונה" שהן חסרות כל ערך (כי, למשל, אלגוריתם שמחזיר False על כל תוצאה אפשרית יענה את התשובה הנכונה בהסתברות טובה על כל קלט, כי רוב הקלטים האקראיים לא יהיו ראשוניים).

משפט 7.16 לכל n פריק אי זוגי, מספר האיברים $1\le a<n$ שעליהם אלגוריתם מילר־רבין מחזיר False הוא לכל היותר $\frac{n-1}{2}$

הוכחה: ההוכחה שלנו תסתמך שוב על משפט לגראנז' מתורת החבורות, בניסוח הכללי יותר שלו: אם G חבורה ו־H תת־חבורה של G, אז $|H|\,|\,|G|$ (גודל H מחלק את גודל G). לא נזדקק למושג הכללי של חבורה ותת־חבורה; די לדבר על החבורה $\mathbb{Z}_n^*$. **תת־חבורה** של $\mathbb{Z}_n^*$ היא תת־קבוצה $\mathbb{Z}_n^*$ שסגורה לפעולת הכפל (אם $a,b\in H$ אז גם $ab\in H$).

ראשית, נניח ש־n אינו מספר קרמייקל, כלומר קיים $1\le a<n$ אחד לפחות כך ש־$a\in\mathbb{Z}_n^*$ ש־$a^{n-1}\not\equiv_n 1$. נגדיר כעת תת־חבורה של $\mathbb{Z}_n^*$ באופן הבא:

$$H=\left\{x\in\mathbb{Z}_n^*\mid x^{n-1}\equiv_n 1\right\}$$

מצד אחד H אינה ריקה כי $1\in H$, ו־H סגורה לכפל (כי $\left(xy\right)^{n-1}=x^{n-1}y^{n-1}\equiv_n 1\cdot1=1$) ולכן היא תת־חבורה של $\mathbb{Z}_n^*$, ומצד שני $H\ne\mathbb{Z}_n^*$ כי $a\notin H$. ממשפט לגראנז' נסיק כעת כי $|H|\le\frac{1}{2}\left|\mathbb{Z}_n^*\right|\le\frac{n-1}{2}$.

בכך הראינו כי מבחן פרמה עובד טוב כמעט לכל המספרים; רק מספרי קרמייקל נותרו המקרה הבעייתי. עבור מקרה זה נגדיר את H בצורה מעט מורכבת יותר, וגם ההוכחה שקיים איבר ב־$\mathbb{Z}_n^*$ שאינו שייך אליה תהיה מורכבת יותר.

כזכור, $n-1=2^kd$ כאשר d אי זוגי. נתבונן כעת בקבוצת כל הזוגות (v,j) כך ש־$v\in\mathbb{Z}_n^*$ ו־$0\le j\le k$ המקיימים $v^{2^jd}\equiv_n -1$. כלומר, אלו כל האיברים שהפעלת מילר־רבין עליהם תוביל ל־-1, מה שיוביל לתשובה לא נכונה של האלגוריתם (כי n מקיים את מבחן פרמה, ומ־-1 מגיעים בצעד הבא אל 1 מבלי לעבור דרך שורש לא טריוויאלי).

זוג אחד מסוג זה בוודאי קיים: $(n-1,0)$, כי $\left(n-1\right)^{2^0\cdot d}\equiv_n -1$. לכן קיים j מקסימלי שעבורו עדיין קיים זוג (v,j) שכזה. כעת נגדיר את H:

$$H=\left\{x\in\mathbb{Z}_n^*\mid x^{2^jd}\equiv_n\pm1\right\}$$

קל לראות ש־H סגורה לכפל והיא לא ריקה שכן $v\in H$ שייך אליה. מכאן שהיא תת־חבורה של $\mathbb{Z}_n^*$. על כן, אם נוכיח כי קיים איבר שאינו שייך ל־H אבל כל האיברים שעליהם האלגוריתם נכשל שייכים ל־H, נסיים כמו קודם.

יהא $a\in\mathbb{Z}_n^*$ כלשהו שעליו האלגוריתם נכשל. נתבונן על a^{2^jd}. אם $a^{2^jd}\equiv_n\pm1$ אז $a\in H$ כמבוקש. אחרת, נתבונן בסדרה $a^{2^jd},a^{2^{j+1}d},\dots,a^{2^kd}$. אנו יודעים כי:

- בסדרה זו -1 אינו מופיע כלל אחרת נקבל סתירה למקסימליות j.
- סדרה זו מסתיימת ב־1 (כי n הוא מספר קרמייקל, ולכן $a^{n-1}\equiv_n 1$ לכל $a\in\mathbb{Z}_n^*$) אך אינה נפתחת ב־1.

נתבונן במקום הראשון שבו מופיע 1 בסדרה. האיבר במקום שלפניו אינו 1 או -1 ולכן הוא שורש לא טריוויאלי של 1 שהאלגוריתם היה אמור לאתר ־ מכאן שהאלגוריתם אינו נכשל על a, בסתירה להנחה שלנו; המסקנה היא שמתקיים $a^{2^jd}\equiv_n\pm1$ ולכן $a\in H$.

נותר האתגר של מציאת איבר $a\in\mathbb{Z}_n^*$ כך ש־$a\notin H$. לצורך כך נסתמך על כך שמספר קרמייקל n אינו חזקה p^t של ראשוני; נוכיח זאת בהמשך. ניקח אם כן פירוק $n=n_1n_2$ כך ש־$(n_1,n_2)=1$. כזכור, מצאנו קודם איבר v כך ש־$v^{2^jd}\equiv_n -1$ ולכן בפרט גם מודולו n_1 נקבל $v^{2^jd}\equiv_{n_1}-1$. על פי משפט השאריות הסיני, קיים a כך ש־

- $a \equiv_{n_1} v$
- $a \equiv_{n_2} 1$

ולכן נקבל

- $a^{2^j d} \equiv_{n_1} -1$
- $a^{2^j d} \equiv_{n_2} 1$

כעת, אם היה מתקיים $a^{2^j d} \equiv_n 1$ בפרט היה נובע מכך $a^{2^j d} \equiv_{n_1} 1$, בסתירה למשוואה הראשונה; ואם היה מתקיים $a^{2^j d} \equiv_n -1$ היה בפרט נובע מכך $a^{2^j d} \equiv_{n_2} -1$ בסתירה למשוואה השניה. המסקנה היא ש־$a^{2^j d} \not\equiv_n \pm 1$ ולכן $a \notin H$.

עדיין יש להראות כי $a \in \mathbb{Z}_n^*$. ראשית, מכיוון ש־$a \equiv_{n_1} v$ ו־$v \in \mathbb{Z}_n^*$ נקבל ש־$(a, n_1) = 1$ כי אם היה מחלק משותף ל־a, n_1 הוא היה מחלק גם את $v = a + n_1 t$ ולכן היה ל־v, n מחלק משותף. נימוק דומה עובד גם עבור n_2 כי $(1, n_2) = 1$ ולכן נקבל $(a, n_2) = 1$. מכאן ש־$(a, n) = 1$ כי כל מחלק משותף של a, n מחלק או את n_1 או את n_2.

נותר רק להשלים את החוב שלנו לפיו מספר קרמייקל לא יכול להיות מהצורה p^t. נניח בשלילה ש־$n = p^t$ כאשר p אי זוגי. במקרה זה, אפשר להוכיח ש־$\mathbb{Z}_{p^t}^*$ הוא חבורה ציקלית, כלומר כל איבריה הם חזקות של איבר נתון יחיד g. מכיוון ש־$\left|\mathbb{Z}_{p^t}^*\right| = \varphi(p^t) = p^{t-1}(p-1)$ אנו יודעים שהסדר של g (המספר החיובי הקטן ביותר s כך ש־$g^s = 1$) הוא $p^{t-1}(p-1)$. בנוסף לכך, מתקיים $g^{n-1} = 1$ שכן n הוא מספר קרמייקל.

נשתמש כעת במשפט מתורת החבורות שקובע שהסדר של איבר מחלק כל חזקה אחרת שלו שמחזירה 1, ונקבל:

$$p^{t-1}(p-1) \mid p^t - 1$$

אבל אם $t > 1$ כמו במקרה שלנו, p מחלק את אגף שמאל אבל לא את אגף ימין, מה שמוביל לסתירה. זה מסיים את ההוכחה. ■

7.4.3 שימוש בפועל באלגוריתם מילר־רבין

הוכחנו כי אלגוריתם מילר־רבין עונה נכון לכל ראשונית ולכל מספר אי זוגי שאינו ראשוני הוא עונה נכון בהסתברות $\frac{1}{2}$. ניתן להוכיח תוצאות חזקות יותר אולם נסתפק במה שעשינו עד כה כדי לקבל אלגוריתם יעיל ואמין לבדיקת ראשוניות.

באופן כללי, ניתן לשפר את הסתברות ההצלחה של אלגוריתם הסתברותי על ידי מספר הרצות שלו. במקרה הנוכחי האלגוריתם הוא בעל טעות "חד־צדדית", מה שאומר שאם הוא החזיר תשובה שלילית ("המספר פריק") ניתן בודאות להחזיר תשובה זו, ולכן ניתן לפעול כך: בהינתן קלט, לבצע מספר הפעלות של מילר־רבין עליו ולהחזיר תשובה חיובית רק אם בכל ההרצות הייתה תשובה חיובית. בצורה זו, ההסתברות לענות נכון על מספר ראשוני היא 1 וההסתברות לענות נכון על מספר לא ראשוני היא 1 פחות הסתברות הכשלון. כדי להיכשל, הכרחי להיכשל בכל ההרצות, וההסתברות לכך היא $\frac{1}{2^t}$ כש־t הוא מספר ההרצות שבוצעו. לכן הסתברות ההצלחה היא $1 - \frac{1}{2^t}$.

אופטימיזציה אחת שתמיד כדאי לנקוט בה היא בדיקה האם n מתחלק בקבוצה כלשהי של ראשוניים קטנים ־ בדיקת חלוקה כזו אינה יקרה מבחינה חישובית ועשויה לחסוך הרצה יקרה יותר של מילר־רבין (ולכן גם אין צורך להטריח את עצמנו בשאלה מה קורה כשמריצים את אלגוריתם מילר־רבין על מספר זוגי).

```
def is_prime(n):
    for p in [2,3,5,7,11,13,17,19]:
        if n % p == 0:
            return False
    for _ in range(10):
        if not miller_rabin(n):
            return False
    return True
```

הקוד הוא דוגמא אקראית בלבד; על פי רוב נבדקת כמות גדולה בהרבה של ראשוניים קטנים, ואין צורך ב־10 הרצות של מילר־רבין.

