אלגוריתמים 2 – סיכום

0 - בסיס לקורס

מתמטיקה בדידה

<u>טבלאות אמת:</u>

טבלת האמת של שלילה

P	$\neg P$
T	F
F	T

טבלת האמת של התנאי

P	Q	$P \to Q$
T	T	T
T	F	F
F	T	T
F	F	T

טבלת האמת של הקוניונקציה

P	Q	$P \wedge Q$
T	T	T
T	F	F
F	T	F
F	F	F

טבלת האמת של הדיסיונקציה

P	Q	$P \vee Q$
T	T	T
T	F	T
F	T	T
F	F	F

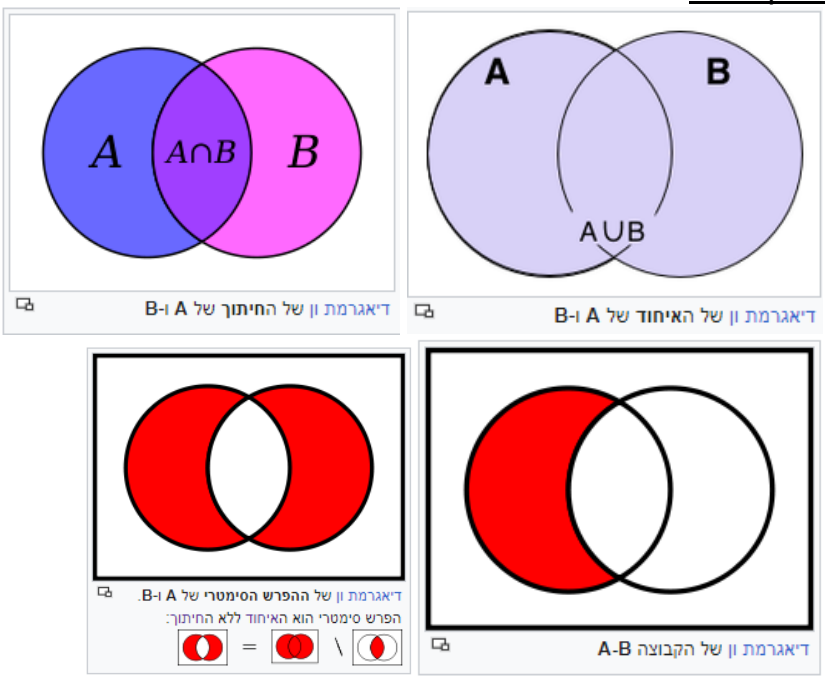
טבלת האמת של התנאי הכפול

P	Q	$P \leftrightarrow Q$
T	T	T
T	F	F
F	T	F
F	F	T

<u>פעולות מתמטיות:</u>

- **פעולה קומוטטיבית** או **פעולה חילופית** היא פעולה בינארית המקיימת את התנאי a * b = b * a לכל a,b (כאשר * מהווה את סוג הפעולה)
- **פעולה אסוציאטיבית** היא פעולה בינארית המקיימת את חוק הקיבוץ, כלומר, לכל a,b,c מתקיים a * (b * c) = (a * b) * c

<u>פעולות על קבוצות:</u>



דיאגרמת ון של **האיחוד** של A ו-B

דיאגרמת ון של **החיתוך** של A ו-B

דיאגרמת ון של הקבוצה A-B

דיאגרמת ון של **ההפרש הסימטרי** של A ו-B.
הפרש סימטרי הוא האיחוד ללא החיתוך:

<u>יחסים:</u>

יחס בינארי או **רלציה** בין קבוצות כלשהן A ו-B הוא קבוצה של זוגות סדורים של איברים, כך שהאיבר הראשון בכל זוג שייך ל-A והשני ל-B.

אם (a,b) שייך לR(יחס כלשהו) נאמר ש-a עומד ביחד ל-b ונסמן aRb.

יחס שקילות:

יחס בינארי R מעל קבוצה A נקרא יחס שקילות אם הוא מקיים את התכונות הבאות:

- **רפלקסיביות:** כל איבר עומד ביחס עם עצמו, כלומר aRa לכל a השייך לקבוצה A.
- **סימטריות:** aRb אם ורק אם bRa. יחס "אח של" שהוא סימטרי.
- **טרנזיטיביות:** אם aRb וגם bRc אזי aRc.

פונקציות:

- **פונקציה**: יחס בו לכל איבר של A קיים זוג סדור *יחיד* שהוא האיבר הראשון שלו.
- **פונקציה חד-חד ערכית**: פונקציה $f: X \to Y$ תקרא **חד-חד ערכית** (חח"ע) אם לכל y בטווח Y קיים לכל היותר x אחד בתחום X המקיים $f(x) = y$
- **פונקציה על**: פונקציה $f: X \to Y$ תקרא **על** אם לכל y בטווח Y קיים לפחות x אחד בתחום X המקיים $f(x) = y$
- **פונקציה הפיכה**: פונקציה שקיימת לה פונקציה הפכית; לכל y קיים x יחיד כך ש-$f(x) = y$ (כלומר, הפונקציה היא חח"ע ועל).

מושגים בסיסיים בקומבינטוריקה:

עצרת של מספר [factorial] מוגדרת עבור מספרים שלמים חיוביים $n = 1,2,3,...$, מסומנת ב-$n!$ ונתונה על ידי הנוסחא

$$n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n = \prod_{j=1}^{n} j$$

3.2 עקרון הכפל

בניסוח הראשון, עקרון הכפל מאפשר חישוב **מספר התוצאות** האפשריות בניסוי רב שלבי:

מספר התוצאות האפשריות בניסוי רב שלבי ניתן על ידי מכפלת מספרי התוצאות האפשריות בכל אחד משלבי הניסוי . כלומר, אם ניסוי מתבצע ב—k שלבים בזה אחר זה כאשר:

בשלב הראשון ישנן n_1 תוצאות אפשריות,
בשלב השני ישנן n_2 תוצאות אפשריות,
,...
בשלב ה—k האחרון ישנן n_k תוצאות אפשריות,

אזי מספר התוצאות השונות בניסוי כולו הוא

$$N = n_1 \cdot n_2 \cdot ... \cdot n_k = \prod_{j=1}^{k} n_j .$$

בניסוח השני (הידוע תחת השם **כלל השרשרת**), עקרון הכפל מאפשר חישוב **ההסתברות** של ניסוי רב שלבי:

עבור המאורע $B = \bigcap_{j=1}^{n} A_j$ מתקיים:

$$P(B) = P(A_1) \cdot P(A_2 / A_1) \cdot P(A_3 / A_1 \cap A_2) \cdot ... \cdot P(A_n / A_1 \cap ... \cap A_{n-1}) .$$

במילים אחרות,

$$P\left(\bigcap_{j=1}^{n} A_j\right) = P(A_1) \prod_{j=2}^{n} P\left(A_j / \bigcap_{k=1}^{j-1} A_k\right) .$$

בחירות: טבלת סיכום

	עם החזרה	בלי החזרה
מסודרת	n^k	P_n^k
לא מסודרת	$P_{n+k-1}(k, n-1)$	C_n^k

$$P_n^k = \frac{n!}{(n-k)!} \qquad C_n^k = \frac{n!}{k!(n-k)!} = \binom{n}{k}.$$

עקרון שובך היונים הוא עיקרון מתמטי הקובע כי אם יש m תאים בשובך שלתוכם יש להכניס m + 1 יונים, קיים בהכרח תא אחד שבו תימצא יותר מיונה אחת.

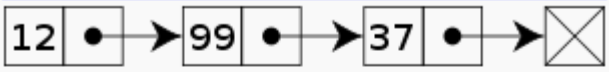
אם יש m תאים בשובך שלתוכם יש להכניס n יונים, אז בהכרח בתא אחד יהיו לפחות $p = \lceil n/m \rceil$ יונים או יותר, (p הוא המספר השלם הקרוב (בעיגול כלפי מעלה) למספר: n/m). כלומר יש תא שמספר היונים בו הוא לפחות כמו הממוצע.

מבני נתונים

רשימה מקושרת

Linked List היא מבנה הנתונים בסיסי לאחסון נתונים. ברשימה המקושרת, אוסף איברים מפוזרים במקומות שונים בזיכרון המחשב (כלומר לא ברצף), כך שבכל איבר מאוחסן הנתון ומצביע לאיבר הבא ברשימה.

רשימה מקושרת



רשימה מקושרת חד-כיוונית

סיבוכיות מקום וזמן

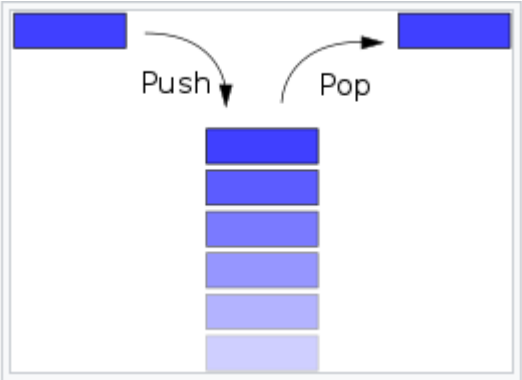
	ממוצע	במקרה הגרוע
זיכרון:	Θ(n)	Θ(n)
חיפוש:	Θ(n)	Θ(n)
הכנסה:	Θ(1)	Θ(1)
הוצאה:	זמן חיפוש+Θ(1)	זמן חיפוש+Θ(1)
שליפה:	Θ(1)	Θ(1)

מחסנית:

Stack היא סוג של מבנה נתונים מופשט הפועל בצורה דומה לזו של מחסנית רובה: האיבר שנכנס ראשון למחסנית יוצא ממנה אחרון (תכונה זו מכונה נכנס אחרון יוצא ראשון - LIFO).

שלוש הפעולות הבסיסיות המגדירות מחסנית הן:

- אתחול - (init) יצירה של מחסנית חדשה. O(1)
- דחיפה - (push) הכנסת איבר חדש לראש המחסנית. O(1)
- שליפה (pop) - פעולה המוציאה את האיבר העליון מהמחסנית. O(1)

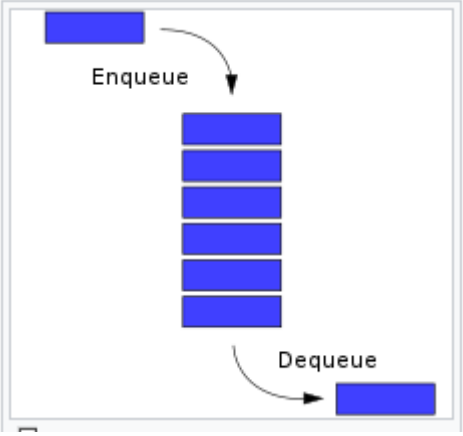


תור:
Queue הוא מבנה נתונים מופשט המוגדר על ידי הפעולות הבאות:

- הכנסה - (enqueue) הוספת אובייקט אחד חדש בסופו של התור
- הוצאה - (dequeue) הוצאתו של האובייקט הנמצא בראש התור
- בדיקה אם התור ריק(isEmpty)
- בדיקת ערך בראש התור(peek)

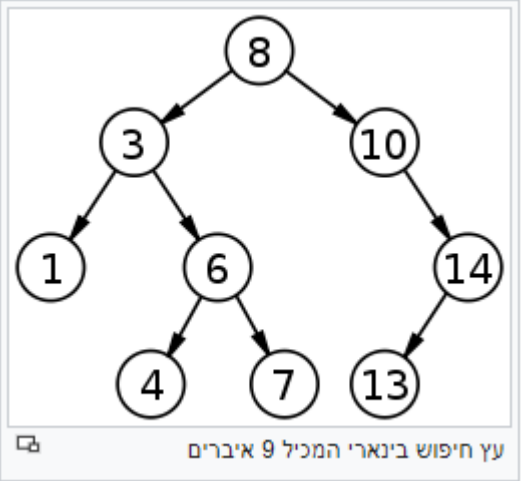
כל הפעולות מתבצעות בסיבוכיות O(1) כלומר במספר פעולות שאיננו תלוי בגודל הקלט.

פעולות ההכנסה וההוצאה בתור מבוססות על העקרון נכנס ראשון - יוצא ראשון FIFO.



עץ בינארי:
מבוסס על עץ מתורת הגרפים וממומש בדומה לרשימה מקושרת שבמקום מצביע לאיבר הבא, לכל איבר בעץ יש מצביע לבן השמאלי ולבן הימני.

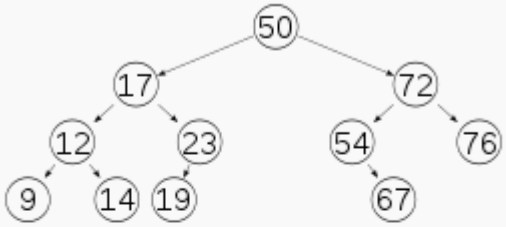
עץ חיפוש בינארי:
עץ בינארי אשר בנוי כך שלכל צומת בעץ, ערך הבן השמאלי קטן מערך הצומת וערך הבן הימני גדול מערך הצומת:



דוגמה לעץ חיפוש בינארי:

עץ AVL הוא מבנה נתונים מסוג עץ חיפוש בינארי מאוזן, שבו הפרש גובהם של שני תתי-העצים של הבנים של כל צומת הוא לכל היותר 1. תכונה זו מבטיחה שניתן יהיה לחפש בעץ, להכניס ולהוציא ממנו נתונים בסיבוכיות של O(logn) במקרה הגרוע ביותר (כאשר n הוא מספר הצמתים בעץ).

עץ AVL



דוגמה לעץ AVL

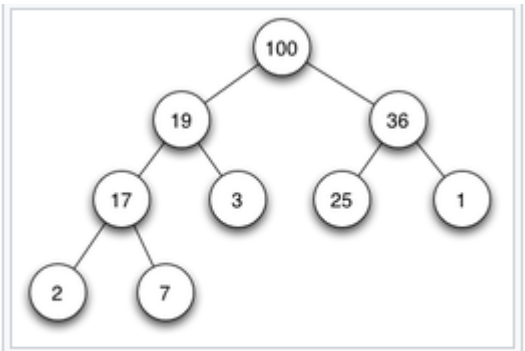
יצירה	
הומצא ב:	1962
ממציא:	גאורגי אדלסון-ולסקי ויבגני לנדיס

סיבוכיות מקום וזמן		
	ממוצע	במקרה הגרוע
זיכרון:	O(n)	O(n)
חיפוש:	O(log n)	O(log n)
הכנסה:	O(log n)	O(log n)
הוצאה:	O(log n)	O(log n)

ערימה:

Heap היא מבנה נתונים בצורת עץ מכוון המקיים תכונה בסיסית, הנקראת

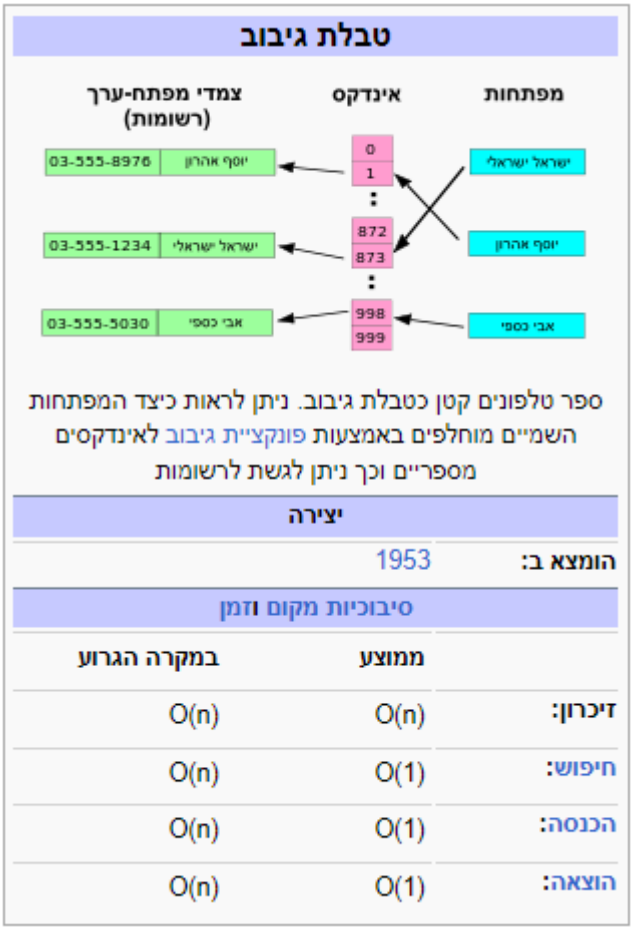
תכונת הערימה: המפתח של כל צומת בעץ קטן ממפתח אביו (בערימת מקסימום). כתוצאה מדרישה זו, הצומת בעל המפתח הגדול ביותר הוא תמיד השורש של העץ, וניתן למצוא אותו בסיבוכיות זמן O(1).



דוגמה לערימה בינארית

טבלת גיבוב:

Hash table היא מבנה נתונים מילוני, אשר נותן גישה לרשומה באמצעות המפתח המתאים לה. המבנה הזה עובד באמצעות הפיכת המפתח על ידי פונקציית הגיבוב, למספר המייצג מיקום במערך שמפנה אל הרשומה המבוקשת. הפעולה העיקרית שבה היא תומכת ביעילות היא אחזור המידע מתוך מבנה הנתונים: בהינתן מפתח נתון (למשל שם של אדם), מצא את הרשומה המתאימה (למשל מספר הטלפון של אותו אדם).



טבלת גיבוב

מפתחות	אינדקס	צמדי מפתח-ערך (רשומות)
ישראל ישראלי	0	
	1	יוסף אהרון 03-555-8976
יוסף אהרון	872	
	873	ישראל ישראלי 03-555-1234
אבי כספי	998	אבי כספי 03-555-5030
	999	

ספר טלפונים קטן כטבלת גיבוב. ניתן לראות כיצד המפתחות השמיים מוחלפים באמצעות פונקציית גיבוב לאינדקסים מספריים וכך ניתן לגשת לרשומות

יצירה	
הומצא ב:	1953

סיבוכיות מקום וזמן		
	ממוצע	במקרה הגרוע
זיכרון:	O(n)	O(n)
חיפוש:	O(1)	O(n)
הכנסה:	O(1)	O(n)
הוצאה:	O(1)	O(n)

אלגוריתמי חיפוש ומיון

<u>מיון בועות:</u>

Bubble sort: בהינתן מערך בגודל n

- עוברים בלולאה מ-0 עד n-1
- בתוך הלולאה, בלולאה נוספת מבצעים swap במידה ו– arr[j] > arr[j+1] (או הפוך, תלוי אם רוצים מערך בסדר עולה או יורד)

מימוש:

```python
def bubbleSort(arr):
    n = len(arr)

    # Traverse through all array elements
    for i in range(n):

        # Last i elements are already in place
        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

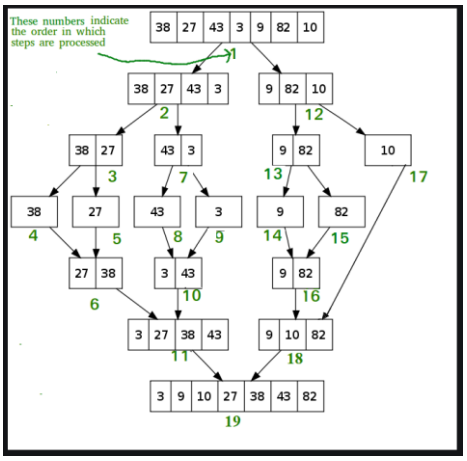
סיבוכיות: O(n^2)

<u>מיון מיזוג:</u>

Merge sort: בהינתן מערך בגודל n

- באופן רקורסיבי מחלקים את המערך כל פעם בחצי עד שנשארים עם איבר בודד ולאחר מכן ממזגים את האיברים לפי הסדר המתאים (עולה\יורד)

המחשה:



מימוש:

```
MergeSort(arr, left, right):
    if left > right
        return
    mid = (left+right)/2
    mergeSort(arr, left, mid)
    mergeSort(arr, mid+1, right)
    merge(arr, left, mid, right)
end
```

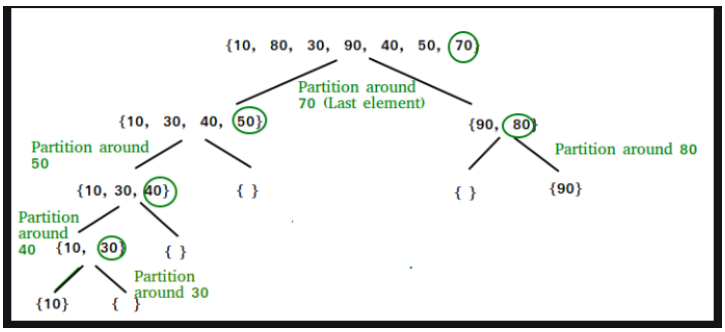
סיבוכיות: O(nlogn)

<u>מיון מהיר:</u>

Quick Sort: בהינתן מערך בגודל n

- בוחרים pivot (בדרך כלל האיבר הכי אחרון, הכי ראשון, האמצעי, או בוחרים באופן רנדומלי)
- לפי בחירת הpivot, עוברים על המערך ומבצעים swap לאיברים שגדולים יותר מpivot מימין לpivot והאיברים הקטנים משמאל לpivot
- באופן רקורסיבי מחלקים את המערך לפי האינדקס של הpivot

המחשה:



מימוש:

```
quickSort(arr, beg, end)
  if (beg < end)
    pivotIndex = partition(arr,beg, end)
    quickSort(arr, beg, pivotIndex)
    quickSort(arr, pivotIndex + 1, end)

partition(arr, beg, end)
  set end as pivotIndex
  pIndex = beg - 1
  for i = beg to end-1
  if arr[i] < pivot
    swap arr[i] and arr[pIndex]
    pIndex++
  swap pivot and arr[pIndex+1]
return pIndex + 1
```

סיבוכיות: O(nlogn) ובמקרה הגרוע (תמיד בוחר את האיבר הכי גדול או הכי קטן במערך): O(n^2)

<u>חיפוש בינארי:</u>

בהינתן מערך ממוין וערך x שיש למצוא את מיקומו:

- נמצא את האינדקס החציון של המערך ונבדוק האם x קטן או גדול מהחציון.
- אם x גדול מהחציון, נקרא באופן רקורסיבי לאלגוריתם במערך שמתחיל מהחציון עד סוף המערך
- אחרת, נקרא באופן רקורסיבי לאלגוריתם עם מערך שמתחיל בleft עד החציון

מימוש:

```
binarySearch(arr, x, low, high)
       if low > high
           return False

       else
           mid = (low + high) / 2
               if x == arr[mid]
               return mid

           else if x > arr[mid]        // x is on the right side
               return binarySearch(arr, x, mid + 1, high)

           else                        // x is on the left side
               return binarySearch(arr, x, low, mid - 1)
```

סיבוכיות: O(log)

<u>חיפוש איבר ה-k בגודלו:</u>

אלגוריתם Select: בהינתן מערך בגודל n ואיבר k שעלינו למצוא בגודלו

- בדומה לquicksort, באמצעות פונקציית הpartition, נפעיל את quicksort רק על הצד המתאים שבו נכנס האינדקס k שעלינו למצוא את ערכו.

מימוש:

```
function quickSelect(list, left, right, k)

   if left = right
      return list[left]

