תיכון ע"ש ב.אוסטרובסקי
מצוינות • מנהיגות • יזמות • חדשנות

פרויקט 5 יח"ל

בתכנון ותכנות מערכות הגנת סייבר

קיץ 2021

שם הפרויקט : פיתוח המשחק "המבוך", תכנות בוט באמצעות AI, ומשחק בין שני משתמשים ברשתות

שם פרטי : בן

שם משפחה : אליאב

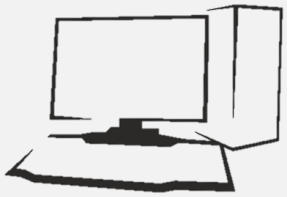
תעודת זהות : 330053893

כיתה : י"ב - **4**

תיכון : אוסטרובסקי

מורה מלווה : אתי בררו

תאריך הגשה : 19.05.21

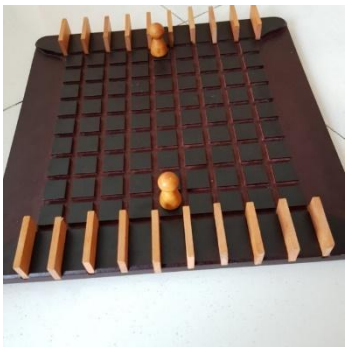


מבוא..... 4
מוטיבציה לפיתוח 5
קהל היעד..... 5
אילוצים ודרישות..... 5
הנחות יסוד ודרישות 5
תרשים ארכיטקטורה 6
מדריך למשתמש 7
תרשים הפעלה 7
הוראות הפעלת המשחק.....8
סביבת הפיתוח 8
תיעוד נושא החקר 9
תכנות מונחה עצמים 9
אלגוריתם חיפוש 9
minimax / alpha beta pruning..... 11
רשתות תקשורת 13
פירוט המודולים..... 14
Pygame 14
Pygame.gfxdraw..... 14
Threading..... 14
Time..... 15
Socket 15
Collections 16
Random 17
Sys..... 17
Datetime..... 17
מדריך למפתח..... 18
מבנה הפרויקט..... 18
__init__.py.....18
תיקיית quoridor.....18
תיקיית network.....19
תיקיית AI.....19
קובץ main.py 19
constants.py - קבועים חשובים..... 20

pieces.py - מחלקת Pawn 21
board.py - מחלקת לוח המשחק Board 22
game.py - מחלקת המשחק Game 25
מסמך algorithm.py - מחלקת AI 27
קובץ client.py - מחלקת Player_Client 29
קובץ server.py - מחלקת השרת 30
קובץ Main.py ומחלקת Main 31
רפלקציה 36
ביבליוגרפיה 37
נספחים 38
constants.py 39
pieces.py 40
board.py 41
game.py 47
client.py 52
server.py 53
algorithm.py 54
main.py 57
דברים שנמחקו: 63
מחלקת Wall 63
פעולה win_possible 64
הפעולה wall_relevant 65
הפעולה relevant_possible_walls 65

מבוא

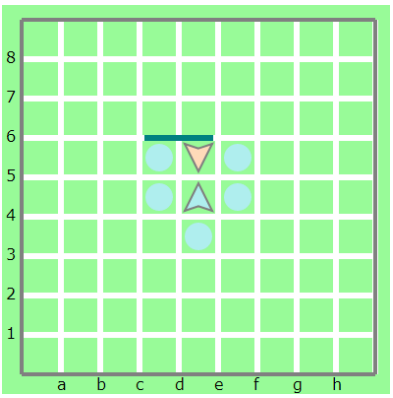
הפרויקט שלי הוא פיתוח משחק המבוך "Quoridor". המבוך הוא משחק קופסה אסטרטגי לשני שחקנים. המשחק לא כולל אלמנט של מזל ומצריך כישורים דדוקטיביים רבים. בשנת 1997, המשחק זכה במשחק השנה בארצות הברית, קנדה, צרפת ובלגיה. המשחק משוחק על לוח של 81 משבצות (9x9). לכל שחקן יש חייל אשר ניתן להזיז אותו משבצת אחת לכל כיוון (ללא אלכסונים). מטרת המשחק היא להיות השחקן הראשון שמגיע לצד השני של הלוח עם החייל שלו כאשר הלוח מתחיל כמו התמונה המצורפת.



במשחק, לכל שחקן יש עשרה מחסומים שבאורך שתי משבצות כפי שמתואר בתמונה. כל שחקן בתורו יכול לבחור בין להזיז את החייל שלו לבין להציב מחסום שיחסום את הדרך של השחקן השני לניצחון. ברגע שמציבים מחסום, לא ניתן להזיז אותו והוא נשאר במקום למשך כל שארית המשחק. כשנגמרים לשחקן המחסומים, הוא יכול רק להזיז את השחקן שלו.

כללים מיוחדים של המשחק:

- אסור ששני מחסומים יעברו אחד דרך השני (יצטלבו). מותר ששניהם ייפגשו רק אם אפשרי פיזית לשים את שני המחסומים.
- דפנות הלוח מהווים מחסומים.
- אם שני השחקנים צמודים, מותר לחלוף על פני השחקן השני (לדלג שתי משבצות) ואסור ששני החיילים יעמדו באותה המשבצת.
- בהמשך לכלל הנ"ל, אם אין באפשרות חייל לדלג על חייל אחר בגלל שנמצא מאחוריו מחסום, הוא יכול לחלוף על החייל השני באלכסון, כל עוד אין מחסום החוסם את התנועה, כפי שמתואר בשרטוט משמאל (בשהתור הוא של השחקן הכחול).
- אין לחסום את השחקן השני לגמרי מהנצחון, תמיד חייבת להיות דרך כלשהי שהשחקן השני יגיע לקצה הלוח. כלומר, אסור לשים מחסומים בצורת ריבוע סביב החייל השני.



תכנתתי את המשחק בשפת Python 3.8 בסביבת העבודה Pycharm של JetBrains.

תכנתתי את המשחק בעזרת תכנות מונחה עצמים (Object Oriented Programming). השחקנים, הלוח והמשחק יהיו אובייקטים.

לאחר שיימתי לתכנת את המשחק עבור שני שחקנים אנושיים מאותו המחשב, תכנתתי בוט שישחק נגד שחקן אנושי שיפעל ב-AI לפי אלגוריתם של עץ החלטות (minimax). זהו אלגוריתם מוכר בתורת המשחקים הנועד למשחקים קומבינטוריים ללא מזל (דמקה, שחמט, המבוך...). תכנתתי בוטים נוספים שכל אחד פועל באסטרטגיה אחרת. הבוט הכי "חכם" הוא ה-minimax.

בנוסף תכנתתי ששני שחקנים יוכלו לשחק ממכשירים שונים אחד נגד השני דרך האינטרנט. השתמשתי בספריית socket בשביל להשתמש ברשתות.

מוטיבציה לפיתוח

הסיבה שבחרתי במשחק "המבוך" היא מפני שהמשחק מאפשר תכנות באמצעות תכנות מונחה עצמים. בנוסף, בגלל שהמשחק לא כולל אלמנט של מזל, לא אצטרך לדאוג ממניע האקראיות. בתאוריה, ה-AI יוכל לשחק בצורה המיטבית.

הסיבה שבחרתי לכתוב את הפרויקט ב-Python 3.8 היא בגלל שהיא גרסה מתקדמת של פייתון, שפת תכנות עילית, שאוכל לכתוב את הקוד שלי בצורה אסתטית וקצרה.

הסיבה שבחרתי להשתמש באלגוריתם שמשתמש בעץ היא בגלל שהמשחק הזה מושלם בשביל האלגוריתם הזה. האלגוריתם בוחן את כל האפשרויות שיכולות להיות ואז בוחר את הטוב ביותר מבין כולן. הסיבה שבחרתי בשימוש ב-sockets און-ליין היא לנסות גם תכנות טכנית מאוד בצורה שהיא בדרך שלא התנסיתי בה לפני כן.

בנוסף, זה משחק ששיחקתי בו לאורך הילדות שלי ביחד עם המשפחה שלי והוא משחק שאני מאוד נהנה ממנו. למרות שחוקי המשחק פשוטים מאוד והמשחק כביכול לא מורכב, האסטרטגיה שהמשחק מתבסס עליה עמוקה מאוד.

קהל היעד

קהל היעד של התוכנה הוא כל אחד בגיל 5+ שיודע לשחק המבוך. בנוסף, צריכים לדעת לתפעל את שורת הפקודה של windows או לדעת להריץ קוד של פייתון.

אילוצים ודרישות

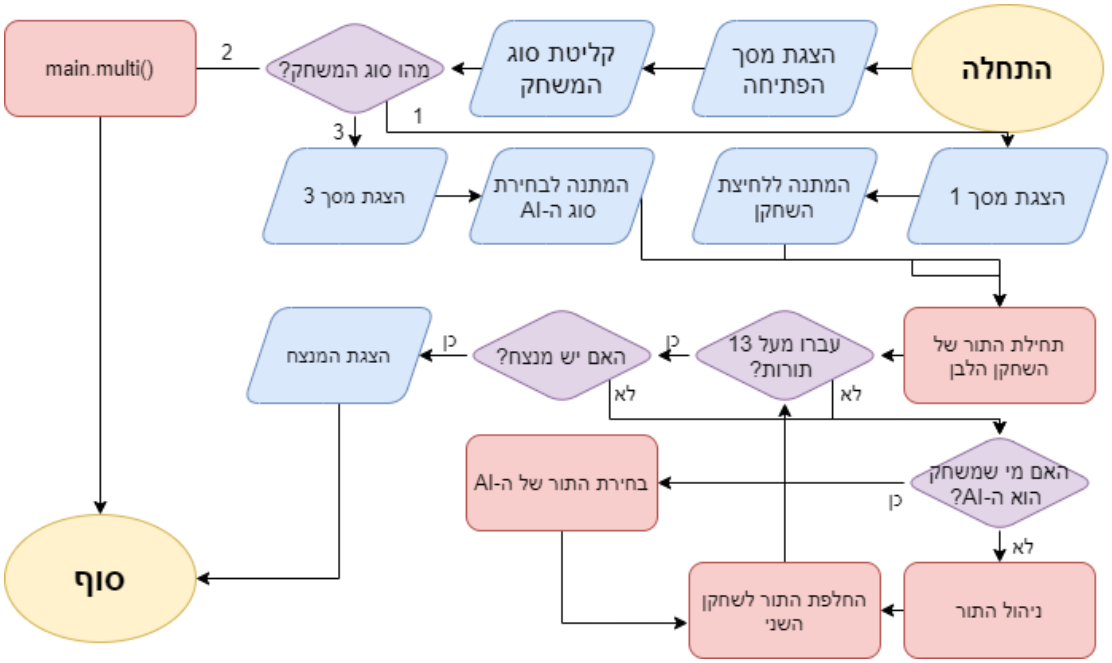
המשחק מתוכנת בפייתון גרסה 3.8.5 ומשתמש בפייגיים גרסה 1.9.6.
בנוסף, משחק הרשתות חייב שכתובת ה-IP של השרת יהיה ידוע לפני שהמשחק מתחיל.

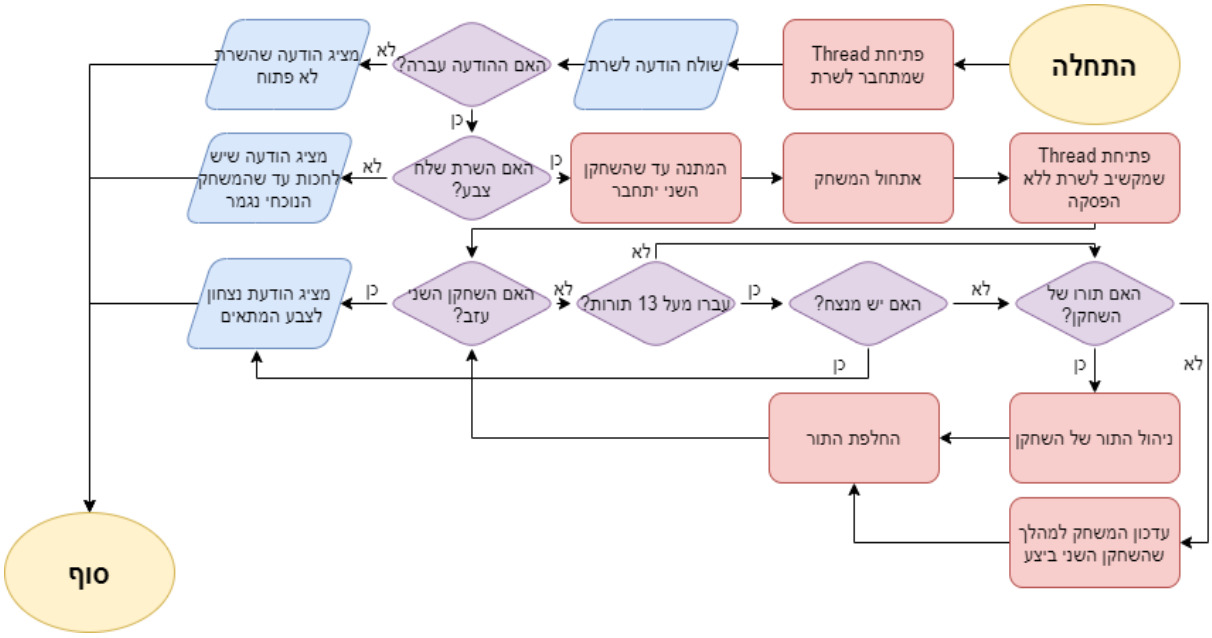
הנחות יסוד ודרישות

כדי להשתמש במשחק, צריך שפייתון בגרסה מתאימה יהיה מותקן במחשב. צריך להוריד pygame. אם אין pycharm או סביבת פיתוח אחרת, צריך להוסיף את פייטון ל-path של המחשב, ולהריץ את הקוד דרך ה-cmd. על מנת לפתוח את המשחק ב-cmd, יש לכתוב python main.py.
אם רוצים להריץ מוד ספציפי של המשחק, אפשר גם לכתוב python main.py [mode] כאשר מספר המוד יהיה במקום הסוגריים המרובעים.

תרשים ארכיטקטורה

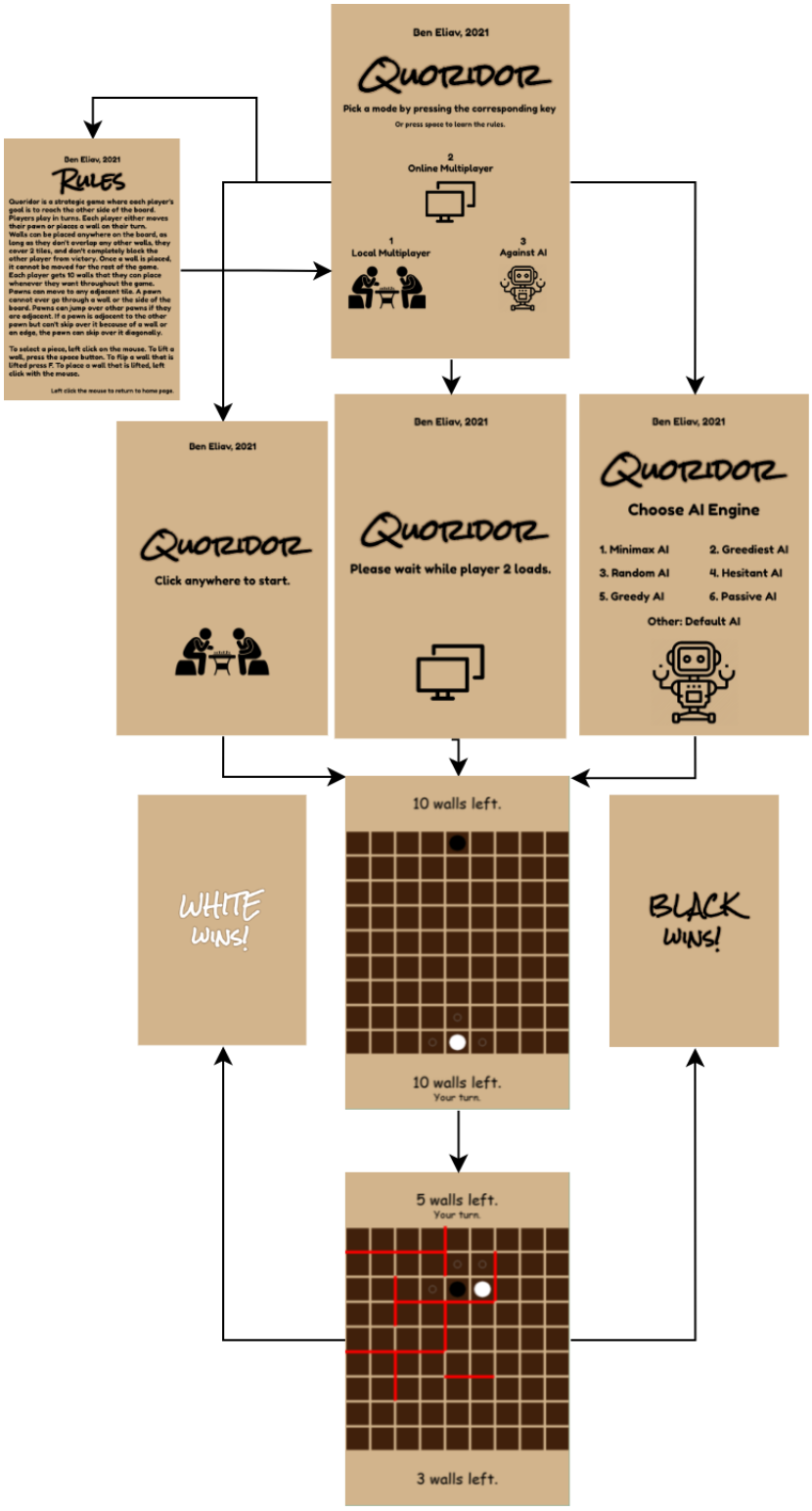
תרשים הזרימה הראשון מתאר את הפעולה main והשני את הפעולה multi.





מדריך למשתמש

תרשים הפעלה



בתחילת הרצת הקוד, יופיע מסך הפתיחה המופיע מלמעלה.

- אם שחקן לוחץ על 1, הוא בוחר באופציית המשחק המקומי בין שני שחקנים. ייפתח המסך שמתחת למסך הפתיחה משמאל. כאשר הוא לוחץ עוד פעם על המסך, המשחק מתחיל.
- אם השחקן לוחץ על 2 במסך הפתיחה, הוא בוחר במשחק אונליין רב משתמשים. אם השרת סגור, תודפס הודעה שצריך לפתוח את השרת והמסך ייסגר. אם השרת פתוח וזה השחקן הראשון שמצטרף, יופיע המסך שמתחת למסך הפתיחה באמצע. אם מי שמצטרף הוא השחקן השני, ישר יתחיל המשחק.
- אם השחקן לוחץ על 3 במסך הפתיחה, הוא בחר במשחק נגד ה-AI. ייפתח המסך שמימין מתחת למסך הפתיחה. לאחר מכן, השחקן בוחר את ה-AI שהוא רוצה לשחק נגדו על ידי הקשת המספר המתאים. לאחר מכן נפתח המשחק.
- אם השחקן לוחץ על רווח, מופיע מסך ההוראות. בשביל לסגור אותו, אפשר ללחוץ על הלחצן השמאלי של העכבר או על מקש ה-ESC.

המשחק לאחר מכן מתנהל בתורות, התור הראשון הוא של השחקן הלבן כפי שניתן לראות בתרשים. כאשר המשחק נגמר, יופיע מסך הניצחון למי שניצח. אם שחקן יוצא באמצע המשחק, גם יופיע מסך הניצחון לפני שהמשחק נסגר.

