חישוב מבוזר

אלי דיין[1]

13 ביולי 2013

[1] <eliyahud@post.tau.ac.il>

תקציר

מסמך זה יביא את סיכומי השיעורים מהקורס חישוב מבוזר, שהועבר על ידי פרופ׳ יהודה אפק בסמסטר ב׳ בשנה״ל תשע״ג.

תוכן עניינים

1 שיעור 1 **3**

1.1 העברת הודעת – Message Passing 4

1.2 הפצה של הודעה (Broadcast) 4

1.2.1 הנחות לגבי המודל 4

1.2.2 המימוש 5

1.2.3 עץ פורש 5

1.2.4 גילוי סיום 6

1.2.5 ניתוח סיבוכיות 6

1.3 Diffusing Computation 7

1.3.1 תיאור הבעיה 7

1.3.2 הצעה לאלגוריתם 7

1.3.3 הצעה טובה יותר 7

1.3.4 שיפור להצעה 7

1.3.5 כמה יוזמים 8

1.4 Snapshots 8

2 שיעור 2 **10**

2.1 Snapshot 10

2.2 רשת סינכרונית 11

2.3 סינכרונייזר – Synchronizer 12

2.3.1 סינכרונייזר α 12

2.3.2 סינכרונייזר β 12

2.3.3 סינכרונייזר γ 12

2.3.4 סיכום 13

2.4 בניית Cluster-ים 14

3 שיעור 3 **15**

3.1 בחירת מנהיג 15

3.1.1 תיאור הבעיה 15

3.1.2 אלגוריתם ראשוני 15

3.1.3 אלגוריתם יעיל יותר (בפאזות) 15

3.1.4 אלגוריתם לרשת סינכרונית 16

3.1.5 גודל הרשת לא ידוע 16

3.1.6 חסם תחתון למספר ההודעות 16

3.1.7 הקשר לעץ פורש 19
3.2 עץ פורש מינימלי – Minimum Spanning Tree (MST) 19

4 שיעור 4 **20**
4.1 עץ פורש מינימלי 20
4.2 Maximal Independent Set 22

5 שיעור 5 **23**
5.1 הרדיוס של בעיה 23
5.2 בעיית הצביעה במעגל 23
5.3 Sequence Transmission Problem (STP) 25

6 שיעור 6 **27**
6.1 בעיית ההסכמה – Consensus 27
6.1.1 התמודדות עם Fail-Stop Fault 28
6.1.2 התמודדות עם Byzantine Fault 29
6.2 מודל זיכרון משותף (Shared Memory) 33

7 שיעור 7 **34**
7.1 בטחון באוגרים בזיכרון משותף 34
7.2 Snapshot 35

8 שיעור 9 **37**
8.1 בעיית ההסכמה 37
8.2 פעולות אטומיות על אוגרים 41

9 שיעור 10 **43**
9.1 מספר Consensus 43
9.2 בנייה אוניברסלית 44

10 שיעור 11 **46**
10.1 היסטוריות 46

11 שיעור 12 **47**
11.1 Splitter 47
11.2 בעיית ה־Renaming 49
11.3 מניעה הדדית (Mutual Exclusion) 49

12 שיעור 13 **51**
12.1 Set Consensus 51
12.2 שקילות המודלים 51

13 שיעור 14 **54**
13.1 יריבים 54

14 שיעור 15 **55**
14.1 אלגוריתמים רנדומיים להסכמה 55

פרק 1

שיעור 1[1]

חישוב מבוזר מורכב מהרבה מעבדים שמחשביסאת אותה המשימה, כאשר כל מעבד יודע חלק מהקלט, והמטרה היא שכל מעבד ידע חלק מהפלט.

דוגמה 1: כל המחשבים באינטרנט מחשבים את המסלולים הקצרים בין כל מחשב לכל מחשב.
אין מחשב שיודע את כל המסלולים הקצרים בין כל זוג מחשבים, אבל כל מחשב יודע את המסלולים הקצרים ממנו לכל מחשב אחר.

דוגמה 2 (חישוב עץ פורש): חישוב עץ פורש. על מעבד הוא צומת בגרף, והוא רואה רק את המעבדים הקרובים אליו.
בסופו של דבר, כל צומת (מעבד) ידע אילו מהקשתות הסמוכות אליו הן חלק מהעץ הפורש.

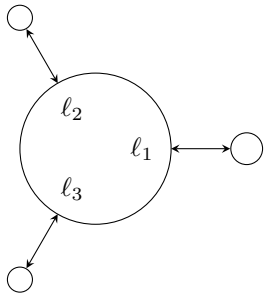
נדבר על חישוב מבוזר, שהוא חישוב א־סינכרוני. זה לא חישוב מקבילי, שמדבר על הרבה מעבדים באותו המחשב (למשל). נטפל גם בחישוב סינכרוני, אבל גם שם יהיה קיים אלמנט אי־הוודאות.
נדבר על מודלים של העברת הודעות ומחשבים מרובי ליבות עסתקשורת לא מסודרת. בכולם יש אי־ודאות. למשל, ברשת, חישוב מתקדם, אבל יש הרבה דברים שאינם ידועים (למשל חיבור בין שני מחשבים נעלם).
המודלים שנדבר עליהם:

1. העברת הודעות (Message Passing).

2. זיכרון משותף: בהתחלה בצדדים התיאורטיים, ואז נעבור למבני נתונים מקביליים.

טרמינולוגיה: מקבילי – Parallel. מבוזר – Concurrent.
המוטיבציה העיקרית לחישוב מבוזר היא עמידות לתקלות – Fault Tolerance. זאת הייתה המוטיבציה להקמת ה־ARPA-Net, האבא של האינטרנט, בתקופת המלחמה הקרה. באמצעותה קיוו להשיג שרידות במקרה של התקפה סובייטית. זה גם נועד למנוע תקלות שנובעות ממחשבים זדוניים שמזייפים חישוב.

[1] סיכום לשיעור שהועבר בתאריך 03.03.2013.



איור 1.1: מבנה של צומת ברשת במודל העברת הודעות

1.1 העברת הודעת – Message Passing

איך שולחים הודעות?

SEND(M on link ℓ_i)

איך מקבלים הודעות? המודל שלנו יהיה מונחה־אירועים (Event-Driven), וקבל הודעה היא אירוע:

1: **Upon Receiving** M **on link** ℓ_i
2: ...
3: SEND(M on link ℓ_j)
4: ...

איך החישוב יתנהל במימוש אמיתי? לכל מחשב יהיה תור של הודעות שהתקבלו. התור יהיה FIFO, ולכן ההודעות יטופלו לפי סדר קבלתן. לא מטפלים בהודעה כל עוד הטיפול בהודעה הקודמת לא הושלם.

1.2 הפצה של הודעה (Broadcast)

1.2.1 הנחות לגבי המודל

- לרשת יש טופולוגיה שהיא גרף קשיר כלשהו, והקשתות לא מכוונות.
- כל קודקוד רואה רק את השכנים שלו, ולא מכיר את שאר הקודקודים.
- לכל מעבד יש זיכרון ושם יחיד.
- הקשתות הן FIFO.
- הודעות לא נאבדות.
- מספר הקודקודים n אינו ידוע.

אלגוריתם 1.1 הפצה של הודעה

```
1: function START(M)                                  ▷ Start broadcast of M
2:     SEND(M to all neighbors)
3: end function

4: Upon Receiving M on link ℓ                         ▷ On any node except initiator
5:     if RECEIVED(M) = false then
6:         RECEIVED(M) ← true
7:         SEND(M to all N(v) \ {ℓ})
8:     end if
```

אלגוריתם 2.1 הפצה של הודעה ופרישת עץ

```
1: function START(M)                                  ▷ Start broadcast of M
2:     SEND(M to all neighbors)
3: end function

4: Upon Receiving M on link ℓ                         ▷ On any node except initiator
5:     if parent = nil then
6:         parent ← {ℓ}
7:         SEND(M to all N(v) \ ℓ)
8:     end if
```

1.2.2 המימוש

קוד ראשוני למימוש נמצא באלגוריתם 1.1.

הגדרה 3 (קבוצת שכנים): קבוצת השכנים של הצומת/מחשב v מסומנת ב־$N(v)$.

במימוש שלנו, $\text{RECEIVED}(M)$ מאותחל להיות `false`.

1.2.3 עץ פורש

בעזרת קבלת ההודעה באלגוריתם, אפשר לפרוש עץ, כפי שניתן לראות באלגוריתם 2.1.

למה 4: האלגוריתם מוצא עץ.

הוכחה: נניח בשלילה שיש מעגל. נסתכל על הפעם האחרונה שמישהו ביצע את שורה 6 במעגל. נקרא לקודקוד v_j.

נאמר כי v_j הוא האחרון לבצע את שורה 6. אבל היא כבר שלח את ההודעה ל־v_{j+1}, בסתירה לקוד.

לכן, אוסף הקשתות המסומנות `parent` הוא חסר מעגלים. כדי להראות שזה עץ, צריך להוכיח שיש בדיוק $n-1$ קשתות.

ל־Initiator אין `parent`, ולכל קודקוד אחר יש. נוכיח את זה: נניח בשלילה שיש קודקוד שלא קיבל את ההודעה. מכיוון שהגרף קשיר, קיים מסלול בינו לבין ה־Initiator.

אלגוריתם 3.1 הפצה של הודעה עם גילוי סיום

```
1:  function START(M)                                   ▷ Start broadcast of M
2:      SEND(M to all neighbors)
3:  end function

4:  Upon Receiving M on link ℓ at any node except root
5:      if parent = nil then
6:          parent ← ℓ
7:          SEND(M to all N(v) \ {ℓ})
8:          if N(v) \ {ℓ} = ∅ then
9:              SEND(Ack to ℓ)
10:         end if
11:     else
12:         SEND(Ack to ℓ)
13:     end if
14: Upon Receiving M on link ℓ at root
15:     SEND(Ack to ℓ)

16: Upon Receiving Ack on link ℓ at any node except root
17:     Mark ℓ as Ack'd
18:     if All N(v) \ {parent} Ack'd then
19:         SEND(Ack to parent)
20:     end if
21: Upon Receiving Ack on link ℓ at root
22:     Mark ℓ as Ack'd                                 ▷ This is the root node
23:     if All nodes acked then
24:         SIGNAL(Terminated)
25:     end if
```

נסתכל על הקודקוד הראשון במסלול שלא קיבל את ההודעה. שכן שלו (על המסלול) כן קיבל אותה, ולכן היה אמור לשלוח אותה, בסתירה. ■

עכשיו אנחנו יודעים שהאלגוריתם פורש עץ ברשת. אלגוריתם זה נקרא Convest Broadcast.

1.2.4 גילוי סיום

נרצה להפוך את האלגוריתם שלנו ל־Broadcast Echo, כדי לקבל גילוי סיום – Termination Detection. אלגוריתם 3.1 מתאר אלגוריתם הפצת הודעה עם גילוי סיום.
נכונות האלגוריתם נובעת מהקוד.

1.2.5 ניתוח סיבוכיות

ננתח את סיבוכיות אלגוריתם הפצת ההודעה, כולל גילוי הסיום.

באופן כללי, כאשר מנתחים סיבוכיות של אלגוריתם, מסתכלים על מספר ההודעות שנשלחו, ועל זמן הריצה.

במקרה הזה, מספר ההודעות שנשלחו: בכל קשת שאינה על העץ עוברות 4 הודעות (פעמיים M ופעמיים Ack). על כל קשת בעץ עוברות 2 הודעות (M ו־Ack). בסך הכל: $4 \cdot (|E| - n + 1) + 2 \cdot (n - 1)$.

לצורך חישובי זמן, נכניס כמה הנחות:

- זמן החישוב בקודקוד הוא 0.
- הזמן שלוקח להודעות לעבור על הקשתות הוא כלשהו. רק לצורך חישוב הסיבוכיות, נגדיר את הזמן שלוקח להודעה לעבור על הקשת האיטית ביותר כיחידת זמן אחת.
- כשמדברים על סיבוכיות זמן, מדברים על המקרה הגרוע ביותר.

במקרה שלנו, סיבוכיות הזמן היא $\mathcal{O}(n)$.

1.3 Diffusing Computation

1.3.1 תיאור הבעיה

יש קודקוד אחד שמתחיל את האלגוריתם. הוא בוחר באופן אקראי חלק מהשכנים שלו, ושולח להם הודעה. כל שכן שמקבל את ההודעה, בוחר באופן אקראי חלק מהשכנים שלו ושולח להם את ההודעה.

נקרא לקודקוד שמתחיל Root. נרצה שהוא יקבל Signal כשה־Diffusing Computation דעך (הסתיים), כלומר Termination Detection of Diffusing Computation.

1.3.2 הצעה לאלגוריתם

לעשות פעמיים Broadcast Echo, ולוודא שפעמיים אין "זבובים" שהתקבלו.

נכונות האלגוריתם נובעת מתכונת ה־FIFO של הקשתות בגרף.

1.3.3 הצעה טובה יותר

האלגוריתם יהיה דומה ל־Broadcast Echo: כל מעבד שמקבל הודעה, בוחר תת־קבוצה אקראית של קבוצת השכנים שלו, ושולח לו את ההודעה. ברגע שקיבל Ack מכל אחד מהשכנים שאליהם נשלחה ההודעה, הוא ישלח Ack לשכן ששלח לו את ההודעה במקור.

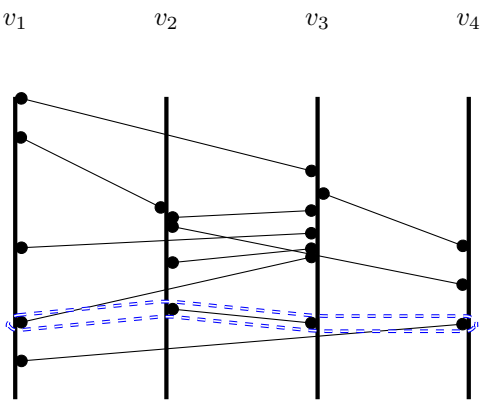
כדי שהאלגוריתם יעבוד, כל שכן צריך ליצור עותק וירטואלי של עצמו (Fork) בכל פעם שהוא מקבל הודעה חדשה לעיבוד. בכל הודעה (M או Ack), יהיה כתוב השם הורטואלי של המעבד, כדי שאפשר יהיה להבין לאיזה מהעותקים של הצומת ההודעה מיועדת.

1.3.4 שיפור להצעה

במקום שהקודקודים יעשו Fork, אפשר להרים את העצים למעלה, ל־v הראשון.

כלומר, כל קודקוד ידע כמה Ack־ים הוא צריך לקבל מכל אחד מהשכנים שלו. יהיה לו רק אבא אמיתי אחד, שאליו הוא ישלח Ack כשהחישוב ידעך בכל תת־העץ שלו. אם הוא קיבל הודעה חדשה לעיבוד, לפני שתת־העץ שלו סיים לבצע את החישובים שלו, הוא ישלח Ack מיידי.

עדיין נשמר כאן מבנה של עת, והסיבוכיות השתפרה (לפחות סיבוכיות הזמן).



איור 1.2: דוגמה ל־Snapshot חוקי. אין הודעה שהתקבלה לפני ה־Snapshot אבל נשלחה אחרי ה־Snapshot.

1.3.5 כמה יוזמים

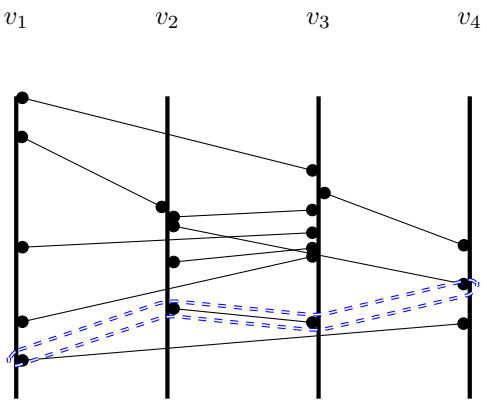
אם רוצים שיהיו כמה יוזמים ל־Diffusing Computation: אפשר לפרוש עץ, ובאמצעות Broadcast Echo על העץ לבדוק אם כולם סיימו.
זה נקרא Static Termination Detection of Diffusing Computation.

