בס"ד

FINAL PROJECT + EXC 4 OPERATING SYSTEMS
REI SHAUL 325390086
RON AVRAHAM 208007005

Part 1-

כדי לייצר מבנה נתונים של גרף, השתמשנו במטריצת שכנויות.(adj_list בתכנית)

Part 2-

כדי לבדוק אם קיים מעגל אוילר ראשית נבדוק שכל הקודקודים בגרף בדרגה זוגית וגם שאין קודקודים מבודדים. אחרת אין מעגל אוילר.

כדי לקבל את המעגל עצמו, כלומר את הקודקודים לפי הסדר בוקטור בנינו פונקציה הממשת את האלגוריתם של Hierholzer ע"י שימוש במחסנית (get_eulerian_cycle בתכנית).

Part 3-

קומפילציה של התכונית:

ע"י מייק פייל נריץ make בטרמינל הפנימי parts1to4/:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ make
```

הקבצים שנוצרו

```
euler    U
```

צילום מסך של יצירת גרף עם 4 קודקודים, 4 צלעות וסיד 3 ע"י שימוש בדגלים:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ ./euler -v 4 -e 4 -s 3
Graph with 4 vertices and 4 edges created
Node 0: (2, weight: 1) (1, weight: 1)
Node 1: (0, weight: 1) (3, weight: 1)
Node 2: (0, weight: 1) (3, weight: 1)
Node 3: (2, weight: 1) (1, weight: 1)
Eulerian circuit found: 0 1 3 2 0
```

ניתן לראות כי נוצר גרף אקראי כזה שיש בו מסלול אוילר ואכן המסלול שהוחזר ע"י האלגוריתם נכון.

צילום מסך עבור מקרה שהגרף האקראי שנוצר אינו מכיל מעגל אוילר:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ ./euler -v 4 -e 4 -s 2
Graph with 4 vertices and 4 edges created
Node 0: (1, weight: 1) (3, weight: 1) (2, weight: 1)
Node 1: (0, weight: 1)
Node 2: (3, weight: 1) (0, weight: 1)
Node 3: (0, weight: 1) (2, weight: 1)
terminate called after throwing an instance of 'std::runtime_error'
  what():  Eulerian circuit does not exist
Aborted (core dumped)
rei@Rei:~/OS FINAL PROJECT/parts1to4$
```

ניתן לראות כי נזרקה שגיאה מתאימה עבור מקרה זה.

בס"ד

Part 4-

Code coverage:
compilation:

לאחר שביצענו make clean
נריץ בטרמינל בתוך התיקיה parts1to4/:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ make coverage
```

נשים לב שהקבצי .gcno נוצרו:

```
graph.gcno   U
```

```
main.gcno   U
```

וכן הקובץ

```
euler   U
```

ואז נריץ מספר מקרים שונים ומגוונים (כדי לכסות כמה שיותר שורות קוד ופונקציות) שכבר הכנו בmake file:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ make coverage-run
```

השרת יחזיר פלט מתאים:

```
rm -f euler graph.o main.o \
      gmon.out gprof_report.txt valgrind_report.txt \
      callgrind.out.* \
          *.gcda *.gcno *.gcov coverage.info
rm -rf coverage_report
make CXXFLAGS="-g -Wall -fprofile-arcs -ftest-coverage" LDFLAGS="-lgcov" euler
make[1]: Entering directory '/home/rei/OS FINAL PROJECT/parts1to4'
g++ -g -Wall -fprofile-arcs -ftest-coverage -c graph.cpp -o graph.o
g++ -g -Wall -fprofile-arcs -ftest-coverage -c main.cpp -o main.o
g++ -g -Wall -fprofile-arcs -ftest-coverage -o euler graph.o main.o
make[1]: Leaving directory '/home/rei/OS FINAL PROJECT/parts1to4'
./euler -v 4 -e 4 -s 3
Graph with 4 vertices and 4 edges created
Node 0: (2, weight: 1) (1, weight: 1)
Node 1: (0, weight: 1) (3, weight: 1)
Node 2: (0, weight: 1) (3, weight: 1)
Node 3: (2, weight: 1) (1, weight: 1)
Eulerian circuit found: 0 1 3 2 0
some changes in the graph:
Removing edge (0, 2)
Edge removed successfully.
```

ואז נריץ פקודה כדי לקבל את הסיכום של gcov:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ make coverage-report
```

נסתכל על החלקים הרלוונטים של תוצאות הכיסוי:

```
File 'main.cpp'
Lines executed:97.83% of 92
Creating 'main.cpp.gcov'
```

```
File 'graph.cpp'
Lines executed:99.13% of 115
Creating 'graph.cpp.gcov'
```

כחלק מהתיעוד השארתי את קבצי main.cpp.gcov, graph.cpp.gcov בתיקיה gcov_files

התקבל כיסוי של 97% משורות הקוד בקובץ main.cpp
וכיסוי של 99% משורות הקוד בקובץ Graph.cpp

בס"ד

מחיקת קבצי gcno, gcda ו gcov

ע"י make clean:

```
rei@Rei:~/OS FINAL PROJECT/parts1to4$ make clean
```

Profiling:

קומפילציה עד דגל פרופיל:

```
rei@Rei:~/OS FINAL PROJECT$ g++ -pg -o euler Graph.cpp main.cpp
```

הרצה:

```
rei@Rei:~/OS FINAL PROJECT$ ./euler -v 4 -e 4 -s 123
```

יצירת דו"ח gprof:

```
rei@Rei:~/OS FINAL PROJECT$ gprof euler gmon.out > gprof_report.txt
```

```
gmon.out          U
gprof_report.txt  U
```

להלן קטע מהקובץ gprof_report.txt :

```
Flat profile:

Each sample counts as 0.01 seconds.
 no time accumulated

  %   cumulative   self              self     total
 time   seconds   seconds    calls  Ts/call  Ts/call  name
  0.00      0.00     0.00      624     0.00     0.00  std::__detail::_Mod<unsigned long, 4294967296ul, 1ul, 0ul, true, true>::__calc(unsigned long)
  0.00      0.00     0.00      624     0.00     0.00  unsigned long std::__detail::__mod<unsigned long, 4294967296ul, 1ul, 0ul>(unsigned long)
  0.00      0.00     0.00      623     0.00     0.00  std::__detail::_Mod<unsigned long, 624ul, 1ul, 0ul, true, true>::__calc(unsigned long)
  0.00      0.00     0.00      623     0.00     0.00  unsigned long std::__detail::__mod<unsigned long, 624ul, 1ul, 0ul>(unsigned long)
  0.00      0.00     0.00      166     0.00     0.00  int const& std::forward<int const&>(std::remove_reference<int const&>::type&)
  0.00      0.00     0.00      132     0.00     0.00  __gnu_cxx::__normal_iterator<graph::Graph::Edge const*, std::vector<graph::Graph::Edge, std::
  0.00      0.00     0.00       94     0.00     0.00  __gnu_cxx::__normal_iterator<graph::Graph::Edge*, std::vector<graph::Graph::Edge, std::alloca
  0.00      0.00     0.00       90     0.00     0.00  int& std::forward<int&>(std::remove_reference<int&>::type&)
  0.00      0.00     0.00       77     0.00     0.00  __gnu_cxx::__normal_iterator<graph::Graph::Edge*, std::vector<graph::Graph::Edge, std::alloca
  0.00      0.00     0.00       74     0.00     0.00  __gnu_cxx::__normal_iterator<std::tuple<int, int, int>*, std::vector<std::tuple<int, int, int
  0.00      0.00     0.00       72     0.00     0.00  graph::Graph::Edge* std::__niter_base<graph::Graph::Edge*>(graph::Graph::Edge*)
  0.00      0.00     0.00       72     0.00     0.00  std::tuple<int, int, int>* std::__niter_base<std::tuple<int, int, int>*>(std::tuple<int, int,
  0.00      0.00     0.00       64     0.00     0.00  bool __gnu_cxx::operator!=<graph::Graph::Edge const*, std::vector<graph::Graph::Edge, std::al
  0.00      0.00     0.00       62     0.00     0.00  __gnu_cxx::__normal_iterator<graph::Graph::Edge const*, std::vector<graph::Graph::Edge, std::
  0.00      0.00     0.00       60     0.00     0.00  __gnu_cxx::__normal_iterator<std::tuple<int, int, int>*, std::vector<std::tuple<int, int, int
  0.00      0.00     0.00       58     0.00     0.00  std::vector<graph::Graph::Edge, std::allocator<graph::Graph::Edge> >::size() const
  0.00      0.00     0.00       56     0.00     0.00  unsigned long const& std::min<unsigned long>(unsigned long const&, unsigned long const&)
  0.00      0.00     0.00       50     0.00     0.00  operator new(unsigned long, void*)
  0.00      0.00     0.00       48     0.00     0.00  std::vector<std::tuple<int, int, int>, std::allocator<std::tuple<int, int, int> > >::size() c
  0.00      0.00     0.00       48     0.00     0.00  graph::Graph::Edge&& std::forward<graph::Graph::Edge>(std::remove_reference<graph::Graph::Edg
  0.00      0.00     0.00       40     0.00     0.00  std::vector<std::vector<graph::Graph::Edge, std::allocator<graph::Graph::Edge> >, std::alloca
  0.00      0.00     0.00       36     0.00     0.00  std::uniform_int_distribution<int>::param_type::a() const
```

ע"י סיכום זה ניתן לנתח כמה פעמים כל פונקציה נקראה, כמה אחוז מהזמן שהתכנית ארכה הפונקציה לקחה וכ"ו.

