Gdańsk, 2021r.

Politechnika Gdańska
Wydział Elektroniki, Telekomunikacji, Informatyki

Sztuczna Inteligencja
Sprawozdanie z projektu
Temat: Sztuczna Inteligencja do *Dyna Blaster*

Wykonali:
Wiktor Krasiński 179987
Jakub Nowak 180503
Przemysław Rośleń 180150

1.Wstęp

1.1. *Dyna Blaster*

Dyna Blaster znany również jako *Boomberman,* jest dość starą grą , mającą swoje początki w latach 80. Tytuł *Dyna Blaster* był używany przez jakiś czas w Europie i jest to pewna wersja serii gier *Bomberman*, na której będziemy opierać nasz projekt. *Dyna Blaster* był dostępny na system operacyjny DOS. Jest to strategiczna gra wideo, oparta na labiryncie.

1.2. Zasady gry *Dyna Blaster*

Aby móc napisać działającą sztuczną inteligencję do gry *Dyna Blaster*, trzeba przede wszytkim poznać zasady działania tej gry. Rozgrywka polega na strategicznym umieszczaniu bomb, które po pewnym czasie eksplodują w wielu kierunkach, aby niszczyć przeszkody i zabijać wrogów. W każdym poziomie gracz ma z góry określony czas na wygranie, pewną liczbę żyć i tablice z punktami. Za każdego przeciwnika i przeszkodę gracz dostaje punkty. Gra kończy się sukcesem dla gracza wtedy, kiedy na planszy pozostanie tylko on, a każdy przeciwnik zostanie zniszczony. Wtedy gracz musi znaleźć portal, który jest ukryty pod jedną z przeszkód. Wchodząc w portal gracz wygrywa poziom. Dodatkowo w przypadku wysadzenia portalu bombą lub gdy minie czas na przejście poziomu, wylatuje kilka dodatkowych przeciwników w formie małych diabełków. Gracz zostaje zabity, jeśli dotknie wroga lub zostanie wciągnięty w eksplozję bomby, następnie odradza się na spawnie, gdy posiada jeszcze życia. W przeciwnym przypadku przegrywa i musi zaczynać poziom od nowa.

1.3. Opis problemu (wygrana gry)

Naszym zadaniem jest napisanie bota, który skutecznie będzie w stanie zniszczyć każdą przeszkodę i zabić każdego przeciwnika, nie wchodząc przy tym pole rażenia eksplozji i przeciwne stworki.

1.4.Podejście do problemu

Uznaliśmy, że niezbędny będzie system pathfindingu, który pomoże agentowi obierać odpowiednią drogę do wyznaczonego przez niego celu. Na początku zdecydowaliśmy się na jeden algorytm, lecz później rozszerzyliśmy program o dwa dodatkowe, by zobaczyć i porównać zachowanie danych algorytmów. Algorytmy, które wybraliśmy to, po kolei:

A*, BFS, DFS

2.Teoria z SI

2.1.Algorytm A*

2.1.1 Wstęp

A* jest heurystycznym algorytmem, który służy do znajdowania najkrótszej ścieżki w grafie ważonym. Algorytm A* jest zupełny i optymalny, czyli znajduje ścieżkę, jeśli taka istnieje i jest to najkrótsza droga, tak długo jak funkcja heurystyki jest odpowiednio wyznaczona. Algorytm A* używa się do takich problemów jak routing problems, poruszanie się botów w grach wideo, przechodzenie labiryntów, rozwiązywanie łamigłówek.

2.1.2 Działanie

Algorytm korzysta z minimalizacji funkcji f(n) w celu wybrania najbardziej odpowiednich wierzchołków do wyznaczenia drogi.
Funkcja f(n), gdzie n to wierzchołek, wygląda następująco:

$f(n) = g(n) + h(n)$

Funkcja g(n) wyznacza koszt rzeczywisty dojścia ze startu do wierzchołka n, dokładniej jest to suma wag krawędzi w ścieżce + waga krawędzi wierzchołka n z ostatnim wierzchołkiem ścieżki.

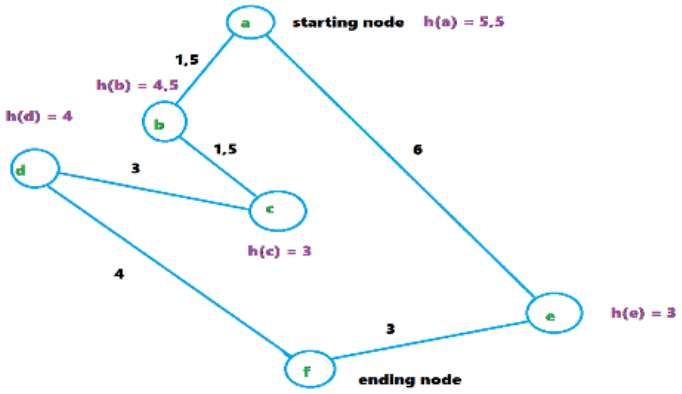
Funkcja h(n) wyznacza estymowany koszt przez określoną heurystykę, z wierzchołka n do wierzchołka końca drogi.

