בעיות בלתי כריעות

דניאל רוזנבלט

מסמך זה הורד מהאתר http://www.underwar.co.il.

מחבר המסמך איננו אחראי לכל נזק, ישיר או עקיף, שיגרם עקב השימוש במידע המופיע במסמך, וכן לנכונות התוכן של הנושאים המופיעים במסמך. עם זאת, המחבר עשה את מירב המאמצים כדי לספק את המידע המדויק והמלא ביותר.

כל הזכויות שמורות ל-**דניאל רוזנבלט**.

בעיות בלתי כריעות

מאת דניאל רוזנבלט, מבוסס על הבנתי את הקורס "מודלים חישוביים" של פרופ' מיכאל טרסי ופרופ' נחום דרשוביץ מאונ' ת"א, אביב 2005. אם נפלו טעויות במאמר זה יש לייחס אותן לחוסר הבנתי את החומר (כפי שהעיר לי המרצה לא אחת), אבל להגנתי קיבלתי בסוף 100.

אני ממליץ על המאמר הזה למתכנתים, אנשים משכילים בכלל ואלה שעומדים ללמוד קורס בנושא בפרט. אני לא בטוח שזה יעזור לכם (אני הולך לבדוק על חברים שלי), אבל אני מקווה שכן. בכל מקרה, זה מעניין מאוד.

1 פתיחה

יש בעיות שמקובל לומר עליהן שהן "בלתי פתירות" כגון פיצוח הצפנת RSA, בעיית הסוכן הנוסע*, ובאופן כללי בעיות שסיבוכיות הזמן שלהן אקפוננציאלית**. הכוונה היא לא שאין פתרון לבעיה, אלא שאין פתרון "סביר", כלומר שהזמן שיקח למחשב בן-ימינו לפתור את הבעיה הוא כזה שאנחנו לא מוכנים לחכות לו. בכלל לא עולה על הדעת שקיימות בעיות שבאמת אין להן פתרון, כלומר אין (ולא תיתכן) תכנית מחשב שיכולה לפתור אותה בזמן סופי, שלא לדבר על סביר.

קשה למצוא דוגמאות מוכרות פרט לבעיית האנטי-וירוס. האנטי וירוסים שנמכרים בחנויות פועלים ע"י השוואת קבצים לקוד של וירוסים מוכרים. למה אין תוכנה שמזהה וירוס בצורה כללית? שפשוט בודקת אם ההרצה של קובץ תיצור עותקים נוספים של הקובץ? יש עם זה בעיה עקרונית, לא טכנית, ובשביל לענות על שאלות עקרוניות בנוגע לתוכנות יש תיאוריות ומודלים.

---

* בעיית הסוכן הנוסע: בהנתן רשימת בתים בהם צריך הסוכן לעבור והמרחקים ביניהם, מהו המסלול הקצר ביותר שיביא את הסוכן לכל הבתים?

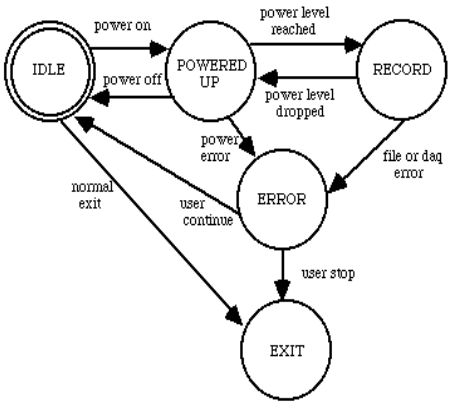
** אלגוריתם פולינומיאלי הוא אלגוריתם, שבהנתן קלט בגודל n, זמן החישוב (מס' הפעולות לפתרון) הוא פולינום ממעלה כלשהי ב-n. לדוגמה, מיון בועות לוקח בערך n^2 פעולות. סימולציות למציאת נקודות עגינה בין מולקולת לוקחות בערך n^7 פעולות, עדיין n בחזקה סופית כלשהי. התכונה של אלגוריתם כזה היא שאם **נכפיל** את n, משך הזמן **יוכפל** פי כמה.
אלגוריתם אקספוננציאל הוא אלגוריתם שדורש מספר פעולות הרבה יותר גדול, לדוגמה 2^n. התכונה של אלגוריתם כזה היא שדי **שנוסיף** קצת ל-n כדי שזמן החישוב **יוכפל**.

