

**Politechnika Śląska**

**Wydział Automatyki, Elektroniki i Informatyki**

**Kierunek Informatyka**

##### Praca dyplomowa magisterska

###### Nawigacja postaci w grach komputerowych

Autor: Radosław Bigaj

Kierujący pracą: dr Ewa Lach

Gliwice, wrzesień 2013

Spis treści

[1. Wstęp 4](#_Toc367397061)

[1.1. Geneza 4](#_Toc367397062)

[1.2. Cel pracy 4](#_Toc367397063)

[1.3. Przewodnik po pracy 4](#_Toc367397064)

[2. Sztuczna inteligencja 6](#_Toc367397065)

[2.1. Początki Sztucznej Inteligencji 6](#_Toc367397066)

[2.2. Przykłady zastosowań sztucznej inteligencji w grach 8](#_Toc367397067)

[2.3. Techniki symulacji sztucznej inteligencji 18](#_Toc367397068)

[3. Znajdowanie ścieżki 22](#_Toc367397069)

[3.1. Graf odnajdowania ścieżki 23](#_Toc367397070)

[1.1. Definicja grafu 24](#_Toc367397071)

[1.2. Grafy ważone 25](#_Toc367397072)

[1.3. Skierowane grafy ważone 28](#_Toc367397073)

[1.4. Terminologia 29](#_Toc367397074)

[1.5. Reprezentacja 30](#_Toc367397075)

[1.6. Algorytm Dijkstry 31](#_Toc367397076)

[1.6.1. Przedstawienie problemu 32](#_Toc367397077)

[1.6.2. Opis algorytmu 33](#_Toc367397078)

[1.7. Algorytm A\* 38](#_Toc367397079)

[1.7.1. Przedstawienie problemu 38](#_Toc367397080)

[1.7.2. Opis algorytmu 39](#_Toc367397081)

[4. Koncepcja inteligentnego agenta 44](#_Toc367397082)

[4.1. Cele badawcze 44](#_Toc367397083)

[4.2. Charakterystyka środowiska graficznego 44](#_Toc367397084)

[4.3. Charakterystyka silnika Unity3D 45](#_Toc367397085)

[4.4. Wybrana metoda optymalizacyjna 48](#_Toc367397086)

[5. Realizacja 51](#_Toc367397087)

[5.1. Konfiguracja biblioteki 51](#_Toc367397088)

[5.2. Implementacja 54](#_Toc367397089)

[5.3. Przegląd map i wygenerowanych grafów 63](#_Toc367397090)

[5.4. Eksperyment 1 – badanie heurystyk 64](#_Toc367397091)

[5.5. Eksperyment 2 – optymalizacja 79](#_Toc367397092)

[6. Podsumowanie i wnioski 82](#_Toc367397093)

[7. Bibliografia 84](#_Toc367397094)

# Wstęp

## Geneza

Nawigacjach w grach komputerowych jest od wielu lat rozwijana i ulepszana oraz stanowi bardzo ważny element każdej gry komputerowej. Gracz komputerowy obecnie jest wymagającym klientem z punktu widzenia branży gier komputerowych. Rozwiązania dostarczane przez branże muszą działać w sposób zadowalający odbiorcę i dotyczy to każdego elementu. Wolno działający system nawigacji może zniechęcić gracza na tyle skutecznie, że gra przestanie mu się podobać, co w efekcie może doprowadzić do porażki gry na rynku. Dlatego warto badać powszechnie stosowane rozwiązania dotyczące nawigacji oraz szukać nowych rozwiązań mogących przyśpieszyć sam proces nawigowania.

## Cel pracy

Celem niniejszej pracy jest przebadanie algorytmów nawigacji stosowanych w grach komputerowych. Nawigacje w grach komputerowych można utożsamić z problemem znajdowania ścieżki. Wynikiem działania takiego algorytmu jest wygenerowana droga z punktu startowego do celu.

W branży gier komputerowych powszechnie stosowanym algorytmem do znajdowania ścieżki jest algorytm A\* (ang. A star) i to on będzie przedmiotem badań. Jest to algorytm heurystyczny, posiadający duże możliwości optymalizacyjne. W pracy zostanie zaimplementowana przykładowa optymalizacja algorytmu A\*.

Na potrzeby badań zostanie zaprojektowane kilka trójwymiarowych map, na których będą odbywać się eksperymenty.

## Przewodnik po pracy

Rozdział drugi zawiera wprowadzenie w zagadnienia związane ze Sztuczną Inteligencją w grach komputerowych. Zawarta w nim jest krótka historia odnosząca się do Sztucznej Inteligencji w grach komputerowych. W rozdziale tym zostały przedstawione również produkcje będące kamieniami milowymi w rozwoju Sztucznej Inteligencji w grach. W rozdziale zawarto również przykładowe techniki symulacji Sztucznej Inteligencji powszechnie stosowane w grach komputerowych.

W rozdziale trzecim zostaje wprowadzone i omówione pojęcie grafu rozumiane przez programistę grafiki komputerowej. Następnie dokonany zostaje opis algorytmów związanych z odnajdywaniem ścieżki.

Rozdział czwarty zawiera postawione cele badawcze, charakterystykę wybranych narzędzi oraz opis wybranej metody optymalizacyjnej.

Piąty rozdział zawiera opis konfiguracji biblioteki użytej do realizacji celu badawczego. W tym rozdziale znajduje się również opis implementacji oraz przegląd zaprojektowanych map. Rozdział ten zawiera również dwa główne eksperymenty, w których dokonano badań efektywności algorytmu A\* ze względu na różne czynniki oraz jego optymalizacji.

Rozdział szósty zawiera wnioski i podsumowanie pracy.

# Sztuczna inteligencja w grach komputerowych

Sztuczna Inteligencja AI (ang. *Artificial Intelligence* ) jest istotnym elementem każdej gry wideo. Obszar zagadnień związanych ze Sztuczną Inteligencją w grach istnieje, właściwie odkąd pojawiły pierwsze gry wideo (rok 1970), jednak same algorytmy pojawiły się już kilkanaście lat wcześniej. Sztuczną Inteligencje w grach nie można utożsamiać ze Sztuczną Inteligencją znaną z badań naukowych. W grach Sztuczna Inteligencja ma na celu stosowanie pewnych trików, których celem jest jedynie symulacja inteligencji. Główne zastosowania AI to [wsk]:

* Podniesienia realizmu świata gry. Stosowana głównie w grach typu cRPG (ang. *Computer Role Playing Games*). Ma za zadanie sterować zachowaniem agentów, z którymi zetknie się bohater gracza.
* Wsparcie podczas walki. Jest to najczęściej spotykana kategoria Sztucznej Inteligencji w grach komputerowych. Stosowana powszechnie w grach strategicznych oraz grach akcji. Sztuczna Inteligencja ma na celu sterowanie agentami podczas walki.
* Relacjonowanie wydarzeń. Stosowane w grach sportowych. Sztuczna inteligencja pełni funkcje związane z trafnym komentowaniem zdarzeń zachodzących w świecie gry na podstawie bieżących działań gracza.
* I wielu innych.

Obszar działania Sztucznej Inteligencji nie kończy się jednak tylko na symulowaniu inteligentnych zachowań, ale może również nadać agentom cechy ludzkie. Po implementacji takiej funkcjonalności w grze może sprawić, że osiągnie ona duży sukces. Efekt taki można uzyskać przenosząc do wirtualnego świata ludzkie niedoskonałości oraz tworząc sposób porozumiewania zawierający nieliniowe dialogi czy duży zasobów słów postaci.

## Początki Sztucznej Inteligencji w grach

Pierwsza faza rozwoju gier komputerowych poświęcała najwięcej czasu wyświetlaniu grafiki, oczywistym tego powodem były ograniczenia czasu pracy procesora, a to grafika w większości gier robiła największa wrażenie. Takie podejście skutkowało tym, że sztuczna inteligencja została zepchnięta na dalszy plan. Przykładowe implementacje zawierały sztywno zakodowane schematy zachowania oraz proste maszyny stanów. W dzisiejszych czasach większość operacji związanej z przetwarzaniem grafiki odbywa się w układach graficznych komputerów GPU ( ang. *graphic processing unit*), a wzrost jakości wyświetlanej grafiki nie przyciąga graczy, którzy wymagają czegoś więcej od gier. Dlatego producenci gier, aby spełnić żądania graczy kładą większy nacisk na rozwój Sztucznej Inteligencji [wsk].

Gra "Tennis for Two" jest jedną z pierwszych gier wideo. Jest to symulacja tenisa ziemnego, w której obraz jest wyświetlany za pomocą oscyloskopu. Została stworzona przez Williama Higinbothama w 1958 roku [wsk]. Pierwszą grą wideo stworzoną specjalne na komputer osobisty było "Space War". Gra została napisana przez S. Russel'a na minikomputer w 1962 roku. Obydwie te gry łączyło to, że wymagały dwóch graczy do rozgrywki. Dopiero w latach siedemdziesiątych zaczęto stosować, pewne proste ustalone schematy odpowiadające za poruszanie się przeciwników, co można traktować jako początki Sztucznej Inteligencji w grach.

Pojęcie przeciwnika komputerowego często w grafice komputerowej określane jest mianem agenta lub agenta komputerowego.

Pierwszą grą, w której gracz posiadał przeciwników był "Pac-Man" wydany w 1979 roku. Agenci komputerowi sprawiali wrażenie inteligentnych - podczas pościgu za postacią gracza, na każdym z rozwidleń dróg agenci mieli różne szanse wyboru losowej drogi lub pogoni za graczem. W efekcie gracz miał odczucie, że komputerowi agenci współpracują ze sobą. Pac-Man zawierał implementacje prostej maszyny stanów, gdzie każdy z czterech agentów mógł gonić lub uciekać przed graczem w labiryncie. Pierwsze gry wideo bazowały na prostych lub bardziej złożonych wzorcach, jak w klasycznych grach "Golden Axe" (1987 rok) czy "Super Mario Brothers" (1985 rok), gdzie przeciwnicy zwykle poruszali się w jednym lub dwóch kierunkach, aż do napotkania gracza[wsk].

Pierwszą grą akcji posiadającą bardziej rozbudowaną Sztuczną Inteligencje jest "Goldeneye 007" (1997 rok). Pozwalała ona reagować agentom odpowiednio na ruch oraz akcję gracza. Komputerowi agenci posiadali zmysł wzroku i byli w stanie zauważyć czy pozostali agenci są martwi. Natomiast w grze "Thief: The Dark Project" (1998 rok) rozgrywka opierała się w znaczniej mierze na symulacji zmysłów wzroku i słuchu. Zostanie ona omówiona dokładniej w kolejnym rozdziale [wsk].

W latach 2001 i 2002 powstały dwie gry, które sprawiły, że gracze z niedowierzaniem patrzyli na poziom Sztucznej Inteligencji. Pierwsza z tych gier to "The Sims" ze studia Maxis, gdzie Sztuczna Inteligencja zajmowała się modelowaniem ludzkich emocji oraz potrzeb, przez co można powiedzieć, że gra była symulatorem życia. Drugą z gier jest "Black and White" ze studia Lionhead Studios. W tej grze komputerowej zachowanie agenta jest sterowane siecią neuronową przez, co może uczyć się w sztucznie stworzonym środowisku. Obecnie jednak większość gier wykorzystuje tylko proste techniki. Powszechnie stosowaną techniką są maszyny stanów oraz jej pochodne.

## Przykłady zastosowań sztucznej inteligencji w grach

W tym rozdziale zostanie przedstawione i opisane kilka przełomowych gier, które dzięki wykorzystaniu Sztucznej Inteligencji odniosły sukces branżowy i stały się rozpoznawalnymi markami, a co więcej niektóre z przedstawionych tytułów są do dzisiaj rozwijane. Poniżej przedstawione gry znalazły się w rankingu serwisu "AiGameDev" (strona: http://aigamedev.com/), w kategorii najbardziej innowacyjnych gier w historii. Dzięki osiągniętemu sukcesowi zapoczątkowały całą serię kolejnych wydań i kontynuacji, jest to jeden z kilku powodów, dla których warto się im przyjrzeć. Przedstawione tutaj gry są swego rodzaju pionierami w swojej klasie. Przyszło im się zmagać z wysokimi wymaganiami przed jakimi stawiał ich silnik Sztucznej Inteligencji, co więcej udało się im te wymagania spełnić, dzięki czemu poniższe tytuły odniosły sukces. Zostanie teraz przedstawione kilka gier z wyżej opisanego rankingu, wybranych na podstawie dostępności materiałów opisujących implementacje Sztucznej Inteligencji.

**Thief (rok 1999)**

Nowatorskie koncepcje wykorzystane w grze:

* Moduł odpowiedzialny za sztuczną inteligencje zbudowany jest z całego systemu czujników, dzięki któremu agenci mogą realistycznie reagować na bodźce świetlne i dźwiękowe.
* Agenci znajdujący się pod kontrolą sztucznej inteligencji korzystają z specjalnych nagrań audio, w celu oznajmienia swojego obecnego stanu. Pozwala to graczowi na zorientowanie się w jakieś sytuacji się znajduje.

**The Sims (rok 2000)**

Nowatorskie koncepcje wykorzystane w grze [wsk]:

* Zamodelowanie wirtualnej emocjonalnej więzi między agentami, dzięki czemu jest możliwe tworzenie związków między nimi.
* Każdy agent w grze ma swój zdefiniowany charakter, umiejętności, podstawowe potrzeby emocjonalne oraz fizyczne, mające wpływ na jego poczynania w grze. Emocje postaci są mierzone w zakresie (-100, 100), a następnie są mapowane do wyjściowej formy szczęścia/nastroju.
* W grze zastosowano inteligentne obiekty, co okazało się pomocne w implementacji zachowań. To obiekt definiuje, jak agent może wejść z nim w interakcje.

**Halo (rok 2001)**

Nowatorskie koncepcje wykorzystane w grze [wsk]:

* Inteligentni agenci potrafiący się kryć przed ostrzałem oraz używać rozważenie dostępnej broni,
* Wykorzystanie drzewa zachowań, które zostało bardzo dobrze przyjęte przez przemysł gier komputerowych,
* Sytuacja na polu bitwy ma wpływ na zachowanie jednostek.

**F.E.A.R. (rok 2005)**

Nowatorskie koncepcje wykorzystane w grze [wsk]:

* Pierwszy raz użyto systemu planowania zadań do generowanie zachowań zależnych od sytuacji.
* Agenci komputerowi sprawnie wykorzystują świat gry tak, aby zwiększyć jego realizm. Otwierają drzwi, znajdują osłony, przechodzą przez okna,
* Dokonano implementacji taktycznych technik walki - atak z flanki, przerywane serie ostrzału.

**Black & White (rok 2001)**

Nowatorskie koncepcje wykorzystane w grze:

* Zastosowanie technik, takich jak rozbudowane drzewa decyzyjne czy sztuczne sieci neuronowe,
* Silnik korzystający z architektury BDI (Przekonanie-Pragnienie-Zamiar ang. *Belief-Desire-Intention*),
* Gra symuluje nie tylko zachowanie stworzenia, ale też życie mieszkańców pracujących tak, aby rozwijać zamieszkiwaną wioskę.

## Techniki symulacji sztucznej inteligencji

Wyróżnia się wiele technik stosowanych do symulacji sztucznej inteligencji w grach komputerowych. Poniżej zostaną omówione najpopularniejsze z nich. W grach bardzo często najprostsze rozwiązania okazują się najlepszymi, dzięki czemu techniki takie jak: maszyny stanów, heurystyczne poszukiwanie drogi czy drzewa decyzyjne zyskały sobie dużą popularność.

**Maszyny stanów.**

Technika maszyny stanów była wykorzystywana w grach już w latach 90. do kontrolowania agentów komputerowych. Maszyny stanów stały się tak popularne i użyteczne, że są stosowane do zarządzania AI, również w najnowszych wysokobudżetowych produkcjach. Wykorzystuje się je też w komputerowych grach fabularnych cRPG(ang. *computer Role Playing Game*) do sterowania dialogami gracza z agentami. Co więcej zarządzają obiektami w grze, przechowują stan rozgrywki (np. zwycięstwo, porażka, wykonane zadanie, postać dotarła do punktu docelowego), przetwarzają komendy gracza oraz zarządzają światem gry.

Maszyna stanów zbudowana jest z pewnej ściśle określonej liczby stanów znajdujących w danej puli rozwiązań. Kolejno zostają przechwycone pewne zdarzenia, które zmieniają stan maszyny. Dzięki temu istnieje możliwość podjęcia jednego lub kilku działań w zależności od stanu w jakim się aktualnie znajduje obiekt gry.

**Heurystyczne poszukiwanie drogi.**

Jednym z problemów jaki rozwiązuje Sztuczna Inteligencja w grach jest określenie najlepszej drogi z punktu A do punktu B na terenie rozgrywki. Technika ta stosowana jest także do rozwiązywania zagadnień bardzo skomplikowanych i złożonych takich jak poruszanie się jednostek w formacjach czy planowanie strategiczne. Rozwiązaniem jakie stosuje się najczęściej dla problemów tego typu jest heurystyczny algorytm A\*. Algorytm ten podczas procesu określania drogi do celu nie szuka jej "na ślepo", tylko szacuje jej najbardziej prawdopodobny kierunek odrzucając inne mniej sensowne ścieżki.

Celem algorytm A\* jest minimalizacja obszaru poszukiwań najlepszej trasy, dzięki ustaleniu pewnego kierunku, który zawęża obszar rozważanych tras. Technika ta oblicza koszt dotarcia do punktu na mapie i dodaje do niego heurystykę określającą przewidywane koszty dotarcia do celu. Heurystyka jest liczona zwykle jako odległość od obecnego punktu do celu ignorując wszelkie przeszkody i ograniczenia umieszczone na mapie. W skrócie A\* sprawdza, po każdym wykonanym ruchu agenta wszystkie możliwe kierunki dalszej trasy i ponownie wybiera możliwe kierunki trasy o jak najniższym koszcie. W momencie, gdy rozważane położenie jest celem, algorytm kończy swoje działanie. W przeciwnym przypadku algorytm przechowuje przyległe położenie, tak aby w przyszłości móc rozważyć inne ścieżki.

