פרויקט סיום במת"מ – חורף ואביב 2019-20

1 פרטים טכנים

הגשה: **בודדים בלבד** (שיתופי פעולה אסורים).
פרסום: **3 באוגוסט, 2020, 14:00**; הגשה: **10 באוגוסט, 2020, 23:55**.
כל סטודנט/ית לו/ה נקבע שירות מילואים, או בעל/ת כל מחוייבות או עניין אישי אחר בתקופת הפרויקט, מתבקש/ת לפנות **מוקדם ככל האפשר** למתרגל האחראי בסמסטר לו הוא/היא רשום/ה (יורי פלדמן או אור אייזקס) כדי לקבוע את זכאותו/ה לתוספת זמן.

2 הגדרת הפרויקט

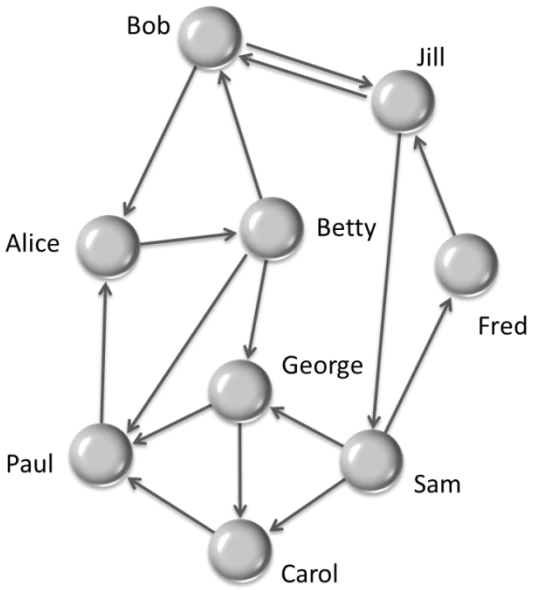
2.1 סקירה

בפרויקט זה יהיה עליכם לממש *מחשבון*. המשתנים במחשבון הם גרפים מכוונים (ראו הגדרה למטה). למחשב יש שני מצבי פעולה: (1) מצב אינטראקטיבי, בו פקודות נקראות מ-`stdin` ופלט נכתב ל-`stdout`; (2) מצב אוטומטי (batch), בו קלט/פלט מבוצע מ/אל קבצים.

2.2 גרף

גרף (מכוון) הינו אובייקט המורכב **מקודקודים ומקשתות**. הקודקודים מייצגים אלמנטים כלשהם (למשל ערים, אנשים, שרתי מחשב, וכו'), והקשתות מחברות ביניהם ומתארות את הקישוריות בין הקודקודים. הקשתות הן מכוונות, במובן שלכל קשת ישנו קודקוד מקור וקודקוד מטרה. הגרף אינו יכול להכיל קשת עצמית (קשת מקודקוד לעצמו) או קשתות מקבילות (שתי קשתות עם בדיוק אותם קודקודי מקור ומטרה). עם זאת, שתי קשתות בכוונים מנוגדים מותרות.

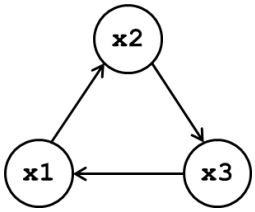
גרף יכול לתאר מגוון רחב של יחסים בין נתונים. לדוגמה, גרף יכול לתאר קישוריות בין אתרי אינטרנט, כאשר הקודקודים הם דפי האינטרנט והקשתות הם הקישורים (links) ביניהם. ניתן באמצעות גרף לייצג גם רשת של קווי רכבת, כאשר הקודקודים הינם תחנות הרכבת, והקשתות הן קווים ישירים בין שתי תחנות. דוגמה נוספת, שמובאת בשרטוט למטה, הינה רשת של משתמשי טינדר – כאשר הקודקודים הם משתמשי האפליקציה, והקשתות מתארות סימון "לייק" של משתמש אחד למשתמש אחר. בדוגמה זו אפשר לראות ש-Bob עשה לייק ל-Alice, Carol עשתה לייק ל-Paul, וכן הלאה. כיוון ש-Bob ו-Jill שניהם עשו לייק זה לזה, האפליקציה עשויה להכיר ביניהם.



בפרוייקט זה, קודקוד בגרף מיוצג ע"י שם יחודי (מחרוזת). למען הסר ספק (בייחוד לסטודנטים מסמסטרים קודמים) יודגש שהגרף <u>איננו</u> גרף גנרי והקודקודים אינם אובייקטים כלליים, אלא הם מאופיינים על ידי השם שלהם בלבד, כאשר שם הקודקוד הוא ייחודי לכל קודקוד (כלומר, לא ייתכנו שני קודקודים באותו הגרף עם שם זהה).

