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, październik 2013

Spis treści

Spis rysunków

Spis tabel

1 Wstęp

Nawigacjach w grach komputerowych jest od wielu lat rozwijana i ulepszana oraz stanowi bardzo ważny element gier komputerowych. Rozwiązania dostarczane przez branże gier komputerowych 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.

1.1 Cel pracy

Celem niniejszej pracy jest przebadanie algorytmów nawigacji stosowanych w grach komputerowych. Nawigacja w grach komputerowych w dużym stopniu polega na problemie 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* i to on będzie przedmiotem badań w pracy. Jest to algorytm heurystyczny, posiadający duże możliwości optymalizacyjne. W pracy zostanie przebadane zastosowanie jednej z obiecujących optymalizacji algorytmu A* jakim jest algorytm Jump Point Search.

Na potrzeby badań zostanie zaprojektowana aplikacja będąca środowiskiem umożliwiającym przeprowadzanie symulacji odnajdywania ścieżki. Do aplikacji będzie można załadować dowolny model środowiska gry, a następnie przeprowadzić na nim zestaw eksperymentów.

1.2 Przewodnik po pracy

Rozdział drugi zawiera wprowadzenie w zagadnienia związane ze Sztuczną Inteligencją w grach komputerowych. Zawarta w nim jest krótka historia oraz produkcje będące kamieniami milowymi w rozwoju Sztucznej Inteligencji w grach. W rozdziale przedstawiono także przykładowe techniki symulacji Sztucznej Inteligencji powszechnie stosowane w grach komputerowych.

W rozdziale trzecim omówiono pojęcie grafu z punktu widzenia programisty grafiki komputerowej oraz opisano algorytmy związane 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 środowisk. Rozdział ten zawiera również eksperymenty, w których dokonano badań efektywności algorytmu A* ze względu na różne czynniki oraz jego optymalizacji.

W rozdziale szóstym ujęto wnioski i podsumowanie pracy.

2 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). Sztuczną Inteligencje w grach nie można utożsamiać ze Sztuczną Inteligencją znaną z badań naukowych. W grach Sztuczna Inteligencja opiera się często na prostych algorytmach, których celem jest jedynie symulacja inteligencji. Główne zastosowania AI w grach to [4]:

- 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.

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.

2.1 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 [4].

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 [7]. 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.

Do określenia przeciwnika komputerowego często w grach komputerowych używa się pojęcia agenta lub agenta komputerowego.

Pierwszą grą, w której gracz posiadał komputerowych 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[4].

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 rozdziale 2.2.

W latach 2001 i 2002 powstały dwie gry, które sprawiły, że gracze z niedowierzaniem patrzyli na ich 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.

2.2 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 [4], 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:

- 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:

- 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.

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.

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

Nowatorskie koncepcje wykorzystane w grze:

- 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.

2.3 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 znajdującego się w środowisku gry 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 w środowisku gry. W skrócie A* sprawdza, po każdym wykonanym ruchu agenta wszystkie możliwe kierunki dalszej trasy i wybiera 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 liście to możliwe posunięcia/decyzje dostępne w aktualnej sytuacji.

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". Logika rozmyta wykorzystywana jest często do odwzorowania emocji w grach (np. "przyjacielski", "obojętny", "wrogi"). Za jej 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 nad 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 umieszczone w środowisku gry [3].

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. Spowoduje ona rozwój świata wraz ze wzrostem doświadczenia gracza, które uzyskał w dotychczasowym procesie rozgrywki [8].

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 [8].

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.

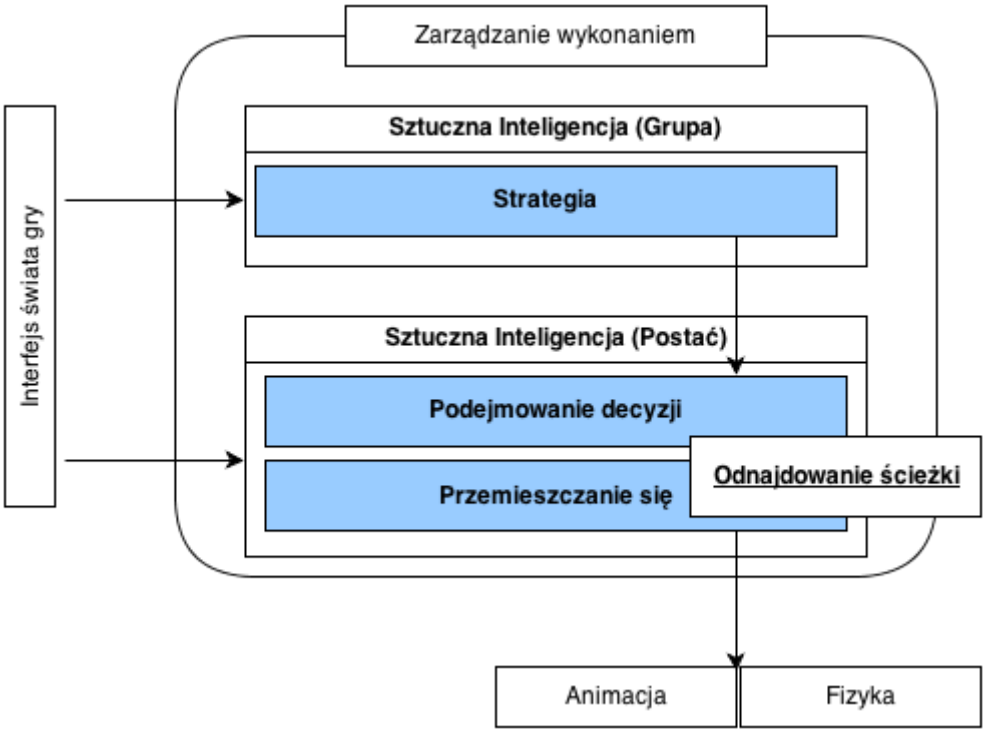
3 Znajdowanie ścieżki

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 w środowisku gry 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 w każdym w silniku gry posiadającym moduł odpowiedzialny za sztuczną inteligencję. W przedstawionym na rysunku 1 modelu rola odnajdywania ścieżki znajduje się pomiędzy modułami odpowiedzialnymi za podejmowanie decyzji oraz poruszanie się postaci. Często odnajdywanie ścieżki jest 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.



Rysunek 1 Przykładowy model sztucznej inteligencji w grach [1]

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 środowiska gry. Wymaga on, aby środowisko gry było reprezentowane przez odpowiednią strukturę danych - jest to zwykle graf ważony o nieujemnych wagach. Ten rozdział wprowadza pojęcie struktury danych grafu. Następnie 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 przyjrzeć się kwestii zmiany geometrii środowiska gry na dane, które mogą zostać przetworzone przez moduł odpowiedzialny za znajdowanie ścieżki. Warto też wspomnieć o istnieniu wielu dziesiątek przydatnych wariacji podstawowego algorytmu A*.

W pracy będzie używane pojęcie efektywności, rozumianego jako efektywność czasowa algorytmu.

3.1 Graf odnajdowania ścieżki

Algorytm A* oraz algorytm Dijkstry nie może pracować bezpośrednio na geometrii, z której zbudowane jest środowisko gry. Jednak każde środowisko gry może

zostać przekształcona do uproszczonej wersji 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 skierowanych grafów o nieujemnych wagach.

3.1.1 Definicja grafu

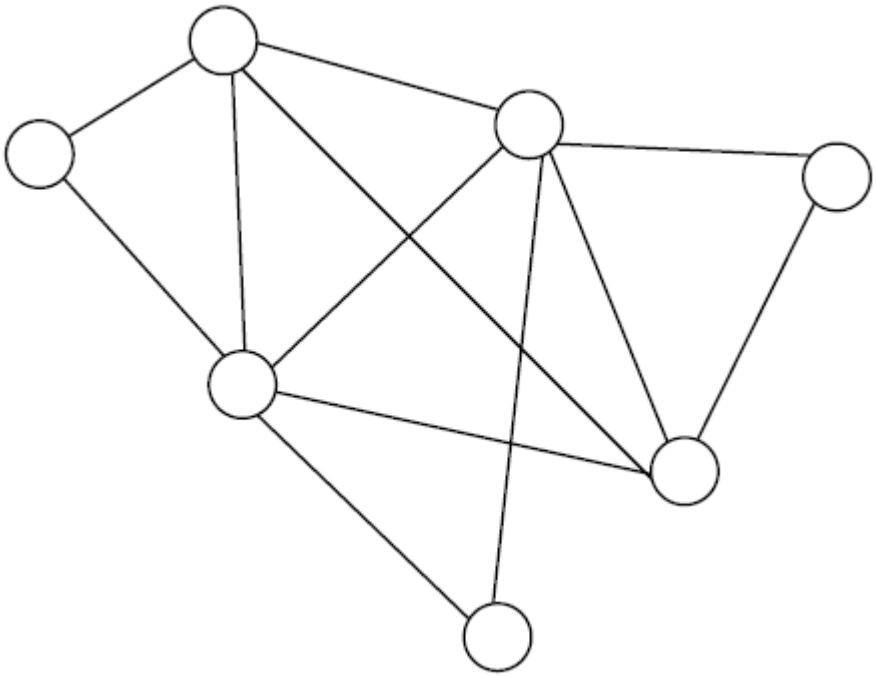
Graf jest matematyczną strukturą często reprezentowaną przy pomocy schematu graficznego. Graf składa się z dwóch rodzajów elementów. Są to węzły często rysowane jako punkty lub koła w schemacie grafu oraz krawędzie będące połączeniami węzłów przedstawiane w postaci odcinków (rysunek 2).

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 środowiska gry, taki jak pokój, piwnica czy schody lub obszar 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 całe środowisko gry jest podzielone na obszary, które są ze sobą połączone. Aby dostać się z jednego miejsca w danym środowisku gry 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. Zatem droga przez graf składa się z zera lub więcej połączeń.

3.1.2 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 wartość ta jest nazywana wagą, natomiast w grach jest ona określana mianem kosztu



Rysunek 2 Schemat grafu [1]

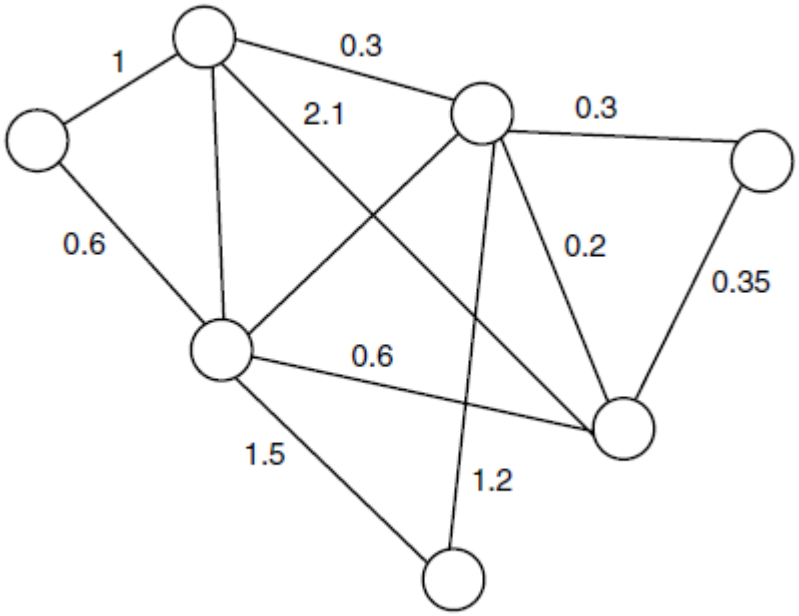
(chociaż graf nadal jest nazywany grafem ważonym, a nie grafem kosztu). Na rysunku 3 przedstawiono graf, w którym z każdym połączeniem jest związana wartość kosztu.

Koszty w module zarządzającym odnajdowaniem ścieżki są zwykle czasem lub odległością. Jeśli węzeł reprezentujący obszar środowiska gry jest położony w dużej odległości od innego takiego węzła, 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.

Na całej trasie przez graf, od węzła początkowego do węzła docelowego, możemy obliczyć całkowity koszt ścieżki. Jest to suma kosztów każdego połączenia na trasie.

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 ze względu

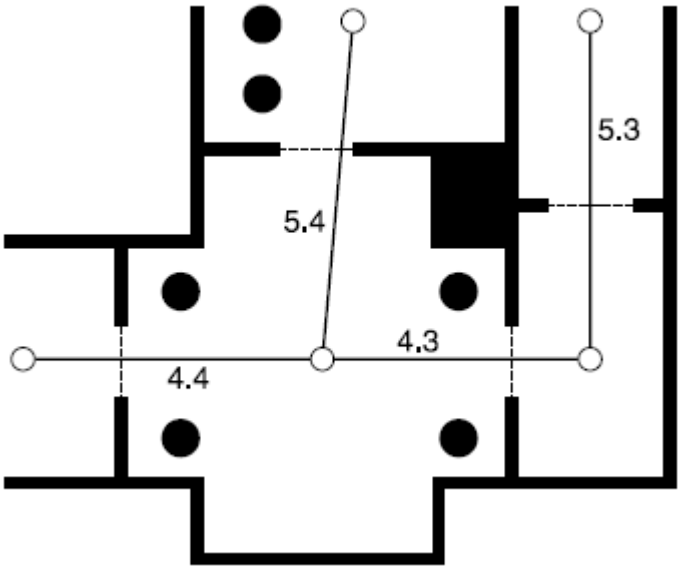


Rysunek 3 Schemat grafu ważonego [1]

na odległości czy czas przez punkt reprezentatywny znajdujący się 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, a co za tym idzie koszt również będzie duży. Rysunek 4 prezentuje wirtualny świat i odpowiadający mu graf, gdzie punkt reprezentatywny jest przyporządkowany do każdego regionu.