Warto dodać, że algorytm będzie znajdywał drogę najkrótszą wtedy i tylko wtedy, gdy koszt zdefiniowanej heurystyki będzie mniejszy lub równy od rzeczywistej drogi do rozwiązania.
Często wyborem określania wartości heurystyki, jest odległość w linii prostej od wierzchołka n do wierzchołka docelowego.
W przypadku gdy h(n) będzie wynosiło 0, algorytm staje się algorytmem Dijkstry.
Złożoność algorytmu jest wykładnicza.

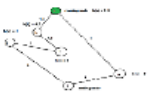
2.1.3 Algorytm

- Stwórz zbiór Open do którego będą ładowane wierzchołki nieodwiedzone, sąsiadujących z odwiedzonymi i zbiór Closed gdzie będą ładowane wierzchołki odwiedzone. Następnie dodaj do zbioru Open początkowy wierzchołek i ustaw wartość g(s) = 0, gdzie s to początkowy wierzchołek.
- Dopóki zbiór Open pozostaje niepusty powtarzaj:
    - Pobierz (i usuń) jeden wierzchołek v ze zbioru Open (jest to wierzchołek najlepszy w sensie wartości f).
    - Jeżeli v jest wierzchołkiem docelowym, to przerwij cały algorytm i zwróć v jako wynik
    - Wygeneruj wszystkie wierzchołki sąsiadujące(prócz rodzica) dla v i dla każdego z nich wykonaj:
        - Jeżeli wierzchołek znajduje się w Closed, to nie badaj go dalej.
        - Policz: g(w) = g(v) + distance(v, w) i f (w) = g(w) + h(w).
        - Jeżeli wierzchołek nie występuje w Open, to dodaj go ustawiając v jako jego rodzic.
        - Jeżeli s wierzchołek występuje w Open, to podmień w nim wartości g i f (oraz poprzednika), o ile są korzystniejsze
    - Włóż v do zbioru Closed

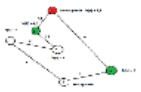
2.1.4. Przykład:



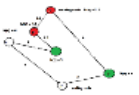
1. f(a) = g(a) + h(a) = 0 + 5,5 = 5,5



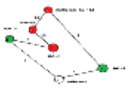
2. f(b) = g(b) + h(b) = 1,5 + 4,5 = 6
   f(e) = g(e) + h(e) = 6 + 3 = 9
   f(b) < f(e)



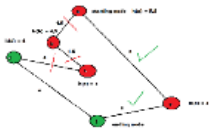
3. f(c) = g(c) + h(c) = 3 + 3 = 6
   f(e) = g(e) + h(e) = 6 + 3 = 9
   f(c) < f(e)



4. f(d) = g(d) + h(d) = 6 + 4 = 10
   f(e) = g(e) + h(e) = 6 + 3 = 9
   f(e) < f(d)



