AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca inżynierska

Mateusz Winiarski

kierunek studiów: **informatyka stosowana**

# Sztuczna inteligencja w grach komputerowych na przykładzie logiki rozmytej, algorytmu stada i problemu najkrótszej ścieżki w grze 2D.

Opiekun: **dr inż. Janusz Malinowski**

Kraków, styczeń 2017

Oświadczam, świadomy odpowiedzialności karnej za poświadczenie nieprawdy, że niniejszą pracę dyplomową wykonałem osobiście i samodzielnie i nie korzystałem ze źródeł innych niż wymienione w pracy.

……

(czytelny podpis)

Ocena merytoryczna opiekuna.

Ocena merytoryczna recenzenta.

# Spis treści:

1. Wstęp. (Tutaj coś ogólnie o sztucznej inteligencji, jakieś wstępne wspomnienie o algorytmach, czemu takie wybrałem)
2. Algorytm szukania najkrótszej ścieżki.
    1. Część teoretyczna - omówienie algorytmów Dijkstry i jego rozwinięcia do A*.
    2. Część implementacyjna - tutaj omówiłbym moją implementację A*. Także zastanawiam się nad dołączeniem tutaj jakichś testów wydajnościowych (czas wyznaczenia ścieżki) czy jakościowych (koszt wyznaczonej ścieżki, długość wyznaczonej ścieżki) mojego algorytmu A* przy różnym sposobie szacowania pozostałej drogi, bądź jego braku (algorytm Dijkstry).
3. Algorytm stada.
    1. Część teoretyczna - omówienie algorytmu stada, reguł w nim zawartych i sposobu jego działania.
    2. Część implementacyjna - omówienie mojej implementacji reguł, tego jak stado jest zarządzane i jak poszczególne zwierzęta się zachowują na podstawie otrzymanych informacji.
4. Logika rozmyta.
    1. Część teoretyczna - omówienie logiki rozmytej samej w sobie, zbiorów rozmytych, działań możliwych na tych zbiorach.
    2. Część implementacyjna - omówienie własnej implementacji prostego silnika logiki rozmytej, pokazanie użytych zbiorów rozmytych oraz omówienie na przykładzie wyboru zadania do wykonania dla kolonisty i akcji do wykonania dla atakującego działania logiki rozmytej.
5. Opis wykorzystanych narzędzi.
    1. Krótki opis SFML.Net.
    2. W zależności od długości poprzednich 3 punktów omówiłbym tu jakieś ciekawsze zagadnienia z C#, które napotkałem w trakcie pisania programu. (Takie jak wbudowana w język obsługa zdarzeń, dopasowanie wzorca dostępne w najnowszej wersji C# itp)
    3. Tutaj zastanawiam się nad położeniem tego rozdziału. Lepiej umieścić go w pracy przed omówieniem algorytmów czy tak jak w tym spisie na końcu?
6. Podsumowanie.

# 1. WSTĘP

Coś co wygląda jak początki wstępu:

AI (od ang. *artificial intelligence* – sztuczna inteligencja) może być symulowana na wiele różnych sposobów. Celem gier komputerowych jest stworzenie intrygującej rozgrywki i świata, który zainteresuje gracza. Nie są one jedna programami mającymi symulować rzeczywistość, więc nie jest od nich wymagane dokładnego odwzorowania środowiska jakie znamy i z jakim się spotykamy na co dzień. Głównie z tego powodu AI w grach nastawione jest na imitację racjonalnego działania, a nie całego procesu myślowego. Co więcej, gry mają za zadanie dawać rozrywkę, zatem komputerowo sterowany przeciwnik również powinien popełniać błędy – jak człowiek, a nie być doskonałą wszechwiedzącą istotą.

Niestety w znacznej części gier, nie tylko tych najmniej rozbudowanych, komputerowi przeciwnicy zwykle korzystają z prostych maszyn stanów, drzew behawioralnych czy wręcz są po prostu oskryptowanym ciągiem następujących po sobie konstrukcji typu „if/else”. Daje to często dość mizerne efekty jeśli chodzi o naturalność ich zachowań przeciwników.

Logika rozmyta wykorzystywana jest często do symulacji pragnień czy emocji bądź dostosowania akcji jednostki, żeby była jak najbardziej „racjonalna” na bazie pewnych określonych cech tejże jednostki. Daje nam to sporą różnorodność zachowań i eliminuje wrażenie walki z zastępem przeciwników-klonów o identycznym zachowaniu.

Algorytm stada natomiast wykorzystywany jest do symulacji poruszania się dużych zbiorowisk obiektów. Dzięki kilku względnie prostym regułom pozwala nadać pewnej grupie obiektów realistyczne zachowania zbiorowe. Podobne są one do co możemy obserwować w stadzie ptaków, ławicy ryb czy roju pszczół.

Problem najkrótszej ścieżki

# 2. ALGORYTM SZUKANIA NAJKRÓTSZEJ ŚCIEŻKI

W większości obecnych gier postacie się poruszają. O ile zaprogramowanie stałej trasy, na przykład patrolu, jest proste, gdyż można określić kilka punktów, między którymi postać będzie się przemieszczać, o tyle napisanie postaci tak, żeby umiała tę trasę znaleźć sama, nie jest już takie trywialne. Sprowadza się ono do znalezienia pewnej ścieżki, najlepiej jak najkrótszej. Po angielsku problem ten określa się mianem: *pathfinding*.

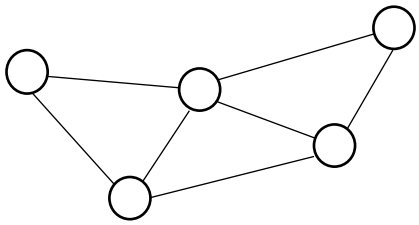
Większość gier do rozwiązania tego problemu wykorzystuje algorytm Dijkstry bądź też jego rozbudowaną, heurystyczną wersję – algorytm A*. Główny powód to fakt, że są one wydajne i łatwe do zaimplementowania[1].