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 algorytmy Dijkstry i algorytmy A* powinny być stosowane wyłącznie z nieujemnymi wagami. Istnieją takie konstrukcje grafu posiadającego ujemne wagi, że algorytm zajmujący się procesem odnajdywania ścieżki zwróci rozsądny wynik. Jednak w większości przypadków, algorytm



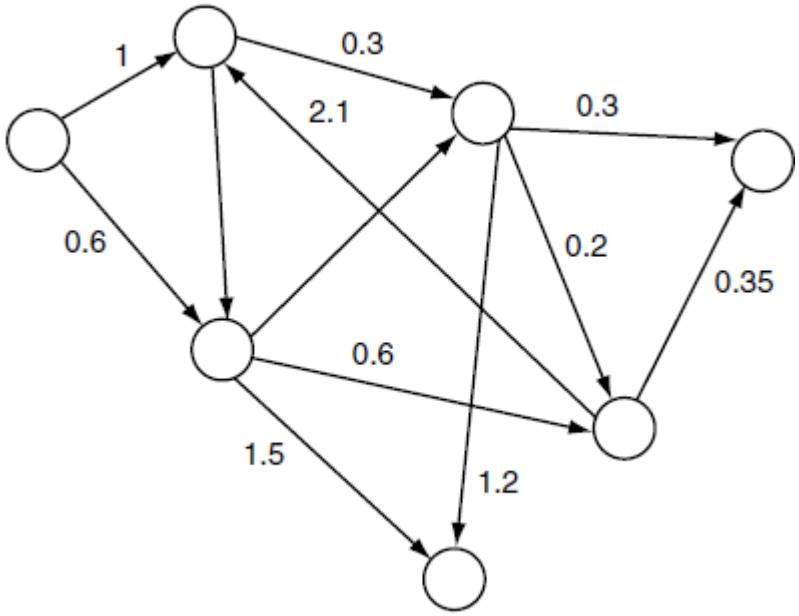
Rysunek 4 Węzły grafu określające regiony [1]

Dijkstry 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 z ujemnymi wagami - takie rozwiązanie nie istnieje. W niniejszej pracy koszt jest zawsze liczbą dodatnią. Twórcy gier wspólnie przyznają, że nigdy nie stosowali ujemnych wag ani algorytmów do nich przystosowanych w procesie tworzenia gier komputerowych [1].

3.1.3 Skierowane grafy ważone

W wielu sytuacjach ważony graf wystarczy do reprezentacji środowiska 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 (rysunek 5). Jest on często używany przez programistów gier komputerowych. Jeśli 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.



Rysunek 5 Skierowany graf ważony [1]

Po drugie dwa połączenia w różnych kierunkach oznaczają, 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 reprezentowane: połączenie A(piętro) z B(magazyn) ma mały koszt, a połączenie z B do A ma większy koszt. Matematycznie w grafie skierowanym pary węzłów, stanowiące połączenie są 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 innymi połączeniami.

3.2 Algorytm Dijkstry

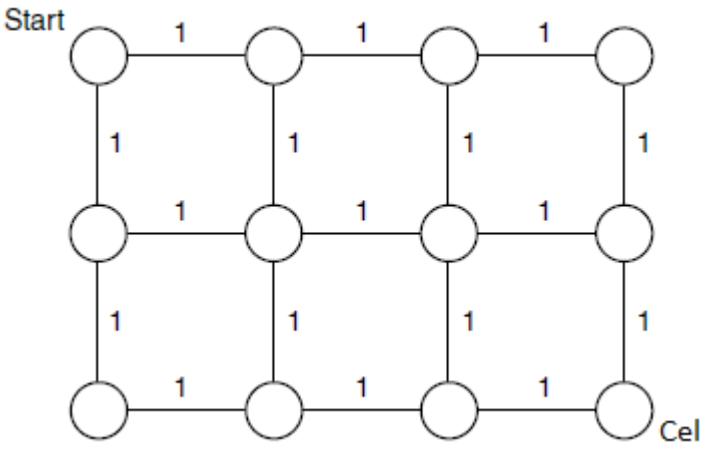
Algorytm Dijkstry został zaprojektowany przez holenderskiego matematyka Edsger'a Dijkstrte do odnajdywania najkrótszych ścieżek do wszystkich węzłów od punktu startowego w grafie. Rozwiązanie tego problemu zawiera w sobie rozwiązanie problemu

odnajdywania ścieżki w grach komputerowych z punktu początkowego do punkty końcowego, ale jest ono osiągane nieefektywnie.

Z tego powodu w praktyce nie korzysta się z algorytmu Dijkstry do odnajdywania ścieżki z punktu A do B w grach. Można go, jednak zastosować do analizy właściwości ogólnych danego środowiska gry w zaawansowanym systemie odnajdywania ścieżki[źródło].

3.2.1 Przedstawienie problemu

Dany jest graf (skierowany o nieujemnych wagach) i dwa węzły (początkowy i końcowy) w tym grafie. Zadaniem algorytmu Dijkstry jest wygenerowanie ścieżki tak, aby całkowity koszt ścieżki był minimalny spośród wszystkich dostępnych ścieżek od startu do celu. Może dojść do sytuacji że będzie istnieć wiele ścieżek o takim samym minimalnym koszcie.



Rysunek 6 Przykład grafu, gdzie każda ścieżka jest optymalna [1]

Rysunek 6 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 zejścia po drabinie do magazynu). Dlatego trzeba dokładnie wiedzieć które połączenia są potrzebne, a lista węzłów byłaby niekompletna. Wiele gier nie robi tego rozróżnienia, zakładając że istnieje najwyżej jedno połączenie pomiędzy dowolną parą węzłów. Pomimo to, jeśli są dwa

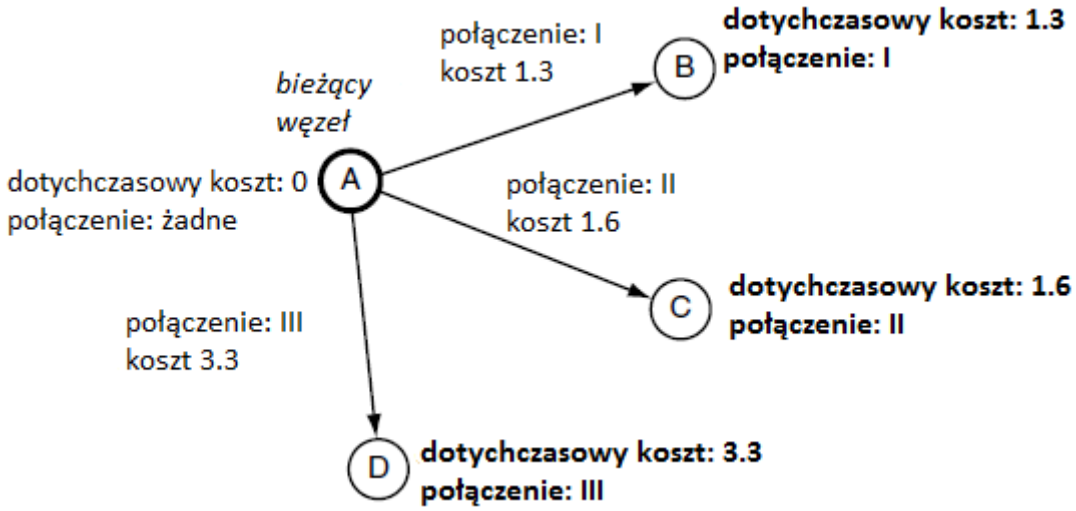
połączenia pomiędzy parą węzłów, to moduł odpowiedzialny za odnajdywanie ścieżki powinien zawsze wybrać to połączenie z najniższym kosztem. W niektórych zastosowaniach, koszt zmienia się w trakcie gry lub pomiędzy różnymi postaciami. Na potrzeby tej pracy ścieżkę identyfikujemy jako listę połączeń.

3.2.2 Opis algorytmu

Algorytm Dijkstry podczas każdej iteracji, rozważa każde z połączeń wychodzących z bieżącego węzła. Dla każdego połączenia znajduje węzeł końcowy i zapisuje całkowity koszt ścieżki znaleziony do tej pory wraz z połączeniami.

W pierwszej iteracji, gdzie węzeł startowy jest bieżącym węzłem, kosztem dotychczasowym dla każdego połączenia w węźle końcowym jest koszt bieżącego połączenia. Rysunek 7 pokazuje przykładową analizę dokonaną w pierwszej iteracji.

Dla kolejnych iteracji dotychczasowy koszt węzła końcowego każdego analizowanego połączenia bieżącego węzła jest sumą kosztów połączeń i kosztów dotychczasowych bieżącego węzła. Rysunek 8 przedstawia kolejną iterację grafu z rysunku 7. Węzłem bieżącym jest węzeł B, którego koszt dotychczasowy był najmniejszy. Koszt dotychczasowy w węźle E jest sumą kosztu dotychczasowego z węzła B i kosztem połączenia IV z B do E.

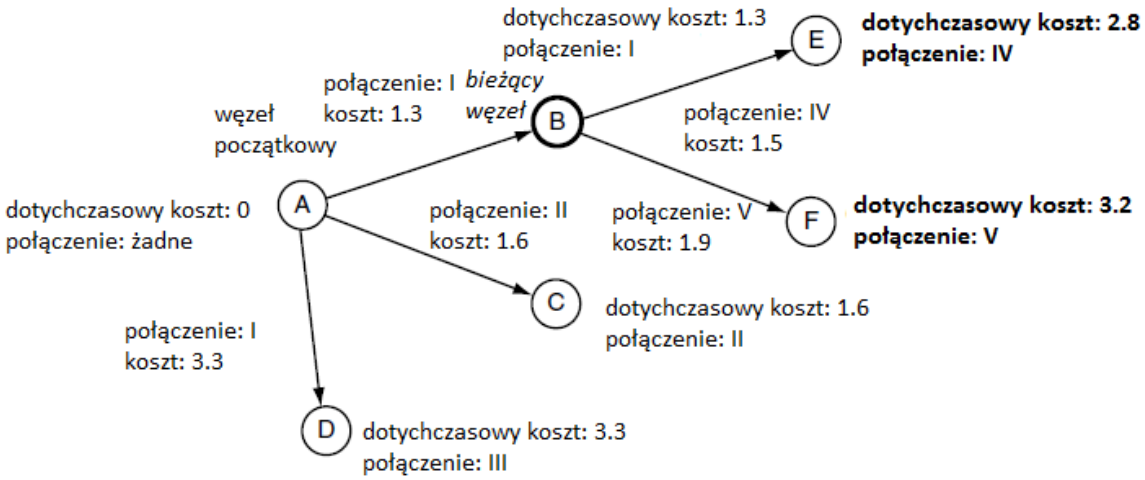


Rysunek 7 Dijkstra - pierwsza iteracja [1]

Lista węzłów

Algorytm korzysta z dwóch list zwanych: listą otwartą i listą zamkniętą. W otwartej liście rejestruje się wszystkie znalezione przez algorytm węzły, które nie miały jeszcze swojej własnej iteracji. Węzły przetworzone są zapisywane w liście zamkniętej.

Na początku lista otwarta zawiera tylko jeden węzeł początkowy (z zerowym



Rysunek 8 Dijkstra - kolejna iteracja [1]

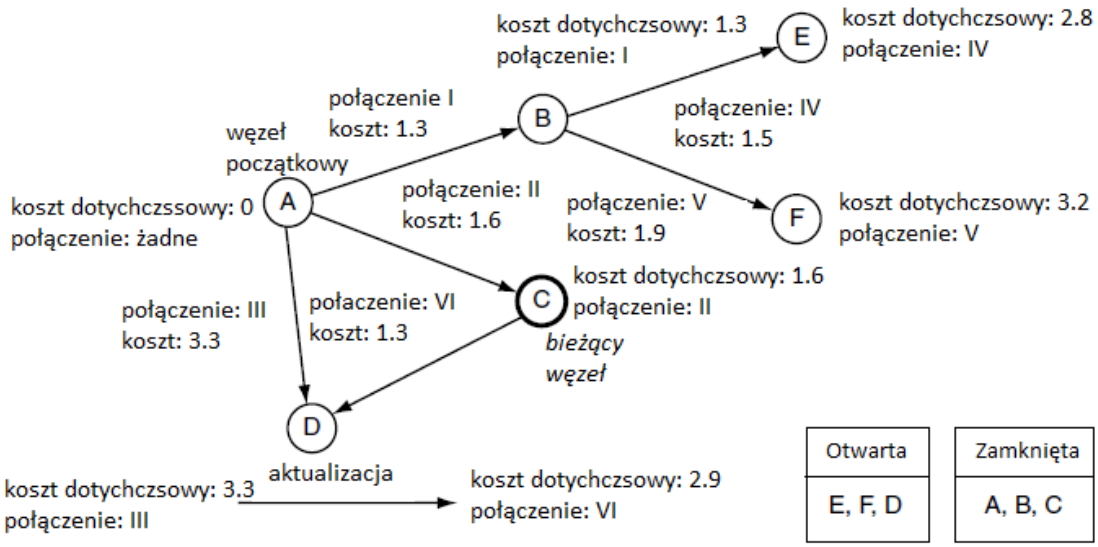
dotychczasowym kosztem), a lista zamknięta jest pusta. Każdy węzeł może należeć do jednej z trzech kategorii: może on należeć do zamkniętej listy, może być też na liście otwartej - będąc odwiedzonym przez inny węzeł, ale jeszcze nie przetworzony w swojej iteracji, może również nie znajdować się na żadnej 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 on przetworzony. Przetworzony węzeł jest usuwany z otwartej listy i dodawany 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, albo zamiast tego, można sprawdzić czy trasa, która obecnie jest rozważana jest lepsza od poprzednio znalezionej. Jeśli wartość kosztu jest większa niż ta aktualnie zarejestrowana, to nie aktualizujemy węzła. Jeśli nowy koszt jest mniejszy, to wtedy dokonywana jest jego aktualizacja.

Rysunek 9 przedstawia aktualizacje otwartego węzła w grafie. Nowa ścieżka przez

węzeł C jest szybsza i zapis dla węzła D jest odpowiednio zaktualizowany.



Rysunek 9 Aktualizacja otwartego węzła[1]

Przerywanie algorytmu

Algorytmu przerywa działanie, kiedy otwarta lista jest pusta. 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 można przerwać algorytm wcześniej. Dokonuje się przerwania w momencie, kiedy węzeł docelowy jest najmniejszym węzłem na liście otwartej.

Na rysunku 9 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.

