מסמך תיאור פרויקט -
חישוב מקבילי ומבוזר של ספירת חיות שריג

תוכן עניינים:

מטרת הפרויקט

ספירת חיות שריג היא הבעיה של ספירת כל הצורות הקשירות בגודל מסוים הניתנות להרכבה מעל שריג. ידועים מספר אלגוריתמים לספירת חיות שריג, חלקם גנריים וחלקם מיועדים לשריגים מסוימים – אך המשותף לכולם הוא שהם לוקחים זמן חישוב אקספוננציאלי.
המטרה של הפרויקט שלנו היא להתמודד עם הבעיה הזאת, ואנו עושים זאת בעזרת:

- חישוב מבוזר – כלומר לאפשר למחשבים רבים לעבוד יחד על פתרון בעיית ספירה שכזו.
- חישוב מקבילי – כלומר ניצול כל כוח החישוב, כך שכל מחשב יוכל לנצל את כל הליבות שלו למשימה.

מטרת הפרויקט היא לאפשר חישוב מקבילי, מבוזר ומהיר של מגוון בעיות ספירה, ולכן בחרנו במימוש האלגוריתם של רדלמייר, אלגוריתם שמאפשר לספור חיות שריג בשריג כללי.
בנוסף, על הפרויקט לעמוד בנקודות הבאות:

- על הפרויקט להיות עמיד לנפילות – כלומר לדעת לשחזר את המצב התקין שהיה בו לפני הנפילה.
- על הפרויקט להיות יעיל ומהיר, ועל כן מימשנו אופטימיזציות נוספות ברמת האלגוריתם.
- על הפרויקט שלנו להתממשק עם פרויקט חיצוני המייצר גרף (המתאים לשריג) בהתאם להנחיות. על הפרויקט שלנו לזהות את תוכן הקובץ ולשחזר ממנו את השריג המתאים.

מבנה הפרויקט

הפרויקט מחולק בגדול ל2 חלקים, החלק החישובי והחלק התקשורתי.

<u>החלק החישובי</u>:

מטרת החלק החישובי היא ממש לבצע את האלגוריתם של רדלמייר על הגרף והפרמטרים שיקבל כקלט. חלק זה מתחלק גם הוא ל3 חלקים – החישובים עבור הלקוח, החישובים עבור השרת, ומבני נתונים משותפים.
בחרנו לממש את חלק זה ב-cpp בגלל שהשפה low level ומתקמפלת ישירות לקוד מכונה, ולכן מספקת זמן ריצה מהיר.

מבני הנתונים המשותפים:

חלק זה כולל את הקבצים:

- graphCreator.cpp, graphCreator.h
- Backup.cpp, backup.h
- defs.h

שני הקבצים הראשונים (graphCreator) מטפלים בהמרת קובץ גרף למבנה cppי, ויצירת מבני גרפים cppים נפוצים (polyominoes, polycubes, polyaminds).
קובץ הגרף הוא קובץ טקסטואלי מהצורה הבאה, המורכב מהשורות הבאות:

```
Data about the graph (Type, size of animal and etc)\n
<Number of nodes>\n
<Id of origin>\n
<id of node> <number of neighbors> <id of neighbor1> <id of neighbor2> …\n
<id of node> <number of neighbors> <id of neighbor1> <id of neighbor2> …\n
...
<id of node> <number of neighbors> <id of neighbor1> <id of neighbor2> …\n
```

שני הקבצים הבאים (backup) מאפשרים שמירת מצב החישוב הנוכחי של אלגוריתם רדלמייר לתוך קובץ (ע"י פונקציית doBackup()), וקריאת מצב החישוב (והמשך הביצוע שלו) מתוך קובץ (ע"י פונקציית recover()).

הקובץ האחרון – defs.h מכיל קבועים והגדרות השימושיים לשאר חלקי הקוד.