התוים אשר יכולים להופיע בשם של קודקוד הם: אותיות באנגלית (קטנות וגדולות), ספרות, וכן התווים '[', ']', ו-';' (סוגריים מלבניים ונקודה-פסיק). במידה וסוגריים מרובעים מופיעים בשם של קודקוד, הם **חייבים** להיות מאוזנים, כלומר, לכל סוגר פותח '[' חייב לבוא סוגר סוגר ']' מתאים אחריו, ולהיפך, לכל סוגר סוגר '[' חייב להיות סוגר פותח '[' מתאים לפניו. לדוגמה, "`[x1[x3]]`" הוא שם קודקוד חוקי, אבל "`]x2[]x1]`" אינו חוקי. בנוסף לכך, ';' יכול להופיע רק בין סוגריים מלבניים, כלומר, "`[;;;]`" הוא חוקי, אבל "`x1[];y`" אינו חוקי.

קשת מכוונת מתוארת ע"י זוג סדור של קודקודים, מופרדים עם פסיק ומוקפים עם '<' ו-'>'. לדוגמה, הקשת `<x1,x2>` מחברת מקודקוד המקור `x1` לקודקוד המטרה `x2`.



פורמאלית, נסמן ב-$V(G)$ ו-$E(G)$ את קבוצת הקודקודים וקבוצת הקשתות של גרף G, בהתאמה. גרף $G=(V,E)$ יירשם כ- $\{v_1,v_2,\dots,v_n \mid e_1,e_2,\dots,e_m\}$, כלומר, רשימה בת n קודקודים ולאחריה רשימה בת m קשתות. למשל, מעגל של שלושה קודקודים (כמצוייר משמאל) יכול להירשם כ- `{x1, x2, x3 | <x1,x2>, <x2,x3>, <x3,x1>}`.

2.3 פעולות על גרפים

נגדיר 5 פעולות בסיסיות על גרפים:

א. + איחוד (union):

$$V(G_1+G_2)=V(G_1)\cup V(G_2) \quad ; \quad E(G_1+G_2)=E(G_1)\cup E(G_2)$$

הסבר: פעולת האיחוד בין שני גרפים מאחדת הן את קבוצות הקודקודים והן את קבוצות הקשתות של שני הגרפים. שימו לב שהקבוצות המקוריות (קודקודים או קשתות) אינן חייבות להיות זרות, אך איבר שמופיע בשניהם יופיע רק פעם אחת בתוצאה. לכן, קודקוד/קשת יהיה בגרף האיחוד אם ורק אם הוא שייך לפחות לאחד משני הגרפים.

ב. ^ חיתוך (intersection):

$$V(G_1 \wedge G_2)=V(G_1)\cap V(G_2) \quad ; \quad E(G_1 \wedge G_2)=E(G_1)\cap E(G_2)$$

הסבר: באופן דומה לאיחוד, פעולת החיתוך בין שני גרפים חותכת הן את קבוצות הקודקודים והן את קבוצות הקשתות של שני הגרפים. לכן, קודקוד/קשת יהיה בגרף החיתוך אם ורק אם הוא שייך לשני הגרפים.

ג. - הפרש (difference):

$$V(G_1-G_2)=\{v\in V(G_1) \mid v\notin V(G_2)\}$$
$$E(G_1-G_2)=\{<v,w>\in E(G_1) \mid v\in V(G_1-G_2)\wedge w\in V(G_1-G_2)\}$$

הסבר: ההפרש בין שני גרפים G_1 ו-G_2 מכיל את כל הקודקודים בגרףG_1 שאינם נמצאים בגרף G_2, ואת כל הקשתות מהגרף G_1 במידה ושני הקודקודים של הקשת הזו קיימים בגרף התוצאה.

ד. * מכפלה (product):

$$V(G_1*G_2)=\{[v;w] \mid v\in V(G_1)\wedge w\in V(G_2)\}$$
$$E(G_1*G_2)=\{<[v_1;w_1],[v_2;w_2]> \mid <v_1,v_2>\in E(G_1)\wedge <w_1,w_2>\in E(G_2)\}$$

הסבר: הקודקודים והקשתות בגרף המכפלה מתקבלים מתוך זוגות של קודקודים/קשתות מהגרפים G_1 ו-G_2. בפרט, לכל זוג אפשרי של קודקוד v מ-G_1 וקודקוד w מ-G_2, יהיה קודקוד בשם [v;w] בגרף התוצאה (שימו לב לשם הקודקוד – בתוך סוגריים מלבניים, עם נקודה-פסיק בין שמות קודקודי המקור, ללא רווחים נוספים). כמו כן, לכל זוג אפשרי של קשתות <v1,v2> מ-G_1 ו-<w1,w2> מ-G_2, תהיה קשת מהקודקוד [v1;w1] לקודקוד [v2;w2] בגרף המכפלה.