**Drzewa decyzyjne.**

Drzewa decyzyjne zostały zaprojektowane z myślą o rozwiązywaniu problemów decyzyjnych oraz tworzeniu planu działania. Są powszechnie stosowane w grach takich jak: szachy czy warcaby.

Drzewo decyzyjne jest reprezentowane w postaci grafu decyzji i możliwych konsekwencji. Węzły odzwierciedlają stan gry, natomiast potomkowie to stany uzyskane po przeprowadzeniu ruchu. Każda ścieżka w takim drzewie kończy się węzłem końcowym. Agent komputerowy dokonuje analizy drzewa rozpatrując tym samym wszystkie możliwe posunięcia dostępne w aktualnej sytuacji. Następnie wybiera te, które uznał za najbardziej korzystne.

**Logika rozmyta.**

Pojęcie logiki rozmytej ma związek z teorią prawdopodobieństwa oraz teorią zbiorów rozmytych. Logika rozmyta jest jedną z logik wielowartościowych posiadającą stany pośrednie pomiędzy wartościami logicznymi (prawda, fałsz) , które określają przynależność do odpowiedniego zbioru. Daje to możliwość rozważenia odpowiedzi na pytanie "jak bardzo?", "ile?" przykładowo: "legion", "wataha", "grupa", "sporo", "kilka". Logia rozmyta wykorzystywana jest często do odwzorowania emocji w grach (np. "przyjacielski", "obojętny", "wrogi"). Za pomocą można modelować sferę uczuciową agentów komputerowych, co poprawia realizm gry.

**Sztuczne sieci neuronowe.**

Sztuczne sieci neuronowe zostały zaprojektowane, aby działać podobnie jak sieci neuronowe w mózgu człowieka. Przetwarzają one sygnały oraz wykonują obliczenia za pomocą neuronów - są to elementy, które wykonują pewne operacje na wejściu. Za pomocą sieci neuronowej agent może się uczyć wraz z postępem gry. Natrafiając na nowy rodzaj sytuacji, dostosowuje się do niej korzystając ze zdobytego doświadczenia.

Sieci neuronowe od wielu lat zamierzano przystosować do tworzenia Sztucznej Inteligencji w grach komputerowych. W 2000 roku swoją premierę miała gra Colin McRae Rally 2.0 - gra będąca symulatorem wyścigów. Gra ta zawierała implementacje sieci neuronowej. Za dane wejściowe przyjmowała ona parametry opisujące trasę jaką miał przejechać agent przykładowo: krzywizna łuku drogi, rodzaj gruntu, parametry techniczne pojazdu, stan pogody. Zadaniem tej sieci było wygenerowanie odpowiednich danych wyjściowych bazując na parametrach wejściowych tak, aby samochód kierowany przez agenta mógł bez problemu przejechać trasę wyścigu.

Z siecią neuronową związane są również pewne problemy. Sieć neuronową do gry można dodać na dwa sposoby. Pierwszym z nich jest dodanie wyuczonej sieci, która została w jakiś sposób przebadana i zwraca sensowe wyniki. Natomiast drugim sposobem jest dodanie niewyuczonej sieci neuronowej. Taka sieć będzie się uczyć dopiero podczas rozgrywki z graczem. Jest to o tyle niebezpieczne, że nie ma kontroli na wynikami zwracanymi przez tą sieć, przez co wyniki mogą być nieprzewidywalne.

**Algorytm stadny.**

W 1987 roku Craig Raynolds przedstawił artykuł, w którym opracował 3 zasady, które w połączeniu umożliwiały grupie agentów realistyczne zbiorowe zachowanie podobne do zachowań stadnych znanych ze świata zwierząt np. ławic ryb, stad ptaków. Raynolds określił te trzy zasady jako sterownie zachowaniem. Prezentują się one następująco:

* spójność - sterownie, którego celem jest zbieranie agentów znajdujących się blisko siebie w odpowiednie grupy lokalne,
* wyrównywanie - rodzaj sterowania dzięki któremu agent może dostosowywać kierunek i prędkość do innych agentów przebywających w pobliżu,
* rozdzielczość - sterownie, w które zapobiega tworzeniu się tłumu w jednym miejscu. Agenci zachowują pewną odległość względem siebie.

Reynolds opracował jeszcze czwartą zasadę, określaną mianem unikania. Jest ona stosowana, aby wirtualni agenci unikali przeszkody umieszczonej na mapie.

W każdym cyklu procesu przemieszczania się agenci każdorazowo sprawdzają środowisko, w jakim w danej chwili przebywają i to jest jedyna informacja jakiej wymaga ten algorytm. Powoduje to, że zmniejszenie wymagań związanych z pamięcią przy sterowaniu wieloma agentami oraz pozwala na szybką reakcję na ewentualną zmianę sytuacji. Podsumowując algorytm ten umożliwia nadanie grupie poruszających się agentów dynamiki ruchu jak jedno ciało oraz zdolności omijania przeszkód czy wrogich postaci.

**Dalszy rozwój sztucznej inteligencji w grach komputerowych**

Wraz z upływem czasu rola sztucznej inteligencji w grach zwiększyła się o analizowanie gry i jej dostosowanie do poziomu gracza. Powoduje ona rozwój świata wraz ze wzrostem doświadczenia gracza, które uzyskał w dotychczasowym procesie rozgrywki.

Istnieją pewne grupy badawcze, zajmujące się projektami gier, w których cały świat ma być kontrolowany przez Sztuczną Inteligencję. Gracze wchodzący do gry mają czuć się, tak jakby wchodzili do realnej rzeczywistości, gdzie każde ich działanie ma wpływ na dalszy przebieg gry.

Analizując obecny stan Sztucznej Inteligencji, w którym twórcy rozwijają każdy element świata, tak aby wydawał się jak najbardziej realny, można pokusić się o stwierdzenie, że w przyszłości Sztuczna Inteligencja będzie odpowiedzialna za odwzorowanie zachowań gracza jako człowieka w świecie gry. Dzięki temu gracz będzie musiał bardziej zaangażować się w grę.

# Znajdowanie ścieżki

Postacie w grach komputerowych muszą się poruszać w wirtualnym świecie. Czasami zdarza się, że ten ruch jest na stałe ustawiony przez programistę. Przykładowo strażnik patrolujący drogę, porusza się ślepo po ogrodzonym terenie. Stałe trasy są łatwe do implementacji i wdrożenia. Niestety bardzo łatwo można spowodować, że obiekt zostanie przesunięty przez inny obiekt (wejdzie z nim w kolizje), co spowoduje, że wypadnie z trasy. Pozwolenie postaci na pewną dowolność w przemieszczaniu może spowodować, że jej wędrówki będą bezcelowe, co więcej postać będzie mogła łatwo utknąć. Bardziej zaawansowane postacie nie wiedzą z góry, gdzie będą musiały się przemieścić. Jednostka wykorzystywana w strategii czasu rzeczywistego może zostać przypisana do dowolnego punktu na mapie przez gracza w dowolnym momencie czasu, patrolujący strażnik może potrzebować przemieścić się do najbliższego punktu alarmowego, żeby wezwać wsparcie, a w grach platformowych może być wymagane, żeby przeciwnicy gonili gracza do przepaści używając dostępnych obszarów terenu.

Dla każdej z tych postaci musi zostać obliczona odpowiednia droga, żeby dostać się tam gdzie jest jej cel. Konieczne jest utworzenie sensownej trasy w jak najkrótszym czasie.

To właśnie jest istotą odnajdywania ścieżki (ang. pathfinding), czasami nazywane także planowaniem ścieżki - znajduję się każdym w silniku gry posiadającym moduł odpowiedzialny za sztuczną inteligencję. W przedstawionym poniżej modelu (Rysunek 1) rola odnajdywania ścieżki znajduje się pomiędzy modułami odpowiedzialnymi za podejmowanie decyzji oraz poruszanie się postaci. Często odnajdywanie ścieżki jest po prostu używane do wykonania wstępnej analizy gdzie postać ma się przesunąć, aby dotrzeć do celu. Sam cel jest wyznaczany przez inną część modułu sztucznej inteligencji, więc można podsumować, że odnajdywanie ścieżki oblicza nam tylko jak dostać się do celu. Do uzyskania pożądanego efektu trzeba zbudować system przemieszczania w taki sposób, aby był wywoływany, kiedy jest potrzeba zaplanowania drogi. Zostanie on omówiony w kolejnych rozdziałach.

Moduł odnajdywania ścieżki może, również zostać umieszczony na siedzeniu kierowcy i zarządzać podejmowaniem decyzji, gdzie się trzeba przemieścić, a także w jaki sposób się dostać do celu. Jest to pewnego rodzaju odmiana modułu odnajdywania ścieżki zwana odnajdywaniem ścieżki otwartego celu (ang. *open goal pathfinding*), może być użyta do pracy zarówna nad ścieżką jaki i miejscem przeznaczenia.



Ilustracja Model sztucznej inteligencji

Większość gier używa funkcjonalności znajdowania ścieżki wykorzystując algorytm A\*(ang. *A star*). Pomimo, że algorytm jest efektywny i łatwy do wdrożenia, to nie może on pracować bezpośrednio na danych zaczerpniętych z poziomu (mapy) gry. Wymaga to, aby poziom gry był reprezentowany przez odpowiednią strukturę danych - jest to zwykle graf ważony o nieujemnych wagach. Ten rozdział wprowadza pojęcie struktury danych grafu, kolejno zostanie omówiony algorytm Dijkstry, pomimo że jest on częściej stosowany w procesie podejmowania decyzji taktycznych niż w procesie odnajdowania ścieżki. Ponieważ struktura danych grafu nie jest jedynym sposobem w jaki większość gier reprezentuje swoje dane, dlatego warto się przyjrzeć się kwestii zmiany geometrii mapy na dane, które mogą zostać przetworzone przez moduł odpowiedzialny na znajdowanie ścieżki. Warto też wspomnieć o wielu dziesiątkach przydatnych wariacji podstawowego algorytmu A\*.

Po raz pierwszy w powyższym akapicie wspomniano o efektywności. Będzie ona rozumiana jako efektywność czasowa algorytmu.

## Graf odnajdowania ścieżki

Algorytm A\* oraz Dijkstra nie może pracować bezpośrednio na geometrii, z której zbudowana jest mapa. Jednak każda mapa może zostać poddana pewnemu procesowi w wyniku, którego otrzymamy jej uproszczoną wersje w postaci grafu. Jeśli proces upraszczania geometrii do grafu wykona się prawidłowo, to plan ścieżki zwrócony przez moduł odpowiedzialny za odnajdowanie ścieżki może zostać użyty oraz przetłumaczony ponownie na warunki gry. Z drugiej strony sam proces upraszczania może pozbawić pewnych informacji, które mogą się okazać znaczące. Złe uproszczenie może oznaczać, że ostateczna trasa nie jest za dobra.

Algorytmy stosowane do odnajdywania ścieżki używają zwykle struktur zwanych skierowanymi grafami o nieujemnych wagach. Dla uproszczenie opisu całego grafu odnajdywania ścieżki jest on wyrażony przez proste struktury grafowe.

## Definicja grafu

Graf jest matematyczną strukturą często reprezentowaną przez schemat graficzny. Graf składa się z dwóch rodzajów elementów. Są to węzły często rysowane jako punkty lub koła w schemacie grafu oraz krawędzie będące połączeniami węzłów przedstawiane w postaci linii. *Ilustracja 8*  przedstawia strukturę grafu.

Formalnie graf składa się ze zbioru węzłów i zestawu połączeń, w którym połączenie jest po prostu nieuporządkowaną parą węzłów.

Każdy węzeł stanowi zwykle pewien region poziomu gry, taki jak pokój, piwnica czy schody lub mały region miejsca na zewnątrz. Połączenia pokazują, które miejsca są połączone. Jeśli pokój sąsiaduje ze schodami, to węzeł reprezentujący pokój będzie miał połączenie z węzłem reprezentującym schody. W ten sposób cały poziom gry jest podzielony na obszary, które są ze sobą połączone. Aby dostać się z jednego miejsca na danym poziomie do drugiego możemy korzystać z połączeń. Jeśli jest możliwość przejścia bezpośrednio z węzła startowego do celu, to problem jest trywialny. W przeciwnym razie możemy użyć połączeń do podróży przez węzły pośrednie znajdujące się na ścieżce.

Droga przez graf składa się z zera lub więcej połączeń. Jeśli początek i koniec są takie same, to nie ma połączenia w ścieżce. Jeśli węzły są połączone, wówczas tylko jedno połączenie jest potrzebne.



Ilustracja Graf

## Grafy ważone

Ważony graf składa się z węzłów i połączeń podobnie jest zwykły graf. Dodatkowo posiada wartość liczbową dla każdego połączenia węzłów. W matematycznej teorii grafów jest to nazywane wagą, natomiast w zastosowaniach stosowanych w grach powierzchnie jest określany mianem kosztu (chociaż graf nadal jest nazywany grafem ważonym, a nie grafem kosztu). Na *ilustracji 9* przedstawiającej graf z każdym połączeniem jest związana wartość kosztu.

Koszty w module zarządzającym odnajdowaniem ścieżki są zwykle reprezentowane czasem lub odległością. Jeśli węzeł reprezentujący platformę jest położony w dużej odległości od węzła reprezentującego następną platformę, to koszt takiego połączenia będzie duży. Podobnie będzie wyglądało to w przypadku przemieszczania się pomiędzy dwoma pokojami, które są pokryte pułapkami - taka podróż będzie trwała długo przez co koszt będzie duży. Koszty w grafie mogą reprezentować więcej niż tylko czas i odległość. Istnieje duża liczba aplikacji z odnajdywaniem ścieżki, w których koszt stanowi kombinacje czasu, odległości i innych współczynników.



Ilustracja Graf posiadający koszty

Na całej trasie przez graf, od węzła początkowego do węzła docelowego, możemy obliczyć całkowity koszt ścieżki. Jest to po prostu suma kosztów każdego połączenia na trasie. Na *ilustracji 10* zaprezentowano przykładowy graf ważony, na którym zostanie policzony koszt przykładowej trasy. Zakładając trasę z punktu A do punktu C przez węzeł B. Koszt jest obliczany następująco: z A do B wynosi 4, następnie z B do C ma wartość 5 to całkowity koszt drogi będzie wynosił 9.



Ilustracja Graf z rozważaną drogą A-B-C

**Reprezentacja punktów w regionie**

Można od razu zauważyć, że jeśli dwa regiony są ze sobą połączone (np. pokój i schody), to odległość pomiędzy nimi będzie wynosić zero. Jeśli gracz stoi w drzwiach, a następnie przemieszcza się do schodów natychmiastowo. Nasuwa się więc pytanie czy zatem wszystkie połączenie mają koszt równy zero? Istnieje tendencja do pomiaru połączenia ze względu na odległości czy czas przez punkt reprezentatywny znajdujący sie w każdym regionie. Zatem dla przykładu punkt reprezentatywny dla pokoju będzie się w jego środku, a dla schodów w ich centrum. Jeśli pokój jest duży a schody są długie to jest całkiem prawdopodobne że odległość między punktami reprezentatywnymi będzie duża, co za tym idzie koszt również będzie duży. Często można spotkać diagramy grafów odnajdywania ścieżek. Przykładem jest Ilustracja 11, gdzie punkt reprezentatywny jest przyporządkowany do każdego regionu.



Ilustracja Węzły grafu określające regiony

**Ograniczenia odnośnie wag w grafie.**

Posiadanie przez krawędź ujemnego kosztu, może wydawać się całkowicie nieuzasadnione. Nie można mieć ujemnego dystansu między węzłami oraz nie można mieć ujemnego czasu, aby dostać się do danego miejsca. Mimo to matematyczna teoria grafów dopuszcza ujemne wagi i mają one bezpośrednie zastosowanie, w niektórych praktycznych problemach. Jednak problemy te znajdują się całkowicie poza zasięgiem świata gier komputerowych i nie będę omawiane na łamach tej pracy. Pisanie algorytmów, które mogą pracować z ujemnymi wagami jest zazwyczaj bardziej złożone, niż dla tych, które mają sztywne wymogi stosowania nieujemnych wag. W szczególności, Dijkstra i algorytmy A\* powinny być stosowane wyłącznie z nieujemnymi wagami. Istnieje taka konstrukcja grafu posiadającego ujemne wagi, że algorytm zajmujący się procesem odnajdywania ścieżki zwróci rozsądny wynik. Jednak w większości przypadków, Dijkstra i A\* wejdzie w pętlę nieskończoną. I nie jest to błąd algorytmów. Z punktu widzenia matematycznego nie ma czegoś takiego jak najkrótsza ścieżka przez wiele grafów w ujemnymi wagami - takie rozwiązanie po prostu nie istnieje. Używając terminu "koszt" w tej pracy brana jest pod uwaga wyłącznie nieujemna waga. Kosz jest zawsze liczbą dodatnią. Twórcy gier wspólnie przyznają, że nigdy nie stosowali ujemnych wag ani algorytmów do nich przystosowanych w procesie tworzenia gier komputerowych.

## Skierowane grafy ważone

W wielu sytuacjach ważony graf wystarczy do reprezentacji poziomu gry i często zdarzają się implementacje grafu w takiej formie. Można jednak pójść o krok dalej. Główne algorytmy służące do odnajdywania ścieżki obsługują bardziej złożone formy grafów, takie jak graf skierowany (Ilustracja 12). Jest on często używany przez programistów gier komputerowych. Możliwe jest, aby poruszać się pomiędzy węzłem A i węzłem B (przykładowo pokój i schody), to możliwe jest też, aby przejść z węzła B do węzła A. Połączenia są dostępne w obie strony, a koszt przejścia jest taki sam w obu kierunkach. Skierowany graf zakłada, że połączenia są dostępne tylko w jednym kierunku. Jeśli postać gracza może dotrzeć z węzła A do węzła B i odwrotnie to będzie to reprezentowane na grafie jako dwa połączenia: jedno z A do B i drugie z B do A. Jest to przydatne w wielu sytuacjach. Po pierwsze, nie jest zawsze tak, że możliwość przejścia z punktu A do B oznacza że w drugą stronę jest to osiągalne. Jeśli węzeł oznacza podłogę na piętrze, a B reprezentują podłogę magazynu znajdującego się pod pokojem, to postać może łatwo spaść z A do B, ale nie będzie w stanie wrócić ponownie.



