הערות:

* כללי: חסר עמוד שער, תוכן עניינים (אפשר לייצר אוטומטית) והקדמה.
* כללי: עדיף פונט ללא סריפים כמו Arial.
* כללי: סדר הפרקים לא ברור לי (אולי כי חסרה הקדמה). על כל פרק צריך שיהיה מצוין האם מדובר במשהו שהוא סקירה או משהו שהוא ספציפי לעבודה.
* כללי: כותרות צריכות להיות בסגנון כותרות (שיניתי במסמך)
* כללי: בפעם הראשונה שמזכירים מונח כדאי לציין בסוגריים את שמו באנגלית.
* מבוא ל-OOP: האם צריך? אני לא בטוח שחייבים את זה. אין במבוא שום דבר שהוא ספציפי ל-C#, הוא נכון לכל שפה.
* אופן פעולת אויבי המשחק: זה צריך להיות כאן? המשחק עדיין לא הוגדר (אולי זה רק עניין של עריכה). כאשר את מזכירה pathfinding שווה להזכיר שימושים מרכזיים של זה (Waze וכו')
* אלגוריתמי חיפוש: צריך לומר כמה מילים על זה ובעיקר – מה מחפשים?
* תיאור המשחק: ראוי לציין שאת המצאת את המשחק ושהמטרה העיקרית שלו היא להציג את האלגוריתם.

## מבוא לתכנות מונחה עצמים ב C#

תכנות מונחה עצמים הוא שם של פרדיגמת תכנות המשתמשת ב"עצמים" לשם תכנות. השיטה נועדה להיות אינטואיטיבית וקרובה יותר לדרך החשיבה של האדם. אתגר התכנות, אשר בעבר היה כיצד לכתוב את הלוגיקה, הוא עכשיו כיצד להגדיר את המידע.

עצמים מוגדרים עלי ידי טיפוסים. **טיפוס** (גם נקרא מחלקה) הוא סוג העצם, ועצם הוא מופע של טיפוס איתו ניתן לעבוד. כל טיפוס יכול לכלול סוגים שונים של איברים. מאפיינים מתארים את הנתונים הנשמרים בטיפוס, מתודות אשר מגדירות פעולות אשר ניתן לבצע, ואירועים המאפשרים תקשור בין מחלקות או עצמים שונים.

כאשר מתייחסים לתכנות מונחה עצמים מועלים שלושה עקרונות. **כימוס** אומר שקבוצה של מאפיינים, מתודות ואיברים אחרים מיוחסת כאיבר אחד – עצם. הגישה אל איברים אלו נעשית רק דרך הפונקציות של האובייקט המהוות את הממשק. בכך שכימוס לא מאפשר שינוי ישיר של משתני האובייקט ניתן למנוע שינויים לא מבוקרים.

**הורשה** מאפשרת להגדיר מחלקה חדשה אשר משתמשת, מרחיבה ומשנה את ההתנהגות המוגדרת במחלקה אחרת. המחלקה ממנה נורשים האיברים נקראת מחלקת הבסיס, והמחלקה היורשת איברים אלו נקראת מחלקה נגזרת. ב C# אין תמיכה בהורשה מרובה, משמע לכל מחלקה נגזרת יכול להיות רק מחלקת בסיס אחת. יכולת זו מאפשרת חיסכון בכתיבת קוד ושימוש חוזר בקוד קיים, מכיוון שאין צורך להגדיר טיפוס חדש כל פעם.

עקרון **הפולימורפיזם**, או בעברית רב צורתיות, הוא היכולת לתת לפונקציה במחלקת הבסיס מימוש שונה במחלקות הנגזרות כך שבזמן ריצה תופעל הפונקציה שמתאימה לטיפוס האובייקט. העיקרון מאפשר להתייחס לאובייקטים מסוגים שונים באופן זהה אם יש להם מחלקת בסיס משותפת. אפשרות זו יוצרת תוכניות נוחות לעבודה מכיוון שניתן להגדיר פונקציות שעל כל מחלקה להכיל, ולהפעיל פונקציות אלו על אובייקט מבלי לדעת מהו הסוג שלו.

