|  |
| --- |
| מכללת עמל ב', פתח תקווה |
| פרוייקט גמר טכנאי הנדסת תכנה |
| נושא הפרוייקט: משחק דמקה סינית |

|  |
| --- |
| שם הסטודנט: הארי זלסקי  ת.ז.: 322521386  מרצה מנחה: ד"ר נילי נווה  מתרגל: מיקי שרוני  שנה: 2019 |

תוכן עניינים

**פרק 1 – נושא הפרוייקט ורקע2**

**פרק 2 – מבנה הנתונים5**

**פרק 3 – האלגוריתם הראשי7**

**פרק 4 – הקוד12**

**פרק 1 – נושא הפרוייקט ורקע**

**דמקה סינית**

דמקה סינית או הלמה כוכב הוא משחק לוח המיועד לשניים עד שישה שחקנים. המשחק הוא גרסה של המשחק הלמה שפותח בשלהי המאה ה-19, והמטרה בו היא להעביר את כלי המשחק, על פי רוב גולות או יתדות, לצד המנוגד של לוח המשחק.



תמונה 1 - לוח משחק דמקה סינית עם שש ערכות משחק

למרות שמו, המשחק דמקה סינית לא הומצא בסין או באסיה כלל. הוא אף אינו גרסה של המשחק דמקה. המשחק הומצא בגרמניה בשנת 1892 תחת השם "שטרן-הלמה" (הלמה כוכב), כגרסה למשחק הלמה האמריקאי. שמו הנוכחי ניתן לו בארצות הברית ב-1928, כאמצעי שיווקי של ביל וג'ק פרסמן, אך בטרם זאת נקרא דמקה הופ צ'ינג.

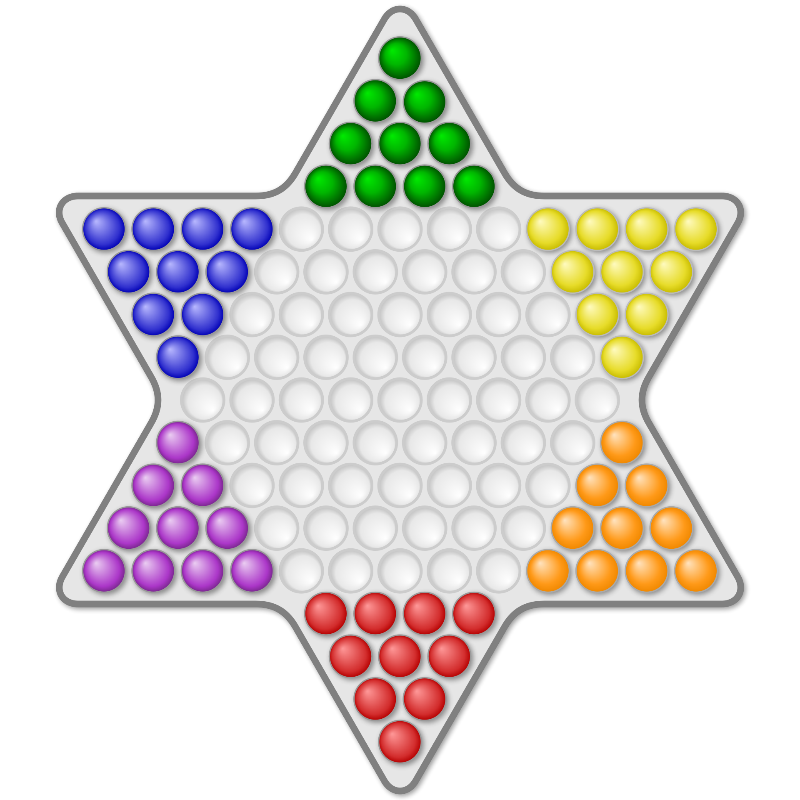
**חוקי המשחק**

**מטרת המשחק**

להעביר את כלי המשחק, על פי רוב גולות או יתדות, לצד המנוגד של לוח המשחק.

**מבנה הלוח**

לוח המשחק עשוי 121 תאים המסודרים בצורת מגן דוד, כאשר המשולשים הקטנים המרכיבים את פינותיו מכילים עשרה תאים כל אחד.



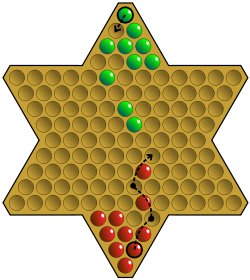
**הכנות למשחק**

בתחילת המשחק מציב כל שחקן עשרה כלי משחק מאותו צבע באחד מקדקודיו של הלוח.

כאשר משחקים שני שחקנים, כמו במקרה שלנו, מוצבים כלי המשחק של כל שחקן בבסיסים מנוגדים.

**מהלך המשחק**

כל שחקן בתורו מזיז כלי משחק אחד, או לתא סמוך פנוי או תוך כדי דילוג מעל כלי משחק אחד אחר, אין זה משנה של איזה שחקן, אל התא הפנוי הבא אחריו. אסור לדלג מעל שני כלי משחק צמודים, אך ניתן לבצע באותו מהלך מספר דילוגים, ובתנאי שבכל דילוג בשרשרת יונח כלי המשחק בתא פנוי. הדילוג יכול להתבצע בכל כיוון, וכלִי המשחק שמדלגים מעליו אינו מושפע מכך.



**המנצח**

הראשון שמצליח להעביר את כל כליו לבסיס הנגדי.

