POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka
SPECJALNOŚĆ: Systemy informatyczne w automatyce

PRACA DYPLOMOWA INŻYNIERSKA

Zastosowanie wybranych algorytmów sztucznej inteligencji w grach komputerowych

Usage of chosen artificial intelligence algorithms in computer games

AUTOR:
Paweł Rynowiecki

PROWADZĄCY PRACĘ:
dr inż., Łukasz Jeleń, jednostka

OCENA PRACY:

WROCŁAW …….

Spis treści

1. Wstęp

Celem pracy dyplomowej było stworzenie gry komputerowej, w której przeciwnicy korzystaliby z algorytmów sztucznej inteligencji. Wybrana została gra typu "Snake". Gracz kontroluje wirtualnego węża i ma na celu zdobycie jak największej ilości pożywienia. Utrudnieniem są przeszkody, z którymi kolizja kończy się śmiercią węża. Przykłady przeszkód: ściany naokoło mapy, ogony przeciwnych węży i jego własny. Pożywienie pojawia się jedno na raz, w losowym miejscu na mapie, które jest aktualnie puste. Gdy jeden z węży trafi na pole z jedzeniem zwiększa długość swojego ogona. Węże mogą poruszać się w tylko wertykalnie oraz horyzontalnie; kierunek poruszania można zmieniać o kąt 90 stopni. Gra wymusza ruch w zadanym kierunku co pewien okres czasu, o jedno pole mapy na raz. Wygrywa najdłuższy wąż.

W założeniu gracz miał współzawodniczyć z 3 przeciwnikami, korzystających z następujących algorytmów: A*, drzewo behawioralne oraz wielowarstwowy perceptron. Projekt udało się w większości zrealizować.

Do wykonania zadania użyto silnika graficznego Unity. Umożliwia on pisanie skryptów w języku C# a następnie przypisywania ich do poszczególnych obiektów gry. Projekt jest ciągle rozwijany, posiada obszerną i ciąglę aktualizowaną dokumentację. Unity jest dostępne warunkowo za darmo, z maksymalnym progiem zarobku z wykorzystaniem ich silnika.

Do implementacji algorytmu sieci neuronowych posłużono się platformą .Net core oraz językiem C#. Ten wybór miał na celu ułatwienie przenoszenia kodu pomiędzy dwoma aplikacjami. Platforma dotnet posługuje się licencją MIT. Oprogramowanie na tej licencji zezwala na kopiowanie, modyfikacje, wykorzystanie prywatne i komercyjne. Twórca natomiast nie bierze za nic odpowiedzialności, ani nie daje żadnej gwarancji na to oprogramowanie.

The main goal of this paper is to describe the work done with the project which focuses on the AI implementation in the real game. The chosen game is “Snake”. The player controls the virtual snake which needs to eat food and also needs to avoid some obstacles, such as walls and other snakes tails as well as its own. Food is spawn once per map in random places. When the snake goes through a field with the food, it increases its size by one. Snakes can only move vertically and horizontally and can change direction of movement only by 90 degrees. The game forces the movement one field at a time, per a given period of time. Game is won by the longest snake.

The following artificial intelligence algorithms were implemented in the project as follows: A* pathfinding algorithm, behavioural tree and artificial neural network algorithm, which unfortunately was not finished.

To complete this task, Unity Game Engine was used. It allows one to write scripts in C# programming language, which can be assigned to particular game objects located in the scene. Unity project is still being developed. It has a wide and proper documentation. Unity allows one to use their engine for free depending on the annual income of the product it was made in.

To implement neural network algorithms, .Net core programming platform was used. C# language was used alongside. This helped greatly with code portability. .Net Core platform uses MIT licence which means that you can copy this software, modify it, use it for commercial purposes, but an author doesn’t take any responsibility for use of the software.

2. Opis użytych technologii

2.1. Unity

Unity jest silnikiem gry. Jest to takie oprogramowanie, które udostępnia szereg potrzebnych funkcjonalności twórcą gier do realizacji swoich projektów w sposób szybki oraz wydajny. Żeby spełnić te warunki musi on wspierać możliwość importowania materiałów 2D oraz 3D stworzonych za pomocą innych popularnych programów, takich jak np. Photoshop czy Maya. Silnik powinien pozwalać również na: tworzenie efektów specjalnych, symulować zjawiska fizyczne, a w szczególności kinematykę, pozwalać na wykonanie animacji oraz dać możliwość zarządzania ścieżką dźwiękową czy np. oświetleniem. Dodatkowo, ważnym aspektem silnika gry jest to by gra wyprodukowana za jego pomocą była możliwie najlepiej zoptymalizowana pod platformę na którą została stworzona.

Unity w wersji 2017.2 daje użytkownikowi do wyboru dwa języki programowania, do implementacji logiki gry: C# oraz Javascript. W pracy korzystano z języka C# i to na nim skupi się reszta podrozdziału.

Język C# został stworzony przez firmę Microsoft jako jedno z narzędzi obsługi platformy .Net, która stała się naturalnym konkurentem maszyny wirtualnej Javy. Jest to język obiektowy, projektowany w latach 1998 - 2001. Napisany program jest kompilowany do kodu pośredniego następnie wykonywanego w środowisku uruchomieniowym takim jak .Net Framework albo Mono. Ważną cechą tego języka jest to, że posiada on odśmiecanie pamięci czyli automatyczną dealokację pamięci stosu.

Unity wspiera platformę .Net w wersji 3.5 a kod uruchamiany jest w środowisku Mono. Wirtualna maszyna, niestety nie jest tak wydajna jak kod natywny, dlatego sam silnik jest napisany w C/C++ a API silnika w C# tylko opakowuje te biblioteki.

Podczas tworzenia gry w Unity można rozróżnić pare kluczowych koncepcji. Koncepcja Teatru (Stage) i Scen (Scenes) służą do prezentacji tego co się dzieje w grze. Teatr jest swoistą przestrzenią świata Unity, w której mogą się odbywać różne Sceny. Każda scena domyślnie zaczyna z kamerą główną (Main Camera), która pozwala na rejestracje i przedstawienie użytkownikowi tego co aktualnie widzi. Zwykle jedna scena prezentuje jeden widok/poziom gry. Do sceny można dodawać inne obiekty, które można podzielić na elementy świata (World Objects) oraz elementy interfejsu użytkownika. Tworzą one hierarchię drzewiastą, którą można swobodnie zarządzać poprzez UI programu. Podczas przeładowania sceny wszystko co należało do poprzedniej zostaje zniszczone. Dodatkowo wspierane są Prefabrykaty (Prefabs). Są to pliki, w których Unity potrafi zapisywać informacje o obiekcie ze sceny. Dzięki temu w łatwy sposób można duplikować obiekty. Najważniejszą cechą Prefabrykatów jest jednak to, że elementy w scenie, dodane za ich pomocą są powiązane. Zatem można wykonać modyfikację Prefabrykatu i zostanie ona zastosowana do wszystkich elementów zależnych.

