AI4Vid_Project

Decision Making with Markov State Machines.

Environment:

The project has been realized using Unity and C#, with the IDE Visual Studio.
All the assets used come from the default 2D and 3D Objects proposed by the Unity Editor.

Summary of the game's rules:

In a 3D environment, 2 groups of agents, controlled by AI, have to fight each other.
Each group has a base where agents can get healed up.

Summary of the game's AI:

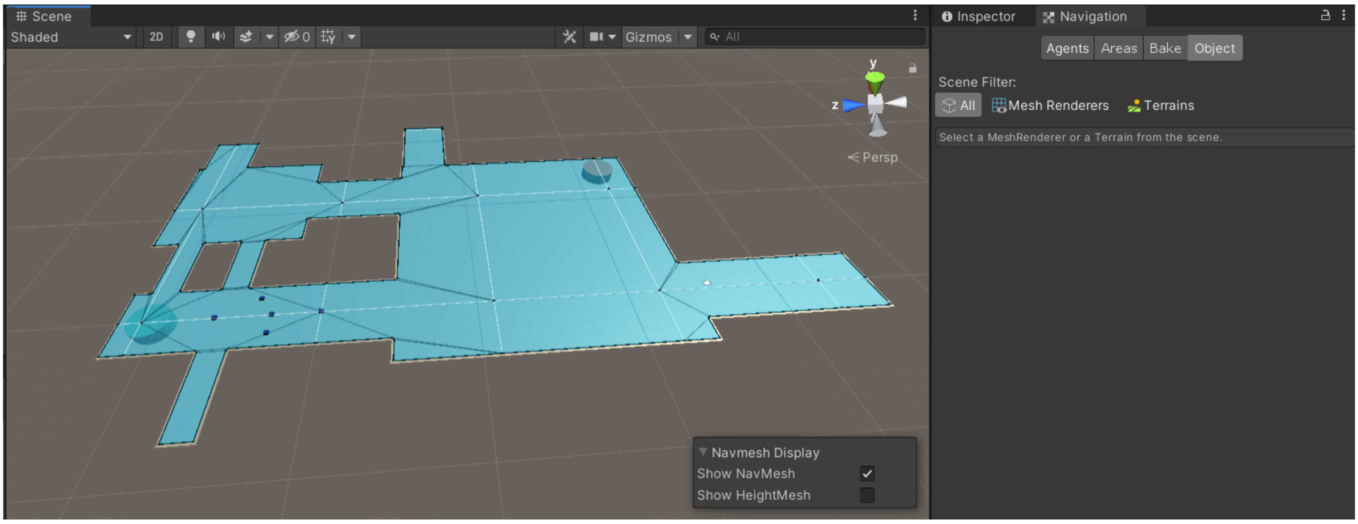
The agents will be able to move to a certain destination and avoid objects.
This will be achieved using Navigation Meshes.

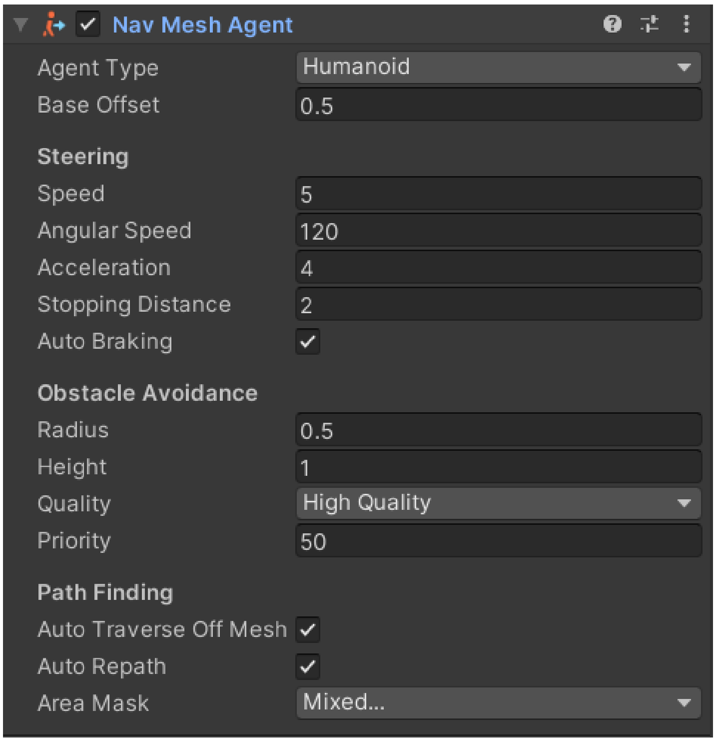
To enrich the behavior, agents will be able to:

- **Collaborate** and **Coordinate** with other agents of the same group
- **Feel Emotions**
    - This will be achieved with 5 emotions (InRage, Brave, Normal, Shy, Scared).
    - Each emotion will be represented as values, grouped in an array called State Vector in the **Markov State Machines**. The values in the array will change when a transition of the Markov State Machine is fired.
    - The emotion that the agent is feeling will affect the behavior of the agent:
        - **FSM** with a state for each emotion
            - A **DT** for each **FSM** state defining a unique tree to create the illusion that the agent is feeling that emotion
                - A **BT** to define the more complex actions.

Pathfinding

Since this is not a project where the main goal is to perform or analyze pathfing, i've solved this problem with Navigation Meshes, build on the map, and Nav Mesh Agent to make the agent move to a particular destination and to make it avoids obstacles and to make it stay on the surface of the map.





The project is built around the goal to make the agent feel emotions.

The emotions that the agent can "feel" are 5:
- In Rage
- Brave
- Normal
- Shy
- Scared

The project can be divided in 2 parts:
1. How to shifts between the emotions that the agent is feeling
2. How the emotion that the agent is feeling affects its behavior

1 - Emotions System

In order to shifts between the 5 emotions, it has been implemented a Markov State Machine as proposed in the book at chapter 5.6.

1.1 - Markov State Machine

A **Markov State Machine** is composed by:

- **A single State**: characterized by a vector, called State Vector of size ***"n"***

- **Transitions**: characterized by:
    - **Condition:** a condition needs to be triggered in order to fire the transition.
    - **Transition's Matrix:** when a transition is fired, the squared matrix of size *"n x n"* associated to the transition, is multiplied with the stateVector.
    - **Actions**: When a transition is fired, the actions, associated to the transition, are executed.

  The transition will not cause the State Machine, since there is only one, but it will cause a change in the vector associated with the state.

1.2 - Markov State Machine Implementation

It has been implemented as a Library MarkovSM, following the library FSM proposed in class with the necessary modification.

1.2.1 - Markov SM Condition

```
// Defer function to trigger activation condition
// Returns true when transition can fire
public delegate bool MarkovSMCondition();
```

1.2.2 - Markov SM Action

```
// Defer function to perform action
public delegate void MarkovSMAction();
```

1.2.3 - Markov SM Transition

```
public class MarkovSMTransition
{
    // The method to evaluate if the transition is ready to fire
    public MarkovSMCondition myCondition;

    // Matrix
    public float[][] myMatrix;

    // A list of actions to perform when this transition fires
    public List<MarkovSMAction> myActions = new List<MarkovSMAction>();

    7 riferimenti
    public MarkovSMTransition(MarkovSMCondition condition, float[][] matrix, MarkovSMAction[] actions = null)
    {
        myMatrix = matrix;
        myCondition = condition;
        if (actions != null) myActions.AddRange(actions);
    }

    // Call all  actions
    1 riferimento
    public void Fire()
    {
        foreach (MarkovSMAction action in myActions) action();
    }
}
```

When instantiating a transition, we need to define the condition that, when true will fire the transition, the matrix that will be multiplied with the state vector and the actions that will be executed when the transition is fired.

1.2.4 - Markov SM State

When instantiated, we need to define the starting vector.
The method VerifyTransitions, checks if one of the conditions of the transitions associated with the state is triggered, in that case the transition is fired.

```
public class MarkovSMState
{
    // A dictionary of transitions and the states they are leading to
    private List<MarkovSMTransition> links;

    public float[] myStateVector;

    1 riferimento
    public MarkovSMState(float[] stateVector)
    {
        links = new List<MarkovSMTransition>();
        myStateVector = new float[5];
        myStateVector = stateVector;
    }

    7 riferimenti
    public void AddTransition(MarkovSMTransition transition)
    {
        links.Add(transition);
    }

    1 riferimento
    public MarkovSMTransition VerifyTransitions()
    {
        foreach (MarkovSMTransition t in links) {
            if (t.myCondition()) return t;
        }
        return null;
    }
}
```