1.4 Snapshots

במודל שלנו, אין זמן כללי, אבל כל צומת יכול להגדיר שעון מקומי. השעון יגדל ב־1 עם כל הודעה שהתקבלה.
אי אפשר לעשות Snapshot לכל החישוב בזמן כלשהו, כי אי אפשר לדבר עם כל הקודקודים בו־זמנית. אפשר לקחת אוסף של נקודות זמן בכל קודקוד, כך שאפשר יהיה להמשיך את ריצת המערכת בצורה תקינה.

הגדרה 5 (Snapshot): Snapshot הוא אוסף של נקודות זמן במעבדים, כך שאין הודעות שהתקבלו לפני ה־Snapshot ונשלחו אחרי ה־Snapshot.

ניתן למצוא דוגמה ל־Snapshot חוקי באיור 1.2, ודוגמה ל־Snapshot שאינו חוקי באיור 1.3.
דרוש אלגוריתם שמוצא Snapshot.
ניתן לעשות זאת בקלות בעזרת Broadcast: כל מעבד עושה את ה־Snapshot שלו כשמגיעה הודעת ה־Broadcast. נכונות ה־Snapshot תתקיים מתכונת ה־FIFO של הקשתות.
את המשך הדיון על הבעיה נמשיך בשיעור הבא.



איור 1.3: דוגמה ל־Snapshot שאינו חוקי. המעבד v_1 ב־Snapshot קיבל את ההודעה שנשלחה מ־v_4, אבל ב־Snapshot v_4 עדיין לא שלח את ההודעה.

פרק 2

שיעור 2[1]

ה־Broadcast הוא מקרה פרטי של Diffusing Computation, וה־Echo הוא ה־Termination Detection.

2.1 Snapshot

לכל קודקוד $i \in \{1, \ldots, n\}$ יש מונה משלו – t^i. המונה מתקדם בכל פעם ש־i מקבל הודעה. זהו מונה א־סינכרוני.

הגדרה 6 (Snapshot): וקטור $\left(t_1^1, t_1^2, \ldots, t_1^n\right)$ הוא *Snapshot* אם אין הודעה M_{ij} (מ־i ל־j) כך ש־M_{ij} נשלחה ב־t_M^i כך ש־$t_M^i > t_1^i$ והתקבלה ב־t_M^j כך ש־$t_M^j \leq t_1^j$.

אלגוריתם טריויאלי ל־Snapshot הקודקוד שרוצה להתחיל חישוב של Snapshot מצלם את המצב המקומי של המעבד, ומייד שולח הודעה לשכנים שיבצעו Snapshot. ככה גם כל מי שמקבל את ההודעה מצלם את המצב, ושולח לשכנים. אם ההודעה כבר התקבלה, אפשר להתעלם ממנה.

למעשה, זה סוג של Diffusing Computation, ואפילו יש לנו אפשרות לקבל Termination Detection בעזרת Echo.

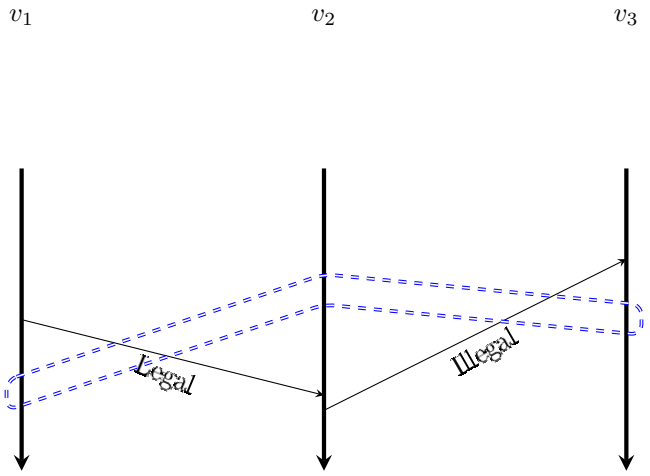
הערה ?: כדאי לשים לב שב־Snapshot חוקי, אין הודעה שנשלחה אחרי והגיעה לפני. אולם יכול להיות מצב שהודעה נשלחה לפני והגיעה אחרי. ניתן לראות דוגמה לכך באיור 2.1.

כל קודקוד, ברגע שהוא מקבל הודעה על Snapshot, מבצע צילום ומתחיל להקליט את כל ההודעות על הקשתות האחרות. ברגע שמקבלים הודעת Snapshot משכן אחר, אפשר להפסיק להקליט הודעות שמגיעות ממנו. ככה אפשר לשמור בתוך ה־Snapshot גם את ההודעות שאמורות להתקבל מהשכנים. ככה לא ייתכן שהודעות הולכות לאיבוד בגלל ה־Snapshot.

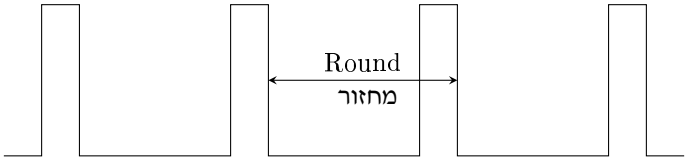
אחר כך, כשחוזרים מתוך Snapshot, כל צומת יכול לסמלץ את ההודעות שאמורות להגיע אליו לפני שהוא מעבד את ההודעות שהגיעו בפועל אחרי החזרה מה־Snapshot.

מה קורה אם כמה קודקודים מתחילים את ה־Snapshot באופן ספונטני, ולא רק קודקוד אחד? זה בדיוק אותו הדבר. אפשר לחשוב שיש שורש וירטואלי, ששלח לכל אחד מהקודקודים שהתחילו את ה־Snapshot הודעת Snapshot, ואז הכל אותו הדבר.

[1] סיכום לשיעור שהועבר בתאריך 10.03.2013.



איור 2.1: דוגמה ל־Snapshot עם הודעה חוקית ובלתי־חוקית



איור 2.2: מחזורים ברשת סינכרונית

2.2 רשת סינכרונית

נסתכל על מודל חדש: רשת סינכרונית. ההנחות:

1. יש שעון (קוצב) שמייצג פולסים. כל פולס מגיע לכל הקודקודים ברשת בו־זמנית.
2. כל ההודעות שקודקוד שלח בתחילת המחזור מגיעות לשכנים לפני הפולס הבא.
3. התנהלות החישוב: כל קודקוד מאותחל למצב התחלתי. בתחילת המחזור, הקודקוד שולח הודעות בהתאם למצב הנוכחי שלו. כשמגיע הפולס, כל ההודעות מהשכנים כבר התקבלו, וכל קודקוד מחשב מצב חדש בהתאם למצב הקודם, ובהתאם להודעות שהתקבלו.

ניתן לראות הסבר גרפי למחזורים ברשת סינכרונית באיור 2.2.

סיבוכיות הזמן של Broadcast & Echo במודל א־סינכרוני היא $\mathcal{O}(n)$, כאשר n הוא מספר הצמתים. במודל סינכרוני, הסיבוכיות היא $\mathcal{O}(d)$, כאשר d הוא קוטר הגרף (graph diameter) של הרשת. העץ שנוצר ברשת סינכרונית על ידי Broadcast & Echo הוא עץ BFS. נרצה לייצר עץ BFS ברשת א־סינכרונית.

נסתכל על בעייה כללית יותר: נניח שנתון לנו אלגוריתם שתוכנן לעבוד ברשת סינכרונית. נרצה להריץ אותו ברשת א־סינכרונית בצורה נכונה.

2.3 סינכרונייזר – Synchronizer

סינכרונייזר יודע להריץ אלגוריתם סינכרוני ברשת א־סינכרונית. יהיו לנו שלושה סוגים של סינכרונייזרים: α, β ו־γ.

הגדרה 8 (Safe): נאמר כי קודקוד v הוא *Safe*, ונסמן $\text{Safe}_i(v)$, אם כל ההודעות ש־v שלח אחרי הפולס ה־i הגיעו ליעדן.

קודקוד יכול לסמלץ או לייצר לעצמו את הפולס ה־$i+1$ אם כל שכניו הם Safe_i.

איך נדע אם v הוא Safe_i? על כל הודעה של האלגוריתם, נשלח Ack. ברגע שקודקוד מסויים קיבל Ack על כל ההודעות ששלח בפולס ה־i, הוא Safe_i.

2.3.1 סינכרונייזר α

כל קודקוד יודיע לכל השכנים שלו שהוא Safe_i ברגע שהוא הופך ל־Safe_i. ברגע שקודקוד מסויים קיבל Safe_i מכל שכניו, הוא יסמלץ את הפולס ה־$i+1$.

ננתח את הסיבוכיות של סינכרונייזר α: נסתכל על התקורה למחזור. על כל הודעה שהאלגוריתם שולח, שולחים גם Ack. זה מכפיל את התעבורה פי 2, ולכן נזניח. אולם ברגע שהופכים ל־Safe_i, מעבירים הודעה. לכן, בכל מחזור עוברות $2 \cdot E$ הודעות Safe_i.

סיבוכיות הזמן היא $\mathcal{O}(1)$: תוך 3 פעמים של ההודעה האיטית ברשת נקבל מכל השכנים Safe_i, ונעבור לפולס הבא.

הערה 9 (סימונים): E, $|E|$ או m מייצגים את מספר הקשתות ברשת, n מייצג את מספר הקודקודים ברשת, h מייצג את גובה העץ.

2.3.2 סינכרונייזר β

נתון לנו עץ פורש ברשת (שנוצר בעזרת Broadcast & Echo). כל עלה ישלח לאבא שלו שהוא Safe_i כשהוא מקבל את כל ה־Ack־ים על ההודעות ששלח. כל צומת בעץ ישלח Safe_i לאבא שלו רק כאשר הוא Safe_i, וכך גם כל תת־העץ שלו. כשהשורש יקבל שכל הילדים שלו הם Safe_i, הוא ישלח Broadcast עם $Pulse_{i+1}$.

סיבוכיות סיבוכיות ההודעות היא $2 \cdot n$. זמן: $2 \cdot h$, אחד על Broadcast, והשני על Echo.

2.3.3 סינכרונייזר γ

נסתכל על המצב הבא: הרשת שלנו מחלוקת ל־Cluster־ים, שבכל אחד מהם עץ פורש, וקודקוד אחד שהוא ה־Cluster Center. כמו כן, בין ה־Cluster־ים יש גשרים (Bridges). קודקוד שיש עליו קשת שהיא גשר יודע שהקשת היא גשר. נסמן ב־B את מספר הגשרים.

איך נראה Pulse?

	הודעות	זמן
α	$2 \cdot \|E\|$	$\mathcal{O}(1)$
β	$2 \cdot n$	$2 \cdot h$
γ	$4 \cdot n + 2 \cdot B$	$\mathcal{O}(h')$

טבלה 2.1: סיבוכיות הזמן וההודעות של הסינכרונייזרים השונים

1. כל שורש (Cluster Center) שולח הפצה (Broadcast) של $Pulse_i$ בעץ של ה־Cluster.
2. בתוך ה־Cluster, כל צומת שמקבל $Pulse_i$, שולח $Pulse_i$ לכל ילדיו בעץ.
3. כל צומת מבצע את החישוב של ה־Pulse ה־i.
4. אחרי שמקבלים Ack על כל הודעה שנשלחה, הקודקוד הוא Safe_i.
5. כאשר קיבלתי Safe_i מכל ילדיי בעץ (של ה־Cluster), ואני Safe_i, אשלח Safe_i להורה בעץ.
6. כאשר שורש מקבל Safe_i מכל ילדיו בעץ, הוא מודיע שהוא ClusterSafe_i.
7. עושים Broadcast ב־Cluster שאומר שה־Cluster הוא ClusterSafe_i.
8. כל קודקוד שמקבל ClusterSafe_i מההורה בעץ שולח ClusterSafe_i לכל ילדיו בעץ, ועל גשרים מ־B הסמוכים אליו.
9. כל קודקוד, אם קיבל:
   (א) ClusterSafe_i על כל הקשתות מ־B הסמוכות אליו.
   (ב) Ack עבור ClusterSafe_i מכל ילדיו בעץ (למעט אם הוא עלה).

   אם שני התנאים מתקיימים, אזי הוא ישלח Ack ל־ClusterSafe_i להורה שלו בעץ.
10. כאשר שורש מקבל Ack ל־ClusterSafe_i מכל ילדיו בעץ, ו־ClusterSafe_i מכל B שסמוך אליו, הוא עובר ל־$Pulse_{i+1}$.

סיבוכיות סיבוכיות ההודעות: בתוך העצים יש $4 \cdot n$ הודעות, ויש גם $2 \cdot B$ הודעות. בסך הכל, סיבוכיות ההודעות היא $4 \cdot n + 2 \cdot B = \mathcal{O}(n + B)$.
סיבוכיות הזמן: $\mathcal{O}(h')$ – גובה העץ הגבוה ביותר ביער.

2.3.4 סיכום

טבלה 2.1 מפרטת את הסיבוכיות של הסינכרונייזרים השונים.
סינכרונייזר α מספק סיבוכיות זמן טובה מאוד (זמן קבוע), אך סיבוכיות ההודעות גדולה מאוד. סינכרונייזר β מספק סיבוכיות הודעות טובה, אך סיבוכיות הזמן עשוייה להיות גדולה מאוד. סינכרונייזר γ נהנה משני העולמות: הן סיבוכיות הזמן שלו והן סיבוכיות ההודעות שלו טובות יותר (אך לא הכי טובות כשכל אחת מהן נמדדת בנפרד).

הביצועים של סינכרונייזר γ תלויים מאוד בבחירת ה־Cluster־ים. לכן, אנחנו נחפש חלוקה ל־Cluster־ים שתיתן סיבוכיות הודעות כמו בסינכרונייזר β, וזמן כמו סינכרונייזר α. כלומר, Cluster־ים בגובה קטן, עם מעט גשרים ביניהם. נראה אלגוריתם לבניית ה־Cluster־ים באופן הזה.

2.4 בניית Cluster־ים

להלן תיאור אלגוריתם לבניית Cluster־ים:

קודקוד אחד מתחיל. הוא ימצא עץ BFS באמצעות שליחת $test_1$ שעושה Broadcast & Echo לשכנים עד הרמה ה־1. לאחר סיום האיטרציה ה־i, שולחים $test_{i+1}$.

ברגע שצומת מקבל את הודעת ה־$test$ הראשונה, הוא ישלח Ack. לכל הודעת $test$ נוספת, הוא ישלח $Reject$. ברגע שהשורש יקבל Ack_{i+1} מכל הילדים שלו, הוא ימשיך לאיטרציה ה־$i+2$.

עכשיו כל צומת זוכר מי האבא שלו ומי הבנים שלו. כל קודקוד ששולח $test$ סופר כמה Ack־ים הוא קיבל, כלומר כמה ילדים יש לו ב־$i+1$. הוא סוכם אותם, ושולח את המספר לאבא שלו. ככה הסכום מתבצע עד השורש, והוא מקבל את מספר הצמתים שנוספו לו בשכבה ה־$i+1$. כלומר, השורש יודע כמה צמתים נוספו לו בכל רמה, ובפרט הוא יודע כמה צמתים יש לו בעץ.

ברגע שמספר הצמתים שנוספו לעץ לא מכפיל את גודל העץ פי 2 לפחות, השורש אומר שלא מרחיבים יותר את העץ. כלומר, מוסיפים שכבה רק אם היא לפחות מכפילה אתמספר הקודקודים בעץ. בתנאי הזה מבטיח ש־$h' < \log n$.

עכשיו נלך ב־DFS, וכל עלה שנגיע אליו יהיה שורש של עץ (Cluster) חדש. בעזרת ה־DFS (שימשיך לתוך ה־Cluster־ים רקורסיבית), נכסה את כל הצמתים בגרף.

מובטח לנו שבכל עץ, $h' < \log n$. מה הגודל של B? נספור כל קשת ב־B לפי הפעם הראשונה שעבר עליה $test$ ו־Ack. Cluster בגודל n_i מוסיף לכל היותר n_i גשרים. לכן:

$$|B| \leq \sum_i n_i = n$$

לכן, סיבוכיות ההודעות של סינכרונייזר γ תהיה $4 \cdot n + 2 \cdot B = \mathcal{O}(n)$, וסיבוכיות הזמן היא $\mathcal{O}(h') \leq \mathcal{O}(\log n)$, כי גובה ה־Cluster הגבוה ביותר חסום על ידי $\log n$.