5. f(f) = g(f) + 0 = 9



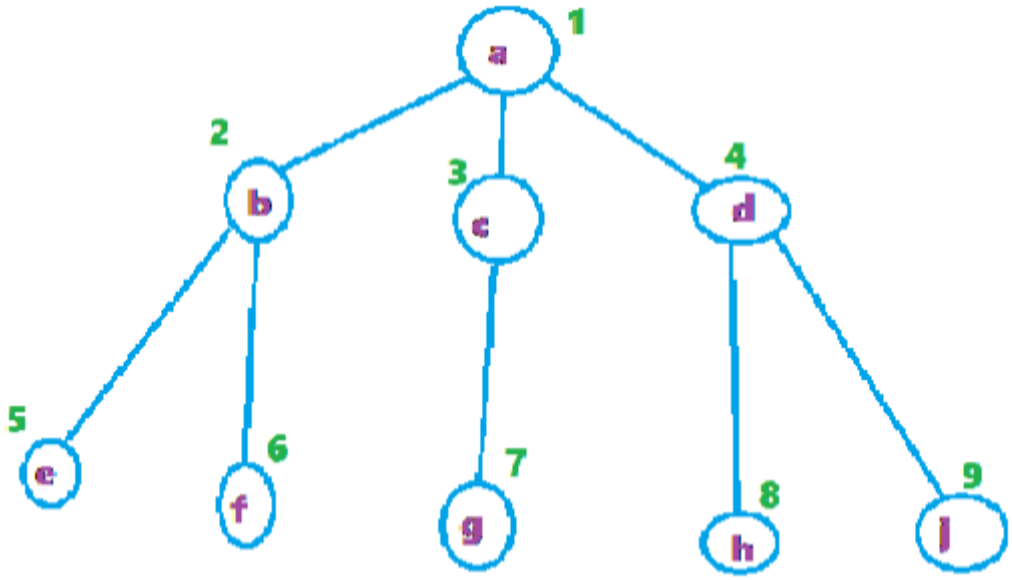
2.2.BFS

2.2.1 Wstęp

BFS czyli przeszukiwanie wszerz, to jeden z prostszych algorytmów do przeszukiwania grafu. Polega na odwiedzeniu wszystkich wierzchołków w grafie. W wyniku działania algorytmu powstaje drzewo o korzeniu startowego wierzchołka przeszukiwania wszerz. Dzięki temu do każdego wierzchołka z korzenia prowadzi jedna ścieżka, tworząc najkrótszą droge z wierzchołka wejściowego do dowolnego wyszukiwanego. Dzięki takiej właściwości algorytmu jest on odpowiedni do użycia go do pathfindingu. Algorytm jest zupełny, czyli znajdzie najkrótszą drogę, jeżeli takowa istnieje.
Algorytm działa na grafach skierowanych i nieskierowanych

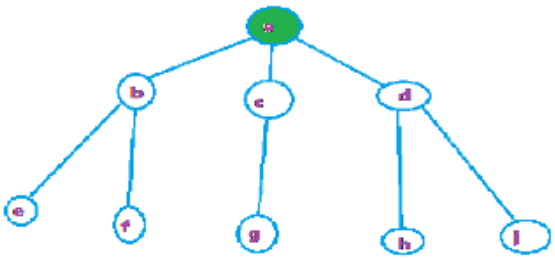
2.2.2 Działanie

Działanie BFS polega na przechodzeniu i odwiedzaniu najpierw każdego sąsiada korzenia, a następnie sąsiadów tych sąsiadów. Działanie całego algorytmu i przeszukiwanie można zobrazować na prostym rysunku:

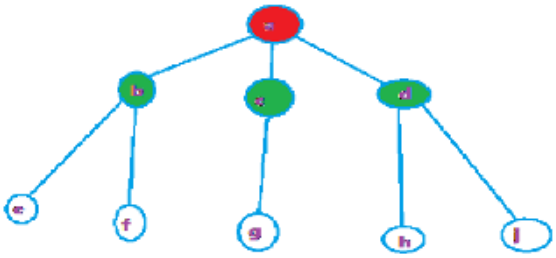


Dokładna kolejność odwiedzania wierzchołków została przedstawiona na tym rysunku:

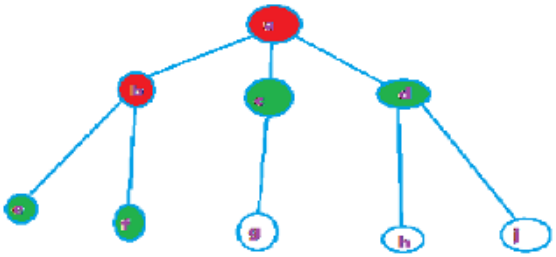
1.



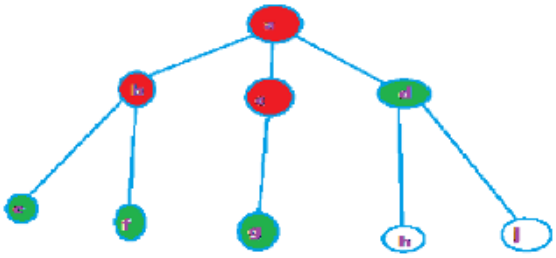
2.



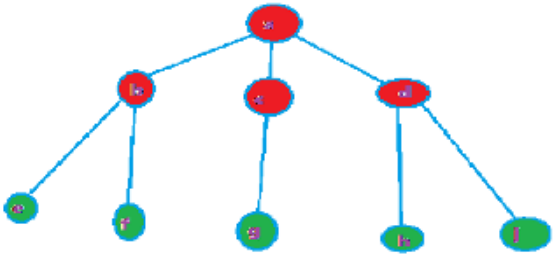
3.



4.



5.



Tutaj trochę bardziej szczegółowo został przedstawiony przebieg przeszukiwania wszerz.
Gdy wierzchołek jest odwiedzony zostaje naznaczony na czerwono, a gdy jest zielony oznacza to, że jest jeszcze do odwiedzenia. W momencie zaznaczania wierzchołka na czerwono, jego dzieci stają się zielone. Odwiedzamy wierzchołki wszerz, ponieważ każdy nowy zielony wierzchołek zostaje dodany na koniec listy FIFO i w ten sposób algorytm będzie odwiedzał najpierw najstarsze dodane zielone wierzchołki. Dzięki temu kolejność przeszukiwania odbywa się wszerz.
Złożoność obliczeniowa BFS'a wynosi v+e gdzie v to liczba wierzchołków, a e to liczba krawędzi.