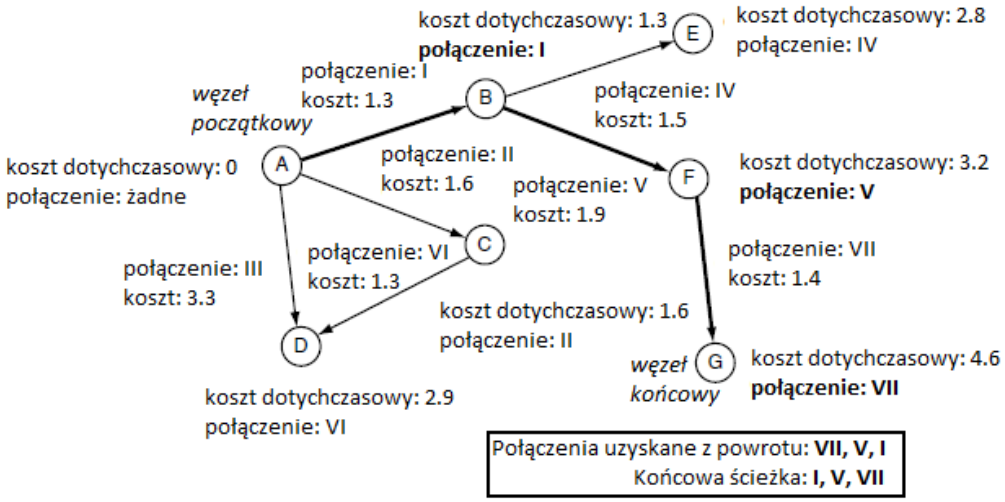
W praktyce ta zasada jest często łamana. Algorytm jest modyfikowany tak, że jeśli zostaje znaleziona ścieżka do celu to zakłada się, że ona jest najkrótsza, nawet jeśli istnieje inna, krótsza - zwykle jest ona krótsza o niewielką wartość liczbową.

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.

Rysunek 10 przedstawia prosty graf zaraz po tym jak algorytm zakończył działanie. Znaleziona lista połączeń zawiera węzły od celu do początku - jej odwrócenie pozwoli uzyskać poszukiwaną drogę.



Rysunek 10 Graf po zakończeniu obliczeń [1]

3.3 Algorytm A*

Odnajdywanie ścieżki w grach odbywa się najczęściej za pomocą algorytmu A*. A* jest prosty w implementacji, bardzo efektywny oraz posiada bardzo duże możliwości optymalizacji. Może również zostać wykorzystany do planowania złożonych akcji dla postaci.

W odróżnieniu od algorytmu Dijkstry, A* został zaprojektowany dla odnajdywania ścieżki z punktu do punktu i nie jest stosowany do rozwiązywania problemu najkrótszej ścieżki w teorii grafów.

3.3.1 Przedstawienie problemu

Problem rozwiązywany przez algorytm A* 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 jej koszt jest minimalny spośród wszystkich dostępnych ścieżek od startu do celu.

3.3.2 Opis algorytmu

Algorytm A* działa na takiej samej zasadzie jak algorytm Dijkstry. Występuje tam otwarta lista z najniższymi dotychczasowymi kosztami oraz lista zamknięta. Algorytm A* działa iteracyjnie. W każdej iteracji jest przetwarzany jeden z węzłów grafu nazywany bieżącym.

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 powszechnie nazywana "wartością heurystyki" węzła i nie może być to wartość ujemna. Proces generowania wartości heurystyki jest punktem kluczowym w procesie implementacji algorytmu A* - zostanie on omówiony później.

Rysunek 11 prezentuje obliczone wartości dla paru węzłów w grafie. Węzły są opisane, również wartościami heurystyki. Przestawione są dwie wartości (dotychczasowy koszt i estymowany koszt całkowity), które informują, że dane węzły zostały przetworzone przez algorytm.

Możemy wyróżnić trzy przykładowe funkcje wyznaczające wartość heurystyki:

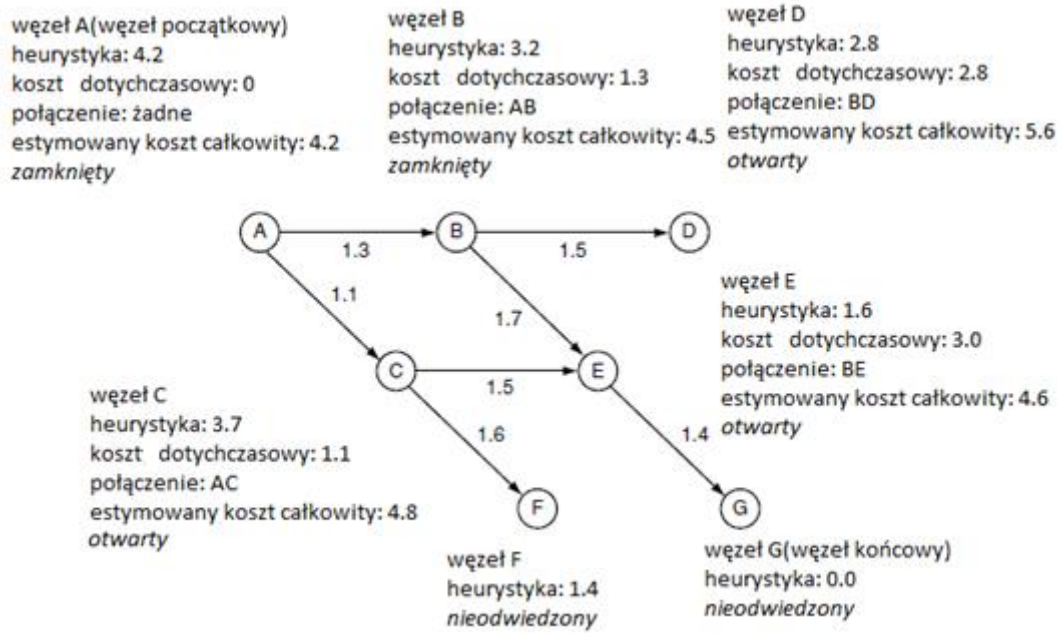
- Manhattan. Wyznacza ona wartość na podstawie obliczenia całkowitej ilości pól koniecznych do przejścia od aktualnej pozycji do celu, dla ruchu poziomego i pionowego. W obliczeniach nie jest brany pod uwagę ruch po skosie oraz ignorowane są wszystkie przeszkody jakie mogą wystąpić na trasie. Wyraża się wzorem 1.
- Diagonalny Manhattan. Funkcja obliczająca estymacje analogicznie do heurystyki Manhattan, bierze pod uwagę ruch po skosie.
- Euklidesowa. Jest to miara euklidesowa na płaszczyźnie. Wyraża się wzorem 2.

$$manhattan(p1, p2) = |p1.x - p2.x| + |p1.y - p2.y| \qquad (1)$$

$$euclidean(p1, p2) = \sqrt[2]{(p1.x - p2.x)^2 + (p1.y - p2.y)^2} \qquad (2)$$

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



Rysunek 11 A* - działanie [1]

są tam węzły, które zostały przetworzone w swojej iteracji. Węzeł z najmniejszym estymowanym kosztem całkowitym jest wybierany do każdej iteracji.

Powoduje to, 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 oblicza 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 wartości kosztu dotychczasowego, a nie estymowanego kosztu całkowitego.

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

To powoduje pewien problem występujący również w algorytmie Dijkstry. 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.

Rysunek 12 przestawia aktualizacje grafu - jest to analogiczna sytuacja do grafu z rysunku 11, 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.

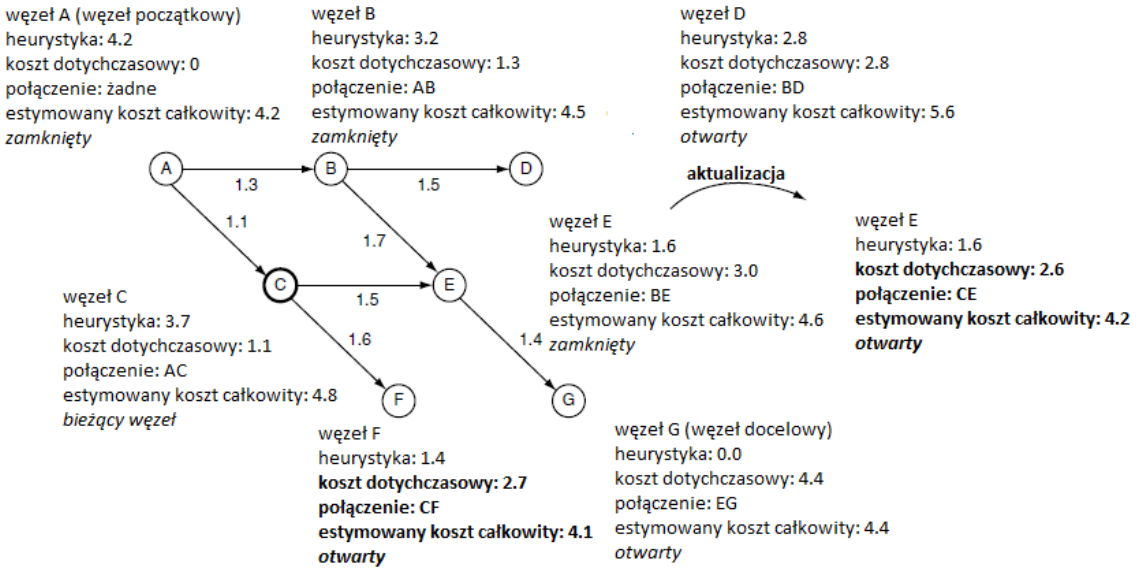
Często jednak zdarza się, że węzeł posiadający najmniejszy estymowany koszt całkowity może zostać zaktualizowany. Nie da się zagwarantować, że węzeł, który jest pierwszym na otwartej liście jest tym, który posiada najkrótszą ścieżkę do tego miejsca. Dlatego przerwanie algorytmu w tym momencie nie daje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

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, jednak nie jest tak duży jak w przypadku zrobienia identycznej rzeczy w algorytmie Dijkstry.

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.



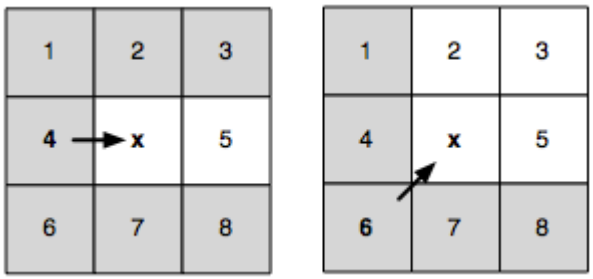
Rysunek 12 Aktualizacja zamkniętego węzła [1]

3.4 Algorytm JPS

Podczas każdej iteracji algorytm A* rozszerza obszar poszukiwań w najlepszym znanym kierunku. Jednak istnieją sytuacje, które mogą prowadzić do spadku jego efektywności. Spadek ten może pojawić się w przypadku przetwarzania dużych przestrzeni. Algorytm JPS(ang. jump point search) pozwala na przyśpieszenie obliczeń szukanej ścieżki.

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

W przypadku algorytmu A* dokonywana jest operacja rozszerzania ścieżki 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 za mało wartościowe. Można tego dokonać określając sytuacje, w której są obecne symetryczne ścieżki i pomijać niektóre węzły. Dwie ścieżki są symetryczne, jeżeli mają ten sam punkt początkowy i końcowy oraz zmiana kolejności składowych wektora punktów ścieżki pozwala przekształcić jedną ścieżkę w drugą. Rysunek 13 przedstawia podstawową idee algorytmu JPS.



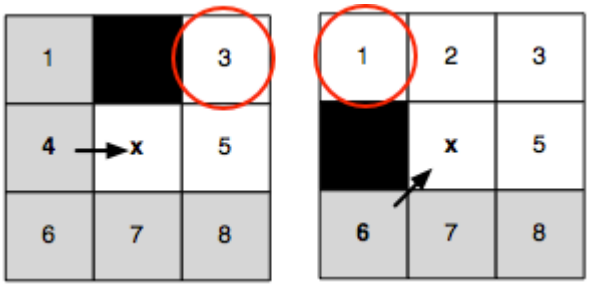
Rysunek 13 Idea algorytmu JPS [5]

Węzeł *x* jest węzłem aktualnie przetwarzanym. 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 bezpośrednio 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 kolorem białym na rysunku 15. 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 ($0 \leq k \leq 2$). Zbiór jest zwykle nazywany zbiorem wymuszonych sąsiadów bieżącego węzła. Rysunek 16 przedstawia taki przypadek.

Węzeł *x* na rysunku 14 jest węzłem aktualnie przetwarzanym, 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 droga z rodzica węzła *x* do każdego z tych węzłów jest zablokowana.



Rysunek 14 Przykład wymuszonych sąsiadów [5]

Wyżej opisane zasady obcinania węzłów stosuje się w następujący sposób: zamiast generować naturalnych i wymuszonych sąsiadów, wykonuje się rekursywne obcinanie zbioru sąsiadów wokół każdego 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 przez bieżący węzeł [2].

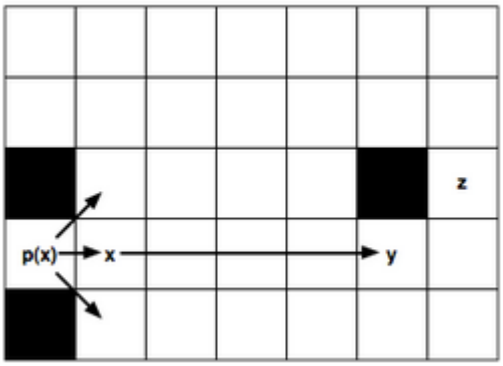
Rekursywny algorytm obcinania sąsiadów musi zapewniać optymalność. Jest ona zagwarantowana przez uporządkowany proces , w jakim przetwarzani są naturalni sąsiedzi (najpierw horyzontalnie/wertykalnie, a następnie diagonalnie) [5]. Rysunek 17 i 18 przedstawia dwa przykłady algorytmu obcinania w akcji. W pierwszym przypadku możemy określić punkt skoku przez rekursje horyzontalną, a w drugim punkt jest określany przez rekursje diagonalną.

Rysunki 15 i 16 przedstawiają przykładowe skoki. Węzeł x jest aktualnie przetwarzanym węzłem *p(x)* jest jego rodzicem.

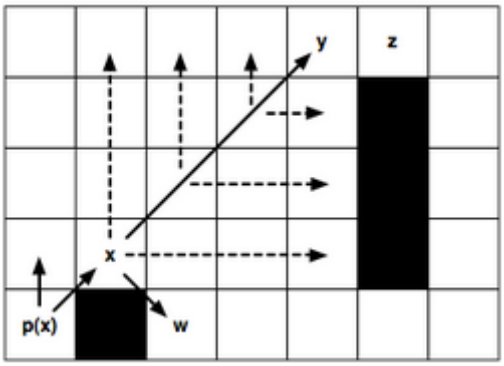
Na rysunku 15 została przedstawiona prosta reguła przycinania i identyfikacji węzła y jako następcy skoku *x*. Węzeł ten jest rozważany, ponieważ ma sąsiada *z*, do którego optymalna ścieżka zgodnie z założeniami musi prowadzić przez punkty *x* i potem *y*.

Zastosowanie rekursji z wariantem przekątnej reguły obcinania (rysunek 16), 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*.