## 2.1. GRAF W PROBLEMIE NAJKRÓTSZEJ ŚCIEŻKI

Niestety jednak zazwyczaj ani *algorytm Dijkstry*, ani *A** nie mogą w bezpośredni sposób działać na strukturach gry. Do poprawnego działania wymagają specyficznej struktury danych – *skierowanego grafu z nieujemnymi wagami*.

### 2.1.1. GRAF

*Graf* jest pewną matematyczną strukturą składającą się ze zbioru wierzchołków i zbioru połączeń między nimi[2]. Często są one reprezentowane wizualnie przez pewne punkty i połączenia między nimi.



*Rysunek 2.1. Przykładowy*

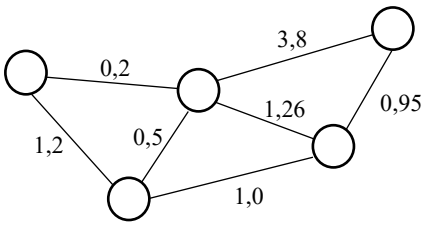
W *pathfindingu* wierzchołki grafu zwykle reprezentują pewne obszary albo punkty na mapie czy planszy. Jeśli między takimi dwoma rejonami gry istnieje bezpośrednie przejście, jest ono reprezentowane przez połączenie między odpowiednimi wierzchołkami.

### 2.1.2. GRAF Z NIEUJEMNYMI WAGAMI

*Graf z wagami* różni się od zwykłego grafu tym, że do jego połączeń dodaje się pewną numeryczną wartość. W matematyce jest ona nazywana wagą, natomiast w wypadku gier częściej - kosztem. Odpowiada ona kosztowi, jakim obarczone jest przemieszczenie się z jednego wierzchołka na drugi. Im jest ona większa, tym trudniej jest przejść pomiędzy danymi wierzchołkami lub zajmuje to więcej czasu.

Koszt ten może odpowiadać na przykład czasowi przejścia lub długości ścieżki, ale nic nie stoi na przeszkodzie, żeby był dowolną kombinacją pewnych parametrów.

[1] Ian Millington, *Artificial intelligence for games*, s. 204
[2] Ian Millington, *Artificial intelligence for games*, s. 205




*Rysunek 2.2. Przykładowy graf z wagami*

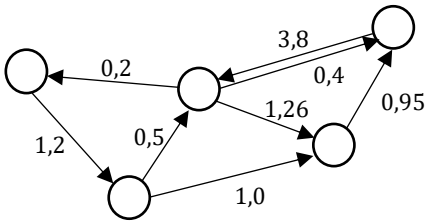
W takim grafie łatwo też określić całkowity koszt dotarcia z jednego wierzchołka na drugi, z nim nie sąsiadujący. Jest on obliczany jako suma poszczególnych przejść pomiędzy nimi.

Pomimo że teoria grafów dopuszcza ujemne wagi, to omawiane algorytmy ich nie dopuszczają. Wprawdzie możliwe jest, że w takich przypadkach algorytm zwróci w pewnych wypadkach poprawny wynik, jednak prawdopodobnym jest też, że *algorytm Dijkstry* czy *A** zapętlą się w takim grafie w nieskończoność[3]. Ponadto, jaki sens miałaby ujemna droga czy ujemny czas, które przecież przez wagę są reprezentowane?

### 2.1.3. GRAF SKIEROWANY

W wielu sytuacjach graf z nieujemnymi wagami jest wystarczający. Jednak łatwo wyobrazić sobie sytuację, w której przejście pomiędzy pewnymi punktami jest możliwe tylko w jedną stronę. Na przykład z niewielkiego urwiska da się łatwo zeskoczyć, ale wspiąć się na nie, bez dodatkowego sprzętu, jest w zasadzie niemożliwe.

Drugą rzeczą, jaką zyskujemy przy użyciu *grafu skierowanego*, jest możliwość ustawienia różnych wag w zależności od kierunku przemieszczania się. Kontynuując poprzedni przykład: jeśli podstawimy drabinę pod urwisko, jesteśmy w stanie się wspiąć, ale czas, jaki musimy na to poświęcić, jest znacznie dłuższy niż podróż w drugą stronę.




*Rysunek 2.3. Przykładowy skierowany graf z wagami*

## 2.2. ALGORYTM DIJKSTRY

Nazwa *algorytmu Dijkstry* pochodzi od nazwiska jego twórcy – Edsgera Wybe'a Dijkstry, holenderskiego matematyka, pioniera informatyki i laureata nagrody Turinga[4] w roku 1972 r.

Pierwotnie nie miał być to algorytm do wyznaczania najkrótszej ścieżki w dokładnie takim znaczeniu, w jakim rozumiemy to w kontekście gier. Miał on rozwiązywać matematyczny problem z teorii grafów o łudząco podobnej nazwie „najkrótszej ścieżki". Podczas gdy w grach przez ten termin rozumie się znalezienie najkrótszej drogi pomiędzy dwoma punktami, to w wypadku teorii grafów jego rozwiązaniem są najkrótsze ścieżki z punktu początkowego do wszystkich innych

[3] Ian Millington, *Artificial intelligence for games*, s. 207-208

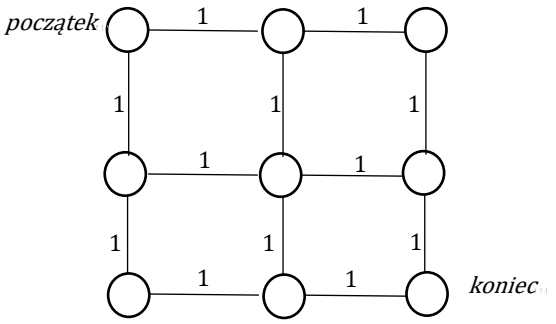
[4] Nagroda Turinga przyznawana corocznie od 1966 za wybitne osiągnięcia w dziedzinie informatyki przez Association for Computing Machinery. Nazwa została ustanowiona dla uczczenia jednego z twórców współczesnej informatyki, brytyjskiego matematyka Alana Turinga. Nagroda ta bywa czasem określana mianem „informatycznego Nobla".

wierzchołków[5]. Oczywiście chociaż jedną ze składowych jest poszukiwana w wypadku gier ścieżka, to jednak takie podejście byłoby marnotrawstwem zasobów i czasu procesora. Dlatego w tym wypadku zmienia się nieco algorytm Dijkstry tak, żeby szukał i zwracał tylko żądaną ścieżkę.