פרק 3

שיעור 3[1]

3.1 בחירת מנהיג

3.1.1 תיאור הבעיה

נתונה קבוצה של מעבדים, שרוצה לבחור מעבד שיהיה ה"מנהיג", כלומר לעשות חישוב שבסופו כל המעבדים מקבלים את הביט 0, פרט למנהיג שמקבל 1.

במצב בו כל המעבדים זהים לחלוטין, עד רמת ה־id, לא ניתן לבחור מנהיג, שכן המעבדים הם לא distinguishing. עובדה זו היא גם תוצאה של מבנה הרשת. לא ניתן לבחור מנהיג כאשר הרשת סימטרית, למשל מעגל, בניגוד למבנה כוכבי.

לכן, נניח את ההנחות הבאות:

- לכל מעבד שם שונה (באורך $\mathcal{O}(\log n)$).
- הרשת היא בטופולוגיה של מעגל.

3.1.2 אלגוריתם ראשוני

כאשר צומת מתעורר, הוא שולח לשני שכניו הודעה עם ה־id שלו. כל צומת שמקבל הודעה מפיץ אותה אם היא גדולה מה־id שלו. אחרת, הוא מפיץ חזרה את ה־id שלו.

סיבוכיות ההודעות של האלגוריתם היא $\mathcal{O}(n^2)$.

3.1.3 אלגוריתם יעיל יותר (בפאזות)

כשצומת מתעורר (בין אם בכוחות עצמו, ובין אם ע"י הודעה משכן), הוא פועמד ושולח את שמו לשכניו. אם הוא גדול משני שכניו, אז הוא local max, ועובר לפאזה הבאה ושולח הודעה נוספת לשני הכיוונים. צומת שאינו local max הוא פרש (לא מנהיג), וככזה הוא פשוט עושה relay להודעות. האלגוריתם מסתיים כאשר צומת מקבל את השם שלו, ואז הוא מנהיג.

לאלגוריתם יש $\log n$ פאזות, כי בכל פאזה חצי מהצמתים פורשים. בכל פאזה נשלחות $2 \cdot n$ הודעות. לכן, בסך הכל, $2 \cdot n \cdot \log n$ הודעות נשלחות.

[1] סיכום לשיעור שהתקיים בתאריך 17.03.2013. מבוסס על סיכום של עופרי זיו.

3.1.4 אלגוריתם לרשת סינכרונית

נניח כעת שהרשת מקיימת:

- הרשת היא מעגל סינכרוני.
- כולם מתחילים ביחד.
- כולם יודעים מראש את n.

האלגוריתם כל צומת i סופר $n/2 - i$ בכל שעון, וכאשר מגיע ל־0, מודיע לכולם שהוא המנהיג. במקרה הזה, המנהיג הוא בעל ה־id הנמוך ביותר.

סיבוכיות n הודעות.

על מנת לוותר על ההנחה שכולם מתעוררים ביחד, נשנה את האלגוריתם כך שכשצומת מתעורר הוא שולח הודעת wake-up לכל כיוון, ונאמר שהודעה זו עוברת פעם אחת בכל קשת. נקבל שתוך לכל היותר $n/2$ פאזות כולם התעוררו.

לכן, כדי להבטיח את נכונות האלגוריתם, נאמר כי כעת כל צומת סופר $|\mathrm{id}| + n/2 + n$ פאזות. זה נועד לפתור את המצב שבו $i+1$ מתעורר $n/2$ פאזות לפני $i_{\min}$.

3.1.5 גודל הרשת לא ידוע

אפשר גם לוותר על ההנחה שגודל הרשת ידוע, לפי האלגוריתם הבא:

כל צומת סופר $2^{|\mathrm{id}|}$ פולסים, ושולח הודעה לשני הכיוונים. כאשר צומת מקבל הודעה, הוא מעביר אותה הלאה אם היא קטנה מה־id שלו, לאחר $2^{|\mathrm{id}|}$ פולסים (מהרגע שקיבל את ההודעה).

גם כאן הנחנו שיש צורך בהודעת התעוררות בתחילת האלגוריתם.

בזמן שצומת 1 שולח הודעה, צומת 2 עובר לכל היותר מחצית מהצמתים ברשת. לכן, סיבוכיות ההודעות:

$$n + \frac{n}{2} + \cdots + 1 \leq 2 \cdot n$$

3.1.6 חסם תחתון למספר ההודעות

נרצה להוכיח שבמודל א־סינכרוני, החסם התחתון של מספר ההודעות הוא פונקציה של $n \cdot \log n$ הודעות.

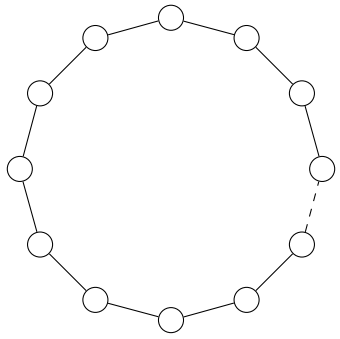
בכדי לבחור מנהיג, חייבת להיות שרשרת הודעות מאחר לכולם.

הגדרה 10 (שרשרת פתוחה): בשרשרת פתוחה נשלחו כל ההודעות על כל הקשתות, פרט לקשת אחת כלשהי. ראה דוגמה באיור 3.1.

נאמר כי נשלחו $M(n)$ הודעות על קשתות הרשת.

למה 11:

$$M(2 \cdot n) \geq 2 \cdot M(n) + n/2$$



איור 3.1: שרשרת פתוחה. לא עברה הודעה בקשת החסרה (המקווקוות), כלומר הודעות שנשלחו על הקשת הזו לא הגיעו לצד השני.

איור 3.2: מקרה הבסיס של האינדוקציה, כאשר $n = 2$: $M(2) = 2$.

הוכחה: נוכיח באינדוקציה.

בסיס: כל צומת חייב לקבל הודעה ולשלוח הודעה. למשל, $M(2) = 2$, $M(3) = 3$ (ראה איורים 3.2 ו־3.3).

צעד: ניתן לאלגוריתם לרוץ על מעגל R_1 של n קודקודים, ולאחר מכן על מעגל R_2 בגודל n כך שבשניהם תישאר קשת פתוחה – (v_1, w_1) ב־R_1 ו־(v_2, w_2) ב־R_2. כעת, נחבר את v_1 עם v_2, ואת w_1 עם w_2. ראה דוגמה באיור 3.4.

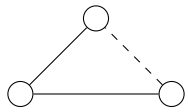
כעת, צד אחד צריך להעביר את ההודעה של המנהיג, כלומר ליצר שרשרת הודעות באורך לפחות $n/2$, כדי שהצד השני יכיר במנהיג. בפועל, אנחנו נחבר רק בין v_1 ל־v_2 או רק בין w_1 ל־w_2, אבל לא את שניהם.

כיצד נבחר איזו קשת ליצור? נשחרר את שתי הקשתות. נראה מי מהן מעבירה יותר הודעות (יותר מ־$n/2$), ועם הידע הזה, נחזיר את הריצה לאחור, ונוסיף רק את הקשת שהעבירה יותר הודעות.

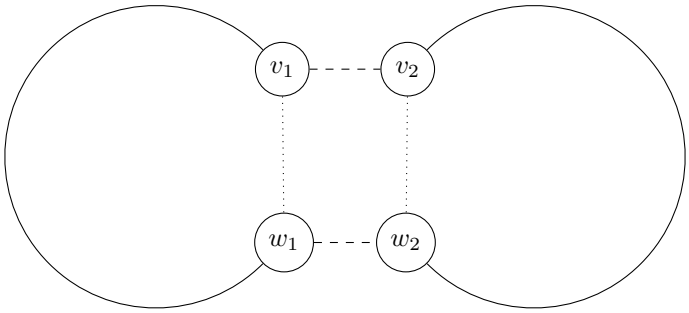
קיבלנו שרשרת פתוחה של $2 \cdot n$ צמתים, כך שעברו $M(n)$ הודעות ב־R_1 (לפי הנחת האינדוקציה), עוד $M(n)$ הודעות ב־R_2 (לפי הנחת האינדוקציה), ועוד $n/2$ הודעות לאחר חיבור R_1 ל־R_2. כלומר:

$$M(2 \cdot n) \geq 2 \cdot M(n) + n/2$$

■



איור 3.3: מקרה הבסיס של האינדוקציה, כאשר $n = 3$: $M(3) = 3$.



איור 3.4: חיבור שני המעגלים R_1 ו־R_2

ברשת סינכרונית, סיבוכיות ההודעות היא $\mathcal{O}(n)$. סיבוכיות הזמן (מספר הסיבובים) – $n \cdot 2^{|\mathrm{id}_{\min}|}$.
מהלמה, אפשר להסיק את המשפט הבא:

משפט 12: אם סיבוכיות הזמן חסומה על ידי T, אזי קיימים id־ים בגודל $M_n(T,n)$ כך שהאלגוריתם ישלח לפחות $\Omega(n \cdot \log n)$ הודעות.

ברשת Full-Mesh א־סינכרונית, סיבוכיות ההודעות נותרת $\mathcal{O}(n \cdot \log n)$. עבור גרף כלשהו, ניתן לקבל סיבוכיות הודעות של $\mathcal{O}(|E| + n \cdot \log n)$.

3.1.7 הקשר לעץ פורש

מציאת מנהיג שקולה למציאת עץ פורש: אם נתון עץ פורש, אז המנהיג הוא שורש העץ. אם נתון מנהיג, אפשר לפרוש עץ כשהמנהיג יוזם Flooding.
גם אם העץ הפורש אינו מושרש, קיימת רדוקציה לינארית מכל אחד מהמקרים.
אם נתון עץ פורש שאינו מושרש, אפשר לבחור את המנהיג באמצעות Collapsing ב־$\mathcal{O}(n)$ הודעות (עד שנותרים שני קודקודים, והם מחליטים לפי ה־id).
אם נתון מנהיג, קל למצוא עץ פורש מושרש (בעזרת Flooding, כמו קודם), שהוא בפרט עץ פורש, ב־$\mathcal{O}(|E|)$ הודעות.

3.2 עץ פורש מינימלי – Minimum Spanning Tree (MST)

כדי לא לאפשר קיום של שני עצים פורשים מינימאליים, לא נאפשר קיום של שתי קשתות בעלות אותו המשקל.
נשיג זאת באופן הבא: קשת בין id_1 ו־id_2, משקלה יהיה:

$$\min\{\mathrm{id}_1, \mathrm{id}_2\} \mid \max\{\mathrm{id}_1, \mathrm{id}_2\}$$

כלומר, שרשור של ה־id המינימלי עם ה־id המקסימלי.
כמו כן, נניח כי הגרף ינו מכוון, וכל קודקוד מכיר רק את הקשתות הסמוכות אליו.
נראה אלגוריתמים לפתרון הבעיה בשיעור הבא.

פרק 4

שיעור 4[1]

4.1 עץ פורש מינימלי

ההנחות שלנו:

1. אין שתי קשתות בעלות אותו המשקל.
2. ניתן לבצע השוואה לקסיקוגרפית של קשתות.

האלגוריתם של בורורקה (*Boururka*) מתואר באלגוריתם 1.4.

למה 13: אין מעגלים בבחירת הקשתות (באיטרציה הראשונה).

הוכחה: נניח בשלילה שיש מעגלים. נסמן את הקשתות בו ב־$e_1, e_2, \ldots, e_k$ ($\forall i \neq j . e_i \neq e_j$). אזי $e_i < e_{i+1}$, וכן $e_k < e_1$ (לפי בחירת הקשתות), בסתירה. ■

ההבדל בין Core ל־Fragment: Core הוא זוג קודקודים. Fragment הוא עץ פורש.

למה 14: אין מעגלים גם באיטרציות הבאות.

[1] סיכום לשיעור שהתקיים בתאריך 07.04.2013.

אלגוריתם 1.4 האלגוריתם של *Boururka* למציאת עץ פורש מינימלי

1: $\mathcal{C}_0 \leftarrow \{V\}$, $\mathcal{F}_0 \leftarrow \{V\}$ ▷ vertices
2: $k \leftarrow 0$
3: **while** there is more than one fragment in $\mathcal{F}_k$ **do**
4: **for all** $F_i \in \mathcal{F}_k$ **do** $\mu_i \leftarrow$ minimum weight outgoing edge in F_i
5: $\mathcal{C}_{k+1} \leftarrow \{e_j \mid e_j = \mu_x = \mu_y, x \neq y\}$
▷ cores, collection of edges selected in both directions
6: $\mathcal{F}_{k+1} \leftarrow \{\forall c \in \mathcal{C}_{k+1} \mid \text{the connected graph around } c \text{ is in } \mathcal{F}_k \cup \{\mu\}\}$
7: $k \leftarrow k + 1$
8: **end while**

הוכחה: נסתכל על כל רכיב קשירות כקודקוד. עם אותו הטיעון כמו בלמה 13, נקבל שאין מעגלים. ■

היינו רוצים לממש את האלגוריתם הזה בצורה מבוזרת, כלומר לממש אלגוריתם מבוזר שיחפש עץ פורש מינימלי.

הצעה 15: כל צומת v ישלח $Connect\,(L=0,v)$ על הקשת המינימלית. כל זוג קודקודים ששלחו וקיבלו $Connect$ האחד לשני יהפכו ל־$Core$.

אחר כך עושים $Find\,(f\text{-}name, L=1)$ (כאשר $f\text{-}name$ הוא השם של ה־$Fragment$) שבו כל קודקוד שולח בדיקה (הודעת $Test\,(L=1, f\text{-}name)$) לקשת המינימלית שיוצאת ממנו. מתקבלת התשובה OK אם היעד לא ב־$f\text{-}name$, או $Reject$ אחרת. התהליך ממשיך עד שמוצאים קשת שיוצאת מה־$Fragment$ לכל קודקוד.

כמו כן, כל קודקוד מעביר את $Find$ לכל אחד מהילדים שלו בעץ. ברגע שקודקוד מוצא קשת מינימלית שיוצאת ממנו אל מחוץ ל־$Fragment$, וקיבל הודעת $EchoMininalWeightOutgoingEdge\,(L, f\text{-}name)$ מכל אחד מהילדים שלו, הוא מעביר הלאה את המשקל המינימלי. כמו כן, כל קודקוד זוכר מאיזה שכן קיבל את המינימום. ככה, השורש יודע מה המשקל של הקשת הקטנה ביותר היוצאת מה־$Fragment$, והמסלול למציאתה יזוע. עליה שולחים $Connect\,(L, f\text{-}name)$, וחוזרים חלילה.

האלגוריתם הזה נכתב על ידי *Gallagher*, *Humblet* ו־*Spira*. המאמר שמסביר את האלגוריתם לא כל כך מובן, אז עדיף להבין את מה שכתוב כאן.

כדי שהאלגוריתם יעבוד על רשת א־סינכרונית, נבצע שינוי יחיד: קודקוד לא עונה להודעה עד שהוא מגיע לרמה (Level) שלה. כלומר, קודקוד ברמה ℓ לא עונה להודעות ברמה יותר גדולה מ־ℓ. עכשיו, לכל צומת יש את המצב שלו, שאינו בהכרח זהה למצב של צמתים אחרים.

ניתוח סיבוכיות

סיבוכיות הודעות באלגוריתם יש לנו הודעות $Test$, $Reject$ ו־OK. היו גם $Connect$, $Find$ ו־$EchoMinimalWeightOutgoingEdge$.

הודעת $Test$ נשלחת $|E|$ פעמים (פעם אחת בכל קשת), והיא תמיד תקבל תשובה מסוג $Reject$ או OK.

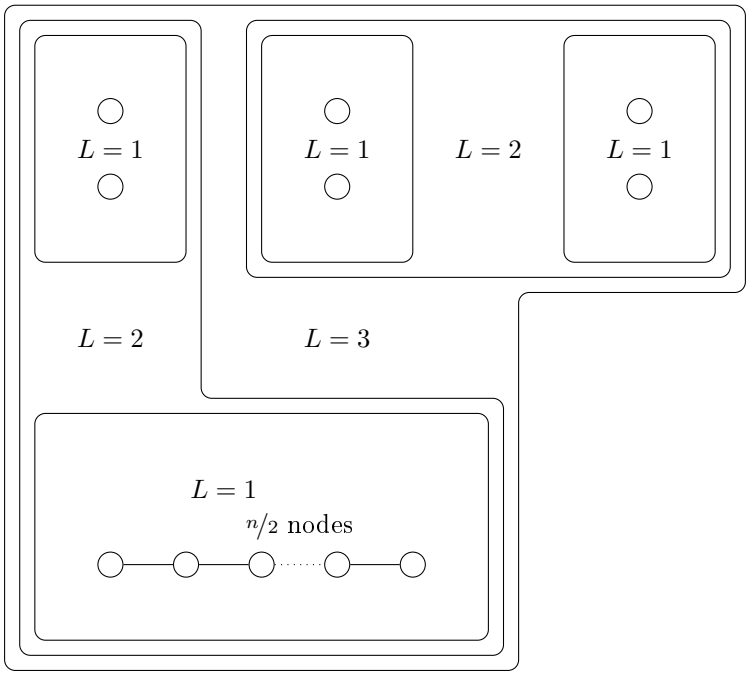
עבור כל רמה, נשלחות n הודעות $Find$. כנ"ל לגבי $EchoMinimalWeightOutgoingEdge$. לכן, בסך הכל נשלחות $n \cdot \log n$ הודעות $Find$.