**פרק 2 – מבנה הנתונים**

לוח המשחק מיוצג על ידי גרף לא מכוון ולא משוקלל בן 121 צמתים (כמספר התאים בלוח). דרגתו המקסימלית של כל צומת היא 6, והמינימאלית היא 2, והוא מחובר בקשת לכל אחד משכניו.

public class GraphFacilities {

public static final int W=25,H=17;

public static CellVertex[][] vertexMat = new CellVertex[H][W];

public static Graph<CellVertex, CCEdge> g =

new SimpleGraph<CellVertex, CCEdge>( CCEdge.class );

}

כל אחד מהצמתים מכיל את הנתונים הבאים:

* שדה המכיל אחד משלושה ערכים אפשריים:
  + 0 – אין חייל בתא.
  + 1 – החייל בתא שייך לשחקן הממוחשב.
  + 2 – החייל בתא שייך לשחקן האנושי.
* שדה המכיל את המיקום של הקדקוד במטריצה.

public class CellVertex {

public int content;// The content of the vertex

private Point l;// The location of the vertex in the graph

}

כל אחת מהקשתות בגרף מכילה את הנתונים הבאים:

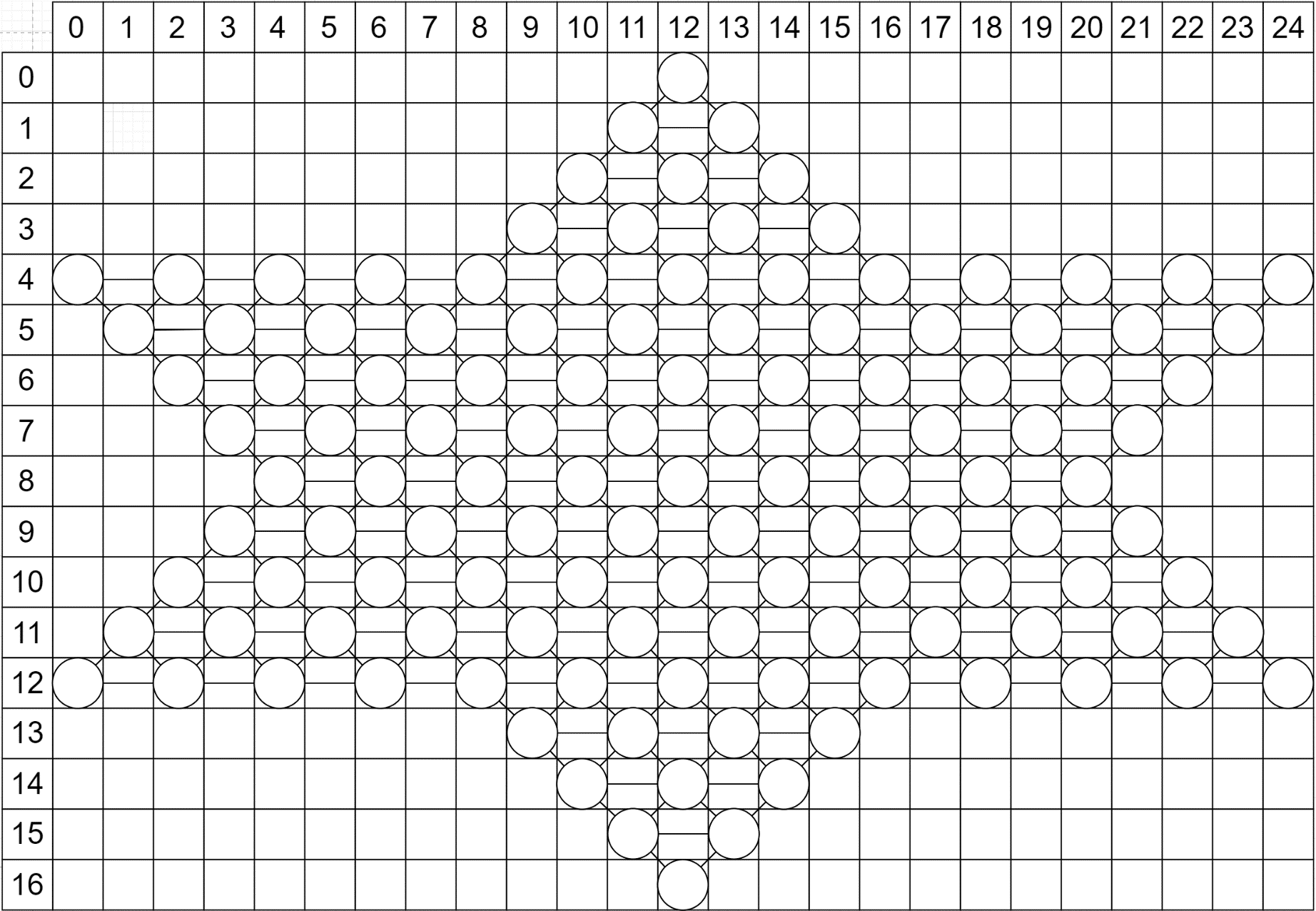
* שדה המכיל את קדקוד המקור
* שדה המכיל את קדקוד היעד