הוראות הפעלת המשחק

- לחיצה על הלחצן של העכבר קוראת לפעולה game.select. הפעולה קודם בודקת אם מחסום מורם. אם מחסום מורם, היא מנסה להציב אותו במקום שבו השחקן לחץ. אם הפעולה מצליחה True מוחזר, אם לא מודפס על המסך שהמקום שהשחקן ניסה להציב את המחסום הוא מקום לא חוקי. אם מחסום לא מורם אבל החייל של השחקן מורם, הפעולה בודקת אם ניתן להזיז את החייל למקום שבו השחקן לחץ. אם כן, הפעולה מזיזה את השחקן. אם לא, החייל משוחרר. אם מחסום וחייל לא מורמים, הפעולה מנסה להרים את החייל שנמצא במקום שהשחקן לחץ. אם יש חייל במקום הזה, הפעולה מרימה אותו וניתן יהיה לראות עיגולים אפורים במשבצות הסמוכות לחייל. אם אין, לא קורה כלום.
- לחיצה על מקש הרווח תקרא לפעולה game.lift_wall. אם השחקן לא השתמש בכל המחסומים שלו, המשחק ירים מחסום. אם מחסום מורם, הפעולה תשחרר אותו. אם חייל מורם כאשר לוחצים על רווח, הוא משוחרר.
- לחיצה על מקש ה-f תסובב את המחסום המורם אם יש מחסום מורם.
- לחיצה על CTRL+Z תעשה "undo" למהלך האחרון שנעשה (במקרה והמשחק הוא לא אונליין). כל שחקן יכול לעשות את זה רק פעם אחת במשחק על מנת לחזור על טעות.

סביבת הפיתוח

השתמשתי בסביבת הפיתוח Pycharm של JetBrains.

תיעוד נושא החקר

תכנות מונחה עצמים

תכנות מונחה עצמים היא פרדיגמת תכנות המבוססת על המושג של אובייקטים. פרדיגמה זו מנסה לחקות את העולם האמיתי, שבו כל דבר הוא אובייקט הנמצא באינטרקציה תמידית עם האובייקטים בסביבה. לכל אובייקט יש תכונות ופעולות משלו. שמורות בתוך האובייקט נתונים (שדות) הקשורים למצב ותכונות שלו וקוד בצורה של פעולות / פרוצדורות. כל אובייקט יכול לשנות את המצב שלו. אובייקטים יכולים להיות באינטרקציה אחד עם השני, לרשת אחד מהשני ולהשפיע על המצב של אובייקטים אחרים.

רוב התכנות המונחה עצמים משתמש במחלקות (classes). כל אובייקט הוא מופע (Instance) של מחלקה מסוימת. לרוב, שפות תכנות התומכות בתכנות מונחה עצמים הן שפות תכנות אימפרטיביות או פרוצדורליות. בכל שפת תכנות, הרעיון של תכנות מונחה עצמים מיושם בצורה אחרת.

נשתמש בעט כדוגמה לאובייקט. לעט יש תכונות כמו צבע הדיו והרוחב שלו המאפיינים את העט. בנוסף, לעט יש מספר פעולות שמשנות את המצב שלו, כמו לפתוח ולסגור. לעט גם יש פעולות שמשנות את המצב של אובייקטים הנמצאים סביבו. למשל, פעולת הכתיבה מקבלת דף ומשנה את המצב של הדף. בתכנות מונחה עצמים יש מספר עקרונות חשובים:

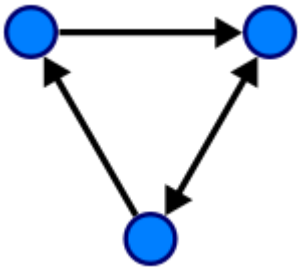
- ירושה (Inheritance): מחלקה יכולה לרשת תכונות ממחלקה אחרת. למשל, יכולה להיות המחלקה "בעל חיים". לבעל חיים מספר פעולות ותכונות המאפיינות אותו כמו גיל וגובה, ופעולות המאפיינות אותו כמו אכילה, ריצה והשמעת קול. אם אנחנו רוצים להגדיר מחלקה של קוף, ניתן לקבוע שהמחלקה "בעל חיים" תהיה מחלקת אב של קוף. המחלקה קוף תירש את כל המאפיינים שיש לבעל חיים. בפייתון, מחלקה יכולה לרשת מיותר ממחלקה אחת.
- קומפוזיציה (Object composition): תכונה של מחלקה יכולה להיות מופע של מחלקה אחרת. למשל, יכולה להיות המחלקה "מנוע". אם נרצה להגדיר מחלקה של "מכונית", אחד המאפיינים שלו יהיה מנוע. ככה, משתמשים באובייקט אחד על מנת להגדיר אובייקט אחר.

כל השפה בפייתון מבוססת על הפרדיגמה של תכנות מונחה עצמים. בפייתון, כל טיפוס וכל פעולה הם אובייקטים. כל טיפוס בפייתון, המוגדרים כחלק מהשפה והמוגדרים על ידי המשתמש, יורשים מהמחלקה object. הפרויקט שלי מבוסס על הרעיון של תכנות מונחה עצמים וכל דבר בפרויקט הוא עצם.

אלגוריתם חיפוש

אלגוריתם חיפוש הוא אלגוריתם שנועד לפתור בעיה של חיפוש. מטרתה היא למצוא מידע כלשהו השמור במבנה נתונים מסוים. אלגוריתם חיפוש פשוט הוא אלגוריתם חיפוש לינארי, העובר על

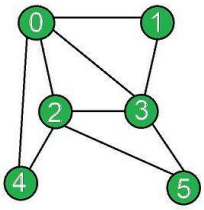
מערך או רשימה מקושרת ומחפש את האיבר הספציפי שמנסים למצוא. לא תמיד ניתן להשתמש באלגוריתם חיפוש לינארי במקרים של מבני נתונים מסובכים יותר. האלגוריתמים BFS ו-DFS הם שני אלגוריתמים לחיפוש בגרף.



גרף הוא מבנה נתונים המורכב מצמתים (vertices) וקשתות (edges). למשל, הגרף משמאל הוא גרף המורכב משלושה צמתים (העיגולים הכחולים) ושלוש קשתות (החצים השחורים). הצמתים הם האיברים עצמם בגרף והקשתות מקשרות בין הצמתים. מבנה נתונים גרף מתאים לייצג מסלולים, רשתות תקשורת ועוד. עצים יכולים להיות גם מיוצגים בגרפים.

גרף לא מכוון הוא גרף שאם אפשר ללכת מצומת V_1 לצומת V_2 דרך קשת E, בהכרח אפשר ללכת מצומת V_2 ל-V_1 דרך אותה הקשת. בגרף מכוון, הקשר הזה לא מתקיים וכל קשת הוא רק חד כיווני. לכן אם אפשר להגיע מצומת V_1 לצומת V_2 וגם מצומת V_2 ל-V_1 , יהיו שתי קשתות שונות. ניתן לראות שהגרף הנ"ל הוא גרף מכוון. שני צמתים נקראים שכנים אם עוברת ביניהם קשת. גרף לא משוקלל הוא גרף שאורכי כל הקשתות שלו שווים, שהמסלול הקצר ביותר הוא בהכרח המסלול שעובר דרך הכי פחות צמתים. בגרף משוקלל, לכל קשת יש אורך או "משקל".

ישנן שתי דרכים עיקריות לייצג גרף, רשימת שכנות ומטריצת שכנות. מטריצת שכנות (מטריצת סמיכויות) מייצגת את הגרף כמערך דו מימדי שכל אינדקס הוא צומת בגרף. במטריצה A, A[i][j] ייצג את משקל הקשת בין הצמתים V_i ו-V_j. בגרף לא משוקלל, 1 מייצג קשת ו-0 מייצג שאין קשת. אפשר לראות משמאל מטריצת שכנות של גרף לא מכוון ולא משוקלל.



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

רשימת שכנות היא ייצוג של גרף בצורה של "מילון". לכל צומת יש רשימה של כל הצמתים שניתן להגיע אליהם מהצומת הנוכחי. ניתן להסתכל על רשימת שכנות כמערך חד מימדי שכל איבר בו הוא רשימה של כל המקומות שניתן להגיע אליהם מהמקום הנוכחי. אין דרך להציג גרף משוקלל בייצוג זה שתהיה הגיונית.

כאשר משתמשים בגרפים, יותר נהוג להשתמש ברשימות שכנות בגלל הדרך האינטואיבית שבה הן מיוצגות. מטריצת שכנות מתאימה לגרפים משוקללים. בפרויקט שלי השתמשתי גם ברשימת שכנות, שמיוצגת כ-defaultdict שהמפתחות שלו הם הצמתים מיוצגים כ-tuples (כל המיקומים על הלוח) והערכים שלו הם sets בשביל שהגישה לכל איבר תהיה מהירה וכי הסדר של הקשתות לא חשוב.

האלגוריתמים Breadth-First Search ו-Depth-First Search הם אלגוריתמים של חיפוש בגרף.

Breadth-First (רוחב ראשון) היא צורת חיפוש בגרף שקודם הולכת לרוחב ואז לעומק. אם משתמשים בחיפוש בשביל למצוא את המסלול בין שני צמתים V_1 לצומת V_2, הוא בודק קודם את כל המסלולים האפשריים באורך 1 מ- V_1, אז כל המסלולים האפשריים באורך 2, וכך הלאה. הוא מפסיק כאשר הוא מגיע לצומת V_2. ניתן לראות שהאלגוריתם תמיד מוצא את המסלול הקצר ביותר מפני שהוא הולך קודם לרוחב.

האלגוריתם מתחיל בצומת ההתחלתי. הוא עובר על כל השכנים של הצומת ובונה מסלול לכל שכן. הוא שומר את כל המסלולים בתוך טור. האלגוריתם רץ עד שהטור ריק. בתוך לולאה, האלגוריתם כל פעם שולף את ראש הטור (מסלול) ורואה את הצומת האחרון שבמסלול. הוא עובר על כל השכנים של הצומת, ואם הם צמתים חדשים (לא נבדקו כבר) בונה מסלול חדש איתם בסוף המסלול. הוא מוסיף את המסלול החדש לסוף הטור ומתחיל שוב עם האיבר הבא בטור. אם מקבלים מסלול שמגיע לצומת הרצוי, מחזירים את המסלול הזה. זה יהיה המסלול הקצר ביותר. אם הטור מתרוקן, זה אומר שאין מסלול שמגיע לצומת הרצוי.

הסיבוכיות של הפעולה היא O(V+E), כאשר V כמות הצמתים ו-E הוא כמות הקשתות. זה המקרה הגרוע שבו עוברים על כל מסלול אפשרי עד שמגיעים למסלול הנכון. לרוב, האלגוריתם יהיה הרבה יותר קצר.

Depth-First (עומק ראשון) היא צורת חיפוש אחרת בגרף שבודק קודם לעומק ואז לרוחב. האלגוריתם מנסה כל מסלול במלואו לפני שהוא מתחיל לבדוק מסלול אחר. הוא יפסיק לבדוק מסלול כאשר הוא יראה שכל השכנים של הצומת האחרון במסלול כבר נמצאים בתוך המסלול. האלגוריתם לא ימצא בהכרח את המסלול הקצר ביותר, אלא ימצא מסלול כלשהו בין הצומת ההתחלתי לצומת הסופי (למקרה שיש).

האלגוריתם מתחיל בצומת ההתחלתי. דוחפים את הצומת ההתחלתי למחסנית כמסלול ההתחלתי. אחר כך, עושים לולאה על המחסנית עד שהיא ריקה. כל פעם שהלולאה רצה, שולפים את האיבר (המסלול) בראשה. עוברים על כל השכנים של הצומת האחרון במסלול ובונים מסלול חדש לכל אחד (שלא כבר נבדק) ודוחפים את המסלול החדש לתחילת המחסנית. אם מגיעים למצב שאין עוד שכנים שלא נבדקו ושהמסלול לא הגיע למטרה, שולפים את המסלול מהמחסנית וממשיכים עם המסלול הבא. ברגע שמגיעים לצומת היעד, מחזירים את המסלול. אם המחסנית ריקה, הפעולה עברה על כל המסלולים ולכן היא תחזיר False. אפשר לכתוב את הפעולה גם כפעולה רקורסיבית שמחסנית הקריאות מדמה את המחסנית של המסלולים.

הסיבוכיות של הפעולה היא גם O(V+E) במקרה הגרוע שבו הפעולה עוברת על כל המסלולים האפשריים. שתי הפעולות עושות את אותו הדבר אם הן עוברות על כל המסלולים האפשריים, רק בסדר שונה. הפעולה DFS יותר חסכונית במקום מפני שלא בונה הרבה מסלולים מבלי לבדוק אותם. BFS יותר מתאים אם המטרה נמצאת קרובה לצומת ההתחלתי ושיש מעט מסלולים שמגיעים למטרה. במקרה שיש הרבה מסלולים אפשריים והמטרה רחוקה מהצומת ההתחלתי, DFS יותר מתאים. היתרון של BFS הוא שהוא יכול לחשב את המסלול הקצר ביותר, בזמן שה-DFS יצטרך לעבור על כל המסלולים בשביל למצוא את הקצר ביותר.

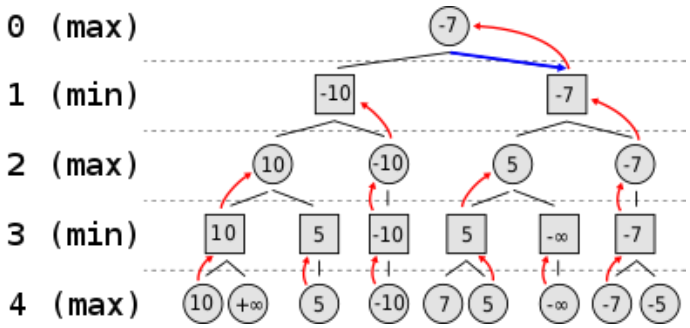
minimax / alpha beta pruning

האלגוריתם minimax הוא אלגוריתם בתורת המשחקים, בינה מלאכותית, תורת ההחלטות ופילוסופיה בשביל לקבל את ההחלטה שתביא לרווח ה**מקס**ימלי וההפסד ה**מינ**ימלי. זה אלגוריתם של בינה מלאכותית, כי המחשב מנסה לחקות את הדרך שבה האדם חושב כאשר הוא משחק משחק אסטרטגי. משחק סכום אפס הוא משחק שבכל שמהלך אחד טוב לשחקן אחד, הוא רע לשחקן השני. המשג "סכום אפס" נובע מזה שאם סוכמים את הרווח של השחקן הראשון ואת ההפסד של השחקן השני, הסכום יתאזן ויהיה 0. המשחק המבוך הוא משחק סכום אפס כי ככל שמהלך הוא טוב יותר לשחקן אחד, הוא רע יותר לשחקן השני.

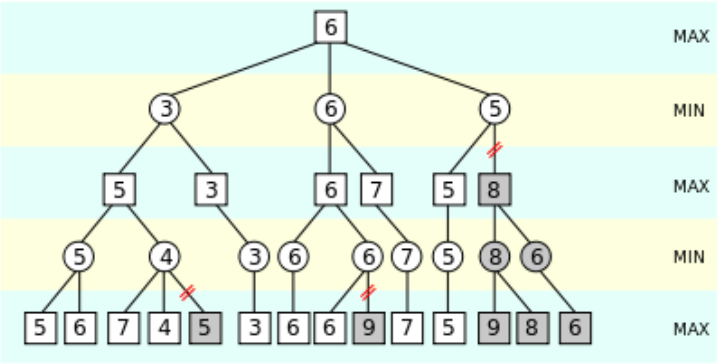
בכל משחק אסטרטגי סכום אפס, קיים אלגוריתם כלשהו שניתן בעזרתו לחשב כמה טוב מהלך מסוים. למשל בשחמט, המהלך הכי טוב הוא לעשות מט לשחקן השני. בתאוריה, תמיד יש מהלך שהוא "הכי טוב".

אלגוריתם המינימקס לוקח כנקודת הנחה ששני השחקנים משחקים בצורה האופטימלית. האלגוריתם הוא אלגוריתם רקורסיבי שבמצב האידאלי היה ממשיך עד סוף המשחק. מפני שהאלגוריתם איטי יותר בצורה מעריכית ככל שיורדים יותר לעומק, לרוב צריך להתפשר על עומק קטן יותר.

המינימקס הוא אלגוריתם רקורסיבי. בשביל לחשב מה המהלך הטוב ביותר בשביל השחקן הלבן, הוא מנסה כל מהלך אפשרי. לכל מהלך, הוא יחשב מה המהלך הכי טוב שהשחור יכול לעשות באמצעות אותו האלגוריתם. המחשב לוקח בחשבון ששני השחקנים רוצים לנצח וירצו לעשות את המהלך הטוב ביותר. זה ממשיך עד לעומק המוגדר בפעולה, או עד שמגיעים למצב של סוף המשחק. כשהאלגוריתם מגיע לעומק המתאים, הפעולה מחזירה את "ערך המשחק" במצב הזה. "ערך המשחק" גדול יותר ככל שהמצב של הלוח יותר טוב לשחקן הלבן וקטן יותר ככל שהמצב של הלוח יותר טוב לשחקן השחור. ערך של 0 אומר ששני השחקנים נמצאים במצב שווה. ניתן לראות באיור משמאל, איך המחשב באמצעות האלגוריתם של המינימקס מוצא את המהלך הטוב ביותר בהתחשבות במהלכים הבאים. במקרה של האיור, המחשב חושב 4 מהלכים קדימה. סיבוכיות האלגוריתם הוא $O(n^m)$ כאשר n הוא כמות המהלכים ו-m הוא העומק של העץ.



בפרויקט שלי, לא השתמשתי באלגוריתם ה-alpha-beta pruning. האלגוריתם הוא אופטימיזציה לאלגוריתם ה-minimax שגורם לו לרוץ יותר מהר על ידי חיסכון בחישוב של מהלכים שלא יכולים להיות המהלכים הנכונים. הוא אלגוריתם חיפוש שבמהלך החיפוש לעומק שמפסיק לחשב פתרונות חלקיים ברגע שברור שהם גרועים מפתרונות שכבר נבדקו. האלגוריתם מחזיר בדיוק את אותו הערך שהמינימקס היה מחזיר, אבל בזמן קצר יותר כי הוא לא מחשב מהלכים שאין סיבה לחשב אותם. ניתן לראות באיור משמאל את אופן הגיזום של האלגוריתם. הסיבוכיות של גיזום אלפא בטא במקרה הגרוע שווה לסיבוכיות של מינימקס תמים. במקרה הטוב ביותר, הסיבוכיות יכולה להיות $O(\sqrt{n^m})$.



רשתות תקשורת

בתכנות עם רשתות, חשוב להבין את הבסיס של רשתות תקשורת. בתקשורת בין שני מחשבים, מתרחשים הרבה פרוטוקולים הפועלים בשכבות שונות. המודל של השימוש באינטרנט להעברת מידע בין מחשבים הוא ה-Internet Protocol Suite - TCP/IP. המודל מורכב מארבע שכבות, שכבת קשר, שכבת רשת, שכבת תעבורה, ושכבת יישום. השכבה הגבוהה ביותר, שכבת היישום (Application), אחראית על הגדרת סוג התקשורת בין שני המחשבים (שרת/לקוח, שני לקוחות). פרוטוקולים חשובים בשכבת היישום הם HTTP (HyperText Transfer Protocol), הנועד להעברת קבצי HTML והמידע המוכל בתוכם, SMTP (Simple Mail Transfer Protocol) להעברת מיילים, FTP (File Transfer Protocol) להעברת קבצים וכך הלאה.