   Select a pivotIndex between left and right

   pivotIndex := partition(list, left, right,
                                   pivotIndex)
   if k = pivotIndex
      return list[k]
   else if k < pivotIndex
      right := pivotIndex - 1
   else
      left := pivotIndex + 1
```

סיבוכיות: O(n) ובמקרה הגרוע O(n^2)

תזכורות מאלגוריתמים 1

<u>זמני ריצה:</u>

ניתוח אלגוריתם:

- הערכת יעילות האלגוריתם = הערכת כמות המשאבים שהאלגוריתם דורש.
- **זמן ריצה** = פונקציה של גודל הקלט המחשבת את מספר צעדי החישוב(פעולות בסיסיות) ומסומנת באות T.
- **מקום** = כמות הזיכרון הדרוש ביחס לגודל הקלט.
- המטרה: מציאת אלגוריתם שזמן ריצתו פחותה ביותר

סוגי זמני ריצה:

- זמן הריצה **הגרוע** ביותר: זמן הריצה "הארוך ביותר" על איזשהו קלט בגודל n
- זמן הריצה **הטוב** ביותר: זמן הריצה "הקצר ביותר" על איזשהו קלט בגודל n
- זמן ריצה **הממוצע**: זמן הריצה "הקצר ביותר" על איזשהו קלט בגודל n

חסמים אסימפטוטיים:

בכל מחשב זמן ביצוע הפקודה יכול להשתנות, אך, השינוי הוא עד כדי מכפלה בקבוע(או תוספת של קבוע).
אנו נתעניין בזמני ריצה מבחינת "סדר גודל" ולא בזמני הריצה המדויקים.
נתעלם מזמן הריצה על קלטיים קטנים, ונתעלם ממכפלה פי קבוע.

o is like < ,
O is like ≤,
ω is like >,
Ω is like ≥ ,
Θ is like =.

<u>נוסחאות נסיגה:</u>
היא משוואה או אי שוויון, המתארים פונקציה באמצעות הערכים שהיא מקבלת על קלטים קטנים יותר.

לדוגמה, נוסחת הנסיגה עבור Merge-Sort:

$$T(n)=\begin{cases}\Theta(1) & if\ n=1\\ 2T(n/2)+\Theta(n) & if\ n>1\end{cases}$$

<u>משפט האב:</u>
אחת מהדרכים לפתור נוסחת נסיגה היא באמצעות משפט האב האומר:

נותנת פתרון מיידי עבור נוסחאות נסיגה מהצורה:

$$T(n)=aT(\frac{n}{b})+f(n)$$

$a\geq 1,\quad b>1,$ חיובית אסימפטוטית פונקציה $f(n)$

<u>משפט האב</u>: יהיו $a \geq 1$ ו $b > 1$ קבועים, $f : N \to R^+$. תהיה $T : N \to R^+$ מוגדרת ע"י

$T(n) = aT(\frac{n}{b}) + f(n)$ אז:

1. אם קיים קבוע $\varepsilon > 0$ כך ש: $f(n) = O(n^{\log_b a - \varepsilon})$ אזי- $T(n) = \Theta(n^{\log_b a})$.
2. אם $f(n) = \Theta(n^{\log_b a})$ אזי- $T(n) = \Theta(n^{\log_b a} \cdot \log_2(n))$.
2. אם $f(n) = \Theta(n^{\log_b a} \log^k n)$ כאשר $k \geq 0$ אזי $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$
3. אם קיים קבוע $\varepsilon > 0$ כך ש: $f(n) = \Omega(n^{\lceil \log_b a + \varepsilon \rceil})$

ואם קיים הקבוע $0<C<1$ כך ש: $a \cdot f(\frac{n}{b}) \leq C \cdot f(n)$ (עבור $n > n_0$) אזי $T(n) = \Theta(f(n))$.

שיטות יסוד אלגוריתמיות

<u>אלגוריתם חמדן:</u>

הוא אלגוריתם המוצא פתרון אופטימלי לבעיה, ע"י כך שבכל שלב נבחרת האפשרות הנראית כטובה ביותר באותו הרגע.

אלגוריתם זה נכון כאשר מתקיימות התכונות הבאות:

תכונת הבחירה החמדנית: קיים פתרון אופטימלי המתחיל בבחירה החמדנית שהאלגוריתם בוחר.

תת-מבנה אופטימלי: לאחר הבחירה החמדנית הראשונה, הבעיה מצטמצמת למציאת פתרון אופטימלי לבעיה המתיישבת עם הבחירה הראשונה.

איך נוכיח נכונות של האלגוריתם החמדני:

נוכיח את שתי התכונות שציינו למעלה. בד"כ עדיף להתחיל עם תכונת הבחירה החמדנית ולהוכיח אותה בשלילה ולאחר מכן התכונה השנייה קל להוכיח באופן טריוויאלי.

לפעמים נצטרך להיעזר באינדוקציה

<u>שיטת הפרד ומשול:</u>

עובד באופן הבא:

- **הפרד:** פצל את הבעיה לכמה תתי בעיות קטנות יותר וזרות
- **משול:** פתור את התת בעיות באופן רקורסיבי
- **צרף:** צרף את כל הפתרונות של התת בעיות לבעיה הגדולה

<u>תכנון דינאמי:</u>

לאיזה בעיות נשתמש בתכנון דינמי?

- בעיות שניתן לפתור בצורה רקורסיבית (הפרד ומשול).
- כך שיש קריאות רבות עם קלט זהה שוב ושוב.

במקום לחשב מחדש כל פעם את הקריאות הרקורסיביות נחשב את הפונקציה על כל קלט פעם אחת בלבד.

שתי צורות חישוב סטנדרטיות:

- בחישוב "מלמעלה למטה"
- בחישוב "מלמטה למעלה"

שיטת העבודה:

פתרונות לשאלות ע"י אלגוריתמי תכנות דינמי מורכבים מהחלקים הבאים:

א. אלגוריתם נאיבי.

ב. אלגוריתם רקורסיבי שכולל:

(א) נוסחא רקורסיבית

(ב) תנאי עצירה

(ג) הסבר רעיוני של הנוסחא ומשמעות של כל אינדקס ומשתנה בנוסחא.

(ד) הוכחת נכונות של הנוסחא הרקורסיבית

ג. אלגוריתם תכנות דינמי שכולל:

(א) תאור מבנה הנתונים (מערך, עץ, טבלה, וכו')

(ב) גודל המבנה

(ג) מה מכילה יחידה בודדת בתוך המבנה

(ד) אופן מילוי המבנה

i. תנאי עצירה

ii. איך משתמשים בנוסחא הרקורסיבית

iii. סדר מילוי היחידות (מלמעלה למטה, משמאל לימין, וכדומה)

(ה) היכן במבנה נמצא הפתרון לבעיה

(ו) פסאדו קוד לאלגוריתם

(ז) ניתוח סיבוכיות זמן ומקום – ולציין אם ניתן לצמצם שימוש בזיכרון (איך ולכמה).

(ח) שחזור פיתרון (אם רלוונטי)

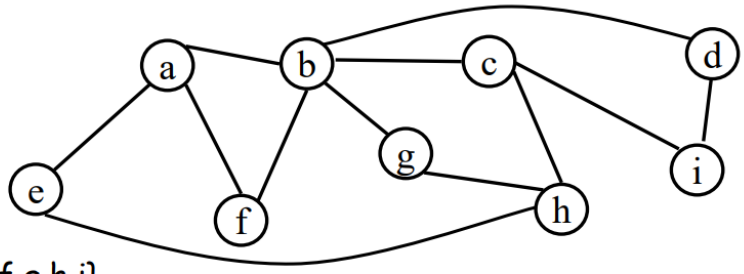
0 – תורת הגרפים מושגי בסיס

גרף לא מכוון

גרף הוא אוסף של קודקודים (צמתים) וקשתות (צלעות)
<u>הגדרה פורמלית:</u>
גרף הוא זוג סדור E,V=(G) כאשר:

- V = קבוצה סופית שאיבריה נקראים **קודקודים** או **צמתים**
- E = קבוצה שאיבריה הם קבוצות בנות שני איברים של V, כלומר: $E = \{ \{u,w\} \mid u,w \in V \}$
  אברי E נקראים **קשתות** או **צלעות**. מסמנים קשת ב- (w,u)

דוגמא:



V={a,b,c,d,e,f,g,h,i}

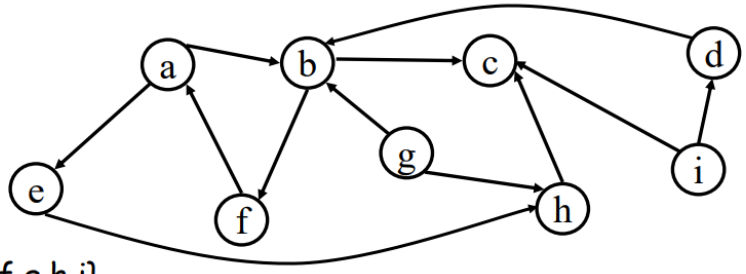
E=
{(a,b),(a,f),(a,e),(b,f),(b,g),(b,c),(b,d),(c,h),(c,i),(d,i),(h,g),(h,e)}

גרף מכוון

בשונה מגרף לא מכוון,

E = קבוצה שאיבריה הם זוגות **סדורים** מ-V^2, כלומר: $E = \{ (u,w) \mid u,w \in V \}$

דוגמא:



V={a,b,c,d,e,f,g,h,i}

E=
{(a,b),(f,a),(a,e),(b,f),(g,b),(b,c),(d,b),(h,c),(i,c),(i,d),(g,h),(e,h)}

דרגה של צומת

<u>הגדרה:</u> יהי $G = (V,E)$ גרף לא מכוון והיו u,v צמתים השייכים ל-V.
אם הקשת (u,v) שייכת ל-E, אז הצמתים u,v נקראים **שכנים**.

<u>הגדרה:</u> נסמן ב-**N(v)** את הקבוצה שמכילה את כל השכנים של v ונקרא לה ה**שכנות** של v.

<u>הגדרה:</u> מספר השכנים של v נקראת ה**דרגה** של v ומסומנת כך: **deg(v).**

<u>הגדרה:</u> בגרף מכוון, ניתן להפריד בין דרגה של קשתות היוצאות מ-v: **deg(v)+**
ובין דרגה של קשתות הנכנסות ל-v: **deg(v)-**

- אם deg(v)=0 אז הוא **צומת מבודד**.
- אם deg(v)=1 אז הוא **עלה**.

<u>הגדרה:</u> **דרגה מינימלית** זוהי דרגה הקטנה ביותר של צומת v כלשהו בגרף ומסומנת כך:
$\delta(G) = \min \{ \deg(v) : v \in V\}$

<u>הגדרה:</u> **דרגה מקסימלית** זוהי דרגה הגדולה ביותר של צומת v כלשהו בגרף ומסומנת כך:
$\Delta(G) = \max\{\deg(v) : v \in V\}$

<u>הגדרה:</u> גרף שבו הדרגה של כל צומת שווה לערך r כלשהו נקרא גרף **r – רגולרי**.

- G הוא r – רגולרי אם ורק אם: $\Delta(G) = \delta(G) = r$.

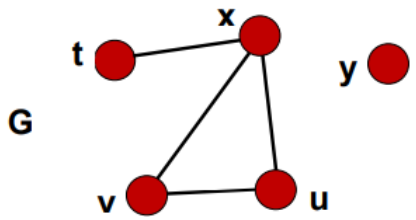
דוגמאות:

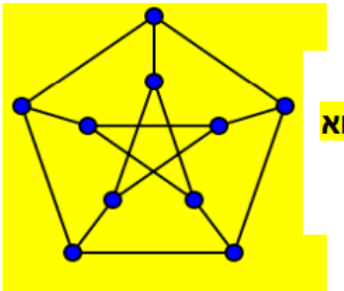
t = עלה, כי $\deg(t) = |N(t)| = |\{x\}| = 1$

y = צומת מבודד, כי $\deg(y) = |N(y)| = |\varnothing| = 0$

$\deg(u) = |N(u)| = 2 \Leftarrow N(u) = \{x, v\}$

$\Delta(G) = \deg(x) = 3$ וגם $\delta(G) = \deg(y) = 0$





הגרף של Petersen הוא 3 – רגולרי

משפט אויילר

<u>הגדרה:</u> בגרף **מכוון** G = (V,E) מתקיים:

$$\sum_{v \in V} deg^{+}(v) = \sum_{v \in V} deg^{-}(v) = |E|$$

<u>הגדרה:</u> בגרף **לא מכוון** G = (V,E) מתקיים:

$$\sum_{v \in V} deg(v) = 2|E|$$

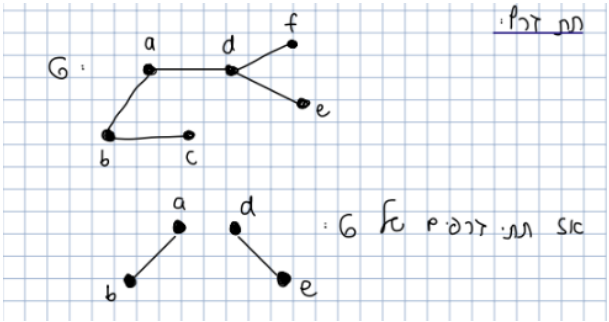
<u>מסקנות:</u> בגרף **לא מכוון** G = (V,E)

- מספר הצמתים בעלי דרגה **אי-זוגית** הוא **זוגי.**
- אם G הוא r – רגולרי וכולל n צמתים, אז לפחות אחד מהמספרים r,n הוא זוגי כי:

$$\sum_{v \in V} deg(v) = n \times r = 2|E|$$

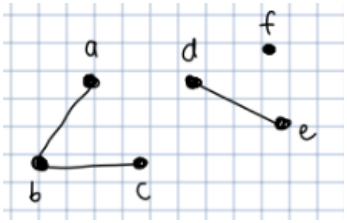
תת גרף

<u>הגדרה:</u> עבור G = (V,E) הגרף G' = (V', E') נקרא **תת גרף** אם V' מוכל ב-V וגם E' מוכל ב-E.



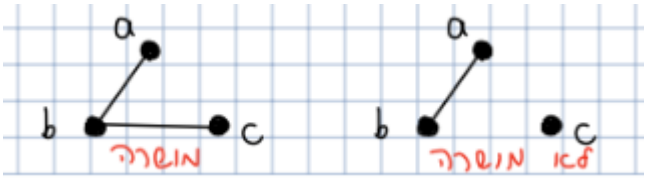
<u>הגדרה:</u> הגרף G' = (V', E') נקרא **תת גרף פורש** אם הוא מכיל את כל הצמתים אך לא בהכרח את כל הקשתות.

דוגמה לתת גרף פורש של הגרף G שהראינו למעלה:



<u>הגדרה:</u> הגרף G’ = (V’, E’) נקרא **תת גרף מושרה** אם הוא תת גרף וגם עבור כל צומת שהתת גרף מכיל, הוא מכיל גם כן את כל הקשתות של אותו צומת כמו בגרף המקורי (ממש כמו לקחת סקרינשוט של חלק מהגרף).

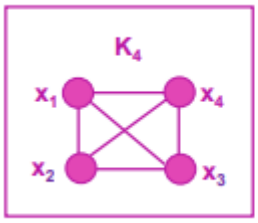
דוגמה לתת גרף מושרה של הגרף G שהראינו למעלה:

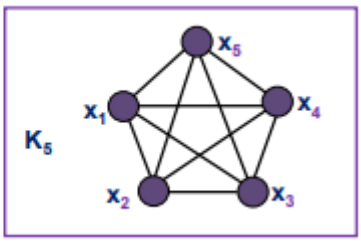


גרף שלם

<u>הגדרה:</u> הגרף G = (V,E) נקרא **גרף שלם** אם מכל צומת יוצאת קשת לכל שאר הצמתים.

- השם הנהוג של גרף זה הוא **קליקה** והוא מסומן באופן הבא: **Kn** כך ש-n מהווה את מספר הצמתים בגרף.





- נוסחה למציאת כמות הקשתות בקליקה:

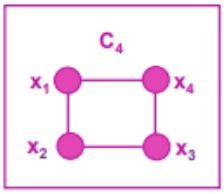
$$|E| = \frac{|V| * (|V| - 1)}{2}$$

- הדרגה של כל צומת היא n-1 ולכן ניתן להגיד שקליקה Kn היא גם **(n-1) – רגולרי.**

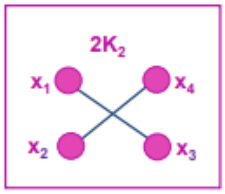
גרף משלים

<u>הגדרה:</u> עבור הגרף G = (V,E) נאמר שהגרף G’ = (V,E’) הוא **גרף משלים** אם הוא מכיל את כל הצמתים של G וגם אם נאחד את קשתות G ו-G’ נקבל קליקה.

המשלים של



הוא



אם G = (V,E) גרף לא מכוון ואילו G' = (V,E') הוא גרף המשלים שלו, אזי:

- $|E| + |\bar{E}| = \frac{n*(n-1)}{2} = \frac{|V|*(|V|-1)}{2}$ (זהה לנוסחה למציאת כמות הקשתות של קליקה)
- $deg_G(v) + deg_{\bar{G}}(v) = n - 1 = |V| - 1$ לכל v השייך ל-V.
- אם G הוא r – רגולרי, אזי G' הוא (n-r-1) – רגולרי.

טיול

<u>הגדרה:</u> אם ישנה דרך להגיע מצומת v לצומת u כלשהן בגרף G, אזי הדרך הזו נקראת **טיול**.

<u>הגדרה:</u> טיול נקרא **טיול סגור** אם הצומת שממנה התחלנו היא גם צומת סיום טיול.

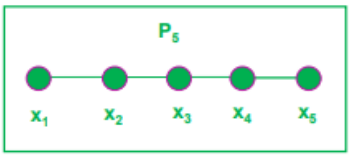
מסלול

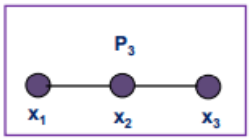
<u>הגדרה:</u> טיול נקרא **מסלול** אם אין חזרה על הקשתות.

<u>הגדרה:</u> מסלול נקרא **מסלול פשוט** אם אין חזרה גם על הצמתים.

- נהוג לסמן מסלול פשוט באופן הבא: Pn כך ש-n מהווה את כמות הצמתים במסלול.

דוגמאות למסלולים פשוטים:





- כמות הקשתות במסלול פשוט Pn הינו n-1: $|E| = |V| - 1$

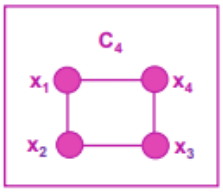
מעגל

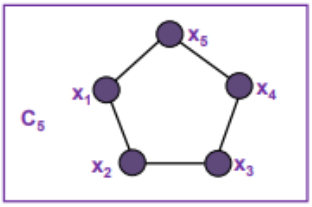
<u>הגדרה:</u> מסלול נקרא **מעגל** אם הוא מתחיל ונגמר באותה הצומת.

<u>הגדרה:</u> מעגל נקרא **מעגל פשוט** אם אין חזרה על הצמתים פרט לצומת התחלה והסוף. (מסלול פשוט שמתחיל ונגמר באותה הצומת)

- נהוג לסמן מעגל פשוט באופן הבא: Cn כך ש-n מהווה את כמות הצמתים במעגל.

דוגמאות למעגלים פשוטים:

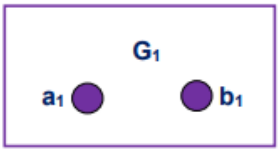


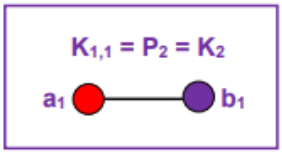


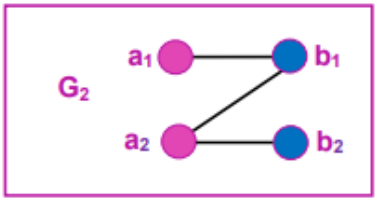
- כמות הקשתות במעגל פשוט Cn הינו n: $|E| = |V|$

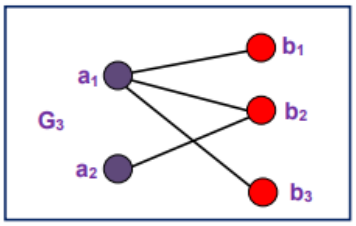
גרף דו צדדי

<u>הגדרה:</u> נאמר שהגרף G = (A,B,E) הוא **גרף דו צדדי** אם האיחוד של A ו-B שווה לכל הצמתים בגרף וגם לא עוברות קשתות בין חברי הקבוצה אלא רק לקבוצה השנייה בלבד.

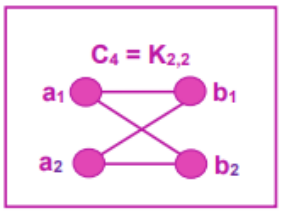


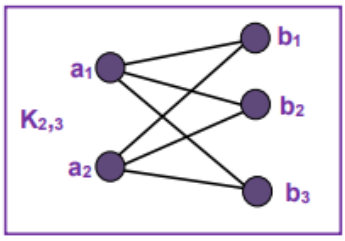






<u>הגדרה:</u> נאמר שגרף דו צדדי G = (A,B,E) הוא **גרף דו צדדי שלם**, אם מכל צומת בקבוצה A יוצאות קשתות לכל צמתי הקבוצה B ולהיפך.





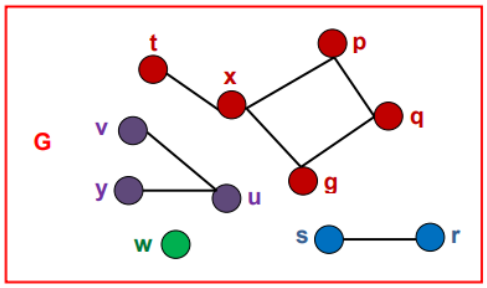
<u>הערה:</u> מספר הקשתות בגרף דו צדדי שלם = |A| * |B|

<u>משפט:</u> גרף לא מכוון G הוא גרף דו צדדי אם ורק אם כל מעגל ב-G הוא בעל מספר זוגי של צלעות.

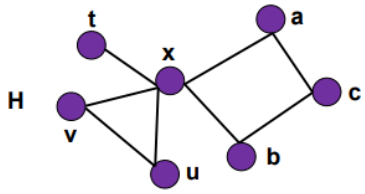
<u>משפט:</u> יהי גרף לא מכוון G, קשיר בעל קבוצת צלעות לא ריקה. אזי:
G הוא דו צדדי ⇔ המספר הכרומטי שווה ל-2 (ראה הסבר בנושא צביעת גרף) ⇔ כל מעגל ב-G הוא בעל מספר זוגי של צלעות.

גרף קשיר

<u>הגדרה:</u> עבור גרף לא מכוון G = (V,E) נאמר שגרף הוא **קשיר** אם לכל צומת v ו-u קיים מסלול. כלומר שניתן להגיע לכל צומת מכל צומת.



גרף לא קשיר = G



גרף קשיר = H

<u>הגדרה:</u> עבור גרף מכוון שעבור כל צומת v ו-u יש מסלול מכוון, נאמר שהגרף הינו **קשיר היטב**(או **קשיר חזק**).

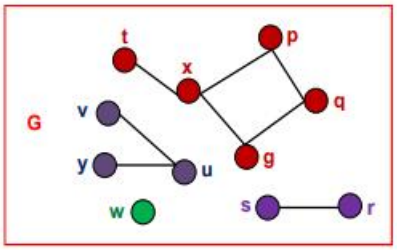
<u>הגדרה:</u> יהי גרף G מכוון ונגדיר G~ = (V,E~) שמוגדר לפי הכלל: (u,v) שייך ל-E~ ⇔ (u,v) או\ו (v,u) שייכים ל-E.

<u>הגדרה:</u> יהי גרף G מכוון ונאמר שהינו **קשיר חלש** אם G~ קשיר.

<u>הגדרה:</u> עבור גרף G = (V,E) **היחס** V x V המוכל בR ומוגדר לפי הכלל הבא:
uRv ⇔ קיים מסלול מצומת u לצומת v בגרף G, הוא **יחס שקילות** מעל V.

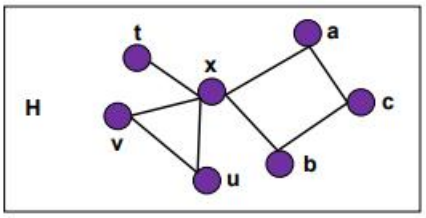
<u>הגדרה:</u> אם G לא מכוון, אזי כל מחלקת שקילות של V/R נקראת **רכיב קשירות** של G.

<u>הגדרה:</u> **גשר** בגרף הוא קשת שהסרתה מהגרף מגדילה את מספר רכיבי הקשירות.



4 רכיבי קשירות (מחלקות שקילות):

V/R = {[y], [s], [w], [x]}



הגרף - קשיר, לכן יש לו רק רכיב קשירות אחד:

V/R= {V}

<u>הגדרה:</u> **צומת הפרדה** בגרף, בדומה לגשר, זוהי צומת שאם נוריד אותה ואת הקשתות המחוברות אליה מהגרף, רכיבי הקשירות יגדלו.

<u>הגדרה:</u> בגרף מכוון הצומת **שורש** זוהי צומת שממנה יש מסלול לכל צומת אחר בגרף.

<u>הגדרה:</u> הגרף המכוון G^t = (V,E^t) שמוגדר לפי הכלל: (v,u) שייך ל-E^t ⇔ (u,v) שייך ל-E ונקרא ה-**transpose** (או **הגרף המוחלף** של G). (כלומר הכיוונים של הקשתות מתהפכים ב-G^t).

<u>הערה:</u> צלע e נמצאת במעגל כלשהו בגרף G ⇔ צלע e אינה גשר

הערה: צומת v שייכת למעגל ⇔ יש צלע e שיוצאת מ-v שאינה גשר (שייכת למעגל)

הערה: יכול להיות יותר משורש אחד בגרף.

הערה: המושג קשיר שייך אך ורק לגרף לא מכוון.

מרחק וקוטר

הגדרה: עבור גרף לא מכוון G = (V,E) נאמר שאורך המסלול הקצר ביותר בין צמתים u ו-v נקרא **מרחק** ונסמן: d = **dist(u,v)** כך ש-d מהווה את הערך של המרחק.

- ערך המרחק בין שני צמתים שאין ביניהם מסלול: ∞

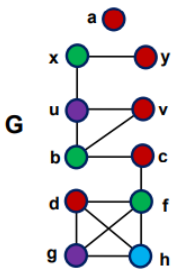
הגדרה: המרחק המקסימלי בגרף G = (V,E) נקרא **קוטר** ונסמן: d = **diam(G)** כך ש-d מהווה את ערך הקוטר.

- עבור גרף לא קשיר ערך הקוטר יהיה תמיד ∞ כי קיימים לפחות שני צמתים שלא קיים מסלול ביניהן.

קליקה

הגדרה: עבור גרף לא מכוון G = (V,E). אם קיימת קבוצת צמתים A המוכלת ב-V וגם לכל הצמתים u ו-v השייכים לA קיימת קשת, אזי A נקראת **קליקה**(=תת גרף שלם של G).

- **ω(G)** = גודל קליקה הגדולה ביותר בגרף G.



A_1 = {u,x,y} אינה קליקה כי (u,y)∉E

A_2 = {u,b,v} - קליקה בגודל 3

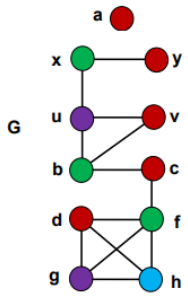
A_3 ={d,f,g,h} קליקה **בגודל מקסימאלי**

ω(G) = 4 **A_3 = קליקה מקסימום**

קבוצה בלתי תלויה

<u>הגדרה:</u> עבור גרף לא מכוון G = (V,E), אם קיימת קבוצת צמתים A המוכלת ב-V וגם לכל הצמתים u ו-v השייכים ל-A **אין אף קשת אחת** המחברת בין הצמתים, אזי A נקראת **קבוצה בלתי תלויה**.

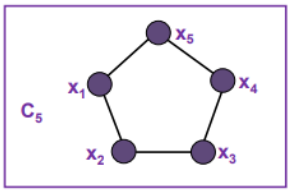
- $\alpha(G)$ = גודל קבוצה בלתי תלויה הכי גדולה בגרף G.
- אם A קבוצה בלתי תלויה ולא קיימת קבוצה בלתי תלויה אחרת ש-A מוכלת בה, אזי A נקראת קבוצה בלתי תלויה **מקסימלית**.
- אם A קבוצה בלתי תלויה וגם |A| = $\alpha(G)$ אז A נקראת קבוצה בלתי תלויה **מקסימום**.
- כל קבוצה בלתי תלויה מקסימום היא גם קבוצה בלתי תלויה מקסימלית אבל **ההפך לא בהכרח נכון**.



- {x,v,c,d} =קבוצה בלתי תלויה בגודל **4**
- **S = {a,y,v,c,d} -** קבוצה בלתי תלויה **מקסימום**

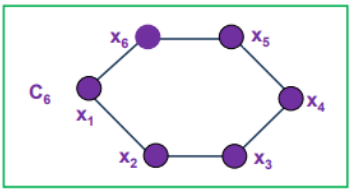
α(G) = |S| = 5

הקבוצה **{a,x,b,f}** היא בלתי תלויה מקסימלית, אך לא מקסימום.



$\alpha(C_5) = |\{x_1, x_3\}| = |\{x_1,x_4\}| = 2$

$\alpha(C_6) = |\{x_1, x_3, x_5\}| = |\{x_2,x_4,x_6\}| = 3$



$\alpha(C_n) = |\{x_1,x_3,...\}| = \lfloor n/2 \rfloor$

קליקה – קבוצה בלתי תלויה

הערות:

(א) $\omega(C_n) = 2 \Leftrightarrow 3 < n$, $\omega(C_3) = 3$.

(ב) $G = K_n \Leftrightarrow \alpha(G) = 1 \Leftrightarrow \omega(G) = n$.

(ג) יהי G = (V,E) לא מכוון ו: H = (V,U) **המשלים** שלו.

1. S בלתי תלויה ב- G $\Leftrightarrow$ S היא קליקה של H
2. **S בלתי תלויה מקסימום ב- G $\Leftrightarrow$ S היא קליקה מקסימום של H**

- **כלומר מתקיימים:** $\alpha(G) = \omega(H)$ & $\alpha(H) = \omega(G)$

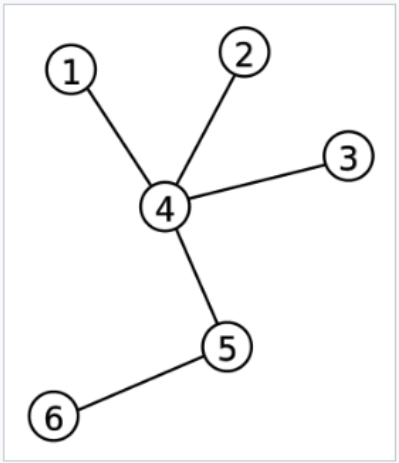
עצים

<u>הגדרה:</u> גרף לא מכוון וללא מעגלים נקרא **יער** ונהוג לסמן אותו כ: **F = (V,E).**

<u>הגדרה:</u> גרף לא מכוון, קשיר וללא מעגלים נקרא **עץ** ונהוג לסמן אותו כ: **T = (V,E)**.

<u>משפט:</u> נתון G = (V,E) גרף לא מכוון בעל |V| ≥ 2 צמתים
אזי הטענות הבאות שקולות זו לזו:
1 – G הוא עץ
2 – G קשיר וגם |E| = |V| - 1
3 – G ללא מעגלים וגם |E| = |V| - 1
4 – G קשיר וכל צלע שלו היא גשר
5 – לכל v ו-u השייכים ל-V קיים מסלול אחד ויחיד ב-G המחבר ביניהן
6 – G ללא מעגלים, אבל ב: G+e קיים מעגל אחד בלבד, לכל צלע e השייכת לגרף המשלים של G

<u>משפט:</u> בכל T = (V,E) עץ המכיל לפחות צלע אחת יש לפחות שני עלים.



בעץ שבתמונה יש 6 צמתים, ולכן 6-1=5 קשתות. המסלול הפשוט היחיד שמחבר את צמתים 2 ו-6 הוא 2-4-5-6.

<u>הערה:</u> אם נוסיף קשת אחת נוספת לעץ נקבל מעגל אחד בגרף