Rysunek 15 Skok diagonalny [5]



Rysunek 16 Horyzontalny skok [5]

4 System nawigacji

4.1 Analiza zagadnienia

Skład problemów, z którymi przychodzi się zmierzyć podczas prac nad grą komputerową, zależy od pewnych ustaleń i wymagań jakie gra ma spełniać. W przypadku dodania do gry systemu nawigacyjnego warto dokonać analizy pewnych zagadnień.

Pierwszym z nich jest charakterystyka środowiska gry. Musi ono mieć odpowiednią reprezentacje, aby mogło zostać odpowiednio przetworzone. Środowiska gry mogą różnić się rozmiarem oraz stopniem skomplikowania. Środowiska o niewielkim stopniu skomplikowania, posiadają małą liczbę możliwych do przejścia ścieżek. Natomiast te o dużym stopniu skomplikowania mają strukturę podobną do labiryntu.

Kolejnym zagadnieniem jest dobór odpowiedniego algorytmu. W grach komputerowych popularnym algorytmem jest algorytm A*. I jest on stosowany w większości gier komputerowych. Jest to algorytm heurystyczny, z którym związane są pewne funkcje heurystyczne opisane w rozdziale 3.3.2. Funkcje te mogą wpływać na czas w jakim zostanie odnaleziona ścieżka, zależnie od rozmiaru środowiska gry oraz stopnia jego skomplikowania. Zatem warto zastanowić się, w którym przypadku zastosować daną funkcję heurystyczną, aby droga została obliczona w jak najkrótszym czasie.

Obecnie producenci gier dostarczają środowiska gier o bardzo dużych rozmiarach, po których porusza się duża liczba agentów komputerowych. Przetwarzając tak dużą liczbę danych warto zastanowić się nad kwestiami optymalizacyjnymi. Jednym z rozwiązań powalających przyśpieszyć proces odnajdowania ścieżki jest implementacja algorytmu Jump Point Search.

4.2 Założenia

Projekt dyplomowy jest trójwymiarowym środowiskiem służącym do badania wydajności algorytmu A*. W skład tego środowiska wchodzą: trójwymiarowy model terenu oraz model agenta komputerowego. Na potrzeby badań zostaną wykonanie cztery modele terenu różniące się stopniem skomplikowania. W każdym ze środowisk gry zostaną umieszczeni agenci, których celem jest dotarcie do wskazanego przez użytkownika celu.

Do procesu odnajdywania ścieżki zostanie użyta gotowa biblioteka *A* pathfinding*

project. Dodatkowo zostanie zaimplementowany oddzielny system nawigacyjny zawierający algorytm Jump Point Search.

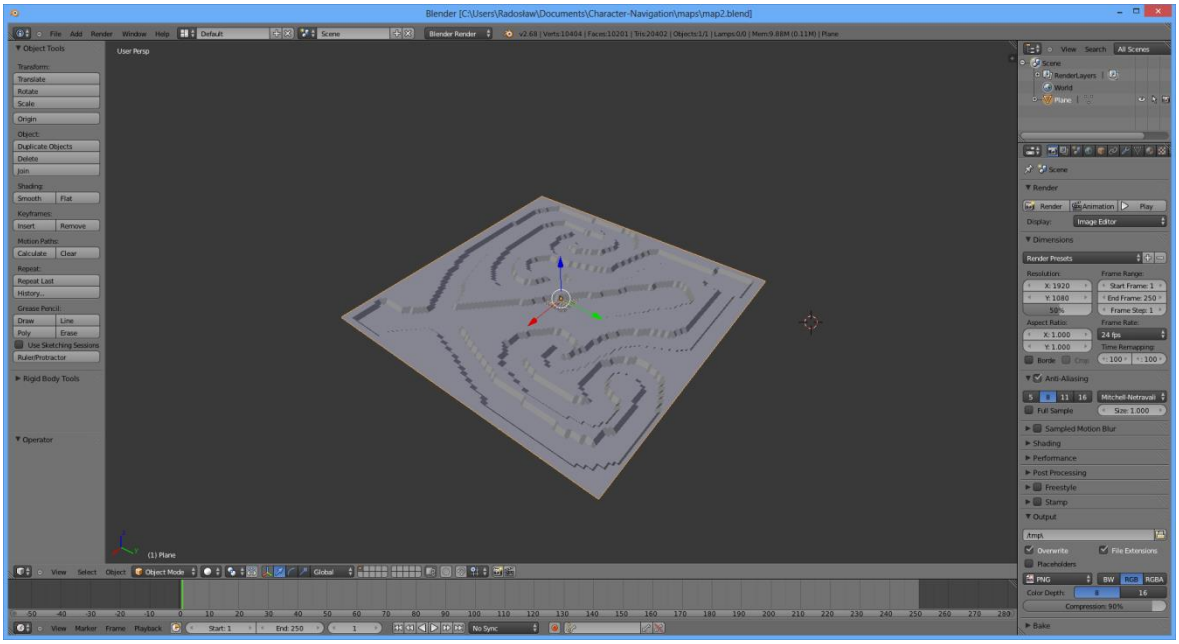
Proces uruchomienia testu będzie wymagał odpowiedniej konfiguracji biblioteki, umieszczenia agentów w środowisku gry oraz wskazania punktu docelowego.

Projekt dyplomowy zostanie wykonany w silniku Unity3D. Projekt ten będzie umożliwiał przeprowadzenie symulacji odnajdywania ścieżki na czterech zaprojektowanych środowiskach gry. Do implementacji zostanie zastosowany język C#. Trójwymiarowy model środowiska zostanie wykonany w programie Blender.

4.3 Narzędzia

4.3.1 Blender

Blender jest darmowym środowiskiem do tworzenia grafiki trójwymiarowej. Oprócz standardowej funkcjonalności (tworzenie statycznych modeli 3D, animacji) posiada też wbudowany silnik gier. Pozwala także modelować środowiska świata gry (rysunek 13).

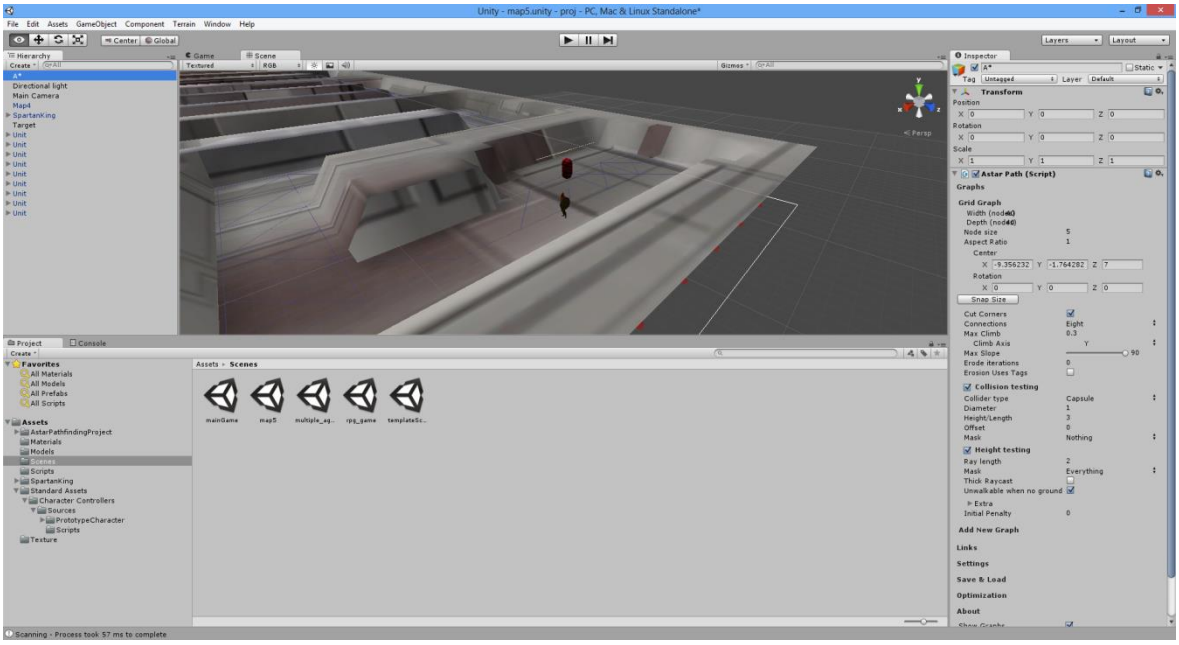


Rysunek 17 Proces tworzenia modelu środowiska gry w programie Blender

4.3.2 Unity3D

Unity3D (rysunek 18) 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 uruchomić w prosty sposób na większości 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.



Rysunek 18 Zrzut ekranu prezentujący środowisko silnika Unity3D

Cechy silnika Unity3D:

Wygodny w użyciu edytor zawierający:

- Podgląd logów wyświetlanych przez grę, a także ostrzeżeń i błędów. Logi wysyłamy do konsoli z poziomu skryptów.
- Inspektor obiektów przy pomocy którego możemy ustalać różne właściwości obiektu takie jak: pozycja, rotacja, skala, parametry skryptów, parametry związane z wyświetlaniem czy fizyką.
- Podgląd widoku hierarchii obiektów na scenie.
- Wspólny widok wszystkich elementów znajdujących się w projekcie: modele, materiały, skrypty, sceny.
- Szybki podgląd jak po kompilacji i uruchomieniu będzie wyglądać gra.
- Widok sceny, po którym projektant może dowolnie się poruszać ustawiać obiekty, światło czy kamerę.

Wyraźny podział na obiekty, komponenty oraz skrypty:

- Obiekt 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.

- Komponent wchodzi w skład obiektów, a konkretnie rozszerza go o nowe funkcjonalności. Filozofia działania Unity3D związana jest właśnie z pracą na komponentach. Komponent przeważnie posiada pewną liczbę parametrów. Przykładowo renderowany na ekranie sześcian (obiekt) składa z następujących elementów: GameObject zawierający komponent MeshFiler (przechowuje dane o geometrii, jako parametr przyjmuje siatkę sześcianu). Następnie obiekt sześcianu zawiera kolejny komponent MeshRenderer, który posiada parametr ustawiający materiał renderowania. Co więcej, do sześcianu można przypisać komponent Rigidbody - w nim ustawiane jest parametry fizyczne dla obiektu. Nadawana jest mu masa oraz gęstość. Dzięki temu istnieje możliwość oddziaływania na obiekt siłą. Komponentem może też być skrypt.
- Komponent skryptowy - 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 opis 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 zebrała się duża społeczność programistów, którzy aktywnie

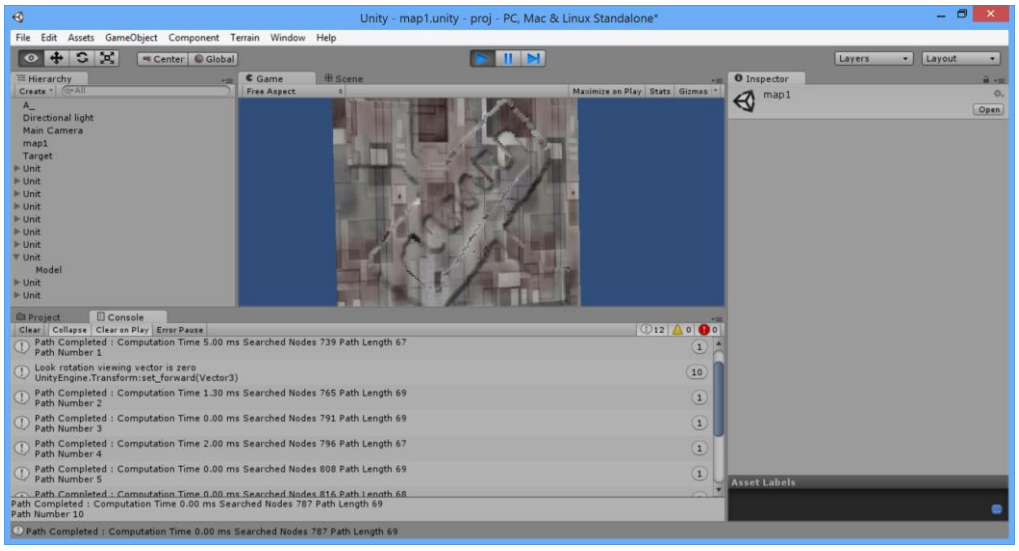
działają na forach dyskusyjnych oraz służą pomocą w przypadku napotkania problemów.

Elastyczność:

- Rozszerzalność - programista może sam pisać własne skrypty i wtyczki rozszerzające funkcjonalność silnika.
- Elementy można przenosić między projektami poprzez eksportowanie/importowanie zasobów gry w postaci jednego pliku.
- Używając prefabrykowanych elementów w trakcie zmiany parametrów jednego z nich, zmiany te są propagowane do pozostałych elementów tego samego typu. Pozwala to przyśpieszyć proces składania świata gry w całość.
- Silnik Unity3D posiada wbudowaną bazę jednostek cieniujących (ang. shader), przetwarzających obiekty na poziomie grafiki i pikseli z wykorzystaniem procesora graficznego GPU. Oczywiście programista może rozbudować tą bazę pisząc swoje własne jednostki cieniujące.

4.4 Specyfikacja zewnętrzna

Po uruchomieniu projektu należy wybrać scenę gry, na której mają odbywać się eksperymenty. W ramach sceny zapisane są informacje o środowisku gry. Sceny są umieszczone w katalogu Assets/Sceen. Każda ze scen posiada skonfigurowaną bibliotekę odpowiedzialną za odnajdywanie ścieżki. Proces konfiguracji A* został opisany w rozdziale 4.4.1, a JPS w rozdziale 4.4.2. Kilka elementów konfiguracyjnych biblioteki ulega zmianie, ponieważ są to parametry eksperymentu.