7.4.4 מציאת ראשוניים גדולים

השימוש שאנו מייעדים למילר־רבין הוא מציאה של מספרים ראשוניים גדולים על מנת להשתמש בהם בבניית מערכת RSA. כאן "גדולים" פירושו "בני מאות ספרות". מילר־רבין יכול לבדוק ביעילות האם מספר בן מאות ספרות הוא ראשוני או לא, אך כיצד **לייצר** מספרים כאלו? הדרך הפשוטה ביותר היא להגריל מספרים גדולים ולבדוק. כאן בא לעזרתנו **משפט המספרים הראשוניים** שאומר שמספר הראשוניים בתחום $\{1,2,\ldots,n\}$ הוא בערך $\frac{n}{\ln n}$, כלומר ההסתברות להגריל ראשוני אקראי מ־1 עד n היא בערך $\frac{1}{\ln n}$ ־ בערך מסדר גודל של מספר הספרות של n. דהיינו, אם אנו מעוניינים להגריל ראשוני מסדר גודל של מאות ספרות, נצטרך לבדוק מאות מספרים בלבד ־ משימה קלה בסיוע מילר־רבין.

```
def generate_prime(digits):
    for _ in range(10000):
        candidate = random.randrange(10**(digits-1), 10**digits)
        if is_prime(candidate):
            return candidate
    return None
```

7.5 מערכת ההצפנה RSA

7.5.1 מבוא למערכות הצפנה

מערכת הצפנה באה לפתור את הבעיה הבאה: נניח שאליס ובוב חולקים ערוץ תקשורת כלשהו (האינטרנט, קו טלפון, יוני דואר, שליח, פרפרים, צעקות) ורוצים להעביר בו אינפורמציה בצורה מאובטחת, כך שגם במקרה שבו ערוץ התקשורת נופל קורבן למתקפה של גורם עוין המידע שאליס ובוב חולקים לא יגיע אל אותו גורם עוין. **מערכת הצפנה** באה לספק יכולת זו, לכל הפחות בצורה חלקית.

הסיטואציה הנפוצה ביותר היא זו שבה הגורם העוין, שנכנה "איב" יכולה אך ורק לצותת לערוץ התקשורת אך לא להשתלט עליו ולשנות בו דברים (כדי להתמודד עם הסיטואציה הבעייתית יותר צריך מערכות מורכבות יותר מאלו שנדבר עליהן). הפתרון הוא **לשנות** את המידע טרם שליחתו כך שייראה כמו ג'יבריש, אך שניתן יהיה לבצע תיקון של המידע לאחר שהתקבל בצד השני. כדי לבצע את "קלקול" ואת "תיקון" המידע מתבססים אליס ובוב על מידע סודי שידוע לשניהם ־ **מפתח ההצפנה**. היה עליהם לשתף את המפתח מראש, בערוץ תקשורת מאובטח (למשל, פגישה פיזית) אולם מרגע שיש להם מפתח משותף הם מסוגלים לשתף באמצעותו כמות גדולה של מידע (אבטחה "מושלמת" ניתן להשיג רק כאשר המפתח הוא חד־פעמי וגודלו כגודל המידע שאותו רוצים לשתף, אולם בעזרת מפתח קטן מאוד אפשר להשיג אבטחה טובה מאוד לכמויות גדולות של מידע).

פורמלית, אם לאליס יש הודעה M שברצונה לשלוח (הודעה כזו נקראת "כתב גלוי", Plaintext) היא קוראת לפונקציה encrypt שמקבלת את ההודעה ואת המפתח הסודי K ומוציאה "כתב סתר" (Ciphertext) שנראה כמו ג'יבריש: $C=\text{encrypt}\left(M,K\right)$. את C אליס שולחת לבוב, והוא משתמש בפונקציה משלו, $M=\text{decrypt}\left(C,K\right)$. אם איב מצותת להתרחשות הזו היא מקבלת את C אך לא את M, ואם שיטת ההצפנה טובה דיה, איב לא תוכל לשחזר את M מתוך C בהינתן היכולות החישוביות המוגבלות שלה.

החיסרון במערכת הצפנה שכזו ברור ־ על אליס ובוב לשתף את המפתח K ביניהם מראש, או להשתמש בערוץ מאובטח לצורך השיתוף שלו. למרות הקושי המהותי הזה, מערכות הצפנה היו בשימוש נרחב לאורך מרבית ההיסטוריה.

בשנות ה־70 של המאה ה־20 הוצעה גישה נוספת להצפנה, שמצליחה להתמודד עם בעיית שיתוף המפתחות ־ גישת **המפתח הפומבי**. בגישה זו אליס ובוב אינם זקוקים למפתח משותף על מנת לקיים תקשורת מאובטחת. נניח שהסיטואציה היא זו שבה אליס רוצה ליזום את התקשורת עם בוב. כדי לאפשר זאת, בוב משתמש בשיטת הצפנה שכוללת **שני** מפתחות שונים, K_{public} ו־$K_{private}$. את המפתח הפומבי K_{public} הוא מפרסם לכלל העולם, ואילו את המפתח הפרטי $K_{private}$ הוא שומר בסוד אך ורק לעצמו.

כאשר אליס רוצה להצפין הודעה M ולשלוח לבוב, היא משתמשת בפונקציה $C=\text{encrypt}\left(M,K_{public}\right)$. התוצאה היא כתב סתר C שגם אליס עצמה אינה יודעת איך לפענח (אם למשל היא שכחה את M פתאום). אליס שולחת את C אל בוב, שמפענח אותו באמצעות פונקציה $M=\text{decrypt}\left(C,K_{private}\right)$ שמתבססת על המפתח הפרטי שלו.

ניתן לחשוב על הסיטואציה באופן ציורי כך:

בהצפנה "רגילה", יש לאליס ובוב מנעולים שננעלים ונפתחים באמצעות אותו מפתח כך שלאליס יש עותק אחד של המפתח ולבוב עותק אחר. כאשר אליס רוצה לשלוח לבוב הודעה היא שמה אותה בקופסה, ומשתמשת במפתח שלה כדי לנעול את

הקופסה באמצעות המנעול שמשותף לה ולבוב. הקופסה עוברת מאליס אל בוב ואיב לא מסוגלת לפתוח אותה מכיוון שאין לה את המפתח; כשהיא מגיעה אל בוב, הוא פותח אותה עם המפתח שבצד שלו.

לעומת זאת, בהצפנה פומבית בוב דואג מבעוד מועד לשים בכיכר העיר קופסה שמלאה בעותקים זהים של מנעול קפיצי. אליס לוקחת לעצמה אחד מהמנעולים ומשתמשת בו כדי לנעול את הקופסה; מרגע זה ואילך היא אינה יכולה לפתוח את הקופסה בעצמה. היא שולחת אותה אל בוב, שפותח אותה בעזרת המפתח הפרטי שלו. בתיאור זה, המפתח הפומבי הוא למעשה המנעול עצמו.

על פי רוב השימוש של הצפנת מפתח פומבי היא ביצירת קשר ראשוני בין שני הצדדים וניהול פרוטוקול שבסופו מיוצר מפתח הצפנה אקראי ששני הצדדים מחזיקים, ומרגע זה ואילך התקשורת נמשכת באמצעות שיטת הצפנה רגילה, מהירה יותר. לכאורה בזאת נפתרה בעיית שיתוף המפתחות של המערכות הקלאסיות, אך בפועל עדיין קיימים היבטים נוספים שיש להתייחס אליהם (אם בוב שם ארגז מלא מנועלים בכיכר העיר מה מונע מאיב להכניס פנימה מנעול משל עצמה ולפתוח את הקופסה שאליס שולחת?) אך לא ניכנס אליהם - ואל הפתרונות הקיימים אליהם - כאן, אלא נסתפק בהצגת מערכת הצפנה באמצעות מפתח ציבורי.

7.5.2 שיטת ההצפנה RSA

הרעיון בשיטת RSA הוא ביצוע הצפנה ופענוח באמצעות העלאה בחזקה מודולרית. המודולוס N והחזקה e שבה מעלים על מנת להצפין נבחרים באקראי ומתפרסמים בפומבי; החזקה d שבה יש להעלות כדי לפענח הודעה מוצפנת ניתנת לחישוב מתוך $e,\varphi(N)$. בטיחות מערכת ההצפנה מסתמכת, אם כן, על שני דברים:

- הקושי של היפוך פעולת ההעלאה בחזקה מודולרית.
- הקושי של חישוב $\varphi(N)$ מתוך N.

נעסוק בהמשך בפירוט רחב יותר בהנחות אלו.

ראינו כבר כי חישוב $\varphi(N)$ הוא משימה קלה אם ידועים הגורמים הראשוניים של N. בפרט, במקרה הפשוט שבו N הוא מכפלה של זוג ראשוניים $N=pq$ הרי ש־$\varphi(N)=(p-1)(q-1)$. על כן, היתרון של בוב על פני יתר העולם יהיה שהוא יבנה את N באופן הבא: יגריל p,q ראשוניים ויחשב מתוכם את $N=pq$. את e הוא יכול להגריל באופן כמעט חופשי. לאחר מכן הוא ישתמש בהיכרות שלו עם p,q על מנת למצוא d שהופך את פעולת ההצפנה; לאחר מכן די לו לשמור את d אצלו ואין לו צורך שמור את p,q (אבל אסור לגלות אותם לעולם!).

אם כן, בהינתן N,e,d, מערכת ההצפנה כוללת את הפונקציות הבאות:

- **הצפנה**: $\text{encrypt}\left(M,(e,N)\right)=M^e \bmod N$
- **פענוח**: $\text{decrypt}\left(C,(d,N)\right)=C^d \bmod N$

כיצד על d להיבחר כדי שיתקיים $\text{decrypt}\left(\text{encrypt}\left(M,(e,N)\right),(d,N)\right)=M$?

אנו מעוניינים שתתקיים המשוואה הבאה: $(M^e)^d\equiv_N M$, כלומר $M^{ed-1}\equiv_N 1$. על פי משפט אוילר, $M^{\varphi(N)}\equiv_N 1$, ולכן אם $\varphi(N)\,|ed-1$, נקבל

$$\begin{aligned}M^{ed-1}&=M^{k\varphi(N)}\\&=\left(M^{\varphi(N)}\right)^k\\&\equiv_N 1^k=1\end{aligned}$$

על מנת שיתקיים $\varphi(N)\,|ed-1$ צריך שיתקיים $ed\equiv_{\varphi(N)}1$, דהיינו d נבחר להיות ההופכי המודולרי של e מודולו $\varphi(N)$.