2 מודלים חישוביים

כדי לענות על השאלה "מה יכול מחשב לעשות?" צריך קודם לענות על השאלה "מהו מחשב?". יש לשאלה זו תשובות אפשריות רבות, ועל כל תשובה אפשר לבסס מודל שמגדיר איך המחשב עובד, ובהתאם לכך – מה הוא יכול לעשות.

מודלים שמאפשרים ניתוח לעומק צריכים להיות מוגדרים בצורה יסודית (שלא משאירה מקום לספק) ומופשטת (כלומר התיאור צריך להיות של "מה זה עושה" ולא של "ממה זה עשוי"), לכן ברמת המודל זה לא משנה ממה עשוי המחשב, כמה זיכרון יש לו ואיזה מעבד. כדוגמאות רשמתי את ההגדרות היסודיות של אוטומט סופי דטרמיניסטי ושל מכונת טיורינג. אתייחס להגדרות הללו בסיכום.

בספרות האקדמית קיים מגוון גדול של מודלים, כאשר רובם וריאציות של אחד המודלים הבסיסיים: אוטומט סופי (Finite Automaton), אוטומט מחסנית (Pushdown Automaton) ומכונת טיורינג (Turing Machine).



איור 1: אוטומט סופי
מתוך: **www.ltrpub.com**
(תודה ל-google שמצא אותה,
אין לי מושג על מה האתר)

אוטומט סופי דטרמיניסטי – הגדרה יסודית

*) אוטומט סופי דטרמיניסטי (Deterministic Finite Automaton) הוא קבוצה:
כאשר:
Q = קבוצה של מצבים.
= אלפבית (קבוצת של סימנים, לדוגמה {0,1}).
= פונקציית מעבר ממצב למצב:
(פונקציה שמקבלת מצב נוכחי ואות נוכחית,
ומחזירה מצב חדש ופלט)
q_0 = מצב התחלתי ().
F = קביצת מצבים סופיים, שהאוטומט עוצר ברגע שהוא מגיע אליהם. ()

הערה: אפשר להגדיר DFA ללא פלט, שמבצע קבלה/דחייה של מחרוזות מעל .

בתחתית של סולם המודלים נמצא האוטומט הסופי (נושא שתלמידי תיכון לומדים לבגרות במחשבים). אוטומט סופי מתואר ע"י קבוצה סופית של "מצבים", כאשר מכל מצב ניתן להתקדם למצב אחר בהתאם לקלט (ראה איור). המודל ה"חזק" ביותר הוא מכונת טיורינג, ומקובל כיום כי המודל הזה מתאר נכונה את יכולות המחשב (כל מחשב, ועל כך בהמשך).

לכל המעוניין, בסוף המאמר יש תרשים המשווה את היכולות של מגוון שפות.

2.1 מכונת טיורינג ושקוליה

נתאר בקצרה את המבנה ואופן הפעולה של מכונת טיורינג. המכונה מורכבת מ"ראש" שיודע לקרוא ולכתוב על גבי סרט נייר אינסופי. על הנייר יש "1"ים ו"0"ים (או סימנים נוספים), והראש יודע לקרוא את מה שעל הנייר, לגלגל את הנייר ימינה או שמאלה ולכתוב עליו "1" או "0" (או סימנים נוספים). זה מה שהמכונה **יכולה** לעשות, ומה שהיא עושה בפועל נקבע ע"פ ה"תכנות" שלה.בכל מכונה יש רשימת כללים בסגנון הזה:

- מצב 0:
  - אם כתוב על הסרט "סיום" כתוב "0", זוז ימינה ועבור ל"מצב 1".
  - אחרת אל תכתוב כלום, זוז ימינה ועבור ל"מצב 0".
- מצב 1:
  - כתוב "סיום" ועבור ל"מצב 2".
- מצב 2:
  - זהו מצב סיום, לא עושים בו כלום.

רשימת הכללים הנ"ל מגדירה מכונה שמבצעת הכפלה ב2- של מס' בינארי. השתמשתי בסרט שבו יש סימן מיוחד לסוף הסרט ("סיום").