החישובים עבור הלקוח:

חלק זה הוא ליבת הפרויקט, והוא ביצוע החישוב עצמו. החלק מורכב מהקובץ redelClient.cpp ומ-redelClient.h.
הממשק של חלק זה הוא דרך הפונקציה executeJob() שמקבלת שני קבצים – קובץ גרף, וקובץ מצב חישוב. פונקציה זו קוראת לפונקציה countSubGraphs() עם המבנים הcpp'ים של הגרף ועם קובץ מצב החישוב.
countSubGraphs מקצה זיכרון עבודה, וקוראת לפונקציה הרקורסיבית recCounterGOTO().
הפונקציה האחרונה מבצעת את כל העבודה. היא קוראת לrecover() מקובץ מצב החישוב שקיבלה, וממשיכה לבצע את אלגוריתם רדלמייר על הגרף הנתון, עד סופו.
הפונקציה תומכת בדברים הבאים:

- הפונקציה מגבה את מצב החישוב הנוכחי שלה מדי BACKUP_INTERVALS חיות שנספרו (ע"י קריאה לdoBackup) על מנת להתמודד עם נפילות.
- הפונקציה מממשת אופטימיזציות, כמו דילוג על הרמה האחרונה, שימוש ב-goto וניהול מחסנית קריאות מצומצמת במקום בקריאה רקורסיבית.

את הקבצים שבחלק זה קימפלנו (יחד עם מבני הנתונים המשותפים) לספריה libRedelClient.so.

החישובים עבור השרת:

חלק זה אחראי על חלוקת משימת ספירה לתתי משימות. החלק מורכב מהקובץ redelServer.cpp ומ-redelServer.h.
הממשק של חלק זה הוא דרך הפונקציה jobsCreator() שמקבלת קובץ גרף, מספר צעדי החישוב (גודל החיה), חסם תחתון למספר תתי המשימות, וpath למיקום תתי המשימות (job/hi עבור המשימות job/hi_0, job/hi_1, job/hi_2, ...).
הפונקציה יוצרת תתי משימות עבור המשימה שהוגדרה. היא מחליטה על העומק בו היא תיצור את המשימות (ע"י decideWhatLevel()), ומריצה את האלגוריתם של רדלמייר עד אותה רמה, ומגבה לתוך קובץ של תת-משימה בכל "עלה" אליו היא מגיעה (ע"י recJobsCreatorWrapper() שקורא ל recJobsCreatorGOTO() הרקורסיבית). קובץ תת המשימה מגדיר את ה"עלה" בתור שורש עץ הספירה, כך שתכנית שתמשיך את החישוב מקובץ זה תחשב בדיוק את תת העץ שתחת אותו ה"עלה", ותסיים כשתחזור אליו. בעצם כל תתי המשימות זרות, והן מכסות יחד את עץ הספירה בכללותו.
על מנת לשחזר את התוצאה הסופית, יש לסכום את כל התוצאות של תתי המשימות.

בחלק זה קיימת גם הפונקציה canIFinishIt() שמאפשרת להריץ לוקלית את אלגוריתם החיפוש הרגיל למשך זמן מוגבל, ולהחזיר האם החישוב הסתיים בהצלחה, ואם כן גם את התוצאה שהתקבלה.

את הקבצים שבחלק זה קימפלנו (יחד עם מבני הנתונים המשותפים) לספריה libRedelServer.so.

החלק התקשורתי:

מטרת החלק התקשורתי היא ליצור תוכנת שרת ולקוח, כך שהשרת יחלק וינהל את כל תתי המשימות והתוצאות החלקיות, וידע לתקשר עם הלקוחות שיבצעו את החישובים בעצמם.
בחרנו לממש את חלק זה בפייתון בגלל זמן פיתוח מהיר, ובשביל לממש תקשורת, מאגר נתונים וממשק משתמש בצורה מהירה.
חלק זה מתחלק ל2 חלקים עיקריים – צד השרת וצד הלקוח.