2.2. .Net Core

.Net Core jest platformą programistyczną typu open source, dostępną na popularnych platformach takich jak Windows, Linux i macOS. Z początku miała to być odchudzona wersją .Neta, którą można łatwo uruchamiać na wielu systemach operacyjnych. Z powodu jej niespodziewanej popularności Microsoft zdecydował się, że nie będzie dłużej rozwijał klasycznego .Neta.

.Net Core posiada szeroki zestaw narzędzi CLI, z których pomocą z łatwością można stworzyć, uruchomić, przetestować projekt. Aktualnie (wersja 2.1) ma wsparcie tylko dla dwóch języków programowania. Jest to C# i F#. W przyszłości ma się pojawić wsparcie dla języka Visual Basic.

Technologia została wybrana w celu, możliwości szybkiej wymiany kodu pomiędzy skryptami w Unity a programem uczącym sieć neuronową.

2.3. Git

Git jest systemem kontroli wersji, stworzonej podczas tworzenia systemu operacyjnego Linux. Wykorzystuje on drzewiasty model zmian.

Wykorzystano Gita w wersji 2.7.4 build na Ubunut oraz serwisu github.com, który umożliwia za darmo przetrzymywanie zdalnej wersji repozytorium.

3. Struktura aplikacji

3.1. Moduły i zależności między modułowe

- *Map* i *Field*:

Map jest obiektem statycznym, który przetrzymuje informację o dostępnych kierunkach i typach pól (*enum*) oraz dwuwymiarową tablicę pól planszy, na której odbywa się cała rozgrywka. Dodatkowo posiada różne metody pozwalające na łatwiejsze wyciąganie informacji o położeniu np. sąsiednich pól. *Field* jest typem obiektów w tablicy znajdującej się w obiekcie *Map*. Daje możliwość transformowania pozycji mapy na pozycję w scenie gry.

Klasy *Map* i *Field*:

```
public class Field {
    public Map.Fields fieldType;
    public bool canWalk;

    public Vector2 pos; // World Position
    public int x, y;    // Map Position
}

public static class Map {
    public enum Fields { empty, wall, tail, food };
    public enum Side { up, right, down, left };
    public static Field[,] map;

    public static Field Neighbour(Field field, Side side){
        var dir = Direction(side);

        return map[field.x + (int)dir.x, field.y + (int)dir.y];
    }

    public static Vector2 Direction(Side side) {
        switch (side) {
            case Side.right:
                return Vector2.right;
            case Side.down:
                return Vector2.down;
            case Side.left:
                return Vector2.left;
            case Side.up:
                return Vector2.up;
            default:
                return Vector2.zero;
        }
    }
}
```

Vector2 jest typem wbudowanym w bibliotekę *Unity.Engine.* Składa się z dwóch pól typu *float* i pozwala na wykonanie wielu operacji na wektorach.

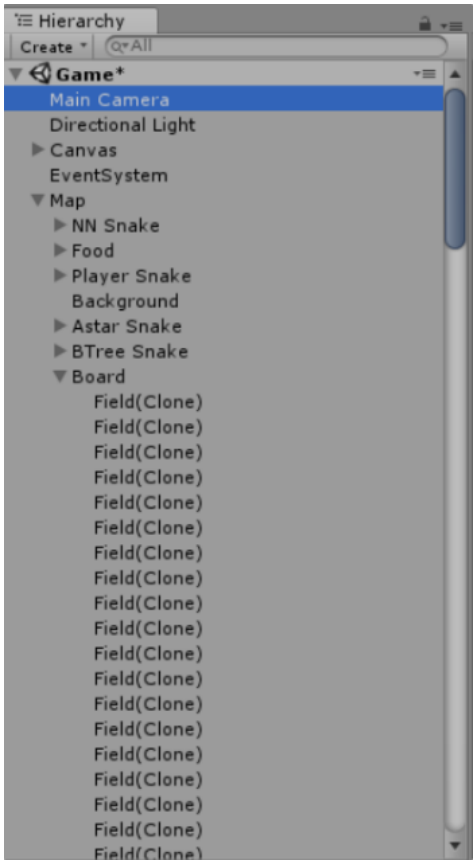
Metoda *Direction* konwertuje typ wyliczeniowy na odpowiadający wektor jednostkowy. Wykorzystywana jest przez metodę *Neighbour*, która potrafi zwrócić sąsiadujące pole na podstawie aktualnego pola i podanej strony.

Dodatkowo został zaimplementowany *MapBuilder*, który przed uruchomieniem rozgrywki tworzy wszystkie potrzebne obiekty w pamięci i wywołuję metody, które graficznie je przedstawią.

Przykład metody w *MapBuilder* do tworzenia ścian w świecie gry:

```
public Transform wall;
public Transform board;

private void CreateWalls() {
    Transform temp;

    foreach (var m in Map.map) {
        if(m.field == Map.Fields.wall) {
            temp = Instantiate(wall, m.pos, Quaternion.identity);
            temp.SetParent(board);
        }
    }
}
```

Każdy obiekt w scenie gry posiada obiekt *Transform*, który służy do manipulacji położeniem, skalą oraz rotacją obiektów. Metoda *Instantiate* pozwala skopiować i umieścić w określonej pozycji na scenie (*pos*), wcześniej przygotowany w formie prefabrykatu, obiekt *wall* z rotacją identyczną do oryginalnego obiektu.Na przechwyconej wartości *Transform* nowo utworzonego obiektu wywołano metodę *SetParent*, która przypisuję go w odpowiednie miejsce w hierarchii obiektów w scenie (rys.1).



Rys.1. Hierarchia obiektów w scenie *Game*.

- *Snake* i *Food*

Obiekt *Food* odpowiada za, losowanie miejsca na mapie, w którym pojawi się jedzenie, natomiast klasa *Snake* kontroluje wszystkie aspekty życia węża. Jego przemieszczanie, konsumpcję pokarmu, wzrost i wiele innych.

Klasa *Food*:

```
public class Food : MonoBehaviour {
    public Transform foodPrefab;

    private Transform food;
    private int x = 0, y = 0;

    void Start() {
        Spawn();
    }

    private void Spawn() {
        food = Instantiate(foodPrefab, FindEmptyField(),
Quaternion.identity);
        food.transform.SetParent(GetComponent<Transform>(), false);
        food.GetComponent<SpriteRenderer>().color = Color.green;
    }

    private Vector2 FindEmptyField() {
        x = 0; y = 0;
        while(!Map.map[x,y].IsEmpty()) {
            x = Random.Range(1, Map.map.GetLength(0));
            y = Random.Range(1, Map.map.GetLength(1));
        }
        Map.map[x,y].ChangeField(Map.Fields.food);

        return Map.map[x,y].pos;
    }
}
```

MonoBehaviour to podstawowy obiekt z którego dziedziczą wszystkie obiekty w scenie. Definiuje on np. w jaki sposób ma się zainicjalizować obiekt w scenie udostępniając metodę *Start* w swoim interfejsie, która wykonuję się raz na początku załadowania sceny. Jeżeli w hierarchii obiektów znajduje się wiele tego typu metod to wywoływane są od góry do dołu listy zależności (rys.1).