המכונה נראית פרימיטיבת למדי, והיא אכן איטית בהשוואה למחשב (איטית מבחינה זו שדרושות הרבה פעולות כדי לממש חישוב פשוט לכאורה, אבל לא מבחינת מהירות הביצוע של פעולה בודדת), אבל היא לא נופלת ממחשב מבחינת היכולות שלה, כלומר: כל מה שניתן לחשב (או לבצע) באמצעות מחשב ניתן לחשב גם באמצעות מכונת טיורינג, ולהיפך.

זה נראה כמו דבר קשה להוכחה אם חושבים על העושר של פעולות שניתן לבצע בעזרת כל שפות התכנות, אבל נזכור שכל תכנית מחשב בסופו של דבר מתקמפלת ל-Assembler (או שפת דומה אחרת), וכל מה שצריך בשביל הוכחה מלאה זה להראות שניתן לחקות כל פקודה ב-Assembler באמצעות מכונת טיורינג (הספרים בתחום מלאים הוכחות לשקילות של שתי המכונות, ויש הוכחות שקילות עבור 4 מכונות שונות (תכנית scheme,

מכונת טיורינג – הגדרה יסודית

*) מכונת טיורינג (Turing Machine) היא קבוצה: $\{Q,\Sigma,\Gamma,\delta,q_0\}$
כאשר:
Q = קבוצה של מצבים.
Σ = אלפבית קלט (שמופיע על הסרט לפני הקריאה).
Γ = אלפבית סרט (שמותר למכונה לכתוב בו ולקרוא).
δ = פונקציית מעבר ממצב למצב: $\delta : (Q \times \Sigma) \to (Q \times \Gamma \times \{L,R\})$
(פונקציה שמקבלת מצב נוכחי ואות נוכחית ומחזירה מצב חדש, פלט לסרט וכיוון התקדמות)
q_0 = מצב התחלתי ($q_0 \in Q$).

הערה: אפשר להגדיר מכונת טיורינג ללא פלט, שמבצע קבלה/דחייה של מחרוזות מעל Σ.

"מכונת מונים", מכונת טיורינג ו-Ram machine (מעבד)) במצגות השעור באתר של פרופ' נחום דרשוויץ: www.cs.tau.ac.il/~nachumd/models).

2.2 הערה על סיבוכיות

ההגדרה האינטואיטיבית ל"סיבוכיות של אלגוריתם" היא "מספר הפעולות הדרוש לאלגוריתם ביחס לגודל הקלט". זו לא ההגדרה היסודית, אבל היא תספיק לעת עתה.

יש הנחה סמויה בהגדרה הזו, והיא שה"אלגוריתם" מסיים את פעולתו בזמן סופי, אחרת אין משמעות ל"יחס" שבהגדרה. אנחנו לא נותנים את דעתנו על כך מתוך הנחה שלכל בעיה יש אלגוריתם שפותר אותה (בזמן סופי), ואם תוכנה נתקעת אז זו אשמת המתכנת, או החומרה או מערכת ההפעלה (לא יעלה על הדעת ש**הבעיה** היא כזו שכל אלגוריתם שפותר אותה ייתקע). לפיכך, כל בעיה שאנו מדברים על הסיבוכיות שלה, סימן שיש לנו עבורה אלגוריתם שעובד ושלעולם לא נתקע (עם חומרה לבחירתנו). במאמר זה אני רוצה לעסוק בבעיות שמושג הסיבוכיות הוא חסר משמעות עבורן כיוון שאין אלגוריתם שפותר אותן בזמן סופי.

2.3 כריעות (על משקל "צניעות")

הגדרה: "בעיה היא **כריעה** אם קיימת תכנית שפותרת אותה, עבור כל קלט, בזמן סופי". היא נקראת גם decidable, recursive או "שייכת ל-R". בהתאם לכך, בעיה היא **לא-כריעה** אם לא קיימת תכנית שפותרת אותה עבור כל קלט בזמן סופי ("לא קיימת" במובן שאין ולא ניתן לכתוב, לעולם, תכנית מחשב שפותרת את הבעיה). אין לי ספק שההגדרות האלה די סתומות בקריאה ראשונה, לכן אתן כמה דוגמאות אחרי שתי השאלות להלן.

איך מוכיחים שבעיה היא כריעה?

מוצאים אלגוריתם שפותר אותה לכל קלט בזמן סופי (דוגמאות לבעיות כריעות: חיפוש מיון, הדפסת תלושי משכורת וכל מה שתוכנות עושות בימינו).

