POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: Automatyka i Robotyka
SPECJALNOŚĆ: Systemy informatyczne w automatyce

PRACA DYPLOMOWA INŻYNIERSKA

Zastosowanie wybranych algorytmów sztucznej inteligencji w grach

Application of chosen artificial intelligence methods in games

AUTOR:
Paweł Jan Rynowiecki

PROWADZĄCY PRACĘ:
dr inż., Łukasz Jeleń,
Katedra Informatyki Technicznej

OCENA PRACY:

WROCŁAW 2018

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. Jeżeli jeden z węży natrafi 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. Unity jest ciągle rozwijane, posiada obszerną i ciąglę aktualizowaną dokumentację. 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 .Net core 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.

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#. Unity wspiera platformę .Net w wersji 3.5 a kod uruchamiany jest w środowisku Mono. Kod zarządzany pomimo tego, że jest wydajny nie jest w stanie konkurować z kodem natywnym, dlatego sam silnik jest napisany w C/C++ [3] a API (*Application Programming Interface*) silnika dostępne w C# tylko opakowuje te biblioteki.

Podczas tworzenia gry w Unity można rozróżnić kilka 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 (*User Interface*) programu (rys.1). Podczas przeładowania sceny wszystko co należało do poprzedniej zostaje zniszczone. Wspaniałą funkcjonalnością Unity są Prefabrykaty (*Prefab*). 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. Sprawia to, że utrzymanie projektu staje się bardzo proste. W projekcie tworzonym na potrzeby tej pracy, każde pole gry jest jednym prefabrykatem. W przypadku potrzeby zmiany wielkości pól mapy wystarczy poprawić wartość w jednym miejscu.

Unity posiada wsparcie dla ogromnej liczby platform. Przy użyciu odpowiedniego SDK (*Software development kit*) można swój projekt opublikować na następujących urządzeniach:

- Komputery personalne (Windows, Linuks i MacOS)
- Platformy mobilne (Android i iOS)
- Konsole do gier (PS4, Xbox One)
- Oraz wiele innych

Stwarza to niesamowitą sposobność twórcą, gdyż przy niewielkim nakładzie pracy projekt może zostać opublikowany na wielu platformach jednocześnie.

By przyśpieszyć proces tworzenia oprogramowania oraz dać szansę samotnym twórcą spieniężyć swoją pracę. Unity otwarło sklep z materiałami do gier (*Asset Store*), gdzie można publikować swoje niestandardowe komponenty, różnego rodzaju grafiki, modele, animacje czy nawet muzykę do gry. Znajdują się tam zarówno darmowe jak i płatne materiały.

Podsumowując, Unity jest wspaniałym narzędziem do tworzenia gier, animacji komputerowych. Spisze się nawet do innych programów, które kładą duży nacisk na intefejs użytkownika. Pozwala na szybkie zarządzanie komponentami nawet przy większych projektach. Udostępnia swoje API w popularnych językach programowania wysokiego poziomu. Posiada kompletną i szczegółową dokumentacje. Wspiera wiele platform i powiązanych z nimi urządzeń.

2.2. .Net Core

.Net Core jest platformą programistyczną typu open source, dostępną na popularnych systemach operacyjnych takich jak Windows, Linuks czy macOS. Została stworzona przez firmę Microsoft. Z początku wydawało się, że miał być to projekt równoległy do .Net Framework. Jednak z powodu jej niespodziewanej popularności Microsoft zdecydował się, że nie będzie dłużej rozwijał klasycznego .Net'a. Od swojego starszego brata odróżnia ją to, że kod źródłowy jest dostępny dla każdego na popularnym serwisie hostingowym GitHub. Dzięki takiemu podejściu można samemu przyczynić się do rozwoju projektu poprzez np. zgłaszanie błędów, propozycje zmian czy nawet własnoręczną modyfikację kodu źródłowego.

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

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 CLR (*Common Language Runtime*). Ważną cechą tego języka jest to, że posiada on odśmiecanie pamięci (*Garbage Collector*) czyli automatyczną dealokację pamięci stosu. Daje to użytkownikowi języka dużą swobodę podczas pracy z dynamiczną alokacją pamięci podręcznej. Niestety takie rozwiązanie też ma swoje wady. Współbieżnie do programu działa proces, który co jakiś czas blokuje działanie programu, by pozwalniać niepotrzebne już zasoby. Powoduje to, że C# nie ma szansy konkurować pod względem szybkości z językami gdzie programista sam zarządza pamięcią jak np. C/C++ czy Rust.

