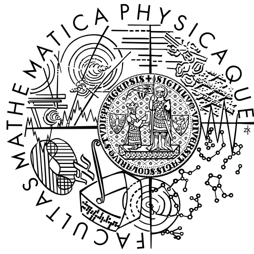

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jakub Arnold

HexMage - Encounter Balancing in Hex Arena

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Mgr. Jakub Gemrot
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2017

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In .... date .... signature of the author

Title: HexMage - Encounter Balancing in Hex Arena

Author: Jakub Arnold

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Department of Software and Computer Science Education

Abstract: Procedural content generation (PCG) is mostly examined in the context of map/environment creation, rather than generating the actual game characters. The goal of this thesis is to design a turn-based RPG-like game with perfect information for which we can generate balanced encounters. The game consists of a hex-based arena in which two teams fight. Each team consists of a few player controller characters with unique abilities. We generate the attributes of these abilities in order to make the encounter balanced. We will also build an AI that can be used to automatically play-test the PCG algorithm. The goal is to generate an equally strong, but different opponent.

Keywords: video games encounter balancing hex arena rpg elements

I would like to thank my supervisor Mgr. Jakub Gemrot for his valuable advice and suggestions. I would also like to thank my girlfriend Zuzana for her endless patience and support.

Contents

Introduction **3**

1 Problem Definition **4**
1.1 Types of PCG currently being used 4
1.2 Thesis Goals 4
1.3 Scope 5
1.4 Used technology 5

2 Game Rules and Mechanics **6**
2.1 Map 7
2.2 Simulator 8
2.2.1 Replay recording 9

3 Game AI **10**
3.1 Monte-Carlo Tree Search 10
3.1.1 Implementation of MCTS and high level actions 11
3.1.2 Playout 12
3.2 Rule based AI 13
3.3 Random AI 13

4 Generating Encounters **14**
4.1 Reducing the Scope of the Problem 14
4.2 Approach 14
4.3 Simulated Annealing 14
4.4 Evolution Strategies 15
4.5 Choice of the Fitness Function 15
4.6 Remarks 16

5 Experiments **19**
5.1 Comparing AIs against each other 19
5.2 Survey 19
5.2.1 Participants 20
5.2.2 Experiment Design 20
5.2.3 Questions 20
5.2.4 Discussion of the Results 21

6 Conclusion **23**
6.1 Future work 23

Bibliography **24**

List of Figures **26**

List of Tables **27**

Abbreviations and Terminology **28**

A Implementation Analysis **29**
A.1 HexMage.GUI . . . 30
A.1.1 Threading and TPL . . . 31
A.2 HexMage.Simulator . . . 31
A.3 Performance . . . 31
A.3.1 Map implementation . . . 32
A.3.2 Game Events and AI Controllers . . . 32

B File formats **33**
B.1 DNA file format . . . 33
B.2 Map File Format . . . 33

C User Documentation **34**
C.1 System Requirements . . . 34
C.2 Generating Experiment Data . . . 34
C.3 Running the Experiment . . . 34
C.4 Playing Custom Games . . . 35

D Programmer Documentation **38**
D.1 Used Libraries . . . 38
D.2 Compilation Instructions . . . 39
D.3 Constants and Command Line Arguments . . . 39
D.4 Running the Experiments . . . 39
D.5 The Game Loop and GameEventHub . . . 40
D.6 Writing Custom AI . . . 40
D.7 Creating Encounters with GameInstance . . . 41
D.7.1 Encounter Setup . . . 42
D.7.2 Invariants and Action Validation . . . 43
D.8 Extending the Game UI . . . 43
D.8.1 Entities and Components . . . 43
D.8.2 Scenes . . . 44

Introduction

An increasing number of computer games is using procedural content generation (PCG) as one of their core mechanics. This is in different contexts, most commonly for generating new levels (e.g. Diablo [8]). Occasionally games even generate player collectible items (e.g. Borderlands [10]). However, there has not been much research on the use of procedural generation for balancing encounters in RPG games. By this we mean procedurally generating enemies that can be defeated by the player, but pose a challenge. A crucial criteria here is that the balance is not simply achieved by creating the enemy as an exact clone of the player, but rather explore the search-space to find an enemy that is not only balanced, but also different from the player.

One particular application for this kind of PCG is automatic difficulty adjustments based on the player's skill. Another possible use could be automatic generation of new and unique enemies based on given constraints, which is the approach we chose in this thesis.

We have created a custom game with mechanics that are simple enough to simulate quickly, yet flexible enough to represent a large search space. There are two teams that fight in a hexagonal arena, each consists of a small number of player controlled characters (mages), and each mage has a small number of abilities. In each turn the player has control over one of his characters, and both move around the map and cast spells, in any order he wishes. The only limit is the number of action points the character has available, which are consumed both by movement and ability usage. The side that first eliminates the opposing team wins.

All information is visible to all players, and all actions are completely deterministic. There is no time limit for the player action, which means the player could theoretically calculate a perfect move given enough time.

The goal of this thesis is to explore PCG options for balancing encounters in turn-based RPG-like games. We design a simple game with flexible mechanics, and build an AI that can be used to approximate the player. We then use evolution strategies to generate opponents of just the right difficulty for the given player team, using AI vs AI combat as a fitness function.

Organization

In Chapter 1 we begin by defining the scope of the work and our general approach. Chapter 2 follows by exploring our game mechanics in detail and explaining the choices behind them. Next in Chapter 3 we will go over our different choices for implementing the AI.

Chapter 4 describes our approach to generating the encounters. The experiments are described in Chapter 5 with a conclusion in Chapter 6. Lastly, the Appendix contains user and programmer documentation, as well as description of the used file formats and an analysis of the implementation.

1. Problem Definition

Tactical turn-based games like Duelyst [5] and Faeria [1] have become very popular in the recent years. It is easy to create more content for such games (new spells, monsters, etc.) by simply changing the attributes of existing content. However, it is difficult to make sure the game stays balanced as the new content is added over time. This is generally done by human play-testing in-house before the game updates are released, and takes large amounts of time.

This thesis aims to solve the problem with *procedural content generation* (PCG). We define PCG as 'the algorithmic creation of game content with little or no user input' [28] We introduce a custom game similar to Duelyst, and show that search based PCG can create new content that is balanced.

1.1 Types of PCG currently being used

Many games utilize different kinds of PCG these days. Large commercial games such as Diablo 2 [8] use PCG to generate unique map layout of certain areas. Some games like the popular Minecraft [20] go to an extreme and procedurally generate the whole world. But PCG is not only limited to large games, even small indie games like Spelunky [22] or Terraria [25] utilize PCG for generating playable levels.

Games also use PCG for item creation, where the specific attributes of an item are generated online (e.g. Borderlands [10]).

1.2 Thesis Goals

Our main goal for this thesis is to generate balanced encounters. For this, we decided to implement a custom game with flexible game mechanics so that there are many different ways to create a balanced encounter. The game is turn-based, zero-sum, with perfect information (see chapter 2).

We implement the game both in the form of a simulator that can be used as a library, and a GUI that a human player can use to play the game and test it. We also implement an AI for the game so that we can automatically evaluate and test games in our PCG algorithm.

In summary, the goals are:

- Design and implement a custom game, both GUI and a simulator of the game mechanics.
- Build an AI that can be used to represent the player in terms of skill.
- Create an algorithm for generating balanced encounters for our game.
- Verify that the AI and encounter balancing works by conducting an experiment.

1.3 Scope

We narrow our scope to turn-based games, since simultaneous move games provide an additional challenge when building an AI, which for us is just a means to help with PCG. Our game zero-sum for two players. We also chose to make the game with perfect information to simplify the creation of AIs.

1.4 Used technology

We chose C# and the .NET platform since it provides a good balance of programmer productivity and performance. C#, while being a high level language, still has the ability to control memory layout of objects to some extent. Most importantly, it has the `struct` keyword which allows for objects which are allocated in-place and have copy semantics, contrast to Java, where all objects are dynamically allocated on the heap. This feature allows for compact representation of data and thus improves locality.

On the other hand, C# also provides high level primitives in the form of LINQ and the Task Parallel Library [19] (see subsection A.1.1). An example of where this is used is projectiles, where one can create a projectile and `await` the task that represents the projectile hitting the target, and chain an animation afterwards. While developing, we also made heavy use of the advanced profiling tools provided for the .NET ecosystem, most notably dotTrace [12].

2. Game Rules and Mechanics

Two teams fight on a hexagonal map (*arena*) of small size (radius of 5–10 hexes, see Figure 2.1). The map contains empty hexes and walls. Each team consists of a small number of player controller characters (*mages* for short), and each mage has a small number of *abilities*, *health*, and *action points*. Players take turns, during which each player has control over one of his mages. The player can issue commands to move around the arena and use abilities. Mages can only walk on empty hexes and cannot cast through walls.

There is also an important distinction between a *turn* and a *round*. A turn means playing all the actions a single mage can do with his action points, and ends when the player decides he is finished playing with that one single character. A round ends when all of the characters have played their turn, and it is at that point when debuffs, AOEs, cooldowns and action points are re-calculated (see below).

Both movement and ability usage costs action points, which are restored at the end of the round. Moving one hex costs one action point. The cost of using an ability varies, and is one of the parameters that we optimize for when looking for a balanaced game. An ability can also have a *cooldown*, which prevents repeated usage for a given number of rounds. Note that the cooldown can be zero, which means the ability can be used multiple times per turn. Abilities also have limited range, which supports positional gameplay.

Abilities can do direct damage to an enemy mage, apply a *debuff* (causing damage and decreasing action points over time), and create an area of effect (*AOE*) debuff that spans multiple hexes in the arena. Both the debuffs and AOEs are applied at the end of each round. Both debuffs and AOEs also have a lifetime, which specifies how many rounds the effect lasts.

The motivation for having cooldowns is that it allows us to generate powerful abilities that don't necessarily take up the whole turn of the player by costing a lot of action points. If the ability is cheap, but has a large cooldown, it can serve a strategic purpose, as the player might want to prepare his position in order to use the ability when the cooldown wears off. AOE abilities also improve positional gameplay, as they can be used to force the enemy player to move out of position.

In summary, the game mechanics are:

- Two players fight in a hexagonal arena.
- Each player has a group of mages.
- Each mage has health, action points, and a number of abilities he can use.
- Movement and ability use costs action points.
- Abilities can cause direct damage, apply a damage over time (debuff), or cause a similar effect in a portion of the arena (AOE).
- When a player finishes playing his mage, he ends his *turn*.

- When all mages play their turn, the *round* ends and debuffs and AOEs are calculated.
- When the mages of one player are eliminated, that player loses. If all of the mages are eliminated in a single turn, the game ends in a draw.
- Both players see everything (perfect information).