*איך אפשר **להוכיח** שלא קיימת אף תכנית שפותרת בעיה (בזמן סופי ולכל קלט)?*

יש 2 דרכים. הדרך הקשה היא להניח שקיימת תכנית שפותרת לכל קלט בזמן סופי, ובאמצעות התכנית הזו לכתוב תכנית אבסורדית (תיכף נראה מה זה). בדרך הקלה נתחיל עם בעיה A, שכבר הוכח כי איננה כריעה (באחת מן הדרכים), ונראה שלו היתה בעיה B כריעה אזי בהכרח גם A היתה כריעה. לדרך השנייה קוראים "רדוקציה" והיא מוגדרת לפרטיה במסגרת.

עתה נתבונן במספר דוגמאות לבעיות לא כריעות ונראה כיצד מוכיחים את אי-כריעותן (הדוגמות וניסוחן לקוחות מתוך הרצאותיו של פרופ' נחום דרשוביץ).

רדוקציה

*)ניתן לומר כי:
"בעיה A ניתן להפוך לבעיה B ע"י רדוקציה", "יש רדוקציה מ-A ל-B","A can be reduced to B","B".
ולרשום זאת כך: $A \propto B$.

*) אם: - אם B כריעה אז A כריעה.
- אם A בלתי כריעה אז B בלתי כריעה.

*) במילים אחרות, הכלי הזה יכול לשמש להוכחת ה הכריעות של A או להוכחת אי הכריעות של B, אבל לא לשום דבר אחר.

2.3.1 בעיית ה- Busy Beaver

הבעיה: בהנתן n (מס' טבעי), מצא את המספר הגדול ביותר שיכולה להדפיס תכנית C שכוללת לכל היותר n תווים (או תכנית scheme, או java, או מכונת טיורינג בעלת n מצבים וכו').

פתרון נאיבי: מה הבעיה? נכתוב תכנית C שמייצרת מחרוזות באורך של עד n, מקמפלת אותן, מריצה אותן וזוכרת את התוצאה הגדולה ביותר שנתקבלה.

למה זה עלול לא לעבוד? כי אנחנו עלולים לייצר תכנית שכוללת את השורה:

```
for (i=1; i>0; )
```

ואז כשנריץ אותה התכנית תיתקע ולא תחזיר ערך לעולם. אבל זהו רק קושי טכני קל, ואפשר להתגבר עליו בקלות. אפשר להתגבר בקלות על המון קשיים טכניים שיעלו במהלך הכתיבה, השאלה היא האם ניתן להתגבר על כולם? התשובה היא שלא, ונוכיח את זה בדרך הקשה.

- נניח שניתן לפתור את הבעיה. נניח שקנינו תוכנה בשם MSBB(n) שעל אריזתה כתוב שהיא מקבלת מספר טבעי n ומחזירה את המספר הגדול ביותר שמחזירה תכנית C באורך של עד n תווים.
- נסמן MSBB(n) את המספר שהתכנית מחזירה בהפעלתה על המספר n.
- נסמן bb(n) את מה שפונקציית Busy Beaver אמורה להחזיר עבור הערך n, היינו המספר הגדול ביותר שמחזירה תכנית C באורך של עד n תווים.
- נסמן ב-N את מספר התווים שבתכנית MSBB.
- נמציא תכנית בשם c(), ללא קלט, כדלקמן:

```
int c (void) {return (MSBB(10*N) + 1);}
```

נניח שזו התכנית כולה, למעט כמה הכרזות כמובן.

- הבה נחשב את מס' התווים שבתכנית c():

$$|c| = |MSBB| + |N| + 100$$

הסבר: התכנית כוללת את כל התווים שב-MSBB, את המספר N (שדורש מס' תווים כמספר הספרות שלו) ועוד כמה תווים, נניח 100, עבור הכרזות ושורת ה-return. לפיכך:

$$|c| = N + \log_{10}(N) + 100 \ <= \ 10N$$

- כעת נסביר למה יש סתירה בתכנית שכתבנו:
  - מצד אחד, כתבנו תכנית שאורכה בתווים קטן מ- 10Nלכן:

$$bb(10N) >= c()$$

  - מצד שני, הפלט של c() גדול מ-MSBB(10N):

$$c() > MSBB(10N)$$

  - נאחד את שני האי-שוויונים:

$$bb(10N) > MSBB(10N)$$

  - וזו סתירה להנחה שלנו, ש-MSBB אכן פותרת את בעיית ה-Busy Beaver.
  - הסתירה יכולה לבוא מכשל בכל אחד משלבי ההוכחה, אבל כל מה שעשינו היה להניח משהו על MSBB ולכתוב תכנית פשוטה ביותר.