### 2.2.1. PROBLEM

Dany jest graf (skierowany z nieujemnymi wagami) oraz dwa z jego wierzchołków zwane początkiem i końcem szukanej ścieżki. Algorytm powinien wygenerować ścieżkę o możliwie najniższym koszcie ze wszystkich możliwych ścieżek pomiędzy początkiem a końcem.

Ścieżek spełniających powyższy warunek może być dowolna liczba. W takim wypadku powinna zostać zwrócona którakolwiek z nich.




*Rysunek 2.4. Przykładowy graf, w którym od początku do końca jest kilka ścieżek o równym koszcie.*

### 2.2.2. ROZWIĄZANIE

W dużym uproszczeniu można powiedzieć, że algorytm Dijkstry działa poprzez *rozlewanie się* w grafie przez występujące w nim połączenia od wierzchołków bliżej początkowego do tych położonych dalej. Zapamiętuje on kierunek poruszania się, dzięki czemu po osiągnieciu końcowego wierzchołka jest w stanie odtworzyć drogę, jaką przebył od początkowego wierzchołka. A sposób, w jaki odbywa się to rozlewanie, gwarantuje, że uzyskana przy odtwarzaniu ścieżka będzie najkrótszą z możliwych[6].

#### 2.2.2.1. PRZETWARZANIE OBECNEGO WĘZŁA

Podczas każdej iteracji przetwarzany jest pewien wierzchołek, zwany dalej obecnym węzłem. Rozważane jest każde wychodzące z niego połączenie wraz z wierzchołkiem, do którego ono prowadzi oraz kosztem całkowitego przejścia z węzła startowego do danego węzła sąsiadującego.

W przypadku, gdy obecnie rozważanym węzłem jest wierzchołek startowy, obliczenie całkowitego kosztu jest trywialne. Jest to po prostu koszt przejścia przez łączącą oba wierzchołki krawędź. Natomiast w przypadku kolejnych iteracji koszt ten obliczany jest jako suma kosztów przejścia po wszystkich krawędziach prowadzących od wierzchołka początkowego.

#### 2.2.2.2. LISTY WĘZŁÓW

Algorytm Dijkstry przechowuje w pamięci dwie listy węzłów, z którymi do tej pory miał styczność. Pierwsza z nich, zwana *listą otwartych*, zawiera wszystkie te węzły, które algorytm *zobaczył*, ale które nie miały jeszcze swojej iteracji. Oznacza to, że w wyniku przetwarzania innego wierzchołka, został dla nich obliczony koszt całkowity przejścia i oznaczony został kierunek, z któ-

[5] Ian Millington, *Artificial intelligence for games*, s. 211
[6] Ian Millington, *Artificial intelligence for games*, s. 211

rego to przejście miało miejsce, ale same jeszcze nie były przetwarzane. Natomiast na drugiej z nich, zwanej *listą zamkniętych*, znajdują się wszystkie te węzły, które przeszły już przez swoją iterację.

Każdy wierzchołek grafu może być w takim wypadku w jednym z trzech stanów:

1. *Otwartym* – wierzchołek jest na liście otwartych i oczekuje na swoją iterację;
2. *Zamkniętym* – wierzchołek przeszedł już swoją iterację;
3. *Nieodwiedzonym* – wierzchołek nie został do tej pory osiągnięty.

Algorytm rozpoczyna się od umieszczenia na liście otwartych wierzchołka startowego, natomiast lista zamkniętych jest pusta. Podczas iteracji wierzchołek z najmniejszym jak do tej pory kosztem całkowitym jest usuwany z listy otwartych i dodawany do listy zamkniętych, a jego każdy sąsiedni wierzchołek jest dodawany do listy otwartych.

W powyższym rozumowaniu może pojawić się jednak pewna komplikacja. Zakłada ono, że za każdym razem, przemieszczając się wzdłuż krawędzi grafu, natrafi się na wierzchołek do tej pory nieodwiedzony. Jednak można równie dobrze trafić na wierzchołek znajdujący się na liście otwartych czy zamkniętych.

#### 2.2.2.3. Obliczanie całkowitego kosztu dla listy zamkniętych i otwartych

Jeśli podczas iteracji dotrzemy do węzła, który został już odwiedzony, to będzie miał on swój całkowity koszt i kierunek, z którego został osiągnięty. Nie można ich tak po prostu ustawić na nowe, ponieważ nadpisałoby to poprzednią pracę wykonaną przez algorytm.

Zamiast tego należy sprawdzić, czy nowo otrzymana droga jest lepsza od poprzedniej. W tym celu obliczamy nowy koszt całkowity dla danego węzła i porównujemy go z kosztem całkowitym zapisanym już dla danego węzła. Jeśli nowa wartość jest większa, a tak będzie w większości wypadków [1], to nie zmieniamy nic i przechodzimy do następnego kroku.

Natomiast jeśli koszt całkowity jest mniejszy, to zmieniamy zarówno koszt całkowity danego węzła, jak i zapisany kierunek. Tak zmieniony węzeł musi znaleźć się na liście otwartych, więc jeśli natrafiliśmy na odwiedzony węzeł o mniejszym koszcie całkowitym na liście zamkniętych, należy przenieść go z powrotem na listę otwartych.

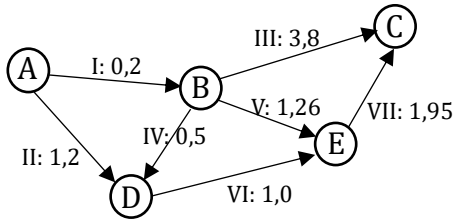
Jednak taka sytuacja w przypadku algorytmu Dijkstry nie ma możliwości zaistnienia z powodu wybierania zawsze węzła o najmniejszym koszcie całkowitym. Jest ona natomiast możliwa w przypadku algorytmu A*, który omawiany będzie później.