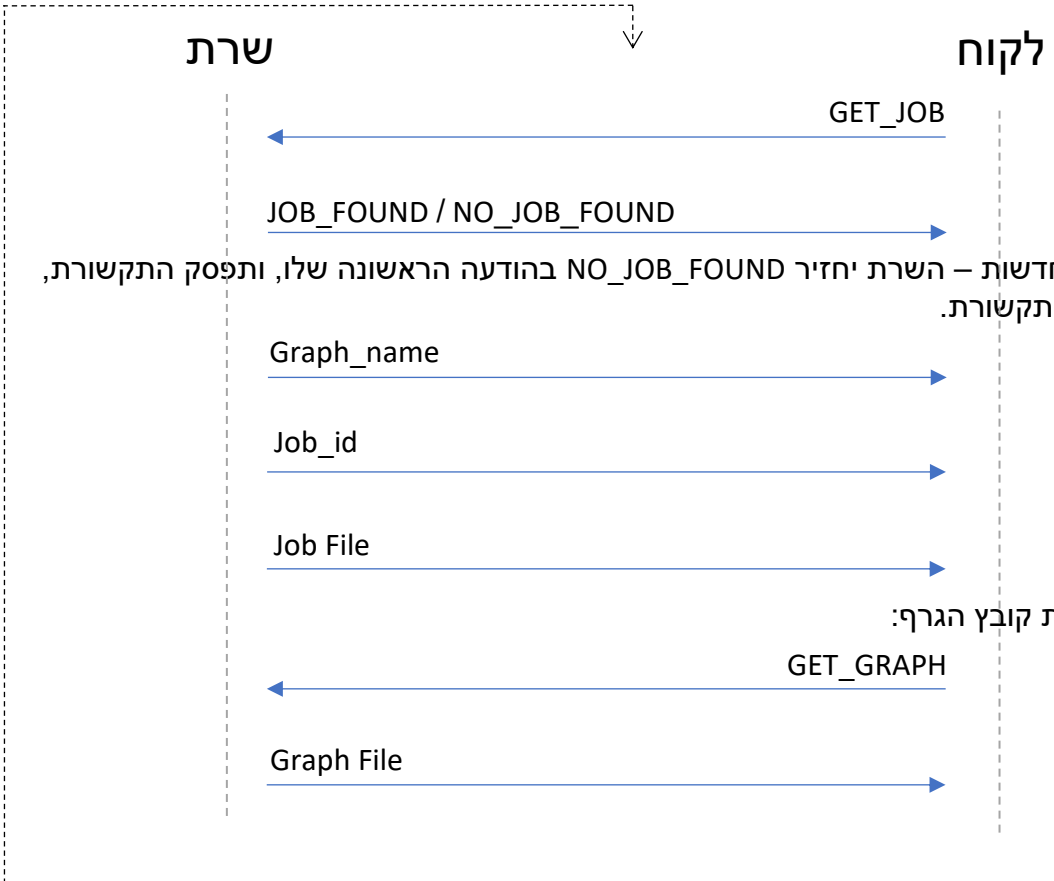
צד הלקוח:

חלק זה אחראי לתקשר עם צד השרת, לקבל ממנו משימות, לבצע אותן ולהחזיר לשרת תשובות. בנוסף, חלק זה מספק ממשק פשוט למשתמש, עליו נפרט במדריך למשתמש.
הקבצים בחלק זה:

- Client.py
- redelClient.py
- defs.py

את המשימות אנו נבצע בעזרת הפונקציות הcppיות שכתבנו בחלק הקודם. נעזרנו בספריה הפייתונית ctypes שמאפשרת לטעון קוד cpp מקומפל ולקרוא לפונקציות בו. הקובץ redelClient.py משמש מעטפת לפונקציה הcppית executeJob.
הקובץ העיקרי הוא client.py, ומטרתו היא לנהל n פועלים (כמספר הליבות במחשב) ולספק לכל אחד מהם משימה מהשרת. הפועל מקבל משימה, קורא למעטפת הפייתונית ל-executeJob ומחזיר את התוצאה. כשפועל מסיים משימה, המנהל ישלח את התוצאה לשרת ויבקש משימה חדשה.