את בניית מערכת ההצפנה ופונקציות ההצפנה והפענוח (הזהות) אפשר לכתוב כך בשפת Python:

```
def generate_rsa_cryptosystem(digits):
    p = generate_prime(digits // 2)
    q = generate_prime(digits // 2)
```

```
    N = p*q
    for _ in range(10):
        e = random.randrange(2,N-1)
        d = modular_inverse(e, (p-1)*(q-1))
        if d is not None:
            return ((N,e),(N,d))
    return None
```

```
def rsa_encrypt(M, public_key):
    N,e = public_key
    return modular_power(M, e, N)
```

```
def rsa_decrypt(C, private_key):
    N,d = private_key
    return modular_power(C, d, N)
```

7.5.3 שיטת RSA-CRT

שיטת RSA-CRT באה לשפר את זמני הפענוח של הודעת RSA מוצפנת בערך פי 4, ובכך לסייע להגדלת הקיבולת של מערכות שצריכות להתמודד עם כמות גדולה של פענוחים בו זמנית. למשל: חנות אלקטרונית שמאות אלפי לקוחות מנסים להתחבר אליה בו זמנית; כל לקוח יכול להרשות לעצמו זמן חישוב גדול של הצפנת ה־RSA שאיתה הוא יוזם את הפניה לחנות, אבל החנות עצמה צריכה להתמודד עם כמות גדולה של פניות בבת אחת ולכן זקוקה לפענוח מהיר.

בשיטת RSA-CRT בניית מערכת ההצפנה מתחילה באופן הרגיל:

1. הגרילו שני מספרים ראשוניים גדולים דיו p, q.
2. חשבו $N = pq$
3. הגרילו $e \in \mathbb{Z}_N^*$.
4. חשבו $d \in \mathbb{Z}_N^*$ כך ש־$ed \equiv_{\varphi(N)} 1$.

ב־RSA רגיל, בשלב זה p, q "ייזרקו לפח" אולם ב־RSA-CRT אנו שומרים אותם, ובנוסף לכך מחשבים:

$d_p = d \bmod (p-1)$

$d_q \equiv d \bmod (q-1)$

המפתח הפומבי יהיה כמקודם (N, e) ואילו המפתח הפרטי יהיה (p, q, d_p, d_q).

הצפנה מבוצעת כמקודם: $\text{encrypt}(M, (e, N)) = M^e \bmod N$.

פענוח מתבצע באופן הבא. ראשית מחשבים:

$M_p = C^{d_p} \bmod p$

$M_q = C^{d_q} \bmod q$

כעת משתמשים במשפט השאריות הסיני כדי למצוא את ה־M היחיד מודולו $N = pq$ המקיים:

$M \equiv_p M_p$

$M \equiv_q M_q$

עלינו להראות כי בצורה זו הפענוח משחזר את ההודעה המקורית. נניח אם כן כי $C = M^e \bmod N$. אז בפרט מתקיים

$M_p \equiv_p M^{ed_p}$

$M_q \equiv_q M^{ed_q}$

כעת נוכיח כי $M \equiv_p M^{ed_p}$: לשם כך די להראות כי $M^{ed_p - 1} \equiv_p 1$. מכיוון ש־$M^{\varphi(p)} \equiv_p 1$, די להראות כי $\varphi(p) | ed_p - 1$, כלומר ש־$ed_p \equiv_{\varphi(p)} 1$.

כזכור, $\varphi(p) = p - 1$.

כמו כן $d \equiv_{p-1} d_p$ ו־$ed \equiv_{\varphi(n)} 1$, כלומר בפרט $ed \equiv_{p-1} 1$

מכל אלו נסיק: $ed_p \equiv_{p-1} ed \equiv_{p-1} 1$ כנדרש.

באותו האופן ניתן להראות כי $M \equiv_q M_q$. משני אלו נובע ש־M הוא אכן פתרון של מערכת המשוואות

$x \equiv_p M_p$
$x \equiv_q M_q$
ומכיוון שהפתרון הוא יחיד מודולו N, אז החישוב שאנו מבצעים באמצעות משפט השאריות הסיני אכן מחזיר את M.

8 מבוא לתורת הסיבוכיות

8.1 המחלקות P ו־NP

תורת הסיבוכיות עוסקת במודלים חישוביים שונים ומשונים ובכוחם החישובי ־ בבעיות שהם יכולים לפתור ואלו שאינם יכולים לפתור. מכיוון שבתחום זה עדיין רב הנסתר על הגלוי, לעתים קרובות תוצאות בתורת הסיבוכיות הן יחסיות: למשל, הוכחה ששני מודלים חישוביים שונים הם שקולים בכוחם, או הוכחה שאם בעיה מסויימת ניתנת לפתרון במודל חישובי מסויים אז הדבר מוכיח את השקילות של מודלים אחרים, וכדומה.

על מנת להגדיר מודל חישובי קונקרטי יש לתת הגדרה מתמטית מדויקת, ולצורך כך נעזרים לעתים קרובות במודל שנקרא **מכונת טיורינג** אך לא נציג כאן בצורה מפורשת. עבור מה שנעשה כאן די לחשוב על שפת תכנות קונקרטית, דוגמת Python, כמגדירה מודל חישובי, כך שכל תוכנית מחשב בשפה זו שייכת למודל החישובי. בפועל כל תוכנית בשפת פייתון ניתנת למימוש גם באמצעות מכונת טיורינג, כאשר ה"מחיר" הוא שעל כל פקודה בשפת Python, כמות הצעדים שמכונת הטיורינג תזדקק לה כדי לממש פקודה זו הוא **פולינומי** בגודל הקלט של המכונה.

נגדיר במדויק מושג זה:

הגדרה 8.1 נאמר שחישוב כלשהו מתבצע **בזמן פולינומי** אם הוא מתבצע בסיבוכיות זמן $O\left(n^{k}\right)$ עבור $k \ge 0$ קבוע כלשהו.

לחישובים פולינומיים יש את התכונה המועילה ש**הרכבה** שלהם היא פולינומית; חישוב שמתבצע בזמן פולינומי יכול להוציא פלט שהוא לכל היותר פולינומי בגודל הקלט המקורי, ואם פלט זה מועבר לחישוב פולינומי שני גודל הפלט שהחישוב השני ייתן יהיה פולינומי בגודל הקלט המקורי. תכונה זו עומדת בבסיס הבחירה להגדיר **חישוב יעיל** בתור חישוב פולינומי, אף שלכאורה זו הגדרה רחבה **מדי** שכן יש חישובים פולינומיים שכלל לא נראים לנו יעילים (למשל אלגוריתם שדורש n^{10000} צעדים). האינטואיציה היא שהמטרה המרכזית שלנו בתורת הסיבוכיות היא לאפיין בעיות ש**אין להן פתרון יעיל**. בפרט אי־קיום של אלגוריתם פולינומי יגרור אי־קיום של פתרון יעיל (תחת הסתייגויות שנתייחס אליהן בהמשך).

רוב הבעיות האלגוריתמיות מנוסחת בתור **בעיות חיפוש**: בהינתן קלט, למצוא פלט המותאם לקלט זה ומקיים **אילוצים** מסויימים. למשל: מציאת התמורה הממיינת של סדרה; מציאת מסלול קל ביותר בין שני צמתים בגרף; מציאת זרימת מקסימום בגרף. לפעמים הפלט האפשרי הוא יחיד ולעתים יש כמה פלטים אפשריים. לבעיות אלו קוראים **בעיות חיפוש** שכן אנו "מחפשים" במרחב הפלטים האפשריים (למשל, מרחב התמורות האפשריות של סדרה) את זה שעונה על התנאים הדרושים לנו (למשל, שהסדרה אחרי הפעלת התמורה תהיה ממויינת).

סוג נוסף של בעיות אלגוריתמיות הוא **בעיות הכרעה**. בהינתן קלט כלשהו, התשובה היא "כן/לא" בלבד, בהתאם לשאלה האם הקלט עונה לקריטריונים מסויימים. למשל, האם גרף מסויים הוא קשיר; האם רשימה היא ממויינת; האם קיים מסלול מאורך לכל היותר 7 בין זוג צמתים נתונים בגרף, וכדומה.

בגלל הפשטות היחסית של הפלט בבעיות הכרעה, הגדרות רבות בתורת הסיבוכיות עוסקות בהן. ההרחבה לבעיות חיפוש ולחישובי פונקציות באופן כללי היא לרוב פשוטה יחסית אחרי שמחלקות הסיבוכיות הרלוונטיות הוגדרו.

עבור בעיות הכרעה, נהוג להשתמש בטרמינולוגיה של **שפה**: שפה L היא אוסף של מילים, כאשר כל מילה היא סדרה סופית של איברים מתוך קבוצה סופית Σ שמכונה **אלפבית**. כל האובייקטים שאנו עוסקים בהם בדרך כלל במסגרת אלגוריתם כלשהו ניתנים לקידוד באמצעות סדרות בינאריות של תווים, כך שהטרמינולוגיה לא מגבילה אותנו.

ב־Σ^* נסמן את אוסף כל המילים מעל Σ, כלומר אוסף הסדרות הסופיות עם איברים מתוך Σ. כעת נוכל להגדיר פורמלית בעיית הכרעה:

הגדרה 8.2 בהינתן שפה L, **בעיית ההכרעה** שמוגדרת על ידי L היא הבעיה הבאה: בהינתן מילה $w \in \Sigma^*$, יש לקבוע האם $w \in L$ או $w \notin L$. נאמר שאלגוריתם A **מקבל** מילה w אם A מסיים את ריצתו על w עם החזרת הפלט "כן", ונאמר ש־A **דוחה** מילה w אם הוא מסיים את ריצתו עליה עם הפלט "לא". נאמר ש־A **מכריע** את השפה L אם לכל $w \in L$, A מקבל את w, ולכל $w \notin L$, A דוחה את w.

כדוגמא פשוטה, אפשר לחשוב על L בתור שפת כל הסדרות הממויינות של מספרים טבעיים, כאשר איבר לדוגמא בשפה זו הוא המילה $[2,5,17]$ (הסוגריים המרובעים והפסיקים הם חלק מהאלפבית במקרה זה). בדיקת שייכות ל־L היא פשוטה: בהינתן w, האלגוריתם עובר סדרתית על התווים בו וממיר אותם למספרים, ובכל פעם שהתקבל מספר חדש בודק אם הוא גדול מקודמו. אם לא ־ האלגוריתם עוצר ועונה "לא" ואחרת בסוף הרשימה הוא עוצר ועונה "כן".