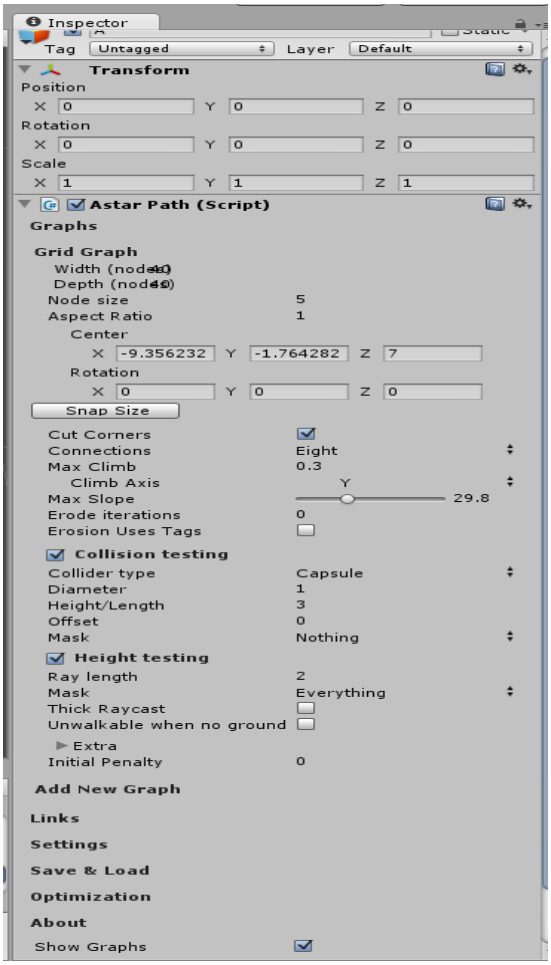
Rysunek 19 Zrzut ekranu z środowiska testowego

Ponadto na scenie można zmienić punkt docelowy agentów komputerowych ustawiając odpowiednio współrzędne obiektu *Target* oraz punkt startowy agentów.

Po ustawieniu parametrów należy włączyć eksperyment przyciskiem *Run* (trójkątny przycisk umieszczony w środku górnej części interfejsu użytkownika). Włącznie eksperymentu powoduje automatyczne przełączenie widoku na środowisko gry, na którym można obserwować poruszających się agentów komputerowych. Agenci przemieszczają się do wcześniej wyznaczonego celu. W dolnej części interfejsu użytkownika znajduje się konsola. Wyświetla ona informacje (czas obliczeń, liczba przeszukanych węzłów oraz długości ścieżki) o każdym agencie biorącym udział w eksperymencie (rysunek 19).

4.4.1 Konfiguracja biblioteki A* Pathfinding Project

Po uruchomieniu silnika Unity3D i zainstalowaniu dostarczonej biblioteki *A* Pathfinding Project* pierwszym krokiem konfiguracyjnym jest dodanie do sceny trójwymiarowego modelu środowiska gry, na którym ma działać moduł odnajdowania ścieżki. Kolejno dodajemy pusty obiekt, który będzie zawierał całą konfigurację biblioteki



Rysunek 20 Parametry konfiguracyjne biblioteki A* Pathfinding Project

(w projekcie nazwany jest on *A**). Do obiektu *A** dodajemy skrypt z biblioteki o nazwie *Astar Path*, w którym dokonywana jest konfiguracja ustawień. Odpowiednia konfiguracja jest bardzo istotna z punktu widzenia optymalizacji oraz napotkanych błędów w czasie testowania aplikacji. Rysunek 20 przedstawia parametry konfiguracyjne widziane z poziomu silnika Unity3D.

Najbardziej istotne parametry konfiguracyjne algorytmu A* to:

1. Rodzaj grafu – np. siatka (*Grid Graph*).
2. Rozmiar węzła grafu. Środowisko gry ma rozmiar 200x200, więc rozmiar węzła może przyjąć 4x4, a długość i szerokość grafu będzie wynosić 50x50. Wartości te będą się zmieniały w przypadku badań wydajności gęstości grafu na 100x100 i 200x200.
3. Po ustaleniu rozmiaru siatki grafu należy ją umieścić tak, aby całą swoją długością i szerokością (współrzędne x i z) obejmowała całe środowisko gry.
4. Parametr *MaxClimb* - opisuje maksymalną pozycje pomiędzy dwoma węzłami, tak aby istniało połączenie.
5. *MaxSlope* - określa maksymalny kąt nachylenia dla węzła, aby można po nim przejść.
6. Biblioteka automatycznie buduje graf przez dokonywanie testów wysokości środowiska gry. Robi to za pomocą rzutów promieni (ang. raycast). Związany jest z tym parametr opisujący długość promienia (*ray length*). Parametr ten pozwala ustalić maksymalną wysokość do jakiej ma być tworzony graf. Dla obszarów położonych powyżej tej wartości graf nie będzie budowany. Spowoduje to, że agenci nie będą mogli się poruszać po takich obszarach.

4.4.2 Konfiguracja modułu JPS

Po uruchomieniu silnika Unity3D z projektem zawierającym moduł JPS należy dokonać konfiguracji środowiska. Konfiguracja odbywa się przez ustawienie odpowiednich parametrów skryptu *JumpPointNavigationMesh*.

Najbardziej istotne parametry to:

1. Punkt początkowy i docelowy ścieżki (*Start Transform* i *Destination Transform*).
2. Parametr *Resolution* określający rozmiar węzła.
3. Maksymalną wysokość między węzłami, tak aby istniało połącznie reprezentuje parametr *Max Height Delta.*

4.5 Specyfikacja wewnętrzna

4.5.1 Struktura skryptów w Unity3D

Skrypty w Unity3D są to klasy dziedziczące po wybudowanej w silnik klasie `MonoBehaviour`. Skrypty te mają dostęp do metod, które można dowolnie nadpisać. Najczęściej nadpisywanymi metodami są:

`Start` – metoda wywoływana przed innymi metodami tylko raz. Pozwala ona na inicjalizacje pól klasy.

`FixedUpdate/Update` – metoda wywoływana jest w każdej klatce. Pozwala aktualizować położenie obiektów.

`OnGUI` – metoda pozwalająca na przechwytywanie zdarzeń związanych z graficznym interfejsem użytkownika (np. przyciśnięcie przycisku).

4.5.2 Skrypt nawigacyjny

Wykorzystanie biblioteki A* Pathfinding Project wymaga implementacji skryptu odpowiedzialnego za przemieszczanie się obiektów po środowisku gry na podstawie ścieżki. Skrypt został dodany (jako komponent) do każdego agenta występującego w grze.

Skrypt posiada pole `target` określające współrzędne celu, do którego mają dostać się agenci. Jest to pole publiczne - takie pola w Unity3D mają możliwość wstawienia obiektu z poziomu silnika.

`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` aktualizuje stan gry.

4.5.3 Skrypty modyfikujące

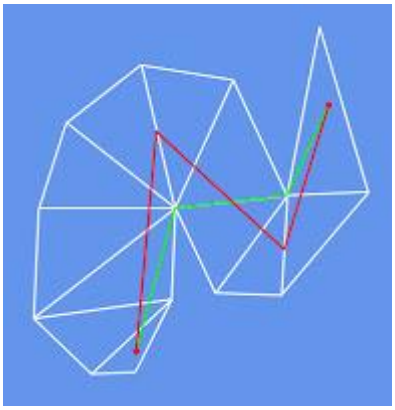
Jakość otrzymanego rozwiązania w postaci ścieżki można rozpatrywać jako czas obliczenia najkrótszej ścieżki. Jednak interesujący jest aspekt jakości w rozumieniu gracza.

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 rodzaje skryptów modyfikujących). Na potrzeby projektu zostały dołączone dwa skrypty modyfikujące:

- *funnel* - jest to prosty algorytm, który znajduje proste odcinki trasy. Rysunek 21 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.



Rysunek 21 Zasada działania skryptu *funnel*

4.5.4 Implementacja algorytmu Jump Point Search

Implementacja optymalizacji algorytmu A* wymagała zaprogramowania całego modułu odnajdowania ścieżki. Poniżej zostaną przedstawione klasy biorące udział

w procesie reprezentacji danych środowiska gry oraz główny skrypt, w którym zawarta jest optymalizacja algorytmu.

Do reprezentacji siatki środowiska gry wymagana była implementacja prostej klasy punktu (Point) opisanej za pomocą pól x i y.

Do reprezentacji węzła w grafie stosowana jest klasa `Node`, zawierająca wartość funkcji kosztu, wartość estymowanej 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 dany węzeł jest węzłem docelowym. Stan ten jest reprezentowane przez wartość

logiczną w polu `mIsDestinationNode`.

Implementacja głównego skryptu znajduje się w klasie *`JumpPointNavigationMesh`*. Pierwsze z jej publicznych pól (`Gizmo`) jest obiektem, który służy do rysowania ścieżki w środowisku gry. 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 środowiska gry.

Wektor `mTerrainPosition` zawiera w sobie współrzędne terenu. Pole to będzie wykorzystane w procesie skalowania siatki i terenu. `mGizmoHeight` jest wektorem opisującym wysokość na jakiej 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 zawierających 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 środowiska gry 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 trójwymiarowe środowisko gry w dwuwymiarowej tablicy, gdzie trzeci wymiar jest reprezentowany przez wartość koloru piksela. Taką tablicę można zapisać w formacie pliku graficznego. 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 świecie – jest to zbiór obiektów reprezentujących znalezioną drogę do celu. Druga z list jest tworzona osobno dla każdego węzła i zawiera listę jego sąsiadów.

Działanie skryptu `JumpPointNavigationMesh` 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 obiektu 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 celu.

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 (`openList`) i zamkniętą (`closedList`).

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.

Następnie sprawdzany jest 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).

Metoda `Jump` zajmuje się rekursywnym rozszerzaniem ścieżki wertykalnie/horyzontalnie i 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.

5 Eksperymenty

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

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

Testy wykonano za pomocą środowiska przedstawionego w rozdziale 4. Do rozwiązania problemu znajdowania ścieżki zastosowany został algorytm A*. Parametrami eksperymentów związanych z badaniem heurystyk są:

- Poziom skomplikowania środowiska gry (od prostej ścieżki po labirynt),
- Użyta funkcja heurystyczna: odległość euklidesową, Manhattan, diagonalny Manhattan (rozdział 3.3.2),
- Wykorzystane skrypty modyfikujące ścieżkę (opisane w rozdziale 5.2),
- Gęstość grafu (obszar poszukiwań zostaje zwiększony x4 i x16).

W przypadku eksperymentu związanego z algorytmem Jump Point Search parametrami są:

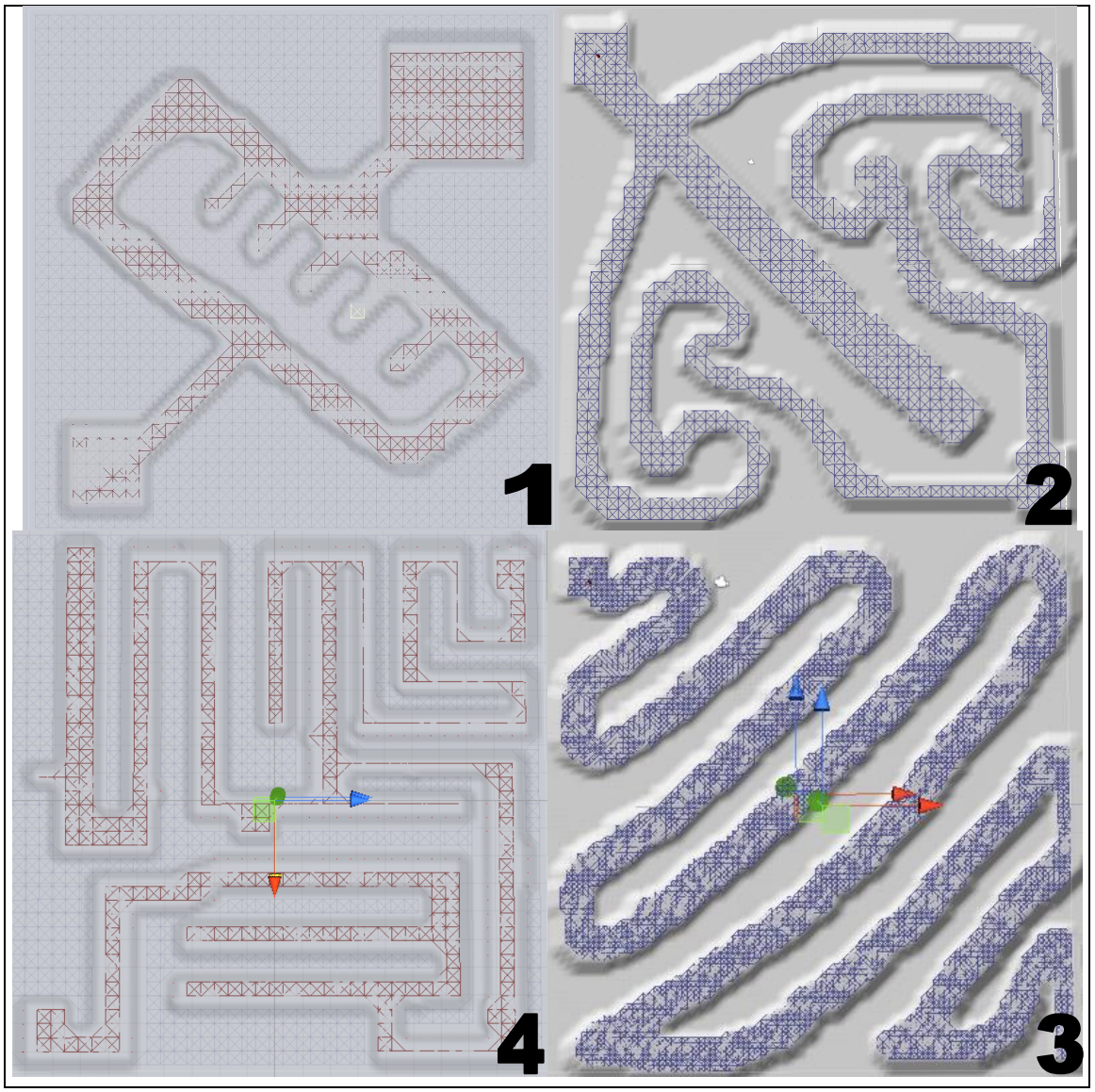
- Poziom skomplikowania środowiska gry (od prostej ścieżki po labirynt),
- Gęstość grafu (obszar poszukiwań zostaje zwiększony x4 i x16).

5.1 Charakterystyka zaprojektowanych środowisk gry

Na rysunku 22 znajdują się cztery modele środowisk wraz z siatką grafu. Obszary niepokryte siatką są niedostępne dla agentów. Modele środowisk gry zostały wykonane przez autora pracy w programie Blender.

Środowiska gry znajdujące się na rysunku 22 można podzielić ze względu na stopnień skomplikowania. Środowiska numer 1 i 2 posiadają prostą strukturę oraz niewielką liczbę alternatywnych ścieżek do zbadania.

Środowiska gry o numerach 3 i 4 posiadają bardziej skomplikowaną strukturę terenu. Środowisko numer 3 posiada jedną ścieżkę, ale jest ona najdłuższa z spośród pozostałych środowisk. Natomiast środowisko gry numer 4 posiada strukturę labiryntu, przez co istnieje wiele alternatywnych ścieżek do rozpatrzenia przez algorytm odnajdujący ścieżkę.