W parze z .Net idzie kolejny produkt Microsoft'u – Visual Studio. Jest to środowisko programistyczne IDE (*Integrated development environment*) czyli program, który służy do produkcji oprogramowania nie tylko poprzez edycję kodu źródłowego. Powinien udostępniać programiście szereg funkcjonalności. Visual Studio posiada ich zawrotną ilość: począwszy od generowania kodu i zarządzania projektem poprzez narzędzia do debugowania, integracji z bazą danych oraz Azure, oraz systemami kontroli wersji takimi jak Git, kończąc na zarządzaniem zewnętrznymi bibliotekami poprzez managera pakietów NuGet.

W skrócie .Net Core jest platformą programistyczną, w której używa się języków zarządzanych takich jak C# czy F#. Dzięki posiadanej strukturze, kod stworzony na jednej maszynie może być swobodnie przenoszony pomiędzy różnymi urządzeniami tak długo jak istnieje dla nich środowisko uruchomieniowe. Jest to projekt z otwartym kodem źródłowym, gdzie każdy może dodać

coś od siebie. Zaleca się używać zintegrowanego środowiska programistycznego jakim jest Visual Studio, który w dużym stopniu potrafi ułatwić pracę, jednakże narzędzia konsolowe nie odstają mocno w tyle.

2.3. Git

Git jest rozproszonym systemem kontroli wersji [7], stworzona przez Linusa Torvaldsa podczas rozwijania jądra systemu operacyjnego Linuks. Wykorzystuje on drzewiasty model zmian, gdzie zależności trzymane są w relacji rodzic – dziecko. Zmiany pomiędzy rewizjami są trzymane w postaci obrazów całego projektu (*Snapshot*).

Kolejne zmiany wprowadzane są w commit'ach. Każdy następny jest dzieckiem poprzedniego – nakładają się na siebie tworząc strukturę drzewiastą. Każde zmiany znajdują się w jakieś konkretnej gałęzi. Gałąź (*branch*) jest zatem serią commit'ów wychodzących z korzenia do samej głowy (*HEAD*). Wszystkie gałęzie wychodzą od korzenia, jednak pierwszą gałęzią, która się pojawia jest gałąź master. Przyjęło się, że jest gałęzią główną każdego projektu, do której powinny trafiać, zaakceptowane przez właściciela repozytorium, zmiany. Pracując z gitem należy utworzyć swoją gałąź i to do niej wprowadzać kolejne rewizję, a na sam koniec połączyć (*merge*) serie dokonanych zmian do gałęzi głównej. Podczas gdy wiele osób pracuje nad projektem, może się okazać, że podczas merge'a git wykrył konflikt – oznacza to, że ten sam kawałek został zmodyfikowany na dwa różne sposoby. Wówczas git daje wybór użytkownikowi, które zmiany chce zachować – proces ten nazywany jest rozwiązywaniem konfliktów.

Program Git jest programem darmowym, dostępny na licencji GNU GPL (*General Public Licence*). Jest często stosowany przy rozwoju oprogramowania. Posiada bardzo elastyczny zestaw narzędzi konsolowych. Git w sprawnych rękach pozwala na absolutną manipulacje historią zmian dokonanych na projekcie. Poniżej zostaną wymienione najbardziej popularne komendy gita wraz z ich krótkim opisem:

- *$ git init*, inicjalizuje lokalne repozytorium gita,
- $ *git clone*, klonuje zdalne repozytorium, przyjmuje jego adres jako parametr,
- $ *git add*, dodaje podane pliki do etapu, z którego mogą być zapisane w rewizji,
- $ *git commit*, dodaje nową rewizję,
- $ *git branch*, tworzy nową gałąź,
- $ *git merge*, pozwala na łączenie zmian z wielu rewizji w jedną.

Pracę z gitem można rozdzielić na pracę na lokalnym repozytorium oraz na repozytorium zdalnym (*remote*). Zwykle zmiany są wprowadzane w repozytorium lokalnym, a gdy użytkownik chce podzielić się dokonanymi zmianami, to „wypycha" (*push*) je do zdalnego repozytorium. Przykładem serwisu oferującego usługi w postaci zdalnego repozytorium jest np. serwis GitHub. Został on również wykorzystany do rozwoju oprogramowania na potrzeby tej pracy. Umożliwia on przetrzymywanie zasobów za darmo o ile utworzone repozytorium pozostaje publicznym. W przypadku projektu zmiany są zachowane w zdalnym repozytorium w w/w serwisie.

Słowem zakończenia: git oferuje efektywną pracę przy małych i dużych projektach. Jest programem kontrolującym przepływ zmian dokonanych podczas życia projektu. Jest szybki i darmowy.

3. Struktura aplikacji

3.1. Wprowadzenie

Stworzenie tej pracy opierało się na wykonaniu gry komputerowej, a następnie szczegółowego przeanalizowania wykonanej pracy. By temu podołać, należało poznać zasadę działania silnika graficznego Unity. Poznać jego możliwości, zapoznać się z dokumentacją. Ponadto trzeba było posiadać znajomość składni języka C# oraz przyswoić zasady działania algorytmów SI (Sztucznej Inteligencji) takich jak: A*, drzewo decyzyjne i SSN (Sztuczne Sieci Neuronowe).