סיבוכיות זמן הריצה של האלגוריתם הזה היא פולינומית ב־$|w|$.

זה מוביל אותנו להגדרה המרכזית הראשונה שלנו: מחלקת הסיבוכיות P. כזכור, אלגוריתם A הוא **פולינומי** אם קיים k כך שלכל קלט w, זמן הריצה של A על w הוא $O\left(|w|^k\right)$.

הגדרה 8.3 נאמר ששפה L שייכת למחלקת הסיבוכיות P אם קיים אלגוריתם פולינומי A שמכריע את L.

כל הבעיות שעסקנו בהן בקורס הזה נפתרו בזמן פולינומי ולכן בניסוח שלהן כבעיות הכרעה ("האם ברשת הזרימה הנתונה קיימת זרימה שערכה לפחות 73?") כולן שייכות ל־P. נציג כעת בעיית הכרעה שאיננו יודעים אם היא שייכת ל־P או לא:

הגדרה 8.4 בהינתן גרף G (שיכול להיות מכוון או לא מכוון), **מסלול המילטוני** ב־G הוא מסלול ב־G כך שכל צומת ב־V מופיע בו בדיוק פעם אחת. **מעגל המילטוני** הוא מעגל ב־G שכל צומת ב־V מופיע בו בדיוק פעם אחת למעט צומת התחלת המעגל ששווה לצומת סיום המעגל.

ההגדרה של מסלול/מעגל המילטוני מזכירה את זו של מסלול/מעגל **אוילרי**. עבור מסלול/מעגל אוילרי, הדרישה היא לעבור בכל **קשתות** G בדיוק פעם אחת; עבור מסלול/מעגל המילטוני הדרישה היא לעבור בכל **צמתי** G בדיוק פעם אחת.

הגדרה 8.5 בעיית המסלול ההמילטוני HL הוא שפת כל הגרפים הלא מכוונים G בעלי מסלול המילטוני. בדומה מגדירים את DHL עבור גרפים מכוונים עם מסלול המילטוני, ואת HC ו־DHC עבור גרפים לא מכוונים/מכוונים עם מעגל המילטוני.

כאמור, איננו יודעים אם HL (או יתר הבעיות שהגדרנו) שייכת ל־P וכפי שנראה בהמשך, ההשערה שלנו היא שאינה שייכת (וזאת בניגוד חריף לבעיית המסלול/מעגל האוילרי, שקיים לה פתרון פולינומי פשוט). עם זאת, יש לבעיה הזו אספקט פשוט אחד: קל לנו **לבדוק** שמסלול נתון הוא המילטוני. כלומר, אם אנו נתקלים בפתרון של הבעיה, קל לנו **לוודא** את נכונות הפתרון. תכונה זו חשובה דיו כדי להצדיק הגדרה פורמלית:

הגדרה 8.6 אלגוריתם A **מוודא** שפה L אם מתקיימים שני התנאים הבאים:

- אם $w\in L$ אז **קיים** $\pi\in\Sigma^*$ כך ש־A מקבל את הקלט (w,π)
- אם $w\notin L$ אז **לכל** $\pi\in\Sigma^*$ A דוחה את הקלט (w,π)

עבור בעיית המסלול ההמילטוני, A יכול להיות אלגוריתם שמקבל את הקלט (G,π) כך ש־$\pi=[v_1,v_2,\ldots,v_k]$ היא רשימת צמתים. A עובר על הרשימה, מוודא שכל $v\in V$ מופיע ברשימה בדיוק פעם אחת; שכל צומת מ־π אכן מופיע ב־V; ושהקשת $v_i\to v_{i+1}$ שייכת ל־E לכל $1\le i<k$. אם π עבר את כל המבחנים הללו, A יקבל את הקלט.

כעת, אם G הוא גרף בעל מסלול המילטוני π אז A יקבל את הקלט (G,π); לעומת זאת אם G אינו בעל מסלול המילטוני, אז A ידחה כל קלט (G,π) כי המבחנים של A יכולים לעבור רק במקרה שבו π הוא מסלול המילטוני. המסקנה היא ש־A מוודא את השפה HL (באופן דומה ניתן לוודא את השפות HC,DHL,DHC).

אנו רוצים לעסוק במקרים שבהם תהליך הוידוא מבוצע ביעילות, כלומר בזמן פולינומי, אולם נשאלת השאלה ־ פולינומי **במה?** אם A צריך להיות פולינומי ב־$|(w,\pi)|$ אז "הוכחה" π יכולה להיות פשוט סדרה ארוכה מאוד של 1־ים כך ש־A יוכל לבצע חישוב ארוך ומסובך לבדיקת w וחישוב זה ייחשב "פולינומי" כי הוא פולינומי באורך π. על כן, נניח כי A פולינומי ב־$|w|$. נשים לב כי במקרה זה, אם π היא מאורך גדול כל כך עד ש־A אינו יכול לקרוא את כל π תוך עמידה במגבלת הזמנים, הרי שאפשר "לקצר" את π ולהשאיר רק את החלק שאותו A מסוגל לקרוא.

כעת ניתן להגדיר את המחלקה המתאימה לתכונת ה"קל לבדוק":

הגדרה 8.7 נאמר ששפה L שייכת למחלקה NP אם קיים אלגוריתם A פולינומי ב־$|w|$ שמוודא את L.

ה־P בשמות המחלקות P,NP הוא מהמילה Polynomial. ה־N שב־NP בא לציין "Nondeterministic" שמתייחס להגדרה אחרת, מיושנת יותר, של מחלקה זו; לא נעסוק בהגדרה זו כאן.

8.2 שאלת $\mathrm{P}=\mathrm{NP}$

אחת מהשאלות הפתוחות המרכזיות במדעי המחשב התיאורטיים, אם לא המרכזית שבהן, היא השאלה האם $\mathrm{P}=\mathrm{NP}$. כיוון אחד של הכלה הוא קל: אם $L\in\mathrm{P}$ עם אלגוריתם A שמכריע אותה, אז אותו A גם מוודא את אותה שפה (כאשר חושבים עליו ככזה שעל הקלט (w,π) פשוט מתעלם מה־π ומבצע את החישוב על w כרגיל) ולכן $\mathrm{P}\subseteq\mathrm{NP}$. מכאן שאת שאלת $\mathrm{P}=\mathrm{NP}$ אפשר לנסח בתור השאלה "האם כל שפה שניתנת לוידוא ביעילות ניתנת גם להכרעה ביעילות".

על פניו, המשמעות של קיום השוויון $\mathrm{P}=\mathrm{NP}$ היא מרחיקת לכת. למשל, רוב שיטות ההצפנה הקיימות הן כאלו שבהן בהינתן מפתח K קל לבדוק אם הוא אכן המפתח הנכון (פשוט מנסים לפענח את ההודעה המוצפנת ורואים אם התוצאה הגיונית; קיימות שיטות הצפנה כדוגמת "פנקס חד פעמי" שבהן גישה זו לא תעבוד) ולכן $\mathrm{P}=\mathrm{NP}$ משמעותו שקל **למצוא** מפתח (כלומר, לשבור את ההצפנה) עבור שיטות אלו. השלכה דומה יש על שלל בעיות שונות ומשונות (מכיוון שקל לבדוק את המבחן באלגוריתמים קומבינטוריים נובע מכך שלפתור את המבחן באלגוריתמים קומבינטוריים זה גם קל). עם זאת, יש להימנע מייחוס משמעויות מופרזות לשוויון הזה; אם בעיה ניתנת לוידוא בזמן $O\left(n\right)$ אבל להכרעה בזמן $O\left(n^{10000}\right)$ כנראה שלא תצא לנו שיטה פרקטית לשבירת צפנים מכך.

האמונה הרווחת בקרב העוסקות והעוסקים במדעי המחשב היא ש־$\mathrm{P}\neq\mathrm{NP}$. כפי שנראה בקרוב, קיימות אלפי בעיות שונות ומשונות מתחומים רבים של מדעי המחשב שפתרון יעיל עבור **אחת** מהן יוכיח ש־$\mathrm{P}=\mathrm{NP}$, אולם למרות האינטרס הגדול לפתור בעיות אלו והשיטות הפרקטיות הרבות שהוצעו כדי להתמודד איתן, לא נמצא אלגוריתם יעיל עבור אף אחת מהן.

למרות זאת, אף שהבעיה תוארה כבר לפני כ־50 שנים, טרם נמצאה הוכחה לכך ש־$\mathrm{P}\neq\mathrm{NP}$. התחושה היא שהטכניקות המתמטיות הקיימות כיום בתורת הסיבוכיות אינן חזקות מספיק כדי להוכיח טענה כמו זו (כך למשל מאמר של Baker-Gill-Solovay משנת 1975 הראה ששיטות מקובלות בתחום כדוגמת **לכסון** שנראה בהמשך לא יכולות לעבוד). במידה מסויימת זה מזכיר את גורלן של בעיות קשות במתמטיקה כדוגמת המשפט האחרון של פרמה, שהיה קל מאוד לניסוח אך הוכח רק לאחר 350 שנים בעזרת טכניקות מתמטיות מתקדמות ביותר שלא היו קיימות בזמנו של פרמה.

ניסוח שקול של $\mathrm{P}=\mathrm{NP}$, שממחיש את הכוח הרב שהשוויון הזה טומן בחובו, הוא "זיהוי יעיל גורר חיפוש יעיל":

משפט 8.8 אם $\mathrm{P}=\mathrm{NP}$ אז לכל שפה $L\in\mathrm{NP}$ ומוודא A עבורה קיים אלגוריתם פולינומי שבהינתן $w\in L$ מחזיר $\pi\in\Sigma^{*}$ כך ש־A מקבל את (w,π).

על מנת להבין את משמעות המשפט, נתבונן במשמעותו עבור שפת המסלול ההמילטוני HL שראינו. אם ניקח בתור A את המוודא שראינו קודם, אז המשפט אומר שאם $\mathrm{P}=\mathrm{NP}$ הרי שבהינתן G, קיים אלגוריתם שמוצא ביעילות מסלול המילטוני עבור G. כלומר, היכולת שלנו **לזהות** מתי π הוא "פתרון נכון" עבור w מתורגמת ליכולת שלנו **למצוא** π כזה.

