A picture containing tool

Description automatically generatedGommers Tijl

How do UtilityAI compare to GOAP for NPCs in a soulslike battle game in their setup and adding of npc actions.

Supervisor: Boury Fries

Coach: Geeroms Kasper

Graduation Work 2024-2025

Digital Arts and Entertainment

Howest.be

A close up of a card

Description automatically generated

Contents

[Abstract & Key words 5](#_Toc187328019)

[Preface 6](#_Toc187328020)

[List of Figures 7](#_Toc187328021)

[Introduction 8](#_Toc187328022)

[Literature Study / Theoretical Framework 9](#_Toc187328023)

[1. Decision-making algortithm 9](#_Toc187328024)

[2. Blackboard 10](#_Toc187328025)

[3. Utility AI 10](#_Toc187328026)

[4. Goal Oriented Action Planner(GOAP) 11](#_Toc187328027)

[Research 14](#_Toc187328028)

[1. Experiment setup 14](#_Toc187328029)

[1.1. Setup time 14](#_Toc187328030)

[1.2. Time to implement a new action in the system 14](#_Toc187328031)

[1.3. behavior in game 14](#_Toc187328032)

[2. How to get this data 15](#_Toc187328033)

[2.1. setup time 15](#_Toc187328034)

[2.2 action implementation 15](#_Toc187328035)

[2.3 behavior in game 15](#_Toc187328036)

[3. Software used 16](#_Toc187328037)

[3.1. unreal engine 5 16](#_Toc187328038)

[3.1.1 SoulsCharacter 16](#_Toc187328039)

[3.1.2 BattleActions 17](#_Toc187328040)

[3.1.3 AI Controller 18](#_Toc187328041)

[3.1.4 Blackboard 18](#_Toc187328042)

[4. AI Behavior base 19](#_Toc187328043)

[5. Implement Utility ai 19](#_Toc187328044)

[6.1 SETUP 19](#_Toc187328045)

[6.1.1 State 19](#_Toc187328046)

[6.1.2 METHODS 19](#_Toc187328047)

[ Calculate score 19](#_Toc187328048)

[ Execute 20](#_Toc187328049)

[6.2 Making Actions 20](#_Toc187328050)

[6.2.1 Move Back and forth 20](#_Toc187328051)

[6.2.2 Move Sideways 21](#_Toc187328052)

[6.2.3 Throw 21](#_Toc187328053)

[6.2.4 Heal 21](#_Toc187328054)

[6.2.5 Block 21](#_Toc187328055)

[6.2.6 Quick attack 22](#_Toc187328056)

[6.2.7 Hard Attack 22](#_Toc187328057)

[7 Implement GOAP 22](#_Toc187328058)

[7.1 SETUP 22](#_Toc187328059)

[7.1.1 State 22](#_Toc187328060)

[7.1.2 Execute 23](#_Toc187328061)

[7.2 Goals 23](#_Toc187328062)

[7.2.1 Attack opponent 23](#_Toc187328063)

[7.2.2 Dodging 24](#_Toc187328064)

[7.2.3 Heal 24](#_Toc187328065)

[7.2.4 More Attacks 25](#_Toc187328066)

[7.2.5 Block 25](#_Toc187328067)

[7.2.6 StopBlock 25](#_Toc187328068)

[7.2.7 Throw 25](#_Toc187328069)

[8 Implement Player experiment 26](#_Toc187328070)

[8.1.1 experiment 26](#_Toc187328071)

[9 Results 26](#_Toc187328072)

[9.1 Setup 26](#_Toc187328073)

[9.2 Actions 28](#_Toc187328074)

[9.2.1 Movement 28](#_Toc187328075)

[9.2.2 Throw 28](#_Toc187328076)

[9.2.3 Heal 29](#_Toc187328077)

[9.2.4 Block 29](#_Toc187328078)

[9.2.5 Quick Attack 29](#_Toc187328079)

[9.2.6 Hard Attack 30](#_Toc187328080)

[9.2.7 Summary 30](#_Toc187328081)

[9.3 Testers 31](#_Toc187328082)

[9.3.1 Wins-Losses 31](#_Toc187328083)

[9.3.2 Stamina 31](#_Toc187328084)

[Discussion 33](#_Toc187328085)

[Conclusion 34](#_Toc187328086)

[Future work 35](#_Toc187328087)

[Critical Reflection 36](#_Toc187328088)

[References 37](#_Toc187328089)

[Acknowledgements 38](#_Toc187328090)

[Appendices 39](#_Toc187328091)

# Abstract & Key words

Abstract:

This research explores the comparative effectiveness of Utility AI and Goal-Oriented Action Planning (GOAP) in managing Non-Player Character (NPC) behaviors within a soulslike battle game environment.

Utility AI, which selects actions based on dynamically calculated utility scores, offers a straightforward setup but can become complex with an increasing number of actions. This approach has been widely used in various game genres, from strategy games to simulations, due to its ability to make informed decisions based on the current game state.

In contrast, GOAP, which involves defining goals and planning sequences of actions to achieve them, presents a more intricate initial setup but excels in flexibility and dynamic adaptability. GOAP has been instrumental in creating more dynamic and responsive NPC behaviors in games, allowing characters to evaluate their current state and choose actions that best align with their goals.

This study evaluates both approaches in terms of setup complexity, ease of adding new NPC actions, flexibility, performance, and long-term maintenance. The findings suggest that while Utility AI may be more performant in simpler scenarios, GOAP's structured planning and adaptability make it more suitable for the complex, dynamic nature of soulslike games. This research contributes to the understanding of AI methodologies in game development, providing insights that can enhance NPC behavior and overall game experience.

Keywords:

Game development, AI, GOAP, UtilityAI, Unreal Enginge, c++, soulslike games.

# Preface

I started this research out of personal interest in NPC behavior in games.

After a previous research on UtilityAI, I was curious how this compares to others, for example Finite State Machine, Behavior Tree or Goal-Oriented Action Planning.

I was still unfamiliar with how Goal-Oriented Action Planning worked but heard good things about it, this motivated me to learn GOAP and compare it to each other so I knew what to use and when in future projects.

# List of Figures

[Figure 1 Finite State Machine 9](#_Toc187342580)

[Figure 2 Behavior Tree 9](#_Toc187342581)

[Figure 3 blackboard 10](#_Toc187342582)

[Figure 4 curves 11](#_Toc187342583)

[Figure 6 action implementation time 30](#_Toc187342584)

[Figure 7 Win/loss results 31](#_Toc187342585)

[Figure 8 average stamina UtilityAI 31](#_Toc187342586)

[Figure 9 average stamina GOAP 32](#_Toc187342587)

# Introduction

In this research paper, we aim to compare two decision-making algorithms, Utility AI and GOAP, in a soulsLike game to identify their respective pros and cons.

soulsLike Game:

These games are known to be more difficult then others, where the player has to approach carefully and look for openings and keep an eye on their stamina and health. Attacks can not be cancelled, which is why it leaves you vulnerable if you time it wrong.