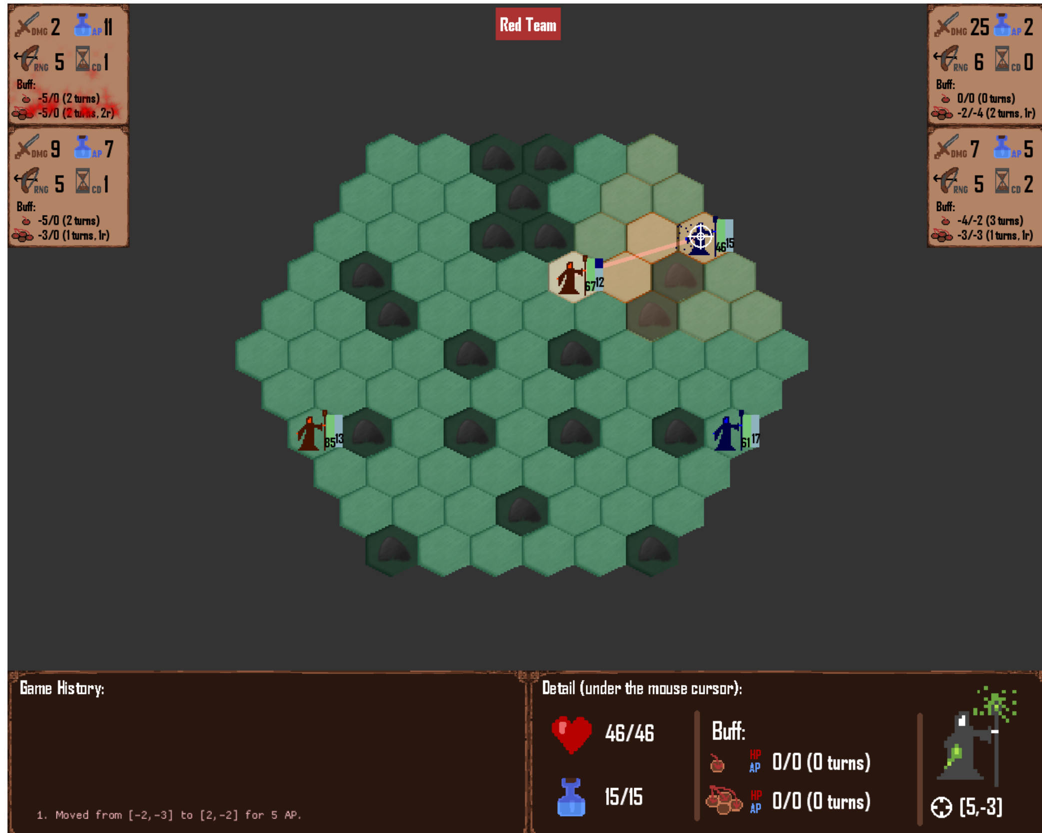

Figure 2.1: User interface of the HexMage game, featuring a 2v2 game. The red player currently has a spell selected and is targeting one of the blue player's mages.

2.1 Map

The map consists of two types of hexes. *Empty* hexes which can be walked on, and *walls*, which can not be stepped and obstruct visibility. The map also contains *starting points* for all of the mages. These are part of the map design and must be created before the encounter generation can start. While a map can have an arbitrary amount of starting points, the teams' sizes must be at most the same as the number of starting point per team.

All maps have a hexagonal shape and are defined by their radius. Figure 2.1 shows an example of a map with a radius of 5. In order to make testing easy, the game also includes a map editor (see Editor user documentation), as shown in Figure 2.2.

Maps can be saved to a file and loaded back again at any time (see the Map File Format section for more details).

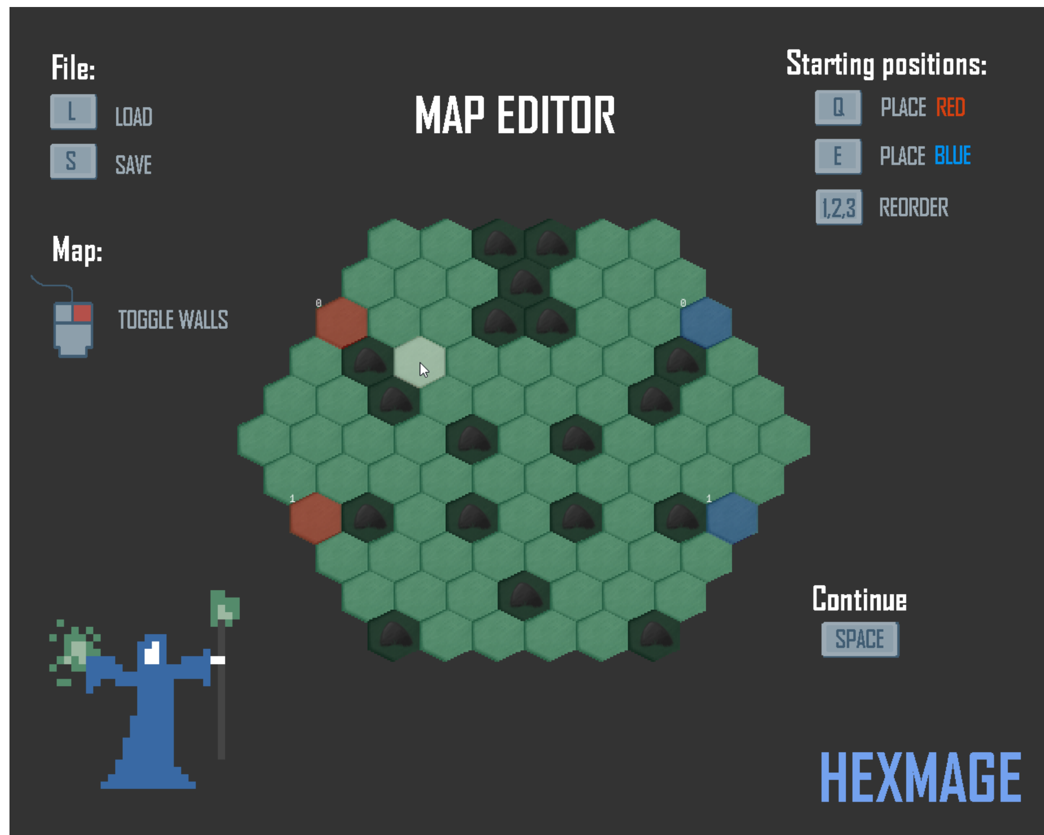

Figure 2.2: User interface of the map editor, showing a map of a radius of 5 hexes. Light green are hexes represent walkable surface, dark hexes represent walls. Red and blue hexes represent starting points.

2.2 Simulator

Part of our game is a simulator that can be used as a library and encapsulates all of the game rules and mechanics. This is then used by both the AI and the PCG algorithm and can thus be run separately from the main game. The game is internally represented by a *state* object, and all of the possible player actions are encapsulated by an *action* object. Playing the game through the library then simply becomes a matter of applying a state transition function $f : (\text{state}, \text{action}) \to \text{state}$.

Here's a list of all possible actions at an arbitrary state:

AbilityUse Use an ability targeting an enemy that is already in range.

Move Move the current mage to a different hex on the map.

EndTurn Finish the current turn.

DefensiveMove Serves the same purpose as *Move*, but carries an additional information in the sense that *DefensiveMove* can only be the last action of the turn.

AttackMove Combines the *Move* and *AbilityUse* actions into one.

NullAction Doesn't do anything and is mostly used as a placeholder in cases no action is possible.

The simulator also verifies that no invalid actions are applied through a thorough list of invariant checks. These are automatically turned off in a release build to make the simulator run as fast as possible.

The simulator is also built to be high performance and can easily run hundreds of thousands to millions of actions per second on a consumer-grade PC. The state object is split into two parts, one that handles the general immutable information that doesn't change as the game progresses (i.e. max hitpoints, ability definitions, etc.), and one that handles all of the mutable data, such as current hitpoints, current positions on the map, etc. This allowed us to make state copies very fast as well, running only at a few microseconds per copy.

2.2.1 Replay recording

The simulator also has a builtin capability for recording replays from a given game. The replay is a recording of all of the actions that occured in a given game, plus a starting state. When replay recording is turned on (see the Command Line Arguments section), the simulator records all actions as they are applied to the game state in a list, and when the game finishes it stores the list into a file along with the initial game state.

3. Game AI

In order to test our encounter balancing approach, we needed to develop an AI that can be used to evaluate match setups. Since our game is very positional and with complicated effects that can span multiple turns, we conclude that manual evaluation of the game state would be difficult. There is also a large amount of actions possible at each turn, and a game is expected to last anywhere from 2 to 5 rounds. Given a 2v2 setup, this would yield $50^{4*5*4} \approx 10^{135}$ possible states (50 actions per turn, 4 turns per round, 5 rounds, 4 mages).

A naive approach using minimax [26] would not be possible as it could not search the whole game tree. Other approaches such as Alpha-Beta pruning [26] are also not possible because of the difficulty of evaluating the current game state. Debuffs, AOEs and cooldowns are difficult to evaluate just by looking at the game state. For example, standing inside of an AOE might cause the mage to lose enough action point that he's not able to use his ability, effectively losing his turn. However, this will only happen in some cases when the AOE has enough of an effect and the mage stands far enough that he has to move and use an ability. Taking visibility into account also increases complexity, as there can be lots of obstacles on the map.

As a result, we chose to use *Monte-Carlo Tree Search* (MCTS), which has both the benefit of not needing a state evaluation function, and searching the tree asymetrically with focus only on the interesting parts.

We present three different approaches and compare them in their strength, specifically a MCTS based AI, a *Rule based AI*, and a *Random AI*. The Random AI will serve as a baseline for benchmarking the other AI implementations. The Rule based AI serves to show that the game can not be simply won by playing greedily, but rather requires positional gameplay. Our goal is to show that MCTS can beat both the Rule based AI and the Random AI.

3.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search [2] is a heuristic search algorithm that focuses its search on the most promising nodes in the game tree. States are evaluated based on the result of a *playout*, which means picking actions based on a policy until a terminal state is reached, at which point it can be easily evaluated. The policy can be for example uniform sampling of the possible actions. The result of the playout is then used to re-calculate rewards for all the parent nodes up the game tree.

The MCTS tree consists of nodes which represent game states. Each node keeps two values: the number N of times it has been visited by MCTS, and the total cumulated reward Q based on simulations. Overall, the algorithm consists of four steps:

Selection Start from the root R and chose the most promising child node until a leaf node N is reached.

Expansion If N is not a terminal node, add a new child state to N.

Simulation/Playout Play the game to finish according to a default policy.

Backpropagation Calculate the reward terminal state of the default policy, then update reward/visited counts on the path from N to R.

There are many variants of MCTS, and the one we chose is UCT (*Upper Confidence Bound for Trees*). It uses the Upper-Confidence Bound (UCB) formula to balance *exploration* and *exploitation* of the game tree. Exploration is the tendency to choose nodes with the highest reward, and explitation is choosing nodes that have been visited in only a few simulations.

$$\text{UCB} = \frac{Q}{N} + C \cdot \sqrt{\frac{\ln ParentN}{N}} \tag{3.1}$$

C is a constant that can be adjusted based on the game mechanics and controls the amount of exploration. The theoretical optimum is $\sqrt{2}$ [2], which is what we picked for HexMage.