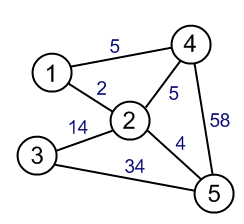
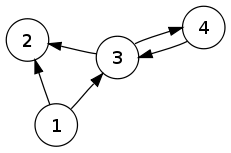
## אופן פעולת אויבי המשחק

אחד הנושאים הראשונים שנגישתי אליהם בתכנון הפרויקט הוא כיצד האויבים יעבדו. הקונספט של משחק דו-מימדי ממבט על הצריך דרך בה האויבים יכולים לעקוב אחר פעולות השחקן ובמידת הצורך לרדוף אחריו. בתיאוריה להגיע ממקום למקום זהו קונספט פשוט: אם יש מכשול בדרך אז עוקפים אותו. כשמגיעים למימוש, לעומת זאת, מתחילות לצוץ שאלות שהתשובה אליהן לאו דווקא ברורה. איך עוקפים מכשול מסוים? האם עקיפת מכשולים תמיד תוביל ליעד? איך מוצאים את הדרך המהירה ביותר? על מנת לענות על שאלות אלו חקרתי תחום במדעי המחשב בשם pathfinding, אשר מתייחס למציאת מסלולים על ידי שימוש בתורת הגרפים.

## מבוא לגרפים: הגדרות

גרף הוא קונספט מתמטי בעל שימוש נרחב במדעי המחשב. קיימות הרבה גרסאות של גרפים, אך כולן מוגדרות על ידי שני מאפיינים: קודקודים וצלעות. כל צלע מצייגת קישור בין שני קודקודים, ובאופן דומה אם שני קודקודים הם שכנים יש צלע המחברת ביניהם.

קיים הבדל בין גרפים מכוונים ולא מכוונים. בגרפים מכוונים הצלע מובילה מקודקוד אחד לשני בכיוון אחד, לכן השם, כאשר בגרפים לא מכוונים כל צלע היא דו כיוונית. בנוסף, גרף משוקלל הוא גרף בו לכל צלע יש משקל המייצג מספר, עלות, אורך או כל מדד אחר.



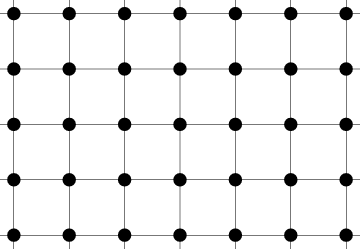
דוגמאות של גרף לא מכוון משוקלל (ימין) וגרף מכוון (שמאל)

### ייצוג מפה על ידי גרף

הצעד הראשון בעבודה עם אלגוריתם חיפוש מבוסס גרפים הוא להבין מה הקלט והפלט. הקלט הוא גרף המוגדר על ידי רשימת קודקודים וצלעות. אם הגרף משוקלל יהיו לצלעות גם ערך מספרי. חשוב להבין שמבחינת האלגוריתם אין אף חשיבות למה הגרף מייצג. הפלט יהיה הקודקודים והצלעות המרכיבים את המסלול.

מכאן ניתן להסיק כי קיימות דרכים רבות לייצג את מפת המשחק בתור גרף. ניתן לייצג חדרים בתור קודקודים ודלתות בתור צלעות. במקרה של עיר ניתן לייצג צמתות בתור קודקודים ורחובות בתור צלעות. הגרף יכול להיות מכוון ועל ידי כך לייצג תנועה חד כיוונית או משוקלל ובכך לייצג קושי תנועה. ההצגה השכיחה ביותר במשחקים היא של טבלה. טבלה היא מקרה פרטי של גרף מכוון משוקלל, בו תאים הם קודקודים ובין התאים הסמוכים מוגדרות הצלעות. לכל תא יש שמונה או ארבעה שכנים, בתלות או קיימת או לא קיימת תנועה אלכסונית.

אציג בפרקים הבאים את ההתפתחות של אלגוריתמי חיפוש בגרפים עם התמקדות בטבלאות.



טבלה בתור גרף

## מבוא לאלגוריתמי חיפוש

כל אלגוריתמי החיפוש שאציג בנויים על עיקרון של חזית מתפשטת המיוצגת על ידי תור. בכל איטרציה של הלולאה החיפוש מוציא קודקוד מהתור, מוצא את שכניו, מעריך כל שכן בנפרד ואם נמצא מתאים על פי התנאים מוסיף אותו לתור. החיפוש מפסיק כאשר כל הקודקודים נבדקו, דבר המיוצג על ידי כך שהתור ריק. לחילופין, החיפוש נגמר כאשר קודקוד היעד נבדק.