מושגים בסיסיים בעצים:

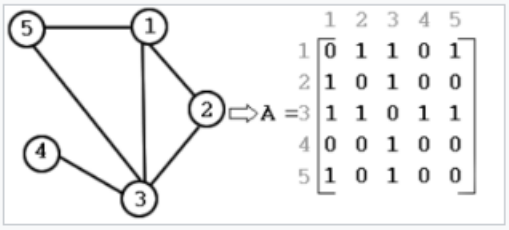
- **שורש** - בגרפים מכוונים, שורש הוא צומת שקיים מסלול ממנו לכל צומת אחר בגרף.
- **רב-עץ** - או **עץ מכוון** - גרף מכוון שגרף התשתית שלו הוא עץ, ושאת אחד מצמתיו ניתן לסמן כשורש העץ, כך שיש מסלול מהשורש לכל צומת בעץ. בהינתן עץ לא מכוון, ניתן להפוך אותו לעץ מכוון על ידי בחירה שרירותית של אחד הצמתים בתור שורש, ובחירת הכיוון שעל הקשתות בצורה שתאפשר מעבר מהשורש לכל צומת בעץ.
- **עץ מושרש** - עץ שקיים בו שורש.
- **עומק של צומת** - מספר הקשתות במסלול בין השורש לצומת.
- **אב ובן** – צומת v הוא האב של צומת w ו-w הוא הבן של צומת v ⇔ ישנה קשת ביניהם ועומקו של v קטן ב-1 מעומקו של w.
- **עלה** - בגרף מכוון, צומת שאין לו בנים. בגרף לא-מכוון, צומת שהדרגה שלו קטנה או שווה 1.
- **קודקוד פנימי** - צומת שיש לו בנים.
- **אב קדמון וצאצא** – צומת v הוא האב קדמון של צומת w ו-w הוא הצאצא של צומת v ⇔ w הוא הבן של v או ש-w הוא הבן של צאצא של v.
- **גובה של צומת** - מספר הקשתות במסלול הארוך ביותר בין הצומת לאחד הצאצאים שלו.
- **גובה העץ** - מספר הקשתות בין השורש לבין העלה שנמצא במסלול הארוך ביותר.
- **תת עץ** - בהינתן עץ T, תת העץ שלו הוא עץ שצמתיו הם צומת v והצאצאים של צומת v מהעץ המקורי T כאשר v הוא שורשו של תת העץ. הקשתות של תת העץ זהות לחלק בעץ המקורי T שעוברות בין הצמתים.
- **עץ פורש** - עץ המוכל בגרף G ומכיל את כל הצמתים שלו.
- **עץ בינארי** – עץ שמכיל שורש ושאר הצמתים מכילים 0, 1 או 2 בנים לכל היותר.

0 – עץ פורש מינימלי

מטריצת שכנות (מטריצת סמיכויות)

<u>הגדרה:</u> **מטריצת שכנות** (גם **מטריצת סמיכויות** או **מטריצת שכנויות**) היא שיטה לייצוג **גרף מכוון** בעל n צמתים בעזרת מטריצה ריבועית בגודל n x n, שערכי תאיה הם 0 או 1. תא (i,j) בגרף מתאר את קיומה (או העדרה) של הקשת המכוונת מצומת i לצומת j. אם אין קשת כזו, הערך בתא במטריצה יהיה 0. אם יש קשת כזו, אזי הערך יהיה 1.

<u>הגדרה:</u> **מטריצת שכנויות** של **גרף לא מכוון** היא סימטרית. זאת משום שבגרף לא מכוון אין כיוון לקשתות ולכן, אם קיימת קשת מ-i ל-j אזי קיימת קשת מ-j ל-i. גרפים שאינם מכוונים נהוג לשמור רק את המשולש העליון של מטריצת השכנויות.



דוגמה למטריצת שכנות של גרף לא מכוון

- עבור גרף שבו אין לולאות עצמיות (כלומר, קשת מקודקוד כלשהו לעצמו), האלכסון הראשי של מטריצת השכנויות המתאימה יהיה 0.
- מטריצת השכנויות של גרף שלם תהיה המטריצה בה כל התאים בעלי הערך 1, פרט לאלכסון הראשי.
- מטריצת השכנויות של גרף חסר קשתות תהיה מטריצת האפס.

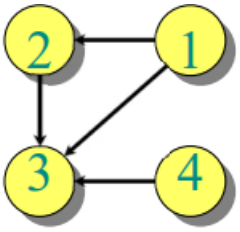
<u>תכונות:</u>

1- הוספה/מחיקת קשת מתבצעת בגישה מיידית, בסיבוכיות זמן של O(1)
2- ניתן לחשב את הגרף המשלים בקלות על ידי חיסור של מטריצת השכנות ממטריצת שכנות של גרף שלם. כך, בכל מקום שהיה בו קשת, כלומר ערך 1, יהיה 0 ובכל מקום שהיה בו 0, יהיה 1.
3- סיבוכיות מקום קטנה יחסית לשמירת המטריצה בזיכרון O(|V^2|)
4- חיפוש שכנים בסיבוכיות של O(1)

<u>מתי כדאי להשתמש במטריצת סמיכויות:</u> כאשר התדירות שבה אנו בודקים שכנות היא גבוהה

רשימת סמיכויות

<u>הגדרה:</u> רשימת סמיכויות גם היא שיטת ייצוג מקובלת לגרף כללי. בשיטה זו, הגרף מיוצג כקבוצה של רשימות מקושרות של השכנים של כל צומת.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

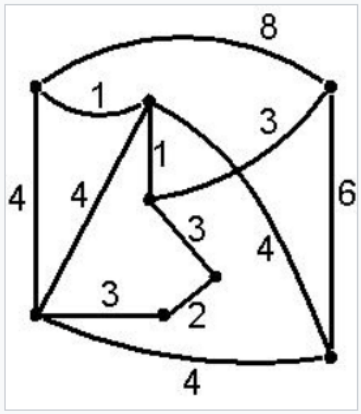
תכונות:

1- חיפוש שכנים בסיבוכיות של O(deg(v)) כאשר v זו הצומת עבורה אנו מחפשים שכנים.
2- סיבוכיות מקום קטנה יותר ממטריצת סמיכויות: O(|V|+|E|)

מתי כדאי להשתמש ברשימת סמיכויות: כאשר יש לנו גרף שהוא דליל: |E| = O(|V|)

גרף ממושקל

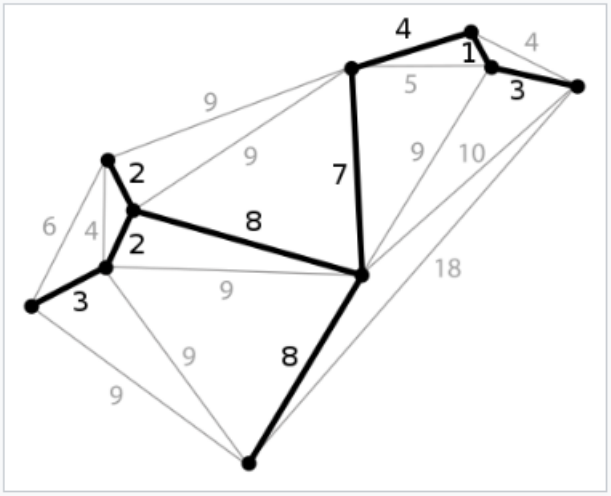
הגדרה: **גרף ממושקל** הוא גרף עבורו לכל קשת בגרף משויך "משקל" - לרוב מספר ממשי.
מבחינה פורמלית, זהו גרף G = (V,E) ופונקציית משקל w: E -> R (R = ממשיים).



דוגמה לגרף ממושקל. המספר הצמוד לכל צלע מסמן את משקלה

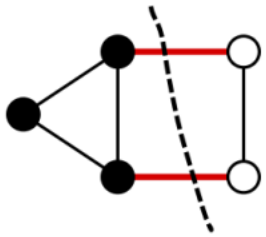
עץ פורש מינימלי

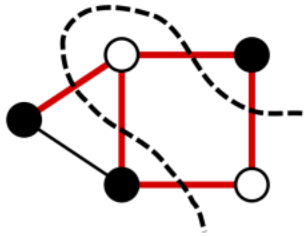
הגדרה: **עץ פורש מינימלי** של גרף הוא עץ פורש שהוא מינימלי בסכום משקלי הקשתות שלו מבין כל העצים הפורשים.



חתך

<u>הגדרה:</u> **חתך** (S,T) בגרף G = (V,E) זוהי חלוקה של V לשתי תתי קבוצות זרות לא ריקות S,T כך ש:
S מוכלת ב-V ואילו T = V – S.

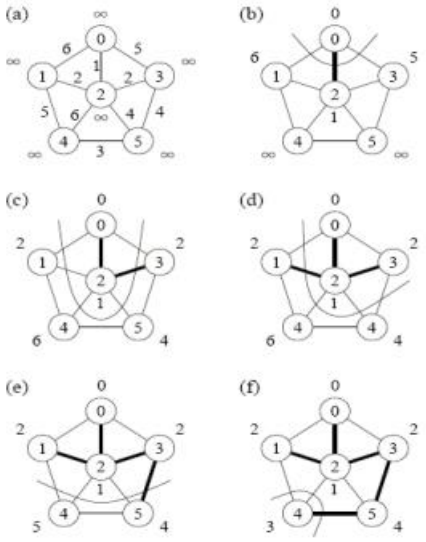




תכונת החתך: תהי A קבוצת צמתים לא ריקה המוכלת ב-V. ותהי e=(u,v) הצלע שחוצה את החתך (A, V-A), כלומר קצה אחד שלה ב-A והקצה השני ב-V-A וכמו כן משקלה של e המינימלי מבין כל הצלעות שחוצות את החתך. אז כל עץ פורש מינימלי כולל בתוכו את הצלע e.

האלגוריתם של פרים

<u>רעיון:</u> מתחילים מצומת יחיד ובכל שלב מוסיפים צלע הכי קלה שמחברת את העץ שבנינו עם המשלים שלו.



פסאודו קוד:

Robert Clay Prim 1921

Prim(G,w,u)

INPUT : $G = (V, E, w)$ – a **connected**, undirected, weighted graph and $u \in V$

OUTPUT : $T_{Prim} = (V_T, E_T)$ = minimum spanning tree

1. $V_T \leftarrow \{u\}$, $E_T \leftarrow \varnothing$
2. **While** $t < |V| - 1$
3. **Do** choose $(x,y) \in E$ of minimum weight $w(x,y)$ such that $x \in V_T$ and $y \in V - V_T$
   $V_T \leftarrow \{y\} \cup V_T$ & $E_T \leftarrow \{(x,y)\} \cup E_T$ & $t \leftarrow t + 1$

סיבוכיות זמן ריצה: O(|E|*log|V|)

איחוד קבוצות זרות

Union מכיל בתוכו 3 פעולות:

1- makeset(n) – יוצר קבוצה עם איבר בודד n.
2- find(n) – מוצא את הקבוצה ש-n שייך אליה.
3- union(a,b) – מאחד שתי קבוצות a ו-b.

האלגוריתם של קרוסקל

הרעיון: שימוש במבנה נתונים איחוד קבוצות זרות.

תיאור האלגוריתם:

- נחשיב כל צומת כקבוצה. כלומר מתחילים עם |V| קבוצות זרות.
- ממיינים את המשקל של הצלעות בסדר עולה.
- מאתחלים t = 0
- רצים בלולאה כל עוד t < |V| -1
- מתחילים מהצלע הכי קטנה (u,v) ובודקים האם u ו-v שייכות לאותה קבוצה (find(u) == find(v)). אם לא, מאחדים את הקבוצות (union(u,v)) וממשיכים ולצלע הבאה.
- מבצעים t++

האלגוריתם:

Kruskal(G,w)

1. **Sort E** as: $w(e_1) \le w(e_2) \le \ldots \le w(e_m)$; **O(E logE)**
2. $E_T \leftarrow \varnothing$, $t \leftarrow 0$ & for each $u \in V$ initialize a set $\{u\}$
3. **While** $\boxed{t < |V| - 1}$
4. **Do** choose next $(x,y) \in E$ of minimum weight
5. **if** find(x) $\neq$ find(y)
   then union(find(x), find(y))
   and $E_T \leftarrow \{(x,y)\} \cup E_T$
6. $t \leftarrow t + 1$

סיבוכיות זמן ריצה: O(|E|*log|V|)

אלגוריתם למציאת עפ"מ שני הכי טוב

הגדרה: **עפ"מ שני הכי טוב** זהו עפ"מ שאם נחליף צלע אחת בו בצלע אחרת שלא נמצאת בו כך שההפרש בין שני העצים הוא המינימלי מבין כל האפשרויות, אזי קיבלנו עפ"מ שני הכי טוב.

האלגוריתם:

Input : **G = (V,E,w)** undirected, connected, weighted graph
Output: best second tree

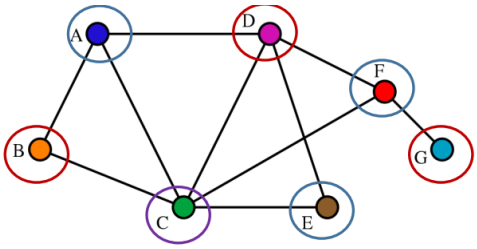
Call Kruskal(G,w) $\rightarrow$ $\mathbf{T_{min} = (V,E_{min})}$ and set $A = E - E_{min}$
Dif(T) = ∞
For every edge $(u,v) \in A = E - E_{min}$ do:
1. **Add (u,v)** to the tree $\mathbf{T_{min}}$, creating a cycle $\mathbf{C_{uv}}$
2. **Find** $(x,y) \in C_{uv} - (u,v)$ the **maximum weight edge**
3. **Compute** the **change** in the tree
   weight dif = w(u,v) – w(x,y)
4. **IF dif < Dif(T) then Dif(T) = dif** and
   $\mathbf{T = T_{min} + (u,v) - (x,y)}$

Return T as a best second spanning tree

הסבר לאלגוריתם: תחילה נמצא עפ"מ באמצעות קרוסקל. נגדיר משתנה Dif(T) שיכיל את המספר המקסימלי ביותר שאפשר והוא יהווה אינדיקציה להפרש המשקלים בין שני העצים.
לאחר מכן בלולאה על כל הצלעות שלא שייכות לעפ"מ שלנו, נחבר את הצלע לעץ ומהמעגל שנוצר נוריד את הצלע המקסימלית.
נשמור את המשקל של העץ החדש בdif ונבדוק האם dif קטן מDif(T). אם כן -> נבצע: Dif(T) = dif
ונשים בT את הערך של העץ החדש. ובסופו של דבר נחזיר את T.

0 – צביעת גרף

המחשת הבעיה:



בעיית חלוקת הצמתים:
למצוא חלוקת הצמתים למספר **מינימלי** של קבוצות **ביילתי תלויות**.

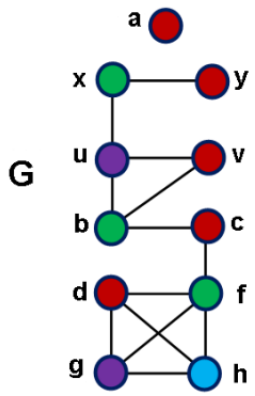
תיאור הבעיה:

נניח שהצמתים A,B,C,D,E,F,G הם הרצאות במכללה עם שעת התחלה ושעת סוף.
מתחת לגרף ניתן לראות ציר זמן שעליו פרושות ההרצאות.
הרצאות **מתיישבות** זו עם זו אלו הרצאות שהשעות שלהן אינן מתנגשות.
הגרף אינטרוולים מייצג את ההרצאות השונות. קשת היוצאת מצומת מהווה התנגשות עם הצומת האחרת (למשל, B מחובר ל-A ול-C, כלומר ההרצאה B מתנגשת עם הרצאות A ו-C ואכן ניתן לשים לב לכך בציר הזמן).
העיגולים הצבעוניים שעוקפים את הצמתים מהווים סימן מזהה של קבוצות בלתי תלויות:
הקבוצה האדומה: B, D ו-G
הקבוצה הכחולה: A, E ו-F
הקבוצה הסגולה: C
נשים לב כי קבוצות בלתי תלויות מהוות וריאציות שבהן ניתן לשבץ הרצאות שמתיישבות זו עם זו.

המטרה: למצוא את המספר המינימלי של חדרים שניתן להשתמש בהם על מנת שההרצאות יתיישבו זו עם זו (למצוא חלוקת הצמתים למספר מינימלי של קבוצות בלתי תלויות).

הגדרה: עבור גרף מכוון G = (V,E). **המספר הכרומטי** $\chi(G)$ מהווה מספר מינימלי של קבוצות בלתי תלויות.

למשל ערך המספר הכרומטי בגרף זה הינו 4:



תיאור האלגוריתם:

נעבור באיטרציה על כל צומת ונצבע את הצומת בצבע שונה מצבעי השכנים שלה.

האלגוריתם:

Greedy coloring (G, L):

// $L = v_1, v_2, \dots, v_n$

1. color[1] = 1
2. for i = 2 to n
   do color[i] ← 0
3. for i = 2 to n
   color[i] = min_color_not_used_by_neighbors(v_i)
4. return d = max(color[ ]) and color[ ]

min_color_not_used_by_neighbors(v)

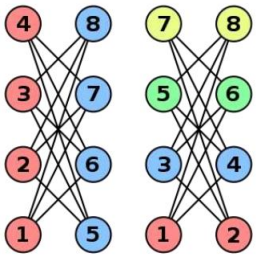
1. for i = 1 to deg(v)+1
   do used_color[i] ← 0
2. for each u∈Adj[v]
   if (1 ≤ color[u] ≤ deg(v))
   used_color[color[u]] = 1
3. for i = 1 to deg(v)+1
   if (used_color[i] == 0)
   return i

הצבע המינימאלי שלא נמצא בין deg(v) השכנים חייב להיות לא גדול מ- deg(v)+1

סיבוכיות O(deg(v))

סיבוכיות זמן ריצה: O(V+E)

- הבעיה הזו נחשבת בעיית NP קשה. וישנה משמעות לסדר שבו עוברים בלולאה על הצמתים. אם נריץ את אותו האלגוריתם על סדר שונה של צמתים עלולים לקבל תוצאות שונות:



אם הצמתים יסודרו ברשימה ע"פ המספר משמאל:
אז נקבל צביעה אופטימלית עם **2** צבעים

אם הצמתים יסודרו ברשימה ע"פ המספר מימין:
אז נקבל צביעה עם **4** צבעים

משפט

יהי **$G=(V,E)$ - גרף אינטרוולים**, אזי:

(א) $\omega(G) = \chi(G)$ = גודל הקליקה הגדולה ביותר ב- G

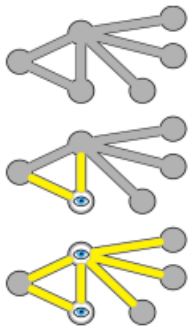
(ב) אם האלגוריתם סורק את הצמתים בסדר עולה של תחילת האינטרוולים אז נקבל צביעה **אופטימלית** ב- $\omega(G) = \chi(G)$ צבעים.

משפט: אם $G = (V,E)$ לא מכוון, אזי: $\chi(G) \geq 2 \Leftrightarrow E \neq \varnothing$.

משפט: אם $G = (V,E)$ לא מכוון ו-H תת גרף של G, אזי: $\chi(H) \leq \chi(G)$

0 – כיסוי קודקודים

<u>הגדרה:</u> בהינתן גרף G = (V,E) נאמר ש**כיסוי קודקודים** זוהי קבוצת קודקודים המוכלת ב-V אשר מכסה את כל הצלעות ב-E (כלומר, במילים אחרות, כיסוי צלעות בעזרת קודקודים).

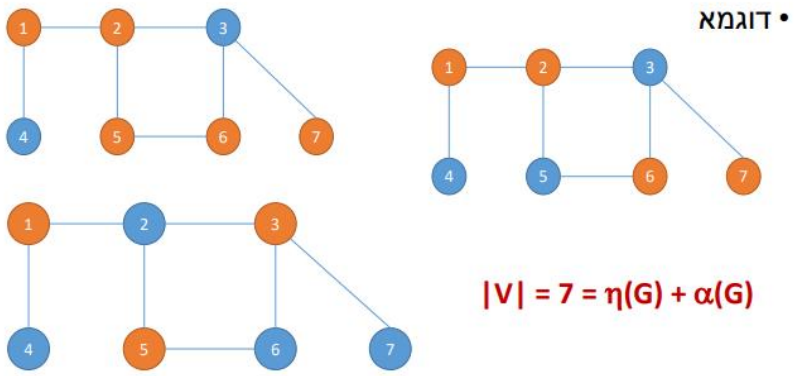


<u>הסבר להמחשה:</u> הקודקודים שעליהם מצוירת עין, הם בעצם מהווים כיסוי קודקודים.

<u>הגדרה:</u> אם A מהווה כיסוי קודקודים ב-G ולא קיים כיסוי B אשר מוכל ב-A, נאמר שכיסוי A הוא **כיסוי מינימלי**.

<u>הגדרה:</u> נסמן ב: $\eta(G)$ = גודל כיסוי מינימלי.

<u>הגדרה:</u> אם A כיסוי קודקודים וגם $\eta(G) = |A|$, אז A נקראת **כיסוי מינימום**.



<u>הסבר להמחשה:</u> הצבע הכתום מהווה כיסוי והצבע הכחול מהווה קודקוד מתוך קבוצה בלתי תלויה.

<u>משפט:</u> $|V| = \eta(G) + \alpha(G)$ מתקיים לכל G = (V,E).

<u>טענה:</u> יהי גרף לא מכוון G = (V,E) ועלה u. אז:

- כל קבוצה בלתי תלויה מקסימום של G מכילה את כל הצמתים של G בעלי דרגה 0 (צמתים מבודדים).
- קיימת קבוצה בלתי תלויה מקסימום שמכילה את u.

אלגוריתמי קירוב

<u>הרעיון:</u> למצוא פתרון פולינומיאלי אשר קרוב לפתרון האופטימלי לבעיות NP-קשות.

<u>אופן חישוב הקירוב:</u>

- נסמן את הפתרון האופטימלי ב-Opt
- נסמן את הפתרון של אלגוריתם קירוב ב-A
- אלגוריתם קירוב נקרא p(n)-approximation algorithm אם:

$$\boldsymbol{\rho(n)} \geq max(\frac{A}{Opt}, \frac{Opt}{A})$$

Minimization problem

Maximization problem

כלומר p(n)=1 זהו אלגוריתם שתמיד מוצא פתרון אופטימלי.

אלגוריתם לחישוב קבוצת כיסוי מקסימום

אלגוריתם זה הינו בעצם בעיית **1-Approzimation Algorithm**, כלומר תמיד מספק פתרון אופטימלי.

<u>הרעיון:</u> לפי המשפט שציינו למעלה, אם נמצא את קבוצת כיסוי מקסימום, הצמתים הנותרים שלא נכללים בקבוצה זו הם בעצם צמתים של כיסוי קודקודים מינימלי.

<u>האלגוריתם:</u>

Greedy_Indep_Set_Forest(F,S)

Input: F = (V,E) a forest

Output: S = a **maximum independent set**

1. **S** ← ∅ and **U** ← **E**
2. while **U** ≠ ∅ {
3. let **(u,v)** be an edge s.t. **v** is <u>a leaf</u>
4. **S** ← **S** ∪ **{v}**
5. remove **u, v** and all the edges incident to them
   }
6. add to **S** all remaining vertices
7. return **S**

<u>תיאור האלגוריתם:</u>

- נסמן ב-S את קבוצת כיסוי מקסימום שנחזיר ונאתחל אותה כקבוצה ריקה.
- נסמן ב-U את העתק של קבוצת הצלעות E.
- בלולאה: כל עוד U שונה מקבוצה ריקה
- נבחר עלה מתוך U ונסמן ב-v את העלה ואת u השכן שלו.
- נוסיף ל-S את העלה v ונמחק את u מקבוצת הקודקודים V ונמחק מ-U את כל הקשתות המקושרות ל-u ול-v

- אחרי יציאה מהלולאה, נוסיף ל-S את הקודקודים שנותרו ב-V.
- נחזיר את S.

<u>סיבוכיות זמן ריצה:</u> O(V+E)

אלגוריתם לחישוב כיסוי קודקודים

אלגוריתם זה הינו בעצם בעיית **2-Approzimation Algorithm.**

<u>האלגוריתם:</u>

Algorithm: ApproxVertexCover-3(G = (V,E))

C =∅ ; E'=E;

while E' ≠ ∅ /* Repeat until every edge is covered: */

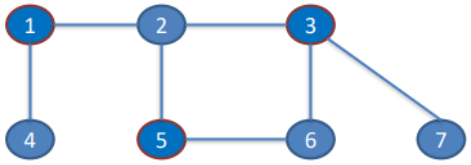
Let e = (u, v) be any edge in E.

C=C ∪ {u,v}

Remove u,v (from V) and all adjacent edges (from E').

return C

O(V+E)



<u>תיאור האלגוריתם:</u>

- נסמן ב-C את כיסוי הקודקודים ונאתחל כקבוצה ריקה.
- נסמן ב-E' את העתק קבוצת הצלעות E.
- בלולאה: כל עוד E' שונה מקבוצה ריקה.
- נבחר עלה (u,v) כלשהו ונסמן ב-e. בעצם u מהווה קודקוד עלה ו-v זה השכן שאליו מחובר.
- נוסיף ל-C את העלה ואת השכן שלו (כלומר את e).
- נמחק מ-V את u,v ואת כל הקשתות שהם מחוברים אליהן מתוך E'.
- נחזיר את C כאשר נצא מהלולאה.

<u>סיבוכיות זמן ריצה:</u> O(V+E)

<u>הסבר למדוע p(n) = 2:</u> נמחיש על הגרף שמצורף לאלגוריתם שלנו למעלה.

- נשים לב כי Opt = 3 כי כיסוי קודקודים מינימלי בגרף הינו {1,5,3}
- אם נפעיל את האלגוריתם שלנו ונבחר את העלים לפי הסדר הבא:
  1 – (7,3)
  2 – (4,1)
  3 – (2,5)
  נגיע ל: A = 6
- נחשב: p(n) >= max(3/6, 6/3) = 2

1 – אלגוריתם סריקת גרף לרוחב - BFS

Breadth **F**irst **S**earch – BFS

שימושים של BFS

- לקבוע האם גרף קשיר.
- לבדוק האם צלע היא גשר.
- לבדוק האם קיים מעגל אי זוגי/או מעגל כלל.
- האם הגרף דו-צדדי.
- האם צומת היא שורש.

הרעיון

נרצה לבצע סריקה שיטתית של צלעות וקודקודי הגרף.
סריקת "רוחב": נבחר צומת התחלתית s. עוברים על כל צומת במרחק 1 מצומת s, ואז על כל צומת במרחק 2, מרחק 3 וכן הלאה..

תיאור האלגוריתם

<u>מבנה נתונים עזר:</u>

- **תור** לצורך שמירת הסדר סריקה של הגרף.
- **צבעים color[]** כדי לסמן את הסטטוס של כל קודקוד:

לבן (white) ← אפור (gray) ← שחור (black)

טרם התגלה← נמצא עכשיו בתור ← יצא מהתור, הטיפול נגמר

- **π[]** מכיל מידע אודות האב של כל קודקוד (למשל, אם עבור קודקוד b האב שלו הוא a אזי π[b]=a).
- **d[]** מכיל מידע אודות המרחק של כל קודקוד מקודקוד התחלה (למשל, אם קודקוד b הוא הבן של a שהוא הבן של s קודקוד ההתחלה, אזי d[b]=2)

<u>קלט:</u> גרף G שמיוצג כרשימת סמיכויות (סיבוכיות מקום O(V+E) וצומת התחלה s.

<u>פלט:</u>

- מערך מרחקים d[] שנותן את ערך המרחק של הצומת מצומת התחלה
- מערך π[] אשר נותן מידע אודות אב של צומת
- עץ BFS: Tbfs = (Vbfs, Ebfs). קבוצה Vbfs מכילה את כל הצמתים של G שעבורם color[v] = black, ואילו הקבוצה Ebfs מכילה את כל הקשתות מהצורה: (π[v], v).

<u>פירוט הצעדים:</u>

- תחילה, נרוץ בלולאה על כל הקודקודים ונאתחל אותם.
  עבור כל קודקוד v נאתחל:
  1 – color[v] = white
  2 – π[v] = Null
  3 – d[v] = ∞
- נאתחל את התור ונוסיף לו את איבר ההתחלה: enqueue(Q,s)
- בלולאה: כל עוד Q לא ריק נבצע:
- נשלוף קודקוד מראש התור: u = dequeue(Q)
- בלולאה פנימית נעבור על כל השכנים של u לפי מטריצת סמיכויות: for each v in Adj[u]

- אם הצבע של v לבן, נבצע:
  color[v] = grey – 1
  π[v] = u – 2
  d[v] = d[u] + 1 – 3
  enqueue(Q,v) – 4
- ברגע שעברנו על כל השכנים, נצבע את הקודקוד הנוכחי בשחור כי סיימנו לעבור עליו:
  color[u] = black

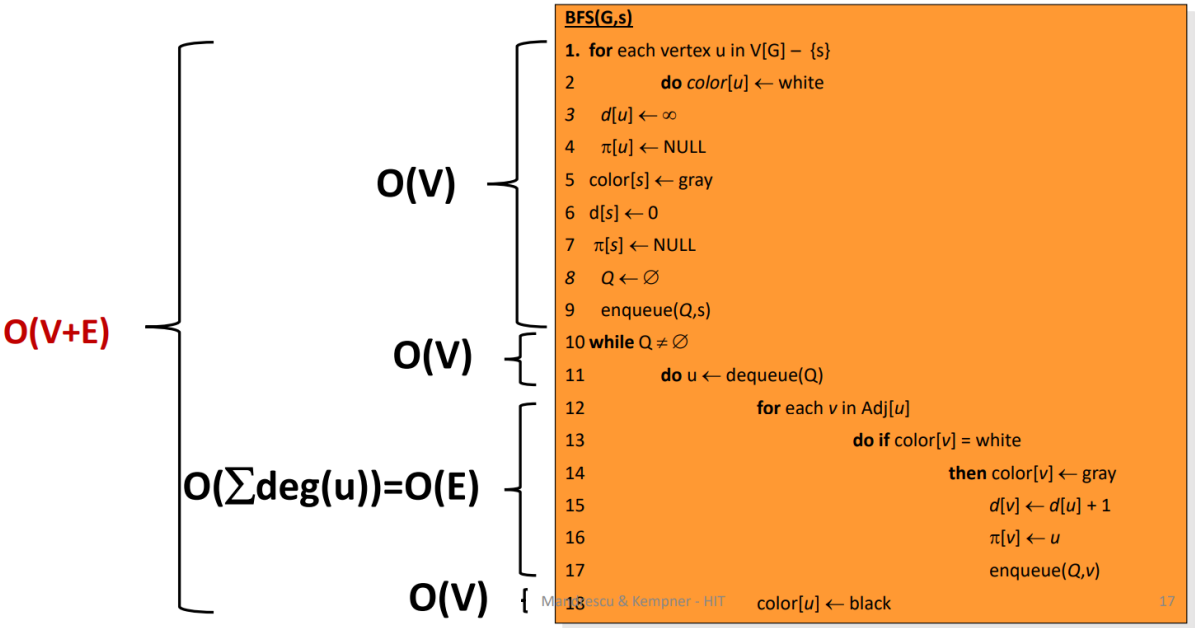
<u>האלגוריתם:</u>