W *Start* wywoływana jest metoda *Spawn*, która ma za zadnie utworzyć obiekt jedzenia w scenie. Ona natomiast zwraca się do *FindEmptyField*, która losuję tak długo współrzędne na mapie, aż znajdzie wolne pole i zwraca współrzędne tego punktu w świecie gry.

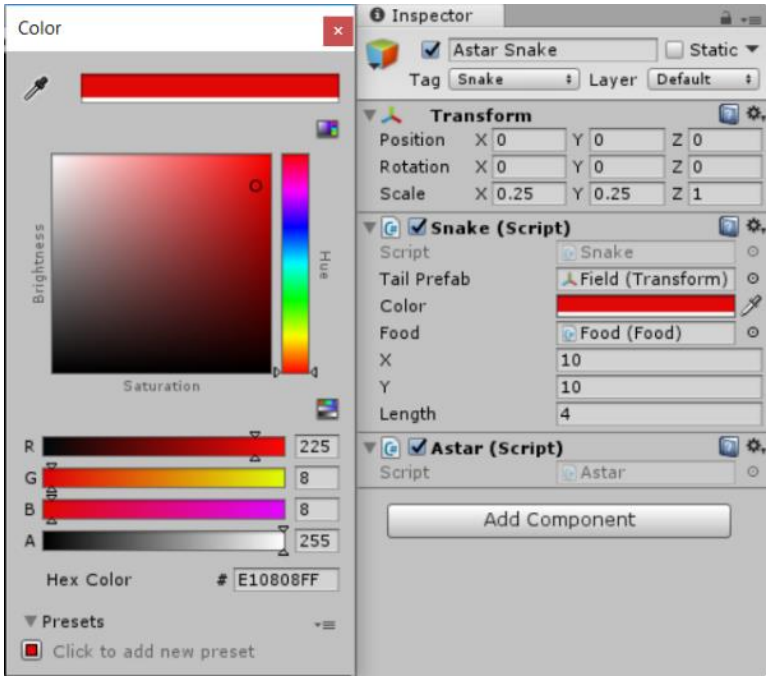
Klasa *Food* posiada dwie zmienne typu *Transform*, w *foodPrefab* trzymany jest prefabrykat obiektu do umieszczenia w scenie. Druga służy w celach dostępowych do klona w scenie, by móc go w dowolnej chwili zmodyfikować tak jak to się dzieję w poniższej metodzie.

```
public void Eat(){
    Destroy(food.gameObject);
    Spawn();
}
```

Klasa Snake:

```
public class Snake : MonoBehaviour {
    public Transform tailPrefab;
    public Color color;
    public Food food;
    public int x,y;
    public int length;

    private List<Field> tail;
    private List<Transform> tailTransforms;
    private Map.Side dir;
}
```

W Unity pola publiczne klasy dziedziczącej po *MonoBehaviour* są dostępne i edytowalne z poziomu edytora. Dobrze to będzie można zwizualizować na klasie *Snake* ponieważ posiada ona dodatkowo obiekt *Color* (rys.2).



Rys.2. Lista komponentów podczas modyfikacji zmiennej *Color* z poziomu edytora Unity.

Dzięki upublicznianiu zmiennych można w łatwy i szybki sposób modyfikować wartości w zależności od instancji komponentu. W tym wypadku żeby w prosty sposób odróżnić węże można wybrać im różne kolory z palety (rys.2).

Klasa *Snake* posiada również 2 Listy – jedna posiada pola mapy a druga obiekty w scenie. Żeby wytłumaczyć w jaki sposób się synchronizują, należy w pierwszej kolejności przedstawić metody odpowiedzialne za sam ruch. Podczas ruchu należy rozpatrzeć jedną z trzech możliwych sytuacji:

- pole jest przeszkodą – zabij węża
- pole jest jedzeniem – zjedz, dodaj ogon na koniec i przesuń się na to pole
- pole jest puste – pole na końcu oznacz jako puste, przesuń się na puste

Metody publicznego interfejsu do poruszania się:

```
public void Move() {
    Move(Map.Neighbour(Head(), Direction()));
}

public void Move(Field next) {
    var last = tail.Last();

    if(!next.IsWalkable()) {
        Die();
        return;
    }

    if(next.IsFood()) {
        food.Eat();
        AddTail(last);
        Slither(next);
    } else {
        last.ChangeField(Map.Fields.empty);
        Slither(next);
    }
}
```

Wywoływane są z komponentów sterujących. Metoda *Die*, powoduje uśmiercenie węża. *AddTail* sprawia, że długość węża zwiększa się o jedno pole w podanym miejscu. Slither powoduje przesunięcie się węża na wybrane pole poprzez zaktualizowanie dwóch list. Na początku następuje przesunięcie zmiennej *tail* się po obiekcie *Map*, a następnie wszystkie elementy ze sceny przesuwamy w odpowiednie pozycje pól zawartych w *tail*.

- Komponenty sterujące

Komponenty sterujące możemy podzielić na 3 typy:

- Główny komponent sterujący, co dany okres czasu wymusza odpowiedź pozostałych.
- Komponent sterujący gracza, czeka na wejście danych z klawiatury.
- Komponent sterujący poszczególnych algorytmów sztucznej inteligencji, wywołuje algorytm SI oraz interpretuje jego odpowiedź.