2.2.3 Algorytm

- Dodaj do kolejki FIFO wierzchołek startowy i oznacz jako odwiedzony.
- Dopóki kolejka pozostaje niepusta powtarzaj:
  - Pobierz (i usuń) jeden wierzchołek v z przodu kolejki
  - Wygeneruj wszystkie wierzchołki sąsiadujące(prócz rodzica) dla v i dla każdego z nich wykonaj:
    - Jeżeli wierzchołek był odwiedzony, to nie badaj go dalej.
    - Jeżeli jednak nie był odwiedzony to go dodaj do kolejki .
  - Oznacz v jako odwiedzony

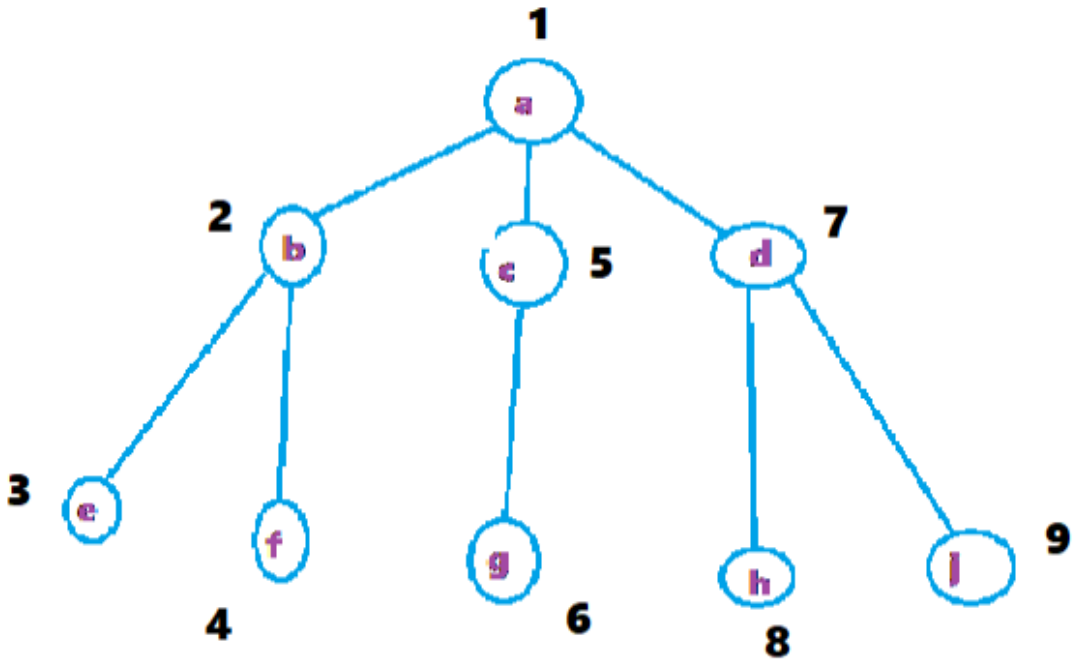
2.3.DFS

2.3.1 Wstęp

DFS czyli przeszukiwanie w głąb, to jeden z prostszych algorytmów do przeszukiwania grafu. Polega na badaniu wszystkich krawędzi z danego wierzchołka. Przy odwiedzeniu wszystkich wierzchołków wychodzących z krawędzi dany wierzchołek zostaje oznaczony jako odwiedzony. DFS służy do rzeczy takich jak szukanie drogi, generowanie labiryntów.

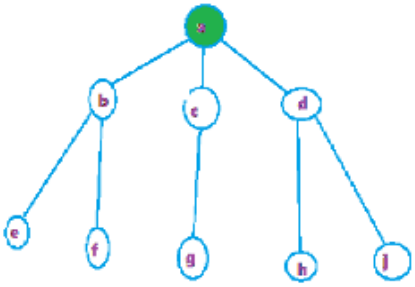
2.3.2 Działanie

DFS działa podobnie jak BFS, lecz zamiast przeszukiwać wszerz, zagłębia się w początkowy wierzchołek aż dany wierzchołek nie będzie posiadał krawędzi. Kolejność odwiedzania wierzchołków jest przedstawiona na tym rysunku:

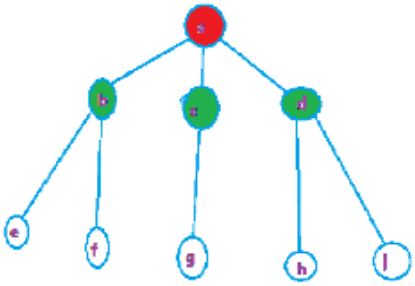


Tutaj w pięciu krokach zostało zobrazowane funkcjonowanie jak działa przeszukiwanie grafu za pomocą DFS’a:

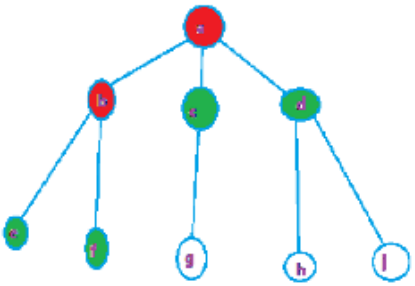
1.