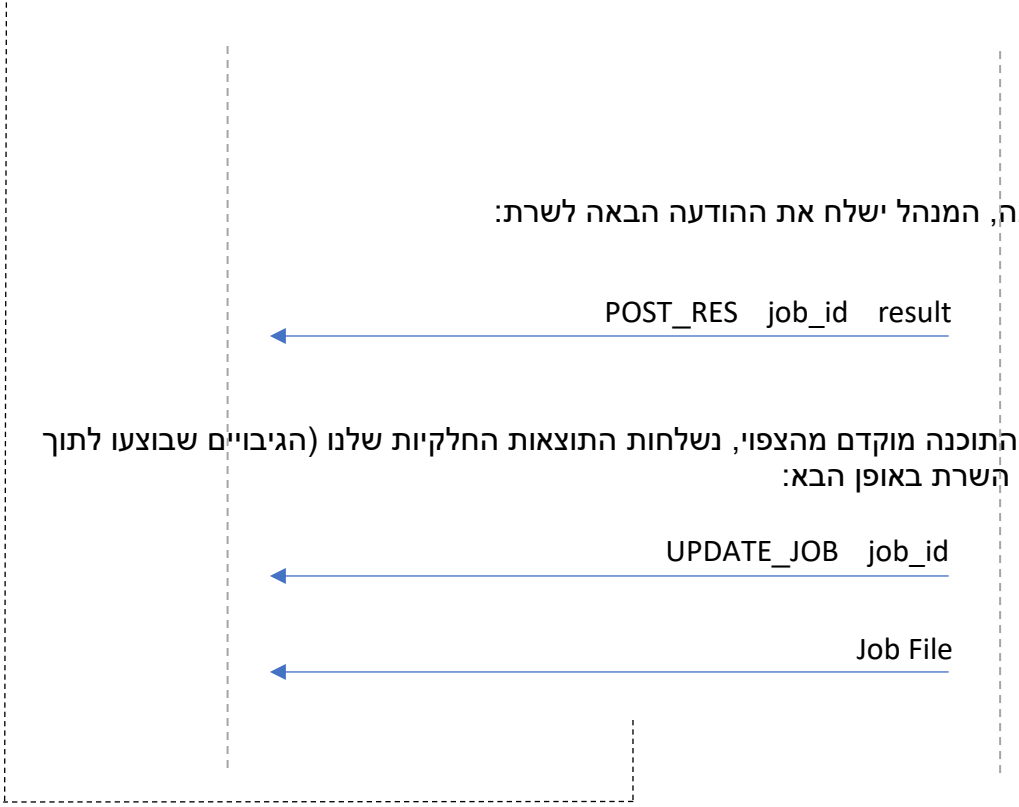
התקשורת בין המנהל לשרת מתבצעת באופן הבא:



אם אין עבודות חדשות – השרת יחזיר NO_JOB_FOUND בהודעה הראשונה שלו, ותפסק התקשורת, אחרת, תמשך התקשורת.

אם ללקוח אין את קובץ הגרף:

כשפועל יחזיר תוצאה, המנהל ישלח את ההודעה הבאה לשרת:



כשהלקוח סוגר את התוכנה מוקדם מהצפוי, נשלחות התוצאות החלקיות שלנו (הגיבויים שבוצעו לתוך קבצי המשימות) אל השרת באופן הבא:

הקובץ defs.py מכיל את כל המחרוזות שבהודעות הללו, הודעות שמוצגות למשתמש, וקבועים שימושיים כמו ה-server ip וה-port שבעזרתם יתקשר.