בס"ד

Valgrind Memcheck:

הרצת התכנית ולאחר מכן הרצת valgrind:

```
rei@Rei:~/OS FINAL PROJECT$ g++ -g -o euler Graph.cpp main.cpp
rei@Rei:~/OS FINAL PROJECT$ valgrind --leak-check=full --show-leak-kinds=all --track-origins=yes ./euler -v 4 -e 4 -s 123
```

סיכום ההרצה:

```
==198816== HEAP SUMMARY:
==198816==     in use at exit: 0 bytes in 0 blocks
==198816==   total heap usage: 44 allocs, 44 frees, 77,565 bytes allocated
==198816==
==198816== All heap blocks were freed -- no leaks are possible
==198816==
==198816== For lists of detected and suppressed errors, rerun with: -s
==198816== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
rei@Rei:~/OS FINAL PROJECT$
```

ניתן לראות שאין דליפות זיכרון וניהול הזיכרון עבר בהצלחה.

Valgrind Callgrind (Call Graph):

נריץ את הפקודה הבאה:

```
rei@Rei:~/OS FINAL PROJECT$ valgrind --tool=callgrind ./euler -v 4 -e 4 -s 123
```

סיכום ההרצה:

```
==198995==
==198995== Events    : Ir
==198995== Collected : 2198971
==198995==
==198995== I   refs:      2,198,971
rei@Rei:~/OS FINAL PROJECT$
```

Events:Ir - זה סופר קריאות לזיכרון

Collected: 2,198,971 - מספר הפעמים שהוראות התכנית נקראו

Part 6-

השרת, מקבל קלט של גרף ע"י קריאה מקובץ בפורמט:
V <vertex num> E <edges num>

<u> <v>

<u> <v>

<u> <v>

בס"ד

דוגמה לשימוש:

ניצור לדוגמה קובץ graph.txt :

```
graph.txt    U
```

עם התוכן לפי הפורמט:

```
graph.txt
1  V 4 E 4
2  0 1
3  0 2
4  1 3
5  2 3
6
```

נקמפל את הקבצים:

```
rei@Rei:~/OS FINAL PROJECT$ g++ -std=c++17 -Wall -O2 server.cpp Graph.cpp -o server
```

נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

נשים לב שהוא מאזין על פורט 5555

נפתח טרמינל נוסף ונריץ את הפקודה הבאה:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < graph.txt
OK 0 2 3 1 0
rei@Rei:~/OS FINAL PROJECT$
```

הדגל q- אומר לסגור את החיבור לאחר שניה 1. קיבלנו שהגרף מאושר כלומר יש בו מעגל אוילר ובנוסף קיבלנו את המעגל עצמו כמצופה

כמו כן, השרת ממשיך להאזין לקלט הבא הוא לא נסגר:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

Part 7: We are choosing to implement this algorithms: Finding MST weight, Finding Hamilton circuit, Finding Max Clique, Finding max flow between node 0 (source) and node (n-1).

נראה שהשרת מקבל קלט בפורמט:

ALG <algorithm name>

V <vertex num> E <edges num>

<u> <v>

<u> <v>

<u> <v>

בס"ד

דוגמה לשימוש:

ניצור לדוגמה קובץ EULER.txt :

algraph.txt M

תוכן הקובץ:

```
algraph.txt
ALG EULER
V 4 E 4
0 1
0 2
2 3
1 3
2 3
```

נקמפל את התכנית ע"י פקודת make :

```
rei@Rei:~/OS FINAL PROJECT$ make
```

server U

נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחר נריץ כך: (הורדנו את האלגוריתם euler מחלק זה של הפרויקט- חלק זה מתמקד בארבעת האלגוריתמים- MST, MAXFLOW,MAXCLIQUE,HAMILTON יש להריץ בקשה לדוגמה עם אלגוריתם אחר).

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < EULER.txt
OK 0 2 3 1 0
```

ואכן קיבלנו את הפלט הנכון מעגל אוילר כפי שציפינו.

בצד השרת- הוא ממשיך להאזין לבקשה הבאה:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

דוגמה עבור אלגוריתם find Hamilton cycle:

ניצור לדוגמה קובץ HAM.txt :

HAM.txt U

תוכן הקובץ:

```
HAM.txt
ALG HAMILTON
V 5
E 7
0 1
1 2
2 3
3 4
4 0
0 2
1 3
```

נקמפל את התכנית ע"י פקודת make :

```
rei@Rei:~/OS FINAL PROJECT$ make
```

server U

בס"ד

נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחר נריץ כך:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < HAM.txt
OK 0 1 2 3 4 0
rei@Rei:~/OS FINAL PROJECT$
```

ואכן קיבלנו את הפלט הנכון מסלול מעגל המילטון כפי שציפינו.

כמו כן, השרת ממשיך להאזין לקלט הבא הוא לא נסגר:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

דוגמה עבור אלגוריתם Finding MST weight:

ניצור לדוגמה קובץ MST.txt :

```
MST.txt          U
```

תוכן הקובץ:

```
MST.txt
ALG MST
V 4
E 5
0 1 1
0 2 3
1 2 1
1 3 4
2 3 2
```

נקמפל את התכנית ע"י פקודת make :

```
rei@Rei:~/OS FINAL PROJECT$ make
```

```
server           U
```

נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחר נריץ כך:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < MST.txt
OK MST WEIGHT: 4
rei@Rei:~/OS FINAL PROJECT$
```

ואכן קיבלנו את הפלט הנכון המשקל המינימלי עבור עץ פורש של הגרף כפי שציפינו.