שכבת התעבורה (Transport), אחראית על ניהול התקשורת בין המחשבים, אמינות הנתונים ואמינות החיבור. הפרוטוקול הכי משומש בשכבת התעבורה הוא TCP, Transmission Control Protocol. פרוטוקול TCP מעביר ביתים של מידע בצורה מהימנה, מסודרת, ומאושרת מבחינת שגיאות. הפרוטוקול מונחה חיבור, כלומר הוא צריך שהשרת והלקוח יהיו מחוברים לפני שהוא מתחיל להעביר מידע. הפרוטוקול יכול לבקש שליחה מחדש של נתונים שלא הגיעו בצורה תקינה. שימושים ב-TCP כוללים את הWorld Wide Web, שליחת אימייל והעברת קבצים. פרוטוקול UDP User Datagram Protocol, הוא פרוטוקול תעבורה אחר שמעביר את כל המידע באופן לא אמין אך מהיר. הפרוטוקול לא בודק בכלל את תקינות המידע ולכן המידע יכול להגיע בסדר לא נכון, להגיע באופן כפול או לא להגיע בכלל. פרוטוקול זה מהיר יותר מTCP ולכן משתמשים בו לצורך real time streaming כמו שיחות וידאו.

שכבת הרשת (Internet) היא השכבה שממפה את הרשת ומנתבת את חבילות המידע לפי מיפוי זה. שכבת הרשת היא השכבה בה מתבצעות כל ההחלטות בנוגע לדרך שבה יועברו הנתונים דרך הרשת. היא זו שתקבע האם קיים קשר בין המקור ליעד, היא תבחר באיזו דרך יעברו הנתונים על פי שיקולים שונים, ביניהם - מהירות, נגישות, יעילות, עומס ועלויות. שכבת הרשת גם מעניקה כתובת לכל מכשיר קצה ברשת. כיום, הפרוטוקול השולט בשכבת הרשת הוא פרוטוקול IP - Internet Protocol. בתוך פרוטוקול IP יש גרסאות שונות, המשומשות ביותר הן IPv4 ו-IPv6. ההבדל העיקרי בין שניהם הוא גודל מרחב הנתונים, IPv4 משתמש ב-32 ביתים, כלומר מספק 4,294,967,296 כתובות ייחודיות. לעומת זאת, IPv6 משתמש במרחב של 128 ביתים, לכן בתאוריה אפשר שיהיו עד 3.4×10^{38} כתובות ייחודיות. כתובת של IPv4 מיוצגת באמצעות ארבעה מספרים מ-0 עד 255, למשל 213.0.56.136. הכתובת 127.0.0.1 שמור ככתובת ה-localhost, הכתובת שבעזרתו המחשב יכול לקרוא לעצמו. פרוטוקול IPv6 הוא הגרסה העדכנית ביותר של IP והוא פותח כתגובה לכך שהאינטרנט מתפתח במהירות גבוהה והגודל של IPv4 לא מספיק. כתובת של IPv6 לדוגמה הוא 2001:db8:3333:4444:5555:6666:7777:8888.

ניתן לראות שבניגוד ל-IPv4, הכתובת מופרדת ל-8 מספרים לעומת 4, ושכל אחד נע מ-0 לFFFF, כלומר 0 עד 65535.

שכבת הקשר (Link) היא השכבה התחתונה של המודל. זו השכבה המתקשרת עם השכבה הפיזית של המחשב, היא מחלקת את החבילות לכתובות הספציפיות שהיא אמורה לחלק אותן אליהן. שכבת הקשר היא לדוגמה Ethernet, WiFi וכו'.

פירוט המודולים

Pygame

Pygame הוא קבוצה של מודולים המשמשים לכתיבת משחקי מחשב. המודול כולל ספריות של גרפיקה ואודיו המקלות על כתיבת משחקים עם גרפיקה טובה. פייגיים רץ על כמעט כל פלטפורמה ומערכת הפעלה והיא שדרוג של הספריה SDL (Simple DirectMedia Layer). פייגיים משתמש בקוד הכתוב בAssembly ו-C באופטימיזציה מירבית.

פייגיים כתוב בצורה פשוטה, קצרה ויעילה. המודול משתמש בתכנות מונחה עצמים וכל תמונה, מסך, ציור, צליל וכו' מהווים עצמים. ניתן להשתמש במודולים ספציפיים בתוך pygame, לא חייבים להשתמש בכל הספריה אם רוצים להשתמש בפייגיים רק למטרה ספציפית.

במשחק, פייגיים הוא מה שמקשר בין המידע השמור במשחק לבין מה שהמשתמשים רואים על המסך. ללא פייגיים, המשחק עדיין היה עובד ברמה התאורטית אבל לא היה אפשר לראות את הלוח, ללחוץ על השחקן, להזיז מחסום או בכלל להבין את מה שקורה במשחק. חלק מהמחלקות העיקריות בפייגיים שהשתמשתי בהן:

`pygame.Surface` – "משטח" - אובייקט שבעזרתו ניתן להציג דברים על המסך

`pygame.font` – גופן - הדרך להציג טקסט

`pygame.Rect` – מלבן

`pygame.display` – המסך של המשחק

Pygame.gfxdraw

זוהי ספריה בתוך פייגיים שאחראית על שרטוט דמויות בגרפיקה טובה יותר. כאשר עושים import pygame, הספריה הזו לא מיובאת וצריך לייבא אותה בנפרד, כי זו ספריה נסיונית. הספרייה משתמשת בטכנולוגיית anti-aliasing בשביל לשרטט קווים, מעגלים וצורות נוספות בצורה יותר אסתטית. אני משתמש בספריה הזו בשביל הפעולות ()Board.drawmoves- ()Pawn.draw.

Threading

תהליכון (thread) הוא מושג במדעי המחשב המתאר פעולה שהמחשב עושה בזמן מסוים. במילים פשוטות, תהליכון הוא ביצוע של משימה מסוימת שהמחשב עושה באופן מבודד מהדברים האחרים שהמחשב עושה. כיום, מערכות הפעלה מאפשרות לנהל מספר תהליכונים בזמן שהוא עובד על תהליך אחד (process). בפייתון במודול threading, קיימת אפשרות לעשות שני דברים בבת אחת.

הדרך שהמחשב מסוגל לעבוד על שני דברים "בו זמנית" נקראת threading או multiprocessing. מפני שלרוב המחשבים יש רק מעבד אחד ולכן לא ניתן באמת להריץ קוד במקביל, המחשב במקום מחליף בין הפעולות שהוא עושה בקצב מהיר ובכך מדמה "מולטיטסקינג".

בפרויקט, שני תהליכונים רצים במקביל. למחלקה main יש פעולה connect שמחברת בין המחשב של השחקן (הלקוח) לשרת. הפעולה הזו רצה במקביל לעדכון של המסך של הלקוח בpygame. הסינטקס שלה הוא:

```
threading.Thread(target=self.connect).start()
```

- threading.Thread() יוצר מופע אנונימי של מחלקת Thread.
- target פונקציית המטרה של התהליכון, הפעולה שהתהליכון האנונימי עושה בזמן שהתהליכון הראשי ממשיך לעדכן את המשחק.
- start() היא פעולה פנימית של Thread שאומר לתהליכון להתחיל לעבוד.

אם לא הייתי משתמש בתהליכונים, לא היה דרך שהמחשב היה יכול גם לקבל מידע מהשרת וגם לעדכן את המסך לפי המידע הזה במקביל.

Time

מחלקת זמן. המחלקה מסוגלת להגיד למחשב להפסיק לפעול לכמות זמן מוגדרת בעזרת הפעולה:

```
time.sleep(s)
```

בפעולה, הקוד עוצר למשך s שניות וממשיך לפעול כרגיל לאחר שנגמרו s השניות. זה חשוב בשביל שהמשתמשים יקלטו מה קורה על המסך, ושמסכים לא יתחלפו תוך חלקיק שנייה.

Socket

שקע (socket) הוא נקודת קצה עבור זרם נתונים בין תהליכים ברשת מחשבים. שקעים נותנים לתהליכים שונים ברשת את היכולת לתקשר אחד עם השני. השקע הוא הבסיס לתקשורת בין מחשבים שונים ותכנות ממשקים רב משתמשים ברשת.

המודול socket הוא חלק מהספריה הסטנדרטית של פייתון ומספק ממשק ל- Berkeley Sockets API. המודול עובד ישירות עם החומרה של המחשב.

כדי לפתוח שקע בפייתון, מאתחלים אותו כך:

```
my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

`socket.socket` יוצר עצם מטיפוס socket שמקבל שני פרמטרים, משפחת הכתובות (שכבת הרשת) וסוג השקע (שכבת התעבורה). `AF_INET` אומר למחשב שמשתמשים במשפחת הכתובות IPv4. הפרמטר `SOCK_STREAM` אומר למחשב שמשתמשים בפרוטוקול TCP. צריך לאתחל את השקע גם בצד של השרת וגם בצד של הלקוח.

בשביל שהשקע של הלקוח יוכל להתחבר לשקע של השרת, יש להשתמש בפעולה:

```
server_socket.bind((HOST,PORT))
client_socket.connect((HOST,PORT))
```

הפעולה bind מעניקה לשרת את הכתובת והפורט. הכתובת HOST הוא כתובת IPv4 של המחשב שבו פועל השרת. הפורט יכול להיות כל פורט במחשב שאינו שמור. אני השתמשתי בפורט 49550.

הפעולה connect מחברת את השקע של הלקוח לשקע הנמצא בכתובת HOST דרך הפורט PORT.

פעולות חשובות נוספות בsocket שיש בהן שימוש בפרויקט שלי:

```
server_socket.listen([backlog])
```

הפעולה listen אומרת לשקע להקשיב לשקעים אחרים. השרת משתמש בזה בשביל ליצור קשר עם השקעים של הלקוחות. listen גורמת לשקע לחכות עד ששקע אחר ייצור איתו קשר. המשתנה backlog הוא משתנה אופציונלי שקובע את כמות החיבורים שהשרת מאפשר להתחבר אליו. כאשר לא מוגדר ה-backlog, המחשב משתמש בbacklog ה-default.

```
server_socket.accept()
```

הפעולה accept באה אחרי הפעולה listen. היא מחזירה tuple של השקע שמתחבר אליו והכתובת שלו.

```
my_socket.sendall(bytestring)
```

הפעולה sendall שולחת מידע ב-bytestring דרך השקע my_socket.

```
my_socket.recv(buffsize)
```

הפעולה recv מחזירה את הנתונים שהתקבל ב-my_socket, בגודל buffsize ביתים (8 סיביות).

בקוד שלי, קיימות שתי מחלקות, שרת ולקוח. הלקוחות הם השחקנים המשתמשים במשחק. השרת הוא האחראי על התקשורת בין הלקוחות ודואג שכל המידע יועבר ביניהן. המחלקות של השרת והלקוח משתמשות בשקעים בשביל ליצור קשר אחת עם השנייה.

Collections

ספריית collections בפייתון היא גם חלק מהספריה הסטנדרטית של פייתון. הספרייה כוללת מבני נתונים שנקראים containers (מכולות) שמהווים תחלופה למבני הנתונים הקיימים בשפה (list, tuple, set כו'). השתמשתי בשני מבני נתונים מספריית collections, הראשונה `collections.deque` והשנייה `collections.defaultdict`.

- collections.deque - דו-תור הוא תור שמשתמש ברשימה מקושרת לשני כיוונים. זה מבנה נתונים טוב להכנסה ושליפה משני הקצוות ומהווה תחלופה הרבה יותר מהירה ויעילה לlist. שליפה מההתחלה של deque הוא O(1) לעומת שליפה מההתחלה של רשימה שהיא O(n), כי צריך להזיז את כל איברי הרשימה שמאלה. לעומת זאת, Random Access, או גישה לאיברים באמצע דו-תור היא O(n), למרות ש-n מחולק במספר קבוע גדול ולכן לא איטי אלא אם כן מדובר בטור ארוך מאוד. גישה לאיברים באמצע רשימה רגילה היא O(1).
- collections.defaultdict - מבנה נתונים המבוסס על dictionary. האימפלמנטציה דומה מאוד, יש מפתחות וערכים. המחשב שומר את הערכים ב-Hash Table לפי ערך ה-hash של המפתח. הפעולה הבונה של defaultdict מקבלת פרמטר default_factory. הפרמטר הזה הוא אובייקט של פעולה בונה של מחלקה כלשהי שלא מקבלת פרמטרים. לאחר מכן, כשמוסיפים key חדש ל-defaultdict, ה-value שלו שווה אוטומטית ל-default_factory. למשל, אם מגדירים שה-default_factory הוא הפעולה int, הערך

הראשוני כשמגדירים key חדש למילון יהיה המספר השלם ה-default, 0. במילים אחרות, defaultdict לעולם לא יזרוק KeyError, כי אם מנסים לחפש מפתח שלא נמצא ב-defaultdict, הוא אוטומטית מוסיף לו את המפתח עם הערך של ה-defaultfactory. אני השתמשתי בבסיס נתונים הזה בשביל לשמור את כל הצעדים האפשריים בפעולה board.all_possible_moves. שם השתמשתי ב-set כ-default_factory. זה קיצר את הקוד וייעל אותו יותר.

Random

ספרייה האחראית על הגרלת דברים פסאודו-אקראיים. מספר פסאודו-אקראי הוא מספר כביכול אקראי אך מבוסס על אלגוריתם קבוע. המנוע שבו הוא משתמש הוא Mersenne Twister, אחד המנועים המשומשים ביותר בעולם להגרלת מספרים אקראיים. חשוב לציין שניתן בתאוריה למצוא את האלגוריתם של random ולכן אין להשתמש בו למערכות הגנה ואבטחה.

במשחק שלי, הספרייה random משומשת לAI בשביל שלא תמיד יעשה את אותו המהלך או יפעל לפי אותה "חשיבה". הפעולה מהספריה שהשתמשתי בה היא:

```
random.randint(1,6)
```

שמגרילה מספר אקראי שלם בין 1 ל-6. כמעט כל פעולה בספריית random משתמשת בפעולה הבסיסית - `random.random()` שמחזיר מספר עשרוני בין 0.0 ל-1.0.

Sys

ספרייה שנותנת גישה למשתנים שה-interpreter (מפרש) של פייתון עובד איתם. המפרש של פייתון הוא מה שמתרגם את פייתון מטקסט לקוד. ה-Interpreter הכי פופולרי של פייתון הוא CPython, ה-Interpreter המקורי של השפה הכתוב בשפה C. השתמשתי ב-sys בשביל לקבל מידע מהמשתמש על המוד של המשחק.

`sys.argv` הוא רשימה של כל הפרמטרים שמכניסים בשורת הפקודה כאשר מריצים קוד מקובץ בפייתון. `sys.argv[0]` הוא שם הקובץ שמריצים. ב-windows, הוא כל ה-path של הקוד. כאשר כותבים בשורת הפקודה:

```
python main.py 3
```

`sys.argv[0]` יהיה ה-path של main, למשל "C:/User/PythonCode/main.py".
`sys.argv[1]` יהיה 3. אפשר להשתמש בזה בשביל להכניס פרמטרים ישר בשורת הפקודה בלי להצטרך לחכות עד שהמשחק נפתח וטוען.

Datetime

ספרייה שמשתמשת בתאריך. מקבלת מידע מהמחשב על התאריך ונותנת פעולות רבות שניתן לעשות על תאריכים. השתמשתי בספריה רק בשביל הפעולה timeit, שמחשבת את התאריך המדויק לפני הרצת פעולה ואחרי, ולאחר מכן מחשבת את כמות הזמן שלקח לפעולה לרוץ.

`datetime.now()` מחזיר את הזמן המדויק עכשיו.

מדריך למפתח

מבנה הפרויקט

Quoridor
———— quoridor
————————— __init__.py
————————— constants.py
————————— pieces.py
————————— board.py
————————— game.py
———— network
————————— __init__.py
————————— client.py
————————— server.py
———— ai
————————— __init__.py
————————— algorithm.py
———— main.py
———— [כל קבצי התמונות]

__init__.py

הסיבה שקובץ זה חוזר על עצמו בכל תיקייה היא בגלל שקובץ __init__ הופך ספריה (directory) רגילה לחבילה (package) של פייתון. זה מאוד שימושי כי אז אפשר לעשות import למחלקות וקבועים השמורים בספריות האלה. למשל, כדי לקבל את המחלקה Pawn מהקובץ pieces.py, אפשר לעשות from quoridor.pieces import Pawn.

תיקיית quoridor

תיקייה הכוללת את כל הקבצים על מנת שיהיה אפשר לשחק את המשחק בין שני שחקנים אנושיים במחשב אחד. כולל את חוקי המשחק, גרפיקה ועוד נתונים בשביל המשחק.

- constants.py: קובץ הכולל את כל המשתנים הקבועים שאני משתמש בהם. הסיבה שבחרתי לשמור את כל המשתנים הקבועים בתוך קובץ היא כי אז ניתן לשנות גדלים בקלות. למשל, אם החלטתי שרוחב החלון יהיה 400 פיקסלים אבל אז שיניתי את דעתי ל500, במקום שאצטרך לחפש בקובץ כל פעם שאני כותב 400, אשנה רק בקובץ constants את הרוחב של החלון. המשתנים הקבועים מתחלקים למספר קטגוריות: גדלים (גודל משבצת, גודל החלון, גודל החיילים וכו'), מיקומים (המיקום ההתחלתי של

השחקנים, המיקום הסופי שצריך להגיע אליו), גופנים (של הטקסט על המסך) וצבעים (tuples של שלושה ערכים המייצגים RGB).
בנוסף לכל המשתנים הקבועים, כתבתי decorator בשם timeit שהשתמשתי בו בשלב הדיבג בשביל לראות כמה זמן לוקחות פונקציות ולנסות לקצר את הפונקציות שלוקחות יותר מדי זמן.

- pieces.py: הקובץ הכולל את המחלקה Pawn. בתחילת הפיתוח גם הייתה מחלקה wall שמחקתי כי המחלקה לא הייתה שימושית, למרות שהמשחק אמור להיות על בסיס הרעיון של תכנות מונחה עצמים. בהמשך אפרט על המחלקה Pawn.
- board.py: הקובץ הכולל את המחלקה Board. בתחילת העבודה, הייתה גם מחלקה Tile. כשראיתי שסיבוכיות זמן הריצה הייתה גבוהה מידי, שילבתי בין המחלקות Board ו-Tile והשארתי רק את Board. גם על המחלקה Board אפרט בהמשך.
- game.py: הקובץ הכולל את המחלקה Game. המחלקה מנהלת את כל מהלך המשחק ואחד המאפיינים שלו הוא מטיפוס Board (קומפוזיציה).

תיקיית network

חבילה המאפשרת לשני שחקנים לשחק אחד מול השני דרך הרשת. הקבצים כתובים בצורה כללית כך שיהיה קל מאוד לשנות את הקוד בתיקייה בשביל משחקים או פרויקטים אחרים הדורשים תקשורת בין שני מחשבים.

- server.py: קובץ שאמור לרוץ בנפרד מהפרויקט. הקובץ פותח עצם מטיפוס Server שאפרט עליו בהמשך. פותחים את הקובץ הזה משורת הפקודה והוא רץ ללא קשר למשחק. הוא מעביר את המידע בין הלקוחות.
- client.py: קובץ המכיל את המחלקה PlayerClient. כאשר מריצים את main.py, אם בוחרים באפשרות השנייה, למחלקה main מצטרפת תכונה נוספת בשם client המקשרת בין המשחק שעל המחשב לבין השרת, והשרת מקשר בין שני הלקוחות. כך מתאפשר משחק דרך רשתות התקשורת.

תיקיית AI

חבילה המאפשרת לשחקן לשחק מול המחשב.

- algorithm.py: קובץ המכיל את כל המנועים של ה-AI שהשתמשתי בהם. קיימת מחלקה AI שרוב הפעולות שלה סטטיות.

קובץ main.py

קובץ המרכז את כל התיקיות האחרות ביחד וזה הקובץ שמריצים בסוף. הקובץ כולל מחלקת main. זו המחלקה העיקרית של המשחק והמחלקה שמתחילה את המשחק באמת.

constants.py - קבועים חשובים

```
# Sizes
WIDTH, HEIGHT = 450, 670
ROWS = 9  # Amount of rows and columns, board needs to be square so rows=columns
MARGIN = 110  # Size of margin
BOARD_HEIGHT = HEIGHT - (2 * MARGIN)  # The height of the board is the height of the window - the heights of the margins

TILE_WIDTH = WIDTH//ROWS  # The width of the tiles is the width of the window // amount of columns
TILE_HEIGHT = BOARD_HEIGHT//ROWS  # The height of the tiles is the height of the board // amount of rows (==TILE_WIDTH)

WALLS = 10
WALL_WIDTH = 5
WALL_HEIGHT = 2 * TILE_HEIGHT  # Each wall is the height of 2 tiles
PAWN_RADIUS = TILE_WIDTH//3  # The diameter of a pawn is 2/3 the width of a tile
MOVE_RADIUS = TILE_WIDTH//7  # The diameter of a possible move is 2/7 the width of a tile

# Positions
WHITE_START = ((ROWS-1)//2, ROWS-1)
BLACK_START = ((ROWS-1)//2, 0)
BOTTOM_ROW = {(i,ROWS-1) for i in range(ROWS)}  # All positions in bottom row
TOP_ROW = {(i,0) for i in range(ROWS)}  # All positions in top row

# Font
FONT = pygame.font.SysFont('Comic Sans ms', 30)
SMALL_FONT = pygame.font.SysFont('Comic Sans ms', 20)

# Colors
RED = (255, 0, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
TAN = (210, 180, 140)
BROWN = (64, 32, 11)
GRAY = (150, 150, 150)
BLUE = (0, 0, 255)
GREEN = (0, 200, 0)
LIGHT_BLUE = (52, 155, 229)
LIGHT_PINK = (214, 105, 255)
```

כפי שצוין קודם, קובץ constants.py הוא הקובץ ששומר את כל המשתנים הקבועים שאני משתמש בהם בקבצים האחרים בפרויקט. שמרתי כמות גדולה של קבועים על מנת שיהיה קל לערוך את הערך של אחד מהם מבלי להצטרך לערוך כל מקום שאני משתמש בקבוע. ניתן לראות בתמונה למעלה את כל המשתנים הקבועים ואת הערכים שלהם. הפעולה היחידה שכתבתי בקובץ היא הפעולה timeit, פעולה (decorator) שהשתמשתי בה בעיקר בשלב הדיבוג בשביל לספור זמן של פעולת ולראות לאיזה פעולות לוקח הכי הרבה זמן לרוץ. כאשר הפעולה העיקרית main מסיימת לרוץ, היא מדפיסה את הזמן שלקח המשחק בעזרת הפעולה הזו.