כיצד ניתן להוכיח את המשפט עבור המקרה הפרטי של מסלול המילטוני? נגדיר שפה חדשה, Partial-HL, שהמילים בה הן זוגות של גרף G והתחלה של מסלול המילטוני עבור G ־ כלומר, מסלול ב־G שעובר בכל צומת של G לכל היותר פעם אחת, וניתן **להמשיך אותו** כדי לקבל מסלול המילטוני ב־G (אבל הוא עצמו לא בהכרח מכיל מספיק צמתים כדי להיות מסלול המילטוני). השפה הזו בבירור שייכת ל־NP כי בהינתן זוג של גרף ומסלול, ה־π עבור המוודא יכלול את המשך המסלול. אם כן, מכך ש־$\mathrm{P}=\mathrm{NP}$ נסיק ש־$\text{Partial-HL}\in\mathrm{P}$.

כעת, בהינתן גרף G, נבנה מסלול המילטוני עבורו כך: נבחר צומת $v\in V$ באופן שרירותי ונתחיל את המסלול ממנו. נבדוק אם G יחד עם המסלול שכבר בנינו שייך ל־Partial-HL; אם כן, זה היה רעיון טוב להתחיל עם v כי אפשר להמשיך עם המסלול שבנינו עד שנגיע למסלול המילטוני. אם לא, נעבור לבחור איבר אחר של V ונבדוק עליו. כך נעבור לכל היותר על כל אברי V ובסיום איטרציה זו או שנדע שאין מסלול המילטוני ב־G או שנדע **בוודאות** על צומת v שמתחיל מסלול המילטוני.

כעת ניקח את המסלול שהתחלנו לבנות, ונוסיף לו את הצומת הבא באותו האופן ־ מבין הצמתים שנותרו, נעבור אחד אחד ונבדוק איזה מהם, לאחר שהוא מתווסף למסלול, מותיר אותנו ב־Partial-HL, וכן הלאה.

בצורה זו נדרשות $|V|$ איטרציות, שבכל אחת מהן $|V|$ צעדים לכל היותר, כך שכל צעד כולל בדיקת שייכות ל־Partial-HL ־ פולינומי בסך הכל.

נעבור כעת להוכחה הכללית של המשפט, שמתבססת על אותה הטכניקה בדיוק, בניסוח פורמלי יותר. **הוכחה:** בהינתן L עם מוודא A פולינומי, נגדיר שפה $L^{\prime}=\left\{ \left(w,\pi^{\prime}\right)\ |\ \exists\pi^{\prime\prime}\in\Sigma^{*}:A\left(w,\pi^{\prime}\pi^{\prime\prime}\right)=\mathrm{True}\right\}$.

$L^{\prime}\in\mathrm{NP}$ שכן קיים לה מוודא $A^{\prime}$ שפועל כך: על קלט $\left(\left(w,\pi^{\prime}\right),\pi^{\prime\prime}\right)$, מריץ את A על $\left(w,\pi^{\prime}\pi^{\prime\prime}\right)$ ועונה כמוהו.

מכיוון שהנחנו ש־$\mathrm{P}=\mathrm{NP}$, נקבל ש־$L^{\prime}\in\mathrm{P}$.

נציג כעת אלגוריתם פולינומי שבהינתן $w\in L$ מחזיר $\pi\in\Sigma^{*}$ כך ש־A מקבל את (w,π):

1. נאתחל $\pi=\varepsilon$ (המילה הריקה, סדרה ריקה של תווים).
2. כל עוד A אינו מקבל את (w,π):
   (א) לכל $\sigma\in\Sigma$:

i. נבדוק האם $(w,\pi\sigma)\in L'$
ii. אם כן, נגדיר $\pi\leftarrow\pi\sigma$

(ב) אם הבדיקה לעיל נכשלה לכל $\sigma\in\Sigma$ נחזיר None

3. נחזיר את π.

ניתן להראות שאלגוריתם זה מחזיר π כמבוקש. ■

8.3 רדוקציות פולינומיות

איננו מכירים פתרון יעיל ל־HL. אבל **נניח** שהיינו מכירים כזה ־ האם היינו יכולים להיעזר בפתרון זה כדי לפתור בעיות נוספות? ראינו דוגמאות לכך במהלך הקורס; היכולת לבצע DFS ביעילות תורגמה באופן מיידי ליכולת לבצע מיון טופולוגי ביעילות; היכולת לבצע BFS ביעילות תורגמה ליכולת למציאת זרימת מקסימום ביעילות (בסיוע אלגוריתם אדמונדס־קארפ), ויכולת זו תורגמה ליכולת למצוא שידוך מקסימום בגרף.

במקרה של שימוש ב־BFS, השתמשנו ב־BFS בתור "קופסה שחורה" שנקראת מספר פעמים במהלך ריצת אדמונדס־קארפ. במקרה של שידוך מקסימום, עשינו דבר פשוט יותר: בהינתן קלט בעיית שידוך מקסימום (כלומר, גרף דו צדדי) **המרנו** את הקלט הזה לקלט לבעיית זרימת מקסימום, פתרנו את בעיית זרימת המקסימום ואז "תרגמנו חזרה" את הפתרון שמצאנו אל פתרון של בעיית שידוך מקסימום.

שתי גישות אלו שימושיות בתורת הסיבוכיות. הראשונה, גישת ה"קופסה שחורה" מתוארת באמצעות המושג הפורמלי של **אורקל** ולא נעסוק בה כאן אף שיש בה עניין רב. השניה, של "המרת קלט של בעיה אחת לקלט של בעיה אחרת" מתוארת באמצעות המושג הפורמלי של **רדוקציית העתקה** או בקיצור "רדוקציה", והיא הכלי שנעסוק בו כאן.

הגדרה 8.9 בהינתן שפות L_1,L_2, **רדוקציה** מ־L_1 אל L_2 היא פונקציה $f:L_1\to L_2$ הניתנת לחישוב אלגוריתמי, כך שמתקיים $x\in L_1 \iff f(x)\in L_2$. אם f ניתנת לחישוב בזמן פולינומי אומרים ש־f היא **רדוקציה פולינומית**. אם קיימת רדוקציה מ־L_1 אל L_2 מסמנים זאת $L_1\le L_2$ ואם הרדוקציה היא פולינומית מסמנים זאת $L_1\le_p L_2$.

הרעיון ברדוקציה הוא להיעזר בפתרון של L_2 כדי לפתור את L_1. לשם כך ממירים את הבעיה ב־L_1 (שמסומנת ב־x) לבעיה $f(x)$ של L_2. נשים לב לכך שאין שום צורך ש־f תהיה על, כלומר אין צורך שהרדוקציה שלנו תיעזר ב"מלוא הכוח" של השפה L_2; ואכן, לעתים קרובות התוצרים של הרדוקציה ייראו כמו מקרים מאוד ספציפיים של L_2.

טענה 8.10 אם $L_2\in\mathrm{P}$ וגם $L_1\le_p L_2$ אז $L_1\in\mathrm{P}$

הוכחה: נניח כי A הוא אלגוריתם פולינומי המכריע את L_2. נכריע את L_1 כך: בהינתן $x\in\Sigma^*$ נחשב את $f(x)$ בזמן פולינומי, נריץ את A על $f(x)$ ונענה כמוה. מכיוון ש־$x\in L_1 \iff f(x)\in L_2$ נקבל ש־$x\in L_1$ אם ורק אם A ענתה "כן", מה שמראה את נכונות הפתרון. כמו כן, זמן הריצה של A על $f(x)$ הוא פולינומי ב־$|f(x)|$ ומכיוון ש־$|f(x)|$ הוא פולינומי ב־$|x|$ נקבל שזמן הריצה של A הוא פולינומי ב־$|x|$. ■

טענה זו מסבירה את הסימון $\le$: אם $L_1\le_p L_2$ אז בכל הנוגע לחישוב פולינומי, L_2 היא בעיה "קשה לפחות כמו" L_1 כי אם ניתן לפתור את L_2 בזמן פולינומי, ניתן לפתור גם את L_1. כמובן, ניתן להפוך את התוצאה החיובית על פיה כדי לקבל תוצאה שלילית:

טענה 8.11 אם $L_1\notin\mathrm{P}$ וגם $L_1\le_p L_2$ אז $L_2\notin\mathrm{P}$

בצורה זו רדוקציה יכולה לשמש כדי להוכיח אי־שייכות של שפה למחלקה כלשהי ־ מראים רדוקציה **אל** השפה משפה שהקושי שלה כבר הוכחץ

נראה דוגמא לרדוקציה בכך שנראה $\mathrm{HC}\le_p\mathrm{HL}$. כלומר, בהינתן גרף לא מכוון G נראה איך לבנות ממנו גרף G' כך שב־G יש **מעגל** המילטוני אם ורק אם ב־G' יש **מסלול** המילטוני.

אם נגדיר סתם $G'=G$, הרדוקציה לא תעבוד. אמנם, אם $G\in\mathrm{HC}$ אז בפרט $G'\in\mathrm{HL}$ כי כל מעגל המילטוני הוא בפרט מסלול המילטוני, אבל בהחלט ייתכן ש־$G'\in\mathrm{HL}$ אבל אין ב־G מסלול המילטוני (למשל, גרף "שרוך").

ננקוט אם כן בתעלול הבא: אם ב־G יש מעגל, אז כל צומת בגרף יכול לשמש בתור נקודת ההתחלה/סיום של המעגל. נבחר צומת שרירותי $v\in V$ ונפצל אותו לשניים, v_1,v_2 כך שלכל $v\neq u\in V$ מתקיים $(v,u)\in E \iff (v_i,u)\in E'$.

כעת נרצה "להכריח" כל מסלול המילטוני בגרף החדש להתחיל ב־v_1 ולהסתיים ב־v_2 או ההפך; לצורך כך נוסיף שני צמתים

w_1, w_2 ואת הקשתות $(w_1, v_1), (w_2, v_2) \in E'$. פרט אליהן w_1, w_2 לא מחוברים לקשתות נוספות ומכיוון שמסלול המילטוני חייב לעבור בצמתים אלו הוא חייב להתחיל ולהסתיים בהם.

כעת, אם מצאנו ב־G' את המסלול ההמילטוני $w_1 \to v_1 \to \ldots \to v_2 \to w_2$ אז על ידי הסרת w_1, w_2 מהמסלול והחלפת v_1, v_2 ב־v נקבל מעגל המילטוני ב־G; ובדומה אפשר להמיר מעגל המילטוני ב־G במסלול המילטוני ב־G', מה שמוכיח את נכונות הרדוקציה. פולינומיות הרדוקציה נובעת מהשינוי הפשוט שביצענו ־ בחירת צומת, הסרה שלו, הוספת כמה צמתים חדשים וקשתות שמחוברות אליהם דורשים כולם זמן פולינומי.