צד השרת:

השרת אחראי על ניהול וריכוז עבודות ותוצאות ביניים, על חלוקה לתתי-משימות, ועל תקשורת עם לקוחות. בנוסף השרת מספק ממשק למשתמש, עליו נפרט במדריך למשתמש.
הקבצים בחלק זה:

- server.py
- redelServer.py
- jobManager.py
- databaseManager.py
- defs.py

את החלוקה למשימות אנו נבצע בעזרת הפונקציות הcppיות שכתבנו בחלק הקודם. כמו בצד הלקוח נעזרנו בספריה ctypes – הקובץ redelServer.py מספק מעטפת לפונקציות הcppיות canIFinishIt ו-jobsCreator.

הקובץ העיקרי הוא server.py, ומטרתו היא להגיב לבקשות מלקוחות ולבקשות מהמשתמש.
על הבקשות מהלקוחות ואופן התקשורת הרחבנו בחלק של צד הלקוח.
הבקשות מהמשתמש מתחלקות ל2 סוגים עיקריים – ניהול מאגר העבודות/משימות (בין היתר – יצירת עבודה חדשה), וקבלת מידע. על קבלת המידע נפרט במדריך למשתמש.

צד השרת משתמש במאגר נתונים לוקלי בשם ZoDB ושומר בו את כל המידע החשוב, כמו עבודות ומשימות, תוצאות ביניים. אם השרת נופל, הוא יודע לשחזר את פעולתו התקינה בעזרת מאגר הנתונים.

ממשק ניהול מאגר הנתונים ממומש במודול databaseManager.py.

jobManager.py מגדיר שלוש מחלקות:

- jobStatus: מייצג תת-משימה אחת.
- jobGroup: מייצג עבודה אחת (אוסף תתי המשימות שלה).
- jobManager: מייצג את מנהל כל העבודות שבמאגר.

jobManager מאפשר ליצור עבודה חדשה (יקרא למעטפת של jobsCreator לחלוקת תתי המשימות), למחוק עבודה, להוסיף/להסיר מתור העבודות הפעילות, לשנות את סדר העבודות בתור. בנוסף מאפשר לספק משימה חדשה ללקוח, לעדכן תוצאה (סופית/חלקית) מלקוח.
כל האובייקטים של שלוש המחלקות נשמרים במאגר הנתונים, וניתנים לשחזור בכל עליית השרת.

השרת מממש מנגנון לפיזור מחדש של משימות המתעכבות יותר מדי זמן.

הקובץ defs.py מכיל את כל המחרוזות שבהודעות התקשורת, הודעות שמוצגות למשתמש, וקבועים שימושיים כמו הport שבעזרתם יתקשר. בנוסף מכיל את הjob_manager ואת הdb_manager הגלובליים.

הוראות התקנה

נדגיש שגם צד הלקוח וגם צד השרת שניהם מיועדים להרצה בסביבת לינוקס.
שימו לב כי אנו מספקים עם תיקיית ההתקנה קבצי so. המקומפלים לארכיטקטורת X86-64, אם ברשותכם מחשב בעל ארכיטקטורה אחרת, אנא קמפלו ליצירת קובץ so מתאים בעזרת:
- עבור libRedelClient.so – הריצו את הסקריפט redel_core/compile-client-core.
- עבור libRedelServer.so – הריצו את הסקריפט redel_core/compile-server-core.

הוראות התקנה למשתמש:

צד הלקוח:

אין צורך בpython מותקן – מצורפת סביבת ההרצה.
כדי להריץ את תכנת הלקוח יש לחלץ את תיקיית ההתקנה (גרסת הלקוח), לעדכן את ipה שבקובץ client/defs.py, ולהריץ ממנה את הסקריפט client/client.py.

צד השרת:

יש צורך בגרסת פייתון 3.6+ מותקנת (נבדק עם 3.8).
כדי להריץ את תכנת השרת יש לחלץ את תיקיית ההתקנה (גרסת השרת), ולהתקין את הספריה הפייתונית ZoDB עם השורה הבאה:

```
pip install ZoDB
```

ולאחר מכן להריץ את הסקריפט server/server.py.