3.2. Moduły i zależności między modułowe

3.2.1. 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 znajdujących się w obiekcie *Map*. Daje możliwość transformowania pozycji mapy na pozycję w scenie gry.

LISTING 1. Klasy *Map* i *Field*

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

    public Vector2 pos; // Pozycja w scenie
    public int x, y;    // Pozycja na mapie
}

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 kierunku ruchu. Dodatkowo został zaimplementowany *MapBuilder*, który przed uruchomieniem rozgrywki tworzy wszystkie potrzebne obiekty w pamięci i wywołuję metody (listing 2.), które graficznie je zaprezentują.

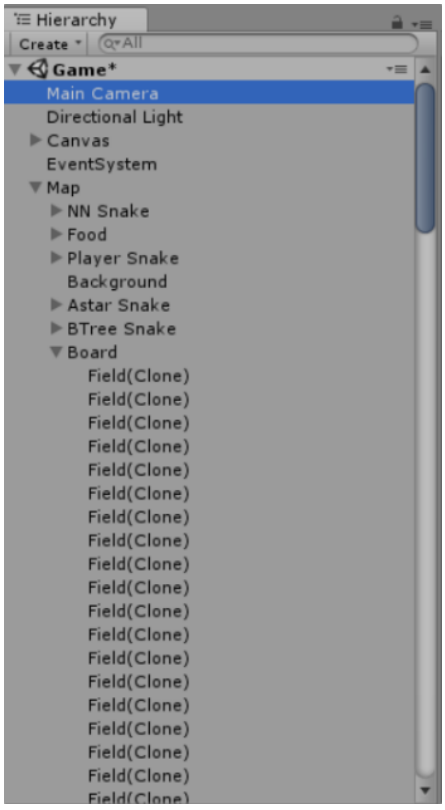
LISTING 2. Przykład metody w *MapBuilder* do tworzenia ścian w świecie gry.

```
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* (listing 2.), 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 (*Quaternion.identity*). 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*.

3.2.2. 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.

LISTING 3. Klasa *Food*.

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

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

    void Start() {
        Spawn();
    }

    public void Eat(){
        Destroy(food.gameObject);
        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 [2] to podstawowy obiekt z którego dziedziczą wszystkie obiekty w scenie (listing 3.). 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. Udostępnia też metodę *Update*, która wykonuje się raz na jedną klatkę gry. 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.

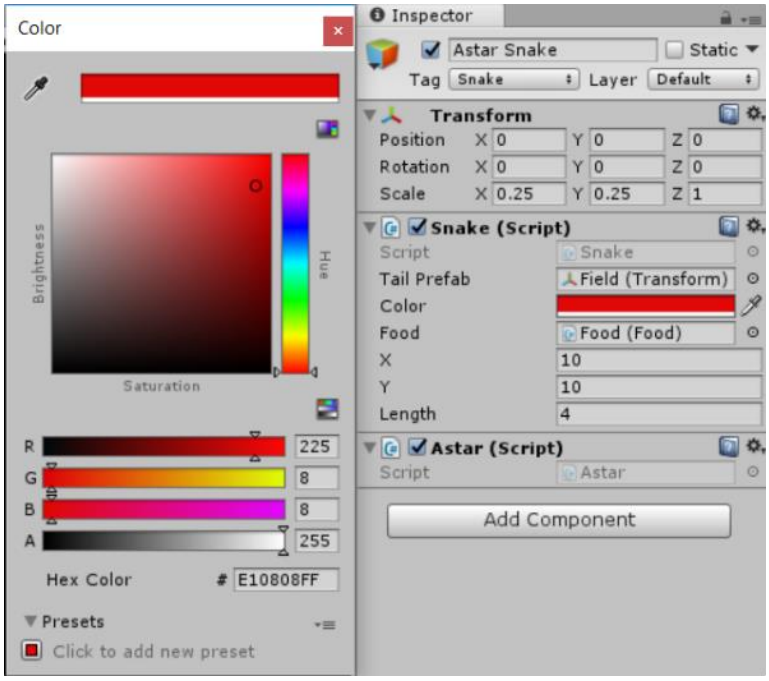
Klasa *Food* posiada dwie zmienne typu *Transform*, w *foodPrefab* trzymany jest prefabrykat obiektu do umieszczenia w scenie. Zmienna *Food* natomiast służy w celach dostępowych do klona w scenie, by móc go w dowolnej chwili zniszczyć (Metoda *Eat*).

LISTING 4. 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* (listing 4.)(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* (listing 4.) 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

LISTING 5. Metody publicznego interfejsu klasy *Snake* 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);
    }
}
```

Metody pokazane na listing 5. wywoływane są z korespondujących 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*.