גם $Connect$ נשלחת $n \cdot \log n$ פעמים, כי בכל רמה נשלחים לכל היותר n הודעות $Connect$ (כי n חוסם את מספר ה־Fragment־ים).

כל קודקוד שולח $Test$ חוזר לכל היותר פעם אחת בכל רמה. לכן, יש $n \cdot \log n$ הודעות $Test$ חוזר. כנ"ל לגבי התשובות ל־$Test$.

בסך הכל, האלגוריתם שולח $2 \cdot |E| + 5 \cdot n \cdot \log n$ הודעות, שזה $\mathcal{O}\left(|E| + n \cdot \log n\right)$ הודעות.

סיבוכיות זמן נסתכל על ה־Fragment עם הרמה המינימלית בגרף. שום דבר לא מעכב אותו, כי הוא הרמה הנמוכה ביותר. הוא עושה Broadcast & Echo עם קצת הודעות $Test$ (בכל צומת), בזמן של $\mathcal{O}\left(n\right)$. מכיוון שיש $\log n$ רמות, הזמן הכולל חסום על ידי $\mathcal{O}\left(n \cdot \log n\right)$.



איור 4.1: הוכחת הדיקות החסם לזמן הריצה של האלגוריתם של Gallagher, Humblet ו־Spira

נראה שזהו החסם העליון המינימלי: נסתכל על דוגמת ההרצה שבאיור 4.1. תמיד יש Fragment בגודל של יותר מ־$n/2$ צמתים, ולכן זמן הריצה שלו ייקח בכל רמה $\mathcal{O}(n)$. בסך הכל, $\mathcal{O}(n \cdot \log n)$.

הערה 16: מכירים אלגוריתם (חדש יותר) שמוצא עץ פורש מינימלי ב־$\mathcal{O}(n)$ זמן, וסיבוכיות ההודעות שלו נשארת $\mathcal{O}(|E| + n \cdot \log n)$. האלגוריתם דואג שלא יהיה רכיב אחד מאוד גדול. זה נעשה בעזרת ההגבלה שברמה ℓ, הגודל של כל Fragment יהיה חסום על ידי 2^ℓ (כלומר $\log|\text{Fragment}| \leq \ell$) כאשר $n \to \infty$.

4.2 Maximal Independent Set

אנחנו רוצים לדעת מה הרדיוס k של השכנים של קודקוד ברשת שמשפיעים על הפתרון שלו. למשל, בצביעת גרפים, מה המרחק של הצומת הרחוק ביותר שמשפיע על הצבע של צומת מסויים?

אפשר למצוא את k בעזרת המודל הסינכרוני, כל שכל הצמתים מתעוררים ביחד, וגודל ההודעות שנשלחות גדול כרצוננו. אם זמן הריצה של האלגוריתם חסום על ידי k, אזי ההודעה הרחוקה ביותר הצליחה לעבור k צמתים בלבד, ולכן k הוא הרדיוס המבוקש.

פרק 5

שיעור 5[1]

5.1 הרדיוס של בעיה

אנחנו רוצים לחקור לוקאליתי (מקומיות) של בעיה. כלומר, מה הרדיוס של שכנים שקודקוד צריך לדעת את הקלטים שלהם כדי להוציא את הפלט (שלו) של כל החישוב. זה נקרא הרדיוס של הבעיה.

למשל, אם רוצים לצבוע ב־n צבעים (ויש n מעבדים), אז כל אחד לוקח את השם שלו בצור הצבע, והרדיוס הוא 0.

5.2 בעיית הצביעה במעגל

המודל שלנו:

- סינכרוני.
- כל המעבדים מתעוררים ביחד.
- ההודעות גדולות כרצוננו.
- לכל מעבר יש id כך ש־$1 \leq id \leq n$ (מיוצג ב־$\mathcal{O}(\log n)$ ביטים).

באופן כללי, אם אפשר לפתור בעיה ב־r סיבובים (פולסים) במודל סינכרוני, אזי הרדיוס של הבעיה הוא r, כי כל מעבד יכול לשלוח בכל סיבוב לכל שכנוי את כל הידוע לו, ואחרי r סיבובים כל מעבד יסמלץ את החלק שלו מהאלגוריתם המבוזר.

נגדיר את המזהה של הקודקוד i אחרי j סיבובים להיות ID_i^j.

באלגוריתם שלנו, כל קודקוד משנה את ה־ID שלו בכל סיבוב. מאתחלים: $ID_i^0 = id_i$.

כדי לחשב את ID_i^j, משווים את ID_i^{j-1} ו־ID_{i-1}^{j-1}. ID_i^j יהיה האינדקס של הביט הראשון שבו הם שונים. במילים אחרות:

$$ID_i^j = \mathrm{LSB}\left(ID_i^{j-1} \oplus ID_{i-1}^{j-1}\right)$$

[1] סיכום לשיעור שהתקיים בתאריך 14.04.2013.

לביטוי הזה נשרשר את ערך הביט הזה ב־ID_i^{j-1}. למשל:

$$\left.\begin{array}{rcccc|c|ccc} ID_i^{j-1} & = & 1 & 0 & 1 & 1 & 0 & 0 & 1 \\ ID_{i-1}^{j-1} & = & 1 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}\right\} \Longrightarrow ID_i^j = \underbrace{011}_{3}1$$

אם בהתחלה, הגודל של ID_i^0 הוא $\log n$ ביטים, בסיבוב השני הגודל יהיה $1+\log\log n$. בסיבוב שאחריו, $1+\log\log\log n$ ביטים. זה ימשיך ככה, עד שנגיע לאורך קבוע – 3 ביטים: b_0, b_1, b_2. הם יהפכו ל־6 צבעים שונים (כי 11 לא יכול להופיע בשני הביטים הראשונים). לכן, מספר האיטרציות הוא $\log^\star n$.

למה קיבלנו צביעה שהיא חוקית? הביט האחרון של כל שני מעבדים שכנים הוא בהכרח שונה, ולכן זוהי צביעה חוקית. זה גם מוכיח שה־ID־ים של כל שני צמתים שכנים יהיו שונים בכל שלב באלגוריתם.

זה האלגוריתם של Richard Cale ו־Uzi Vishkin.

בהינתן צביעה של 6 צבעים, אפשר להפוך אותה ל־Maximal Independent Set: כל מי שהצבע שלו הוא 6, מודיע לשכניו שהוא בפנים. בסיבוב הבא, כל מי שהצבע שלו הוא 5, ואין לו שכן שהודיע שהוא בפנים, מודיע לשכניו שהוא בפנים, וכך הלאה. כל מי ששומע מאחד השכנים שלו שהשכן בפנים, יודע שהוא בחוץ. התהליך דורש 4־5 סיבובים.

טענה 17: אי אפשר לצבוע מעגל במספר קבוע של צבעים (למשל 6) בפחות מ־$\log^\star n$ סיבובים. כלומר, הרדיוס של בעיית הצביעה הוא $\log^\star n$.

הוכחה: נניח שקיים אלגוריתם שצובע ב־k סיבובים. אזי קיימת פונקציה f שמקבלת $id_x, id_{x+1}, \ldots, id_{x+k}$ ומחזירה צבע, למשל ב־$\{1,2,3,4,5,6\}$.

נגדיר גרף חדש – $G_k = (V_{G_k}, E_{G_k})$. קבוצת הקודקודים V_{G_k} היא סדרות על id באורך k מונוטוניות עולות ממש. הקשתות מוגדרות באופן הבא: $(v,w) \in E_{G_k}$ אם ורק אם $v = (id_x, id_{x+1}, \ldots, id_{x+k})$ ו־$w = (id_{x+1}, id_{x+2}, \ldots, id_{x+k}, y)$, כאשר $y > id_{x+k}$.

לא ייתכן שאם קיימת קשת בין v ל־w אזי v ו־w יקבלו את אותו הצבע על ידי f, כי אפשר לבנות מעגל מהצמתים הללו, ואז f תחזיר צבעים שונים לפי ההנחה. לכן, f היא צביעה חוקית של G_k.

הטענה שלנו היא שמספר הצביעה מקיים:

$$\chi\left(G_k^n\right) \geq \underbrace{\log\log\cdots\log n}_{k \text{ times}}$$

כאשר ה־n ב־G_k^n מייצג את ה־id־ים של הצמתים – $\{1,2,\ldots,n\}$. אם נראה את זה, נסיק שאחרי איטרציה אחת, יהיו $\log n$ צבעים. כדי להגיע למספר קבוע של צבעים, נצטרך ש־$k = \log^\star n$, כאשר $\log^\star n$ הוא מספר הפעמים שצריך לשרשר את $\log$ כדי שהפונקציה שהתקבלה תיתן מספר שהוא לכל היותר 1 עבור n. כלומר:

$$\underbrace{\log\log\cdots\log n}_{\log^\star n \text{ times}} \leq 1$$

נוכיח את זה באינדוקציה:

נתחיל עם G_0 – גרף מלא על n קודקודים. $\chi(G_0) = n$.

נניח על G_k. נוכיח על G_{k+1}. נראה ש־G_{k+1} הוא ה־Line Graph של G_k. ב־Line Graph, הקודקודים הם הקשתות של הגרף המקורי. הקשתות ב־Line Graph מחברות בין שתי קשתות סמוכות (הגרף מכוון).

למה G_{k+1} הוא ה־Line Graph של G_k? כל קודקוד ב־G_{k+1} הוא השוואה בין השמות של שני קודקודים סמוכים, ועוד ביט אחד.
נתונה צביעה של G_{k+1}. ניתן צביעה ל־G_k. הצבע של $v \in V$ יהיה קבוצת הצבעים של הקשתות הנכנסות אליו (לפי הצביעה ב־G_{k+1}). הקבוצה אינה מסודרת.
האם זו צביעה חוקית? כן, כי צבע הקשת הנכנסת לא מופיע בצבע הקודקוד ממנו הקשת יצאה, כי הצביעה של G_{k+1} חוקית. לכן, $\chi(G_k) \leq 2^{\chi(G_{k+1})}$. נוציא log:

$$\chi(G_k) \geq \log \chi(G_{k-1}) \geq \log\log \chi(G_{k-2}) \geq \cdots \geq \log^\star \chi(G_0) = \log^\star n$$

■

זאת ההוכחה של Nati Linial.

הערה 18: טענה 17 מדברת על אלגוריתמים דטרמיניסטיים בלבד.

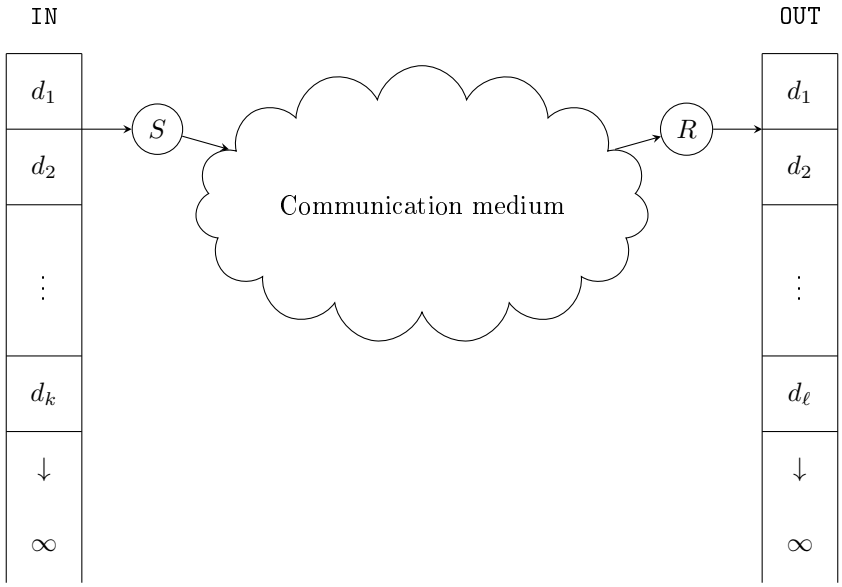
בהינתן Maximal Independent Set של מעגל, אפשר להגיע לצביעה של 3 צבעים: מי שנמצא ב־Maximal Independent Set ייצבע ב־0, ובין כל זוג צמתים ב־Maximal In-dependent Set יש לכל היותר 2 צמתים (אחרת ה־Independent Set לא היה מקסימלי). אותם צובעים ב־1 ו־2 לפי הסדר.

5.3 Sequence Transmission Problem (STP)

נתון לנו המודל כפי שמופיע באיור 5.1. יש לנו שני מעבדים: מעבד שולח S ומעבד מקבל R. לכל אחד מהם יש מערך. S קורא ערכים מתוך `IN` ושולח אותם ל־R דרך התווך התקשורתי. R כותב את מה שהוא מקבל ל־`OUT`.
הדרישות שלנו:

1. `OUT` יהיה רישא (Prefix) של `IN` בכל נקודה בזמן.
2. תחת תנאי הוגנות על התווך התקשורתי, תוך זמן סופי (בכל נקודה בזמן), יתווסף עוד איבר ל־`OUT`.

אם התווך הוא FIFO, No-Less ו־No-Duplication אזי הפתרון טריוויאלי. אם התווך הוא FIFO, Lossy ו־No-Duplication, אזי הפתרון יותר קשה. אם מוסיפים גם Duplication, הפתרון עוד יותר קשה. הפתרון קשה עוד יותר אם לא מובטח לנו FIFO.
נוסיף עוד דרישה: ה־Head של כל הודעה צריך להיות בגודל קבוע.



איור 5.1: המודל ב־STP

פרק 6

שיעור 6[1]

6.1 בעיית ההסכמה – Consensus

תיאור הבעיה: כל מעבד i מקבל ביט $x_i \in \{0,1\}$ כקלט, ומחזיר ביט $y_i \in \{0,1\}$ כפלט. כל המעבדים צריכים להחליט על ביט הפלט, תחת התנאים הבאים:

הסכמה (Safety) כל המעבדים מסכימים על אותו הפלט. כלומר, בסיום האלגוריתם, $\forall i,j.y_i = y_j$.

תקינות (Validity) ערך ההסכמה היה קלט של אחד המעבדים, כלומר $\exists j.y_i = x_j$.

חוסר־המתנה (Wait-Free) הריצה מסתיימת בזמן סופי.

כל y_i מאותחל להיות $\perp$ (don't care). בסוף האלגוריתם, אסור ש־y_i יהיה $\perp$.

פתרון טריויאלי: בוחרים מנהיג, וכולם מסכימים על מה שהוא אומר. כלומר, $y_i = x_j$, כאשר j הוא המנהיג.

מה שמעניין אותנו הוא פתרון הבעיה תוך התמודדות עם נפילות. קיימים הרבה מודלים לנפילות. אנחנו נדבר על נפילות של מעבדים במודלים הבאים:

1. Fail-Stop faults.

   כל מעבד עובד בצורה תקינה, עד שפתאום "נגמרת לו הסוללה", והוא שובת לנצח.

2. Byzantine faults.

   המעבד עובד בצורה תקינה, עד שברגע מסויים הוא מפסיק לעבוד, משתגע ועושה מה שהוא רוצה. יכול להיות גם מצב של קואליציה של מעבדים שמשתפים פעולה כדי להרוס את ריצת האלגוריתם.

   בנפילות כאלה, רוצים ש־$\forall i,j.y_i = y_j$ יתקיים בכל המעבדים התקינים.

כל עוד לא נאמר אחרת, מעכשיו כל הרשתות הן קליקות, כלומר גרף מלא, שבו יש קשת בין כל זוג מעבדים.