הוראות התקנה למפתח:

כל קבצי המקור של הפרויקט נמצאים ב- https://github.com/tomhea/CountingPolyominoes.

מפתח שרוצה לשנות את קבצי הcpp יצטרך לקמפל אותם מחדש לקבצי so בעזרת הסקריפט שציינו לעיל.
סקריפטים אלו ייצרו את קבצי הספרייה libRedelClient.so ו-libRedelServer.so בתיקיות המתאימות להם.

לא נדרשת פעולה מיוחדת על מנת לשנות את קבצי הפייתון.

מדריך למשתמש

גם תוכנת הלקוח וגם תוכנת השרת מציאות ממשק טקסטואלי עם מגוון פקודות.
בשתי התוכנות ניתן לראות את רשימת הפקודות המלאה ע"י פקודת help (או פשוט h).

הוראות שימוש בתוכנת הלקוח:

תוכנת הלקוח נבנתה כדי להיות מאוד פשוטה – פותחים אותה והיא כבר מתחילה לעבוד.
אם היא מוצאת את השרת – היא תתחיל לבצע משימות ולהחזיר תוצאות, ואם היא לא מוצאת – היא תחכה ותחפש מחדש את השרת באופן אוטומטי.

תוכנת הלקוח מציעה את השירותים הבאים (הפקודות הבאות):

- עזרה (help) – הדפסת פירוט על כל הפקודות האפשריות.
- קבלת סטטיסטיקות (stats) – מספר המשימות שהתוכנה סיימה בסשן זה, יחד עם מספר החיות שספרה.
- כיבוי/הדלקת הדפסות (prints) – בעיקר הדפסות של תוצאות חדשות שחושבו. כבוי בברירת מחדל.
- עדכון השרת (update) – עדכון השרת בכל התוצאות החלקיות שחושבו עד כה (משימות פעילות שלא הסתיימו עדיין).
- יציאה חלקה (close) – כיבוי תוכנת הלקוח, תוך עדכון השרת על כל התוצאות החלקיות.

הוראות שימוש בתוכנת השרת:

תוכנת השרת מגובה ע"י מאגר נתונים לוקלי (באופן אוטומטי), וכאשר תוכנת השרת עולה מחדש היא משחזרת את הנתונים באופן אוטומטי מאותו מאגר.

תוכנת השרת מציעה מגוון גדול של שירותים:

- עזרה (help) – הדפסת פירוט על כל הפקודות האפשריות.
- יצירת תתי משימות (create) – הגדרת משימה (מתן שם) וחלוקה שלה לתתי משימות.
- התחלת משימה (add) - העברת משימה לתור המשימות הפעילות.
- הפסקת משימה (stop) – הוצאת משימה מתור המשימות הפעילות.
- מחיקת משימה (delete) – מחיקת כל המידע הקשור למשימה (כולל תוצאות ותתי משימות).
- הוספת גרף (graph) – רישום קובץ גרף במערכת.
- הסרת גרף (ungraph) – הסרת הרישום של קובץ גרף.
- הצגת רשימות (list) – הצגת הגרפים הרשומים, המשימות הקיימות (ואחוזים) ואת תור המשימות הפעילות.
- הצגת אחוזים (%) – הצגת אחוז תתי המשימות שהושלמו.
- הצגת תוצאות (#) – הצגת התוצאה העדכנית ביותר למשימה זו (מספר החיות שנספרו עד כה), ותחזית משוערת למספר הסופי.

- סדר עדיפויות (prio) – הצגת סדר העדיפויות של המשימות הפעילות, או שינוי מיקום של משימה בסדר העדיפויות.
- תזמון משימות מתעכבות (resched) – חלוקה מחדש של כל תתי המשימות (תחת משימה מסויימת) שהתעכבו יותר מX דקות.
- יציאה חלקה (close) – כיבוי תוכנת השרת.