1.2.5 - Markov SM

The Markov SM class requires the MarkovSM State when instantiated and will operate the multiplication, via the method Multiply, with the matrix of the transition that is fired and with the state vector of the state.

```
public class MarkovSM
{
    public MarkovSMState current; // Current state

    1 riferimento
    public MarkovSM(MarkovSMState state)
    {
        current = state;
    }

    1 riferimento
    public void Update()
    { // NOTE: this is NOT a MonoBehaviour
        MarkovSMTransition transition = current.VerifyTransitions();
        if (transition != null) {
            // (1) Multiply the matrix of that transition with the state vector
            current.myStateVector = Multiply(transition.myMatrix, current.myStateVector);

            // (2) Execute the actions associated with the transition
            transition.Fire();
        }
    }

    1 riferimento
    public float[] Multiply(float[][] matrix, float[] _stateVector)
    {
        float[] vector = new float[5];
        for (int i = 0; i < 5; i++) {
            for (int j = 0; j < 5; j++) {
                vector[i] += matrix[i][j] * _stateVector[j];
            }
        }
        return vector;
    }
}
```

1.3 - EmotionsSystem

The library it's used in the monobehaviour Emotions_Sytem

1.3.1 - Markov SM State, State Vector

Each emotion is stored inside the stateVector as a single float value.
The vector is passed as a parameter at the Markov SM State creation.

```
private float[] stateVector;
```

```
void Start()
{
    stateVector = new float[]{
        0.0f,
        0.0f,
        1.0f,
        0.0f,
        0.0f
    };

    MarkovSMState markovSMState = new MarkovSMState(stateVector);
```

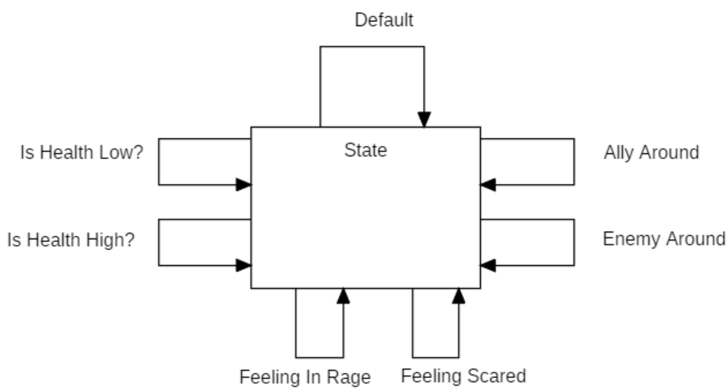
1.3.2 - Markov SM Transitions

Each transition needs:
- Matrix
- Condition

and can execute some Actions.

The script also requires Health, the script in charge of controlling the health value of the agent, and DecisionMaker component since some of the functions are used as conditions in EmotionsSystem script.

There are 7 transitions:



1. Default

As suggested in the book there is a default transition that fires when no transition has been fired for some time, in order to reset the default value of the state vector.

```
// Default transition
MarkovSMTransition m_t_default = new MarkovSMTransition(TimerOff, defaultMatrix);
m_t_default.myActions.Add(m_a_Default);
markovSMState.AddTransition(m_t_default);
```

a. Default Matrix:

```
1 riferimento
public void TransitionMatrixDefaultInit()
{
    // Declare a jagged array.
    defaultMatrix = new float[5][];

    defaultMatrix[0] = new float[5] { 0.1f, 0.1f, 0.1f, 0.1f, 0.1f };
    defaultMatrix[1] = new float[5] { 0.1f, 0.1f, 0.2f, 0.1f, 0.1f };
    defaultMatrix[2] = new float[5] { 0.1f, 0.5f, 1.0f, 0.5f, 0.1f };
    defaultMatrix[3] = new float[5] { 0.1f, 0.1f, 0.2f, 0.1f, 0.1f };
    defaultMatrix[4] = new float[5] { 0.1f, 0.1f, 0.1f, 0.1f, 0.1f };
}
```

b. Condition

```
public bool TimerOff()
{
    if (Time.time >= startTime + timer) {
        return true;
    }
    return false;
}
```

- timer is a public float attribute
- start time is a private float attribute

```
private float startTime;
public float timer = 5f;
```

c. Action

The only action executed by the transition is ResetTimer, to reset the starting time of the timer. This action is present in all the transitions, for the reason expressed before.

```
public void ResetTimer()
{
    startTime = Time.time;
}
```