3.2.3. Komponenty sterujące

Komponenty sterujące mają za zadanie wymieniać informację pomiędzy algorytmami, interpretować dostarczone dane oraz zmieniać kierunek ruchu węża. Można je 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.

Najwyżej w hierarchii z w/w przedstawionych jest główny komponent sterujący, który dzieli się na dwa etapy działania (listing 6.) zebranie wszystkich komponentów sterujących oraz (listing 7.) wymuszenie ruchu.

LISTING 6. Zebranie komponentów sterujących przez komponent nadrzędny.

```
void Start ()
{
    this.astar  = GameObject.Find("Astar Snake").GetComponent<Astar>();
    this.player = GameObject.Find("Player Snake").GetComponent<Player>();
    this.nn     = GameObject.Find("NN Snake").GetComponent<NN>();
    this.bt     = GameObject.Find("BTree Snake").GetComponent<BTree>();

    InvokeRepeating("MoveSnakes", 0.3f, 0.3f);
}
```

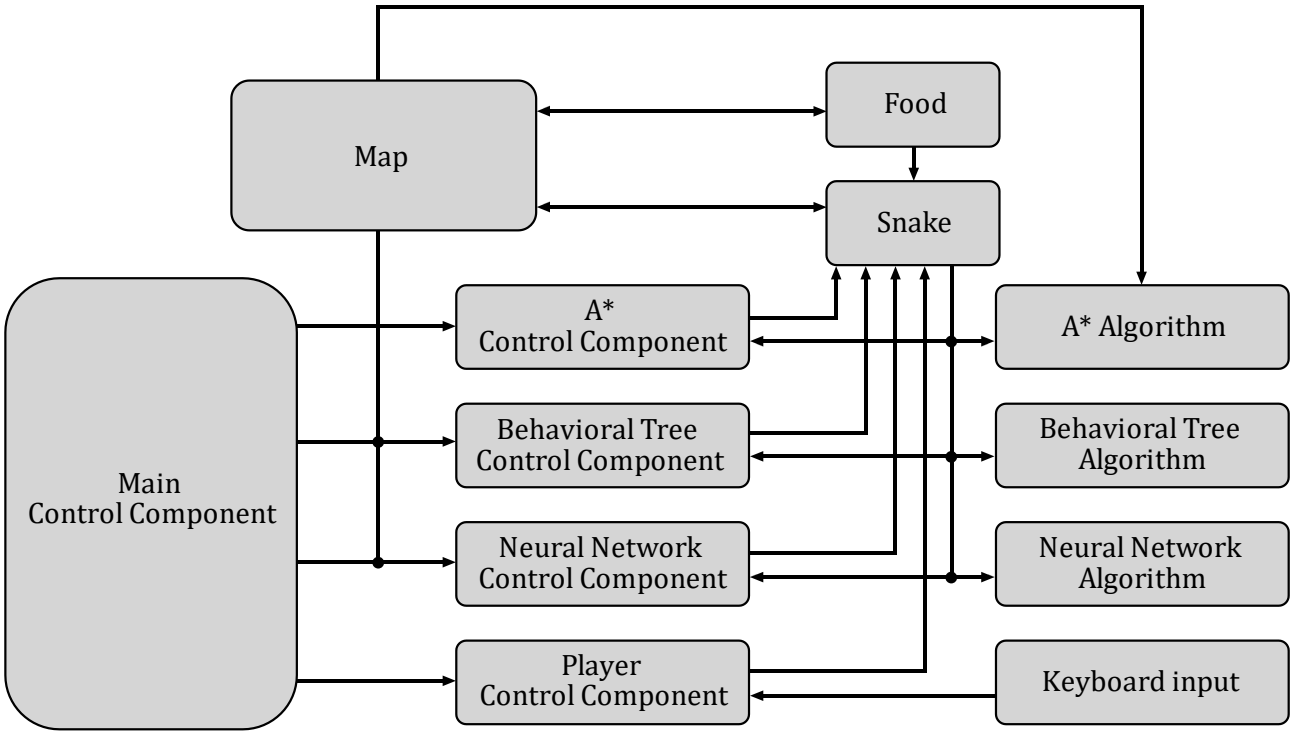
W listingu 6. używana jest metoda *Find*, która znajduje się w obiekcie bazowym każdego innego obiektu w Unity – *GameObject*. W samym C# tą samą rolę spełnia *Object*. Są to klasy z których dziedziczą wszystkie pozostałe klasy w programie. Metoda *Find* przeszukuję wszystkie byty w scenie szukając po nazwie. W kroku następnym celem wywołania *GetComponent<T>* jest zebranie komponentów sterujących odpowiednich algorytmów. Na samym końcu tworzymy event wywołania metody *MoveSnakes* co 0.3s (Metoda *InvokeRepeating*).