One of the advantages of MCTS is that unlike minimax, it grows the search tree asymmetrically towards the most promising actions.

3.1.1 Implementation of MCTS and high level actions

After some experimentation, we've settled down for three high level actions that represent most of what a player might want to do.

AbilityUse Use an ability targeting an enemy that is already in range.

AttackMove Move into the range of an enemy and use an ability.

DefensiveMove Look for a place on the map that is not visible to the enemy and move there (to avoid damage).

Combined actions help significantly reduce the depth of the game tree (see Figure 3.1 for an example of a game tree). Most prominent is the fact that we don't allow arbitrary *Move* actions, but only *DefensiveMove*.

In an average setup we can expect each mage to have tens of possible *Move* actions. Specifically, given our 2v2 setup on a 10-hex map there could up nearly up to a 100 move actions possible. Given the game mechanics, it is also completely valid to move from A to B and then from B to C, instead of moving directly from A to C. While this is something the player might want to utilize, it makes no sense to separate the *Move* actions from the point of the AI.

It also doesn't make any sense to execute a *DefensiveMove* actions (with the goal to hide), and then try to do anything else. If we wanted to attack after moving, the action would be *AttackMove*, and if we just wanted to move to a different position, we could always move there directly.

The reason we chose to split *AbilityUse* and *AttackMove* is because one mage can use an ability multiple times per his turn. This creates a need for a raw *AbilityUse* action. The *AttackMove* action is also necessary since the enemy might be out of range.

One thing we explicitly do not consider is moving towards an enemy without the intent of using an ability or hiding. A player could theoretically move into

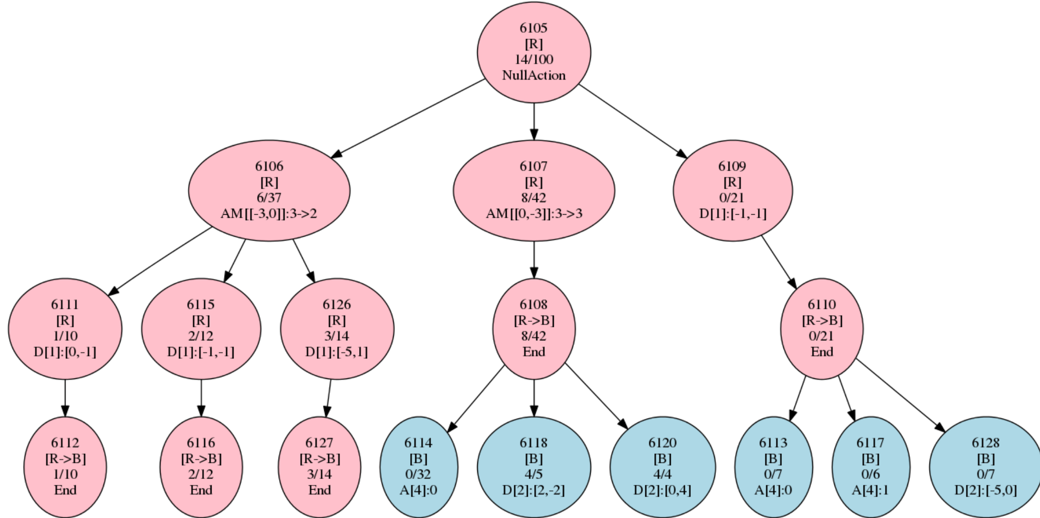

Figure 3.1: An exmple of a MCTS game tree after 100 iterations. Each node contains (from top to bottom) a node identifier, currently active team, cumulated reward / total number of visits, and the action type. The nodes are also colored based on the team color. The actions are shown in shortcuts to reduce tree width (AM - AttackMove, D - DefensiveMove, End - End Turn, NullAction - empty action representing the root node).

a position that is vulnerable, but at the same time not allowing him to use an ability afterwards. We made this decision based on the layout of our map which should almost always allow the player to move forward while also hiding behind cover.

3.1.2 Playout

Given the number of possible actions at each turn, and the number of turns needed to finish the game, we decided that a random playout as is generally suggested with MCTS is not feasible [2]. The games are usually short (between 2–5 rounds), which means a single bad action can decide the whole game. A uniform action selection would not yield a representative playout.

We thus chose to make our MCTS playout deterministic. The strategy is similar to that of our *Rule based AI*, but is made simpler in order to avoid generating a heatmap on each turn. The strategy is as follows:

- If there is an enemy in sight, use the best possible ability against him.
- If there is no enemy in sight, pick an enemy and move towards him.

Having an aggressive set of rules both makes the games shorter, and allows for faster generaiton of actions than that of our *Rule based AI*, since we don't have to account for positional information.

The main benefit of the playout strategy is measuring the effect of debuffs, AOEs and cooldowns, which unlike damage/health would be difficult to calculate in closed form given a state of the game.

Our hypothesis is that MCTS can play reasonably well even against a human player, which is later tested in an experiment (see Chapter 5). Next we built a

Rule based AI which picks its actions based on a fixed set of predefined rules. These are mostly designed for an aggressive-style gameplay in order to make simulated games shorter. Lastly, we present a *Random AI* which makes uniform random choice of actions from a pool of generated actions.

3.2 Rule based AI

To get a reasonable comparison between both MCTS and the *Random AI* we built an aggresive *Rule based AI*, which does not simulate the playout in any way, and only does simple analysis of the board state before choosing what to do. The rule order is as follows:

- If there is an enemy in sight, use the best possible ability against him.
- If there is no enemy in sight, but we have an ability that can be used, pick a target towards which we can move and use our ability.
- Otherwise, hide from enemy sight.

The structure of these rules is similar to the actions generated by MCTS, but there is no additional search or game-state evaluation. The rules are simply picked from top to bottom.

We found that even with such a simple set of rules the AI can play reasonably well against a human opponent.

3.3 Random AI

In order to establish a baseline when evaluating our *Rule based AI* and MCTS. The *Random AI* uses the same mechanics as MCTS for generating possible actions, but choses among them randomly. This can be seen as taking a random walk down the MCTS search tree, and as such, the Random AI doesn't play completely random.

The benefit of re-using the MCTS tree is that we get a reasonable benchmark against both the rule-based AI and MCTS. If we generated actions completely at random, the *Random AI* would walk around the map without doing much of anything, as the number of *Move* actions greatly outnumbers the *AbilityUse* actions. To give a rough estimate, in a 2v2 game where each mage has 10 action points, there would be at most 2 *AbilityUse* actions at each point in the game, but up to 100 *Move* actions (considering a small map of radius 5).

There are also certain limitations imposed on the choice of MCTS actions to allow for faster search, from which the *Random AI* can benefit. For example, it is not allowed to use a *Move* action twice in a row, as these could always be collapsed into a single *Move* actions to the final target destination. This limitation alone reduces the search tree greatly.

By making the *Random AI* smarter, we can get a better estimate of just how better both the *Rule based AI*. This should serve as a benchmark against a player who would pick actions mostly at random, while also putting at least some thought into not doing things that are completely pointless, such as moving from A to B and back from B to A in a single turn.

4. Generating Encounters

There are many possible attributes that can be generated. We could change the size of each team, the number of abilities of each mage, starting positions of the map, or even change the positions of the walls.

4.1 Reducing the Scope of the Problem

We chose to reduce the scope by creating a small fixed map on which all encounters will be played and balanced. We also fix the team size of both teams to 2 mages, and each mage to 2 abilities. While this significantly reduces the possibilities to achieve balance, there are still a lot of parameters that can be tweaked, as described in the next section.

We also put lower and upper bounds on most of the numeric values of attributes of mages and their abilities. See attached programmer documentation in the Appendix D for more details about how attributes are represented.

4.2 Approach

We approach generating encounters as a search based problems with two different approaches, Simulated Annealing [26] and Evolution Strategies [27].

To make the search algorithms as general as possible, we serialize our internal game representation into a single vector of normalized floating point values (called DNA, see section B.1). The algorithm then does not need to understand our game mechanics and restrictions and simply treats the DNA as a vector of floating point values.

In the case of our experiments, we chose to stick with 2v2 games on a fixed map, where each mage has only two abilities. This was chosen both with respect to our questionnaire, and running time of the algorithms. Choosing a larger team setup or a bigger map (or many different maps) would be difficult for the participants, and would also take much longer to compute our experimental data.

Taking these restrictions into mind, the DNA would then take up 96 floating point values, specifically:

$$2 \text{ Teams} \times 2 \text{ Mages} \times (HP + AP + 2 \times \text{AbilitySize}) = 96$$

since Ability Size = 11 as we need to serialize damage, cost, range, cooldown, debuff (HP damage, AP damage, lifetime), and AOE (debuff + lifetime).

4.3 Simulated Annealing

Our initial implementation of generating the encounters was using Simulated Annealing [26]. Simulated annealing works similar to a hill climbing algorithm, except that instead of always picking the best neighbor, we use a probability distribution. Neighbors with better fitness have a higher probability, but we might still take a path downhill. The amount of randomness is controlled by an

external variable called temperature (or energy). As the algorithm progresses, the temperature is slowly reduced and thus allows it to converge. The main advantage over hill climbing is that simulated annealing does not immediately get stuck in a local optima, as there is always a chance it will move to a state with lower fitness value. In contrast, a hill climbing algorithm [26] would always pick a neighbor with a better fitness value.

However, we had difficulty getting good results and the algorithm almost never converged. As a result, we ran an experiment to sample the search space at roughly 20 million different points, and measured the change in fitness in the neighborhood of each point. We found that in each point's neighborhood, there are roughly 5 times more points that have worse fitness than the ones that are an improvement over the current point. We also found that most of these downward changes were much steeper between 4–7x than the improving points. Our suspicion is that this is the main cause of failure of Simulated Annealing, which simply fails to find the upward slope.

4.4 Evolution Strategies

For this reason we chose to try another approach, specificially Evolution Strategies (ES) [27]. ES works by taking a random sample of the neighbourhood of the current value, evaluating the fitness of each of the neighbours, and moving to a state that is the weighted average of the neighbours with respect to their fitness. This process is iterated until a state with suitable fitness value is found. In our experiments we have found this approach to consistently converge much faster than simulated annealing.

ES also yields itself to easy parallelization, especially if the evaluation of the fitness function is complicated. Since all of the neighbouring samples are completely independent, they can be evaluated completely in parallel.

Specifically, we sample by choosing from a uniform distribution from the range $[0.75; 1.25]$ (determined by the `MutationDelta` constant, see section D.3) and take that as a percentage update to one value in the DNA vector at a random index. We repeat this process with a probability of $p = 0.35$ (determined by the `SecondMutationProb` constant). Overall, we sample 40 neighbours this way during each iteration of ES (determined by `TeamsPerGeneration` constant).

4.5 Choice of the Fitness Function

In order to evaluate the balance of a matchup, we evaluated the following three criteria:

Balance Unsurprisingly, part of the fitness function is comparing the strength of both teams against each other. We consider games that end in a draw balanced. If the game doesn't end in a draw, we measure the remaining HP percentage of the winner, and the lower it is, the more balanced the game was.