#### 2.2.2.4. Zakończenie algorytmu

W przypadku pierwotnego wykorzystania algorytmu Dijkstry kończy on swoje działanie, gdy lista otwartych jest pusta, co jest równoznaczne z przeglądnięciem wszystkich osiągalnych wierzchołków.

Jednak w wypadku szukania najkrótszej ścieżki można jego działanie zakończyć wcześniej. Wystarczy, że przerwiemy algorytm w momencie, gdy na liście otwartych to właśnie węzeł docelowy jest tym o najmniejszym koszcie całkowitym. Oznacza to także, że odkryliśmy węzeł docelowy w jednej z poprzednich iteracji.

Jednak przerwanie algorytmu w wypadku, gdy pierwszy raz wśród sąsiednich węzłów obecnie iterowanego wierzchołka znajdzie się cel, nie gwarantuje jeszcze tego, że wygenerowana w




*Rysunek 2.5. Nie zawsze pierwsza ścieżka jest najkrótsza.*

ten sposób ścieżka będzie najkrótsza.

Rozważmy sytuację z rysunku powyżej, gdzie przyjmiemy, że węzeł $A$ to początek szukanej ścieżki, a węzeł $C$ – koniec. Jeżeli przerwalibyśmy algorytm zaraz przy pierwszym napotkaniu węzła $C$, to otrzymalibyśmy ścieżkę $A \to B \to C$ o całkowitym koszcie $0{,}2 + 3{,}8 = 4{,}0$, natomiast najkrótszą ścieżką jest ścieżka $A \to B \to E \to C$ ponieważ jej koszt to $0{,}2 + 1{,}26 + 1{,}95 = 3{,}41$.

Jednak w praktyce programiści często wykorzystują tę drobną metodę przyspieszenia zwróconego wyniku[7]. Ma to miejsce z kilku powodów. Po pierwsze pierwsza droga zazwyczaj jest najkrótsza. Po drugie, nawet jeśli tak nie jest, to różnica w koszcie całkowitym tych dróg jest zazwyczaj nieznaczna.

#### 2.2.2.5. REKONSTRUKCJA ŚCIEŻKI

Na sam koniec algorytmu otrzymujemy pole końcowe wraz z połączeniem do węzła, z którego do niego doszliśmy. W tak przygotowanym grafie odtworzenie całej ścieżki jest proste. Wystarczy iterować się *w głąb* wyznaczonych połączeń z wierzchołka końcowego, aż otrzyma się wierzchołek początkowy. Podczas każdej iteracji należy dodać obecny wierzchołek na początek listy, która reprezentuje szukaną ścieżkę. Jeśli jednak dodajemy go na koniec, należy tak otrzymaną listę odwrócić.

### 2.2.3. PREZENTACJA DZIAŁANIA

Prezentacja działania algorytmu Dijkstry dla grafu z rysunku 2.5.

W iteracji pierwszej dodane zostaną dwa węzły – $B$ i $D$ – ponieważ sąsiadują one z węzłem $A$. Z kolei on sam zostanie przeniesiony na listę zamkniętych.

W drugiej iteracji przetwarzane będą węzły sąsiadujące z $B$ – $C$, $D$ i $E$ – ze względu na to, że węzeł ten ma jak do tej pory najmniejszy koszt całkowity. W tej iteracji zostanie także zaktualizowany koszt całkowity i połączenie dla węzła $D$, ponieważ droga przez węzeł $B$ jest „tańsza” niż przejście bezpośrednio z $A$. W tym momencie pierwszy raz dochodzimy do węzła końcowego $C$, jednak nie można mieć pewności, że otrzymana w ten sposób ścieżka jest „najtańszą” z możliwych.

W kolejnej iteracji jedynym, co się wykona, będzie przeniesienie badanego węzła $D$ z listy otwartych na listę zamkniętych. Do sąsiadującego z nim węzła $E$ droga $\cdots \to B \to D$ byłaby „droższa” niż droga tylko przez węzeł $B$, więc nie zostanie on uaktualniony.

[7] Ian Millington, *Artificial intelligence for games*, s. 214-215

W tej chwili to właśnie węzeł $E$ ma najmniejszy koszt całkowity i zostanie przebadany jego sąsiad – węzeł C. Na koniec iteracji na liście otwartych najmniejszy koszt całkowity ma końcowy węzeł $C$, więc można stwierdzić, że otrzymana w ten sposób droga jest najbardziej optymalną.

| $i$ | Obecny węzeł | Lista otwartych | | | | Lista zamkniętych | | | Otrzymana ścieżka |
|---|---|---|---|---|---|---|---|---|---|
| | | nr | Węzeł | Koszt | Połączenie | Węzeł | Koszt | Połączenie | |
| | | 1. | $A$ | 0 | – | – | – | – | |
| 1 | $A$ | 1. | $B$ | 0,2 | $I$ | $A$ | 0 | – | |
| | | 2. | $D$ | 1,2 | $II$ | | | | |
| 2 | $B$ | 1. | $D$ | 0,7 | $IV$ | $A$ | 0 | – | $A \rightarrow B \rightarrow C$ |
| | | 2. | $E$ | 1,26 | $V$ | $B$ | 0,2 | $I$ | |
| | | 3. | $C$ | 3,8 | $III$ | | | | |
| 3 | $D$ | 1. | $E$ | 1,26 | $V$ | $A$ | 0 | – | |
| | | 2. | $C$ | 3,8 | $III$ | $B$ | 0,2 | $I$ | |
| | | | | | | $D$ | 0,7 | $IV$ | |
| 4 | $E$ | 1. | $C$ | 3,41 | $VII$ | $A$ | 0 | – | $A \rightarrow B \rightarrow E \rightarrow C$ |
| | | | | | | $B$ | 0,2 | $I$ | |
| | | | | | | $D$ | 0,7 | $IV$ | |
| | | | | | | $E$ | 1,26 | $V$ | |

*Tabela 2.1 Wyniki poszczególnych iteracji algorytmu Dijkstry i ścieżki, które można by otrzymać w wypadku przerwania algorytmu w danej iteracji dla grafu z rysunku powyżej*

## 2.3. ALGORYTM A*

Szukanie najkrótszej ścieżki to w wypadku gier niemalże synonim *algorytmu A** (wym. *A gwiazdka* lub z ang. *A star*). Dzieje się tak, ponieważ jest on prosty do zaimplementowania, szybki, a dodatkowo daje duże pole do manewru przy optymalizacji.