ה. ! משלים (complement):

$$V(!G)=V(G) \quad ; \quad E(!G)=\{<v,w> \mid v\in V(G)\wedge w\in V(G)\wedge <v,w>\notin E(G)\}$$

הסבר: פעולת המשלים על גרף יוצרת גרף בו קבוצת הקדקודים זהה לזו של הגרף המקורי, ואילו קבוצת הקשתות מכילה את כל הקשתות האפשריות שלא היו בגרף המקורי. כלומר, קשת $<v,w>$ תופיע בגרף המשלים אם ורק אם v ו-w הם קודקודים בגרף G, והקשת $<v,w>$ אינה קיימת בגרף המקורי.

2.4 הפעלת המחשבון

המחשבון (תוכנית בשם gcalc) פועל כמו shell. אם הוא מורץ ללא כל פרמטרים, הוא מציג את ה-prompt " Gcalc>"., ובו המשתמש מקליד פקודות ולאחריהן <ENTER>. הפקודה מבוצעת, מודפס פלט במידת הצורך, וה-prompt מוצג שוב. סוגי הפקודות האפשריים הם:

א. הגדרה (או הגדרה מחדש) של משתנה גרף. למשל:

```
Gcalc> G1={a,b | <a,b>}
Gcalc> G2=G1
```

ב. השמת התוצאה של ביטוי (הכולל אופרטור בודד) לתוך משתנה אחר. למשל:

```
Gcalc> G1 = !G0
Gcalc> G3 = G1 + G2
```

ג. הפעלת אחת מהפקודות הנתמכות ע"י המחשבון, המוגדרות בחלק הבא. למשל:

```
Gcalc> print(G4)
```

<u>שימו לב</u>:

- שם של משתנה גרף יכול להכיל תוים אלפאנומריים בלבד (אותיות או ספרות). התו הראשון חייב להיות אות.
- בכל התרגיל יש חשיבות לאותיות קטנות וגדולות (למשל, g1 ו-G1 הם משתנים שונים, x1 ו-X1 הם קודקודים שונים).
- רווחים **יכולים** להפריד בין אלמנטים של ביטוי או פקודה, אבל אסור שיופיעו בתוך שמות (של משתנה, קודקוד, פונקציה, וכו'). למשל, הפקודה "G3 = G1+ G2" מותרת, כמו גם הפקודה "print (G4)". עם זאת הפקודה "G 1 = !G2" והפקודה "prin t(G3)" אינן חוקיות.
- אם הביטוי שנמצא בצד ימין של השמה כולל שגיאה, אז ערך המשתנה בצד שמאל לא משתנה.
- בכל מקרה של שגיאה בפקודה מצד המשתמש (למשל תחביר שגוי, פונקצייה לא מוכרת, וכו') יש להדפיס הודעת שגיאה ל-stdout. הודעת השגיאה צריכה להיות בדיוק שורה אחת, ולאחריה ה-prompt צריך להיות מוצג למשתמש בשורה הבא כרגיל. הפורמט של כל הודעות השגיאה <u>חייב</u> להיות "Error: <xxx>", כאשר <xxx> הוא הודעת שגיאה אינפורמטיבית כלשהי אשר עליכם לנסח בעצמכם.
- משתנה גרף המופיע בפעם הראשונה בצד ימין של ביטוי (או כארגומנט של פקודה) מבלי שהוגדר קודם גורר שגיאה.

2.5 פקודות המחשבון

בנוסף להשמות ולאופרטורים, המחשבון תומך בפקודות הבאות:

א. `print(G)`: מדפיס את התוכן של גרף G.
הפלט כולל רשימה של קודקודי הגרף, לאחריהם הסימן $, ולבסוף רשימת הקשתות. הקודקודים והקשתות יודפסו **בסדר לקסיקוגרפי**, כל אחד בשורה נפרדת. קשת תודפס משמאל לימין כך: שם קודקוד המקור, רווח בודד, שם קודקוד המטרה. למשל:

```
Gcalc> G={x1, x2, x3 | <x1,x2>, <x2,x3>, <x3,x1>}
Gcalc> print(G)
x1
x2
x3
$
x1 x2
x2 x3
x3 x1
```

ב. `delete(G)`: מוחק את המשתנה G.

