Learning Player Behavior Models to Enable Cooperative Planning for Non-Player Characters

Stephen Chen

CMU-CS-17-130
December 2017

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee
Maxim Likhachev, Chair
Nathan Beckmann

Submitted in partial fulfillment of the requirements for the degree of Master of Science

Keywords: Non-player Characters; Player modeling; Behavior learning; Online classification; Cooperative planning

Abstract

In video games, non-player characters (NPCs) tend to have trouble initiating cooperative tactics because players usually have an unspoken preference for how to solve a challenge. For example, in a "Capture the Flag" game where players have to reach an objective, it is usually difficult to program an NPC to accommodate both players that want to stealthily reach the goal and players that want to aggressively charge toward the goal. To enable NPCs to take a more active role in supporting players, we advocate learning a human-player model from in-game demonstrations to aid NPC planners in finding cooperative plans. To realize this, we first developed a classification algorithm which could learn player behaviors unsupervised, and then integrated it in our NPC AI planning framework for the game *The Elder Scrolls: Skyrim*. Specifically, we applied concepts of activity analysis from pervasive computing to identify and cluster similar sequences of game events which captured distinctive player behaviors in demonstrations. We generalized these clusters as player models which could predict player trajectories for the behavior pattern. We evaluated our player behavior classification system in a toy version of *Skyrim* to illustrate its performance. We also combined the classifier with our *Skyrim* NPC AI planner to demonstrate the classifier's practical application. The classifier associated each player behavior with supporting NPC demonstrations, which were used to learn search heuristics like Training Graphs for the situation. With our classifier's player model and the learned heuristics, our planner was able to use multi-heuristic A* to find cooperative plans that solved "Capture the Flag" scenarios in *Skyrim* which would have otherwise required manual engineering to solve.

Dedicated to my parents.

Acknowledgements

First and foremost, I would like to thank my thesis advisor Dr. Maxim Likhachev. He was the one who provided me the initial opportunity to work on an AI research project in the context of video games. He expressed interest in every stage of my research and supplied valuable insights and advice that helped guide this work.

I would also like to thank John Drake, a fellow graduate student and lab member, who was responsible for leading the NPC planning work which motivated my thesis. His existing work with *The Elder Scrolls V: Skyrim* provided me a huge tool set which streamlined the implementation and testing stages of my work. It has been a great pleasure working with him to bring advancements in AI techniques to a video game we both enjoyed.

I would like to extend this thanks to Eric Wong, another graduate peer in our Machine Learning department. On top of being an amazing teaching assistant for the machine learning course I took during my ungraduate program, he answered many of my learning related questions and provided many suggestions that bettered my methodology.

In addition, I would like to thank Dr. Nathan Beckmann for being my second thesis committee member. I am grateful for his interest in my work and taking the time to review my thesis.

Finally, I would like to thank Tracy Farbacher and Peter Steenkiste for running and managing the School of Computer Science's Fifth Year Masters Program. I would like to thank Bethesda Game Studios for developing the awesome game *Skyrim* and for providing the *Creation Kit* which allowed us fans to apply our own work to the game. And thanks to all of my friends and family who have supported and encouraged me along the way.

Contents

Abstract iii

Acknowledgements vii

1 Introduction 1

2 Player Behavior Classifier 5

2.1 Using Playstyles as Behaviors Classes 5

2.2 Problem Statement 6

2.3 Representing Demonstrations as Activities 7

2.4 Activities as n-Gram Histograms 8

2.5 Clustering Activities 9

2.5.1 Similarity Metric 9

2.5.2 Clustering by Finding Dominant Sets in an Edge-Weighted Graph 10

2.5.3 Computing Dominant Sets with Replicator Equations 12

2.6 Activity Classification 14

2.7 Improvements to the Similarity Metric 15

2.8 A New System for Robust Online Classification 16

2.8.1 Overall System Description 17

2.8.2 Adjusting for Confidence in the Activity Classifier 17

2.8.3 Computing the Classification at the Next Time Step 19

2.9 Using Discounted Histograms to Cull Old Events 21

2.10 Summary 23

3 Classifier Experiments and Results **25**

3.1 Simple *Skyrim* . . . 25

3.1.1 Game Description . . . 25

3.1.2 Events in Simple *Skyrim* . . . 27

3.1.3 Map Layouts . . . 28

3.2 Playstyle Demonstrations . . . 29

3.3 Clustering Demonstrations into Playstyle Classes . . . 29

3.4 Metrics . . . 32

3.4.1 Total Entropy of a Clustering . . . 32

3.4.2 Adjusted Rand Index . . . 33

3.5 Hyper-Parameters . . . 34

3.5.1 Normal κ_s vs Weighted κ_s . . . 34

3.5.2 Using Union vs Concatenation . . . 35

3.5.3 Deleting Duplicate Demonstrations . . . 35

3.5.4 Changing n in the Activity Histogram . . . 35

3.6 Validation Results . . . 36

3.7 Resulting Clustering . . . 38

3.8 Evaluating Classification Ability . . . 38

3.9 Responding to Switching Playstyles . . . 40

3.10 Shortcomings . . . 41

3.11 Summary . . . 42

4 Cooperative Planning for Non-Player Characters **45**

4.1 Heuristic Graph Search . . . 45

4.1.1 Training Graph Heuristic . . . 47

4.1.2 Multi-Heuristic A* . . . 47

4.2 Toward Cooperative Planning . . . 48

4.3 A Cooperative *Skyrim* Scenario . . . 49

4.4 Implementation and Integration . . . 51

4.4.1 *Skyrim* NPC Planning Framework . . . 51

4.4.2 Integrated Design 52
4.4.3 Running the Behavior Classifier 52
4.5 Results 54
4.5.1 Feasibility of New Cooperative Plans 54
4.5.2 Speeding Up Search with Supporting NPC Demonstration 57
4.6 Summary 59

5 Related Works **61**
5.1 Plan Recognition 61
5.2 Learning Approaches 63

6 Conclusion **65**

Bibliography **67**

List of Figures

2.1 Example Activity Sequence 8
2.2 Example Activity Histogram 8
2.3 Bayesian Network for Classification X_{t+1} 18

3.1 Simple *Skyrim* 26
3.2 Simple *Skyrim* Maps 30
3.3 Demonstration of Switching Playstyles 40
3.4 Impact of Discount Factor on Class Belief 42

4.1 *Skyrim* Scenario Map Layout 50
4.2 *Skyrim* Scenario with Labeled Points 51
4.3 Updates to the Planning Pipeline 52
4.4 Examples of COMBAT and SNEAK in *Skyrim* Scenario 53
4.5 NPC Plan without a Player Model 55
4.6 NPC Plan with a Player Model 56
4.7 NPC Plan with SNEAK Player, Weighted A* with Shortest Path Heuristic 57
4.8 NPC Plan with SNEAK Player, Weighted A* with T-Graph Heuristic . . . 59
4.9 NPC Plan with SNEAK Player, Multi-Heuristic A* with T-Graph Heuristic 60

List of Tables

3.1 Damage dealt by the player's weapons . . . 27
3.2 Enemy Statistics in Simple *Skyrim* . . . 27
3.3 Playstyle guidelines for Simple *Skyrim* . . . 31
3.4 Example Demonstration Event Sequences . . . 32
3.5 Clustering performance of various hyper-parameters when $n = \{1, 2, 3\}$. . . 36
3.6 Clustering performance of different n in the activity histogram, using normal κ_s and list concatenation. . . . 37
3.7 Clustering performance of different n in the activity histogram, using weighted κ_s and set union. . . . 37
3.8 Clustering of Demonstrations from Maps 1-4 . . . 38
3.9 Classifcation of Map 5 Demonstrations . . . 39
3.10 Delay (in number of events) until classification switch . . . 41

4.1 NPC planning results for when Player follows COMBAT playstyle . . . 56
4.2 NPC planning results for different search algorithms when player is following the SNEAK playstyle . . . 58

Chapter 1

Introduction

Modern video games are becoming increasingly complex. With advancements in processing power, game developers can boost the entertainment value and immersion of their virtual world by packing more sophisticated gameplay mechanisms into their game. For certain genres like action adventure, the rising complexity has caused developing smart AI for non-player characters (NPCs) to become increasingly difficult, particularly because of two prominent trends. Games now tend to give players many different options to tackle a single challenge, and NPCs are now expected to cooperate with players in solving challenges.

Giving players multiple options or tactics to solve a problem is not a new pattern in video gaming, but recent titles have consistently expanded the boundaries of player choice. In particular, action adventure role-playing games now usually feature an avatar with highly customizable abilities to fit the preferences of the player. For instance, *The Elder Scrolls V: Skyrim* (*Skyrim*) includes a diverse selection of weaponry, abilities, and special traits that allow players to heavily customize their avatar. This customization allows players to approach challenges in their own preferred playstyle, instead of one predetermined by the developers. For example, there are many dungeons in *Skyrim* that feature a "Capture the Flag" game scenario: there is a treasure chest at the end of the dungeon with many enemies guarding it, and the player must retrieve the treasure without dying. Players can customize their character to be knight-like or thief-like, giving them a choice between aggressively fighting through the enemies or stealthily slipping past them. Some players might choose a middle ground between these two

strategies, or event invent something entirely different.

Now, the increasing customizability is not necessarily an issue for all NPC AI, particularly those NPCs that operate outside of the player's playstyle preference like enemy NPCs. However, newer games tend to feature ally NPCs that are expected to cooperate with players in tackling certain challenges. *Skyrim* and *Borderlands 2* both feature companion NPCs which accompany the player on their adventure and aid in combat. Similarly, multiplayer team games like *Left for Dead* and *Rocket League* sometimes substitute in NPC characters when there are not enough human players to fill out the team. In both cases, the computer controlled NPC needs to operate not only intelligently, but also cooperatively with the human players. The cooperative aspects in these games require that NPCs be able to see beyond maximizing their own individual objectives, and actually participate in coordinated team strategies with human players.

Coordinating these strategies becomes increasingly difficult when the number of playstyles or tactics available to the player is large. First, if there is a large number of interacting gameplay mechanisms, it can be difficult for developers to foresee all possible approaches to a challenge. Second, it is hard to program an all-purpose NPC that accommodate all players, especially those with novel playstyles or tactics. In the *Skyrim* example, an aggressive NPC that can help a knight player fight a hoard of enemies is likely useless to a thief player that wants to remain undetected. To complicate the matter, a knight-like player may have self-imposed rules that make an aggressive NPC unattractive to their playstyle. Perhaps the player only wants to fight easy enemies and not hard ones, or maybe the player wants to be stealthy if there are too many enemies in one area. In these cases, an NPC programmed for the prototypical knight player will quickly ruin the strategy of these cautious players.

We thus propose that NPCs learn these player strategies directly from human demonstrations, rather than requiring developers to anticipate these cases and hard coding the NPC response. These demonstrations can be recorded during the development of the game, perhaps during playtesting, or they can be recorded after the game's release,

when players are actively playing the game. These demonstrations will be used to create models for the players' different playstyles and tactics, and it should be easy to create new models as new demonstrations of new playstyles appear. To realize this, we will draw upon activity analysis techniques used in pervasive computing to identify high-level human activities. We will identify similar behavior patterns in the demonstrations in order to differentiate distinct playstyles. We can then use the relevant demonstrations to construct player models that can predict the player's next move. During active gameplay, we classify the player's current behavior as one of our discovered playstyle and then pass the relevant model to an NPC planner. This planner will allow the NPC to accommodate the player, even if the game environment differs from the demonstrations. Furthermore, once we learn the players' playstyles, we can provide demonstrations of supporting NPC behavior for each of the playstyles and use these supporting demonstrations to help guide the NPCs planning process.

Our goal is to reduce the developers' work in trying to hard-code appropriate behaviors for NPCs. We simplify the process of recognizing human playstyles by learning them directly from player demonstrations which are easily attainable. As we will discuss in Chapter 4, we can associate these playstyles with supporting NPC demonstrations, which can be used to guide NPC planning. This will cut out the work of hard-coding NPC support behaviors, and again rely on easily attainable demonstrations of good support behavior. Our system will also allow NPCs to be flexible to new playstyles and tactics, as this learning process can continue even after the game ships.

At a high level, this thesis contributes the following:

- The Player Behavior Classifier, a complete pipeline for learning and classifying player behaviors in video games, based on existing activity analysis techniques [13, 21].

- Modifications to the existing learning algorithm to enable better performance for our domain.

- A novel method to handle online behavior classification during active gameplay

through the use of Bayesian Belief Networks [22] and our newly proposed Discounted Histograms.

- A novel method to integrate the Player Behavior Classifier into an NPC planning system, which involves the use of Training Graph Heuristic [6] and Multi-Heuristic A* [1] for fast planning times.
- Results which demonstrate how the combined system can enable an NPC planner to find cooperative solutions to a task.

Chapter 2

Player Behavior Classifier

In this chapter, we present the Player Behavior Classifier, our solution to modeling player behavior. We begin by explain how player behaviors are typically divided into distinct classes known as **playstyles**. We then formalize the problem of learning player behavior patterns from demonstrations as an offline unsupervised learning problem and an online classification problem. After that, we explain how to apply an existing activity analysis technique to our domain, which involves clustering activities via Dominant Sets [13, 21]. Next, we build upon the existing technique by presenting our own modifications to similarity function. Finally, we introduce a new Bayesian Belief Network [22] that will allow robust online classification, as well as a new concept called the Discounted Histogram, which will allow our classifier to adapt to changing player behaviors.

2.1 Using Playstyles as Behaviors Classes

Our ultimate goal is to aid an NPC planner in finding plans which factor in the current human-player behaviors. To this end, the planner must be able to generate a model for the current player's behavior so that it can extrapolate their next moves during planning. Luckily, we can expect players to behave consistently to according to some **playstyle**, a self-imposed set of rules or guidelines for playing the game. We can thus expect the player's plans or tactics devised under a particular playstyle to look relatively the same. For example, in *Skyrim*, a player that is playing a "thief" playstyle is

likely to devise plans or employ tactics that let them sneak past any enemy guards in a dungeon. So long as the player adheres to the "thief" playstyle, we can expect that in future dungeons, the player will continue creating similar plans that let them sneak past the enemies. In other words, the combat choices the player makes on-the-fly are closely related to the overarching playstyle. Thus, if we can recognize the player's current playstyle, then we are likely able to accurately predict their next move.

However, players are not tied down to one particular playstyle when playing a game. A thief avatar in *Skyrim* can just as easily become a knight if the player decides to equip a sword and shield. The plans and tactics employed by a knight will likely be distinct from the plans of a thief. As a result, there could potentially be multiple playstyles for which we need models. The fact that players can easily switch between playstyles also means we need to be able to choose between our different models on the fly.

With this in mind, we examine two problems. One, how do we build the possible classes of the player playstyles? And second, how do we recognize which playstyle the player is currently following?