Ilustracja Graf skierowany ważony

Po drugie posiadając dwa połączenia w różnych kierunkach, oznacza to, że mogą istnieć dwa różne koszty. Za przykład można ponownie podać pokój na piętrze oraz magazyn znajdujący się pod nim, z tym, że do naszego świata dodajemy drabinę. Zastanawiając się o kosztach w kategoriach czasu skok z pokoju do magazynu nie zajmuje w ogóle czasu, ale może upłynąć kilka sekund zanim postać gracza zdąży się wspiąć z powrotem na górę po drabinie. Ponieważ koszty są związane z każdym połączeniem mogą być po prostu reprezentowane: połączenie A(piętro) z B(magazyn) ma mały koszt, a połączenie z B do A ma większy koszt. Matematycznie graf skierowanie jest identyczny jak graf nieskierowany z wyjątkiem pary węzłów, które stanowią połączenie jest teraz uporządkowane. Podczas gdy połączenie [węzeł A, węzeł B, koszt] w nieskierowanym grafie jest identyczne do połączenia [węzeł B, węzeł A, koszt] (tak długo jak koszty są identyczne ) w grafie skierowanych są one po prostu innymi połączeniami.

## Terminologia

Terminologia dotycząca gafów jest zmienna. W literaturze matematycznej często można natknąć się na określenie wierzchołek niż węzeł oraz krawędź niż połączenie (oczywiście zamiast pojęcia wagi stosowany jest koszt jak już było wspominane). Wielu programistów, którzy aktywnie angażują się w badania nad Sztuczną Inteligencją stosuje terminologie naruszającą matematyczną literaturę. Może to okazać się mylące w kontekście procesu tworzenia gier, ponieważ pojęcie wierzchołka ma zupełnie inne znaczenie. Niestety nie ma spójnej terminologii dotyczącej grafów w kontekście znajdowania ścieżki. Można spotkać w różnego typu artykułach oraz na konferencjach poświęconych programowaniu gier określenie "kropka" na węzeł, natomiast łuk, ścieżka, łącze i linia są określeniami dla połączenia. Na potrzeby tej pracy będzie stosowana terminologia węzłów i połączeń, ponieważ jest to powszechne, dość sensowne (w przeciwieństwo do kropek i linii) oraz jednoznaczne (łuki i wierzchołki mają znaczenie zarówno w grafice komputerowej). Ponadto, podczas omawiania skierowanych grafów o nieujemnych wagach prawie każda pozycja literaturowa dotycząca zagadnienia odnajdowania ścieżki nazywa te grafy właśnie w taki sposób .

## Reprezentacja

Reprezentacja grafu musi być przedstawiona w taki sposób, żeby algorytmy odnajdywania ścieżki takie jakie A\* czy Dijkstra mogły pracować na nich. Jak można się przekonać algorytmy muszą zdobyć informacje o połączeniach wychodzących z danego węzła. I dla każdego z tych połączeń muszą one mieć dostęp kosztu i punktu przeznaczenia.

Można dokonać reprezentacji grafu za pomocą następującego interfejsu:

interface Graph

{

/\*Zwraca tablice połączeń (interfejs Connection) wychodzących z podanego jako parametr węzła \*/

Connection [] getConnetions(Node fromNode);

}

interface Connection

{

/\* Zwraca nieujemny koszt połączenia \*/

uint getCost();

/\* Zwraca węzeł z które to połączenie pochodzi \*/

Node getFromNode();

/\* Zwraca węzeł do którego to połączenie prowadzi \*/

Node getToNode();

}

Interfejs Graph będzie zwracał tablicę obiektów połączeń dla każdego węzła, który został o to zapytany. Za pomocą tych obiektów można pobrać koszt oraz węzeł końcowy. Prosta implementacja takiej klasy będzie przechowywać połączenia dla każdego węzła oraz dzięki niej można w prosty sposób zwrócić listę tych połączeń.

Bardziej zaawansowana implementacja może liczyć koszt tylko wtedy, kiedy jest to wymagane, korzystając z obecnej informacji na temat struktury poziomu. Należy zauważyć, że nie ma określonego typu danych dla węzła w tym interfejsie, ponieważ nie ma potrzeby specyfikowania go. W wielu przypadkach wystarczy tylko nadać węzłowi unikalny numer i użyć liczb całkowitych jako typ danych. Jest to bardzo dobra implementacja, ponieważ otwiera pewne specyficzne i bardzo szybkie możliwości optymalizacyjne algorytmu A\*.

## Algorytm Dijkstry

Nazwa algorytm Dijkstry wzięła sie od holenderskiego matematyka Edsger'a Dijkstrty, który jest jego twórcą. Algorytm Dijkstry nie był pierwotnie zaprojektowany do odnajdywania ścieżki w rozumieniu gier komputerowych. Został on zaprojektowany, aby rozwiązać problem matematycznej teorii grafów, łudząco nazywany "najkrótsza droga".

Gdzie odnajdywanie ścieżki w grach ma jeden punkt początkowy i jeden punkt docelowy, natomiast algorytm znajdujący najkrótszą ścieżkę jest zaprojektowany tak, aby znaleźć najkrótsze trasy do wszystkich węzłów od punktu startowego. Rozwiązanie tego problemu zawiera w sobie rozwiązanie problemu odnajdywania ścieżki w grach komputerowych, ale

jest to duże marnotrawstwo, jeśli potem w kolejnych krokach musimy odrzucać wszystkie pozostałe ścieżki. Można by pokusić się o pewną modyfikację algorytmu, która wygeneruje interesującą z punktu widzenia programisty ścieżkę, ale wciąż jest ona tworzona nieefektywnie.

Z tego powodu bardzo praktycznie w ogóle nie korzysta się z algorytmu Dijkstry do rozwiązania problemu odnajdywania ścieżki. Jego wykorzystanie zostało raz zaimplementowane, jednak nie jako algorytm odnajdywania ścieżki, ale do analizy właściwości ogólnych danej mapy w zaawansowanym systemie odnajdywania ścieżki w pewniej symulacji wojskowej. Niemniej jednak, jest to ważny algorytm taktycznej analizy i ma zastosowanie w kilku innych obszarach Sztucznej Inteligencji w grach. Zbadana zostanie tutaj jego prostsza wersja jako ogólnego algorytmu to odnajdywania ścieżki,

## Przedstawienie problemu

Dany jest graf (skierowany o nieujemnych wagach) i dwa węzły (początkowy i końcowy) w tym grafie. Zadaniem algorytmu jest wygenerowanie ścieżki tak, aby całkowity koszt ścieżki był minimalny spośród wszystkich ścieżek od startu do celu. Może dojść do sytuacji że będzie istnieć wiele ścieżek o takim samym minimalnym koszcie. Ilustracja 13 zawiera 10 możliwych ścieżek, wszystkie z tym samym kosztem minimalnym. Gdy istnieje więcej niże jedna optymalna ścieżka, oczekuje się, że tylko jedna z nich zostanie zwrócona, bez znaczenia która. Warto wspomnieć, że oczekiwany rezultat powinien składać się z pewnego zbioru połączeń, a nie węzłów. Dwa węzły mogą być połączone przez więcej niż jedno połączenie i każde połączenie może mieć inny koszt (przykładowo koszt zeskoku z piętra do magazynu lub koszt wspięcia się po drabinie z magazynu na piętra). Dlatego trzeba dokładnie wiedzieć które połączenia są potrzebne, a lista węzłów byłaby niekompletna. Wiele gier nie robi tego rozróżnienia. Istnieje tam co najwyżej jedno połączenie pomiędzy dowolną parą węzłów. Pomimo to, jeśli są dwa połączenia pomiędzy parą węzłów, to moduł odpowiedzialny za odnajdywanie ścieżki powinien zawsze wybrać to połączenie z najniższym kosztem. W niektórych zastosowaniach, jednakże koszt zmienia się w trakcie gry lub pomiędzy różnymi postaciami oraz w przypadku śledzenia wielu połączeń okazuje się przydatne. Tak właśnie można zamknąć temat w którym algorytm ma sobie radzić z wieloma połączeniami. Na potrzeby tej pracy należy przyjąć, że ścieżkę będziemy identyfikować jako listę połączeń.



Ilustracja Przykład grafu, gdzie każda ścieżka jest optymalna

## Opis algorytmu

Nieformalnie, algorytm Dijkstry działa przez rozsiew od węzła początkowego wzdłuż swoich połączeń. Podczas rozprzestrzeniania się do, co raz bardziej odległych węzłów prowadzi rejestr kierunku, z którego pochodzi (można to sobie wyobrazić jako rysunek strzałek wykonanych kredą na podłodze, tak aby wskazać drogę powrotną). W końcu jak algorytm dotrze do celu zaczyna on podążać z powrotem po "strzałkach" do punktu startowego, generując tym samym kompletną ścieżkę. Ze względu na sposób działania przez proces rozprzestrzeniani, gwarantuje to strzałki z kredy wskazują zawsze najkrótszą drogę do punktu startowego. Bardziej szczegółowo można powiedzieć, że Dijkstra działa iteracyjnie. W każdej iteracji rozważany jest jeden węzeł grafu i podążą wychodzącymi z niego połączeniami. W pierwszej iteracji sprawdzi węzeł początkowy. W kolejnych krokach wybiera węzeł do rozważenia za pomocą algorytmu, który zostanie potem krótko omówiony. Na potrzeby nazewnictwa iteracji węzła zostanie przyjęte określenie "bieżący węzeł".

**Przetwarzanie bieżącego węzła**

Podczas iteracji, rozważa się każde wychodzące połączenie z bieżącego węzła. Dla każdego połączeni znajduje się węzeł końcowy i zapisuje całkowity koszt ścieżki znaleziony do tej pory wraz z połączeniami, do których dotarł algorytm.

W pierwszej iteracji, gdzie węzeł startowy jest bieżącym węzłem, kosztem dotychczasowym dla każdego połączenia w węźle końcowym jest po prostu koszt połączenia. Ilustracja 14 pokazuje sytuacje pierwszej iteracji. Każdy węzeł połączony z węzłem startowym ma koszt dotychczasowy równy kosztowi połączenia, które doprowadziło go tam - w tym przypadku to po prostu zapis wagi z tego połączenia.

Dla kolejnych iteracji dotychczasowy koszt węzła końcowego każdego połączenia jest sumą kosztów połączeń i kosztów dotychczasowych bieżącego węzła (to znaczy węzeł, z którego połączenie przychodzi.) Ilustracja 15 przedstawia kolejną iteracja tego samego grafu. Tutaj koszt dotychczasowy w węźle E jest sumą kosztu dotychczasowego z węzła B i kosztem połączenia IV z B do E.

W implementacji algorytmu, nie ma różnicy między pierwszą i kolejną iteracją. Ustalając koszt dotychczasowych wartości węzła początkowego jako 0 można skorzystać z jednego kawałka kodu dla wszystkich iteracji algorytmu.



Ilustracja Dijkastra - pierwsza iteracja



Ilustracja Dijkstra - kolejna iteracja

**Lista węzłów**

Algorytm śledzi ścieżki wszystkich węzłów, które miały miejsce w dwóch listach zwanych: lista otwarta i lista zamknięta. W otwartej liście rejestruje się wszystkie węzły dostępne, które nie miały jeszcze swojej własnej iteracji. Węzły przetworzone są zapisywane w liście zamkniętej. Na początku lista otwarta zawiera tylko węzeł początkowy (z zerowym dotychczasowym kosztem), a lista zamknięta jest pusta. Każdy węzeł może być należeć do jednej z trzech kategorii: może on należeć do zamkniętej listy - został on przetworzony w procesie swojej własnej iteracji, może być na liście otwartej - będąc odwiedzonym przez inny węzeł, ale jeszcze nie przetworzony w swojej iteracji, może również nie znajdować się na żadnej z wymienionych wyżej list. Czasami mówi się, że węzeł jest zamknięty, otwarty lub nieodwiedzony. W każdej iteracji, algorytm wybiera węzeł z listy otwartej, który ma najmniejszy dotychczasowy koszt. Następnie zostaje przetworzony w normalny sposób. Przetworzony węzeł jest usuwany z otwartej listy i dodany do listy zamkniętej. Jest jeszcze jedna komplikacja. Kiedy algorytm podąża za połączeniem od bieżącego węzła, zakłada się, że ostatecznie znajdzie się w węźle nieodwiedzonym. Może się zdarzyć taka sytuacja, że zamiast tego algorytm kończy w węźle, który jest otwarty lub zamknięty, aby rozwiązać taki przypadek trzeba postąpić nieco inaczej.

**Obliczanie dotychczasowych kosztów dla węzłów otwartych i zamkniętych**

W przypadku, gdy algorytm dochodzi do otwartego lub zamkniętego węzła podczas iteracji, to rozpatrywany węzeł będzie już posiadał dotychczasowych koszt i zapis połączenia, które doprowadziło do niego. Teraz wystarczy wstawić wartość która nadpisze wynik poprzedniej iteracji algorytmu. Zamiast tego, można sprawdzić czy trasa, która obecnie jest rozważana jest lepsza od poprzednio znalezionej. Obliczenie dotychczasowego kosztu przebiega normalnie - jeśli wartość kosztu jest większa niż ta aktualnie zarejestrowana, to nie aktualizujemy węzła w ogóle i nie zmieniamy jego listy. Jeśli nowy koszt jest mniejszy niż obecny dotychczasowy koszt węzeł, to wtedy dokonywana jest aktualizacja węzła nową, lepszą wartość, kolejno jest zapisywana w połączeniu. Następnie węzeł powinien zostać umieszczony na liście otwartej. W przypadku jeśli był on poprzednio na liście zamkniętej, to zostaje z niej usunięty. Podsumowując Dijkstra nie znajdzie lepszej trasy do zamkniętego węzła, więc może sprawdzić czy węzeł jest zamknięty najpierw i kontynuować dalsze testy dotychczasowych kosztów.

Fachowa implementacja algorytmu Dijkstry właśnie tak działa. W kolejnych rozdziałach można się jednak przekonać, że przypadku algorytmu A\* jest jednak inaczej, jakkolwiek w obydwu przypadkach muszą zostać sprawdzone najszybsze ścieżki. Ilustracja 16 przedstawia aktualizacje otwartego węzła w grafie. Nowa ścieżka przez węzeł C jest szybsza i zapis dla węzła D jest odpowiednio zaktualizowany.



Ilustracja Aktualizacja otwartego węzła

**Przerywanie algorytmu**

Podstawowa wersja algorytmu przerywa działanie, kiedy otwarta lista jest pusta, a oznacza to, że każdy węzeł został rozważony oraz wszystkie węzły znajdują się na liście zamkniętej.

W przypadku odnajdywania ścieżki obiektem zainteresowania jest tylko węzeł docelowy, więc dzięki temu można przerwać algorytm wcześniej. Dokonuje się przerwania w momencie, kiedy węzeł docelowy jest najmniejszym węzłem na liście otwartej.

Warto zwrócić uwagę na warunek przerwania. Algorytm nie jest przerywany w momencie napotkania węzła docelowego.

Ilustracja 17. Węzeł D jest węzłem docelowym i zostaje odnaleziony podczas przetwarzania węzła B. W tym momencie algorytm zostaje przerwany i zwracana jest ścieżka A-B-D, która nie jest najkrótszą ścieżką. Aby mieć pewność, że znaleziona ścieżka jest najkrótszą, algorytm musi działać dalej, dopóki cel nie będzie miał najmniejszego dotychczasowego kosztu. W tym i tylko w tym punkcie obliczeń, każdy inny punkt(znajdujący się na liście otwartej lub nieprzetworzony) będzie posiadał dłuższą drogę.

W praktyce ta zasada jest często łamana. Algorytm jest modyfikowany tak, że jeśli zostaje znaleziona ścieżka do celu to zakłada się, że ona jest najkrótsza, nawet jeśli istnieje inna, krótsza - zwykle jest ona o krótsza o niewielką wartość liczbową.

Z tego powodu większość programistów implementując algorytm Dijkstry przerywa algorytm odnajdywania ścieżki dopóki nie natknie się on na węzeł końcowy, zamiast pozwolić algorytmowi działać dalej, dopóki węzeł docelowy zostanie wybrany z otwartej listy.

**Odtwarzanie ścieżki**

Ostatnim krokiem algorytmu jest odtworzenie ścieżki. Dokonuje się tego zaczynając od punktu końcowego i podąża się połączeniami, które doprowadziły ścieżkę do tego miejsca. Można powiedzieć, że dokonuje się powrotu i szukanym punktem jest punkt początkowy. Kontynuuje się proces zapamiętując połączenie dopóki punkt startowy nie znajdzie się na powrotnej ścieżce. Ostatnim krokiem jest odwrócenie kolejności połączeń ścieżki i można zwrócić gotowe rozwiązanie.

Ilustacja17 przestawia prosty graf zaraz po tym jak algorytm zakończył działanie. Znaleziona lista połączeń zawiera węzły od celu do początku - jej odwrócenie pozwoli uzyskać poszukiwaną drogę.



Ilustracja Graf po zakończeniu obliczeń

## Algorytm A\*

Odnajdywanie ścieżki w grach jest synonimem algorytmu A\*. A\* jest prosty w implementacji, bardzo efektywny oraz posiada bardzo duże możliwości optymalizacji. Każdy moduł zajmujący się odnajdywaniem ścieżki na przestrzeni 10 lat wyżywał pewnej wariacji algorytmu A\* jako jego kluczowego algorytmu. Może również zostać wykorzystany do planowania złożonych akcji dla postaci.