ג. `who`: מציג רשימה של כל המשתנים (גרפים) המוגדרים, כל משתנה בשורה נפרדת, **מסודרים באופן לקסיקוגרפי**. למשל:

```
Gcalc> who
G1
G2
G3
```

ד. `reset`: מוחק את כל הגרפים הקיימים. לאחר ביצוע הפקודה `reset`, המחשבון חוזר למצבו ההתחלתי.

ה. `quit`: יוצא מהתוכנית.

<u>הערה</u>: שימו לב ששמות של פונקציות הינם **מלים שמורות**, כלומר, הם אינם יכולים לשמש כשמות של משתנים.

2.6 מצב אוטומטי (batch)

התוכנית רצה במצב אוטומטי כאשר היא מורצת משורת הפקודה עם **שני ארגומנטים**, קובץ קלט וקובץ פלט. למשל:

```
gcalc input.txt output.txt
```

במצב זה אין אינטראציה עם המשתמש, **ולא מודפסים prompt-ים**. במקום זאת, התוכנית מריצה את כל הפקודות בקובץ הקלט, שורה אחר שורה (כאילו הוקלדו ע״י המשתמש), וכותבת את הפלט, כולל כל הודעות השגיאה, לתוך קובץ הפלט. התוכנית ממשיכה להריץ כל שורה בקובץ הקלט, ומפסיקה <u>רק</u> כאשר היא מגיעה לסוף הקובץ, או מגיעה לפקודת `quit`, או אם מתרחשת שגיאה פטאלית (fatal error, כלומר שגיאה שאינה מאפשרת לתוכנית להמשיך, למשל מחסור בזכרון). שימו לב שקובץ הפלט צריך להכיל את כל שורות הפלט שנוצרו ע״י התוכנית, כולל הודעות השגיאה למשתמש, למעט ה-prompts מהסוג "`Gcalc>`" שכאמור אינן מיוצרות כלל במצב אוטומטי.

3 הרצה לדוגמה

```
Gcalc> G1={x1,x2,x3,x4|<x4,x1>, <x3,x4>,<x2,x3>,<x1,x2>}
Gcalc> G2 = { x1,y1 | <x1,y1> }
Gcalc> H = G1 + G2
Gcalc> print(H)
x1
x2
x3
x4
y1
$
x1 x2
x1 y1
x2 x3
x3 x4
x4 x1
Gcalc> who
G1
G2
H
Gcalc> reset
Gcalc> print(H)
Error: Undefined variable 'H'
Gcalc> G1 {x,y|<y,x>}
Error: Unrecognized command 'G1 {x,y|<y,x>}'
Gcalc> quit
```

4 תכונות מתקדמות

בנוסף לתכונות הבסיסיות שתוארו עד כה, על המחשבון לתמוך בתכונות המתקדמות הבאות. כל תכונה **תבדק ותוערך בפני עצמה**, ויינתן לה ניקוד בנפרד. לכן, באפשרותכם לתמוך ב**קבוצה חלקית של תכונות** על מנת לקבל ניקוד חלקי.

4.1 ביטויים מורכבים

המחשבון יכול לתמוך ב**אחד או יותר** מסוגי הביטויים המורכבים הבאים. שימו לב שכל אחד מהם ייבדק בנפרד.

א. **רצף של אופרטורים**: תכונה זו מאפשרת לביטוי להכיל רצף של כמה אופרטורים. במקרה זה, כל האופרטורים הבינאריים מוערכים **משמאל לימין** (כלומר, לכולם יש את אותה הקדימות). לאופרטור האונארי '!' יש תמיד העדיפות הגבוהה ביותר, והוא פועל על הארגומט שנמצא מימינו. למשל:

```
G3 = G1+G2*G2      <-- evaluated as (G1+G2)*G2
G4 = G1+!G2*G2     <-- evaluated as (G1+(!G2))*G2
```

ב. **קבועי גרף (graph literals)**: תכונה זו מאפשרת לביטוי לערב שמות של משתנים וקבועי גרף. למשל:

```
G2 = G1 + {a,b|<a,b>}
```

ג. **סוגריים**: תכונה זו מאפשרת להשתמש בסוגריים על מנת לשנות את סדר ההערכה של רצף של אופרטורים. שימו לב שסעיף זה רלבנטי רק אם סעיף א' (רצף אופרטורים) מומש. למשל:

```
G4 = G1+(G2*G2)      <-- without the parentheses, would be evaluated as (G1+G2)*G2
```

מחשבון התומך בכל התכונות הנ"ל יאפשר לכתוב פקודות מורכבות כגון:

```
Gcalc> G4 = G3 + (G1*({a,b|<a,b>}+G2))
```

4.2 שמירה וטעינה של גרפים

תכונה זו מאפשרת לשמור ולטעון גרפים מקובץ. פורמט הקובץ שבו יישמר הגרף (גרף אחד בלבד בקובץ) מתואר למטה. הפקודות שיש לתמוך בהן בסעיף זה הן:

א. `save(G, filename)`: שומרת את הגרף G לתוך הקובץ בשם הנקוב.
הסיומת של שם הקובץ תהיה בד"כ `.gc`, אם כי זו אינה דרישה מחייבת.