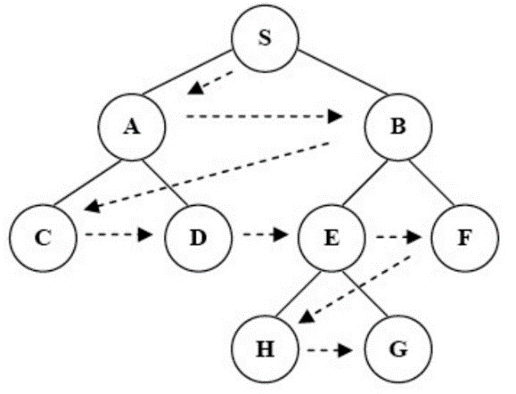
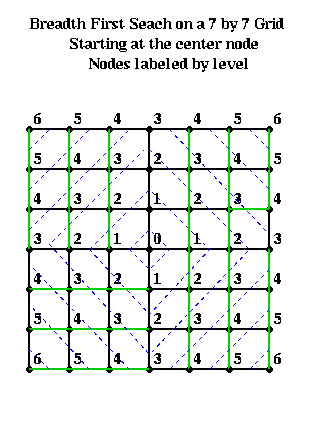
הווריאציות השונות של האלגוריתם תלוית ביישום הפונקציה של הוצאה מהתור. כאשר נבחר האלמנט הישן ביותר, כמו בדוגמה הנתונה, ייתקבל חיפוש לרוחב. כאשר נבחר האלמנט החדש ביותר ייתקבל חיפוש לעומק, Best-First search כאשר נבחר האלמנט בעל הערך הכי טוב ועוד.

ניתן לחלק חיפושים לשני סוגים: חיפוש עץ וחיפוש גרף. בחיפוש גרף קיים תנאי הבודק האם קודקוד נבדק לפני שהוא מתווסף לתור על מנת למנוע חזרות. חיפוש עץ, לעומת זאת, יכול לעבור על אותו הקודקוד מספר פעמים, אפילו אינסוף אם הגרף מכיל לולאה, אבל כל מעבר מתאים לשכן שונה בעץ. לאלגוריתמים המוצגים במסמך זה מצב בו יש חזרה לא מתאים מכיוון שכאשר קודקוד מסויים נבדק הדרך המובילה אליו היא בוודאות הטובה ביותר. לפיכך, לכולם תנאי המונע מעבר חוזר, דבר המגדיר אותם כחיפושים גרפיים.

כאשר אשווה אלגוריתמים שונים אתייחס לאופן בו מתבצעת פעולת ההוצאה מהתור, למאפיינים אשר נשמרים בכל קודקוד ולתנאי המונע חזרה.

### אלגוריתם חיפוש לרוחב (Breadth-first search)

האופן הכי בסיסי ליישם חיפוש מבוסס על התפשטות בשכבות עד ההגעה אל היעד נקרא "חיפוש לרוחב". החיפוש מתחיל בקודקוד מסוים, וכל פעם השכבה הבאה של שכנים הופכת לחזית.



הדגמת ריצת ההאלגוריתם על גרף כללי וטבלה

Breadth\_first\_search(graph, start, finish)

{

frontier = new queue()

frontier.enqueue(start)

start.parent = null //property

while(frontier.count != 0)

{

current = frontier.dequeue() //get next function

if(next.equals(finish)) //early exit

break

foreach(node next in graph.neighbors(current))

{

if(next.parent == null) //check condition

{

next.parent = current

frontier.enqueue(next)

}

}

}

**עצירה מוקדמת:** רלוונטית כאשר רוצים למצוא דרך לקודקוד מסויים. ברירת המחדל היא למצוא את הדרך מקודקוד מסויים לכל הקודקודים האחרים. כאשר אין בכך צורך ניתן להפסיק את החיפוש לאחר שמגיעים ליעד. במקרים שבהם דרך לא קיימת החיפוש יימשך כרגיל – עד שנבדקו כל הקודקודים, מצב אשר נקרא "הגרוע ביותר" לצורך הערכת סיבוכיות.

פעולת ההוצאה בחיפוש לרוחב מחזירה את הקודקוד הותיק ביותר, ולכן יהיה בעל מספר הצעדים הקטן ביותר הנדרש על מנת להגיע אליו. התוצאה היא שהאלגוריתם מתרחב באופן סימטרי בכל הכיוונים. המאפיין היחיד הנשמר לכל קודקוד הוא השכן ממנו התווסף, והוא גם התנאי המונע מעבר חוזר. אם קיים הורה משמע הקודקוד נבדק ואין צורך לבצע בדיקה חוזרת.