```
BFS(G,s)
for each vertex u in V[G] – {s}
        do color[u] ← white
  d[u] ← ∞
  π[u] ← NULL
color[s] ← gray
d[s] ← 0
π[s] ← NULL
Q ← ∅
enqueue(Q,s)
while Q ≠ ∅
        do u ← dequeue(Q)
                for each v in Adj[u]
                        do if color[v] = white
                                then color[v] ← gray
                                        d[v] ← d[u] + 1
                                        π[v] ← u
                                        enqueue(Q,v)
                color[u] ← black
```

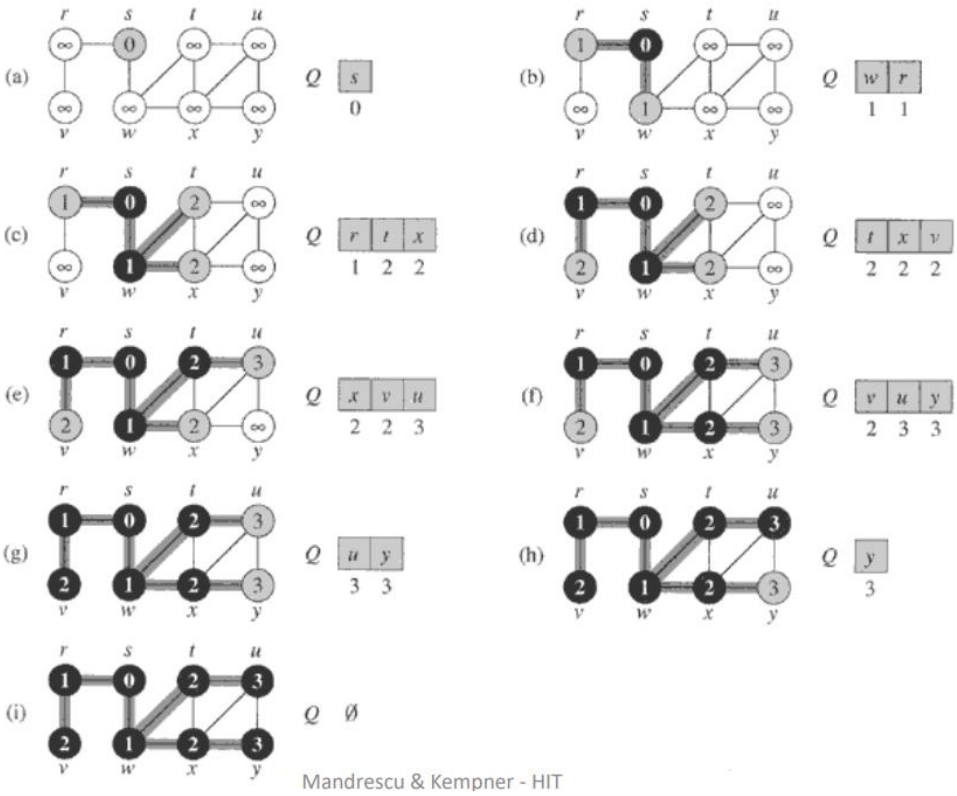
Input: G = (V,E) - (un)directed graph and $s \in V$

Output: d[ ], π[ ] ($T_{BFS} = (V_{BFS}, E_{BFS})$), color[ ]

<u>סיבוכיות זמן ריצה:</u>



המחשה:



הערות

לאחר הרצת אלגוריתם BFS על גרף G = (V,E) לא מכוון ניתן לומר:

- כל צמתי הגרף G נצבעו בשחור ⇔ d[v] != ∞ לכל v ⇔ **G קשיר**.
- צומת v כלשהו נשאר צבוע לבן ⇔ d[v] = ∞ ⇔ אין מסלול ב-G מ-s ל-v ⇔ **G אינו קשיר**.

לאחר הרצת אלגוריתם BFS על גרף G = (V,E) מכוון ניתן לומר:

- כל צמתי הגרף נצבעו בשחור ⇔ s הוא **שורש**.
  שורש = יש מסלול ממנו לכל צומת בגרף.
- צומת v כלשהו נשאר צבוע לבן ⇔ d[v] = ∞ ⇔ אין מסלול מכוון ב-G מ-s ל-v (s אינו שורש).

משפטים

<u>משפט:</u> לאחר הרצת BFS(G,s) מתקיים: d[v] = dist(v,s) לכל v בגרף G מכוון וגם לא מכוון.

<u>משפט:</u> לאחר הרצת BFS(G,s) מתקיים:

- אם G לא מכוון, אז $T_{BFS} = (V_{BFS}, E_{BFS})$ הוא עץ פורש של רכיב הקשירות של s בגרף G.
- אם G מכוון, אז $T_{BFS} = (V_{BFS}, E_{BFS})$ הוא עץ מכוון בעל שורש ב-s.

<u>משפט:</u> לכל צלע בגרף (u,v) מתקיים dist(s,v) <= dist(s,u) + 1.

<u>מסקנה:</u> לאחר הרצת BFS(G,s) לכל צלע בגרף (u,v) מתקיים: |d[u]-d[v]| <= 1

אלגוריתם למציאת המסלול הקצר ביותר

אם לאחר הרצת BFS(G,s) הצומת v נצבע בשחור, אז ב T_{BFS} קיים מסלול יחיד מ-s ל-v. זהו **מסלול הקצר ביותר** ב-G מ-s ל-v, אותו ניתן למצוא על ידי האלגוריתם הרקורסיבי הבא:

.

- **Print-Path(G,s,v)** ↔ **O(V)**

```
if v = s
 then print  s
 else  if  π[v] = NULL
        then  print "no path exists from  s  to  v
        else   Print-Path(G, s, π[v])
               print  v
```

<u>הערה:</u> יתכן שב-G קיימים עוד מסלולים קצרים ביותר מ-s ל-v.

אלגוריתם למציאת גרף דו-צדדי

<u>משפט:</u> יהי G = (V,E) גרף לא מכוון, קשיר ו-E לא קבוצה ריקה. אזי:
G הוא גרף דו-צדדי ⇔ לא קיימת צלע (u,v) כך ש: d[u] = d[v] כאשר d[x] = dist(s,x) לכל x השייך ל-V, לאחר הרצת BFS(G,s).

<u>באמצעות שימוש במשפט ניצור אלגוריתם למציאת גרף דו-צדדי:</u>

```
select  s∈V
call  BFS(G,s)                          O(V+E)
for each  (u,v)∈E                       O(E)
   if d[u] = d[v]
      then return "G ≠ bipartite"
return  "G = bipartite"
```

<u>סיבוכיות האלגוריתם:</u> מכיוון שהגרף קשיר אזי: |E| >= |V| - 1 >= |V|/2
ולכן: O(E).

אלגוריתם לחישוב קוטר בגרף

אלגוריתם לחישוב קוטר בכל גרף:

```
Algorithm_Diam(G)
diam = 0
for each s∈V {
   call BFS(G, s)
    last_vertex ← y  (where d[y] = max{d[v] : v∈V})
    if  diam < d[last_vertex]
          then  diam ←  d[last_vertex]
                }
return  diam(G) = diam
```

last_vertex=הצומת האחרון ב: Q

d[last_vertex] = max{d[v] : v∈V}

$O(|V|\cdot(|E|+|V|))= O(|V|\cdot(|E|)$

גרף קשיר

אלגוריתם לחישוב קוטר בעץ בלבד:

משפט: יהי T = (V,E) עץ ו-s צומת כלשהו בגרף. יהיה a צומת בגרף כך ש: **dist(s, a) = max{dist(s, v) : v∈V}**. אזי:

- a הוא עלה של T.
- קיים צומת b כך ש: dist(a,b)=diam(T)

נשתמש במשפט זה ונבנה את האלגוריתם:

```
Diam_Tree(T)
select s∈V
call BFS(G, s)
a ← last_vertex
call  BFS(T,a)
diam(T) = d[last_vertex]
```

last_vertex=הצומת האחרון ב: Q

d[last_vertex] = max{d[v] : v∈V}

$O(|E|+|V|) = O(|E|) = O(|V|)$

גרף הוא עץ

$|V| \geq |E| = |V| - 1 \geq |V|/2$

2 – אלגוריתם סריקת גרף לעומק - DFS

Depth **F**irst **S**earch – **DFS**

הרעיון

הסריקה מתבצעת ל**"עומק".** כלומר מתחילים מצומת מסוים ויורדים לכל עומקה בשונה מאלגוריתם BFS שבו עשינו סריקה ל"רוחב" והתחלנו מצומת מסוים ועברנו תחילה על כל שכניו ואז עברנו להבאים בתור.

במהלך הסריקה לעומק נעזרים ב**מחסנית**(רקורסיה) לביצוע הסריקה.

תיאור האלגוריתם

<u>מבני נתונים עזר:</u>

- טיימר – משתנה counter שמאותחל ב-0 ובכל שלב עולה ערכו ב-1
- מערך צבעים []color שעוזר לנו לסמן צמתים שאותם סרקנו, צמתים בתהליך סריקה וצמתים שעוד לא סרקנו. בדומה ל-BFS: לבן – עוד לא נכנס לסריקה, אפור – בסריקה, שחור – עבר סריקה.

<u>קלט:</u> גרף G = (V,E) מכוון או לא מכוון שמיוצג כרשימת סמיכויות (סיבוכיות מקום O(V+E)).

<u>פלט:</u>

- מערך []π שמכיל מידע על האב של כל צומת בגרף.
- מערך []b שמכיל את זמן ההתחלה שבו עברנו על הצומת.
- מערך []f שמכיל את זמן הסיום שבו סיימנו לסרוק את הצומת ואת בניה.
- יער DFS: Tdfs = (Vdfs, Edfs). קבוצה Vdfs מכילה את כל הצמתים של G ואילו הקבוצה Edfs מכילה את כל הקשתות מהצורה: (v ,[v]π).

<u>פירוט הצעדים:</u>

- תחילה, נרוץ בלולאה על כל הקודקודים ונאתחל אותם.
  עבור כל קודקוד v נאתחל:
  1 – color[v] = white
  2 – π[v] = Null
- נאתחל את הטיימר: time = 0
- בלולאה על כל הצמתים של G, נקרא לפונקציה DFS-Visit(v)
- בתוך הפונקציה החדשה, נעדכן את צבע של v לאפור, נקדם את time ב-1 ונשים במערך b[v] = time.
- בלולאה על כל השכנים של v נבצע: אם הצבע של השכן u לבן, נצבע אותו באפור, נעדכן את המערך π[u] = v ונקרא רקורסיבית ל-DFS-Visit(u).
- ברגע שסיימנו לעבור רקורסיבית על כל השכנים, נצבע את v בשחור, נקדם את time ב-1 ונשים במערך f[v]=time.

האלגוריתם:

DFS

Input: G = (V,E) - (un)directed graph

Output: Forest $T_{DFS} = (V_{BFS}, E_{BFS})$,

π[ ],b[ ], f[ ]

DFS(*G*)

```
for each vertex u ∈ V[G]
    do color[u] ← white
       π[u] ← NULL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = white
          then DFS-Visit(u)
```

DFS-Visit(*u*)

```
color[u] ← GRAY
time ← time + 1
b[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

b[u]– זמן גילוי **f[u]** – זמן נסיגה (סיום טיפול)

סיבוכיות זמן ריצה:

DFS(*G*)

```
for each vertex u ∈ V[G]
    do color[u] ← white
       π[u] ← NULL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = white
          then DFS-Visit(u)
```

O(V)