W odróżnieniu do algorytmu Dijkstry, A\* został zaprojektowany dla odnajdywania ścieżki z punktu do punktu i nie jest stosowany do rozwiązywania problemu najkrótszej ścieżki w teorii grafów. A\* może zostać śmiało rozszerzony o bardziej zaawansowane przypadki.

## Przedstawienie problemu

Problem jest identyczny do tego przedstawionego w algorytmie Dijkstry.

Dany jest graf (skierowany o nieujemnych wagach) i dwa węzły w tym grafie (startowy i docelowy). Zadaniem algorytmu jest wygenerowanie takiej ścieżki, że całkowity koszt jej jest minimalny spośród wszystkich dostępnych ścieżek od startu do celu. Wyliczona ścieżka jest tak naprawdę listą połączeń od węzła początkowego do końca.

## Opis algorytmu

Nieformalnie, algorytm działa na takiej samej zasadzie jak Dijkstra. Podobnie występuje tam otwarta lista w najniższymi dotychczasowymi kosztami, ponadto zostaje tam wybrany węzeł, który jest najbardziej prawdopodobny do dotarcia do celu. Użyte sformułowanie "najbardziej prawdopodobny" mówi, że jest to kontrolowane przez heurystykę. Jeśli heurystyka została dobrana trafnie to algorytm będzie efektywny, w przeciwnym wypadku efektywność będzie gorsza od Dijkstry.

Algorytm A\* działa iteracyjnie. W każdej iteracji jest przetwarzany jeden węzeł grafu i podąża dalej wychodzącymi połączeniami. Węzeł (ponownie nazywany bieżącym węzłem) jest wybierany używając algorytmu selekcji podobnego do tego zastosowanego w Dijkstrze, ale z znaczącą różnicą heurystyki.

**Przetwarzanie bieżącego węzła**

Podczas iteracji A\* rozważa każde wychodzące połączenie z bieżącego węzła. Dla każdego połączenia znajduje węzeł końcowy i przypisuje całkowity koszt ścieżki tymczasowej i połączenia, z którego przyszedł - tak jak miało to miejsce w poprzednim algorytmie .

Dodatkowo algorytm przypisuje jeszcze jedną wartość: estymacje całkowitego kosztu dla ścieżki od węzła startowego przez obecny węzeł do celu (ta wartość zostanie teraz nazwana: estymowany koszt całkowity). Ta estymacja jest sumą dwóch wartości: kosztu dotychczasowego i wartości określającej jak daleko węzeł znajduje się od celu. Estymacja jest generowana przez oddzielną część kodu, która nie jest częścią algorytmu.

Estymacja jest powszechnie nazywana "wartością heurystyki" węzła, nie może być to wartość ujemna (wynika to z tego, że koszty w grafie są nieujemne, więc bezsensownym było by, aby estymacja miała wartość ujemną). Proces generowania wartości heurystyki jest punktem kluczowym w procesie implementacji algorytmu A\* - zostanie on omówiony później.

*Ilustracja 18* prezentuje obliczone wartości dla paru węzłów w grafie. Węzły są opisane, również wartościami heurystyki oraz są przestawione dwie wartości (dotychczasowy koszt, estymowany koszt całkowity), które informują, że dane węzły zostały przetworzone przez algorytm.



Ilustracja A\* - działanie

**Lista węzłów**

Jak w poprzednim przypadku algorytm trzyma w liście otwartej węzły, które zostały odwiedzone, ale nie przetworzone, a w liście zamkniętej te które zostały już przetworzone. Węzły są przenoszone do listy otwartej jeśli były węzłami końcowymi na liście dotychczasowo rozważanych połączeń. W przypadku listy zamkniętej - umieszczane są tam węzły, które zostały przetworzone w swojej iteracji.

W odróżnieniu od poprzedniego algorytmu węzeł z najmniejszym estymowanym kosztem całkowitym jest wybierany do każdej iteracji. Jest to prawie zawsze inny węzeł niż ten posiadający najmniejszy koszt dotychczasowy.

Ta różnica powoduje, że algorytm będzie przeszukiwał bardziej obiecujące węzły. Jeśli węzeł ma małą wartość estymowanego kosztu całkowitego, to musi on mieć też relatywnie niski koszt dotychczasowy i relatywnie małą estymacje dystansu do przebycia do celu. Jeśli estymacje są dokładne to węzły, które są bliższe celu są brane pod uwagę najpierw, nakierowując poszukiwania na najbardziej korzystny obszar.

**Obliczanie kosztu dotychczasowego dla otwartej i zamkniętej listy**

Jak wspomniano poprzednio podczas przebiegu procesu algorytmu może on dotrzeć do węzła oznaczonego jako zamknięty lub otwarty i trzeba dokonać aktualizacji zapisanych wartości.

Algorytm analogicznie dokonuje obliczenia wartość kosztu dotychczasowego i jeśli nowa wartość jest mniejsza od tej istniejącej w węźle, to dokonywana jest aktualizacja węzła. Warto zwrócić uwagę na fakt, że dokonywane jest porównanie na wartości kosztu dotychczasowego, a nie na estymowanym koszcie całkowitym.

W porównaniu do Dijkstry, A\* może znaleźć lepsze drogi do węzłów, które już znajdują się na liście zamkniętej. Jeśli poprzednia estymacja była bardzo optymistyczna, to węzeł możesz zostać przetworzony z przekonaniem, że był to najlepszy wybór, jednak faktycznie nie był.

To powoduje pewien problem. Jeśli niepewny węzeł zostanie przetworzony i umieszczony na liście zamkniętej, to znaczy że wszystkie jego połączenia zostały sprawdzone. Może się zdarzyć, że zbiór wszystkich węzłów ma koszt dotychczasowy obliczony na podstawie kosztu jednego z niepewnych węzłów. W takim przypadku aktualizacja tylko tego węzła nie wystarczy. Należy dokonać aktualizacji wszystkich połączeń, przez propagacje nowej wartości. W przypadku węzła na liście otwartej nie jest to konieczne - jak wiadomo połączenia węzłów na liście otwartej nie zostały jeszcze przetworzone. Istnieje metoda, która pozwoli na ponownie przeliczenie i propagacje nowej wartości. Można to osiągnąć poprzez usunięcie węzła z listy zamkniętej i umieszczenie go na liście otwartej. Algorytm będzie dalej kontynuował swój proces przetwarzając usunięty węzeł, umieszczając go ponownie na liście zamkniętej. Każdy węzeł, którego wartość kosztu jest związana z ponownie rozpatrywanym węzłem, zostanie przetworzony jeszcze raz.

*Ilustracja 19* przestawia aktualizacje grafu - jest to analogiczna sytuacja do poprzednio przedstawionego grafu, lecz dwie iteracje później. Przedstawia on sytuacje, w której aktualizowany jest zamknięty węzeł. Nowa trasa do węzła E przez węzeł C jest szybsza, więc dane węzła E odpowiednio aktualizowane i zostaje on umieszczony na otwartej liście. W następnej iteracji wartość węzła G zostaje zmieniona. Tak, więc węzły znajdujące się na zamkniętej liście mają zmienioną wartość kosztu i zostają z niej usunięte oraz przeniesione do listy otwartej. Otwarte węzły, które mają zmienione wartość zostają na otwartej liście.

**Przerywanie algorytmu**

W wielu implementacjach A\* przerywa swoje działanie, kiedy węzeł docelowy jest najmniejszym węzłem na liście otwartej.

Często jednak zdarza sie, że węzeł posiadający najmniejszy estymowany koszt całkowity może zostać zaktualizowany. Nie da się zagwarantować, że węzeł, który jest pierwszym na otwartej liście jest tym, który posiada najkrótszą ścieżkę do tego miejsca. Dlatego przerwanie algorytmu w tym momencie nie daje gwarancji, że została znaleziona najkrótsza ścieżka.

Jest to oczywiste, że proces algorytmu mógłby trwać troszkę dłużej, żeby wygenerować optymalny wynik. Można to osiągnąć dzięki przerwaniu wykonania algorytmu w momencie, w którym węzeł na otwartej liście z najmniejszym dotychczasowym kosztem ma ten koszt większy niż koszt znalezionej ścieżki do celu. Wtedy i tylko wtedy jest zagwarantowane, że żadna przyszła ścieżka nie będzie krótsza.

Implementacje A\* biorą pod uwagę fakt, że mogą one zwrócić teoretycznie nieoptymalny rezultat. Istnieje jednak funkcja heurystyczna, która kontroluje algorytm. Zależnie od wyboru takiej funkcji jest zagwarantowane, że zostaną zwrócone optymalne rezultaty lub można świadomie dopuścić nieoptymalne wyniki, co daje szybszy czas wykonania.

Z racji tego, że proces A\* często jest bliski znalezieniu optymalnego wyniku, duża liczba implementacji przerywa algorytm, kiedy cel jest pierwszym odwiedzonym węzłem bez czekania, żeby był pierwszy na otwartej liście. Wzrost wydajności nie jest tak duży jak w przypadku zrobienia identycznej rzeczy w Dijkstrze.

**Odtwarzanie ścieżki**

Proces odtwarzania ścieżki zaczyna się od miejsca docelowego i gromadzi połączenia cofając się do punktu startowego. Połączenia muszą zostać odwrócone, żeby utworzyć poprawną ścieżkę.



Ilustracja Aktualizacja zamkniętego węzła

# Koncepcja inteligentnego agenta

## Cele badawcze

Głównym celem będzie zbadanie wydajności algorytmów służących do nawigacji. Do tego celu zostanie wykorzystany moduł odnajdowania ścieżki. Przeprowadzone eksperymenty można podzielić na dwie części:

* Badania heurystyk oraz jakości uzyskanej drogi.
* Przyśpieszanie algorytmu A\* oraz analiza efektywności.

Narzędzia techniczne zastosowane do realizacji celu badawczego:

* Silnik Unity3D – zastosowany do zaprojektowanie trójwymiarowego świata.
* Środowisko programistyczne MonoDevelop – wykorzystane do implementacji skryptów. Jest dostarczane wraz z silnikiem Unity3D.
* Program graficzny Blender – użyty do wykonania trójwymiarowych modeli map.

## Charakterystyka środowiska graficznego

Blender jest środowiskiem do tworzenia grafiki trójwymiarowej. Oprósz standardowej funkcjonalności odpowiedzialnej za tworzenie statycznych modeli 3D czy animacji posiada też wbudowany silnik gier. Blender jest darmowy. Poniżej znajduje się zrzut ekranu przedstawiający postęp prac nad modelowaniem jeden z map świata gry.

Ilustracja Proces tworzenia mapy w programie Blender

## Charakterystyka silnika Unity3D

Unity3D jest silnikiem przeznaczonym do tworzenia gier komputerowych. Jest to w pełni zintegrowany potężny silnik renderujący. Posiada on komplet narzędzi wspomagających tworzenie interaktywnych treści 3D. Każdą zaprojektowaną aplikację można opublikować w prosty sposób na większość z dostępnych platform (Windows, Linux, Mac, Android, Windows Phone, iPhone, XBOX360, Playstation 3), zaznaczając, na którą z nich ma zostać zbudowana aplikacja. Unity3D jest dostępny w dwóch wersjach: darmowej: Basic oraz płatnej Pro.

Często programiści piszący gry komputerowe zaczynają ten proces od napisania silnika gry. Silnik ten będzie przetwarzał zasoby graficzne, zajmował się fizyką, będzie posiadał wbudowany edytor map oraz posiadał implementacje samej gry. W zasadzie każdy programista, który kiedykolwiek zajmował się implementacją gry, spotkał się z tworzeniem silnika. Zwykle jest to ostatni etap prac programisty, ponieważ w tym etapie programista zmaga się z własnymi błędami projektowymi oraz poprawia w nieskończoność błędy algorytmiczne. Tym samym koncentruje się na rozwoju silnika, a zapomina o głównym celu jakim jest stworzenie gry. Oto kilka argumentów przemawiających za tym, aby do realizacji tej pracy wykorzystać właśnie silnik Unity3D:

Wygodny w użyciu edytor zawierający:

* + Zakładka "Console" - znajduje się tam podgląd logów wyświetlanych przez grę, a także ostrzeżeń i błędów. Logi wysyłamy do konsoli z poziomu skryptów.
  + Zakładka "Inspector" - inspektor obiektów przy pomocy którego możemy ustalać różne właściwości obiektu takie jak: pozycja, rotacja, skala, parametry skryptów, parametry związane z wyświetlaniem czy fizyką.
  + Zakładka "Hierarchy" - panel widoku hierarchii obiektów na scenie
  + Zakładka "Project" - panel na którym znajduje się widok wszystkich elementów znajdujących się w projekcie: modele, materiały, skrypty, sceny.
  + Zakładka "Game" - panel przeznaczony do szybkiego testowania gry. Projektant ma tutaj podgląd jak po kompilacji i uruchomieniu będzie wyglądać gra.
  + Zakładka "Scene" - w tym oknie znajduje się widok sceny, po którym projektant może dowolnie się poruszać ustawiać obiekty, światło czy kamerę

Wyraźny podział na obiekty, komponenty oraz skrypty

* + Objekty ("GameObjects") - jest to rozszerzalny byt abstrakcyjny, który w bazowej postaci posiada transformacje, nazwę, dzieci, oraz może mieć rodzica. Może oczywiście być pustym obiektem i nie posiadać żadnej reprezentacji na ekranie, a może być również dowolnym modelem 3D, prymitywem, światłem, kamerą i w pełni oskryptowanym samochodem.
  + Komponenty ("Components") - Komponenty wchodzą w skład obiektów, a konkretnie rozszerzają je o nowe funkcjonalności. Filozofia działania Unity3D związana się właśnie pracą na komponentach. Komponent przeważnie posiada pewną liczbę parametrów. Przykładowo renderując na ekranie sześcian, to taki obiekt składa z następujących elementów: GameObject (pusty) zawiera komponent MeshFiler - przechowuje dane o geometrii, jako parametr przyjmuje siatkę sześcianu. Następnie obiekt ten zawiera kolejny komponent MeshRenderer - posiada parametr ustawiający materiał renderowania. Co więcej, obiekt może zawierać komponent Rigidbody - w nim ustawiane jest ciało fizyczne dla obiektu. Nadawana jest mu masę oraz gęstość. Dzięki temu istnieje możliwość oddziaływania na obiekt siłą. Komponentem może też być skrypt.
  + Komponent skrypt - jest to kluczowy komponent dla programisty. Pisząc jeden skrypt można go dodać do wielu obiektów. Oczywiście do obiektu można przypisać więcej niż jeden skrypt. Skrypty mogą być pisane w językach JavaScript, C# lub Boo. Skrypty są kompilowane w locie, więc na bieżąco widzimy czy nie został popełniony błąd. W przypadku wystąpienia wyjątku (np. NullPointerException) nie powoduje to zamknięcie Unity3D, tylko wstrzymanie działania aplikacji (aktywna pauza), dzięki czemu można sprawdzić w logach gdzie wystąpił błąd.

Bogata pomoc

* + Unity Manual - dokumentacja zawierająca opis okien, sposób poruszania się po edytorze, skróty klawiszowe, itp.
  + Reference Manual - zawiera opisz poszczególnych części edytora np. elementów odpowiedzialnych za fizykę, dźwięk czy rendering
  + Scripting Reference - znajduje się tutaj opis wszystkich klas, do których możemy się odwoływać z poziomu skryptów. Większość z klas i metod jest wsparta prostymi przykładami w trzech językach, które wspiera Unity3D: JavaScript, C# oraz Boo.
  + W sieci znajduje się duża liczba kursów oraz samouczków do Unity3D, także w wersji wideo.
  + Wokół Unity3D znajduje się duża społeczność programistów, którzy aktywnie działają na forach dyskusyjnych oraz służą pomocą w przypadku napotkania problemów

Elastyczność

* + Rozszerzalność - programista może sam pisać własne skrypty i wtyczki (ang. plugin)
  + Prostota eksportów - elementy może przenosić między projektami poprzez eksportowanie/importowanie paczek z zasobami gry (ang. packages)
  + Prefabrykowane elementy (ang. prefabs) - Dodając do sceny pewną liczbę prefabrykowanych elementów powoduje to, że w trakcie zmiany parametrów jednego z nich, zmiany te są propagowane do pozostałych elementów tego samego typu. Pozwala to przyśpieszyć proces składania poziomu w całość oraz testowanie prototypów.
  + Silnik Unity3D posiada wbudowaną bazę jednostek cieniujących (ang. shader). Oczywiście programista może rozbudować tą bazę piszą swoje własne.



Ilustracja Zrzut ekranu prezentujący środowisko silnika Unity3D

## Wybrana metoda optymalizacyjna

Podczas każdej iteracji algorytm A\* rozszerza obszar poszukiwań w najlepszym znanym kierunku. Jednak istnieją sytuacje, które mogą prowadzić do spadku jego efektywności. Spadek ten pojawia się w przypadku przetwarzania dużych przestrzeni.

W tej części pracy zostanie omówiony algorytm JPS(ang. Jump Point Search). Jego działanie pozwoli na przyśpieszenie obliczeń szukanej ścieżki.

Algorytm JPS można opisać w postaci dwóch prostych zasad obcinania, które są stosowane w rekursywnym przeszukiwaniu. Pierwsza zasada dotyczy wertykalnego i horyzontalnego poruszania się, natomiast druga diagonalnego. Kluczowe w obydwóch przypadkach jest obcinanie zbioru bezpośrednich sąsiadów wokół węzła, próbując udowodnić, że optymalna ścieżka (symetrycznie lub nie) istnieje od rodzica bieżącego węzła do każdego sąsiada, a ścieżka nie wymaga odwiedzania aktualnie rozpatrywanego węzła.