8.4 בעיות NP־שלמות

הרדוקציה $\mathrm{HC} \le_p \mathrm{HL}$ שהצגנו הייתה "יצירתית" ־ השתמשנו בתעלול כלשהו עבורה שהתבסס בצורה חזקה על המאפיינים של שתי הבעיות שביניהן עשינו רדוקציה. האם יש משהו כללי יותר שאפשר לעשות? האם יש דרך שיטתית כלשהי לבצע רדוקציה בין שפות?

תוצאה מפתיעה ומעניינת במיוחד היא שאכן יש דרך כזו עבור שפות ב־NP (וגם עבור מחלקות נוספות של סיבוכיות שלא נעסוק בהן). נפתח עם הגדרה:

הגדרה 8.12 שפה L נקראת **NP־קשה** אם לכל $L' \in \mathrm{NP}$ קיימת רדוקציה $L' \le_p L$. אם בנוסף $L \in \mathrm{NP}$ אז L נקראת **NP־שלמה**

השפה הראשונה שעליה הוכח כי היא NP־שלמה היא השפה SAT שנציג כעת, ודורשת מושגים בסיסיים מתחשיב הפסוקים בלוגיקה:

הגדרה 8.13 (פסוקי CNF)

- **משתנה בוליאני** x הוא משתנה שיכול לקבל ערך T או F.
- **ליטרל** הוא משתנה x או שלילה $\neg x$ של משתנה.
- **פסוקית CNF** היא ביטוי מהצורה $C = (l_1 \lor l_2 \lor \ldots \lor l_k)$ כאשר כל l_i הוא ליטרל.
- **פסוק CNF** הוא ביטוי מהצורה $C_1 \land C_2 \land \ldots \land C_n$ כאשר כל C_i הוא פסוקית CNF.

הרעיון בפסוקי CNF הוא שניתן **להציב** במשתנים ערכי אמת:

הגדרה 8.14 **השמה** τ לפסוק CNF היא פונקציה $\tau : \{x_1, \ldots, x_n\} \to \{\mathrm{T,F}\}$ מקבוצת המשתנים שמופיעים בפסוק לקבוצת ערכי האמת האפשריים.

בהינתן השמה, ניתן לחשב את ערך האמת שהיא נותנת לפסוק כולו באופן הבא: לכל ליטרל, אם הוא מהצורה x אז הוא מקבל את ערך האמת שההשמה נתנה למשתנה x ואם הוא מהצורה $\neg x$ הוא מקבל את ערך האמת שההשמה לא נתנה ל־x. פסוקית CNF מקבלת ערך T אם לפחות אחד מהליטרלים בה קיבל T. פסוק CNF מקבל ערך T אם כל פסוקיות ה־CNF שלו קיבלו ערך T.

ניתן להוכיח כי כל טבלת אמת ניתנת למימוש באמצעות פסוק CNF אך לא נעשה זאת כאן. אנו מתעניינים בשאלה פשוטה יותר ־ האם פסוק ה־CNF שלנו הוא זהותית F או לא.

הגדרה 8.15 פסוק CNF הוא **ספיק** אם קיימת השמה כלשהי שנותנת לו את הערך T.

כך למשל הפסוק $(x_1 \lor x_2) \land (\neg x_1 \lor x_3)$ הוא ספיק על ידי ההשמה שנותנת T ל־x_1, x_3 ולא משנה מה ל־x_2. לעומת זאת הפסוק $(x_1 \lor x_2) \land (\neg x_1) \land (\neg x_2)$ אינו ספיק.

כעת ניתן להגדיר את השפה שלנו:

הגדרה 8.16 השפה SAT היא שפת כל פסוקי ה־CNF הספיקים.

קל לראות ש־SAT שייכת ל־NP: בהינתן פסוק φ, ה"הוכחה" π לשייכותו ל־SAT תהיה פשוט השמה שמספקת אותו. השמה שכזו היא בסך הכל סדרה של n ביטים כך ש־n הוא מספר המשתנים השונים שמופיעים ב־φ כך ש־$|\pi| \le |\varphi|$ ובפרט π פולינומי בגודלו ובדיקת ספיקות בהינתן ההשמה היא פולינומית.

הסיבה לחשיבות הגדולה של SAT היא המשפט הבא:

משפט 8.17 (משפט קוק־לוין): SAT היא NP־שלמה.

לא נוכיח משפט זה כאן. ההוכחה היא טכנית אך אינה קשה בצורה חריגה; אולם על מנת להוכיח בצורה פורמלית את המשפט אנחנו צריכים מודל פורמלי של חישוב, ונמנענו מלעשות זאת. הרעיון שמאחורי ההוכחה הוא זה: בהינתן שפה $L \in \mathrm{NP}$ כלשהי לוקחים מודל חישוב פשוט עבור ודוא שלה ־ **מכונת טיורינג**. אופן הפעולה של מכונת טיורינג הוא פשוט דיו עד כדי כך שניתן לייצג את כל המצבים האפשריים של המכונה במהלך כל החישוב בידי כמות לא גדולה של משתנים בוליאניים, ולכתוב פסוק שמוודא שהמשתנים הללו 1) מקודדים את המצב התחלתי של המכונה 2) מקודדים את ההגעה של המכונה למצב מקבל ו־3) מקודדים את החישוב שהמכונה מבצעת.

דהיינו, הוכחת המשפט מצביעה על דרך שיטתית לייצר רדוקציה מ־L אל SAT בהינתן שיש לנו מכונת טיורינג עבור L (קיימת כזו שכן $L \in \mathrm{NP}$). אפשר לחשוב עליו כאומר ש־SAT היא שפה "מורכבת מספיק" כדי שאפשר יהיה לקודד כל בעיה אחרת ב־NP באמצעות אובייקטים מהעולם ש־SAT עוסקת בו.

למרבה השמחה, מרגע שבידינו שפה NP־שלמה אחת, קל יותר להוכיח כי גם שפות אחרות הן כאלו:

משפט 8.18 אם L_1 היא שפה NP־קשה ו־$L_1 \le_p L_2$ אז גם L_2 היא NP־קשה.

הוכחה: תהא $L \in \mathrm{NP}$ כלשהי. אז $L \le_p L_1$ מכיוון ש־L_1 NP־קשה. תהא $f : L \to L_1$ הרדוקציה הפולינומית המתאימה ו־$g : L_1 \to L_2$ הרדוקציה הפולינומית המתאימה ל־$L_1 \le_p L_2$. אז $gf : L \to L_2$ היא רדוקציה פולינומית (הפולינומיות של זמן הריצה נובעת שוב מכך שהרכבת פולינומים היא פולינום). ■

כלומר, אם אנו רוצים להוכיח ששפה כלשהי ב־NP היא NP־שלמה, אנחנו מתחילים משפה NP־שלמה מוכרת ועושים רדוקציה **ממנה** אל השפה שאנחנו רוצים להוכיח את הקושי שלה (זו נקודה שקל להתבלבל בה ולנסות למצוא רדוקציה בכיוון השני, למרות שקיום רדוקציה כזו כבר מובטח לנו ממילא).

מה המשמעות של כך ששפה היא NP־שלמה? מסקנה אחת היא מיידית:

משפט 8.19 אם L היא NP־שלמה ו־$L \in \mathrm{P}$ אז $\mathrm{P} = \mathrm{NP}$.

הוכחה: לכל $L' \in \mathrm{NP}$ קיימת רדוקציה $L' \le_p L$ ולכן אם $L \in \mathrm{P}$ אז $L' \in \mathrm{P}$. ■

משמעות הדבר היא שאם $\mathrm{P} \ne \mathrm{NP}$ אז בודאות כל שפה NP־שלמה אינה ב־P; שפות אלו הן במובן מסויים "השפות הקשות ביותר ב־NP".

8.5 דוגמאות לשפות NP־שלמות

נציג כעת מספר דוגמאות לשפות NP־שלמות. נציג את השפות עצמן וסקיצה של רדוקציה אליהן משפה NP־שלמה מוכרת. לא נוכיח שהשפות הן ב־NP; הדבר לרוב פשוט למדי.

8.5.1 השפה 3SAT

השפה 3SAT זהה ל־SAT הרגילה פרט לכך שכל פסוקית מכילה בדיוק שלושה ליטרלים (השפה הדומה 2SAT שייכת ל־P). רדוקציה $\mathrm{SAT} \le_p 3\mathrm{SAT}$ מתבססת על פירוק כל פסוקית CNF של הפסוק המקורי ל"שרשרת" של פסוקיות מגודל 3 בעזרת משתני עזר חדשים:

$$(l_1 \vee \ldots \vee l_n) \mapsto (l_1 \vee l_2 \vee b_1) \wedge (\neg b_1 \vee l_3 \vee b_2) \wedge (\neg b_2 \vee l_4 \vee b_3) \wedge \ldots \wedge (\neg b_{n-3} \vee l_{n-1} \vee l_n)$$

כל b משתתף בשתי פסוקיות בדיוק, פעם כ־b ופעם כשלילתו, ולכן על ידי הצבה ל־b מסויים אפשר "לבטל" את אחת הפסוקיות. יש $n-3$ משתני b שכאלו אבל $n-2$ פסוקיות ולכן אחרי השמה ל־n־ים עדיין נשארת פסוקית אחת שחייבת להסתפק בעזרת אחד ה־l־ים, כך שהפסוק החדש מסתפק תחת השמה נתונה למשתנים של ה־l־ים אם ורק אם ההשמה הזו סיפקה את הפסוקית המקורית.

8.5.2 השפה Vertex Cover

כיסוי בצמתים של גרף לא מכוון G הוא תת־קבוצה $A \subseteq V$ כך שלכל $(u,v) \in E$, $u \in A$ או $v \in A$ (או שניהם).