I want to find out which system can handle this while having the actions:

* Quick Attack: fast but weak attack, average stamina cost.
* Hard Attack: slow but devastating, is strong against block, high stamina cost.
* Block: Put received damage to stamina instead of health, slows down movement drastically, no stamina cost to execute.
* Heal: Takes long time to execute, heals halve of its health when completed, high stamina cost.
* Throw: Send an object flying to the opponent, low damage, high stamina cost.

UtilityAI:

Decision-making forms the core of any AI system. There are many different approaches to decision-making. One of the most robust and powerful systems we’ve encountered is a utility-based system. The general concept of a utility-based system is that every possible action is scored at once and one of the top scoring actions is chosen. By itself, this is a very simple and straightforward approach[1].

GOAP:

By planning in real-time, we can simulate the affect of various factors on reasoning, and adapt behavior to correspond. With First Encounter Assault Recon (F.E.A.R.), we demonstrated that planning in real-time is a practical solution for the current generation of games. Moving forward, planning looks to be a promising solution for modeling group and social behaviors, increasing characters’ command of language, and designing cooperative characters[2].

This brings us to the following research question: *How can each of them be used in this soulslike battle game, how do they compare to each other on setup, implementing each action, their reaction to the actions and their maintainability?*

*Gameplay-wise, how are each in their management of stamina and reactions to the player. Comparing the time put in the actions in both systems, which one pays off the most.*

The following hypotheses were formulated:

* Null hypothesis (H0): There are no remarkable differences, they both work just as well and as easily implemented as the other.
* Working hypothesis 1 (H1): UtilityAI is easiest to setup, but hard to maintain after more actions gets implemented. GOAP will be much easier to maintain and ends up more stable at the end.

# Literature Study / Theoretical Framework

## Decision-making algortithm

An npc in games that attacks you, runs away or is goofing around are all different kinds of behaviors he can do. What behavior he will do is decided by his decision-making algorithm.

What they all have in common is that they use the world state (game environment) to decide on their next behavior. A commonly known way to get this state values is by the use of a Blackboard, which will be explained in the next chapter.[9]

e.g. You are patrolling an area (behavior = patrol), when you suddenly notice an enemy (state enemyFound = true), so you run to the enemy (behavior = GoToEnemy), when you reach him you attack (state enemyInRange = true , behavior = AttackEnemy).

Most popular form of AI in modern computer games [10] are the Finite State Machine(FSM) or the Behavior Tree(BT), in this research we will be talking about UtilityAI and Goal Oriented Action Planner(GOAP) and find how they differ in selecting new behavior.

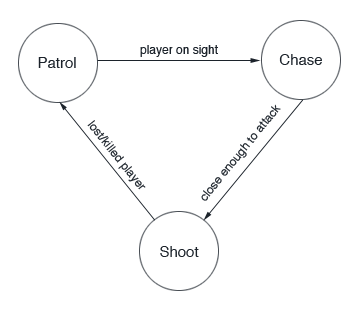


Figure 1 Finite State Machine

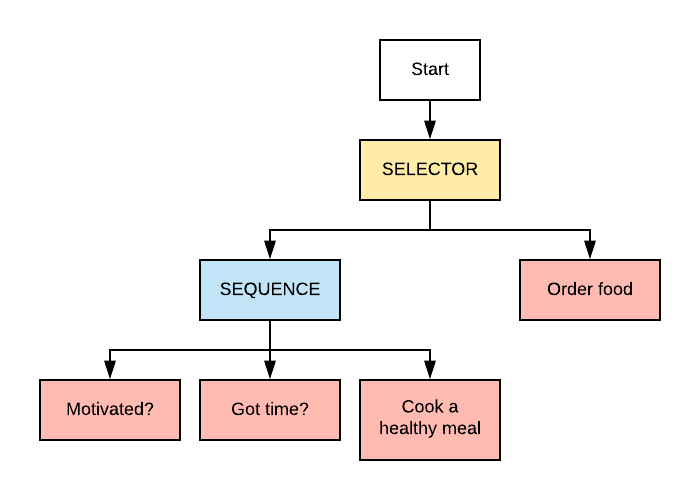


Figure 2 Behavior Tree

## Blackboard

The blackboard architecture is a flexible and powerful expert system framework. It represents a general approach to problem solving that is useful in many domains of applications especially in the area of intelligent control. The blackboard architecture can provide an environment for achieving intelligent control behaviors in many AI systems[4].

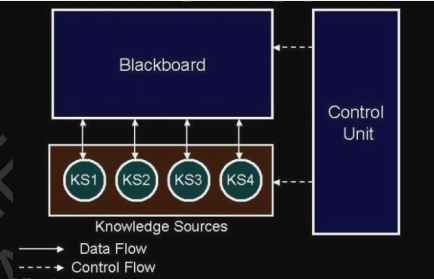


Figure 3 blackboard

Knowledge Sources: these are independent agents in the game, e.g. player character, door of a building, plants in the scene,…

The Control unit is the agent that owns the blackboard.

When an agent his state changes he notifies the blackboard, when the decision-making algorithm of other agents need that value he gets it from the blackboard instead of trying to look for connection with that previous agent and get his value.

## Utility AI

* **What is UtilityAI**

Utility Based AI is a method of assigning scores to actions  based on some heuristic, usually involving [Curves](https://grail.com.pl/documentation/documentation/1.2.0/manual/utility/curve.html) and Weights. Then algorithm analyzes those scores and chooses the most suitable [Behavior](https://grail.com.pl/documentation/documentation/1.2.0/manual/general/behavior.html).[3]

When calculating utility scores, it’s important to be consistent. Because utility scores are compared to each other to come up with a final decision, they must all be on the same scale across the entire system. As you’ll see later in this article, scores are often combined in meaningful ways to produce other scores. Therefore, using normalized scores (values that go from 0–1) provide a reasonable starting point[1].