Game length We also put an interval restriction on game length. Ideally we'd want the game to last at least 2 rounds.

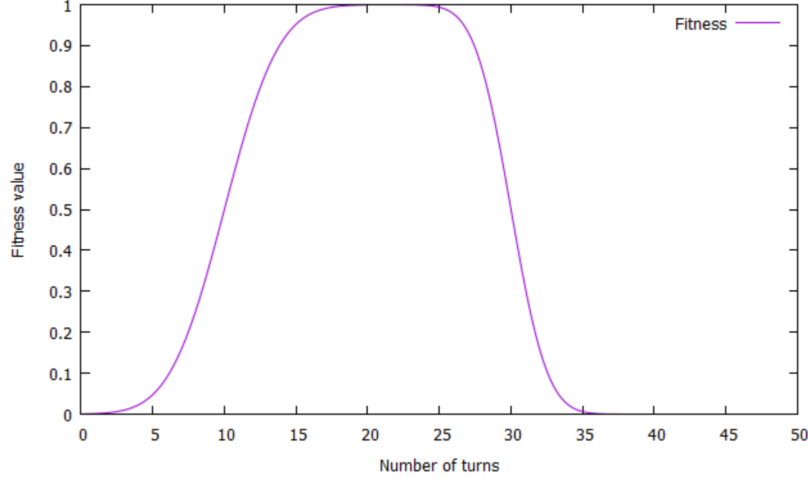

Figure 4.1: Fitness values for games of a given length, determined by two combined cumulative normal distribution functions.

Team difference Lastly, since we don't to create balance by making both teams identical, we added a third criteria that measures the difference of both teams.

The combined fitness function is calculated as the average of the three. See Figure 4.1 for an example plot of how the fitness function converges using Evolution Strategies. It is worth noting here that evaluating the fitness function (the *Balance* part specifically) is computationally intensive, as it requires the whole game to be played out till the end. We also do this evaluation using an ensemble average of multiple different AIs. Specifically, we use both the Rule based AI and MCTS in the *Balance* playout and take their average. This helps assure that the game is balanced despite multiple playstyles, as both the Rule based AI and MCTS play differently (see Table 5.1 for their comparison). You can see how the fitness function converges in Figure 4.3.

The combined formula is $\frac{(1-H)+L+D}{3}$ where H is the percentage of total health left at the end of the game, and

$$L = \begin{cases} f(\text{length}|10,3) & \text{if length} < 20 \\ f(40-\text{length}|10,2) & \text{otherwise} \end{cases}$$

where f is the cumulative distribution function for the normal distribution (see Figure 4.1). Lastly, $D = \frac{1}{1+\exp(-60R+1.5)}$ where R is the euclidean distance between the two teams (normalized to $[0;1]$, see Figure 4.2). Note that the constants the above mentioned formulas were chosen by hand as to fit the game design goals (minimum game length, etc.).

4.6 Remarks

During the experiment we encountered multiple ways ES tried to exploit the game mechanics to maximize the balance fitness function in ways that were undesirable. One example being when the resulting games end up being short as the algorithm

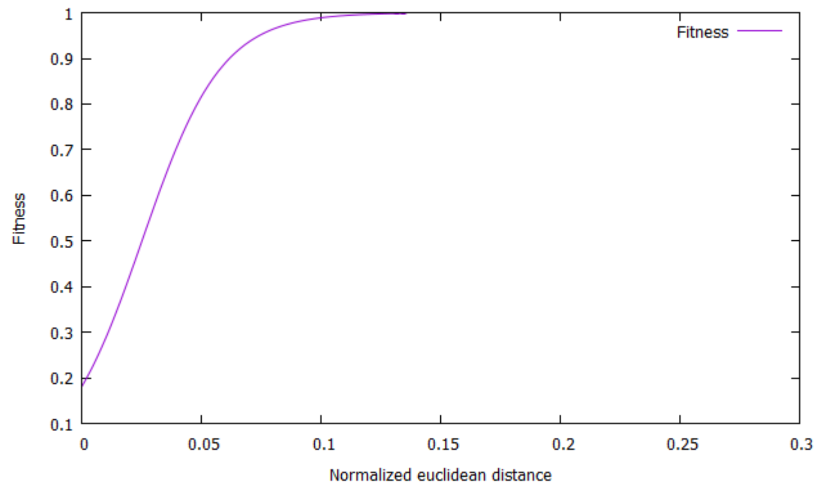

Figure 4.2: Fitness values for normalized euclidean distance between the two teams. Showing only the section where distance is less than 30%.

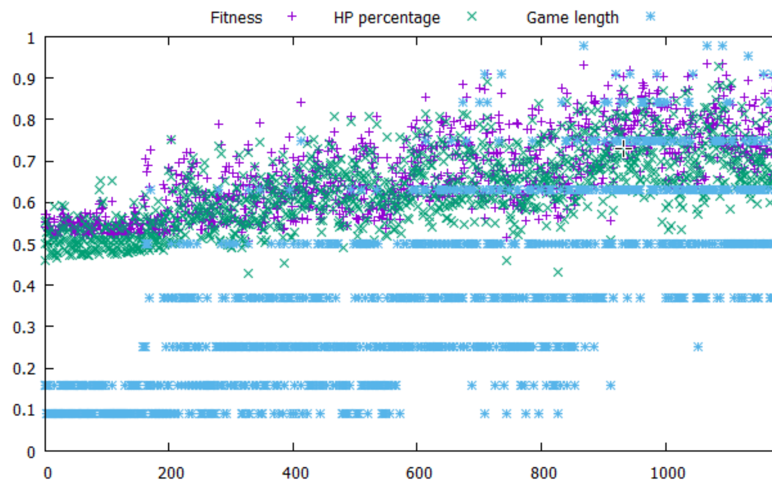

Figure 4.3: An exmple of a how the combined fitness function converges over rougly 1200 iterations. Green dots represent the *Balance* fitness measuring HP percentage left at the end of the game, blue dots represent game length, and purple is the combined fitness. We're not showing team difference data points as in this plot.

generates mages with lots of AOE abilities that cover the whole map in the first turn, resulting in immediate death of all characters.

We balance this by introducing an additional fitness function for game length in the form of a cumulative normal distribution with mean around 10 turns.

Lastly, the team difference is measured as an euclidean distance between the DNA vectors of each team. Again, we use a cumulative normal distribution to put a lower bound on the team difference.

5. Experiments

First we show a small experiment of comparing our different AI implementations against each other to measure their relative strength. This should uncover if there are any underlying flaws in the implementation. Next we present our user survey, of which the first goal is to verify the strength of our MCTS implementation. A second goal is to measure the balance of the generated encounters. Since the problem of game balance is largely defined by the players' perception, we can only measure it by asking the users in a survey.

5.1 Comparing AIs against each other

To compare the strength of our AIs against each other, we simulated 1000 randomly generated games and had the AIs play against each other. Both teams were generated at random and there was no step taken to make them balanced. Instead, each AI played both sides of the match. See Table 5.1 for results.

	MCTS	Rule	Random
MCTS	N/A	63%	88%
Rule	37%	N/A	82%
Random	12%	18%	N/A

Table 5.1: Table showing win percentages over 1000 games between our different AI implementations. MCTS beats Random 88% of the time, MCTS beats Rule 63% of the time, and Rule beats Random 82% of the time.

As we can see, both MCTS and the Rule Based AI can beat our Random AI with a significant margin, and MCTS can also consistently beat the Rule Based AI, despite playing for both generated teams. From this we conclude that:

1. A greedy aggressive strategy doesn't always win, which means the game mechanics are rich enough to reward strategic thinking.
2. Thinking ahead (as MCTS does) provides enough value to be significant in terms of win rate.

5.2 Survey

We've conducted a small survey to verify the results of our encounter balancing algorithm. The participants played 126 games in total and filled in a questionnaire after each game. We only had 7 participants, most of which have played the 20 games in the survey. The survey was done online within a group of students, as we couldn't get a larger group due to limited resources. However, given the total number of games played, we still believe that it has enough value to present in this thesis.

The goal of the survey is to show that games are balanced both by asking how players felt about the balance of the game, and by measuring the games won/lost.

Encounters with either 0% or 100% winrate will likely not be balanced. The goal is to also measure how the players percieve our MCTS AI implementation.

5.2.1 Participants

The participants were all computer science students. All of them had at least some experience with computer games, and were presented the mechanics of HexMage before conducting the experiment.

5.2.2 Experiment Design

The goal of the survey is to measure two things. One, if the generated encounters are actually balanced. And two, if the AI is strong enough to pose a challenge to the player. We will measure the overall winrate of the players, and also compare how the AI did in games the player considered balanced.

The experiment consists of 20 different games of HexMage. All of the games are played on the same map that was hand-designed beforehand. This was to allow the participants to better get familiar with the game and think ahead. The games are structured so that each player has 2 mages, each with 2 abilities. This was to reduce the cognitive overhead for the participants and allow them to more easily adjust to the 20 completely different scenarios.

The first 10 of the 20 games had the player team hand designed, and the opponent (played by the AI) generated with our PCG algorithm. The remaining 10 games had both teams generated with no manual tweaks or changes. Having some of the games hand designed allows us to show that the PCG algorithm can balance against constraints that aren't completely random. A hand designed team might have features that are rare in the search space. All of the games are played against the MCTS based AI with a fixed number of iterations.

5.2.3 Questions

The questions were the following, answered on scale (1 - definitely no, to 7 - definitely yes), showing short codes for Table 5.2 and Table 5.3. We intend to measure three things: the strength of the AI, the overall difficulty, and the balance of the encounter.

Balanced: The game was balanced.

Challenge: The game was challenging.

Unsure: I wasn't sure who was going to win.

Smart: The AI played smart.

Difficult: The game was difficult.

Strategy: The AI showed strategic thinking.

The intended pairing of the questions is *Balanced* and *Unsure*, *Smart* and *Strategy*, and *Challenge* and *Difficult*.

5.2.4 Discussion of the Results

As we can see in Table 5.2, the questions for balance strongly correlate. We also see many correlations between questions regarding strength of the AI and difficulty.

	balanced	unsure
balanced	1.00	0.73
unsure	0.73	1.00

Table 5.2: Correlation table between people who said the game was balanced and who were unsure about the result.

	challenge	smart	difficult	strategy
challenge	1.00	0.61	0.46	0.63
smart	0.61	1.00	0.39	0.82
difficult	0.46	0.39	1.00	0.39
strategy	0.63	0.82	0.39	1.00

Table 5.3: Correlation table between people who said the game was challenging, the AI was smart, the game was difficult to play and the AI showed strategic behavior.