כמו כן, השרת ממשיך להאזין לקלט הבא הוא לא נסגר:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בס"ד

דוגמה עבור אלגוריתם Finding max flow between node 0 (source) and node (n-1):

ניצור לדוגמה קובץ MAXFL.txt :

MAXFL.txt U

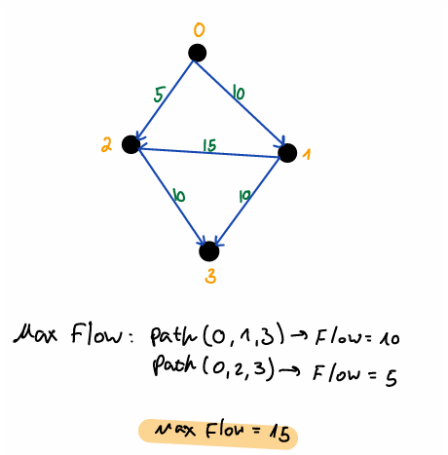
תוכן הקובץ:

```
MAXFL.txt
ALG MAXFLOW
V 4
E 5
0 1 10
0 2 5
1 2 15
1 3 10
2 3 10
```

נקמפל את התכנית ע"י פקודת make :

```
rei@Rei:~/OS FINAL PROJECT$ make
```

server U



נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחר נריץ כך:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < MAXFL.txt
OK MAXFLOW 15
```

ואכן קיבלנו את הפלט הנכון הזרימה המקסימלית שניתן להזרים בגרף הנתון היא 15 יחידות מידה כפי שציפינו.

כמו כן, השרת ממשיך להאזין לקלט הבא הוא לא נסגר:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

דוגמה עבור אלגוריתם Finding Max Clique:

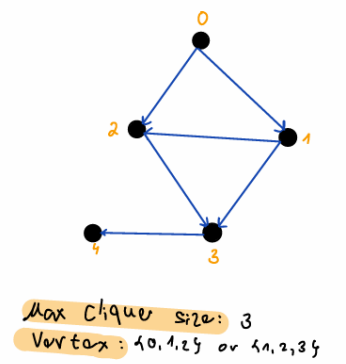
ניצור לדוגמה קובץ MAX CLIQUE.txt :

MAXCLIQUE.txt U

תוכן הקובץ:

```
MAXCLIQUE.txt
ALG MAXCLIQUE
V 5
E 6
0 1
0 2
1 2
1 3
2 3
3 4
```

הגרף בצורה ויזואלית:



בס"ד

נקמפל את התכנית ע"י פקודת make :

```
rei@Rei:~/OS FINAL PROJECT$ make
```

```
server    U
```

נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחר נריץ כך:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < MAXCLIQUE.txt
OK MAXCLIQUE SIZE 3
CLIQUEMEMBERS: 0 1 2
rei@Rei:~/OS FINAL PROJECT$
```

ואכן קיבלנו את הפלט הנכון הקליקה המקסימלית הינה בגודל 3 והיא מורכבת מקודקודים {1,2,3} כפי שציפינו.

כמו כן, השרת ממשיך להאזין לקלט הבא הוא לא נסגר:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

Part 8-

a+b:

עכשיו כל חיבור מלקוח צריך להיות מטופל ע"י thread נפרד, במקום שה server יטפל בלקוח אחד בלבד בכל רגע.

כדי לאפשר זאת- נחליף את השורות קוד האלו:

```
handleClient(cfd); // Handle the client connection

close(cfd); // Close the client connection
```

בשורות:

```
    std::thread t([cfd]() {

      handleClient(cfd);

      close(cfd);

    });

    t.detach(); // Run in the background without join
```

//this 4 lines are for creating a new thread to handle the client connection

בס"ד

נראה כעת שני בקשות לשרת משני טרמינלים שונים בקשה אחת תביא גרף ספיציפי עם פירוט הקודקודים והצלעות, ובקשה שניה תבוא עם מספר קודקודים וצלעות ותבקש ליצור גרף רנדומלי:
ניצור קבצים מתאימים לקריאה:

```
GRAPH.txt
GRAPH
V 4
E 5
0 1 10
0 2 5
1 2 15
1 3 10
2 3 10
```

GRAPH.txt U

```
RANDOM.txt
RANDOM
V 10
E 15
```

RANDOM.txt U

נקמפל עם make
ונריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחד נריץ:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < GRAPH.txt
OK MST WEIGHT: 25
OK MAX FLOW 15
OK HAM VERTEX: 0 1 3 2 0
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 1 2
rei@Rei:~/OS FINAL PROJECT$
```

ובטרמינל שני נריץ:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 39
OK MAX FLOW 16
OK HAM VERTEX: 0 4 2 6 5 7 1 8 3 9 0
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 4 9
```

נשים לב, שכל ארבעת האלגוריתמים הורצו על הגרפים ובכל שורה יש את תוצאת האלגוריתם כך שבשורה הראשונה תוצאת אלגוריתם MST, בשורה השנייה תוצאת האלגוריתם MAX FLOW, בשורה השלישית תוצאת אלגוריתם HAMILTON, ובשורה הרביעית תוצאת האלגוריתם MAX CLIQUE