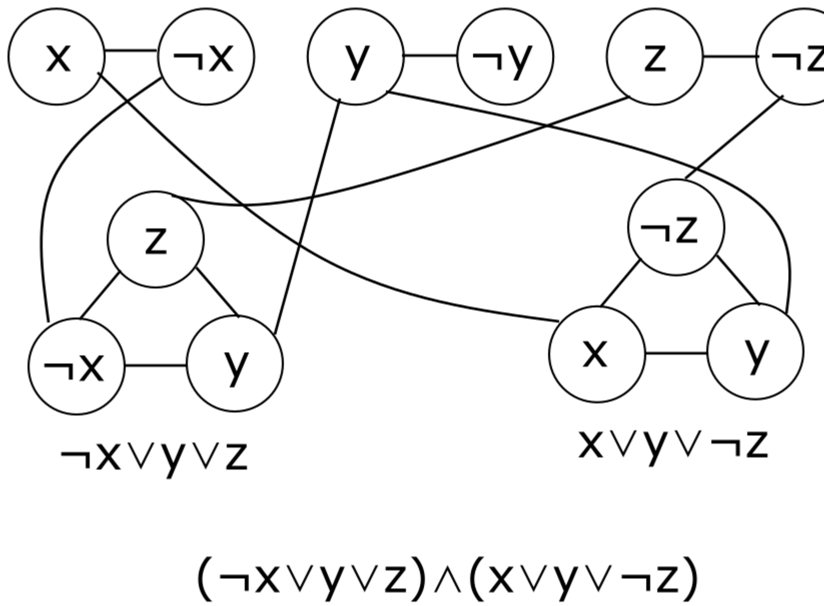
השפה Vertex Cover ובקיצור VC מורכבת מזוגות (G,k) כך ש־G גרף לא מכוון ו־k מספר טבעי, כך שקיים ל־G כיסוי בצמתים מגודל לכל היותר m.

נראה רדוקציה $3\mathrm{SAT} \le_p \mathrm{VC}$:

בהינתן פסוק φ עם n משתנים ו־m פסוקיות, נבנה גרף G בצורה הבאה:

ראשית, לכל משתנה x ב־φ נוסיף צמתים שמסומנים ב־x וב־$\neg x$ ונחבר אותם בקשת. על מנת לכסות קשתות אלו, יש צורך לבחור לפחות צומת אחד מכל זוג. את הפרמטר k של גודל הכיסוי בצמתים נכוונן כך שיהיה ניתן לבחור גם **לכל היותר** צומת אחד מכל זוג, כך שניתן יהיה לזהות את בחירת הצמתים עם בחירת **השמה** לפסוק המקורי.

בנוסף, לכל פסוקית $(l_1 \vee l_2 \vee l_3)$ נוסיף שלושה צמתים המסומנים ב־l_1, l_2, l_3 כך שכל זוג צמתים מתוכם מחובר בקשת. על מנת לכסות את ה"משולש" הזה הכרחי לבחור שניים מבין שלושת הצמתים שבו.



קיבלנו שכדי לכסות את הקשתות שתיארנו עד כה, כיסוי בצמתים חייב להכיל לפחות $n+2m$ איברים. נקבע כעת $k=n+2m$, מה ש"מכריח" כל כיסוי פוטנציאלי בצמתים לכלול בדיוק צומת אחד מכל "רכיב משתנה" ושני צמתים מכל "רכיב פסוקית".

לסיום, לכל צומת ששייך ל"רכיב פסוקית" ומתאים לליטרל l, נחבר צומת זה בקשת עם הצומת ב"רכיב משתנה" שמסומן ב־l. בצורה זו, אם כל שלושת הליטרלים של רכיב פסוקית אחד קיבלו F בהשמה, נהיה חייבים להוסיף את שלושת צמתי הליטרלים ונחרוג מהחסם של k.

8.5.3 השפות Independent Set ו־Clique

בהינתן גרף לא מכוון G, **קבוצה בלתי תלויה** ב־G היא קבוצה $A \subseteq V$ כך שלכל $u,v \in A$ מתקיים $(u,v) \notin E$.

המושג המשלים הוא **קליק** ב־G שהוא קבוצה $A \subseteq V$ כך שלכל $u,v \in A$ מתקיים $(u,v) \in E$.

בהתאם, השפה IS כוללת את כל הזוגות (G,k) כך שבגרף G יש קבוצה בלתי תלויה מגודל k לכל הפחות, והשפה CLIQUE את כל הזוגות (G,k) כך שבגרף G יש קליק מגודל k לכל הפחות.

קל לראות כי $\mathrm{IS} \le_p \mathrm{CLIQUE}$ על ידי הרדוקציה שמעבירה את הגרף $G=(V,E)$ לגרף (V,E') כך ש־$E' = \{(u,v) \mid (u,v) \notin E\}$ ומשאירה את k ללא שינוי. אותה רדוקציה בדיוק מראה גם כי $\mathrm{CLIQUE} \le_p \mathrm{IS}$. כלומר, די להראות שאחת משפות אלו היא NP־שלמה כדי להראות זאת עבור שתיהן.

נראה רדוקציה $\mathrm{IS} \le_p \mathrm{VC}$: $f(G,k) = (G, |V| - k)$.

על מנת לראות את נכונות הרדוקציה, נשים לב לכך שאם A היא כיסוי בצמתים של V, אז $V \backslash A$ היא קבוצה בלתי תלויה. זאת מכיוון שלכל קשת $(u,v) \in E$, או ש־$u \in A$ או ש־$v \in A$ (מהגדרת כיסוי בצמתים) כך שלא ייתכן ש־$u,v \in V \backslash A$ שניהם.

אם כן, בגרף יש כיסוי בצמתים מגודל לכל היותר k אם ורק אם יש בו קבוצה בלתי תלויה מגודל $|V|-k$ לפחות, כנדרש.

8.5.4 השפה 3COL

נגדיר k־צביעה של גרף לא מכוון בתור פונקציה $f:V\to\{1,2,\dots,k\}$ כך שלכל $(u,v)\in E$ מתקיים $f(u)\neq f(v)$.

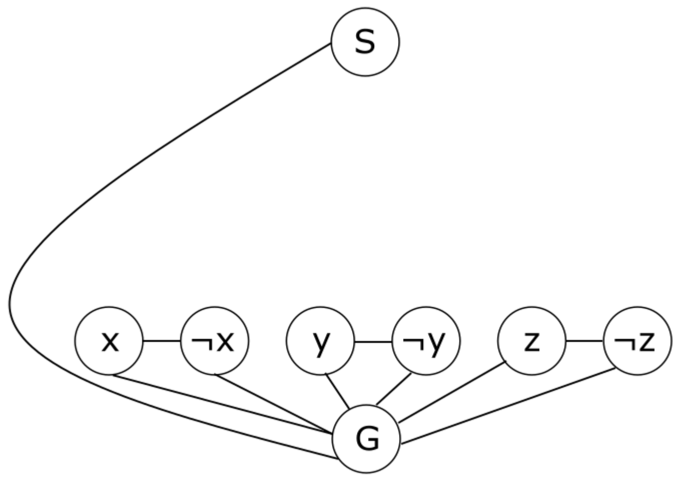
נאמר שגרף הוא k־צביע אם קיימת לו k־צביעה.

נגדיר את השפה 3COL בתור שפת הגרפים ה־3־צביעים. נוכיח שבעיה זו היא NP־קשה על ידי רדוקציה $3\mathrm{SAT}\le_p 3\mathrm{COL}$ (לעומת זאת, $2\mathrm{COL}\in\mathrm{P}$).

הרעיון הוא בניה של גרף שבו יהיו צמתים עבור כל משתנה וליטרל שלו, בדומה לכיסוי בצמתים, כך שמבין שלושת הצבעים שבהם הגרף נצבע, צבע אחד ייצג את T, שני את F ושלישי יהיה "נייטרלי" ולא ישתתף בצביעה של צמתי הליטרלים. בנוסף, לכל פסוקית נבנה תת־גרף שיהיה 3 צביע אם ורק אם אחד מצמתי הליטרלים שמחוברים לתת־הגרף צבוע בצבע של T.

המבנה הכללי של הגרף, ללא תיאור של רכיבי הפסוקיות לעת עתה, הוא זה:

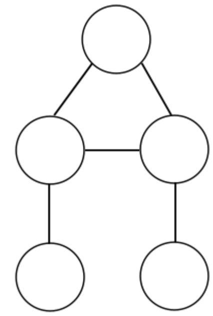
ראשית יהיו בגרף צמתים לכל ליטרל כך שזוג הצמתים שמתאימים לאותו משתנה מחוברים בקשת כך שמובטח שייצבעו בצבעים שונים (שהולכים לייצג את T ו־F) שנית, נוסיף לגרף צומת מיוחד, "G" ("קרקע") שיחובר לכל צמתי הליטרלים ובכך יהיה מובטח שהצבע שלו (שעליו נחשוב כ"נייטרלי") יהיה שונה מהצבעים של צבעי הליטרלים. כמו כן נוסיף עוד צומת מיוחד "S" ("שמיים") שיחובר אל הקרקע ובכך יהיה מובטח שצבעו ייצג את אחד מערכי האמת. נבחר לחשוב על הצבע של S בתור ערך האמת F ובהתאם לכך נבנה את רכיבי הפסוקיות.



כדי להבין איך נבנה רכיב פסוקית, ראשית נביט בדוגמא פשוטה יותר, עבור פסוקית בת שני ליטרלים. אנו רוצים לבנות רכיב בעל שני צמתי "כניסה" (אלו יהיו צמתי הליטרלים עצמם) וצומת "יציאה" כך שמתקיים:

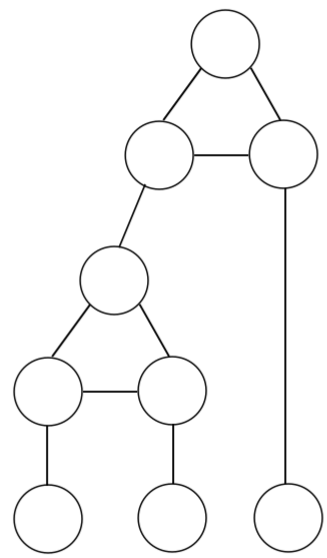
- אם לפחות אחד מצמתי הכניסה צבוע ב־T אז קיימת צביעה של הרכיב שבו צומת היציאה צבוע ב־T
- אם שני צמתי הכניסה צבועים ב־F אז בכל צביעה של הרכיב צומת היציאה צבוע ב־F

האפקט המבוקש מושג על ידי חיבור של **משולש** לשני צמתי הקלט:

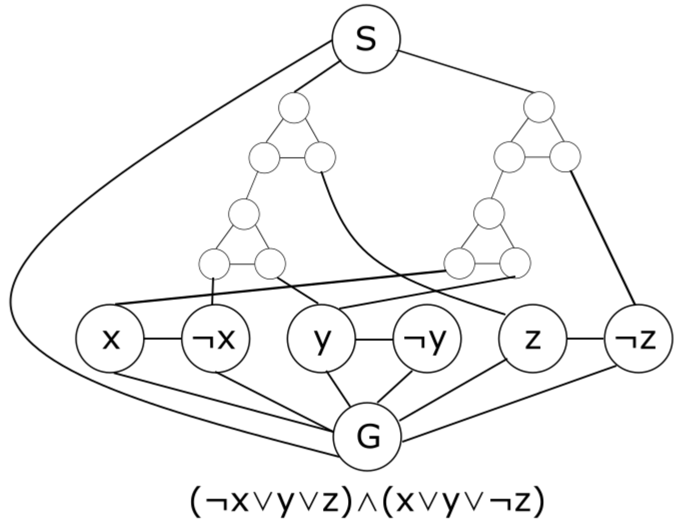


אם שני צמתי הקלט צבועים באותו צבע, אז שני הצמתים שמעליהם יהיו חייבים להיות צבועים בשני הצבעים האחרים (כאן הצבע השלישי, ה"נייטרלי" של הצביעה בא לידי ביטוי) ואז הצומת העליון יהיה חייב להיות צבוע באותו צבע כמו צמתי הקלט. לעומת זאת, אם שני צמתי הקלט צבועים בצבעים שונים (כלומר, אחד הוא T והשני הוא F) אז עדיין **אפשר** לצבוע את שני הצמתים שמעליהם ב־F והצבע הנייטרלי: נצבע את הצומת שמחובר ל־T ב־F ואת הצומת שמחובר ל־F בצבע הנייטרלי. זה יאפשר לנו לצבוע את הצומת העליון ב־T.