חשוב להבין שההוכחה הנ"ל אומרת ש"לא ניתן לממש את פונקציית bb(n)" אבל היא לא אומרת לנו את הסיבה לכך שלא נוכל לממש אותה. כעת אפשר לשלב את הקושי הטכני שבו נתקלנו בהתחלה עם הקושי העקרוני שהוכחנו את קיומו כדי להבין מה עלולה

להיות הבעיה המעשית שתמנע את מימוש הפונקציה MSBB: היא תיתקע. אולי נוכל לכתוב תכנית שמנבאת אם תכנית אחרת תיתקע? לא, וזו הדוגמה הבאה.

2.3.2 בעיית העצירה (The Halting problem)

זו הבעיה הלא-כריעה הבסיסית ביותר ומרבית ההוכחות "בדרך הקלה" מבוססות עליה (ישירות או בעקיפין).

הבעיה: בהנתן תכנית C (שמקבלת מספר טבעי ומחזירה מספר טבעי) שנכנה f, וקלט עבור f שנכנה x, האם ההרצה של f על x תסתיים תוך זמן סופי?

נעבור ישר להוכחה שבעיית העצירה היא בלתי כריעה:

- שוב, נניח שקנינו תכנה MSH(f,x) שמחזירה ערך true אם ההרצה f(x) אי פעם עוצרת ומחזירה false אם ההרצה לעולם לא עוצרת.
- נסמן halt(f,x) את הפתרון האמיתי לבעיית העצירה עם f ו-x.
- נכתוב תכנית בשם loopy() שנכנסת ללולאה אינסופית ולעולם לא עוצרת.
- ונכתוב תכנית בשם c(f) כדלקמן:

```
bool c (f)
{
    if( MSH(f,f) )
        loopy();
    return (true);
}
```

אם זה נראה בעייתי להעביר את f כארגומנט מספרי ל-MSH, נזכור שאפשר לקודד כל מחרוזת (תכנית) למספר טבעי (לדוגמה, ע"י שרשור ערכי ה-ASCII של התווים לפי הסדר) והקידוד הפיך לגמרי, לכן אפשר לומר ששני הארגומנטים לבעיה הם מספריים.

- השאלה עכשיו היא: מה תחזיר הרצת הפונקציה c על הפונקציה c? יש כמובן 3 אפשרויות:
  - אם ההפעלה תחזיר true, הרי ש- MSH(c,c) == false, כלומר הרצת c(c) אמורה להימשך לנצח לפי MSH, אבל אנחנו מקבלים תשובה בזמן סופי! זו סתירה להנחה ש-MSH אכן

פותר את בעיית העצירה (כי התשובה שהוא נותן לא נכונה: הוא אומר ש(c)c ימשך לנצח, אבל הוא טועה).

- אם ההפעלה תחזיר false, זה קצת בעייה כי לא קודדנו שום אפשרות כזו...
- אם ההפעלה לא תסתיים לעולם, אחד מהשניים קרה:
  - MSH נתקע, וזו סתירה להנחה ש-MSH תמיד עונה בזמן סופי.
  - נכנסנו ל-()loopy כיוון ש- MSH(c,c) == true.

כלומר לפי MSH הפעלת c(c) אמורה להסתיים תוך זמן סופי, אבל עבורנו היא נתקעה! סתירה!

מה המסקנה שנותרת? הנחת המוצא שגויה! ההנחה שבכלל קיימת תכנית שמנבאת (תוך זמן סופי) אם הפעלת פונקציה f על ערך x תסתיים אי-פעם, עבור כל f ו-x. למה? כי מצאנו זוג ערכים (c ו- c) שעבורם אין סיכוי שהתכנית תעבוד. ההוכחה הזו מהווה בסיס לאינספור הוכחות אי-כריעות, ואני רוצה לפחות פעם אחת להדגים את הדרך ה"קלה" להוכחת אי כריעות.