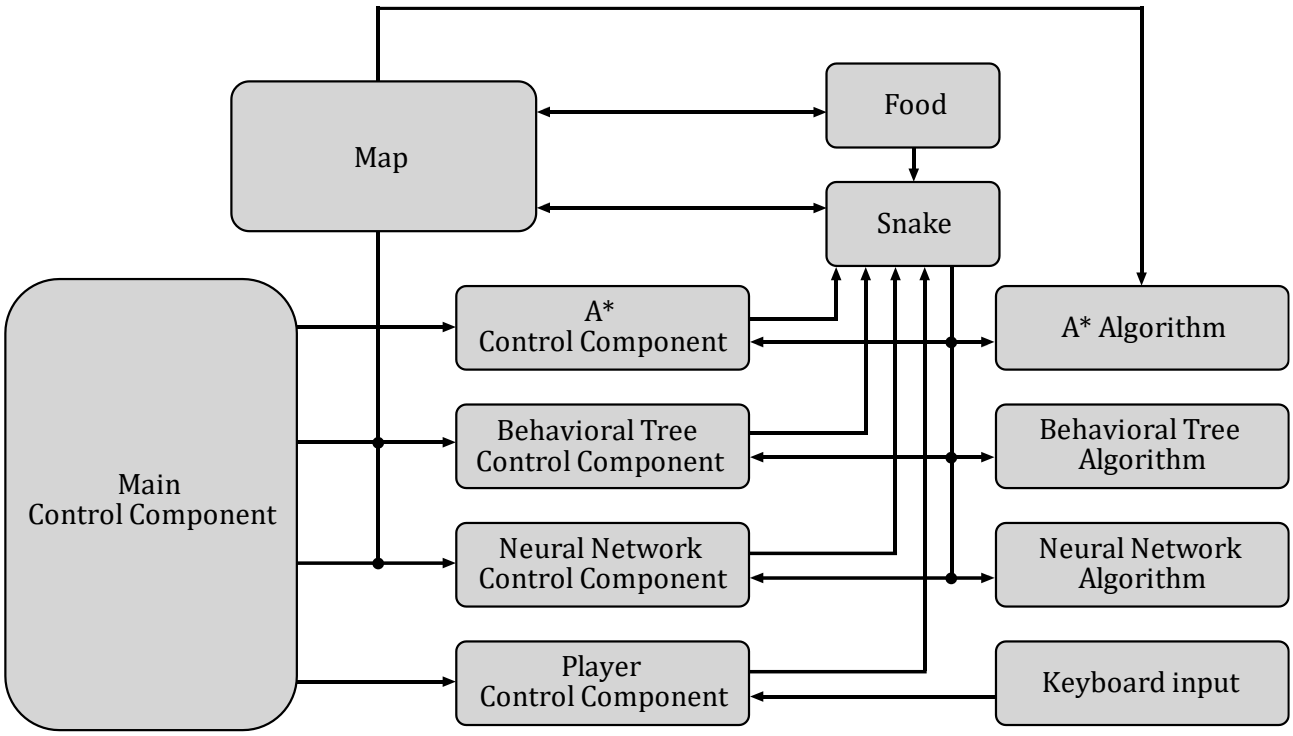
- Algorytmy sztucznej inteligencji

Zaimplementowano następujące algorytmy SI:

- Drzewo Behawioralne,
- Wyszukiwanie ścieżek - A*
- Sieć Neuronowa – Wielowarstwowy perceptron

Zostaną dokładnie opisane w kolejnych podrozdziałach.

Poniżej przedstawiono mapę zależności pomiędzy poszczególnymi komponentami (rys.3).

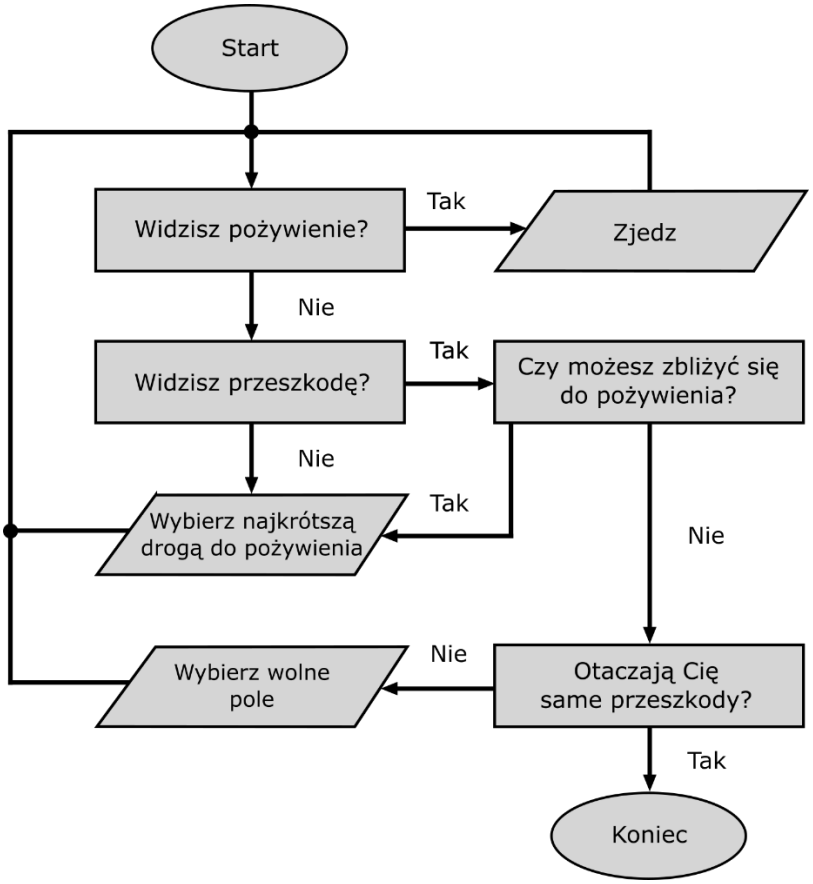


Rys.3. Schemat wymiany informacji pomiędzy komponentami projektu.

Jak widać na schemacie (rys.3), główny kontroler sterujący wyzwala specyficzne obiekty kontrolujące. Do Algorytmu drzewa behawioralnego i sieci neuronowej dostarczana jest mapa, z której wyłuskują niezbędne dane. Z mapy bezpośrednio korzysta algorytm A*. Każdy z komponentów kontrolujących ma za zadnie przekazać informację o podjętej decyzji do swojej instancji węża, który już zadba o swoje położenie na mapie.

3.2. Drzewo Behawioralne

Inaczej zwane drzewem decyzyjnym. Algorytm polega na przejściu poprzez wszystkie możliwe przypadki i podjęcie decyzji na podstawie sytuacji, w której aktualnie się znalazł. W projekcie algorytm ten dostaje następujące informacje: kierunek przemieszczenia, wektor przesunięcia względem pożywienia oraz ograniczoną percepcje do 3 sąsiadujących pól, na które może się udać. Priorytetem algorytmu jest minimalizacja wektora pomiędzy głowa węża a pożywieniem o ile jest to możliwe.



Rys. 4. Struktura algorytmu drzewa behawioralnego zaimplementowanego w projekcie.

Przygotowanie zmiennych:

```
var head        = snake.Head();
var dir         = snake.Direction();
var forward     = Map.Neighbour(head, dir);
var right       = Map.Neighbour(head, Map.Right(dir));
var left        = Map.Neighbour(head, Map.Left(dir));
var neighbours  = new Field[] {forward, right, left};
var dist        = new Perception(snake).DistanceToFood();
```

Zmienne *head* i *dir* mają zwiększyć czytelność, *forward*, *right*, *left* to pola w odpowiedniej kolejności: przed, z prawej i z lewej względem głowy węża i jego aktualnego kierunku ruchu. Zmienna *dist* jest wektorem pomiędzy głową a pożywieniem.

Implementacja algorytmu:

```
public Map.Side Run(Snake snake)
{
    var head        = snake.Head();
    var dir         = snake.Direction();
    var forward     = Map.Neighbour(head, dir);
    var right       = Map.Neighbour(head, Map.Right(dir));
    var left        = Map.Neighbour(head, Map.Left(dir));
    var neighbours  = new Field[] {forward, right, left};
    var dist        = new Perception(snake).DistanceToFood();

    var desired = GetDesiredDirection(dist);
    if(CanDirectionBeFollowed(head, neighbours, desired)) {
        return desired;
    }

    var alternative = GetAlternativeDirection(dist);
    if(CanDirectionBeFollowed(head, neighbours, alternative)) {
        return alternative;
    }
    else {
        foreach (var neighbour in neighbours) {
            if(neighbour.CanWalk()) {
                return Map.Direction(head, neighbour);
            }
        }
    }
    return dir;
}
```

Przy implementacji zastosowano kilka skrótów logicznych względem schematu (rys. 4), dzięki czemu otrzymano prosty i zwięzły kod.