כעת ננקוט באותו תעלול עבור **שלושה** ליטרלי קלט פשוט על ידי שימוש כפול ברכיב שהצגנו: נבנה את הרכיב על שניים משלושת צמתי הקלט, ואז נשתמש בפלט שלו בתור צומת קלט נוסף לעותק של הרכיב, כשהליטרל השלישי הוא צומת הקלט השני:



כל שנותר הוא לרכב את הרכיבים הללו לגרף שכבר בנינו ־ הקלטים יהיו צמתי הליטרלים, והפלטים יחוברו כולם אל S כדי להבטיח שאם אחד מהפלטים יקבל את הערך F, הצביעה של הגרף תיהרס (כי F הוא הרי צבעו של S).



שימו לב כי ה־3 של 3SAT וה־3 של 3COL התבטאו בבניה בדרכים שונות לגמרי; עבור 3COL הדבר התבטא בכך שיכלנו להשתמש בצבע "נייטרלי" ובכך לבנות את הרכיבים שמדמים חישוב של ערך האמת של פסוקיות; ה־3 של 3SAT התבטא בכך שהרכיבים הללו הם פשוטים יחסית ומורכבים משרשור בודד של הרכיב לשני ליטרלים על עצמו (עבור פסוקיות של k ליטרלים היינו נזקקים ל־$k-2$ שרשורים שכאלו).

8.5.5 תכנון בשלמים

בעיית **תכנון לינארי** (Linear programming) היא בעיית אופטימיזציה תחת אילוצים. למשל, בעיה מהצורה "מצאו x,y,z שממקסמים את הפונקציה $2x+3y+5z$ תחת האילוצים $x+y\le 4$ ו־$1\le x+z$". ה**לינאריות** בשם הבעיה נובעת מכך שפונקציית המטרה והאילוצים הם כולם לינאריים במשתנים.

ניסוח כללי לבעיית תכנון לינארי הוא זה:

מצאו וקטור $x\in\mathbb{R}^n$

אשר ממקסם את $c\cdot x$ עבור $c\in\mathbb{R}^n$ (c מגדיר את פונקציית המטרה)

תחת האילוצים $Ax\le b$ עבור $A\in\mathbb{R}^{m\times n}$ ו־$b\in\mathbb{R}^m$ (A ו־b מגדירים m אילוצי אי שוויונים שונים)

בעיית התכנון הלינארי היא בעיה מרכזית במדעי המחשב וקיימים שלל אלגוריתמים לפתרונה, כולל אלגוריתמים פולינומיים. אלגוריתמים אלו מסתמכים על היכולת של המשתנים x להכיל ערכים **שבריים**; כאשר דורשים שהערכים של ה־x־ים יהיו מספרים **שלמים**, אפילו בעיית ההכרעה המתאימה (לבדוק אם קיים פתרון כלשהו שעונה על האילוצים, בלי למקסם פונקציית מטרה) הופכת ל־NP־שלמה, וזאת אפילו אם מגבילים את ערכי המשתנים להיות 0 ו־1 בלבד.

נסמן ב־ILP את השפה המתאימה: שפת כל הזוגות (A,b) כך שקיים פתרון $x\in\mathbb{Z}^n$ לאילוצים $Ax\le b$.

נראה רדוקציה פולינומית $3\mathrm{SAT}\le_p\mathrm{ILP}$:

יהא φ פסוק 3CNF כלשהו. יהיו $x_1,\dots,x_n$ המשתנים שמופיעים בו; אלו יהיו גם משתני בעיית התכנון בשלמים שלנו. הרעיון יהיה שהשמה של 0 בהם תתורגם להשמה של F והשמה של 1 תתורגם להשמה של T.

ראשית, לכל משתנה x_i שמופיע בפסוק ה־CNF נוסיף שני אי שווינים:

$x_i\le 1$

$0\le x_i$ (פורמלית, אי שוויון זה הוא $-x_i\le 0$)

נדגים תרגום של פסוקית לאי שוויון: למשל עבור הפסוקית $(x_1\vee\neg x_2\vee x_3)$ נבנה את אי השוויון

$x_1+(1-x_2)+x_3\ge 1$

ובאופן דומה כל פסוקית תתורגם לסכום של שלושה ביטויים כאשר כל ביטוי הוא או משתנה x (אם הליטרל המקורי היה x) או $1-x$ (אם הליטרל המקורי היה $\neg x$)

8.6 המשמעות של "קושי" בעיות NP־קשות

לסיום הנושא, יש צורך להבהיר מה בעצם משמעות התוצאות שראינו.

הוכחנו עבור מספר בעיות שהן NP־שלמות. המשמעות הפרקטית של הוכחה כזו היא: "אין טעם לחפש אלגוריתם שפותר את הבעיה ביעילות במקרה הגרוע ביותר", כדוגמת היתר האלגוריתמים שראינו במהלך הקורס. אמנם, אלגוריתם כזה **עשוי**

להתקיים, אולם מציאה שלו תפתור בעיה פתוחה מרכזית במדעי המחשב מזה כ־50 שנים שהאמונה הרווחת היא שהפתרון שלה הוא **הפוך**. דהיינו, אפילו אם אלגוריתם כזה קיים מציאה שלו תהיה קשה ביותר וכלל לא מובטח שהוא יהיה שימושי בפועל.

האם מכך ניתן להסיק שאם בעיה היא NP־שלמה אז אפשר לאבד כל תקווה לפתרון שלה? **חד משמעית לא**.

בכל הקורס עסקנו בניתוח סיבוכיות **במקרה הגרוע ביותר**, אולם בשימושים פרקטיים לא בהכרח צץ המקרה הגרוע ביותר, ודאי שלא ברוב הפעמים שבהן האלגוריתם מופעל. אם אלגוריתם רץ היטב על 99 אחוז מהקלטים שצצים בשימושים פרקטיים, הוא ימצא שימוש פרקטי גם אם אין לנו ביסוס תיאורטי לטיב שלו.

כזה הוא בדיוק המצב בתחום שעוסק באלגוריתמים שפותרים את בעיית SAT, שמכונים SAT-solvers. אלגוריתמים אלו לרוב מבצעים וריאציה על אלגוריתם DPLL שהוא אלגוריתם אקספוננציאלי לפתרון SAT בעזרת Backtracking. וריאציות חכמות על שיטה בסיסית זו כדוגמת CDCL (Conflict-driven clause learning) רצות היטב בפועל על מקרים פרקטיים רבים, אף שלא קשה להנדס קלט שעליהן הן ייכשלו בצורה מביכה (וגם על מקרים פרקטיים רבים הן נכשלות, כמובן).

מכיוון שבעיות רבות ניתנות לתרגום לבעיית SAT, ניתן להשתמש ב־SAT solvers כדי לפתור בעיות NP־שלמות עבור מקרים פרקטיים רבים. בנוסף, עבור גדלי קלט "סבירים" אפילו אלגוריתמים פשוטים למדי יעבדו ־ דוגמא קלאסית היא **סודוקו** שבגרסתו הכללית הוא בעיה NP־שלמה אך ניתן בנקל לתכנת אלגוריתם שיפתור מהר את גרסת ה־9×9 הנפוצה שלו.

שאלה מעניינת אחרת היא האם אלגוריתמים **הסתברותיים** יכולים לפתור ביעילות בעיות NP־שלמות גם במקרה הגרוע ביותר (דהיינו, זמן הריצה שלהם יהיה פולינומי במקרה הגרוע ביותר, והם יטעו רק, נאמר, ב־$\frac{1}{3}$ מן ההרצות שלהם). בלשון תורת הסיבוכיות זוהי השאלה האם $\mathrm{NP} \subseteq \mathrm{BPP}$, שהיא שאלה פתוחה בימינו (אם כי גם עבורה ההשערה היא שהתשובה היא שלילית).

נקודה אחרת שיש לתת עליה את הדעת היא שלא כל בעיה קשה ב־NP חייבת להיות NP־שלמה. הדוגמא הקלאסית היא בעיית **הפירוק לגורמים** שלא ידוע לה אלגוריתם יעיל (אפילו לא ברמה הפרקטית ולא במובן של המקרה הגרוע ביותר) אך אין הוכחה שהיא NP־שלמה וההשערה הרווחת היא שהיא אינה כזו. מאחר ואם $\mathrm{P} = \mathrm{NP}$ אז כל שפה (למעט השפה הריקה ושפת כל המילים) ב־NP היא NP־שלמה, לא נוכח להוכיח שפירוק לגורמים אינה בעיה NP־שלמה מבלי לפתור את שאלת $\mathrm{P} = \mathrm{NP}$ "על הדרך".

עם זאת, עדיין ניתן לשאול את השאלה ־ אם **נניח** ש־$\mathrm{P} \neq \mathrm{NP}$, האם ניתן להוכיח כעת כי פירוק לגורמים אינה NP־שלמה? זוהי עדיין שאלה פתוחה, אך ידועות שפות אחרות, שנבנות בצורה מלאכותית מתוך SAT, כך שאם $\mathrm{P} \neq \mathrm{NP}$ אז הן לא ב־P אך גם אינן NP־שלמות (זוהי תוצאה של **משפט לדנר**).