[1] סיכום לשיעור שהתקיים בתאריך 21.04.2013.

מודלי	Fail-Stop	Byzantine
סינכרוני	$f+1$ סיבובים	$f < \frac{n}{3}$
א־סינכרוני	אי אפשר	אי אפשר

טבלה 6.1: פתרונות לבעיית ההסכמה תחת נפילות מסוגים שונים של f מעבדים ברשת עם n מעבדים

אלגוריתם 1.6 דוגמה לריצת המעבד P_i באלגוריתם הכרעה סינכרוני שמאפשר Fail-Stop של לכל היותר f מעבדים

```
1: V_i ← {x_i}
2: for round r = 1, 2, ..., f + 1 do
3:     Send new items in V_i to all neighbors
4:     Receive S ← {new messages}
5:     V_i ← V_i ∪ S
6: end for
7: y_i ← max V_i                          ▷ could also be y_i ← min V_i
```

בטבלה 6.1, f הוא מספר המעבדים שנופלים ו־n הוא מספר המעבדים הכולל ברשת. הטבלה מתארת פתרונות לבעיית ההסכמה בכל אחד מתנאי הנפילות. הטבלה מדברת על מודל של העברת הודעות (Message Passing).

את ההוכחה שאין פתרון ל־Fail-Stop ברשת א־סינכרונית כתבו Fischer, Lynch ו־Petterson (FLP). לא נראה אותה, כי היא מסובכת. בגדול, היא עוברת למודל של זיכרון משותף.

6.1.1 התמודדות עם Fail-Stop Fault

נראה אלגוריתם הסכמה סינכרוני שעובד עם Fail-Stop, כאשר לכל היותר f מעבדים נופלים. האלגוריתם פועל $f+1$ סיבובים, ובכל סיבוב שולח את כל הידוע לו לכל שכניו. בסיבוב ה־$f+1$, מחליטים על הערך המקסימלי שנצפה. דוגמה לריצת המעבד P_i מופיעה באלגוריתם 1.6.

טענה 19: אחרי $f+1$ מחזורים, לכל i ו־j, $V_i = V_j$.

הוכחה: נניח שלא. אזי קיים $v \in V_i$ כך ש־$v \notin V_j$. אזי קיים P_{f+1} שנפל בסיבוב ה־$f+1$, ששלח את v ל־P_i, אבל לא הספיק לשלוח את v ל־P_j.

אם אף אחד לא נפל בסיבוב ה־$f+1$, אזי מכיוון שהגרף מלא, כולם מכירים בסוף הסיבוב את כל הקלטים. לכן, P_{f+1} קיים. כמו כן, קיים מעבד P_f שנפל בסיבוב ה־f, משיקול דומה. אם קיים סיבוב שבו אף אחד לא נפל, לכל שני מעבדים P_i ו־P_j, $V_i = V_j$ בסוף הסיבוב, כלומר כל המעבדים מכירים את כל הקלטים.

נמשיך כך: קיים P_{f-1} שנפל בסיבוב $f-1$, קיים P_k שנפל בסיבוב k, וכן קיים P_1 שנפל בסיבוב 1. לכן, $P_1, P_2, \ldots, P_{f+1}$ הם מעבדים שונים שנפלו במהלך הריצה. הם $f+1$ מעבדים שונים, בסתירה! ■

איך יכול להיות שלא כולם קיבלו הודעה ממעבד שנפל, אלא רק חלק מהמעבדים קיבלו אותה? יש שם פעולה של לשלוח הודעה לכולם, אבל בעצם זה לא מבוצע בבת־אחת, אלא לולאה ששולחת לכל אחד מהשכנים. יכול להיות שהמעבד נפל באמצע הלולאה.

מסקנה 20: אחרי $f+1$ מחזורים, לכל שני מעבדים P_i ו־P_j שלא נפלו, $y_i = y_j$.

6.1.2 התמודדות עם Byzantine Fault

נפתור עכשיו את הבעיה במודל סינכרוני עם Byzantine faults. את מספר המעבדים הכולל מסמנים ב־n, ואת מספר הרמאים מסמנים ב־t.

למשל, עבור $n = 3$ ו־$t = 1$: נניח שקיים אלגוריתם הסכמה שיודע להתמודד עם Byzantine faults. לכל מעבד יש שתי אפשויות לביט הקלט שלו. נחבר במשושה אותם כפי שמתואר באיור 6.1. במשושה, אף אחד מהמעבדים אינו רמאי.

כאשר A ו־B מקבלים שניהם את הקלטים $x_A = 0$ ו־$x_B = 0$, במשושה ובאיור 6.2 הם אמורים לשלוח אחד לשני בדיוק את אותן ההודעות. נניח ש־C רמאי ברשת הזאת (של המשולש), והוא שולח הודעה ל־A כאילו ש־$x_C = 0$, ושולח הודעה ל־B כאילו ש־$x_C = 1$. מההנחה, האלגוריתם יודע להתמודד עם המצב הזה, ולכן $y_A = 0 = y_B$, וכך יהיה גם במשושה.

כאשר A ו־C מקבלים שניהם את הקלטים $x_A = 1$ ו־$x_C = 1$, במשושה ובאיור 6.3 הם אמורים לשלוח אחד לשני בדיוק את אותן ההודעות. נניח ש־B רמאי ברשת הזאת (של המשולש), והוא שולח ל־A הודעה כאילו ש־$x_B = 1$, ושולח הודעה ל־C כאילו ש־$x_B = 1$. מההנחה, האלגוריתם יודע להתמודד עם המצב הזה, ולכן $y_A = 1 = y_C$, וכך יהיה גם במשושה.

נסתכל על חלק אחר מהמשושה, כאשר B מקבל את הקלט $x_B = 0$ ו־C מקבל את הקלט $x_C = 1$, כפי שמתואר ברשת שבאיור 6.4. אזי הם אמורים לשלוח אחד לשני בדיוק את אותן ההודעות כפי שמתואר במשושה. אם A רמאי ברשת הזאת (של המשולש), והוא שולח ל־B הודעה כאילו $x_A = 0$, אזי B יחליט $y_B = 0$ (כפי שראינו באיור 6.2). כמו כן, אם A שולח ל־C הודעה כאילו $x_A = 1$, אזי C יחליט $y_C = 1$ (כפי שראינו באיור 6.3). זאת סתירה להסכמה של האלגוריתם.

זאת ההוכחה של Fischer, Lynch ו־Merrit לכך שאלגוריתם כזה לא קיים.

מה עם המקרה הכללי? נגיד ש־$n = 3 \cdot t$. עושים את אותו הדבר בכפולות של 3. במקום על קודקודים, נסתכל על צבירים של t קודקודים, כך שגרף שלם של n קודקודים מורכב מחיבור של 3 צבירים כאלה. ניתן לראות הדגמה לכך באיור 6.5.

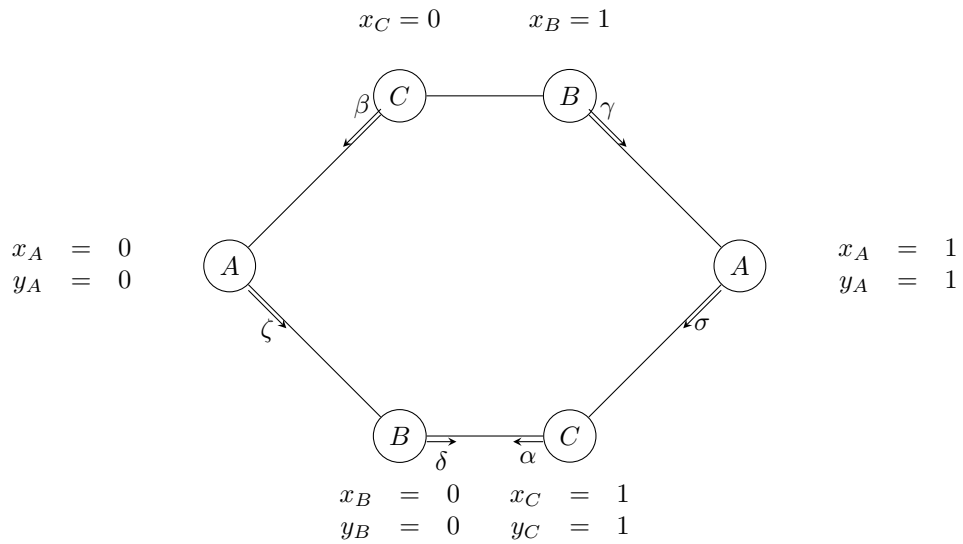
למעשה, הוכחנו שאי אפשר לפתור את הבעיה אם יש שליש או יותר רמאיים. עכשיו צריך לראות אלגוריתם שפותר את הבעיה אם פחות משליש מהמעבדים רמאיים. אלגוריתם כזה הוא האלגוריתם של Berman ו־Garay. האלגוריתם מתואר באלגוריתם 2.6. האלגוריתם מניח שלמעבדים יש שמות ב־$\{1, 2, \ldots, n\}$, וכן ש־$n > 3 \cdot t$.

משפט 21: אלגוריתם *2.6* הוא אלגוריתם הסכמה ב־$3 \cdot (t+1)$ סיבובים.

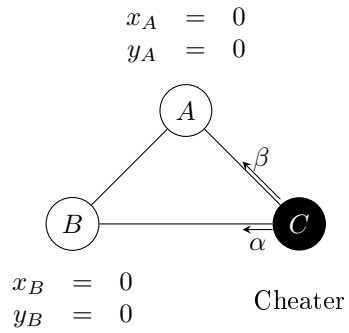
הוכחה: נשים לב לעובדות הבאות:

- בסוף סבב ההחלפות הראשון (Exchange #1), קיים $v \in \{0, 1\}$ כך שלכל המעבדים התקינים, הערך של V הוא v או 2.

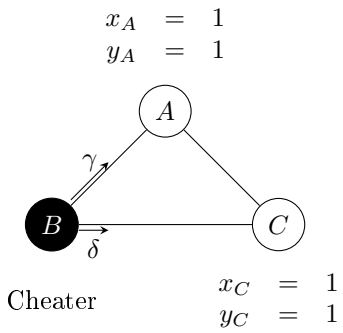
  נניח בשלילה שבמעבד x, $C_x(0) \geq n - t$, ובמעבד y, $C_y(1) \geq n - t$. נשים לב שכל מעבד תקין שולח את אותה ההודעה לכל המעבדים האחרים, ויש פחות מ־$n/3$ מעבדים שקרנים. לכן, יש פחות מ־$n/3$ הודעות שמתקבלות בצורה שונה אצל x ו־y.



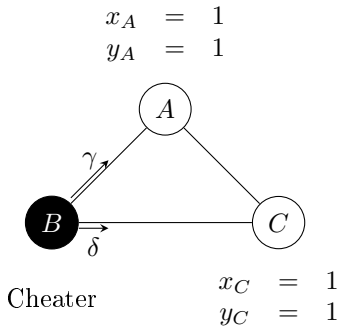
איור 6.1: רשת מעגלית עם 6 מעבדים. כל מעבד חושב שהוא נמצא ברשת עם 3 מעבדים בלבד (כאילו הייתה גרף מלא של 3 מעבדים), ומתנהג בהתאם. ברשת הזאת אין מעבדים רמאים, אבל אנחנו מקליטים את ההודעות שכל אחד מהמעבדים שולח.



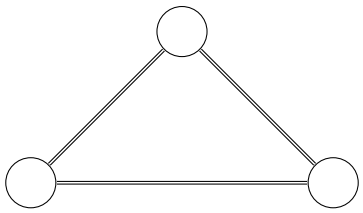
איור 6.2: רשת עם 3 מעבדים, כאשר C הוא רמאי, ומדמה שליחה של הודעות ממצבי קלט שונים ל־A ו־B.



איור 6.3: רשת עם 3 מעבדים, כאשר B הוא רמאי, ומדמה שליחה של הודעות ממצבי קלט שונים ל־A ו־C.



איור 6.4: רשת עם 3 מעבדים, כאשר A הוא רמאי, ומדמה שליחה של הודעות ממצבי קלט שונים ל־B ו־C. נראה שעכשיו B ו־C כבר לא מסכימים על אותו ביט הפלט.



איור 6.5: כל צומת בגרף הוא צביר של t קודקודים המחוברים ביניהם. כל קודקוד בצביר מחובר לשני הקודקודים המתאימים לו בצבירים האחרים.

אלגוריתם 2.6 האלגוריתם של Berman ו־Garay להסכמה עם Byzantine faults

```
1:  V ← x_i
2:  for m = 1, 2, ..., t + 1 do
                                                    ▷ Exchange #1
3:      send V
4:      V ← 2
5:      for k = 0, 1 do
6:          C(k) ← the number of received k's
7:          if C(k) ≥ n − t then V ← k
8:      end for
                                                    ▷ Exchange #2
9:      send V
10:     for k = 2, 1, 0 do
11:         D(k) ← the number of received k's
12:         if D(k) > t then V ← k
13:     end for
                                                    ▷ Exchange #3
14:     if m = i then send V
15:     if V = 2 or D(V) < n − t then V ← min{1, received message}
16: end for
17: return V
```

לכן, קיימים יותר מ־$\frac{2\cdot n}{3}$ מעבדים ששלחו את אותו הערך ל־x ול־y. נשים לב: $n-t > n-\frac{n}{3} = \frac{2\cdot n}{3}$. מכיוון ש־$C_x(0) \geq n-t$, $C_x(1) < n/3$. מצד שני, $C_y(1) \geq n-t > \frac{2\cdot n}{3}$. לכן, קיימים יותר מ־$n/3$ מעבדים ששלחו הודעה שונה ל־x ול־y, בסתירה.

- באופן דומה, בסוף סבב ההחלפות השני (Exchange #2), עם אותו ה־v מקודם, הערך של V אצל כל המעבדים התקינים הוא v או 2.
- אם בתחילת הסבב הערך של V הוא $v \in \{0,1\}$ בכל המעבדים התקינים, אז הוא לא משתנה עד לסוף הסבב.

  במצב הזה, כל המעבדים התקינים יקבלו לפחות $n-t$ הודעות עם הערך v, ואז הערך של V תמיד יהיה v.

יהי g המזהה הקטן ביותר של מעבד תקין. בסוף הסבב השלישי (Exchange #3) של האיטרציה ה־g בלולאה, מושגת הסכמה, כי אחד משני המצבים הבאים מתקיים:

1. כל המעבדים התקינים מקבלים את הערך של ה"מלך" של האיטרציה (שהוא g).
2. חלק מהמעבדים התקינים מתעלמים מההודעה של g כי אצלם $D(V) \geq n-t$. אבל אז $D(V) > t$ אצל כל המעבדים התקינים, ולכן הערכים שלהם של V זהים בסוף הסבב השלישי, בין אם הם מקבלים את הערך של ה"מלך" ובין אם לאו.

מכיוון ש־$g \leq t+1$, נקבל שכל המעבדים התקינים ידעו את ערך ההסכמה בסוף האיטרציה ה־$t+1$. בכל איטרציה יש 3 סיבובים, ולכן תוך $3\cdot(t+1)$ סיבובים, ערך ההסכמה יוסכם בין כל המעבדים התקינים. ■

6.2 מודל זיכרון משותף (Shared Memory)

במודל יש n מעבדים. לכל מעבד יש זיכרון פרטי מקומי שלו, ויש גם זיכרון המשותף לכולם. בפעולה אטומית, אפשר לקרוא (Load) או לכתוב (Store) לזיכרון המשותף.

הזיכרון מחולק לתאים. נקרא לתאים האלה אוגרים, או Registers באנגלית. בגישה היא לתאים, ולא לזיכרון השלם.

נרצה להראות קשר בין מודל העברת ההודעות (א־סינכרונית) למודל הזיכרון המשותף. נרצה להראות שכל בעיה הניתנת לפתרון באמצעות העברת הודעות א־סינכרונית, ניתנת לפתרון באמצעות זיכרון משותף, ולהיפך.

ניתן להפעיל אלגוריתם העברת הודעות במודל זיכרון משותף אם לכל זוג מעבדים המחוברים בהעברת הודעות יש שני תורים בזיכרון המשותף, כאשר כל מעבד קורא מתור אחד וכותב לתור אחר[2]. מעבד כלשהו קורא כל הזמן ובודק אם יש הודעות נכנסות בכל אחד מהתורים הנכנסים אליו (לפי הסדר).