We can also take a look at the responses normalized to 0 and 1 where 1 means the response was at least 4. Looking at Table 5.4, we can see that players tend to consider the game difficult if they lost and vice versa.

	balanced	challenge	unsure	smart	difficult	strategy	won
balanced	1.00	0.35	0.53	0.21	0.07	0.22	0.15
challenge	0.35	1.00	0.30	0.42	0.48	0.36	-0.20
unsure	0.53	0.30	1.00	0.17	0.16	0.14	0.07
smart	0.21	0.42	0.17	1.00	0.49	0.79	-0.32
difficult	0.07	0.48	0.16	0.49	1.00	0.42	-0.62
strategy	0.22	0.36	0.14	0.79	0.42	1.00	-0.22
won	0.15	-0.20	0.07	-0.32	-0.62	-0.22	1.00

Table 5.4: Table of correlations after normalizing reponse values from 1–7 to 0–1 where 1 means the original response was at least 0.

The total win rate of the players was only 38%, which shows that our MCTS AI is a formidable opponent. Table 5.5 show the mean of the other responses.

Looking further at the results, in 4 out of the 20 games the players lost 100% of the time, and in 1 out of the 20 games the players won 100% of the time. In the resulting 15 games there were both cases when a player won and when a player lost. We re-evaluated those games manually and found that in one case the game trully could not have been won, as one of the player mages always got killed too early on and did not get to do any damage. On the other hand, the second

	mean
balanced	0.60%
challenge	0.71%
unsure	0.57%
smart	0.78%
difficult	0.76%
strategy	0.78%

Table 5.5: Table of mean responses to all the questions in the questionnaire.

game we tried we won rather decisively, even though all of the participants in the experiment lost. The strategy to winning strategy was however to play very defensively and calculate the opponents AP, which is something the participants probably did not have a chance to do.

As for the third game, we also confirm it as impossible to win, as the opponent had a cheap ability and powerful ability and could easily stay out of range while doing damage. The last of the always-lost games was also similar in the sense that the opponent had a cheap ability he could use over and over again, and once he got a small advantage he could push through to win the game without any resistance. While the games were close in terms of HP at the end, there was no clear way to win.

In light of these results, there is a possibility for future work by incorporating an additional fitness metric that makes sure the player can win the game, and not just that the games end up being close. However, considering the players voted on 60% of the games as balanced, we have definitely met our goal of generating balanced encounters.

Note that all of the 20 games were generated without any human intervention, and the only input from the user was to set the constants for attributes' range. The whole process of generating a balanced encounter is completely automatic and replicable.

6. Conclusion

We have designed and developed a working game with a simulator, developed multiple AI bots, built an algorithm for generating balanced encounters and tested everything in a number of experiments.

Our experimental findings confirm that search based procedural generation is a viable solution for balancing encounters. While we didn't achieve perfect accuracy in generating balanced results, our participants considered around 60% of the games to be balanced. This is confirmed by our win rate results, where 75% of the games were not completely won/lost by the players. After manually checking the remaining 25% we found that one of the games was actually possible to win, which would raise the accuracy to 80%. We think of this as a success, as the games were generated with no human input or modifications to the data.

Our MCTS AI also showed to be working very well, as it decisively defeated both the Random, and the Rule based AI. It also did well against the human players in the experiment, reaching 62% win rate, and having around 75% of the games be voted as *smart* and *challenging* by the participants.

6.1 Future work

There are many opportunities for future work. One would be to develop a more sophisticated fitness function that takes into account problems mentioned in chapter 5. Team sizes could also be adjusted and allow changing the number of mages in a team and the number of their abilities as part of the evolution. Lastly, one could consider balancing regardless of the map the game is going to be played on, and generate the team such that it can adjust to different scenarios.

We also restricted a few mechanics in order to help things converge faster. For example, healing spells are not allowed in our simulator. Enabling them however runs the risk of games taking much longer, and they might not even finish if the amount of healing is greater than the amount of damage. One could also consider allowing AOE abilities to be targeted on the ground, and not just on the enemy mages. While this is possible in many of the popular games, it greatly increases the number of possible actions in each state where an AOE ability is available.

Bibliography

[1] Abrakam. Faeria. [PC CD-ROM], 2017.

[2] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[3] David Capello. Aseprite. `http://aseprite.org/`, 2001.

[4] Jurgen Van Gael Christoph Ruegg, Marcus Cuda. Math.net numerics. `https://numerics.mathdotnet.com/`, 2017. Accessed: 2017-05-09.

[5] Counterplay Games. Duelyst. [PC CD-ROM], 2016.

[6] Douglas Crockford. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, March 2014.

[7] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen North, Gordon Woodhull, Short Description, and Lucent Technologies. Graphviz — open source graph drawing tools. In *Lecture Notes in Computer Science*, pages 483–484. Springer-Verlag, 2001.

[8] NA: Blizzard Entertainment. Diablo 2. [PC CD-ROM], 2000.

[9] Free Software Foundation. Bash. `https://www.gnu.org/software/bash/`, 2007. Accessed: 2017-05-09.

[10] 2K Games. Borderlands. [PC CD-ROM], 2009.

[11] Intel. Intel core i7-5820k processor. `http://ark.intel.com/products/82932/Intel-Core-i7-5820K-Processor-15M-Cache-up-to-3_60-GHz`, 2014. Accessed: 2017-05-11.

[12] JetBrains. dottrace. `https://www.jetbrains.com/profiler/`.

[13] Microsoft. Microsoft public license. `https://opensource.org/licenses/MS-PL`, 2001. Accessed: 2017-05-09.

[14] Microsoft. Xna. `https://web.archive.org/web/20080517163516/http://msdn.microsoft.com/xna/`, 2004. Accessed: 2017-05-09.

[15] Microsoft. Windows forms. `https://msdn.microsoft.com/en-us/library/dd30h2yb(v=vs.110).aspx`, 2006. Accessed: 2017-05-09.

[16] Microsoft. Windows presentation foundation. `https://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx`, 2006. Accessed: 2017-05-09.

[17] Microsoft. Nuget. `https://www.nuget.org/`, 2010. Accessed: 2017-05-09.

[18] Microsoft. Microsoft asynchronous programming with async and await. https://msdn.microsoft.com/library/hh191443(vs.110).aspx, 2012. Accessed: 2017-05-09.

[19] Microsoft. Microsoft TPL. https://msdn.microsoft.com/en-us/library/dd460717(v=vs.110).aspx, 2012. Accessed: 2017-05-09.

[20] Mojang. Minecraft. [PC CD-ROM], 2011.

[21] MonoGame Team. Monogame. http://www.monogame.net/, 2009.

[22] LLC Mossmouth. Spelunky. [PC CD-ROM], 2008.

[23] James Newton-King. Json.net. http://www.newtonsoft.com/json, 2017. Accessed: 2017-05-09.

[24] Amit Patel. Hexagonal grids. http://www.redblobgames.com/grids/hexagons/, 2013. Accessed: 2017-05-09.

[25] Re-Logic. Terraria. [PC CD-ROM], 2011.

[26] Stuart Jonathan Russell and Peter Norvig. *Artificial intelligence: a modern approach (3rd edition)*. Prentice Hall, 2009.

[27] T. Salimans, J. Ho, X. Chen, and I. Sutskever. Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *ArXiv e-prints*, March 2017.

[28] Noor Shaker, Julian Togelius, and Mark J. Nelson. *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[29] MonoGame Tool. Monogame pipeline tool. http://www.monogame.org/documentation/?page=Using_The_Pipeline_Tool, 2009. Accessed: 2017-05-09.

[30] Mads Torgersen. What's new in c# 7.0. https://blogs.msdn.microsoft.com/dotnet/2016/08/24/whats-new-in-csharp-7-0/, 2016. Accessed: 2017-05-11.

[31] Unity Technologies. Unity 3d. https://unity3d.com/, 2005. Accessed: 2017-05-09.

[32] Ben Watson. *Writing High-performance. NET Code*. Ben Watson, 2014.

[33] Thomas Williams, Colin Kelley, and many others. Gnuplot: an interactive plotting program. http://gnuplot.sourceforge.net/, April 2013.

[34] Mario Zechner. libgdx. https://libgdx.badlogicgames.com/, 2014. Accessed: 2017-05-09.

List of Figures

2.1 User interface of the HexMage game, featuring a 2v2 game. The red player currently has a spell selected and is targeting one of the blue player's mages. 7

2.2 User interface of the map editor, showing a map of a radius of 5 hexes. Light green are hexes represent walkable surface, dark hexes represent walls. Red and blue hexes represent starting points. 8

3.1 An exmple of a MCTS game tree after 100 iterations. Each node contains (from top to bottom) a node identifier, currently active team, cumulated reward / total number of visits, and the action type. The nodes are also colored based on the team color. The actions are shown in shortcuts to reduce tree width (AM - Attack-Move, D - DefensiveMove, End - End Turn, NullAction - empty action representing the root node). 12

4.1 Fitness values for games of a given length, determined by two combined cumulative normal distribution functions. 16

4.2 Fitness values for normalized euclidean distance between the two teams. Showing only the section where distance is less than 30%. 17

4.3 An exmple of a how the combined fitness function converges over rougly 1200 iterations. Green dots represent the *Balance* fitness measuring HP percentage left at the end of the game, blue dots represent game length, and purple is the combined fitness. We're not showing team difference data points as in this plot. 17

C.1 The main screen of the questionnaire. It shows a list of games which must be played, along with their status. Finished games are displayed in gray and have a *.done* extension, unfinished games are displayed in white. A keyboard shortcut information is also showed. 35

C.2 Team selection scene which allows the user to pick AIs and team sizes before starting the game. The controls are described on the game screen. 36

C.3 (*Left*) An image shown when the mage does not have enough action points for the given ability. (*Middle*) An image of the currently selected and active ability. (*Right*) An image shown when the given ability is on cooldown. 37

List of Tables

5.1 Table showing win percentages over 1000 games between our different AI implementations. MCTS beats Random 88% of the time, MCTS beats Rule 63% of the time, and Rule beats Random 82% of the time. 19
5.2 Correlation table between people who said the game was balanced and who were unsure about the result. 21
5.3 Correlation table between people who said the game was challenging, the AI was smart, the game was difficult to play and the AI showed strategic behavior. 21
5.4 Table of correlations after normalizing reponse values from 1–7 to 0–1 where 1 means the original response was at least 0. 21
5.5 Table of mean responses to all the questions in the questionnaire. 22

Abbreviations and Terminology

MCTS Monte-Carlo Tree Search

UCT Upper Confidence Bound for Trees

UCB Upper Confidence Bound for the multi-armed bandit problem.

Winrate The percentage of games a player has won in a set number of games. For example winning 3 out of 10 games gives a 33% winrate

Turn Playing a single mage until the `EndTurn` action is considered a *turn*. Each mage gets to play one turn within a single round.

Round When all mages play their turn it is considered one round. Debuffs and AOEs are calculated at the end of each round, and their lifetime is decreased.

Mage A player or AI controller character.

Damage The effect of reducing health of a character. Ability with 5 damage reduces the health of the target by 5.

Action point A resource which can be spent both on movement and ability usage. Moving one hex costs one action point, and using an ability costs as many as is defined in the ability's description.

Cooldown If an ability has an associated *cooldown*, using it disables the ability for the number of rounds as specified by the cooldown attribute.