W przeciwieństwie do algorytmu Dijkstry, A* jest zaprojektowany do nawigacji punkt-punkt, a nie do rozwiązywania problemu najkrótszej ścieżki w rozumieniu teorii grafów.

### 2.3.1. PROBLEM

Problem i założenia tego algorytmu są identyczne z algorytmem Dijkstry. Potrzebny jest skierowany graf z nieujemnymi wagami. Dwa wierzchołki z tego grafu oznaczają początek i koniec szukanej ścieżki. Przy tak zadanych danych algorytm powinien wygenerować najkrótszą możliwą ścieżkę.

### 2.3.2. ROZWIĄZANIE

Rozwiązanie problemu jest niemalże identyczne jak omawianego wcześniej algorytmu Dijkstry. Jednak zmienia się nieco definicja kosztu całkowitego. W tym algorytmie reprezentuje on sumę kosztu dotychczasowej ścieżki i *oszacowanego* kosztu ścieżki do węzła końcowego.

Wydajność algorytmu w dużej mierze zależy od doboru sposobu owego oszacowania. Jeśli oszacowanie będzie dokładne, to algorytm będzie wydajny, w przeciwnym wypadku algorytm A* może być nawet mniej efektywny od algorytmu Dijkstry[8].

[8] Ian Millington, *Artificial intelligence for games*, s. 223

Ze względu na niewielkie różnice w samym algorytmie omówię tylko te jego aspekty, które są nowe w stosunku do algorytmu Dijkstry.

#### 2.3.2.1. HEURYSTYKA

W informatyce *heurystyka* (od starogreckiego: *εὑρίσκω [heurískō]* – znaleźć, odkryć[9]) jest to technika używana przy znajdowaniu rozwiązań wtedy, gdy zwykłe algorytmy są niewystarczająco wydajne bądź nieznane. Metoda ta nie daje gwarancji znalezienia rozwiązania optymalnego, a często nawet nie daje gwarancji znalezienia rozwiązania w ogóle. Wykorzystywana jest też często do znajdowania bądź oszacowywania pewnych rozwiązań przybliżonych czy cząstkowych, na podstawie, których później wyliczane jest rozwiązanie ostateczne[10]. To też ma miejsce w algorytmie A*.

W momencie, gdy w algorytmie Dijkstry podczas iteracji obliczana jest wartość całkowitego kosztu dotarcia do sąsiedniego węzła, w tym algorytmie „wyliczana" jest też pewna nieznana wartość wynikająca z heurystyki (często zwana *zmienną heurystyczną*), która ma określać pozostały koszt dotarcia z owego sąsiedniego węzła do wierzchołka końcowego[11].

Jako że nie jest możliwe, żeby ten koszt był znany (wtedy przecież znana byłaby również szukana ścieżka), musi być znana jakaś metoda umożliwiająca jego oszacowanie. Oczywiście istnieje ich wiele i są one w dużym stopniu zależne od sposobu prezentacji przestrzeni w grafie.

Jeśli heurystyka zawsze niedoszacowuje pozostałego kosztu, algorytm będzie potrzebował więcej czasu na znalezienie najkrótszej ścieżki. W takim przypadku wyznaczona ścieżka będzie ścieżką o najmniejszym koszcie z możliwych i dokładnie tą samą, która byłaby wyznaczona przez algorytm Dijkstry. Jednakże jeśli choć raz nastąpi przeszacowanie, to nie można już tego zagwarantować[12].

W przeciwnym wypadku, jeśli heurystyka zawsze przeszacowuje wartość pozostałego kosztu, algorytm będzie miał tendencję do generowania krótszych ścieżek, z mniejszą ilością węzłów, nawet jeśli miałyby one większy koszt całkowity. Jednak w ten sposób można wiele zyskać na szybkości algorytmu[12]. Oczywiście jest pewna granica przeszacowania, powyżej której algorytm zaczyna bardzo szybko tracić na wydajności.

##### 2.3.2.1.1. HEURYSTYKA EUKLIDESOWA

Najbardziej naturalnym sposobem oszacowania pozostałej drogi jest obliczenie zwykłej euklidesowej odległości pomiędzy dwoma punktami. Gwarantuje to niemalże zawsze niedoszacowanie zmiennej heurystycznej. W najgorszym przypadku oszacowanie i zmienna heurystyczna będą sobie równe.

##### 2.3.2.1.2. HEURYSTYKA OPARTA NA METRYCE MANHATTAN

Heurystyka ta często wykorzystywana jest w grach, w których przestrzeń opiera się na siatce połączonych ze sobą kwadratów (ang. *grid-like game*)[13]. Odległość w tej metryce określa się wzorem:

[9] https://en.wikipedia.org/wiki/Heuristic [13 listopada 2016 r.], https://pl.wiktionary.org/wiki/εὑρίσκω [13 listopada 2016 r.]
[10] https://en.wikipedia.org/wiki/Heuristic_(computer_science) [13 listopada 2016 r.]
[11] Mat Buckland *Programming Game AI by* Example s. 241, Ian Millington, *Artificial intelligence for games*, s. 223-224
[12] Ian Millington, *Artificial intelligence for games*, s. 239-240
[13] Mat Buckland *Programming Game AI by* Example s. 246

$$d_n(A,B) = \sum_{k=1}^{n} |A_k - B_k|,$$

gdzie $a, b$ to punkty w przestrzeni, a $n$ to ilość wymiarów przestrzeni. W podanych grach najczęściej przestrzeń jest dwuwymiarowa, w której współrzędne punktów są oznaczane w następujący sposób:

$$P = (P_x, P_y).$$

Sprowadza to powyższy wzór do następującego:

$$d_2(A,B) = |A_x - B_x| + |A_y - B_y|.$$

Głównym powodem używania tej metryki zamiast metryki euklidesowej jest fakt, że nie trzeba obliczać pierwiastka kwadratowego, co jest bardzo kosztowną operacją.