מסקנה 22: אם יש אלגוריתם שפותר את בעיית ההסכמה במודל העברת הודעות, יש גם אלגוריתם שפותר אותה בזיכרון משותף. לכן, אם אין אלגוריתם שפותר את בעיית ההסכמה בזיכרון משותף, אזי אין אלגוריתם שפותר אותה בהעברת הודעות.

[2] זה מזכיר קצת Pipe־ים ב־Linux.

פרק 7

שיעור 7[1]

7.1 בטחון באוגרים בזיכרון משותף

בשונה מבחישוב סדרתי, כאשר החישוב מבוזר, אי אפשר להסתכל על פעולה כנקודה בזמן, אלא כקטע בזמן, עם התחלה וסוף. כך למשל, כתיבה לאוגר מתחילה בשלב כלשהו ומסתיימת בשלב כלשהו. יכול להיות שבאמצע מגיע תהליך אחר שרוצה לקרוא את ערך האוגר. מה הוא יקבל?

זה תלוי בסוג האוגר. לרוב, ההבדל הוא במימוש בחומרה, אבל מבחינה לוגית־תיאורטית אפשר להגדיר כמה אוגרים שונים, לפי רמות הבטחון שהם מספקים. את האוגרים ניתן לחלק ל־3 רמות ביטחון:

- Safe.

  בין כתיבות לאוגר, הערך של האוגר הוא הערך שנכתב בכתיבה האחרונה. במהלך פעולת (או פעולות) כתיבה, הערך שייקרא מהאוגר הוא ערך כלשהו. כלומר, גם אם כותבים 0 לאוגר שהערך הקודם שלו היה 0, יכול להיות שנקרא ממנו 1.

- Regular.

  כמו Safe, אבל בזמן כתיבה לאוגר, קריאה מהאוגר תחזיר את הערך הישן של האוגר או את הערך החדש שלו. למשל, אם הערך באוגר הוא 0, ומתחילים לכתוב אליו 1, עד שפעולת הכתיבה לא תסתיים, הקריאות יכולות להחזיר 0 או 1.

  יכול להיות שתהיה פעולת קריאה שהחזירה 0, והתחילה מפש אחרי שפעולת קריאה אחרת הסתיימה והחזירה 1. התכונה הזאת נקראת *Linearizable*, ואוגר Regular לא מקיים אותה.

- Atomic.

  כמו Regular, אבל גם Linearizable.

כמו כן, אפשר להסתכל על האוגרים לפי כמות הכותבים והקוראים מהם. הסוגים האפשריים הם Single-Reader Single-Writer (SRSW), Multiple-Reader Single-Wrter (MRSW) ו־Multiple-Reader Multiple-Writer (MRMW). בנוסף, כל אוגר יכול להיות בינארי (שהערכים בו הם 0 או 1), או רב־ערכי (Multi-Value).

[1] סיכום לשיעור שהתקיים בתאריך 28.04.2013.

אלגוריתם 1.7 SnapScan למציאת Snapshot עדכני ביותר

```
1: function Collect
2:     return all snapshots from all registers as a local memory array
3: end function

4: function SnapScan
5:     a ← Collect()
6:     b ← Collect()
7:     if a ≠ b then
8:         existsk.a[k] ≠ b[k]
9:         return the snapshot at b[k]
10:    else
11:        return any snapshot in b
12:    end if
13: end function
```

בסופו של דבר, אפשר לבנות אוגר Atomic Multi-Values Multiple-Reader Multiple-Writer מאוסף של הרבה אוגרים מסוג Safe Binary Single-Reader Single-Writer. כלומר, מבחינה תיאורטית, שני האוגרים שקולים ביכולות החישוביות שלהם, כי אפשר לבנות אחד מהשני. הבניה מאוד יפה, אבל לא נראה אותה פה.

7.2 Snapshot

איך עושים Snapshot בעזרת אוגרי Atomic Multi-Valued MRMW?

נניח שאנחנו כבר יודעים לעשות Snapshot. אז כל אוגר יכיל מקום לערך שלו ול־Snapshot שלם, וכל פעם שכותבים אליו, כותבים בנוסף לערך גם את ה־Snapshot.

בכל פעם שאנחנו נרצה Snapshot, אפשר לקחת Snapshot מתוך אוגר כלשהו, אבל זה יהיה Snapshot ישן. איך מוצאים Snapshot חדש? ניתן למצוא אלגוריתם שעושה את זה באלגוריתם 1.7.

הבעיה עם האלגוריתם הזה היא שהוא תמיד יחזיר לנו Snapshot ישן, אלא אם שום דבר לא השתנה בין שני ה־Collect־ים. תיקון לבעיה ניתן למצוא באלגוריתם 2.7.

אם תפסנו את k זז בפעם השניה, אזי הוא הספיק לקחת Snapshot לפני ה־Collect השני. לכן, זה Snapshot חדש. נשים לב שיש לנו לכל היותר n לולאות, כאשר n הוא מספר האוגרים ב־Snapshot.

אלגוריתם 2.7 תיקון למציאת Snapshot עדכני ביותר

```
1:  function SnapScan
2:      Moved ← ∅
3:      loop
4:          a ← Collect()
5:          b ← Collect()
6:          if a = b then return any snapshot in b
7:          else
8:              ∃k.a[k] ≠ b[k]
9:              if k ∈ Moved then return the snapshot in b[k]
10:             else Moved ← Moved ∪ {k}
11:         end if
12:     end loop
13: end function
```

פרק 8

שיעור 9[1]

8.1 בעיית ההסכמה

נתונים לנו שני מעבדים, היכולים לקרוא ולכתוב. אנחנו מנסים לפתור את בעיית ההסכמה – Consensus. הפתרון צריך לקיים את ה־Safety וה־Validity שראינו קודם.
בנוסף, נדרוש שהפתרון יהיה Wait-Free. כל מעבד יכול להיות Fail-Stop, ויש לנו Atomic Registers.

הגדרה 23 (מצב דו־ערכי ומצב חד־ערכי): נתבונן בעץ ההחלטות (והפעולות) באלגוריתם ההסכמה. מכיוון שהפעולות הן Linearizable, אפשר להסתכל על נקודה בזמן שבה כל פעולה קורית, וסדר ביצוע הפעולות הוא המשפיע על ההסכמה. לכן, יש לנו עץ, כאשר כל קשת היא פעולה של מעבד כלשהו מתוך המצב הנוכחי.
מצב דו־ערכי הוא מצב שממנו ניתן להגיע לשתי החלטות שונות. מצב חד־ערכי הוא מצב שבלי תלות בריצה ובתזמונים של פעולות המעבדים, תמיד יוסכם על אותו הערך.

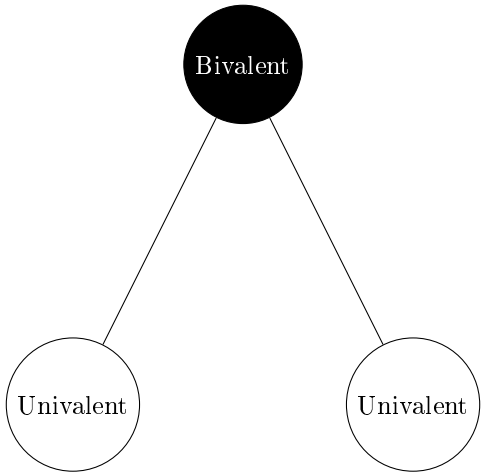
נניח שיש מצב התחלתי דו־ערכי. אזי קיים עץ, שנבנה מההתנהגות של כל אחד מהמעבדים וסדר ביצוע הפעולות של כל אחד מהמעבדים (אין אפשרות שבה שני המעבדים מבצעים פעולה בו־זמנית, כי האוגרים הם Linearizable). במילים אחרות, עץ הריצות נקבע לפי האלגוריתם והקלט. הצעדים בעץ (ההתקדמות בפועל) נקבעת לפי הריצה.
במודל שלנו, אם מעבד רץ לבד (כי המעבד השני נפל, או פשוט לא תקשר עם העולם עדיין), והקלט שלו הוא 1 (למשל), אזי הוא חייב להגיע להסכמה על 1 תוך זמן סופי. מכיוון שהאלגוריתם הוא Wait-Free, המעבד יהיה חייב לקבל את הקלט שלו בתור הפלט.

טענה 24: קיים מצב התחלתי דו־ערכי.

הוכחה: מכיוון שהאלגוריתם הוא Wait-Free, אם ניתן לכל אחד מהמעבדים לרוץ לבד, עוד לפני שהשני מתעורר, ולכל אחד מהמעבדים קלט שונה, תמיד הפלט יהיה הקלט של המעבד שרץ. ■

טענה 25: קיים מצב דו־ערכי, ששני המצבים היוצאים ממנו (הילדים שלו בעץ) הם חד־ערכיים, עם ערכים שונים.

[1] סיכום לשיעור שהתקיים בתאריך 05.05.2013.



איור 8.1: מצב קריטי

הוכחה: העלים (בהם התקבלה ההחלטה) הם חד־ערכיים. מכיוון שהמצב ההתחלתי דו־ערכי, נקבל את התוצאה מכך שהריצה מסתיימת בזמן סופי, כלומר בסדרת פעולות סופית. ■

הגדרה 26 (מצב קריטי): בעץ ריצות של אלגוריתם הסכמה, מצב קריטי הוא מצב דו־ערכי ששני בניו הם חד־ערכיים.

ניתן למעבדים שלנו את השמות P ו־Q. נניח כי c הוא מצב קריטי. נניח שהצעד הבא של כל אחד מהצדדים הוא צעד פנימי־מקומי. מחד, אם P יבצע את הצעד הראשון, וירוץ לבד, האלגוריתם יחליט על הפלט α. מאידך, אם Q יבצע את הצעד המקומי הראשון, ואז P ירוץ לבד, הוא לא ידע ש־Q ביצע את הצעד, כי הוא מקומי ב־Q, ולכן יחליט על α, בסתירה!

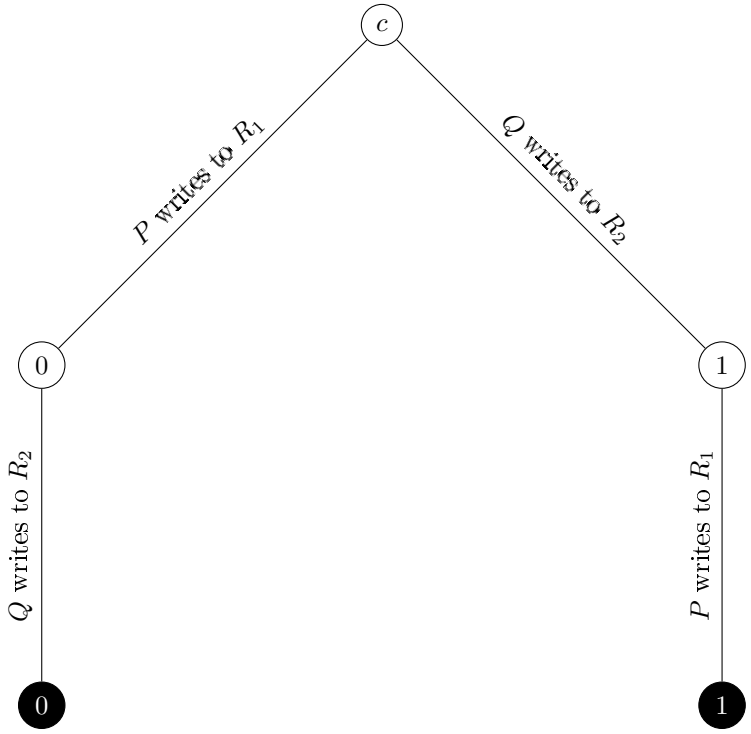
גם אם אחד הצעדים הוא קריאה מהזיכרון המשותף (בה"כ של Q), קיבלנו מצב דומה: אם P ירוץ לבד, הוא יחליט על α. אם Q יקרא, ואז P ירוץ לבד, P לא ידע ש־Q קרא, ולכן יחליט בכל זאת על α, בסתירה לכך ש־Q הוא מצב קריטי.

אם הצעדים הבאים של P ו־Q הם כתיבות לאוגרים שונים, אז גם נקבל סתירה, כי נשים לב שאין דרך להבדיל בין המצבים הבאים:

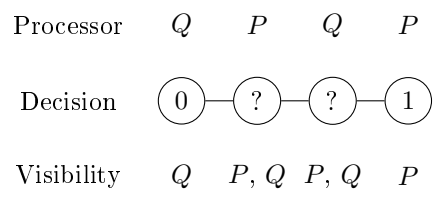
1. הפעולה הראשונה היא ש־P כתב ל־R_1, והפעולה הבאה היא ש־Q כתב ל־R_2.
2. הפעולה הראשונה היא ש־Q כתב ל־R_2, והפעולה הבאה היא ש־P כתב ל־R_1.

לא ניתן להבחין בין שני המצבים האלו (המצבים השחורים באיור 8.2), מכיוון שאי אפשר לדעת איזה מעבד ביצע את הכתיבה שלו קודם, בסתירה לכך ש־c הוא מצב קריטי.

לכן, שני המעבדים בהכרח כותבים לאותו אוגר. אם P ירוץ ראשון ולבד, הוא יחליט על α. אם Q רק יכתוב לאוגר, ואז P ירוץ לבד, אזי P לא יראה את הכתיבה של Q, ולכן יחליט על α, בסתירה לכך ש־c הוא מצב קריטי.



איור 8.2: לא ניתן להבחין בין המצבים השחורים, בסתירה לכך ש־c הוא מצב קריטי



איור 8.3: המצבים האפשריים. קווים מחברים בין זוגות מצבים שאפשר שיתקיימו באותה הריצה, ולכן ערך ההסכמה בהם צריך להיות זהה. ערכי ההסכמה כתובים במקום שהם קבועים ללא קשר לריצה.

זה מוכיח שבמודל זיכרון משותף א־סינכרוני אי אפשר לממש אלגוריתם הסכמה בעזרת אוגרים אטומיים בצורה שהיא Wait-Free. אפשר לומר "עם נפילות" במקום "א־סינכרוני".

הערה 27: הדרישה ל־Fail-Stop כאן לא הכרחית.

נגדיר מה זה Wait-Free עם n מעבדים במערכת שתומכת ב־$n-1$ נפילות. למה תומכים ב־$n-1$ נפילות? כי אם מעבד יודע שלכל היותר מעבד אחד נופל מתוך 100, אזי אפשר לחכות ש־99 מעבדים יכתבו, כי זה יקרה בוודאות בזמן סופי. במודל שלנו, למעבד אסור להניח שאף אחד יכתוב שום דבר בזמן סופי.

בהמשך נראה שאם יש $n = 2$ מעבדים, ו־1 נופל, אי אפשר לבצע החלטה (בדרך אחרת ממה שראינו קודם). אחר כך נראה שאם יש n מעבדים, ו־1 נופל, אי אפשר לפתור את בעיית ההסכמה.

טענה 28: נניח שיש 2 מעבדים ו־1 יכול ליפול. אזי אי אפשר להחליט על הסכמה.

הוכחה: לא ייתכן שמחליטים אחרי 0 צעדים, כי אז כל מעבד יחזיר את הקלט שלו, והפלטים עשויים להיות שונים.

נניח שהצעדים של המעבדים הם הכי מסובכים שאפשר לחשוב עליהם – Immediate Snapshot. אפשר לכתוב ולקרוא את כל מה שכתבו ביחד איתו. מבחינה חישובית, זה מודל יותר חזק מאוגר אטומי. מעכשיו הצעדים של P הם מהצורה $W_P R_P$, כי הם כוללים קריאה וכתיבה אטומית בזיכרון המשותף (ללא עיבוד באמצע).

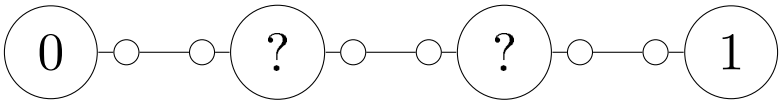
נניח שהמודל הוא Single-Writer Multiple-Reader. זה גם לא מגביל אותנו, כי ראינו בשיעור הקודם שזה שקול ל־Multiple-Writer Multiple-Reader.

אז אי אפשר לעשות את זה ב־0 צעדים. מה עם צעד 1? יכול להיות ש־Q עשה $W_Q R_Q$ ולא ראה את P. יכול להיות ש־P עשה $W_P R_P$ ולא ראה את Q. יכול להיות ש־P או Q (או שניהם) רואים את השני.