Rysunek 22 Projekty środowisk gry

5.2 Charakterystyka sprzętu zastosowanego do eksperymentów.

Eksperymenty zostały przeprowadzone na komputerze o parametrach zawartych w tabeli 1.

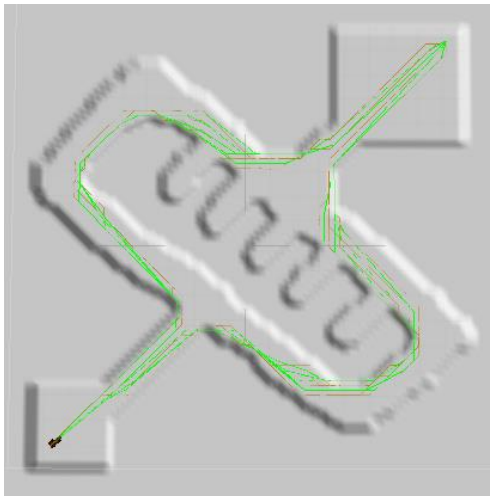
Tabela 1 Charakterystyka urządzenia

Procesor	Intel Core i7-3630QM (4 rdzenie, od 2.40 GHz do 3.40 GHz, 6 MB cache)
Karta grafiki	2 x NVIDIA GeForce GT 650M SLI
Dysk twardy	Samsung SSD 840 Series 120 GB
Pamięć	16 GB (SO-DIMM DDR3, 1600 MHz)
System operacyjny	Microsoft Windows 8 x64

5.3 Eksperyment 1 – badanie heurystyk

Dla każdego z zaprojektowanych środowisk gry zostały przeprowadzone badania. Polegały one na umieszczeniu dziesięciu komputerowych agentów w wybranym punkcie startowym, następnie został im wskazany punkt docelowy, do którego mieli się udać.

Podczas eksperymentów mierzony był: czas obliczeń algorytmu A* w milisekundach, liczba odwiedzonych węzłów w procesie wyszukiwania oraz długość obliczonej ścieżki w węzłach. Liczba odwiedzonych węzłów została wyrażona procentową wartością w stosunku do rozmiaru środowiska (kolumna: przeszukany obszar).



Rysunek 23 Pierwsze środowisko z wygenerowaną ścieżką

Eksperyment 1.1

Tabela 2 przedstawia wyniki eksperymentu 1.1 w środowisku 1 o rozmiarze 50x50 bez włączonych modyfikatorów ścieżki dla dziesięciu agentów. Czasy wykonywania obliczeń są mierzone w milisekundach i dla heurystyki euklidesowej 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 euklidesową.

Tabela 2 Wyniki badań dla eksperymentu 1.1 (środowisko 1, rozmiar 50x50, bez modyfikatorów)

Rozmiar	50x50								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	1,59	48,44	67	2,1	65,28	67	1,7	55,04	67

Eksperyment 1.2

Tabela 3 przestawia symulacje dla eksperymentu 1.2. Rozmiar środowiska 1 wynosi 50x50. Do agentów zostały dołączone komponenty skrypty modyfikacyjne (opisane w rozdziale 4.4.1). Zestawiając eksperymenty 1.1 oraz 1.2, 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 20% węzłów więcej.

Rysunek 23 przedstawia wygenerowaną ścieżkę (kolor zielony) z eksperymentu 1.2 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.

Tabela 3 Wyniki badań dla eksperymentu 1.2 (środowisko 1, rozmiar 50x50, włączone modyfikatory)

Rozmiar	50x50								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	5	48,44	67	5	65,28	69	6,01	55,04	67
Agent 2.	2	52,04	69	2	70,52	67	2	56,76	69
Agent 3.	0,99	54,04	69	3	75,84	69	2	58,32	67
Agent 4.	1	55,2	67	2	75,6	67	1	58,64	69
Agent 5.	2	56,56	69	2	79,04	71	1	60,48	69
Agent 6.	1	57,68	68	2	77,32	68	2	58,92	67
Agent 7.	1	56,92	69	2	82,88	68	2	59,28	69
Agent 8.	2,01	57,64	69	3,01	82,76	69	1	59,84	68
Agent 9.	1	58,44	70	1,98	80,84	71	1	59,96	69
Agent 10.	1	59,16	70	2	83,08	68	1	62	67
Wartości średnie	**1,7**	**55,612**	**68,7**	**2,49**	**77,316**	**68,7**	**1,90**	**58,924**	**68,1**

Eksperyment 1.3

Tabela 4 przestawia wyniki eksperymentu 1.3 Rozmiar badanego środowiska 1 wynosi 100x100. Do eksperymentu nie wykorzystano skryptów modyfikacyjnych. Eksperyment ten posiada identyczne parametry jak eksperyment 1.1, jednak algorytm ma teraz do przeszukania 4 razy więcej węzłów niż w poprzednim wypadku. Można zauważyć, że liczba odwiedzonych węzłów wzrosła znacząco. W przypadku metody

Manhattan przyrost przekroczył 5000 węzłów, co jest procentowo dużą liczbą (~25%). Czasowo najlepsze wyniki uzyskała heurystyka euklidesowa.

Tabela 4 Wyniki badań dla eksperymentu 1.3 (środowisko 1, rozmiar 100x100, bez modyfikatorów)

Rozmiar	100x100								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	5,39	48,87	132	8,01	73,89	132	7,47	58,39	132

Eksperyment 1.4

Tabela 5 przedstawia wyniki eksperymentu 1.4 przeprowadzonego w środowisku 1 o rozmiarach 100x100. W eksperymencie zastosowano skrypty modyfikacyjne, co obciążyło czasowo najbardziej heurystykę diagonal Manhattan i uzyskała ona najgorszy wynik czasowy ze wszystkich heurystyk. Pomimo tego heurystyka Manhattan w dalszym ciągu odwiedza największą liczbę węzłów.

Tabela 5 Wyniki badań dla eksperymentu 1.4 (środowisko 1, rozmiar 100x100, włączone modyfikatory)

Rozmiar	100x100								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	8,99	48,87	131	10,96	73,89	131	12	58,39	131
Agent 2.	3,99	50,97	133	7,98	75,78	132	7	59,92	133
Agent 3.	6	51,96	137	8,98	77,71	137	10,01	60,89	132
Agent 4.	6	53,79	135	8,99	78,04	132	14,01	61,2	137
Agent 5.	7	55,34	138	9,97	80,14	134	16,01	62,25	135
Agent 6.	7	56,63	134	7,98	79,79	137	11	68,9	137
Agent 7.	6,99	57,39	138	10,97	85,42	133	9,01	64,25	138
Agent 8.	5,99	58,39	138	10,98	88,15	138	16,01	63,76	134
Agent 9.	7	58,97	136	12,04	89,14	137	10	64,26	136
Agent 10.	7,01	59,89	140	10,98	88,32	139	8,98	64,57	140
Wartości średnie	**6,59**	**55,22**	**136**	**9,98**	**81,638**	**135**	**11,4**	**62,839**	**135,3**

Eksperyment 1.5

Tabela 6 przedstawia wyniki eksperymentu 1.5 dla środowiska gry 1 o rozmiarach

200x200. Jest to największe środowisko na jakim zostaną przeprowadzone badania. Porównując obecny rezultat z eksperymentem 1.1 można zauważyć, że czasy obliczeń są kilkunastokrotnie większe. Zwiększenie rozmiaru środowiska gry spowodowało, że heurystyka diagonal Manhattan osiągnęła najdłuższy czas wykonania.

Tabela 6 Wyniki badań dla eksperymentu 1.5 (środowisko 1, rozmiar 200x200, bez modyfikatorów)

Rozmiar	200x200								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci:	21,84	50,03	263	39,06	31313	263	44,89	24318	263

Eksperyment 1.6

Dodając do środowiska numer 1 skrypty modyfikujące można zauważyć wzrost czasu obliczeń. W przypadku heurystyki euklidesowej wzrost ten zwiększył się o 14[ms], Manhattan o 15[ms]. Natomiast heurystyka diagonal Manhattan wykonała się wolnej tylko o 5[ms]. Wyniki eksperymentu 1.6 przedstawia tabela 7.

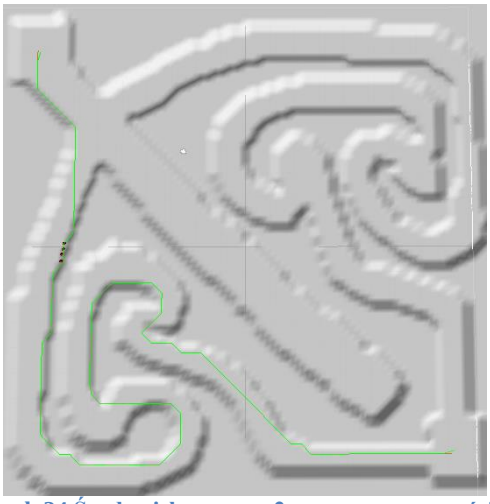
Analizując wyniki eksperymentu 1.6 można dojść do wniosku, że najlepsze wyniki osiągnęła heurystyka wykorzystująca odległość euklidesową.

Tabela 7 Wyniki badań dla eksperymentu 1.6 (środowisko 1, rozmiar 200x200, włączone modyfikatory)

Rozmiar	200x200								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	32,96	50,03	263	34,96	78,28	263	57,96	60,795	263
Agent 2.	42,99	50,77	264	34,84	80,55	264	32,52	60,695	264
Agent 3.	22,01	51,27	266	37,93	82,42	272	61,99	60,925	265
Agent 4.	28,12	52,18	272	47,93	85,30	264	32,97	61,6825	272
Agent 5.	27,21	52,96	269	54,88	87,28	268	48,9	61,9875	268
Agent 6.	27,73	53,34	274	41,88	86,34	272	55,92	61,8525	273
Agent 7.	37,08	54,07	271	66,8	89,29	270	62,05	61,8275	272
Agent 8.	43,03	55,19	273	49,96	86,84	274	40	62,5175	275
Agent 9.	40,98	56,26	274	97,97	90,5	275	60,03	62,52	273
Agent 10.	52,08	56,89	274	81,05	88,17	270	41,99	62,8975	277
Wartości średnie:	**35,41**	**53,299**	**270**	**54,82**	**85,50**	**269,2**	**49,43**	**61,77**	**270,2**

Eksperyment 2.1

Tabela 8 przedstawia wyniki eksperymentu 2.1. W tym eksperymencie badane jest środowisko 2 o rozmiarze 50x50 przy wyłączonych skryptach modyfikacyjnych. Porównując wyniki z eksperymentem 1.1 można zauważyć, że osiągnięte czasy są podobne. Podobnie jak w przypadku wcześniejszych eksperymentów heurystyka Manhattan dokonuje przeglądu największej liczby węzłów. Jednak czasowo najszybciej wykonał się algorytm korzystający z heurystyki diagonalnej Manhattan, tym samym odwiedzając najmniejszą liczbę węzłów. Środowisko, na którym przeprowadzono badania reprezentuje rysunek 24.



Rysunek 24 Środowisko numer 2 z wygenerowaną ścieżką

Tabela 8 Wyniki badań dla eksperymentu 2.1 (środowisko 2, rozmiar 50x50, bez modyfikatorów)

Rozmiar	50x50								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	1,7	53,92	117	1,827	61,48	117	1,495	51,36	117

Eksperyment 2.2

W tabeli 9 przedstawiono wyniki działania eksperymentu 2.2. Rozmiar środowiska 2 gry wynosi 50x50 oraz zostały włączone skrypty modyfikacyjne. Porównując symulacje z eksperymentem 2.1, heurystyka Diagonal Manhattan osiągnęła lepszy wynik. Przypadek badawczy wykorzystujący odległość euklidesową jest minimalnie gorszy. Natomiast proces odnajdywania ścieżki wykorzystującego heurystykę Manhattan wykonał się najwolniej przeglądając największą liczbę węzłów. Dodanie skryptów

modyfikacyjnych spowodowało, że algorytm przeszukał o około 7% więcej obszaru środowiska gry.

Tabela 9 Wyniki badań dla eksperymentu 2.2 (środowisko 2, rozmiar 50x50, włączone modyfikatory)

Rozmiar	50x50								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	5	53,92	117	6	61,48	117	5	51,36	117
Agent 2.	2	56,72	117	1	63,76	117	1	55,68	117
Agent 3.	1	57,04	118	2	63,28	117	2	53,48	117
Agent 4.	2	59,64	118	1,96	67,84	119	1	59,52	119
Agent 5.	1	59,36	119	1,99	67,92	119	2	59,68	118
Agent 6.	1	59,4	120	3,03	67,36	120	1	59,44	118
Agent 7.	2	61,04	117	2	70,96	118	1	60,28	119
Agent 8.	1	61	121	3	70,04	119	1	61,52	119
Agent 9.	2	61,12	131	3	70,04	119	1	61,64	118
Agent 10.	1,99	63	122	1	72,04	122	2	63,08	131
Wartości średnie	**1,899**	**59,22**	**120**	**2,498**	**67,47**	**118**	**1,7**	**58,568**	**119**

Eksperyment 2.3

Wyniki eksperymentu 2.3 w środowisku 2 zostały zawarte w tabeli 10. Rozmiar badanego środowiska 2 wynosi 100x100. W eksperymencie nie wykorzystano skryptów modyfikacyjnych. Można zauważyć, że czasy dla heurystyki liczonej odległością euklidesową oraz diagonalnym Manhattan czasowo zwracają podobne wartości. Jednak biorąc pod uwagę liczbę odwiedzonych węzłów to diagonalny Manhattan odwiedza ich najmniej.

Tabela 10 Wyniki badań dla eksperymentu 2.3 (środowisko 2, rozmiar 100x100, bez modyfikatorów)

Rozmiar	100x100								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	6,083	59,43	231	10,58	69,17	231	6,296	58,17	231

Eksperyment 2.4

Tabela 11 przestawia wyniki eksperymentu 2.4. Na potrzeby eksperymentu zostały włączone skrypty modyfikujące. Rozmiar środowiska gry 2 wnosi 100x100. Najlepszy wynik uzyskała heurystyka euklidesowa, natomiast minimalnie gorsza okazała się

diagonalna Manhattan. Włącznie skryptów modyfikujących obciążyło jednakowo heurystyki posiadające najlepsze wyniki czasowe. Heurystyka Manhattan osiągnęła najgorszy wynik czasowy oraz przetworzyła największą ilość węzłów.