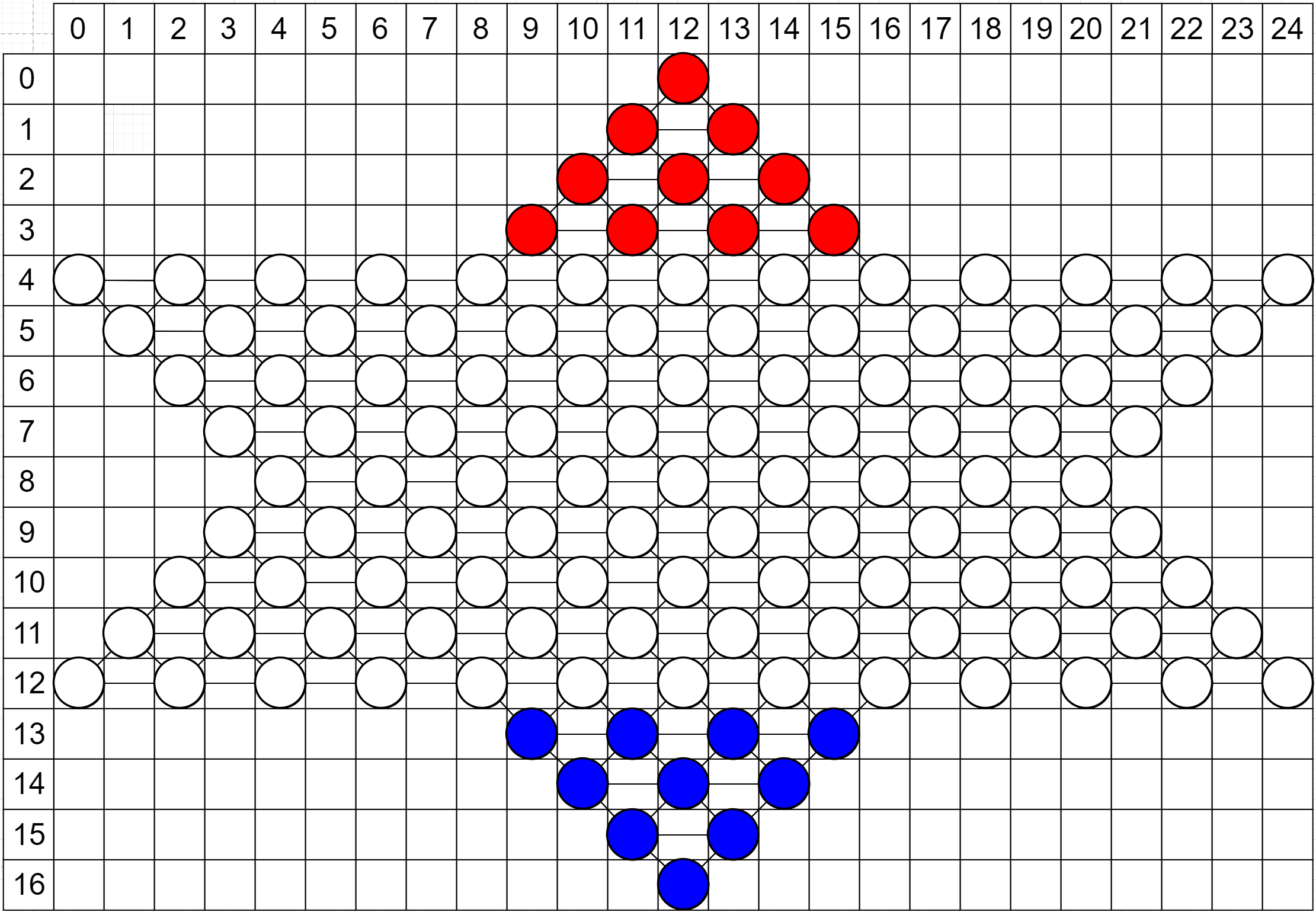
public class CCEdge {

private CellVertex srcVertx;//source vertex

private CellVertex destVertx;//destination vertex

}

****בנוסף, הצמתים ממוקמים במטריצה שבאמצעותה אנו מקבלים את המיקום של הקדקוד על לוח המשחק, ומדפיסים את הלוח על המסך:

****בתחילת המשחק, התכנית יוצרת מערך של כל הקשתות האפשריות

ומעדכנת את הגרף לפי הקשתות שנוצרו.

בכל תור, השחקן (האנושי והממוחשב) מקבלים את כל המקומות אליהם הם יכולים לעבור עם החייל שאותו בחרו.

מתוך המקומות האפשריים הם בוחרים אחד אליו הם עוברים. בסוף כל תור התכנית בודקת האם יש מנצח.

אם אין מריצה עדכון של הקשתות בגרף ומעבירה את התור לשחקן השני.

הסיבות לשימוש בגרף על פני מבנה נתונים אחר הם:

* ניתן למצוא את כל הקפיצות מקדקוד כלשהו ביעילות O(1)
* ניתן לחשב מסלול בזמן ריצה לינארי O(E LOG(V))

**פרק 3 – האלגוריתם הראשי**

**האסטרטגיות לכל תור של המחשב: AI**

1. חשב את כמות השחקנים שנשארו בחוץ (פונקציה\_1).
2. אם רק שחקן אחד נשאר בחוץ:
   1. עבור כל חייל במערך החיילים של המחשב:
      1. אם החייל מחוץ לבסיס:
         1. עבור כל תא בבסיס היעד:
            1. אם התא ריק:

הגדר את התא כנקודת הסיום של המסלול.