LISTING 7. Wymuszenie ruchu co okres czasu – główny komponent sterujący.

```
private void MoveSnakes()
{
    this.player.MoveSnake();
    this.astar.MoveSnake();
    this.nn.MoveSnake();
    this.bt.MoveSnake();
}
```

3.2.4. Mapa aplikacji

Poniżej przedstawiono schemat działania aplikacji: wymiany danych oraz spis wszystkich komponentów biorących udział w życiu aplikacji.

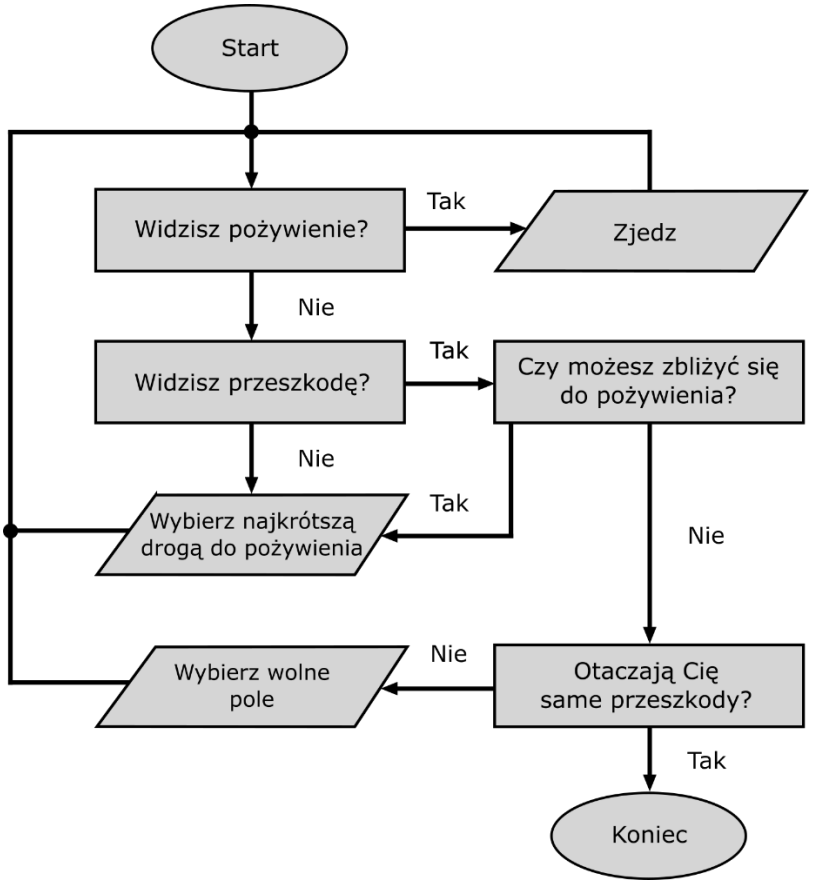


Rys.3. Schemat wymiany informacji pomiędzy komponentami projektu.

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 zadba o swoje położenie na mapie.

3.3. 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.

LISTING 8. Przygotowanie zmiennych w wywołaniu algorytmu.

```
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.

LISTING 9. Implementacja algorytmu drzewa decyzyjnego.

```
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;
}
```

Podczas implementacji (listing 9.) zastosowano kilka skrótów logicznych względem schematu (rys. 4), dzięki czemu otrzymano prosty i zwięzły kod.

Logika znajdowania optymalnego kierunku, wykorzystuje fakt że są cztery możliwe scenariusze, zbliża/oddala się w jednej z 2 zmiennych. Algorytm stara się minimalizować straty wykorzystując do tego dwie funkcję kierunkowe. W listingu 10. znajduje się przykład implementacji jednej z nich.

LISTING 10. Logika znajdowania kierunku w algorytmie drzewa behawioralnego.

```
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);
}
```

Metody znajdujące kierunek korzystają z funkcji matematycznej:

$$f(x) = -\frac{x}{|x|},$$

która normalizuje odległość do wektora i zmienia jego zwrot. Metody *GetDesiredDirection* i *GetAlternativeDirection* różnią się tylko znakiem nierówności. 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 użyte są obie to jedna z nich musi dać poprawną odpowiedź.

Ostatnią z metod wykorzystanych w algorytmie to *CanDirectionBeFollowed* (listing 11.). W niej następuję iteracja po podanych polach w celu sprawdzenia czy pole jest wolne i czy wygenerowany kierunek jest możliwy do spełnienia.

LISTING 11. Sprawdzenie możliwości podążania obranym kierunkiem.

```
    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. W praktyce wydaje się nawet być bardziej odporny niż implementacja algorytmu A* np. na sytuację, w których nie istnieje ścieżka do pożywienia.

3.4. 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* .