2.2 Problem Statement

Let us formalize the exact problem statement of behavior modeling for playstyles in video games. We first start in an offline context. We are given a set of complete in-game replays or demonstrations (whose representation will be discussed soon). We need to build models of human playstyles from these demonstrations. Then, once we have models, we move to an online setting where we are given a live stream of data (a live replay) from a player currently playing the game. We must then classify which model the player is following and then output that model to an NPC planner. In this formulation, the offline problem of learning playstyle models can be solved before or after a video game ships. This means developers can create an initial batch of playstyle models during game development and then continue collecting playstyle data for new models even after the game's release.

In the next sections, we present a solution to this problem based on the algorithms provided by Hamid et al. [13], which were originally used for classifying human activities in a bookstore loading dock. This system, known as the Player Behavior Classifier, will treat demonstrations as high-level activities, which we can cluster into disjunctive activity classes. These activity classes will represent playstyles. Once we have these playstyle classes, we can identify new live replays as members of one of these classes and produce an appropriate model of that class.

2.3 Representing Demonstrations as Activities

To model player behaviors, we begin by establishing a groundwork for how to describe actions and behaviors. We define an **event** to be a representation of some interaction of agents or objects in the game setting. In a the *Skyrim* context, these events can be as specific as swinging a sword to something as abstract as "initiating combat." An **activity** is an ordered sequence of events, or in other words, a discrete time series that describes everything that transpires in terms of events. We refer to this representation of activities as an *activity sequence* or *event sequence*.

Depending on the granularity of the events, an activity could describe a the movement of a character–where events are the characters' world coordinates–or an activity could describe a high-level plan, tactic, or strategy–where the events are more abstract like "flank the enemy" then "engage in melee combat." Given more abstract events, activities can better capture what occurs at a high level. It is thus favorable to define events which describe critical points in the activities we want to analyze. In our context, we want our activities to describe playstyles or playstyle specific tactics, which means our events have to be as abstract as "initiating combat" and "sneaking past an enemy." As example activity sequence is given in figure 2.1.

We assume that the demonstrations given to us in the offline phase are in the form of event sequences, instead of raw state logs. The ability to extract abstract high level features from state data is highly domain dependent and would require either careful hand engineering or another learning algorithm. Furthermore, games generally already have

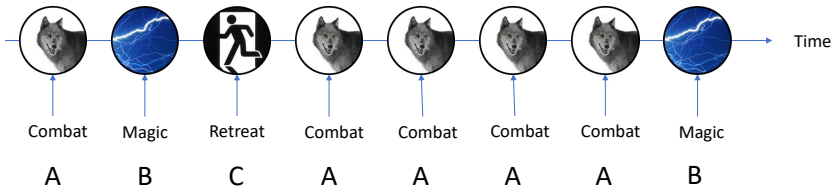


FIGURE 2.1: An example activity sequence from *Skyrim*. The player initially engages a wolf, uses magic, and then retreats. The player then later engages four wolves, and uses magic. These events can be thought of as types A, B, C, creating the string $ABCAAAAB$.

A	B	C	AB	BC	CA	AA	ABC	BCA	CAA	AAA	AAB
5	2	1	2	1	1	3	1	1	1	2	1

FIGURE 2.2: An example activity histogram with $n = \{1, 2, 3\}$ for the sequence $ABCAAAAB$ (given in figure 2.1).

a notion of important high level events in their code base. For example, state machines, which are common in game AI, encode high-level information about the character's current behavior. It is easier to log the high-level events occuring in these state machines than it is to analyze raw game state and infer the corresponding events. We thus leave the potential problem of extracting event sequences from raw data to other works.

2.4 Activities as n-Gram Histograms

Once we have demonstrations in the form of activity sequences, we further break them down into an **activity histograms**. An activity histogram is simply an n-Gram, or a frequency count of the substrings in the activity sequence. We simply count the number of instances of each unique substring of length n in the event sequence, and the resulting histogram is a feature vector that represents the activity. The activity histogram is a good feature vector for the activity sequence, because temporal structure in long sequences can often be captured by its smaller local relationships. An example activity histogram can be found in figure 2.2.

Hamid et al. provide further analysis of the representational power of n-Grams and

show that they provide features that are as informative as a Vector Space Model (word-count model) and Hidden Markov Models when it comes to discriminating classes of activities [13]. Note that in the case $n = 1$, an n-Gram is equivalent to a naive Vector Space Model. We can also use multiple values of n in the n-Gram, e.g. keeping frequency counts of substrings of length 1, 2, and 3. We use the notation $n = \{1, 2, 3\}$ to indicate that our n-Gram keeps frequency counts of substrings of more than one length.

As n increases, we capture the structure of an activity more rigidly. Of course, larger n suffers from higher dimensionality and requires more space to represent. In addition, it has been shown that, in the context of capturing player behaviors, n-Grams tend to perform worse after $n = 5$, as the natural randomness in human actions make it rare to see a long event string twice [20].

The correct choice of n is highly domain dependent, and we investigate the choice in our experiments. It is generally accepted that $n = 3$ provides a good representation of human behaviors.

2.5 Clustering Activities

2.5.1 Similarity Metric

We now define a similarity metric between activities that will help us cluster them into classes.

Let A and B be activity sequences, and let their corresponding activity histograms be H_A and H_B. Let S_A and S_B represent the set of the unique substrings with non-zero count in H_A and H_B respectively. Let $f(s \mid H_A)$ be the frequency of substring s in histogram H_A. Then the similarity metric is defined as follows:

$$\mathsf{sim}(A, B) = 1 - \sum_{s \in S_A, S_B} \kappa_s \frac{|f(s \mid H_A) - f(s \mid H_B)|}{f(s \mid H_A) + f(s \mid H_B)} \tag{2.1}$$

where $\kappa_s \in [0, 1]$ is the *importance* of substring s, and $\sum_{s \in S_A, S_B} \kappa_s = 1$. In the original formulation by Hamid et. al. [13], $\kappa_s = \frac{1}{|S_A| + |S_B|}$ for all s, where $|\cdot|$ is the set cardinality operator. We will discuss our proposed improvements on this metric in section 2.7.

The similarity metric is a function that always returns a value in $[0, 1]$. Activities which are exactly the same will have similarity 1, and activities with mutually exclusive substrings will have similarity 0. Note that the total number of unique substrings that are mutually exclusive across the two activities will directly impact the similarity metric. For substrings which are present in both activities, it is the difference in their frequency counts that affects the metric. This metric has the property that it is commutative.

There are other formulations for similarity between discrete time series, such as min-edit distance [17] or dynamic time warping [3]. These are not well suited for activity analysis since they are sensitive to input size and assumes that sequences need to have the same global ordering of events to be similar, which may be too strict for human activities which have strong local structure but weak global structure. Dynamic time warping also requires a metric of distance between the discrete events, which is not easily definable between game events. It is not intuitively clear what the distance between "initiating combat" and "sneaking past a guard" is.

2.5.2 Clustering by Finding Dominant Sets in an Edge-Weighted Graph

We now present the method to group activities into disjunctive classes. In our context, this will separate demonstration activities into clusters which represent distinctive playstyles.

Suppose we are given K demonstration activities. We represent the activities and their similarities to each other as a complete edge-weighted graph of size K. Each vertex represents an activity histogram, and each edges has weight equal to the similarity of the two activities they connect.

We equate the problem of discovering activities classes to finding the "maximal cliques" of this edge-weighted graph. Normally, a maximal clique is defined as the largest subset of vertices which are all adjacent to each other. However, since our graph is complete, our notion of "maximal clique" refers to finding a subset of vertices that are all connected to each other by high weight edges. We call such a subset a **dominant**

set, which is described more thoroughly in [21]. The clustering algorithm involves finding the dominant set, removing those vertices from the graph, and then repeating the process until no more dominant sets can be found. Each set thus becomes a class, and anything remaining is assigned into the found classes.

We will now formalize the notion of a dominant set. We represent our demonstration activities as an undirected edge-weighted complete graph with no self loops. Our K activities form the vertex set V and are labeled by $1, \ldots, K$. The edge set is $E \subseteq V \times V$. The weights are given by $\mathsf{sim}(i, j)$.

The weighted adjacency matrix, or **similarity matrix**, of our graph is a $K \times K$ square matrix whose elements are given by:

$$a_{ij} = \begin{cases} \mathsf{sim}(i,j) & i \neq j \\ 0 & \textit{otherwise} \end{cases} \tag{2.2}$$

where sim is our defined similarity metric (equation 2.1).

Next, we introduce "average weighted degree," which is the average weight of the set of edges touching a vertex. This measure of how similar a vertex i is to a set of other vertices S. Let $i \in V$ and $S \subseteq V$.

$$\mathsf{awdeg}_S(i) = \frac{1}{|S|} \sum_{j \in S} a_{ij} \tag{2.3}$$

In addition, we introduce a notion of the similarity of i and j with respect to how similar i is to a set S of vertices. For $i, j \in V$, $S \subseteq V$ and $j \notin S$:

$$\phi_S(i,j) = a_{ij} - \mathsf{awdeg}_S(i) \tag{2.4}$$

Note that ϕ can produces values in $[-1, 1]$. To understand this function, consider a cluster of vertices S where i is very close (similar) to that cluster and j is very far away (dissimilar). $\phi_S(i,j)$ would thus be a very negative value. Intuitively this makes sense because i and j are far away from each other, especially when we consider how close i is to the cluster S. Conversely, suppose S is a very scattered cluster, but i and j are very

close to each other. $\phi_S(i,j)$ would this be very positive, because i and j are very similar when we consider how dissimilar i is from the set S.

This brings us to a final definition, which evaluates how similar a vertex i is to a set of vertices S with respect to the overall similarities of the other vertices in the set. In other words, if we consider the other vertices in S as a cluster, how well does i sit with that cluster? Let $S \subseteq V$, and $i \in S$. We make the following recursive definition:

$$w_S(i) = \begin{cases} 1 & \text{if } |S| = 1 \\ \sum_{j \in S \setminus \{i\}} \phi_{S\setminus\{i\}}(j,i) w_{S\setminus\{i\}}(j) & \text{otherwise} \end{cases} \tag{2.5}$$

Informally, the recursive case expresses the following: how well vertex i sits with the cluster S depends on how similar i is to each other vertex j in S, weighted by how well each j sits with S.

We can now say that a set of vertices $S \subseteq V$ is dominant, if $w_S(i) > 0$ for all $i \in S$, and $w_{S\cup\{i\}}(i) < 0$ for all $i \notin S$. The first condition represents internal homogeneity: all vertices in the dominant set are very similar to all the other vertices in the dominant set. The second condition represents external inhomogeneity: all vertices outside the dominant set should be dissimilar to the entire dominant set.

2.5.3 Computing Dominant Sets with Replicator Equations

Solving for the subset S that satisifies Equation 2.5 is combinatorially hard. As such, we instead rely on solving a related quadratic program using a continuous optimization technique. The solution to this quadratic program will tell us exactly which vertices are in the dominant set.

The quadratic program can be formulated as such:

$$\begin{aligned} &\text{maximize} && \tfrac{1}{2} x^T A x \\ &\text{subject to} && x \in \Delta \end{aligned} \tag{2.6}$$

Here, A is the similarity matrix as given by equation 2.2. Δ is the standard simplex of $\mathbb{R}^K$, i.e. the set of vectors $x \in \mathbb{R}^K$ such that each element $x_i \geq 0$ and $\sum_{i=1}^{k} x_i = 1$. Pavan and Pelillo provide a proof for why the solution to this quadratic program corresponds to the dominant set of an edge-weighted graph [21].

To solve this quadratic program, we use a continuous optimization technique known as *replicator equations*. This is essentially Newton's Method of optimization which finds a stationary point. The optimzation starts with a guess of what x should be, and then proceed to step in the direction that will cause the optimization function to have a first derivative of zero. x is repeated stepped until convergence, indicating we have reached an optima. As proved in [21], the optima of this quadratic program is a global maximum.

Let $x_i(t)$ represent the ith element of a vector x at time step t. We set $x(0)$ to be the barycenter of the feasible region Δ, i.e. $x_i(0) = \frac{1}{K}$ for each i. Then, the optimization step is:

$$x_i(t+1) = x_i(t)\frac{(Ax(t))_i}{x(t)^T Ax(t)} \tag{2.7}$$

where $(Ax(t))_i$ represents the ith element of the vector resulting from the matrix multiplication. The equation is stepped until $x(t+1) \approx x(t)$, at which point we have reached an optimal solution x^*.

The optimal solution x^* represents:

$$x_i^* = \begin{cases} \frac{w_D(i)}{W(D)} & \text{if } i \in D \\ 0 & \text{otherwise} \end{cases} \tag{2.8}$$

where D is the dominant vertex set of the graph, $w_D(i)$ is as defined in equation 2.5, and $W(D) = \sum_{i \in D} w_D(i)$.

We see that if the ith element of of x^* is non-zero, then vertex i is in the graph's dominant set. This allows us to extract the dominant set D from x^*. Furthermore, since $\sum_i x_i = 1$, each non-zero x_i term actually represents the *participation*, or weight, of vertex i in the dominant set D. This fact will become useful in the next section. Let

the *participation* of a vertex i in dominant set D be defined as:

$$p_D(i) = x_i^* \tag{2.9}$$

Finally, we group together the demonstrations represented by the vertices in D and treat the group a unique activity class. We remove those vertices in D from the graph, and then repeat the process of finding the next dominant set in the new graph. The process repeats until no new dominant sets can be found.

2.6 Activity Classification

So far, we have described how to represent our demonstrations as activities and how to cluster these activities into disjunctive activity classes that represent playstyles. This concludes the offline learning phase of the Player Behavior Classifier. We now move onto the classification problem: how do we determine which playstyle the player is pursuing?

Suppose we let a player play the game for a bit, and we record the sequence of events that are triggered. These events form an activity sequence which we can express as an activity histogram. Let τ represent this new activity. Let C be the set of classes that we discovered in the offline analysis. Each activity class $c \in C$ is computed via the replicator equations discussed in the previous section. Each activity $j \in c$ has some participation $p_c(j)$ in the class. We can compute a similarity score of the new activity sequence τ to each existing classes $c \in C$ with the following function:

$$A_c(\tau) = \sum_{j \in c} \mathsf{sim}(\tau, j) p_c(j) \tag{2.10}$$

This function is simply the weighted average of similarities between τ and the activities in class c. Note that A_c takes on values in $[0, 1]$.

From here, we can classify the activity as a member of class c^* by just taking the highest similarity.

$$c^* = \arg\max_{c \in C} A_c(\tau) \tag{2.11}$$

Alternatively, we can normalize the similarities and produce a probability distribution over the possible classes. Let the normalizing factor be $\alpha = \sum_{c \in C} A_c(\tau)$. If c^* is a random variable indicating the classification, then for any class $c \in C$:

$$Pr[c^* = c] = \frac{A_c(\tau)}{\alpha} \tag{2.12}$$

This formulation fails if $\alpha = 0$, which can happen if the new activity τ matches none of the of the existing activity classes. In our system, we circumvented this issue by creating a special exception for this case, returning the uniform distribution instead. In a more robust system, this special case should be handled more carefully, as the activity τ is likely part of an activity class that was not seen during training.