Logika znajdowania optymalnego kierunku, opiera się na tym że są cztery możliwe scenariusze, zbliża/oddala się po jednej z 2 zmiennych. Algorytm stara się minimalizować największą ze współrzędnych. Logika znajdowania kierunku:

```
private Map.Side GetDesiredDirection(Vector2 dist) {
    Vector2 dir;

    if(Mathf.Abs(dist.x) > Mathf.Abs(dist.y)) {
        dir = new Vector2(-dist.x / Mathf.Abs(dist.x), 0);
    }
    else {
        dir = new Vector2(0, -dist.y / Mathf.Abs(dist.y));
    }

    return Map.Direction(dir);
}
```

Wykorzystuje ona funkcję matematyczną: $f(x) = -\frac{x}{|x|}$, która normalizuje odległość do wektora i zmienia jego zwrot. Metody *GetDesiredDirection* i *GetAlternativeDirection* różnią się tylko znakiem większości. Każda z nich rozpatruje wszystkie kierunki do pożywienia ale z odmiennym priorytetem. *GetDesiredDirection* zawsze zwróci kierunek minimalizujący większą współrzędną, natomiast *GetAlternativeDirection* odwrotnie – minimalizuje współrzędną mniejszą. Warto zauważyć fakt, że każda z funkcji - *GetDesiredDirection* i *GetAlternativeDirection,*

faworyzuje jeden scenariusz i nie bierze pod uwagę faktu że wąż nie potrafi zmienić kierunku ruchu o 180°. Jednak gdy użyje się ich obu to jedna z nich zwróci poprawną odpowiedź.

Ostatnią z metod wykorzystanych w algorytmie to CanDirectionBeFollowed. W niej następuję iteracja po podanych polach oraz sprawdzenie czy pole jest wolne i czy jego kierunek się zgadza z tym oczekiwanym.

```
    private bool CanDirectionBeFollowed(Field start, Field[] neighbours,
Map.Side dir) {
        foreach (var neighbour in neighbours) {
            if(neighbour.CanWalk() && dir == Map.Direction(start, neighbour)){
                return true;
            }
        }

        return false;
    }
```

Algorytm jest nieprzystosowany do wychodzenia z trudnych sytuacji – takich jak np. zamknięcie samego siebie. Jednak podczas testowania okazało się nie być to wielkim problemem gdyż w większości sytuacji, zmierzając wprost do pożywienia, są nikłe szanse na przegraną z własnego powodu. Nawet w teorii wydaje się być bardziej odporny niż implementacja algorytmu A* na sytuację, w których nie istnieje ścieżka do pożywienia.

3.3. A*

Algorytm znajduję najkrótszą ścieżkę w grafie ważonym. W przypadku tego projektu był to graf z wagami równymi. Algorytm jest zupełny – zawsze odnajdzie ścieżkę, o ile istnieje i jest ona najkrótsza. Polega na odwiedzaniu sąsiednich pól i uzupełnianiu kosztów przejścia jak i również przypisaniu pola z którego się przemieszczono. W celu zaimplementowania algorytmu trzeba było rozszerzyć interfejs publiczny klasy *Field* :

```
public class Field {
    public Map.Fields fieldType;
    public bool canWalk;

    public Vector2 pos; // World Position
    public int x, y;    // Map Position

    // A* Specific Variables
    public Field parent;
    public int h, g;
    public int f {
      get { return h+g; }
    }
}
```

Zmienne g,h i f to zmienne oznaczające koszty. Przyjęło się je tak nazywać od funkcji matematycznych stojących za algorytmem:

- g(x) jest funkcją zwracającą odległość od punktu startu,
- h(x) to funkcja heurystyczna, estymująca przybliżony koszt przejścia do celu
- f(x) jest równa ich sumie, f(x) = g(x) + h(x).

W przypadku programu stworzonego w ramach pracy dyplomowej zmienna h zawiera dane dokładne, co gwarantuje, że algorytm zwraca zawsze optymalne rozwiązanie. Zmienna *parent* jest ustawiana podczas przejścia do następnego pola. Jest to kluczowa operacja, gdyż pod koniec działania algorytmu z tych zmiennych będziemy w stanie odzyskać obraną ścieżkę.

Koncepcja algorytmu polega na posiadaniu dwóch kontenerów: węzłów otwartych (do odwiedzenia) oraz węzłów zamkniętych (odwiedzonych). Jego zadanie polega na uzupełnianiu i wybieraniu najtańszego węzła z listy otwartych. Węzeł zostaje przeniesiony do listy zamkniętych. Pole odwiedzone może zostać przeniesione do węzłów otwartych gdy spełni odpowiednie warunki kosztów. Co iterację, algorytm powinien sprawdzać czy ma dostępne pola otwarte oraz czy dotarł do celu.

Cały proces zaczyna się od prostej metody:

```
private Field FindPath() {
    if (this.PathExists()) {
        return RetracePath();
    }
    else {
        return this.start;
    }
}
```

Gdy ścieżki nie odnaleziono, metoda daje sygnał modułowi sterującemu, że następny krok to punkt startowy; wtedy on nie podejmuje żadnej akcji i czeka na zmianę sytuacji na planszy nie zmieniając kierunku poruszania się węża.

Sposób w jaki sprawdzamy czy ścieżka istnieje jest przedstawiona w poniższych metodach:

```
    private bool PathExists() {
        List<Field>    open   = new List<Field>();
        HashSet<Field> closed = new HashSet<Field>();
        this.UpdateCosts(this.start, this.start, 0);
        open.Add(this.start);

        while (open.Count > 0) {
            Field current = FindCheapest(open);
            open.Remove(current);
            closed.Add(current);

            if (current == this.target){
                return true;
            }

            this.AddPossibleSteps(current, open, closed);
        }

        return false;
    }
```

W metodzie *PathExists* wykonuje się trzon algorytmu A*. *FindCheapest* przeszukuje wszystkie pola otwarte i zwraca pole, którego f (g+h) jest najmniejsze, ale jednak cała logika związana ze zbieraniem, wycenianiem i zapamiętywaniem węzłów zaszyta jest w metodzie *AddPossibleSteps* oraz w metodach jej zależnych.

```
  private void AddPossibleSteps(Field current, List<Field> open, HashSet<Field>
closed) {
        foreach (Field neighbour in Map.GetNeighbours(current)) {

            if (closed.Contains(neighbour)) {
                var movementCost = current.g + GetDistance(current, neighbour);

                if (!neighbour.CanWalk() || movementCost > neighbour.g) {
                    continue;
                }
            }

            AddStep(current, neighbour, open);
        }
    }
```

Metoda *AddPossibleSteps* przeszukuje sąsiednie pola w poszukiwaniu potencjalnych węzłów otwartych. Gdy pole nie zostało odwiedzone, automatycznie jest klasyfikowane jako możliwe do odwiedzenia i jest przekazywane metodzie *AddStep*:

```
    private void AddStep(Field current, Field neighbour,  List<Field> open) {
        var movementCost = current.g + GetDistance(current, neighbour);

        if (!open.Contains(neighbour) || movementCost < neighbour.g) {

            this.UpdateCosts(current, neighbour, movementCost);

            if (!open.Contains(neighbour) && neighbour.CanWalk()) {
                open.Add(neighbour);
            }
        }
    }
```

Ta natomiast aktualizuje koszty przejścia i dodaje pole tylko w wypadku gdy nie ma go w liście otwartych węzłów i można po nim się poruszać.