DFS-Visit(*u*)

```
color[u] ← GRAY
time ← time + 1
b[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
          then π[v] ← u
               DFS-Visit(v)
color[u] ← BLACK
time ← time + 1
f[u] ← time
```

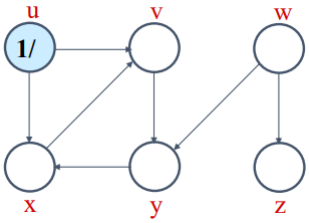
O(V)

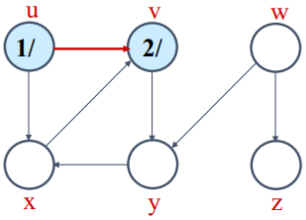
O(Σdeg(v)) = O(E)

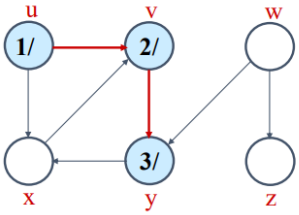
O(V)

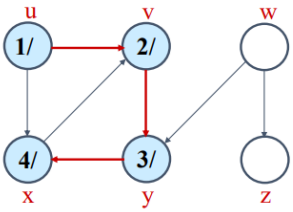
סיבוכיות כוללת: O(V+E)

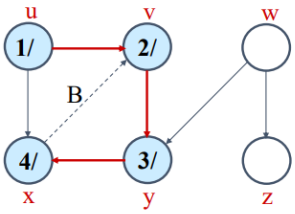
המחשה (משמאל לימין, למעלה למטה):

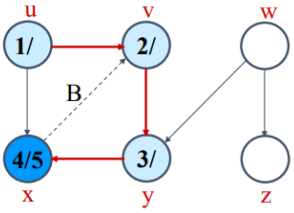


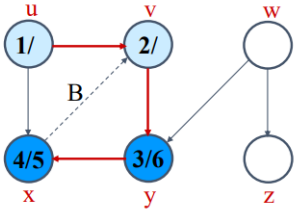


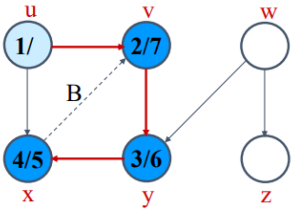


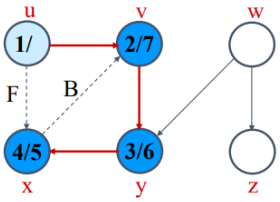


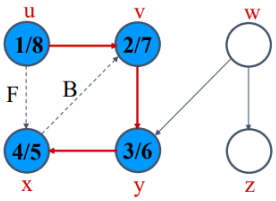


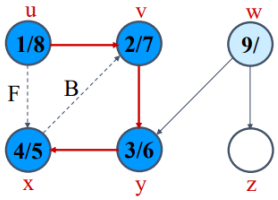


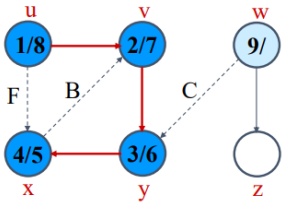


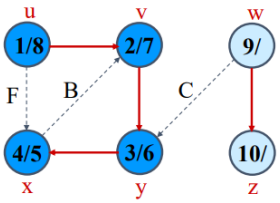


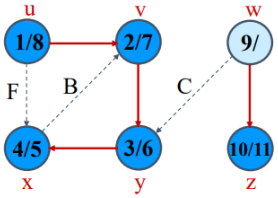


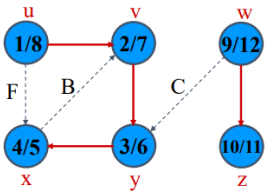












הערות

אחרי הרצת אלגוריתם DFS בגרף G מכוון (או לא מכוון) ניתן לומר:

- כל צומת נצבע בשחור: V = Vdfs.
- בגרף G לא מכוון, מספר הצמתים עם NIL שווה למספר רכיבי קשירות בגרף.
- בכל רגע בהרצה, הצמתים האפורים הם מסלול ב-G.

<u>הערה לגבי פרמטרים של האלגוריתם:</u> ניתן בנוסף לגרף להביא גם צומת ספציפי ממנו נרצה להתחיל.

<u>סיווג קשתות:</u> הסיווג של הצלעות תלוי בסדר הסריקה של ה-DFS.

<u>הרצת DFS פעמיים על אותו הגרף:</u>

- האם יש אותו מספר צלעות? <u>לא מכוון:</u> כן, <u>מכוון:</u> לא.
- האם יש אותו מספר עלים? <u>לא מכוון:</u> לא, <u>מכוון:</u> כן.

משפטים

<u>משפט עצים פורשים ביער העומק:</u> אחרי הרצת DFS על גרף לא מכוון G נוצר יער עומק Tdfs כך שכל עץ בתוכו הוא עץ פורש של רכיבי קשירות של G (כלומר עבור כל רכיב קשירות יש ב-Tdfs עץ פורש עבורו).

<u>הגדרה:</u> אם u מתגלה בזמן שהצומת v צבוע באפור, אזי u הוא **צאצא** של v ו-v נקרא **אב קדמון** של u.

<u>משפט הסוגריים:</u> לכל u ו-v ב-V מתקיים בדיוק 1 מתוך שלושת המקרים הבאים:

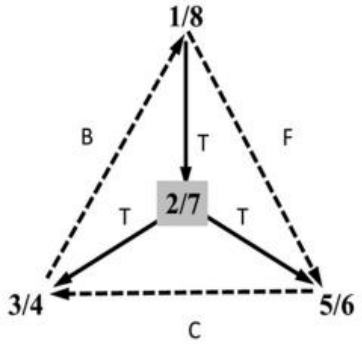
1- [[b[u],f[u]] איחוד [b[v],f[v]] שווה לקבוצה ריקה. כלומר אף אחד משני הצמתים אינו צאצא אחד של השני.
2- [[b[u],f[u]] מוכל בתוך [b[v],f[v]]. כלומר u צאצא של v.
3- [b[v],f[v]] מוכל בתוך [[b[u],f[u]]. כלומר v צאצא של u.

<u>משפט המסלול הלבן:</u> צומת v הוא צאצא של צומת u ב-Tdfs של G = (V,E) (מכוון או לא מכוון) אם ורק אם בזמן b[u] שבו מתגלה u קיים מסלול בגרף מ-u ל-v שמורכב כולו מצמתים לבנים.

סיווג קשתות

<u>סיווג קשתות בגרף מכוון:</u> יהיה G גרף מכוון, Tdfs יער העומק שנוצר וקשת e = (u,v) מתוך הגרף.

- הקשת e = (u,v) היא **קשת עץ** (**T**ree edge) אם היא שייכת ל-Tdfs.
- הקשת e = (u,v) היא **קשת קדמית** (**F**orward edge) אם היא אינה שייכת ל-Tdfs וגם v הוא צאצא של u.
- הקשת e = (u,v) היא **קשת אחורית** (**B**ack edge) אם היא אינה שייכת ל-Tdfs וגם v הוא אב קדמון של u.
- הקשת e = (u,v) היא **קשת חוצה(או קשת הצידה)** (**C**ross edge) אם היא אינה שייכת ל-Tdfs וגם v אינו צאצא של u וגם u אינו צאצא של v.



<u>אפיון קשתות על פי צבעים במהלך ריצת ה-DFS:</u>

ניתן לסווג כל קשת e = (u,v) בגרף G על פי צבעיו של הצמתים כאשר הקשת נבדקת לראשונה במהלך ריצת ה-DFS באופן הבא:

קשת **עץ**: מאפור ללבן. (b[u]<b[v]<f[v]<f[u])

קשת **אחורית**: מאפור לאפור. (b[v]<b[u]<**f[u]<f[v]**)

קשת **קדמית**: מאפור לשחור. (b[u]<b[v]<f[v]<f[u])

קשת **חוצה**: מאפור לשחור. (b[v]<f[v]<b[u]<f[u])

<u>סיווג קשתות בגרף לא מכוון:</u> בגרף לא מכוון יש רק קשת עץ וקשת אחורית.

אלגוריתם למציאת מסלול

DFS, מציאת מסלול

אם **G = (V,E)** **לא מכוון**, אז **אחרי** הרצת האלגוריתם **DFS(G)** מצומת הראשון **s** ב-T_{DFS} קיים <u>מסלול יחיד</u> מ- **s** לכל צומת **v** הנמצא ברכיב קשירות של **s** שניתן למצוא על ידי האלגוריתם הרקורסיבי הבא (מסלול זה אינו **בהכרח** מסלול **קצר ביותר** שיש ב-**G** מ-**s** ל-**v** ויתכן גם שב-**G** קיימים עוד מסלולים):

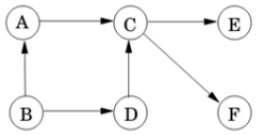
- **Print-Path(G,s,v)** ↔ **O(V)**

```
if v = s
 then print  s
 else  if  π[v] = NULL
        then  print "no path exists from  s  to  v"
        else   Print-Path(G, s, π[v])
               print  v
```

<u>איך בודקים ב-DFS אם יש מסלול בין u ל-v?</u> ניתן לבדוק אם u צאצא של v.

גרף א-ציקלי

Directed **A**cyclic **G**raph - **DAG**



זהו גרף מכוון חסר מעגלים (גמ"ל) ונקרא גרף א-ציקלי.

<u>אלגוריתם להכרעת אי-צקליות של גרף:</u> הרץ DFS, אם במהלך הריצה התגלתה קשת אחורית החזר "G הוא DAG" ואחרת "G אינו DAG".
<u>סיבוכיות:</u> O(V+E).

נכונות האלגוריתם נובעת מ-2 טענות:

<u>טענה ראשונה:</u> אם G אינו DAG אז **בהכרח** בכל הרצת DFS תתקבל קשת אחורית.

טענה שניה: אם G אכן DAG אז **בהכרח** בכל הרצת DFS לא תתקבל שום קשת אחורית.

מיון טופולוגי – אלגוריתם TOP_SORT

מיון טופולוגי של G היא רשימה של כל הצמתים ב-G כך שאם (u,v) שייכת ל-E, אז u מופיעה לפני v ברשימה.

לכן, ניתן לצייר את G בצורה "ליניארית": כל הצמתים בקו אופקי והכיוון של כל צלע הוא משמאל לימין.

אלגוריתם למיון טופולוגי TOP_SORT:

תיאור האלגוריתם:

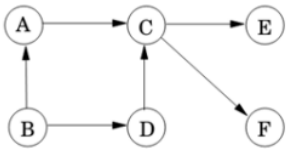
נקרא לאלגוריתם DFS וניצור את הרשימה הטופולוגית כך שהקודקודים יהיו ממוינים לפי זמן סוף בסדר יורד.

מיון טופולוגי

TOPOLOGICAL_SORT(G)

Input: a directed acyclic graph G = (V, E)

Output: L = a listing of vertices



1. call DFS( ) to compute **f**[**u**] for each vertex **u**

2. when a vertex **u** is **finished** (**f**[**u**] is computed),

insert it onto the **front of a linked list L**

3. return the linked list of vertices L.

סיבוכיות האלגוריתם: **O(|E|+|V|).**

משפטים:

משפט ראשון: אלגוריתם זה מחשב מיון טופולוגי לכל G גמ"ל.

משפט שני: יהי G גרף מכוון. אזי ל-G יש מיון טופולוגי ⇔ G חסר מעגלים מכוונים.

כלומר, מיון טופולוגי לא יעבוד עבור G מכוון עם מעגלים.

הערות:

- אם צומת a קדם לצומת b, הצומת b לא חייב להגיע ממש אחרי a, אבל כן צריך להיות אחרי a.
  למשל: a->b c
  המיונים הטופולוגיים האפשריים לגרף זה:
  1- a,b,c
  2- a,c,b
  3- c,a,b

מיון טופולוגי דוגמה:

צומת	מספר קורס	שם הקורס	דרישות קדם
v_1	M13 .1	מתמטיקה II	M12
v_2	M14 .2	מתמטיקה III	M13
v_3	M20 .3	מתמטיקה IV	M14, Phys
v_4	M12 .4	מתמטיקה I	אין
v_5	CS15 .5	מדעי המחשב II	M14 CS10,
v_6	Phys .6	פיסיקה	אין
v_7	CS10 .7	מדעי המחשב I	M12
v_8	CS20 .8	מדעי המחשב III	CS10, CS15, ENG
v_9	CS24 .9	מדעי המחשב IV	CS15,M20, CS20, Phys
v_{10}	ENG .10	אנגלית III	אין

שאלה: מהו סדר הקורסים המאפשר למלא את כל הדרישות קדם?

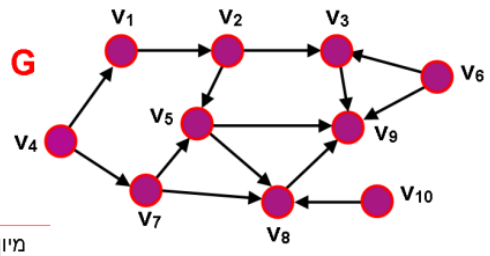
פתרון:

נגדיר גרף מכוון $G = (V,E)$ לפי הכללים:

$V = \{v_i : 1 \le i \le n\}$ כאשר n = מספר הקורסים

ו: v_i הוא צומת שמותאם לקורס i

$(v_i , v_j) \in E$ $\Leftrightarrow$ הקורס שמותאם לצומת v_i הוא קורס קדם לקורס שמותאם לצומת v_j



מיון טופולוגי של G $\Leftarrow$ סדר אפשרי שמקיים התנאי

כלומר אנו יוצרים סדר טופולוגי שכל "מאורע" (במקרה שלנו, קורסים) יהיו מסודרים בסדר הגיוני. בדוגמה שלנו, אנו נסדר את הגרף כך שמקורסי קדם תצא קשת לקורסים התלויים בקורסי הקדם.

ניעזר באלגוריתם זה כאשר נרצה ליצור סדר מסוים, כמו סידור לו"ז בין דברים תלויים אחד בשני. כמו כן במצבים שבהם ישנו סדר פעולות מסוים שישנן פעולות שתלויות אחת בשנייה.

אלגוריתם לבדיקה האם G הינו קשיר חזק

ניעזר במשפט:

משפט: יהי G גרף מכוון, אזי כל הטענות שקולות זו לזו:

- G הוא קשיר חזק.
- כל צומת של G היא שורש של G.
- G^t הוא קשיר חזק.
- כל צומת של G^t היא שורש של G^t.
- קיים צומת v כך שהינו שורש של G וגם של G^t.

אלגוריתם שבודק האם **G = (V,E) קשיר חזק**.

Is_strongly_connected(G)

select $s \in V$

call BFS(G,s) (DFS_VISIT(G,s))

if $|V| \neq |V_{BFS}|$ (if No_of_NIL >1)

then return "G is not strongly connected" // **s** is not a root of G

call BFS(G^T,s) (DFS_VISIT(G^T,s))

if $|V| \neq |V_{BFS}|$ (if No_of_NIL >1)

then return "G ≠ strongly connected (but **s** is a root of G)"

else return "G is strongly connected"

O(|E|+|V|)

הערה: No_of_NIL = מס הצמתים עם π=NIL אחרי DFS

נשים לב שכל התתי משפטים באים לידי ביטוי. בחרנו צומת אקראי ובדקנו אם G וגם ה-G מוחלף שלו והראינו שצומת זה הוא שורש בשניהם.

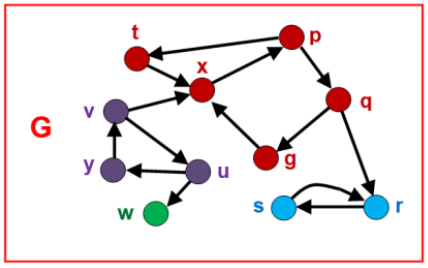
רכיב קשירות חזק (רק"ח)

Strongly **C**onnected **C**omponent - **SCC**

<u>משפט:</u> יהי G = (V,E) גרף מכוון. היחס R המוכל ב- V x V מוגדר לפי הכלל: uRv ⇔ u = v או קיימים מסלולים מכוונים מ-u ל-v וגם מ-v ל-u הוא יחס שקילות מעל V.

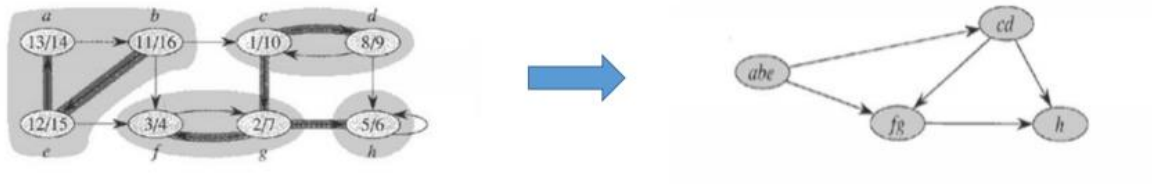
<u>הגדרה:</u> אם G גרף מכוון אז כל מחלקת שקילות V/R נקראת **רכיב קשירות חזק(רק"ח)** של G.

במילים אחרות, רכיב קשירות חזק זה רכיב שבו קיים מסלול מכל צומת ברכיב אל שאר הצמתים ולהפך.



גרף העל (גרף רק"חים)

<u>הגדרה:</u> גרף העל(או גרף רק"חים) של גרף מכוון G = (V,E) ומסומן Gscc = (Vscc, Escc) הוא גרף אשר Vscc מכיל בתוכו צומת אחד עבור כל רק"ח ב-G וכמו כן כל זוג רק"חים שמחוברים בלפחות קשת אחת ב-E, אזי Escc מכיל לכל מקרה כזה קשת אחד יחידה שמחברת בין צמתי Vscc.



<u>משפט:</u> יהי G = (V,E) גרף מכוון וגם H1 = (V1,E1), H2 = (V2, H2) רק"חים של G.

אז מתקיימים:

- V1 חיתוך V2 שווה לקבוצה ריקה.
- אם קיימת צלע (u,v) ב-E כך ש-u שייך ל-V1 וגם v שייך ל-V2 אז לא קיימת אף צלע (a,b) ב-E כך ש-a שייך ל-V2 ואילו b שייך ל-V1.
- אם קיים מסלול מכוון Puv כך ש-u שייך ל-V1 ו-v שייך ל-V2 אז לא קיים אף מסלול מכוון Pab כך ש-a שייך ל-V2 ו-b שייך ל-V1.

<u>מסקנה:</u> גרף העל Gscc הוא גרף א-ציקלי.

<u>הגדרה:</u> תהי S קבוצת צמתים. נגדיר: $f[S] = \max_{v \in S} f[v]$.

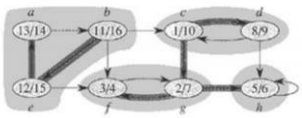
<u>משפט:</u> יהי G = (V,E) גרף מכוון וגם H1 = (V1,E1), H2 = (V2, H2) רק"חים של G.
אם קיימת צלע (u,v) ב-E כך ש-u שייך ל-V1 וגם v שייך ל-V2 אז אחרי ביצוע DFS(G) מתקיים: f[H1] > f[H2].

<u>מסקנה:</u> אם קיימת צלע (v,u) ב-E^t כך ש-u שייך ל-V1 וגם v שייך ל-V2 אז אחרי ביצוע DFS(G) מתקיים: f[H1] < f[H2]. (כלומר כל קשת שיוצאת מרק"ח ב-G^t מובילה לרק"ח עם זמן נסיגה גדול יותר ב-G).

<u>הערה:</u> אם יש צלעות בין שני רק"חים אז הצלעות חייבות להיות באותו כיוון כי אחרת לא באמת יש לנו שני רק"חים אלא רק"ח אחד שמרכיב את שניהם.

אלגוריתם SCC

STRONGLY_CONNECTED_COMPONENTS(G)
Kosaraju-Sharir's algorithm 1978,1981

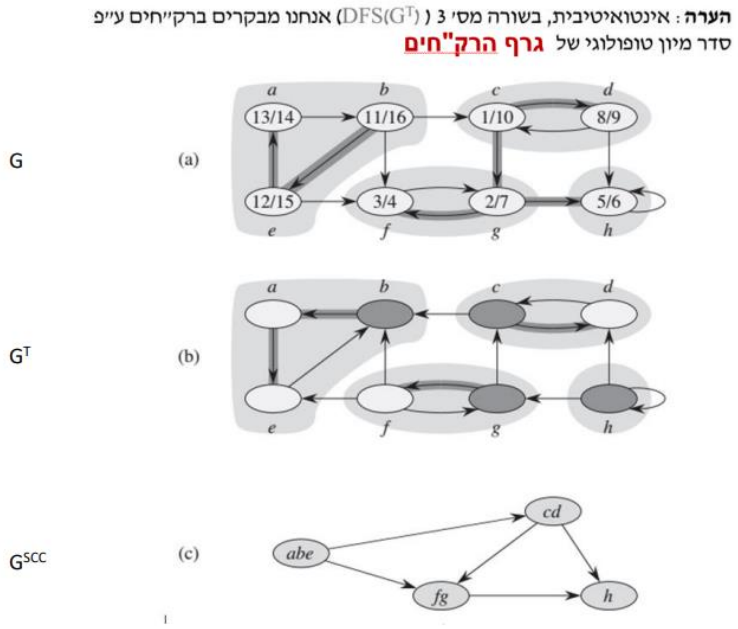


Input: G a directed graph
Output: its S.C.C.

1. **call DFS(G)** to compute **f[ ]**
2. **compute** the transpose G^T of **G**
3. **call DFS(G^T)**, but in the main loop of **DFS**, consider the vertices in **order of decreasing f[ ]**
4. **output** the vertices of each tree in the **DFS(G^T)** forest of step **3** as a strongly connected component.

O(|E|+|V|)

הסבר של האלגוריתם: בעצם מריצים בפעם הראשונה DFS על G ובפעם השנייה מריצים אותו על G^t ומאחורי הקלעים גורמים לסדר של הסריקה להיות לפי סדר יורד של זמני הסיום של הצמתים מהרצה הראשונה. ובעצם כך אנו מקבלים את SCC.



הגדרה: יהי G = (V,E) גרף לא מכוון. הגרף המכוון G* = (V,E*) מתקבל מהוספת כיוון לכל צלע ב-G ונקרא **המכוונת של G**.

משפט רובינס: יהי G גרף לא מכוון. אז ל-G יש מכוונת G* קשיר חזק ⇔ G קשיר ללא גשרים.

אלגוריתם STRONG

אלגוריתם למציאת מכוונת G* קשיר חזק:

Strong(G, G*)

Input: G = (V, E) an undirected, connected and **bridgeless** graph

Output: G* = (V, E*) a strongly connected graph

Step 1. Carry out a DFS**(G)** (DFS_Visit(G,s))

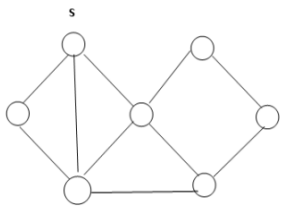
Step 2. For each edge e of G,

if **e** is a tree edge (in DFS tree)

orient it away from the tree root,

and if **e** is a back edge,

orient it from the descendant to the ancestor



3 – אלגוריתם למציאת מרחקים ממושקלים חיוביים - דייקסטרה

אלגוריתם זה עוסק בגרפים ממושקלים, מכוונים או לא.

הרעיון

הסריקה, בדומה ל-BFS מתבצעת **לרוחב** והמטרה של אלגוריתם זה **למצוא את המרחק הקצר ביותר** מצומת ההתחלה שאנו נותנים לאלגוריתם. המרחק אינו כפי שלמדנו עד כה, כאשר הגרף ממושקל, כשמדובר על כמות הקשתות הקצר ביותר שעלינו לעבור על מנת לעבור לצומת הנדרשת אלא מדובר על **המרחק עם סכום משקלים המינימלי ביותר** מצומת התחלה לצומת הרצויה.

את המרחק נסמן בדלטא: $\delta(s,v)$ (כאשר כאן מדובר על המרחק בין צומת s לצומת v).

הערה: דלטא מהווה מרחק בגרף ממושקל, אך יכולה להכיל יותר קשתות ממרחק במונח שהכרנו בגרף לא ממושקל.

תיאור האלגוריתם

<u>מבני נתונים עזר:</u>

- ערמת מינימום/מערך לצורך שליפה של צומת שערך מרחקה הנמוך ביותר בכל איטרציה.
- מערך צבעים []color שעוזר לנו לסמן צמתים שאותם סרקנו, צמתים בתהליך סריקה וצמתים שעוד לא סרקנו. בדומה ל-BFS: לבן – עוד לא נכנס לסריקה, שחור – עבר סריקה.