2. Ally Around

When there are at least 2 allies around, the transition is fired, increasing the value of Brave.

```
MarkovSMTransition m_t1 = new MarkovSMTransition(AllyAround, braveMatrix);
m_t1.myActions.Add(m_a_Default);
markovSMState.AddTransition(m_t1);
```

```
public void TransitionMatrixBraveInit()
{
    braveMatrix = new float[5][];

    braveMatrix[0] = new float[5] { 1.1f, 0.1f, 0.0f, 0.0f, 0.0f };
    braveMatrix[1] = new float[5] { 0.1f, 1.3f, 0.1f, 0.0f, 0.0f };
    braveMatrix[2] = new float[5] { 0.0f, 0.0f, 0.8f, 0.0f, 0.0f };
    braveMatrix[3] = new float[5] { 0.0f, 0.0f, 0.0f, 0.7f, 0.0f };
    braveMatrix[4] = new float[5] { 0.0f, 0.0f, 0.0f, 0.0f, 0.6f };
}
```

```
public bool AllyAround()
{
    int n = 0;

    float sightRange = GetComponent<DecisionMaker>().sightRange;

    // Otherwise i can calculate Angle with every target /in range
    GameObject[] allies = GameObject.FindGameObjectsWithTag(tag);

    foreach (GameObject ally in allies) {
        //Check distance
        if (Vector3.Distance(transform.position, ally.transform.position) < sightRange) {
            n++;
        }
    }

    if (n >= 3)
        return true;
    return false;
}
```

3. Enemy Around

When there are at least 2 enemies around, the transition is fired, increasing the value of shy.

```
MarkovSMTransition m_t2 = new MarkovSMTransition(EnemyAround, shyMatrix);
m_t2.myActions.Add(m_a_Default);
markovSMState.AddTransition(m_t2);
```

```
public void TransitionMatrixShyInit()
{
    shyMatrix = new float[5][];

    shyMatrix[0] = new float[5] { 0.6f, 0.0f, 0.0f, 0.0f, 0.0f };
    shyMatrix[1] = new float[5] { 0.0f, 0.7f, 0.0f, 0.0f, 0.0f };
    shyMatrix[2] = new float[5] { 0.0f, 0.0f, 0.8f, 0.0f, 0.0f };
    shyMatrix[3] = new float[5] { 0.0f, 0.0f, 0.1f, 1.3f, 0.1f };
    shyMatrix[4] = new float[5] { 0.0f, 0.0f, 0.0f, 0.1f, 1.1f };
}
```

```
public bool EnemyAround()
{
    int n = 0;
    string enemyTag = GetComponent<DecisionMaker>().enemyTag;
    float sightRange = GetComponent<DecisionMaker>().sightRange;

    // Otherwise i can calculate Angle with every target /in range
    GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag);

    foreach (GameObject enemy in enemies) {
        //Check distance
        if (Vector3.Distance(transform.position, enemy.transform.position) < sightRange) {
            n++;
        }
    }

    if (n >= 3)
        return true;
    return false;
}
```

4. Is Health Low

When the health of the agent is low, the transition fires, increasing the value of Scared.

```
MarkovSMTransition m_t3 = new MarkovSMTransition(GetComponent<Health>().IsHealthLow, scaredMatrix);
m_t3.myActions.Add(m_a_Default);
markovSMState.AddTransition(m_t3);
```

```
public void TransitionMatrixScaredInit()
{
    scaredMatrix = new float[5][];

    scaredMatrix[0] = new float[5] { 0.6f, 0.0f, 0.0f, 0.0f, 0.0f };
    scaredMatrix[1] = new float[5] { 0.0f, 0.7f, 0.0f, 0.0f, 0.0f };
    scaredMatrix[2] = new float[5] { 0.0f, 0.0f, 0.8f, 0.0f, 0.0f };
    scaredMatrix[3] = new float[5] { 0.0f, 0.0f, 0.1f, 1.3f, 0.1f };
    scaredMatrix[4] = new float[5] { 0.0f, 0.0f, 0.0f, 0.1f, 1.1f };
}
```

The function IsHealthLow() is inside the script Health

```
public bool IsHealthLow()
{
    if (this.health <= healthLow)
        return true;
    return false;
}
```