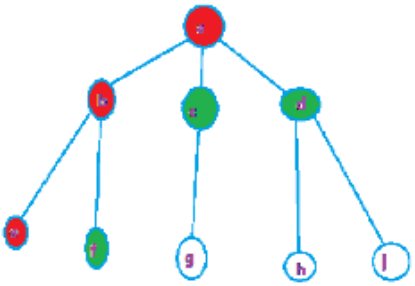
2.



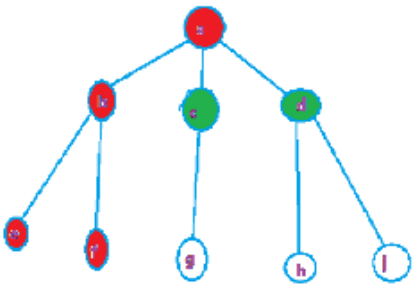
3.



4.



5.



2.3.3 Algorytm

- Oznacz korzeń jako odwiedzony.
- Wygeneruj wszystkie wierzchołki sąsiadujące(prócz rodzica) dla v i dla każdego z nich wykonaj:
  - Jeżeli wierzchołek sąsiadujący w jest nieodwiedzony oznacz na odwiedzony
  - Jeżeli wierzchołek sąsiadujący nie ma sąsiadów nie rób nic, w przeciwnym razie wywołaj tą funkcje rekursywnie dla wierzchołka sąsiadującego.

3. Opis realizacji zadania

3.1. Technologia

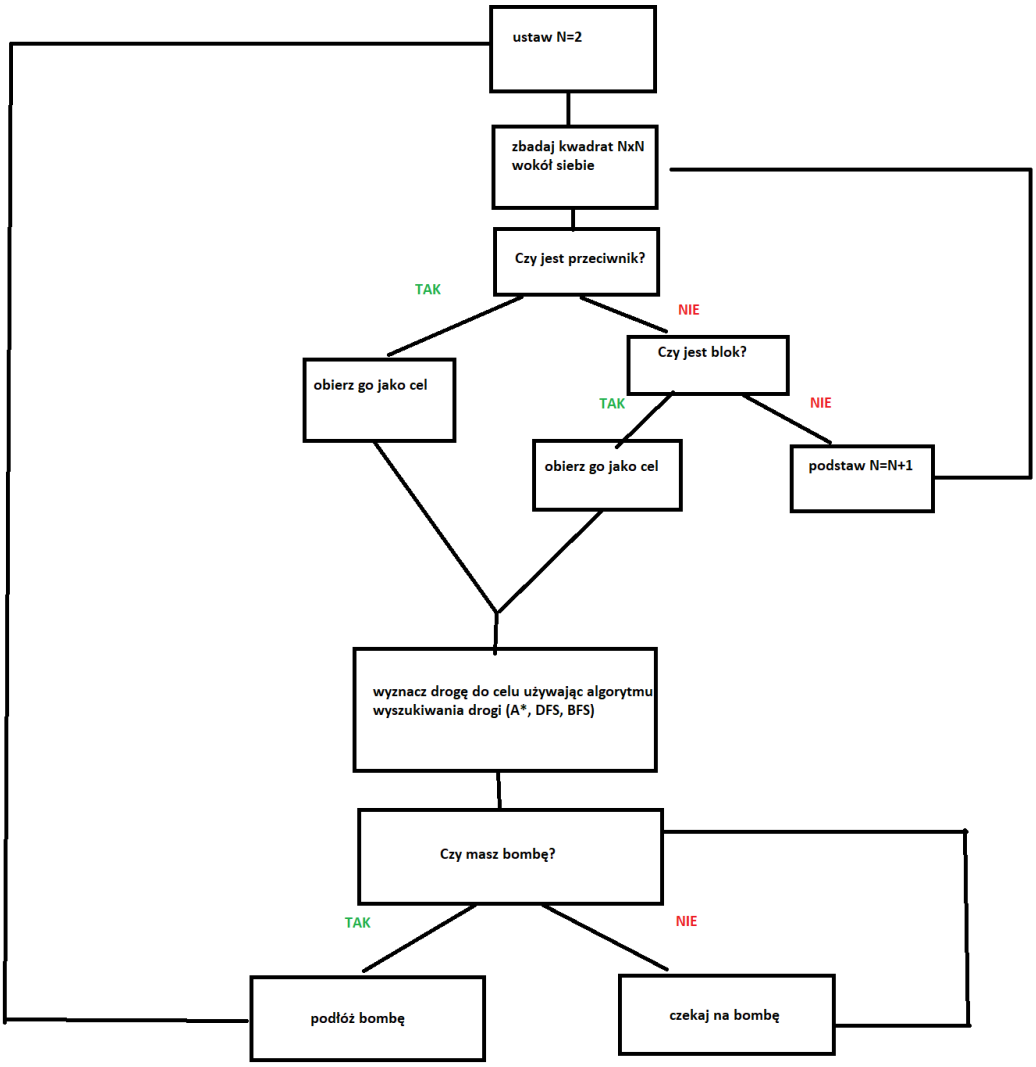
W celu implementacji naszego projektu, postanowiliśmy wykorzystać skryptowy język programowania Python w wersji 3. Do wizualizacji graficznej zdecydowaliśmy się skorzystać z biblioteki *PyGame*. Środowisko programistyczne oraz uruchomieniowe (IDE), które użyliśmy, to *PyCharm Community Edition*.