- **[]π** מכיל מידע אודות האב של כל קודקוד (למשל, אם עבור קודקוד b האב שלו הוא a אזי π[b]=a).
- **[]d** מכיל מידע אודות המרחק בעל המשקל המינימלי של כל קודקוד מקודקוד התחלה.

<u>קלט:</u> גרף G שמיוצג כרשימת סמיכויות (סיבוכיות מקום O(V+E)), פונקציית משקל(כי מדובר על גרף ממושקל) וצומת התחלה s.

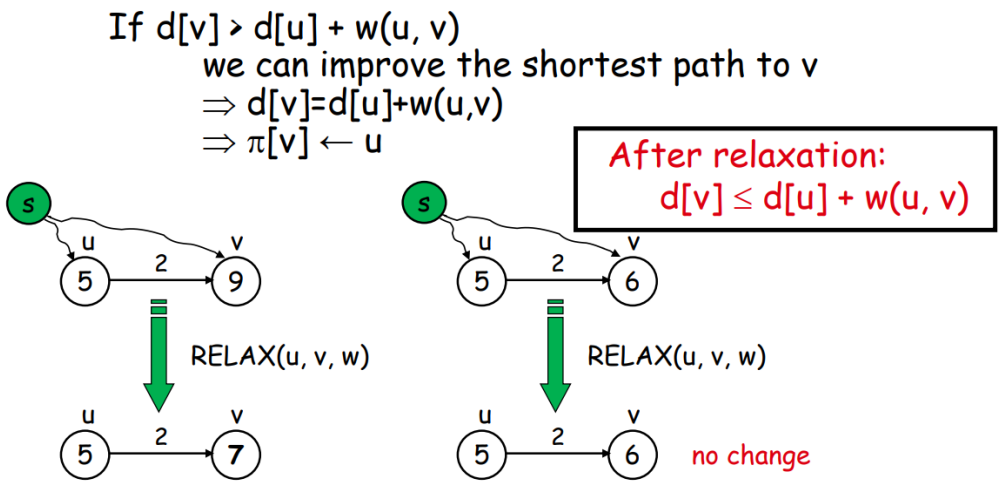
<u>פלט:</u>

- מערך מרחקים []d שנותן את ערך המרחק עם המשקל המינימלי של הצומת מצומת התחלה.
- מערך []π אשר נותן מידע אודות אב של צומת.
- עץ דייקסטרה: Tdij = (Vdij, Edij). קבוצה Vdij מכילה את כל הצמתים של G שעבורם d[v] != ∞ אינסוף, ואילו הקבוצה Edij מכילה את כל הקשתות מהצורה: (π[v], v).
  עבור גרף לא מכוון נקבל עץ פורש של רכיב קשירות של G הכולל את s.
  עבור גרף מכוון נקבל עץ מכוון בעל שורש s.

<u>פירוט הצעדים:</u>

- תחילה, נרוץ בלולאה על כל הקודקודים ונאתחל אותם.
  עבור כל קודקוד v נאתחל:
  1 – color[v] = white
  2 – π[v] = Null
  3 – d[v] = ∞
  עבור קודקוד ההתחלתי, נאתחל את d[v] ל-0
- בכל איטרציה, נבחר את הצומת עם המרחק הקצר ביותר ונעבור על השכנים של הצומת שצבען לבן.

- האלגוריתם מבוסס relaxation, כלומר מבוסס שיפור. לכן אם נסמן ב-u את הצומת שאיתה אנו מבצעים את האיטרציה ונסמן ב-v את השכן שאותו אנו בודקים כעת, אזי נבצע:



- לאחר שעברנו על כל השכנים הלבנים של u, נצבע אותו בשחור ונמשיך לצומת הבא בתור שצבעו לבן וערך מרחקו הקצר ביותר.

האלגוריתם:

Dijkstra's Algorithm (G,w,s)

```
// Step 1:
  for each vertex  v∈V
        d[v] = ∞ and π[v] = NIL
   d[s] = 0
   S ← ∅ ; Q ← V ;
// Step 2
  while Q ≠ ∅
     do u ← EXTRACT-MIN(Q)
     if  d[u] = ∞  then  break
         S ← S ∪ {u}
         for each vertex v ∈ Adj[u] ∩ Q
            do RELAX(u, v, w)

return {d[v], π[v] : v∈V}
```

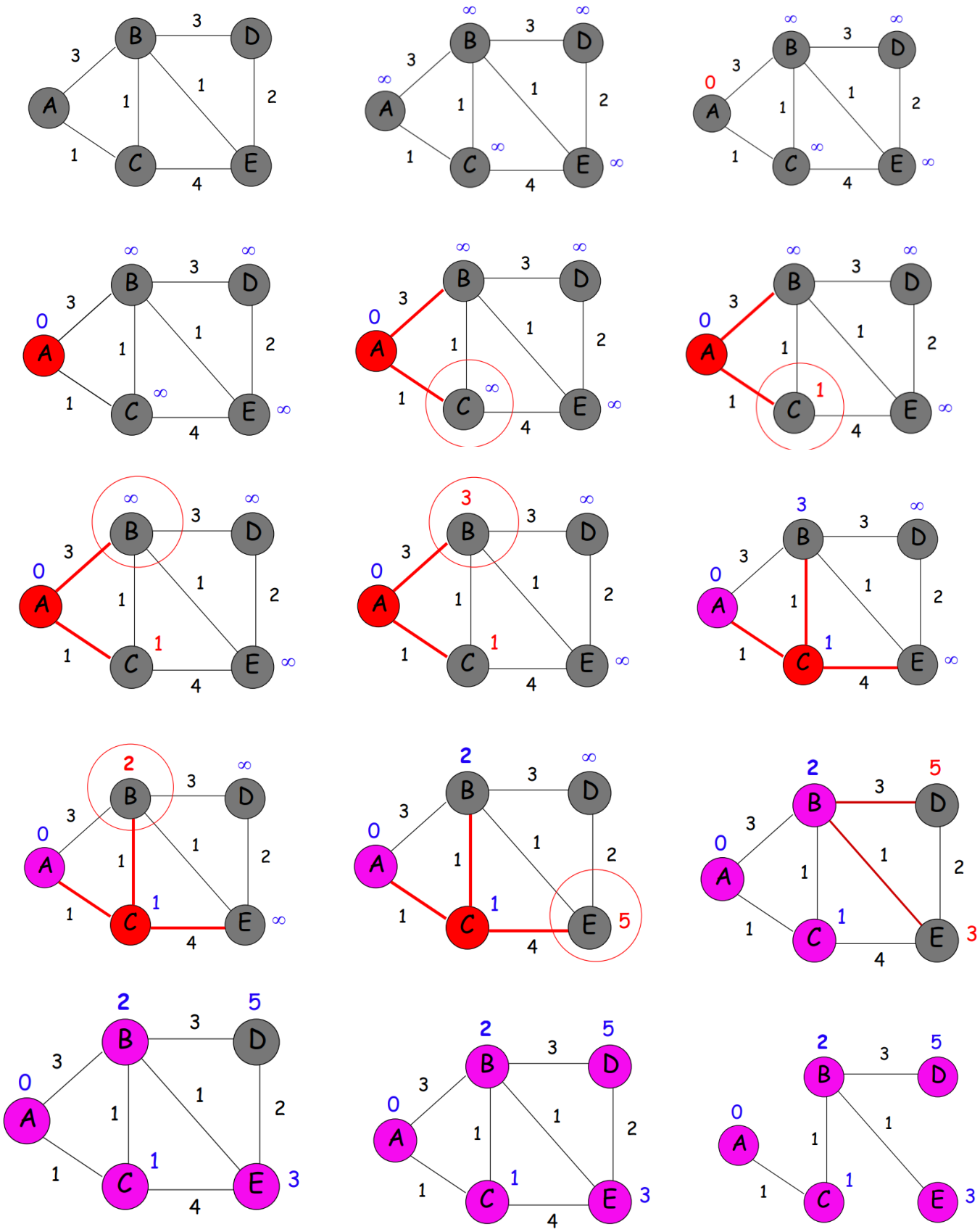
```
if d[u] + w(u,v) < d[v]:
      d[v] = d[u] + w(u,v)
         π[v] = u
```

סיבוכיות:

עבור גרפים דלילים (E = O(V)) עדיף להשתמש בערימת מינימום לצורך שליפת הצומת עם המרחק המינימלי ואז הסיבוכיות של האלגוריתם תהיה: O(V*log(V)).

עבור גרפים לא דלילים (E = O(V^2)) עדיף להשתמש במערך כי סיבוכיות O(V^2) עדיפה על O(E*log(V)).

המחשה (משמאל לימין, למעלה למטה):



משפטים

משפט : **אחרי ביצוע Dijkstra(G,w,s)** מתקיימים:

(א) אם **G לא מכוון**, אז $T_{Dij} = (V_{Dij}, E_{Dij})$ הוא **עץ פורש** של רכיב הקשירות של **G** שכולל את **s**.
אם **G מכוון**, אז $T_{Dij} = (V_{Dij}, E_{Dij})$ הוא עץ מכוון בעל שורש ב- **s**;

(ב) $d[v] = \delta(s,v) = dist(s,v)$ לכל $v \in V$ ב- **G** (לא) מכוון;

הערות

- אלגוריתם דייקסטרה לא עובד עבור גרפים עם משקל שלילי.
- אם מדובר במשקלים קלים מאוד (משקל מקסימלי 2 או 3 או קצת יותר) אז יותר יעיל להגדיל את הגרף בצורה כזו שנניח המשקל של קשת מסוימת היא 3, אזי ליצור בין הצמתים של אותה הקשת, 2 צמתים נוספים כך שמשקל של כל קשת יהפוך ל1. להפעיל את שיטת העבודה הזו על כל קשת בגרף כל שהגרף שלנו יהפוך לגרף בעל משקל מקסימלי 1 ונוכל להפעיל את אלגוריתם BFS ולמצוא את המרחקים הרצויים.

אלגוריתם למציאת מסלול

הערה: נניח שאחרי הרצת **Dijkstra(G,w,s)** הצומת $v \in V$ הוא ב-T_{Dij} . אז ב- T_{dij} קיים מסלול יחיד מ-**s** ל-**v** (הוא מסלול **קצר ביותר** שיש ב-**G** מ-**s** ל-**v** ואולי ב- **G** קיימים עוד מסלולים) שניתן למצוא על ידי האלגוריתם הבא:

```
Print-Path(G,s,v)  ↔  O(V)  time
if v = s
   then print  s
else  if  π[v] = NIL
        then  print "no path exists from  s  to  v

        else  Print-Path(G, s, π[v])
              print  v
```

4 – אלגוריתמים למציאת מרחקים ממושקלים גם שליליים

אלגוריתם בלמן פורד

מוטיבציה

בדומה לאלגוריתם דייקסטרה, גם אלגוריתם זה נועד למצוא את המרחק הקל ביותר מצומת ההתחלה s. אך אלגוריתם דייקסטרה אינו תומך במשקלים שליליים ובנוסף אם ישנו מעגל עם משקל שלילי בגרף, אזי הערך דלטא עבור שני הצמתים יכול להיות לא תקין (מינוס אינסוף).

נמחיש באמצעות דוגמה:

$s \to a$: only one path

$\delta(s, a) = w(s, a) = 3$

$s \to b$: only one path

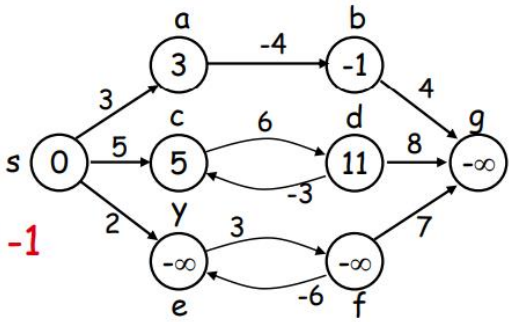
$\delta(s, b) = w(s, a) + w(a, b) = -1$

$s \to c$: infinitely many paths

$\langle s, c\rangle, \langle s, c, d, c\rangle, \langle s, c, d, c, d, c\rangle$

cycle has positive weight (6 - 3 = 3)

$\langle s, c\rangle$ is shortest path with weight $\delta(s, b) = w(s, c) = 5$

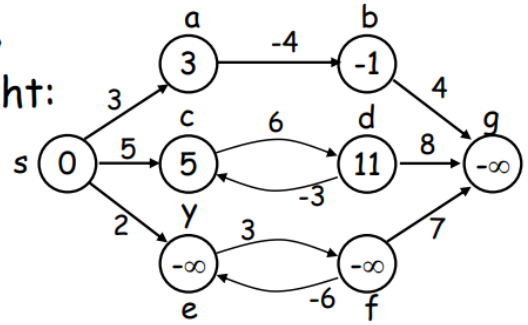


כפי שניתן לראות s -> c הינו בעייתי והינה כמה נוספים:

$s \to e$: infinitely many paths:

- $\langle s, e\rangle, \langle s, e, f, e\rangle, \langle s, e, f, e, f, e\rangle$
- cycle $\langle e, f, e\rangle$ has negative weight:

  3 + (- 6) = -3



- can find paths from s to e with arbitrarily large negative weights
- $\delta(s, e) = -\infty \Rightarrow$ no shortest path exists between s and e
- Similarly: $\delta(s, f) = -\infty$,

  $\delta(s, g) = -\infty$

<u>האם המסלולים הקלים יכולים להכיל מעגלים?</u>

- מעגל שלילי – **לא**. כי אינו מוגדר היטב (כפי שראינו עם המינוס אינסוף)
- מעגל חיובי – **לא**. אם נסיר את המעגל נקבל מסלול טוב יותר.
- מעגל ניטרלי שערכו שווה ל0 – **לא**. הוא מיותר וניתן להגיע לאותה התוצאה אם נוריד אותו.

<u>הערה חשובה:</u> מסלול לא יכול להיות גדול יותר מ V – 1.

הרעיון

המטרה זהה לדייקסטרה, למצוא את המרחקים הקלים ביותר מצומת ההתחלה s. בנוסף, גם כאן נבצע relaxation ובשונה מדייקסטרה, אנו נבצע relaxation V – 1 פעמים וזאת על מנת לוודא שאורך המסלול לא עולה על V – 1. בנוסף, נבצע בדיקה נוספת לאחר ההרצה של הrelaxation על מנת לבדוק אם קיימים מעגלים בגרף. אם נתקלנו במעגל, נחזיר FALSE כי לא ניתן לשער את המרחקים הקלים ואחרת נחזיר TRUE עם הפלט המתאים.

תיאור האלגוריתם

מבני עזר, קלט ופלט זהים לדייקסטרה (פרט לכך שאין צורך במערך color).

<u>פירוט הצעדים:</u>

- תחילה, נרוץ בלולאה על כל הקודקודים ונאתחל אותם.
  עבור כל קודקוד v נאתחל:
  π[v] = Null – 1
  d[v] = ∞ – 2
  עבור קודקוד ההתחלתי, נאתחל את d[v] ל-0
- נבצע RELAX בלולאה שרצה V – 1 פעמים על כל הקשתות של G.
- נבצע בדיקת מעגלים ונרוץ בלולאה שוב על כל הקשתות של G ונבצע RELAX. אם נכנסנו לתוך הIF של הRELAX, נחזיר FALSE כי יש לנו מעגל ואם עברנו על כל הקשתות ולא נכנסנו לתוך הIF, נחזיר את הפלט.

<u>האלגוריתם:</u>

Bellman-Ford Algorithm

```
Bellman-Ford (G,s) //G - digraph
// Step 1: ------- initialization ---------------- O(|V|)
   for each vertex  v∈V
       d[v] = ∞ and π[v] = NIL
   d[s] = 0
// Step 2 ............................... relax edges repeatedly
       for i from 1 to |V|-1:
           for each edge (u, v) ∈ E  ........................ O(|V||E|)
               if d[u] + w(u,v) < d[v]:
                           d[v] = d[u] + w(u,v)
                           π[v] = u
// Step 3  ........... check for negative-weight cycles ---------- O(|E|)
       for each edge (u, v) ∈ E
           if d[u] + w(u,v) < d[v]:
               then return error "Graph contains a negative-weight cycle"
return {d[v], π[v] : v∈V}
```

Complexity : O(|V||E|)

<u>סיבוכיות:</u> מקרה גרוע O(V*E) מקרה הטוב ביותר O(E).

הערות

- האלגוריתם עובד על **גרף מכוון בלבד**.
- תמיד יש לנו לכל v: $d[v] \geq \delta(s, v)$
- הערכה של המרחק הקל ביותר שמתקבל מהאלגוריתם לעולם לא יעלה. פעולת הrelaxation רק גורמת לערך לרדת עוד אך לא לעלות.

משפטים

- אם G לא מכיל בתוכו מעגלים שליליים אזי לאחר הרצת האלגוריתם נקבל כי: $d[v] = \delta(s, v)$ לכל v לאחר V – 1 פעולות RELAX.

נקודת מבט אחרת

בעצם, אלגוריתם בלמן פורד בא מעולם התכנות הדינאמי:

Let d(i, j) = cost of the shortest path from s to i that uses at most j edges.

$$d(i, j) = \begin{cases} 0 & \text{if } i = s \wedge j = 0 \\ \infty & \text{if } i \neq s \wedge j = 0 \\ \min(\{d(k, j-1) + w(k, i) : (k,i) \in E\} \cup \{d(i, j-1)\}) & \text{if } j > 0 \end{cases}$$

אלגוריתם DAG-Shortest Path

הרעיון

בדומה לאלגוריתם בלמן פורד, גם כאן מדובר באלגוריתם המאפשר מציאה של מרחקים בגרפים ממושקלים מכוונים ומאפשרים משקל שלילי.
ההבדל הוא שכאן מתמקדים בגרפים א-ציקליים בלבד.

תיאור האלגוריתם

הקלט, הפלט ומבני עזר זהים לאלגוריתם בלמן פורד.

<u>פירוט הצעדים:</u>

- תחילה מריצים אלגוריתם TOP_SORT וממיינים טופולוגית את הצמתים של הגרף.
- מאתחלים כל צומת באופן זהה כפי שאתחלנו בבלמן פורד.
- בלולאה רצים על הצמתים לפי הסדר של מיון הטופולוגי ובתוך הלולאה רצים על כל השכנים של הצומת ומבצעים על כל שכן RELAX.

<u>הערה:</u> הRELAX מתבצע בהתאם לצומת ההתחלה s שהבאנו לאלגוריתם.

האלגוריתם:

```
DAG-SHORTEST PATHS(G, s)
    TOPOLOGICALLY-SORT the vertices of G
    INITIALIZE-SINGLE-SOURCE(G, s)
    for each vertex u taken in topologically sorted order
    do
        for each v in Adj[u] do
            RELAX(u, v)
```

סיבוכיות:

המיון הטופולוגי: O(V+E)

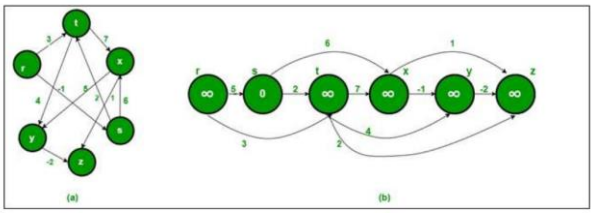
האתחול: O(V)

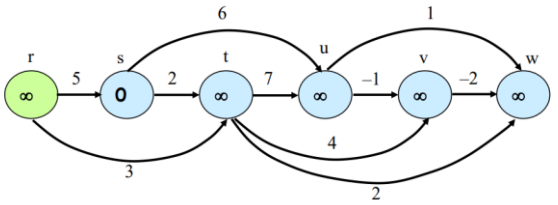
המעבר על כל צומת במיון: O(V)

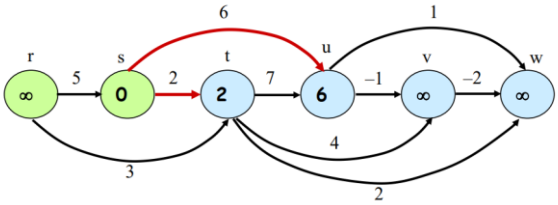
המעבר על כל שכן של שכל כל צומת + RELAX: O(E)

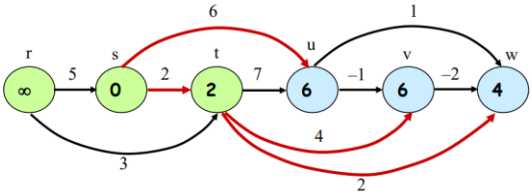
סה"כ: O(V+E)

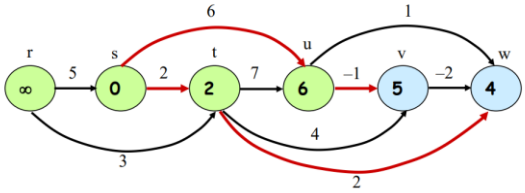
המחשה:

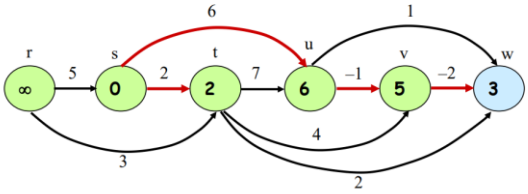

(a)
(b)

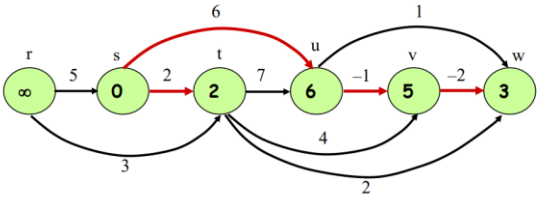

r s t u v w
∞ 0 ∞ ∞ ∞ ∞
5 2 7 −1 −2
6 1 4 3 2


r s t u v w
∞ 0 2 6 ∞ ∞
5 2 7 −1 −2
6 1 4 3 2


r s t u v w
∞ 0 2 6 6 4
5 2 7 −1 −2
6 1 4 3 2


r s t u v w
∞ 0 2 6 5 4
5 2 7 −1 −2
6 1 4 3 2


r s t u v w
∞ 0 2 6 5 3
5 2 7 −1 −2
6 1 4 3 2


r s t u v w
∞ 0 2 6 5 3
5 2 7 −1 −2
6 1 4 3 2

משפטים

- **משפט** : אחרי ביצוע DAG-SHORTEST PATHS(G,w,s) מתקיים:
  $d[v] = \delta(s,v) = dist(s,v)$ לכל $v \in V$ ב- G מכוון ללא מעגלים;

אלגוריתם CPM

Critical **P**ath **M**ethod – **CPM**

הרעיון

האלגוריתם בא לפתור בעיה ספציפית הקשורה לפרויקט.

כעת נגדיר מה זה פרויקט:

פרויקט: אוסף פעילויות הקשורות זו בזו וכולן מכוונות למטרה אחת.

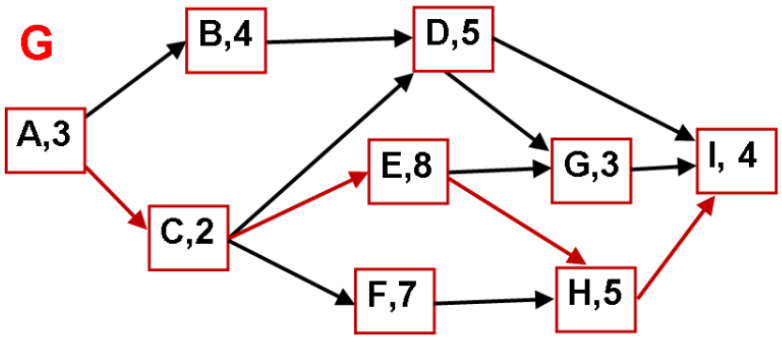
מודל מתמטי לייצוג פרויקט: גרף א-ציקלי, מכוון G = (V,E) שבו:

- הצמתים מייצגים פעילויות.
- הצלעות מייצגות את אילוצי הקדימויות.
- קיים צומת "התחלה" שממנו יוצאים לצמתים(=משימות) הבאות ואין צומת הנכנסת אליו.
- קיים צומת "סיום" שאין צומת היוצאת ממנו ואליו נכנסות הצמתים האחרונות.
- לכל צומת יש "זמן ביצוע" המבוטא באמצעות פונקציית משקל שהינה על **הצומת** עצמה ולא על הקשת.

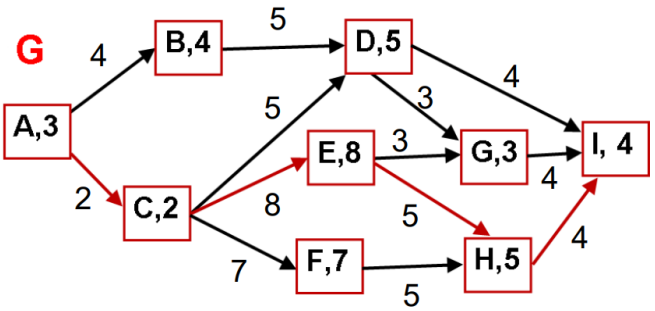
האלגוריתם CPM מוצא את **המסלול הקריטי** – **הזמן המינימלי** להשלים את הפרויקט.