הערה- כדי לבדוק שאכן שני הבקשות רצו במקביל זו לזו השתמשתי בשורה הזו שהוספתי לפונקציה

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 39
OK MAX FLOW 2
ERR NO HAMILTONIAN CYCLE
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 6 4
rei@Rei:~/OS FINAL PROJECT$
```

בס"ד

handle client שבקוד השרת:
std::this_thread::sleep_for(std::chrono::seconds(5));

למעשה, עבור שרת התומך במולטי טרד שני הבקשות יסתיימו לאחר 5 שניות. עבור שרת שלא תומך במולטי טרד, בקשה אחת תסתיים לאחר 5 שניות והבקשה השנייה תסתיים לאחר 5 שניות נוספות, סך הכל שני הבקשות יסתיימו לאחר 10 שניות כי הם לא רצו במקביל.

ייתכן שנקבל שהגרף הרנדומלי שנוצר או אפילו שהגרף שנתנו כקלט אינו מכיל מעגל המילטוני שזה הגיוני ואז נקבל הודעה מתאימה:

Part 9-

ממשיכים עם השרת כמו בסעיף 8 שמקבל קלט בונה גרף ומריץ את ארבעת האלגוריתמים רק שכעת ישנם לפחות 4 אובייקטים פעילים שכל אחד אחראי להריץ אלגוריתם אחד מהארבעה HAMILTON,MAXCLIQUE,MAXFLOW,MST
נבנה צינור שיפרק את התהליך לכמה שלבים כך שכל שלב רץ בטרד נפרד(אובייקט פעיל). כל שלב מקבל קלט, עושה את החישוב שלו ומעביר את הפלט לשלב הבא.
תחת name space של גרף מימשנו מחלקה threadpool עם פונקציות stageWorker, sinkWorker
כאשר הראשונה מקבלת "עבודה" מתוך התור מריצה את האלגוריתם שלה ושולחת את התוצאה לתור הבא. והאחרונה היא למעשה השלב האחרון בתור הפייפליין- היא אוספת את התוצאה הסופית ומגדיר אותה ב promise. ה future ב hendle client מחכה לסיום ה sink_worker כדי לקבל את התוצאה וממשיך לשלוח אותה ללקוח.
כמו כן, לכל אלגוריתם יש active object אחד שמריץ את העבודה שלו על טרד נפרד.
לבסוף שילבנו את זה בתכנית השרת כתחלופה לשיטה הקודמת שבה עברנו בלולאה על כל אלגוריתם שביצע את עבודתו.

נקמפל ע"י make
ונריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
```

בטרמינל אחר נריץ :

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < GRAPH.txt
OK MST WEIGHT: 25
OK MAX FLOW 15
OK HAM VERTEX: 0 1 3 2 0
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 1 2
rei@Rei:~/OS FINAL PROJECT$
```

בס"ד

נשים לב, שאכן קיבלנו את התוצאות עבור כלל האלגוריתמים שרצו על הגרף.
כדי לוודא שאכן כל האלגוריתמים רצו במקביל זה לזה
הוספתי את השורות האלו בפונקציה stage_worker

```
std::cout << "[" << algName << "] starting job on thread "
        << std::this_thread::get_id() << std::endl;
```

וזה כדי לעקוב אחר המספר המזהה עבור כל טרד.
עבור הרצה זו, נראה שבצד השרת קיבלנו:

```
[MST] starting job on thread 131466767886016
[MST] running on thread 131466767886016
[MAXFLOW] starting job on thread 131466759493312
[MAXFLOW] running on thread 131466759493312
[HAMILTON] starting job on thread 131466751100608
[HAMILTON] running on thread 131466751100608
[MAXCLIQUE] starting job on thread 131466667226816
[MAXCLIQUE] running on thread 131466667226816
```

ניתן לראות שעבור כל אלגוריתם נפתח טרד משלו ואכן הם רצו במקביל זה לזה.
וגם השרת לא נסגר, הוא עדיין מאזין לבקשה הבאה.

Part 10-
valgrind analysis-

<u>Memcheck:</u>

הוספנו מטרות נוספות בקובץ make file:

```
# Run with valgrind memcheck
valgrind_memcheck: $(TARGET)
    valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all ./$(TARGET)

# Run with valgrind helgrind
valgrind_helgrind: $(TARGET)
    valgrind --tool=helgrind ./$(TARGET)

# Run with valgrind callgrind (profiling)
valgrind_callgrind: $(TARGET)
    valgrind --tool=callgrind ./$(TARGET)
```

בס"ד

נריץ make ואז את הפקודה make valgrind_memcheack:

```
rei@Rei:~/OS FINAL PROJECT$ make valgrind_memcheck
valgrind --tool=memcheck --leak-check=full --show-leak-kinds=all ./server
==331808== Memcheck, a memory error detector
==331808== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==331808== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==331808== Command: ./server
==331808==
Server listening on port 5555 ...
```

נפתח טרמינל נוסף ונריץ בו בקשה מהשרת:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 40
OK MAX FLOW 15
OK HAM VERTEX: 0 2 7 5 6 8 4 1 3 9 0
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 5 7
rei@Rei:~/OS FINAL PROJECT$
```

תגובה מהשרת:

```
Server listening on port 5555 ...
[MST] starting job 1 on thread 102815424
[MST] job 1 moving to next stage
[MAXFLOW] starting job 1 on thread 111208128
[MAXFLOW] job 1 moving to next stage
[HAMILTON] starting job 1 on thread 119600832
[HAMILTON] job 1 moving to next stage
[MAXCLIQUE] starting job 1 on thread 127993536
[MAXCLIQUE] job 1 moving to next stage
sinkWorker: processing job 1
sinkWorker: notified job 1
```

נסגור את צד השרת עם ctrl+c

נקבל בסוף הפלט סיכום על הדליפות של התכנית:

```
==331808== LEAK SUMMARY:
==331808==    definitely lost: 0 bytes in 0 blocks
==331808==    indirectly lost: 0 bytes in 0 blocks
==331808==      possibly lost: 1,824 bytes in 6 blocks
==331808==    still reachable: 76,904 bytes in 20 blocks
==331808==         suppressed: 0 bytes in 0 blocks
==331808==
==331808== For lists of detected and suppressed errors, rerun with: -s
==331808== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
make: *** [makefile:23: valgrind_memcheck] Interrupt

rei@Rei:~/OS FINAL PROJECT$
```

הערה: נוכל להריץ "make valgrind_memcheck ARGS="-s
ולחזור על התהליך עם פתיחת טרמינל נוסף ולהריץ בקשה מהשרת ואז לסגור את צד השרת
ולקבל סיכום מעט יותר מדויק.

בקשר לסיכום הדליפה קיבלנו שהקוד נקי מדליפות זיכרון וכל מה שנשאר זה הקצאות פנימיות של מערכת/ספריות חלק מהארכיטקטורה של thread pool, singleton.

בס"ד

Helgrind:

במערכות מרובות־threads (כמו ה־server הזה עם ה־pipeline) כמה threads יכולים לגשת לאותו משתנה/משאב **במקביל**. אם שני טרדים קוראים וכותבים לאותו זיכרון בלי תיאום כמו condition_variable, mutex אז הסדר של מי שכותב ראשון נקבע ע"י מהירות/תזמון מקרי של מערכת ההפעלה וזה נקרא race condition. (זה מסוכן כי זה יכול לגרום לתוצאות לא צפויות, קריסות אקראיות וכ"ו ולכן יש לבדוק ע"י Helgrind).

תחילה נריץ make clean ואז make

ואז נריץ את הפקודה make valgrind_helgrind:

```
rei@Rei:~/OS FINAL PROJECT$ make valgrind_helgrind
valgrind --tool=helgrind ./server
==328205== Helgrind, a thread error detector
==328205== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==328205== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==328205== Command: ./server
==328205==
Server listening on port 5555 ...
```

בטרמינל שני נריץ לדוגמה:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < GRAPH.txt
OK MST WEIGHT: 25
OK MAX FLOW 15
OK HAM VERTEX: 0 1 3 2 0
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 1 2
rei@Rei:~/OS FINAL PROJECT$
```

נחזור לטרמינל הראשון ונסגור אותו ע"י ctrl+c ונקבל את הסיכום:

```
Server listening on port 5555 ...
[MST] starting job 1 on thread 102852288
[MST] job 1 moving to next stage
[MAXFLOW] starting job 1 on thread 111244992
[MAXFLOW] job 1 moving to next stage
[HAMILTON] starting job 1 on thread 119637696
[HAMILTON] job 1 moving to next stage
[MAXCLIQUE] starting job 1 on thread 128030400
[MAXCLIQUE] job 1 moving to next stage
sinkWorker: processing job 1
sinkWorker: notified job 1
^C==327549==
==327549== Process terminating with default action of signal 2 (SIGINT)
==327549==    at 0x4C4083D: accept (accept.c:26)
==327549==    by 0x121849: main (main.cpp:41)
==327549==
==327549== Use --history-level=approx or =none to gain increased speed, at
==327549== the cost of reduced accuracy of conflicting-access information
==327549== For lists of detected and suppressed errors, rerun with: -s
==327549== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 104 from 17)
make: *** [makefile:27: valgrind_helgrind] Interrupt
```

לפי הסיכום ניתן לראות שאין "possible data race" ושההרצה הסתיימה בהצלחה וללא שגיאות.

בס"ד

Cg:

לא חובה אבל מומלץ קודם להריץ תחילה make clean ואז make

ואז נריץ את הפקודה make valgrind_callgrind:

```
rei@Rei:~/OS FINAL PROJECT$ make valgrind_callgrind
valgrind --tool=callgrind ./server
==337177== Callgrind, a call-graph generating cache profiler
==337177== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==337177== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==337177== Command: ./server
==337177==
==337177== For interactive control, run 'callgrind_control -h'.
Server listening on port 5555 ...
```

ואז בטרמינל אחר נעשה כמה הרצות של בקשות מהשרת:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 31
OK MAX FLOW 12
ERR NO HAMILTONIAN CYCLE
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 1 5 9
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < GRAPH.txt
OK MST WEIGHT: 25
OK MAX FLOW 15
OK HAM VERTEX: 0 1 3 2 0
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 1 2
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 60
OK MAX FLOW 9
ERR NO HAMILTONIAN CYCLE
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 6 1
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 30
OK MAX FLOW 17
ERR NO HAMILTONIAN CYCLE
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 6 1
rei@Rei:~/OS FINAL PROJECT$
```

נסגור את השרת עם ctrl+c:

```
^C==337177==
==337177== Process terminating with default action of signal 2 (SIGINT)
==337177==    at 0x4C2083D: accept (accept.c:26)
==337177==    by 0x121849: main (main.cpp:41)
==337177==
==337177== Events    : Ir
==337177== Collected : 309568329
==337177==
==337177== I   refs:      309,568,329
make: *** [makefile:32: valgrind_callgrind] Interrupt

rei@Rei:~/OS FINAL PROJECT$
```

נשים לב שנפתח לנו קובץ חדש בשם- callgrind.out<num>

כדי לפתוח אותו באופן גרפי נריץ את הפקודה:

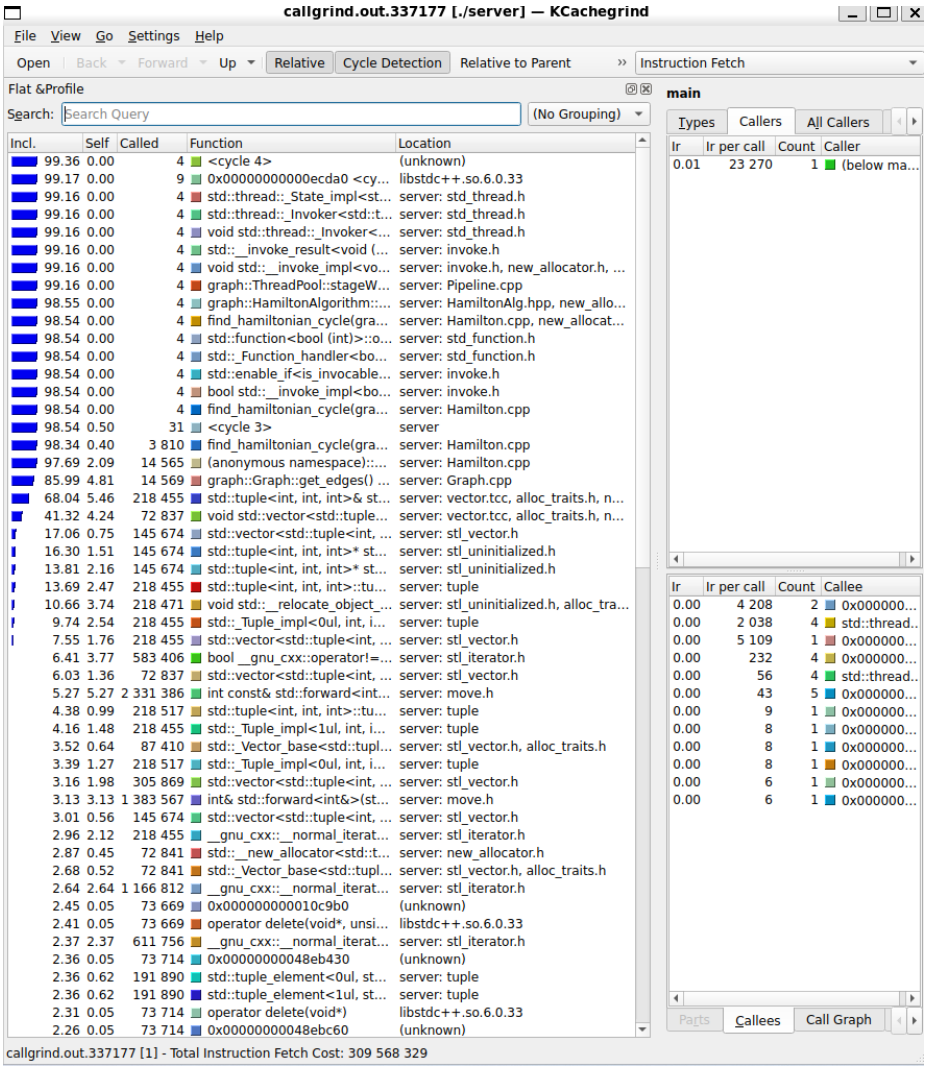
```
rei@Rei:~/OS FINAL PROJECT$ kcachegrind callgrind.out.337177
```

בס"ד

ואז יפתח לנו ממשק גרפי ששם נוכל לנתח את הקוד ולראות איפה הוא מבזבז הכי הרבה משאבים(זמן) ואיזה פונקציות הכי יקרות.