Aktualizacja kosztów i wycenianie przejścia to klucz do poprawnego działania algorytmu:

```
private void UpdateCosts(Field current, Field neighbour, int movementCost) {
    neighbour.g = movementCost;
    neighbour.h = GetDistance(neighbour, this.target);
    neighbour.parent = current;
}

private int GetDistance(Field from, Field to) {
    int distanceX = Math.Abs(from.x - to.x);
    int distanceY = Math.Abs(from.y - to.y);

    return distanceX + distanceY;
}
```

Jako, że wąż potrafi poruszać się tylko po prostych, algorytm wyznaczania ceny przejścia *GetDistance* jest banalnie prosty i polega na zsumowaniu odległości po współrzędnych x i y.

W ostatnim kroku algorytmu należy odtworzyć ścieżkę. Służy do tego metoda *RetracePath*:

```
private Field RetracePath() {
    Field current = this.target;
    this.path.Clear();
    this.path.Add(current);

    while(current.parent != this.start) {
        current = current.parent;
        this.path.Add(current);
    }

    return current;
}
```

Zapisuję ona ścieżkę do zmiennej *path* i zwraca ostatni element ścieżki. Tylko ten jest potrzebny komponentowi sterującemu. Algorytm w celu optymalizacji działania wyszukuje ścieżkę tylko jeżeli nie może podążać wcześniej już wytyczoną:

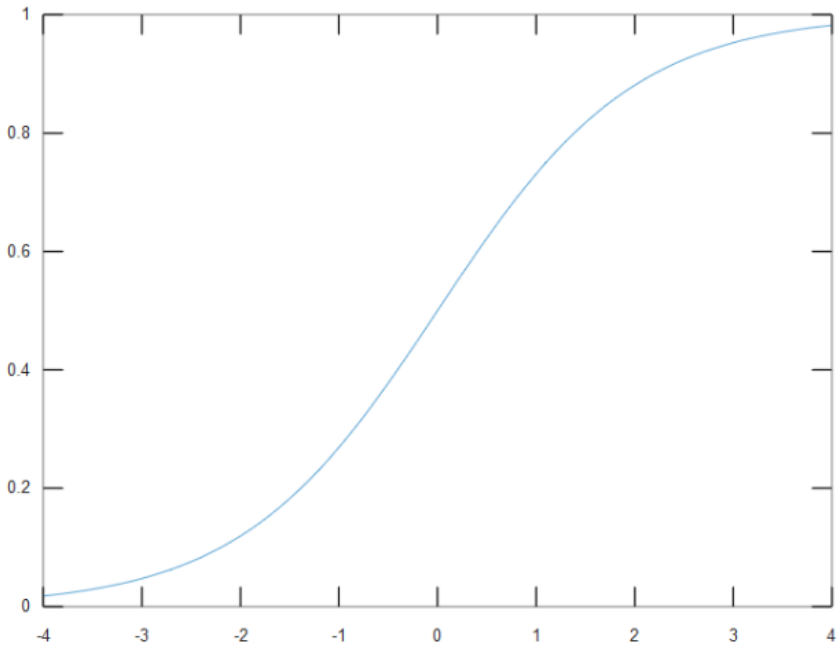
```
public Field FindPath(Field start, Field target) {
    this.start  = start;
    this.target = target;

    if(!this.IsEmpty()) {
        this.path.Remove(this.path.Last());

        if(this.IsActual()) {
            return this.path.Last();
        }
    }

    return FindPath();
}
```

Jeżeli ścieżka nie jest pusta i jest aktualna – można nią dojść do celu, zwróć jej następny element. W innym wypadku wyznacz nową.

3.4. Sieci Neuronowe – Wielowarstwowy perceptron

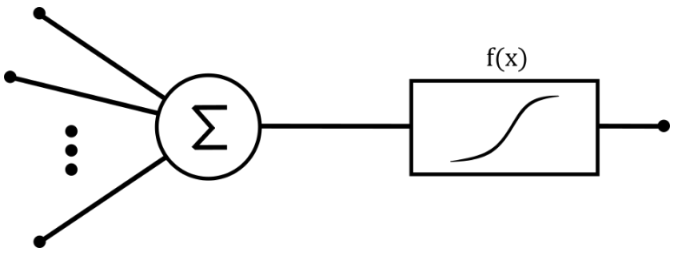
Sieci neuronowe zostały zainspirowane biologicznymi zjawiskami zachodzącymi w ludzkim mózgu. Podstawową jednostką obliczeniową jest perceptron równoważny jednej komórce nerwowej. Taki perceptron sumuje cały sygnał wchodzący i podejmuję decyzję o aktywacji. Decyzja ta jest zależna od funkcji aktywacji, która jest funkcją matematyczną standaryzującą sygnał. Przy perceptronach zwykle wykorzystuje się funkcję skoku jednostkowego albo sigmoidalną (wygładzona funkcja progowa). Choć istnieje ich wiele więcej, w zależności od budowy sieci i jej zastosowań, to projekcie wykorzystano tylko unipolarne sigmoidalne funkcje aktywacji.

Unipolarna funkcja sigmoidalna definiuje się wzorem: $f(x) = \frac{1}{1+e^{-\alpha x}}$, a jej wykres wygląda następująco (rys.5):



Rys.5. Wykres unipolarnej sigmoidy ze współczynikiem α = 1.

Szybkość narastania takiej funkcji sigmoidalnej zależy od współczynnika α, gdy α→∞, to funkcja sigmoidalna dąży do funkcji skoku jednostkowego.



Rys.6. Schemat budowy perceptronu.

Właściwością pojedynczego perceptronu (rys.6) jest to, że potrafi rozwiązywać tylko problemy liniowo separowalne. Dlatego żeby rozwiązywać bardziej skomplikowane zadania, trzeba użyć wielu współpracujących ze sobą neuronów tworząc strukturę nazywaną sieciami neuronowymi.

Istnieje wiele różnych typów sieci. Można wyznaczyć kilka szczególnych klas tych obiektów – sieci jednokierunkowe, sieci rekurencyjne (z pamięcią), sieci samoorganizujące, sieci radialne. Temat tych struktur istnieje już dość długo bo już od lat 50 ubiegłego stulecia. W latach 60 pojawił się pierwszy model perceptronu. Amerykański psycholog Frank Roseblatt był jego twórcą. Przeprowadzał on szereg eksperymentów w dziedzinie działania mózgu oraz pomięci. Po swoim sławnym odkryciu przeprowadzał eksperymenty na szczurach, w których ekstraktował umysły nauczonych osobników i wprowadzał je do osobników niewytrenowanych. Wyniki eksperymentu wskazują, że pamięć nie jest lub jest w nieznacznym stopniu przenoszona pomiędzy osobnikami.

Sieci neuronowe znajdują szerokie zastosowanie w rozpoznawaniu mowy i pisma. Mają też bardzo obiecujące wyniki w medycynie, gdzie wykorzystywane są np. do wczesnego rozpoznawania zmian nowotworowych czy w celach diagnostycznych. Ekonomiści też często sięgają po sieci neuronowe w celu przewidzenia stanu giełdy papierów wartościowych.