* + - 1. חשב את המסלול הקצר ביותר ממיקום החייל למיקום נקודת הסיום של המסלול.
      2. הגדר את נקודת ההתחלה של הצעד כמיקום הראשון במסלול (נקודת ההתחלה).
      3. הגדר את נקודת הסיום של הצעד כמיקום השני במסלול.
      4. בצע את הצעד (פונקציה\_9).
      5. עדכן את מיקום החייל למיקום החדש.
      6. סיים את התור (פונקציה\_7).

1. אם קיים חייל היכול להמשיך תנועת [[1]](#footnote-2)S (פונקציה\_2):
   1. מצא את מיקומו במערך החיילים של המחשב (פונקציה\_3).
   2. הגדר את מיקום תחילת התנועה למיקום החייל.
   3. הגדר את מיקום סוף התנועה למיקום סוף תנועת ה-S (אותה עמודה וארבע שורות למטה).
   4. בצע את התנועה (פונקציה\_9).
   5. עדכן את מיקום החייל למיקומו החדש.
   6. סיים את התור (פונקציה\_7).
2. אחרת:
   1. אם החייל החיצוני ביותר[[2]](#footnote-3) מצד שמאל נמצא במקומו המקורי:
      1. הגדר את מיקום תחילת התנועה למיקום החייל.
      2. הגדר את מיקום סוף התנועה למיקום (שורה אחת למטה, עמודה אחת ימינה).
      3. בצע את התנועה (פונקציה\_9).
      4. עדכן את מיקום החייל למיקומו החדש.
      5. סיים את התור (פונקציה\_7).
   2. אם החייל החיצוני ביותר מצד ימין נמצא במקומו המקורי:
      1. הגדר את מיקום תחילת התנועה למיקום החייל.
      2. הגדר את מיקום סוף התנועה למיקום (שורה אחת למטה, עמודה אחת שמאלה).
      3. בצע את התנועה (פונקציה\_9).
      4. עדכן את מיקום החייל למיקומו החדש.
      5. סיים את התור (פונקציה\_7).
   3. אחרת:
      1. מצא את הקפיצה הרחוקה ביותר (פונקציה\_4):
         1. אם הקפיצה קיימת:
            1. הגדר את תחילת התנועה למיקום החייל הקופץ.
            2. הגדר את סוף התנועה למיקום סיום הקפיצה.
            3. בצע את התנועה (פונקציה\_9).
            4. מצא את החייל שביצע את הקפיצה, ועדכן את מיקומו למיקום החדש.
            5. סיים את התור (פונקציה\_7).
         2. אחרת:
            1. עבור כל חייל בגרף מהחייל האחורי ביותר:

הגדר את מיקום תחילת התנועה למיקומו הנוכחי.

אפס את הלוח (פונקציה\_8).

מצא את כל המהלכים האפשריים של החייל (פונקציה\_5).

אם הוא יכול לבצע צעד קדימה ולכיוון מרכז הלוח:

הגדר את מיקום סוף התנועה למיקום המוגדר.

בצע את התנועה (פונקציה\_9).

מצא את החייל שביצע את הקפיצה, ועדכן את מיקומו למיקום החדש.

סיים את התור (פונקציה\_7).

אם הוא נמצא במרכז הלוח ומסוגל לבצע צעד קדימה לאחד הכיוונים:

הגדר את מיקום סוף התנועה למיקום המוגדר.

בצע את הצעד (פונקציה\_9).

מצא את החייל שביצע את הקפיצה, ועדכן את מיקומו למיקום החדש.

סיים את התור (פונקציה\_7).

אם הוא מסוגל לבצע צעד לכיוון מרכז הלוח באותה שורה בה הוא נמצא:

הגדר את מיקום סוף התנועה למיקום המוגדר.

בצע את הצעד (פונקציה\_9).

מצא את החייל שביצע את הקפיצה, ועדכן את מיקומו למיקום החדש.

סיים את התור (פונקציה\_7).

אם הוא מסוגל לבצע צעד קדימה אך לא למרכז הלוח:

הגדר את מיקום סוף התנועה למיקום המוגדר.

בצע את הצעד (פונקציה\_9).

מצא את החייל שביצע את הקפיצה, ועדכן את מיקומו למיקום החדש.

סיים את התור (פונקציה\_7).

**פונקציה\_1: countOutsideSoldiers**

1. הגדר מונה עם ערך ראשוני 10.
2. עבור כל תא בבסיס היעד:
   1. אם בתא יש חייל, הקטן את המונה ב-1.
3. החזר את המונה.

**פונקציה\_2: is\_S\_TurnExist**

1. עבור כל תא בלוח מלמטה:
   1. אם בתא יש חייל של המחשב:
   2. אפס את הלוח (פונקציה\_8).
   3. מצא את כל המהלכים האפשריים מהתא (פונקציה\_5).
      1. אם {קיים חייל של המחשב במקומות (אותה עמודה, שתי שורות למטה) וגם [(עמודה אחת שמאלה, שורה אחת למטה) וגם (עמודה אחת שמאלה, שלוש שורות למטה)] וגם מטרה במקומות (אותה עמודה, ארבע שורות למטה) וגם (שתי עמודות שמאלה, שתי שורות למטה)} או {קיים חייל של המחשב במקומות (אותה עמודה, שתי שורות למטה) וגם [(עמודה אחת ימינה, שורה אחת למטה) וגם (עמודה אחת ימינה, שלוש שורות למטה)] וגם מטרה במקומות (אותה עמודה, ארבע שורות למטה) וגם (שתי עמודות ימינה, שתי שורות למטה)}:
         1. החזר את המיקום של החייל.
2. החזר ריק.