דוגמא של פרויקט:

פעילות	זמן ביצוע	מקדימים מידיים
A = s	3	-
B	4	A
C	2	A
D	5	B, C
E	6	C
F	7	C
G	3	D, E
H	5	E, F
I = t	8	D, G, H



הערה: ניתן "להמיר" את המשקלים שעל הצמתים לצלעות באופן הבא:



חשוב להבין: בגלל התלות של המשימות אחת בשנייה, מדובר בעצם על מציאת **המרחק המקסימלי** בין A ל-I.

מדוע: ניקח למשל את המשימה D שתלויה במשימות B ו-C.
אם היינו מחשבים את המרחק המינימלי מ-A ל-D היינו מקבלים כי המרחק המינימלי הינו: 3 + 2 + 5 = 10
אך זה אינו נכון במצב של פרויקט כי D תלוי גם בסיום המשימה B שיותר ארוכה ולכן הזמן המינימלי להשלמת משימה D הינו: 3 + 4 + 5 = 12.

תיאור האלגוריתם

בעצם האלגוריתם זהה לחלוטין לאלגוריתם DAG-Shortest Path אך ההבדל העיקרי הינו ב-RELAX.
בביצוע RELAX, אנחנו בודקים האם המרחק של השכן גדול מהמרחק של צומת האב + המשקל שלו ואם כן, אז נבצע עדכון של מרחק השכן ונוריד את ערכו.
כאן מדובר בעצם הפוך. אם המרחק של השכן קטן מהמרחק של צומת אביו + המשקל שלו, אז נבצע עדכון של המרחק ונגדיל את ערכו.

האלגוריתם:

Critical Path Method (CPM)

Algorithm for finding CRITICAL PATH

Input: $G = (V,E,w)$ dag weighted in vertices
Output: finish(x), $x \in V$ and CP

```
1. Topological_Sort(G)
2. INITIALIZE:  finish(x) = 0 , for all  x
   for the first vertex in topological order a:
   finish(a)=w(a)
3. for each x in topologically sorted order
            for each y in Adj[x] do
              if (finish(y)<finish(x)+w(y))
                  finish(y)=finish(x)+w(y)
                  π[y] ← x
4.     return finish[], π[]
```

סיבוכיות: בדומה ל-DAG, מדובר על O(V+E).

5 – אלגוריתם Floyd-Warshall

הרעיון

גם כאן מדובר על גרף מכוון וממושקל ומדובר בבעיית מציאת המרחק(דלטא) הקל ביותר.
אלגוריתם זה עוסק בבעיה רחבה יותר, כאשר נרצה לחשב את המרחקים של כל הקודקודים בגרף.
לשם כך, אם נשתמש באלגוריתמים שלמדנו עד כה:

- אם יש משקל שלילי, בשביל למצוא את המרחקים של כל הקודקודים בגרף, נצטרך להריץ את אלגוריתם בלמן פורד |V| פעמים (פעם אחת לכל קודקוד) והסיבוכיות של זה: O(V^4)
- אם אין משקל שלילי, נוכל להריץ דייקסטרה ונקבל סיבוכיות קצת יותר יעילה: O(V^3)

כדי שנוכל לשפר את זמן הריצה של גרף מכוון עם משקל שלילי, נוכל להיעזר באלגוריתם פלויד-וורשל.
גם אלגוריתם זה, כמו בלמן פורד, יכול לקבל משקלים שליליים אך אינו תומך במעגלים שליליים.
עם אלגוריתם פלויד-וורשל נוכל לשפר את הסיבוכיות ל: O(V^3).

גם אלגוריתם זה בדומה לבלמן פורד, מבוסס על תכנות דינמי.

תיאור האלגוריתם

<u>קלט:</u> גרף מכוון ממושקל G = (V,E,w) שמיוצג באמצעות מטריצת מרחקים W = (w(i,j)) כך ש:

$$w(i,j) = 0 \text{ if } i=j,$$
$$w(i,j) = \infty \text{ if } (i,j) \notin E.$$
$$w(i,j) = \text{“weight of edge } (i,j)\text{”}$$

<u>הערה:</u> במידה ויש צלע (I,j) אז המשקל שיופיע בטבלת הקלט זה המשקל הישיר שיש ביניהן (אפילו שאולי יש משקל יותר קל, אבל את זה האלגוריתם יחשב במטריצת פלט).

<u>פלט:</u> מטריצה D = (dij) שתכיל את המידע אודות המרחק מקודקוד i לקודקוד j.

בניית נוסחה רקורסיבית

הנוסחה

נניח שיש לנו קבוצת צמתים: V = {1,2,3,…n}.

נסמן ב- d(i,j)^(k) כך ש- 0<=k<=n והסימון הזה יהווה את המשקל המינימלי בין i ל-j באמצעות {1,2,..,k} צמתים.

אזי נוסחת הנסיגה שלנו שפותרת את הבעיה:

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k > 0 \end{cases}$$

<u>הסבר:</u>

אם k = 0, אז זה אומר שהמסלול בין i ל-j לא יכול לעבור דרך שום צומת נוספת, כלומר מדובר פשוט על המשקל של הצלע ij.

אחרת, כאשר נרצה לחשב d(i,j)^(k) יש לנו כבר את הערך של: d(i,j)^(k-1) ועכשיו יש לנו שתי אופציות:

1- המסלול הקצר ביותר מ-i ל-j אינו כולל את הצומת k
2- המסלול הקצר ביותר כן כולל את הצומת k ועל מנת לקבל את d(i,j)^(k) אזי נבצע חיבור בין d(i,k)^(k-1) לבין d(k,j)^(k-1). כלומר מחשבים את המסלול הקצר ביותר מ-i ל-k ואת המסלול הקצר ביותר בין k ל-j.

ומבצעים min(1,2) והערך המינימלי מבין השניים נכנס למטריצה.

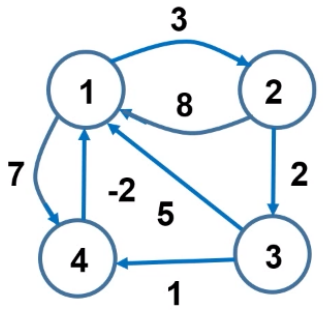
האלגוריתם

```
FLOYD-WARSHALL(W)
n ← rows[W]
D(0) ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            d_ij^(k) ← min (d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

<u>סיבוכיות:</u> O(N^3)

<u>המחשה:</u>

נדגים המחשה חלקית בשביל להבין רעיונית איך האלגוריתם עובד:



$D^{(0)}$	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	-2	∞	∞	0

$d_{23}^{(1)} = \min\{ d_{23}^{(0)}, d_{21}^{(0)} + d_{13}^{(0)} \} = \min\{2, 8+\infty\} = 2$
$d_{24}^{(1)} = \min\{ d_{24}^{(0)}, d_{21}^{(0)} + d_{14}^{(0)} \} = \min\{\infty, 8+7\} = 15$
$d_{32}^{(1)} = \min\{ d_{32}^{(0)}, d_{31}^{(0)} + d_{12}^{(0)} \} = \min\{\infty, 5+3\} = 8$
$d_{34}^{(1)} = \min\{ d_{34}^{(0)}, d_{31}^{(0)} + d_{14}^{(0)} \} = \min\{1, 5+7\} = 1$
$d_{42}^{(1)} = \min\{ d_{42}^{(0)}, d_{41}^{(0)} + d_{14}^{(0)} \} = \min\{\infty, -2+3\} = 1$
$d_{43}^{(1)} = \min\{ d_{43}^{(0)}, d_{41}^{(0)} + d_{13}^{(0)} \} = \min\{\infty, -2+\infty\} = \infty$

$D^{(1)}$	1	2	3	4
1	0	3	∞	7
2	8	0	2	15
3	5	8	0	1
4	-2	1	∞	0

<u>הסבר:</u> הטבלה D(0) בעצם מכילה את האתחול הראשוני של המטריצה. אם יש קשת בין i ל-j אז נרשום את המשקל שלה ואחרת נמלא אינסוף במשבצת.

הטבלה D(1) בעצם אומרת שניתן להשתמש במסלול שלנו בצומת 1 וניתן לראות את החישובים של כל אחד מהתאים ואיך הם משתנים בהתאם לאלגוריתם.

שחזור מסלול

כעת נרצה ליצור אלגוריתם שבאמצעות המטריצה שיצרנו באלגוריתם הקודם, שנוכל לשחזר את המסלול מ-i ל-j.

באלגוריתמים הקודמים שלמדנו היה לנו מערך ז, כאן נגדיר באופן דומה:

מטריצת מסלולים P

תשמור ב P[i][j] את הצומת הפנימי האחרון במסלול הקצר ביותר בין i ל-j.

לכן אם המסלול הקצר ביותר בין i ל-j הוא: <i, v1, v2, …. , vk, j>

אז: P[i][j] = vk.

אתחול:

$$P[i][j] = \begin{cases} i, (i,j) \in E \\ NIL, otherwise \end{cases}$$

דוגמא למטריצת מסלולים:

NIL	3	4	5	1
4	NIL	4	2	1
4	3	NIL	2	1
4	3	4	NIL	1
4	3	4	5	NIL

נשחזר מסלול הקצר ביותר מ-1 ל-2

P[1][2]=3 → **1**,…**3**,2

P[1][3]=4 → **1**,…,**4**,3,2

P[1][4]=5 →**1**,…,**5**,4,3,2

P[1][5]=1 →**1,5,4,3,2**

<u>הסבר:</u> ניסינו למצוא מסלול מ1 ל2. חישבנו P[1][2] וקיבלנו 3, כלומר יש לנו מסלול מ1 עד 3 שעובר דרך כל מני צמתים וישר לאחר מכן מגיעים ל2.
בשביל למצוא את מה שקורה בין 1 ל3, חישבנו את P[1][3] וקיבלנו 4, כלומר יש לנו מסלול מ1 עד 4 שעובר דרך כל מני צמתים וישר לאחר מכן מגיעים ל3.
עד כה קיבלנו: 4,3,2 , … ,1
באותו האופן ממשיכים כך עד שמגיעים לתנאי העצירה.

אלגוריתם לאתחול P

```
INITIALIZE_PATH_MATRIX(W)

for i ← 1 to n
    for j ← 1 to n
        if (i≠j) and (W[i][j] ≠ ∞) //(i,j)∈E
            p_ij =i //P[i][j] = p_ij
        else
            p_ij =NIL
return P
```

<u>סיבוכיות:</u> O(n^2)

האלגוריתם להדפסת המסלול

```
Print_Path(P,i,j)

if i = j
    then print  I
else  if  P[i][j] = NIL
    then  print "no path exists from  i  to  j"
else
        Print-Path(P, i, P[i][j])
        print  j
```

<u>סיבוכיות:</u> O(V)

ייעול האלגוריתם

האלגוריתם שציינו משתמש ב-(n+1) מטריצות D, האם זה הכרחי? והאם אפשר לעשות אותו יותר יעיל?

ניתן לשפר על ידי שימוש ב-2 מטריצות בכל פעם. בהתחלה מאתחלים את מטריצה D(0) ומשתמשים בה בשביל לבנות את מטריצה D(1) אך לאחר שסיימנו לבנות את D(1) ניתן לדרוס את המטריצה D(0) ולשים בה את D(1) ובאופן דומה ממשיכים לבנות את שאר המטריצות:

```
FLOYD-WARSHALL (W)

n      ←   rows[W]
D(0)   ←   W
P      ←   INITIALIZE _PATH_MATRIX(W)
  for k  ←  1 to n{
     for i  ←  1 to n
        for j  ←  1 to n
           if (d_ij^(0) > d_ik^(0)+d_kj^(0))
                d_ij^(1)  ←  d_ik^(0)+d_kj^(0))
                p_ij  ←  p_kj
           else
                d_ij^(1)  ←  d_ij^(0)
           }
     D(0)  ←  D(1)
return D(n)                    Shortest Path Problems 5
```

האם ניתן לשפר ולהשתמש במטריצה אחת?

נשים לב כי כדי לחשב את dij אנחנו מסתפקים רק בשורה k ובעמודה k ולא צריכים את שאר השורות והעמודות וזה נותן לנו את האפשרות להסתפק במטריצה אחת בלבד וכך נקבל את האלגוריתם הסופי:

האלגוריתם הסופי

FLOYD-WARSHALL (W)

```
n ← rows[W]
D ← W
P ← INITIALIZE _PATH_MATRIX(W)
for k ← 1 to n
    for i ← 1 to n
        for j ← 1 to n
            if (d_ij > d_ik + d_kj)
                d_ij ← d_ik + d_kj
                p_ij ← p_kj
return D
```

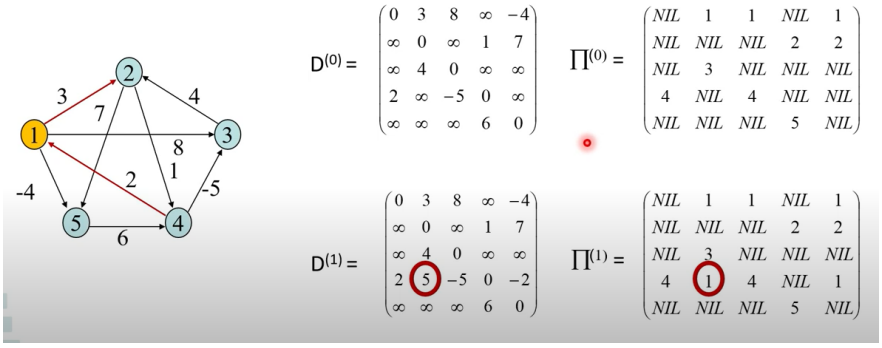
<u>סיבוכיות:</u> נשארת זהה O(n^3) אבל שיפרנו את סיבוכיות המקום שלנו.

<u>המחשה:</u>

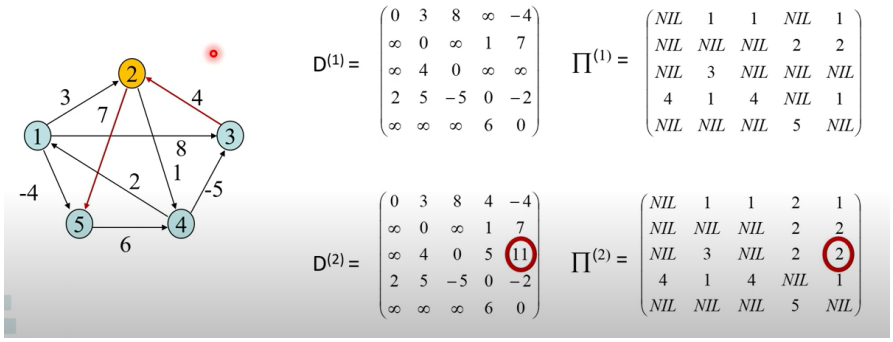
לצורך ההמחשה הפרדנו בין D(0) ל-D(1) וכן הלאה ובאופן דומה גם עבור מטריצה P אך באלגוריתם מדובר במטריצה אחת בלבד שאנו דורסים את ערכיה:

- העיגולים האדומים במטריצות מסמנים דוגמה לשינויים בין איטרציה אחת לשנייה (יכולים להיות כמה שינויים נוספים, אבל בכל תמונה נדגים שינוי ספציפי ב-D והמקביל אליו ב-P).

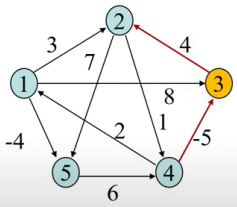
K = 1:



K = 2:



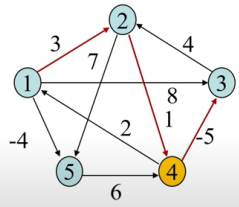
:K = 3



$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 1 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

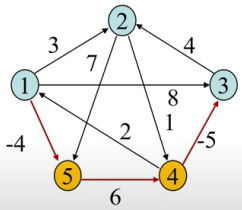
:K = 4



$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} NIL & 1 & 1 & 2 & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & 2 & 2 \\ 4 & 3 & 4 & NIL & 1 \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

:K = 5



$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} NIL & 1 & 4 & 2 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} NIL & 3 & 4 & 5 & 1 \\ 4 & NIL & 4 & 2 & 1 \\ 4 & 3 & NIL & 2 & 1 \\ 4 & 3 & 4 & NIL & 1 \\ 4 & 3 & 4 & 5 & NIL \end{pmatrix}$$

בדיקה האם קיים מעגל שלילי בגרף

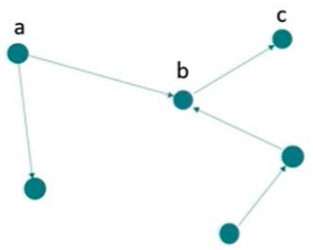
<u>טענה:</u> אם הגרף ללא מעגלים שליליים, אז ערך האלכסון יישאר ללא שינוי (=0) בזמן הרצת האלגוריתם.

<u>מסקנה:</u> אם יש מעגל שלילי אז אחד מערכי האלכסון יהיה שלילי ושונה מ-0 וכך נוכל לזהות אם יש לנו מעגל שלילי בגרף.

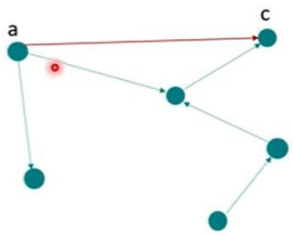
בעיית סגור טרנזיטיבי

זהו אלגוריתם אב קדמון לאלגוריתם פלויד-וורשל שלנו.

לפני שנגדיר את הבעיה, נגדיר טרנזיטיביות בגרף:

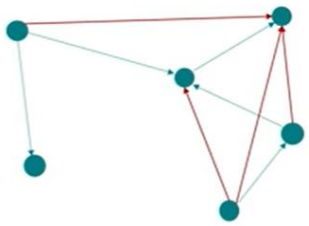


נשים לב שיש לנו קשת (a,b) וקשת (b,c) וכאשר קיימת קשת זה אומר שיש בין הצמתים יחס.
אזי הטרנזיטיביות באה לידי ביטוי בכך שעל סמך הקשתות שציינו קודם גם (a,c) הוא יחס.



הקשת המסומנת באדום הינה היחס החדש שמצאנו בזכות הטרנזיטיביות.

באופן דומה נמצא את כל היחסים שנובעים מטרנזיטיביות ונסמן אותם באדום:



<u>הגדרה:</u> **סגור טרנזיטיבי** של גרף מכוון G = (V,E) הוא גרף מכוון Gtr = (V,Etr) כך שהצלע (u,v) שייכת לEtr אם ורק אם קיים מסלול מ-u ל-v ב-G.

כלומר, סגור טרנזיטיבי הוא גרף מכוון שבו קיימת קשת המחברת בין כל שני צמתים אשר יש מסלול ביניהם בגרף המקורי.

הבעיה שלנו עוסקת ביצירת סגור טרנזיטיבי.

תיאור האלגוריתם

<u>קלט:</u> גרף מכוון לא ממושקל G אשר מיוצג כמטריצת סמיכויות באופן הבא: לכל i,j צמתים ב-G, המטריצה W[i][j] שווה ל-0 אם אין קשת בין i ל-j ו-1 אחרת.

<u>פלט:</u> סגור טרנזיטיבי המיוצג באמצעות מטריצה T[i][j] כך שלכל i,j צמתים ב-G, המטריצה T[i][j] שווה ל-0 אם אין מסלול בין i ל-j ו-1 אחרת.

<u>הצעה לפתרון:</u> להשתמש באלגוריתם פלויד-וורשל עם משקל לכל צלע = 1 ולהמיר את מטריצת התוצאה כך שכל ערך ששונה מאינסוף יהפוך ל-1 ומה ששווה לאינסוף יהפוך ל-0.

האם ניתן להגיע לפתרון יעיל יותר?

כן. באמצעות אופרטורים בוליאניים מכיוון שמדובר על תוצאה שכוללת בתוכה 0 או 1 בלבד.

נבנה נוסחת נסיגה בדומה לפלויד-וורשל:

$$t_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k=0, \\ t_{ij}^{(k-1)} \vee (t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}) & \text{if } k \geq 1. \end{cases}$$

<u>האלגוריתם:</u>