Tabela 11 Wyniki badań dla eksperymentu 2.4 (środowisko 2, rozmiar 100x100, włączone modyfikatory)

Rozmiar	100x100								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	12	59,43	231	10,99	69,17	231	15	58,17	231
Agent 2.	7	60,57	231	8	73,65	231	9,01	59,1	231
Agent 3.	6,97	61,38	234	7,98	77,32	235	8	60,21	235
Agent 4.	7,01	62,07	233	10	88,02	235	7,99	62,45	234
Agent 5.	6,97	64,51	234	11	84	235	8,81	63,48	236
Agent 6.	9	64,96	237	9,93	82,69	240	7,93	64,69	236
Agent 7.	8,01	65,81	235	11	87,16	241	10	65,45	234
Agent 8.	9	65,32	238	11	84,49	234	7	65,53	241
Agent 9.	8	66,25	260	11,04	80,81	236	6	67,39	260
Agent 10.	8,01	67,31	245	10,97	83,86	239	7	67,24	241
Wartości średnie	**8,197**	**63,761**	**237,8**	**10,191**	**81,117**	**235,7**	**8,674**	**63,371**	**237,9**

Eksperyment 2.5

Tabela 12 przedstawia wyniki eksperymentu 2.5 dla środowiska 2 o rozmiarze 200x200. W eksperymencie nie wykorzystano skryptów modyfikujących. Wzrost obszaru przeszukiwań nie wyłonił najlepszej heurystyki. Wyniki osiągnięte przez heurystyki diagonalny Manhattan oraz euklidesową są porównywalne.

Tabela 12 Wyniki badań dla eksperymentu 2.5 (środowisko 2, rozmiar 200x200, bez modyfikatorów)

Rozmiar	200x200								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	31,04	60,27	461	39,12	74,75	461	30,88	59,76	461

Eksperyment 2.6

Wyniki eksperymentu 2.6 zostały umieszczone w tabeli 13. Badania zostały

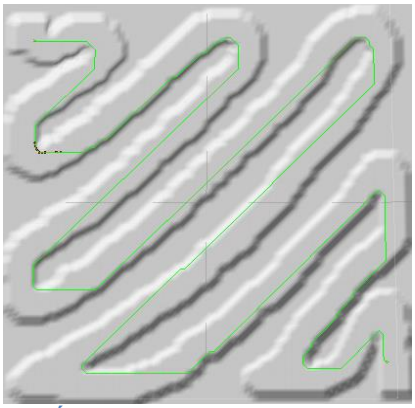
przeprowadzone na środowisku 2 o rozmiarze 200x200. Dodanie skryptów modyfikacyjnych spowodowało, że minimalnie lepsza od heurystyki euklidesowej okazała się heurystyka diagonal Manhattan. Natomiast najdłuższy czas wykonania osiągnęła heurystyka Manhattan.

Tabela 13 Wyniki badań dla eksperymentu 2.6 (środowisko 2, rozmiar 200x200, włączone modyfikatory)

Rozmiar	200x200								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	31,75	60,27	461	33,94	74,75	461	30,67	59,76	461
Agent 2.	32,93	60,72	467	39,14	75,47	466	24,98	60,19	465
Agent 3.	34,94	61,80	466	42,91	80,39	470	24,99	60,30	467
Agent 4.	35,05	61,73	473	43,91	83,56	471	30,99	61,19	467
Agent 5.	32,95	61,88	472	47,87	89,27	475	28,99	61,22	474
Agent 6.	32,98	62,53	476	44,81	87,78	480	46,99	61,88	475
Agent 7.	34,96	62,77	475	45,91	89,72	477	48,98	62,27	477
Agent 8.	35,99	63,10	485	52,93	96,21	489	30,96	62,67	483
Agent 9.	34	62,69	486	66,94	101,72	489	32	64,14	487
Agent 10.	35,94	63,23	487	54,8	103,62	489	33,98	63,9	488
Wartości średnie	**34,14**	**62,07**	**474,8**	**47,316**	**88,25**	**476,7**	**33,35**	**61,75**	**474,4**

Eksperyment 3.*

Rysunek 25 przedstawia środowisko gry 3 z wygenerowaną ścieżką. Agenci rozpoczynają ruch z lewego górnego narożnika. Warto zwrócić uwagę na strukturę modelu środowiska. Nie zawiera ona alternatywnych dróg – postacie podążając jedną ścieżką docierają do punktu docelowego. Algorytm sumarycznie ma w tym przypadku do przeszukania procentowo większy obszar w porównaniu do poprzednich eksperymentów. Środowisko numer 3 jest pierwszym z środowisk o bardziej skomplikowanej strukturze terenu. Na rysunku można zauważyć zieloną linie oznaczającą obliczoną ścieżkę.



Rysunek 25 Środowisko numer 3 z wygenerowaną ścieżką

Eksperyment 3.1

Tabela 14 zawiera wyniki eksperymentu 3.1 w środowisku 3 o rozmiarze 50x50. W tej symulacji heurystyka euklidesowa, 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.

Tabela 14 Wyniki badań dla eksperymentu 3.1 (środowisko 3, rozmiar 50x50, bez modyfikatorów)

Rozmiar	50x50								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	1,597	46,52	224	1,596	47,84	224	1,398	45,68	224

Eksperyment 3.2

W tabeli 15 zostały zaprezentowane wyniki eksperymentu 3.2 dla środowiska 3 przy włączonych modyfikatorach ścieżki. Narzut czasowy wynikający z zastosowania skryptów jest niewielki i wynosi on od 0,7[ms] do 1.5[ms]. Wszystkie badane heurystyki osiągnęły podobne czasy obliczeń. Warto zwrócić uwagę, że heurystyka Manhattan odwiedziła więcej węzłów od pozostałych heurystyk.

Tabela 15 Wyniki badań dla eksperymentu 3.2 (środowisko 3, rozmiar 50x50, włączone modyfikatory)

Rozmiar	50x50								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	5	47,28	224	5	50	224	5,01	46,4	224
Agent 2.	0,99	50,96	225	3	62,88	225	2	49,08	225
Agent 3.	3	51,8	225	2	64,44	226	1	51,44	225
Agent 4.	2	53,04	228	2	57,52	229	2	51,64	226
Agent 5.	1	52,92	228	2	68,4	230	1	52,72	228
Agent 6.	2	54,76	228	3	67,32	227	1	53,04	229
Agent 7.	3	59,32	231	4,01	60,72	229	1	53,04	225
Agent 8.	3	62,28	231	3	65,2	231	3	55,36	230
Agent 9.	3	59,84	232	4,02	70,8	228	2	59,16	231
Agent 10.	2	63,52	233	1	62,32	232	2	58,64	233
Wartości średnie	**2,499**	**55,57**	**228,5**	**2,903**	**62,96**	**228,1**	**2,001**	**53,05**	**227,6**

Eksperyment 3.3

W eksperymencie 3.3 zbadane zostało środowisko gry numer 3. Rozmiar środowiska wynosi 100x100. Do badania nie wykorzystano skryptów modyfikujących. Wartości w tabeli 16 wskazują na przewagę heurystyki diagonalnej Manhattan. Heurystyka Manhattan podobnie jak w pozostałych przypadkach przetwarza największą liczbę węzłów oraz posiada wysoką wartość czasu obliczeń.

Tabela 16 Wyniki badań dla eksperymentu 3.3 (środowisko 3, rozmiar 100x100, bez modyfikatorów)

Rozmiar	100x100								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	5,883	49,46	452	7,488	55,68	452	5,291	47,66	452

Eksperyment 3.4

Wyniki symulacji dla środowiska 3 o rozmiarze 100x100 z włączonymi skryptami modyfikującymi przedstawione zostały w tabeli 17. Wskazują one, że heurystyki euklidesowa oraz diagonalna Manhattan wykonały się w podobnych czasach. Analizując eksperyment 3.4 można dojść do wniosku, że dodanie skryptów modyfikacyjnych jest bardziej obciążające czasowo dla heurystyki diagonalnej Manhattan.

Tabela 17 Wyniki badań dla eksperymentu 3.4 (środowisko 3, rozmiar 100x100, włączone modyfikatory)

Rozmiar	100x100								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	10	49,8	452	10	56,07	452	7,01	50,97	452
Agent 2.	6,99	51,73	453	8,99	70,06	453	6	52,11	453
Agent 3.	6,97	52,15	456	11,01	70,5	454	7,01	53,14	457
Agent 4.	7	54,47	457	8,98	71,82	456	7,98	55,22	458
Agent 5.	7	55,35	460	9,96	69,83	459	7,9	55,94	458
Agent 6.	7,99	57,47	465	10,01	69,47	458	7	56,67	461
Agent 7.	7	59,89	461	10,01	70,18	466	7	60,34	462
Agent 8.	7	63,21	470	18,02	76,76	464	7,9	63,62	472
Agent 9.	5,98	65,34	475	13,02	78,01	470	7,01	63,82	482
Agent 10.	7	60,72	464	15,02	71,74	473	7,9	64,2	471
Wartości średnie	**7,29**	**57,01**	**461,3**	**11,50**	**70,44**	**460,5**	**7,27**	**57,60**	**462,6**

Eksperyment 3.5

Eksperyment 3.5 został wykonany w środowisku numer 3 o rozmiarze 200x200. W eksperymencie nie wykorzystano skryptów modyfikujących. Analizując wyniki umieszczone w tabeli 18 można zauważyć stopniową przewagę heurystyki diagonalnej Manhattan. W poprzednich eksperymentach, gdzie rozmiar środowiska wynosił 200x200 heurystyka ta uzyskiwała większe czas obliczeń, w tym przypadku jest jednak inaczej.

Tabela 18 Wyniki badań dla eksperymentu 3.5 (środowisko 3, rozmiar 200x200, bez modyfikatorów)

Rozmiar	200x200								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	42,66	50,56	901	67,36	59,49	901	40,81	52,16	901

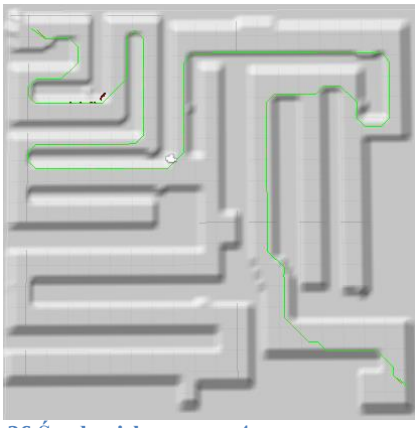
Eksperyment 3.6

Do agentów poruszających się po środowisku 3 o rozmiarach 200x200 dodano skrypty modyfikujące. Wyniki symulacji poszukiwania drogi znajdują się w tabeli 19. Warto zwrócić uwagę, że dodanie skryptów modyfikujących najmniej obciążyło heurystykę diagonal Manhattan. Natomiast pozostałe heurystyki wykonały się o 15[ms](euklidesowa) i 11[ms] (Manhattan) dłużej.

Tabela 19 Wyniki badań dla eksperymentu 3.6 (środowisko 3, rozmiar 200x200, włączone modyfikatory)

Rozmiar	200x200								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	28,87	50,56	901	39,99	59,49	901	27,98	52,16	901
Agent 2.	34,87	51,05	904	31,92	64,85	904	40,04	52,55	904
Agent 3.	37,95	51,53	907	73,93	70,43	905	64,14	53,09	907
Agent 4.	54,92	53,16	912	81,08	73,63	912	47,05	53,58	908
Agent 5.	60,95	53,77	911	70,89	72,46	915	42,98	54,55	915
Agent 6.	64,87	54,15	915	80,92	72,27	916	33,05	55,4	920
Agent 7.	68,94	55,09	915	73,94	74,16	916	69,08	55,98	920
Agent 8.	77,01	57,42	919	78,04	75,06	924	35,01	57,14	924
Agent 9.	60,95	62,49	929	88,07	76,84	928	39,86	57,395	926
Agent 10.	80,99	64,42	933	127,57	109,39	935	47,04	55,21	940
Wartości średnie	**57,03**	**55,36**	**914,6**	**78,48**	**74,86**	**915,6**	**44,62**	**54,70**	**916,5**

Eksperyment 4.*



Rysunek 26 Środowisko numer 4 z wygenerowaną ścieżką

Ostatnim środowiskiem, na którym są przeprowadzane eksperymenty jest środowisko gry posiadające strukturę labiryntu (rysunek 25). Agenci zostali umieszczani w lewym górnym rogu, a ich celem jest dotarcie do punktu końcowego znajdującego się w dolnym prawym rogu środowiska gry. Zielona linia oznacza ścieżkę, którą podążają agenci.

Eksperyment 4.1

Eksperymentu 4.1 został przeprowadzony na środowisku 4 (rysunek 26) o rozmiarze 50x50. W eksperymencie nie zastosowano skryptów modyfikacyjnych. Wyniki badań zostały przedstawione w tabeli 20. Największy czas na uzyskała heurystyka Manhattan. Natomiast porównywalne wyniki uzyskały dwie pozostałe heurystyki.

Tabela 20 Wyniki badań dla eksperymentu 4.1 (środowisko 4, rozmiar 50x50, bez modyfikatorów)

Rozmiar	50x50								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	3,502	33,6	181	4,102	39,08	181	3,501	33,92	181

Eksperyment 4.2

W tabeli 21 zostały umieszczone wyniki eksperymentu 4.2. W testowanym środowisku 4 zostały dodane skrypty modyfikujące. Rozmiar środowiska wynosi 50x50. Dodanie skryptów modyfikacyjnych w przypadku takich rozmiarów środowiska gry obciążyło czasowo heurystykę diagonalną Manhattan. Uzyskała ona gorszy czas od najlepszej w tym przypadku heurystyki euklidesowej.

Tabela 21 Wyniki badań dla eksperymentu 4.2 (środowisko 4, rozmiar 50x50, włączone modyfikatory)

Rozmiar grafu:	50x50								
Modyfikator:	Funnel, Alternative Path								
Heurystyka:	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	14,01	36,44	181	13,01	40,12	181	14,01	34,76	181
Agent 2.	2	38,32	181	5	65,32	181	3	39,56	181
Agent 3.	2	39,24	181	4,01	67,16	181	2	42,08	181
Agent 4.	3	39,28	184	7	62,48	181	4	40,8	183
Agent 5.	2	40,92	183	3,01	55,44	184	2	43,08	183
Agent 6.	3	42,04	186	4	49,16	184	4	42,96	184
Agent 7.	2	41,48	184	4	70,88	183	3	43,2	185
Agent 8.	5	41,64	184	6	68,24	186	3	44,4	186
Agent 9.	3	43,04	188	4	70,36	185	3	44,52	183
Agent 10.	2	43,32	188	4	67,2	185	2	46	187
Wartości średnie	**3,8**	**40,57**	**184**	**5,403**	**61,63**	**183,1**	**4,001**	**42,13**	**183,4**

Eksperyment 4.3

Kolejna symulacja została przeprowadzona w środowisku numer 4 o rozmiarach 100x100. W eksperymencie 4.3 z powodu braku skryptów modyfikujących, wyniki heurystyk euklidesowej i diagonalnej Manhattan są zbliżone (tabela 22). Natomiast algorytm w przypadku wykorzystania heurystyki Manhattan przeszukuje procentowo większy rozmiar środowiska gry.