* **Curves**:

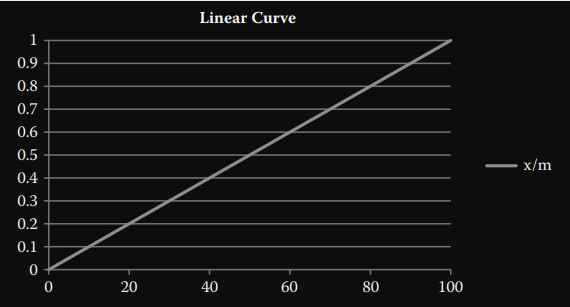
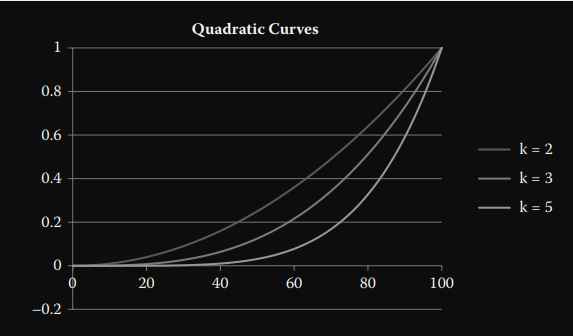
 

Figure 4 curves

The curves used by the algorithm determine the score (Y-value) the action gets compared to its world state (X-value).

The Y-axis is normalized but the X-axis can be whatever is needed.

E.g. score calculated on the distance to the opponent.

When the distance is lower then 50, he is close enough to hit Y = 1. When the distance is over 350, he is to far to matter Y = 0.

* **Weights**:

Several curves can be used to calculate the score of 1 action, but not both curves are equally important.

To adjust this, we can give weight to the curves.

The weights are a number that we multiply the Y value from the curve, and important is that we keep the total weight of all curves.

To calculate the score of this action is then by adding all the weighted Y values and then divide by the total weight.

Score = (0.5\*2 + 0.8 \*0.5)/2.5

* **Summary**

Every possible [Behavior](https://grail.com.pl/documentation/documentation/1.2.0/manual/general/behavior.html) get one or more curves with their perspective weight assigned to it. On every update the score of this behavior is calculated. The behavior with the highest score gets executed.

## Goal Oriented Action Planner(GOAP)

* **What is GOAP**

Goal oriented action planning is an artificial intelligence system for agents that allows them to plan a sequence of actions to satisfy a particular goal. The particular sequence of actions depends not only on the goal but also on the current state of the world and the agent. This means that if the same goal is supplied for different agents or world states, you can get a completely different sequence of actions., which makes the AI more dynamic and realistic[5].

The most important parts of this algorithm are Goals, Actions, Planner and the WorldState.

* **Goals**

Goal is the endpoint the AI wants to achieve, this is where the plan of actions is build upon. E.g. KillPlayer, KeepHealthUp

The way the goal is selected is by priority, the goals are ranked by priority in a list and the first valid one will be selected.

The is valid state means if the goal is worth to be pursued [6].

* **WorldState**

WorldState is the state of the game, the collection of all the important worldvariables needed for the algorithm.

The current WorldState is a representation of how the game currently is, this can be a list of bools or enums.

Every goal has a desired WorldState, this means that for the goal to be achieved the current WorldState has to be equal to the desired WorldState.

* **Action**

An action is something that the agent does. Usually it is just playing an animation and a sound, and changing a little bit of state (for instance, adding firewood). Opening a door is a different action (and animation) than picking up a pencil. An action is encapsulated, and should not have to worry about what the other actions are.

To help GOAP determine what actions we want to use, each action is given a cost. A high cost action will not be chosen over a lower cost action. When we sequence the actions together, we add up the costs and then choose the sequence with the lowest cost.[5]

Actions have a Satisfy WorldState and can have a Desired WorldState just like the Goals.

* **Planner**

The planner will start from the selected goal and will look at his desired worldState.

For every state in this collection that has to be achieved, he will look for an action to achieve this.

The planner will go over the entire list of actions, if he finds an action that satisfies the desired state, he stores the cost, looks if that action has Dsired WorldState as well and looks further until no more desired state necessary.

The planner uses A\* pathfinding to get the cheapest plan to reach all Desired WorldState.

* **A\***

The *A (A-star) pathfinding algorithm*\* is a popular and efficient method used to find the shortest path between two points on a graph, such as a map. It combines the strengths of Dijkstra's algorithm and a heuristic approach to optimize the search process[7].

* **Summary**

GOAP will create a plan based on the first most important goal that he needs to achieve, he makes this plan by selecting all the actions needed to satisfy all the Desired WorldState using the A\* pathfinding to ensure the most proficient one.

# Research

## Experiment setup

What data do we need to find out which system is most useful for what type of project.

### Setup time

How long does it take to setup each system?

🡪This is good to know for when a developer has a big or small project in mind. With only a small one, it is not interesting to use a system with a big setup time where it will not even be used to its full strength.

### Time to implement a new action in the system

Is it a straightforward approach to make it do the action at the time you want it to?

When you add a new action to the list, will it disrupt the previous action their execution?

### behavior in game

We set up 2 in-game rules for both algorithms to follow.

The AI has to win from the player.

* + - Dodge/block when player attacks.
    - Attack when player is close.
    - Disrupt player.

The AI has to keep his stamina above 0.4.

Stamina is chosen because it is really important in soulslike games. Without stamina you become a sitting duck. We chose 0.4 because when lower than this, blocking has big chance to fail and healing wont be possible.

Will the fully implemented system act as we programmed it to do?

🡪When you implement the system to act in a certain way, it is important that this will happen. E.g. When set it that the AI will heal itself, you don’t want him to rush at the target to attack.

## How to get this data

### setup time

We will gather this information by creating our own implementation of both systems in a soulslike combat game we designed for it.

🡪Log the time it takes to have a working implementation on each of them. We start with UtillityAI then GOAP.

Input variables: The research and paper prototype, only start logging when know what to do.

Output variables: The logged time.

### action implementation

Continue on previous implementation, we will create signals for specific actions to be executed by the character.

For every type of action the character can do, there has to be a at least 1 signal of them each.

This signal has to only be send at the by us designed situation.

🡪We log the time to create every signal.

🡪After every new signal is added, test if previous still work as before. If not, redo last created signal or adjust previous on.

Input variables: The strategy we want both AI’s to follow when fighting his opponent.

Output variables: The logged time.

### behavior in game

* We set the rule that the AI has to attack player while keeping his stamina usage in check.

This will be done during the creation of the signals.

We choose for stamina because this is the thing he needs for everything. If he fights with no regards for it, runs empty, he loses the ability to execute the correct action on critical times.

E.g. Heal when his health is low and far away. Disrupt the opponent healing by throwing a rock or attack him.

