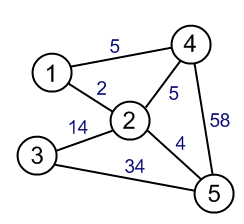
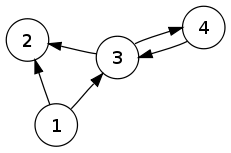
**אופן פעולת אויבי המשחק**

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.

קיימת הבדלה 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.



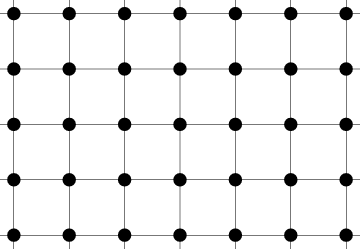
דוגמאות של גרף לא מכוון משוקלל (ימין) וגרף מכוון (שמאל)

**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.

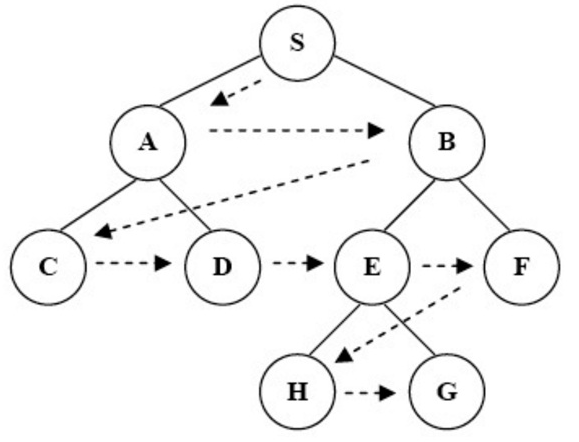
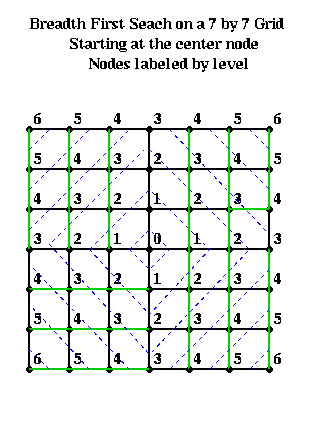
מכאן ניתן להסיק כי קיימות דרכים רבות לייצג את מפת המשחק בתור גרף. ניתן לייצג חדרים בתור קודקודים ודלתות בתור צלעות. ניתן לייצג צמתות בתור קודקודים ורחובות בתור צלעות. הגרף יכול להיות מכוון ועל ידי כך לייצג תנועה חד כיוונית או משוקלל ובכך לייצג קושי תנועה. ההצגה השכיחה ביותר במשחקים היא של טבלה. טבלה 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

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

העיקרון מאחורי אלגוריתמי החיפוש שאציג הוא גילוי במעגל מתרחב הנקרא החזית. חיפוש זה מבוסס על התפשטות שכבות עד הגעה אל היעד. האופן הכי פשוט ליישם עיקרון זה 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

while(frontier.count != 0) //while there are nodes left to check

{

current = frontier.dequeue()

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

{

if(next.parent == null) //if a node wasnt checked

{

next.parent = current

frontier.enqueue(next)

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

break

}

}

}

(לא בטוחה אם להוסיף הסבר מעין זה)

האלגוריתם יוצר תור חדש המכיל את הקודקוד ההתחלתי ומציב את ההורה כריק. הלולאה היא עיקר הקוד. התנאי הוא כל עוד החזית לא ריקה כל איטרציה תבדוק את כל הקודקודים השכנים של הקודקוד הנוכחי . אם לקודקוד אין הורה (משמע לא נבדק) הוא יתווסף לחזית, ההורה שלו יסומן כקודקוד הנוכחי, ואם הוא המטרה תבוצע עצירה מוקדמת.

**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.

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. באופן כזה ניתן לשחזר את הדרך מקודקוד המטרה על ידי מעקב אחרי ההורה.

Get\_path(graph, finish)

{

path = new list()

node previous = finish.parent

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

{

path.add(previous)

previous = previous.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

יוצרים משתנה מסוג קודקוד ומציבים אותו בתור ההורה של המטרה. יוצרים לולאה אשר התנאי שלה הוא שהמשתנה החדש לא ריק. אם לא נמצאה דרך הלולאה לא תתחיל. בתוך הלולאה מוסיפים לרשימת הדרך את הקודקוד הנוכחי, ומשנים את הנוכחי להורה שלו. לבסוף האלגוריתם מחזיר את הדרך.

זהו האלגוריתם הפשוט ביותר למציאת דרכים העובד על כל סוג של גרף. במקרה בו הגרף לא משוקלל זו תהיה הדרך הקצרה ביותר, אך בגרף משוקלל תיתכן דרך יותר קצרה. זה 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 יותר מפותח.

**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 "עלות התנועה" בכל קודקוד על מנת למצוא את הדרך בעל העלות הנמוכה ביותר. עלות התנועה היא סכום ערכי הצלעות אשר מרכיבות את המסלול אל אותו הקודקוד.

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

{

foreach(node n in graph)

{

n.cost = infinity

}

frontier = new priorityQueue()

start.cost = 0

start.parent = null

frontier.enqueue(start, start.cost)

while(frontier.count != 0)

{

current = frontier.dequeue()

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

{

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

if(newCost < next.cost)

{

next.cost = newCost

next.parent = current

frontier.enqueue(next, next.cost)

if(next.equals(finish))

break

}

}

}

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).

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). ניתן לייעל את האלגוריתם על פי סוג הגרף, אם מספר הצלעות נתון וכדומה. בדוגמה הנתונה סיבוכיות המקום תהיה O(|V|).

השימוש בעלות התנועה כעדיפות בתור מבטיח כי החזית מתפשטת על פי המרחק מהמרכז ולא סדר הכנסה. במקרה הזה הדרך שתימצא תמיד תהיה הקצרה ביותר.

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

**Greedy Algorithm:** מתקדם על פי הבחירה הטובה ביותר בכל שלב מקומי בתקווה למצוא דרך התהווה פתרון, אשר נקראת פונקציה היוריסטית. ברוב המקרים אלגוריתם חמדן לא ימצא את הפתרון האופטימלי הכללי, אבל יש סיכוי כי ימצא פתרונות מקומיים המדמים כזה בזמן סביר.

**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.

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.

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

{

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

}

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

{

frontier = new priorityQueue()

start.cost = Heuristic(start, finish)

start.parent = null

frontier.enqueue(start, start.cost)

while(frontier.count != 0)

{

current = frontier.dequeue()

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)

if(next.equals(finish))

break

}

}

}