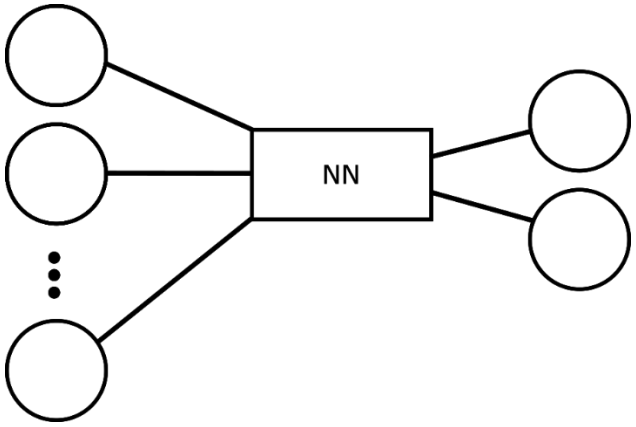
Wielowarstwowy perceptron wykorzystuje uczenie nadzorowane. Oznacza to, że jest nauczyciel (zwykle algorytm nadzorujący uczenie), który przygotowuję serie pytań i odpowiedzi na nie, po czym koryguje błędne odpowiedzi sieci neuronowej. Do uczenia wykorzystuje się algorytm wstecznej propagacji błędu który polega na propagowaniu błędu na warstwy ukryte i modyfikację ich wag.

W ramach projektu stworzono bardzo podstawową bibliotekę pozwalającą na budowę wielowarstwowego perceptronu o dowolnej wielkości z sigmoidalnymi funkcjami aktywacji, uczenie, zapisanie i wczytanie go z pliku. W dalszej części pod słowem implementacja ukrywa się wykorzystana budowa sieci oraz dostosowane do niej przykłady uczące, ponieważ to jest najbardziej pracochłonny proces – dobieranie parametrów sieci oraz przykładów uczących.

Implementacja tego algorytmu miała wiele iteracji podczas życia projektu. W pierwszej algorytm został zaimplementowany bezpośrednio w silniku gry i tam przeszedł parę przygód. Na początku pomysł był prosty - algorytm dostaje całą mapę i dostaje przykłady z zarejestrowanych gier. Zapisywano aktualny stan mapy oraz odpowiedź gracza na dany stan rzeczy. Próba okazała się totalnym fiaskiem, ponieważ złożoność obliczeniowa zależała w dużej mierze od wielkości mapy, a do tego brak jakichkolwiek zauważalnych efektów spowodował szybką zmianę podejścia do problemu. W następnej próbie uproszczono model danych wejściowych, do sześciu elementów: pięć pól wokoło głowy i informacja w postaci binarnej czy porusza się w kierunku pożywienia. Przygotowano dane uczące ręcznie, zwykle w momentach krytycznych, gdzie reakcja ze strony sieci była wymagana. Pomysł okazał się działać, ale jego skuteczność malała ze wzrostem wielkości mapy. Przypuszczalnie sieć wiedziała jak się zachowywać gdy w otoczeniu występowały przeszkody, więc zwykle podążała wokół mapy trzymając się ścian. Z dotychczasowym doświadczeniem podjęto decyzję by rozwinąć ten pomysł. Wymieniono informację binarną na zestaw 2 wektorów dwuwymiarowych – przesunięcie wektorowe od pożywienia oraz wersor z informacją po której osi wąż się porusza. Niestety to podejście nigdy w pełni nie zadziałało pomimo wielokrotnych prób – 3 implementacji w 2 różnych językach. Pierwsza implementacja odbyła się w samym Unity. Szybko się okazało, że moc obliczeniowa jest niewystarczająca. Próbowano zreimplementować całą bibliotekę w języku Rust, ale porzucono ten pomysł, gdyż język okazał się być wówczas zbyt skomplikowany. Ostatecznie wybrano .Net Core, ponieważ można było swobodnie przenieść istniejący kod ze skryptów z Unity. Tam okazało się, że przygotowane ręcznie

przykłady są niezbyt różnorodne. Utworzono metody generujące przykłady. Sieć dalej nie chciała się nauczyć, a więc wprowadzono losowość do tworzenia konfiguarcji sieci, różne funkcje aktywacji per warstwa, różne liczba warstw i neuronów w tych warstwach. Do zoptymalizowania zastosowano prosty algorytm genetyczny i żeby działał sprawnie używał przetwarzania wielowątkowego. Jednak i tutaj się nie powiodło.

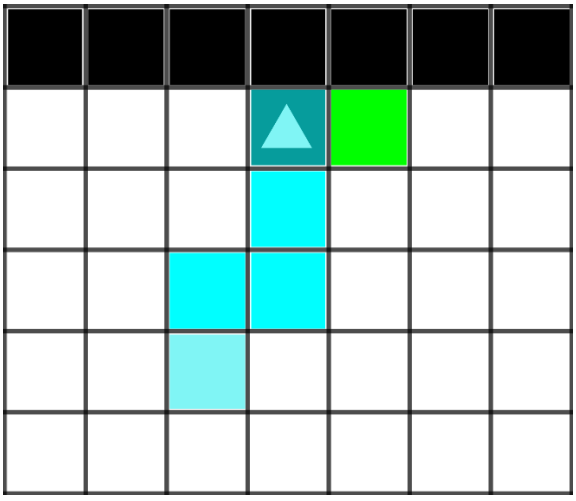
Ostatnim podejściem wykonanym w stronę próby nauczenia sieci, było uczenie nadzorowane algorytmem drzewa behawioralnego. Niestety zabrakło czasu na wdrożenie tej iteracji.



Rys. 7. Model przedstawiający ogólny zamysł jednokierunkowej sieci neuronowej.

Powyżej (rys.7.) przedstawiono ogólny model sieci zastosowanej w projekcie. Sygnał przepływa od lewej do prawej. Struktura sieci jest bliżej niezdefiniowana, ponieważ użyto wielu różnych możliwych kombinacji, a w końcowej fazie projektu wprowadzono generowanie warstw oraz liczby neuronów w każdej warstwie losowo. Najważniejszym jednak elementem tego modelu jest schemat wejść i wyjść. Aktualnie sieć posiada siedem wejść i dwa wyjścia. Schemat informacji dostarczanych do sieci jest taki sam jak w przypadku drzewa behawioralnego. Z jednym małym wyjątkiem – wszystkie pola mapy są rzutowane z enumeratora *Fields* (typ pola) na wartość liczbową. Na wyjściu sieci oczekiwano odpowiedzi w postaci kombinacji 2 bitów:

- [0,0],[1,1] – nie zmieniaj kierunku
- [1,0] – skręć w prawo
- [0,1] – skręć w lewo



Rys.8. Przykładowa sytuacja podczas gry.

Przykładowy format danych wejściowych do sieci neuronowej:

Pierwsze trzy wartości to pola otaczające głowę w formacie [lewa, na wprost, prawa]:

- -1.0, to pożywienie
- 0.0, to pole puste,
- 1.0, to pole ze ścianą,
- 2.0, to pole z wężem.