* The AI has to be able to win from the player.

Even if he keeps an eye on his stamina, if he cant win it wont matter.

🡪To test the 2 above behaviors, we will let both implementations be tested by a human player.

We will log both the stamina from the AI and the player and the wins/loses.

By having the stamina of the player in comparison with the AI, we can see how the player gets pressured by the AI and also to confirm that the player his output will be similar to both AIs.

We will ask 15 people. Age 15 – 40. Experienced with soulslike games as testers.

The age is chosen for their combination of reflexes and experience in games.[8].

Younger is not chosen because of their lack of patience needed for soulslike games to keep stamina in check.

Every tester will play 3 minutes against UtillityAI implementation, then 3 minutes against the GOAP.

Input variables: The game will equal in all ways for both the AI and player.

Output variables: The logged stamina of both systems and the player, the logged wins/loses.

## Software used

### unreal engine 5

Started with the 3theperson template. Made a new GameMode and controller for the player. Developed 5 BattleActions representing the 5 actions that both the player and AI need to be able to do. Made sure there is a constant *Lock on* to each other.

Create the AIController that can execute both algorithms.

Create the blackboard, his keys and update this every tick.

### 3.1.1 SoulsCharacter

This is the character which will be controlled by the player and AI, they will both have the exact same stats and actions.

State:

* 5 BattleActions:
* MovementComponent (inherited from AActor)
* HealthComponent
* StaminaComponent
* KnockbackComponent
* A TRingBuffer of BattleActionBase: This is the queue of actions that are selected to execute. It pops from the front and gets added in the back. The popped action gets executed. An action will be automatically removed after a set amount of time.

When an action gets executed the player is unable to execute another action, a timer will start that will enable the player again when finished.

When character gets stunned from a hit. The current action and the action Queue gets reset and will be unable to activate a new action for as long as the stun endures.

### BattleActions

BattleActions are all derived from the class BattleActionBase.

State:

* Cost: the cost of stamina to execute this BattleAction.
* Damage: the amount it subtracts from the character he hits.
* ExecutionTime: this is the time it takes to finish the BattleAction, while it is active the characterMovement is disabled.
* KnockTime: the amount of time the hit character will be stunned and receives a knockback. The knockback itself is handled by the KnockbackComponent of the player.

Methods:

* Execute: This is the animation of the action.
* DeActivate: This is an event triggered from the baseClass to stop executing the animation.
* CollisionHandling: This is responsible for the reaction of getting hit.

When he hits the character body different things can happen

* + Was character already stunned? Nothing will happen then.
  + Was character blocking? If this was true, the damage will be dealt to the staminaBar instead of the healthBar. No knockback will be executed.
  + Was the stamina amount to low to block? Then the stamina will be zero and he will receive knockback and health damage.
  + Was there no block? Then character will receive the knockback and damage to his health.

While BattleActionBase is in c++, the BattleActions themselves are blueprints, this is where they get their own specific state and execution, set in the editor.

For there to be meaning to the action/reaction of the AI, there need to be some kind of rock/paper/scissor to these BattleActions.

E.g.

Block is perfect against QuickAttack but weak against HardAttack while HardAttack is easily countered by quickAttack.

Throw is weak and costly but perfect when the opponent is far away and starts to heal.

### AI Controller

Called Npc\_Controller.

State:

* AIBehaviourBase m\_SelectedBehaviourSystem: Thid is the selected Decision-making algorithm, the brain of the Character.
* Blackboard: The only place where it gets created and updated.
* ASoulsCharacter called m\_NpcRefrence: Reference to its controlled character

Methods:

On Tick, This will execute the *SelectedBehaviourSystem* , returning an EAction enum. A switch case will handle this enum to the concrete action that needs to be executed.

EAction

* Idle ,
* WalkForward ,
* WalkSideways ,
* WalkBackwards,
* Block ,
* StopBlock ,
* Heal,
* Throw ,
* QuickAttack,
* HardAttack

From then on it will work same way as for the player, e.g. pressing forward arrow equals the reaction in code as when the enum WalkForwards is returned.

The time an action is allowed to stay in the queue is for the AI (0.2s) shorter then for the player(2.0s).

### Blackboard

State:

* String Key
* Float value

Methods:

* AddBlackboardValue
* UpdateKey
* GetKeyValue

This is created in the AIController and updated every tick there.

Keys used in both algorithms:

* IsEnemyBlocking : returns 1 when opponent is using Block.
* EnemyDistance : returns distance to opponent.
* Stamina : returns stamina
* Health : returns health.
* EnemyHealth : returns opponent health.
* IsEnemyHealing : returns 1 if opponent is using Heal.
* IsEnemyThrowing : returns 1 if opponent is using Throw.
* IsEnemyAttacking : returns 1 if opponent is using any type of attack.
* IsEnemyQuickAttack : returns 1 if opponent is using QuickAttack.
* IsEnemyFrontalAttack : returns 1 if opponent is using HardAttack or Throw.
* RandomFloat : returns a random float between 0 -1.
* IsBlocking : returns 1 when is using Block.
* IsHealing : returns 1 when is using Heal.

## AI Behavior base

AIBehaviorBase is the base class of the Decision-making algorithms, holds reference to the blackboard and the virtual method Execute().

## Implement Utility ai

### SETUP

### State

List of ActionScores.

* + ActionType: the EAction return value.
  + List of Scores
    - Weight
    - Curve.
    - BlackboardKey: used as the X value in the curve.

### METHODS

### Calculate score

For every ActionScore his every score, his score is calculated as previously explained in LITERATURE STUDY 3.

### Execute

Local state:

* + - Float highestScore = 0.5. Startvalue.
    - EAction selectedAction = Idle.

Function:

The highest scoring action received from the CalculateScore methode his ActionType gets used to return to the AIController.

If no Action scores higher then 0.5, Idle will be the chosen action. this is to eliminate scores when none were actually valid.

### Making Actions

We created 1 ActionScore for each EActionType, every tick it will go over all of them and decide what input to give to the AIController.

We logged the time to create them, and explain the step by step procedure on how I got the final result of our UtilityAI.

### Move Back and forth

The 2 ActionScores I started with where the MoveBackWards and MoveForwards. This was to start with giving the AI the correct distance to the player at all time.

He now moves back and forwards to keep his distance between 150 and 200.

Backward to dodge:

We added a graph to the MoveBackWards ActionScore that holds opponent IsQuickAttacking. When this value gets one, The returned score has to be higher then his forward score to make sure he dodges.

Making sure this happened took a lot of iterating, writing down situations to compare results when choosing different weights on the graphs.