סיבוכיות הזמן של חיפוש לרוחב היאO(|V| + |E|), מאחר שבמקרה הכי גרוע (אין מסלול) כל קודקוד וצלע ייבדקו פעם אחת. |V| הוא מספר הקודקודים ו |E| מספר הצלעות. הטווח של O(|E|) נע בין O(1) ל )²O(|V|, בתלות בכמות הצלעות בין קודקודים בגרף. סיבוכיות המרחב תלויה ביצוג של הגרף, אך בדוגמה שנתתי היא תהיה O(|V|) מכיוון שנשמרת כמות זיכרון קבועה לכל קודקוד.

חיפוש לרוחב לאו דווקא בונה את הדרך אך מסביר איך לבקר בכל הגרף. זהו אלגוריתם עם שימושים רבים מעבר למציאת מסלולים. אפשר להשתמש בו בתור חיפוש אם שומרים לכל קודקוד את ה"הורה" שלו – השכן שלו אשר ממנו ניתן להגיע אליו. ניתן לשחזר את הדרך מקודקוד המטרה על ידי מעקב אחרי ההורה, כמו רשימה מקושרת רגילה.

Get\_path(graph, finish)

{

path = new list()

node next = finish.parent

while(next != null) //traces the parent back until the start

{

path.add(next)

previous = next.parent

}

path.reverse()

return(path) //if a path wasnt found the path returned will be empty

}

פסאודו קוד המשחזר את המסלול הנבנה על ידי חיפוש

זהו האלגוריתם הפשוט ביותר למציאת דרכים העובד על כל סוג של גרף. הוא נחשב לפשוט ביותר מכיוון שהוא ממצה את כל המסלולים המכוונים האפשריים ומחזיר את האחד הנגמר ביעד.

במקרה בו הגרף לא משוקלל זו תהיה הדרך הקצרה ביותר, אך בגרף משוקלל תיתכן דרך יותר קצרה מפני שהאלגוריתם מתייחס רק לשכבה או מספר הצעדים של מהקודקוד ההתחלתי. על מנת למצוא את הדרך האופטימלית בגרף משוקלל, בו לצלעות יש ערכים, יש צורך באלגוריתם המתייחס לערכים אלו.

### אלגוריתם דייקסטרה (Dijkstra’s algorithm)

יצירתו של אדסחר דייקסטרה אשר פורסמה ב 9195, לאלגוריתם הזה אלמנטים נוספים על חיפוש לרוחב אשר מאפשרים עבודה עם גרף משוקלל. המשמעותי ביותר הוא שמירת סכום ערכי הצלעות אשר מרכיבות את המסלול אל אותו הקודקוד בתור משתנה הנקרא עלות תנועה. משתנה זה מאפשר להעריך כל דרך מתמטית על מנת למצוא את הטובה ביותר.

Dijkstra’s\_algorithm(graph, start, finish)

{

frontier = new priorityQueue()

start.cost = 0

start.parent = null

frontier.enqueue(start, start.cost)

while(frontier.count != 0)

{

current = frontier.dequeue()

if(next.equals(finish))

break

foreach(node next in graph.neighbors(current))

{

newCost = current.cost + next.distance(current)

if(next.parent == null || newCost < next.cost)

{

next.cost = newCost

next.parent = current

frontier.enqueue(next, next.cost)