5. Is Health High

When the health of the agent is high, the transition fires, increasing the value of InRage.

```
MarkovSMTransition m_t4 = new MarkovSMTransition(GetComponent<Health>().IsHealthHigh, inRageMatrix);
m_t4.myActions.Add(m_a_Default);
markovSMState.AddTransition(m_t4);
```

```
public void TransitionMatrixRageInit()
{
    inRageMatrix = new float[5][];

    inRageMatrix[0] = new float[5] { 1.3f, 0.1f, 0.1f, 0.0f, 0.0f };
    inRageMatrix[1] = new float[5] { 0.0f, 1.1f, 0.1f, 0.0f, 0.0f };
    inRageMatrix[2] = new float[5] { 0.0f, 0.0f, 0.8f, 0.0f, 0.0f };
    inRageMatrix[3] = new float[5] { 0.0f, 0.0f, 0.0f, 0.7f, 0.0f };
    inRageMatrix[4] = new float[5] { 0.0f, 0.0f, 0.0f, 0.0f, 0.6f };
}
```

The function IsHealthHigh() is inside the script Health

```
public bool IsHealthHigh()
{
    if (this.health >= healthHigh)
        return true;
    return false;
}
```

6. - 7. Feeling In Rage and Feeling Scared

Feeling In Rage and Feeling Scared are 2 special transitions.
They are fired when a flag, a boolean variable for Feeling in rage is inRage and for Feeling Scared is scared. These flags are activated by the script Decision Maker: when a nearby agent is feeling InRage or Scared, it spreads this strong emotion to the allies around it, and so it turns true the flags of the allies.
This behavior has the goal to introduce some elements of coordination between agents through emotions.

```
// Feeling inRage
MarkovSMTransition m_t_FinRage = new MarkovSMTransition(FeelingInRage, inRageMatrix);
m_t_FinRage.myActions.Add(m_a_Default);
m_t_FinRage.myActions.Add(m_a_ResetInRage);
markovSMState.AddTransition(m_t_FinRage);

// Feeling scared
MarkovSMTransition m_t_Fscared = new MarkovSMTransition(FeelingScared, scaredMatrix);
m_t_Fscared.myActions.Add(m_a_Default);
m_t_Fscared.myActions.Add(m_a_ResetScared);
markovSMState.AddTransition(m_t_Fscared);
```

Each transition, after it is fired, reset the flag associated with it.

Once the state and its transitions are correctly initialized, the coroutine can start, as any other FSM.

```
    markovSM = new MarkovSM(markovSMState);

    StartCoroutine(Patrol());
}

Messaggio Unity | 0 riferimenti
void Update()
{

}

1 riferimento
public IEnumerator Patrol()
{
    while (true) {
        markovSM.Update();
        yield return new WaitForSeconds(reactionTime);
    }
}
```

Other functions present in the script are the one related with understanding which emotion the agent is feeling. These functions will be called by the Decision Maker script when it needs to understand which emotion is the agent feeling in order to mutate its behavior.

Taking in example the evaluation of the emotion InRage:

```
#region Emotions Evaluation
2 riferimenti
public bool InRage()
{
    if (ToPercentage(Total(stateVector), stateVector[0]) >= UnityEngine.Random.Range(0f, 100f))
        return true;
    return false;
}
```

```
public float ToPercentage(float tot, float a)
{
    float x = (100 * a)/tot;
    return x;
}

5 riferimenti
public float Total(float[] vec)
{
    float total = 0f;

    for(int i = 0; i < 5; i++)
    {
        float a = vec[i];
        total += a;
    }

    return total;
}
```

The function InRage() verifies if the agent is feeling InRage, it compares the value of that emotion, reworked as a percentage with the total value of the vector, and a random number form 0 - 100, if that random number is lower than the emotion value made as a percentage, it returns true.

2 - Decision Maker

This script has the goal to change the behavior of the agent reflecting which emotion it is feeling.