W przypadku algorytmu A\* dokonywana jest operacja rozszerzania w najprostszy z możliwych sposobów: poprzez dodanie węzła bezpośredniego sąsiada, bez badania następnego. Warto zastanowić się czy nie można pójść o krok dalej i pominąć niektóre węzły, które można intuicyjnie uznać mało wartościowe. Można tego dokonać określając sytuacje, w której są obecne symetryczne ścieżki i pomijać niektóre węzły. Rysunek 1 przedstawia podstawową idee algorytmu JPS.



Węzeł x jest obecnie rozszerzany. Strzałka wskazuje kierunek podróży od jego rodzica, horyzontalnie lub diagonalnie. W obu przypadkach można od razu wykluczyć wszystkie szare węzły sąsiadów, ponieważ mogą zostać one osiągnięte optymalnie z rodzica węzła x bez potrzeby przechodzenia z węzła x.

Pojęcie naturalnego sąsiada bieżącego węzła będzie odnosić się do zbioru węzłów pozostałych po procesie obcinania. Są one oznaczone na kolor biały na rysunku 1. W idealnym przypadku rozważany jest zbiór naturalny sąsiadów podczas procesu rozszerzania. Jednak w niektórych przypadkach , obecność przeszkód może oznaczać, że trzeba również rozważyć mały zbiór k-dodatkowych węzłów (0<= k <= 2). Zbiór jest zwykle nazywany zbiorem wymuszonych sąsiadów bieżącego węzła. Rysunek x przedstawia taki przypadek.

Węzeł x na rysunku x jest węzłem aktualnie rozszerzanym, strzałka wskazuje kierunek trasy od rodzica, horyzontalnie i diagonalnie. Należy zauważyć, że gdy węzeł x jest obok przeszkody to wyróżnieni sąsiedzi nie mogą zostać usunięci. Każda alternatywna optymalna droga z rodzica węzła X do każdego z tych węzłów jest zablokowana.



Wyżej opisane zasady obcinania węzłów stosuje się w następujący sposób: zamiast generować naturalnych i wymuszonych sąsiadów, wykonuje się rekursywne obcinanie zbioru sąsiadów wokół każdego takiego węzła. Celem tej operacji jest wyeliminowanie symetrii przez rekursywne przeskoki pomiędzy węzłami, do których można dotrzeć ścieżką nie przechodzącą przez bieżący węzeł. Rekursja jest przerywana w momencie, kiedy algorytm natrafi na przeszkodę lub następcę punktu skoku. Punkty skoku są o tyle interesujące, ponieważ posiadają one sąsiadów, do których nie można dotrzeć żadną inną alternatywną symetryczną ścieżką. Podsumowując optymalna ścieżka musi przechodzić prze bieżący węzeł.

Rekursywny algorytm obcinania sąsiadów musi zapewniać optymalność. Jest ona zagwarantowana przez uporządkowany proces, w jakim przetwarzani są naturalni sąsiedzi (najpierw horyzontalnie/wertykalnie, a następnie diagonalnie). Rysunek x przedstawia dwa przykłady algorytmu obcinania w akcji. W pierwszym przypadku możemy określić punkt skoku przez rekursje horyzontalną, a w drugim punkt jest określany przez rekursje diagonalną.



Rysunek Skok diagonalny Rysunek Horyzontalny skok

Rysunki x i y przedstawiają przykładowe skoki. Węzeł x jest aktualnie rozszerzanym węzłem p(x) jest jego rodzicem.

Na rysunku 1 została przedstawiona prosta reguła przycinania i identyfikacji węzła y jako następcy skoku x. Węzeł ten jest rozważany, ponieważ ma sąsiada z, do którego optymalna ścieżka zgodnie z założeniami musi prowadzić przez punkty x i potem y.

Zastosowanie rekursji z wariantem przekątnej reguły obcinania na rysunku 2, powoduje wyznaczenie węzła y jak sukcesora x. Warto zwrócić uwagę, że przed każdym krokiem rozszerzania przekątnej najpierw odbywa się rekursja horyzontalna i wertykalna (linie przerywane). Tylko wtedy, gdy obie rekursje (wertykalna i horyzontalna) nie zwrócą punktu skoku wykonuje się ponownie krok diagonalny. Węzeł ‘w’ w tym przypadku jest wymuszonym sąsiadem węzła x.

# Realizacja

Zaprojektowane zostało proste środowisko testowe korzystające z systemu nawigacji postaci. Do rozwiązania problemu znajdowania ścieżki zastosowany został algorytm A\*. Warto zwrócić uwagę na kwestie wydajnościowe tego algorytmu biorąc pod uwagę następujące czynniki badane w pierwszym eksperymencie:

* Poziom skomplikowania mapy (od prostej ścieżki po labirynt),
* Użytą funkcje heurystyczną: odległość euklidesową, Manhattan, diagonalny Manhattan,
* Wykorzystane skrypty modyfikujące ścieżkę (zostaną opisane później),
* Gęstość grafu (obszar poszukiwań zostaje zwiększony x4 i x16).

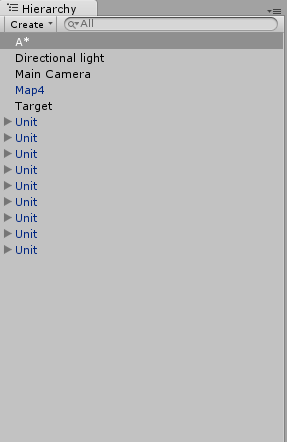
W przypadku drugiego eksperymentu:

* Poziom przyśpieszenia czasowego wynikającego z zastosowania optymalizacji.

## Konfiguracja biblioteki

Po uruchomieniu silnika Unity3D i zainstalowaniu paczki z biblioteką 'A\* Pathfinding Project' pierwszym krokiem jest dodanie do sceny mapy, na której ma działać moduł odnajdowania ścieżki. Kolejno dodajemy pusty obiekt, który będzie zawierał całą konfigurację biblioteki (w projekcie nazwany jest on 'A\*' - Ilutstacja xx reprezentuje widok hierarchii projektu). Do obiektu A\* dodajemy skrypt z biblioteki o nazwie 'Astar Path', którym dokonywana jest konfiguracja ustawień. Odpowiednia konfiguracja jest bardzo istotna z punktu widzenia optymalizacji oraz napotkanych błędów w czasie testowania aplikacji. Poniżej znajduję się część najbardziej istotnych parametrów konfiguracyjnych:

* Rodzaj grafu – wybrana została siatka (Grid Graph).
* Rozmiar węzła grafu. Mapa ma rozmiar 200x200, więc przyjęto rozmiar węzła 5x5, zatem długości i szerokości grafu będzie wynosić 50x50. Wartości te będą się zmieniały w przypadku badań wydajności gęstości grafu na 100x100 i 200x200.
* Po ustaleniu rozmiaru siatki grafu należy ją umieścić tak aby całą swoją długością i szerokością (współrzędne x i z) obejmowały całą mapę.
* Parametr *MaxClimb* - opisuje maksymalną pozycje pomiędzy dwoma węzłami, tak aby istniało połączenie
* *MaxSlope* - określa maksymalny kąt nachylenia dla węzła aby można po nim przejść.
* Biblioteka umożliwia budowanie grafu przez dokonywanie testów wysokości terenu. Robi to za pomocą rzucania promieni (ang. raycast). Związany jest z tym parametr opisujący długość tego promienia (ray length). Przykładowo projektując budynek o wysokości 10, w którym agent będzie miał możliwość poruszania się po każdym piętrze wartość ta powinna wynosić również 10. W przypadku przygotowanych map gracz porucza się w dolnie i nie może wejść na górę. Wysokość całej mapy wynosi 3, więc wysokość dla jakiej tworzony jest graf jest ustawiona na 2.



## Implementacja

**Implementacja skryptu nawigującego**

Ostatnim elementem przed uruchomieniem aplikacji jest implementacja skryptu odpowiedzialnego za przemieszczanie się obiektów po mapie na podstawie ścieżki. Skrypt został dodany (jako komponent) do każdego agenta występującego w grze.

using UnityEngine;

using System.Collections;

using Pathfinding;

public class AI\_Shifter : MonoBehaviour {

public Transform target;

private Seeker seeker;

Path path;

CharacterController agentController;

int currentWaypoint;

float speed = 10.0f;

float maxWaypointDistance = 2.0f;

void Start () {

seeker = GetComponent<Seeker>();

seeker.StartPath(transform.position,

target.position,

OnPathCompleted);

agentController = GetComponent<CharacterController>();

}

public void OnPathCompleted(Path p)

{

if (!p.error)

{

path = p ;

currentWaypoint = 0 ;

}

else{

Debug.Log(p.error);

}

}

void FixedUpdate()

{

if (path == null)

return;

if (currentWaypoint >= path.vectorPath.Count)

{

return;

}

Vector3 direction = (path.vectorPath[currentWaypoint] - transform.position).normalized \* speed;

agentController.SimpleMove(direction);

gameObject.transform.forward = direction.normalized;

float distance = Vector3.Distance(transform.position, path.vectorPath[currentWaypoint]);

if (distance < maxWaypointDistance){

currentWaypoint++;

} }

}

Pole Transform target określa współrzędne celu, do którego mają dostać się agenci. Jest to pole publiczne - takie pola w Unity3D mają możliwość wstawienia obiektu z poziomu silnika. Rysunek x przedstawia dodany skrypt do obiektu agenta oraz widoczne pole Target z ustawionym celem.



Rysunek Dodany skrypt do agenta

Seeker jest skryptem dostarczonym wraz z biblioteką. Unity3D pozwala na traktowanie skryptów jak obiekty, co więcej można pobrać taki skrypt dodany do obiektu i wywołać z niego odpowiednie metody. Do pola Path zostanie przypisana ścieżka, którą musi przejść agent - jest ona zwracana przez wywołanie zwrotne metody ze skryptu Seeker.

Skrypt najpierw wykonuje metodę Start(), w której pobierany jest skrypt z biblioteki. Z pobranego skryptu wywoływana jest metoda StartPath. Przyjmuje ona za parametry aktualną pozycję agenta, cel oraz wywołanie zwrotne metody, która przekaże obliczoną ścieżkę do skryptu. Metoda FixedUpdate wykonuje się wielokrotnie więcej razy na klatkę.

**Skrypty modyfikujące**

Jakość otrzymanego rozwiązania w postaci ścieżki można rozpatrywać jako czas obliczenia najkrótszej ścieżki. Jednak interesujący jest aspekt jakości w rozumieniu gracza. Mianowicie czy postacie poruszające się po ścieżce robią to w sposób taki, jak poruszałby się po niej człowiek? Na podstawie przeprowadzonych badań można dojść do wniosku, że wygenerowanie ścieżki i zaprogramowanie agentów, nie daje ładnego wizualnego rezultatu. Agenci poruszają się dokładnie po ścieżce. Przykładowo wykonując zakręt linia skrętu przedstawia złączenie odcinków, a nie wygładzony wycinek koła. Dlatego do podniesienia jakości wygenerowanego rezultatu stosuje się skrypty modyfikujące.

Skrypt modyfikujący jest komponentem dołączonym do agenta (agent może posiadać maksymalnie 2 skrypty modyfikujące). Na potrzeby projektu zostały dołączone dwa skrypty modyfikujące:

* funnel - jest to prosty algorytm, który znajduje proste odcinki trasy. Poniższa ilustracja przedstawia efekt działania tego algorytmu. Czerwona linia to ścieżka wygenerowana przez główny algorytm, natomiast zielona jest modyfikacją tej ścieżki.



* alternative path - generuje alternatywne ścieżki dla agentów. Powoduje to, że agenci nie poruszają się jeden za drugim.

Zastosowanie skryptów modyfikujących podnosi jakoś znalezionego rozwiązania z punktu widzenia gracza. Cały proces obróbki takiej ścieżki wymaga jednak wykonania dodatkowych obliczeń .

**Implementacja części optymalizacyjnej**

Implementacja optymalizacji algorytmu A\* wymagała zaprogramowania całego modułu odnajdowania ścieżki. Poniżej zostaną przedstawione klasy biorące udział w procesie reprezentacji danych mapy oraz główny skrypt, w którym zawarta jest optymalizacja algorytmu.

Do reprezentacji siatki mapy wymagana była implementacja prostej klasy punktu (Point). Posiada konstruktor przyjmujący współrzędne x i y opisujące punkt na siatce oraz właściwości umożliwiające ustawienie wartości współrzędnych.

namespace JumpPointSearach

{

public class Point

{

private int mX;

private int mY;

public int X {

get { return this.mX;}

set { this.mX = value;}

}

public int Y {

get { return this.mY;}

set { this.mY = value;}

}

public Point (int x, int y)

{

this.mX = x;

this.mY = y;

}

}

}

Do reprezentacji węzła w grafie stosowana jest klasa Node. Poniżej znajduje się listing x Zawierający opis pól tej klasy.

#region Fields

private float mCostFunction;

private readonly float mHeuristicEstimateCost = 0;

private float mCostFromStart = 0;

private List<Node> mNeighbors = null;

private readonly int mX = 0;

private readonly int mY = 0;

private Vector3 mNodePosition;

private Node mParentNode = null;

private readonly bool mIsDestinationNode = false;

#endregion

Pole mCostFunction zawiera wartość funkcji kosztu obliczone poprzez zsumowanie wartości estymowaej wartości (mHeuristicEstimateCost) oraz dotychczas obliczonego kosztu (mCostFromStart). Z każdym węzłem związana jest lista sąsiadujących z nim węzłów – mNeighbors. Reprezentacja węzła na siatce jest związana z polami mX i mY, natomiast punkt w przestrzeni reprezentowany jest przez wektor mNodePosition. W trakcie działania algorytmu każdy węzeł będzie posiadał rodzica za wyjątkiem pierwszego węzła, który będzie miał ustawione pole mParentNode na wartość null. Dodatkową ważną informacją przechowywaną w klasie repezentującej węzeł jest to czy dane węzeł jest węzłem docelowym. Stan ten jest reprezentowane przez wartość logiczną w polu mIsDestinationNode. Każde z pól posiada właściwość pobierającą lub ustawiającą dane pole, o ile nie jest tylko do odczytu (posiada słowo kluczowe *readonly*).

Na listingu x został przedstawiony konstruktor obiektu *Node:*

#region Constructors

public Node (Node parentNode, int x, int y, Vector3 nodePosition, Vector3 destinationPosition)

{

if (parentNode != null) {

mParentNode = parentNode;

mCostFromStart = mParentNode.CostFromStart + Node.estimate (mParentNode.NodePosition, nodePosition);

}

mX = x;

mY = y;

mNodePosition = nodePosition;

mIsDestinationNode = (mNodePosition == destinationPosition);

mHeuristicEstimateCost = Node.estimate (nodePosition, destinationPosition);

// f = g + h

mCostFunction = mCostFromStart + mHeuristicEstimateCost;

}

#endregion

Konstruktor za parametry przyjmuje węzeł rodzica, współrzędne x,y oraz wektory określające położenie węzła i punkt docelowy. Kolejno są ustawiane pola klasy oraz liczone estymacje. Do obliczenia estymacji wykorzystana jest metoda Manhattan (diagonalna). Obliczone wartość heurystyk i dotychczasowego kosztu są zapisywane w polu mCostFunction.

Implementacja głównego skryptu znajduje się w klasie *JumpPointNavigationMesh*. Rozszerza ona klasę MonoBehaviour, co powoduje, że klasa może być wykorzystana jak skrypt i być użyta jak komponent. Wykorzystanie klasy jako komponent pozwoli w prosty sposób zainicjalizować wartość pól publicznych zawartych w klasie z poziomu interfejsu graficznego silnika Unity3D. Definiując pojęcie skryptu w Unity3D można powiedzieć, że jest to klasa dziedzicząca po klasie MonoBehaviour. Na listingu x znajdują pola publiczne zawarte w głównym skrypcie.

#region Public fields

public GameObject Gizmo;

public Material PathMaterial;

public Transform StartTransform;

public Transform DestinationTransform;

public int resolution = 2;

public int ApproximationRadius = 1;

public float DistanceTolerance = 4.0f;

public float MaxHeightDelta = 0.001f;

public Vector3 InitialMappingPoint;

#endregion

Pierwsze z publicznych pól (Gizmo) jest obiektem, który posłuży do rysowania ścieżki na mapie. Może to być przykładowo jeden z prymitywów takich jak: kula czy sześcian. PathMaterial jest to materiał określający kolor ścieżki. StartTransform i DestinationTransform są obiektami określającymi początek i koniec ścieżki. Pole resolution określa jaką część powierzchni terenu będzie zajmowała graficzna reprezentacja węzła. Algorytm wykonujący obliczenia musi uwzględnić obiekty znajdujące się na scenie. Takimi obiektami są np. drzewa. Wartości ApproximationRadius oraz DistanceTolerance opisują tolerancje oraz promień wokół, którego będą usuwane węzły ze względu na znajdujące się tam obiekty blokujące ścieżkę. MaxHeightDelta określa do jakiego poziomu wysokości będzie budowany graf. Zwiększanie tej wartość spowoduje, że graf zacznie być generowany dla terenów położonych wyżej, co w efekcie pozwoli postaci poruszać się po wyższych partiach terenu.

Listing x przedstawia pola prywatne.

#region Fields

private Vector3 mTerrainPosition;

private Vector3 mGizmoHeight = new Vector3();

private Vector3 mMeshScale;

private Vector3 mTerrainScale;

private float[,] mHeightMap;

private int mWidth = 0;

private int mHeight = 0;

private int mTerrainResolution = 4;

private List<Object> mPaths = new List<Object> ();

private List<Node> mNeighbors = null;

#endregion

Wektor mTerrainPosition zawiera w sobie współrzędne terenu. Pole to będzie wykorzystane w procesie skalowania siatki i terenu. mGizmoHeight jest wektorem opisującym wysokość na jakieś będą rysowane znaczniki opisujące trasę. Pola mMeshScale i mTerrainScale opisują skalę siatki i terenu. mTerrainScale określa rozmiar terenu 3D, a mMeshScale zawiera przeskalowaną wartość mTerrainScale względem ustawionej rozdzielczości mapy wysokości. Pola te będą wykorzystywane podczas odczytywania współrzędnych świata posiadając współrzędne siatki oraz analogicznie w odwrotnym przypadku.