### Move Sideways

Main use : Dodging the opponent HardAttack.

Graphs using BlackboardKeys:

* EnemyDistance: No need to move sideways if not in range of attack.

Weight: 1.0.

* IsEnemyFrontalAttack: Only necessary when opponent is using this.

Weight : 2.0.

* RandomFloat : For at times when he is idle in front of the player, he would move sideways from time to time.

Weight : 0.5.

### Throw

Graphs using BlackboardKeys:

* EnemyDistance: No need to throw when in attack reach.

Weight: 1.0.

* IsEnemyHealing: Only necessary when opponent is using Heal.

Weight : 2.0.

### Heal

Graphs using BlackboardKeys:

* EnemyDistance: Need to be on safe distance.

Weight: 1.0.

* Health: Only necessary when health is below 0.5.

Weight : 1.5.

### Block

Graphs using BlackboardKeys:

* EnemyDistance: Only when in reach.

Weight: 1.0.

* Stamina: Only use with stamina above 0.8.

Weight : 2.0.

* IsEnemyQuickAttack : Only use to block Quick attack. Hard attack will deal great damage to stamina during block.

Weight: 3.0

### Quick attack

Graphs using BlackboardKeys:

* EnemyDistance: Only when in reach.

Weight: 1.0.

* Stamina: To make sure he keeps stamina high.

Weight : 1.0.

* RandomFloat: To make sure he switches between Hard and Quick Attack.

Weight: 2.0.

### Hard Attack

Graphs using BlackboardKeys:

* EnemyDistance: Only when in reach.

Weight: 1.0.

* Stamina: To make sure he keeps stamina high.

Weight : 1.0.

* RandomFloat: To make sure he switches between Hard and Quick Attack.

Weight: 2.0.

* IsEnemyBlocking: Hard Attack is strong against block so score needs to be higher when opponent is blocking.

## Implement GOAP

### SETUP

### State

* WorldState CurrentWorldState: The current state of the game, updated by the blackboard, used for the planning.
* WorldState ComparedWorldState: Used to get the comparison between the current Goal/Action Desired State and the CurrentWorldState.
* m\_AllGOAPActions : List of all Actions.
* m\_AllGOAPGoals : List of all Goals.
* m\_CurrentPlan: The current list of Actions to be executed by the AIController. Output from the planner.
* m\_CurrentGoal: Current selected Goal where the planning is made for.
* M\_CurrentAction: The activated Action, popped from the m\_CurrentPlan. This is being executed.

### Execute

On execute, the algorithm will start with checking if there is already a Goal selected and if it is still valid. When not valid anymore he will quit and empty the m\_CurrentPlan.

if there is a plan. This is by seeing if the m\_CurrentPlan which is an TArray of GOAPAction is empty.

When empty

* Will select the first valid goal.
* Create a plan by looking at the goal his DesiredWorldState. For every Desired State
  + Compare the State with CurrentWorldState to see if some state is already satisfied.
  + Loop over all actions available in the algoritm and compare their SatisfyWorldState with the DesiredWorldState that still needs to be satisfied. When more actions are found he will pick the one with the lowest score.
    - Score: cost + heuristic. Cost is set on the action. Heuristic is the amount of state to be satisfied on that same action.
  + This will happen recursively for when the chosen action has DesiredState as well.
* The planner will make the shortest plan for each desiredState, state by state. E.g. When DesiredState is DistanceIsClose and NeedHighStamina. He will first make the plan for Distance, add it to the m\_CurrentPlan. Then make the plan for HighStamina and add that to back of m\_CurrentPlan.

When there is a plan, the m\_CurrentAction will be what is popped from the back of the m\_CurrentGoal. This will not happen if there is already an active m\_CurrentAction.

This action will be updated and returns the EAction input for the AIController. When the action is valid, this will be set to nullptr.

### Goals

### Attack opponent

The GOAPGoal *BP\_GGKillPlayer* is the first goal created, thus the only goal in current list of goals for the GOAP planner to select, and is valid unless the player his Health <= 0.0.

The DesiredWorldState of this Goal is m\_GoalState = Fight. All Goal WorldState has only 1 DesiredState which is the top one called m\_GoalState.

There is only 1 GOAPAction that has a SatisfyWorldState equal to that. *BP\_GAQuickAttack* . This action returns the EAction::QuickAttack and has 2 DesiredWorldState

* M\_DistanceToOpponentState
* M\_ConditionState : AI his health/stamina condition.

The AI start with full health and stamina so this get satisfied by the currentWorldState.

For the distance he needs to look for an action that satisfies this. BP\_GAForward returns EAction::Forward and satisfies this state, since it is the onle action that satisfies this and he has no desiredState by himself. The plan is created.

M\_CurrentPlan = {. *BP\_GAQuickAttack* , BP\_GAForward }.

The algorithm will pop BP\_GAForward, update it and returns EAction::Forward so the AI will move toward the player this tick.

This Action has no update, The action his IsVallid method will return true when the distance is < 160. This will be checked by comparing it with the blackboard.

### Dodging

Created the BP\_GGRetreat goal, This goal is valid when the AI is within a distance of 300 to the player. The general goal of this is to create distance from the player or dodge his attacks.

m\_GoalState = Dodge. This state can be satisfied by

* *BP\_GABackward* . has cost = 1.0 and heuristic = 0.0. score = 1.0
* *BP\_GALeft* . has cost 0.5 and heuristic = 1.0. score = 1.5

The heuristic from BP\_GALeft is from his DesiredState m\_OpponentActionState = HardAttack.

When the opponent is using hardAttack, the currentWorldState put this to satisfied, making the heuristic score = 0.0. What makes the total score = 0.5 + 0.0.

🡪Conclusion: When opponent is using his HardAttack . He will take BP\_GALeft in the plan, else he will take BP\_GABackward.

### Heal

Created the BP\_GGHealthManagement goal. This goal is valid when the AI his health gets lower then 0.4. The general goal is to put himself to safe distance and heal.

m\_GoalState = Heal. This state can be satisfied by

* BP\_GAHeal. EAction::heal. Has 1 DesiredWorldState m\_DistanceToOpponent = HealDistance.
  + BP\_GABackwardToHeal.this goal will be satisfied when distance > 650.

M\_CurrentPlan = { BP\_GAHeal, BP\_GABackwardToHeal}.

### More Attacks

Starting from the BP\_GGKillPlayer from earlier, I added more attackoptions.

* BP\_GATaunt: returns EAction::Idle, Desired WorldState m\_OpponentDistanceState = close.

Will go to face the opponent to taunt him to attack.