This concludes this system for classifying activities, or playstyles in our context. We henceforth refer to this system as the **activity classifier**.

2.7 Improvements to the Similarity Metric

The activity classifier we have discussed thus far has been an application of Hamid et. al.'s work [13], using video game events and activities. We now propose some modifications to the existing similarity metric to improve the clustering performance.

First recall the similarity metric:

$$\mathsf{sim}(A, B) = 1 - \sum_{s \in S_A, S_B} \kappa_s \frac{|f(s \mid H_A) - f(s \mid H_B)|}{f(s \mid H_A) + f(s \mid H_B)} \tag{2.1 revisited}$$

where A, B are activities, H_A, H_B are their activity histograms, and S_A, S_B are the unique substrings in H_A and H_B.

In the original formulation, the notation $s \in S_A, S_B$ means once for every element in S_A and once again for every element in S_B. In other words, $s \in S_A, S_B$ is interpreted as iterating through the concatenation of two lists ($S_A + S_B$). The main takeaway is that $S_A + S_B$ is different from the set union $S_A \cup S_B$, which deletes repeat substrings. As a result, the original formulation gives more weight to subtrings which are found in both activities than substrings which are found only in one. We thus propose a modification to the similarity metric: to use set union instead of list concatenation. The choice of $S_A \cup S_B$ (set union) over $S_A + S_B$ (list concatenation) will affect the values produced by the similarity metric and will thus affect the clustering produced by the dominant sets algorithm. We explore the empirical impact of choosing $S_A + S_B$ vs $S_A \cup S_B$ in our experiments in chapter 3.

The original work also set $\kappa_s = \frac{1}{|S_A|+|S_B|}$ for all s, where $|\cdot|$ is the set cardinality operator. κ_s was originally a single constant κ, which was intended to be a normalizing factor. It gave each substring equal weight in contributing to the similarity metric. We propose instead to use κ_s, a constant parameterized by substring s, so that we can potentially weight certain substrings more heavily than others. For example, we might choose to weigh substrings of length 3 as more important than substrings of length 1, since being able to match longer substrings is more important when measuring similarity. We explore the empirical impact of changing the value of κ_s for different s in our experiments in chapter 3.

2.8 A New System for Robust Online Classification

We now move onto addressing other issues in online classification. The activity classifier presented requires that an ongoing event sequence be built in order for classification to occur. However, in some cases, events might not be triggered frequently enough to allow our system to constantly update its guess of the current playstyle. In addition, when the an online gameplay run first begins, there is very little event data for the activity classifier system to use. Previous works only employed the activity classifier in an offline setting, meaning they only classified completed activity sequences rather than

partial sequences. Ideally, we desire a system that can account for the lack of data in an online setting, and remain flexible in its classifications as the player moves around in the game world. We now present a new Bayesian Belief Network [22] that will mediate our classifications when there is little event data.

2.8.1 Overall System Description

Suppose our activity classifier has discovered activity classes C during offline training. Let τ_t represent the online event sequence that we witness by game time t. Let $AC_t \in C$ be a random variable that describes the activity classifier's classification of τ_t.

We introduce a new random variable $X_t \in C$, which will describes our actual online classification at time t. We initialize $Pr[X_0]$ according to some prior. This can be based on the percentage of activity examples per activity-class, or it can be a uniform distribution. We are interested in the distribution of X_{t+1}, the classification at the next time step.

To compute the distribution, we need a few additional random variables:

- Q_t is a boolean that is true if AC_t is an accurate classification.
- X'_t is an adjusted classification at time t that arbitrates X_t and AC_t by factoring the confidence in the activity-classifier ($Pr[Q_t]$).
- OBS_t is the set of in-game observations at time t. The exact nature of these observations are discussed in 2.8.3.
- Z_t is an boolean that is true if the player shows tendencies of switching between activity classes at time t.

Figure 2.3 is a Bayesian network that illustrates dependencies of these variables. The problem at hand is to compute $Pr[X_{t+1} \mid \mathsf{OBS}_t]$. To this end, we examine how the distributions X'_t and X_{t+1} are computed.

2.8.2 Adjusting for Confidence in the Activity Classifier

X'_t is an intermittent value and is used to arbitrate the a potential lack of data in the active activity sequence τ_t. If our system has not seen many game events yet, then

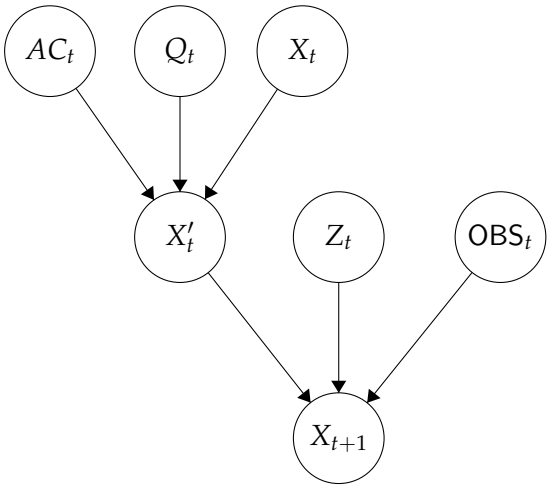


FIGURE 2.3: Bayesian Network for Classification X_{t+1}

the chance that activity-classifier's result (AC_t) is correct is probably low. This is what $Pr[Q_t]$ aims to capture.

Suppose that the ongoing activity sequence is τ_t. $Pr[Q_t]$ should be our confidence in the activity-classifier. We want this probability to reflect the fact that the activity-classifier is more accurate as the number of events in τ increases. In other words

$$Pr[Q_t = 0] \propto \frac{1}{|\tau_t|} \tag{2.13}$$

i.e. the chance AC_t is wrong decreases with more events.

In our system, we opted for the following equation:

$$Pr[Q_t = 1] = k - \frac{k}{r|\tau_t| + 1} \tag{2.14}$$

where k is a value in $[0, 1]$ which describes the maximum probability that the activity classifier is correct, and r is a positive real which describe how fast the classifier becomes correct as we see more events.

This gives us, for some $i, j \in C$:

$$Pr[X'_t = c \mid AC_t = i, X_t = j, Q_t = 1] = \begin{cases} 1 & \text{if } c = i \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

and

$$Pr[X'_t = c \mid AC_t = i, X_t = j, Q_t = 0] = \begin{cases} 1 & \text{if } c = j \\ 0 & \text{otherwise} \end{cases} \tag{2.16}$$

The distribution of X'_t then is simply:

$$Pr[X'_t = c] = Pr[AC_t = c]Pr[Q_t = 1] + Pr[X_t = c]Pr[Q_t = 0] \tag{2.17}$$

Intuitively, if Q_t is true, then we use the activity classification AC_t. Otherwise, if Q_t is false, then we go with the classification given at the previous timestep X_t.

2.8.3 Computing the Classification at the Next Time Step

With the distribution of X'_t, we can now move on to computing the classification at the next time step X_{t+1}. Here we assume we are given the set of observations OBS_t.

OBS_t is a vector containing the following: Consider all of the game events E that could possibly be achieved at time t. We assume that, given the current game state, we can compute a distance to each available game event. For example, a player might be 2 meters away from a "combat start" event. OBS_t will be a vector of how these distances have changed between time steps. Formally, for game event $e \in E$, the element $\mathsf{OBS}_t(e)$ is the difference in distance to event e from the game state at time t to the game state at time $t + 1$. Intuitively, these are the delta distances to each of the game events. Negative values mean the player is approaching a game event, whereas positive values mean the player is moving away.

X_{t+1} also depends on another variable Z_t. Z_t is true if the player has a tendency to switch playstyles or activity classes. For our work, we simply assume that $Pr[Z] = 0.5$, but this probability could also be the result of a more involved computation.

With this variable, we can say that given Z_t is false:

$$Pr[X_{t+1} \mid \mathsf{OBS}_t, Z_t = 0] = Pr[X'_t] \tag{2.18}$$

i.e. the classification should be the same as the previous classification if we are given that the player does not have any tendency to switch playstyles.

The question remains how to compute $Pr[X_{t+1} \mid \mathsf{OBS}_t, Z_t = 1]$. We propose the following. Look at all the game events which have a negative delta distance in OBS_t. For each of these events e, run the activity classifier on a new activity sequence $\tau_t^e = \tau_t + [e]$, which is the current activity τ_t with e appended. Let τ_t^* be a random variable which takes on one of these speculated activity sequence (e.g. τ_t^* could be τ_t^e). Let $Y_t \in C$ be the random variable for the speculative activity classification of τ_t^*. Then, we define:

$$Pr[X_{t+1} \mid \mathsf{OBS}_t, Z_t = 1] = Pr[Y_t \mid \mathsf{OBS}_t] \tag{2.19}$$

meaning the distribution of the next classification is equivalent to trying to classify all possible future activity sequences.

Furthermore:

$$Pr[X_{t+1} \mid \mathsf{OBS}_t, Z_t = 1] = \sum_{\tau_t^e} Pr[Y_t \mid \mathsf{OBS}_t, \tau_t^* = \tau_t^e] Pr[\tau_t^* = \tau_t^e \mid \mathsf{OBS}_t] \tag{2.20}$$

where $Pr[Y_t \mid \mathsf{OBS}_t, \tau_t^* = \tau_t^e]$ is the distribution produced by running the activity classifier on activity sequence τ_t^e.

And finally, we define

$$Pr[\tau_t^* = \tau_t^e \mid \mathsf{OBS}_t] \propto -\mathsf{OBS}_t(e) \tag{2.21}$$

In other words, the likelihood of τ_t^e being the next activity sequence is based on the observation of how big of a step we took toward game event e. In our implementation, we simply took all of the negative delta distances, and normalized them into a probability distribution.

Intuitively, we have considered all possible next events and run our activity classifier on a speculation of what the next activity sequence could be. These speculated activity sequences are weighted according to the probability that they occur, which is computed based on whether the player is moving towards those events.

This finally brings us to:

$$Pr[X_{t+1} \mid OBS_t] = Pr[Z_t = 0]Pr[X'_t] + Pr[Z = 1]Pr[Y_t \mid \mathsf{OBS}_t] \tag{2.22}$$

This complete formulation does the following:

- It factors in confidence in the activity classifier, weighing the activity classifier lower when there are few activities in the online activity sequence.
- It provides a classification at the resolution of the game time step, rather than at the resolution of triggered game events. This is important when game events are not triggered frequently.
- It factors in the motion of the player and how fast they are approaching new game events in addition to the results of the current activity classification.
- Following the previous point, it provides a classification distribution even when there are zero or few events in the online activity sequence. At zero events, the classification is dependent on just the motions of the player. As more events are seen, this formulation provides a smooth transition to relying more on the activity classifier.

2.9 Using Discounted Histograms to Cull Old Events

We now address a problem that arises from the histogram representation of activities: a histogram can become polluted with old event substrings if an activity sequence runs on for too long. Suppose in an online activity sequence, a player follows the playstyle of a knight for 50 events, and then follows the playstyle of a thief for 50 events. The activity classifier would probably not be able to change between its classification from

"knight" to "thief" until all 50 thief events are seen, because the 50 initial knight events dominate the activity histogram. In a practical game application, we can potentially reset the online activity sequence between loading screens, but what if a player spends a long time in one environment without triggering a load screen? The activity sequence would run on for very long, and events seen very early in the sequence would continue to play a factor in the activity's classification by the end.

A potential solution is to window the activity sequence: once the online activity sequence sees some number of events, we start deleting the oldest events. One issue with this approach is that we may not necessarily know a correct window size. It is not clear how many events will be required in the online activity sequence for our activity classifier to produce a reasonable classification. Furthermore, in the worst case, if we fix window size w and have w events belonging to activity class A in the activity histogram, we may need to see at least $\frac{w}{2}$ events of activity class B before our activity classifier considers B as a possible classification.

We thus propose a new concept called the **discounted histograms**. Instead of a normal n-Gram, we will use the discounted histogram representation as our activity histograms. A discounted histogram works much like a regular n-Gram histogram, except that it lowers the importance of older event substrings by a discount factor. Instead of a frequency count of each unique event substring, we have sum of contributions from each instance of a unique substring.

First, let us formalize how a regular n-Gram is constructed. Suppose we are given an activity sequence τ. Let S be the set of unique event substrings up to length n that occur in τ. For non-empty substrings $s \in S$, we define an indicator function $c(s, i)$ that returns 1 if the $|s|$ length substring that is i events away from the end is s, and 0 otherwise.

For example, suppose we have events A, B, C, and an activity sequence $\tau = [A, B, C, B]$. Then:

- $c([C, B], 0) = 1$
- $c([B, C], 1) = 1$
- $c([A], 3) = 1$

- $c([A], 0) = 0$

In the regular histogram H_τ constructed from τ, the frequency of substring $s \in S$ in is:

$$f(s \mid H_\tau) = \sum_{i=0}^{|\tau|-|s|} c(s, i) \tag{2.23}$$

This simple counts the number of instances of s in τ.

To construct a discounted histogram, we present a simple modification of equation 2.23 to discount older activities. Fix a discounter factor $\gamma \in (0, 1]$. We define f to instead be:

$$f(s \mid H_\tau) = \sum_{i=0}^{|\tau|-|s|} c(s, i) \cdot \gamma^i \tag{2.24}$$

Event substrings that occur at the end of the activity sequence have contribution 1. The event substring that is one event away from the end now has a contribution of γ. The event substring two events away from the end has a contribution of γ^2, and so forth. As an activity sequence grows large, substrings that occur early in τ effectively contribute nothing to the discounted histogram.

Note that our original definition of the similarity metric sim (equation 2.1) can still remain the same and will preserve its original properties. Between two activities, for mutually exclusive substrings, it is the number of these substrings that directly impact the similarity. For substrings that occur in both activities, it is now the difference in the substring's contribution scores.

We demonstrate in our experiments in chapter 3 that the use of discounted histograms allows the activity classifier to better react to changes in playstyle during online classification.

2.10 Summary

All of the algorithms described above come together to form a complete system that we call the Player Behavior Classifier. To recap, in-game demonstrations are first provided in the form of event sequences which we call activities. We construct activity histograms

out of these sequences, and then use our similarity metric to cluster similar activities via a technique called dominant sets. Once we have activity classes of similar activities, we can classify new activities according to the learned classes.

For our domain, we also modify the existing similarity metric to use set union and a string parameterized importance factor. We include a Bayesian Belief Network to enable online classification even when the activity classifier has not seen many game events. We further use discounted histograms to prevent old game events from polluting the activity histogram. In the next chapter, we will examine the performance of the Player Behavior Classifier in both an offline and online setting.