LISTING 12. Rozszerzenie klasy *Field* o dodatkowe pola.

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

    public Vector2 pos; // Pozycja w scenie
    public int x, y;    // Pozycja na mapie

    // Zmienne dla algorytmu A*
    public Field parent;
    public int h, g;
    public int f {
      get { return h+g; }
    }
}
```

Zmienne g,h i f (listing 12.) to zmienne oznaczające koszty przejścia pomiędzy węzłami. 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* (listing 12.) 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 [8,9] 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.

LISTING 13. Warunkowe odtworzenie poprawnej ścieżki.

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

Algorytm odtwarza ścieżkę tylko gdy zostanie ona odnaleziona w innym wypadku 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.

LISTING 14. Sprawdzenie czy ścieżka istnieje – algorytm A*.

```
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* (listing 14.) wykonuje się trzon algorytmu A* (listing 14.). *FindCheapest* przeszukuje wszystkie pola otwarte i zwraca pole, którego f (g+h) jest najmniejsze. Cała logika związana ze zbieraniem, wycenianiem i zapamiętywaniem węzłów ukryta jest w metodzie *AddPossibleSteps* (listing 15.) oraz w metodach jej zależnych.

LISTING 15. Przeszukiwanie sąsiednich pól.

```
  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*.

LISTING 16. Warunek dodania pola do listy węzłów otwartych.

```
    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);
            }
        }
    }
```

W listingu 16. algorytm aktualizuje koszty przejścia i dodaje pole tylko w wypadku gdy nie ma go w liście otwartych węzłów i można po nim się poruszać.

LISTING 17. Aktualizacja kosztów i wyceniana przejścia.

```
    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;
    }
```

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 jeżeli zostanie ona znaleziona. Służy do tego metoda *RetracePath*:

LISTING 18. Odtworzenie wyznaczonej ścieżki.

```
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;
}
```

Ścieżka zostaje zapisana do zmiennej *path* i zwracany jest ostatni element ścieżki (listing 18.). Tylko następny krok jest potrzebny komponentowi sterującemu. Algorytm w celu optymalizacji działania wyszukuje ścieżkę tylko jeżeli nie może podążać wcześniej już wytyczoną.

LISTING 19. Publiczny interfejs algorytmu A*.