Debuff A temporary effect of an ability which causes the target to low health and action points for a given amount of time. Debuffs have a defined lifetime, which is the number of rounds they last.

AOE Area of Effect debuffs are similar to *Debuffs*, but they're placed on the ground instead of the targeted mage. AOEs have a defined radius which they affect, and they also have an associated debuff which is applied to a mage standing within the radius at the end of a round.

A. Implementation Analysis

Since the whole project is implemented in C# on the .NET platform. As such, it is structured into multiple Visual Studio *projects* under a single *solution*. A solution is a collection of projects that are built together on the .NET platform. The projects are the following:

HexMage.Simulator Contains all of the game's core logic, as well as implementation of the AIs and encounter balancing. This project is also compatible with Mono on Linux.

HexMage.Benchmarks Small project that wraps all of the experiments in a command line interface. This project is also compatible with Mono on Linux.

HexMage.GUI Contains the GUI of the game, which includes the arena, map editor, team generation and the questionnaire interface. It also has the ability to re-play games stored in replay files.

HexMage.Tests Small project containing just a few unit tests for the most critical features.

There is also a directory structure under the *data* directory that is used by the projects

data/graphs Contains GraphViz DOT [7] files for the MCTS tree. These are used mostly for debugging purposes and their generation can be turned on and off by defining a *DOTGRAPH* compile time constant (see *UctAlgorithm.cs* for details). Each iteration of MCTS is stored in a separate file.

data/images Output directory for the generated image graphs from the *graphs* folder.

data/manual-teams Contains manually designed teams for the questionnaire. Each team is stored in its own JSON [6] file.

data/questionnaire Contains the generated encounters for our questionnaire. Each encounter is stored in a separate file. The file format is rather simple and described in the DNA file format section.

data/replays Contains replays recorded from games in JSON format. Replay recording can be turned on with the *RecordReplays* command line argument. See the command line section for more details on how to provide command line arguments.

data/save-files Contains saved DNA results in both the short DNA format and in JSON format (see section B.1 for details). This folder is used by the search algorithm, which stores DNA with high enough fitness value.

There are also a few misc files in the data folder that serve mostly for data processing. Most notably the `generate.sh` file contains a Bash [9] script for generating MCTS tree images from the `data/graphs` folder. We will now over go the internal structure of the project.

A.1 HexMage.GUI

The GUI is written using the MonoGame [21] library (version 3.5), which provides capabilities such as sprite rendering and user input handling. It also asset loading (textures, sprites, fonts) and compiles them together with the application. All of the assets can be found under `HexMage.GUI/Content`. Most of the graphics was done using the Aseprite [3] pixel art editor. Now onto the project structure.

We chose MonoGame because we wanted to use a higher level language, and given the performance constraints, the only choice was between C#with MonoGame and Java with libGDX [34]. We chose C#as it has the tools for more compact data structures and better development tools. We didn't consider larger game engines such as Unity3D [31] since our game is rather simple, and we wanted to avoid spending most of the time trying to figure out how to do particular things in a big framework, but rather focus on the important parts.

The game is organized into *Scenes*, where each Scene represents a single screen for the user. The scenes are the following (located under `HexMage.GUI/Scenes`):

MapEditorScene.cs Shows the map editor.

TeamSelectionScene.cs Allows the user to generate teams and assign AIs to play the game.

ArenaScene.cs Shows the current game, this is where the game is played.

QuestionnaireScene.cs Serves as a UI for our survey. It shows the participants their progress.

Each scene contains an arbitrary amount of root *Entities*. Each entity can have any amount of child Entities, and also any amount of *Components* attached. Entities can also optionally have a *Renderer* (subclass of `IRenderer`) if they wish to be displayed to the user. Other than that, entities contain positional information, such as a relative position with respect to their parent, the order in which they are rendered, and a flags (i.e. `Active` flag which determines if the entity is shown/updated on each frame). The Components are what give Entities most of their functionality, and can be an arbitrary class that extend from the `Component` parent. The most important part of a Component is their `Update` function, which gets called on every frame.

All of the user interface is implemented in terms of Entities and Components. We also provide some basic layout primitives, such as a `VerticalLayout`, which lays out its children vertically as a list (with an optional spacing in between). We also implemented a small UI library in the form of clickable *Buttons* which can have an arbitrary callback attached. The important part of the UI are the `GameBoardController` and `GameBoardRenderer` classes, which implement most of the logic for the game arena.

On top of the scene mechanism, we also provide basic user input handling in the form of the `InputManager` class, and 2d camera in the `Camera2D` class. This allows the user to zoom in/out and move around the map easily. The GUI also contains its own logging mechanism with different log levels.

A.1.1 Threading and TPL

While the GUI is single threaded, we make use of the TPL for asynchronous task processing. This is implemented using a custom `SynchronizationContext` subclass, which makes sure asynchronous callbacks are executed within the main event loop of the GUI. Implementation-wise it would have been simpler to use a worker thread and communicate with queues, but our approach provides much easier and flexible API for the developer. By implementing a global subclass of `SynchronizationContext` we can make use of the .NET 4.x `async` [18] feature.

Callbacks of `async` methods are then automatically scheduled onto our custom event loop and are executed right before the `Layout` and `Update` lifecycle methods are called on the root entities (see Appendix D). This also allows actions to be interleaved with the regular game loop without having to worry about locks and shared mutable state. We still have the ability to execute tasks on a thread pool using the `Task.Run` method from the TPL [19], but at the same time we can specify the callback to run on the game event loop using the await `await` thanks to our custom `SynchronizationContext`.

A.2 HexMage.Simulator

The main core of our program is the simulator. It contains all of the game rules and mechanics and can be used as a standalone library. The main class is `GameInstance` which encapsulates the game state. It delegates most of its functionality to the following classes:

Map Contains all of the logic related to the map itself, such as whether each hex is empty or a wall, and also starting point information

Pathfinder Calculates and caches pathfinding and visibility information for the given map.

MobManager Stores immutable information about the mages and their abilities. For example maximum amounts of HP and action points, ability definitions, etc.

GameState Contains state information which changes as the game is played. This contains the current values, such as current HP, action points and position on the map for each mage.

TurnManager Encapsulates all of the logic regarding turns and rounds.

GameEventHub Handles the main game loop and handles switching turns between players, and detects whether the game has finished.

A.3 Performance

We have spent a considerable amount of time optimizing the simulator to run as fast as possible. Most of the game state related data is stored in tightly packed arrays of structures. This was done to reduce the number of heap allocations and

reduce overall pressure on the garbage collector [32]. It also improves copy time of the game state, which happens during each iteration of the MCTS algorithm.

With `DefensiveMove` generation enabled, MCTS can run around 30000 iterations per second, and the simulator running around 150000 actions per second (on the Intel i7-5820k Processor [11]). Note that most of the time is spent on generating high level actions. When `DefensiveMove` actions are disabled, MCTS can run nearly twice as fast since it can avoid re-calculating which positions on the map are safe. Improving the high level action generation is a possibility for future work. More elaborate heuristics could allow MCTS to handle larger map sizes and perform even better as the search depth is limited by the number of iterations it can run within a given amount of time.

A.3.1 Map implementation

Since the map is composed of hexagons, some special care was taken to implement things properly. Most of the algorithms such as calculating visibility and drawing layout were inspired by Amit Patel's Hexagonal Grids article [24].

We use a 2-dimensional array to store the hexagonal map (class `HexMap<T>`) and restrict the valid coordinates to only those with a distance of N from the center, which makes the resulting map have a hexagonal shape as well (with a radius of N).

A.3.2 Game Events and AI Controllers

Our main game loop is encapsulated by the `GameEventHub` class. It does not however pick which action each player is doing, that is the job of a `IMobController`. Each AI we implemented has its own implementation of the `IMobController` interface, and there is also one for the human player called `PlayerController`. This allows us to write the game loop without knowing who is going to play the game. The responsibility of the controller is only to dispatch proper actions during its turn.

Also worth noting that since we're not using threading in the UI, but rather asynchronously queued tasks in the event loop, we can't simply block and wait for the `PlayerController` to respond with a given action, as that would freeze the game UI. Instead, we provide an asynchronous game loop in the form of `GameEventHub.SlowMainLoop`. It handles turn management automatically and all the programmer has to provide is an implementation of the `IMobController` (see Appendix D).

B. File formats

B.1 DNA file format

The DNA vectors are scored in simple text files. The format is as follows:

- Each team is stored on a separate line.
- The line first contains two numbers, one for the number of mages in a team, and one for the number of abilities of each mage.
- Then follows the serialized vector for each mage in the order the mages were defined.
- Each mage is stored as his health, action points, and a list of abilities.
- Each ability is stored as damage, cost, range, cooldown, debuff, AOE.
- Debuff is stored as damage, action point damage and lifetime.
- AOE is stored as radius and the effect as if it was a debuff.

A reference implementation is in the `GenomeLoader.cs` file which contains both serialization and deserialization. Note that the starting positions of the mages are not stored in their DNA, as that information an attribute of the map on which the game is played. See the Map file format section for more details.

We also have an alternative file format for the DNA which can be seen in

B.2 Map File Format

The map is stored in a plain JSON file [6]. An example can be found in the `data/map.json` file. The file defines the size of the map, whether each hex contains a wall or is empty, and the starting positions of each team. It is important to note that although there can be any number of starting positions on a given map, the team size must not exceed the number of starting positions. If the user tries to start a game with larger teams than the number of starting positions, the program will detect the error and not start the game.

C. User Documentation

C.1 System Requirements

All of the code is written for C#6.0. The `HexMage.GUI` project runs on Windows under .NET framework 4.5 (or newer). It also requires up to date graphics drivers, and optionally audio drivers when running with `--EnableAudio=true`. The experiments in `HexMage.Benchmarks` can be run on Linux under a recent version of Mono (tested on 4.x). See section D.2 for more detailed instructions on how to compile and run the project.

C.2 Generating Experiment Data

The data for our survey are under the `data/manual-teams` directory. Each file represents a single team for which a suitable opponent will be generated by our algorithm. Any number of files can be added.

In order to generate the data for the experiment, simply run the `HexMage.Benchmarks.exe` program with no command line arguments (preferably in Release mode) and everything will be generated automatically. The results are stored under `data/questionnaire` in separate files in the DNA file format section B.1. Generating the data takes around 1 hour on Intel i7-5820k Processor [11]. We've disabled parallelization in order to make the data generation deterministic and the whole experiment reproducible. An initial seed is defined under `Constants.RandomSeed` and is used by all of the internal random number generators. Parallel execution can be re-enabled by defining a compile time `PARALLEL` constant.

C.3 Running the Experiment

Running the `HexMage.GUI.exe` program will run the experiment by default. It loads the data from `data/questionnaire` and presents a list of the required games on the main screen (see section C.3). The state of the questionnaire is maintained in the files themselves. Specifically, finished games will be renamed to have a `.done` extension. This allows the user to resume at any point in time, and also makes it easy to revert back to an initial state if something goes wrong. Pressing `Ctrl-Q` will reset all of the games to unplayed state. This is meant mostly for debugging.