Chapter 3

Classifier Experiments and Results

In this chapter, we describe our implementation of the Player Behavior Classifier outlined in chapter 2 in the context of a simpler version of *Skyrim*. We then outline a series of experiments that test the performance our classifier with various algorithm hyper-parameters, such as the similarity metric changes mentioned in section 2.7. Finally, we present the empirical results and discuss the implications on how the system would perform in the full fledged *Skyrim* game.

3.1 Simple *Skyrim*

We implemented a simpler version of the *Skyrim* game in the Unity Game Engine. This 3D top-down game contains a subset of the features found in *Skyrim*.

3.1.1 Game Description

The game environment is a simple 2D map composed of hallways and enemies, viewed from a birds-eye perspective. The goal is for the player to navigate through the hallways to the goal, killing enemies along the way as necessary. The player moves using WASD controls and uses the mouse to aim and fire their weapon. A screenshot of the game is provided in figure 3.1.

The player is given two weapons: a sword and a bow. The sword is a melee weapon, and when swung can deal damage to nearby enemies. The bow is a ranged weapon, and when fired will launch an arrow from the player, dealing damage to all enemies in the

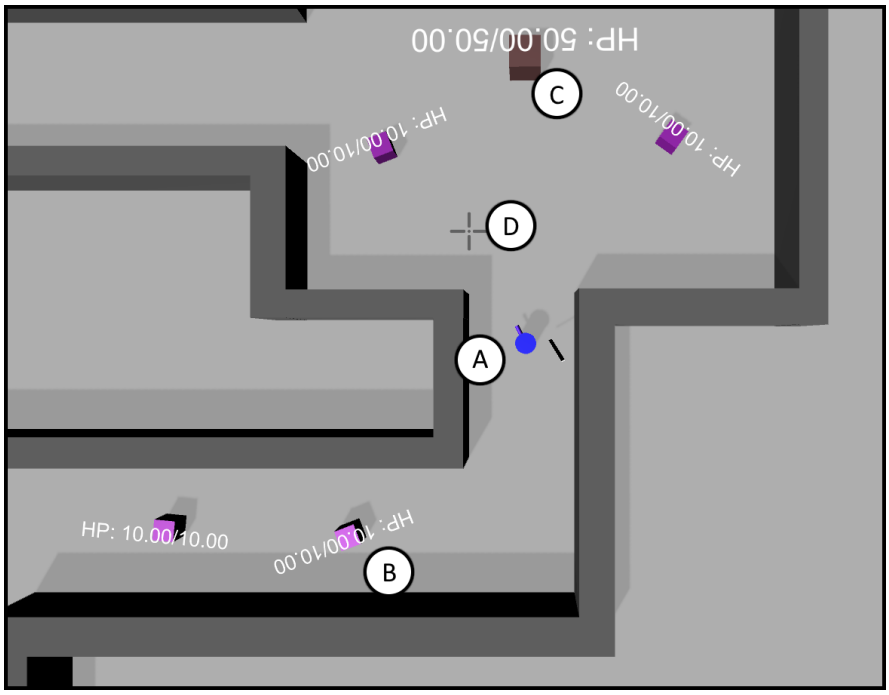


FIGURE 3.1: A screenshot of Simple *Skyrim*. 4 point of interested have been marked. A: the player character. B: a spider enemy. C: a bear enemy. D: the mouse cursor

line to the mouse cursor. The bow can be fired an unlimited number of times, as there is no ammunition system. The sword deals more damage than the bow. The player equips only one weapon at a time, and has the option to instantly switch between the two at any time. Any damage dealt to an enemy is subtracted from the enemy's health point total. The player can also enter a "sneak" gait. While sneaking, the player moves at half speed, but the enemies' detection radii are massively decreased (enemy properties will be discussed next). If an enemy is not currently in its combat state and the player is sneaking, then the player can launch a "sneak attack" on an enemy with either the sword or the bow. A "sneak attack" with a sword deals three times the normal sword damage, whereas a "sneak attack" with the bow deals only two times the normal bow damage. A damage table is provided in table 3.1. Note that a sneak attack will cause the enemy to enter its combat state, so a player cannot attain two consecutive sneak attacks. Player have a set amount of health points, but for our experiments, we ignore player health and assume the player cannot die.

	Normal	Sneak Attack
Sword	10	30
Bow	5	10

TABLE 3.1: Damage dealt by the player's weapons

Enemy	Difficulty	Health
Spider	EASY	10
Wolf	MEDIUM	20
Bear	HARD	50

TABLE 3.2: Enemy Statistics in Simple *Skyrim*

There are three kinds of enemies: spiders, wolves, and bears. Each has its own difficulty rating and a set number of health points. The statistics for each are outlined in table 3.2. Every enemy is either in an idle state or in a combat state. While in an idle state, enemies stand still at a predefined home location or move towards their home location if they have left it. Each enemies has a set detection radius around them. If the player enters this radius, the enemy will enter its combat state. The player can decrease this detection radius by entering sneak mode. An enemy will also enter combat state if they take damage from the player. In combat state, an enemy will continuously pursue the player so long as they are within the detection radius. The enemy will also attack the player once close enough. Since we assume the player cannot die, this attack does not actually deal any damage. If the player manages to move outside the enemy's detection radius and remain outside the radius for 5 seconds, then the enemy will return to its idle state. If the enemy reaches 0 health points due to the player dealing damage, then the enemy dies and is removed from the map.

3.1.2 Events in Simple *Skyrim*

In the follow section, we use the variable d to denote a difficulty rating, which could be EASY, MEDIUM, or HARD. The variable r is a boolean variable taking on TRUE or FALSE.

We define the following events for our game:

- *CombatStart*(d, r): A monster of d difficulty entered its combat state from idle state. r is TRUE if the player is wielding a ranged weapon, and is FALSE if the player is wielding a melee weapon.

- *EnemyDeath*(d): A monster of d difficulty has died.

- *SneakInEnemyRange*(d): the player is currently in sneak mode and has entered the normal detection radius of an enemy of difficulty d without triggering the enemy's combat state. Recall that an enemy has massively decreased detection radius while the player is in sneak mode. This means that a sneaking player can enter and exit the original detection radius of an enemy without triggering the combat state.

- *Retreat*(d): There are two possible triggers for this event. One, a monster of difficulty d has transitioned from combat state to idle state. Two, the player is sneaking and has left the enemy's original detection radius. (This second trigger can only happen folowing in *SneakInEnemyRange* event.)

- *SneakAttack*(d, r): The player landed a sneak attack on an enemy of difficulty d. r is TRUE if the sneak attack was performed with a ranged weapon and FALSE if it was performed with a melee weapon.

Two events are equal if they share the same identifier and arguments. This brings us to a total of 21 unique events. Note that in this formulation, if we have multiple spider enemies of EASY difficulty, we cannot differentiate *EnemyDeath* events between either of them. This abstraction allows us to focus on the type of enemy instead of a specific instance of an enemy.

Our game automatically registers these game events when the trigger occurs.

3.1.3 Map Layouts

We constructed 5 different game maps in Simple *Skyrim*. Their layouts are given in figures 3.2. The circular P represents the player's starting location. The squares S, W, B

represent the home locations of spiders, wolves, and bears respectively. And the odd shaped G represents the goal.

3.2 Playstyle Demonstrations

We outlined 6 possible playstyles in Simple *Skyrim* with informal directives (see table 3.3). These playstyles are representative of the types of playstyles we might encounter in the actual *Skyrim* game. We assume that the training demonstrations follow only one of these outlined playstyles, and that the demonstrating player does not switch playstyles during the demonstration.

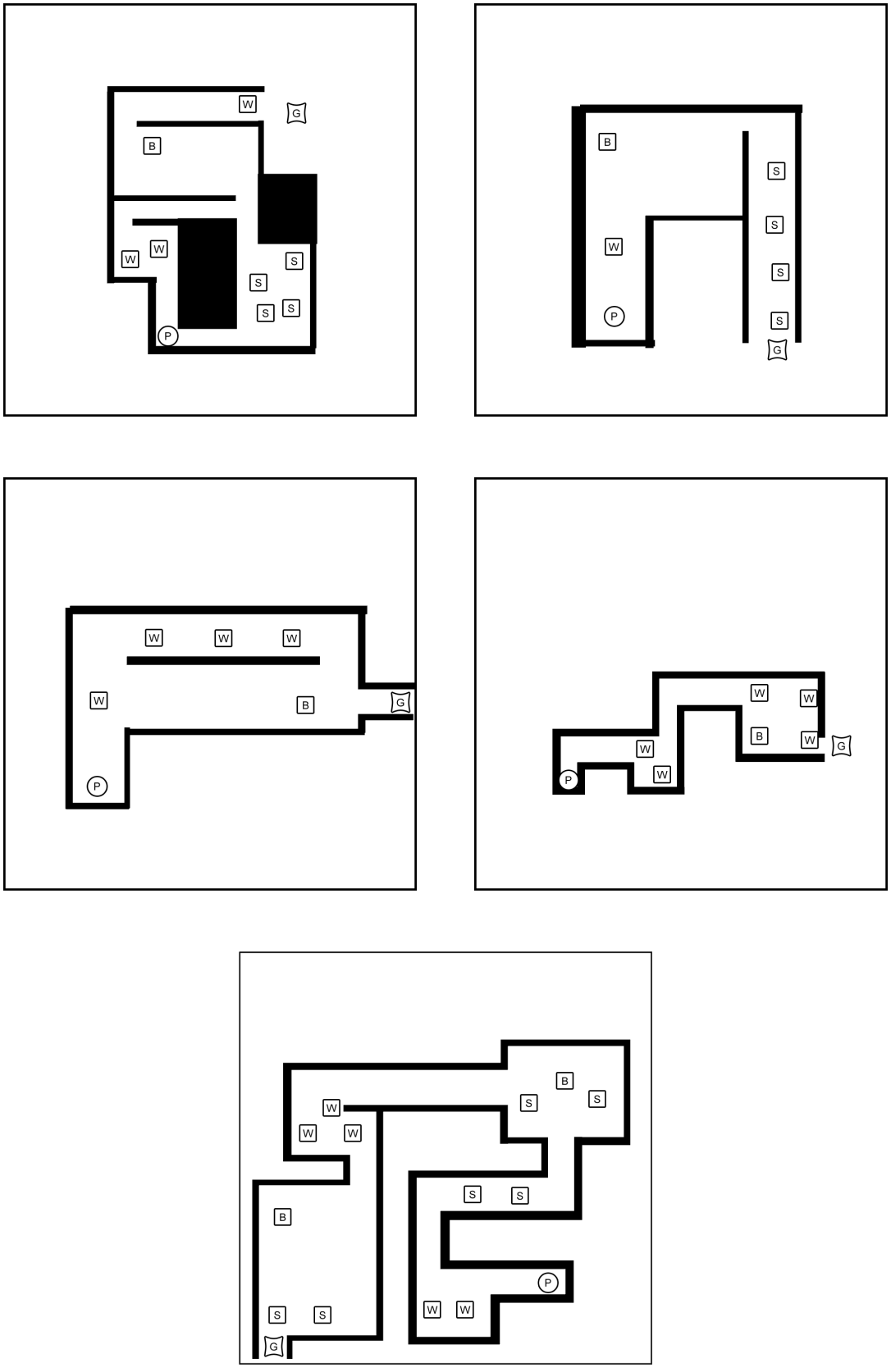
We collected demonstration data in the following way: For maps 1-4, we performed two human demonstrations of each playstyle on the map for a total of 48 demonstrations. For map 5, we performed one demonstration of each playstyle for an additional 6 demonstrations. We used the demonstrations collected from maps 1-4 as our training dataset for offline playstyle learning, and we reserved the data from map 5 to be a test set of online event sequences.

These demonstrations are of human players generally following the guidelines set forth by our informal descriptions. As a result, demonstrations of the same playstyle on the same map may not necessarily be identical. We provide examples of the recorded event sequences in table 3.4. The provided examples also demonstrate how the same playstyle can vary on the same map.

3.3 Clustering Demonstrations into Playstyle Classes

We ran our algorithm to cluster activities (as described in section 2.5) on the 48 demonstrations obtained from maps 1-4 in order to learn playstles from demonstration. We evaluated the results of the clustering using the metrics *total entropy of a clustering* and *Adjusted Rand Index* (ARI). We use these metrics to determine the following hyperparameters in our clustering algorithm: the n in the activity histogram (mentioned in section 2.4), the κ_s in the similarity metric sim (mentioned in section 2.5.1), whether

FIGURE 3.2: Simple *Skyrim* Maps. Top-Left: Map 1, Top-Right: Map 2, Middle-Left: Map 3, Middle-Right: Map 4, Bottom: Map 5



Playstyle Name	Description
MELEE	Defeat all enemies in any order with a melee weapon, even if the enemy is not on the shortest length path to the goal. Once all enemies are defeated, move toward the goal. Do not use sneak mode.
ARCHER	Move toward the goal according to the shortest length path. Defeat any enemies obstructing the path with a ranged weapon. Initiate all combat from outside of the enemy's detection radius. Prioritize low difficulty enemies before higher difficulty enemies. Do not use sneak mode.
SNEAK	Move toward the goal according to the shortest length path. Use sneak mode to avoid detection whenever possible. If combat is unavoidable (i.e. the enemy's detection radius completely obstructs the path), initiate combat with a melee sneak attack. Use only melee weapons during combat.
MMO	Move toward the goal according to the shortest length path. If an enemy is near the path, initiate combat by launching a ranged attack. Wait for the enemy to enter melee range, and then defeat the enemy with melee attacks. If there are multiple enemies in a group, only engage one enemy at a time in this manner. Also if there are multiple enemies in a group, prioritize the lowest difficulty enemies before the harder enemies. If the weakest enemy cannot be easily targeted with a ranged attack, use sneak mode to move into a more advantageous position to launch the initial ranged attack. Otherwise, do not use sneak mode. *The playstyle name is a reference to a common tactic in Massively-Multiplayer Online Role-Playing Games (MMORPGs): players lure away a single enemy from a group of enemies, so they can focus on only one enemy at a time.*
BOSS	Defeat all enemies on the map with a melee weapon before moving towards the goal. If there is a HARD difficulty enemy, use sneak mode to initiate combat with it, starting with a sneak attack. Exit sneak mode immediately after landing the sneak attack.
SPEED	Move toward the goal according to the shortest length path. Do not use sneak mode. Ignore all combat, even if enemies give chase. (Enemies will initiate combat when the player moves past them, but the player should ignore them.)

TABLE 3.3: Playstyle guidelines for Simple *Skyrim*

Index	Map 1, MELEE, Demo 1	Map 1, MELEE, Demo 2
0	$CombatStart(MEDIUM, FALSE)$	$CombatStart(EASY, FALSE)$
1	$CombatStart(MEDIUM, FALSE)$	$CombatStart(EASY, FALSE)$
2	$EnemyDeath(MEDIUM)$	$EnemyDeath(EASY)$
3	$EnemyDeath(MEDIUM)$	$CombatStart(EASY, FALSE)$
...	...	...