}

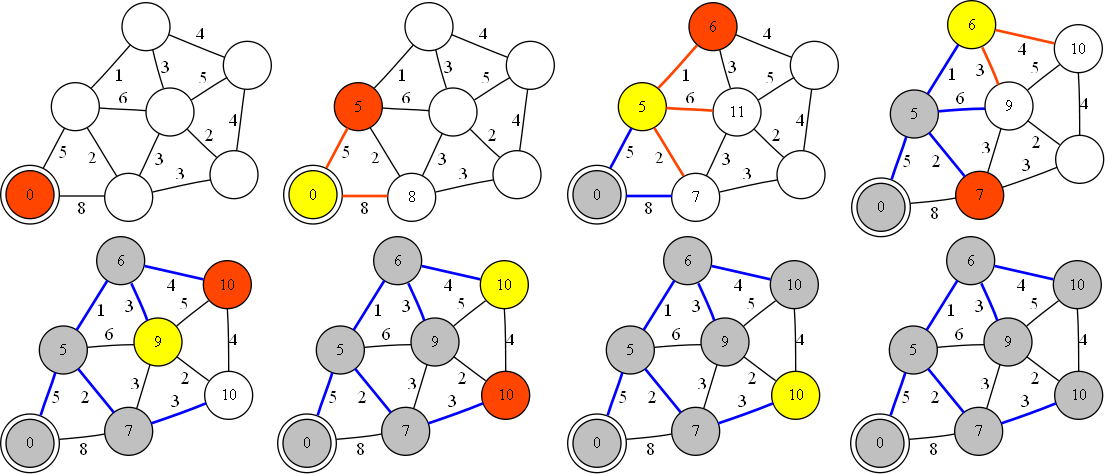
}

}

מימוש כללי של האלגוריתם של דייקסטרה המשתמש בתור עדיפויות

**תור עדיפויות:** תור אשר סדר איבריו נקבע על פי עדיפות מספרית הניתנת לכל איבר בעת הכנסתו לתור. הצצה והוצאת איבר מחזירים את האיבר בעל העדיפות המספרית הגבוהה ביותר. קיימים מספר מימושים של תור עדיפויות בעלי סיבוכיויות שונות. הסיבוכיות של ההכנסה (או ההוצאה, תלוי במימוש) היא או O(n) או O(log n) מכיוון שבהכנסה יש צורך לעבור על האיברים הקיימים בתור על מנת לקבוע היכן להכניס את האיבר החדש.

פעולת ההוצאה באלגוריתם של דייקסטרה מחזירה את הקודקוד בעל עלות התזוזה הכוללת הנמוכה ביותר, לפיכך גם הקרוב ביותר להתחלה. האלגוריתם ימצא את הדרך הקצרה ביותר לכל קודקוד שיחקור. נשמרים השכן ממנו התווסף ועלות התנועה. התנאי המונע חזרה שונה, ועכשיו יש שני תנאים אפשריים לבדיקה חוזרת. האחד הוא אם הקודקוד לא נבדק, כמו בחיפוש לרוחב. השני הוא אם הדרך החדשה שנבדקת יותר טובה מהדרך השמורה.



הדגמת ריצת האלגוריתם על גרף לא מכוון על מנת למצוא את כל המסלולים מקודקוד מסויים

לאלגוריתם של דייקסטרה יש הרבה מימושים שונים בעלי סיבוכיות שונה ויעוד שונה. באלגוריתם המקורי לא היה שימוש בתור עדיפויות, ולכן הסיבוכיות שלו הייתה )²O(|V| בהשוואה ל

O(|V|log |V| + |E|) כאשר משתמשים בערימת פיבונאצ'י (סוג מבנה נתונים המשמש מטרה דומה לתור עדיפויות). בדוגמה הנתונה סיבוכיות המקום תהיה O(|V|).

השימוש במשתנה עלות התנועה כעדיפות בתור מבטיח כי הקודקודים בעלי הערך הכי טוב נבדקים קודם, ולפיכך החזית מתפשטת על פי המרחק מהמרכז ולא סדר ההכנסה. במקרה הזה הדרך שתימצא תמיד תהיה הקצרה ביותר.

### חיפוש היוריסטי (Greedy Best-first search)

כאשר יש צורך למצוא דרך בין שני מקומות ולא בין מקום אחד לרבים ניתן לגרום לחזית להתפשט בכיוון המבטיח ביותר. האופן שבו קובעים את המשתנה ההיוריסטי הזה תלוי בסוג הגרף – אם מדובר בעץ ניתן לבחור בצלע הקצרה לקודקוד הנוכחי, אך האלגוריתם הכי אפקטיבי כאשר ניתן להעריך את הבחירה של כל קודקוד ביחס לכל הגרף. לדוגמה, במקרה של טבלה המשתנה ההיוריסטי יוערך על ידי מדידת מרחק מקודקוד היעד.

האלגוריתם ההיוריסטי מתקדם על פי הבחירה הטובה ביותר בכל שלב מקומי בתקווה למצוא דרך התהווה פתרון. אלגוריתם כזה נקרא Greedy (חמדן). ברוב המקרים אלגוריתם חמדן לא ימצא את הפתרון האופטימלי הכללי, אבל יש סיכוי כי ימצא פתרונות מקומיים המדמים כזה בזמן סביר.