כפי שניתן לראות באיור 8.3, אם Q רץ לבד, הוא יחליט על הקלט של עצמו (נניח 0). באופן דומה, אם P רץ לבד, הוא יחליט על הקלט של עצמו (נניח 1). אבל מה יקרה עם P שירוץ אחרי Q? הוא יחליט 0. גם Q שרץ אחרי P יחליט 1.

אבל אם P ו־Q רצים ביחד, הם יראו בדיוק את אותו המצב כאילו שאחד היה רץ לפני השני. לכן, ההחלטות שלהם מנוגדות, בסתירה!

מה קורה אם מחליטים תוך 2 צעדים? נחלק כל קשת בגרף שלנו ל־3 קשתות ו־2 קודקודים, כפי שניתן לראות באיור 8.4.



איור 8.4: הכללה להחלטה תוך 2 צעדים של הגרף באיור 8.3.

אלגוריתם 1.8 אלגוריתם החלטה למעבד ה־0 (מתוך זוג) שאינו Wait-Free

```
1: write A[0] ← x
2: SA ← SNAPSHOT(A⃗)
3: write B[0] ← SA
4: SB ← SNAPSHOT(B⃗)

5: if SA[1] = ⊥ then decide x
6: if SB[1] = (⊥, y) then decide y
7: while B[1] = ⊥ do read B[1]
8: decide min B⃗
```

ניתן להמשיך כך באינדוקציה, ולקבל שלא ניתן לפתור את בעיית ההסכמה תוך זמן סופי. ■

איך נראה שאי אפשר לפתור הסכמה עבור n מעבדים? נניח בשלילה שקיים אלגוריתם שפותר החלטה עבור n מעבדים, כאשר 1 מהם עשוי ליפול. בהינתן הקוד (שפת המכונה, למשל) של המעבדים $P_0, P_1, \ldots, P_{n-1}$, כך שבתחילת הקוד כתוב הקלט של המעבד, נראה דרך לפתור החלטה ל־2 מעבדים בעזרת הקודים האלה.

אלגוריתם ההחלטה על המעבד ה־0 יראה כפי שמוצג באלגוריתם 1.8. האלגוריתם הזה הוא לא Wait-Free, כי יש שם לולאת while. זאת הסיבה לכך שהוא לא עומד בסתירה לטענות הקודמות. כמו כן, יכול להיות שניתקע ב־while באלגוריתם בהמתנה למעבד השני.

את הסימולציה נעשה באופן הבא: כל מחשב יבצע הסכמה על כל אחד מהקלטים של n המעבדים לאלגוריתם המקורי. הסימולציה תעבוד באופן הזה: נסמלץ 7 צעדים מכל מחשב, ואז נעבור הלאה. אם המחשב השני ייתקע, אז יכול להיות שנחכה בלולאה בהחלטה על הקלט של אחד המחשבים המסומלצים. מכיוון שהאלגוריתם הנתון הוא Wait-Free, ומאפשר נפילה של מחשב אחד, אפשר להמשיך באלגוריתם ולהגיע להסכמה. ככה קיבלנו הסכמה שהיא Wait-Free עבור 2 מעבדים, בסתירה!

8.2 פעולות אטומיות על אוגרים

נניח עכשיו שנתון לנו אוגר R המאותחל ל־0. בצעד אטומי אחד, Test&Set (או TAS) עושים את המתואר באלגוריתם 2.8.

חשוב לשים לב שזהו רק תיאור (Specification) של מה שקורה בפעולה אטומית. אי אפשר להיתקע באמצע הקוד כאן.

אלגוריתם 2.8 Test&Set

```
1: T ← R
2: R ← 1
3: return T
```

אלגוריתם 3.8 אלגוריתם הסכמה בין 2 מעבדים בעזרת אוגרי Test&Set

```
1: write Input[i] ← v
2: t ← TAS(R)
3: if t = 0 then decide v
4: else decide Input[1 − i]
```

אלגוריתם הסכמה בין 2 מעבדים בעזרת אוגרי Test&Set מוצג באלגוריתם 3.8. באלגוריתם, $i \in \{0, 1\}$ הוא המזהה של כל אחד מהמעבדים.

טענה 29: במערכת עם Test&Set, ואוגרים אטומיים לקריאה וכתיבה, אין אלגוריתם Wait-Free להסכמה בין 3 מעבדים.

הוכחה: כמו קודם, קיים מצב התחלתי דו־ערכי. כמו כן, קיים מצב קריטי c, שמביא למצבים חד־ערכיים. כמו כן, לא יכול להיות שהצעדים הם מקומיים, קריאות או כתיבות. לכן, הצעד הבא הוא Test&Set, ועל אותו האוגר (משיקול דומה למה שראינו קודם).

אם P או Q עשו TAS ראשונים (אחד מהם), ו־R ממשיך לבדו, הוא לא יכול לדעת מי היה הראשון לבצע את הפעולה. במקרה שבו P ו־Q מביאים להחלטות שונות, נקבל את הסתירה. ■

פרק 9

שיעור 10[1]

9.1 מספר Consensus

כמה הבהרות לגבי השיעור הקודם: כשהוכחנו שאין אלגוריתם א־סינכרוני Wait-Free שמבצע הסכמה בין n מעבדים, כאשר לכל היותר 1 נופל, ראינו אלגוריתם הסכמה בין שני מעבדים המכיל לולאה. אם מעבד מגיע לבד ללולאה, אזי הוא מחליט לבד. אם שני המעבדים סיימו את קטע החישוב הישר וסיבוב שלם בלולאה, אז שניהם מחליטים.

מסקנה 30: לא קיים אלגוריתם הסכמה Wait-Free במערכת זיכרון משותף עם קריאות וכתיבות אטומיות, אפילו רק עם נפילה אחת (Fail-Stop).

איך נשליך מכך על מבני נתונים אחרים? נניח שקיים מימוש אטומי של תור, המאפשר ביצוע Enqueue ו־Dequeue. נראה איך מממשים באמצעותו אלגוריתם הסכמה, ואז נסיק כי לא ניתן לממש את התור באמצעות קריאות וכתיבות אטומיות לזיכרון המשותף. האלגוריתם מתואר באלגוריתם 1.9.

מניחים שהמערך A הוא משותף בזיכרון, וכן כי התור מאותחל כך שהאיבר הראשון שהוכנס אליו הוא כחול, וכל השאר ירוקים.

למעשה, ראינו באלגוריתם שמספיק שהתור יהיה מאותחל ויתמוך ב־Dequeue (כי לא השתמשנו כאן ב־Enqueue). אין כאן שום דבר המיוחד לתורים. אפשרת לעשות את זה עם כל אובייקט שתומך בפעולה הזאת (או בפעולה דומה). למשל, מחסנית (עם Pop).

הגדרה 31 (מספר Consensus): במערכת של זיכרון משותף עם קריאות וכתיבות אטומיות, ומספר לא מוגבל של עותקים של האובייקט O, נגדיר את מספר ה־$Consensus$ של O להיות המספר הגדול ביותר של מעבדים שיכולים להסכים.

[1] סיכום לשיעור שהתקיים בתאריך 12.05.2013.

אלגוריתם 1.9 אלגוריתם הסכמה באמצעות תור אטומי

```
1: write A[i] ← x_i                          ▷ x_i is my input
2: t ← DEQUEUE(Q)
3: if t = • then decide x_i                  ▷ t is "blue"
4: else decide A[1 − i]
```

אלגוריתם 2.9 הפעולה האטומית של אובייקט SWAP

Ensure: The value of R will be swapped by v
1: **function** SWAP(v, R)
2: **read** $t \leftarrow R$
3: **write** $R \leftarrow v$
4: **return** t
5: **end function**

מספר Consensus	אובייקט
1	אוגרים לכתיב וקריאה אטומית
2	תור, מחסנית, Test&Set, SWAP, Fetch&Add ועוד
⋮	
∞	Compare&SWAP

טבלה 9.1: מספרי ה־Consensus לאובייקטים שונים

בשיעור הקודם, ראינו שמספר ה־Consensus של Test&Set הוא לפחות 2 וקטן מ־3. לכן, הוא בדיוק 3. עכשיו ראינו שמספר ה־Consensus של תור הוא לפחות 2. קיימת הוכחה שהוא בדיוק 2.

נגדיר אובייקט חדש – SWAP. הפעולה האטומית שלו מוגדרת באלגוריתם 2.9. אפשר לומר ש־$\texttt{TAS}(R) = \texttt{SWAP}(1, R)$.

אפשר להראות מימוש להסכמה עם SWAP ל־2 מעבדים, בדומה למימוש שראינו עבור Test&Set. איך נראה שאי אפשר לממש הסכמה ל־3 מעבדים עם SWAP? כמו שראינו קודם, עם המצבים הקריטיים.

נניח שאם P או R עושים פעולה ראשונים, מחליטים 0, ואם Q עושה פעולה ראשון, מחליטים 1. בהכרח כולם כותבים לאותו האוגר. אם R כותב r, ואז Q כותב q, אם P ירוץ לבד, הוא יחליט 0. אבל הוא יראה בדיוק את אותו המצב כאילו ש־Q היה כותב q מייד אחרי המצב הקריטי (בלי ש־R יכתוב כלום), ואז P היה צריך לרוץ לבד ולהחליט 1, בסתירה.

באופן כללי, אפשר לסווג אובייקטים לפי מספר ה־Consensus שלהם. ניתן לראות מספרי Consensus של חלק מהאובייקטים בטבלה 9.1.

נגדיר את האובייקט Compare&SWAP (או בקיצור CAS), לפי הפעולה האטומית שמתוארת באלגוריתם 3.9.

כדי לממש הסכמה עם Compare&SWAP, נאתחל את האובייקט שיכיל את הערך $\perp$. כל מעבד ינסה לכתוב את המזהה שלו לתוך האובייקט. היחיד שיצליח הוא המנהיג, וכל האחרים משתמשים בערך שלו. הקוד מתואר באלגוריתם 4.9.

9.2 בנייה אוניברסלית

בהינתן אובייקטי Compare&SWAP, או מימוש אלגוריתם הסכמה תקין, אפשר לבנות כל אובייקט, כך שיהיה מקבילי לפי ה־Sequential Specification שלו. המימוש לא יעיל

אלגוריתם 3.9 Compare&SWAP

Ensure: If the value of R is o, it will be changed to n. In any case, the old value of R is returned

```
1: function CAS(o, n, R)
2:     read t ← R
3:     if t = o then write R ← n
4:     return t
5: end function
```

אלגוריתם 4.9 הסכמה באמצעות Compare&SWAP

```
1: T ← CAS(⊥, x_i, R)
2: if T = ⊥ then decide x_i          ▷ I am the leader
3: else decide T                     ▷ Someone else is the leader
```

במיוחד, אבל מבחינה חישובית הוא עובד.

הרעיון הוא שבכל פעם שמעבד מנסה לבצע פעולה על האובייקט, הוא יסמלץ את הפעולה בצורה מקומית, ואז ינסה להסכים (יחד עם כל המעבדים האחרים) מה הפעולה שקורית בפועל. אם המעבד לא הצליח לגרום לפעולה שלו להיות מוסכמת על השאר, הוא יעדכן את האובייקט המקומי שלו, ואז יפעיל את הפעולה שלו מחדש וינסה לגרום להסכמה על הפעולה שלו הפעם.

אחת הבעיות שיכולה להיות באלגוריתם כללי כזה היא הרעבה (Starvation) של אחד המעבדים, שלא יצליח לגרום לפעולה שלו להיות מוסכמת על כולם אף פעם. יש אלגוריתמים שמתמודדים עם המצבים האלה, ורמת הסיבוך שלהם עולה בהתאם.

פרק 10

שיעור 11[1]

10.1 היסטוריות

הגדרה 32 (היסטוריה סדרתית חוקית): היסטוריה סדרתית H היא חוקית אם לכל אובייקט x בהיסטוריה, $H \mid x$ (הטלה של H על x) היא סדרה חוקית של פעולות.

הגדרה 33 (היסטוריה Linearizable): היסטוריה היא *Linearizable* אם ניתן להרחיבה להיסטוריה סדרתית חוקית על ידי הוספת Responses לקריאות שמחכות, או מחיקה של קריאות שמחכות.

משפט 34: היסטוריה H היא *Linearizable* אם לכל אובייקט x, $H \mid x$ היא גם *Linearizable*.

הגדרה 35 (Sequence Consistent): היסטוריה H היא *Sequence Consistent* אם אפשר למתוח או להזיז את ציר הזמן של כל מעבד (בנפרד) ולקבל היסטוריה עקבית.
אלגוריתם הוא *Deadlock-Free* אם מישהו מהמעבדים יתקדם תוך זמן סופי.

הגדרה 36 (אלגוריתם Starvation-Free): אלגוריתם הוא *Starvation-Free* אם כל אחד מהמעבדים יתקדם תוך זמן סופי.

הגדרה 37 (אלגוריתם Lock-Free): אלגוריתם הוא *Lock-Free* אם מישהו מאלו שקראו לפונקציה יחזור תוך זמן סופי.

הגדרה 38 (אלגוריתם Wait-Free): אלגוריתם הוא *Wait-Free* אם כל מי שקרה לפונקציה יחזור תוך זמן סופי.

[1] סיכום לשיעור שהתקיים בתאריך 19.05.2013.

פרק 11

שיעור 12[1]

11.1 Splitter

ה־Splitter, כפי שמתואר באיור 11.1, הוא רכיב שפועל לפי הדרישות הבאות:

1. אם תהליך מגיע לבד ל־Splitter, הוא ינצח תוך זמן סופי.
2. אם הגיעו k תהליכים, אז לכל היותר $k-1$ יקבלו L (כלומר ילכו שמאלה).

הגדרה 39 (אלגוריתם Adaptive): אלגוריתם הוא *Adaptive* אם סיבוכיות הפעולות בו היא פונקציה של מספר התהליכים שמשתתפים בו־זמנית.

נרצה לעשות Collect, כלומר לאסוף את כל הקלטים של כל התהליכים בצורה שהיא Adaptive. כלומר, שהסיבוכיות של האלגוריתם תהיה $f(k)$, כאשר k הוא מספר המשתתפים.

בצורה דומה ל־Renaming, אפשר לעשות את זה ב־$\mathcal{O}(k^2)$, באמצעות החזקה של מטריצה של Splitter־ים בגודל $k \times k$. איך אפשר לעשות את זה ב־$\mathcal{O}(k)$?

נחזיק עץ בינארי של Splitter־ים. הולכים ימינה ושמאלה לפי R ו־L, בהתאמה. המנצח ב־Splitter יכתוב את הערך שלו באוגר המקושר מהעלה בו הוא ניצח.

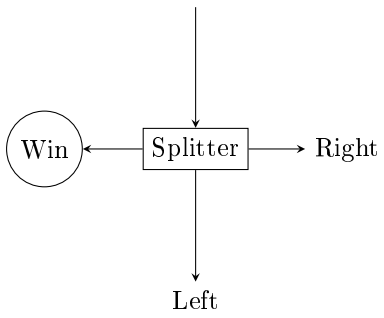
אם יש n מעבדים, אבל רק k מתוכם פעילים, גודל העץ יהיה 2^n, כי המסלול הארוך ביותר הוא בגודל n. אם באו k תהליכים, נעבור $\mathcal{O}(k)$ Splitter־ים במקרה הגרוע ביותר, כלומר k צעדים עד שננצח בוודאות.

נניח שהצמתים בעץ מסומנים ב־X, עד שכותבים עליהם, ואז מסמנים אותם ב־W. אזי יהיו k צמתים שמסומנים ב־W. איך עושים את ה־Collect? אפשר לסרוק את העץ, בדרך הכי רגילה, עם BFS. כדאי גם להוסיף לכל צומת בעץ דגל שנקרא visitted, שיאותחל כלא מסומן, ויסומן בביקור בצומת (כלומר מעבר ב־Splitter).

לכן, בסך הכל, ביצוע של Collect לוקח $\mathcal{O}(k)$. כך גם ההרשמה (Register) של הקלטים לוקחת $\mathcal{O}(k)$ זמן.

מימוש Splitter יהיו לנו 2 אוגרים: x ו־y. כל מעבד כותב את השם שלו ב־x. y מאותחל ל־0. אם $y = 1$, הולכים ימינה. אחרת, כותבים 1 ב־y, ובודקים אם השם שלנו עדיין כתוב ב־x. אם כן, ניצחנו. אחרת, הולכים שמאלה. פסאודו־קוד של המימוש ניתן למצוא באלגוריתם 1.11.