TABLE 3.4: Examples of demonstration event sequences. The left demonstration is the first demonstration of playstyle MELEE on map 1. The right demonstration is the second of playstyle MELEE on map 2. Note that, despite being the same playstyle on the same map, they are not identical due to multiple existence of paths to the goal in map 1. The human demonstrator opted to go down different paths between the two demonstrations.

union or concatenation is used in the formula (also mentioned in section 2.5.1), and whether or not we deleted duplicate demonstrations.

Section 3.4 reviews how our metrics are calculated. Section 3.5 describe the hyper-parameters we tuned for our clustering. Section 3.6 examines how well the each set of selected hyper-parameters performs in terms of clustering. Section 3.7 presents the final clustering that we obtained after validation. Section 3.8 discusses how well our learned clusters accurately classify the online test data from map 5.

3.4 Metrics

3.4.1 Total Entropy of a Clustering

Let N be the number of demonstrations. Let the true classes (or clusters) be given by C. In our case, $C = \{MELEE, ARCHER, SNEAK, MMO, BOSS, SPEED\}$, and we have 8 demonstrations per class.

Let our learned classes be the set Ω. Let N_ω be the number of demonstrations in class $\omega \in \Omega$. The total entropy of a clustering is thus define as:

$$H(\Omega) = \sum_{\omega \in \Omega} H(\omega) \frac{N_\omega}{N} \tag{3.1}$$

where $H(\omega)$ is the entropy of cluster ω, which is given by:

$$H(\omega) = -\sum_{c \in C} \frac{\omega_c}{N_\omega} \log_2 \frac{\omega_c}{N_\omega} \tag{3.2}$$

where ω_c is the number of demonstrations in ω that have a true labeling of class c.

A perfect clustering will have an entropy of 0, and a cluster of demonstrations which do not belong together will have a positive value. Entropy effectively measures how many mistakes there are in a cluster.

3.4.2 Adjusted Rand Index

The Adjusted Rand Index is a metric for evaluating clustering which rewards putting pairs of items that belong together in the same cluster and putting items that don't belong together in separate clusters. It also punishes putting items that belong together in separate clusters, and putting items that don't belong together in the same cluster.

ARI takes values from $[-1, 1]$, where negative values are worse than random labelings, and positive values are better than random labeling.

Suppose we have n data points and are given two different clustering (or partitions) of the points (e.g. one being ground truth, the other produced by our algorithm). Specifically, let clustering $X = \{X_1, X_2, \ldots, X_r\}$ and clustering $Y = \{Y_1, Y_2, \ldots Y_s\}$. We define a contingency table $[n_{ij}]$ where entries are given by:

$$n_{ij} = |X_i \cap Y_j| \tag{3.3}$$

Then, the sum of the rows are given by a_i and the sum of the columns are given by b_j as such:

$$a_i = \sum_{j=1}^{s} n_{ij} \tag{3.4}$$

$$b_j = \sum_{i=1}^{r} n_{ij} \tag{3.5}$$

ARI is given by:

$$\frac{\sum_{i=1}^{r}\sum_{j=1}^{s}\binom{n_{ij}}{2} - \frac{1}{\binom{n}{2}}\sum_{i=1}^{r}\binom{a_i}{2}\sum_{j=1}^{s}\binom{b_j}{2}}{\frac{1}{2}\left(\sum_{i=1}^{r}\binom{a_i}{2} + \sum_{j=1}^{s}\binom{b_j}{2}\right) - \frac{1}{\binom{n}{2}}\sum_{i=1}^{r}\binom{a_i}{2}\sum_{j=1}^{s}\binom{b_j}{2}} \tag{3.6}$$

We use ARI as a metric in addition to entropy because entropy goes towards 0 (a good evaluation) if we create a class for every demonstration. However, this defeats the purpose of clustering, so we cannot rely on only entropy as an evaluation metric. ARI is negatively impacted if our algorithm needlessly creates too many clusters. ARI will thus put the entropy score in perspective.

3.5 Hyper-Parameters

3.5.1 Normal κ_s vs Weighted κ_s

Recall the similarity metric (equation 2.1) described in section 2.2.

$$\mathsf{sim}(A,B) = 1 - \sum_{s \in S_A, S_B} \kappa_s \frac{|f(s \mid H_A) - f(s \mid H_B)|}{f(s \mid H_A) + f(s \mid H_B)} \tag{2.1 revisited}$$

The equation has normalizing factor κ_s for each term of summation.

Suppose S is the set of unique substrings with non-zero counts in the two activities being compared. The *normal* κ_s term is:

$$\kappa_s = \frac{1}{|S|} \tag{3.7}$$

This is a normalizing factor which gives equal weight to every substring.

We proposed a *weighted* κ_s factor instead, which gives more weight to longer substrings. For substring s, we set κ_s to be:

$$\kappa_s = \frac{|s|}{\sum_{s \in S} |s|} \tag{3.8}$$

where $|s|$ means the length of the substring s. This will intuitively increase the similarity of two activities that share long substrings.

The empirical effect of the two different κ_s is presented in table 3.5.

3.5.2 Using Union vs Concatenation

Again recall the similarity metric:

$$\text{sim}(A, B) = 1 - \sum_{s \in S_A, S_B} \kappa_s \frac{|f(s \mid H_A) - f(s \mid H_B)|}{f(s \mid H_A) + f(s \mid H_B)} \qquad \text{(2.1 revisited)}$$

The originally intended summation is over all elements of S_A and all elements of S_B. Effectively, if S_A and S_B are lists, then the summation is over $S_A + S_B$, the concatenation of the two lists. However, this will give double the weight to substrings which appear in both S_A and S_B.

We propose instead the use of $S_A \cup S_B$, which will cause each substring to only have one term in the summation instead of a potential 2.

We show the empirical effect of changing between list concatenation (Concat) and set union (Union) in table 3.5.

3.5.3 Deleting Duplicate Demonstrations

Of the 48 demonstrations from maps 1-4, there are 7 which are exact duplicates of another in the set. Intuitively, including duplicate demonstrations will cause the discovered clusters to capture more focused or rigid patterns in the activities. Since we are trying to capture playstyles which are full of human-error and variation, it makes sense to remove duplicate demonstrations to encourage more scattered clusters.

The empirical impact of removing duplicate demonstrations is given in table 3.5.

3.5.4 Changing n in the Activity Histogram

In section 2.4, we discussed that the value of n in the activity histogram tends to be domain dependent. Higher values of n capture more temporal relations between events,

	Duplicate Demos Removed			Duplicate Demos Kept		
Hyper-Parameters	Classes Found	Total Entropy	ARI	Classes Found	Total Entropy	ARI
Normal κ_s, Concat	10	0.403	0.502	13	0.393	0.412
Normal κ_s, Union	13	0.341	0.396	15	0.365	0.332
Weighted κ_s, Concat	12	0.283	0.479	15	0.240	0.400
Weighted κ_s, Union	12	0.222	0.495	14	0.354	0.365

TABLE 3.5: Clustering performance of various hyper-parameters when $n = \{1, 2, 3\}$.

whereas low n focuses more on the frequency of particular events. Recall that we used the notation $n = \{1, 2, 3\}$ to describe an event histogram that keeps track of substrings of multiple lengths (in this example, lengths 1, 2, and 3).

We show an example of how the value of n can affect clustering performance in table 3.6.

3.6 Validation Results

We first fixed $n = \{1, 2, 3\}$ for our activity grams and varied the hyper-parameters described above: κ_s, set union vs list concatenation, and duplicate deletion. The results are summarized in table 3.5. We are primarily interested in the lowest entropy clustering, using ARI and number of classes found to help put the entropy metric in context.

Keeping duplicate demonstrations tends to have varying effects on the total entropy, but generally increases the number of clusters found. We would prefer to keep the number of clusters lower so that we can generalize the demonstrations better, so we opt for always removing duplicate demonstrations.

Also seen in the table is that the normal κ_s formulation tends to have higher entropy when comapred to the weighted κ_s. One trade off is that we get more clusters with weighted κ_s, but the large reduction in entropy seems to be worthwhile.

Similarly, using set union over list concatenation seems to increase the number of clusters found when using normal κ_s, but decreases the number of clusters when using weighed κ_s.

n Choice	Classes Found	Total Entropy	ARI
$n = 1$	7	0.329	0.734
$n = \{1, 2, 3\}$	10	0.403	0.502

TABLE 3.6: Clustering performance of different n in the activity histogram, using normal κ_s and list concatenation.

n Choice	Classes Found	Total Entropy	ARI
$n = 1$	8	0.408	0.589
$n = \{1, 2\}$	11	0.399	0.476
$n = \{1, 2, 3\}$	12	0.222	0.495

TABLE 3.7: Clustering performance of different n in the activity histogram, using weighted κ_s and set union.

We then looked at the impact of varying the n in the activity histogram. These results are summarized in tables 3.6 and 3.7. Table 3.6 uses the original formulation with normal κ_s and list concatenation, whereas table 3.7 uses the weighted κ_s and set union which we found to be best.

The results show that $n = 1$, or a naive vector space model, actually does fairly well with the normal κ_s and list concatenation. It discovers 7 different activity classes, which is fairly close to the true value of 6 classes. This suggests that the playstyles we outlined are well characterized just by how many types of one game event are triggered.

However, this set of hyper-parameters did produce a clustering with higher entropy than when we used $n = \{1, 2, 3\}$ with weighted κ_s and set union. This is likely because more clusters were produced in the latter setup, but also because $n = 1$ fails to capture any temporal structure of the game events. We also see that $n = 1$ is not as good with weighted κ_s and set union, so the performance of $n = 1$ seems to be rather variable.

Depending on how much we value generalization of the demonstration data, this clustering may be more appealing as it minimizes the number of clusters discovered. For our purposes though, we opted to minimize entropy.

Class Label	Demonstrations
0	map1_BOSS_01, map1_BOSS_02, map2_MELEE_01, map2_BOSS_01, map3_BOSS_01
1	map2_ARCHER_01, map2_ARCHER_02, map3_ARCHER_01, map4_ARCHER_01
2	map4_SNEAK_01, map4_SNEAK_02, map4_BOSS_02
3	map1_MMO_01, map1_MMO_02, map2_MMO_01, map2_MMO_02, map3_MMO_01
4	map1_MELEE_01, map4_MELEE_01, map4_MELEE_02
5	map1_SPEED_02, map4_SPEED_01, map4_SPEED_02
6	map2_SNEAK_01, map2_SNEAK_02, map3_SNEAK_01
7	map1_ARCHER_01, map1_ARCHER_02, map4_ARCHER_02
8	map4_MMO_01, map4_MMO_02
9	map1_MELEE_02, map3_MELEE_01, map3_MELEE_02
10	map1_SPEED_01, map2_SPEED_01, map3_SPEED_01
11	map1_SNEAK_01, map1_SNEAK_02, map4_BOSS_01

TABLE 3.8: Clustering of Demonstrations from Maps 1-4

3.7 Resulting Clustering

Based on the results above, we decided to use $n = \{1, 2, 3\}$ for our activity histogram, weighted κ_s, the set union instead of list concatenation, and deleting all duplicate demonstrations. This combination of hyper-parameters minimized the total entropy.

With these hyper-paramters, the clustering produces 12 classes. We show how the 41 demonstrations were partitioned in table 3.8. The demonstration names are given by the map they were performed on, the playstyle they demonstrate, and an identifier number.

3.8 Evaluating Classification Ability

We now use our learned playstyle classes to perform activity classification on the demonstrations from map 5. For each event sequence τ, we compute its classification with

Replay Name	True Playstyle	Predicted Class Label	Support Density
map5_MELEE	MELEE	4	1.0
map5_ARCHER	ARCHER	1	1.0
map5_SNEAK	SNEAK	6	1.0
map5_MMO	MMO	3	1.0
map5_BOSS	BOSS	0	0.8
map5_SPEED	SPEED	5	1.0

TABLE 3.9: Classifcation of Map 5 Demonstrations

equation 2.6. We can then evaluate our labeling of the new demonstrations according to ground truth.

We present the results of the classification in table 3.9. Our metric of ARI is no longer useful in this scenario because we have six test demonstrations and each one is different classes. ARI requires there to be more than one demonstration per class to produce a score.

The entropy of this assignment is 0, meaning that we have correctly put all of the demonstrations into their own class. But placing each demonstration into its own class is not particularly informative. We are interested in whether each new demonstration was placed into a class containing demonstrations of the same playstyle.

We thus rely on the *support density* of correct labels in the cluster each demonstration was placed. For a activity instance with true label c and predicted label ω, *support density* is the fraction of instances in cluster ω which have true label c. For example, if a new demonstration τ has a correct playstyle labeling of SNEAK and was classified as class label 2, we would compute the fraction of SNEAK demonstrations in class 2 (which turns out to be 0.667). We summarize the results in table 3.9.

With the exception of map5_BOSS, all demonstrations fell into a class with a support density of 1.0, meaning they were all grouped with demonstrations of the same playstyle. Even map5_BOSS had a support density of 0.8, where 4 out of the 5 demonstrations in the class had the same BOSS playstyle.

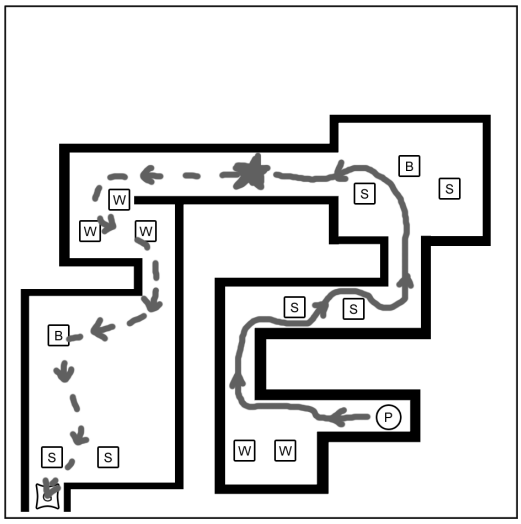


FIGURE 3.3: A demonstration of a player switching playstyles while going through Map 5. The solid gray line represents the player following the SNEAK playstyle, whereas the dotted gray line represents the player following the MELEE playstyle.

3.9 Responding to Switching Playstyles

We now present one more demonstration that we had used as a test online activity sequence. This demonstration took place on map 5, and the path taken by the player is illustrated in figure 3.3. Initially, the player started the demonstration by following the playstyle guidelines for SNEAK. This is shown in the figure as a solid gray line. When the player reached the star icon, he switched to following the guidelines for a MELEE playstyle. The path of the MELEE playstyle behavior is illustrated in the figure as a dotted gray line.

For this particular test, we investigated if discounting older activity substrings with discounted histograms (described in section 2.9) impacted how quickly the activity classifier can react to changes in playstyle.

The demonstration was 24 game events long. The switch from SNEAK to MELEE playstyle occurred after the 12th game event. We summarize in table 3.10 the total delay in measured in number of events until the classifier was able to make a new classification. Below $\gamma = 0.7$, the classifier actually misclassified the sequence into a class that has 0 support density in either MELEE or SNEAK.