```
def timeit(func):
    def inner(*args, **kwargs):
        tim = datetime.now()
        x = func(*args, **kwargs)
        print(datetime.now()-tim)
        return x
    return inner
```

הפעולה מקבלת פעולה אחרת כפרמטר. בתוכה, היא מגדירה פעולה חדשה שמבצעת את הפעולה המקורית, אבל כעת היא שומרת את הזמן כשהיא מתחילה לרוץ. כשהיא מסיימת להריץ את הפעולה, היא מחשבת כמה זמן עבר. הפעולה timeit מחזירה את הפעולה הפנימית שהיא עכשיו הגדירה.

pieces.py - מחלקת Pawn

```python
class Pawn:
    def __init__(self, color, pos):
        self.color = color  # Color: (R,G,B)
        self.pos = pos  # Pos: (row, column)
        self.x = self.y = 0  # Real location of piece
        self.calc_pos()  # Initiating self.x and self.y

    def __repr__(self):...

    def clone(self):...

    def calc_pos(self):...

    def move(self, new_pos):...

    def draw(self, win):...

    def __str__(self):...
```

מחלקת Pawn מורכבת מהתכונות הבאות:

- self.color: צבע החייל. מיוצג ב-tuple של שלושה מספרים שלמים מ0 עד 255 המייצגים את ערך ה-RGB של הצבע. במשחק, הצבע יהיה או WHITE או BLACK.
- self.pos: המיקום של החייל על הלוח. tuple של שני מספרים שלמים מ-0 עד 8.
- self.x: המיקום האמיתי של החייל על ציר ה-x. מספר שלם בין 0 ל-WIDTH.
- self.y: המיקום האמיתי של החייל על ציר ה-y. מספר שלם בין MARGIN ל-MARGIN + BOARD_HEIGHT.

פעולות פנימיות (תמיד מקבלות את self) של המחלקה Pawn:

- __init__: פעולה בונה של חייל. מקבל צבע ומיקום על הלוח ושומר אותם בתוך האובייקט של ה-Pawn. לאחר מכן קורא לפעולה calc_pos.
- calc_pos: פעולה המקבלת רק את self ומחשבת את x ו-y של החייל בהתאם ל-pos.
- move: מקבל מיקום חדש ומשנה את pos למיקום החדש. לאחר מכן קורא לפעולה calc_pos.
- draw: מקבל מסך (pygame.Surface) ומצייר עליו עיגול שמייצג את החייל באמצעות pygame.gfxdraw.filled_circle. משתמש בקבוע PAWN_RADIUS בשביל לקבוע את רדיוס החייל.
- __str__ ו-__repr__: פעולות המייצגות את החייל כמחרוזת.
- clone: פעולה המחזירה חייל עם אותם הנתונים כמו החייל. משומש כאשר צריך להעתיק את המשחק בשביל ה-minimax והמחסנית של ה-undo.

כל הפעולות במחלקת Pawn הן בסיבוכיות זמן ריצה קבועה O(1) למעט draw שתלוי בסיבוכיות הלא ידועה של pygame.gfxdraw.filled_circle. המחלקה Pawn פשוטה מאוד.

בהתחלה, הייתה מחלקה נוספת בשם Wall שנמחקה בעקבות זמן ריצה ארוך מדי. המחלקה wall הייתה דומה מאוד למחלקה pawn ושתיהן ירשו מהמחלקה piece. מחלקת wall הוחלפה ב-dictionary. בדוגמה למטה, המשתנה wall הוא מחסום שכרגע נבחר על ידי המשתמש והכיוון שלו בכיוון האנכי. המיקום של המחסום הוא המיקום של העכבר כאשר sel = True.

wall = {‘sel’:True, ‘dir’:1}

board.py - מחלקת לוח המשחק Board

מחלקת Board היא המחלקה שבה מתרחש המשחק. הלוח שומר את כל חוקי המשחק, את רוב הגרפיקה ואת המיקומים של החיילים והמחסומים. למחלקה יש שתי תכונות:

```
class Board:
    def __init__(self):
        self.board = []
        self.create_board()
        self.pieces = (Pawn(WHITE, WHITE_START), Pawn(BLACK, BLACK_START))

    def create_board(self):
        """
        Creates Board as 2d list. Creates ROWS*ROWS board of tiles, adds white piece on bottom and black piece on top
        """
        self.board = [[{'occupied':False, 'pos':(j*TILE_WIDTH, MARGIN+i*TILE_HEIGHT),
                        'walls':[False for _ in range(4)]} for i in range(ROWS)] for j in range(ROWS)]
        self.board[WHITE_START[0]][WHITE_START[1]]['occupied'] = True
        self.board[BLACK_START[0]][BLACK_START[1]]['occupied'] = True
```

- self.pieces - רשומה (tuple) של שני ה-Pawns. השחקן הלבן שמור במקום ה-0 והשחקן השחור במקום ה-1.
- self.board - מערך דו מימדי שכל איבר בו הוא dictionary עם 3 מפתחות: occupied יהיה False אם אין במשבצת חייל ו-True אחרת. pos הוא המיקום האמיתי של המשבצת על המסך. walls הוא רשימה באורך 4 שמבטאת את הקצוות של כל משבצת, אם יש בכל משבצת מחסום לידו. 0 - משמאל, 1 - מעל, 2 - מימין, 3 - מתחת.

הפעולות העיקריות במחלקת board הן:

- פעולה בונה ו-create_board(self): מכין את לוח המשחק והחיילים למצב ההתחלתי. סיבוכיות לינארית כאשר n הוא כמות המשבצות.
- הפעולה clone(self) - מחזירה לוח עם נתונים זהים ללוח הזה. חשוב כאשר צריך להעתיק את הלוח.
- הפעולה draw(self, win) - מציירת את הלוח על מסך (שהפעולה מקבלת כפרמטר). מציירת את כל המשבצות, את החיילים, את המחסומים וכו'. סיבוכיות לא ידועה מפני שתלויה בסיבוכיות של הפעולות של פייגיים. לפחות O(n) כש-n הוא כמות המשבצות.
- הפעולה move(self, piece, pos) - מזיז חייל ממשבצת אחת למשבצת אחרת. לא בודק אם ההזזה חוקית. פעולה טכנית שמשנה את ערכי occupied ב- self.board. O(1).
- הפעולות place_wall(self, wall) ו-unplace_wall(self,wall) - מציב מחסום על הלוח. place_wall לא בודק אם מיקום המחסום חוקי. unplace_wall לא בודק אם יש מחסום לפני שהוא מוריד אותו. פעולות טכניות שמשנה את ערכי walls ב-self.board. O(1) .
- הפעולה get_piece(self, pos)- מקבלת מיקום של משבצת ומחזירה את העצם של החייל שנמצא במשבצת, או 0 אם המשבצת ריקה. סיבוכיות קבועה.
- הפעולה winner - מחזירה WHITE אם השחקן הלבן נמצא בשורה העליונה ו-BLACK אם השחקן השחור נמצא בשורה התחתונה. מחזיר None במקרה שאין עדיין מנצח. WHITE ו-BLACK הם צבעים שהוגדרו בקובץ constants. סיבוכיות קבועה.
- הפעולה can_place_tech(self, wall) - מקבל פרמטר wall שהוא tuple של המיקום והכיוון של המחסום שרוצים להציב. בודק אם ניתן להציב את המחסום מבחינה טכנית, הוא לא חוצה/עובר על פני מחסום אחר או אחד מדפנות הלוח. סיבוכיות קבועה.
- הפעולה can_place(wall) - מופרדת מ-can_place_tech בשביל שהמחשב לא יריץ את הפעולה הזאת למקרה ש-can_place_tech הוא False. הפעולה קודם בודקת את

can_place_tech. אם הוא False, הפעולה מחזירה False. אם לא, הפעולה מציבה את המחסום לרגע, מריצה את הפעולה DFS בשביל לבדוק אם שני השחקנים יכולים להגיע לקצה השני אם מציבים את המחסום, ואז משתמשת ב-unplace_wall. מחזירה True אם DFS החזיר True בשני המקרים. במקרה הגרוע, סיבוכיות לינארית O(V+E) כאשר V הוא כמות המשבצת ו-E הוא כמות המעברים בין כל משבצת, אם ה-DFS עבר על כל המשבצות והמעברים.

- הפעולה possible_moves(self) - פעולה שמטרתה לבנות מבנה נתונים של גרף moves המיוצג בצורת Adjacency List מיושם ב-defaultdict(set). לכל מיקום על לוח המשחק יש מפתח, וה-set ששמור כ-value שלו הוא כל המיקומים שניתן להגיע אליהם מהמיקום שהחייל נמצא בו. הפעולה כוללת הרבה תנאים, בודקת אם נמצאים מחסומים או אם החייל השני צמוד אליו. בסוף מחזיר את moves. סיבוכיות הפעולה הוא סיבוכיות לינארית, כאשר *n* הוא כמות המשבצות על הלוח.
- הפעולות DFS ו-BFS_SP הן אלגוריתמים חיפוש בגרפים או עצים. קיצור של Depth First Search ו-Breadth First Search - Shortest Path. פירטתי עליהם בתיעוד נושא החקר.

```python
    def DFS(self, color, moves):
        """
        Depth first search - Find if there is a path from piece of color color to the goal

        :param color: Color of piece being checked
        :param moves: Graph of moves
        :return: True if there is a path, False if there isn't
        """
        start = self.pieces[color == BLACK].pos  # start is the position of the piece of color 'color'
        goal = TOP_ROW if color == WHITE else BOTTOM_ROW  # if piece is black, they need to get to bottom, vice versa
        seen, stack = set(), [start]  # seen is set of visited positions, stack is the stack used to check
        while stack:  # while the stack isn't empty
            node = stack.pop()  # node is the last element of the stack, a pos on the board
            seen.add(node)  # add current node to all nodes that have been checked
            for neighbor in moves[node]:  # for every move that can be done at pos node
                if neighbor in goal:  # if neighbor is in the goal row
                    return True
                if neighbor not in seen:  # if neighbor hasn't been checked yet
```

```python
def BFS_SP(self, color, moves):
    """
    Breadth First Search: Shortest path from start to goal.

    :param color: Color of piece
    :param moves: Graph (default dict) of possible moves
    :return: Shortest path or infinity if no path
    """
    start = self.pieces[color==BLACK].pos  # start is the position of the pawn of color Color
    goal = TOP_ROW if color==WHITE else BOTTOM_ROW
    seen = set()  # set of all edges (keys) already checked
    queue = deque()
    queue.append([start])  # queue is a queue of lists of the shortest paths
    if start in goal:
        return []
    while queue:
        path = queue.popleft()  # one path is removed from the head of the queue
        node = path[-1]  # node = last edge in path
        if node not in seen:  # if node hasn't been explored yet, if it was explored, this couldn't be shortest path
            neighbors = moves[node]  # neighbors -> list of all edges that node can reach
            for neighbor in neighbors:
                new_path = list(path)  # queue will append a new path for each of the neighbors of node
                new_path.append(neighbor)
                queue.append(new_path)
                if neighbor in goal:
                    return new_path  # if the end has been reached, this is the shortest path
            seen.add(node)  # check given node as seen
    return float("Inf")  # If the queue was completely checked and emptied, there's no path
```

```python
def possible_moves(self):
    moves = defaultdict(set)
    for x in range(ROWS):
        for y in range(ROWS):
            walls = self.board[x][y]['walls']
            if y > 0 and not walls[1]:  # Moving up -> No wall above and not on top row
                if self.board[x][y-1]['occupied']:  # If the tile above is occupied
                    if y-1>0 and not self.board[x][y-1]['walls'][1]:  # If no wall (or border) over piece above
                        moves[(x,y)].add((x, y-2))
                    else:  # If there's a wall over the piece above
                        if not self.board[x][y-1]['walls'][0] and x>0:  # If there's no wall left of piece above
                            moves[(x,y)].add((x-1,y-1))
                        if not self.board[x][y-1]['walls'][2] and x<ROWS-1:  # If no wall right of piece above
                            moves[(x,y)].add((x+1,y-1))
                else:  # If the tile above is empty
                    moves[(x,y)].add((x, y-1))
            if x > 0 and not walls[0]:  # Moving left -> No wall on the left and not on first column
                if self.board[x-1][y]['occupied']:  # If the tile to the left is occupied
                    if x-1>0 and not self.board[x-1][y]['walls'][0]:  # If there's no wall behind piece to the left
                        moves[(x,y)].add((x-2, y))
                    else:  # If there is a wall (or border) behind the piece to the left
                        if not self.board[x-1][y]['walls'][1] and y>0:  # If there's no wall above piece to the left
                            moves[(x,y)].add((x-1,y-1))
                        if not self.board[x-1][y]['walls'][3] and y<ROWS-1:  # If no wall below piece to the left
                            moves[(x,y)].add((x-1,y+1))
                else:  # If the tile to the left is empty
                    moves[(x,y)].add((x-1, y))
            if x < ROWS - 1 and not walls[2]:  # Moving right -> No wall on the right and not on last column
                if self.board[x+1][y]['occupied']:  # If the tile to the right is occupied
                    if x+1<ROWS-1 and not self.board[x+1][y]['walls'][2]:  # If no wall behind piece to right
                        moves[(x,y)].add((x+2, y))
                    else:  # If there is a wall (or border) behind piece to right
                        if not self.board[x+1][y]['walls'][1] and y>0:  # If there's no wall above piece to right
                            moves[(x,y)].add((x+1,y-1))
                        if not self.board[x+1][y]['walls'][3] and y<ROWS-1:  # If no wall below piece to right
                            moves[(x,y)].add((x+1,y+1))
                else:  # If the tile to the right is empty
                    moves[(x,y)].add((x+1, y))

            if y < ROWS - 1 and not walls[3]:  # Moving down -> No wall beneath and not on bottom row
                if self.board[x][y+1]['occupied']:  # If the tile below is occupied
                    if y+1<ROWS-1 and not self.board[x][y+1]['walls'][3]:  # If there's no wall under piece below
                        moves[(x,y)].add((x, y+2))
                    else:  # If there's a wall under the piece below
                        if not self.board[x][y+1]['walls'][0] and x>0:  # If there's no wall left of piece below
                            moves[(x,y)].add((x-1,y+1))
                        if not self.board[x][y+1]['walls'][2] and x<ROWS-1:  # If no wall right of piece below
                            moves[(x,y)].add((x+1,y+1))
                else:  # If the tile below is empty
                    moves[(x,y)].add((x, y+1))
    return moves
```

game.py - מחלקת המשחק Game

מחלקת Game משלבת את המחלקה של Board עם עוד חוקים, מנהלת את מהלך המשחק ובסוף מכינה את המשחק לשחקן.