[1] סיכום לשיעור שהתקיים בתאריך 26.05.2013



איור 11.1: תרשים של Splitter

אלגוריתם 1.11 מימיש Splitter

Require: The identification of the current process is i
1: **write** $x \leftarrow i$
2: **read** $t \leftarrow y$
3: **if** $y = 1$ **then return** go right
4: **write** $y \leftarrow 1$
5: **if** $x \neq i$ **then return** go left
6: **else return** win

11.2 בעיית ה־Renaming

יש לנו מעבדים שמגיעים ממרחב שמות בגודל $|M|$. רק $n \ll |M|$ מעבדים מתעוררים, ואנחנו רוצים להתייחס אליהם בשמות יותר קטנים. הבעיה נקראת גם Weak-Renaming. נראה בהמשך אלגוריתם שבוחר שמות מתוך $2 \cdot n - 1$ שמות.

הבעיה שימושית למשל במימוש ליבות של מערכות הפעלה, שבהן יש הרבה תהליכים, אבל מעט מהם פעילים, והיינו רוצים לתת שמות קטנים יותר לתהליכים הפעילים.

הבעיה של Strong-Renaming: אם התעוררו k מעבדים, נרצה שמות מתוך $2 \cdot k - 1$. הבעיה נקראת גם Adaptive-Renaming.

האלגוריתם יעבוד באופן הבא: השמות יהיו $1, 2, \ldots, 2 \cdot n - 1$. כל מעבד בהתחלה מרים דגל שאומר שהוא בחר את השם 1, ואז מוודא שהוא היחיד שהרים את הדגל. אם הוא היחיד – מזל טוב! זה יהיה השם שלו. אחרת, הוא יוריד את הדגל, ויעבור למספר (השם) הבא.

איך הוא יבחר את המספר הבא? בהתחלה, הוא יעשה `Collect` על המערך, ויראה מי המעבדים הפעילים. הוא יממין אותם לפי המזהים הישנים שלהם, וכך הוא ידע מה מיקומו. נניח שהמקום שלו במיון הוא x. אזי הוא ינסה לתפוס את המקום הפנוי ה־x במערך של המזהים החדשים (הקטנים).

למה 40: האלגוריתם מסתיים.

הוכחה: נניח שכל מי שנשאר בחיים לא יסיים את האלגוריתם. אזי יש Suffix של הריצה שימשיך לאינסוף.

נסתכל על תת־מרחב השמות שעדיין פנויים. נסמן אותו ב־E. לא יכול להיות שהשם הקטן ביותר ייתפס יותר מפעם אחת על ידי אותו המעבד, כי כל המעבדים כבר התעוררו, או שלא יתעוררו יותר. אזי יש רק מעבד אחד עם השם הקטן ביותר, ולכן רק הוא ינסה (אם בכלל) לתפוס את השם הקטן ביותר.

נניח שהשם של המעבד הקטן ביותר הוא r (כלומר יש $r - 1$ קטנים ממנו, שכולם כבר עצרו). המעבד הזה יצליח לתפוס את השם הפנוי ה־r הכי קטן אחרי שכל המעבדים יסיימו סיבוב אחד של בחירת שם. זאת סתירה. ■

למה 41: אם k מעבדים התעוררו, השמות יהיו עד $2 \cdot k - 1$.

הוכחה: במקרה הגרוע, כל $k - 1$ המעבדים הקטנים ביותר יסיימו לבחור $k - 1$ מקומות המעבד ה־k לא יצליח לבחור את 1, ואז הוא יבחר את השם ה־k הפנוי בראשון, שהוא:

$$k - 1 + k = 2 \cdot k - 1$$

■

11.3 מניעה הדדית (Mutual Exclusion)

ראינו כבר אלגוריתמים בסיסיים למניעה הדדית של כניסה לקטע קוד קריטי. נראה עכשיו דברים מסובכים יותר, אבל יעילים יותר במבחן התוצאה.

נרצה שהאלגוריתם שלנו יהיה הוגן, כלומר שיקיים FIFO, כלומר התהליכים מקבלים את המנעול לפי סדר הבקשה. לכן, יהיה לנו בקבלת המנעול קטע שנקרא Doorway, ואז Waiting. ב־Doorway לוקחים מספר, וב־Waiting מחכים למספר. האלגוריתם הזה נקרא אלגוריתם הפאפייה (*The Bakery Algorithm*).

אלגוריתם 2.11 אלגוריתם המאפייה

```
1: choosing[i] ← true
2: number[i] ← 1 + max{number[j] | 1 ≤ j ≤ n}
3: choosing[i] ← false
4: for j = 1, 2, ..., n do
5:     await choosing[j] = false
6:     await (number[j] = 0) ∨ ((number[j], j) ≥ (number[i], i))
7: end for
                                                    ▷ Critical Section
8: number[i] ← 0
```

איך ניקח מספרים? אי אפשר לקחת מספר לפי הממתין המקסימלי ועוד 1, כי יכול להיווצר מצב שבו שני מעבדים (או יותר) לוקחים את אותו המספר. גם מיון לקסיקוגרפי לא מספיק טובף כי הוא לא חייב לקיים בהכרח את המניעה ההדדית. אלגוריתם 2.11 מתאר את הקוד של אלגוריתם המאפייה.

אנחנו צריכים להיות זהירים כשמחשבים את המקסימום כשבוחרים את $number[i]$. לא כדאי לנו להשתמש במצביעים לפי מספר המעבד. המימוש הנאיבי יעבוד כאן, אין שום סיבה להתחכם ולקפוץ מעל הפופיק.

הבעיה באלגוריתם המאפייה היא שה־$number[i]$ לא חסומים. במימושים אמיתיים, יכול להיות Overflow.

נראה את אלגוריתס המאפייה השחור־לבן (*Black-White Bakery Algorithm*): לכל מספר יהיה גם ביט של צבע – שחור או לבן. יהיה צבע במקום משותף בזיכרון, שמכריז על הצבע שבוחרים עכשיו. כשמעבד יוצא מהקטע הקריטי, אם הצבע של המספר שלו זהה לצבא של המספרים החדשים שייכנסו עכשיו, הוא ישנה את הצבע.

פרק 12

שיעור 13[1]

12.1 Set Consensus

ב־(k,n)-Set Consensus, כל מעבד i מחזיר פלט v_i, שהיה הקלט של (לפחות) אחד המעבדים. לא מחזירים יותר מ־k פלטים שונים, $k < n$.

נראה את השקילות בין זיכרון משותף להעברת הודעות: נניח שיש f מעבדים שיכולים לקבל Fail-Stop, $f < n/2$. נממש אוגר SRSW.

פעולת `Write` (v): שולח את v לכולם, ומחכה ל־$\lfloor \frac{n}{2} \rfloor + 1$ הודעות Ack. הפעולה מקבלת גם seq (מספר סידורי).

פעולת `Read`: שולח Req לכולם. מחכה לקבל $\lfloor \frac{n}{2} \rfloor + 1$ הודעות. בוחר את התשובה עם ה־seq הגדול ביותר.

המימוש הזה הוא לאוגר Atomic Single-Reader Single-Writer, או Safe Multiple-Reader Single-Writer. נראה עכשיו מימוש לאוגר Atomic Multiple-Reader Single-Writer.

השינוי צריך להיות ב־`Read`. בסוף הקריאה, הקרא שולח את ערך הקריאה שלו, ומחכה ל־$\lfloor \frac{n}{2} \rfloor + 1$ הודעות Ack.

תרגיל 42: נממש $\lfloor \frac{n}{2} \rfloor$-Set Consensus בהעברת הודעות:

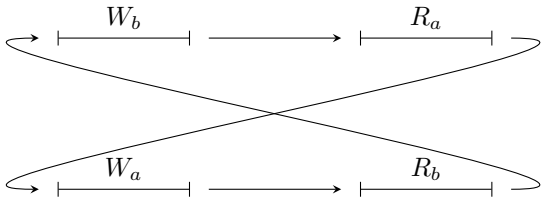
1. שולח לכולם את הערך שלו.
2. חכה ל־$\lfloor \frac{n}{2} \rfloor + 1$ תשובות.
3. החזר את התשובה המקסימלית.

יוחזרו לכל היותר $\lfloor \frac{n}{2} \rfloor$ תשובות, כי כל ה־$\lceil \frac{n}{2} \rceil$ התחתונים תמיד יגיעו ביחד עם מישהו יותר גדול.

12.2 שקילות המודלים

נראה עכשיו שמודל העברת הודעות (פשוט יותר ממה שראינו עד עכשיו) שקול לזיכרון משותף עם אוגרי Atomic Read-Write Wait-Free. כלומר, כל פעולה שניתן לפתור באחד מהם, ניתן לפתור גם בשני.

[1] סיכום לשיעור שהתקיים בתאריך 02.06.2013.



איור 12.1: כפי שניתן לראות באיור, לא יכול להיות שקיים מעבד שלא ראה את הכתיבה של המעבד השני

למודל החדש התכונות הבאות:

1. זהו מודל העברת הודעות.
2. המודל סינכרוני.
3. הרשת היא בתצורה של גרף מלא.
4. יש יריב, שבכל מחזור מותר לו לחסל חלק מההודעות.
5. כל אחד שולח את כל ההודעות שקיבל אי־פעם לכל השכנים שלו.

ההודעות ששרגדו בכל סיבוב מגדירות גרף מכוון, $G = (V, E)$, כאשר V היא קבוצת המעבדים, והקשת $(v, w) \in E$ אם ורק אם היריב לא הרג את ההודעה שנשלחה מ־v ל־w. לכן, היריב מוגדר למעשה על ידי אוסף של גרפים מכוונים.

נגדיר את היריב כך שאם G שייך ליריב, אזי גם $G \subseteq H$ ביריב. כמו כן, אם G ביריב, אזי גם כל גרף איזומורפי ל־G נמצא ביריב.

דוגמאות ליריבים אפשריים:

- כל תת־גרף קשיר היטב הוא יריב.
- $TOUR$(Tournament Graph): בין כל שני קודקודים קיימת קשת אחת (באחד הכיוונים).
- TP (Traversal Path): מסלול מכוון שעובר דרך כל הקודקודים.
- $PAIRS$: בכל סיבוב, משאירים רק קשת אחת, כך שאחרי $\binom{n}{2}$ סיבובים, עברנו על כל הקשתות. כל קשת יכולה להתאפשר רק בכיוון אחד.

טענה 43: יריב $TOUR$ שקול לזיכרון משותף עם אוגרי Read-Write Wait-Free.

הוכחה: נניח שכל מעבד כותב באוגר אחד וקורא מהשני (ויש רק 2 מעבדים). לא יכול להיות שכל אחד מהמעבדים לא ראה את השני, כפי שמודגם באיור 12.1. בפרט, זה מוכיח שכל דבר שניתן לחישוב ב־$TOUR$ ניתן גם לחישוב בעזרת אוגר Read-Write Wait-Free.

נראה עכשיו שכל משימה שפתירה עם אוגרי Read-Write Wait-Free, פתירה גם עם $TOUR$. אחרי שני מחזורים, קיים מלך, כלומר קודקוד שכל הקודקודים האחרים שמעו ממנו, כלומר הקלט שלו הגיע לכולם. זה טעון הוכחה שלא נראה עכשיו.

נראה אלגוריתם ל־$TOUR$ שמממש Non-Blocking Write באמצעות זה שהמלך יכתוב בשני מחזורים. איך המלך ידע שהוא המלך? בסיבוב השלישי, כל קודקוד יגיד לשאר הקודקודים אם הוא שמע ממנו, או שלא.

בהינתן קודקוד v, המצב שלו לגבי u יכול להיות אחד מהבאים:

1. לא שמעתי ממנו. אזי הוא בהכרח שמע ממני.
2. הוא הודיע לי ששמע ממני.
3. הוא הודיע לי שלא שמע ממני.

לכן, תמיד קל לדעת אם כל שער המעבדים שמעו ממני או לא. אם אני יודע שכולם שמעו ממני, אז סיימתי את הכתיבה. איך נראית קריאה? באמצעות כתיבה, כמו שראינו קודם.

ככה יוצא שהאלגוריתם שלנו הוא Non-Blocking, כי תמיד מישהו מצליח להתקדם. ■

קל לראות את השקילות בין $TOUR$ ל־$PAIRS$, כי הם למעשה די דומים.

פרק 13

שיעור 14[1]

13.1 יריבים

למה 44: נרצה להוכיח שבהינתן יריב $TOUR$, יש קודקוד שכולם שמעו ממנו תוך שני מחזורים. נסמן את הקשתות ב־$TOUR$ במחזור הראשון בכחולות, ובמחזור השני באדומות.

אזי קיים קודקוד v, כל שלכל x יש קשת כחולה או אדומה מ־x ל־v, או שיש y כך שיש קשת כחולה מ־x ל־y, וקשת אדומה מ־y ל־v.

הוכחה: נניח בשלילה שאין קודקוד כזה. ניקח קודקוד כלשהו a.

אם a לא "מלך", אזי יש קודקוד b שלא שמע מ־a לאחר שני סיבובים. נזכור שהגרף שלנו הוא Tournament Graph בכל סיבוב, כלומר בכל שלב, בין כל שני קודקודים עוברת הודעה בכיוון כלשהו.

נסמן ב־R_b את קבוצת הקודקודים ש־b שלח להם הודעה במחזור השני, וב־B_a את קבוצת הקודקודים ש־a שלח להם הודעה במחזור הראשון. נרצה להוכיח:

$$|R_b| \geq |B_a| + 1$$

נעשה משהו לא ברור, ונקבל c שעבורו $|R_c| \geq |B_b| + 1$. נמשיך כך, ונקבל $a, b, c, \ldots$. בוודאי זה יוצר מעגל, כי הגרף שלנו בגודל סופי.

ניקח את תת־הגרף המושרה על ידי קודקודי המעגל. גם בתת־הגרף, אי השוויונות מתקיימים, כי גם שם אין מלך. סתירה! ■

[1] סיכום לשיעור שהתקיים בתאריך 09.06.2013.

פרק 14

שיעור 15[1]

14.1 אלגוריתמים רנדומיים להסכמה

אנחנו מחפשים אלגוריתם רנדומי עם הטלת מטבעות בזיכרון משותף עם אוגרי קריאה וכתיבה ו־n מעבדים.

באופן כללי, קיימים שני סוגים של אלגוריתמים רנדומיים:

1. הזמן יכול ללכת ל־∞ בהסתברות מאוד נמוכה, אבל הנכונות של הפלט תמיד נשמרת.
2. הזמן חסום, ומקבלים שגיאה (כלומר פלט לא נכון) בהסתברות מאוד נמוכה.

אנחנו מתעניינים בסוג הראשון של האלגוריתמים.

ניזכר ב־Randm Walk: אם בכל נקודה בזמן מחליטים לעלות למעלה בהסתברות $1/2$, ולרדת למטה בהסתברות $1/2$, ומתחילים ב־0, אזי תוחלת הזמן שנעלה מעל a או נרד מתחת ל־$-a$ היא a^2 צעדים. גם אם קיים יריב שבוחר כל פעם אם להטיל מטבע או להתרחק מ־0, נקבל תוחלת של a^2 (עם שינוי קל בקבועים).

אלגוריתם ההסכמה שלנו יהיה כפי שמתואר באלגוריתם 1.14.

לכל אחד מהמעבדים יש 3 אוגרים משלו, שהם MRSW שאליהם הוא כותב. פעולת הקריאה קוראת את הסכומים של כל האוגרים המתאימים בכל אחד מהמעבדים לתוך האוגרים של המעבד שלי.

[1] סיכום לשיעור שהתקיים בתאריך 16.06.2013.

אלגוריתם 1.14 אלגוריתם רנדומי להסכמה

Require: My input is i
increment a_i
loop
  read a_0, a_1 and c
  if $c > 2 \cdot n$ **then decide** 1
  if $c < -2 \cdot n$ **then decide** 0
  if $a_1 = 0$ or $c \leq -(a_0 + a_1)$ **then** decrement c
  else if $a_0 = 0$ or $c \geq (a_0 + a_1)$ **then** increment c
  else
    $coin \leftarrow$ CoinFlip()
    if $coin = 0$ **then** decrement c
    else increment c
  end if
end loop