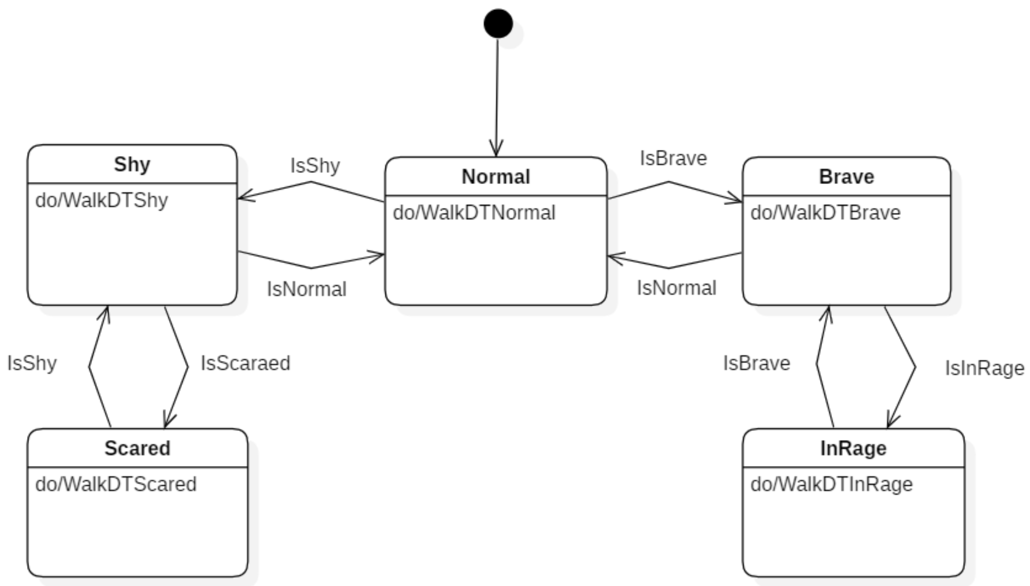
This component is divided in 3 main levels:

1. **Finite State Machine**
   In charge of understanding which emotion is the agent feeling, by consulting the script seen before
2. **Decision Tree**
   In charge of defining a behavior in line with the emotion that the agent is feeling
3. **Behavior Tree**
   In charge of defining how an action is performed

In the project are included all the libraries of FSM, DT and BT seen in class.

2.1 - FSM

The purpose of the FSM is to query the EmotionsSystem to understand which emotion is the agent feeling and switch between states.
Each state is identified by the name of the emotion and each has a Stay Action that executes the DT associated with the state.

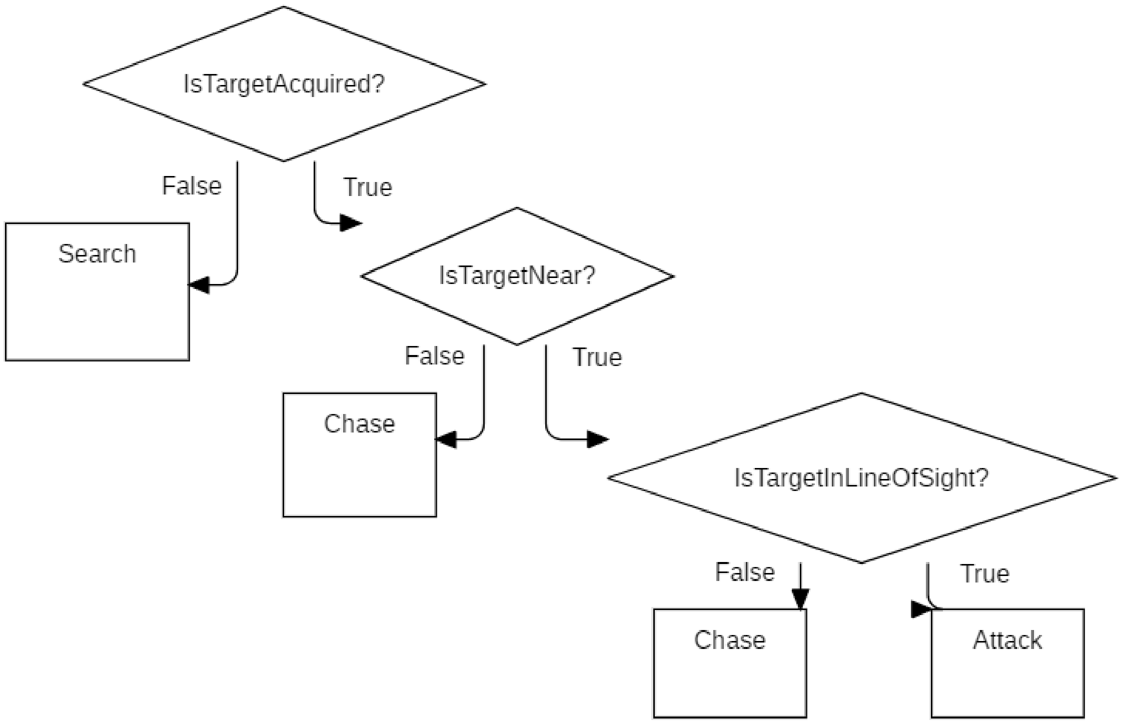


2.2 - DT

There is a unique Decision Tree for each FSM State.