2.3.3 בעיית אי-העצירה-לכל-קלט ($\overline{Halt^*(f)}$)

הבעיה: בהנתן תכנית f, האם התכנית f נתקעת לכל קלט אפשרי?

- ננית שיש פתרון לבעיה, תכנית בשם MSH*(f)!.
- כעת, ע"י שימוש ב- MSH*(f)! נבנה תכנה שפותרת, עבור כל f ו-x את בעיית העצירה הרגילה!
- עבור כל f וכל x ניתן לכתוב פונקציה f' כדלקמן:

```
int f' (y) { return (f(x));}
```

נשים לב שהתכנית f' מתעלמת לחלוטין מהקלט שלה, ומריצה את f על x.

- זאת אומרת, שלו היינו רוצים לפתור את בעיית העצירה הרגילה עבור f ו-x, היינו צריכים פשוט לכתוב תכנית f' שכזו, והיינו מקבלים את התשובה ע"י הפעלת MSH*(f)!': אם התשובה היתה true, הרי זה בגלל ש-f(x) נתקעת, ואם התשובה היתה false, הרי זה כי f(x) לא נתקעת.

- זאת אומרת שבעיית העצירה הרגילה היא כן כריעה (מצאנו דרך לפתור אותה לכל f ו-x בזמן סופי). זו סתירה, כיוון שכבר הוכחנו שהיא לא כריעה.
- איפה הפגם בהוכחה אם כן? בהנחה שניתן לכתוב תכנית *MSH!. כך הוכחנו שאי אפשר לכתוב תכנית *MSH! או תכנית שקולה.

זאת הוכחה כדין, אבל היא לא כל כך מעניינת. הבאתי אותה כדי שנוכל להגיע להוכחה כן מעניינת:

2.3.4 בעיית השקילות של תכניות.

הבעיה: לכל שתי תכניות f ו-g, האם הן שקולות? כלומר האם לכל קלט x מתקיים ש-f(x)==g(x), ואם אחת נתקעת אז גם השנייה נתקעת?

- נניח שקיימת תכנה שפותרת את הבעיה, נקרא לה לשם שינוי EQ(f,g), והיא מחזירה true אם התכניות שקולות ו-false אם לאו.
- כעת, באמצעות התכנה EQ נראה שאפשר לפתור את בעיית אי-העצירה-לכל-קלט. ניקח פונקציה f שאנחנו מעוניינים לפתור עבורה את בעיית אי-העצירה-לכל-קלט, ונפעיל עליה את הפונקציה הבאה:

```
return ( EQ(f, loopy()) );
```

- הפונקציה הזו תגיד לנו בדיוק האם f נתקעת לכל קלט! אבל כבר הוכחנו שלא ניתן לענות על שאלה זו לכל קלט בזמן סופי, לכן המסקנה היא ש-EQ היא בלתי ניתנת למימוש. לא ניתן להשוות שתי פונקציות מבחינת התפקוד שלהן. אפשר כמובן להשוות את המבנה (הסינטקס).

זה מתקשר לבעיית האנטי-וירוס: בבעיה זו אנו רוצים לדעת אם תכנית נתונה (הקובץ החשוד כוירוס) מבצע דבר מה (מעתיק את עצמו, במקרה של וירוס). זו לא בדיוק בעיית EQ אבל זו בעיה דומה, ובכל מקרה תיכף נכיר את משפט RICE שבאמת יפתיע אותנו.

2.3.5 משפט RICE

נתחיל עם הניסוח העממי והמהמם של המשפט: "כל בעיה מעניינת אודות תכנית מחשב היא בלתי כריעה".

מהי בעיה "מעניינת"? בעיה צריכה לקיים שני תנאים כדי להיות מעניינת:

- להיות לא-טריוויאלית (non-trivial)
  - בעיה לא טריוויאלית: בעיה שיש תכניות שעבורן התשובה היא true ויש כאלה שעבורן התשובה היא false.
  - דוגמה: "האם התכנית מבצעת מיון?"
  - דוגמה נגדית: "האם התכנית מבצעת מיון בזמן לינארי?"