3.2. Założenia implementacyjne

W jednym z wcześniejszych podrozdziałów został zaprezentowany podstawowy koncept gry Dyna Blaster. Nie jest on jednak dostatecznie precyzyjny, żeby bezpośrednio przejść do implementacji w pełni funkcjonalnej gry. Należało powziąć pewne założenia, aby dało się to zrealizować.
Gracz, posiadający jedno życie, w swoim ekwipunku posiada tylko jedną bombę, która powraca do jego plecaka w momencie eksplozji wcześniejszej bomby. Rywalami agenta są stwory, w liczbie trzech, które poruszają się w linii prostej. W przypadku rozwidlenia lub napotkania ściany, przeciwnik posiada pewną mało prawdopodobną szansą na zmianę kierunku. Jeśli agent wejdzie w stwora albo w momencie wybuchu znajdzie się w promieniu eksplozji, traci swoje życie. Prędkość ruchu gracza jest pięciokrotnie większa od prędkości poruszania się stworów, żeby mógł je efektywnie gonić. Rozgrywka kończy się, gdy zostaną zniszczone wszystkie bloki i pokonani wszyscy przeciwnicy (przypadek zwycięski) albo gdy gracz straci życie (przypadek przegranej). W celu uproszczenia implementacji, nie zdecydowaliśmy się na super-moce ani na portal.

3.3. Schemat działania agenta

Nasz agent próbujący zwyciężyć poziom, opiera się na poniższym schemacie decyzyjnym:



3.4 Unikanie śmierci

Mówiąc o sztucznej inteligencji, mamy na myśli mechanizm/istotę nierzeczywistą, która zachowuje się rozumnie. Dlatego nasz wykreowany agent powinien inteligentnie unikać śmierci. Powinien on usilnie dążyć do osiągnięcia celu, jednak stosując podejście „bezpieczeństwo najważniejsze”. Dlatego w przypadku bezpośredniego zagrożenia utraty życia agent podejmuje odpowiednie kroki, by przeżyć.
Po podłożeniu bomby, gracz oznacza wszystkie pola rażenia, jako „niebezpieczne”. W praktyce oznacza to, że może przechodzić przez te pola, do momentu, w którym nie pozostanie bardzo niewiele czasu do wybuchu. Wtedy natomiast, pod żadnym pozorem nie

wejdzie na te pola rażenia.
Kolejnym czynnikiem samozachowawczym jaki podejmuje agent jest niepodchodzenie do wrogów bliżej na 2 pola od niego.

3.5 Zabijanie stworków

Do wygrania poziomu konieczny jest efektywny sposób zabijania stworów. Dlatego agent sprawdza kierunek, w którym porusza się przeciwnik i kieruje się na pole, które jest oddalone o 2 pola od stwora i do którego ten stwór aktualnie zmierza. To sprawia, że w wielu przypadkach udaje mu się skutecznie przeprowadzić atak.

3.4. Algorytmy wyszukiwania ścieżki

3.4.1. Informacje wstępne

Mimo tego, że zaimplementowane przez nas algorytmy różnią się pewnymi aspektami, to jednak możemy wyróżnić pewne cechy wspólne w tym projekcie. Każdy z nich, na początku, wyznacza punkt startowy, którym zawsze jest obecna pozycja agenta, oraz punkt końcowy, za którego znalezienie odpowiada funkcja *determineTarget()*. W ciele tej funkcji wyznaczany jest kwadrat N x N pól wokół agenta (przy startowym N=2). Jeśli znaleziony zostanie przeciwnik w tym obszarze, to zostanie on obrany za nasz cel. W przeciwnym razie, obrany zostanie najbliższy blok znajdujący się w kwadracie. Gdy jednak ten warunek również nie zostanie spełniony, to procedura jest powtarzana z argumentem N+1. Uproszczony schemat znajduje się na rysunku powyżej.
Niezwykle istotną cechą wspólną wszystkich wykorzystanych algorytmów jest fakt, że po każdym „kroczku” agenta, wyznaczają one optymalną trasę ponownie. Takie działanie to konieczność, ponieważ na życie agenta nieustannie czyhają stworki, które w pewnych momentach poruszają się losowo i przy bezmyślnym przechodzeniu przez raz wyznaczoną drogę, mogłoby się okazać, że na tej ścieżce pojawiłby się w pewnym momencie przeciwnik, który zjadłby naszego gracza. Taki rezultat nie byłby satysfakcjonujący.
W celu efektywnego korzystania z algorytmów, utworzyliśmy tablicę obiektów węzłowych – każdy z właściwościami takimi jak: wspX, wspY, rodzic, itd.
Po znalezieniu celu, wyznaczana jest ścieżka przy użyciu propagacji wstecznej – czyli począwszy od węzła końcowego idziemy po rodzicach aż do początku i na końcu odwracamy tablicę.