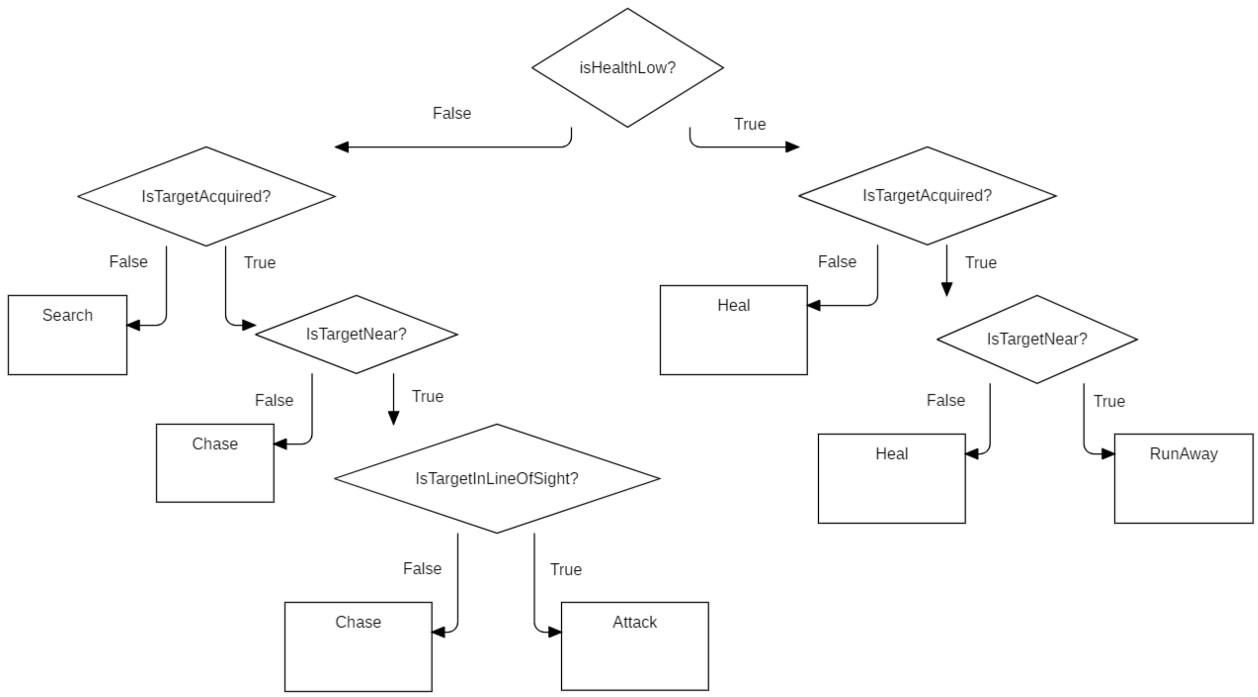
2.2.1 - DT InRage

To better create the illusion that the agent is feeling a strong emotion as been In Rage, it does not take in consideration the agent's health and only looks, chases or attacks the target.