```
Transitive Closure(W)
n ← rows[W]
T ← W
for k ← 1 to n
    do for i ← 1 to n
        do for j ← 1 to n
            t_ij ← t_ij ∨ (t_ik ∧ t_kj)
return T
```

<u>סיבוכיות:</u> O(n^3)

<u>סיכום:</u> האלגוריתם הזה הרבה יותר פשוט מפלויד-וורשל והמטרה שלו רק למצוא האם קיים מסלול בין שני צמתים אך לא מעניין אותנו מהו המסלול ומה משקלו.

6 – רשתות זרימה

הרעיון

בתורת הגרפים, **רשת זרימה** היא סוג מיוחד של גרף מכוון, שמשמש למידול בעיות שמערבות מעבר של חומר בין מקומות. ניתן להשתמש ברשתות זרימה כדי למדל:

- זרימה של נוזל בצינורות.
- מעבר של מידע ברשתות תקשורת.
- מעבר של תנועה בכביש.
- זרם ברשתות חשמל.

לרשתות זרימה ישנם גם שימושים לפתרון בעיות תאורטיות.

<u>באופן פורמלי:</u>

נתונה רשת זרימה $N = (G,s,t,c)$, כאשר:

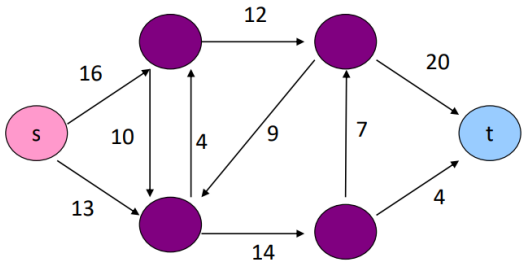
- $G = (V,E)$ **גרף מכוון.**
- $s,t \in V$ **צומת מקור (source)** s **וצומת בור (sink)** t **.**
- $c: E \to \mathbb{R}$ **פונקציית קיבול (Capacity)**

$c(e)$**- הזרימה המקסימלית שניתן להעביר דרך קשת** $e \in E$**.**

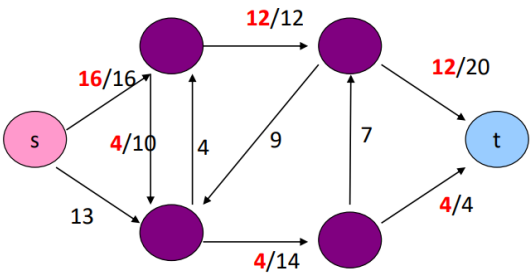
<u>הבעיות שנעסוק בהן:</u> מציאת קצב מקסימלי שניתן להזרים חומר מ-s ל-t מבלי לחרוג מקיבולי הצלעות.

<u>המחשה:</u>

ניקח את רשת הזרימה הבאה ונדגים עליה:



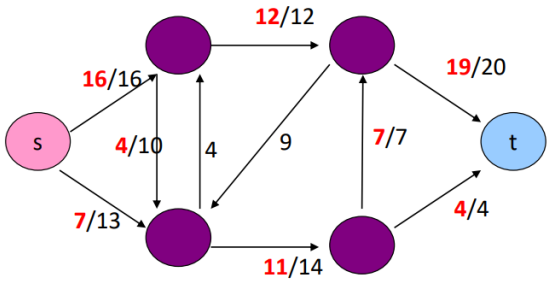
כך נראה קצב = 16:



16

הסבר: יוצאים מצומת s ובוחרים בקשת עם קיבולת 16 ומזרימים את כל הנוזל. כעת בצומת החדשה יש לנו 16 ליטר נוזל שעלינו לפזר הלאה.
יש לנו 2 קשתות שניתן לצאת אליהן: אחת עם קיבולת 4 והשנייה 12, נפזר את הנוזלים ביניהן.
מהצומת אליה שהזרמנו 12 ליטר, נעביר את ה12 ליטר אל תוך צומת t.
מסקנה: הצלחנו להזרים בדקה 16 ליטרים של נוזלים וזו הקיבולת המקסימלית למסלול הזה.

האם ניתן להגיע לקיבולת יותר מקסימלית? כן. המחשה:



23

הסבר: יוצאנו מצומת s 16 ליטר לצומת אחת ו7 ליטר לצומת השנייה. בצומת עם ה16 ליטר פיזרנו את הנוזלים לצומת אחת 12 ולשנייה 4. הצומת שקיבלה 4 ליטר מעבירה אותם לצומת עם ה7 ליטר שפיזרנו מs וכעת יש לנו בצומת 11 ליטר שאותם אנו מעבירים לצומת עם הקיבולת 14. מהצומת הזו אנו מפזרים את המים לצומת אחת 7 ליטר ולצומת t את ה4 הנותרים.
כעת נחזור לצומת שקיבלה בהתחלה 12 ליטר. הוזרמו אליה 7 ליטרים נוספים ויש לנו 19 ליטר שאותם אנו מעבירים לצומת t.
סה"כ הוצאנו בדקה 23 ליטר, המסלול המקסימלי.

האתגר הוא בעצם למצוא את המסלול המקסימלי ולהוכיח שהוא אכן המקסימלי.

פונקציית הזרימה

פירמול הבעיה

נגדיר כעת באופן פורמלי את הבעיה וניצור את פונקציית הזרימה f:

פונקציה $f:V\times V\to R$ נקראת **פונקצית זרימה** אם מתקיימים התנאים:

- **אילוץ הקיבול** (capacity constraints): על אף צלע לא מוזרם יותר מהקיבול שלה: $f(u,v) \le c(u,v)$ לכל $u,v\in V$

הסבר לאילוץ הקיבול: אם הקיבול הוא X אזי לא ניתן להעביר יותר מ-X חומר שעוברים דרך הקשת.

- **אנטי-סימטריה** (skew symmetry): $f(u,v) = -f(v,u)$ לכל $u,v\in V$

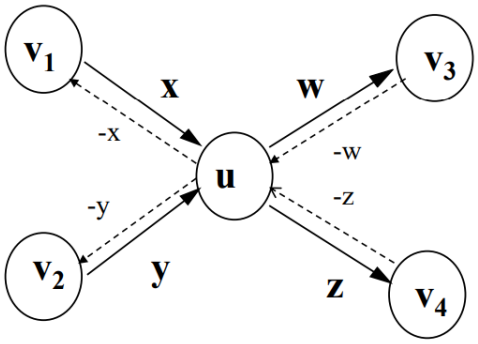
הסבר לאנטי-סימטריה: בעצם לכל קשת עם קיבול X קיימת קשת זהה עם הכיוון הנגדי שערכה X- (גם אם אינה מופיעה בגרף אנו יוצאים מנקודת הנחה שהיא קיימת).

- בתוספת להערה בסוגריים, אפשר להחשיב את הרשת זרימה כגרף שלם וכל הקשתות שאינן מופיעות בגרף ואינן סימטריות לשום קשת שכן קיימת בגרף, אזי הקיבול שלהן הוא 0.

• **שימור הזרימה** (flow conservation): לכל קדקוד u השונה מ s ו t
כמות הזרימה היוצאת מ **u** שווה לכמות הזרימה הנכנסת.

$$\sum_{v\in V} f(u,v) = 0 \quad \text{מתקיים} \quad u \neq s, t \quad \text{לכל}$$

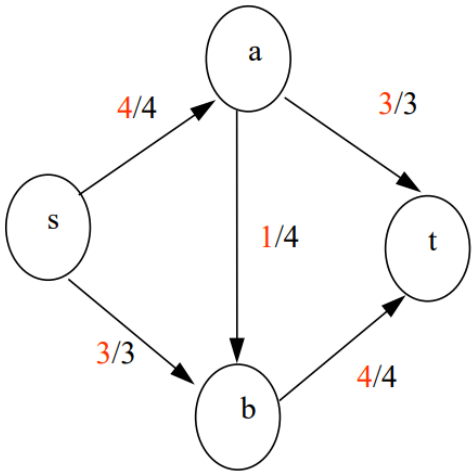
<u>הסבר לשימור הזרימה:</u>



שימור הזרימה: מה שנכנס ל u , כלומר x+y , שווה למה שיוצא,
כלומר z+w . בעזרת האנטי-סימטריה , זה שקול ל: $\sum_{v\in V} f(u,v) = 0$

$$\sum_{v\in V} f(u,v) = -x + -y + w + z = 0 \quad \text{כלומר:}$$

דוגמה לפונקציית זרימה:



אילוץ הקיבול: $f(s,a) \le c(s,a)$ וכנ"ל ביחס לשאר ערכי הזרימה

אנטי-סימטריה: $f(a,s) = -4, \; f(s,a) = 4$

חוק שימור הזרימה: $f(a,s) + f(a,b) + f(a,t) = -4 + 1 + 3 = 0$

• הערה: הפונקציה בעצם זה הצבע האדום המומחש.

ערכה של פונקציית הזרימה

נגדיר: ערך הזרימה |f| : הזרימה שיוצאת מ- s: $|f| = \sum_{v \in V} f(s,v)$

טענה: $\sum_{v \in V} f(v,t) = \sum_{v \in V} f(s,v) = |f|$

טענה שהיא טריוויאלית כי זה ברור שכל מה שיוצא מ-s זה כל מה שיכנס ל-t.

שיטת פורד-פלקרסון

הגדרות

צלע רוויה: צלע שבה עוברת זרימה מרבית אפשרית: f(u,v)=c(u,v)

צלע משפרת: צלע (u,v) שאפשר "להזרים" לאורכה עוד זרימה:

f(u,v)<c(u,v)

מסלול שיפור : מסלול מ- s ל- t שמורכב כולו מצלעות משפרות.

הערה: במסלול שיפור מותר ללכת "נגד הכיוון" – אפשרות להתחרט!

הערה: הגרף המקורי הוא מכוון, אך במסלול משפר מותר ללכת "נגד הכיוון".

רשתות שיוריות:

קיבול שיורי (residual capacity) c_f:

כמות הזרימה שאפשר עדיין להזרים לאורך צלע: $\boxed{c_f(u,v) = c(u,v) - f(u,v)}$

צלע משפרת: צלע (u,v) משפרת אם $c_f(u,v) > 0$.

מסלול שיפור : מסלול מ- s ל- t שמורכב כולו מצלעות משפרות.

הקיבול השיורי של מסלול P: הקיבול השיורי המינימלי לאורך P:

(צוואר הבקבוק) $c_f(P) = \min_{(u,v) \in P} c_f(u,v)$

אם P הוא מסלול שיפור, אזי $c_f(P) > 0$

הגרף השיורי: הגרף G_f שמכיל את כל הצלעות המשפרות של G,

כאשר משקלי הצלעות הם הקיבולים השיוריים c_f .

⇦ כל מסלול מ- s ל- t ב- G_f הוא מסלול שיפור ב- G ולהפך.

הרעיון

באמצעות גרף שיורי אנו יכולים להגיע למסלול המקסימלי. הסיבה לכך היא שהגרף השיורי מאפשר לנו את היכולת לחזור אחורה ולתת לנו הזדמנות "להתחרט" ולחזור אחורה ולשנות את מסלול הזרימה.

תיאור האלגוריתם

ברמה המילולית, תחילה אנו מאפסים את כל פונקציות הזרימה של כל הקשתות ויוצרים גרף שיורי Gf שנכון לאתחול זהה לגרף המקורי G.

בלולאה כל עוד קיים מסלול כלשהו בגרף השיורי Gf מצומת התחלה s לצומת היעד t נבצע:

באמצעות אלגוריתמים שאנו כבר מכירים DFS\BFS אנו מוצאים מסלול P כלשהו מצומת התחלה s לצומת היעד t בגרף השיורי Gf.

ברגע שמצאנו מסלול, נחלץ את הקיבול השיורי Cf(P) של המסלול P, שבעצם שווה לערך של הקשת עם פונקציית קיבול המינימלית.

כעת נעדכן את הגרפים בהתאם:

בלולאה על כל קשת (u,v) במסלול P נבצע:

עבור גרף G:

$$f[u,v] = f[u,v]+c_f(p)$$
$$f[v,u] = -f[u,v]$$

עבור גרף Gf:

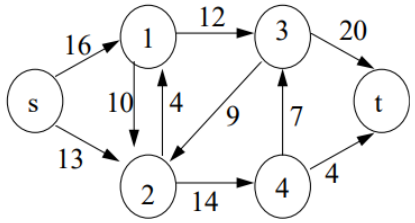
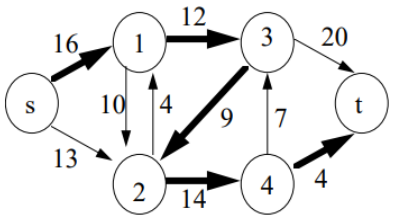
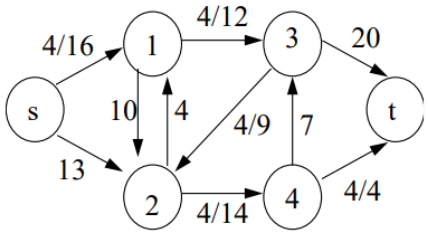
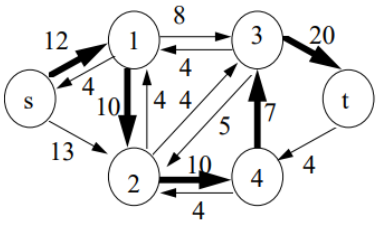
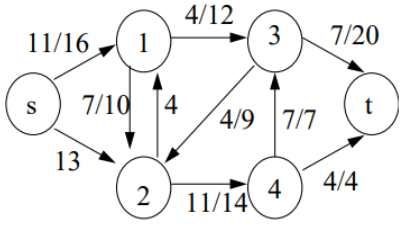
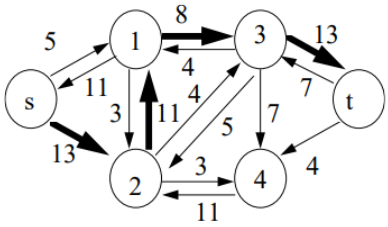
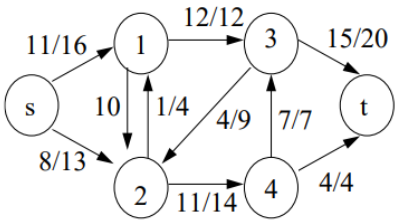
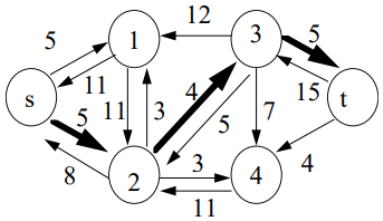
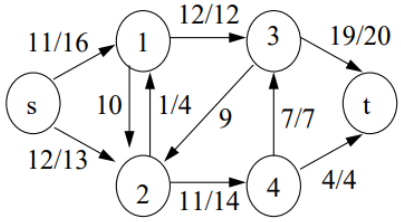
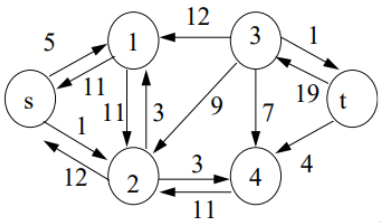
$$c_f(u,v) = c(u,v) - f(u,v)$$
$$c_f(v,u) = c(v,u) - f(v,u)$$

<u>האלגוריתם:</u>

The basic Ford-Fulkerson algorithm

```
FORD-FULKERSON(G,s,t,c)
for each edge (u,v)∈E[G]
    do f[u,v] = 0
       f[v,u] = 0
G_f=G
while there exists a path p from s to t in the residual network G_f
    do c_f(p) = min{c_f(u,v): (u,v) is in p}
       for each edge (u,v) in p
           do f[u,v] = f[u,v]+c_f(p)
              f[v,u] = -f[u,v]
              c_f(u,v) = c(u,v) – f(u,v)
              c_f(v,u) = c(v,u) – f(v,u)
```

המחשה:

הגרף G והזרימה הנוכחית	הגרף G_f והמסלול המשפר P	$C_f(P)$
		4
		7
		8
		4
		אין מסלול משפר

HIT- Yulia Kempner

HIT- Yulia Kempner

משפטים ומסקנות

<u>טענה:</u>

יהיו f ו-f' פונקציות זרימה בגרפים G ו - G_f. אז הפונקציה $f+f'$ היא פונקציית זרימה עבור G, וכמו כן $|f+f'|=|f|+|f'|$

<u>מסקנה הנובעת מהטענה:</u>

תהי f פונקציית זרימה עבור G, ויהי P מסלול שיפור ב $-G_f$.
אז $f^*=f+f_P$ היא פונקציית זרימה עבור G ו -
$$|f^*|=|f|+|f_p|>|f|$$

ניתוח זמן ריצה

כעת ננתח את זמן הריצה של האלגוריתם.

- <u>חלק של האתחול:</u> מכיוון שאנו מאתחלים את כל הקשתות מדובר על O(E).
- <u>לולאה פנימית שבה אנו רצים על המסלול:</u> O(E) כי רצים על הקשתות של המסלול שהחסם העליון שלו הוא E.
- <u>לולאה הראשית שבה אנו רצים כל עוד קיים מסלול ב-Gf:</u> זמן ריצה של איטרציה אחת הינה O(V+E) כי אנו מריצים אלגוריתם DFS או BFS שזהו זמן הריצה שלהם.

נשים לב כי O(V+E) = O(E) כי V = O(E). ולכן סה"כ נקבל שזמן הריצה הוא: O(E) * כמות האיטרציות.

כמה איטרציות נבצע?

נסמן ב-f* את הזרימה המקסימלית ביותר.

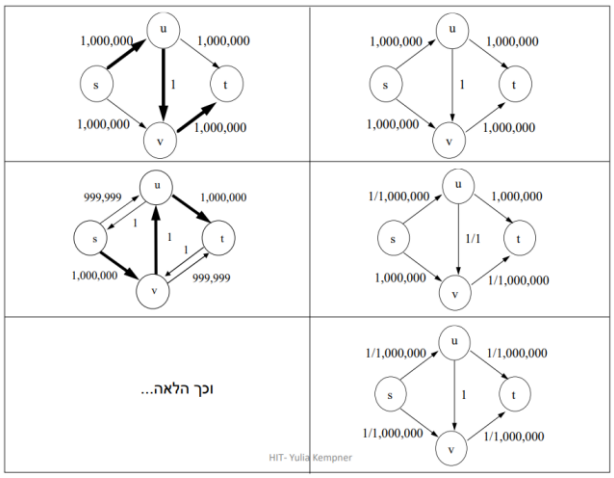
<u>טענה:</u> אם כל הקיבולים שלמים אז ייתכנו לכל היותר f* איטרציות.

למה? כי בכל איטרציה הזרימה גדלה לפחות ב-1.

<u>ולכן הסיבוכיות של האלגוריתם היא:</u> O(Ef*)

<u>הערה:</u> האלגוריתם לא בהכרח פולינומיאלי מכיוון שתלוי בגודל הקלט. כלומר, הערך f* מיוצג ב-log(f*) ביטים ואם הגודל הקלט הוא n אז נקבל: n = log(f*) כלומר: f* = 2^n, כלומר קיבלנו זמן ריצה אקספוננציאלי.

המחשה:



<u>טענה:</u> החסם הדוק, כלומר, ניתן לבחור מסלולים משפרים באופן כזה שיגרום לכך שאכן יהיו f* איטרציות.

אלגוריתם אדמונד-קארפ

הרעיון

בעצם האלגוריתם זהה לחלוטין לאלגוריתם של פורד-פלקרסון וההבדל היחידי הוא שמציאת המסלול ש-Gf מתבצעת תמיד על ידי BFS.

איך זה עוזר לנו? בכך שעל ידי BFS אנחנו תמיד נמצא את המסלול הקצר ביותר ובכך אנו מקבלים שמספר האיטרציות שהלולאה הראשית עושה הוא O(VE) ובכך ייעלנו את זמן ריצת האלגוריתם.

כמו כן בשונה מאלגוריתם פורד-פלקרסון, אלגוריתם זה כן יכול לעבוד עם קיבולים לא שלמים, אלא רציונליים.

סיבוכיות האלגוריתם

נקבל שהסיבוכיות הינה O(min{VE^2, Ef*}).

תכונת החתך בנושא רשתות זרימה

בהרצאות למדנו את מושג החתך שמתקשר לרשתות זרימה והמשפטים הקשורים אליו שעזרו לנו להוכיח את נכונות האלגוריתמים שציינו קודם.

בסיכום נציג את תכונת החתך, הטענות והמשפטים שנובעים ממנה.

חתך ברשת: חתך (S,T) של רשת זרימה הוא חלוקה של **V** לשתי קבוצות זרות, S ו- T = V\S כאשר s∈S, t∈T.

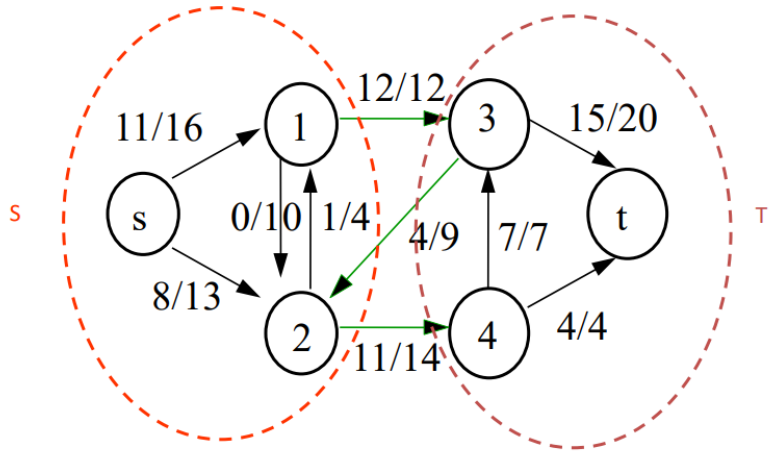
קיבול חתך = סכום **הקיבולים** החוצים את החתך

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

זרימה בחתך = סכום **הזרימות** החוצות את החתך

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v)$$

<u>דוגמה:</u>



c(S,T) = c(1,3) + c(2,4) +c(2,3)= 12 + 14 + 0 = 26

f(S,T) = f(1,3) + f(2,4) + f(2,3) = 12 + 11 + (–4) = 19

למת החתך:

למת החתך: תהי f זרימה ויהי (S,T) חתך. אז $f(S,T) = |f|$

מסקנה: לכל חתך (S,T) של רשת זרימה N, ולכל זרימה f, מתקיים $c(S,T) \geq |f|$.

הסבר: מהלמה לעיל מתקיים: $f(S,T) = |f|$
מחוק הקיבול מתקיים: $c(S,T) \geq f(S,T)$
ומשניהם מסיקים: $c(S,T) \geq |f|$

משפט Max flow, Min cut

משפט: תהי f זרימה ברשת N=(G,s,t,c). התנאים הבאים שקולים:

(1) f זרימה מקסימלית ב- G.

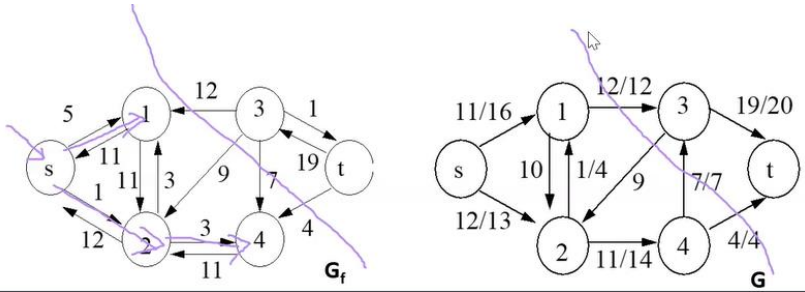
(2) בגרף השיורי G_f אין מסלול מ- s ל- t (אין מסלול שיפור ב- G).

(3) קיים חתך (S,T) ב-G שעבורו $c(S,T) = |f|$

שיטה למציאת חתך מינימלי

להריץ אלגוריתם אדמונד-קארפ וברגע שנגיע לסיום שבו בגרף השיורי אין מסלול מ-s ל-t, אז החתך יכיל בקבוצה S את כל הקודקודים שניתן להגיע אליהם מ-s וקבוצה T יכיל את השאר.

המחשה:



כלומר החתך המינימלי שלנו מכיל בקבוצה S את הקודקודים s,1,2,3 וקבוצה T מכילה את הקודקודים t,3

7 – זיווגים

הגדרות

יהי G = (V,E) גרף לא מכוון.

זיווג (Matching): אם M מוכל ב-E ולכל (a,b),(c,d) קשתות המוכלות ב-M אין צומת משותף, אז M נקראת זיווג (Matching).

גודל זיווג: מספר צלעות בקבוצה M, כלומר |M|.

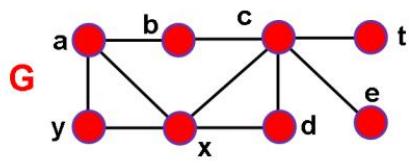
זיווג מקסימום (Maximum Matching): זיווג בגודל מקסימלי ומסומן: $\mu(G)$

זיווג מושלם (Perfect Matching): זיווג שמכסה את כל הצמתים בגרף.

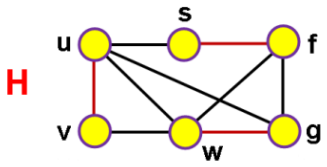
<u>הערות:</u>

- כל זיווג מושלם הוא גם זיווג מקסימום.
- אם ב-G יש זיווג מושלם אז |V| זוגי.

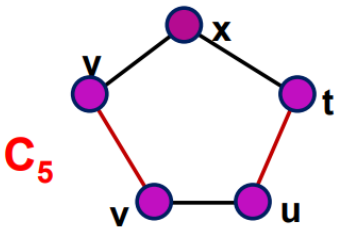
<u>המחשה:</u>



M1={(a,b),(x,d)}-matching,
M2={(a,b),(y,x),(c,e)} – maximum, $\mu(G)$ =3
{(a,b),(c,d), (c,t)} – is not matching



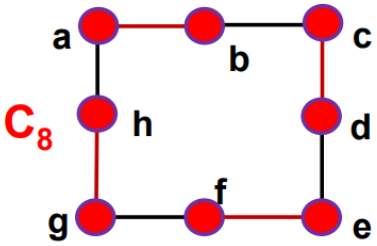
M = {(u,v),(s,f),(g,w)}
זיווג מושלם (וגם מקסימום)



$\mu(C_5) = |\{v,y),(u,t)\}| = 2$

$\mu(C_{2k+1}) = k = \lfloor (2k+1)/2 \rfloor$

ל: C_{2k+1} יש 2k+1 זיווגי מקסימום



$\mu(C_{2k}) = k = (2k)/2$

ל: C_{2k} יש 2 זיווגים מושלמים (וגם מקסימום)

זיווגים בגרף דו צדדי

הגדרות

- **גרף דו צדדי:** גרף שניתן לחלק אותו לשתי קבוצות בלתי תלויות (הגדרה מפורטת עם המחשה ראה בסיכום בנושאי העבר).
- להמחשה נסמן את שתי הקבוצות הבלתי תלויות ב-L (קבוצה שמאלית) וב-R (קבוצה ימנית). יהי G גרף לא מכוון ודו צדדי בעל קבוצות L ו-R וגם קיימת קבוצה X המוכלת ב-L.

אזי נסמן: $N(X) = \cup \{N(v) : v \in X\}$ כלומר קבוצה שמכילה את כל השכנים של X בקבוצה R.

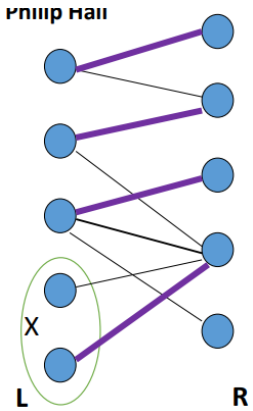
<u>הערה:</u> ב-G יש זיווג M שמכסה את L => $|X| \le |N(X)|$ לכל $X \subseteq L$.

משפט HALL

יהי G = (E,V) גרף לא מכוון דו צדדי כך ש-E מורכב מקבוצות בלתי תלויות L ו-R.

ב-G קיים זיווג שמכסה את L ⇔ $|X| \le |N(X)|$ לכל $X \subseteq L$.

<u>המחשה:</u>



קיבלנו |X| = 2 , |N(X)| = 1 כלומר |X|>|N(X)| ולכן לא קיים זיווג שמכסה את L.

משפט החתונה

ב-G קיים זיווג מושלם ⇔ |L| = |R| וגם $|X| \le |N(X)|$ לכל $X \subseteq L$.

משפט König-Egeváry

תזכורת:

$\alpha(G)$: גודל קבוצה בלתי תלויה הכי גדולה בגרף G.

$\eta(G)$: גודל מינימלי של כיסוי קודקודים.

המשפט אומר:

$$\alpha(G) + \mu(G) = |V|$$
$$\text{or} \quad \mu(G) = \eta(G) .$$