Dwie następne wartości to wektor kierunku ruchu w formacie [x, y]

- $x=1 \wedge y=0$, ruch w prawo,
- $x=-1 \wedge y=0$, ruch w lewo,
- $x=0 \wedge y=1$, ruch w górę,
- $x=0 \wedge y=-1$, ruch w dół.

Pozostałe dwie wartości mówią jak daleko znajduje się pożywienie. Staje się ono środkiem układu współrzędnych, a wartości wektora mówią o tym w jakim miejscu w tym układzie znajduje się głowa węża, również wyrażone w formacie [x, y].

Przykład praktyczny formatu danych na podstawie zadanej sytuacji w grze (rys.8.):
[0.0, 1.0, -1.0, 0.0, 1.0, -1.0, 0.0]

Obliczenia sygnałów w sieciach neuronowych wykonuje się na macierzach. Podczas tworzenia projektu skorzystałem, z gotowej biblioteki do obliczeń macierzowych stworzoną przez Ivan'a Kuckir'a, udostępnioną na licencji MIT.

Implementacja:

```
public class NeuralNetwork {
    private Matrix[] nn;

    public NeuralNetwork(int[] layers) {
        var lsl = layers.Length - 1;
        this.nn = new Matrix[lsl];

        for(int i = 0; i < lsl; i++) {
            this.nn[i] = Matrix.RandomMatrix(layers[i] + 1, layers[i+1]);
        }
    }
}
```

Klasa *NeuralNetwork* posiada tablice macierzy wag odpowiednich połączeń neuronowych w każdej warstwie. Inicjalizajcja klasy polega na stworzeniu zadanej ilości macierzy z losowymi wartościami wag z przedziału (-1,1).

Uczenie sieci:

```
public bool Learn(Matrix signal, Matrix anwser) {
    int layers = nn.Length;

    Matrix[] signals = RunAndReturnSignals(signal, layers);
    Matrix delta = anwser - signals[layers];
    BackPropagation(signals, delta, layers);

    return Test(signal, anwser);
}
```

W poprzednim listingu przedstawiono metodę uczącą sieć pojedynczego zagadnienia. Wywoływana jest w komponencie kontrolujący sieć neuronową. Użyto w niej trzech wewnętrznych metod obiektu *NeuralNetwork*: *RunAndReturnSignals*, *BackPropagation* oraz *Test*. Pierwsza z nich jedyne co robi to wymnaża poszczególne macierze sygnałów i wag ze sobą, a następnie zwraca je w tablicy. BackPropagation ma na celu realizować algorytm wstecznej propagacji błędu, a *Test* przetestować odpowiedź sieci i stwierdzić czy udzieliła ona poprawnej odpowiedzi po wywołaniu metody uczącej.

Algorytm wstecznej propagacji błędu:

```
        void BackPropagation(Matrix[] signals, Matrix delta, int layers) {
            Matrix signal;

            for(int i = layers; i > 0; i--) {
                signal = signals[i];
                if(signal.rows > delta.rows)
                    signal = RemoveBias(signal);

                Matrix error = FindError(signal, delta);
                Matrix dnn = learningRate * signals[i-1] *
Matrix.Transpose(error);
                nn[i-1] += dnn;

                delta = RemoveBias(nn[i-1]) * error;
            }
        }
```

Algorytm polega na szacowaniu błędu każdej warstwy na podstawie iloczynu różniczki funkcji aktywacji i różnicy sygnału propagowanego wstecz błędu (*delta*) – metoda *FindError*. Następnie należy dostosować wagi w sieci neuronowej z określoną siłą – przemnożenie błędu przez małą stałą wartość szybkości uczenia (*learningRate*).

4. Wnioski

Pisanie gier komputerowych bywa bardzo pouczające, w szczególności w dziedzinie optymalizacji wydajności. Trzeba mocno oszczędzać zasoby sprzętowe, tym bardziej gdy próbuje się implementować złożone obliczeniowo algorytmy, takie jak użyte w pracy np. A* czy SSN. Implementacja algorytmów sztucznej inteligencji w pracy była zabiegiem czysto naukowym. Unity oferuję algorytm wyszukiwania ścieżek czy budowania drzew decyzyjnych, które z pewnością wydajnościowo sprawdzą się znacznie lepiej niż te zaimplementowane w C# i odpalane na maszynie wirtualnej. Nie powinno się „wymyślać koła na nowo", ale takie doświadczenie zawsze jest pomocne podczas chociażby koncepcyjnego zrozumienia zagadnienia i uniknięcia błędów związanych z wykorzystaniem już gotowych materiałów. Bardzo podobnie można byłoby powiedzieć o implementacji własnej biblioteki SNN. Jednak tutaj przydałoby się dodać, że naprawdę nie warto tworzyć sieci neuronowych od samych podstaw w celu ich użycia. Istnieje szereg bibliotek które, w łatwy i co najważniejsze – skuteczny sposób pozwolą na skonstruowanie sieci o potrzebnych parametrach i właściwościach. Nie opłacalnym jest stosować sieci neuronowych do rozwiązania problemów, które można rozwiązać algorytmem dokładnym czy heurystycznym, ponieważ dają one znacznie lepsze wyniki. Dodatkowo, trzeba zwrócić uwagę na to, że zastosowana w projekcie architektura sieci jest bardzo przestarzała.

Podczas pisania pracy zdałem sobie sprawę jak dużo mnie kosztowało stworzenie SSN w stosunku do pozostałych algorytmów, a jakie otrzymałem z tego korzyści. Aktualna implementacja SSN nie zadziałała, a gdyby jednak udało się ją uruchomić to w najlepszym wypadku byłaby gorszym drzewem decyzyjnym. W aktualnym stanie SSN czasem zareaguje na bodźce, by chwile później zacząć kręcić się wokół siebie albo uderzać głową w ścianę.

Pomimo traumatycznych przeżyć, uważam tą pracę za cenną naukę. Wymagała ona wytrwałości w dążeniu do celu, tworzeniu dobrej jakości oprogramowania by nie zagubić się w całym projekcie oraz nauczyła sceptycyzmu do SSN.

Bibliografia:

Literatura:

1. McIlwraith, D., Marmanis, H. and Babenko, D. (2016). *Allgorithms of the intelligent web*. Shelter Island, NY: Manning Publications Co.

Źródła internetowe:

1. Dokumentacja Unity, https://docs.unity3d.com/Manual/index.html, 21.11.2018
2. Matheus Amazonas, *How does Unity3D scripting work under the hood*, https://sometimesicode.wordpress.com/2014/12/22/how-does-unity-work-under-the-hood/, 22.12.2014
3. Artykuł na temat istoty silnika gry, https://unity3d.com/what-is-a-game-engine
4. Dokumentacja .Net Core, https://docs.microsoft.com/en-us/dotnet/core/index, 08.01.2018
5. Frank Rossenblatt, John T. Farrow and Sam Rhine , The transfer of learned behaviour from trained to untrained rats by mean of brain extracts, http://www.pnas.org/content/pnas/55/3/548.full.pdf, 20.12.1965