```
class Game:
    """
    Game class
    """
    def __init__(self, init=True):
        if init:
            self.init()
        else:
            self.started = False

    def init(self):
        """
        Start game (or reset)
        """
        self.started = True
        self.turns = 0
        self.checked_for_winner = False
        self.selected = None  # No tile is selected
        self.wall_selected = {'sel':False, 'dir':1}  # No wall is lifted
        self.turn = WHITE  # First turn is WHITE
        self.board = Board()  # Create board
        self.winner = lambda: self.board.winner()  # Returns winner (None if no one is winning)
        self.valid_moves = []  # List of valid moves for selected piece (currently empty because no piece is selected)
        self.walls_remaining = [WALLS,WALLS]  # First is player 1, second is player 2. Walls left for each player
```

למחלקה Game יש תכונות רבות:

- self.started - אמת אם המשחק התחיל, False אם לא.
- self.turns - סופר כמה תורות עברו.
- self.checked_for_winner - משומש במחלקה Main, אמת אם כבר בדקו אם יש מנצח בתור הזה, False אחרת.
- self.selected - None אם לא בחרו בשום שחקן, עצם מטיפוס Pawn אם בחרו בשחקן.
- self.wall_selected - תכונה שמייצגת עצם של מחסום. Dictionary עם התכונות sel ו-dir. Sel הוא אמת אם הרימו מחסום ו-False אם המחסום לא מורם. Dir 0 אם המחסום אופקי ו-1 אם המחסום אנכי.
- self.turn - WHITE אם תורו של השחקן הלבן, BLACK אם תורו של השחקן השחור.
- self.board - לוח המשחק.
- self.valid_moves - set ריק כשאין שחקן ב- selected, אם יש שחקן ב-selected אז זה set של כל המהלכים שהוא יכול לעשות.
- self.walls_remaining - רשימה באורך 2 עם כמות המחסומים שנשארו לשני השחקנים.

הפעולות במחלקת Game מסתמכות על הפעולות של Board. ביחד, הן מרכיבות משחק שניתן לשחק בו.

- self.winner() - פונקציית lambda שמחזירה את self.board.winner()
- self.flip() - משנה את הכיוון של המחסום שעכשיו מורם.
- move(self, pos) - מזיז את החייל ב-self.selected למיקום pos, בתנאי שהוא מיקום שמותר להזיז את החייל אליו. מחזיר True אם הזיז את השחקן ו-False אחרת.

- place(self, pos) - מציב את self.wall_selected במיקום pos. בניגוד לפעולות במחלקה wall, קודם בודק שהצעד חוקי על ידי board.can_place(). מחזיר True אם הצליח להציב את המחסום ו-False אם לא. O(V+E), ראה board.can_place(wall).
- place_ai(self, pos, dir) - מציב מחסום בכיוון dir במיקום pos. לא בודק אם המהלך חוקי, כי ה-AI בודק בעצמו אם המהלך חוקי לפני שהוא מציב את המחסום.
  clone(self) - מחזיר משחק עם אותם הנתונים כמו המשחק הנוכחי.
  draw_moves(self, win) - מצייר את self.valid_moves כמעגלים אפורים על win בשביל שהשחקן יוכל לדעת לאן הוא יכול ללכת.
- walls_left(self, win, color=None) - כותב טקסט על המסך win. מציג לכל שחקן כמה מחסומים נשארו לו וכותב של מי התור. אם זה משחק און-ליין, color = הצבע של הלקוח.
- lift_wall(self) - מרים את wall_selected. הופך את sel ל-True. בנוסף, אם selected לא היה None, משנה אותו ל-None. אם self.walls_remaining הוא 0 בשביל השחקן, הוא לא יוכל להרים עוד מחסום.
- select(self,pos) - זו הפעולה שתרוץ כל פעם שהשחקן לוחץ על המסך. אם מחסום מורם - מנסים להציב אותו. אם חייל נבחר, מנסים להזיז אותו. אם לוחצים על חייל, בוחרים אותו. אם לוחצים מחוץ למסך או על משבצת שאין עליה כלום, מבטל את הלחיצה הקודמת שהתרחשה.
- next_turn(self) - מחליף את self.turn. מאתחל ערכים כמו self.wall_selected, self.valid_moves.
- update(self, win, pos=None, color=None) - מצייר את הלוח בכל פריים של המשחק. עושה self.draw_moves, self.walls_left, self.board.draw. pos הוא המיקום של העכבר למקרה שמחסום מורם. color הוא הצבע של הלקוח בתנאי שהמשחק הוא און ליין.
- evaluate(self) - מחזיר ערך שמייצג את המצב של המשחק. ככל שהערך גבוה יותר, המשחק יותר לטובת השחקן הלבן. ככל שהערך נמוך יותר, הוא יותר לטובת השחקן השחור. משתמש ב-BFS_SP בשביל למדוד את המרחק של השחקן מהנצחון. משומש בשביל אלגוריתם ה-minimax.

מסמך algorithm.py - מחלקת AI

מחלקת AI מנהלת את המנוע של ה-AI במצב שהמשחק הוא מסוג 3. אפרט על הפירוש של AI ועל האלגוריתמים שהשתמשתי בהם בפרק הלימוד העצמי והחקר. המחלקה כוללת מנועים שונים של AI שאחד הוא minimax.

```
class AI:

    ALL_WALLS_HORIZONTAL = *((i,j) for i in range(ROWS-1) for j in range(1,ROWS)),
    ALL_WALLS_VERTICAL = *((i,j) for i in range(1,ROWS) for j in range(ROWS-1)),

    def __init__(self, typ=0):...

    def do(self, game):...

    @staticmethod
    def place_wall_above(game):...

    @staticmethod
    def go_shortest_path(game):...

    @staticmethod
    def minimax(game, depth, maximizing_player):...

    @staticmethod
    def greediest_ai(game):...

    @staticmethod
    def random_ai(game):...

    @staticmethod
    def hesitant_ai(game):...

    @staticmethod
    def greedy_ai(game):...

    @staticmethod
    def passive_ai(game):...

    @staticmethod
    @timeit
    def pick_move(game):...
```

למחלקה יש רק תכונה אחת, type, שהוא הסוג של ה-AI שהמחשב הולך להריץ. יש בסך הכל 7 סוגים.

1. minimax - משתמש באלגוריתם מינימקס בשביל לחשב את הצעד הטוב ביותר.
2. greediest - ינסה בכל הזדמנות לשים מחסום לשחקן השני. כשלא יוכל, ילך במסלול הכי קצר לנצחון.
3. random - יגריל מספר בין 1 ל-2. אם 1, ינסה להציב מחסום מעל השחקן השני. אם לא יצליח להציב את המחסום או יגריל 2, ילך במסלול הכי קצר לנצחון.
4. hesitant - יגריל מספר בין 1 ל-4. אם 4, ינסה להציב מחסום מעל השחקן השני. אם לא יצליח או יגריל מספר שונה מ-4, הולך במסלול הכי קצר.
5. greedy - יגריל מספר בין 1 ל-3. אם 1 או 2 ינסה להציב מחסום מעל השחקן השני. אם לא יצליח או יגריל מספר 3, ילך במסלול הכי קצר לניצחון.
6. passive - תמיד הולך במסלול הכי קצר לנצחון.

אם ה-type הוא 0 או לא נמצא בין 1 ל-6, ה-type שמור כ-default type. כל תור, הוא יגריל מספר בין 2 ל-6 ואז יבחר במנוע המתאים שיחשב את הצעד הבא.

```
@staticmethod
def minimax(game, depth, maximizing_player):
    """
    Minimax Algorithm

    :param game: Game that is being checked
    :param depth: How many recursions have been done
    :param maximizing_player: True if white, False if white
    :return: Best minimax value, game after the best minimax value has been done
    """
    x = game.winner()
    if x is not None:
        print('f')
        return float('inf') if x == WHITE else float('-inf'), game
    if depth == 2:  # If depth has reached 2 or if the game is over
        return game.evaluate(), game
    best = float('-inf') if maximizing_player else float('inf')  # min value if maximizing, max if
minimizing
    best_move = None
    piece = game.board.pieces[game.turn == BLACK].pos  # position of piece that is moving
    game.select((piece[0] * TILE_WIDTH, piece[1] * TILE_HEIGHT + MARGIN))  # select piece to get valid
moves
    for move in game.valid_moves:  # For every move in the valid moves
        new_game = game.clone()  # Create copy of game
        piece = new_game.board.pieces[new_game.turn == BLACK].pos  # position of piece that is moving
        new_game.select((piece[0] * TILE_WIDTH, piece[1] * TILE_HEIGHT + MARGIN))
        new_game.move(move)  # move piece to current valid move
        # noinspection PyUnresolvedReferences
        value = AI.minimax(new_game, depth + 1, not maximizing_player)[0]  # recursion, does minimax
        if maximizing_player:  # minimax
            best = max(best, value)
        else:
            best = min(best, value)
        if best == value:
            best_move = new_game  # if the current value is the best value, the best move is the move
that was done
    if game.walls_remaining[not maximizing_player]:  # if there are any walls left for the current
player
        for wall in AI.ALL_WALLS_HORIZONTAL:  # for each possible wall
            if game.board.can_place((wall, 0)):
                new_game = game.clone()
                new_game.place_ai(wall, 0)
                value = AI.minimax(new_game, depth + 1, not maximizing_player)[0]
            else:
                continue
            if maximizing_player:
                best = max(best, value)
            else:
                best = min(best, value)
            if best == value:
                best_move = new_game
        if game.turns > 10:
            for wall in AI.ALL_WALLS_VERTICAL:
                new_game = game.clone()
                if new_game.board.can_place((wall, 1)):
                    new_game.place_ai(wall, 1)
                    value = AI.minimax(new_game, depth + 1, not maximizing_player)[0]
                else:
                    continue
                if maximizing_player:
                    best = max(best, value)
                else:
                    best = min(best, value)
                if best == value:
                    best_move = new_game
    return best, best_move
```

קובץ client.py - מחלקת Player_Client

המחלקה מנהלת את הצד של הלקוח במשחק אונליין. הלקוחות הם שני השחקנים.

```
class Player_Client:

    HOST = 'localhost'
    PORT = 49550

    def __init__(self, game):
        self.main = game
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    def close(self):
        self.socket.close()

    def con(self):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((Player_Client.HOST, Player_Client.PORT))
        self.color = self.socket.recv(8).decode('UTF-8')
        self.socket.recv(1024)
        self.main.connected = True

    def request(self):
        self.main.que.append(self.socket.recv(1000).decode('UTF-8'))

    def send(self, info):
        self.socket.send(info.encode('UTF-8'))
```

תכונות:

- HOST - כתובת ה-IP של השרת. מטעמי פרטיות החלפתי את הכתובת ב-localhost, אבל חייבת להיות כתובת ה-IP האמיתית של השרת בשביל לשחק בין שני מחשבים שונים.
- PORT - הפורט שדרכו השקע מתחבר.
- self.main - האובייקט ממחלקת main הקורא ל-Player_Client
- self.socket - השקע שאיתו מתחברים לשרת
- self.color - הצבע של השחקן במשחק

פעולות:

- פעולה בונה - מכינה את השקע
- close(self) - סוגרת את השקע בסוף המשחק
- con(self) - מתחבר לשרת בתחילת המשחק. שומרת את הצבע של השחקן.
- request(self) - מוסיף את ההודעה שהלקוח קיבל מהשרת לטור של המחלקה Main שמקבל הודעות
- send(self) - שולח מידע לשרת

קובץ server.py - מחלקת השרת

מחלקת PlayerOneLeft היא הודעת שגיאה מיוחדת שקוראים לה כאשר השחקן הראשון עזב את המשחק לפני שהשחקן השני הספיק להצטרף.

```
class Server:

    HOST = socket.gethostbyname(socket.gethostname())
    PORT = 49550

    def __init__(self):...

    def connect(self):...

    def send(self):
        while True:
            try:
                self.conn2.sendall(self.conn1.recv(1000))
                self.conn1.sendall(self.conn2.recv(1000))
            except (ConnectionResetError, ConnectionAbortedError):
                return


if __name__ == "__main__":
    print('Server Initiated.')
    try:
        x = Server()
    except PlayerOneLeft:
        print('Server shutting down because a player left before the game began. Please start over.')
    print('Server Closed.')
```

תכונות:

- HOST - כתובת ה-IP של המחשב
- PORT - הפורט שבו השקע מתחבר
- self.socket - השקע של השרת
- self.conn1, self.conn2 - השקעים של הלקוחות המחוברים לשרת
- self.addr1, self.addr2 - הכתובות של הלקוחות המחוברים לשרת

פעולות:

- פעולה בונה - מאתחלת את השקע וקוראת ל-()self.connect
- connect(self) - מתחבר לשני הלקוחות. שולח לכל לקוח את הצבע שלו. למקרה של שגיאה בגלל שאחד השחקנים עזב לפני שהמשחק התחיל, מעלה את החריגה PlayerOneLeft. בסוף קורא ל-()self.send.
- send(self) - לולאה שרצה לנצח (עד שאחד השחקנים עוזב) ומעבירה מידע מאחד הלקוחות לשני.

כאשר מריצים את הקובץ server.py משורת הפקודה, זה פותח שרת. הקוד של השרת יכול להיות מופרד לגמרי משאר הפרויקט.

קובץ Main.py ומחלקת Main

```
class Main:

    SCREENS = [pygame.image.load('Quoridor-Pick Mode.png'), pygame.image.load('Quoridor-Local.png'),
               pygame.image.load('Quoridor-Multiplayer wait.png'), pygame.image.load('Quoridor-AI.png'),
               pygame.image.load('Quoridor-Rules.png'), pygame.image.load('White wins.png'),
               pygame.image.load('Black wins.png')]  # list of all images shown throughout game.

    def __init__(self, typ):
        pygame.init()
        self.WIN = pygame.display.set_mode((WIDTH, HEIGHT))  # creates pygame display as main window
        pygame.display.set_caption('Quoridor')
        self.game = Game(False)  # self.game is a game which isn't initiated yet
        self.type = typ  # type of game. 1: local, 2: online 3: vs ai
        self.ai = AI()  # AI engine, set as default value
        quoridor = self.SCREENS[0]  # opening screen
        self.WIN.blit(quoridor, (0,0))
```

מחלקת Main היא המחלקה שמרכזת את כל המשחק למחלקה אחת. זו המחלקה המורכבת ביותר בפרויקט.

תכונות:

- self.WIN - מסך המשחק של פייגיים.
- Main.SCREENS - רשימה של כל התמונות שמופיעות לאורך המשחק.
- self.game - המשחק. לא מאותחל.
- self.type - סוג המשחק, 1 - בין שני שחקנים באותו מחשב. 2 - בין שני שחקנים באינטרנט. 3 - בין השחקן למחשב.
- self.ai - מנוע ה-AI. חשוב רק אם ה-type הוא 3.

תכונות במקרה שה-type 2:

- self.client - עצם מטיפוס Player_Client שמקבל את self.
- self.playing - תכונה שכל עוד שהיא True, המשחק ממשיך לרוץ.
- self.connected - True אם מחובר לשרת, False אחרת.
- self.que - הטור (Deque) שהמידע שמתקבל מהשקע נכנס אליו.

פעולות:

- white_wins(self) ו-black_wins(self): מציג מסך שמראה שהשחקן הלבן / השחור ניצח.
- connect(self) - מחברת את המשחק לשרת.
- client_listen(self) - תמיד מקשיב בזמן שהמשחק רץ.
- wait(self) - מציג את מסך הפתיחה עד שהשחקן בוחר
- main(self) - הפעולה העיקרית שמנהלת את כל המשחק. ניתן לראות איך הוא פועל בתרשים הארכיטקטורה.
- multi(self) - הפעולה העיקרית במצב של משחק אונליין. ניתן לראות את אופי הפעולה שלה בתרשים הארכיטקטורה

הפעולות main ו-multi מופיעות למטה. בשביל לראות איך הן פועלות, ראה את תרשים הארכיטקטורה.

```
@timeit  # print time game was running in the end
    def main(self):
        """
        Game
        """
        self.wait()  # while the player hasn't selected the mode, show loading screen.
        if self.type == 4:  # if player closed the game in self.wait
            return
        if self.type == 2:
            self.multi()
            return
        run = True
        undo_clicked = [1,1,True] if self.type == 1 else [1,0,True]
        """each human player can undo once throughout game. When undo_clicked[2] = True, undo
can't be clicked
         (first turn or if ctrl z is already pressed)"""
        while run:
            screen = self.SCREENS[self.type]  # 1 if local multiplayer, 3 if AI
            if not self.game.started:  # while game hasn't been initiated
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:  # close screen
                        run = False
                    if event.type == pygame.MOUSEBUTTONUP and self.type == 1:  # start local
game when screen is clicked
                        self.game.init()
                        self.game_stack.append(Game())  # add initial game to stack for undo
                    if self.type == 3:
                        if event.type == pygame.KEYUP:  # set up AI based on number clicked
                            typ = pygame.key.name(event.key)
                            try:
                                self.ai = AI(int(typ))
                            except ValueError:  # a string or incorrect number was entered
                                self.ai = AI()
                            self.game.init()
                            self.game_stack.append(Game())  # the first item in the game
stack is the initial game
                self.WIN.blit(screen, (0,0))
                pygame.display.update()
            else:
                if self.game.turns > 13 and not self.game.checked_for_winner:  # check for
winner only when turn starts
                    win = self.game.winner()
                    if win:  # and only if its technically possible for there to be a winner
(14 turns)
                        self.winners[win]()
                        break
                    self.game.checked_for_winner = True
                if self.type == 1 or self.game.turn==WHITE:  # if a human player is playing
                    for event in pygame.event.get():
                        if event.type == pygame.QUIT:  # if the game was closed
                            self.winners[tuple(255-i for i in self.game.turn)]()  # display
player who didn't quit
                            run = False
```

```
                        if event.type == pygame.MOUSEBUTTONUP:  # select tile
                            pos = pygame.mouse.get_pos()
                            temp = self.game.turns
                            self.game.select(pos)
                            if self.game.turns > temp and any(undo_clicked[0:2]):  # if the
turn has changed
                                self.game_stack.append(self.game.clone())  # and undo hasn't
been used by both players
                                self.game_stack[-1].turns = self.game.turns
                                undo_clicked[2] = False

                        if event.type == pygame.KEYUP:
                            if event.key == pygame.K_SPACE:  # lift wall
                                [self.game.lift_wall,
self.game.unlift][self.game.wall_selected['sel']]()  # lift if not
                                # lifted, don't lift if lifted
                            if event.key == pygame.K_f:  # flip wall
                                self.game.flip()
                            if event.key == pygame.K_z and self.game.turns > 0:
                                undo_clicked[2] = False
                    keys = pygame.key.get_pressed()
                    if keys[pygame.K_LCTRL] and keys[pygame.K_z] and \
                            undo_clicked[self.game.turn==BLACK] and not undo_clicked[2]:
                        self.game_stack.pop()
                        self.game = self.game_stack[-1]
                        undo_clicked[self.game.turn==WHITE] = 0
                        print(f'{"White" if self.game.turn == WHITE or self.type == 3 else
"Black"} undid turn.')
                        undo_clicked[2] = True
                        if self.type == 3:
                            self.game_stack.pop()
                            self.game = self.game_stack[-1]

                elif run and self.type == 3:  # it's ai's turn
                    self.ai_move()
                    if undo_clicked[0]:
                        self.game_stack.append(self.game.clone())
                        self.game_stack[-1].turns = self.game.turns

                if run:
                    self.game.update(self.WIN, pygame.mouse.get_pos())  # always update the
screen
```

```
    def multi(self):
        """
        Multiplayer Game.
        """
        self.que = collections.deque()  # queue that will save all of the received messages
        self.connected = False  # will be true when the other thread will connect to the
server
        self.playing = True  # will be false when the game is over / aborted. used to
communicate between the threads
        Thread(target=self.connect).start()  # thread to connect to server while screen loads
        self.WIN.blit(self.SCREENS[2], (0,0))
        time.sleep(0.5)  # wait for other thread to finish
        try:
            self.client.send('hello!')  # try to send message to server
        except OSError:  # raised if there's no server
            print("Server hasn't been opened yet.")
            return
        while not self.connected and self.playing:
            try:
                self.client.color  # see if the server sent a color or if it's currently
running a different game.
            except AttributeError:  # self.client doesnt have the attribute color because
it's stuck on recv(8) in con
                print('A game is currently taking place. Please wait until the current game
ends.')
                self.client.close()
                return
            for event in pygame.event.get():
                if event.type == pygame.QUIT:  # X has been clicked while waiting for other
player to connect
                    self.playing = False
                    self.client.send('Q')
                    self.client.close()
            pygame.display.update()
        self.game.init()  # start game
        run = True  # for game loop
        Thread(target=self.client_listen).start()  # start constant listening
        turns = {BLACK:'B', WHITE:'W'}  # (0,0,0):'B", (255,255,255):'W"
        if self.client.color == 'Q':  # if the other player left before the game started
            print('The white player has left the game.')
            self.black_wins()
            return
        print(f"You are {'Black' if self.client.color=='B' else 'White'}.")  # prints on
screen player's color
        wall_selected_multi = False  # boolean to draw wall on board while the wall is lifted
        while run:
            if not self.playing:  # if the other thread stopped
                return
            if self.game.turns > 13 and not self.game.checked_for_winner:  # if it's possible
for there to be a winner
                win = self.game.winner()
                if win:
                    self.winners[win]()
                    self.playing = False
                    self.client.close()
                    break
                self.game.checked_for_winner = True  # there's no winner. wait until next
move and don't check again
            if turns[self.game.turn] == self.client.color:  # if it's the player's turn
                for event in pygame.event.get():
```