#### 2.3.2.1.3. HEURYSTYKA NULL

Jest to bardzo prosta heurystyka, która w każdym wypadku oszacowuje pozostałą odległość na zero. W takim przypadku koszt całkowity jest z powrotem równy kosztowi ścieżki do badanego węzła, co sprowadza algorytm A* do algorytmu Dijkstry.

### 2.3.2.2. ZMIANY W ALGORYTMIE WYNIKAJĄCE Z ZASTOSOWANIA HEURYSTYKI

Heurystyka wymusza pewne zmiany w samym algorytmie względem algorytmu Dijkstry.

Po pierwsze wybierany do iteracji jest za każdym razem węzeł o najniższym koszcie całkowitej drogi, a nie jak poprzednio o najmniejszym koszcie drogi umożliwiającej dotarcie do niego. Pozwala to algorytmowi iterować się najpierw przez węzły, które są bardziej obiecujące i wydają się być bliżej szukanego celu. W ten sposób wybierane są takie węzły, które mają relatywnie niski zarówno koszt dotarcia do nich, jak i oszacowany koszt dotarcia od nich do celu.

Po drugie w przypadku natrafienia na węzeł znajdujący się już na liście otwartych bądź zamkniętych możliwe jest, że trzeba będzie poprawić jego wartości. O ile dla algorytmu Dijkstry tylko w przypadku węzłów z listy otwartych należało poprawiać wartości dotychczasowego kosztu ścieżki, o tyle w wypadku algorytmu A* należy to samo zrobić, gdy koszt dotarcia do węzła jest niższy dla węzłów znajdujących się już na liście zamkniętych. Dzieje się tak, ponieważ A* może znaleźć lepszą drogę do węzła, przez który już wcześniej się iterował. W takim przypadku trzeba by tę mniejszą wartość dotychczasowego kosztu rozpropagować przez wszystkie wychodzce połączenia danego węzła, co wymagałoby sporo pracy. Żeby uniknąć tego problemu, wystarczy usunąć dany węzeł z listy zamkniętych i dodać wraz z nowymi wartościami na listę otwartych, żeby mógł być z powrotem przetworzony.

Wiele implementacji algorytmu A* kończy jego działanie w momencie, gdy na liście otwartych końcowy węzeł ma najmniejszy całkowity koszt. Jednak, zgodnie z tym, co napisane jest powyżej, może okazać się, że zajdzie potrzeba zmiany wartości w tym węźle z powodu znalezienia krótszej ścieżki. Żeby umożliwić znalezienie tej ścieżki, należy poczekać do momentu, aż węzeł z najmniejszym dotychczasowym kosztem ścieżki będzie mieć ten koszt co najmniej równy całkowitemu kosztowi węzła końcowego. Jest to w zasadzie taki sam warunek końcowy, jak dla algorytmu Dijkstry, ale przedstawiony z nieco innej perspektywy.

### 2.3.3. PREZENTACJA DZIAŁANIA

Jako plansza w dalszych rozważaniach przyjęty jest poniższy rysunek. Na Planszy koszt przejścia pomiędzy kafelkami liczony jest według wzoru:

$$k = d * f$$

Gdzie $k$ to koszt. $d$ to odległość między kafelkami, a $f$ to pewna zmienna zależna od koloru kafelka, która przyjmuje następujące wartości:

- 1 przy przechodzeniu na zielony kafelek;
- 2 przy przechodzeniu na żółty kafelek;
- 0,5 przy przechodzeniu na szary kafelek.

Natomiast przejście na niebieski kafelek jest niemożliwe ($f = \infty$). Odległości liczone są zgodnie z metryką euklidesową, na której oparta będzie również heurystyka. Dla ułatwienia obliczeń wartości będą zaokrąglane do dwóch miejsc po przecinku. W nawiasach podano współrzędne poszczególnych pól. Jako startowe pole przyjęto pole $A$, natomiast za końcowe przyjęto pole $D$.

| | | | |
|---|---|---|---|
| $\boldsymbol{A(0,0)}$ **POCZĄTEK** | $B(1,0)$ | $C(2,0)$ | $\boldsymbol{D(3,0)}$ **KONIEC** |
| $E(0,1)$ | $F(1,1)$ | $G(2,1)$ | $H(3,1)$ |
| $I(0,2)$ | $J(1,2)$ | $K(2,2)$ | $L(3,2)$ |
| $M(0,3)$ | $N(1,3)$ | $O(2,3)$ | $P(3,3)$ |

*Rysunek 2.6 Przykładowa plansza gry.*

Na samym początku nie ma zbyt wielkiego wyboru i z węzła $A$ można przemieścić się tylko na węzeł $E$. Jest to obarczone kosztem równym $k = 1 * 1 = 1$, funkcja heurystyczna natomiast dla tego węzła zwróci wartość $h = \sqrt{(0-3)^2 + (1-0)^2} = \sqrt{10} \approx 3{,}16$.

W drugiej iteracji badany węzły $I$ i $J$. W przypadku węzła $I$ sytuacja jest analogiczna do poprzedniej. Natomiast węzeł $J$ leży po przekątnej węzła $E$ więc odległość jest równa $d = \sqrt{2} \approx 1{,}41$, a skoro $J$ jest kafelkiem żółtym to koszt przejścia jest równy $k = 1{,}41 * 2 = 2{,}82$.

W trzeciej iteracji to węzeł $I$ ma najmniejszy koszt całkowity i do listy otwartych dodawani są jego sąsiedzi. Jednym z jego sąsiadów jest badany już wcześniej węzeł $J$, który obecnie znajduje się na liście otwartych. Jednak koszt przejścia $\cdots \rightarrow E \rightarrow I \rightarrow J$ jest równy 4, co jest wyższą wartością niż bezpośrednie przejście z $E$ do $J$ wyliczone w poprzedniej iteracji.

Kolejnym wybranym do iteracji węzłem jest $J$. W tym kroku na liście otwartych pojawiają się kafelki $G$, $K$ i $O$.