The next game can be started by pressing `Spacebar`, which immediately picks the next game on the list and loads it. After the game is finished, the questionnaire is opened in the default browser and the game identifier is pre-filled (we used Google Forms). The form itself is configured in `QuestionnaireScene.cs`. After the user finishes playing the game, they should quit the program and relaunch it to proceed to the next game. We did this for stability purposes since the questionnaire was run remotely and we had no way to assist the user in case of errors.

We also provide the exact ZIP file we used to run the experiment with all of the generated games to make it easy to reproduce the results without having to run the encounter generation script.

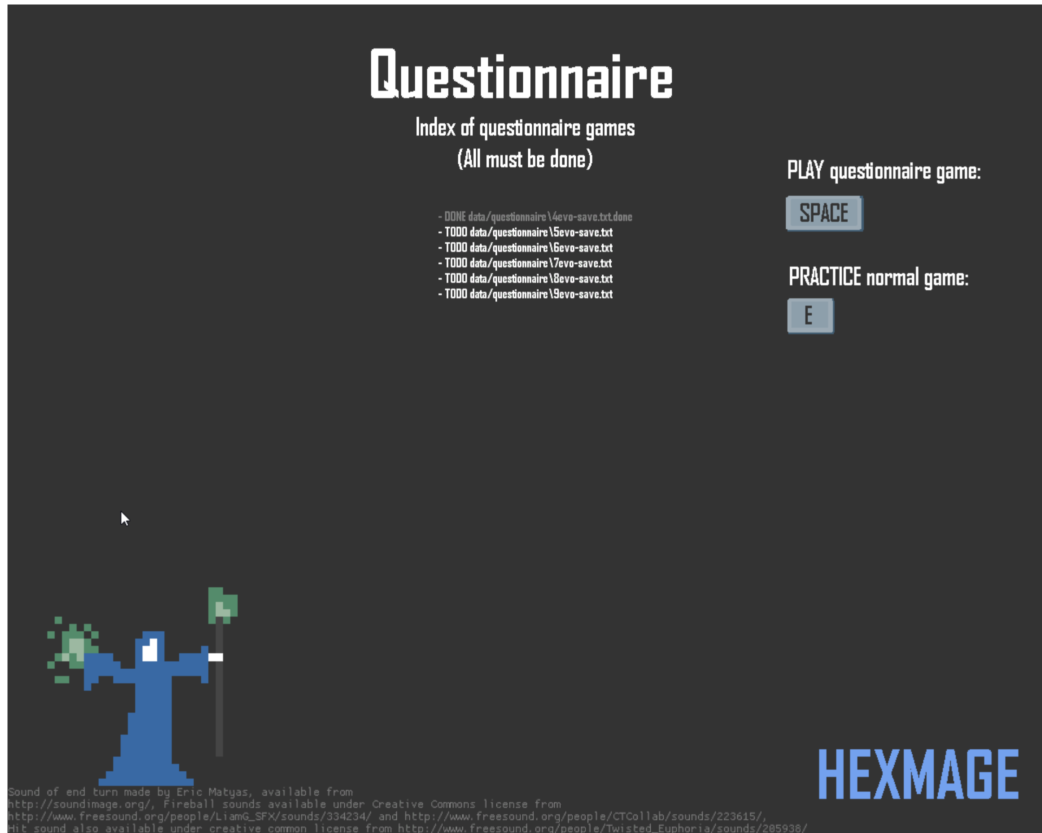

Figure C.1: The main screen of the questionnaire. It shows a list of games which must be played, along with their status. Finished games are displayed in gray and have a *.done* extension, unfinished games are displayed in white. A keyboard shortcut information is also showed.

C.4 Playing Custom Games

The user can also press `E` on the initial screen to go into *Practice mode*. This opens up a map editor (see Figure 2.2) where the user can create a custom map, save it to a JSON file (see section B.2), load saved files, set startup positions, and continue to the AI selection by pressing `Space`. This allows the user to select AIs for both of the teams and set the team sizes (see section C.4). At any point the user can press `Ctrl-R` to go back to the initial scene. This can be done in game as well.

Once the user is happy with the team selection, they can press `Space` to start the game, which brings them to the arena scene (see Figure 2.1). It shows the current game state in the middle of the screen, abilities of the current mage on the left, abilities of the mage under the mouse cursor on the right, the current team at the top, the game history in bottom left, and the detail of a hex under the mouse cursor on the bottom right. Since there is no hidden information, the player can easily see what abilities his opponents have. Figure C.4 shows the

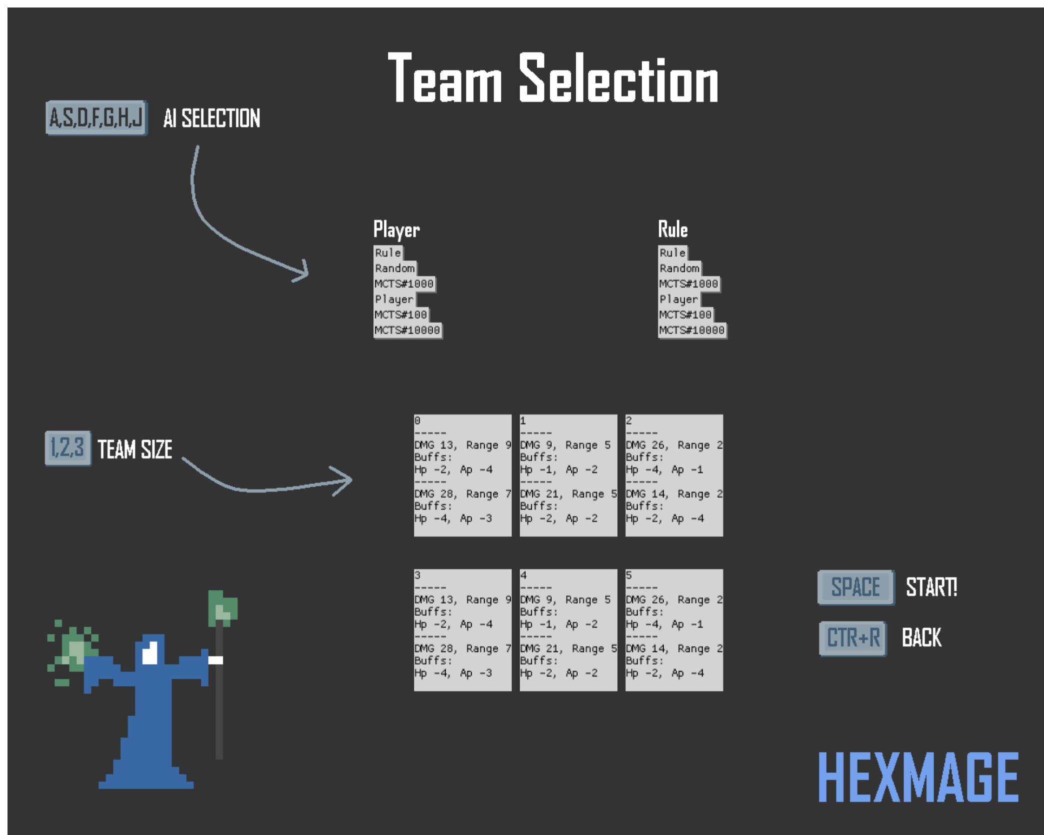

Figure C.2: Team selection scene which allows the user to pick AIs and team sizes before starting the game. The controls are described on the game screen.

Figure C.3: (*Left*) An image shown when the mage does not have enough action points for the given ability. (*Middle*) An image of the currently selected and active ability. (*Right*) An image shown when the given ability is on cooldown.

images representing different ability states. The currently active mage has their underlying hex colored in yellow.

When no ability is selected, the player moves his current mage by clicking the left mouse button on an empty hex. The game displays the distance to the hex in the bottom right corner, as well as drawing the path the mage will walk on. If the target hex is a wall or an enemy is standing on it, the path is not shown, indicating that the mage can not move to the target hex.

Abilities can be selected either by clicking on them with the left mouse button, or with the number keys on the keyboard (`1` for the first, `2` for the second, etc.). When an ability is active, it is shown glowing as shown in Figure C.4. With an active ability, the player can left-click on an enemy mage to use the ability on him. A visiblity indicator is drawn when hovering over an enemy to show whether the enemy is in range and the ability can be used on him.

After the player is finished with his turn, they can press `Space` to finish the turn and move on to the next character.

D. Programmer Documentation

D.1 Used Libraries

While developing desktop GUI applications on Windows provides us with many options, the choices narrow down significantly when the application is a game. As we already established earlier, our choice of language is C# as it provides the ability to write high level code while maintaining key performance characteristics such as in-place allocated value objects [32]. Firstly there is Windows Forms [15] and Windows Presentation Foundation [16] (WPF), which are both excellent desktop GUI libraries, but are not fit for building games as generally require a completely custom user interface. While both Windows Forms and WPF have the ability to embed an OpenGL or DirectX context as a component in their native window components, such approach forfeits all of the benefits of the library and forces the user to implement everything by hand.

Then there is the Unity3D game engine [31] which has a large number of features game developers could need. Unfortunately Unity3D brings its own build system and enforces the developer to structure his game the way Unity3D developers intended. The engine is mostly built for 3D games (with some support for 2D) and as such provides lots of advanced features mainly useful to game developers building larger games. More importantly, Unity3D does not provide any support for tile based movement, hexagonal grids, pathfinding on custom tile-based map, and has a very complicated user interface library tailored mostly for cross platform development. In result, we conclude that using Unity3D would result in us building a lot of things from scratch anyway.

This leaves us with the last contender, which is MonoGame [21], a successor of the previously famous but no longer maintained XNA [14]. MonoGame is Open Source and available under the Microsoft Public License [13]. It abstract away the rendering backend and provides a simple API for rendering sprites. It also handles asset packaging (images, sounds, fonts), user input, and provides some basic data structures and algorithms for computer graphics.

We also make use of the Math.NET Numerics [4] library for some numerical computation and probability distribution functions, and Newtonsoft Json.NET [23] for serialization and deserialization of the JSON files. Both of these can be installed easily via NuGet [17] package manager. MonoGame itself needs to be installed with the installer as defined by the instructions on the website [21]. Note we only tested with the version 3.5, but newer releases should be compatible as well.

GNU Plot [33] is also required to generate plots of encounter balancing if the `GnuPlot` constant is set to `true`. It needs to be available in the system `PATH` via `gnuplot.exe` in order to generate the plots (see `GnuPlot.cs` for more details). When the constant is set to `false`, no plots will be generated an the program does not need to be present.

D.2 Compilation Instructions

There are two ways to compile our project. First is when one wants to run the `HexMage.GUI` project, which requires the libraries specified in section D.1. These can be installed via NuGet [17], but one also needs to install MonoGame separately in order to use the MonoGame Pipeline Tool [29], which compiles the binary assets under `HexMage.GUI/Content` to a binary format which MonoGame can load efficiently. This is not required for compiling and running the `HexMage.Benchmarks` project.