```
                    if event.type == pygame.QUIT:  # if X was clicked
                        print('You forfeit.')
                        self.client.send('Q')  # let other client know that game is over
                        self.winners[self.client.color]()
                        self.client.close()
                        return
                    if event.type == pygame.MOUSEBUTTONUP:  # screen was clicked
                        ms = pygame.mouse.get_pos()
                        legal = self.game.select(ms)
                        if legal != 'illegal':  # if the move was legal
                            st = f"S{ms[0]:3d},{ms[1]:3d}"  # send coordinates to other
player, always length 8
                            wall_selected_multi = False  # for updating screen
                            self.client.send(st)
                        else:
                            self.client.send('L')
                    if event.type == pygame.KEYUP:
                        if event.key == pygame.K_SPACE:
                            self.game.lift_wall()
                            wall_selected_multi = True
                            self.client.send('L')
                        if event.key == pygame.K_f:
                            self.game.flip()
                            self.client.send('F')
            else:
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        print('You forfeit.')
                        self.client.send('Q')
                        self.winners[self.client.color]()
                        self.client.close()
                        return

            self.game.update(self.WIN, pygame.mouse.get_pos() if wall_selected_multi else
None, self.client.color)
            self.client.send('0')  # send stream of data at all times
            if len(self.que) > 0:  # if the request queue has data
                received = self.que.popleft()  # get earliest message
                place = received.find('S')  # if message contains S, returns position of s,
else return -1
                if place != -1:  # if message contains S
                    self.game.select((int(received[place+1:place+4]),
int(received[place+5:place+8])))  # select at pos
                if 'L' in received:  # if other player lifted wall
                    self.game.lift_wall()
                if 'F' in received:  # if other player flipped wall
                    self.game.flip()
                if 'Q' in received:
                    self.playing = False
                    print('The other player has left the game.')
                    self.winners[{'B':BLACK, 'W':WHITE}[self.client.color]]()
                    return
        return 0
```

רפלקציה

תהליך העבודה של הפרויקט בסייבר היה תהליך משמעותי שהתפתחתי הרבה בתחום מדעי המחשב והבנת מערכות.

היעד ההתחלתי של הפרויקט היה להכין את המשחק "המבוך" בין שני שחקנים ולתכנת AI שיידע לשחק נגד שחקן אחד. עמדתי ביעד שלי ובנוסף הוספתי חלק של רשתות בשביל שיהיו 3 מימדים למשחק. אני גאה בתוצר הסופי ורואה בו את העמידה שלי באתגרים והיצר הסקרן שלי שתמיד רצה לשפר את המשחק עוד ועוד.

הקושי הגדול ביותר בתכנות המשחק היה זמן הריצה של הקוד. בתחילת העבודה, למחשב היה יכול לקחת עד דקה וחצי לחשב את המהלך של ה-AI. הצלחתי להתגבר על הקושי הזה באמצעות שימוש באלגוריתמים שונים וצמצום בקוד של המשחק. השתמשתי בידע שרכשתי על מבני נתונים מתקדמים יותר כמו גרפים ו-defaultdict בשביל לכתוב קוד שיהיה יותר יעיל מבחינת מקום ומבחינת זמן. הצלחתי לצמצם את הזמן שייקח למחשב לחשב מהלך ל-8 שניות בלבד במקרה הגרוע (המספר המדויק משתנה ממחשב למחשב). כל פעולה שכתבתי, כתבתי בדרך שתהיה הכי יעילה וחסכנית - במקום זמן ואורך הקוד.

אנשים שנעזרתי בהם הם בעיקר המורה אתי ששלחה לי חומרים חשובים בשביל ללמוד עוד ולשפר את הקוד. בנוסף נעזרתי בקהילת Stack Overflow ופרויקטים דומים שמצאתי ב- open source ב-Github. לא מצאתי קוד דומה למשחק שלי מבחינת AI או רשתות ולכן אני מביא חידוש, ובסגוף תהליך העבודה אעלה את הקוד ל-Github בשביל שמתכנתים חדשים ילמדו לפתח משחקים ויבינו את מה שהם עושים בזכות התיעוד שלי.

להמשך, אני לוקח כלים חשובים. עד שהתחלתי לעבוד על הפרויקט, הייתי מודע לסיבוכיות זמן ריצה ולסימון ה-O הגדולה, אבל לא הבנתי את החשיבות שלה. כיום, אני מבין שבעיקר בתכנות משחקי מחשב, הדבר החשוב ביותר אחרי שהמשחק עובד, הוא לדאוג שייקח לקוד כמה שפחות זמן לרוץ. בנוסף, השתמשתי ב-Exception Handling, דבר שלרוב אני נמנע להשתמש בו ובמקום משתמש בהרבה if-ים מיותרים. טיפול בחריגים בדרך הזו קיצר לי את העבודה בצורה משמעותית ולעתיד אדע. בנוסף, נכנסתי לעולם הרשתות ולמדתי דברים חדשים על איך תקשורת בין מחשבים עובדת. התחלתי להתעניין בפייתון בכיתה ה' ומאז, כל פרויקט שאני עובד עליו אני מרגיש שאני לומד יותר ויותר על השפה. השתמשתי במודולים שאף פעם לא התנסיתי בהם והעבודה הייתה מאוד חדשה בשבילי. בסוף תהליך העבודה אני מרגיש כמו תכניתן הרבה יותר מנוסה וממה שהייתי לפני.

ביבליוגרפיה

- https://github.com/bojotamara/python-chess/blob/master/modules/pieces.py- משחק שחמט מתוכנת בפייתון עם minimax
- http://lode.ameije.com/quoridor/Rules/quoridor_rules.html - חוקי המבוך
- https://github.com/techwithtim/Python-Checkers - משחק דמקה מתוכנת בפייתון
- https://medium.com/cantors-paradise/dijkstras-shortest-path-algorithm-in-python-d955744c7064 - אלגוריתם חיפוש של דייקסטרה.
- https://www.mcs.anl.gov/~itf/dbpp/text/node35.html - אלגוריתמים שונים של חיפוש.
- https://www.geeksforgeeks.org/building-an-undirected-graph-and-finding-shortest-path-using-dictionaries-in-python/ - אלגוריתם BFS בפייתון עם dictionaries.
- https://cis.temple.edu/~vasilis/Courses/CIS603/Lectures/l7.html#:~:text=The%20time%20complexity%20of%20minimax,the%20leaves%20of%20the%20tree.
- https://docs.python.org/3/library/ - תיעוד של פייתון 3 וכל הספריות שלו.
- https://en.wikipedia.org/wiki/ - ויקיפדיה בשביל רשתות, אלגוריתם חיפוש, minimax, מבני נתונים, רקע על המבוך.

נספחים

במהלך כתיבת הפרויקט, היו הרבה פעולות, מחלקות וחישובים שלא עשיתי בהן שימוש בקוד הסופי. זה נובע ממספר גורמים: סיבוכיות זמן ריצה גבוהה מידי, סיבוכיות מקום גבוהה מידי, קוד ארוך ומסובך מדי או חוסר רצון לשנות את הקוד הקיים בשביל להתאים לקוד החדש.

רציתי להשתמש ב-asyncio בשביל לטפל במסך ה-pygame שלא ייתקע בזמן שהמחשב מחשב את המהלך. החלטתי לוותר על השימוש בזה בגלל שהוא האריך את הזמן שלקח למחשב לחשב את המהלך פי יותר משלושה. זהו חסרון בפרויקט שלי, בזמן שהמחשב עושה את המהלך לא ניתן לעשות כלום והמחשב אף עלול לחשוב שהמשחק נתקע.

רציתי להשתמש ב-Context Manager בשביל לטפל ברשתות. חשבתי שיהיה יותר הגיוני שה-Server וה-Client ייסגרו לבד בסוף הקוד וניתן יהיה לכתוב:

```
with Server() as s:
```

לא השתמשתי בזה בסוף כי זה לא התאים לקוד הנוכחי ולא רציתי לשנות את התשתיות הקיימות של הקוד. בנוסף, לא היה בזה צורך אמיתי וזה רק היה מפשט את הקוד.

כתבתי פעולה win_possible שמחשבת אם ניצחון הוא אפשרי. זו הייתה פעולה רקורסיבית שעברה על כל משבצת ועל כל השכנים של המשבצת והחזירה אמת אם ניתן להגיע מההתחלה של הלוח לסוף. זה היה אלגוריתם שכתבתי בעצמי לפני שלמדתי על אלגוריתמים של חיפוש, ובסוף החלפתי את הפעולה הזו בפעולות BFS ו-DFS.

כתבתי מחלקה Tree, עץ החלטות, ופעולה game_tree שבונה עץ החלטות של כל המשחקים האפשריים. האלגוריתם של המינימקס היה רץ על העץ, במקום על המשחק. וויתרתי על המחלקה והפעולה כי הן לא היו שימושיות ובמקום השתמשתי במבנה של המינימקס בשביל להוות עץ.

כתבתי מחלקה Wall כפי שפירטתי בפרק pieces.py. מחקתי אותה בשביל לחסוך במקום כי ראיתי שהפעולה הזו לא שימושית וגורמת לבעיות של מקום במחשב כי בכל מקרה היה צורך רק במחסום אחד בכל עת, ולכן לא הייתה סיבה לשמור 20 מחסומים שצריך להעתיק אותם אלפי פעמים בתוך ה-minimax.

מצורף בעמודים הבאים כל הקוד של הפרויקט (נכון ל-17.05.21):

constants.py

```
import pygame.font
from datetime import datetime
pygame.font.init()

# Sizes
WIDTH, HEIGHT = 450, 670
ROWS = 9  # Amount of rows and columns, board needs to be square so rows=columns
MARGIN = 110  # Size of margin
BOARD_HEIGHT = HEIGHT - (2 * MARGIN)  # The height of the board is the height of the window -
the heights of the margins

TILE_WIDTH = WIDTH//ROWS  # The width of the tiles is the width of the window // amount of
columns
TILE_HEIGHT = BOARD_HEIGHT//ROWS  # The height of the tiles is the height of the board //
amount of rows (==TILE_WIDTH)

WALLS = 10
WALL_WIDTH = 5
WALL_HEIGHT = 2 * TILE_HEIGHT  # Each wall is the height of 2 tiles
PAWN_RADIUS = TILE_WIDTH//3  # The diameter of a pawn is 2/3 the width of a tile
MOVE_RADIUS = TILE_WIDTH//7  # The diameter of a possible move is 2/7 the width of a tile

# Positions
WHITE_START = ((ROWS-1)//2, ROWS-1)
BLACK_START = ((ROWS-1)//2, 0)
BOTTOM_ROW = {(i,ROWS-1) for i in range(ROWS)}  # All positions in bottom row
TOP_ROW = {(i,0) for i in range(ROWS)}  # All positions in top row

# Font
FONT = pygame.font.SysFont('Comic Sans ms', 30)
SMALL_FONT = pygame.font.SysFont('Comic Sans ms', 20)

# Colors - RGB form tuples
RED = (255, 0, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
TAN = (210, 180, 140)
BROWN = (64, 32, 11)
GRAY = (150, 150, 150)
BLUE = (0, 0, 255)
GREEN = (0, 200, 0)
LIGHT_BLUE = (52, 155, 229)
LIGHT_PINK = (214, 105, 255)


def timeit(func):
    def inner(*args, **kwargs):
        tim = datetime.now()
        x = func(*args, **kwargs)
        print(f'{func.__name__} executed in {datetime.now()-tim}')
        return x
    return inner
```

pieces.py

```
import pygame
import pygame.gfxdraw
from .constants import *


class Pawn:
    def __init__(self, color, pos):
        self.color = color  # Color: (R,G,B)
        self.pos = pos  # Pos: (row, column)
        self.x = self.y = 0  # Real location of piece
        self.calc_pos()  # Initiating self.x and self.y

    def __repr__(self):
        return f"{'Black' if self.color==BLACK else 'White'} pawn at {self.pos}"

    def clone(self):
        a = Pawn(self.color, self.pos)
        return a

    def calc_pos(self):
        """x,y will be equal to real x,y position on screen"""
        self.x = self.pos[0]*TILE_WIDTH + TILE_WIDTH // 2
        self.y = self.pos[1]*TILE_HEIGHT + TILE_HEIGHT//2 + MARGIN

    def move(self, new_pos):
        """Move piece from self.pos to new_pos"""
        self.pos = new_pos
        self.calc_pos()

    def draw(self, win):
        """Draw piece on win"""
        pygame.gfxdraw.filled_circle(win, self.x, self.y, PAWN_RADIUS, self.color)

    def __str__(self):
        return f"Pawn at ({self.pos[0]}, {self.pos[1]})"
```

board.py

```
import pygame.gfxdraw
from .pieces import *
from collections import defaultdict, deque


class Board:
    def __init__(self):
        self.board = []
        self.create_board()
        self.pieces = (Pawn(WHITE, WHITE_START), Pawn(BLACK, BLACK_START))

    def create_board(self):
        """
        Creates Board as 2d list. Creates ROWS*ROWS board of tiles, adds white piece on
bottom and black piece on top
        """
        self.board = [[{'occupied':False, 'pos':(j*TILE_WIDTH, MARGIN+i*TILE_HEIGHT),
                        'walls':[False for _ in range(4)]} for i in range(ROWS)] for j in
range(ROWS)]
        self.board[WHITE_START[0]][WHITE_START[1]]['occupied'] = True
        self.board[BLACK_START[0]][BLACK_START[1]]['occupied'] = True

    def __getitem__(self, item):
        return self.board[item]

    def clone(self):
        a = Board()
        for i in range(len(self.board)):
            for j in range(len(self.board[i])):
                a.board[i][j]['occupied'] = self.board[i][j]['occupied']
                a.board[i][j]['pos'] = self.board[i][j]['pos']
                walls = self.board[i][j]['walls']
                a.board[i][j]['walls'] = [*walls]
        a.pieces = (self.pieces[0].clone(), self.pieces[1].clone())
        return a

    def get_piece(self, pos):
        """:param pos: Row,col of tile
        :return: The piece that is in the tile. (0 if no piece is in tile)"""
        if self.pieces[0].pos == pos:
            return self.pieces[0]
        elif self.pieces[1].pos == pos:
            return self.pieces[1]
        return 0

    def draw(self, win):
        """
        Draws board (and margins)

        :param win: Screen (window)
        """
        pygame.draw.rect(win, TAN, pygame.Rect(0, 0, WIDTH, MARGIN))  # Top margin
        pygame.draw.rect(win, BROWN, pygame.Rect(0, MARGIN, WIDTH, BOARD_HEIGHT))  # Board
background
        pygame.draw.rect(win, TAN, pygame.Rect(0, MARGIN + BOARD_HEIGHT, WIDTH, MARGIN))  #
Bottom margin
        for i in range(ROWS):
```

```
            for j in range(ROWS):  # For every single tile on board
                rect = pygame.Rect(self.board[i][j]['pos'], (TILE_WIDTH, TILE_HEIGHT))  #
Pygame rectangle of tile
                pygame.draw.rect(win, TAN, rect, 4)
                if self.board[i][j]['walls'][0]:  # If the item isn't False
                    pygame.draw.line(win, RED, (i * TILE_WIDTH, j * TILE_HEIGHT + MARGIN -
1),
                                     (i * TILE_WIDTH, (j + 1) * TILE_HEIGHT+MARGIN),
WALL_WIDTH + 1)
                if self.board[i][j]['walls'][1]:
                    pygame.draw.line(win, RED, (i * TILE_WIDTH - 1, j * TILE_HEIGHT +
MARGIN),
                                     ((i + 1) * TILE_WIDTH, j * TILE_HEIGHT + MARGIN),
WALL_WIDTH + 1)
        for pawn in self.pieces:
            pawn.draw(win)

    def move(self, piece, pos):
        """
        Move piece from one position to another

        :param piece: Moving piece
        :param pos: New position
        """
        self.board[piece.pos[0]][piece.pos[1]]['occupied'] = False  # The tile the piece was
in is no longer occupied
        self.board[pos[0]][pos[1]]['occupied'] = True  # The tile the piece is moving to is
now occupied.
        piece.move(pos)

    def place_wall(self, wall):
        """
        Place wall in pos

        :param wall: Tuple (pos, dir)
        """
        x,y = wall[0]  # x:row, y:col
        if wall[1] == 1:  # If wall is vertical
            self.board[x][y]['walls'][0] = True
            self.board[x-1][y]['walls'][2] = True
            self.board[x][y+1]['walls'][0] = True
            self.board[x-1][y+1]['walls'][2] = True
        else:  # If wall is horizontal
            self.board[x][y]['walls'][1] = True
            self.board[x][y-1]['walls'][3] = True
            self.board[x+1][y]['walls'][1] = True
            self.board[x+1][y-1]['walls'][3] = True

    def unplace_wall(self, wall):
        """
        Remove wall from board.

        :param wall: Tuple of (pos, dir)
        """
        x,y = wall[0]  # row,col of top/left point of wall
        if wall[1] == 1:
            self.board[x][y]['walls'][0] = 0
            self.board[x-1][y]['walls'][2] = 0
            self.board[x][y+1]['walls'][0] = 0
            self.board[x-1][y+1]['walls'][2] = 0
```

```
        else:  # If wall is horizontal
            self.board[x][y]['walls'][1] = 0
            self.board[x][y-1]['walls'][3] = 0
            self.board[x+1][y]['walls'][1] = 0
            self.board[x+1][y-1]['walls'][3] = 0

    def can_place_tech(self, wall):
        """
        Checks if given wall can be placed in given pos technically (not crossing other walls
or the side of board)

        :param wall: Tuple -> pos, direction
        :return: True if wall can be placed, False if not (Doesn't place wall either case)
        """
        x, y = wall[0]
        if y > ROWS-1 or y < 0:
            return False
        if wall[1] == 1:
            if x == 0 or x == ROWS or y >= ROWS-1 or y < 0:
                return False
            if self.board[x][y]['walls'][0] or self.board[x][y+1]['walls'][0]:  # If theres a
wall in same col and pos
                return False
            if self.board[x-1][y+1]['walls'][1] and self.board[x][y+1]['walls'][1]:  # If
wall is crossing another wall
                return False
        else:
            if x >= ROWS-1 or x < 0 or y == 0 or y == ROWS:
                return False
            if self.board[x][y]['walls'][1] or self.board[x+1][y]['walls'][1]:  # If theres a
wall in same row and pos
                return False
            if self.board[x+1][y-1]['walls'][0] and self.board[x+1][y]['walls'][0]:  # If
wall is crossing another wall
                return False
        return True

    def can_place(self, wall):
        """
        Checks if a wall can be placed at pos

        :param wall: Tuple -> (pos, dir) -> ((int,int), int)
        :return: Whether wall can be placed or not
        """
        x = False
        if self.can_place_tech(wall):  # If walls aren't intercepting, crossing each other
            self.place_wall(wall)  # Place wall (only for sake of seeing if move is illegal
            possible = self.possible_moves()  # Only needed for win_possible function. Dict
of all moves
            x = self.DFS(WHITE, possible) and self.DFS(BLACK, possible)  # True if there is a
path to end
            self.unplace_wall(wall)  # Unplace wall (this function is only supposed to check
if the wall can be placed)
        return x  # If move is legal, return true else false

    def possible_moves(self):
        moves = defaultdict(set)
        for x in range(ROWS):
            for y in range(ROWS):
                walls = self.board[x][y]['walls']
```

```
                if y > 0 and not walls[1]:  # Moving up -> No wall above and not on top row
                    if self.board[x][y-1]['occupied']:  # If the tile above is occupied
                        if y-1>0 and not self.board[x][y-1]['walls'][1]:  # If no wall (or
border) over piece above
                            moves[(x,y)].add((x, y-2))
                        else:  # If there's a wall over the piece above
                            if not self.board[x][y-1]['walls'][0] and x>0:  # If there's no
wall left of piece above
                                moves[(x,y)].add((x-1,y-1))
                            if not self.board[x][y-1]['walls'][2] and x<ROWS-1:  # If no wall
right of piece above
                                moves[(x,y)].add((x+1,y-1))
                    else:  # If the tile above is empty
                        moves[(x,y)].add((x, y-1))
                if x > 0 and not walls[0]:  # Moving left -> No wall on the left and not on
first column
                    if self.board[x-1][y]['occupied']:  # If the tile to the left is occupied
                        if x-1>0 and not self.board[x-1][y]['walls'][0]:  # If there's no
wall behind piece to the left
                            moves[(x,y)].add((x-2, y))
                        else:  # If there is a wall (or border) behind the piece to the left
                            if not self.board[x-1][y]['walls'][1] and y>0:  # If there's no
wall above piece to the left
                                moves[(x,y)].add((x-1,y-1))
                            if not self.board[x-1][y]['walls'][3] and y<ROWS-1:  # If no wall
below piece to the left
                                moves[(x,y)].add((x-1,y+1))
                    else:  # If the tile to the left is empty
                        moves[(x,y)].add((x-1, y))
                if x < ROWS - 1 and not walls[2]:  # Moving right -> No wall on the right and
not on last column
                    if self.board[x+1][y]['occupied']:  # If the tile to the right is
occupied
                        if x+1<ROWS-1 and not self.board[x+1][y]['walls'][2]:  # If no wall
behind piece to right
                            moves[(x,y)].add((x+2, y))
                        else:  # If there is a wall (or border) behind piece to right
                            if not self.board[x+1][y]['walls'][1] and y>0:  # If there's no
wall above piece to right
                                moves[(x,y)].add((x+1,y-1))
                            if not self.board[x+1][y]['walls'][3] and y<ROWS-1:  # If no wall
below piece to right
                                moves[(x,y)].add((x+1,y+1))
                    else:  # If the tile to the right is empty
                        moves[(x,y)].add((x+1, y))

                if y < ROWS - 1 and not walls[3]:  # Moving down -> No wall beneath and not
on bottom row
                    if self.board[x][y+1]['occupied']:  # If the tile below is occupied
                        if y+1<ROWS-1 and not self.board[x][y+1]['walls'][3]:  # If there's
no wall under piece below
                            moves[(x,y)].add((x, y+2))
                        else:  # If there's a wall under the piece below
                            if not self.board[x][y+1]['walls'][0] and x>0:  # If there's no
wall left of piece below
                                moves[(x,y)].add((x-1,y+1))
                            if not self.board[x][y+1]['walls'][2] and x<ROWS-1:  # If no wall
right of piece below
                                moves[(x,y)].add((x+1,y+1))
                    else:  # If the tile below is empty
```

```
                    moves[(x,y)].add((x, y+1))
        return moves

    def BFS_SP(self, color, moves):
        """
        Breadth First Search: Shortest path from start to goal.