- להיות סמנטית (semantic)
  - בעיה סמנטית היא בעיה שעבור כל שתי תכניות שקולות התשובה זהה, כלומר זו שאלה על מה שהתכנית עושה ולא על הסינטקס.
  - דוגמה: "האם הפלט תמיד גדול מהקלט?".
  - דוגמה נגדית: "האם יש בתכנית לולאת while?".

למי שרוצה, הוכחה קלה למשפט במצגות השעור של נחום דרשוביץ (www.cs.tau.ac.il/~nachumd/models).

3 סיכום

מתוך הנחה שהקורא לא התעמק בהגדרות היסודיות של האוטומט הסופי ושל מכונת טיורינג, אני מזמין את הקורא להשוות ביניהן. יקח לכם זמן, כי ההבדל בהגדרה הוא מזערי. עם זאת, ההבדל בכושר החישוב הוא עצום: מכונת טיורינג היא מחשב לכל דבר ואילו אוטומט סופי לא מסוגל אפילו להשוות את מספר ה"0"ים למספר ה"1"ים במחרוזת בינארית.

ההבחנה בין בעיות שמודל מסוים "יכול לפתור" לבעיות שהוא "לא יכול לפתור" נוטעת בי תחושה, שלמרות שה"בעיות" שאנו מדברים עליהן הן כולן מעשה ידי אדם, יש להן קיום משל עצמן והן מתחלקות לקבוצות: בעיות שניתן לפתור ע"י אוטומט סופי, בעיות שניתן לפתור ע"י אוטומט מחסנית, ע"י מכונת טיורינג, וכו' (יש תרשים הרבה יותר מלא בעמ' הבא), כאילו הן היו שם עוד לפני שהמצאנו מחשבים ותכניות ולפני שניסחנו אותן במילים.

עוד רמז לקיום העצמאי של "הבעיות" אפשר לראות בחלוקה של הבעיות הכריעות ל-P, NP ו-NPC (ועוד). בעיות ב-P הן בעיות שנמצא להן אלגוריתם לפתרון בזמן פולינומיאלי. בעיות ב-NPC הן קבוצת בעיות שלא נמצא להן אלגוריתם לפתרון בזמן פולינומיאלי, אבל מספיק שנפתור אחת בזמן פולינומיאלי וכל האחרות תיפולנה כפרי בשל. בעיות ב-NP הן בעיות שלא נמצא להן אלגוריתם לפתרון בזמן פולינומאלי, ואם נמצא זה יהיה סבבה, אבל לא יותר מזה (על הבעיות ב-NP וב-NPC חל גם תנאי, שבהנתן הצעה לפתרון ניתן לבדוק אותה בזמן פולינומיאלי). בעיות מעניינות וחשובות נמצאות בקבוצה NPC, ומדענים עדיין לא יודעים אם יש לחלוקה הזו משמעות אמיתית או שפשוט עוד לא מצאנו אלגוריתמים מחוכמים מספיק.