ב. `G=load(filename)`: טוענת את הגרף השמור בקובץ הנקוב לתוך המשתנה G.

שימו לב ששם הקובץ בשתי הפקודות נכתב ללא מרכאות, למשל `save(G, graph.gc)`.
כמו כן, שם הקובץ אינו יכול להכיל תווי פסיק.

<u>קבצים בינאריים</u>:

הגרף יישמר בקובץ בעל פורמט <u>בינארי</u>. פורמט בינארי שונה מהפורמט הרגיל (טקסט) שהכרנו בקורס עד כה. בניגוד לקובץ טקסט, שבו כותבים וקוראים **מחרוזות** מהקובץ, בקובץ בינארי כותבים וקוראים **בתים של זיכרון** באופן ישיר. מדובר בפורמט בסיסי יותר מקובץ טקסט, ויתרונו העיקרי הוא בכך שהוא יכול לאחסן כל סוג של מידע ללא צורך בהמרה של המידע למחרוזת כדי לשמור אותו. בנוסף, הקובץ הנוצר הוא לרוב קטן יותר מקובץ טקסט.

חישבו למשל על משתנה `num` מטיפוס `int` המכיל את המספר 12345678. נניח שנרצה לשמור אותו לקובץ. במקרה של קובץ טקסט (שאליו נכתוב את `num` באמצעות פקודה כמו `outfile<<num`), ראשית המספר יומר למחרוזת, ולאחר מכן התווים "12345678" ייכתבו בזה אחר זה לקובץ. בסה"כ, הקובץ יכיל 8 תווים, ויהיה בגודל 8 בתים על הדיסק. לעומת זאת, במקרה של קובץ בינארי נכתוב את 4 הבתים של המשתנה `num` <u>כפי שהם בזיכרון</u>, ללא כל המרה, ישירות לקובץ.

במקרה הזה למשל נכתוב את רצף הביטים 10111100011000010100111 0 לקובץ (למעשה אנו נכתוב בדיוק 32 ביטים לקובץ, השמטנו כאן כמה אפסים מובילים לשם קיצור). הקובץ הנוצר יהיה לפיכך תמיד בגודל 4 בתים על הדיסק, ללא תלות בערכו של num.

כתיבה וקריאה של קובץ בינארי היא מאוד פשוטה ב-C++. ראשית, כשפותחים את הקובץ לקריאה/כתיבה, יש לציין שהקובץ הוא במצב בינארי, כך:

```
std::ifstream infile("filename", std::ios_base::binary);  // פותח את הקובץ לקריאה במצב בינארי
std::ofstream outfile("filename", std::ios_base::binary); // פותח את הקובץ לכתיבה במצב בינארי
```

שנית, קוראים וכותבים אל הקובץ ישירות מערכים של בתים (כמערכים של char*) כך:

```
infile.read(buffer, n);      // קורא n בתים מ-infile לתוך המערך buffer מטיפוס char*
outfile.write(buffer, n);    // כותב n בתים מהמערך buffer (מטיפוס char*) לתוך הקובץ outfile
```

לדוגמה, ניתן לקרוא ולכתוב את המשתנה int num לקובץ בינארי פתוח על ידי המרתו למערך בתים באורך 4, וקריאה או כתיבה של רצף הבתים הזה ישירות:

```
infile.read((char*)&num, sizeof(int));
outfile.write((const char*)&num, sizeof(int));
```

מבנה הקובץ של גרף:

כאמור, גרף יאוחסן בקובץ בינארי, כאשר כל המידע של הגרף (הקודקודים והקשתות שלו) יאוחסנו כרצף בתים אחד ארוך. מבנה הקובץ יהיה כדלקמן (כאשר כל ביטוי בתוך <> מייצג רצף של בתים המופיעים בזה אחר זה בקובץ):

<num_vertices><num_edges><vertex1><vertex2>…<vertexN><edge1><edge2>…<edgeM>

הסבר:

<num_vertices> הוא מספר הקודקודים בגרף (4 בתים, unsigned int).
<num_edges> הוא מספר הקשתות בגרף (4 בתים, unsigned int).
<vertex_i> הוא שם הקודקוד ה-i-י. ארבעת הבתים הראשונים יכילו את מספר התוים במחרוזת (ללא NULL) (4 בתים, unsigned int), ושאר הבתים יכילו את שם הקודקוד עצמו כרצף של תוים (תו אחד בכל בית, ללא NULL).
<edge_j> הוא הקשת ה-j-ית. מיוצגת ע"י זוג שמות של קודקודים (מקור ומטרה) כפי שתואר לעיל (כלומר, כל שם מורכב ממספר התווים + התווים עצמם, ללא NULL או תו מפריד אחר).