        :param color: Color of piece
        :param moves: Graph (default dict) of possible moves
        :return: Shortest path or infinity if no path
        """
        start = self.pieces[color==BLACK].pos  # start is the position of the pawn of color
Color
        goal = TOP_ROW if color==WHITE else BOTTOM_ROW
        seen = set()  # set of all edges (keys) already checked
        queue = deque()
        queue.append([start])  # queue is a queue of lists of the shortest paths
        if start in goal:
            return [start]
        while queue:
            path = queue.popleft()  # one path is removed from the head of the queue
            node = path[-1]  # node = last edge in path
            if node not in seen:  # if node hasn't been explored yet, if it was explored,
this couldn't be shortest path
                neighbors = moves[node]  # neighbors -> list of all edges that node can reach
                for neighbor in neighbors:
                    new_path = [*path, neighbor]  # queue will append a new path for each of
the neighbors of node
                    queue.append(new_path)
                    if neighbor in goal:
                        return new_path  # if the end has been reached, this is the shortest
path
                seen.add(node)  # check given node as seen
        return float("Inf")  # If the queue was completely checked and emptied, there's no
path

    def DFS(self, color, moves):
        """
        Depth first search - Find if there is a path from piece of color color to the goal

        :param color: Color of piece being checked
        :param moves: Graph of moves
        :return: True if there is a path, False if there isn't
        """
        start = self.pieces[color == BLACK].pos  # start is the position of the piece of
color 'color'
        goal = TOP_ROW if color == WHITE else BOTTOM_ROW  # if piece is black, they need to
get to bottom, vice versa
        seen, stack = set(), [start]  # seen is set of visited positions, stack is the stack
used to check
        while stack:  # while the stack isn't empty
            node = stack.pop()  # node is the last element of the stack, a pos on the board
            seen.add(node)  # add current node to all nodes that have been checked
            for neighbor in moves[node]:  # for every move that can be done at pos node
                if neighbor in goal:  # if neighbor is in the goal row
                    return True
                if neighbor not in seen:  # if neighbor hasn't been checked yet
                    stack.append(neighbor)  # add neighbor to the stack to be checked
        return False  # the stack is empty and all possibilities were checked.
```

```
    def winner(self):
        """
        :return: WHITE if any of the tiles on top are occupied by white, BLACK if any of the
tiles on bottom are
        occupied by black, None if neither.
        """
        if self.pieces[0].pos[1] == 0:  # if white piece is in top row
            return WHITE
        if self.pieces[1].pos[1] == ROWS-1:  # if black piece is in bottom row
            return BLACK
```

game.py

```
from .board import Board
from .pieces import *


class Game:
    """
    Game class
    """
    def __init__(self, init=True):
        if init:
            self.init()
        else:
            self.started = False

    def init(self):
        """
        Start game (or reset)
        """
        self.started = True
        self.turns = 0
        self.checked_for_winner = False
        self.selected = None  # No tile is selected
        self.wall_selected = {'sel':False, 'dir':1}  # No wall is lifted
        self.turn = WHITE  # First turn is WHITE
        self.board = Board()  # Create board
        self.winner = lambda: self.board.winner()  # Returns winner (None if no one is
winning)
        self.valid_moves = set()  # Set of valid moves for selected piece (currently empty
because no piece is selected)
        self.walls_remaining = [WALLS,WALLS]  # First is player 1, second is player 2. Walls
left for each player

    def clone(self):
        """
        Creates a copy of the game

        :return: Copy
        """
        new_game = Game()
        new_game.turn = self.turn
        new_game.board = self.board.clone()
        new_game.walls_remaining = [*self.walls_remaining]
        return new_game

    def place(self, pos):
        """
        Place a wall at position pos on board

        :param pos: Position on board
        :return: True if wall is placed
        """
        x = self.turn == BLACK  # x=0/False if turn is white and 1/True if turn is black
        if self.board.can_place((pos, self.wall_selected['dir'])):  # If the placement isn't
intercepting another wall
            self.board.place_wall((pos,self.wall_selected['dir']))  # Place the wall in pos
            self.walls_remaining[x] -= 1
            self.wall_selected['dir'] = 1
```

```
            self.next_turn()
            return True  # Wall was successfully placed
        return False

    def place_ai(self, pos, dir):
        x = self.turn == BLACK
        self.board.place_wall((pos, dir))
        self.walls_remaining[x] -= 1
        self.next_turn()

    def flip(self):
        """
        Flip selected wall. Does nothing if no wall is selected
        """
        if self.wall_selected['sel']:
            self.wall_selected['dir'] = 1 - self.wall_selected['dir']

    def select(self, pos):
        """
        Select tile in pos

        :param pos: Given x,y position of selected section
        :return: True if worked, False if not
        """
        board_pos = (pos[0] // TILE_WIDTH, (pos[1] - MARGIN) // TILE_HEIGHT)
        if self.wall_selected['sel']:  # If a wall is being placed by a human player
            board_pos = (round(pos[0]/TILE_WIDTH), round((pos[1] - MARGIN) / TILE_HEIGHT))  #
Closest position to mouse
            placed = self.place(board_pos)
            if not placed:  # If failed to place a wall
                self.wall_selected['sel'] = False
                print('Illegal Move')
                return 'illegal'
            return True
        elif self.selected:  # If a piece is currently selected
            result = self.move(board_pos)  # Try to move the currently selected piece to the
pos
            if not result:  # If unable to move the piece
                self.selected = None  # Unselect piece
                self.valid_moves = set()  # No piece selected so no possible moves
                self.select(pos)  # Select the new tile that was clicked.
            return result

        if board_pos[1] > ROWS-1 or board_pos[0] > ROWS-1:
            return False  # If selected beyond range, return False
        piece = self.board.get_piece(board_pos)  # Piece at tile that was selected (if no
piece, piece=0)

        if piece != 0 and piece.color == self.turn:
            self.selected = piece  # Next time the board is clicked, the select function will
run on this piece
            self.valid_moves = self.board.possible_moves()[(piece.pos[0], piece.pos[1])]  #
Update valid moves
            return True  # The piece has been successfully selected
        return False  # No piece has been selected

    def move(self, pos):
        """
        Move piece
```

```
        :param pos: Position which the selected piece (self.selected) will be moving to
        :return: True if piece was able to move according to rules, false otherwise
        """
        if pos[1]>ROWS-1 or pos[0]>ROWS-1:  # If pos is above or below the board
            return False  # Can't move the piece above the board or under, return false

        piece = self.board.get_piece(pos)  # Needs to be 0, if not then the piece cannot move
to the given place
        if self.selected and piece == 0 and pos in self.valid_moves:
            self.board.move(self.selected, pos)  # Move the selected piece to the pos if it's
a valid move
            self.next_turn()
        else:
            return False  # Unable to move piece, return False
        return True  # Piece successfully moved

    def next_turn(self):
        """
        Reset the selections and change the turn
        """
        self.valid_moves = set()  # There are no valid moves because no pawn has been
selected
        self.wall_selected['sel'] = False  # Unselect any walls when changing turns
        self.turn=BLACK if self.turn==WHITE else WHITE
        self.turns += 1
        self.checked_for_winner = False

    def draw_moves(self, win):
        """
        Draw all possible moves for selected pawn.

        :param win: Game window
        """
        for move in self.valid_moves:
            row, col = move
            pygame.gfxdraw.aacircle(win, row*TILE_WIDTH + TILE_WIDTH // 2,
                                    col*TILE_HEIGHT + TILE_HEIGHT // 2 + MARGIN, MOVE_RADIUS,
GRAY)

    def walls_left(self, win, color=None):
        """
        Writes in margins how many walls are left for each player

        :param win: Game window
        :param color: If the game is multiplayer, color is the color of the client
        """
        for i in range(2):
            n = FONT.render(f'{self.walls_remaining[i]} walls left.', True, BLACK)
            w, h = n.get_size()
            win.blit(n, ((WIDTH - w) // 2, (MARGIN + BOARD_HEIGHT) * (1 - i) + (MARGIN - h)
// 2))  # writing on center
        if color:
            if self.turn == BLACK and color == 'B' or self.turn == WHITE and color == 'W':
                n = SMALL_FONT.render("Your turn.", True, BLACK)
            else:
                n=SMALL_FONT.render("Other player's turn", True, BLACK)
            w, h = n.get_size()
            win.blit(n, ((WIDTH - w) // 2, (MARGIN + BOARD_HEIGHT) * (1 - (color == 'B')) +
(MARGIN - h) // 2 + 30))
        else:
```

```
            n = SMALL_FONT.render("Your turn.", True, BLACK)
            w, h = n.get_size()
            win.blit(n, ((WIDTH - w) // 2, (MARGIN + BOARD_HEIGHT) * (1 -(self.turn ==
BLACK)) +(MARGIN - h) // 2 + 30))

    def lift_wall(self):
        """
        Lift a wall. The wall that will be lifted is the first wall in the player's list that
hasn't been placed yet.
        This function will only be used for human players because the computer can
automatically place a wall without
        having to lift it first

        :return: Whether wall is lifted.
        """
        if self.selected:
            self.select((ROWS,ROWS))  # If a piece was chosen, unselect the piece and select
a wall instead.
        turn = self.turn == BLACK
        if self.walls_remaining[turn] == 0:
            return False  # If player 1 has no walls left, they can't lift another wall
        self.wall_selected['sel'] = not self.wall_selected['sel']  # if wall is lifted,
unlift. else, lift.
        return True  # function successfully completed

    def update(self, win, pos=None, color=None):
        """
        Every frame, the update function will run. This takes care of the graphics so that
they truly remain correct
        throughout each frame

        :param win: Window
        :param pos: Mouse position
        :param color: In the case of an online game, the color of the client.
        """
        self.board.draw(win)  # This will draw the tiles, walls and pawns
        self.draw_moves(win)  # This will draw the possible moves as long as there is a
selected piece
        self.walls_left(win, color)  # This updates in the margins that they will have the
correct amount written
        if self.wall_selected['sel']:  # If wall is being lifted, constantly make the wall
follow the position of mouse
            if pos is not None:
                pygame.draw.line(win, RED, (pos[0]-1, pos[1]-1), (pos[0]-1+WALL_HEIGHT*(1-
self.wall_selected['dir']),
                                                                  pos[1]-
1+WALL_HEIGHT*(self.wall_selected['dir'])),
                                 WALL_WIDTH+1)
        pygame.display.update()

    def evaluate(self):
        """
        Used for ai

        :return: The value of the current position (how good it is for the white player).
White will want to
        maximize this value while black will want to minimize it (minimax). Value composed of
distance from
        """
        possible = self.board.possible_moves()
```

```
        try:
            return len(self.board.BFS_SP(BLACK, possible))-
len(self.board.BFS_SP(WHITE,possible)) + \
                    (self.walls_remaining[0]-self.walls_remaining[1])*0.1
        except TypeError:
            return float('inf')

    def unlift(self):
        """
        Stops lifting wall (sel = false)
        """
        self.wall_selected['sel'] = False

    def make_horizontal(self):
        """
        Makes wall horizontal (dir = 0)
        """
        self.wall_selected['dir'] = 0

    def __repr__(self):
        return f"Game with board {self.board}. {'White' if self.turn==WHITE else 'Black'}
turn."
```

client.py

```
import socket


class Player_Client:

    HOST = 'localhost'
    PORT = 49550

    def __init__(self, game):
        self.main = game  # self.main.client = self
        self.socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)

    def close(self):
        self.socket.close()

    def con(self):  # connect to server
        self.socket.connect((Player_Client.HOST, Player_Client.PORT))
# connect to host, server
        self.color = self.socket.recv(8).decode('UTF-8')  # color of
player received by server
        self.socket.recv(1024)  # wait for signal that the game
started
        self.main.connected = True  # inform main that connection has
been established

    def request(self):  # receive info from server through socket and
append to main's queue
        self.main.que.append(self.socket.recv(1000).decode('UTF-8'))

    def send(self, info):  # send info to server through socket
        self.socket.send(info.encode('UTF-8'))
```

server.py

```
import socket
import time

class PlayerOneLeft(Exception):  # special exception when player one leaves the game
    pass

class Server:

    HOST = 'localhost'
    PORT = 49550

    def __init__(self):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)  # ipv4, tcp
        self.socket.bind((Server.HOST, Server.PORT))  # binding socket to address and port
        self.conn1 = self.conn2 = self.address1 = self.address2 = None
        self.connect()

    def connect(self):
        self.socket.listen()  # wait for socket to connect
        self.conn1, self.address1 = self.socket.accept()  # conn1 and address1 are the info
of the first socket
        self.conn1.sendall(b'W')  # give the first client color W
        self.socket.listen()  # wait for second client to connect
        self.conn2, self.address2 = self.socket.accept()  # conn2 and address2 are the info
of the second socket
        try:
            x = self.conn1.recv(1000).decode('UTF-8')  # check if the first player is still
connected
        except ConnectionResetError:
            self.conn2.send(b'Q')  # tell second client that the first player left
            raise PlayerOneLeft()
        if 'Q' in x:
            raise PlayerOneLeft()
        self.conn2.sendall(b'B')  # if everything worked, second client is black
        self.conn1.sendall(b'2')  # telling player 1 that the game started
        time.sleep(0.1)  # wait so that the messages won't be sent as one
        self.conn2.sendall(b'2')  # telling player 2 that the game started
        self.send()

    def send(self):  # transfer information between the clients
        while True:
            try:
                self.conn2.sendall(self.conn1.recv(1000))
                self.conn1.sendall(self.conn2.recv(1000))
            except (ConnectionResetError, ConnectionAbortedError):  # if one of the players
leaves, stop loop and close
                self.socket.close()
                return

if __name__ == "__main__":
    print('Server Initiated.')
    try:
        server = Server()
    except PlayerOneLeft:
        print('Server shutting down because a player left before the game began. Please start
over.')
    print('Server Closed.')
```

algorithm.py

```
from quoridor.constants import *
import random


class AI:

    ALL_WALLS_HORIZONTAL = *((i,j) for i in range(ROWS-1) for j in range(1,ROWS)),  # tuple
of all horizontal placements
    ALL_WALLS_VERTICAL = *((i,j) for i in range(1,ROWS) for j in range(ROWS-1)),  # tuple of
all vertical placements

    def __init__(self, typ=0):
        if typ not in range(7):  # if the user enters a number other than the range 0-6
            raise ValueError
        self.type = typ
        if typ == 1:
            print('You have selected Minimax, the best performing AI. Take in mind it may
take several seconds to make'
                  'a decision')

    def do(self, game):
        ais = [0, self.pick_move, self.greediest_ai, self.random_ai, self.hesitant_ai,
self.greedy_ai, self.passive_ai]
        if self.type != 0:
            game = ais[self.type](game)
        else:
            self.type = random.randint(2,6)
            self.do(game)
            self.type = 0
        return game

    @staticmethod
    def place_wall_above(game):
        """
        Places wall over the white player
        :param game: Game
        :return: True if wall was placed, False otherwise
        """
        white_piece = game.board.pieces[0].pos
        if game.lift_wall():
            game.make_horizontal()
            if game.place(white_piece):  # try placing right over piece
                return True
            game.lift_wall()
            if game.place((white_piece[0] - 1, white_piece[1])):  # try placing to the left
over piece
                return True
            game.unlift()
        return False

    @staticmethod
    def go_shortest_path(game):
        """
        Makes AI take the shortest path based on the BFS
        :param game:
        :return:
        """
```

```
        piece = game.board.pieces[1].pos  # position of black piece
        game.select((piece[0] * TILE_WIDTH, piece[1] * TILE_HEIGHT + MARGIN))  # select black
piece
        possible_moves = game.board.possible_moves()
        best_path = game.board.BFS_SP(BLACK, possible_moves)
        chosen = best_path[1]
        game.select((chosen[0] * TILE_WIDTH, chosen[1] * TILE_HEIGHT + MARGIN))

    @staticmethod
    def minimax(game, depth, maximizing_player):
        """
        Minimax Algorithm

        :param game: Game that is being checked
        :param depth: How many recursions have been done
        :param maximizing_player: True if white, False if white
        :return: Best minimax value, game after the best minimax value has been done
        """
        x = game.winner()
        if x is not None:
            print('f')
            return float('inf') if x == WHITE else float('-inf'), game
        if depth == 2:  # If depth has reached 2 or if the game is over
            return game.evaluate(), game
        best = float('-inf') if maximizing_player else float('inf')  # min value if
maximizing, max if minimizing
        best_move = None
        piece = game.board.pieces[game.turn == BLACK].pos  # position of piece that is moving
        game.select((piece[0] * TILE_WIDTH, piece[1] * TILE_HEIGHT + MARGIN))  # select piece
to get valid moves
        for move in game.valid_moves:  # For every move in the valid moves
            new_game = game.clone()  # Create copy of game
            piece = new_game.board.pieces[new_game.turn == BLACK].pos  # position of piece
that is moving
            new_game.select((piece[0] * TILE_WIDTH, piece[1] * TILE_HEIGHT + MARGIN))
            new_game.move(move)  # move piece to current valid move
            # noinspection PyUnresolvedReferences
            value = AI.minimax(new_game, depth + 1, not maximizing_player)[0]  # recursion,
does minimax
            if maximizing_player:  # minimax
                best = max(best, value)
            else:
                best = min(best, value)
            if best == value:
                best_move = new_game  # if the current value is the best value, the best move
is the move that was done
        if game.walls_remaining[not maximizing_player]:  # if there are any walls left for
the current player
            for wall in AI.ALL_WALLS_HORIZONTAL:  # for each possible wall
                if game.board.can_place((wall, 0)):
                    new_game = game.clone()
                    new_game.place_ai(wall, 0)
                    value = AI.minimax(new_game, depth + 1, not maximizing_player)[0]
                else:
                    continue
                if maximizing_player:
                    best = max(best, value)
                else:
                    best = min(best, value)
                if best == value:
```