Obiekt terenu w Unity3D jest zapisany w specyficznej strukturze danych, którą programista może pobrać i przetwarzać w dowolny sposób. Struktura ta nazywana jest mapą wysokości (ang. heightmap). Zawiera ona pełen opis struktury mapy zapisany w tablicy dwuwymiarowej przechowującej wartości zmiennoprzecinkowe. Indeksy tablicy opisują położenie punktów mapy na osiach X i Z natomiast wartość zmiennoprzecinkowa określa wysokość punktu w terenie (oś Y). Tym sposobem można przykładowo zapisać trójwymiarową mapę do dwuwymiarowego pliku graficznego, gdzie trzeci wymiar jest reprezentowany przez wartość koloru piksela. W omawianym skrypcie mapa jest przechowywana w tablicy mHeightMap. Pola mWidth i mHeight określają początkowo długość i szerokość mapy wysokości, a następnie siatkę grafu. Kolejnymi polami są dwie listy: mPaths oraz mNeighbors. Pierwsza z nich reprezentuje ścieżkę w trójwymiarowym świcie – jest to zbiór obiektów reprezentujących znalezioną drogę do celu. Druga z list jest tworzona osobno dla każdego węzła i zawiera listę jego sąsiadów.

Skrypt składa się z kilku głównych metod oraz metod pomocniczych, ze względu na dużą ilość lilnii kodu nie zostanie tutaj umieszona pełna implementacja.

Działanie skryptu rozpoczyna się od wykonania metody Start(). W niej pobierany jest obiekt terenu (TerrainCollider), który jest przekazywany od metody InitTerrainData(). Metoda tak pobiera długość i szerokość mapy wysokości z przekazanego obiekty terenu, ustawiana jest zmienna opisująca rozmiar terenu oraz dokonywane jest skalowanie mapy wysokości terenu do zadanej skali. Następnie tworzona jest nowa mapa wysokości z przeskalowanymi wartościami.

Jedną z metod dostępnych w skrypcie jest metoda OnGUI() zajmująca się wyświetlaniem interfejsu graficznego i jego interakcji z użytkownikiem. Zawiera ona kod rysujący przycisk odpowiedzialny za rozpoczęcie procesu odnajdywania ścieżki.

Po kliknięciu na przycisk wywoływana jest metoda FindPath przyjmująca za argumenty dwa wektory początkowy i cel. Metoda została przedstawiona w listingu.

private IEnumerator FindPath (Vector3 start, Vector3 destinationTarget)

{

foreach (Object path in mPaths) {

Destroy (path);

}

mPaths.Clear ();

List<Vector3> paths = new List<Vector3> ();

SortedHeap<Node> openList = new SortedHeap<Node> (new NodeComparer ());

SortedHeap<Node> closedList = new SortedHeap<Node> (new NodeComparer ());

Point startPoint = GetGridPosition (start);

Point destinationPoint = GetGridPosition (destinationTarget);

if (startPoint.X == destinationPoint.X && startPoint.Y == destinationPoint.Y) {

yield break;

}

Vector3 worldStart = GetWorldPosition (startPoint.X, startPoint.Y);

Vector3 worldDestination = GetWorldPosition (destinationPoint.X, destinationPoint.Y);

openList.Push (new Node (null, startPoint.X, startPoint.Y, worldStart, worldDestination));

// Check open list

// GetNeighbors

// Make a jump

}

Ze względu na duży rozmiar metody listing x zawiera początek implementacji metody. Pierwszym krokiem jest wyczyszczenie listy mPaths zawierającej obiekty rysujące ścieżkę w wirtualnym świeci 3D. Następnie tworzona jest lista paths zawierająca punkty ścieżki grafu. Implementacja metody zawiera listy: otwartą() i zamkniętą. Listy te są znane z algorytmu A\* są one dokładnie opisane w rozdziale x dotyczącym opisu działania tego algorytmu.

Następnym krokiem jest pobranie punktów z przestrzeni świata 3D i zwrócenia ich reprezentacji na siatce grafu zajmuje się tym metoda GetGridPosition. Potem tworzony jest węzeł początkowy i dodawany do otwartej listy.

Dalszy bieg procesu sprawdza stan otwartej listy. W przypadku gdy znajdują się na niej jakieś elementy są one pobierane i przetwarzane. Pierwszy krok przetwarzania polega na sprawdzeniu warunku końca algorytmu, czyli jeśli aktualnie rozważany węzeł jest węzłem końcowym, to następuje budowanie ścieżki na siatce grafu. W przeciwnym wypadku wykonywana jest operacja pobrania sąsiadów (GetNeighbors()).

Metoda GetNeighbors() dokonuje analizy sąsiadów aktualnie rozpatrywanego węzła. Wylicza ona odległość od bieżącego węzła do sąsiada oraz dokonuje analizy sąsiadów pod względem tego czy dany węzeł jest osiągalny przez algorytm (znajduje się powyżej dopuszczalnego limitu wysokości lub znajduje się na nim przeszkoda). Za parametry wejściowe przyjmuje ona aktualnie rozpatrywany węzeł oraz węzeł docelowy. Wartością zwracaną jest lista węzłów (sąsiadów).

Po znalezieniu sąsiadów analizowanego węzła jest wywoływana metoda Jump. Jej implementacja znajduje się w listingu x.

private Point Jump (int x, int y, int parentX, int parentY, Vector3 destinationTarget)  
{  
    int dx = x - parentX;  
    int dy = y - parentY;  
      
    if (isReachable (parentX, parentY, x, y) == false) {  
        return null;  
        } else if (GetWorldPosition (x, y) == destinationTarget) {  
        return new Point (x, y);  
    }

    //forced diagonal neighbors  
    if (dx != 0 && dy != 0) {  
        if ((isReachable (parentX, parentY, x - dx, y + dy) &&  
        isReachable (parentX, parentY, x - dx, y) == false) ||  
        (isReachable (parentX, parentY, x + dx, y - dy) &&  
        isReachable (parentX, parentY, x, y - dy) == false)) {  
            return new Point (x, y);  
        }  
        } else { //horizontal||vertical  
        if (dx != 0) {  
            if ((isReachable (parentX, parentY, x + dx, y + 1)  
            && isReachable (parentX, parentY, x, y + 1) == false) ||  
            (isReachable (parentX, parentY, x + dx, y - 1)  
            && isReachable (parentX, parentY, x, y - 1) == false)) {  
                return new Point (x, y);  
            }  
            } else {  
            if ((isReachable (parentX, parentY, x + 1, y + dy)  
            && isReachable (parentX, parentY, x + 1, y) == false) ||  
            (isReachable (parentX, parentY, x - 1, y + dy)  
            && isReachable (parentX, parentY, x - 1, y) == false)) {  
                return new Point (x, y);  
            }  
        }  
    }  
      
    if (dx != 0 && dy != 0) {//check for horizontal||vertical jump points  
        if (Jump (x + dx, y, x, y, destinationTarget) != null ||  
        Jump (x, y + dy, x, y, destinationTarget) != null) {  
            return new Point (x, y);  
        }  
    }

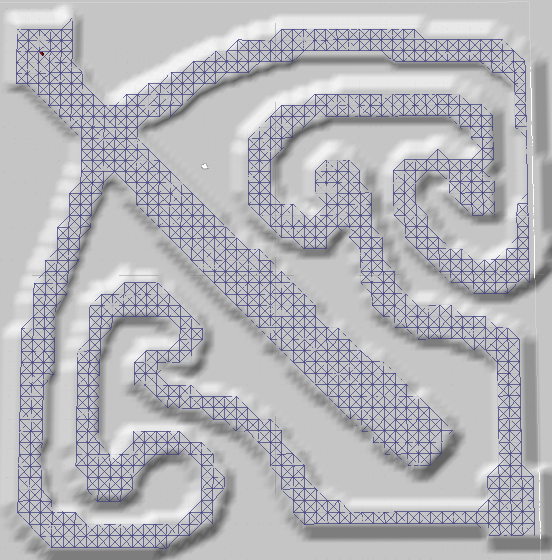
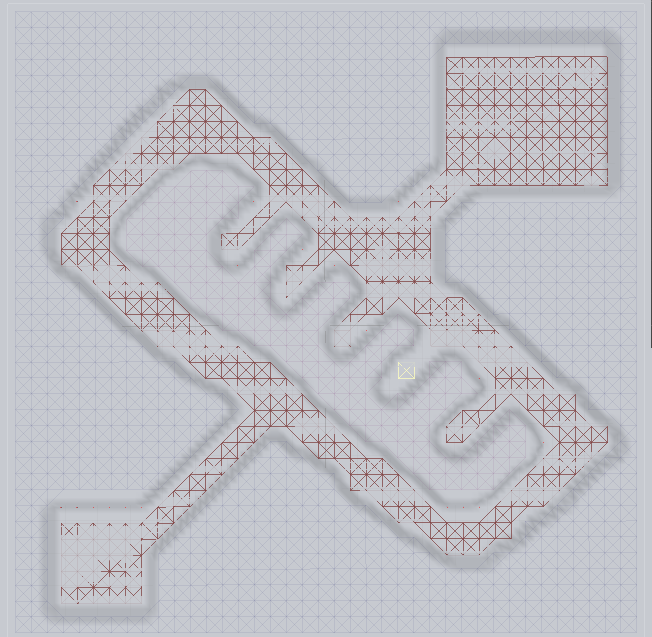
    // check diagonals jump points  
    if (isReachable (parentX, parentY, x + dx, y) || isReachable (parentX, parentY, x, y + dy)) {  
        return Jump (x + dx, y + dy, x, y, destinationTarget);  
        } else {  
        return null;  
    }       
}

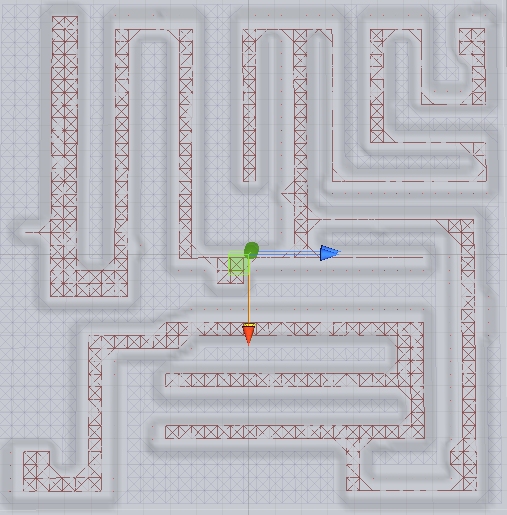
Metoda Jump() zajmuje się rekursywnym rozszerzaniem ścieżki wertykalnie, horyzontalnie lub diagonalnie tworząc kolejne punkty skoku. Za parametry przyjmuje współrzędne dziecka(x i y), współrzędne rodzica (parentX, parentY) oraz punkt docelowy (). Wartością zwracaną jest koordynata punktu skoku.

W pierwszym kroku jest sprawdzane czy analizowana para węzłów dziecko-rodzic posiada połączenie oraz czy węzeł dziecka nie jest węzłem docelowym. Kolejno analizowane są przypadki wymuszonych sąsiadów i zwracane są ich koordynaty jeśli tacy sąsiedzi istnieją. Następnie sprawdzane są zgodnie z założeniami punkty wertykalne i horyzontalne w pierwszej kolejności, a następnie poddawane są analizie punkty diagonalne. W każdym z tych przypadków następuje rekursywne wywołanie metody Jump(), która zwraca punkt, do którego należy wykonać skok.

## Przegląd map i wygenerowanych grafów

Poniżej znajdują się modele map z podpiętym pod nie skryptem wyznaczającym graf - obszar, którym postać może się poruszać. Obszary niepokryte siatką grafu są niedostępne dla agentów.





## Eksperyment 1 – badanie heurystyk

Dla każdej z wygenerowanych map zostały przeprowadzone testy. Polegały one na umieszczeniu dziesięciu komputerowych agentów w wybranym punkcie startowym, następnie został im wskazany punkt docelowy, do którego mieli się udać.

Wykonano zapis czasu wykonania obliczeń ścieżki dla każdego agenta, ilości węzłów jaka zostało odwiedzona oraz ile wynosiła długość ścieżki.

Poniżej znajdują się wyniki pomiarów dla każdej z zaprojektowanych map.

**Mapa 1**

****

Powyższa ilustracja przedstawia wygenerowaną ścieżkę z zastosowaniem skryptów modyfikacyjnych. Warto zauważyć, że skrypt "alternative path" spowodował, wygenerowanie dwóch zupełnie innych ścieżek. Jest to często występujący błąd w grach strategicznych. Wskazując grupie agentów cel, do którego mają dotrzeć powoduje, że w pewnym momencie droga, którą zmierzają do celu jest zupełnie inna.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 1,595 | 1211 | 224 | 2,1 | 1632 | 224 | 1,7 | 1376 | 224 |

Tabela 1 przedstawia wyniki symulacji na mapie 50x50 bez włączonych modyfikatorów ścieżki. Czasy wykonywania obliczeń są mierzone w milisekundach i dla heurystyki eukidesowej i diagonalnego Manhattan są wręcz identyczne. Wykorzystanie metody Manhattan spowodowało, że algorytm odwiedził najwięcej węzłów z pośród badanych heurystyk. Najmniej węzłów zostało odwiedzonych stosując heurystykę eukidesową.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 5 | 1211 | 224 | 5 | 1632 | 224 | 6,01 | 1376 | 224 |
| Agent 2. | 2 | 1301 | 225 | 2 | 1763 | 225 | 2 | 1419 | 225 |
| Agent 3. | 0,99 | 1351 | 225 | 3 | 1896 | 226 | 2 | 1458 | 225 |
| Agent 4. | 1 | 1380 | 228 | 2 | 1890 | 229 | 1 | 1466 | 226 |
| Agent 5. | 2 | 1414 | 228 | 2 | 1976 | 230 | 1 | 1512 | 228 |
| Agent 6. | 1 | 1442 | 228 | 2 | 1933 | 227 | 2 | 1473 | 229 |
| Agent 7. | 1 | 1423 | 231 | 2 | 2072 | 229 | 2 | 1482 | 225 |
| Agent 8. | 2,01 | 1441 | 231 | 3,01 | 2069 | 231 | 1 | 1496 | 230 |
| Agent 9. | 1 | 1461 | 232 | 1,98 | 2021 | 228 | 1 | 1499 | 231 |
| Agent 10. | 1 | 1479 | 233 | 2 | 2077 | 232 | 1 | 1550 | 233 |
| Wartości średnie | **1,7** | **1390,3** | **228,5** | **2,499** | **1932,9** | **228,1** | **1,901** | **1473,1** | **227,6** |

Powyższa tabela (Tabela 2) przestawia symulacje dokonaną w identycznym środowisku jak Tabela 1, jednak do agentów zostały dołączone komponenty poprawiające jakość generowanych ścieżek. Zestawiając obydwie tabele, można dojść do wniosku, że narzut czasowy związany z wykorzystaniem skryptów poprawiających jakość jest niewielki - rzędu ≤ 0,4[ms]. Natomiast obliczenia ścieżki w metodzie Manhattan wymagały odwiedzin ponad 400 węzłów więcej.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 5,399 | 4887 | 452 | 8,011 | 7389 | 452 | 7,473 | 5839 | 452 |

Tabela 3 przestawia identyczną symulację jak w tabeli 1, jednak algorytm ma teraz do przeszukania 4 razy więcej węzłów niż w poprzednim wypadku. Można zauważyć, że liczba odwiedzonych węzłów wzrosła znacząco, a przypadku metody Manhattan przyrost wyniósł nawet 5457 węzłów. Czasowo najlepsze wyniki uzyskała heurystyka eukidesowa, a najdłużej wykonywał się algorytm używając heurystyki Manhattan.