Greedy\_best\_first\_search(graph, start, finish)

{

frontier = new priorityQueue()

start.cost = Heuristic(start, finish) //heuristic function

start.parent = null

frontier.enqueue(start, start.cost)

while(frontier.count != 0)

{

current = frontier.dequeue()

if(current.equals(finish))

break

foreach(node next in graph.neighbors(current))

{

if(next.parent == null)

{

next.cost = next.Heuristic(next, finish)

next.parent = current

frontier.enqueue(next, next.cost)

}

}

}

int Heuristic(a, b) //Manhattan distance on a square grid

{

return(Math.Abs(a.X - b.X) + Math.Abs(a.Y - b.Y))

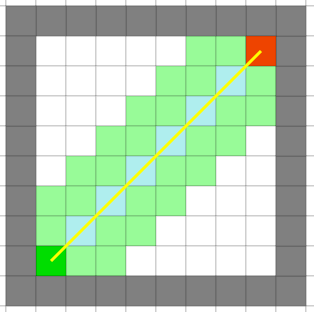
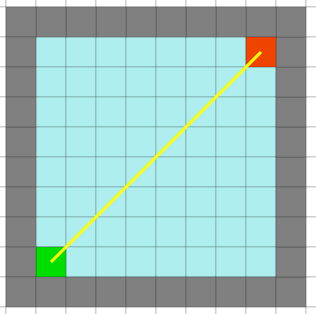
}

**מרחק מנהטן:** משתנה היוריסטי בגרפים מסוג טבלה. המרחק הוא סכום הפרשי הקורדינציות של שתי הנקודות. כאשר אין תנועה אלכסונית מרחק מנהטן יחזיר את המרחק הקצר ביותר מהיעד בלי התחשבות במכשולים. השם מבוסס על הגיאוגרפיה המוכרת של רחובות ניו-יורק.

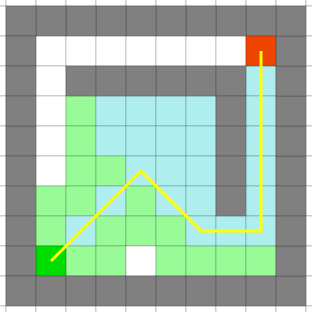
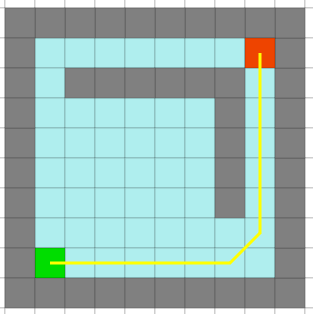
**Best-first search:** אלגוריתם המתקדם כל איטרציה לקודקוד המבטיח ביותר על פי חוק קבוע מראש. החוק הזה בדרך כלל מיוצג על ידי ערך מספרי משתנה התלוי באלגוריתם. אלגוריתם של דייקסטרה המשתמש בתור עדיפויות, למשל, משתמש בסכום עלות התנועה. כאשר BFS מסתמך אך ורק על משתנה היוריסטי הוא נקרא Greedy Best-first search.

בדומה לאלגוריתם של דייקסטרה, חיפוש היוריסטי מחזיר את הקודקוד בעל ערך התזוזה הטוב ביותר, רק שהפעם הוא המרחק המשוער מהסוף. האלגוריתם יתפשט בכיוון הכי מבטיח, היעד, ורק אם לא יגיע אליו יחזור ויבדוק דרכים שונות. אין חובה לשמור את המשתנה ההיוריסטי, אך בדוגמה שלי בחרתי לשמור אותו בנוסף להורה. אין טעם לבדוק מחדש קודקודים מכיוון שהמשתנה ההיוריסטי לא משתנה, ולפיכך התנאי היחיד הוא האם קיים הורה.

חיפוש היוריסטי לא תמיד ימצא את הדרך הטובה ביותר, ובמספר מימושים יכול להיתקע ולא למצוא דרך בכלל, אך יעשה זאת יותר מהר מחיפוש רוחבי. היתרונות והחסרונות של האלגוריתם נעשים ברורים כאשר הדרך האופטימלית אינה לינארית.