**פונקציה\_3: findCpuPlayerIndex**

1. אם המיקום שהועבר לא ריק:
   1. עבור כל חייל במערך החיילים של המחשב:
      1. אם מיקום החייל שווה למיקום שהועבר:
         1. החזר את אינדקס החייל.
2. החזר -1.

**פונקציה\_4: findFarthestJump**

1. הגדר את המהלך הרחוק ביותר והשווה לריק.
2. עבור כל חייל של המחשב מלמעלה למטה:
   1. אפס את הלוח (פונקציה\_8).
   2. מצא את כל הקפיצות האפשריות שלו (פונקציה\_6).
   3. עבור כל מהלך אפשרי מהשורה של החייל ומטה:
      1. אם המהלך הרחוק ביותר שווה לריק:
         1. אם התנועה מקדמת את החייל קדימה או למרכז:
            1. הגדר את התנועה לתנועה הרחוקה ביותר.
      2. אחרת:
         1. אם התנועה מקדמת את החייל רחוק יותר קדימה מהתנועה הקודמת או אם התנועה מקדמת את החייל באותה שורה כמו התנועה הקודמת אך קרוב יותר למרכז:
            1. הגדר את התנועה לתנועה הרחוקה ביותר.
3. החזר את התנועה הרחוקה ביותר.

**פונקציה\_5: optinalPlays**

1. צור קבוצת מיקומים ריקה.
2. עבור כל קשת היוצאת מהמיקום שהועבר לפונקציה:
   1. אם התא בצד השני של הקשת לא מכיל חייל:
      1. הכנס לתא ערך של מטרה אפשרית.
      2. הוסף את מיקום התא לקבוצת המיקומים
      3. אם הקשת היא קפיצה מעל חייל אחר:
         1. מצא את כל הקפיצות האפשריות מהתא (פונקציה\_6).
         2. הוסף אותן לקבוצת המיקומים.
3. החזר את קבוצת המיקומים.

**פונקציה\_6:** **optinalPlaysLen3**

1. צור קבוצת מיקומים ריקה.
2. עבור כל קשת היוצאת מהמיקום שהועבר לפונקציה:
   1. אם התא בצד השני ריק:
      1. אם הקשת היא קפיצה מעל חייל אחר:
         1. הכנס לתא ערך של מטרה אפשרית.
         2. הוסף את מיקום התא לקבוצת המיקומים.
         3. מצא את כל הקפיצות האפשריות מהתא (פונקציה\_6).
         4. הוסף אותן לקבוצת המיקומים.
3. החזר את קבוצת המיקומים.

**פונקציה\_7: endTurn**

1. אפס את הלוח.
2. אם אין ניצחון (פונקציה\_11):
   1. החלף את השחקן.
   2. אפס את מקום הנגיעה.

**פונקציה\_8: resetBoard**

1. עבור כל תא בגרף:
   1. אם אין בו חיילים אך הוא לא ריק (חייל לשעבר או מטרה):
      1. רוקן את התא.
2. עדכן את הקשתות בגרף (פונקציה\_10).

**פונקציה\_9: Move**

1. הכנס חייל למיקום החדש.
2. הכנס חייל עבר למיקום הישן.

**פונקציה\_10: updateGraph**

1. עבור כל קשת במחסן הקשתות:
   1. אם הקשת מחברת שני תאים צמודים:
      1. אם הקשת לא קיימת, הוסף את הקשת.
   2. אחרת:
      1. אם בתא שהקשת מדלגת עליו יש חייל:
         1. אם הקשת לא קיימת, הוסף את הקשת.
      2. אחרת:
         1. אם הקשת קיימת, מחק את הקשת.

**פונקציה\_11: win**

1. צור משתנה בוליאני עבור כל שחקן עם הערך אמת.
2. עבור כל תא בבסיסי היעד של שני הצדדים:
   1. אם בתא יש שחקן מהצד השני, הכנס את ערכו של המשתנה של השחקן לתוכו, אחת הכנס שקר.
3. אם המשתנה של השחקן הראשון מכיל אמת:
   1. הצג הודעה מתאימה.
   2. החזר אמת.
4. אם המשתנה של השחקן השני מציג אמת:
   1. הצג הודעה מתאימה.
   2. החזר אמת.
5. החזר שקר.

**פרק 4 – הקוד**

**AI:**

public void AI() {

int i;

// If only one player is outside

if (countOutsideSoldiers() == 1) {

CellVertex dest = null;

for (i = 0; i < cpuSoldiers.length; i++) {

if (cpuSoldiers[i].y < 13) {

for (int h = 13; h < GraphFacilities.H; h++) {

for (int j = 9 + (h - 13); j < 16 - (h - 13); j += 2) {

if (GraphFacilities.vertexMat[h][j].content

== PlayerAffiliation.NONE) {

dest = GraphFacilities.vertexMat[h][j];

}

}

}

List<CellVertex> path = GraphFacilities.findShortestPathLength(

GraphFacilities.vertexMat[cpuSoldiers[i].y][cpuSoldiers[i].x],

dest).getVertexList();

setTx(path.get(0).getLocation().x);

setTy(path.get(0).getLocation().y);

Move(path.get(1).getLocation().y, path.get(1).getLocation().x);

cpuSoldiers[i].setLocation(path.get(1).getLocation());

return;