3.4.2. Algorytm A*

Implementacja algorytmu A*, przystosowana do naszego projektu:

```
def Astar(self, grid, enemys, targetNode = None, destinationNode = None):
    openList = []
    closed = []
    startNode = grid[int(self.posX / 4)][int(self.posY / 4)]
    openList.append(startNode)
    if destinationNode is None or targetNode is None:
        targetNode, destinationNode = self.determineTarget(grid, enemys)
    found = False
    while True:
        if len(openList) == 0:
            return False
        current = self.findSmallestFx(openList)
        openList.remove(current)
        closed.append(current)

        if current == destinationNode:
            found = True
            break

        neighbours = self.getNeighbours(grid, current)

        for neighbour in neighbours:
            if neighbour in closed:
                continue

            if neighbour.parent is None:
                neighbour.parent = current
            newGx = self.findGx(neighbour, startNode)

            if newGx < neighbour.gx or neighbour not in openList:
                neighbour.gx = newGx
                neighbour.hx = self.findHx(neighbour, targetNode)
                neighbour.fx = neighbour.gx + neighbour.hx
                neighbour.parent = current
                if neighbour not in openList:
                    openList.append(neighbour)
```

Legenda:

- *openList* – tablica węzłów branych pod uwagę do odwiedzenia w przyszłości
- *closed* – tablica odwiedzonych węzłów
- *grid* – tablica wszystkich węzłów na mapie
- *targetNode* – węzeł będący naszym celem

- *destinationNode* – węzeł będący destynacją (sąsiad węzła celu)
- *current* – aktualnie rozpatrywany węzeł
- *findSmallestFx(tab)* – metoda wyszukująca węzeł z najmniejszą wartością funkcji f(x), czyli najmniejszą sumą g(x) + h(x)
- *neighbours* – tablica węzłów sąsiadujących z obecnym, na które może wejść agent
- *node.gx* – wartość funkcji g(x), czyli kosztu dojścia do danego węzła od startu
- *node.hx* – wartość funkcji h(x), czyli heurystyczny koszt dojścia do końca od danego węzła
- *node.parent* – rodzic danego węzła
- *node.fx* – ogólna wartość danego węzła (g(x) + h(x))

3.4.3. Algorytm Breadth-first search (BFS)

Nasza wersja algorytmu BFS do znalezienia ścieżki do celu:

```
def BFS(self, grid, enemys, targetNode = None, destinationNode = None):
    found = False
    open = list()
    visited = list()
    rootNode = grid[int(self.posX / 4)][int(self.posY / 4)]
    open.append(rootNode)
    if destinationNode is None or targetNode is None:
        targetNode, destinationNode = self.determineTarget(grid, enemys)
    while len(open) > 0:
        currentNode = open.pop(0)
        visited.append(currentNode)
        if currentNode == destinationNode:
            found = True
            break
        for neighbour in self.getNeighbours(grid, currentNode):
            if neighbour not in visited and neighbour not in open:
                open.append(neighbour)
                neighbour.parent = currentNode
```

Legenda:

- *visited* – tablica odwiedzonych węzłów
- *rootNode* – węzeł startowy

Reszta zmiennych jest analogiczna do algorytmu A*.

3.4.4. Algorytm Depth-first search (DFS)

Implementacja algorytmu DFS w naszej grze:

```
def DFS(self, grid, enemys, targetNode = None, destinationNode = None):
    found = False
    open = list()
    visited = list()
    rootNode = grid[int(self.posX / 4)][int(self.posY / 4)]
    open.append(rootNode)
    if destinationNode is None or targetNode is None:
        targetNode, destinationNode = self.determineTarget(grid, enemys)
    while len(open) > 0:
        currentNode = open.pop()
        visited.append(currentNode)
        if currentNode == destinationNode:
            found = True
            break

        for neighbour in self.getNeighbours(grid, currentNode):
            if neighbour not in visited and neighbour not in open:
                open.append(neighbour)
                neighbour.parent = currentNode
```

Wszystkie zmienne pełnią analogiczne role jak w algorytmie BFS.