Discount γ	Delay in Class Switch
1.0	11
0.9	7
0.8	5
0.7	5

TABLE 3.10: Delay (in number of events) until classification switch

To better understand this, we present a graphical representation of the classifier's belief in figure 3.4. Instead of a single classification, we asked the classifier for a probability distribution of its classification (equation 2.12). We then looked at the support density of MELEE and SNEAK for each class, and computed a weighted sum of support densities according to the class probabilities. We then computed the ratio of the MELEE probability as compared to the SNEAK probability, giving us the likelihood of a MELEE playstyle as opposed to SNEAK. A ratio of 1.0 means the classifier highly believes MELEE over sneak. A ratio of 0.0 means it believes SNEAK over MELEE. A ratio of 0.5 means the classifier equally believes that the playstyle could MELEE or SNEAK. A perfect classifier should see a perfect step from 0.0 to 1.0 at the 13th event, when the playstyle switches from SNEAK to MELEE.

We can see that without discounting, the classifier fails to firmly change its classification even by the end of the demonstration, sitting at a ratio of 0.557. Meanwhile, $\gamma = 0.8$ allows the classifier to cross the 0.5 ratio threshold by 17th event.

3.10 Shortcomings

One benefit and shortcoming with our approach is that clustering via dominant sets produces variable number of clusters. The benefit of this is that, unlike a clustering algorithm like *k*-means [19], we do not need to guess how many clusters we could possibly need. The downside is that our algorithm can create a needless number of clusters. In fact, in the worst case it could create a cluster for each demonstration. Such a clustering has an entropy of 0, but is practically useless because it does not identify any similar structure between activities. Our result clustering had 12 classes, which is twice

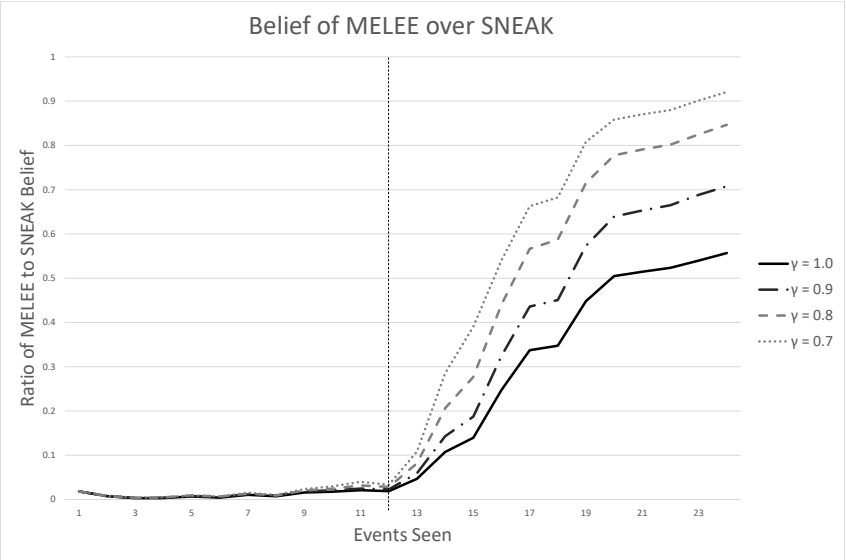


FIGURE 3.4: The likelihood of a MELEE playstyle compared to a SNEAK playstyle as the classifier saw more events. The playstyle switch occurred after the 12th event. A perfect classifier should see a step from 0.0 to 1.0 at the 13th event.

the number of ground truth classes. This left us with 3 demonstrations per class on average, which may not be sufficient to properly generalize the playstyle.

3.11 Summary

In the context of Simple *Skyrim*, our learning system demonstrates exemplary performance in identifying playstyle classes and classifying new gameplay instances. We introduced the concepts of a weighted normalizing factor κ_s and using union instead of concatenation as potential hyper-parameters to our unsupervised learning algorithm. We showed that they are effective in reducing the overall entropy of the produced clustering. We also showed how removing duplicate demonstrations in the training data can help loosen the resulting clusters, which helps in the context of human playstyles which are prone to high variations. We determined that an activity histogram with $n = 1$ (a naive vector space model) could work well as a hyper-parameter value since

it empirically created the smallest number of clusters out of all our hyper-parameter configurations, but we ultimately opted for $n = \{1, 2, 3\}$ since it had lower entropy. Our evaluation of the learned clusters on test demonstrations from Map 5 showed near perfect classification rates, as evidenced by high support densities of true labels. And finally, we showed how the inclusion of a discount factor in an online setting can encourage the classifier to quickly change its classification in response to a significant playstyle switch.

Chapter 4

Cooperative Planning for Non-Player Characters

In this chapter, we provide an example of how the Player Behavior Classifier can be incorporated in an NPC planner to help with cooperative planning, particularly in the context of the full game of *Skyrim*. Our work builds upon the *Skyrim* NPC planning framework presented in John Drake's thesis [5] (in preparation). We start by reviewing NPC planning, which typically consists of heuristic graph search. Then, we present the problem of planning for cooperative tasks and discuss some of the complications involved. We next explain how we integrated our Player Behavior Classifier with an NPC planner to enable cooperative planning, particularly in the context of a cooperative *Skyrim* scenario. We also discuss how we incorporated search techniques like Training Graph Heuristic [6] and Multi-Heuristic A* [1] to reduce planning times. Finally, we present the results of planning with our integrated system and analyze the cooperative plans that are produced.

4.1 Heuristic Graph Search

Let us first consider an NPC AI being presented with a challenge that can be solved just by the AI. For example, in the context of *Skyrim*, suppose there is a hallway of spider monsters and an AI player must arrive at the other end of the hallway without dying.

Such a task is formulated as a graph search problem. Game states are represented as nodes in a graph, and the NPC must find a plan of actions leading through the game states that will allow it to arrive at a desired goal state which solves the problem. These actions are inputs like "move forward" or "swing sword." Each action is assigned a cost, such as the time taken to perform the action. Actions that get the NPC killed would cost infinite time. By searching the graph for the least cost path to the goal, the NPC finds a sequence of actions that avoid death while also reaching the other side of the hallway in the shortest amount of time. The Dijkstra search algorithm would find the least cost path [4].

A single node in the graph is a game state, which could be a dump of many game variables, such as: the NPC's health, the NPC's x-, y-, z-positions, the NPC's gait (walking, running, sneaking), enemy healths, and enemy positions. The high dimensionality of the state creates a very large graph to search, making it very difficult to find an optimal solution in a short planning time. We thus introduce the idea of a heuristic, which approximates how close each game state is to the goal, i.e. the cost-to-goal. A reasonable heuristics must be *admissible*, meaning it must underestimate the true cost to the goal. With an admissible heuristic, an NPC can prune the search space by focusing on states which bring it closer to the goal, all while guaranteeing that the resulting solution is optimal. An example of a simple admissible heuristic is the shortest path distance from the NPC agent to the goal location, ignoring all other state variables. This will lead the planner to favor actions that physically move the NPC toward the goal. A* search is the basic search algorithm that puts heuristics into practice [14].

A planner can further prune the space by inflating the heuristic value by some $\epsilon > 1$. In doing so, the planner sacrifices optimality, but typically only up to a known suboptimality bound. In return, the planning times are dramatically reduced since a larger portion of the search graph becomes unfavorable when evaluated by the inflated heuristic. Weighted A* search is general name for basic inflated heuristic search algorithm.

In some cases, a heuristic can guide a search into a portion of the state space that is a dead end. This pocket of state space is often referred to as a *local minima*, and requires the

search to expend a large amount of effort exploring the local region before it backtracks enough to find a way around the dead end. Consider for example a hallway with a very strong bear enemy, and the NPC's goal is to reach the other side of the bear. A shortest path heuristic would pull the NPC's search right into the bear, whereupon the bear would kill the NPC, preventing the search from advancing any further. Due to the heuristic, the search will keep trying to explore alternatives in the nearby region around the bear to try and circumvent the dead end. This could be a waste of effort, especially if the true solution might be to turn back and find another path around the bear. With regards to heuristics then, we are thus interested in informative heuristics that can avoid local minima.

4.1.1 Training Graph Heuristic

One heuristic of interest is the Training Graph Heuristic (T-Graph) [6], which is based on the Experience Graph (E-Graph) Heuristic [23, 24]. The E-Graph heuristic takes a prior demonstration of a plan through the search space and produces a heuristic value which attempts to drive the search along the same trajectory. The T-Graph Heuristic builds on this by providing a smooth gradient of heuristic values along the demonstrated path. The T-Graph Heuristic was incorporated in NPC planning for *Skyrim* in [6] and produced favorable results in pulling an NPC agent around an unkillable bear enemy.

4.1.2 Multi-Heuristic A*

To make the search faster, we can also leverage the use of several heuristics at once. Multi-heuristic A* is a search algorithm which takes advantage of multiple heuristics to escape local minima when one heuristic fails [1]. The algorithm advances the search by doing a round-robin across the different heuristics. If one heuristic happens to lead the search into a local minima, another heuristic can focus the search towards a different direction out of the local minima. These heuristics are allowed to be inadmissible, meaning that they can overestimate the cost-to-goal and consequently drive the search in a

wrong direction. For example, an inadmissible heuristic can always push the NPC to move backwards, even if moving backwards pulls the NPC away from the goal. However, such a heuristic could be useful in the hallway with a bear scenario, because it will cause the search to explore options other than running into the bear. Normally, inadmissible heuristics can result in an incomplete search, but Multi-Heuristic A* still guarantees completeness and bounded suboptimality by using an admissible anchor heuristic to control the inadmissible heuristics.

The benefits of using Multi-Heuristic A* with NPC planning was investigated in [7].

4.2 Toward Cooperative Planning

Thus far, we have only considered planing for NPCs in a vacuum. In other words, the NPC is solving problems that rely only on the NPCs abilities. We now want to consider how an NPC can plan to perform cooperative tasks with human players. These are tasks which require the NPC and a human player to execute actions which could depend on each other. We cannot just rely on the NPC optimizing its own rewards or costs, but rather, it needs to know what the player is doing in order to complete the task.

To motivate cooperative planning, consider the following "capture the flag" scenario in Skyrim: there is a chest of treasure which the player or NPC must obtain, but there are multiple bandits guarding the chest. If the NPC or player approaches for frontal combat, they would be overwhelmed by the bandits' strength. One cooperative strategy would be for one of two players to lead the bandits away, while the other steals the treasure. This kind of coordinated effort goes beyond what an NPC can plan in isolation. In fact, in an solo planner's state space, this plan is not even feasible solution because there is nothing in the planner that says the treasure will obtained after the NPC leads the bandits away. This type of plan only becomes feasible when the planner is capable of knowing that the other player will steal the treasure in the meantime.

We are thus interested in providing a model of the human player that can predict the player's trajectory during search. This is where our classification and modeling

system come in: we analyze the player's behaviors and create models of them, so that the planner can predict player actions during the search.

One issue that arises from adding a player model is that the player's state is now added to the search space, thereby making the NPC's search extremely difficult. This is where T-Graph heuristic and Multi-Heuristic A* will help maintain reasonable planning times.

4.3 A Cooperative *Skyrim* Scenario

We now present the *Skyrim* game scenario which we aimed to solve. This scenario will both illustrate the benefit of cooperative planning and motivate why the Player Behavior Classifier is needed.

First, in terms of gameplay features, the *Skyrim* game operates much like the Simple *Skyrim* game outlined in section 3.1. The main game features we are concerned about are melee attacks and the sneak mode. The scenario map we are interested is a modified dungeon from the official game. The dungeon map is illustrated in figure 4.1, and the specific scenario we crafted for map is given in figure 4.2. The enemies, the start (S) of the player and NPC, and the goal position (G) are marked in the figure. The goal of the scenario is for the NPC to reach the goal location in the shortest time possible (i.e. the cost for the planning problem is time).

The enemies present in this scenario are a bear (B), basic wolfs (W), and super wolfs (W*). The bear requires both the NPC and player working together to defeat it. If either the player or NPC approaches alone, the bear will win in a fight. Sneaking has no effect on the bear. The bear will engage a character in combat regardless if they are in sneak mode.

The basic wolves (W) are simple enemies which can be defeated by the player or NPC in one attack. The super wolves (W*) cannot be defeated, and will instead defeat the player or NPC in one attack. The super wolves, however, cannot see any character that is in sneak mode. They effectively have a detection radius of zero against a character that is sneaking.

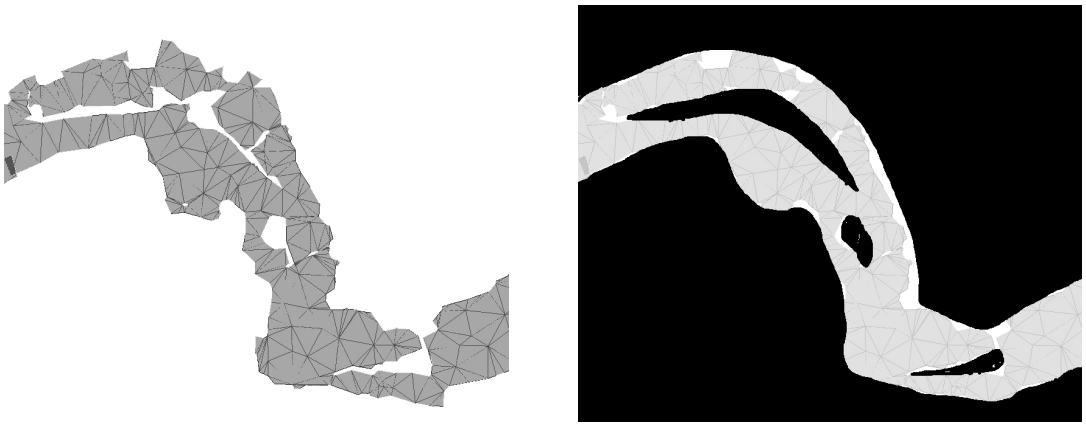


FIGURE 4.1: *Skyrim* scenario map. Left, the raw NavMesh. Right, a clarified version of the map, where obstacles are colored black.

At a high level, the scenario can be summarized as follows. There are two paths to goal: one through the hallway with the bear and one through the hallway with super wolves. The bear path can only be traversed if the player and NPC cooperate in fighting it. The super wolves path can only be traversed via sneak mode. When sneaking, a character moves very slowly, and so the wolves path will take a much longer time to traverse if taken. Thus the shortest time path is actually through the bear, if the bear can be defeated.

To know if the bear can be defeated, the NPC must be able to predict the player's intention. If the player wants to fight the bear, then the NPC can assist to unlock the shorter path. If the player would rather avoid the bear, then the NPC must also plan accordingly. Note that it is not sufficient to randomly guess what the player is going to do (fight the bear or sneak around). Suppose the NPC wrongly assumes the player will avoid the bear when in actuality the player will fight the bear, then the NPC will plan for the path away from the bear, and the human player will die fighting alone (likely expecting help from the NPC). Suppose instead the NPC wrongly assumes that the player will fight the bear. Then the NPC will plan into fighting the bear and die alone because the player is actually avoiding the bear. As a result, knowing what kind of the playstyle the player is following is critical to this scenario.