Do agentów poruszającej się na omawianej mapie dodano skrypty modyfikacyjne. Tabela 4 przedstawia wyniki badań przeprowadzonych na grafie o rozmiarach 100x100. Zastosowanie skryptów modyfikacyjnych obciążyło czasowo najbardziej heurystykę diagonal Manhattan i uzyskała najgorszy wynik ze wszystkich heurystyk. Pomimo tego heurystyka Manhattan w dalszym ciągu odwiedza największą liczbę węzłów.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 8,99 | 4887 | 452 | 10,96 | 7389 | 452 | 12 | 5839 | 452 |
| Agent 2. | 3,99 | 5097 | 453 | 7,98 | 7578 | 453 | 7 | 5992 | 453 |
| Agent 3. | 6 | 5196 | 456 | 8,98 | 7771 | 454 | 10,01 | 6089 | 457 |
| Agent 4. | 6 | 5379 | 457 | 8,99 | 7804 | 456 | 14,01 | 6120 | 458 |
| Agent 5. | 7 | 5534 | 460 | 9,97 | 8014 | 459 | 16,01 | 6225 | 458 |
| Agent 6. | 7 | 5663 | 465 | 7,98 | 7979 | 458 | 11 | 6890 | 461 |
| Agent 7. | 6,99 | 5739 | 461 | 10,97 | 8542 | 466 | 9,01 | 6425 | 462 |
| Agent 8. | 5,99 | 5839 | 470 | 10,98 | 8815 | 464 | 16,01 | 6376 | 472 |
| Agent 9. | 7 | 5897 | 475 | 12,04 | 8914 | 470 | 10 | 6426 | 482 |
| Agent 10. | 7,01 | 5989 | 464 | 10,98 | 8832 | 473 | 8,98 | 6457 | 471 |
| Wartości średnie: | **6,597** | **5522** | **461,3** | **9,983** | **8163,8** | **460,5** | **11,403** | **6283,9** | **462,6** |

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 21,84 | 20013 | 901 | 39,063 | 31313 | 901 | 44,892 | 24318 | 901 |

Tabela 5 przedstawia wyniki badań mapy o rozmiarach 200x200. Jest to największy graf na jakim zostaną przeprowadzone testy. Porównując go z Tabelą 1 można zauważyć, że czasy obliczeń są kilkunastokrotnie większe. Zwiększenie rozmiaru mapy spowodowało, że heurystyka diagonal Manhattan osiągnęła najdłuższy czas wykonania porównując wyniki z eksperymentem przedstawionym w tabeli x.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 32,96 | 20013 | 901 | 34,96 | 31313 | 901 | 57,96 | 24318 | 901 |
| Agent 2. | 42,99 | 20311 | 904 | 34,84 | 32220 | 904 | 32,52 | 24278 | 904 |
| Agent 3. | 22,01 | 20508 | 907 | 37,93 | 32971 | 905 | 61,99 | 24370 | 907 |
| Agent 4. | 28,12 | 20874 | 912 | 47,93 | 34122 | 912 | 32,97 | 24673 | 908 |
| Agent 5. | 27,21 | 21184 | 911 | 54,88 | 34914 | 915 | 48,9 | 24795 | 915 |
| Agent 6. | 27,73 | 21338 | 915 | 41,88 | 34538 | 916 | 55,92 | 24741 | 920 |
| Agent 7. | 37,08 | 21628 | 915 | 66,8 | 35716 | 916 | 62,05 | 24731 | 920 |
| Agent 8. | 43,03 | 22076 | 919 | 49,96 | 34737 | 924 | 40 | 25007 | 924 |
| Agent 9. | 40,98 | 22506 | 929 | 97,97 | 36200 | 928 | 60,03 | 25008 | 926 |
| Agent 10. | 52,08 | 22758 | 933 | 81,05 | 35270 | 935 | 41,99 | 25159 | 940 |
| Wartości średnie: | **35,419** | **21319,6** | **914,6** | **54,82** | **34200,1** | **915,6** | **49,433** | **24708** | **916,5** |

Dodając do mapy nr 2 skrypty modyfikujące można zauważyć wzrost czasu obliczeń. W przypadku heurystyki eukidesowej wzrost ten zwiększył się o 14[ms], Manhattan o 15[ms]. Natomiast heurystyka diagonal Manhattan wykonała się wolnej tylko o 5[ms]

Analizując powyższe wyniki można dojść do wniosku, że najlepsze wyniki osiągnęła heurystyka wykorzystująca odległość eukidesową.

Jest pierwsza z czterech omawianych map i posiada prostą strukturę.

**Mapa 2**



.

Tabela 7 przedstawia wyniki działania algorytmu z różnymi heurystykami. Porównując wyniki z pierwszą mapą można zauważyć, że osiągnięte czasy są podobne. Podobnie jak w mapie nr 1 heurystyka Manhattan dokonuje przeglądu największej liczby węzłów. Jednak czasowo najszybciej wykonał się algorytm korzystający z heurystyki diagonalnej Manhattan, tym samym odwiedzając najmniejszą liczbę węzłów.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 1,7 | 1348 | 181 | 1,827 | 1537 | 181 | 1,495 | 1284 | 181 |

W poniższej tabeli (Tabela 8) przedstawiono wyniki działania algorytmu ze skryptami modyfikującymi. Porównując symulacje z poprzednią heurystyka Diagonal Manhattan osiągnęła lepszy wynik. Przypadek testowy wykorzystujący odległość eukidesową jest minimalnie gorszy. Natomiast proces odnajdywania ścieżki wykorzystującego heurystykę Manhattan wykonał się najwolniej przeglądając również największą liczbę węzłów.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 5 | 1348 | 181 | 6 | 1537 | 181 | 5 | 1284 | 181 |
| Agent 2. | 2 | 1418 | 181 | 1 | 1594 | 181 | 1 | 1392 | 181 |
| Agent 3. | 1 | 1426 | 181 | 2 | 1582 | 181 | 2 | 1337 | 181 |
| Agent 4. | 2 | 1491 | 184 | 1,96 | 1696 | 181 | 1 | 1488 | 183 |
| Agent 5. | 1 | 1484 | 183 | 1,99 | 1698 | 184 | 2 | 1492 | 183 |
| Agent 6. | 1 | 1485 | 186 | 3,03 | 1684 | 184 | 1 | 1486 | 184 |
| Agent 7. | 2 | 1526 | 184 | 2 | 1774 | 183 | 1 | 1507 | 185 |
| Agent 8. | 1 | 1525 | 184 | 3 | 1751 | 186 | 1 | 1538 | 186 |
| Agent 9. | 2 | 1528 | 188 | 3 | 1751 | 185 | 1 | 1541 | 183 |
| Agent 10. | 1,99 | 1575 | 188 | 1 | 1801 | 185 | 2 | 1577 | 187 |
| Wartości średnie: | **1,899** | **1480,6** | **184** | **2,498** | **1686,8** | **183,1** | **1,7** | **1464,2** | **183,4** |

Wyniki eksperymentu na mapie 2. zostały zawarte w tabeli 9. Można zauważyć, że czasy dla heurystyki liczonej odległością eukidesową oraz diagonalnym Manhattan czasowo zwracają podobne wartości. Jednak biorąc pod uwagę liczbę odwiedzonych węzłów to diagonalny typ tej heurystyki odwiedza ich najmniej.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas [ms] | Liczba odwiedzin | Długość | Czas [ms] | Liczba odwiedzin | Długość | Czas [ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 6,083 | 5943 | 363 | 10,5805 | 6917 | 363 | 6,296 | 5817 | 363 |

Dodanie skryptów modyfikacyjnych spowodowało to, że heurystyka, która osiągała do tej pory najlepsze wyniki czasowe ma największy czas wyszukania drogi. Wyniki tych badań przedstawia tabela 10. Heurystyka Manhattan osiągnęła w tym przypadku najmniejszy wynik czasowy.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas [ms] | Liczba odwiedzin | Długość | Czas [ms] | Liczba odwiedzin | Długość | Czas [ms] | Liczba odwiedzin | Długość |
| Agent 1. | 12 | 5943 | 363 | 10,99 | 6917 | 363 | 15 | 5817 | 363 |
| Agent 2. | 7 | 6057 | 364 | 8 | 7365 | 363 | 9,01 | 5910 | 363 |
| Agent 3. | 6,97 | 6138 | 367 | 7,98 | 7732 | 368 | 8 | 6021 | 370 |
| Agent 4. | 7,01 | 6207 | 372 | 10 | 8802 | 366 | 7,99 | 6245 | 367 |
| Agent 5. | 6,97 | 6451 | 374 | 11 | 8400 | 370 | 8,81 | 6348 | 369 |
| Agent 6. | 9 | 6496 | 375 | 9,93 | 8269 | 374 | 7,93 | 6469 | 370 |
| Agent 7. | 8,01 | 6581 | 380 | 11 | 8716 | 370 | 10 | 6545 | 375 |
| Agent 8. | 9 | 6532 | 376 | 11 | 8449 | 371 | 7 | 6553 | 370 |
| Agent 9. | 8 | 6625 | 377 | 11,04 | 8081 | 373 | 6 | 6739 | 374 |
| Agent 10. | 8,01 | 6731 | 379 | 10,97 | 8386 | 372 | 7 | 6724 | 378 |
| Wartości średnie: | **8,197** | **6376,1** | **372,7** | **10,191** | **8111,7** | **369** | **8,674** | **6337,1** | **369,9** |

Włącznie skryptów poprawiających jakość generowanych ścieżek obciążyło jednakowo heurystyki posiadające najlepsze wyniki czasowe. Heurystyka Manhattan osiągnęła przewidywalny najgorszy wynik czasowy oraz przetworzyła największą ilość węzłów.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 31,041 | 24109 | 716 | 39,121 | 29903 | 717 | 30,881 | 23906 | 716 |

Tabela x przedstawia wyniki symulacji dla mapy o rozmiarze 200x200. Wzrost obszaru przeszukiwań nie wyłonił najlepszej heurystyki. Wyniki osiągnięte przez heurystyki diagonalny Manhattan oraz eukidesową są porównywalne.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 31,75 | 24109 | 716 | 33,94 | 29903 | 717 | 30,67 | 23906 | 716 |
| Agent 2. | 32,93 | 24289 | 727 | 39,14 | 30188 | 751 | 24,98 | 24078 | 721 |
| Agent 3. | 34,94 | 24723 | 726 | 42,91 | 32157 | 732 | 24,99 | 24122 | 724 |
| Agent 4. | 35,05 | 24695 | 736 | 43,91 | 33426 | 731 | 30,99 | 24477 | 724 |
| Agent 5. | 32,95 | 24754 | 744 | 47,87 | 35708 | 741 | 28,99 | 24491 | 734 |
| Agent 6. | 32,98 | 25012 | 737 | 44,81 | 35113 | 751 | 46,99 | 24752 | 745 |
| Agent 7. | 34,96 | 25111 | 747 | 45,91 | 35889 | 752 | 48,98 | 24910 | 746 |
| Agent 8. | 35,99 | 25243 | 748 | 52,93 | 38486 | 749 | 30,96 | 25070 | 749 |
| Agent 9. | 34 | 25078 | 757 | 66,94 | 40690 | 752 | 32 | 25656 | 755 |
| Agent 10. | 35,94 | 25292 | 756 | 54,8 | 41450 | 769 | 33,98 | 25563 | 753 |
| Wartości średnie: | **34,149** | **24830,6** | **739,4** | **47,316** | **35301** | **744,5** | **33,353** | **24702,5** | **736,7** |

Wyniki kolejnego eksperymentu zostały umieszczone w tabeli x. Dodanie skryptów modyfikacyjnych spowodowało, że minimalnie lepsza od heurystyki eukidesowej okazała się heurystyka diagonal Manhattan. Natomiast najdłuższy czas wykonania osiągnęła heurystyka Manhattan.

Mapa 3(1)



Rysunek x przedstawia mapę z wygenerowaną ścieżką. Agenci rozpoczynają ruch z lewego górnego narożnika mapy. Warto zwrócić uwagę na konstrukcje mapy. Nie zawiera ona alternatywnych dróg – postacie podążając jedną ścieżką docierają do punktu docelowego. Algorytm sumarycznie ma w tym przypadku do przeszukania procentowo większy obszar. Mapa x jest pierwszą z map zawierającą bardziej skomplikowaną strukturę terenu.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 1,597 | 1163 | 117 | 1,596 | 1196 | 117 | 1,398 | 1042 | 117 |

Tabela x przedstawia wyniki symulacji na mapie 3. W tej symulacji heurystyka eukidesowa, która do tej pory osiągała najlepsze wyniki osiągnęła wynik podobny do najgorszej z dotychczas testowanych heurystyk. Najszybciej wykonała się heurystyka diagonalna Manhattan oraz odwiedziła najmniej węzłów.

W tabeli x zostały zaprezentowane wyniki dla mapy x przy włączonych modyfikatorach ścieżki. Narzut czasowy wynikający z zastosowania skryptów jest niewielki i wynosi on od 0,7[ms] do 1.5[ms].

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 5 | 1182 | 117 | 5 | 1250 | 117 | 5,01 | 1160 | 117 |
| Agent 2. | 0,99 | 1274 | 117 | 3 | 1572 | 117 | 2 | 1227 | 117 |
| Agent 3. | 3 | 1295 | 118 | 2 | 1611 | 117 | 1 | 1286 | 117 |
| Agent 4. | 2 | 1326 | 118 | 2 | 1438 | 119 | 2 | 1291 | 119 |
| Agent 5. | 1 | 1323 | 119 | 2 | 1710 | 119 | 1 | 1318 | 118 |
| Agent 6. | 2 | 1369 | 120 | 3 | 1683 | 120 | 1 | 1326 | 118 |
| Agent 7. | 3 | 1483 | 117 | 4,01 | 1518 | 118 | 1 | 1326 | 119 |
| Agent 8. | 3 | 1557 | 121 | 3 | 1630 | 119 | 3 | 1384 | 119 |
| Agent 9. | 3 | 1496 | 131 | 4,02 | 1770 | 119 | 2 | 1479 | 118 |
| Agent 10. | 2 | 1588 | 122 | 1 | 1558 | 122 | 2 | 1466 | 131 |
| Wartości średnie | **2,499** | **1389,3** | **120** | **2,903** | **1574** | **118,7** | **2,001** | **1326,3** | **119,3** |

Mapa numer x została powiększona do rozmiaru 100x100. Wartości w tabeli x wskazują na przewagę heurystyki diagonalnej Manhattan. Metoda Manhatan podobnie jak w pozostałych przypadkach przetwarza największą liczbę węzłów oraz posiada wysoką wartość czasu obliczeń.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 5,883333 | 4946 | 231 | 7,488 | 5568 | 231 | 5,291 | 4766 | 231 |

Wyniki symulacji przedstawione w tabeli x wskazują, że heurystyki euidesowa oraz diagonalna Manhattan wykonały się w podobnych czasach. W poprzedniej symulacji heurystyka diagonal Manhattan posiadała najkrótszy czas wykonania. Można więc dojść do wniosku, że dodanie skryptów modyfikacyjnych jest bardziej obciążające czasowo dla tej właśnie heurystyki.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 10 | 4980 | 231 | 10 | 5607 | 231 | 7,01 | 5097 | 231 |
| Agent 2. | 6,99 | 5173 | 231 | 8,99 | 7006 | 231 | 6 | 5211 | 231 |
| Agent 3. | 6,97 | 5215 | 234 | 11,01 | 7050 | 235 | 7,01 | 5314 | 235 |
| Agent 4. | 7 | 5447 | 233 | 8,98 | 7182 | 235 | 7,98 | 5522 | 234 |
| Agent 5. | 7 | 5535 | 234 | 9,96 | 6983 | 235 | 7,9 | 5594 | 236 |
| Agent 6. | 7,99 | 5747 | 237 | 10,01 | 6947 | 240 | 7 | 5667 | 236 |
| Agent 7. | 7 | 5989 | 235 | 10,01 | 7018 | 241 | 7 | 6034 | 234 |
| Agent 8. | 7 | 6321 | 238 | 18,02 | 7676 | 234 | 7,9 | 6362 | 241 |
| Agent 9. | 5,98 | 6534 | 260 | 13,02 | 7801 | 236 | 7,01 | 6382 | 260 |
| Agent 10. | 7 | 6072 | 245 | 15,02 | 7174 | 239 | 7,9 | 6420 | 241 |
| Wartości średnie | **7,293** | **5701,3** | **237,8** | **11,502** | **7044,4** | **235,7** | **7,271** | **5760,3** | **237,9** |

Mapę poddawaną analizie zwiększono do rozmiaru 200x200. Analizując wyniki można zauważyć stopniową przewagę heurystyki diagonalnej Manhattan. W poprzednich badanych mapach o rozmiarze 200x200 heurystyka ta uzyskiwała większe czas obliczeń.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 42,669 | 20224 | 461 | 67,367 | 23799 | 461 | 40,816 | 20064 | 461 |

Do agentów poruszających się po mapie x dodano skrypty modyfikujące. Wyniki symulacji poszukiwania drogi znajdują się w tabeli x. Warto zwrócić uwagę, że dodanie skryptów modyfikujących najmniej obciążyło heurystykę diagonal Manhattan. Natomiast pozostałe heurystyki wykonały się o 15[ms](eukidesowa) i 11[ms] (Manhattan) dłużej.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 28,87 | 20224 | 461 | 39,99 | 23799 | 461 | 27,98 | 20864 | 461 |
| Agent 2. | 34,87 | 20420 | 467 | 31,92 | 25941 | 466 | 40,04 | 21023 | 465 |
| Agent 3. | 37,95 | 20613 | 466 | 73,93 | 28172 | 470 | 64,14 | 21237 | 467 |
| Agent 4. | 54,92 | 21266 | 473 | 81,08 | 29453 | 471 | 47,05 | 21434 | 467 |
| Agent 5. | 60,95 | 21508 | 472 | 70,89 | 28984 | 475 | 42,98 | 21821 | 474 |
| Agent 6. | 64,87 | 21660 | 476 | 80,92 | 28910 | 480 | 33,05 | 22163 | 475 |
| Agent 7. | 68,94 | 22038 | 475 | 73,94 | 29667 | 477 | 69,08 | 22393 | 477 |
| Agent 8. | 77,01 | 22968 | 485 | 78,04 | 30025 | 489 | 35,01 | 22856 | 483 |
| Agent 9. | 60,95 | 24997 | 486 | 88,07 | 30739 | 489 | 39,86 | 22958 | 487 |
| Agent 10. | 80,99 | 25768 | 487 | 127,57 | 43759 | 489 | 47,04 | 22084 | 488 |
| Wartości średnie | **57,032** | **22146,2** | **474,8** | **78,4844** | **29944,9** | **476,7** | **44,623** | **21883,3** | **474,4** |

Mapa 4

Ostatnią analizowaną mapą jest mapa posiadająca strukturę labiryntu. Agencie zostali umieszczani w … rogu, a ich celem jest dotarcie do punktu końcowego znajdującego się w … rogu mapy.