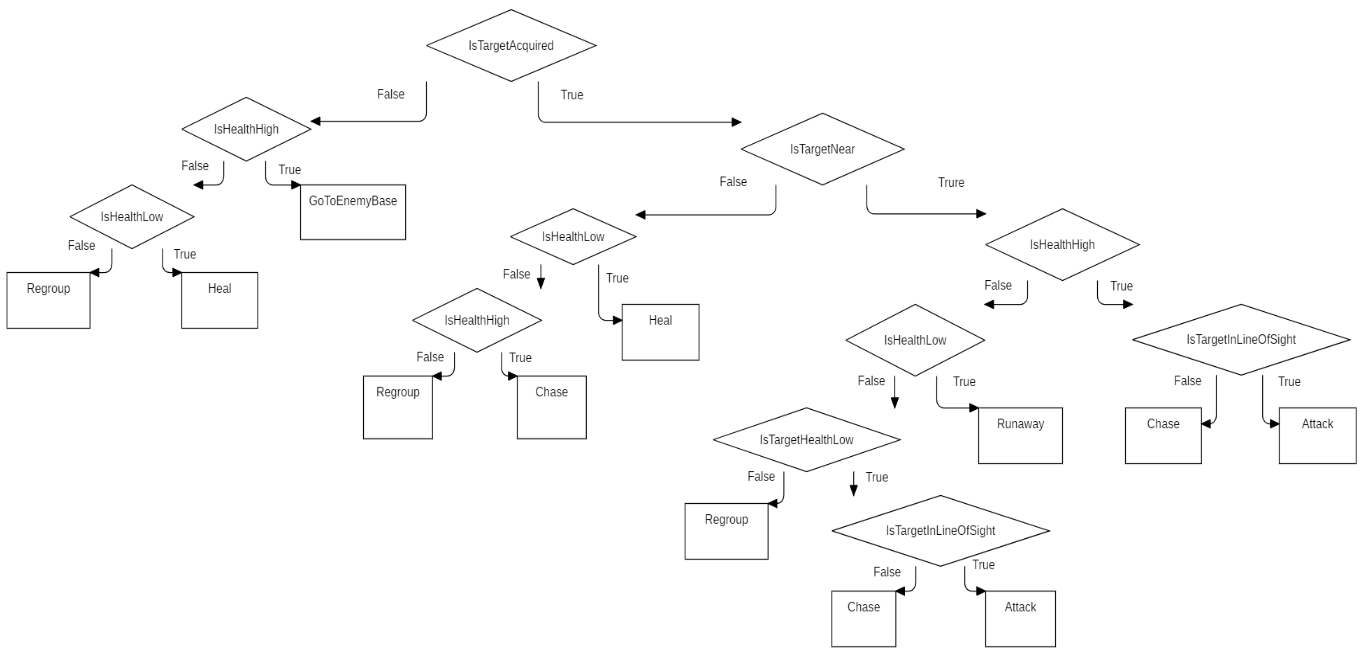


2.2.2 - DT Brave

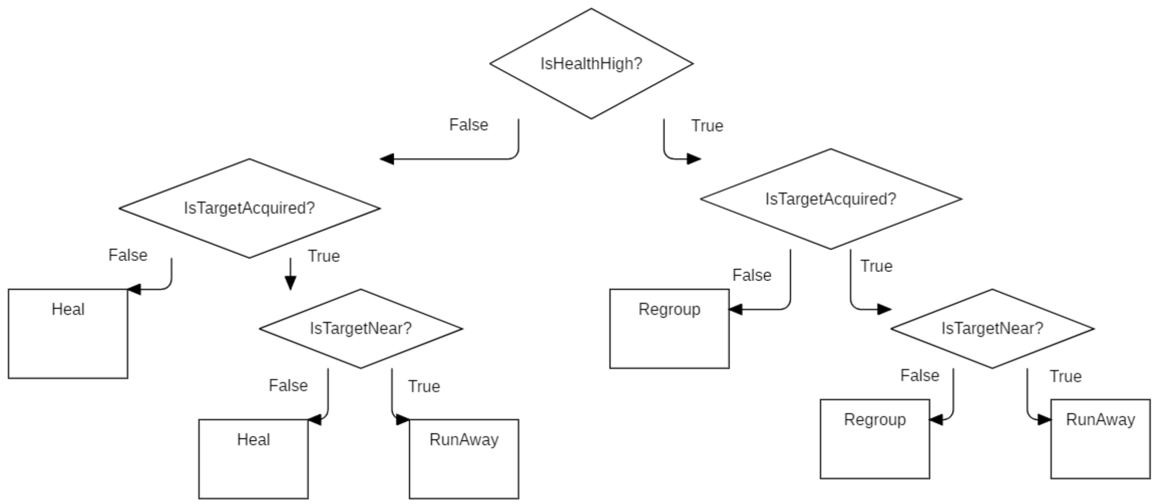
To better create the illusion that the agent is feeling Brave, it does take in consideration the agent's health but only make that it is not too low,



2.2.3 - DT Normal



2.2.4 - DT Shy



2.2.5 - DT Scared