```
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();
}
```

W listingu 19. wyrażona jest następująca logika: jeżeli ścieżka istnieje i można nią dotrzeć do celu, zwróć jej następny element. W każdym innym wypadku wyznacz nową.

Słowem podsumowania – algorytm A* stosowany jest zwykle jako jeden wśród kilku algorytmów SI w branży gier komputerowych. W tym projekcie jednak został zaimplementowany jako osobny algorytm sztucznej inteligencji, ponieważ celem gry jest zdobycie jak największej ilości pożywienia, a by tego dokonać trzeba znaleźć do niego jednoznaczną i najkrótszą drogę.

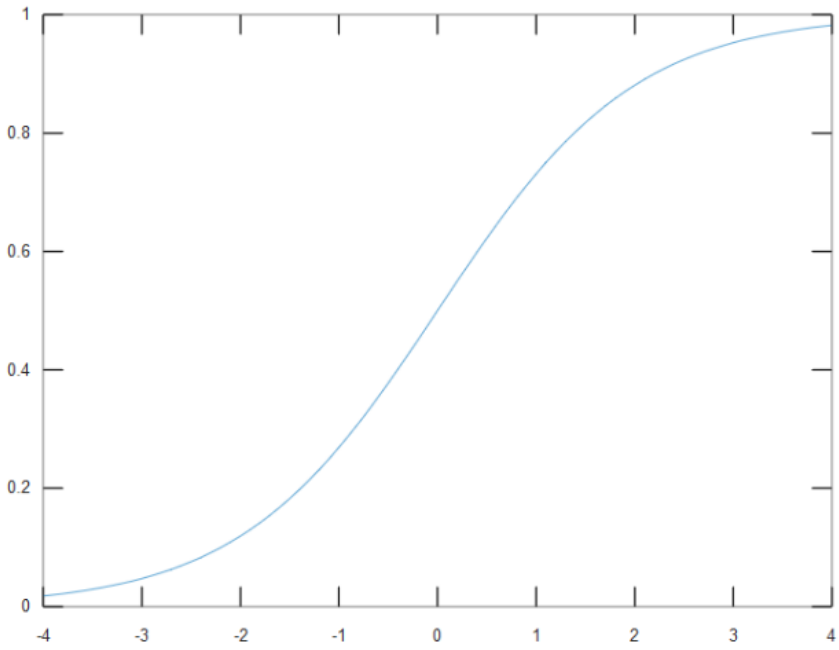
3.5. Sieci Neuronowe – Wielowarstwowy perceptron

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 w projekcie wykorzystano tylko unipolarne sigmoidalne funkcje aktywacji.

Unipolarna funkcja sigmoidalna definiuje się wzorem:

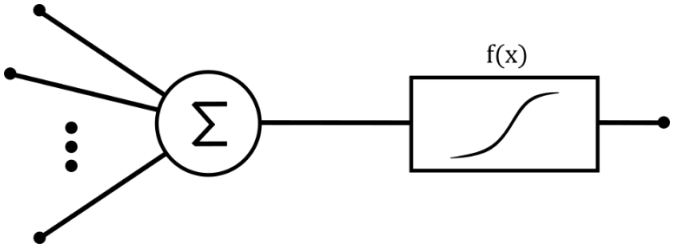
$$f(x) = \frac{1}{1+e^{-\alpha x}},$$

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. Wykres został wykonany w oprogramowaniu Octave Online z niewielką pomocą dokumentacji programu Matlab [10,11].



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 informacje do napisania tego rozdziału zostały zaczerpnięte z książki pt. „Inteligentna sieć algorytmy przyszłości" oraz zajęć i materiałów doń dr inż. Piotra Ciskowskiego [1,12].

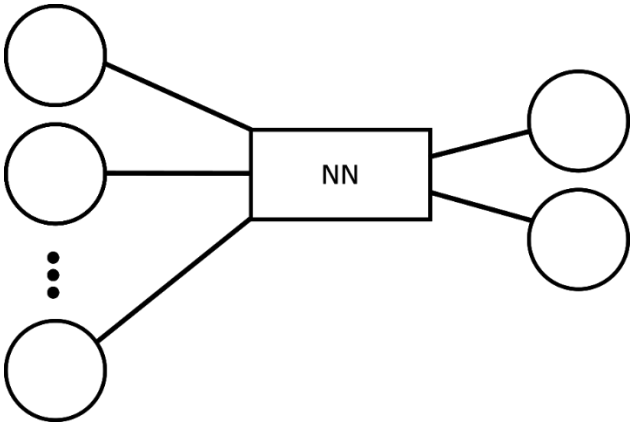
Istnieje wiele różnych typów sieci. Można wyznaczyć kilka szczególnych klas tych obiektów – sieci jednokierunkowe, sieci rekurencyjne (z pamięcią), sieci samoorganizujące, sieci radialne. Temat tych struktur poruszany jest już dość długo bo już od lat 50 ubiegłego stulecia. W latach 60 pojawił się pierwszy model perceptronu. Amerykański psycholog Frank Roseblatt był jego twórcą. Przeprowadzał on szereg eksperymentów w dziedzinie działania mózgu oraz pamięci. Po swoim sławnym odkryciu m.in. 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 [14].

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 SSN w celu próby przewidzenia wahań na giełdzie 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 unipolarnymi sigmoidalnymi funkcjami aktywacji, uczenie, zapisanie i wczytanie go z pliku. 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 na wejście dostanie całą mape oraz w postaci przykładów 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. Brak jakichkolwiek zauważalnych efektów spowodował szybką zmianę podejścia do problemu. W następnej próbie uproszczono model danych wejściowych, do sześciu elementów: pięć pól mapy naokoło głowy i informacja w postaci binarnej czy wąż 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. Na podstawie zdobytego doświadczenia podjęto decyzję by rozwinąć ten pomysł. Wymieniono informację binarną na zestaw 2 wektorów dwuwymiarowych – przesunięcie wektorowe głowy węża od pożywienia oraz wersor z informacją po której osi wąż się porusza. Niestety to podejście nigdy w pełni nie zadziałało pomimo wielokrotnych prób. Pierwsza implementacja odbyła się w samym Unity. Szybko się okazało, że moc obliczeniowa jest niewystarczająca. 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, więc wprowadzono losowość w tworzeniu konfiguracji sieci: różna liczba warstw i neuronów w tych warstwach. Do zoptymalizowania zastosowano prosty algorytm genetyczny. Żeby działał sprawnie używał przetwarzania wielowątkowego. Ostatnim podejściem wykonanym w stronę próby nauczenia

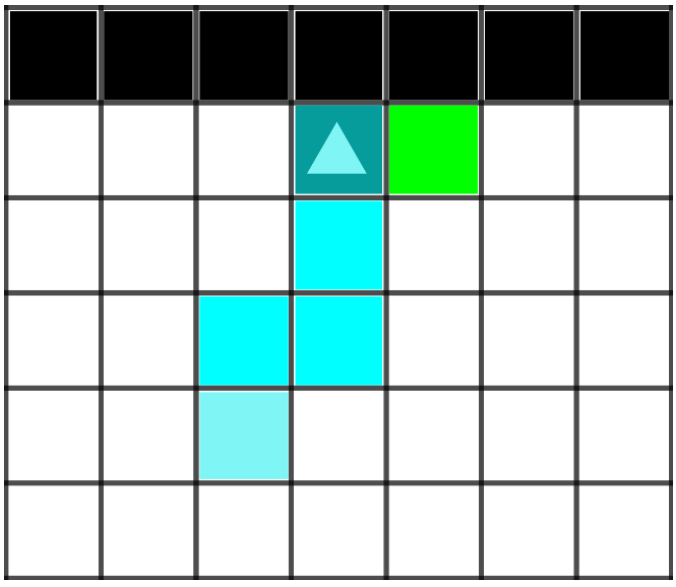
takiej struktury sieci neuronowej, było uczenie nadzorowane algorytmem drzewa behawioralnego. Niestety zabrakło czasu na wdrożenie tej iteracji.



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 typu wyliczeniowego *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, pole z pożywieniem,
- 0.0, pole puste,
- 1.0, pole ze ścianą,
- 2.0, 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. Pole z jedzeniem staje się wówczas 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 skorzystano, z gotowej biblioteki do obliczeń macierzowych stworzoną przez Ivan'a Kuckir'a, udostępnioną na licencji MIT [13].

LISTING 20. Implementacja klasy *NeuralNetwork*.