}

}

}

// If there's an S turn available

if ((i = findCpuPlayerIndex(is\_S\_TurnExist())) > -1) {

setTx(cpuSoldiers[i].x);

setTy(cpuSoldiers[i].y);

Move(cpuSoldiers[i].y + 4, cpuSoldiers[i].x);

cpuSoldiers[i].y += 4;

} else {

if (cpuSoldiers[6].getLocation().equals(new Point(9, 3))

&& GraphFacilities.vertexMat[4][10].content

!= PlayerAffiliation.HUMAN\_PLAYER

&& GraphFacilities.vertexMat[4][10].content

!= PlayerAffiliation.CPU\_PLAYER) {

setTx(9);

setTy(3);

Move(4, 10);

cpuSoldiers[6].setLocation(10, 4);

} else if (cpuSoldiers[9].getLocation().equals(new Point(15, 3))

&& GraphFacilities.vertexMat[4][14].content

!= PlayerAffiliation.HUMAN\_PLAYER

&& GraphFacilities.vertexMat[4][14].content

!= PlayerAffiliation.CPU\_PLAYER) {

setTx(15);

setTy(3);

Move(4, 14);

cpuSoldiers[9].setLocation(14, 4);

} else {

CCEdge jump = findFarthestJump();

if (jump != null) {

setTx(jump.getSrcVertx().getLocation().x);

setTy(jump.getSrcVertx().getLocation().y);

Move(jump.getDestVertx().getLocation().y,

jump.getDestVertx().getLocation().x);

cpuSoldiers[findCpuPlayerIndex(

jump.getSrcVertx().getLocation())].setLocation(

jump.getDestVertx().getLocation());

} else {

for (i = 0; i < GraphFacilities.H; i++) {

for (i = 0; i < GraphFacilities.H; i++) {

for (int j = 0; j < GraphFacilities.W; j++) {

if (GraphFacilities.vertexMat[i][j] != null

&& GraphFacilities.vertexMat[i][j].content

== PlayerAffiliation.CPU\_PLAYER) {

setTx(j);

setTy(i);

resetBoard();

Set<Point> points = optinalPlays(tx, ty);

Iterator<Point> it = points.iterator();

while (it.hasNext()) {

Point p = it.next();

if ((p.y > ty

&& ((Math.abs(tx - GraphFacilities.W / 2)

> Math.abs(p.x - GraphFacilities.W / 2))

|| (Math.abs(tx - GraphFacilities.W / 2)

== 0)))) {

Move(p.y, p.x);

cpuSoldiers[findCpuPlayerIndex(

new Point(tx, ty))].setLocation(p);

return;

}

}

it = points.iterator();

while (it.hasNext()) {

Point p = it.next();

if ((p.y == ty

&& Math.abs(tx - GraphFacilities.W / 2)

> Math.abs(p.x - GraphFacilities.W / 2))) {

Move(p.y, p.x);

cpuSoldiers[findCpuPlayerIndex(

new Point(tx, ty))].setLocation(p);

return;

}

}

it = points.iterator();

while (it.hasNext()) {

Point p = it.next();

if (p.y > i) {

Move(p.y, p.x);

cpuSoldiers[findCpuPlayerIndex(

new Point(tx, ty))].setLocation(p);

return;

}

}

}

}

}

}

}

}

}

}

**countOutsideSoldiers:**

private int countOutsideSoldiers() {

int counter = 10;

for (int i = 13; i < GraphFacilities.H; i++) {

for (int j = 9 + (i - 13); j < 16 - (i - 13); j += 2) {

if (GraphFacilities.vertexMat[i][j].content

== PlayerAffiliation.CPU\_PLAYER) {

counter--;

}

}

}

return counter;

}

**Move:**

public void Move(int y, int x) {

GraphFacilities.vertexMat[y][x].content = player;

GraphFacilities.vertexMat[ty][tx].content = 99;

}

**is\_S\_TurnExist:**

private Point is\_S\_TurnExist() {

for (int i = GraphFacilities.vertexMat.length - 1; i >= 0; i--) {

for (int j = GraphFacilities.vertexMat[i].length - 1; j >= 0; j--) {

if (GraphFacilities.vertexMat[i][j] != null

&& GraphFacilities.vertexMat[i][j].content

== PlayerAffiliation.CPU\_PLAYER) {

resetBoard();

optinalPlays(j, i);

if (i + 4 < GraphFacilities.vertexMat.length

&& GraphFacilities.vertexMat[i + 4][j] != null

&& j - 1 > 0 && j + 1 < GraphFacilities.vertexMat[i].length

&& GraphFacilities.vertexMat[i + 2][j].content

== PlayerAffiliation.CPU\_PLAYER

&& GraphFacilities.vertexMat[i + 4][j].content

== PlayerAffiliation.POSSIBLE\_CPU\_TARGET

&& ((GraphFacilities.vertexMat[i + 1][j + 1].content

== PlayerAffiliation.CPU\_PLAYER

&& GraphFacilities.vertexMat[i + 3][j + 1].content

== PlayerAffiliation.CPU\_PLAYER

&& GraphFacilities.vertexMat[i + 2][j + 2].content

== PlayerAffiliation.POSSIBLE\_CPU\_TARGET)

|| (GraphFacilities.vertexMat[i + 1][j - 1].content

== PlayerAffiliation.CPU\_PLAYER

&& GraphFacilities.vertexMat[i + 3][j - 1].content

== PlayerAffiliation.CPU\_PLAYER

&& GraphFacilities.vertexMat[i + 2][j - 2].content

== PlayerAffiliation.POSSIBLE\_CPU\_TARGET))) {

return new Point(j, i);