* BP\_GACombo: returns EAction::HardAttack, Desired WorldState m\_ActionState = QuickAttack.

Will go to player and execute a quick attack directly followed by a HardAttack.

The way this goal selects is by taking the one with the lowest score. The heauristic is equal with BP\_GATaunt and BP\_GACombo. Their cost is a random number between 0.5 and 3. They will get a new value after every new plan. BP\_GAQuickAttack his cost stay at 1.0.

### Block

On the BP\_GGRetreat goal, it will dodge attacks when he is in range of player. So to have him dodge the throwing rocks, we added a isVallid state to the goal *isOpponentThrowing* . Now this goal will still be valid when distance>650.

BP\_GABlock: cost = 0.25, heuristic = 1.0. returns EAction::Block.

When opponent is using throw, heuristic will become 0.0, making this the cheapest action for the plan.

### StopBlock

When he starts a block he wont stop blocking unless he starts an attack, tis is very annoying because he moves really slowly when blocking.

I decided to create a new goal BP\_GGStopBlock. Te only thing this does is stop blocking when this goal is valid. His priority is just after BP\_GGRetreat and before the BP\_GGKillPlayer.

### Throw

When the opponent decides to heal himself when he somehow was able to get far away from the AI, we need to throw a rock at him.

BP\_GGInteruptHealing: this goal has the highest priority, its only purpose is to throw a rock when the opponent starts healing when The AI wasn’t able to catch him in his attacks.

## Implement Player experiment

### experiment

We put in a system that logs the stamina from the player and the AI every tick as a csv file.

After every death, in the GameMode it will keep track of every win and loss and log it after the 3 min fight csv file.

When attaining the results, the 4 stamina values will be put on graphs stamina on time. We will make an average of all 15 graphs.

After receiving the files from every tester, we will make an average of the stamina usage of both systems and put those of the player in the same graph.

We will also take the average of the wins/losses.

Compare the 2 averaged graphs of AI/Player with their win/Loss.

## Results

### Setup

UtilityAI:

When we created this implementation, we had a lot of trouble converting this code correctly to unreal editor. This caused us double the time it took to only create the algorithm. Since this had nothing to do with the algoritm itself, the time spend on it will be deducted from the total set up time.

Total time = 9 hours 2 minutes.

Implementation time = **4 hours 22minutes.**

See full logs in appendix 1.

GOAP:

It took a week of research and paper prototyping to start this implementation because GOAP’s setup is a lot more complex than the previous one.

We started by creating the basic classes of Goals, Actions, WorldState, and the GOAPBehavior itself. Then we began on improving each of them and start with creating the A\* planner. For this planner we needed a Calculate Heuristic method. Since the AI doesn’t need to create large complex plans, we decided to have the heuristic return the amount of desired state the action has.

It was also important that before he calculates this heuristic, it should look first at the current WorldState. Once it does that, we started on making the planner.

The planner would go over every action, when he finds one that satisfies the desired state, he calculate total score (action cost + heuristic), stores this and continue looking through every action, looking for a lower scoring one. When he finished going through, recursively will do the same for that lowest scoring action if that action has a desired state.

For this WorldState we had some trouble figuring out how to do this best, we ended up giving every action a desired state and a Satisfing state. So it wil look for every desired state to an action which has that same state in his satisfying state. In the planner we use a Compared state which holds the current WorldState in account.

It is now able to create a planning with the actions we created for the goals we created as well.

Afterwards we made more methods to check if current goal remains valid (if not -> Terminate plan) and methods for checking when the current action is finished.

🡪The setup is now fully ready to be used to create plans for their first valid goal.

Total time: **16hours 55min.**

See full logs in appendix 2.

### Actions

### Movement

* UtilityAI:

We created 3 graphs for backwards movement and 2 for forward movement. His dodge failed regularly so we worked more on the graphs and their assigned weights.

Total time**: 6hours 8 min.**

We created graphs so he would move sideways on random moment and/when opponent is using HardAttack.

Total time: **13min.** No need to work long on their graphs. On moments when he was idle before he will now move sideways depending on the random number or when he needs to dodge.

See full logs in appendix 3.a.

* + - GOAP:

We created the BP\_GGRetreat Goal with his desired state, this will be valid when the AI is in reach off the player and will then move backward. We needed to create the move backward action with his desired and satisfying state.

We created the Move Forward action with his desired and satisfying state for the Attack Goal.

Total time:  **1hour 31min**.

See full logs in appendix 4.a.

### Throw

* + - UtilityAI:

We Made the AI throw when his opponent uses heal at a distance. Some of this time was used to reform the Throw BattleAction.

Total time: **3hours 13min.**

See full logs in appendix 3.b.

* GOAP:

We created the BP\_GGInteruptHealing goal with his WorldState and the action Throw with both his WorldStates. When his opponent starts to heal, he will create a plan to throw a rock at him.

Total time: **17min**.

See full logs in appendix 4.b.

### Heal

* + - UtilityAI:

When the AI has to heal he first need to get at a safe distance. So first we had to readjust the Backward/Forward Movement ActionScores. We did this by adding the graph dealing with his health.

Total time: **1 hour 28min**.

See full logs in appendix 3.c.

* + - GOAP:

We created the BP\_GGHealthManagment goal with his WorldState and the Heal Action. This goal has the highest priority. When the AI his health gets below 0.4, this goal becomes valid and will make plan to walk backwards and heal once in healDistance. This goal will become invalid when it takes longer then 3 seconds to complete.

Total time: **33min.**

See full logs in appendix 4.c.

### Block

* + - UtilityAI:

When AI has full stamina, it will be safer to block the quickAttack instead of dodging. It also keeps him closer for a counterattack.

Total time: **1hour 56min**. No need to change the other graphs but we had to be sure the block will only be called at when is being attacked and when he has the stamina and when he is in reach of opponent.

See full logs in appendix 3.d.

* + - GOAP:

In the BP\_GGRetreat goal we created earlier, we made a new option for a plan to create. We created the Block Action with both his WorldStates.

This took longer then expected because I changed my mind several times on how to use block and the way to stop blocking.

Total time: **1 hour** **03min.**

See full logs in appendix 4.d.

### Quick Attack

* UtilityAI:

We made the AI use QuickAttack when he is in range, has the stamina and when a random number is above 0.9.

The range of random number is 0-1, we choose +0.9 so that he will not endlessly use that attack but leaves it open for idle or HardAttack. Making him less predictable.

Total time: **29min**.

See full logs in appendix 3.e.