4.3 פונקציות עם ארגומנטים כלליים

תכונה זו מאפשרת לפונקציה print (ואופציונלית גם לפונקצייה save אם בחרתם לממש את 4.2) לקבל *כל* ביטוי חוקי, ולא רק שם של משתנה. למשל:

```
print(G1+G2)
print(!G3)
print({a,b|<a,b>} * {c,d|<c,d>})     <-- If advanced expressions are implemented
save({a,b|<a,b>}, graph.gc)          <-- If save/load were implemented
```

כמו בסעיפים קודמים, גם כאן כל אופציה כזו תבדק בנפרד – כלומר, ניתן לממש את התכונה הזו רק עבור ביטויים בסיסיים, גם עבור ביטויים מורכבים, עבור הפונקצייה print ו/או עבור הפונקצייה save (אם רלוונטי).

4.4 ממשק ל-Python

תכונה זו מאפשרת ליצור גרפים ולבצע פעולות פשוטות עליהם מתוך פייתון. בפרט, אנו נרצה לחשוף את הפעולות על גרפים מסעיף 2.3 כך שניתן יהיה להשתמש בהן ישירות מ-python, תוך שימוש ב-SWIG כפי שנלמד. עליכם לספק את הפונקציות הבאות, שייקראו מתוך python:

א. `create()`: תחזיר מצביע לגרף חדש.

ב. `destroy(graph)`: תשחרר את הגרף שיתקבל כפרמטר.

ג. `addVertex(graph, v)`: תוסיף את הקודקוד v (הנתון כמחרוזת) לגרף. הפונק' תחזיר את graph (כדי לאפשר שרשור פעולות).

ד. `addEdge(graph, v1, v2)` – תוסיף את הקשת <v1,v2> לגרף (v1,v2 מחרוזות שמציינות את שמות הקודקודים). הפונק' תחזיר את graph (כדי לאפשר שרשור פעולות).

ה. `disp(graph)` – תדפיס את הגרף ל-stdout בפורמט של פקודת print (מסעיף 2.5א).

ובנוסף, את הפונקציות הבאות שמבצעות את הפעולות על גרפים המתוארות בסעיף 2.3. הפרמטר האחרון של כל אחת מפונקציות אלו יהיה מצביע לגרף (שנוצר מראש) שלתוכו תיכתב **תוצאת הפעולה** (ותדרוס את התוכן הנוכחי של הגרף). על כל פונקציה להחזיר את הפרמטר graph_out על מנת לאפשר שרשור פעולות:

ו. `graphUnion(graph_in1, graph_in2, graph_out)`

ז. `graphIntersection(graph_in1, graph_in2, graph_out)`

ח. `graphDifference(graph_in1, graph_in2, graph_out)`

ט. `graphProduct(graph_in1, graph_in2, graph_out)`

י. `graphComplement(graph_in, graph_out)`

<u>הערות</u>:

- במקרה של כישלון עליכם להדפיס ל-stdout הודעת שגיאה בת בדיוק שורה אחת. בדומה למחשבון, הפורמט של כל הודעות השגיאה <u>חייב</u> להיות "Error: <xxx>", כאשר <xxx> הוא הודעת שגיאה אינפורמטיבית כלשהי אשר עליכם לנסח בעצמכם. לאחר הדפסת הודעת השגיאה, הפונקציה תסתיים כרגיל (פונקציה שמחזירה גרף תחזיר nullptr במקום כדי לציין שקרתה שגיאה). בפרט, שגיאה בפונקציית ממשק לא תגרור סגירה של תהליך ה-python.
- על הממשק שאתם חושפים ל-python לתמוך בכל הכללים של שמות הקודקודים והגדרות הפעולות כפי שפורטו מעלה עבור המחשבון.

<u>הנחיות הידור</u>:

יצירת קובץ הספרייה (.so) עבור python תעבוד בשלושה שלבים:

א. יצירה של קובץ libgraph.a שיכיל בתוכו את כל קבצי ה-o. הנדרשים לשלבים הבאים (קוד המימוש של המחלקה שמתארת גרף וייתכן ממשק מעטפת עבורה). כלל ה-makefile המתאים (כאשר OBJS הוגדר לרשימת קבצי ה-o. הנדרשים):

```
libgraph.a: $(OBJS)
        ar -rs $@ $^
```

(אתם כמובן יכולים גם להשתמש בפקודה ar מחוץ ל-makefile).