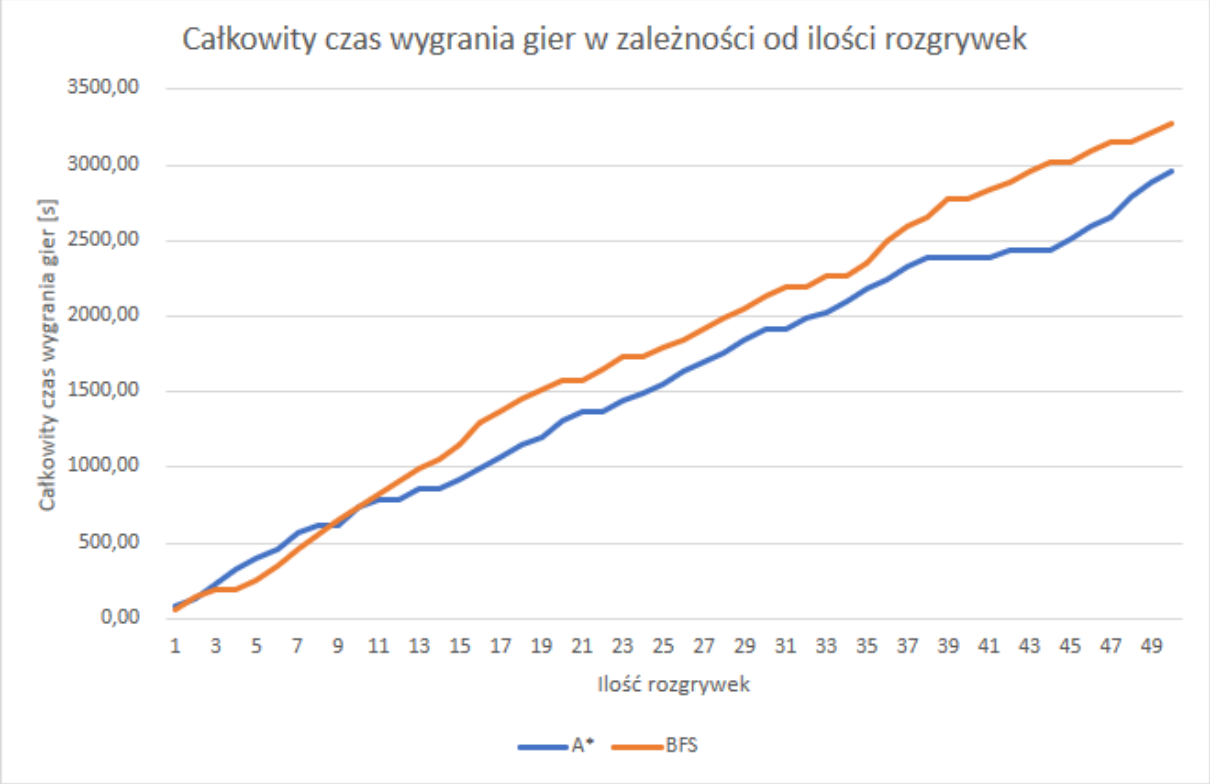
4. Porównanie algorytmów

4.1. Opis eksperymentu

W celu porównania skuteczności działania algorytmów rozwiązujących problem *Dyna Blaster*, wykonaliśmy po 50 prób wygrania etapu dla każdej metody. Pod uwagę braliśmy procent wygranych rozgrywek oraz czas, w którym dany agent wygrał grę. W przypadku przegranej czas nie został uwzględniany w podliczaniu średniego czasu ukończenia. Agenci w każdej iteracji grali na identycznych mapach, ale każda z 50 map była inna, losowa.

4.1. Wyniki

	wygrane	średni czas [s]
A*	80% (40/50)	73.89
BFS	84% (42/50)	77.98
Gracz	100% (50/50)	63.69



Algorytm Depth-first search (DFS) okazał się nieprzystosowany do naszego podejścia, czyli wyznaczania nowej ścieżki po każdym małym „kroczku” agenta. Algorytm ten w wielu przypadkach zapętlał się, uniemożliwiając dalszą grę, a co za tym idzie, testy również nie mogły zostać ukończone z jakkolwiek satysfakcjonującym wynikiem.

4.2. Analiza wyników

Z powyższej tabeli, na pierwszy rzut oka, wydaje się, że algorytm BFS działa bardziej zachowawczo – ginie w mniejszej liczbie przypadków, ale jednocześnie wygranie poziomu zajmuje mu średnio więcej czasu. Jednakże różnice te są na tyle małe (rzędu kilku procent), że nie są one miarodajne. Co więcej, w naszej implementacji gry, wyniki mogą być wypaczane przez pewną losowość wynikającą z niedeterministycznego zachowania w każdej jednostce czasu przez stwory znajdujące się na mapie. Z tego względu, liczba testów powinna być zdecydowanie większa, np. 1000 gier na każdy algorytm, żebyśmy mogli wyciągać konkretne wnioski, które odzwierciedlałyby faktyczny stan rzeczy. Dodatkowo, mapa użyta w projekcie jest na tyle mała, że różnica w czasie wyznaczenia ścieżki jest bardzo mała, niemalże niezauważalna.

5. Podsumowanie

W całym projekcie wykonaliśmy:

- modyfikację środowiska już w części zaimplementowanego
    - dodanie przeciwników
    - dodanie własnych grafik (dzięki uprzejmości Oldmana)
    - dodanie menu końcowego
- implementację algorytmów: A*, BFS, DFS
- implementację drzewa zachowań agenta w różnych stanach, aby działał inteligentnie
- testy algorytmów i gracza
- sprawozdanie