* GOAP:

We created the BP\_GGKillPlayer goal with the action QuickAttack, it uses the Action move forward in the planner.

Total time: **15min.**

See full logs in appendix 4.e.

### Hard Attack

* UtilityAI:

HardAttack shines the best when he is smashing through a block of his opponent. So we focused this action his use on that moment.

Total time**: 1hour 1min**. We want it to focus mostly as a reaction to opponent block, but he also has to use it randomly as attack but lot less as the QuickAttack because the HardAttack will leave him open for counterattack.

See full logs in appendix 3.f.

* GOAP:

We added more actions to be used for the BP\_GGKillPlayer goal; Combo Action, HardAttack action and Taunt.

Total time: **59min.**

See full logs in appendix 4.f.

### Summary

* UtilityAI:

All of his actions are in a delicate balance with each other, if you change the weight on one, it can cause the entire balance to break.

This makes adding new Actions more and more difficult unless it is pointed at a moment when he is still Idle.

Especially the back/forward movement is very sensitive, we realized that his distance to the player started to become incorrect when his health gets lower.

Setup time: **4 hours 22minutes.**

Actions time: **14 hours 28min**.

* GOAP:

The setup takes some time but it gets easier to farther you get. There are many ways on how you can create this and I know I can still keep working to improve it. But for this research it achieves what he needs to be able to do.

For new goals/actions, it only got easier the more I started to create. I can keep adding new actions without fear for it to break.

The only thing that can cause trouble in the long run is its fixed priority for the goals. It makes it easy to ensure the algorithm will do what is necessary, but sometimes a priority can change depending on the situation.

Setup time: **16hours 55min**.

Goals/actions time: **4hours 33min .**

In this table, we rounded to the nearest 0.5 hours.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Setup(hours)** | **actions(hours)** | **easy to add new actions** |
| GOAP | 17 | 4,5 | TRUE |
| UtilityAI | 4.5 | 14,5 | FALSE |

Figure 6 action implementation time

### Testers

### Wins-Losses

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Wins** | **Loses** | **Draw** | | **Kills** | **Deaths** | |
| GOAP | 12 | 0 | 3 | 59 | | | 6 |
| UtilityAI | 12 | 0 | 3 | 52 | | | 8 |

Figure 7 Win/loss results

Both GOAP and UtilityAI performed equally to each other against the testers.

### Stamina

We see the average stamina curve during the 3 minute fight with the testers of both algorithms.

Figure 8 average stamina UtilityAI

Figure 9 average stamina GOAP

# Discussion

When overviewing these results, it seems like UtilityAI is easy to setup but hard to maintain. While GOAP is hard to set up but easy to use.

During the player experiment we noticed that GOAP succeeded greatly at the 2 in-game rules while UtilityAI failed at the stamina control. This doesn’t mean that UtilityAI is not able to do this but that it is really hard to set it.

We also realized that he sometimes attacks out of reach once he gets damage to his health. This happened because he wants to move back to heal when he gets damage, while wanting to move forward to attack. Situations like this shows that when adding too many graphs when wanting to add functionality, it turns it into a very delicate balance and might not work properly in game.

The amount of time put into the actions compared to GOAP already says that it took three times to implement a same type of behavior and still didn’t succeed for both in-game rules.

So for soulslike games, UtilityAI is not that suitable compared to GOAP since they need a lot of actions and many different ways to combine these actions. So it has to stay simple, the long setup time is worth it.

If we took a game with fewer actions or with actions that has a clear distinct timing to use, UtilityAI would have been perfect.

# Conclusion

As concluded in the working hypothesis 1 , UtilityAI proved to be easy to setup but really difficult to maintain while GOAP was difficult to setup but easy to maintain.

It also proofed easy for GOAP to follow in-game rules you want to setup.

# Future work

Creating an hybrid of UtilityAI and GOAP, I realized that GOAP has to most stable and easiest to create action/reactions to opponent in a soulslike game but his fixed priority destroys the creativity and more flexible behaviour.

Here UtilityAI can help by picking the best score for the goal to plan for.

Examples:

* Adding emotions to the AI by having the score be dependent on it and will pick action plans following this emotion.
* Have the AI learn during the fight. We can do this by keeping count on the moves that work and fails and keep that in account when creating score for the goal to pick.

# Critical Reflection

I realized late in the semester that I was way to focused on creating the project then actually looking for something to research. This was because I got to excited by the idea of learning a new decision-making algorithm and comparing it with the ones I know. I ended up lose a lot of time because of this and feel like I missed out on a interesting way to compare them for this paper.

My results are really depended on how I implemented both behaviours, in this case UtilityAI didn’t succeed in accomplishing both in-game rules but he possibly could have if I putted more time in or did it in a different way.

But I am still proud on the project and the new possibilities I learned for future projects.

# References

[1] David “Rez” Graham (14/09/2015) An Introduction to Utility Theory. <https://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf>

[2] Jeff Orkin (08/10/2008) Three States and a Plan: The A.I. of F.E.A.R.

[3]  Antora default UI() Utility Based AI https://grail.com.pl/documentation/documentation/1.2.0/manual/utility/utility.html

[4] Grantham Kwok-Hung Pang (27/08/2011) BLACKBOARD ARCHITECTURE FOR INTELLIGENT CONTROL

[5] [**Brent Owens**](https://tutsplus.com/authors/brent-owens) (23/04/2014) Goal Oriented Action Planning for a Smarter AI

[6] GDC (09/10/2017) Goal-Oriented Action Planning: Ten Years of AI Programminghttps://www.youtube.com/watch?v=gm7K68663rA&list=LL&index=7&t=443s

[7] Copilot(2024) A\* pathfinding

[8] Mona Erfani1 , Magy Seif El-Nasr1 , David Milam1 , Bardia Aghabeigi1 , Beth Aileen Lameman1 , Bernhard E. Riecke1 , Hamid Maygoli1 , and Sang Mah2 (30/07/2010) The Effect of Age, Gender, and Previous Gaming Experience on Game Play Performance https://link.springer.com/chapter/10.1007/978-3-642-15231-3\_33

[9] Barbara Bickham (20/10/2023) The Science Behind Game AI: Understanding the Algorithms and Techniques. <https://www.trailyn.com/the-science-behind-game-ai-understanding-the-algorithms-and-techniques/#techniques-for-game-ai-decision-making>

[10] [George Britton](https://papers.ssrn.com/sol3/cf_dev/AbsByAuth.cfm?per_id=4450893) (01/06/2023) Decision-Making Decisions: Artificial Intelligence In Computer Games https://papers.ssrn.com/sol3/papers.cfm?abstract\_id=4513757

# Acknowledgements

I am grateful that Fries Boury wanted to be my supervisor and for all the guidance he gave me, especially when I got stuck on how to do my comparison.