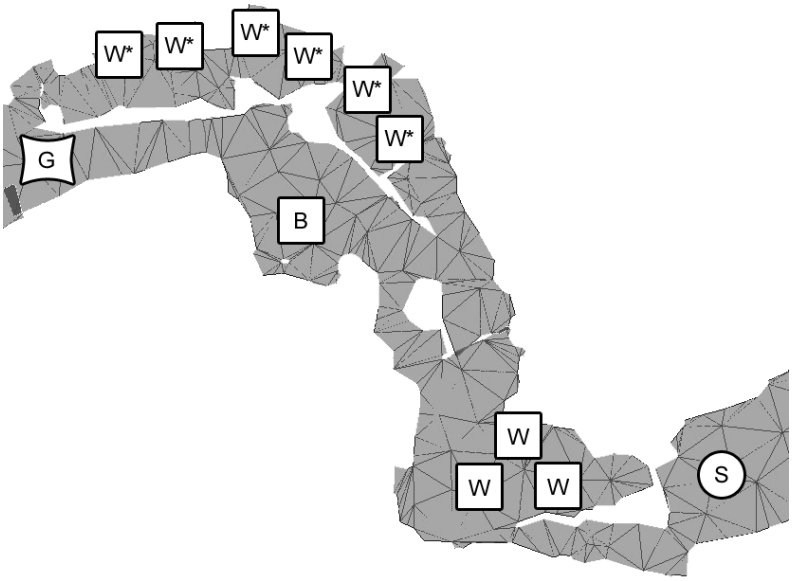


FIGURE 4.2: *Skyrim* scenario map with markers. Circle S is the start position for both the player and NPC. Square W is a basic wolf enemy. Square W* is a super wolf enemy. Square B is a bear enemy. Star G is the goal position.

4.4 Implementation and Integration

4.4.1 *Skyrim* NPC Planning Framework

The details of the *Skyrim* NPC planning framework are more thoroughly described in John Drake's thesis [5]. Here, we provide a brief explanation of the planning framework so that we can better explain how our Player Behavior Classifier integrates with it.

The existing planning framework includes a simple model of the *Skyrim* game coded in C++, which focused primarily on the features of melee attacks and the sneak mode. Planning algorithms can interface with the game model to perform forward simulation of the world for graph search. Each step of the simulation is deterministic, but enemies are coded to respond accordingly to different NPC actions.

The state representation includes game state variables for the player, the NPC, and all the enemies in a map. The actions available are for NPC movement, sneaking, and attacking. Prior to our work, the game model did not include any model for the player, so the player's state data would never change during forward simulation.

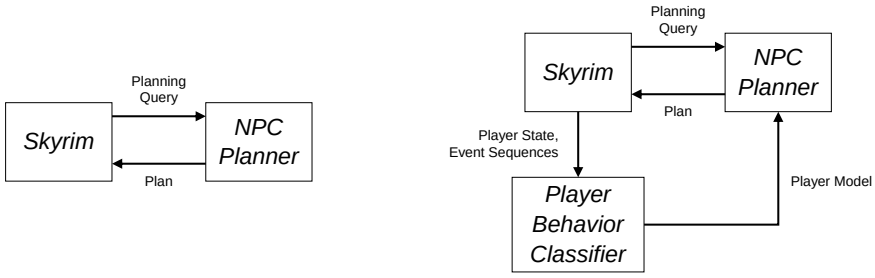


FIGURE 4.3: The addition of the Player Behavior Classifier. Whenever a planning query is sent to the NPC Planner, the Player Behavior Classifier will also send its most likely player model to the NPC Planner.

The framework also uses Bethesda's *Creation Kit* and *Skyrim Script Extender* to constructed a pipeline between NPCs in the *Skyrim* game and the active planner. This pipeline allows the planner to receive planning queries from the game and to send solution plans back to the to the NPCs in game. In most cases, the performance of planning could be evaluated outside of *Skyrim*. *Skyrim* was often only used as a final visualization platform for plans.

4.4.2 Integrated Design

Our goal in integrating our behavior classifier was to provide the NPC planner a player model during its planning stage. The changes we made in the planning pipeline are shown in figure 4.3.

Now, *Skyrim* would continuously send player state and event data to the Player Behavior Classifier. Whenever a planning query was sent to the NPC planner, our Player Behavior Classifier would also send a model of the currently predicted playstyle class to the NPC planner. The planner could then use this during its search.

4.4.3 Running the Behavior Classifier

We defined the same game events as we did in Simple *Skyrim* and configured the *Skyrim* game to record these events using the *Creation Kit* and *Skyrim Script Extender*. We created the described scenario in the game, but with a few minor tweaks. We made the bear and the super wolves weaker, so that a single human player can defeat them. We made

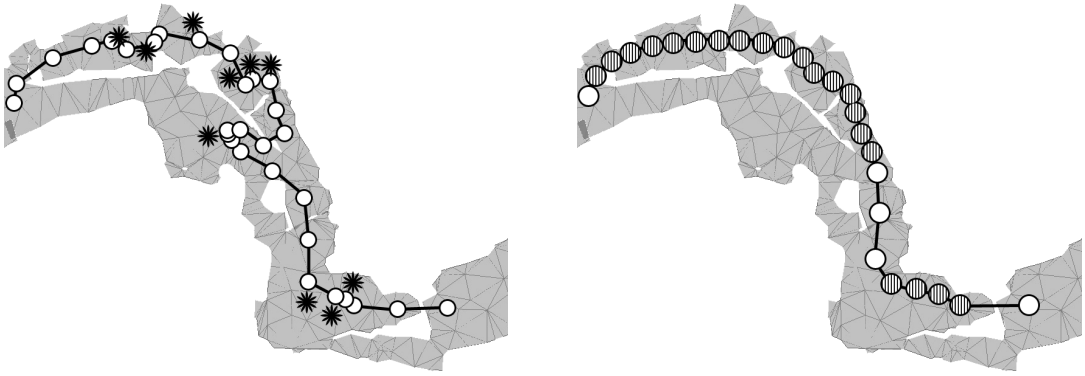


FIGURE 4.4: Example replays of COMBAT and SNEAK in the *Skyrim* scenario. Left is COMBAT and right is SNEAK. The nodes here are not events, but the player's position. The striped nodes indicate the player is sneaking. The black stars indicate an enemy was defeated.

this tweak so that we could focus on recording playstyle demonstrations for the player without an NPC ally.

We recorded demonstrations for two distinctive playstyles: COMBAT and SNEAK. In the COMBAT playstyle, the player defeats all the enemies on his way to the goal. In the SNEAK playstyle, the player uses sneak mode to avoid all combat to reach the goal. The recorded demonstrations included both the event sequence and the full state log. Example demonstrations of COMBAT and SNEAK are given in figure 4.4.

Next, we learned the playstyle from the given demonstrations through activity clustering. This prepared the activity classifier for the next stage, to classify player behaviors in an online setting. From here, whenever we played the *Skyrim* game, the classifier could report which playstyle we were most likely following.

To actual model the playstyles, we chose an exemplar demonstration from the playstyle class and then created a player model which reproduced the demonstration. This became the model we send the planner whenever an NPC planning query was sent. Note that for our scenario, we assumed that the dungeon layout remained the same, so a simple replay of a previous demonstration would suffice. However, in a real setting, the player model would have to be something more sophisticated, such as an n-Gram Predictor trained off all demonstrations in a cluster and a local planner that could figure out how to achieve predicted game events.

After learning and modeling the playstyles, we updated the dungeon to match the exact scenario: the bear was made stronger, the super wolves were now deadly, and the NPC ally was added. We played the *Skyrim* game up according to one of the playstyles (fight everything, or sneak) up to the point after the basic wolves, but before the split between the two paths. This prepped our activity classifier with a few events so that it could make an accurate playstyle prediction.

Finally, we sent the planning query to the NPC Planner. At the same time, the Player Behavior Classifier sent a player model of the predicted playstyle to the NPC Planner. This completed the setup, and now our NPC Planner could attempt its cooperative planning.

4.5 Results

All of our planning results in the following sections were generated on a machine with 2.59 GHz Intel i7-6500U CPU, 8.00 GB of RAM, and 64-bit Windows 10 Professional OS. Our code was compiled with Microsoft Visual Studio 2013 and was run on a single thread.

4.5.1 Feasibility of New Cooperative Plans

We first report the results for when the player is following the COMBAT playstyle, because this will illustrate the feasibility of new cooperative plans. In this case, we took control of the player and defeated the first pack of normal wolves at the start. Then we sent the planning query to the NPC planner. Our Player Behavior Classifier recognized this initial set of player actions as the COMBAT playstyle, and sent the corresponding player model.

First, if the player model is not used in the planner (like how a solitary NPC planner might work), then the resulting plan resembles the one in figure 4.5. The plan in the figure is generated with weighted A* using shortest path heuristic to the goal, where the heuristic weight is $\epsilon = 100$. Note how the NPC initially approaches the bear hallway (because it is the shortest distance path) but then must skirt around through the wolves

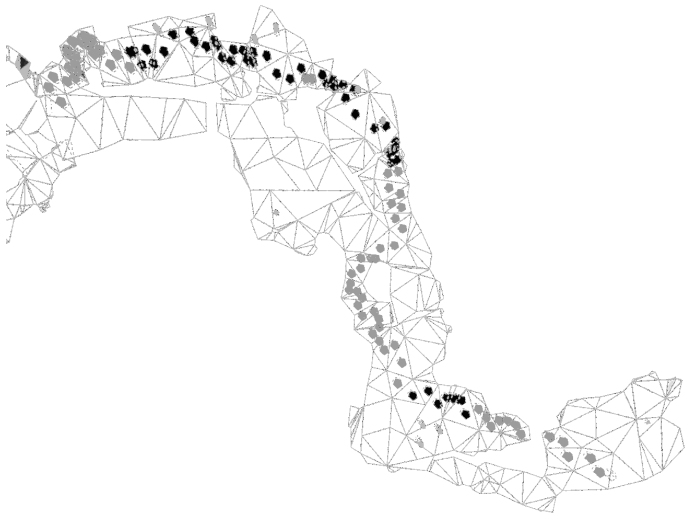


FIGURE 4.5: A plan produced by simple A* with shortest path heuristic when no player model is used. The black points indicate the player is sneaking. Note how the NPC must detour around the area with the bear enemy because it cannot defeat the bear by itself.

because the NPC cannot defeat the bear alone. The path length (cost) of this plan is 133.2 seconds.

Compare this to figure 4.6, which is a plan generated using the same search algorithm but this time factoring in the player model. Note how the plan can actually progress through the bear, because the player model predicts that the player will aid in fighting the bear when the NPC approaches. The path length of this plan is only 24.2 seconds.

In table 4.1, we report metrics such as the number of states expanded, planning time, and path cost between these two planning instances. Note that the path length of the cooperative plan shows a significant reduction in path cost as expected.

Something else worth noting though is that the planning time increased. This was because the search space became harder due to the addition of the player model. In this particular planning query, the increase in search space size did not pose a problem since there were no extreme local minima. However, that is not necessarily true of all queries, as we will see in the next section.

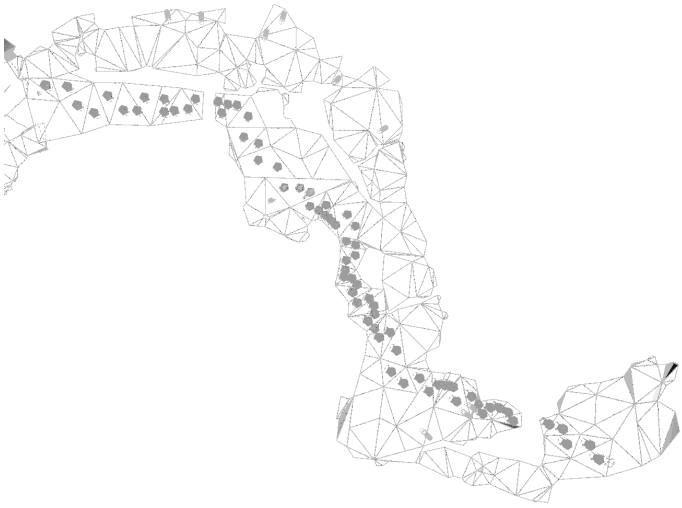


FIGURE 4.6: A plan produced by A* with shortest path heuristic when a player model has been provided by the classifier. Now that the planner can forward simulate the player, the planner can find a path through the Bear enemy, as the combined abilities of the player and NPC can defeat the Bear.

	States Expanded	Planning Time	Path Cost
No Player Model	487	0.135 sec	133.2 sec
With Player Model	1797	0.497 sec	24.3 sec

TABLE 4.1: NPC planning results for when Player follows COMBAT playstyle

FIGURE 4.7: NPC Plan with SNEAK player, using weighted A* with shortest path heuristic and $\epsilon = 100$. The darker dots indicate the player is sneaking.

4.5.2 Speeding Up Search with Supporting NPC Demonstration

In this section we report the planning results for when the player is following the SNEAK playstyle. In this case, we controlled the human-player to sneak pack the initial set of wolves. After sneaking past them, we sent the planning query to the NPC. Our Player Behavior Classifier recognized this as the SNEAK playstyle and sent the according model.

We expect in this case that the NPC plan should resemble the plan in figure 4.5. The NPC should give up on the bear path and take the super wolf path. This is indeed the case as shown in figure 4.7. However, using weighted A* with shortest path heuristic, the planning time was over 5 minutes long (see table 4.2) due to the player state increasing the dimensionality of the search space. The local minima at the bear is exasperated as a result.

Our solution to this is to provide a supporting NPC demonstration to the planner so that it can be used as a T-Graph heuristic [6]. This supporting demonstration can be associated with a particular playstyle class in the Player Behavior Classifier. When a planning query is sent to the NPC planner, the Player Behavior Classifier not only sends a player model, but also an NPC demonstration with it. For the SNEAK playstyle,

Algorithm (and Heuristic)	States Expanded	Planning Time	Path Cost
Weighted A* Shortest Path	898,008	306.003 sec	140.1 sec
Weighted A* T-Graph ($\epsilon = 10$, $\epsilon^T = 10$)	578,781	185.249 sec	137.1 sec
Multi-Heuristic A* T-Graph ($\epsilon = 10$, $\epsilon^T = 10$)	345,208	117.892 sec	197.3 sec
Weighted A* T-Graph ($\epsilon = 20$, $\epsilon^T = 5$)	17,261	4.985 sec	137.1 sec
Multi-Heuristic A* T-Graph ($\epsilon = 20$, $\epsilon^T = 5$)	9,356	2.047 sec	166.7 sec

TABLE 4.2: NPC planning results for different search algorithms when player is following the SNEAK playstyle

we decided the most appropriate supporting NPC demonstration would actually the SNEAK demonstration itself. To be precise, the demonstration used is the one shown in figure 4.4. The NPC should imitate the player in sneaking past the super wolves.