```
                best_move = new_game
        if game.turns > 10:
            for wall in AI.ALL_WALLS_VERTICAL:
                new_game = game.clone()
                if new_game.board.can_place((wall, 1)):
                    new_game.place_ai(wall, 1)
                    value = AI.minimax(new_game, depth + 1, not maximizing_player)[0]
                else:
                    continue
                if maximizing_player:
                    best = max(best, value)
                else:
                    best = min(best, value)
                if best == value:
                    best_move = new_game
    return best, best_move

@staticmethod
def greediest_ai(game):
    if AI.place_wall_above(game):
        return game
    AI.go_shortest_path(game)
    return game

@staticmethod
def random_ai(game):
    x = random.randint(0,1)
    if x == 1:
        if AI.place_wall_above(game):
            return game
    AI.go_shortest_path(game)
    return game

@staticmethod
def hesitant_ai(game):
    x = random.randint(1,4)
    if x == 4:
        if AI.place_wall_above(game):
            return game
    AI.go_shortest_path(game)
    return game

@staticmethod
def greedy_ai(game):
    x = random.randint(1,3)
    if x != 3:
        if AI.place_wall_above(game):
            return game
    AI.go_shortest_path(game)
    return game

@staticmethod
def passive_ai(game):
    AI.go_shortest_path(game)
    return game

@staticmethod
@timeit
def pick_move(game):
    return AI.minimax(game, 0, False)[1]
```

main.py

```
import pygame
import pygame.gfxdraw
import collections
import sys
from quoridor.constants import *
from quoridor.game import Game
from ai.algorithm import AI
from network.client import Player_Client
from threading import Thread
import time


class Main:

    SCREENS = [pygame.image.load('Quoridor-Pick Mode.png'), pygame.image.load('Quoridor-
Local.png'),
               pygame.image.load('Quoridor-Multiplayer wait.png'),
pygame.image.load('Quoridor-AI.png'),
               pygame.image.load('Quoridor-Rules.png'), pygame.image.load('White wins.png'),
               pygame.image.load('Black wins.png')]  # list of all images shown throughout
game.

    def __init__(self, typ):
        pygame.init()
        self.WIN = pygame.display.set_mode((WIDTH, HEIGHT))  # creates pygame display as main
window
        pygame.display.set_caption('Quoridor')
        self.game = Game(False)  # self.game is a game which isn't initiated yet
        self.type = typ  # type of game. 1: local, 2: online 3: vs ai
        self.ai = AI()  # AI engine, set as default value
        self.game_stack = []  # stack of all games
        self.winners = {BLACK:self.black_wins, WHITE:self.white_wins,'W':self.black_wins,
'B':self.white_wins}
        # key to function to save pointless if statements. 'W' calls to black wins and vice
versa because it will be
        # called when that color forfeits.

    def black_wins(self):
        """
        Display screen "black wins"
        """
        time.sleep(0.2)
        print('Black wins.')
        self.WIN.blit(self.SCREENS[6], (0, 0))
        pygame.display.update()
        time.sleep(1)

    def white_wins(self):
        """
        Display screen "white wins"
        :return:
        """
        time.sleep(0.2)
        print('White wins.')
        self.WIN.blit(self.SCREENS[5], (0,0))
        pygame.display.update()
        time.sleep(1)
```

```
    def ai_move(self):
        """
        Do move for AI
        """
        temp = self.game.turns  # to save the amount of turns
        self.game = self.ai.do(self.game)
        self.game.turns = temp + 1  # correct amount of turns
        self.game.select((0, 0))  # deselect piece

    def wait(self):
        screen = self.SCREENS[0]  # opening screen
        self.WIN.blit(screen, (0, 0))
        while self.type == 0:  # while no type has been selected
            pygame.display.update()
            for event in pygame.event.get():
                if event.type == pygame.QUIT:
                    self.type = 4  # close screen
                if event.type == pygame.KEYUP:
                    if event.key == pygame.K_1:  # local multiplayer
                        self.type = 1
                    if event.key == pygame.K_2:  # online multiplayer
                        self.type = 2
                    if event.key == pygame.K_3:  # against ai
                        self.type = 3
                    if event.key == pygame.K_SPACE:  # show rules
                        self.WIN.blit(self.SCREENS[4], (0, 0))
                    if event.key == pygame.K_ESCAPE:  # close rules
                        self.WIN.blit(self.SCREENS[0], (0, 0))
                if event.type == pygame.MOUSEBUTTONUP:  # close rules
                    self.WIN.blit(self.SCREENS[0], (0, 0))

    @timeit  # print time game was running in the end
    def main(self):
        """
        Game
        """
        self.wait()  # while the player hasn't selected the mode, show loading screen.
        if self.type == 4:  # if player closed the game in self.wait
            return
        if self.type == 2:
            self.multi()
            return
        run = True
        undo_clicked = [1,1,True] if self.type == 1 else [1,0,True]
        """each human player can undo once throughout game. When undo_clicked[2] = True, undo
can't be clicked
         (first turn or if ctrl z is already pressed)"""
        while run:
            screen = self.SCREENS[self.type]  # 1 if local multiplayer, 3 if AI
            if not self.game.started:  # while game hasn't been initiated
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:  # close screen
                        run = False
                    if event.type == pygame.MOUSEBUTTONUP and self.type == 1:  # start local
game when screen is clicked
                        self.game.init()
                        self.game_stack.append(Game())  # add initial game to stack for undo
                    if self.type == 3:
                        if event.type == pygame.KEYUP:  # set up AI based on number clicked
```

```
                                typ = pygame.key.name(event.key)
                                try:
                                    self.ai = AI(int(typ))
                                except ValueError:  # a string or incorrect number was entered
                                    self.ai = AI()
                                self.game.init()
                                self.game_stack.append(Game())  # the first item in the game
stack is the initial game
                    self.WIN.blit(screen, (0,0))
                    pygame.display.update()
                else:
                    if self.game.turns > 13 and not self.game.checked_for_winner:  # check for
winner only when turn starts
                        win = self.game.winner()
                        if win:  # and only if its technically possible for there to be a winner
(14 turns)
                            self.winners[win]()
                            break
                        self.game.checked_for_winner = True
                    if self.type == 1 or self.game.turn==WHITE:  # if a human player is playing
                        for event in pygame.event.get():
                            if event.type == pygame.QUIT:  # if the game was closed
                                self.winners[tuple(255-i for i in self.game.turn)]()  # display
player who didn't quit
                                run = False

                            if event.type == pygame.MOUSEBUTTONUP:  # select tile
                                pos = pygame.mouse.get_pos()
                                temp = self.game.turns
                                self.game.select(pos)
                                if self.game.turns > temp and any(undo_clicked[0:2]):  # if the
turn has changed
                                    self.game_stack.append(self.game.clone())  # and undo hasn't
been used by both players
                                    self.game_stack[-1].turns = self.game.turns
                                    undo_clicked[2] = False

                            if event.type == pygame.KEYUP:
                                if event.key == pygame.K_SPACE:  # lift wall
                                    [self.game.lift_wall,
self.game.unlift][self.game.wall_selected['sel']]()  # lift if not
                                    # lifted, don't lift if lifted
                                if event.key == pygame.K_f:  # flip wall
                                    self.game.flip()
                                if event.key == pygame.K_z and self.game.turns > 0:
                                    undo_clicked[2] = False
                        keys = pygame.key.get_pressed()
                        if keys[pygame.K_LCTRL] and keys[pygame.K_z] and \
                                undo_clicked[self.game.turn==BLACK] and not undo_clicked[2]:
                            self.game_stack.pop()
                            self.game = self.game_stack[-1]
                            undo_clicked[self.game.turn==WHITE] = 0
                            print(f'{"White" if self.game.turn == WHITE or self.type == 3 else
"Black"} undid turn.')
                            undo_clicked[2] = True
                            if self.type == 3:
                                self.game_stack.pop()
                                self.game = self.game_stack[-1]

                    elif run and self.type == 3:  # it's ai's turn
```

```
                    self.ai_move()
                    if undo_clicked[0]:
                        self.game_stack.append(self.game.clone())
                        self.game_stack[-1].turns = self.game.turns

                if run:
                    self.game.update(self.WIN, pygame.mouse.get_pos())  # always update the
screen

    def connect(self):
        """
        Create client object with main object and connect to server. If connection fails,
catch exception.
        """
        self.client = Player_Client(self)  # self.client -> client object. client.main ->
self
        try:
            self.client.con()  # connect to server
        except (ConnectionAbortedError, ConnectionRefusedError, TimeoutError):
            return  # if unable to connect to server, do nothing.

    def client_listen(self):
        """
        Function that will always be running in the second thread, fills self.que with TCP
values.
        """
        while self.playing:
            try:
                self.client.request()
            except ConnectionResetError:  # the connection has been closed - TCP RST received
                self.playing = False
                self.client.close()
            except ConnectionAbortedError:  # the connection has been forced to close due to
exception
                return

    def multi(self):
        """
        Multiplayer Game.
        """
        self.que = collections.deque()  # queue that will save all of the received messages
        self.connected = False  # will be true when the other thread will connect to the
server
        self.playing = True  # will be false when the game is over / aborted. used to
communicate between the threads
        Thread(target=self.connect).start()  # thread to connect to server while screen loads
        self.WIN.blit(self.SCREENS[2], (0,0))
        time.sleep(0.5)  # wait for other thread to finish
        try:
            self.client.send('hello!')  # try to send message to server
        except OSError:  # raised if there's no server
            print("Server hasn't been opened yet.")
            return
        while not self.connected and self.playing:
            try:
                self.client.color  # see if the server sent a color or if it's currently
running a different game.
            except AttributeError:  # self.client doesnt have the attribute color because
it's stuck on recv(8) in con
                print('A game is currently taking place. Please wait until the current game
```

```
ends.')
                self.client.close()
                return
            for event in pygame.event.get():
                if event.type == pygame.QUIT:  # X has been clicked while waiting for other
player to connect
                    self.playing = False
                    self.client.send('Q')
                    self.client.close()
            pygame.display.update()
        self.game.init()  # start game
        run = True  # for game loop
        Thread(target=self.client_listen).start()  # start constant listening
        turns = {BLACK:'B', WHITE:'W'}  # (0,0,0):'B', (255,255,255):'W"
        if self.client.color == 'Q':  # if the other player left before the game started
            print('The white player has left the game.')
            self.black_wins()
            return
        print(f"You are {'Black' if self.client.color=='B' else 'White'}.")  # prints on
screen player's color
        wall_selected_multi = False  # boolean to draw wall on board while the wall is lifted
        while run:
            if not self.playing:  # if the other thread stopped
                return
            if self.game.turns > 13 and not self.game.checked_for_winner:  # if it's possible
for there to be a winner
                win = self.game.winner()
                if win:
                    self.winners[win]()
                    self.playing = False
                    self.client.close()
                    break
                self.game.checked_for_winner = True  # there's no winner. wait until next
move and don't check again
            if turns[self.game.turn] == self.client.color:  # if it's the player's turn
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:  # if X was clicked
                        print('You forfeit.')
                        self.client.send('Q')  # let other client know that game is over
                        self.winners[self.client.color]()
                        self.client.close()
                        return
                    if event.type == pygame.MOUSEBUTTONUP:  # screen was clicked
                        ms = pygame.mouse.get_pos()
                        legal = self.game.select(ms)
                        if legal != 'illegal':  # if the move was legal
                            st = f"S{ms[0]:3d},{ms[1]:3d}"  # send coordinates to other
player, always length 8
                            wall_selected_multi = False  # for updating screen
                            self.client.send(st)
                        else:
                            self.client.send('L')
                    if event.type == pygame.KEYUP:
                        if event.key == pygame.K_SPACE:
                            self.game.lift_wall()
                            wall_selected_multi = True
                            self.client.send('L')
                        if event.key == pygame.K_f:
                            self.game.flip()
                            self.client.send('F')
```

```
            else:
                for event in pygame.event.get():
                    if event.type == pygame.QUIT:
                        print('You forfeit.')
                        self.client.send('Q')
                        self.winners[self.client.color]()
                        self.client.close()
                        return

            self.game.update(self.WIN, pygame.mouse.get_pos() if wall_selected_multi else
None, self.client.color)
            self.client.send('0')  # send stream of data at all times
            if len(self.que) > 0:  # if the request queue has data
                received = self.que.popleft()  # get earliest message
                place = received.find('S')  # if message contains S, returns position of s,
else return -1
                if place != -1:  # if message contains S
                    self.game.select((int(received[place+1:place+4]),
int(received[place+5:place+8])))  # select at pos
                if 'L' in received:  # if other player lifted wall
                    self.game.lift_wall()
                if 'F' in received:  # if other player flipped wall
                    self.game.flip()
                if 'Q' in received:
                    self.playing = False
                    print('The other player has left the game.')
                    self.winners[{'B':BLACK, 'W':WHITE}[self.client.color]]()
                    return
        return 0


if __name__ == '__main__':
    try:
        x = int(sys.argv[1])
    except (IndexError, ValueError):
        x = 0
    create = Main(x)
    create.main()
```

דברים שנמחקו:

מחלקת Wall

```
class Wall:
    def __init__(self, dir, color):
        """Creates wall

        :param dir: Direction Horizontal-0, Vertical-1
        :param color: Color"""

        self.color = color
        self.pos = ((0,0),(0,2))
        self.dir = dir
        self.x1 = self.x2 = self.y1 = self.y2 = 0
        self.placed = False  # Will be true when the wall is placed
on the board and can no longer be moved.
        self.lifted = False  # Will be true when the wall is selected
(when the player presses space)
        self.find_pos((0, 0))  # Will find the pos of the second
point of the wall.

    def clone(self):
        a = Wall(self.dir, self.color)
        a.placed = self.placed
        a.lifted = self.lifted
        a.pos = self.pos
        a.calc_pos()
        return a

    def find_pos(self, first_pos):
        """Updates self.pos to be list of the two points on the edges
of wall

        :param first_pos: Pos of top point (if dir==1) or left point
(if dir==0)
        :return:"""
        if self.placed:
            if self.dir == 0:  # self.pos will be [(left row, left
col), (right row, right col)] (left row == right row)
                self.pos = (first_pos, (first_pos[0]+2,
first_pos[1]))
            else:  # self.pos will be [(top row, top col), (bottom
row, bottom col)] (top col == bottom col)
                self.pos = (first_pos, (first_pos[0],
first_pos[1]+2))
        else:
            if self.dir == 0:  # self.pos will be [(left x, left y),
(right x, right y)] (left x == right x)
                self.pos = (first_pos, (first_pos[0]+WALL_HEIGHT,
first_pos[1]))
            else:  # self.pos will be [(top x, top y), (bottom x,
bottom y)] (top x == bottom x)
                self.pos = (first_pos, (first_pos[0],
first_pos[1]+WALL_HEIGHT))
        self.calc_pos()

    def calc_pos(self):
```

```
        if self.placed:  # if wall is placed x1,x2,y1,y2 will be the
real x,y values of pos
            self.x1 = self.pos[0][0] * TILE_WIDTH
            self.x2 = self.pos[1][0] * TILE_WIDTH
            self.y1 = self.pos[0][1] * TILE_HEIGHT + MARGIN
            self.y2 = self.pos[1][1] * TILE_HEIGHT + MARGIN
        else:  # if wall isn't placed, x1,x2,y1,y2 will be equal to
pos
            self.x1 = self.pos[0][0]
            self.x2 = self.pos[1][0]
            self.y1 = self.pos[0][1]
            self.y2 = self.pos[1][1]

    def flip(self):
        """Flip wall (change direction)"""
        if not self.placed:
            self.dir = 1 - self.dir

    def lift(self, pos):
        """Lift wall before placing"""
        self.lifted = True
        self.find_pos(pos)  # find pos based on where it's lifted at
(mouse pos)

    def place(self, pos):
        #Place wall
        self.lifted = False
        self.placed = True
        self.find_pos(pos)

    def unplace(self):
        # Remove wall
        self.placed = False
        self.find_pos(self.pos[0])  # reset pos

    def __str__(self):
        return f"Wall at {self.pos[0]}, {self.pos[1]}"

    def __repr__(self):
        return str(self)
```

פעולה win_possible

```
def win_possible(self, pos, path, color):
    # Recursive function that checks if it's possible foa piece of
color 'color' to win from its current pos 'pos'
    # :param pos: Position on board that is being checked
    # :param path: List of all tiles that were already checked (so
that it doesn't keep checking the same tiles)
    # :param color: Color of piece that is being checked (white needs
to get to top, black needs to get to bottom)
    if pos in path:
        return False
    path.append(pos)
    if (color == WHITE and pos in TOP_ROW) or (color==BLACK and pos
in BOTTOM_ROW):
        return True
    else:
        return any([self.win_possible(p, path, color) for p in
```

```
self.possible[pos]])
        # Return true if it is r possible to win from any of the
tiles that the piece can move to
```

הפעולה wall_relevant

```
def wall_relevant(self, pos, dir, turn):
    x,y = pos
    px, py = self.board.pieces[turn == BLACK].pos  # Piece x, piece y
    if self.lift_wall() and
self.board.can_place_tech(self.wall_selected, pos):
        if dir: # Vertical
            if x==px and y==py or x-1==px and y==py or x==px and
y+1==py or x-1==px and y+1==py:  #Adjacent to piece
                return self.board.can_place(self.wall_selected, pos)
            if (y>0 and (self.board[x][y]['walls'][1] or
self.board[x][y-1]['walls'][0])) or
            (y+2<ROWS and self.board[x][y+2]['walls'][0]):
                return self.board.can_place(self.wall_selected, pos)
            try:
                if self.board[x][y+1]['walls'][1] or
self.board[x][y+1]['walls'][3] or
                self.board[x-1][y]['walls'][1] or self.board[x-
1][y+1]['walls'][1] or
                self.board[x-1][y+1]['walls'][3]:
                    return self.board.can_place(self.wall_selected,
pos)
            except IndexError:
                print(f"{x}, {y+1}, {self.board[x][y+1]['walls']}")
                return False
        else:
            if x==px and y==py or x+1==px and y==py or x==px and y-
1==py or x+1==px and y-1==py:
                return self.board.can_place(self.wall_selected, pos)
            if (x>0 and (self.board[x-1][y]['walls'][1]) or
self.board[x][y]['walls'][0]) or
            (x+2<ROWS and self.board[x+2][y]['walls'][1]):
                return self.board.can_place(self.wall_selected, pos)
            if self.board[x+1][y]['walls'][0] or
self.board[x+1][y]['walls'][2] or self.board[x][y-1]['walls'][0] or
             self.board[x+1][y-1]['walls'][0] or self.board[x+1][y-
1]['walls'][2]:
                return self.board.can_place(self.wall_selected, pos)
    return False
```

הפעולה relevant_possible_walls

```
def relevant_possible_walls(self):
    #This will be a heavy function because it will check for each
wall placement possible if the wall can be placed.
    #This function will be called at the beginning of the turn for
the ai, therefore no wall has been lifted yet.
    #:return: List of all possible wall positions. walls = []
```

```
if self.lift_wall():  # Lift outside of board
    for i in range(ROWS):  # Check all vertical walls
        for j in range(ROWS):
            if self.wall_relevant((i,j), 1, self.turn):
                walls.append(((i,j), 1))
            self.flip()
            if self.wall_relevant((i,j), 0, self.turn):
                walls.append(((i,j),0))
            self.flip()
    self.wall_selected['sel'] = False
return walls
```

[הפעולה gametree נמחקה.