ב. יצירה של ממשק python ע"י swig. עליכם להגיש את הקובץ graph.i (ראו פרק הוראות הגשה).

```
swig -python graph.i -o graph_wrap.c
```

ג. יצירת קובץ הספריה עבור python מתוך הממשק הפייתון וקובץ הספריה שיצרנו בסעיפים הקודמים:

```
g++ -DNDEBUG --pedantic-errors -Wall -Werror -I/usr/local/include/python3.6m
-fPIC -shared graph_wrap.c libgraph.a -o _graph.so
```

<u>שימו לב</u>: השורה יוצרת קובץ ספריה עבור python3.6. ניתן להפעיל אותו בשרת ע"י הקלדת python3.6 כמו בדוגמה מטה.

הרצה לדוגמה של ממשק ה-python:

```
[mtm@csl3 ~]$ python3.6
Python 3.6.3 (default, Jan 27 2019, 09:41:42)
[GCC 4.8.5 20150623 (Red Hat 4.8.5-28)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import graph as gw #graph wrapper
>>> g1 = gw.create()
>>> gw.disp(g1)
$
>>> g1 = gw.addVertex(g1, 'a')
>>> g1 = gw.addVertex(g1, 'b')
>>> gw.disp(gw.addEdge(g1, 'a', 'b'))
a
b
$
a b
>>> gw.addEdge(g1, 'a', 'c')
Error: Cannot add edge <a,c> due to missing endpoint.
>>> g2 = gw.create()
>>> g2 = gw.addVertex(g2, 'c')
>>> gw.disp(gw.graphUnion(g1, g2, g1))
a
b
c
$
a b
>>> gw.disp(g1)
a
b
c
$
a b
>>> gw.destroy(g2)
>>> gw.destroy(g1)
```

5 הגשת טסט (קבצי בדיקה)

בנוסף לקוד הפרוייקט, על כל הגשה לכלול טסט אחד של התוכנית שהגשתם (קובץ קלט וקובץ פלט), המדגים שימוש בתוכנה שלכם, כולל בדיקות של תכונות המחשבון, מקרי קצה וכן הלאה. מומלץ להרכיב את קובץ הטסט במהלך פיתוח התוכנית שלכם, מבדיקות שאתם עורכים באופן שגרתי.

קבצי הטסט ייקראו בשמות test_in.txt ו-test_out.txt, ויכללו בספרייה הראשית של הפרוייקט, ליד הקוד. קובץ הקלט יכול לכלול לכל היותר 100 שורות. הטסטים יורצו משורת הפקודה כך:

```
gcalc test_in.txt test_out.txt
```

הציון עבור חלק זה הינו **5% מהציון הסופי**, והוא יחושב בצורה של "**תחרות" בין הפרוייקטים**. כלומר, כל טסט ייבדק כנגד ההגשות האחרות של הפרוייקט, וציון גבוה יותר יינתן ככל שהוא יצליח **להכשיל** הגשות רבות יותר. כמובן שהטסט נדרש להיות נכון – כלומר, עליו להצליח על התוכנית שלכם, וכן על פתרון "בית ספר" שנכתב על ידי צוות הקורס. מומלץ שקובץ הטסט יקיף מצבים שונים ככל הניתן, וייתמקד במקרים המורכבים יותר שהתוכנה שלכם צפויה להתקל בהם.

6 מסמך תיאור הפרויקט

כל הגשה של הפרוייקט צריכה להיות מלווה בקובץ PDF קצר המתאר את התכן (design) הכללי של הפרויקט. הקובץ צריך לכלול את שני החלקים הבאים:

א. דיאגרמת מחלקות: דיאגרמה המראה את המחלקות העיקריות בפרויקט ואת הקשרים ביניהן. על הדיאגרמה לכלול רק את המחלקות העיקריות המהוות את הלוגיקה המרכזית של התוכנית, ולא מחלקות תמיכה קטנות כמו חריגות (exceptions). באפשרותכם להשתמש ב-PowerPoint, Visio, או כל תוכנה אחרת להכנת הדיאגרמה.

ב. תיאור התכן: סקירה מילולית של תכן הפרויקט, המסבירה בקצרה את המחלקות המופיעות בדיאגרמת המחלקות, את התפקידים שלהן, והקשרים ביניהן. הסקירה צריכה להיות לא ארוכה (1-2 פסקאות, כחצי עמוד), ויכולה להיות כתובה באנגלית או בעברית.