כאשר הדרך לא חסומה ניתן לראות כי אלגוריתם היוריסטי (מימין) מבקר רק בקודקודים הרצויים

ההחלטות המקומיות מובילות לדרך לא אופטימלית, בהשוואה לחיפוש רוחב המתייחס למכשול

**אלגוריתם A\***

השימוש בדייקסטרה תמיד יוביל לדרך האופטימלית, אך הוא מבזבז זמן בלחקור כיוונים לא מבטיחים. אלגוריתם היוריסטי חוקר כיוונים מבטיחים אבל הוא לאו דווקא ימצא את הדרך האופטימלית. איי סטאר משתמש גם בעלות התנועה הצבורה וגם במרחק המוערך ליעד על מנת למצוא את המסלול.

A\_star\_algorithm(graph, start, finish)

{

frontier = new priorityQueue()

start.g = 0 //g -> cost value to reach the node

start.h = start.Heuristic(finish) // h -> heuristic value

start.f = start.g + start.h //f -> combined movement cost

start.parent = null

frontier.enqueue(start, start.f)

while(frontier.count != 0)

{

current = frontier.dequeue()

if(current.equals(finish))

break

foreach(node next in graph.neighbors(current))

{

newCost = current.g + next.distance(current)

if(start.parent == null || newCost < next.g)

{

next.g = newCost

next.h = next.Heuristic(finish)

next.f = next.g + next.h

next.parent = current

frontier.enqueue(next, next.f)

}

}

}

אלגוריתם איי סטאר הוא השילוב של דייקסטרה וחיפוש היוריסטי.

## תיאור המשחק

Pathfinding.exe הוא משחק דו מימדי ממבט על בו השחקן מנווט דרך שלב, מתחמק מאויבים המנסים לסגור עליו, ולבסוף להגיע למשבצת המטרה. לוח המשחק או המפה הוא טבלה ריבועית בה התאים יכולים להיות ריקים או מלאים. קיים שחקן אחד ומספר אויבים הזזים לאחר כל מהלך של המשתמש. המשחק נגמר כשהשחקן מגיע לנקודת המטרה המוגדרת בשלב, במקרה הנקרא נצחון, או כאשר האויבים משיגים את השחקן, הנקרא הפסד. השחקן יזוז בהתאם לקלט המשתמש בכיווני החצים, אך לא יוכל לחצות משבצות המוגדרות מלאות. האויבים יתחילו לרדוף אחרי השחקן כאשר הוא במרחק מסויים מהם, הנקרא טווח. הרדיפה נעשית על ידי תזוזה בדרך הקצרה ביותר לכיוון השחקן כפי שהיא נקבעת על ידי אלגוריתם מציאת המסלול. על פי ברירת המחדל השלבים נוצרים אקראית, אך ניתן לשמור שלבים לקובץ טקסט ולטעון אותם. ניתן גם לערוך קבצים אלו ידנית. ניתן תיאורטית לשחק בכל גודל של לוח, אך קיימות הגבלות מטעמים מעשיים כגון זיכרון או גודל החלון.

אציג את המימוש של המשחק בפרקים: לוגיקה, קלט, גרפיקה, שמע ואחסון. כל פרק אציג את קטעי הקוד הרלוונטיים. ואסביר כיצד הם עובדים.

**לוגיקה**

//Main properties

private bool[,] map;

private Player myPlayer;

private List<Enemy> enemies = new List<Enemy>();

לוח המשחק הוא מערך בוליאני דו מימדי, אשר התנאי מייצג האם המשבצת ניתנת לחצייה. קיים שחקן אחד מסוג Player ורשימת אויבים מסוג Enemy.

public class Player

{

private Point p; //location

private int radius = 4; //detection radius

private Main form; //Main form

public Player(Point location, Main form) //initialize Player

{

P = location;

this.form = form;

}

public void Move(Direction dir) //checks if the cell is traversable, if so moves in said direction

public List<Point> DisplayRange() //gets the points that represent the player's detection radius

}

מחלקת Player. יש צורך במחלקה הראשית על מנת לבדוק את לוח המשחק.

public class Enemy

{

private Point p; //corrent location

private Point to; //goal

private Main form;

private PathFinder2 pfinder;

private Queue<Point> path = new Queue<Point>();

private bool active = false;

public Enemy(Point position, Main form) //initalize Enemy

{

this.P = position;

this.form = form;

pfinder = new PathFinder2(form.Map, P, P);

}

public void Move() //moves one step

public void Detected(Point at) //finds path to player

public bool IsInRange() //checks if this enemy is in the player's detection range

}