Następnie przetwarzany jest węzeł $G$. W tej iteracji szukany węzeł $D$ pierwszy raz pojawi się na liście otwartych, jednak nie można mieć pewności, że będzie to optymalna trasa do niego, pomimo że będzie to najkrótsza z możliwych.

W kolejnym kroku sprawdzamy sąsiadów węzła $H$. Okazuje się, że droga $\cdots \rightarrow G \rightarrow H \rightarrow D$ jest mniej kosztowna niż byłoby bezpośrednio przejście $\cdots \rightarrow G \rightarrow D$ z poprzedniej iteracji.

Jednak z powodów omówionych wcześniej nie możemy w następnej iteracji przy przetwarzaniu kafelka $D$ przerwać algorytmu. Są na liście otwartych kafelki o mniejszym koszcie przejścia do nich niż on.

W następnych krokach węzły $L$, $K$, $N$ oraz $M$ zostaną przeniesione kolejno na listę zamkniętych. Pole $C$ można pominąć ponieważ wiadomo, że dopóki ma większy koszt dotarcia do niego od startu niż od startu do końcowego kafelka $D$, to nie da się z niego utworzyć lepszej ścieżki.

W tym momencie wiadomo, że znaleziona wcześniej ścieżka jest „najtańszą" z możliwych.

| $i$ | Węzeł | Lista otwartych | | | | | | Lista zamkniętych | | | | | Otrzymana ścieżka |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | nr | Węzeł | Koszt | Heu. | Sum. | Poł | Węzeł | Koszt | Heu. | Sum. | Poł | |
| | | 1. | A | 0 | - | - | - | - | - | - | - | - | |
| 1 | A | 1. | E | 1 | 3,16 | 4,16 | A | A | 0 | - | - | - | |
| 2 | E | 1. | I | 2 | 3,6 | 5,6 | E | A | 0 | - | - | - | |
| | | 2. | J | 3,82 | 2.83 | 6,65 | E | E | 1 | 3,16 | 4,16 | A | |
| 3 | I | 1. | J | 3,82 | 2.83 | 6,65 | E | A | 0 | - | - | - | |
| | | 2. | N | 4 | 4.24 | 8,24 | I | E | 1 | 3,16 | 4,16 | A | |
| | | 3. | M | 4,82 | 3,61 | 8,43 | I | I | 2 | 3,6 | 5,6 | E | |
| 4 | J | 1. | G | 4,32 | 1,41 | 5,73 | J | A | 0 | - | - | - | |
| | | 2. | K | 5,82 | 2.24 | 8,06 | J | E | 1 | 3,16 | 4,16 | A | |
| | | 3. | N | 4 | 4.24 | 8,24 | I | I | 2 | 3,6 | 5,6 | E | |
| | | 4. | M | 4,82 | 3,61 | 8,43 | I | J | 3,82 | 2.83 | 6,65 | E | |
| 5 | G | 1. | H | 4,82 | 1 | 5,82 | G | A | 0 | - | - | - | $A \to E \to J \to G \to D$ |
| | | 2. | D | 7,14 | 0 | 7,14 | G | E | 1 | 3,16 | 4,16 | A | |
| | | 3. | L | 5,73 | 2 | 7,73 | G | I | 2 | 3,6 | 5,6 | E | |
| | | 4. | K | 5,82 | 2.24 | 8,06 | J | J | 3,82 | 2.83 | 6,65 | E | |
| | | 5. | N | 4 | 4.24 | 8,24 | I | G | 4,32 | 1,41 | 5,73 | J | |
| | | 6. | C | 7,32 | 1 | 8,32 | G | | | | | | |
| | | 7. | M | 4,82 | 3,61 | 8,43 | I | | | | | | |
| 6 | H | 1. | D | 6,48 | 0 | 6,48 | G | A | 0 | - | - | - | $A \to E \to J \to G \to H \to D$ |
| | | 2. | L | 5,73 | 2 | 7,73 | G | E | 1 | 3,16 | 4,16 | A | |
| | | 3. | K | 5,82 | 2.24 | 8,06 | J | I | 2 | 3,6 | 5,6 | E | |
| | | 4. | N | 4 | 4.24 | 8,24 | I | J | 3,82 | 2.83 | 6,65 | E | |
| | | 5. | C | 7,32 | 1 | 8,32 | G | G | 4,32 | 1,41 | 5,73 | J | |
| | | 6. | M | 4,82 | 3,61 | 8,43 | I | H | 4,82 | 1 | 5,82 | G | |
| 7 | D | 1. | L | 5,73 | 2 | 7,73 | G | A | 0 | - | - | - | |
| | | 2. | K | 5,82 | 2.24 | 8,06 | J | E | 1 | 3,16 | 4,16 | A | |
| | | 3. | N | 4 | 4.24 | 8,24 | I | I | 2 | 3,6 | 5,6 | E | |
| | | 4. | C | 7,32 | 1 | 8,32 | G | J | 3,82 | 2.83 | 6,65 | E | |
| | | 5. | M | 4,82 | 3,61 | 8,43 | I | G | 4,32 | 1,41 | 5,73 | J | |
| | | | | | | | | H | 4,82 | 1 | 5,82 | G | |
| | | | | | | | | D | 6,48 | 0 | 6,48 | G | |
| 8 | L | 1. | C | 7,32 | 1 | 8,32 | G | A | 0 | - | - | - | |
| 9 | K | | | | | | | E | 1 | 3,16 | 4,16 | A | |
| 10 | N | | | | | | | I | 2 | 3,6 | 5,6 | E | |
| 11 | M | | | | | | | J | 3,82 | 2.83 | 6,65 | E | |
| | | | | | | | | G | 4,32 | 1,41 | 5,73 | J | |
| | | | | | | | | H | 4,82 | 1 | 5,82 | G | |
| | | | | | | | | D | 6,48 | 0 | 6,48 | G | |
| | | | | | | | | L | 5,73 | 2 | 7,73 | G | |
| | | | | | | | | K | 5,82 | 2.24 | 8,06 | J | |
| | | | | | | | | N | 4 | 4.24 | 8,24 | I | |
| | | | | | | | | M | 4,82 | 3,61 | 8,43 | I | |