קובץ התכן אמור להיות קצר – לרוב עמוד אחד, ועד 2 עמודים במידה והדיאגרמה תופסת מקום רב.

7 הגשה

7.1 קומפילציה

על הפרויקט להיות ניתן לקמפול ע"י קובץ `Makefile` בדיוק בשם זה. על קובץ ה-makefile ליצור קובץ הרצה (executable) בשם `gcalc`. ה-makefile צריך לכלול את האפשרויות הבאות:

א. `make clean`: ביטול כל קובצי קוד מכונה (object) והרצה (executable), וקבצים זמניים של Unix.
ב. `make`: יצירת קובץ ההרצה `gcalc`.
ג. `make libgraph.a`: יצירת קובץ הספריה כשלב ביניים ליצירת ממשק פייתון.
ד. `make tar`: יצירת קובץ zip (בשם `gcalc.zip`) הכולל את כל הקבצים המוגשים.

שימו לב שעל ה-makefile לבנות מחדש רק את הקבצים אשר אינם מעודכנים.

7.2 פרוצדורה

על קובץ ההגשה להיות במבנה

```
gcalc.zip/
├── design.pdf
├── Makefile
├── test_in.txt
├── test_out.txt
└── ... other files
```

הנחיות:

- עליכם להגיש **באתר הסמסטר שלכם**.
- שימו לב: השתמשו אך ורק ב-zip. פורמט אחר לא יתקבל. אין חשיבות לשם קובץ ה-zip המוגש (gcalc.zip בדוגמאות מעלה).
- מותר להגיש את הפרויקט מספר פעמים. **רק ההגשה האחרונה נחשבת**.
- על מנת לבטח את עצמכם נגד תקלות בהגשה האוטומטית **שימרו תמונת מסך ה-webcourse עם קוד האישור עבור ההגשה**.
- **בכל קוד הפרוייקט, אין צורך לעמוד ב-code convensions כלשהם (אם כי הדבר כמובן מומלץ על מנת לשפר את קריאות הקוד ואת יעילות העבודה).**

8 רמזים

א. ניתן ומומלץ להשתמש ב**מבני נתונים של STL** במימוש שלכם. מבני נתונים רלבנטיים במיוחד לפרויקט הם `std::vector`, `std::set`, ו-`set::map`. העדיפו להשתמש במבני נתונים קיימים על פני מימושם מחדש, על מנת לפשט את הקוד שלכם ולמנוע באגים.

ב. ניתן להעזר בעובדה ש-`std::set` מאחסנת את האברים שלה בסדר עולה. גם `std::map` שומרת את האברים בסדר עולה (לפי ערך המפתח). האיטרטורים של מבנים אלו מחזירים את האברים המאוחסנים בסדר זה.

ג. מחלקות כמו `std::string`, `std::vector`, ו-`std::map` מספקות פונקציונליות גם מעבר למה שלמדנו בכיתה. רצוי ומומלץ **להשתמש בreferences של C++ במהלך מימוש פרויקט זה** על מנת לגלות אילו פונקציות נוספות הן מספקות. באפשרותכם למצוא גם אינפורמציה ספציפית (כמו פרמטרים של פונקציות, ערכים מוחזרים, חריגות, וכו'), בתיעוד זה.

ד. קובץ ה-header הסטנדרטי `<cctype>` מספק מספר פונקציות שימושיות לעבודה עם תוים, כדאי לבדוק אותו.

ה. השתמשו בטבלת סמלים (symbol table), מטיפוס `std::map`, כדי לאחסן את שמות כל המשתנים הידועים למחשבון.

ו. פונקציות נוספות מהספרייה של C++ אשר עשויות להיות שימושיות לכם עבור פרויקט זה, כוללות את הפונקצייה `std::to_string()` (ממירה מספר למחרוזת) והפונקצייה `std::getline()` (קוראת שורה בודדת מקובץ פתוח או מ-`stdin` לתוך מחרוזת).

ז. שימו לב שכל הודעות השגיאה השוטפות המיועדות למשתמש (פקודות לא חוקיות ובעיות דומות אחרות) צריכות להיכתב ל-`stdout` **ולא** ל-`stderr`. רק **שגיאות פטאליות**, אשר לא יכולות להיות מטופלות כהלכה ודורשות את הפסקת ריצת התוכנית, צריכות להיכתב ל-`stderr`. כאשר התוכנית מורצת במצב אוטומטי (batch mode), כל ההודעות המודפסות ל-`stdout` צריכות להיות מופנות לקובץ הפלט, למעט הודעות שגיאה פטאליות (המודפסות ל-`stderr`) שצריכות עדיין להיות מודפסות אל המסך ולא לקובץ הפלט.

בהצלחה!