```
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 (*Matrix*) wag odpowiednich połączeń neuronowych w każdej warstwie (listing 20.). Inicjalizacja klasy polega na stworzeniu zadanej ilości macierzy z losowymi wartościami wag z przedziału (-1,1).

LISTING 21. Realizacja uczenia 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 (21.) 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, *Test* przetestować odpowiedź sieci i stwierdzić czy udzieliła ona poprawnej odpowiedzi po wywołaniu metody uczącej.

LISTING 22. 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 z listingu 22. polega na szacowaniu błędu (*Find Error*) każdej warstwy na podstawie iloczynu pochodnej funkcji aktywacji i różnicy sygnału propagowanego wstecz (*delta*). Po znalezieniu błędu należy dostosować wagi w sieci neuronowej z określoną siłą – przemnożenie błędu przez małą stałą wartość szybkości uczenia (*learningRate*).

Podsumowując, algorytm SSN jest algorytmem, który podczas swojego działania potrafi polepszać swoje umiejętności rozwiązywania danego problemu. Obliczenia dokonywane w algorytmie warto przetrzymywać w macierzach. By stworzyć sieć o wielu warstwach należy zastosować algorytm wstecznej propagacji błędu, który będzie modyfikować wagi w warstwach ukrytych.

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#. 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.

Stworzenie tego projektu nauczyło mnie tego, że powinno się dobierać stosowne narzędzia i rozwiązania w zależności od problemu. Zastosowanie wielowarstwowego perceptronu jako algorytmu sztucznej inteligencji wydawało mi się dobrym pomysłem na początku. W praktyce okazało się, że była to znacznie gorsza implementacja algorytmu dokładnego. Praca nad projektem również poprawiła moje zdolności tworzenia oprogramowania. Wymagała ona pisania jasnego i przejrzystego kodu źródłowego.

Bibliografia:

Literatura:

[1] McIlwraith, D., Marmanis, H. i Babenko, D. *Inteligentna sieć algorytmy przyszłości* Wydanie II, Helion, Polska, 2017 s. 153 - 175

Źródła internetowe:

[2] Dokumentacja Unity, https://docs.unity3d.com/Manual/index.html, 21.11.2018

[3] Amazonas, M., *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

[4] Artykuł na temat istoty silnika gry, https://unity3d.com/what-is-a-game-engine, 30.11.2018

[5] Dokumentacja .Net Core, https://docs.microsoft.com/en-us/dotnet/core/index, 08.01.2018

[6] Wagner, B., Wenzel, M., Latham, L., Onderka, P. i inni, *A Tour of the C# language*, https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/, 08.10.2016

[7] Dokumentacja programu Git, https://git-scm.com/docs, 30.11.2018

[8] Red Blob Games, *Introduction to A** https://www.redblobgames.com/pathfinding/a-star/introduction.html, 26.05.2014

[9] A* pseudokod, http://mat.uab.cat/~alseda/MasterOpt/AStar-Algorithm.pdf, 30.11.2018

[10] Dokumentacja programu Matlab, https://uk.mathworks.com/help/matlab/, 30.11.2018

[11] Program Octave Online, https://octave-online.net/, 30.11.2018

[12] Ciskowski, P., Materiały do zajęć, http://piotr.ciskowski.staff.iiar.pwr.wroc.pl/dydaktyka/materialySN.htm#laboratorium, 30.11.2018

[13] Kuckir, I., Biblioteka Matrix, http://blog.ivank.net/lightweight-matrix-class-in-c-strassen-algorithm-lu-decomposition.html, 2017

[14] Rossenblatt, F., Farrow, J. T., Rhine, T. S., *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

Spis Listingów