## 2.4. Implementacja

### 2.4.1. Kod źródłowy

#### 2.4.1.1. Funkcja algorytmu A*

```
1.  public static IList<MapField> AStar(MapField from, MapField to,
2.                                      Func<MapField, MapField, float> heuristic,
3.                                      params MapField[] forbidenFields) {
4.
5.     // ktores z pol nie istnieje
6.     if ( (from == null) || (to == null) ) throw new NullReferenceException();
7.
8.     // pole jest niedostepne
9.     if ( !to.IsAvaliable ) throw new FieldNotAvaliableException();
10.
11.    // pola from i to leza kolo siebie
12.    if ( from.Neighbour.Any(neighbour => neighbour == to) )
13.       return new List<MapField>(2) { from, to, };
14.
15.    //Lita pol do przeszukania
16.    List<PathFindingNode> openList = new List<PathFindingNode> { new PathFindingNode(from) };
17.
18.    //Lista przeszukanych pol
19.    List<PathFindingNode> closeList = new List<PathFindingNode>(
20.                  forbidenFields.Select(forbidenField => new PathFindingNode(forbidenField)));
21.
22.    bool achivewedGoal = false;
23.
24.    //dopoki jakiekolwiek pole moze zostac jeszcze przebadane
25.    while ( openList.Count != 0 ) {
26.       PathFindingNode current = openList.RemoveAtAndGet(0);
27.       closeList.Add(current);
28.
29.       // znaleziono pole do ktorego dazylismy
30.       if ( current.This == to ) achivewedGoal = true;
31.
32.       if ( achivewedGoal ) {
33.          float minValInOpenList = openList.Select(node => node.CostFromStart)
34.                                           .Min();
35.
36.          if ( minValInOpenList < current.CostFromStart ) return ReconstructPath(current);
37.       }
38.
39.       Parallel.ForEach(current.This.Neighbour.Where(neighbour => neighbour.IsAvaliable),
40.                                                      neighbour => {
41.          PathFindingNode neighbourNode = new PathFindingNode(neighbour) {
42.                                             Parent = current,
43.                                             CostFromStart = current.CostFromStart +
44.                                                EuclideanDistance(current.This, neighbour) *
45.                                                current.This.Cost,
46.                                             CostToEnd = heuristic(neighbour, to)
47.                                          };
48.
49.          //Zmiana parametrow sciezki jesli droga do sasiada jest krotsza z obecnego pola
50.          //niz ustalona wczesniej
51.          if ( closeList.Contains(neighbourNode) ) {
52.             PathFindingNode oldNode = closeList[closeList.IndexOf(neighbourNode)];
53.             if ( neighbourNode.CostFromStart < oldNode.CostFromStart ) {
54.                lock ( closeListMutex_ ) closeList.Remove(oldNode);
55.                lock ( openListMutex_ ) openList.Add(neighbourNode);
56.             }
57.          } else if ( openList.Contains(neighbourNode) ) {
58.             PathFindingNode oldNode;
59.             lock ( openListMutex_ ) oldNode = openList[openList.IndexOf(neighbourNode)];
60.             if ( neighbourNode.CostFromStart < oldNode.CostFromStart ) {
61.                oldNode.CostFromStart = neighbourNode.CostFromStart;
62.                oldNode.Parent = neighbourNode.Parent;
```

```
63.             }
64.          } else {
65.             lock ( openListMutex_ ) openList.Add(neighbourNode);
66.          } });
67.
68.       openList.Sort();
69.    }
70.
71.    //jesli niedotarto do zadanego celu
72.    throw new FieldNotAvaliableException();
73. }
```

#### 2.4.1.2. Funkcja rekonstruująca ścieżkę

```
1.  private static IList<MapField> ReconstructPath(PathFindingNode node) {
2.     List<MapField> path = new List<MapField>();
3.
4.     while ( true ) {
5.        path.Add(node.This);
6.        if ( node.Parent == null ) break;
7.        node = node.Parent;
8.     }
9.
10.    path.Reverse();
11.    return path;
12. }
```

#### 2.4.1.3. Klasa reprezentująca węzeł grafu wraz z połączeniami

```
1.  private class PathFindingNode : IComparable<PathFindingNode> {
2.     internal PathFindingNode(MapField This) {
3.        this.This = This;
4.        Parent = null;
5.
6.        CostFromStart = 0f;
7.        CostToEnd = float.MinValue;
8.     }
9.
10.    public MapField This { get; }
11.    public PathFindingNode Parent { get; set; }
12.
13.    public float CostFromStart { get; set; }
14.    public float CostToEnd { get; set; }
15.
16.    public float AllCost { get { return CostFromStart + CostToEnd; }
17. }
```

#### 2.4.1.4. Funkcje heurystyczne

##### 2.4.1.4.1. Heurystyka Euklideoswa

```
1.  public static float EuclideanDistance(MapField from, MapField to) {
2.     return (float)Sqrt((from.MapPosition.X - to.MapPosition.X) *
3.                        (from.MapPosition.X - to.MapPosition.X) +
4.                        (from.MapPosition.Y - to.MapPosition.Y) *
5.                        (from.MapPosition.Y - to.MapPosition.Y));
6.  }
```

##### 2.4.1.4.2. Heurystyka Manhatan

```
1.  public static float ManhattanDistance(MapField from, MapField to) {
2.     return Abs(from.MapPosition.X - to.MapPosition.X) +
3.            Abs(from.MapPosition.Y - to.MapPosition.Y);
4.  }
```

##### 2.4.1.4.3. Heurystyka Null

```
1.  public static float NullDistance(MapField from, MapField to) {
2.     return 0f;
```

```
3. }
```

### 2.4.2. OBJAŚNIENIA

**2.4.3.**

**2.4.4.**

**2.4.5.**

**2.4.6.**

**2.4.7.**

3. Algorytm stada
4. Logika rozmyta
5. Opis wykorzystanych narzędzi
6. Podsumowanie
7. Bibliografia