}

}

}

}

return null;

}

**findCpuPlayerIndex:**

private int findCpuPlayerIndex(Point p) {

if (p != null) {

for (int i = 0; i < cpuSoldiers.length; i++) {

if (cpuSoldiers[i].equals(p)) {

return i;

}

}

}

return -1;

}

**endTurn:**

public void endTurn() {

resetBoard();

if (!win()) {

player = (player == PlayerAffiliation.CPU\_PLAYER)

? PlayerAffiliation.HUMAN\_PLAYER

: PlayerAffiliation.CPU\_PLAYER;

ty = 0;

tx = 0;

}

}

**resetBoard:**

public void resetBoard() {

for (int i = 0; i < H; i++) {

for (int j = 0; j < W; j++) {

if (GraphFacilities.vertexMat[i][j] != null

&& GraphFacilities.vertexMat[i][j].content > 9) {

GraphFacilities.vertexMat[i][j].content = PlayerAffiliation.NONE;

}

}

}

GraphFacilities.updateGraph(coordMat);

}

**findFarthestJump:**

private CCEdge findFarthestJump() {

CCEdge jump = null;

for (int i = 0; i < GraphFacilities.vertexMat.length; i++) {

for (int j = 0; j < GraphFacilities.vertexMat[i].length; j++) {

if (GraphFacilities.vertexMat[i][j] != null

&& GraphFacilities.vertexMat[i][j].content

== PlayerAffiliation.CPU\_PLAYER) {

resetBoard();

optinalPlaysLen3(j, i);

for (int h = i; h < GraphFacilities.vertexMat.length; h++) {

for (int g = 0; g < GraphFacilities.vertexMat[h].length; g++) {

if (GraphFacilities.vertexMat[h][g] != null

&& GraphFacilities.vertexMat[h][g].content

== PlayerAffiliation.POSSIBLE\_CPU\_TARGET) {

if (jump != null) {

if ((jump.getDestVertx().getLocation().y < h)

|| (jump.getDestVertx().getLocation().y == h

&& Math.abs(

jump.getDestVertx().getLocation().x

- (GraphFacilities.W / 2))

> Math.abs(g - (GraphFacilities.W / 2)))) {

if ((i < h)

|| (i == h

&& Math.abs(j - (GraphFacilities.W / 2))

> Math.abs(

g - (GraphFacilities.W / 2)))) {

jump.setSrcVertx(

GraphFacilities.vertexMat[i][j]);

jump.setDestVertx(

GraphFacilities.vertexMat[h][g]);

}

}

} else {

if ((i < h)

|| (i == h

&& Math.abs(j - (GraphFacilities.W / 2))

> Math.abs(g - (GraphFacilities.W / 2)))) {

jump = new CCEdge(GraphFacilities.vertexMat[i][j],

GraphFacilities.vertexMat[h][g]);

}

}

}

}

}

}

}

}

return jump;

}

**updateGraph:**

public static void updateGraph(Point[][] edgeArr) {

for (int i = 0; i < edgeArr.length; i++) {

if (edgeArr[i].length == 2) {

addEdge(edgeArr[i][0].y, edgeArr[i][0].x, edgeArr[i][1].y, edgeArr[i][1].x);

} else {

if (vertexMat[edgeArr[i][1].y][edgeArr[i][1].x].getContent()

== PlayerAffiliation.CPU\_PLAYER

|| vertexMat[edgeArr[i][1].y][edgeArr[i][1].x].getContent()

== PlayerAffiliation.HUMAN\_PLAYER) {

addEdge(edgeArr[i][0].y, edgeArr[i][0].x,

edgeArr[i][2].y, edgeArr[i][2].x);

} else {

g.removeEdge(vertexMat[edgeArr[i][0].y][edgeArr[i][0].x],

vertexMat[edgeArr[i][2].y][edgeArr[i][2].x]);

}

}

}

}

**optinalPlays:**

public Set<Point> optinalPlays(int x, int y) {

Set<Point> endPoints = new HashSet<>();

Set<CCEdge> edges

= GraphFacilities.g.outgoingEdgesOf(GraphFacilities.vertexMat[y][x]);

Iterator<CCEdge> it = edges.iterator();

while (it.hasNext()) {

CCEdge edge = it.next();

// Source isn't the origin.

if (!edge.getSrcVertx().getLocation().equals(new Point(x, y))) {

// Cell empty

if (edge.getSrcVertx().content != PlayerAffiliation.CPU\_PLAYER

&& edge.getSrcVertx().content != PlayerAffiliation.HUMAN\_PLAYER) {

if (player == PlayerAffiliation.HUMAN\_PLAYER) {

edge.getSrcVertx().content

= PlayerAffiliation.POSSIBLE\_HUMAN\_TARGET;

} else {

edge.getSrcVertx().content

= PlayerAffiliation.POSSIBLE\_CPU\_TARGET;

}

endPoints.add(edge.getSrcVertx().getLocation());

if (edge.getSrcVertx().getLocation().y - 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().y + 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().x - 4

== edge.getDestVertx().getLocation().x

|| edge.getSrcVertx().getLocation().x + 4

== edge.getDestVertx().getLocation().x) {

endPoints.addAll(

optinalPlaysLen3(edge.getSrcVertx().getLocation().x,

edge.getSrcVertx().getLocation().y));