(נפתחת גם חלונית קטנה בהתחלה שניתן לסגור)

הממשק שנפתח-



למעשה, קוביות גדולות יותר מייצגות פונקציות "יקרות" יותר רצות הרבה או שהן לוקחות הרבה זמן.

Inclusive (עלות כוללת)- כל ההוראות שבוזבזו *בתוך הפונקציה וגם בפונקציות שהיא קראה להן.*

Self (עלות עצמית)- רק ההוראות שבתוך הפונקציה עצמה(בלי מה שהיא קראה) ואם הערך הזה גבוהה זה סימן שהיא צוואר בקבוק.

הפונקציות הכי יקרות: hamiltonian_cycle, find_hamiltonian_cycle

Graph::edges ועוד פונקציות של vector ו־tuple

ה־ **Hamiltonian Cycle** הוא כנראה צוואר הבקבוק המרכזי.

למעשה, אם ניכנס ל Hamilton.cpp נראה שהפונקציה has_edge היא היקרה ביותר-

בכל קריאה היא:

שולפת את כל הרשימות של הקשתות auto edges = G.get_edges();

בס"ד

עושה לולאה על כולן ((for (auto &e : edges)

כלומר בכל בדיקה אם יש קשת בין שני קודקודים עוברים על כל הקשתות מחדש וזה קורה בתוך DFS רקורסיבי אלפי או מאות פעמים. סדר גודל O(n!*m)
m- מס' הקשתות בגרף, n- מספר הקודקודים בגרף.

Part 11-

Code Coverage-

עבור חלקים 1 עד 4: כבר בוצע בתחילת הפרויקט
עבור החלק השני של הפרויקט:
תחילה נריץ בטרמינל אחד את הפקודה:

```
rei@Rei:~/OS FINAL PROJECT$ make coverage-build
```

(ניתן לראות במייק פייל את הקומפילציה עבור פקודה זו)
באותו טרמינל נריץ את השרת:

```
rei@Rei:~/OS FINAL PROJECT$ ./server
Server listening on port 5555 ...
Type 'exit' to shutdown gracefully, or use Ctrl+C
```

ואז בטרמינל אחר נריץ מגוון רחב של קלטים (בקשות) לשרת לדוגמה:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < testGraph.txt
OK MST WEIGHT: 7
OK MAX FLOW 0
ERR NO HAMILTONIAN CYCLE
OK MAX CLIQUE SIZE: 2 CLIQUE MEMBERS: 0 1
```

(הפקודה שמריצים זה: nc -q 1 localhost 5555 <testGraph.txt)
השורות שמתחת זה הפלט מהשרת.

עוד דוגמאות לקלטים:

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < RANDOM.txt
OK MST WEIGHT: 35
OK MAX FLOW 12
ERR NO HAMILTONIAN CYCLE
OK MAX CLIQUE SIZE: 3 CLIQUE MEMBERS: 0 7 3
```

```
rei@Rei:~/OS FINAL PROJECT$ nc -q 1 localhost 5555 < testGraph.txt
ERR PARSE FAILED: expected 'V <num vertices>'
```

בטרמינל של השרת נוכל להזין פקודות שונות:

```
help
Available commands: exit, quit, status, help
fgh
Unknown command: 'fgh'. Type 'help' for available commands.
```

כל זה במטרה לכסות כמה שיותר שורות קוד ופונקציות.

בס"ד

נחזור לטרמינל של השרת ונזין- exit:

```
exit
Shutting down server gracefully...
Shutting down server...
Server shutdown complete.
rei@Rei:~/OS FINAL PROJECT$
```

בטרמינל שלישי חדש נריץ את הפקודה:
make coverage-report
אין צילום כיוון שהפלט נורא ארוך וזה מעבר לגבולות הגלילה.
לאחר פקודה זו נקבל פלט ארוך מאוד של כל הנתונים של הקבצים ואחוזי השורות שהתבצעו מתוכם
להלן צילומים מקבצים נבחרים לדוגמה:

```
File 'src/main.cpp'
Lines executed:84.62% of 65
Creating 'main.cpp.gcov'
```

```
File 'src/server.cpp'
Lines executed:89.77% of 88
Creating 'server.cpp.gcov'
```

```
File 'src/Graph.cpp'
Lines executed:94.59% of 37
Creating 'Graph.cpp.gcov'
```

```
File 'src/algorithms/MST.cpp'
Lines executed:100.00% of 24
Creating 'MST.cpp.gcov'
```

```
File 'src/algorithms/Hamilton.cpp'
Lines executed:96.97% of 33
Creating 'Hamilton.cpp.gcov'
```

```
File 'src/algorithms/MaxClique.cpp'
Lines executed:100.00% of 37
Creating 'MaxClique.cpp.gcov'
```

```
File 'src/Pipeline.cpp'
Lines executed:97.78% of 45
Creating 'Pipeline.cpp.gcov'
```

אם נרצה נניח לבדוק אילו שורות קוד לא בוצעו ב Graph.cpp נריץ:

```
grep "####" Graph.cpp.gcov | head -20
```

בס"ד