For the T-Graph heuristic, we tried various combination of T-Graph inflation factors ϵ^T and overall inflation factors ϵ which had a combined product $\epsilon\epsilon^T = 100$. We report results in particular for $\epsilon^T = 10$ and $\epsilon = 10$ and also $\epsilon^T = 5$ and $\epsilon = 20$. Figure 4.8 shows one of resulting plans when using T-Graphs. It resembles the plan without T-Graphs, but it differs in that the T-Graph does not approach the left hallway. The T-Graph Heuristic dramatically reduced the planning time by up to 98% depending on the epsilon parameters (see table 4.2).

We also tried using Multi-Heuristic A* [1] instead of just weighted A* to see if we could further reduce planning times when using a player model. We used shorted path distance as our anchor heuristic, the T-Graph as one of the inadmissible heuristics, and an extra inadmissible heuristic which favored states where the NPC was in sneak mode. This extra heuristic was computed by taking the shortest path distance and adding a constant penalty if the NPC was not sneaking. We used a inadmissible heuristic inflation factor of $w_1 = 100$, which was approximately equivalent to how our weighted A* searches were using inflation factors of 100. We also used an anchor heuristic inflation factor of $w_2 = 100$, meaning the anchor search was rarely expanded. (See [1] for how

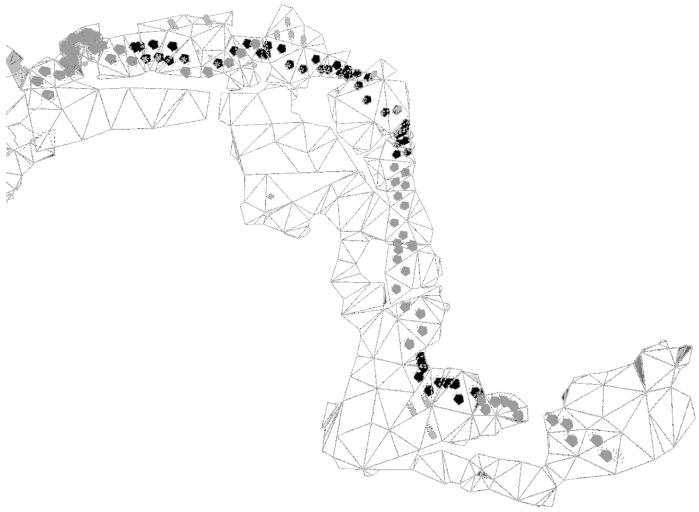


FIGURE 4.8: NPC Plan with SNEAK player, using weighted A* with T-Graph heuristic and $\epsilon^T = 10\ \epsilon = 10$. The darker dots indicate the player is sneaking.

details on inflation factors are used.) For the T-Graph heuristic, we again also tried different T-Graph inflation factors so that $\epsilon\epsilon^T = w_1$. The inflation factors used are the same as our weighted A* search with T-Graph, so that the results are comparable.

An example of the plan produced by Multi-Heuristic A* is shown in figure 4.9. Again it is comparable to the weighted A* searches, except there are more spots where the NPC sneaks due to the extra heuristic. The complete performance metrics for our Multi-Heuristic A* searches are summarized in table 4.2. In both cases, Multi-Heuristic A* outperformed weighted A* with T-Graphs in planning time at the expense of a less optimal path.

4.6 Summary

In this chapter, we showed how the Player Behavior Classifier algorithm outlined in chapter 2 could be applied in the context of cooperative planning for NPCs in *Skyrim*. We described how the classifier fits within a *Skyrim* NPC planing framework: it provided the NPC planner a player model and a supporting NPC demonstration for the current predicted playstyle. We presented results which illustrated how the planner

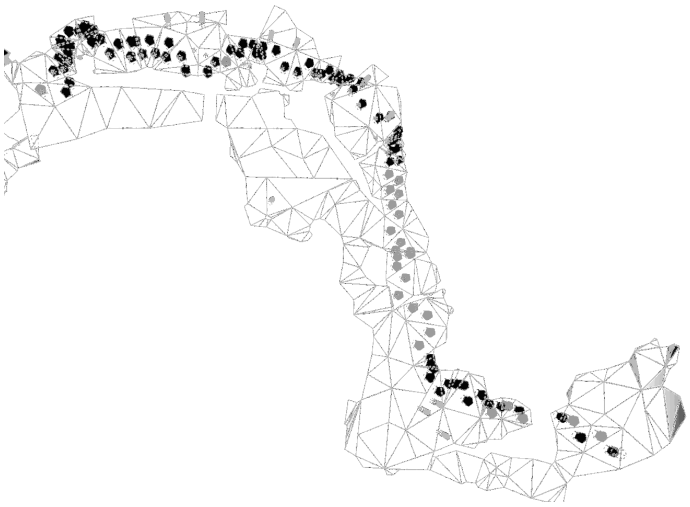


FIGURE 4.9: NPC Plan with SNEAK player, using Multi-Heuristic A* with inflation factors $w_1 = 100$ and $w_2 = 100$. T-Graph heuristic used $\epsilon^T = 10$ and $\epsilon = 10$ so that $\epsilon\epsilon^T = w_1$. The darker dots indicate the player is sneaking.

could find cooperative solutions with the addition of the player model. We further presented how Multi-Heuristic A* and T-Graph heuristic could help speed up the search, since adding a player to the search state made the search more difficult.

Chapter 5

Related Works

In this chapter, we discuss how others have approached the problem of learning human behaviors in the context cooperative AI agents.

5.1 Plan Recognition

The problem of identifying the intention of a human is often referred to as "Plan Recognition" in the literature. The main problem setup is that an AI agent is presented with observations of the actions of another agent (usually human) and must hypothesize the intended goal and plan.

Kautz et. al. demonstrated how to express particular plan recognition problem with first-order logic [2, 16]. Their technique involves representing the problem domain as an action hierarchy, showing how abstract tasks are specialized to specific tasks and how large actions are broken down into smaller steps. This is formalized as an action taxonomy, a set of inference axioms, which allow an initial set of observations to be reduced to the task (plan) they are contributing to. The formalization depends heavily on a closed-world assumption: all actions and high level plans are well defined, and every action must be a part of one of these plans. The rigidity of this technique makes it difficult to apply to the context of human playstyles in video games, as our plans are not necessarily well defined and we have no guarantees that all player actions are purposeful. Furthermore, this technique does not allow us to address our first problem:

we do not know the playstyles to begin with, so we cannot define an action taxonomy with a closed-world assumption.

Another approach to plan recognition is through the use of functional grammars. Geib et. al. presented such a grammar to build an AI agent that would help with setting a table [11, 9, 10]. Their technique defined an action lexicon which is a set of substitution rules that associate each observable action with its possible goals and the preconditions and postconditions actions. By performing all possible parses of an observed set of actions, they could compute a probability distribution of hypothesized plans. Included in this computation were the set of subgoals that still needed to be performed in order to realize the hypothesized plan. This allowed their agent to aid humans by working on subgoals that would realize the overarching goal. Much like the first-order logic variant of plan recognition, an action lexicon requires that plans and their subgoals be defined up front. This is too strict of a requirement, as our goal is avoid having to define playstyles and instead learn from demonstrations. In addition, the benefit of identifying unsatisfied subgoals is limited, as the subgoals presented in the paper are independent pieces of work that only require one worker. The cooperation here is that a human can work in parallel with an AI agent on separate tasks. However, in cooperative games, we generally need the ally NPC to work together with the human-player on a single task.

Intille and Bobick presented a different approach using Bayesian Belief Networks to recognize multiperson actions [15]. In their paper, they attempted to classify videos of different plays in American Football. They used a pipeline of feeding visual belief networks into temporal analysis networks into a final play recognition network. Overall, the algorithm attempts to identify individual player actions and the temporal relationships among them to determine the likelihoods of the multiperson plan is being executed. This technique however requires detailed descriptions of the temporal relationships in a high level plays, which are even less readily available for video game playstyles.

5.2 Learning Approaches

In the pervasive computing domiain, Gu et. al. presented a solution to recognizing human activities through the use of a concept called Emerging Patterns [12, 18]. An Emerging Pattern is set of observations that serves as a good discriminators between different classes of data. By using these discriminators, they were able to score new instances of observations on what household activity they represented. This approach requires that a dataset of observations for known activities are given in advance. Furthermore, the observations must be labeled, so that Emerging Patterns can be mined by comparing the observations in one activity class to another. In otherwords, this solution is a supervised learning algorithm. In our domain, we are not necessarily given labels for the playstyles. We hope to discover playstyles through demonstrations, and thus we need an unsupervised learning method.

Chapter 6

Conclusion

In many modern video games, NPCs are still unable to work with human players on cooperative tasks in a sophisticated matter. The key missing component in NPC planers is that they do not have a way to recognize player intentions and are unable to predict player actions during planning. Specifically, they are missing an accurate player model that can predict trajectories.

To address this, we introduced the Player Behavior Classifier, which can learn playstyles from demonstrations via unsupervised learning. We explained how playstyles are classes of activity sequences which have similar patterns of events. We showed how we can learn these classes via an existing activity clustering technique, which is implemented by finding dominant sets in a weighted similarity graph [13, 21]. We built on top of existing work by proposing new modifications to the similarity metric, a Bayesian Belief Network formulation that can account for uncertainty during online classification, and a new concept called discounted histograms which prevents old events from polluting activity histograms. We presented our process of tuning various hyper-parameters, and showed how our modifications to the similarity metric and other choices created the best clusterings for demonstration activities from Simple *Skyrim*. We presented results which showed how effective this clustering algorithm was and how well the system performed in online classification.

We then motivated the use of the Player Behavior Classifier in the context of cooperative planning for NPCs and described how we integrated the classifier with a *Skyrim* NPC planning framework [5]. We demonstrated that an NPC planner could now find

new cooperative solutions, thanks to our provided player model. To balance the increased search difficulty due to the addition of the player model, we applied search techniques like T-Graphs [6] and Multi-Heuristic A* [1] and showed that these were effective in reducing planning times with the player model.

We will close by describing some avenues of future work. Our work here has formalized the problem of learning playstyles as an unsupervised learning problem. We presented activity histograms and discounted histograms as way to extract a feature vector from an event sequence. We also provided a similarity metric as a measure between two vectors, which can easily be turned into a distance or loss function. By formalizing these concepts, we foresee that others could potentially apply other clustering algorithms, such as k-means [19] or message passing techniques [8], and evaluate their effectiveness in learning playstyles.

We also noted that the are likely more sophisticated ways of creating a concrete model out of the demonstrations in a playstyle class, such as using an n-gram predictor and local event planner. We hope to see how more sophisticated models perform, especially when the game environment is significantly different from the demonstrations.

Bibliography

[1] Sandip Aine et al. "Multi-Heuristic A*". In: *International Journal of Robotics Research (IJRR)* (2015).

[2] James Allen et al. "A Formal Theory of Plan Recognition and its Implementation". In: *Reasoning about plans*. Morgan Kaufmann, 1991. Chap. 2, pp. 69–126.

[3] Donald J Berndt and James Clifford. "Using dynamic time warping to find patterns in time series." In: *KDD workshop*. Vol. 10. 16. Seattle, WA. 1994, pp. 359–370.

[4] E.W. Dijkstra. "A note on two problems in connexion with graphs". English. In: *Numerische Mathematik* 1.1 (1959), pp. 269–271. ISSN: 0029-599X. DOI: 10.1007/BF01386390. URL: http://dx.doi.org/10.1007/BF01386390.

[5] John Drake. "Planning For Non-Player Characters by Learning From Demonstration". PhD thesis. 2018.

[6] John Drake, Alla Safonova, and Maxim Likhachev. "Demonstration-Based Training of Non-Player Character Tactical Behaviors". In: *Proceedings of the Twelfth Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*. 2016.

[7] John Drake, Alla Safonova, and Maxim Likhachev. *Towards Adaptability of Demonstration-Based Training of NPC Behavior*. 2017. URL: https://aaai.org/ocs/index.php/AIIDE/AIIDE17/paper/view/15858.

[8] Brendan J Frey and Delbert Dueck. "Clustering by passing messages between data points". In: *science* 315.5814 (2007), pp. 972–976.

[9] Christopher Geib et al. “Building Helpful Virtual Agents Using Plan Recognition and Planning”. In: *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2016.

[10] Christopher W Geib. “Delaying Commitment in Plan Recognition Using Combinatory Categorial Grammars.” In: *IJCAI*. 2009, pp. 1702–1707.

[11] Christopher W Geib and Robert P Goldman. “Recognizing Plans with Loops Represented in a Lexicalized Grammar.” In: *AAAI*. 2011.

[12] Tao Gu et al. “epsicar: An emerging patterns based approach to sequential, interleaved and concurrent activity recognition”. In: *Pervasive Computing and Communications, 2009. PerCom 2009. IEEE International Conference on*. IEEE. 2009, pp. 1–9.

[13] Raffay Hamid et al. “A novel sequence representation for unsupervised analysis of human activities”. In: *Artificial Intelligence* 173.14 (2009), pp. 1221–1244.

[14] P.E. Hart, N.J. Nilsson, and B. Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *Systems Science and Cybernetics, IEEE Transactions on* 4.2 (1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[15] Stephen S Intille and Aaron F Bobick. “Recognizing planned, multiperson action”. In: *Computer Vision and Image Understanding* 81.3 (2001), pp. 414–445.

[16] Henry A Kautz and James F Allen. “Generalized Plan Recognition.” In: *AAAI*. Vol. 86. 3237. 1986, p. 5.

[17] V. I. Levenshtein. “Binary Codes Capable of Correcting Deletions, Insertions and Reversals”. In: *Soviet Physics Doklady* 10 (Feb. 1966), p. 707.

[18] Jinyan Li, Guimei Liu, and Limsoon Wong. “Mining statistically important equivalence classes and delta-discriminative emerging patterns”. In: *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2007, pp. 430–439.

[19] Stuart Lloyd. “Least squares quantization in PCM”. In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[20] Ian Millington and John Funge. *Artificial Intelligence for Games, Second Edition*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN: 0123747317, 9780123747310.

[21] Massimiliano Pavan and Marcello Pelillo. "A new graph-theoretic approach to clustering and segmentation". In: *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2003, pp. I–I.

[22] Judea Pearl. "Bayesian networks: A model of self-activated memory for evidential reasoning". In: *Proceedings of the 7th Conference of the Cognitive Science Society, 1985*. 1985, pp. 329–334.

[23] Mike Phillips et al. "E-Graphs: Bootstrapping Planning with Experience Graphs". In: *Proceedings of Robotics: Science and Systems*. Sydney, Australia, 2012.

[24] Mike Phillips et al. "Learning to Plan for Constrained Manipulation from Demonstrations". In: *Proceedings of Robotics: Science and Systems*. Berlin, Germany, 2013.