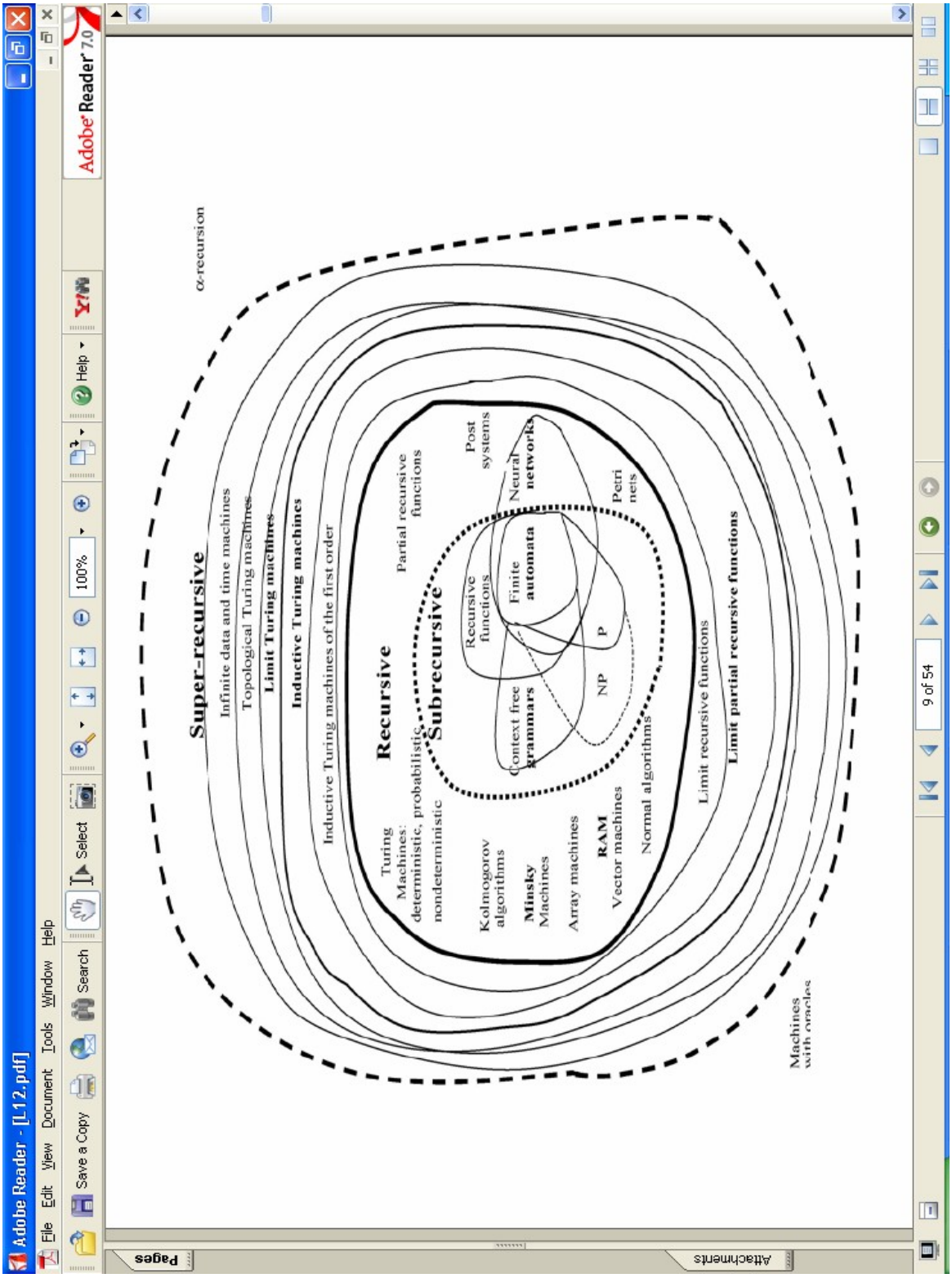
בנימה אקטואלית, חוקרים רבים מנסים "לפרוץ את גבולות טיורינג" ולמצוא מודל שיפתור בעיות שאינן כריעות במסגרת המודל של טיורינג, אבל עוד לא נמצא אחד שכזה (אם כי היו טענות לכך).

ארשה לעצמי להעיר עוד הערה מתחום מרוחק אבל קרוב לליבי: בתחום שנקרא "Quantum Computation" הולך ונבנה מודל שיגדיר היטב את האפשרויות הגלומות במחשב קוונטי. בשלב הזה אין ודאות אם המודל הקוונטי חזק (או שמא חלש) יותר מהמודל של טיורינג. גם בתחום המהירות לא בטוח אם אחד מהשניים "עדיף" על האחר; במודל הקוונטי נמצאה דרך לפצח הצפנת RSA בזמן פולינומיאלי, אולם זו לא בעיה ב-NPC כך שהפיצוח, למרות היותו הישג מכובד (שממתין למעצבי המחשב כדי לממש אותו), נותר לבדו. מאמצים רבים מושקעים כדי לבדוק אם המודל יפצח בעיות ב-NPC, כי אם כן זה יהיה תמריץ אדיר לפיתוח מחשבים קוונטיים שיפתחו המוני אפשרויות חדשות בפני עולם המחשבים.

מקורות:

- המצגות באתר הקורס "מודלים חישוביים" של פרופ' מיכאל טרסי ופרופ' נחום דרשוביץ (*www.cs.tau.ac.il/~nachumd/models*).

- "*The Emperor's new Mind*" מאת Roger Penrose, פרק ראשון (כולל הסבר ידידותי אך מהותי לגבי מכונות טיורינג).



איור 2: קבוצות שבעיות שנפתרות ע"י אלגוריתמים מכל מודל