Tabela 22 Wyniki badań dla eksperymentu 4.3 (środowisko 4, rozmiar 100x100, bez modyfikatorów)

Rozmiar	100x100								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	11,29	34,63	363	15,41	42,94	363	11,91	35,92	363

Eksperyment 4.4

Odnosząc się do poprzednich eksperymentów można było się spodziewać, że dodanie skryptów modyfikujących obciąży czasowo bardziej heurystykę diagonalną Manhattan i osiągnie gorszy czas wykonania. Warto zwrócić uwagę, że w eksperymencie

4.4 zwiększanie rozmiaru środowiska gry 4 do 100x100, powoduje heurystyka diagonalna Manhattan osiąga najlepsze wyniki (tabela 23).

Tabela 23 Wyniki badań dla eksperymentu 4.4 (środowisko 4, rozmiar 100x100, włączone modyfikatory)

Rozmiar	100x100								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	22	35,38	363	25,01	43,51	363	22,04	36,4	363
Agent 2.	10,01	36,62	364	32,04	90	363	10,01	38,02	363
Agent 3.	10,02	38,6	367	31,04	92,46	368	10,01	38,88	370
Agent 4.	12,01	37,83	372	35,09	89,91	366	12,02	41,2	367
Agent 5.	12,01	39,27	374	32,05	92,2	370	12,01	39,4	369
Agent 6.	11,02	39,74	375	31	87,9	374	11	40,04	370
Agent 7.	10,01	40,3	380	40,03	111,15	370	11,02	40,94	375
Agent 8.	13,01	40,4	376	39,31	107,53	371	12,01	41,36	370
Agent 9.	11,01	41,54	377	45,13	113,43	373	11,01	41,74	374
Agent 10.	10,95	41,77	379	40,05	109,71	372	11,02	42,66	378
Wartości średnie	**12,2**	**39,14**	**372,7**	**35,075**	**93,78**	**369**	**12,21**	**40,06**	**369,9**

Eksperyment 4.5

Tabela 24 zawiera wyniki badań przeprowadzonych dla największych rozmiarów środowiska 4 : 200x200. Najlepszy wynik czasowy osiągnęła heurystyka diagonalna Manhattan, co potwierdzaj jej krótszy czas obliczeń w środowiskach o dużych rozmiarach. Heurystyka euklidesowa osiągnęła o 6[ms] gorszy czas w porównaniu do najlepszego wyniku dla omawianej symulacji.

Tabela 24 Wyniki badań dla eksperymentu 4.5 (środowisko 4, rozmiar 200x200, bez modyfikatorów)

Rozmiar	200x200								
Modyfikator	brak modyfikatora								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agenci	46,943	35,78	716	66,78	47,71	716	40,096	37,17	716

Eksperyment 4.6

Dodanie skryptów modyfikacyjnych do środowiska 4 pozwoliło na uzyskanie wyników umieszczonych w tabeli 25. 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 euklidesowej.

Tabela 25 Wyniki badań dla eksperymentu 4.6 (środowisko 4, rozmiar 200x200, włączone modyfikatory)

Rozmiar	200x200								
Modyfikator	Funnel, Alternative Path								
Heurystyka	Euclidean			Manhattan			Diagonal Manhattan		
	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość	Czas [ms]	Przeszukany obszar [%]	Długość
Agent 1.	59,04	36,48	716	79,06	48,77	717	41,05	38	716
Agent 2.	44,07	38,08	727	76,46	53,14	751	47,97	38,51	721
Agent 3.	50,08	40,63	726	83,1	62,06	732	50,06	39,87	724
Agent 4.	51,03	40,44	736	89,09	62,55	731	53,02	39,25	724
Agent 5.	57,07	41,15	744	97,11	63,89	741	50,04	40,41	734
Agent 6.	58,06	40,79	737	90,8	63,12	751	51,74	41,52	745
Agent 7.	48,05	40,27	747	178,15	120,28	752	42,06	40,16	746
Agent 8.	52,05	41,24	748	183,2	125,57	749	43,21	40,65	749
Agent 9.	52,01	40,74	757	194,15	125,52	752	46,05	40,62	755
Agent 10.	57,04	41,23	756	194,11	127,02	769	45,05	40,53	753
Wartości średnie	**52,85**	**40,11**	**739,4**	**126,52**	**85,19**	**744,5**	**47,02**	**39,95**	**736,7**

5.4 Eksperyment 2 – badanie algorytmu Jump Point Search

W tym eksperymencie zostanie przeanalizowany ten sam zestaw środowisk gry co w eksperymentach 1.1-4.6. Na każdym z środowisk zostanie wywołany skrypt z algorytmem JPS. Rozmiar środowisk będzie zwiększany identycznie jak w poprzednich eksperymentach (1.1-4.6). Dzięki temu jest możliwe dokonanie porównania czasowego algorytmu A* (eksperymenty przedstawione w rozdziale 5.3) oraz algorytmu Jump Point Search. Eksperyment dokonuje analizy heurystyki diagonal Manhattan.

Eksperymenty były przeprowadzane dla dziesięciu agentów. Dla każdego z agentów został zapisany wynik obliczeń algorytmu znajdującego ścieżkę do celu. Wynik czasowy został uśredniony. W celach porównawczych, w ostatnim wierszu zestawu tabel 26-29 został zapisany uśredniony wynik algorytmu A*, będący średnią wartością otrzymaną w eksperymentach 1.1-4.6 dla heurystyki diagonalnej Manhattan przy wyłączonych skryptach modyfikacyjnych.

Eksperyment 5.1

Tabela 26 przedstawia wyniki eksperymentu 5.1 przeprowadzanego w środowisku numer 1. Eksperyment został przeprowadzony na środowisku 1 o rozmiarach 200x200, 100x100 oraz 50x50. Można zauważyć, że dla małych i średnich środowisk wynik czasowy jest identyczny. W przypadku środowiska o największym rozmiarze algorytm JPS wykonuje się o 19[ms] szybciej.

Tabela 26 Wyniki działania algorytmu JPS, eksperyment 5.1

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 – eksperyment 1.5, 1.3, 1.1:	44,89	7,473	1,7

Eksperyment 5.2

Eksperyment 5.2 został przeprowadzony w środowisku gry numer 2, którego wyniki zostały zawarte w tabeli 27. Eksperyment wykonał się dla rozmiarów środowiska: 200x200, 100x100, 50x50. Można zauważyć, że algorytm JPS zwraca lepszy wynik, niż rezultat otrzymany przez algorytm A*. W przypadku środowiska średniej wielkości wynik czasowy dla algorytmu JPS jest o 1[ms] lepszy (wykonał się 10% szybciej) . Warto zauważyć, że algorytm JPS uzyskuje najlepsze przyśpieszenie dla dużych środowisk.

Tabela 27Wyniki działania algorytmu JPS, eksperyment 5.2

Rozmiar:	200x200	100x100	50x50
Agent nr:	Rezultat czasowy [ms]		
1	19,84	8,52	4,72
2	13,85	4,82	1,47

Rozmiar:	200x200	100x100	50x50
Agent nr:	Rezultat czasowy [ms]		
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 – eksperyment 2.5, 2.3, 2.1:	30,88	6,29	1,495

Eksperyment 5.3

Przed analizą eksperymentu 5.3 warto zwrócić uwagę na strukturę środowiska gry numer 3. Posiada ono 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 eksperymentu uzyskane czasy (tabela 28) dla rozmiarów 50x50 i 100x100 są mniejsze od tych z algorytmu A*. Analizując środowisko o rozmiarze 200x200 algorytm osiągnął mniejszy czas zestawiając go z wynikiem z eksperymentu 3.5, jednak nie jest on tak niski jak w przypadku mniejszych środowisk.

Tabela 28 Wyniki działania algorytmu JPS, eksperyment 5.3

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 – eksperyment 3.5, 3.3, 3.1:	40,81	5,291	1,398

Eksperyment 5.4

Wyniki eksperymentu 5.4 w środowisku 4 znajdują się w tabeli 29. Eksperyment został przeprowadzony dla rozmiarów środowisk 200x200, 100x100 oraz 50x50. Środowisko to zawiera skomplikowaną strukturę, więc uzyskane rezultaty czasowe są większe od poprzednio analizowanych. Zestawiając wyniki dla poszczególnych środowisk można zauważyć, że dla każdego rozmiaru środowiska zastosowanie algorytmu JPS skraca czas obliczeń. Przyśpieszenie czasowe zwiększa się wraz z rozmiarem środowiska gry.

Tabela 29 Wyniki działania algorytmu JPS, eksperyment 5.4

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 – eksperyment 4.5, 4.3, 4.1:	40,09	11,91	3,501

5.5 Wnioski

Badania nad heurystykami algorytmu A* wykazały, że dla środowisk posiadających prostą strukturę i niewielkie rozmiary najlepszym rozwiązaniem jest zastosowanie heurystyki euklidesowej. Uzyskuje ona najmniejszy czas przeznaczony na odnalezienie ścieżki. Odnośnie skryptów modyfikujących, które poprawiają jakość ścieżek, warto wspomnieć o tym, że dla środowisk o prostej strukturze narzut czasowy związany z zastosowaniem tych skryptów nie jest wysoki (od 0,2[ms] do 3[ms], co stanowi 20% średniej czasu poświęconego na wykonanie algorytmu). Jednak w przypadku środowisk gier o dużych rozmiarach (200x200) zastosowanie skryptu modyfikującego obciążyło obliczeniowo heurystykę euklidesową, aż o 15[ms] (co stanowi 40% czasu). Podczas gdy obciążenie czasowe heurystyki diagonalnej Manhattan wyniosło zaledwie 5[ms] (10 % czasu). W tym przypadku programista powinien się zastanowić nad wyborem heurystyki, ponieważ dla większych środowisk obciążenie czasowe heurystyki

euklidesowej może spowodować spadek efektywności algorytmu na korzyść jednej z pozostałych heurystyk.

Dalsza część badań dotyczyła środowisk o złożonej strukturze (np. środowisko gry o strukturze labiryntu). Wyniki badań wykazały, że dla małych i średnich środowisk gry zarówno heurystyka euklidesowa 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], co stanowi 4% narzutu czasowego. Natomiast heurystyka Manhattan jak do tej pory uzyskała najgorszy rezultat czasowy, co więcej narzut wynikający z zastosowania skryptów wyniósł, aż 20[ms] (~57% wolniej). W przypadku środowisk 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 środowisk było ono niewielkie. Natomiast w przypadku dużych środowisk wynosiło ono około 50%. Interesującym faktem jest, że przyśpieszenie równe 80% zostało zanotowane dla jednego z środowisk o skomplikowanej strukturze. Struktura ta była o tyle specyficzna, że posiadała rozległą przestrzeń. Dzięki czemu algorytm JPS mógł wykonać duże „skoki” w środowisku gry 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, co jest bardziej odczuwalne w przypadku dużych środowisk gry.

6 Podsumowanie

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 zostało zaprojektowane środowisko badawcze, które umożliwia przeprowadzanie eksperymentów na badanych algorytmach. Zostały przeprowadzone pomiary czasu wykonania algorytmów oraz liczby odwiedzonych węzłów w grafie. Eksperymenty te zostały wykonane na czterech różnych środowiskach gry. Środowiska te różniły się w swoje strukturze. Pierwsze z nich posiadały mniej skomplikowaną strukturę, a ostatnia miała formę labiryntu. Dodatkowo obszar poszukiwań dla każdego środowiska był zwiększany z rozmiaru 50x50 do 100x100 oraz następnie do 200x200.

Badania wykazały, że struktura środowiska gry oraz zastosowana funkcja heurystyczna mają wpływ na wynik czasowy algorytmu A*. W przypadku środowisk o dużych rozmiarach różnice czasowe są większe. Zastosowanie algorytmu JPS pozwoliło przyśpieszyć proces wyszukiwania ścieżki o wartość od 7% do 80% w zależności od zastosowanego środowiska gry.

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, co może rzutować na porażkę danej gry komputerowej. Na czas rozwiązania mają wpływ czynniki takie jak: rozmiar i poziom skomplikowania środowiska gry, zastosowana funkcja heurystyczna oraz dodatkowe przetwarzanie drogi mające na celu poprawienie jej jakości (np. wygładzanie ścieżek). Istotnym elementem systemu nawigacji jest czas w jakim zostaje odnaleziona ścieżka w środowisku gry. Czas ten może zostać zmniejszony w przypadku implementacji algorytmu Jump Point Search, który został przedstawiony w tej pracy.

Bibliografia

1. **Ian Millington**, *Artifical Intelligence for Games*, Elsevier 2006.
2. **Daniel Harabor, Alban Grasiten**, *Online Graph Pruning for Pathfinding on Grid Maps*, NICTA, The Australian National University, [Online] http://grastien.net/ban/articles/hg-aaai11.pdf [06.10.2013].
3. **DeLoura M.**, *Perełki programowania gier – Vademecum profesjonalisty*, Helion 2002.
4. **A.J. Champandard**, *Top 10 Most Influential AI Games*, [Online] http://aigamedev.com/open/highlights/top-ai-games/ [05.10.2013].
5. **Daniel Harabor,** Shortest Path, 2011 [Online] http://harablog.wordpress.com/category/pathfinding/ [06.10.2013].
6. **Nikhil Krishnaswamy,** *Comparison of Efficieny in Pathfinding Algorithms in Game Development*, 2009.
7. **James Wexler,** *Artifical Intelligence in Games*, [Online] http://www.cs.rochester.edu/~brown/242/assts/termprojs/games.pdf [06.10.2013].
8. **Czapelski M.,** Gry na serio, 2004, [Online] http://www.pcworld.pl/artykuly/38243/Gry.na.serio.html [06.10.2013].

Dodatek A.

Spis zawartości dołączonej płyty CD:

- Praca dyplomowa w formacie *.pdf
- Dokumentacja kodów źródłowych
- Aplikacja systemu nawigacji
- Modele zaprojektowanych środowisk gry