}

}

} else {

// Cell empty

if (edge.getDestVertx().content

!= PlayerAffiliation.CPU\_PLAYER

&& edge.getDestVertx().content

!= PlayerAffiliation.HUMAN\_PLAYER) {

if (player == PlayerAffiliation.HUMAN\_PLAYER) {

edge.getDestVertx().content

= PlayerAffiliation.POSSIBLE\_HUMAN\_TARGET;

} else {

edge.getDestVertx().content

= PlayerAffiliation.POSSIBLE\_CPU\_TARGET;

}

endPoints.add(edge.getDestVertx().getLocation());

if (edge.getSrcVertx().getLocation().y - 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().y + 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().x - 4

== edge.getDestVertx().getLocation().x

|| edge.getSrcVertx().getLocation().x + 4

== edge.getDestVertx().getLocation().x) {

endPoints.addAll(

optinalPlaysLen3(edge.getDestVertx().getLocation().x,

edge.getDestVertx().getLocation().y));

}

}

}

}

return endPoints;

}

**optinalPlaysLen3**

public Set<Point> optinalPlaysLen3(int x, int y) {

Set<Point> endPoints = new HashSet<>();

Set<CCEdge> edges

= GraphFacilities.g.outgoingEdgesOf(GraphFacilities.vertexMat[y][x]);

Iterator<CCEdge> it = edges.iterator();

while (it.hasNext()) {

CCEdge edge = it.next();

// Source isn't the origin.

if (!edge.getSrcVertx().getLocation().equals(new Point(x, y))) {

// Cell empty

if (edge.getSrcVertx().content == PlayerAffiliation.NONE) {

if (edge.getSrcVertx().getLocation().y - 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().y + 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().x - 4

== edge.getDestVertx().getLocation().x

|| edge.getSrcVertx().getLocation().x + 4

== edge.getDestVertx().getLocation().x) {

if (player == PlayerAffiliation.HUMAN\_PLAYER) {

edge.getSrcVertx().content

= PlayerAffiliation.POSSIBLE\_HUMAN\_TARGET;

} else {

edge.getSrcVertx().content

= PlayerAffiliation.POSSIBLE\_CPU\_TARGET;

}

endPoints.add(edge.getSrcVertx().getLocation());

endPoints.addAll(

optinalPlaysLen3(edge.getSrcVertx().getLocation().x,

edge.getSrcVertx().getLocation().y));

}

}

} else {

// Cell empty

if (edge.getDestVertx().content == PlayerAffiliation.NONE) {

if (edge.getSrcVertx().getLocation().y - 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().y + 2

== edge.getDestVertx().getLocation().y

|| edge.getSrcVertx().getLocation().x - 4

== edge.getDestVertx().getLocation().x

|| edge.getSrcVertx().getLocation().x + 4

== edge.getDestVertx().getLocation().x) {

if (player == PlayerAffiliation.HUMAN\_PLAYER) {

edge.getDestVertx().content

= PlayerAffiliation.POSSIBLE\_HUMAN\_TARGET;

} else {

edge.getDestVertx().content

= PlayerAffiliation.POSSIBLE\_CPU\_TARGET;

}

endPoints.add(edge.getDestVertx().getLocation());

endPoints.addAll(

optinalPlaysLen3(edge.getDestVertx().getLocation().x,

edge.getDestVertx().getLocation().y));

}

}

}

}

return endPoints;

}

**Win:**

public boolean win() {

boolean p1 = true, p2 = true;

for (int i = 0; i < 10 && (p1 || p2); i++) {

p1 = (GraphFacilities.vertexMat[win[i][0]][win[i][1]].content

== PlayerAffiliation.HUMAN\_PLAYER && p1);

p2 = (GraphFacilities.vertexMat[(H - 1) - win[i][0]][win[i][1]].content

== PlayerAffiliation.CPU\_PLAYER && p2);

}

if (p1) {

if (status == 2) {

JOptionPane.showConfirmDialog(null, "Blue Player Won!", "Winner",

JOptionPane.CLOSED\_OPTION, JOptionPane.INFORMATION\_MESSAGE);

} else {

JOptionPane.showConfirmDialog(null, "You Won!", "Winner",

JOptionPane.CLOSED\_OPTION, JOptionPane.INFORMATION\_MESSAGE);

}

player = PlayerAffiliation.NONE;

return true;

} else if (p2) {

if (status == 2) {

JOptionPane.showConfirmDialog(null, "Red Player Won!", "Winner",

JOptionPane.CLOSED\_OPTION, JOptionPane.INFORMATION\_MESSAGE);

} else {

JOptionPane.showConfirmDialog(null, "The Computer Won!", "Winner",

JOptionPane.CLOSED\_OPTION, JOptionPane.INFORMATION\_MESSAGE);

}

player = PlayerAffiliation.NONE;

return true;

}

return false;

}

1. תנועת S היא תנועת קפיצה המערבת 4 חיילים של השחקן. החייל האחורי ביותר בכל תור קופץ שתי קפיצות כך שמיקומו הסופי הוא 4 שורות קדימה ובאותה עמודה. [↑](#footnote-ref-2)
2. החייל החיצוני ביותר הוא חייל הנמצא באחת הפינות של הבסיס המשולש (הפינות הצמודות לשטח הנייטראלי בלוח) **בתחילת המשחק**. [↑](#footnote-ref-3)