Wyniki symlacji na przedstwionej mapie o rozmiar 50x50 zostały przedstawione w tabeli x. Najwięszy czas na znalezienie ścieżki uzyskała heurystyka Manhattan. Natomiast porównywalne wyniki uzyskały dwie pozostałe heurystyki,

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agenci komputerowi: | 3,502 | 840 | 67 | 4,102 | 977 | 67 | 3,501 | 848 | 67 |

Dodanie skryptów modyfikacyjnych w przypadku takich rozmiarów mapy obciążyło czasowo heurystykę diagonalną Manhattan. W poprzedniej symulacji (tabela x) wyniki czasowe były identyczne. Uzyskała ona gorszy czas od najlepszej w tym przypadku heurystyki eukidesowej.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 50x50 | | | | | | | | |
| Modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 14,01 | 911 | 67 | 13,01 | 1003 | 69 | 14,01 | 869 | 67 |
| Agent 2. | 2 | 958 | 69 | 5 | 1633 | 67 | 3 | 989 | 69 |
| Agent 3. | 2 | 981 | 69 | 4,01 | 1679 | 69 | 2 | 1052 | 67 |
| Agent 4. | 3 | 982 | 67 | 7 | 1562 | 67 | 4 | 1020 | 69 |
| Agent 5. | 2 | 1023 | 69 | 3,01 | 1386 | 71 | 2 | 1077 | 69 |
| Agent 6. | 3 | 1051 | 68 | 4 | 1229 | 68 | 4 | 1074 | 67 |
| Agent 7. | 2 | 1037 | 69 | 4 | 1772 | 68 | 3 | 1080 | 69 |
| Agent 8. | 5 | 1041 | 69 | 6 | 1706 | 69 | 3 | 1110 | 68 |
| Agent 9. | 3 | 1076 | 70 | 4 | 1759 | 71 | 3 | 1113 | 69 |
| Agent 10. | 2 | 1083 | 70 | 4 | 1680 | 68 | 2 | 1150 | 67 |
| Średnia: | **3,801** | **1014,3** | **68,7** | **5,403** | **1540,9** | **68,7** | **4,001** | **1053,4** | **68,1** |

Kolejna symulacja została przeprowadzona na mapie powiększonej do rozmiaru 100x100. W tym przypadku z powodu braku skryptów modyfikujących wyniki heurystyk eukidesowej i diagonalnej Manhattan są zbliżone.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Wartości średnie | 11,297 | 3463 | 132 | 15,415 | 4294 | 132 | 11,911 | 3592 | 132 |

Odnosząc się do poprzednich wyników można było się spodziewać, że dodanie skryptów modyfikujących obciąży czasowo bardziej heurystykę diagonalną Manhattan i osiągnie gorszy czas wykonania. Warto zwrócić uwagę, że w przypadku zwiększania się rozmiaru mapy heurystyka ta osiąga najlepsze wyniki spośród pozostałych heurystyk.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 100x100 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 22 | 3538 | 131 | 25,01 | 4351 | 131 | 22,04 | 3640 | 131 |
| Agent 2. | 10,01 | 3662 | 133 | 32,04 | 9000 | 132 | 10,01 | 3802 | 133 |
| Agent 3. | 10,02 | 3860 | 137 | 31,04 | 9246 | 137 | 10,01 | 3888 | 132 |
| Agent 4. | 12,01 | 3783 | 135 | 35,09 | 8991 | 132 | 12,02 | 4120 | 137 |
| Agent 5. | 12,01 | 3927 | 138 | 32,05 | 9220 | 134 | 12,01 | 3940 | 135 |
| Agent 6. | 11,02 | 3974 | 134 | 31 | 8790 | 137 | 11 | 4004 | 137 |
| Agent 7. | 10,01 | 4030 | 138 | 40,03 | 11115 | 133 | 11,02 | 4094 | 138 |
| Agent 8. | 13,01 | 4040 | 138 | 39,31 | 10753 | 138 | 12,01 | 4136 | 134 |
| Agent 9. | 11,01 | 4154 | 136 | 45,13 | 11343 | 137 | 11,01 | 4174 | 136 |
| Agent 10. | 10,95 | 4177 | 140 | 40,05 | 10971 | 139 | 11,02 | 4266 | 140 |
| Wartości średnie: | **12,205** | **3914,5** | **136** | **35,075** | **9378** | **135** | **12,215** | **4006,4** | **135,3** |

Tabela x zawiera wyniki badań przeprowadzonych dla największych rozmiarów mapy. Najlepszy wynik czasowy osiągnęła heursytka diagonalna Manhattan, co potwierdzaj jej efektywność na mapach o dużych rozmiarach. Heurystyka eukidesowa osiągnęła o 6[ms] gorszy czas w porównaniu najleszego wyniku dla omawianej symulacji.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | brak modyfikatora | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Wartości średnie: | 46,943 | 14315 | 263 | 66,782 | 19087 | 263 | 40,096 | 10869 | 263 |

Dodanie skryptów modyfikacyjnych do mapy x pozwoliło na uzyskanie wyników umieszczonych w tabeli x. Metoda Manhattan uzyskała najgorszy wynik jej czas obliczeń jest 2.5 razy większy od pozostałych heurystyk. Najlepszy wynik czasowy uzyskała heurystyka diagonalna Manhattan. Wykonała się szybciej o 5[ms] szybciej w porównaniu do heurystyki eukidesowej.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Rozmiar grafu: | 200x200 | | | | | | | | |
| Włączony modyfikator: | Funnel, Alternative Path | | | | | | | | |
| Heurystyka: | Euclidean | | | Manhattan | | | Diagonal Manhattan | | |
|  | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość | Czas[ms] | Liczba odwiedzin | Długość |
| Agent 1. | 59,04 | 14594 | 263 | 79,06 | 19510 | 263 | 41,05 | 15202 | 263 |
| Agent 2. | 44,07 | 15234 | 264 | 76,46 | 21256 | 264 | 47,97 | 15406 | 264 |
| Agent 3. | 50,08 | 16255 | 266 | 83,1 | 24827 | 272 | 50,06 | 15950 | 265 |
| Agent 4. | 51,03 | 16178 | 272 | 89,09 | 25021 | 264 | 53,02 | 15700 | 272 |
| Agent 5. | 57,07 | 16462 | 269 | 97,11 | 25559 | 268 | 50,04 | 16165 | 268 |
| Agent 6. | 58,06 | 16316 | 274 | 90,8 | 25248 | 272 | 51,74 | 16608 | 273 |
| Agent 7. | 48,05 | 16109 | 271 | 178,15 | 48113 | 270 | 42,06 | 16066 | 272 |
| Agent 8. | 52,05 | 16499 | 273 | 183,2 | 50231 | 274 | 43,21 | 16260 | 275 |
| Agent 9. | 52,01 | 16299 | 274 | 194,15 | 50208 | 275 | 46,05 | 16249 | 273 |
| Agent 10. | 57,04 | 16494 | 274 | 194,11 | 50808 | 270 | 45,05 | 16214 | 277 |
| Wartości średnie | **52,85** | **16044** | **270** | **126,523** | **34078,1** | **269,2** | **47,025** | **15982** | **270,2** |

## Eksperyment 2 – optymalizacja

W tym eksperymencie zostanie przeanalizowany ten sam zestaw map. Na każdej z map zostanie wywołany skrypt z algorytmem JPS. Rozmiar map będzie zwiększany identycznie jak w poprzedniej analizie. Dzięki temu jest możliwe dokonanie porównania czasowego algorytmu przed i po optymalizacji. Eksperyment dokonuje analizy heurystyki diagonal Manhattan.

Poniżej zostaną przedstawione tabele z wynikami. Eksperyment był przeprowadzany dla dziesięciu agentów, dla każdego z nich został zapisany wynik obliczeń algorytmu znajdującego ścieżkę do celu. Wynik czasowy następnie został uśredniony. W ostatnim wierszu został zapisany uśredniony wynik algorytmu bez optymalizacji otrzymany w eksperymencie numer 1 dla heurystyki diagonalnej Manhattan oraz przy wyłączonych skryptach modyfikacyjnych.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mapa 1** | | | |
| Rozmiar: | 200x200 | 100x100 | 50x50 |
| Agent nr | Rezultat czasowy [ms] | | |
| 1 | 23,51 | 9,77 | 3,34 |
| 2 | 26,61 | 6,54 | 1,42 |
| 3 | 26,19 | 6,57 | 1,53 |
| 4 | 26,48 | 6,78 | 1,43 |
| 5 | 23,13 | 6,64 | 1,41 |
| 6 | 25,8 | 8,74 | 1,41 |
| 7 | 26,3 | 4,86 | 1,41 |
| 8 | 25,84 | 5,13 | 1,41 |
| 9 | 25,94 | 6,86 | 1,82 |
| 10 | 25,95 | 6,95 | 1,05 |
| Średnia | **25,575** | **6,884** | **1,623** |
| Średnia -poprzedni eksperyment | 44,89 | 7,473 | 1,7 |

Tabela x przedstawia wyniki eksperymentu przeprowadzanego na mapie pierwszej. Można zauważyć, że dla małych i średnich map wynik czasowy jest porównywalny. W przypadku mapy o największym rozmiarze algorytm JPS wykonuje się o 19[ms] szybciej.

W analizie mapy numer 2 przedstawionej w tabeli x można zauważyć, że algorytm JPS zwraca gorszy wynik niż rezultat otrzymany bez optymalizacji . W przypadku mapy średniej wielkości wynik czasowy jest o 1[ms] lepszy. Warto zauważyć, że optymalizacja uzyskuje najlepsze przyśpieszenie dla dużych map.

|  |  |  |  |
| --- | --- | --- | --- |
| **Map 2** | | | |
| Rozmiar: | 200x200 | 100x100 | 50x50 |
| Agent nr | Rezultat czasowy [ms] | | |
| 1 | 19,84 | 8,52 | 4,72 |
| 2 | 13,85 | 4,82 | 1,47 |
| 3 | 18,98 | 5,37 | 1,66 |
| 4 | 13,98 | 4,82 | 1,46 |
| 5 | 13,9 | 4,84 | 1,46 |
| 6 | 14,35 | 5,49 | 1,67 |
| 7 | 14,28 | 4,81 | 1,62 |
| 8 | 13,93 | 9,74 | 1,03 |
| 9 | 14,45 | 4,82 | 6,25 |
| 10 | 19,41 | 3,41 | 1,59 |
| Średnia | **15,697** | **5,664** | **2,293** |
| Średnia -poprzedni eksperyment | 30,88 | 6,29 | 1,495 |

Przed analizą mapy x warto zwrócić uwagę na strukturę tej mapy. Posiada ona proste ścieżki. Ta właściwość pozwoli algorytmowi JPS na wstawienie mniejszej liczby punktów skoku – jeden będzie zaczynał się na początku prostego odcinka drogi, a drugi na jego końcu. W wyniku tego uzyskane czasy dla rozmiarów 50x50 i 100x100 są mniejsze od tych z przed optymalizacji. Analizując mapę o rozmiarze 200x200 algorytm osiągnął mniejszy czas zestawiając go z wynikiem z poprzedniego eksperymentu, jednak nie jest on tak niski jak w przypadku mniejszych map.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mapa 3** | | | |
| Rozmiar: | 200x200 | 100x100 | 50x50 |
| Agent nr | Rezultat czasowy [ms] | | |
| 1 | 35,24 | 3,13 | 3,26 |
| 2 | 32,5 | 0,68 | 0,5 |
| 3 | 29,67 | 0,6 | 0,56 |
| 4 | 29,02 | 0,29 | 0,48 |
| 5 | 33,3 | 0,29 | 0,34 |
| 6 | 34,94 | 0,3 | 0,46 |
| 7 | 35,5 | 0,29 | 0,47 |
| 8 | 39,22 | 0,28 | 0,46 |
| 9 | 29,43 | 0,59 | 0,48 |
| 10 | 34,65 | 0,37 | 0,35 |
| Średnia | **33,347** | **0,682** | **0,736** |
| Średnia -poprzedni eksperyment | 40,81 | 5,291 | 1,398 |

Wyniki badań nad mapą x znajdują się w tabeli x. Mapa x zawiera skomplikowaną strukturę, więc uzyskane rezultaty czasowe są większe od poprzednio analizowanych. Zestawiając wyniki dla poszczególnych map można zauważyć, że dla każdego rozmiaru mapy optymalizacja skraca czas obliczeń. Przyśpieszenie czasowe zwiększa się wraz z rozmiarem mapy.

|  |  |  |  |
| --- | --- | --- | --- |
| **Mapa 4** | | | |
| Rozmiar: | 200x200 | 100x100 | 50z50 |
| Agent nr | Rezultat czasowy [ms] | | |
| 1 | 31,53 | 7,54 | 5,91 |
| 2 | 22,09 | 7,31 | 1,65 |
| 3 | 20,38 | 7,26 | 1,87 |
| 4 | 22,54 | 7,3 | 2,43 |
| 5 | 20,76 | 7,7 | 2,14 |
| 6 | 22,6 | 7,68 | 2,16 |
| 7 | 26,99 | 7,56 | 4,59 |
| 8 | 22,34 | 7,44 | 2,22 |
| 9 | 26,96 | 7,61 | 2,15 |
| 10 | 22,57 | 7,44 | 2,14 |
| Średnia | **23,876** | **7,484** | **2,726** |
| Średnia -poprzedni eksperyment | 40,09 | 11,91 | 3,501 |

# Podsumowanie i wnioski

Celem pracy było dokonanie analizy technik nawigacji stosowanych w grach komputerowych. Powszechnie stosowanym algorytmem do nawigacji postaci jest algorytm A\*. Wykorzystana biblioteka pomogła w analizie wydajności tego algorytmu ze względu na stosowaną funkcję heurystyczną. Dodatkowo dokonano implementacji algorytmu JPS przyśpieszającej działanie wyszukiwania ścieżki.

Cele pracy zostały zrealizowane. Na potrzeby analizy w zostały przeprowadzone pomiary czasu wykonania algorytmu oraz ilości odwiedzonych węzłów w grafie. Badanie te zostały przeprowadzony na czterech różnych mapach. Mapy te różniły się w swoje strukturze. Pierwsze z nich posiadały mniej skomplikowaną strukturę, a ostatnia miała formę labiryntu. Dodatkowo obszar poszukiwań dla każdej mapy był zwiększany z rozmiaru 50x50 do 100x100 oraz następnie do 200x200.

Rozwój gier komputerowych powoduje, że twórcy dostarczają graczom rozmaite światy rozgrywki. Światy te mogą być zróżnicowane pod względem struktury oraz wielkości. Z reguły światy te zawierają postacie sterowane przez komputer. Postacie te wyposażone są moduł nawigacyjny zawierający algorytm odnajdujący ścieżkę. Dzięki niemu są one w stanie poruszać się po mapie. Taki algorytm jest zwykle algorytmem heurystycznym. Warto zatem dokonać analizy światów rozgrywki i wybrać dla każdego z nich najlepszą heurystykę pod względem czasu wykonania. W branży gier komputerowych niezwykle popularny jest algorytm A\*, ze względu na szybkość oraz duże możliwości optymalizacyjne.

Badania nad heurystykami wykazały, że dla map posiadających prostą strukturę i niewielkie rozmiary najlepszym rozwiązaniem jest zastosowanie heurystyki eukidesowej. Uzyskuje ona najmniejszy czas przeznaczony na odnalezienie ścieżki. Odnośnie skryptów modyfikujących, które poprawiają jakość ścieżek, warto wspomnieć o tym, że dla map o prostej strukturze narzut czasowy związany z zastosowaniem tych skryptów nie jest wysoki (od 0,2[ms] do 3[ms]). Jednak w przypadku map o dużych rozmiarach (200x200) zastosowanie skryptu modyfikującego obciążyło obliczeniowo heurystykę eukidesową aż o 15[ms]. Podczas gdy obciążenie czasowe heurystyki diagonalnej Manhattan wyniosło zaledwie 5[ms]. W tym przypadku programista powinien się zastanowić nad wyborem heurystyki, ponieważ dla większych map obciążenie czasowe heurystyki eukidesowej może spowodować spadek efektywności algorytmu na korzyść jednej z pozostałych heurystyk.

Dalsza część badań dotyczyła map o złożonej strukturze (np. mapa w kształcie labiryntu) . Wyniki badań wykazały, że dla małych i średnich map zarówno heurystyka eukidesowa jak i diagonalna Manhattan uzyskują podobne wyniki czasowe oraz przetwarzają zbliżoną liczbę węzłów w grafie. W tym przypadku narzut czasowy spowodowany wykorzystaniem skryptów modyfikujących spowolnił wykonanie wyżej wymienionych heurystyk od 0,3[ms] do 3[ms]. Natomiast heurystyka Manhattan jak do tego pory uzyskała najgorszy rezultat czasowy, co więcej narzut wynikający z zastosowania skryptów wyniósł, aż 20[ms]. W przypadku map o dużych rozmiarach zdecydowanie najlepiej zastosować heurystykę diagonalną Manhattan. Uzyskała ona najkrótszy czas obliczeń. Zastosowanie skryptów modyfikujących, co prawda spowolniło czas wykonania algorytmu, jednak mimo to algorytm wykonał się najszybciej.

Ostatni eksperyment badawczy wykazał, że zastosowanie optymalizacji JPS przyśpiesza wykonanie algorytmu od 7% do 80%. Rozrzut przyśpieszenia wynika z tego, że dla małych map było ono niewielkie. Natomiast w przypadku dużych map wynosiło ono około 50%. Przyśpieszenie równe 80% zostało zanotowane dla jednej z map o skomplikowanej strukturze. Struktura ta była o tyle specyficzna, że posiadała rozległą przestrzeń. Dzięki czemu algorytm JPS mógł wykonać duże „skoki” na mapie pomijając duże obszary, tym samym wykluczając je z dalszej analizy. Podsumowując optymalizacja przyniosła bardzo dobre efekty i warto podjąć się implementacji algorytmu JPS, który przyśpieszy proces nawigacji postaci, a zwłaszcza w przypadku dużych map.

Zagadnienie nawigacji w grach komputerowych jest ściśle związane z problemem odnajdowania ścieżki. Problem ten przedstawiono na łamach tej pracy. Badania wykazały, że warto zwrócić uwagę, na to czy wybrane rozwiązanie będzie efektywne. Rozwiązania zajmujące długi okres czasu są nie do przyjęcia przy obecnym poziomie gier komputerowych. Gracz jako odbiorca może szybko się zniechęcić grą, w której system nawigacji postaci działa wolno postaci, co może rzutować na sukces, bądź porażkę danej gry komputerowej. Na efektywność rozwiązania mają wpływ czynniki takie jak: rozmiar i poziom skomplikowania mapy, zastosowana funkcja heurystyczna oraz dodatkowe przetwarzanie drogi mające na celu poprawienie jej jakości (np. wygładzanie ścieżek). Istotnym elementem systemu nawigacji jest jego optymalizacja, która może znacznie przyśpieszyć czas wyszukiwania ścieżki.

# Bibliografia