After the MonoGame Pipeline Tool is installed, use it to open `HexMage.GUI/Content/Content.mcgb` and compile the assets (optionally this could be done through the Visual Studio build system). Next, use NuGet from within Visual Studio 2015 to install the packages. This is done by right-clicking on the `HexMage` solution in the *Solution Explorer* and clicking on *Restore NuGet Packages*. Lastly, select the `HexMage.GUI` project as a startup project and click the *Run* button. The compilation can also be done from within the *Developer Command Prompt for VS 2015* by going into the root `HexMage` directory and running `msbuild HexMage.sln`.

Compiling under Mono is similar, but only works for the `HexMage.Benchmarks` and `HexMage.Simualtor` projects. First go to the `HexMage.Benchmarks` directory and run `nuget restore`. Then, build the project either using the Mono command `xbuild` (or `xbuild /p:Configuration=Release` for a Release build), or if the `make` command is available, you can simply run `make` within the project's directory and it will use its `Makefile` to run `xbuild` automatically.

D.3 Constants and Command Line Arguments

All of the important constants are configurable through command line arguments, both for the `HexMage.Benchmarks` and `HexMage.GUI` projects. The constant defaults are configured in `Constants.cs`. When the programs are run, the command line arguments are inspected and set respective values in the `Constants.cs` class via reflection. Adding a new command line argument is thus as simple as adding a new property to the `Constants.cs` file.

The program is then run as following:

```
HexMage.Benchmarks.exe --EnableSounds=true \
                       --TeamsPerGeneration=40
```

will set the `EnableSounds` constant to `true` and `TeamsPerGeneration` to 40. Constant types are automatically inferred by the reflection mechanism in .NET and can thus be specified without any additional boilerplate. The meaning of each constant is documented in the `Constants.cs` file.

D.4 Running the Experiments

There are multiple experiments that can be easily run through the command line via the `HexMage.Benchmarks.exe` file (or by running the `HexMage.Benchmarks` project in Visual Studio).

The following command line arguments are possible (in addition to the constants defined in section D.3):

mcts-measure-speed Runs a MCTS speed benchmark which simply generates a random game and keeps playing it over and over again. The main purpose is for running MCTS itself under a profiler.

compare-ai Runs an AI comparison benchmark. The types of AI that are selected are controlled via the `MctsBenchType` constant (see section D.3). The selected AIs are run against each other on random games and the resulting win rate is measured.

space-stats Samples the search space at random points, looks around at neighbours and measures the ratio of upward/downward slopes with their respective fitness change.

When none of these arguments are specified, the encounter balancing algorithm is run on the questionnaire seed data (see section 5.2).

D.5 The Game Loop and GameEventHub

The main game loop in the GUI is represented by a `GameEventHub` class. During each turn, it queries the game state for the current `IMobController` (see section D.6) and calls its `SlowPlayTurn` method to have the controller execute the turn. After the call returns, the `GameEventHub` automatically ends the turn and moves on to the next character in the turn order.

If the player wishes to pause the game, the `GameEventHub` will delay the game loop until the game is unpaused, which means the game can only be paused at the end of each turn. This however isn't visible to the user, as they can press pause at any time, and if the AI is currently playing, the game will pause once it finishes its turn.

D.6 Writing Custom AI

The simulator is written in a way which makes it easy to extend it with new AI implementations. One only has to implement the `IMobController` interface. Two methods have to be subclassed:

void FastPlayTurn(GameEventHub eventHub)
A synchronous implementation of playing a single turn. The `eventHub` object is how the AI author can perform actions in the currently played game. It exposes a method `void FastPlayAction(UctAction action)` which will take an arbitrary action and execute it on the currently played game. Note that instead of passing an `EndTurnAction` the controller should instead return from the `FastPlayTurn` method. It is the responsibility of the caller to end the turn.

Task SlowPlayTurn(GameEventHub eventHub)

An asychronous variant of the above function works exactly the same way, except that the caller `await`s on the returned task. It is used in the GUI game loop which executes asynchronously with the main GUI event loop.

Another useful class is `ActionGenerator` which can provide some of the logic behind MCTS and the Rule based AI. Namely the `PossibleActions` generates a list of high level actions which MCTS uses in its decision tree. For more detailed information see the API reference in the attachments. Also see subsection D.7.2 for additional information on how to generate and check the validity of actions.

D.7 Creating Encounters with GameInstance

As described in section A.2 the `GameInstance` class is the main type in the simulator. It represents the complete encounter together with current game state. Since there are a lot of things that need to be configured, we also provide helpers in the form of `GameSetup` class that can generate a new game with a given map, number of mages, and abilities.

An important concept we need to describe first is the use of *identifiers* (or handles) throughout the simulator. These are integer types that serve as an identifier of an structure stored and managed internally by the simulator. An example might be using a `int abilityId` parameter instead of a whole `AbilityInfo` class. This allows us to keep the data stored compactly in internal data structures while avoiding unnecessary copying. Since the encounter setup is done only once at the beginning of the game and no abilities/mages are added/removed in the process, we simplified the identifiers to work as indexes directly into the internal array data structures.

While this breaks encapsulation to some extent, it allows us to remove a layer of indirection in the form of lookup tables. An example of this might be accessing a `MobInstance` as `game.State.MobInstances[id]`. A reason for direct array access is that C# doesn't allow the use of arbitrary references and would introduce unnecessary copying in some cases when the client wants to extract an object, modify it, and store it back. This feature could be handled with a new feature of C# 7 called *ref return values* [30] , which allows code to return a reference to a value type from arbitrary functions. However, at the time of writing most of the code C# 7 was not yet available, which made us stick with the more verbose, yet still functional approach.

It is also worth noting that `GameInstance` and all of its members support shallow and deep copying. A shallow copy can be done by the `CopyStateOnly` method which only makes a copy of the state attributes and thus does not allow the modification of abilities and the map. On the other hand, a deep copy done using `DeepCopy` will copy everything to a new instance and can be used in cases when one wishes to modify the definitions of abilities. This is used when generating the encounters.

D.7.1 Encounter Setup

There are multiple ways one can create an encounter. First we describe all of the necessary structures and how to create them:

Map

Represents the whole map, can be either created programatically, or loaded from a file using the `Map.Load(string filename)`. Apart from configuring which hexes are walls and which are empty, it is also important to configure the number and positions of starting points for both teams. The game can not be started with a team larger than the number of starting points for that color.

MobManager

Keeps all of the immutable information related to the encounter. This consists of `AbilityInfo` and `MobInfo` instances. There is no need to create the `MobManager` manually as the `GameInstance` will take care of it. One can also provide the `GameInstance` with an existing `MobManager` in case it was created beforehand. Such case might be when de-serializing content from a file for example.

AbilityInfo

Represents a single ability of a mage. It must be added to the `GameInstance` exclusively through the use of `AddAbilityWithInfo`, since there is an additional internal setup that must be carried out by the `MobManager`.

MobInfo and MobInstance

Together these two structures represent a single mage (called *Mob*, short for *mobile object*, internally). `MobInfo` describes the immutable characteristics, such as maximum amount of health and action points, which abilities are available, and the team the mage was assigned to. Note that there is an extra attribute called *initiative* which detemrines a turn order, but was not used in our experiment or encounter generation to simplify the number of variables that needed to be optimized. `MobInfo` objects can be accessed under `game.MobManager.MobInfos`. The `MobInstance` on the other hand contains all of the *current* information, such as current health and action points, position, and debuffs. As such it is stored in the `GameState` object under `game.state.MobInstances`.

Apart from the above mentioned, the `GameState` also holds information about cooldowns and AOEs, and keeps track of the state of the game (whether the game has finished or not).

We also provide a simpler abstractions that are not used directly by the simulator, but store the whole encounter directly and are used in serialization. This is handled mostly by the `GenomeLoader` class together with the `Team` and `DNA` objects. Both `Team` and `DNA` represent the same data but in different format, and can be converted between themselves. If one has already serialized a setup and loaded it with the `JsonLoader` object, it can be then converted directly into a `GameInstance` using `GameSetup.UnpackTeamsIntoGame` method (see API documentation for details). Another good reference can be both the

`AiRandomController` and `AiRuleBasedController` which are both rather simple and contain only the necessary boilerplate to get things working.

D.7.2 Invariants and Action Validation

Since our game mechanics are rather complicated, we provide a set of runtime checks that verify most of the game mechanics. These can be found under `GameInvariants` and are mostly turned on only when the `DEBUG` compile constant is defined. This is done automatically when doing a Debug build in Visual Studio, and is automatically turned off in Release mode. We made these checks conditional since they can be rather expensive and slow down the simulation a lot.

The class also provides some predicates which are not debug-only and are used by the simulator to check if a given action is valid. For example, one can check if an ability can be used on a target with the `IsAbilityUsable` method. This will check if the target is an enemy, alivem, visible from our current position, if we have enough action points, if the ability is not on cooldown and if the target is within range. These helpers can also serve the programmer developing an AI for our game, as all of the game logic has been encapsulated and the programmer can simply check if some action is valid, without having to implement all of the game rules themself.

D.8 Extending the Game UI

The UI is structured as a tree of `Entity` objects, each with a number of attached `Component` instances. Each `Entity` has an information about its position and a parent relationship. It can also optionally have a `Renderer` attached (subclass of the `IRenderer` interface), which takes care of the actual rendering. The `IRenderer` has a simple interface that only requires implementing a `Render` method, which can do any arbitrary rendering using the MonoGame [21] API.

D.8.1 Entities and Components

A new `Component` can be either created by sub-classing, or by using a shorthand with the `LambdaComponent` class, which allows components to be defined simply by their `Update` method provided as a lambda function. There are no restrictions on the `Update` function, except for deleting entities. This must be done by the use of the scene's `DestroyEntity` method which will queue up the delete until a next frame update. The reason is that we can't know if the deleted entity still needs its components to be updated within the current frame, or if they were already updated, or if we are actually deleting the parent of the current entity.

We also provide two additional helpers for delaying actions. Firstly, there is `AfterUpdate` which takes an arbitrary action and runs it after all of the `Update` functions have been called. And then there is also `DelayFor`, which works similarly but also takes a time for which the action should be delayed. The time keeping is managed automatically by the `GameScene`.

Similarly, newly added entities need to be initialized, and as such one should always use the `AddAndInitializeRootEntity` when adding a root entity. Child

entities are initialized automatically by their parents. This leads us to the component and entity lifecycle. First each entity is initialized with their respective `Initialize` method, then each root entity has its `Layout` method called (and recursively calls down to children automatically), then all root entities have their `Update` method called, and lastly, all entities which have a `IRenderer` attached have their `Render` methods called. During the `Update` call, the `GameScene` will also automatically call `Update` on the entitie's components.

D.8.2 Scenes

Each screen in the game is represented by a `GameScene` object. These objects are automatically managed by the `SceneManager` which keeps a stack of scenes and executes the lifecycle loop on the currently top scene. Scenes can terminate themselves using the `Terminate` call which will destroy the scene at the end of its event loop, and new scenes can be added using `LoadNewScene`.