Kasper Geeroms for coaching me and Dr. Alexander Deweppe for providing all the background I needed for creating a paper.

Of course I also want to thank my friends and classmates who tested my project and gave me valuable feedback.

# Appendices

1. Time logs set up UtilityAI
2. Time logs set up GOAP
3. Time logs UtilityAI Actions
   1. Movement
   2. Throw
   3. Heal
   4. Block
   5. Quick Attack
   6. Hard Attack
4. Time logs GOAP Goals/Actions
   1. Movement
   2. Throw
   3. Heal
   4. Block
   5. Quick Attack
   6. Hard Attack
5. Github link
6. Time logs set up UtilityAI

I started this implementation on 10-11-2024.

10-11-2024 | 13:27 - 16:47. + 19:00 - 20:02.

11-11-2024 |20:05 -21:33. Debugging. Everything was implemented but caused crashes after few runs.

13-11-2024 | 20:00 – 21:17. Continue debugging. I was able to fix the trouble. Had to do with my wrong use of unreal at the time.

15-11-2024 | 22:02 – 23:57 Worked on the AIBehaviourBase and How to read from the UnrealGraphs.

Complete **setup** took a total of **9 hours 2 minutes**. This time included the blackboard key creation and AIBehaviourBase, also lost a lot of time debugging when I used Unreal in a wrong way. Of this, **4 hours 22 minutes** was the actual time to set up this specific behavior.

1. Time logs set up GOAP

Start implementing in unreal:

6-12-2024

13:24 – 14:01. Created classes for Goals, actions and GOAPBehvaiour. 🡪37min

16:55-17:14. Created WorldState class. 🡪9min

17:14 - 18:17. Created the Execute function of the GOAPBehaviour. 🡪1h 3min

20:02 – 21:04. Worked on the WorldState and started the Planner methode of GOAP. 🡪1h 2min

🡪2h 51min

7-12-2024

13:05-13:34. Finished GOAP planner in code. Improved WorldState class. 🡪29min

13:34 - 15:26. Made it possible to create assets from the code in editor. 🡪1h 52min

15:26 - 16:26. Made Goal and Action With his desired State in editor for testing. 🡪1h

16:26 - 19:10. Made IsVallid method for actions in editior, using the blackboard. 🡪2h 44min

🡪2h 15min

8-12-2024

15:13 – 16:40. Finished basic use of the GOAP system in editor. 🡪1h 27min

9-12-2024

20:50 - 22:07. Made Idle and WalkAction for testing the update of an action to finished state.🡪1h 17min

10-12-2024

21:00 – 22:42.Update of currentWorldState to be compared with Action/goal desiredState for validation or heuristic calculations. 🡪1h 42min

11-12-2024

21:12 – 23:09. Use the CurrentWorldState in the planner. 🡪1h 57min

18-12-2024

14:00-17:30 Try creating plan when there are more then 1 desired state. 🡪3h 30min

19:00-20:00 Planner reates best smallest possible plan keeping CurrentWorldState in mind 🡪1h

and each of there costs.

🡪4h 30min

19-12-2024

19:32- 19:54. Set a timet on every goal, put them temporary invalid when he takes to long 🡪22min

when to achieve his plan. Every goal got his RunMaxTime and TimeOutTime.

20-12-2024

18:53-19:27. Increased goal his IsVallid, check over more values instead of 1. 🡪34min

🡪The setup is now fully ready to be used to create plans for their first valid goal.

Total time: **16hours 55min.**

1. Time logs UtilityAI Actions
   1. Movement

Start 23-11-2024 from 16:30 – 18:29 and 20:18 – 23:24. Created 3 graphs for backwards movement and 2 for forwardMovement.

24-11-2024 from 15:10-16:13. His dodge failed regulary so worked more on the graphs.

Total time**: 6hours 8 min.**

Start 24-11-2024 from 18:00-18:13. Creeated graphs so he would move sideways on random moment and/when opponent is using HardAttack.

Total time: **13min.** This was fairly straightforward. On moments when he was idle before he will now move sideways depending on the random number or when he needs to dodge.

* 1. Throw

Start 24-11-2024 from 18:13 – 21:26. Make the AI throw when opponent uses heal at a distance. Some of this time was used to reform the Throw BattleAction.

Total time: **3hours 13min.**

* 1. Heal

Start 25-11-2024 from 21:04 – 22:32. When the AI has to heal he first need to get at a safe distance. So first we had to readjust the Backward/Forward Movement ActionScores. We did this by adding the graph dealing with his health.

Total time: **1 hour 28min**.

* 1. Block

Start 26-11-2024 from 20:02 – 21:58. When AI has full stamina, it will be safer to block the quickAttack instead of dodging. It also keeps him closer for a counterattack.

Total time: **1hour 56min**. No need to change the other graphs but we had to be sure the block will only be called at when is being attacked and when he has the stamina and when he is in reach of opponent.

* 1. Quick Attack

Start 27-11-2024 from 19:58 – 20:29. Made the AI use QuickAttack when he is close, has the stamina and when a random number is above 0.9.

The range of random number is 0-1, we choose +0.9 so that he will not endlessly use that attack but leaves it open for idle or HardAttack. Making him less predictable.

Total time: **29min**.

* 1. Hard Attack

Start 27-11-2024 from 20:29 – 21:30.

Total time**: 1hour 1min**. We want it to focus mostly as a reaction to opponent block, but he also must use it randomly as attack but lot less as the QuickAttack because the HardAttack will leave him open for counterattack.

1. Time logs GOAP Actions
   1. Movement

19-12-2024| 14:15-14:54 . Moving backward 🡪 39min

20-12-2024| 19:46-20:38 Moving sideways 🡪 52min

**🡪1hour 31min**.

* 1. Throw

20-12-2024 | 20:38 – 20:55. Made goal and his action assets. 🡪 17min.

* 1. Heal

20-12-2024| 18:20 – 18:53 . Moving backward then heal 🡪 33min

* 1. Block

21-12-2024 | 15:42 – 16:01. Blockasset created. 🡪19min.

21-12-2024| 16:19 – 17:03. StopBlock goal and action assets created. 🡪 44min.

* 1. Quick Attack

Was created during the setup, took around **15 min**.

* 1. Hard Attack

19-12-2024 | 20:30- 21:29. Taunt and combo assets created. 🡪59min.

1. Github to project link

<https://github.com/TGoncpp/GradworkFramework>